DNA Computing: Modelling in Formal Languages and Combinatorics on Words, and Complexity Estimation

by

Zihao Wang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2022

© Zihao Wang 2022

Examining Committee Membership

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

External Examiner:	Cezar Câmpeanu School of Mathematical and Computational Sciences University of Prince Edward Island
Supervisor(s):	Lila Kari David R. Cheriton School of Computer Science University of Waterloo
Internal Member:	Ming Li David R. Cheriton School of Computer Science University of Waterloo
	Jeffrey O. Shallit David R. Cheriton School of Computer Science University of Waterloo
Internal-External Member:	Andrew C. Doxey Department of Biology University of Waterloo

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

The main contribution of this thesis consists of the following published articles:

- "Word blending in formal languages" published in *Fundamenta Informaticae* is the extended revised journal version of a conference paper I co-authored during my MSc study titled "Word blending in formal languages: the Brangelina effect." In particular, this paper contains complete proofs for all results (omitted if already included in the conference paper), numerous examples, expanded discussion of inverses of blending, new results concerning the iterated blending of languages, and revised state complexity results. The authors are ordered by last name alphabetically. The major individual contributions are listed below.

  Srujan K. Enaganti—topic.

  Lila Kari—topic, research ideas and proofs, results, and manuscript writing and editing.

  Timothy Ng—topic, research ideas and proofs, results, and manuscript writing and editing.

  Zihao Wang—topic, research ideas and proofs, results, and manuscript writing and editing (in particular, Section 3.4.1 and Section 3.4.2).

- The paper "Conjugate word blending: Formal model and experimental implementation by XPCR" was published in *Natural Computing*. The authors are ordered by last name alphabetically. The major individual contributions are listed below.

  Francesco Bellamoli—topic, research ideas, biology experiments, and results.

  Giuditta Franco—topic, research ideas, biology experiments, results, and manuscript writing and editing.

  Lila Kari—topic, research ideas and proofs, results, and manuscript writing and editing.

  Silvia Lampis—topic, research ideas, biology experiments, and results.

  Timothy Ng—topic, research ideas and proofs, results, and manuscript writing and editing.

  Zihao Wang—topic, research ideas and proofs, results, and manuscript writing and editing (in particular, Section 3.5).

- The paper “Involutive Fibonacci words” was published in *Journal of Automata, Languages and Combinatorics*. The authors are ordered by last name alphabetically. The major individual contributions are listed below.

  Lila Kari—topic, research ideas and proofs, results, and manuscript writing and editing.

  Manasi S. Kulkarni—topic, research ideas and proofs, results, and manuscript writing and editing.

  Kalpana Mahalingam—topic, research ideas and proofs, results, and manuscript writing and editing.

  Zihao Wang—topic, research ideas and proofs, results, and manuscript writing and editing (in particular, Section 4.1, Section 4.3, and Section 4.4).

- The paper “Primitivity of atom Watson-Crick Fibonacci words” was published in *Journal of Automata, Languages and Combinatorics*. The authors are ordered by last name alphabetically. The major individual contributions are listed below.

  Lila Kari—topic, research ideas and proofs, results, and manuscript writing and editing.

  Kalpana Mahalingam—topic, research ideas and proofs, results, and manuscript writing and editing.

  Palak Pandoh—topic, research ideas, proofs, and results.

  Zihao Wang—topic, research ideas and proofs, results, and manuscript writing and editing (in particular, Section 4.2, Section 4.6, and Section 4.7).

- The paper “As good as it gets: A scaling comparison of DNA computing, network biocomputing, and electronic computing approaches to an NP-complete problem” was published in *New Journal of Physics*. The first two authors contributed equally. The major individual contributions are listed below.

  Ayyappasamy S. Perumal—topic, research ideas, data curation and analysis, visualization, biology analysis, and manuscript writing and editing.

  Zihao Wang—topic, research ideas, data curation and analysis, visualization, and manuscript writing and editing (in particular, Section 5.2 and Section 5.5).

  Giulia Ippoliti—topic, research ideas, and results.

  Falco C. M. J. M. van Delft—topic, research ideas, results, biochemistry analysis, and manuscript writing and editing.

Lila Kari—topic, research ideas, results, and manuscript writing and editing.

Dan V. Nicolau—topic, research ideas, results, manuscript writing and editing, and biochemistry analysis.

Abstract

DNA computing, an essential area of unconventional computing research, encodes problems using DNA molecules and solves them using biological processes. This thesis contributes to the theoretical research in DNA computing by modelling biological processes as computations and by studying formal language and combinatorics on words concepts motivated by DNA processes. It also contributes to the experimental research in DNA computing by a scaling comparison between DNA computing and other models of computation.

First, for theoretical DNA computing research, we propose a new word operation inspired by a DNA wet lab protocol called cross-pairing polymerase chain reaction (XPCR). We define and study a word operation called word blending that models and generalizes an unexpected outcome of XPCR. The input words are $\alpha w \gamma_1$ and $\gamma_2 w \beta$ that share a non-empty overlap w, and the output is the word $\alpha w \beta$. Closure properties of the Chomsky families of languages under this operation and its iterated version, the existence of a solution to equations involving this operation, and its state complexity are studied. To follow the XPCR experimental requirement closely, a new word operation called conjugate word blending is defined, where the subwords γ_1 and γ_2 are required to be identical. Closure properties of the Chomsky families of languages under this operation and the XPCR experiments that motivate and implement it are presented.

Second, we generalize the sequence of Fibonacci words inspired by biological concepts on DNA. The sequence of Fibonacci words is an infinite sequence of words obtained from two initial letters $f_1 = a$ and $f_2 = b$, by the recursive definition $f_{n+2} = f_{n+1} \cdot f_n$, for all positive integers $n \geq 1$, where "$\cdot$" denotes word concatenation. After we propose a unified terminology for different types of Fibonacci words and corresponding results in the extensive literature on the topic, we define and explore involutive Fibonacci words motivated by ideas stemming from theoretical studies of DNA computing. The relationship between different involutive Fibonacci words and their borderedness and primitivity are studied.

Third, we analyze the practicability of DNA computing experiments since DNA computing and other unconventional computing methods that solve computationally challenging problems often have the limitation that the space of potential solutions grows exponentially with their sizes. For such problems, DNA computing algorithms may achieve a linear time complexity with an exponential space complexity as a trade-off. Using the subset sum problem as the benchmark problem, we present a scaling comparison of the DNA computing (DNA-C) approach with the network biocomputing (NB-C) and the electronic computing (E-C) approaches, where the volume, computing time, and energy required, relative to the input size, are compared. Our analysis shows that E-C uses a tiny volume compared to that required by DNA-C and NB-C, at the cost of the E-C computing time being outperformed first by DNA-C and then by NB-C. In addition, NB-C appears to be more energy efficient than DNA-C for some input sets, and E-C is always an order of magnitude less energy efficient than DNA-C.

Acknowledgements

First and foremost, I want to thank my supervisor Professor Lila Kari for her support in my academic and personal life over the past eight years. Without her guidance, comments, and editing, I could not have finished this thesis.

I would also like to thank my examining committee members, Cezar Câmpeanu, Andrew C. Doxey, Ming Li, and Jeffrey O. Shallit, for reviewing and commenting on my thesis and attending my defence.

Finally, I thank my colleagues, Fatemeh Alipour, Wanxin Li, Pablo Millan Arias, Timothy Ng, and Gurjit S. Randhawa, for fostering a constructive research environment and collaborators, Francesco Bellamoli, Srujan K. Enaganti, Giuditta Franco, Manasi S. Kulkarni, Silvia Lampis, Kalpana Mahalingam, Dan V. Nicolau, Palak Pandoh, Ayyappasamy S. Perumal, and Falco C. M. J. M. van Delft, for fruitful collaboration.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The central dogma of molecular biology explains the flow of genetic information within a biological system, where the information in DNA can be transferred to DNA or to protein molecules via RNA. Although proteins are the building blocks of life, they are created according to the information stored in DNA. Similarly to the information stored as bits in electronic computers, the information in DNA can also be manipulated and used for computation, and this area of research is called *DNA computing*. Research in this area started about thirty years ago when Adleman solved a seven-city instance of the *directed Hamiltonian path problem* using DNA [3]. This new approach inspired the study of DNA computing both theoretically and experimentally. In this thesis, we study three separate but inter-related research topics in DNA computing: modelling biology processes as a formal language operation, studying combinatorics on words inspired by DNA, and complexity estimation of DNA computing experiments.

Recall that a formal model of computation, called the *Turing machine*, was invented by Alan Turing in 1936, and the Church-Turing thesis states that all effective computations can be carried out by a Turing machine [52]. There are also some formal systems defined by some biologically inspired word operations that have been shown to have computational power equivalent to that of the Turing machine, such as those in [150,152,165]. Therefore, modelling DNA processes as bio-operations and studying these bio-operations as formal language operations are the first steps, which may lead to the creation of a DNA-based computer. The DNA processes that can be used to implement computations include naturally-occurring biology phenomena, such as the actions of enzymes on DNA strands, and biology experiment protocols and techniques. For example, the action of DNA polymerase was modelled as *template-directed extension* [74], and gene recombination by the lab protocol called *cross-pairing polymerase chain reaction (XPCR)* was formalized as *overlap assembly* [73]. These operations were studied as formal language operations, and properties, such as closure property, decidability, and state complexity, were considered. In this thesis, we first define and study two bio-operations inspired by the unexpected outcome of XPCR when the input strands are of a particular type.

Another research direction of theoretical DNA computing is related to combinatorics on words, which dates back to the early 1900s when Axel Thue proved the existence of square-free words of an arbitrarily large length [20]. The field of combinatorics on words studies the structure of words, such as borderedness, palindromes, repetitions, and counting. Combining the field of DNA computing and the field of combinatorics on words offers an exciting opportunity to study the structure of words that can be generated by systems defined by some bio-operations. For example, the sequence of Fibonacci words obtained from two initial letters by a recursive definition was well studied in the field of combinatorics on words. In this thesis, we then study a generalization of the Fibonacci words motivated by ideas stemming from theoretical studies of DNA computing.

For electronic computers, different algorithms for a problem can be compared using their time and space complexity, and a similar idea can be applied to DNA computing procedures and other natural computing algorithms for solving computational problems. For the experimental aspect of DNA computing, the time, space, and energy complexity of a DNA computing procedure for a benchmark problem can be considered. Among potential benchmark problems, those belonging to the class referred to as *nondeterministic polynomial time complete (NP-complete)* are of most significant interest since they are related to practical problems in logic, graphs, arithmetic, biology, and chemistry. Finally, in this thesis, a scalability comparison among the classical electronic computing approach, a DNA computing algorithm, and a network biocomputing solution (another type of natural computing that uses biotic motile agents to implement computations) to a benchmark problem called the *subset sum problem (SSP)* will be given.

1.1 Thesis overview

This thesis is organized as follows.

Chapter 2 gives an overview of DNA computing. It starts with a brief biological background on DNA, followed by a description of the three main steps of a DNA computing procedure. It ends with an example of a DNA computing procedure implemented to solve a seven-node instance of the directed Hamiltonian path problem. This chapter is based on the paper [215] published in *New Journal of Physics*.

In Chapter 3, we first introduce formal language operations and some biologically inspired word operations. After reviewing the details of a lab protocol used to extract DNA strands, we discuss a bio-operation that models its outcome under normal circumstances. A generalization of the unexpected outcome of this lab protocol in case the input strands are of a particular type is modelled as a word operation called *word blending*, and another word operation that is more closely related to the experimental requirements is studied, followed by the description of the wet lab experiments that motivate these operations. This chapter is based on the paper [76] published in *Fundamenta Informaticae* and the paper [15] published in *Natural Computing*.

Chapter 4 gives a unified terminology of different types of Fibonacci words studied in the literature and defines several generalizations of Fibonacci words, called *involutive Fibonacci words*, inspired by DNA Watson-Crick complementarity. The interrelationships between involutive Fibonacci words and the borderedness and primitivity of involutive Fibonacci words are discussed. This chapter is based on the papers [145,149] published in *Journal of Automata, Languages and Combinatorics*.

Chapter 5 gives a scaling comparison of DNA computing, network biocomputing, and electronic computing approaches to the subset sum problem (SSP). After describing a DNA computing approach and a network biocomputing approach to SSP, the pre-computing costs, the volume needed for the computation, the run time, and the energy costs of different approaches for SSP are compared. This chapter is based on the paper [215] published in *New Journal of Physics*.

In Chapter 6, we conclude this thesis by reviewing the results and by suggesting future work.

1.2 Notation

This section lists basic notation used throughout this thesis. Notation used only within a chapter will be described in the beginning of that chapter.

An *alphabet* Σ is a finite non-empty set of symbols. We let Σ^* denote the set of all words over an alphabet Σ, including the *empty word* λ, and Σ^+ denotes the set of all non-empty words over an alphabet Σ. If α and β are two words over an alphabet Σ, their concatenation $\alpha\beta$ is also a word over Σ. The empty word λ is an identity with respect to concatenation, where for all words $\alpha \in \Sigma^*$, we have $\alpha\lambda = \lambda\alpha = \alpha$. If α is a word over an alphabet Σ, the word α^i, $i \in \mathbb{N}$, is a word over Σ obtained by concatenating i copies of the word α. For all words $\alpha \in \Sigma^*$, we have $\alpha^0 = \lambda$, and for all numbers $i \in \mathbb{N}$, we have $\lambda^i = \lambda$.

The reverse or mirror image of a word is defined as $\lambda = \lambda^r$ and $(a_1a_2\cdots a_n)^r = a_n\cdots a_2a_1$, where $a_i \in \Sigma$ for all $1 \leq i \leq n$. The length of the word w is denoted by $|w|$, and the number of occurrences of the letter a in a word w is denoted by $|w|_a$. For a positive integer i, we let Σ^i denote the set of all words of length i over Σ. For words $w, x, y, z \in \Sigma^*$, where $w = xyz$, we call the subwords x, y, and z *prefix*, *infix*, and *suffix* of w, respectively. The sets $\text{pref}(w)$, $\text{inf}(w)$, and $\text{suff}(w)$ contain all prefixes, infixes, and suffixes of w, respectively. A prefix, suffix, or infix α of a word w is said to be proper if $\alpha \neq w$, and it is said to be non-empty if $\alpha \neq \lambda$. The sets $\text{Pref}(w)$, $\text{Inf}(w)$, and $\text{Suff}(w)$ contain all proper prefixes, infixes, and suffixes of w, respectively. The sets $\text{pref}'(w)$ and $\text{suff}'(w)$ contain all non-empty prefixes and suffixes of w, respectively. For a word α, the set of all letters that occur in α is denoted by $\text{Alph}(\alpha)$. A *language* L is a subset of Σ^*. The complement of a language $L \subseteq \Sigma^*$ is $L^c = \Sigma^* \setminus L$, the concatenation of two languages L_1 and L_2 is defined as $L_1L_2 = \{uv \mid u \in L_1, v \in L_2\}$, and the power of a language L is defined as $L^0 = \{\lambda\}$, $L^1 = L$, and $L^i = L^{i-1}L$ for all integers $i \geq 2$.

Binary formal language operations $\diamond$ can be considered as a function $\diamond : 2^{\Sigma^*} \times 2^{\Sigma^*} \to 2^{\Sigma^*}$. Similarly, unary formal language operations $\diamond$ can be viewed as a function $\diamond : 2^{\Sigma^*} \to 2^{\Sigma^*}$. A binary word operation $\Box^r$ is called the reversed operation of $\Box$ if for all $\alpha, \beta \in \Sigma^*$, we have that $\alpha \Box^r \beta = \beta \Box \alpha$.

In this thesis, we use the following conventions for variables unless noted otherwise:

- Integer variables are denoted by some lowercase letters from the middle of the English alphabet, such as i, j, and k.
- Word variables are denoted by some letters from the Greek alphabet and some lowercase letters from the end of the English alphabet, such as α, β, u, v, and w.
- Letter variables are denoted by lowercase letters from the beginning of the English alphabet, such as a, b, and c.
- Set variables are denoted by upper case letters from the English alphabet, such as L and S.

Chapter 2

Overview of DNA Computing

This chapter gives an overview of DNA computing. In Section 2.1, a concise biological background on DNA is given. In Section 2.2, the three main steps of a DNA computing procedure are described. The first step is to encode information as DNA, as detailed in Section 2.2.1. The second step is the DNA computation, which consists of a succession of bio-operations, and some of the main bio-operations used in the DNA computing experiments are listed in Section 2.2.2. The last step is reading the output of the DNA computation, as detailed in Section 2.2.3. In Section 2.3, an example of a DNA computing procedure implemented to solve a seven-node instance of the directed Hamiltonian path problem is described.

Parts of Section 2.2 and Section 2.3 are adapted from a paper I co-authored [215], titled "As good as it gets: A scaling comparison of DNA computing, network biocomputing, and electronic computing approaches to an NP-complete problem."

2.1 Biological background

As shown in Figure 2.1 (right panel), a *deoxyribonucleic acid (DNA)* double strand is composed of two chains of *nucleotides*, and these chains are called *DNA single strands*. Each nucleotide (Figure 2.1, left panel) is composed of a nitrogenous base called a *nucleobase*, a sugar, and a phosphate group. There are four kinds of nucleobases: *adenine (A)*, *cytosine (C)*, *guanine (G)*, and *thymine (T)*. A nucleotide with the nucleobase A (respectively C, G, and T) is called nucleotide A (respectively C, G, and T). Illustrated in Figure 2.2, there are five carbon atoms labelled from $1'$ to $5'$ in the sugar, and the sugar contains a hydroxyl group (composed of one hydrogen atom and one oxygen atom) connected to its 3′ carbon. The nucleobase is connected to the 1′ carbon, and the phosphate group is connected to the 5′ carbon.

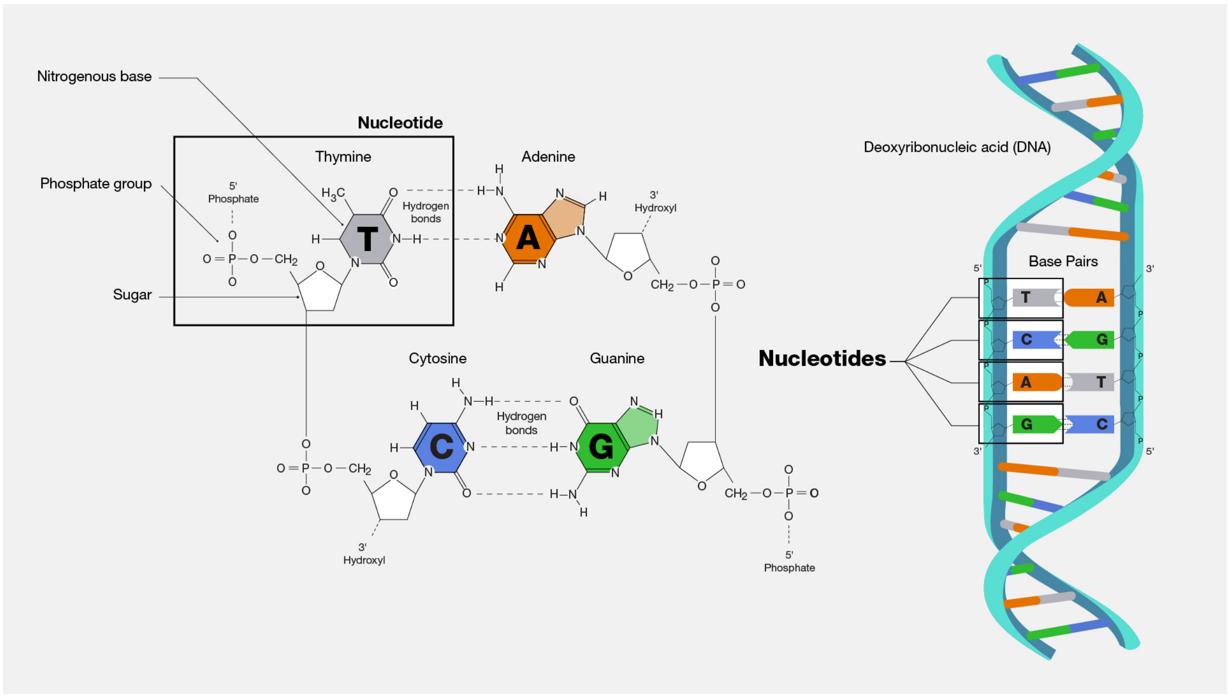


Figure 2.1: DNA double strand [29]. Left panel: The part surrounded by the rectangle is one example of a nucleotide. A nucleotide is composed of a nitrogenous base, a phosphate group, and a sugar. There are four kinds of nucleotides differentiated by their nucleobases (adenine [A, orange], cytosine [C, blue], guanine [G, green], and thymine [T, grey]). Note that the sizes of different nucleobases are different, and nucleobases A and C are larger than nucleobases G and T. The labelling of carbon atoms in a nucleotide is shown in Figure 2.2. Two nucleotides can be connected by a covalent bond (a solid line) between the sugar of one nucleotide and the phosphate group of the other. Two nucleotides with compatible nucleobases (T with A and C with G) can form hydrogen bonds (dashed lines) between their nucleobases, where two bonds are formed between T and A, and three bonds are formed between C and G. Right panel: A chain of nucleotides connected by covalent bonds is a DNA single strand. A DNA single strand is directional, and its direction is from its 5′ end to its 3′ end. A DNA double strand is composed of two such strands connected by the hydrogen bonds between nucleobases.

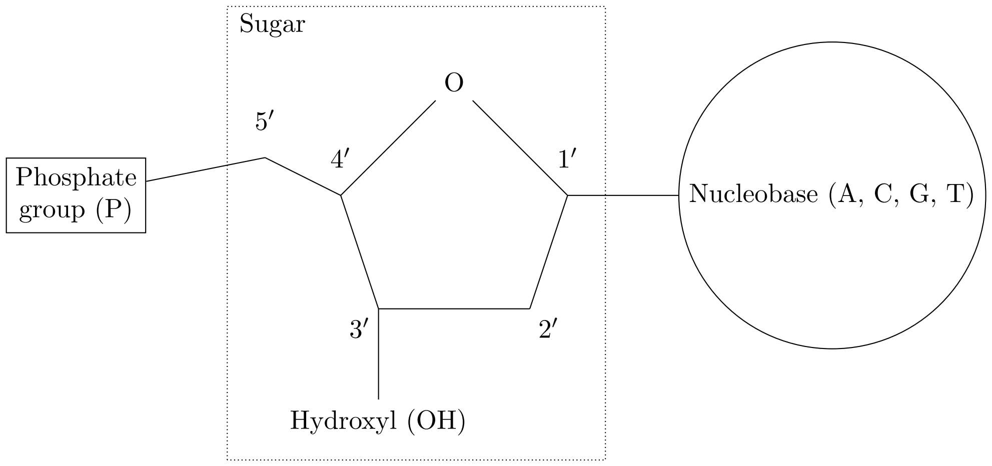


Figure 2.2: Labelling of carbon atoms in a nucleotide. A DNA nucleotide is composed of a nucleobase (adenine [A], cytosine [C], guanine [G], or thymine [T]), a sugar (containing a hydroxyl group), and a phosphate group. There are five carbon atoms in the sugar (represented by junctions of lines), and they are labelled from $1'$ to $5'$. The nucleobase is connected to the $1'$ carbon, the hydroxyl group is connected to the $3'$ carbon, and the phosphate group is connected to the $5'$ carbon.

Two neighbour nucleotides in a DNA single strand are connected by a covalent bond between the hydroxyl group of the sugar of one nucleotide and the phosphate group of the other. The alternating sugars and phosphate groups connected by covalent bonds form what is called the *backbone* of a DNA single strand. A DNA single strand has an unconnected phosphate group on a nucleotide at one end, and this end is called the $5'$ end of this DNA single strand. The other end of the DNA single strand is called the $3'$ end of this DNA single strand.

In this thesis, two types of simplified graphics are used to represent DNA single strands, as shown in Figure 2.3.

The notational convention used in this thesis is that a word $a_1a_2 \cdots a_n$ over the DNA alphabet $\Delta = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$ represents the DNA single strand $a_1a_2 \cdots a_n$ read in the $5'$ to $3'$ direction. For example, the DNA single strands in Figure 2.3 is represented by the word CGTACGTACG.

Two DNA single strands can form a DNA double strand by Watson-Crick complementarity, as detailed below. Nucleotides A can base pair with nucleotides T, and nucleotides C can base pair with nucleotides G. This is called the *Watson-Crick complementarity of bases*, and it connects two nucleotides by creating hydrogen bonds between their nucleobases (rep-

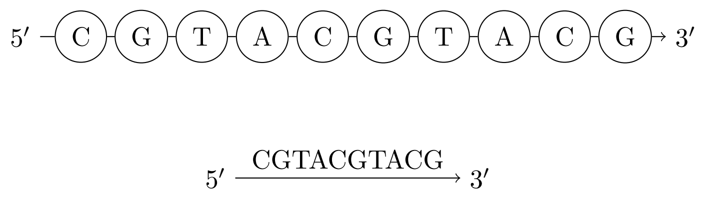


Figure 2.3: Simplified graphical representations of a single-stranded DNA molecule CG-TACGTACG. Top panel: Each circle represents a nucleotide, and its type is denoted by the letter in the circle. The 5′ end is the end labelled with 5′ and no arrow, and the 3′ end is the end labelled with 3′ and an outgoing arrow. The solid lines between nodes represent the backbone. Bottom panel: A similar concept is adopted, but the sequence of nucleotides is represented by a word instead.

resented by dashed lines between nucleobases in Figure 2.1). Note that the nucleobases A and G are larger, and the nucleobases T and C are smaller. Each pair of complementary bases contains a larger base and a small base. Two hydrogen bonds are formed between A and T, and three hydrogen bonds are formed between C and G. Two DNA single strands of opposite orientations and with complementary nucleobases at each position binding together via hydrogen bonds form a DNA double strand in a process called *Watson-Crick base pairing*. Formally, two DNA single strands $\alpha = a_1 a_2 \cdots a_n$, $\beta = b_1 b_2 \cdots b_n$ of the same length $n \in \mathbb{N}$ over the DNA alphabet Δ are said to be Watson-Crick complementary to each other if $\langle a_i, b_j \rangle \in \{\langle \mathrm{A}, \mathrm{T} \rangle, \langle \mathrm{T}, \mathrm{A} \rangle, \langle \mathrm{C}, \mathrm{G} \rangle, \langle \mathrm{G}, \mathrm{C} \rangle\}$, where $1 \leq i \leq n$ and $j = n + 1 - i$. For a DNA single strand $\alpha \in \Delta^*$, its Watson-Crick complement is denoted by $\overline{\alpha}$. For example, two DNA single strands CCTAGGTACG and CGTACCTAGG are Watson-Crick complementary to each other as shown in Figure 2.4.

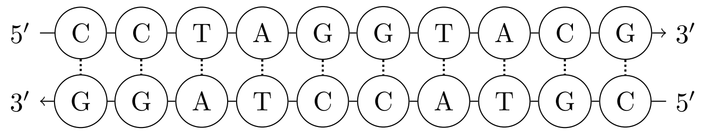


Figure 2.4: Two Watson-Crick complementary DNA single strands CCTAGGTACG and CGTACCTAGG (see conventional notations for representing a DNA strand as a word) bound to each other via Watson-Crick complementarity to form a DNA double strand with blunt ends. Hydrogen bonds between two complementary nucleobases are represented by dotted lines.

If two DNA single strands are of equal length and are Watson-Crick complementary to each other, they form a DNA double strand with *blunt ends* (see Figure 2.4). It is also possible that two DNA strands are partially Watson-Crick complementary, in which case they can form *partially double-stranded* molecules, such as the one in Figure 2.5. In this case, the single-stranded portion of the partially double-stranded molecules is called an *overhang* or a *sticky end* since it has the potential to bind to yet another DNA single strand.

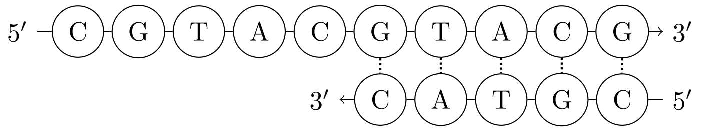


Figure 2.5: The DNA double strand with an overhang CGTAC consisting of two DNA single strands CGTACGTACG and CGTAC. This DNA double strand has one blunt end on the right and one sticky end on the left.

Similarly, if two subsequences of a DNA single strand are complementary, this strand may form intramolecular (intramolecular refers to the structures formed within a single DNA molecule, while intermolecular refers to the structures formed with two or more DNA molecules) structures, such as stem-loops, which are more commonly known as *hairpins* (Figure 2.6).

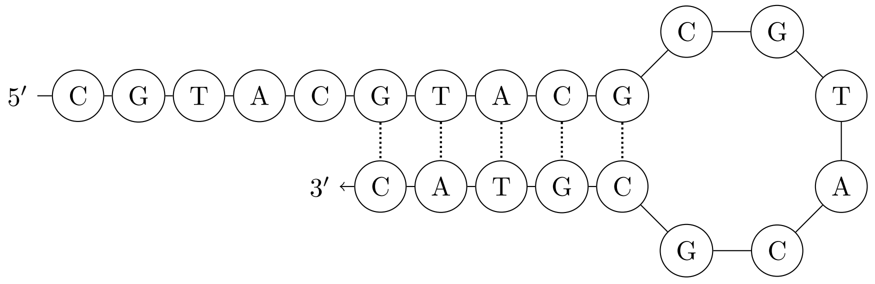


Figure 2.6: The DNA single strand CGTACGTACGCGTACGCGTAC forming a hairpin structure because of Watson-Crick complementarity, pairing G with C and A with T. The orientation of the DNA strand is denoted by its two ends labelled with 5′ and 3′ to indicate their different chemical characteristics.

In addition to the linear DNA single strands mentioned above, there are circular DNA single stands. Two circular DNA single strands with complementary sequences can form a circular DNA double strand called a *plasmid*. Plasmids can be found naturally in bacteria and some other microscopic organisms.

2.2 DNA computing

In the same way we use the letters of the English alphabet to write text, and electronic computers use bits 0 and 1 to store data, Nature uses the four nucleotides (A, C, G, and T) to write genetic information as DNA strands. The possibility of encoding the input of some computational problems as DNA strands and the proved ability of such biochemical processes as cutting and pasting of DNA strands to implement arithmetic and logic operations have led to the development of the field of DNA computing and molecular programming [3]. The three main steps of a DNA computing procedure, which are described in detail in the remainder of this section, are as follows:

1. Encoding the input information as DNA strands;
2. DNA computation; and
3. Reading the output of the DNA computation.

2.2.1 Step 1: Encoding information as DNA

To encode information using DNA, one first selects an encoding scheme, mapping symbols onto words over the DNA alphabet $\Delta = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$. For example, in [53], the letters of the Latin alphabet were encoded using the following triplets:

$$a = \mathrm{CGA}, b = \mathrm{CCA}, c = \mathrm{GTT}, d = \mathrm{TTG}, e = \mathrm{GGT},$$

$$n = \mathrm{TCT}, o = \mathrm{GGC}, r = \mathrm{TCA}, t = \mathrm{TTC}, \ldots$$

With this encoding, the English text "to be or not to be" becomes the DNA single strand shown in Figure 2.7.

$$5' \xrightarrow{\text{TTC GGC CCA GGT GGC TCA TCT GGC TTC TTC GGC CCA GGT}} 3'$$

Figure 2.7: The DNA single strand encoding the English text "to be or not to be" using the encoding scheme described in text.

Then, DNA single strands that encode information can be synthesized. DNA synthesis is one of the most basic bio-operations used in DNA computing. DNA solid-state synthesis is based on a method by which the initial nucleotide is bound to a solid support, and successive nucleotides are added step-by-step, from the $3'$ end to the $5'$ end, in a reactant solution. While the above encoding example is purely hypothetical, DNA strands of lengths of up to 1,000 base pairs (bp) can be readily synthesized, using fully automated DNA synthesizers, in under 24 hours [163]. The synthesis of DNA strands longer than 10 kilo base pairs (kbp) is more challenging and time consuming. There are two major approaches to synthesizing such strands—namely, chemical synthesis and enzymatic synthesis; these approaches both involve multiple rounds of synthesis of short DNA fragments, followed by their assembly into longer DNA strands. A commonly used tool like BioBrick assembly [105] could take 30 to 45 days to synthesize DNA strands longer than 10 kbp.

It should be noted that, in most DNA computing experiments, DNA-encoded information is not associated with a memory location but consists of infinitesimal free-floating DNA strands in a solution. Because of Watson-Crick complementarity, these free-floating DNA strands can interact with each other in ways that are programmed but also in ways that are undesired. For example, information-encoding DNA single strands can contain Watson-Crick complementary substrands within themselves causing them to form hairpins and rendering them unavailable for Watson-Crick complementarity interactions with other information-encoding strands. Several DNA computing studies explore choices for encoding information as DNA strands that ensure that undesirable base pairing does not occur [97, 133, 144, 181, 185, 212, 225, 266]. Exceptions are surface-based DNA computing experiments, such as those reported in [40, 169], wherein data-encoding DNA strands are fixed to a solid surface.

2.2.2 Step 2: DNA computation

A DNA computation consists of a succession of bio-operations. In this section, we list some of the main bio-operations used in DNA computing experiments.

Watson-Crick base pairing, also called *hybridization* or *annealing*, occurs because of the Watson-Crick complementarity of bases (A with T and C with G), and it is the principal bio-operation underlying most—if not all—DNA computations. Watson-Crick base pairing connects two nucleotides by creating hydrogen bonds between their nucleobases (three hydrogen bonds between G and C and two hydrogen bonds between A and T). Since a hydrogen bond is weaker than the covalent bond in the backbone of DNA molecules, it will break when subjected to a high temperature. The process using heat to break DNA double strands into their constituent DNA single strands is called *melting* or *denaturation*. Watson-Crick base pairing was utilized in the first proof-of-concept DNA computing experiment, which solved a seven-node instance of the directed Hamiltonian path problem [3].

It was also used in the DNA computing experiment that solved a 20-variable instance of the 3-*satisfiability problem*, which marked the first instance of a DNA computation solving a problem beyond the normal range of unaided human computation [25]. Finally, Watson-Crick base pairing was the crucial computational primitive used in the algorithmic self-assembly of DNA tiles utilized, such as for the implementation of logic gates [180], stochastic computing [2], and cellular automata [238], and it was used in molecular algorithms using reprogrammable DNA self-assembly [275].

Cutting, also called *restriction enzyme digestion*, is a bio-operation implemented by restriction endonuclease enzymes. A restriction enzyme cuts double-stranded DNA into fragments at or near an enzyme-specific pattern known as the *restriction site*, and the location of cutting is called the *restriction point*. Restriction enzyme digestion is inexpensive from the point of view of energy consumption because no external energy is needed [272]. The result of cutting a DNA double strand at a restriction site (by cutting the backbones of each of its two component single strands) is either two DNA double strands with new complementary sticky ends or two DNA double strands with new blunt ends. For example, the restriction site of the restriction enzyme *EcoRI* is shown in Figure 2.8, and the restriction site of the restriction enzyme *SmaI* is shown in Figure 2.9. Note that the vertical arrows indicate the restriction point. A cut by the restriction enzyme EcoRI creates two new sticky ends AATT, whereas a cut by the restriction enzyme SmaI creates two blunt ends.

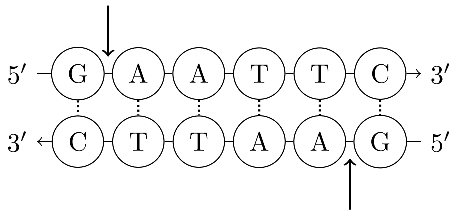


Figure 2.8: The restriction site of the restriction enzyme EcoRI is GAATTC. The restriction point is between the first and second nucleotides from the 5′ end of each of the DNA single strands. The result of the restriction enzyme digestion is two new partially double-stranded DNA strands, each with a sticky end AATT.

Cutting was used to generate DNA molecules encoding a potential solution for the subset sum problem [102], the maximal independent set problem [97], and the minimal dominating set problem [103]. Some enzymes cut non-specifically, outside their restriction site, and they have also been employed for computations, such as in the wet lab implementation of a programmable finite automaton using the *FoKI* enzyme [16, 18].

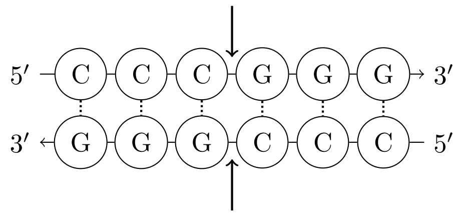


Figure 2.9: The restriction site of the restriction enzyme SamI is CCCGGG. The restriction point is between the third and fourth nucleotides from the 5′ end of each of the DNA single strands. The result of the restriction enzyme digestion is two new DNA double strands with blunt ends.

Pasting, also called *ligation*, is a bio-operation that accomplishes the opposite of cutting, and this is implemented by DNA ligase enzymes that can join together DNA strand backbones. Some DNA ligases join together DNA double strands with complementary sticky ends, whereas others join together blunt-ended DNA double strands. For example, in Figure 2.10, ligation can join a partially double-stranded DNA strand with a sticky end GCTA and a partially double-stranded DNA strand with a sticky end TAGC.

Compared with cutting, ligation is energy intensive because ligases consume *adenosine triphosphate (ATP)* (ATP is an organic compound and an energy-storing molecule; it is considered the "universal battery" because it provides the energy required by many processes in living cells, such as muscle contraction, nerve impulse propagation, and chemical synthesis). One ATP molecule is consumed while connecting each pair of nucleotides on the same strand by a new covalent bond, so the process shown in Figure 2.10 requires two ATP molecules. Ligation was used in [3] to generate potential solutions for an instance of the directed Hamiltonian path problem. It was also used with restriction enzyme digestion to remove part of a DNA double strand for the *maximal independent set problem* [97].

Polymerase chain reaction (PCR) is often used in DNA computing experiments to amplify (produce an exponential increase of) DNA double strands called *templates*. PCR is affected by the DNA polymerase enzyme, which extends a *primer* (short DNA sequence that base pairs to the 3′ end of one of the single strands of the template) nucleotide by nucleotide until it produces a full Watson-Crick complement of the template. As shown in Figure 2.11, the inputs of PCR include a template (DNA double strand in the middle of the first row of Figure 2.11), two primers (DNA single strands α and $\overline{\gamma}$ at the left and right, respectively, of the first row of Figure 2.11), nucleotides (not shown), and DNA polymerase (not shown). The primer $\overline{\gamma}$ can hybridize with one strand of the template $\alpha\beta\gamma$ at the 3′ end, and the primer α can hybridize with the other strand of the template $\overline{\gamma}\overline{\beta}\overline{\alpha}$ at the 3′ end. PCR involves multiple cycles, and each cycle includes denaturation, annealing, and

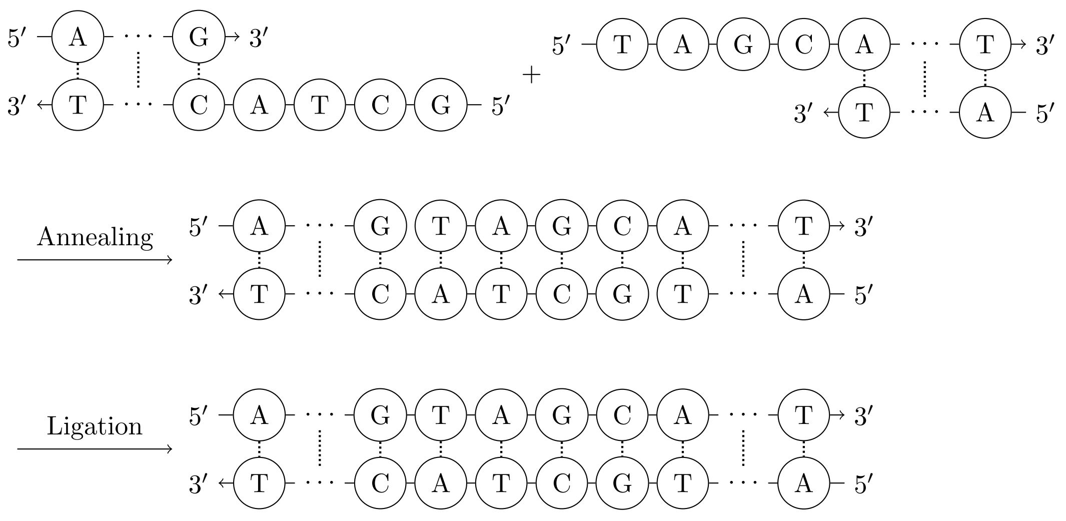


Figure 2.10: Two partially double-stranded DNA strands with complementary sticky ends can be joined to form one DNA double strand by ligation. The input to ligation consists of two partially double-stranded DNA strands with sticky ends GCTA (top left) and TAGC (top right). In the process of ligation, the sticky ends hybridize with each other due to Watson-Crick complementarity (middle), and the enzyme ligase connects the broken backbones to form a DNA double strand (bottom).

extension. During denaturation, the DNA double strands break into their constituent DNA single strands. This is followed by annealing, caused by the decrease of the temperature of the mixture, during which the primers hybridize with the DNA single strands from the template. The last step of this cycle is extension, during which DNA polymerase extends the primers in the $3'$ to the $5'$ direction using the free-floating nucleotides present in the solution until it produces a full complement of each single strand comprising the template. After one cycle of PCR, one copy of the template becomes two copies of the template, and 2^n copies of the template can be generated after n cycles. Whiplash PCR (a variation of PCR where the template and the primers are on the same DNA single strand) has been specifically used in the implementation of a finite state machine by hairpin formation [242]. When used in non-standard ways, PCR can produce a combinatorial richness of molecules, but it may also behave in ways which are complex and difficult to control [111, 134, 175].

Cloning is another method of amplifying so-called “target genes” (DNA double strands). Here, a target gene is inserted into a plasmid to create a recombinant DNA molecule, consisting of the initial plasmid and the inserted target gene, as shown in Figure 2.12. A restriction enzyme is chosen to cut the plasmid and the target gene with the property

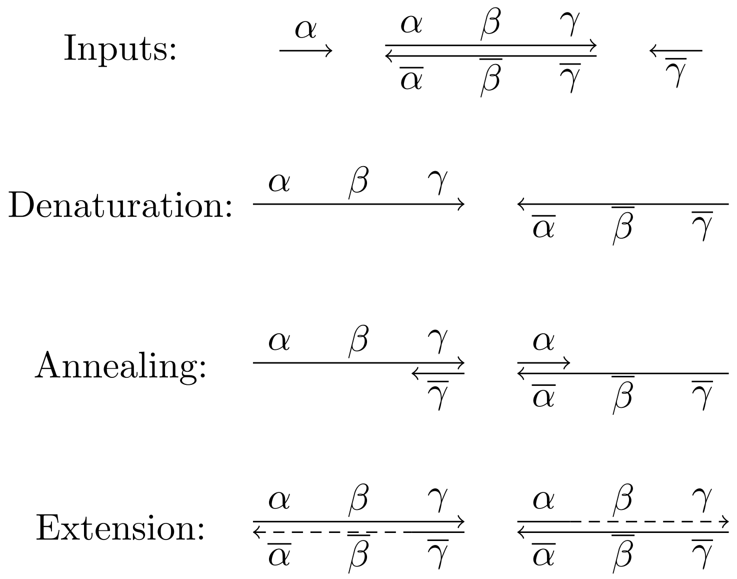


Figure 2.11: One cycle of polymerase chain reaction (PCR) generates two copies of the template. The inputs include a template (DNA double strand in the first row), two primers (DNA single strands α and $\overline{\gamma}$ in the first row), nucleotides (not shown), and DNA polymerase (not shown). One cycle of PCR has three steps: denaturation, annealing, and extension. During the denaturation step, the template breaks into single strands $\alpha\beta\gamma$ and $\overline{\gamma}\overline{\beta}\overline{\alpha}$ because of an increase in temperature. During the annealing step, the single strands from the template hybridize with primers ($\alpha\beta\gamma$ with $\overline{\gamma}$ and $\overline{\gamma}\overline{\beta}\overline{\alpha}$ with α) because of the decrease in temperature. Finally, during the extension step, the primers are extended by DNA polymerase using the free-floating nucleotides present in the solution according to the template strands.

that its recognition site has one occurrence on the plasmid and occurs once on each end of the target gene. After restriction enzyme digestion, the plasmid and the target gene have complementary sticky ends, and they can hybridize to form a recombinant DNA molecule. This process is used in the traditional cloning procedure, and difficulties arise if suitable natural restriction sites are unavailable on the target gene. There are other methods to create a recombinant DNA molecule that avoid this problem [22, 265, 282]. This recombinant is then introduced to a bacterium, such as *Escherichia coli (E. coli)*. Since bacteria multiply exponentially, this process will lead to an exponential increase in the target gene contained in their recombinant plasmids. This bio-operation was used in a DNA computing algorithm to solve a six-bit instance of the *maximal clique problem* [97].

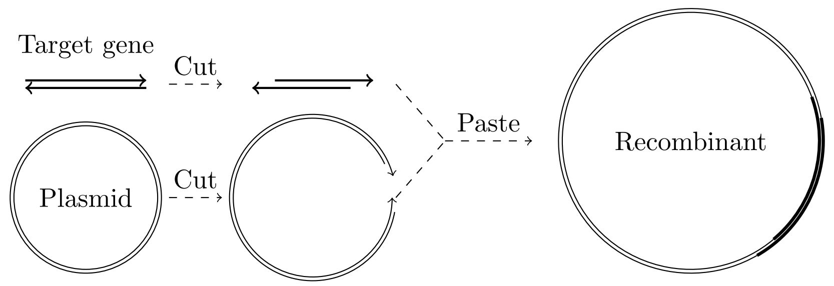


Figure 2.12: The process to create a recombinant circular DNA molecule from a DNA double strand containing the target gene and a plasmid. The target gene is cut by a restriction enzyme whose restriction site occurs once on each end of the target gene. The same restriction site occurs once on the plasmid, and the plasmid is cut by the same restriction enzyme. After these cuts, the resulting strands from the target genes and the plasmid have complementary sticky ends. Therefore, ligation can be used to create a recombinant plasmid consisting of the initial plasmid and the inserted target gene.

Gel electrophoresis is used to separate DNA strands by length. DNA molecules are negatively charged because they have a backbone of negatively charged phosphate groups Therefore, if DNA molecules are put in a gel with an electric current applied, they will migrate toward the positively charged end. Short DNA molecules move through a gel faster because they are able to better navigate the small space available, so smaller DNA molecules move more quickly toward the positively charged end than larger DNA molecules do. It was shown in [101] that the speed of migration of a DNA molecule in a gel is inversely proportional to the log of the length of the DNA molecule. The procedure of gel electrophoresis is shown in Figure 2.13. At one end of the gel, there are wells where the DNA samples will be loaded. The negative electrode is placed at the end of the gel with wells, and the positive electrode is placed at the other end of the gel. After migrating through the gel, the DNA molecules are separated by size. A group of DNA molecules of the same length is visible as a band on the gel if its total weight is at least one nanogram [90,129]. A band is darker if there are more DNA molecules of that length. There are high-resolution types of electrophoresis (e.g., non-denaturing polyacrylamide gel electrophoresis [PAGE] or 2% agarose gel electrophoresis) that can differentiate DNA molecules with as small as a one base pair difference in length. This bio-operation was used for selecting the DNA strands encoding the solution to the directed Hamiltonian path problem in [3] and for selecting the solution to the subset sum problem in [102].

Extraction is used to select DNA strands that contain a certain pattern α as a substrand

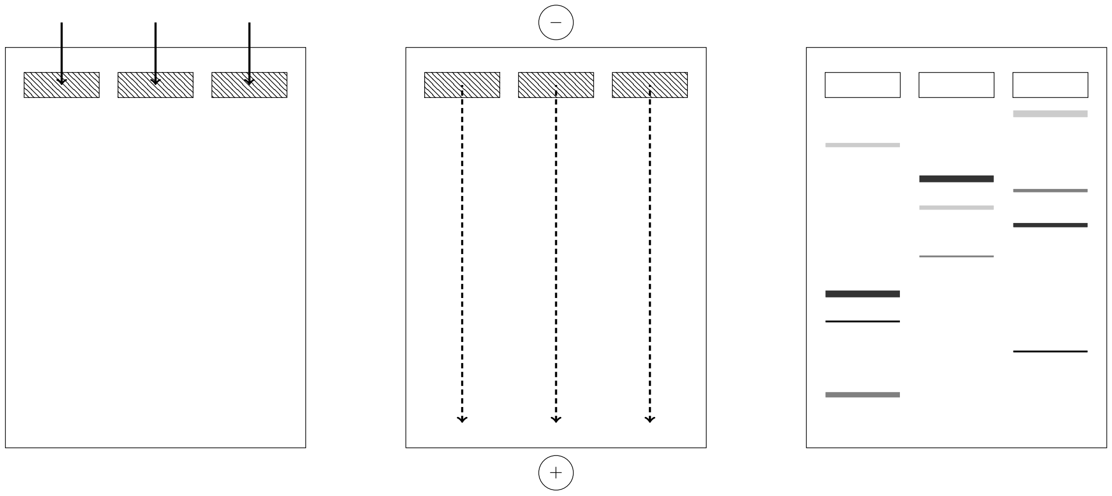


Figure 2.13: Gel electrophoresis procedure [167]. Left panel: DNA samples are loaded into the wells (small rectangles with shades on the top) at one end of the gel (big rectangle). Middle panel: An electric current is applied to the gel, with the negative electrode placed at the end of the gel with the wells and the positive electrode placed at the other end of the gel. Right panel: The DNA molecules migrate toward the positively charged end and are separated by size. The shortest DNA molecules migrate the farthest from the wells, while the longest remain closest to the wells. DNA molecules are visible as bands, and a band is darker if there are more DNA molecules of that length.

using hybridization between DNA molecules. A DNA single strand $\overline{\alpha}$ is created and chemically immobilized to a solid surface. Therefore, when a mixture of DNA strands is passed over the surface, those DNA molecules with the Watson-Crick complementary substrand α are connected to the surface by hybridization with the immobilized strands. After other DNA strands are washed away, the remaining DNA molecules contain the pattern α. This bio-operation was used, for example, in [3] to select the paths that pass through a given node.

DNA strand displacement is a more recent bio-operation, and it can be polymerase-based or toehold-mediated. The input of both types of DNA strand displacement consists of a partially double-stranded DNA molecule called the input strand and a DNA single strand called the fuel strand. The input strand contains a single-stranded substrand that is Watson-Crick complementary to part of the fuel strand, so they can hybridize to form a DNA complex. For toehold-mediated DNA strand displacement, a single strand of the input strand is replaced by the fuel strand because the new DNA duplex is thermodynamically favoured. An example is shown in Figure 2.14, where the input strand is a partially double-stranded DNA molecule, and it has an overhang that is complementary to one end of the fuel strand. However, as shown in Figure 2.15, for polymerase-based DNA strand displacement, a single strand of the input strand is replaced by a new strand created by DNA polymerase with the fuel strand as the primer and the other strand of the input strand as the template.

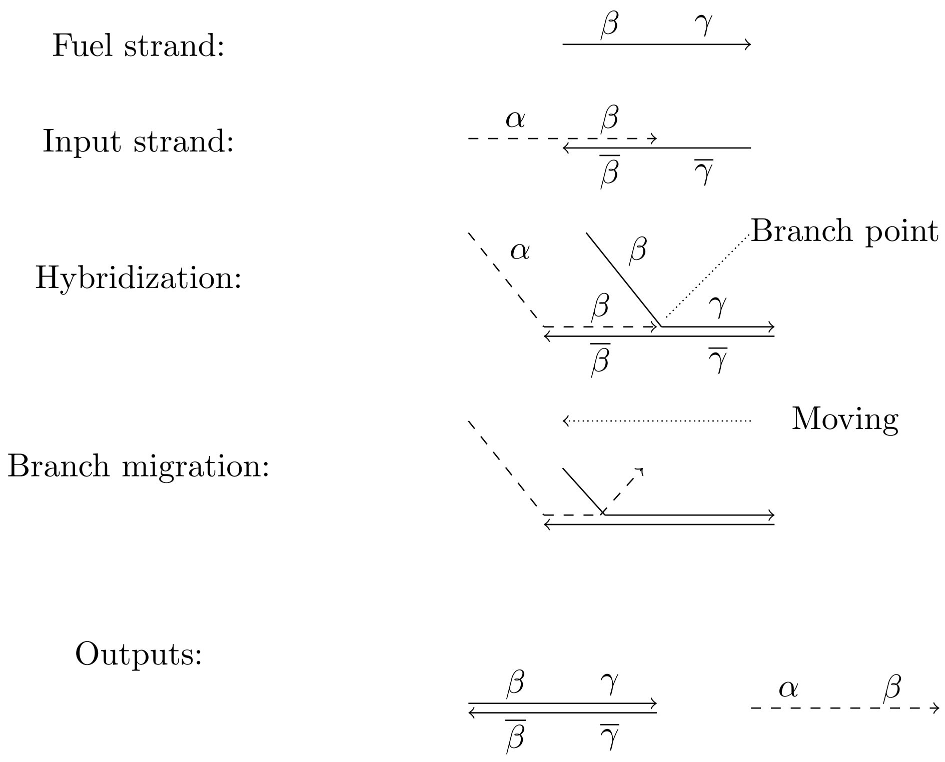


Figure 2.14: Toehold-mediated DNA strand displacement. The input consists of a partially double-stranded DNA molecule (input strand) and a DNA single strand (fuel strand). The input strand has strands $\alpha\beta$ and $\overline{\beta\gamma}$ and overhangs α and $\overline{\gamma}$, and the fuel strand is $\beta\gamma$. They can hybridize to form a DNA complex involving three single strands, where the "bottom strand" of the input strand ($\overline{\beta\gamma}$) hybridizes with the "top strand" of the input strand $\alpha\beta$ as well as the fuel strand ($\beta\gamma$). The branch point is the point where the invading fuel strand meets the "top strand" of the input strand. The invading fuel strand displaces the "top strand" of the input strand, and this moves the branch point in a process called branch migration. Finally, the "top strand" of the input strand is displaced, and a new recombinant DNA double strand is produced.

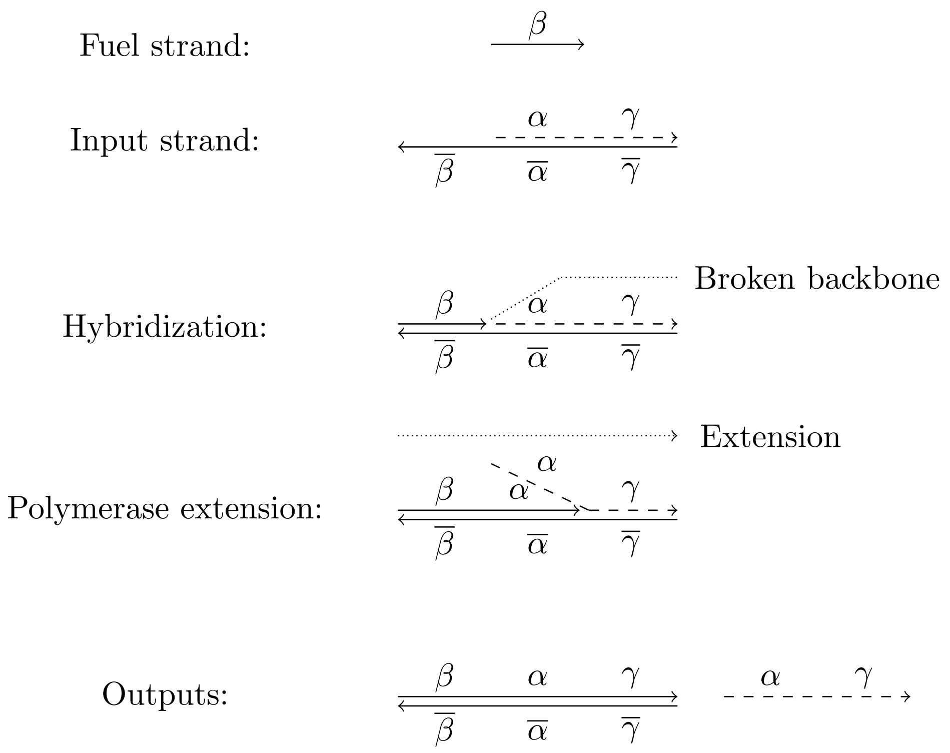


Figure 2.15: Polymerase-based DNA strand displacement. The input consists of a partially double-stranded DNA molecule (input strand) and a DNA single strand (fuel strand). The input strand has strands $\alpha\gamma$ and $\overline{\beta\alpha\gamma}$ and a sticky end $\overline{\beta}$, and the fuel strand is β. They can hybridize to form a DNA complex involving three single strands, where the "bottom strand" of the input strand ($\overline{\beta\alpha\gamma}$) hybridizes with the "top strand" of the input strand ($\alpha\gamma$) and the fuel strand (β). Next, polymerase extends the invading fuel strand toward the $3'$ end using the bottom strand as the template. After the polymerase extension, the substrand $\alpha\gamma$ is displaced, and a DNA double strand $\beta\alpha\gamma$ is produced.

DNA strand displacement was used to implement DNA-based artificial neural networks that can recognize 100-bit hand-written digits [42,227] and to implement artificial biochemical circuits [283]. It has also been used for computing the square root of a number [226,255], for building full adders and logic gates [258], and for implementing chemical reaction networks [250]. In addition, DNA strand displacement is used in so-called *localized DNA computing*, whereby logic circuits are implemented by spatially arranging reactive DNA hairpins on a DNA origami surface [40], potentially reducing the computation time by orders of magnitude. Among these examples, toehold-mediated DNA strand displacement was used in [40,42,226,227,283], and polymerase-based DNA strand displacement was used in [250, 255, 258].

2.2.3 Step 3: Reading the output of the DNA computation

At the end of a DNA computation, the DNA strands representing the output of the computation can be read out and decoded.

In the case of computational problems with yes/no answers, solving the problems could amount to the verification if a solution exists instead of asking for a solution. In this case, solving them could amount to the verification of the presence or absence of a certain solution-encoding DNA strand in the test tube that contains the output of the DNA computation. In such problems, gel electrophoresis can be used to detect DNA molecules of a specific length, and extraction can be used to detect DNA strands that contain a certain pattern as a substrand. If a solution is required, sequencing can be used to read out the information stored in the output DNA strands.

One sequencing method uses special chemically modified nucleotides (dideoxynucleoside triphosphates [ddNTPs]); these act as "chain terminators" during PCR as follows: A sequencing primer is annealed to the DNA template to be read. A DNA polymerase then extends the primer. The extension reaction is split into four tubes, each containing a different chain terminator nucleotide mixed with standard nucleotides. For example, a tube would contain chemically modified nucleotides C (ddCTP), as well as the standard nucleotides (dATP, dGTP, dCTP, and dTTP). In this tube, extension of the primer by the polymerase enzyme produces all prefixes ending in G (complement of C) of the complement of the original strand. A separation of these strands by length, using gel electrophoresis, allows the determination of the position of all Gs (complements of Cs). Combining the results obtained in this way for all four nucleotides allows the reconstruction of the original sequence.

Thousands of genomes have been sequenced since the completion of the Human Genome Project (HGP) [271] using a technique called *shotgun sequencing*. Here, short genome fragments called "reads" are sequenced; computer programs then use the overlapping ends of different reads to assemble them into a continuous sequence. Such conventional methods for reading the output of a DNA computation, including Illumina-based sequencing methods, are straightforward, rapid, and accurate (one percent maximum error), and DNA sequences of the size of the human genome (over three billion base pairs) can now be sequenced in one to three days [233]. In contrast with these sequential sequencing approaches, nanopore-based sequencing techniques parallelize the sequencing process, and despite being more error-prone (10% maximum error), they are now preferred for their speed, sequencer size, and convenience of use. Such single-molecule sequencers can process as many as 16,000 reads simultaneously using a parallel operating array of nanopores. Using nanopore sequencing methods, DNA strands of a total length of up to 90 million base pairs (with contiguous DNA strands as long as 60,000 bp being read) can be sequenced in 18 hours [37, 233].

2.3 Some NP-complete problems solved using DNA computing

A *decision problem* is a problem that can be phrased as a yes/no problem. A decision problem is said to be *polynomial-time solvable* if there exists a deterministic single-tape Turing machine that can solve the problem using polynomial time [254], and the complexity class P consists of all such problems. For example, the problem to decide whether a number is a prime number is in P [5]. A decision problem is said to be in *nondeterministic polynomial time* if a solution can be verified by a deterministic Turing machine in polynomial time [54], and the complexity class NP consists of all such problems. Equivalently, a decision problem is said to be in NP if it can by solved by some nondeterministic Turing machine in polynomial time [254]. In addition, a decision problem is said to be *NP-hard* if all NP problems are reducible to this problem in polynomial time. Informally, NP-hard problems are at least as hard as every NP problem. A decision problem is said to be in the class NP-complete if it is NP and NP-hard [234]. Each NP-complete problem has a solution space that grows exponentially with its size. The question of whether or not polynomial-time algorithms exist for NP-complete problems is one of the foremost open problems in computer science. If a polynomial time algorithm were found for a single NP-complete problem, all NP problems would be solvable in polynomial time. (It is widely thought that this is not the case.) Some NP-complete problems and reductions between these problems were listed in [155]. Thus, every NP-complete problem is significant; some of the most well-known NP-complete problems are defined below.

Many NP-complete problems are about graphs.

A *Hamiltonian path* is a path in a graph that visits each node exactly once. The path is said to be directed if the graph is directed, which means that the edges between nodes are directional; otherwise, it is said to be undirected. Given a (directed) graph, a start node, and an end node, the (directed) Hamiltonian path problem (HPP) asks if there exists a (directed) Hamiltonian path in the graph from the start node to the end node.

An *independent set of a graph* is a set of nodes such that there exist no edges between every pair of nodes in the set. A maximal independent set of a graph is an independent set such that, for each node outside the set, there exists an edge between this node and a node in the set. Given a graph, the maximal independent set problem (MIS) asks how many nodes are in a maximal independent set of the graph. This problem has a related NP-complete decision problem: Given a graph and a positive integer t, does there exist an independent set S of the graph such that $|S| = t$?

A *dominating set of a graph* is a set of nodes such that, for each node in the graph, either the node is in the set, or there exists an edge between the node and a node in the set. A dominating set of a graph is said to be minimal if it has the smallest cardinality among all dominating sets of that graph. Given a graph, the *minimal dominating set problem (MDS)* asks how many nodes are in a minimal dominating set of the graph. This problem has a related NP-complete decision problem: Given a graph and a positive integer t, does there exist a dominating set S of the graph such that $|S| = t$?

A *clique of an undirected graph* is a set of nodes such that there exists an edge between each pair of nodes in this set. A clique of a graph is said to be maximal if it has the largest cardinality among all cliques in that graph. Given a graph, the maximal clique problem asks how many nodes are in a maximal clique of the graph. This problem has a related NP-complete decision problem: Given a graph and a positive integer t, does there exist a clique S of the graph such that $|S| = t$?

Some NP-complete problems are about mathematical programming and logic.

Consider a set of items $S = \{s_i \mid 1 \leq i \leq n\}$, where $n \in \mathbb{N}$, and each item s_i is associated with a weight w_i and a value v_i for $1 \leq i \leq n$, and a positive integer $t \in \mathbb{N}$ called the target. The *knapsack problem* asks for the numbers n_i, $1 \leq i \leq n$, such that the total value $\sum_{1 \leq i \leq n} v_i \times n_i$ is maximized while the total weight is not over the target, that is $\sum_{1 \leq i \leq n} w_i \times n_i \leq t$. This problem has a related NP-complete decision problem: Given a set of items with weights and values and two positive integers t and k, does there exist a solution where the total value is k without exceeding the weight limit t? If the numbers have the property that $n_i \in \{0, 1\}$ ($n_i \leq c \in \mathbb{N}$ and $n_i \in \mathbb{N}$, respectively), for $1 \leq i \leq n$, the problem is called the 0-1 *(bounded and unbounded, respectively) knapsack problem.*

The subset sum problem (SSP) can be viewed as a special case of the knapsack problem. Given a finite set $S \subset \mathbb{N}$ of positive integers and a positive integer target number $t \in \mathbb{N}$, the subset sum problem asks if there exists a subset $S' \subseteq S$ such that $t = \Sigma_{s \in S'} s$.

A propositional variable is a variable that can be either true or false, which are complementary to each other. The binary connective "or" is denoted by $\vee$, and its result is true if and only if at least one of its operands is true. The binary connective "and" is denoted by $\wedge$, and its result is true if and only if both its operands are true. The unary connective "not" is denoted by $\neg$, and its result is true if and only if its operand is false. A propositional formula can be a propositional variable, $\neg B$, $D \wedge B$, or $D \vee B$, where B and D are propositional formulas. Propositional variables and their complements ("not" of the propositional variables) are called *literals*. A *disjunctive clause* is a single literal or a list of literals connected by $\vee$. A propositional formula is in *conjunctive normal form (CNF)* if it is a single disjunctive clause or a list of disjunctive clauses connected by $\wedge$. A truth valuation assigns true or false to all propositional variables, and a formula can be evaluated to true or false using a truth valuation according to the definitions of logic connectives. A propositional formula is said to be *satisfiable* if there exists a truth valuation that determines this formula to be true. Given a propositional formula in CNF, the satisfiability problem (SAT) asks if this formula is satisfiable. If all of its disjunctive clauses have at most three literals, the problem is called the 3-satisfiability problem (3-SAT).

The three steps of a DNA computing procedure, described in Section 2.2 (encoding the input information as DNA strands, performing the DNA computation, and reading out the output of the DNA computation), have been used to solve instances of such NP-complete problems and their related optimization problems as the directed HPP [3], the MIS problem [97], the MDS problem [103], the knapsack problem and its variant, SSP, [7, 102, 257], the 3-SAT problem [25, 26, 130, 169, 203], and the maximal clique problem [207]. These are summarized in Table 2.1.

Since the area of DNA computing started when Adleman used DNA to solve a seven-node instance of the directed Hamiltonian path problem [3], we use this procedure as an example of DNA computing. The input directed graph is as shown in Figure 2.16, the start node is 0, and the end node is 6. The problem has a unique solution, namely the directed Hamiltonian path $0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6$.

In the first step of this DNA computing approach, this graph was encoded as DNA single strands as follows. For each node i in the graph, where $0 \leq i \leq 6$, it was encoded by a DNA single strand O_i of length 20. For each directed edge from node i to node j in the graph, $1 \leq i \leq 6$ and $0 \leq j \leq 5$, it was encoded by a DNA single strand $O_{i \to j}$ of length 20 that was the concatenation of the second half of O_i and the first half of O_j. For each directed outgoing edge from the start node 0 to node j in the graph, $0 \leq j \leq 5$, it was encoded by a DNA single strand $O_{0 \to j}$ of length 30 that was the concatenation of O_0 and the first half of O_j. Similarly, for each directed incoming edge to the end node 6 from node i in the graph, $1 \leq i \leq 6$, it was encoded by a DNA single strand $O_{i \to 6}$ of length 30 that was the concatenation of the second half of O_i and O_6. Lastly, if there was a directed edge from the start node 0 to the end node 6, it would be encoded by a DNA single strand $O_{0 \to 6}$ of length 40 that was the concatenation of O_0 and O_6.

Problem	Cardinality	Bio-operations	Reference
Directed HPP	7 nodes	Annealing, PCR, extraction, gel electrophoresis	[3]
MIS	6 nodes	Cutting, pasting, cloning, gel electrophoresis, sequencing	[97]
MDS	6 nodes	Cutting, pasting, cloning, PCR, gel electrophoresis	[103]
Knapsack/SSP	7 items	Cutting, pasting, cloning, gel electrophoresis, sequencing	[102]
	3 items	Pasting, PCR, gel electrophoresis, annealing, sequencing	[7]
	3 items	Cutting, pasting, gel electrophoresis, annealing	[257]
SAT	6 variables 20 variables 10 variables	PCR, sequencing, extraction, gel electrophoresis, annealing, pasting	[26] [25] [130]
	4 variables	Extraction, cutting, PCR, annealing	[169]
	6 variables	Annealing (hairpins), cutting, pasting, PCR, sequencing, gel electrophoresis	[242]
	4 variables	Annealing (competitive), extraction	[260]
	4 variables	Cutting, pasting (ligase chain reaction), cloning, gel electrophoresis, extraction, sequencing	[273]
Maximal clique	6 nodes	Annealing, PCR (overlap extension), gel electrophoresis, cloning, sequencing, cutting	[207]

Table 2.1: List of some DNA computing algorithms (procedures) solving NP-complete problems and their related optimization problems and the bio-operations used. This list includes the directed Hamiltonian path problem (HPP), the maximal independent set problem (MIS), the minimal dominating set problem (MDS), the knapsack problem, the subset sum problem (SSP), variations of the satisfiability problem (SAT), and the maximal clique problem. Depending on the problem, the cardinality of an instance of a problem is the number of nodes in the input graph, the cardinality of the input set, or the number of propositional variables.

With this encoding, if DNA single strands $\overline{O_i}$, where $1 \leq i \leq 5$, are mixed with DNA single strands for edges, then partially double-stranded DNA molecules representing paths through the graph can be generated after hybridization and ligation. For each partially double-stranded DNA molecule representing a path, one of its strands represents a list of edges that this path passes through. Because of the encoding of the nodes and edges, this strands also represents a (partial) list of nodes that this path visits. The similar information is also stored in the other strand, where the orders of nodes and edges are reversed, and the Watson-Crick complement of the encodings are used. For example, as shown in Figure 2.17, a partially double-stranded DNA molecule that represents the path $2 \to 3 \to 4$ can be generated from the DNA single strand ($\overline{O_3}$) complementary to the encoding of node 3 and the DNA single strands encoding the two directed edges from node

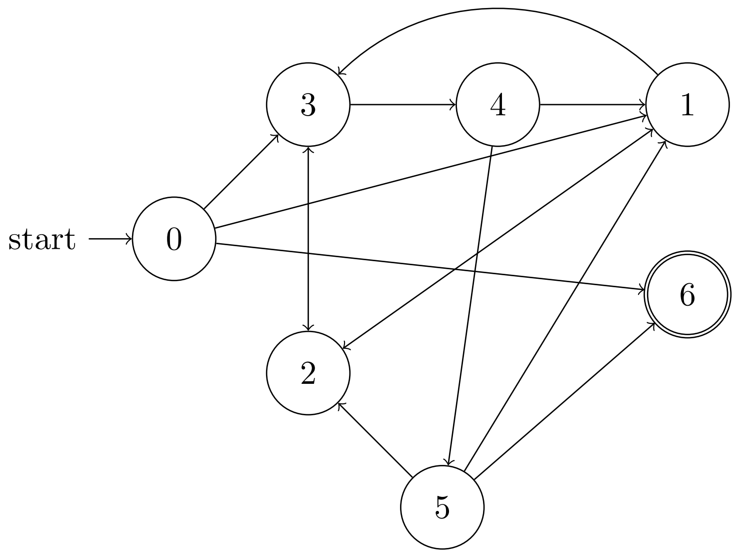


Figure 2.16: Input graph of the seven-node instance of the directed Hamiltonian path problem solved by Adleman [3]. The start node is 0 (with an incoming node from start), and the end node is 6 (double circle). The problem has a unique solution, namely the directed Hamiltonian path $0 \to 1 \to 2 \to 3 \to 4 \to 5 \to 6$.

2 to node 3 ($O_{2\to3}$) and from node 3 to node 4 ($O_{3\to4}$). Its top strand represents a path passing through the edge from node 2 to node 3, followed by the edge from node 3 to node 4.

Next is the actual DNA computation, and the algorithm is as follows:

1. Generate random paths through the graph. This is achieved by annealing DNA single strands $\overline{O_i}$, $1 \leq i \leq 5$, with DNA single strands encoding each of the directed edges. The backbones of these DNA double strands are connected by ligation. In [3], approximately 3×10^{13} copies of each input DNA single strand are used. Since infinitesimal free-floating DNA strands are in great quantity, they can interact with each other randomly through hybridization. Therefore, all possible paths through the graph are generated with high probability (theoretically, even one copy of each path is sufficient for this computation).

2. Select all paths that start with node 0 and end with node 6 from the pool of above randomly generated paths. This is achieved using PCR to amplify the DNA double strands generated from the last step with primers O_0 and $\overline{O_6}$. With these two primers,

only those DNA (sub)strands that have O_0 at one end and O_6 at the other end are amplified by PCR.

3. Select all paths that have the desired length. Since every Hamiltonian path would have to traverse seven nodes, a solution to the problem, if one exists, would be a DNA strand of length 140 bp. Gel electrophoresis is used to select those strands of 140 bp in length.

4. Select all paths that traverse all remaining nodes 1, 2, 3, 4, and 5 from the paths that remain after the selection in the previous step. This is achieved by sequentially extracting DNA strands that contain O_1, O_2, O_3, O_4, and O_5 as a substrand. For node 1, the DNA strands with O_1 as a substrand are selected by extraction, with the DNA single strands $\overline{O_1}$ as the sequence attached to the surface. The remaining paths go through a similar selection for the remaining nodes in a sequential order.

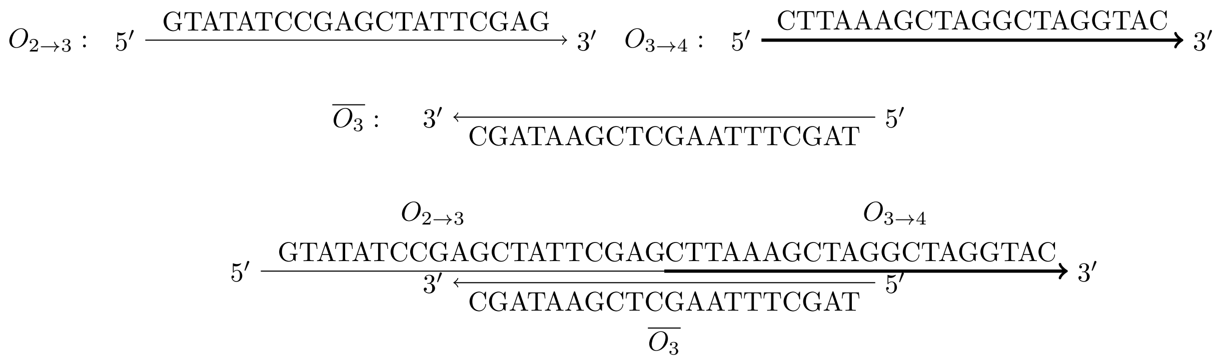


Figure 2.17: Example of a path formed by connecting two compatible edges using the node that they have in common [3]. Top: The directed edges from node 2 to node 3 and from node 3 to node 4 are encoded by the DNA single strands $O_{2\to3}$ and $O_{3\to4}$. Middle: Node 3 is encoded by O_3, and the DNA single strand $\overline{O_3}$ is used in computation. Bottom: The above strands can recombine to form a partially double-stranded DNA molecule after hybridization and ligation. Its top strand represents a path passing through the edge from node 2 to node 3, followed by the edge from node 3 to node 4. It also contains all the nodes that two neighbour edges in the path have in common. In this case, it is node 3.

After these selections, the remaining paths (if any) are Hamiltonian paths from node 0 to node 6 because each of them starts with node 0 and ends with node 6 (by step 2), each traverses all seven nodes (by step 3), and each traverses each of the nodes at least once (by step 4).

The last step is the result readout, which can be done by checking if there are DNA double strands encoding paths left in the solution. The resulting DNA double strands can be amplified by PCR, and we can use gel electrophoresis to check the existence of DNA molecules. The answer to the Hamiltonian path problem is "yes" if there are DNA molecules of length 140 bp left and "no" otherwise. If there exists a Hamiltonian path, and one wants to know what is in the path, sequencing can be used to read the DNA sequence; it can then be decoded using the encoding scheme described above.

Chapter 3

Word Blending and Other Formal Models of Bio-operations

Since a seven-city instance of the directed Hamiltonian path problem was solved using programmed DNA interactions in a test tube [3], there has been an ongoing effort to model biological processes on DNA as computations. We continue this research direction by modelling the unexpected outcomes of a DNA wet lab protocol, when particular types of input DNA strands are its input, as two new bio-operations, and by studying their formal language properties.

In Section 3.1, an introduction to formal language operations is given. In Section 3.2, some biologically inspired word operations in the literature are described. In Section 3.3, the details of cross-pairing polymerase chain reaction (XPCR) is given. In addition, a previously studied bio-operation inspired by XPCR is discussed. In Section 3.4, a word operation that models a generalization of a special case of XPCR is defined and studied. In Section 3.5, a new word operation that is more closely related with the XPCR experimental requirement is proposed and studied, and Section 3.6 describes the wet lab experiments that motivate and implement this operation. In Section 3.7, a conclusion and possible future work are discussed.

Section 3.4 contains updated results from a paper I co-authored [76] titled "Word blending in formal languages." This paper is the journal version of a conference paper I co-authored during my Master's studies [75] titled "Word blending in formal languages: the Brangelina effect."

Section 3.5 and Section 3.6 contain theoretical and experimental results from a paper I co-authored [15] titled "Conjugate word blending: Formal model and experimental implementation by XPCR."

The co-authors of [15] include an experimental team who implemented the XPCR experiments and generated the gel electrophoresis results in Section 3.6.

3.1 Introduction to formal language operations

This section gives a summary of the aspects of formal language study that are related to this thesis, and it includes the closure properties of some families of languages under some operations, their left- and right-inverses, and their descriptional state complexity.

3.1.1 Closure properties

Among different families of languages, the Chomsky hierarchy of languages is well known and well studied. The four families of languages in the Chomsky hierarchy are regular, context-free, context-sensitive, and recursively enumerable languages. Languages are categorized into these different families according to their generative grammar.

The following definitions are from [243].

Definition 3.1. *A* grammar *is a quadruple $G = (V_N, V_T, a_0, F)$, where V_N is the non-terminal alphabet, V_T is the terminal alphabet, $a_0 \in V_N$ is the initial symbol, and $F \subseteq \{\alpha \to \beta \mid \alpha, \beta \in \{V_N \cup V_T\}^*, \mathrm{Alph}(\alpha) \cap V_N \neq \emptyset\}$ is the set of production rules.*

Definition 3.2. *A grammar $G = (V_N, V_T, a_0, F)$ is called:*

- *a* type-3 *grammar if*

$$F \subseteq \{a \to b\alpha \mid a, b \in V_N, \alpha \in V_T^*\} \cup \{a \to \alpha \mid a \in V_N, \alpha \in V_T^*\};$$

- *a* type-2 *grammar if $F \subseteq \{a \to \alpha \mid a \in V_N, \alpha \in \{V_N \cup V_T\}^*\}$;*
- *a* type-1 *grammar if*

$$F \subseteq \{\alpha_1 a \alpha_2 \to \alpha_1 \beta \alpha_2 \mid a \in V_N, \alpha_1, \alpha_2, \beta \in (V_N \cup V_T)^*, \beta \neq \lambda\};\ \textit{and}$$

- *a* type-0 *grammar if there are no restrictions on F.*

Note that these types of grammar are hierarchical, where a type-i grammar is also a type-j grammar, $0 \leq j \leq i \leq 3$. The Chomsky hierarchy of languages can thus be defined as follows:

Definition 3.3. *[48] A language is said to be:*

- regular *if it can be generated by a type-*3 *grammar;*
- context free *if it can be generated by a type-*2 *grammar;*
- context sensitive *if it can be generated by a type-*1 *grammar; and*
- recursively enumerable *if it can be generated by a type-*0 *grammar.*

The family of regular (context-free, context-sensitive, and recursively enumerable, respectively) languages is denoted by REG (CF, CS, and RE, respectively). A language L is said to be finite if it contains a finite number of words, and the family of finite languages is denoted by FIN.

In addition to generative grammars, there are simple idealized machines that recognize or accept languages [243].

Definition 3.4. *A* non-deterministic finite automaton (NFA) *is a quintuple $M = (S, V_T, s_0, A, F)$, where S is the set of states, V_T is the alphabet, $s_0 \in S$ is the initial state, $A \subseteq S$ is the set of final states, and $F \subseteq \{sa \to s' \mid s, s' \in S, a \in V_T\} \cup \{s \to s' \mid s, s' \in S\}$ is the set of transitions.*

The set of transitions F of an NFA can also be written as a function $\delta : S \times (\Sigma \cup \{\lambda\}) \to 2^S$, defined as follows. Given states $s, s' \in S$ and a letter $a \in \Sigma$, there is a transition of the form $sa \to s' \in F$ if and only if we have $s' \in \delta(s, a)$; there is a transition of the form $s \to s' \in F$ if and only if we have $s' \in \delta(s, \lambda)$.

Definition 3.5. *A* deterministic finite automaton (DFA) *is a quintuple $M = (S, V_T, s_0, A, F)$, where S is the set of states, V_T is the alphabet, $s_0 \in S$ is the initial state, $A \subseteq S$ is the set of final states, and $F \subseteq \{sa \to s' \mid s, s' \in S, a \in V_T\}$ is the set of transitions. Moreover, F contains at most one transition $sa \to s'$ for each pair (s, a), where $s, s' \in S$ and $a \in V_T$.*

Definition 3.6. *A* pushdown automaton (PDA) *is a septuple $M = (S, V_I, V_Z, z_0, s_0, A, F)$, where S is the set of states, V_I is the input alphabet, V_Z is the stack alphabet, $z_0 \in V_Z$ is the initial stack letter, $s_0 \in S$ is the initial state, $A \subseteq S$ is the set of final states, and $F \subseteq \{zs_ia \to \alpha s_j \mid z \in V_Z, a \in V_I, \alpha \in V_Z^*, s_i, s_j \in S\} \cup \{zs_i \to \alpha s_j \mid z \in V_Z, \alpha \in V_Z^*, s_i, s_j \in S\}$ is the set of transitions.*

Definition 3.7. *A* linear bounded automaton (LBA) *is a sextuple* $M = (S, V_I, V_T, s_0, A, F)$*, where* S *is the set of states,* $V_I \subseteq V_T$ *is the input alphabet,* V_T *is the tape alphabet,* $s_0 \in S$ *is the initial state,* $A \subseteq S$ *is the set of final states, and* $F \subseteq \{s_i a \to s_j b \mid s_i, s_j \in S, a, b \in V_T\} \cup \{s_i a \to a s_j \mid s_i, s_j \in S, a \in V_T\} \cup \{c s_i a \to s_j c a \mid s_i, s_j \in S, a, c \in V_T\}$ *is the set of transitions. Moreover, for each triple* (s_i, a, s_j)*, where* $s_i, s_j \in S$ *and* $a \in V_T$*, the set of transitions* F *either contains no rules of the form* $c s_i a \to s_j c a$ *or contains all rules of the form* $c s_i a \to s_j c a$ *for all* $c \in V_T$*.*

Definition 3.8. *A Turing machine (TM) is a septuple* $M = (S, V_I, V_1, V_T, s_0, A, F)$*, where* S *is the set of states,* V_T *is the tape alphabet,* $\# \in V_T$ *is the tape boundary marker,* $V_1 = V_T \setminus \{\#\}$ *is a non-empty alphabet,* $V_I \subset V_1$ *is the input alphabet,* $s_0 \in S$ *is the initial state,* $A \subseteq S$ *is the set of final states,* $\Box \in V_1$ *denotes an empty cell on the tape, and* $F \subseteq \{s_i a \to s_j b \mid s_i, s_j \in S, a, b \in V_1\} \cup \{s_i a c \to a s_j c \mid s_i, s_j \in S, a, c \in V_1\} \cup \{s_i a \# \to a s_j \Box \# \mid s_i, s_j \in S, a \in V_1\} \cup \{c s_i a \to s_j c a \mid s_i, s_j \in S, a, c \in V_1\} \cup \{\# s_i a \to \# s_j \Box a \mid s_i, s_j \in S, a \in V_1\}$ *is the set of transitions.*

In addition, for each triple (s_i, a, s_j)*, where* $s_i, s_j \in S, a \in V_1$*, the set of transitions* F *either contains no rules of the form* $s_i a c \to a s_j c$ *and* $s_i a \# \to a s_j \Box \#$ *(*$c s_i a \to s_j c a$ *and* $\# s_i a \to \# s_j \Box a$*) or contains all rules of the form* $s_i a c \to a s_j c$ *and* $s_i a \# \to a s_j \Box \#$ *(*$c s_i a \to s_j c a$ *and* $\# s_i a \to \# s_j \Box a$*) for all* $c \in V_1$*.*

Moreover, for each pair (s_i, a)*, where* $s_i \in S, a \in V_1$*, there is at most one transition of the form* $s_i a \# \to a s_j \Box \#$*,* $\# s_i a \to \# s_j \Box a$ *and* $s_i a \to s_j b$ *in* F*, where* $s_j \in S$ *and* $b \in V_1$*.*

The relationships among the Chomsky hierarchy of languages, generative grammars, and recognition machines are summarized in Table 3.1 with examples.

Family	Grammar	Machine	Example
REG	Type-3	NFA/DFA	$\{a^n \mid n \geq 0\}$
CF	Type-2	PDA	$\{a^n b^n \mid n \geq 0\}$
CS	Type-1	LBA	$\{a^n b^n c^n \mid n \geq 0\}$
RE	Type-0	TM	$\{w \mid w$ encodes a TM that halts with empty input.$\}$

Table 3.1: The relationships among the Chomsky hierarchy of languages, generative grammars, and recognition machines with examples.

A family of languages $\mathcal{L}$ is said to be closed under an operation $\diamond$ if application of the operation $\diamond$ on languages in the family $\mathcal{L}$ always produces a language in the same family $\mathcal{L}$. Definitions of some operations are given in Table 3.2, and the closure properties of the Chomsky hierarchy of languages under these operations are summarized in Table 3.3.

Operation	Definition
Union	$L_1 \cup L_2 = \{\alpha \mid \alpha \in L_1 \text{ or } \alpha \in L_2\}$
Concatenation	$L_1 L_2 = \{\alpha\beta \mid \alpha \in L_1, \beta \in L_2\}$
Kleene star	$L^* = \bigcup_{i=0}^{\infty} L^i$
Kleene plus	$L^+ = \bigcup_{i=1}^{\infty} L^i$
Reverse	$L^r = \{a_1 a_2 \cdots a_k \mid a_k a_{k-1} \cdots a_1 \in L\}$
Substitution	$\sigma(L) = \{\alpha \mid \exists \beta \in L : \alpha \in \sigma(\beta)\}$ $\sigma(\lambda) = \lambda, \sigma(\alpha\beta) = \sigma(\alpha)\sigma(\beta)$ $\forall a \in \Sigma : \sigma(a) \subseteq \Sigma_a^*$
Homomorphism	$\sigma(L) = \{\alpha \mid \exists \beta \in L : \alpha \in \sigma(\beta)\}$ $\sigma(\lambda) = \lambda, \sigma(\alpha\beta) = \sigma(\alpha)\sigma(\beta)$ $\forall a \in \Sigma : \sigma(a) \in \Sigma_a^*$
Intersection	$L_1 \cap L_2 = \{\alpha \mid \alpha \in L_1, \alpha \in L_2\}$
Set difference	$L_1 \setminus L_2 = \{\alpha \mid \alpha \in L_1, \alpha \notin L_2\}$
Complementation	$L^c = \{\alpha \in \Sigma^* \mid \alpha \notin L\} = \Sigma^* \setminus L$
Left quotient	$L_2^{-1_l} L_1 = \{\beta \mid \alpha\beta \in L_1, \alpha \in L_2\}$
Left derivative	$\partial_\alpha L = \{\beta \mid \alpha\beta \in L\}$
Right quotient	$L_1 L_2^{-1_r} = \{\alpha \mid \alpha\beta \in L_1, \beta \in L_2\}$
Right derivative	$\partial_\beta^r L = \{\alpha \mid \alpha\beta \in L\}$
Prefix Suffix Infix	$\mathrm{pref}(L) = \{\alpha \mid \alpha\beta \in L, \alpha, \beta \in \Sigma^*\}$ $\mathrm{suff}(L) = \{\beta \mid \alpha\beta \in L, \alpha, \beta \in \Sigma^*\}$ $\mathrm{inf}(L) = \{\beta \mid \alpha_1\beta\alpha_2 \in L, \alpha_1, \alpha_2, \beta \in \Sigma^*\}$
Sequential insertion	$L_1 \leftarrow L_2 = \bigcup_{\alpha \in L_1, \beta \in L_2} (\alpha \leftarrow \beta)$ $\alpha \leftarrow \beta = \{\alpha_1\beta\alpha_2 \mid \alpha = \alpha_1\alpha_2\}$
Parallel insertion	$L_1 \Leftarrow L_2 = \bigcup_{\alpha \in L_1, \beta \in L_2} (\alpha \Leftarrow \beta)$ $\alpha \Leftarrow L = \{\beta_0 a_1 \beta_1 a_2 \cdots a_k \beta_k \mid \alpha = a_1 a_2 \cdots a_k, \beta_0, \beta_1, \ldots, \beta_k \in L\}$
Sequential deletion	$L_1 \rightarrow L_2 = \bigcup_{\alpha \in L_1, \beta \in L_2} (\alpha \rightarrow \beta)$ $\alpha \rightarrow \beta = \{\alpha_1\alpha_2 \mid \alpha = \alpha_1\beta\alpha_2, \alpha_1, \alpha_2 \in \Sigma^*\}$
Dipolar deletion	$L_1 \leftrightarrow L_2 = \bigcup_{\alpha \in L_1, \beta \in L_2} (\alpha \leftrightarrow \beta)$ $\alpha \leftrightarrow \beta = \{\alpha_2 \mid \alpha = \alpha_1\alpha_2\alpha_3, \beta = \alpha_1\alpha_3, \alpha_1, \alpha_2, \alpha_3 \in \Sigma^*\}$
Shuffle	$L_1 \amalg L_2 = \bigcup_{\alpha \in L_1, \beta \in L_2} (\alpha \amalg \beta)$ $\alpha \amalg \beta = \{\alpha_1\beta_1\alpha_2\beta_2 \cdots \alpha_k\beta_k \mid \alpha = \alpha_1\alpha_2 \cdots \alpha_k, \beta = \beta_1\beta_2 \cdots \beta_k,$ $k \in \mathbb{N}^+, \alpha_i, \beta_i \in \Sigma^*, 1 \leq i \leq k\}$
Cyclic shift	$\mathrm{SHIFT}(L) = \{\alpha\beta \mid \beta\alpha \in L\}$

Table 3.2: Definitions of some formal language operations. Languages are denoted by L, alphabets are denoted by Σ, words are denoted by lowercase letters from the Greek alphabet, and letters are denoted by lowercase letters from the English alphabet.

Operation	REG	CF	CS	RE	References
Union	Y	Y	Y	Y	[243]
Concatenation	Y	Y	Y	Y	[243]
Kleene star	Y	Y	Y	Y	[243]
Kleene plus	Y	Y	Y	Y	[243]
Reverse	Y	Y	Y	Y	[243]
Substitution	Y	Y	N	Y	[243]
Homomorphism	Y	Y	N	Y	[243]
Intersection	Y	N ($L_1 \in$ REG or $L_2 \in$ REG)	Y	Y	[166, 234, 243, 246]
Set difference	Y	N	Y	N	[243, 246]
Complementation	Y	N	Y	N	[117, 164, 243, 254]
Left/right quotient	Y	N ($L_2 \in$ REG)	N ($L_2 \in$ FIN)	N	[88, 139, 243]
Left/right derivative	Y	Y	Y	Y	[243]
Prefix Suffix Infix	Y	Y	N	Y	[243]
Sequential insertion	Y	Y	Y	Y	[139]
Parallel insertion	Y	Y	Y	Y	[139]
Sequential deletion	Y	N	N	Y	[140]
Dipolar deletion	Y	N	N	Y	[140]
Shuffle	Y	N	Y	Y	[118, 139]
Cyclic shift	Y	Y	Y	Y	[182, 206]

Table 3.3: Closure properties of the Chomsky family of languages under formal language operations. Definitions are given in Table 3.2. The families of the languages considered are finite, regular, context free, context sensitive, and recursively enumerable denoted by FIN, REG, CF, CS, and RE, respectively. A letter "Y" in a cell indicates that the family of the column is closed under the operation of the row, and a letter "N" in a cell indicates that the family of the column is not closed under the operation of the row. In the case of N, the condition surrounded by brackets is the exceptional case, and L_1 and L_2 are the left and right operands of the binary operation.

3.1.2 Invertible operations

Consider an algebraic equation of the form $ax = b$, where a and b are known numbers, and x is an unknown number. Its solution can be calculated by $x = b/a$, where quotient can recover the right operand of multiplication from its left operand and product. A similar idea can be used to solve equations involving languages and formal language operations. For example, given two known languages L and R and an unknown language Y over an alphabet Σ, decision problems related to whether or not there exists a solution to $LY = R$ were studied in [141].

In this section, language equations involving binary word operations are considered. For example, consider an equation of the form $L \diamond Y = R$ or $X \diamond L = R$, where $\diamond$ is a binary word operation, L and R are known languages, and X and Y are unknown languages. The question "whether or not there exists a solution to the equation" was studied in [141, 143, 154].

Using the idea that quotient can recover an operand of multiplication from its other operand and product, the left- and right-inverse of a binary word operation can be defined as follows.

Definition 3.9. *A binary word operation $\diamond$ is called the* left-inverse *of $\Box$ if for all $\alpha, \beta, \gamma \in \Sigma^*$, we have that $\gamma \in (\alpha \Box \beta)$ if and only if $\alpha \in (\gamma \diamond \beta)$. Similarly, a binary word operation $\diamond$ is called the* right-inverse *of $\Box$ if for all $\alpha, \beta, \gamma \in \Sigma^*$, we have that $\gamma \in (\alpha \Box \beta)$ if and only if $\beta \in (\alpha \diamond \gamma)$.*

For example, concatenation and right quotient are the left-inverses of each other, and sequential insertion and reversed dipolar deletion are the right-inverses of each other.

Using the definition of left- and right-inverses of binary word operations, the following propositions from [141] provide a way to find the solution to a language equation of the form $L \diamond Y = R$ or $X \diamond L = R$.

Proposition 3.10. *Consider an equation $L \diamond Y = R$. If there is a solution to this equation, the language $Y' = (L \Box R^c)^c$ is also a solution to the equation, where $\Box$ is the right-inverse of $\diamond$, and all the solutions to this equation are subsets of Y'.*

Proposition 3.11. *Consider an equation $X \diamond L = R$. If there is a solution to the equation, the language $X' = (R^c \Box L)^c$ is a solution to the equation, where $\Box$ is the left-inverse of $\diamond$, and all the solutions to the equation are subsets of X'.*

Note that X' and Y' are the maximal solutions.

Using the above two propositions, the decidability of the existence of a (singleton) solution to an equation can be determined.

For example, consider an alphabet Σ, two regular languages L, R over Σ, and binary operations $\diamond$ and $\Box$, where the family of regular languages is closed under them, and they are the right-inverses to each other. We want to determine whether the question "does a solution to the equation $L \diamond Y = R$ exist?" is decidable or not. A maximal solution $Y' = (L \Box R^c)^c$ can be effectively constructed because the family of regular languages is closed under $\Box$ and complementation. A solution exists if and only if the regular language $L \diamond Y'$ equals the regular language R. Since testing the equivalence of two regular languages is decidable [6], the existence of a solution to the equation is also decidable.

3.1.3 State complexity

In this section, we briefly introduce state complexity and offer results about the state complexity of operations under which the family of regular language is closed.

The size of a DFA $M = (S, V_T, s_0, A, F)$ can be determined by either the number of states $|S|$ or the number of transitions $|F|$. These are related measurements because the number of transitions of M is at most $|V_T| \cdot |S|$. The state complexity of a regular language L, denoted by $\mathrm{sc}(L)$, is defined as the number of states in the minimal complete DFA M that recognizes L. A DFA is said to be *complete* if there is a transition $sa \to s'$ in F for every state $s \in S$ and letter $a \in V_T$. A complete DFA is said to be *minimal* if there does not exist any complete DFA with fewer states recognizing the same language. For every regular language, there is a unique minimal complete DFA up to the renaming of the states, and it can be generated using the Myhill-Nerode theorem [199, 252].

For example, the minimal complete DFA recognizing the regular language $L = a\{a, b\}^*b$ is shown in Figure 3.1, which gives up $\mathrm{sc}(L) = 4$.

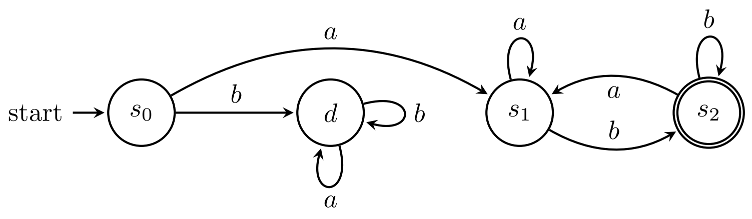


Figure 3.1: The minimal complete DFA recognizing the regular language $a\{a, b\}^*b$.

A regular language L is called an m-state DFA language if L can be recognized by a DFA with m states.

The definition of state complexity can be extended to a regularity-preserving operation (an operation that outputs a regular language whenever its operand[s] are regular) as follows.

Definition 3.12. *Let $\diamond$ be a regularity-preserving operation taking k operands, and let L_i be an m_i-state DFA language for $1 \leq i \leq k$. The state complexity of $\diamond$ is the number of states, in terms of m_i, $1 \leq i \leq k$, that is sufficient and necessary for a minimal complete DFA to recognize the output of $\diamond$ on operands L_i, $1 \leq i \leq k$.*

To show that the state complexity of $\diamond$ is a tight upper bound, a family of languages for each operand should be provided such that the number of states of the minimal complete DFA recognizing the output reaches that bound. These families of languages are called *witnesses*.

For example, consider an alphabet Σ, an m-state regular language L_m over Σ, and an n-state regular language L_n over Σ, where $m = 1$, $n \geq 2$, and $|\Sigma| \geq 2$, the state complexity of concatenation $\mathrm{sc}(L_m L_n)$ is $2^n - 2^{n-1}$, and the witnesses are $L_m = \{a, b\}^*$ and $L_n = \{ubv \in \{a, b\}^* \mid |v|_a \equiv n - 2 \pmod{n-1}\}$ [279].

The state complexities of concatenation, union, intersection, complementation, reverse, and Kleene star were studied in [125–127, 189, 229, 278, 279]. These are summarized in Table 3.4.

In addition to the above mentioned operations, the state complexities of other operations, such as left/right quotient [279] and left/right derivative [71], were studied. Other variations, such as the state complexity regarding NFA [71, 125–128, 187] and state complexity considering such sub-regular languages as finite languages [33, 34, 93, 109, 110, 278], were studied.

A family of languages called *universal witnesses* was identified in [30]. These can be used as witnesses for many of the above mentioned operations and compositions of operations. The conditions about the universal witnesses can be used as heuristics when searching for witnesses for a given upper bound.

It has also been proved that no algorithm exists to calculate the state complexity of an operation that is a composition of regularity-preserving operations with known state complexities [244].

Operation	State complexity	Condition	References
L_mL_n	$2^n - 2^{n-1}$	$m = 1, n \geq 2, \|\Sigma\| >= 2$	[279]
	$m2^n - 2^{n-1}$	$m \geq 2, n \geq 2, \|\Sigma\| >= 2$	[126]
	m	$m \geq 1, n = 1, \|\Sigma\| >= 1$	[279]
	mn	$m = 1, n \geq 2, \|\Sigma\| = 1, \gcd(m, n) = 1$	[279]
$L_m \cup L_n$	mn	$m \geq 1, n \geq 1, \|\Sigma\| \geq 2$	[279]
		$m \geq 1, n \geq 1, \|\Sigma\| = 1, \gcd(m, n) = 1$	[278]
$L_m \cap L_n$	mn	$m \geq 1, n \geq 1, \|\Sigma\| \geq 2$	[229, 279]
		$m \geq 1, n \geq 1, \|\Sigma\| = 1, \gcd(m, n) = 1$	[278]
L_m^c	m	$m \geq 1, n \geq 1, \|\Sigma\| \geq 1$	[125]
L_m^r	2^m	$m \geq 1, n \geq 1, \|\Sigma\| \geq 2$	[127]
	m	$m \geq 1, n \geq 1, \|\Sigma\| = 1$	[189]
L_m^*	$2^{m-1} + 2^{m-k-1}$	$m \geq 2, k = \|A \setminus \{s_0\}\| \geq 1, \|\Sigma\| \geq 2$	[279]
	m	$m \geq 2, \|A \setminus \{s_0\}\| = 0, \|\Sigma\| \geq 1$	
	2	$m = 1, \|\Sigma\| \geq 1$	
	$(m-1)^2 + 1$	$m \geq 2, \|A \setminus \{s_0\}\| \geq 1, \|\Sigma\| = 1$	

Table 3.4: Summary of the state complexities of concatenation, union, intersection, complementation, reverse, and Kleene star. The operands are two languages L_m and L_n over an alphabet Σ, and they can be recognized by minimal complete DFA with m and n states, respectively. The state complexities depend on conditions like the number of states, the size of the alphabet, the number of states in the set of final states, and whether m and n are co-prime. The witnesses given in the paper appear in the reference column.

3.2 Biologically inspired word operations

This section lists biologically inspired word operations. They formalize DNA processes that are:

- naturally occurring—duplication, inversion, transposition, repeat deletion, and block substitution;
- related to the actions of enzymes on DNA strands—splicing, hairpin completion, and template-directed extension; and
- related to lab protocols with multiple steps—contextual insertion/deletion and site-directed deletion/insertion.

First, there are many naturally occurring DNA processes that modify DNA strands, such as mutations, insertions, and deletions. Genes are segments of DNA that can determine the traits of an organism, a chromosome is a linear collection of genes, and a genome is the complete set of DNA of an organism. There are many known processes of genome rearrangements, and the following bio-operations are related to them.

The process where a segment of a chromosome is reversed end-to-end is called *chromosomal inversion*. For example, consider words u, v, and w over an alphabet Σ, the word uvw becomes $uv^r w$ after a chromosomal inversion. This process is formalized as *inversion* in [59]. The process, where the whole segment does not reverse completely during inversion, is formalized as *pseudo inversion* [43]. For a chromosome, multiple inversions may happen at the same time. This process is modelled as *non-overlapping inversion* [156]. A related operation called *block reversal* was studied in [174]. *Hairpin inverted repeat* models the situation where an inversion occurs at the head of a hairpin structure [57].

The process where a segment of a chromosome is moved to a new location in the chromosome is called *transposition*. For example, the word $u\alpha\beta v$ becomes $v\beta\alpha u$. This process is formalized as *transposition* in [59].

The process called *duplication* copies a segment of a chromosome to a new location. For example, the word uvw becomes $uvvw$. This process is formalized as *duplication* in [59]. A variation of duplication that identifies a set of subwords that can be duplicated was studied in [119]. In addition, duplication with length limitation was considered in [121]. Copying errors can occur during duplication, such as the duplicated part having an altered sequence (modelled by *pseudo duplication* [44]) or the duplicated part being reversed (modelled by *reverse duplication* [44, 58]). The bio-operation *repeat deletion* models the process where the duplicated substrand of DNA is skipped, and it was studied in [119].

The phenomenon of errors occurring in DNA strands, where part of a DNA strand is replaced with another strand of the same length, is modelled by the bio-operation called *block substitution*, defined and studied in [146].

Next, we consider the DNA processes that are related to the actions of enzymes, such as restriction enzymes, ligases, and DNA polymerases.

The DNA recombination achieved by restriction enzyme digestion and ligation is modelled as a bio-operation called *splicing*, as studied in [95, 221, 223].

Given an alphabet Σ, a splicing rule r is defined as $r = u_1\#u_2\$u_3\#u_4$, where $\#, \$ \notin \Sigma$ and $u_1, u_2, u_3, u_4 \in \Sigma^*$ [223]. In a splicing rule $u_1\#u_2\$u_3\#u_4$, the words u_1 and u_4 are called *visible sites*, while u_2 and u_3 are called *invisible sites*. For two strings $x = x_1u_1u_2x_2$ and $y = y_1u_3u_4y_2$, applying the rule $r = u_1\#u_2\$u_3\#u_4$ produces a string $z = x_1u_1u_4y_2$, and this is denoted by $(x, y) \vdash^r z$. Splicing rules of the form $w\#\lambda\$w\#\lambda$, where $w \in \Sigma^+$, are called *null-context splicing rules* [95].

A splicing scheme is a pair $\sigma = (\Sigma, \mathcal{R})$, where Σ is an alphabet, and $\mathcal{R}$ is a set of splicing rules. A splicing scheme $\sigma = (\Sigma, \mathcal{R})$ is said to be regular (context free, context sensitive, respectively) if $\mathcal{R}$ is a regular (context-free, context-sensitive, respectively) language contained in $\Sigma^*\#\Sigma^*\$\Sigma^*\#\Sigma^*$. The closure properties of various families of languages under a splicing scheme were studied in [96].

A splicing system is a splicing scheme together with an initial language, and the language generated by a splicing system is the set of all words obtained by iteratively applying splicing rules on the initial language. A splicing system is called a *simple splicing system* [184] if the splicing rules are restricted to rules of the form $a\#\lambda\$a\#\lambda$ for $a \in \Sigma$. In [222], it is shown that the families of regular, context-free, and recursively enumerable languages are closed under splicing systems with a regular splicing scheme and finitely many visible sites. Splicing systems with a regular splicing scheme and a finite initial language have the same computational power as the Turing machine [210, 224].

As mentioned in Section 2.1, a DNA single strand can form an intramolecular structure called a hairpin if it contains compatible substrands. The resulting hairpin structure may have a partially double-stranded stem, and a polymerase enzyme can be used to extend the stem to become fully double stranded. This process is formalized as a unary word operation called *hairpin completion*, defined in [41], where the closure properties under it and its iterated version have been studied. Its variation with length restrictions on the sticky end was defined and studied in [120, 162]. Other related bio-operations were studied in [120, 176–179].

The process where a DNA polymerase enzyme extends a primer according to a template is formalized as template-directed extension, introduced in [74]. The families of regular and recursively enumerable languages are closed under it, but the families of context-free and context-sensitive languages are not.

Lastly, there are some lab protocols that are used to modify DNA strands.

Insertions and deletions of nucleotide sequences can be achieved by a biology lab technique called *PCR site-specific oligonucleotide mutagenesis* [64], and these processes can be modelled as language operations called *contextual insertion* and *contextual deletion* defined and studied in [153]. Closure properties under contextual insertion and contextual deletion and the decidability of the existence of solutions to equations involving them were studied in [153]. They can be used to defined a computation system called an *insertion-deletion system* that has the same computational power as the Turing machine [151, 153, 211, 259]. Related operations called *site-directed deletion/insertion* were defined and studied in [45–47, 173].

3.3 Cross-pairing polymerase chain reaction (XPCR) and overlap assembly

Cross-pairing polymerase chain reaction (XPCR) is a wet lab procedure introduced in [84] for extracting all the strands that contain a given pattern (a substring) from a heterogeneous pool of DNA strands. It was employed to implement several DNA recombination algorithms [86], to create the solution space for a SAT problem [82], and for mutagenesis [85]. The combinatorial power of this technique was explained using logical-symbolic schemes in [175], while algorithms to create combinatorial libraries were experimented in [85] and improved in [83], where all permutations of the three genes were generated. The process of XPCR is shown in Figure 3.2, where the inputs of XPCR are $\alpha X\gamma$ and $\gamma Y\beta$, and the output is $\alpha X\gamma Y\beta$.

The wet lab experiments reported in this chapter involve three different genes: *dbtAa* (Ferredoxyn Reductase, 1,019 bp), *dbtAb* (Ferredoxyn, 311 bp), and *dbtAd* (β Dioxygenase subunit, 518 bp), extracted from the widely studied bacterial strain *Burkholderia fungorum DBT1* [62]. In the following, we denote these three genes by the capital letters A, B, and D, respectively, and the primers (21 bp long DNA sequences used in XPCR) by α, β, and γ. The primer α (β and γ, respectively) is TTCTACAAGGAGGATATTACC (GATATCAGGTACATCTCCATA and ATATTGGAGGAGGTATACAAC, respectively). The experimental results of XPCR to concatenate pairs of genes (D with B, B with D, A with B, and D with A) using primers α, β, and γ are shown in Figure 3.3.

The process of XPCR illustrated in Figure 3.3 was formalized as a binary word operation called overlap assembly in [73].

Definition 3.13. *Given two words $x, y \in \Sigma^*$, the overlap assembly of x with y is defined by*

$$x \overline{\odot}\, y = \{uvw \mid x = uv, y = vw, u, w \in \Sigma^*, v \in \Sigma^+\}.$$

This operation was studied under the name *linear self-assembly* in [55] and *chop* in [108]. It can also be considered a special case of a *semantic shuffle on trajectories*, as studied in [65]. The iteration of this process was called *parallel overlap assembly*, and it was used to generate combinations of partially double-stranded DNA molecules [137].

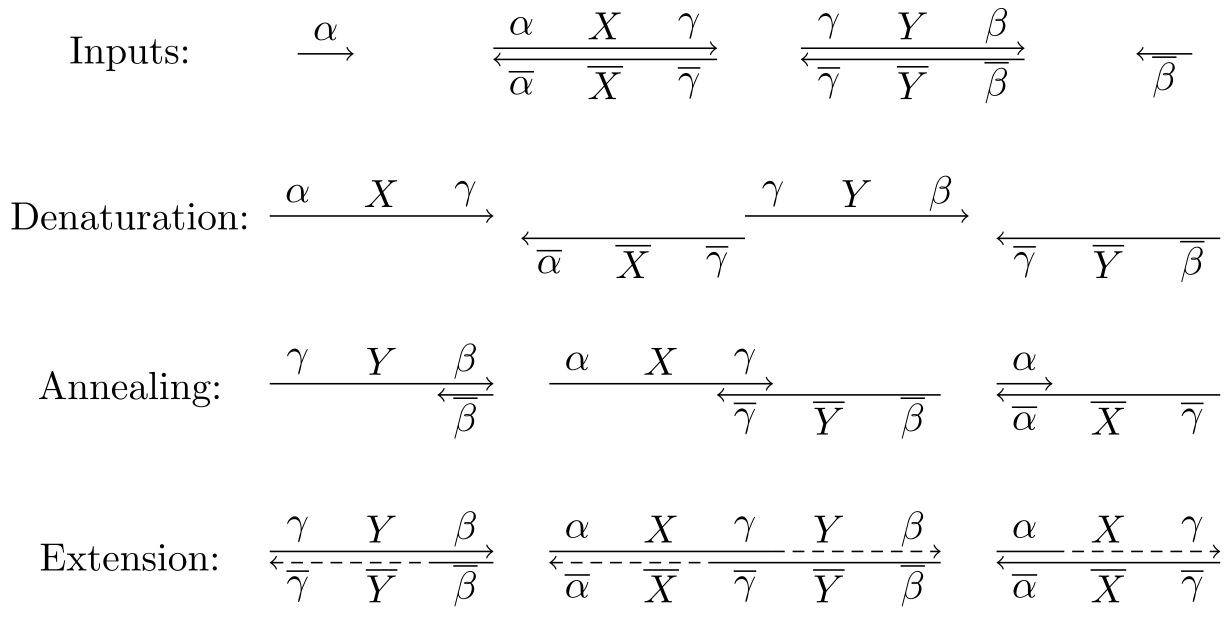


Figure 3.2: Cross-pairing polymerase chain reaction (XPCR). XPCR can be used to create and amplify a recombination of two genes. The inputs include two templates (DNA double strands $\alpha X\gamma$ and $\gamma Y\beta$ in the first row), two primers (DNA single strands α and $\overline{\beta}$ in the first row), nucleotides (not shown), and DNA polymerase (not shown). One cycle of XPCR has the three following steps: denaturation, annealing, and extension. During the denaturation step of the first cycle, the templates break into single strands $\alpha X\gamma$, $\overline{\alpha X\gamma}$, $\gamma Y\beta$, and $\overline{\gamma Y\beta}$ because of an increase in temperature. During the annealing step of the first cycle, two of the single strands from the templates hybridize with primers ($\gamma Y\beta$ with $\overline{\beta}$ and $\overline{\alpha X\gamma}$ with α), and the other two single strands from the templates hybridize with each other ($\alpha X\gamma$ with $\overline{\gamma Y\beta}$) because of the decrease in temperature. Finally, during the extension step of the first cycle, the primers and the single strands of the recombinant are extended by DNA polymerases using the free-floating nucleotides present in the solution according to the template strands. In the following cycles, XPCR acts similarly to PCR, with primers α and $\overline{\beta}$ and a template $\alpha X\gamma Y\beta$. The recombinant DNA double strand $\alpha X\gamma Y\beta$ is amplified exponentially.

The closure properties of the Chomsky families of languages under overlap assembly, the decidabilities of some decision problems (such as emptiness, context-freeness and finiteness about the languages generated by overlap assembly), as well as properties about languages generated by the iterative application of overlap assembly, were studied in [72, 73]. In addition, the state complexity of overlap assembly was studied in [31].

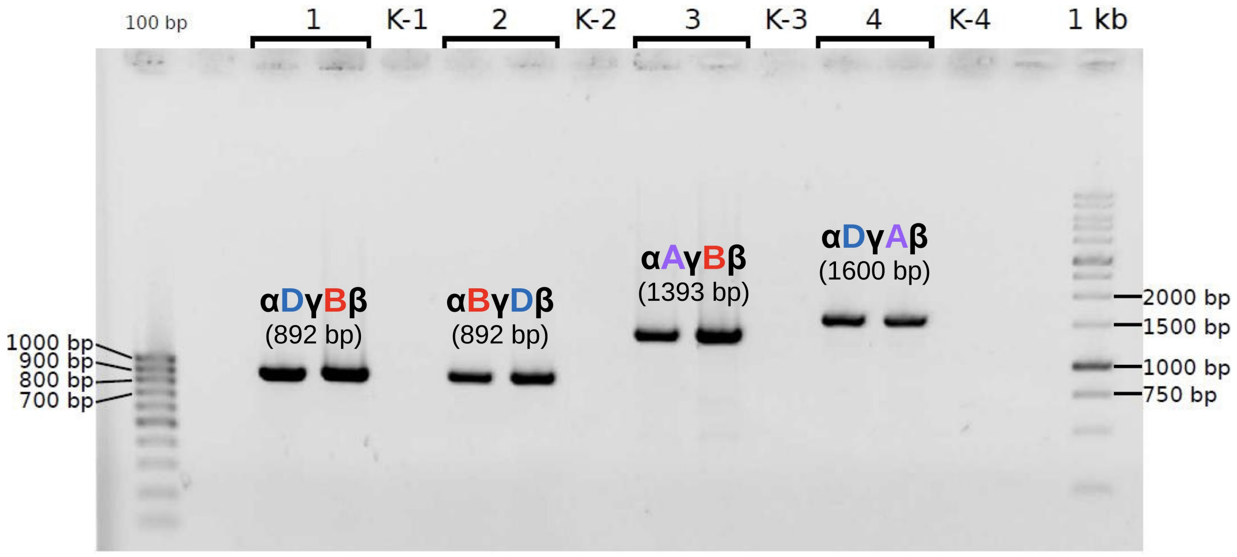


Figure 3.3: XPCR with templates containing different genes [83]. Lane 1: XPCR with templates $\alpha D\gamma$ (560 bp) and $\gamma B\beta$ (353 bp) amplifies the strand $\alpha D\gamma B\beta$ (892 bp). Lane 2: XPCR with templates $\alpha B\gamma$ (353 bp) and $\gamma D\beta$ (560 bp) amplifies the strand $\alpha B\gamma D\beta$ (892 bp). Lane 3: XPCR with templates $\alpha A\gamma$ (1,061 bp) and $\gamma B\beta$ (353 bp) amplifies the strand $\alpha A\gamma B\beta$ (1,393 bp). Lane 4: XPCR with templates $\alpha D\gamma$ (560 bp) and $\gamma A\beta$ (1,061 bp) amplifies the strand $\alpha D\gamma A\beta$ (1,600 bp). Lanes K-1, K-2, K-3, and K-4: Negative controls without templates for the reactions in Lanes 1, 2, 3, and 4, respectively.

As shown in Definition 3.13, the operands x and y can be decomposed as $x = uv$ and $y = vw$, where u and w are non-overlapping parts, and v is the overlapping part. Some related operations are summarized as follows with different length restrictions on these (non-)overlapping parts of the operands.

- *Restricted assembly*: The non-overlapping parts u and w are non-empty [55].
- Word operation denoted by $\bigotimes$: The concatenation of the non-overlapping parts uw are non-empty, but the overlapping part v can be empty [122]. Note that $a^* \bigotimes b^* = \{a^+b^*\} \cup \{a^*b^+\}$, whereas $a^* \overline{\odot} b^* = \emptyset$.
- *Max chop* denoted by $\odot_{\max}$: Only the longest overlapping part is considered [108].

- *Min chop* denoted by $\bigodot_{\min}$: Only the shortest overlapping part is considered [108].
- *Short concatenation*: Only the longest overlapping part is considered, and the overlapping part can be empty [35].
- Word operation denoted by $\bigodot_N$: The length of the overlapped part is required to be at least $N \in \mathbb{N}^+$ [66].
- *Fusion* denoted by $\bigodot$/*Latin product*: The overlapped part is a single character [10, 89, 106, 107].

3.4 Word blending

This section contains updated results from a paper I co-authored [76], and if a proof was already included in [75], it will be omitted.

As shown in Section 3.3, the XPCR procedure has been successfully used to join two different genes if they are attached to compatible primers [83]. Formally, $\alpha X\gamma$ and $\gamma Y\beta$ were combined to produce $\alpha X\gamma Y\beta$ (X and Y are gene sequences, and α, γ, and β are primers). However, when $X = Y$, that is, when two sequences containing the same gene were combined by XPCR, the result was not as expected. More specifically, when using XPCR with the input $\alpha X\gamma$, $\gamma X\beta$, α, and β, instead of obtaining the expected $\alpha X\gamma X\beta$, the experiments repeatedly produced the result $\alpha X\beta$.

In this section, we define and investigate a binary word and language operation called *word blending*, which formalizes the following generalization of this experimentally observed outcome of XPCR: The word blending of two words $\alpha X\gamma_1$ and $\gamma_2 X\beta$ that share a non-empty overlap X results in $\alpha X\beta$. Note that while $\gamma_1 = \gamma_2$ was required by XPCR, it is not required for our definition of the operation. Interestingly, this phenomenon has been observed independently in linguistics [91], under the name "blend word" or "portmanteau," and is responsible for the creation of words in the English language such as sm$\underline{\text{o}}$g (sm$\underline{\text{o}}$ke + f$\underline{\text{o}}$g), labrad$\underline{\text{o}}$odle (labrad$\underline{\text{o}}$r + p$\underline{\text{o}}$odle), emoticon (emot$\underline{\text{i}}$on + $\underline{\text{i}}$con), and Brangelina (Br$\underline{\text{a}}$d + $\underline{\text{A}}$ngelina).

We model this string recombination as follows.

Definition 3.14. *Given two words x, y over an alphabet Σ, we define the word blending or, simply, blending of x with y as*

$$x \bowtie y = \{z \in \Sigma^+ \mid \exists \alpha, \beta, \gamma_1, \gamma_2 \in \Sigma^*, \exists w \in \Sigma^+ : x = \alpha w \gamma_1, y = \gamma_2 w \beta, z = \alpha w \beta\}.$$

Note that the result of blending between two words is the empty set (the operation is undefined) if the words do not share a non-empty infix. If one or both words are empty, the operation is similarly undefined. The definition of blending can be extended to languages L_1 and L_2 by

$$L_1 \bowtie L_2 = \bigcup_{x \in L_1, y \in L_2} x \bowtie y.$$

First, for example, consider $u = bbac$ and $v = caab$. Then we have

$$u \bowtie v = \{b, bb, bbab, bbaab, bbacaab\},$$

$$v \bowtie u = \{c, cac, caac, caabac, caabbac\}.$$

Next, consider $L_1 = a^*b^*$ and $L_2 = b^*c^*$. Then we have $L_1 \bowtie L_2 = a^*b^+c^*$ and $L_2 \bowtie L_1 = b^+$. It is clear from this that blending is not commutative.

We emphasize that the definition of blending is a generalization of the experimentally observed outcome of XPCR on DNA strings. Indeed, to match the experimentally observed process, we would have to take $\gamma_1 = \gamma_2$ in Definition 3.14. We also note that for a realistic model, we would need additional restrictions, such as the fact that the words w, γ_1, and γ_2 should be of a sufficient length and that these words should not appear as a substring in the other strings involved.

The blending operation resembles the Latin product [89] and the crossover operation [36, 184]. The Latin product, denoted by $\diamond$, was defined as $u \diamond v = u'av'$ for $u = u'a$ and $v = av'$, where $a \in \Sigma$ and $u', v' \in \Sigma^*$, and, by definition, $u \diamond \lambda = \lambda \diamond u = u$. Even though, as will be proved in Lemma 3.15, word blending is equivalent to single-letter-overlap word blending, the Latin product differs from blending: In the Latin product, the overlap and blending can only occur at the extremities of the words, while in word blending it can happen anywhere inside the two operand words. Given a subset of the alphabet $M \subseteq \Sigma$, the *crossover* operation $\sharp_M$ is defined as $L_1 \sharp_M L_2 = \text{pref}(L_1) \diamond_M \text{suff}(L_2)$, where $\diamond_M$ is a restriction of the Latin product, defined by

$$u \diamond_M v = \begin{cases} u'av' & \text{if } u = u'a,\ v = av',\ a \in M,\ u', v' \in \Sigma^*, \\ \text{undefined} & \text{otherwise.} \end{cases}$$

By definition, $u \diamond_M \lambda = \lambda \diamond_M u = u$, and the crossover on languages is defined as $L_1 \diamond_M L_2 = \bigcup_{u \in L_1, v \in L_2} (u \diamond_M v)$. The blending operation resembles the crossover operation $\sharp_M$ with $M = \Sigma$. However, due to its biological motivation, unlike the Latin product and the crossover operation, blending is not defined when one of the operands is λ, and this leads to additional differences. For example, the crossover using $M = \Sigma = \{a, b\}$ between ab and ba is $ab\sharp_\Sigma ba = \mathrm{pref}(ab) \diamond \mathrm{suff}(ba) = \{\lambda, a, b, ab, ba, aba\}$, while the Latin product of the same words is $ab \diamond ba = \{aba\}$, and the blending between the same words is $ab \bowtie ba = \{a, aba\}$. Also, both the Latin product and the crossover operation $\sharp_M$ for a given M are associative, while word blending is not. For example, $(ba \bowtie a) \bowtie b^*a = ba \bowtie b^*a = b^+a$, while $ba \bowtie (a \bowtie b^*a) = ba \bowtie a = ba$.

One can extend the blending operation to an iterated version as follows. For $L \subseteq \Sigma^*$, the *iterated (word) blending* of L is defined by $L^{\bowtie_0} = L$, and $L^{\bowtie_i} = L \bowtie L^{\bowtie_{i-1}}$, $i \geq 1$. We define the iterated blending closure of L by

$$L^{\bowtie_*} = \bigcup_{i \geq 0} L^{\bowtie_i}.$$

We observe that the result of the iterated blending operation resembles the result of a splicing system. It is easy to see that the language $L^{\bowtie_*}$ can be generated using a set of null-context splicing rules. In fact, we can show that iterated blending can be expressed as a simple splicing system, where the crossover operation described above was also introduced as "one step" of a simple splicing system. The relationship between iterated blending and splicing will be discussed in greater detail in Section 3.4.1.

3.4.1 Closure properties

In this section, we prove that the families of regular, context-free, and recursively enumerable languages are closed under blending, but that the family of context-sensitive languages is not. The section also contains closure properties of the Chomsky families under the right- and left-inverse of blending as well as under iterated blending.

The following lemma shows that blending is equivalent to a restricted version where only one-letter overlaps are utilized. A similar simplification was made for synchronized insertion and deletion and hairpin inversion by Daley et al. [56].

Lemma 3.15. *If x, y are non-empty words over Σ, then*

$$x \bowtie y = \{z \in \Sigma^+ \mid \exists \alpha, \beta, \gamma_1, \gamma_2 \in \Sigma^*, \exists a \in \Sigma : x = \alpha a \gamma_1, y = \gamma_2 a \beta, z = \alpha a \beta\}.$$

Proof. Let A denote the right-hand side of the equality. The inclusion $A \subseteq x \bowtie y$ is obvious by the definition of blending. To prove the converse, let $z \in x \bowtie y$. Then $z = \alpha w \beta$ for some $\alpha, \beta, \gamma_1, \gamma_2 \in \Sigma^*$ and $w \in \Sigma^+$ such that $x = \alpha w \gamma_1$, $y = \gamma_2 w \beta$. As $w \in \Sigma^+$, w can be written as $w = w_1 a$, where $w_1 \in \Sigma^*$ and $a \in \Sigma$. It follows that $x = \alpha w_1 a \gamma_1$, $y = \gamma_2 w_1 a \beta$, and $z = \alpha w_1 a \beta$, that is, $x = \alpha' a \gamma_1$, $y = \gamma_2' a \beta$, and $z = \alpha' a \beta$ with $\alpha' = \alpha w_1 \in \Sigma^*$ and $\gamma_2' = \gamma_2 w_1 \in \Sigma^*$. Thus, $z \in A$, which proves equality. $\square$

The above lemma indicates that word blending, $\bowtie$, is a generalization of the chop operation (also called fusion), as studied in [106,107], whereby $u \odot v$ equals $u'av'$ if $u = u'a$, $v = av'$, $u', v' \in \Sigma^*$, and $a \in \Sigma$, and it is undefined otherwise. Also note that the chop operation differs from the Latin product only when one of the operands is λ, in which case the result of the Latin product equals the other operand, while the chop operation (fusion) is undefined. Lemma 3.15 can be extended to languages in the natural way. From this lemma, we can show that the blending of two languages can be obtained by combining the right quotient, catenation, left quotient, and union operations as follows.

Proposition 3.16. *Given languages $L_1, L_2 \subseteq \Sigma^+$,*

$$L_1 \bowtie L_2 = \bigcup_{a \in \Sigma} \left(L_1 (a\Sigma^*)^{-1_r}\right) a \left((\Sigma^* a)^{-1_l} L_2\right).$$

Proof. Let $z \in L_1 \bowtie L_2$. Then, by Lemma 3.15, $z = \alpha a \beta$ for some $x \in L_1$ and $y \in L_2$ such that $x = \alpha a \gamma_1$ and $y = \gamma_2 a \beta$, where $a \in \Sigma$ and $\alpha, \beta, \gamma_1, \gamma_2 \in \Sigma^*$. It is clear that $\alpha \in L_1 (a\Sigma^*)^{-1_r}$ and $\beta \in (\Sigma^* a)^{-1_l} L_2$, so $z = \alpha a \beta \in (L_1 (a\Sigma^*)^{-1_r})\, a\, ((\Sigma^* a)^{-1_l} L_2)$.

Conversely, let $z \in \bigcup_{a \in \Sigma} (L_1 (a\Sigma^*)^{-1_r})\, a\, ((\Sigma^* a)^{-1_l} L_2)$. Then there exists a letter $a \in \Sigma$ and words $\alpha, \gamma_1, \gamma_2, \beta \in \Sigma^*$, such that $z = \alpha a \beta$, where $x = \alpha a \gamma_1 \in L_1$ and $y = \gamma_2 a \beta \in L_2$, which implies that $z \in L_1 \bowtie L_2$. $\square$

Corollary 3.17. *The families of regular, context-free, and recursively enumerable languages are closed under blending.*

Proof. It follows from Proposition 3.16 and the fact that families of regular, context-free, and recursively enumerable languages are closed under left/right quotient with regular languages, catenation, and union [243]. $\square$

A counterexample can be found to prove the following proposition [75].

Proposition 3.18. *The family of context-sensitive languages is not closed under blending.*

Next, we consider the left- and right-inverses of blending.

The right-inverse of an operation does not always resemble a binary operation in the traditional sense. For example, if we define the operation $u * v = \{a^{|u|} \mid a \in \Sigma\}$, where Σ

is the alphabet, then the right-inverse of this operation, $x *_r^{-1} y$, equals Σ^* if $y = a^{|x|}$ and equals $\emptyset$ if $y \neq a^{|x|}$. However, the relation $y \in x *_r^{-1} (x * y)$ still holds, and this will be sufficient for solving language equations involving binary word operations, as detailed in Section 3.4.2.

Also note that the right-inverse of a binary word operation inherits some of the properties of the original binary operation. For example, a standard property of many familiar binary word operations (catenation, left/right quotient, insertion, deletion, shuffle, etc.) is that they have a right (left) identity: A binary word operation with a *right identity* is a binary word operation $\oplus$ with the property $u \oplus \lambda = \{u\}$ for all $u \in \Sigma^*$. The notion of binary word operation with a *left identity* is defined similarly. For example, catenation, insertion, and shuffle are binary operations with both right and left identities, while deletion and left/right quotient are binary word operations with a right identity only. The operation of word blending has neither a right identity nor a left identity. In general, the following result holds.

Proposition 3.19. *If $\oplus$ is a binary word operation with a left identity, then its right-inverse $\oplus_r^{-1}$ is also a binary operation with a left identity.*

Proof. Let v be a word in $(\lambda \oplus_r^{-1} w)$. By definition of the right-inverse, $v \in (\lambda \oplus_r^{-1} w)$ if and only if $w \in (\lambda \oplus v)$, which, in turn, equals $\{v\}$, as $\oplus$ is an operation with a left identity. Thus, for all $v \in (\lambda \oplus_r^{-1} w)$, we have that $v = w$, which means that $\{w\} = (\lambda \oplus_r^{-1} w)$, so $\oplus_r^{-1}$ is a binary operation with a left identity. $\square$

Proposition 3.20. *The right-inverse of a binary word operation $\oplus$ is unique.*

Proof. Assume the contrary, that is, assume that a binary word operation $\oplus$ has two distinct right-inverses $\oplus_1$ and $\oplus_2$. By definition, for all words $u, y, w \in \Sigma^*$, the following relations hold: $w \in (u \oplus y)$ if and only if $y \in (u \oplus_1 w)$, and $w \in (u \oplus y)$ if and only if $y \in (u \oplus_2 w)$.

Since $\oplus_1$ and $\oplus_2$ are distinct, there exist some words α, β such that $(\alpha \oplus_1 \beta) \neq (\alpha \oplus_2 \beta)$. Without loss of generality, assume that there exists a word $\gamma \in (\alpha \oplus_1 \beta)$ such that $\gamma \notin (\alpha \oplus_2 \beta)$.

By definition of $\oplus_1$, from $\gamma \in (\alpha \oplus_1 \beta)$, we have that $\beta \in (\alpha \oplus \gamma)$, which implies that $\gamma \in (\alpha \oplus_2 \beta)$ by definition of $\oplus_2$. This contradicts our assumption about γ, so the right-inverse of a binary word operation is unique. $\square$

Definition 3.21. *For two words $u, w \in \Sigma^*$, the binary word operation $\bowtie_r^{-1}$ is defined as*

$$u \bowtie_r^{-1} w = \bigcup_{a \in \Sigma} \Sigma^* a (u (a\Sigma^*)^{-1_r} a)^{-1_l} w.$$

The definition of $\bowtie_r^{-1}$ *can be extended to languages by*

$$L_1 \bowtie_r^{-1} L_2 = \bigcup_{u \in L_1, w \in L_2} \left(\bigcup_{a \in \Sigma} \Sigma^* a (u (a\Sigma^*)^{-1_r} a)^{-1_l} w) \right).$$

We observe that, similar to insertion, deletion, and other binary word operations [141], the result of $u \bowtie_r^{-1} w$ is in general a set of words (not necessarily a singleton word). In general, the operation $\bowtie_r^{-1}$ can be used to recover the right operand of blending: The right operand (a word) belongs to the set obtained by applying $\bowtie_r^{-1}$ to the other operand and one of the words in the result of blending.

For example, let $\Sigma = \{a, b\}$. We have that $aabb \bowtie_r^{-1} aab = \Sigma^* aab \cup \Sigma^* ab \cup \Sigma^* b = \Sigma^* b$. The result of blending $aabb$ with ab is $aabb \bowtie ab = \{ab, \underline{aab}, aabb\}$. The right operand ab is an element of the set obtained by applying $\bowtie_r^{-1}$ to the other operand and one of the words of the result of $\bowtie$: $ab \in aabb \bowtie_r^{-1} \underline{aab} = \Sigma^* b$.

We can show that the operation $\bowtie_r^{-1}$ is the unique right-inverse of $\bowtie$ [75].

Proposition 3.22. *The operation* $\bowtie_r^{-1}$ *is the right-inverse of* $\bowtie$.

Corollary 3.23. *The operation* $\bowtie_r^{-1}$ *is the unique right-inverse of the blending operation* $\bowtie$.

We also study the closure properties of the Chomsky families of languages under $\bowtie_r^{-1}$.

Corollary 3.24. *The families of regular and recursively enumerable languages are closed under the right-inverse of blending. Moreover, if* L_1 *is an arbitrary language and* L_2 *is a regular language, then* $L_1 \bowtie_r^{-1} L_2$ *is regular; if* L_1 *is a regular language and* L_2 *is a context-free language, then* $L_1 \bowtie_r^{-1} L_2$ *is context free.*

Proof. The proof follows from Proposition 3.22. The family of regular languages is closed under right quotient and catenation, and the family of context-free languages is closed under union, catenation, and left quotient with regular languages [141, 243], so the right-inverse of blending of a regular language with a context-free language is context free. □

A counterexample can be found to prove the following proposition [75].

Proposition 3.25. *The family of context-free languages is not closed under the right-inverse of blending.*

Proposition 3.26. *The family of context-sensitive languages is not closed under the right-inverse of blending.*

Proof. Let L_0 be a recursively enumerable language over Σ, which is not context sensitive. It is known that a context-sensitive language L_1 over $\Sigma \cup \{a, b\}$ with $a, b \notin \Sigma$ can be constructed such that L_1 is a subset of $\{a^i b\alpha \mid \alpha \in L_0, i \geq 0\}$ and, in addition, for every $\alpha \in L_0$, there is an $i \geq 0$ such that $a^i b\alpha \in L_1$ (see, for example, [243]).

The result now follows as $(L_1 \bowtie_r^{-1} \{a^* b\}) \cap b\Sigma^* = \{b\alpha \mid b \notin \Sigma, \alpha \in L_0\} = bL_0$, which is not context sensitive, and the family of context-sensitive languages is closed under intersection with regular languages. □

Next, we consider the left-inverse of blending. Similarly to Proposition 3.19, we can prove that:

Proposition 3.27. *If $\oplus$ is a binary word operation with a right identity, then its left-inverse $\oplus_l^{-1}$ is also a binary operation with a right identity.*

Define now the binary word operation $w \bowtie_l^{-1} v = (v^r \bowtie_r^{-1} w^r)^r$. The operation $\bowtie_l^{-1}$ is the left-inverse of $\bowtie$, and this can be proved similarly to Proposition 3.22 by invoking the mirror image operation. Because all the families of languages in the Chomsky hierarchy are closed under mirror image, their closure properties under the left-inverse of blending are the same as their closure properties under the right-inverse of blending. Lastly, similar to Proposition 3.20, we can also prove that the left-inverse of a binary operation is unique, and, as a consequence, the operation $\bowtie_l^{-1}$ is the unique left-inverse of the blending operation.

We now consider the iterated blending operation $\bowtie_*$. Recall that, as mentioned in Section 3.4, for every language $L \subseteq \Sigma^*$, the language $L^{\bowtie_*}$ can be generated by a splicing system with null-context splicing rules. This connection, together with Proposition 3.16, allows us to express iterated blending using so-called simple splicing systems [184].

Recall that simple splicing schemes are splicing schemes with splicing rules of the form $a\#\lambda\$a\#\lambda$ for $a \in \Sigma$. Note that for two languages L_1 and L_2 over Σ, we now have that

$$L_1 \bowtie L_2 = \bigcup_{x \in L_1, y \in L_2} \sigma_{\bowtie}(x, y),$$

where $\sigma_{\bowtie}$ is the simple splicing scheme $\sigma_{\bowtie} = (\Sigma, R)$ with $R = \{a\#\lambda\$a\#\lambda \mid a \in \Sigma\}$. Note that in a simple splicing scheme that expresses word blending, each and every letter of the alphabet must be used in a splicing rule. This observation, together with Proposition 3.16 which showed that the blending of two languages can be written as $L_1 \bowtie L_2 = \bigcup_{a \in \Sigma}(L_1(a\Sigma^*)^{-1_r})a((\Sigma^* a)^{-1_l} L_2)$, gives us the following result.

Proposition 3.28. *For every language $L \subseteq \Sigma^*$, we have $\sigma_{\bowtie}(L) = L \cup (L \bowtie L)$ and $\sigma^*_{\bowtie}(L) = L^{\bowtie_*}$.*

We note that the splicing scheme $\sigma_{\bowtie}$ is finite, since the number of rules depends only on the number of symbols in Σ, and is unary (its rules use words of length at most 1). We also note that, even though in [184] consideration is restricted to the case when L is a finite language, the properties of the splicing systems obtained therein imply the following closure properties.

Proposition 3.29. *The families of regular, context-free, and recursively enumerable languages are closed under iterated blending.*

It was shown in [184] that the class of languages generated by simple splicing systems is a subclass of the class of strictly locally testable languages, which is a subregular language class. The authors also showed that the class of languages generated by simple splicing systems contains the class of finite languages. This is done by constructing a simple splicing system with a finite language L as the initial language and introducing a new alphabet symbol to be used with a splicing rule. The effect is that no splicing can be performed in this system since no words in L contain the new symbol, so no new words are generated. This approach does not work in the case of iterated blending because we cannot restrict the use of blending, which must occur whenever two words have a one-letter overlap. The following example shows that, even though iterated blending is related to simple splicing, there are differences between the two: In contrast to the case of simple splicing (whereby every finite language can be generated by a simple splicing scheme), there exist finite languages that cannot be generated via iterated blending.

For example, let $L = \{aa\}$. We will show that L cannot be generated via iterated blending. Suppose that there exists a language B such that $B^{\bowtie_*} = L$. Then there exist words $u, v \in B$ such that $aa \in u \bowtie v$. This means either $u = aau'$ and $v = v'a$ for some $u', v' \in \Sigma^*$ or $u = au'$ and $v = v'aa$ for some $u', v' \in \Sigma^*$. In either case, together with $aa \in B^{\bowtie_*}$, we have $a^+ \subseteq B^{\bowtie_*} = L$, which is a contradiction.

3.4.2 Decision problems

This section investigates some decision problems related to the blending operation, such as the existence of solutions to language equations of the type $X \bowtie L = R$ and $L \bowtie Y = R$ (where L, R are given known languages and X, Y are unknown languages), the closure of languages under $\bowtie_*$, and the existence of a (finite) language base for a given language.

We also show that, for a given alphabet, there exist finitely many languages that can be obtained by iterated blending from an initial language.

Consider the question whether or not there exists a (singleton) solution to language equations of the type $X \bowtie L = R$ and $L \bowtie Y = R$ (where L, R are given known languages and X, Y are unknown languages), and we have the following results [75].

Proposition 3.30. *The existence of a solution Y to the equation $L \bowtie Y = R$ is decidable for the given regular languages L and R.*

We note that the solution to a language equation of the type $L \bowtie Y = R$ does not need to be unique.

For example, let $\Sigma = \{a, b\}$. The equation $\{a^n b^n | n \geq 0\} \bowtie Y = a^+ b^+$ has the maximal solution $Y_{\max} = a^* b^+ \cup \{\lambda\}$, but it also has the solution $Y = b^+ \subset Y_{\max}$.

As another example, the equation $\{a^n b^n | n \geq 0\} \bowtie Y = a^+$ has the maximal solution $Y_{\max} = (\{a^n b^n | n \geq 0\} \bowtie_r^{-1} (a^+)^c)^c = (\{a^n b^n | n \geq 0\} \bowtie_r^{-1} (\{\lambda\} \cup \Sigma^* b \Sigma^*))^c = (\{a^n b^n | n \geq 0\} \bowtie_r^{-1} \Sigma^* b \Sigma^*)^c = (\Sigma^* b \Sigma^*)^c = a^*$, but it also has the solution $Y = a^+ \subset Y_{\max}$.

Proposition 3.31. *The existence of a singleton solution $\{w\}$ to the equation $L \bowtie \{w\} = R$ is decidable for regular languages L and R.*

Proposition 3.32. *The existence of a singleton solution $\{w\}$ to the equation $L \bowtie \{w\} = R$ is undecidable for regular languages R and context-free languages L.*

Corollary 3.33. *The existence of a solution Y to the equation $L \bowtie Y = R$ is undecidable for regular languages R and context-free languages L.*

Proposition 3.34. *The existence of a (singleton) solution X to the equation $X \bowtie L = R$ is decidable for regular languages L and R, and it is undecidable for regular languages R and context-free languages L.*

Next, we consider the decidability of the question of whether a language is closed under iterated blending.

Proposition 3.35. *Let L be a regular language. It is decidable whether or not L is closed under $\bowtie_*$.*

Proof. By Proposition 3.28 and [221], we can construct an NFA A' that recognizes $L^{\bowtie_*}$. The testing equivalence of two NFAs is known to be decidable [112], so testing whether $L = L^{\bowtie_*}$ is decidable. $\square$

Proposition 3.36. *It is undecidable whether or not L is closed under $\bowtie_*$, where L is context free.*

Proof. Assume the contrary. Under this assumption, given an arbitrary context-free language L over an alphabet Σ, and a letter $\#$ not in Σ, we claim that the context-free language $B = \#L(\#\Sigma^*)^*$ is closed under $\bowtie_*$ if and only if $L = \Sigma^*$ or $L = \emptyset$.

It is clear that if $L = \Sigma^*$ or $L = \emptyset$, then B is closed under $\bowtie_*$. Now, we consider the other implication and assume that B is closed under $\bowtie_*$. If $B = \emptyset$, then $L = \emptyset$. If $B \neq \emptyset$, then $L \neq \emptyset$. Consider the languages $\#L$ and $\#L(\#\Sigma^*)^+$, both included in B. As B is closed under $\bowtie_*$, the blending of these two languages should be included in $B^{\bowtie_*}$, that is, $\#L \bowtie \#L(\#\Sigma^*)^+ = (\#\Sigma^*)^+ \subseteq B^{\bowtie_*}$. Thus, $B = (\#\Sigma^*)^+$, and this implies that $L = \Sigma^*$.

Thus, if we could decide the problem in the proposition, since emptiness is decidable for context-free languages [112], we would be able to decide whether or not $L = \Sigma^*$ for arbitrary context-free languages L, which is impossible [112]. □

Let $L, B \subseteq \Sigma^*$ be two languages. We say that B is a *base* of L (with respect to $\bowtie_*$) if $L = B^{\bowtie_*}$. In [184], it is shown that it is decidable whether or not a regular language is generated by a simple splicing scheme and a finite language base, and we can prove similar results with respect to $\bowtie_*$.

Note that a language L is closed under $\bowtie_*$ if and only if it has a base with respect to $\bowtie_*$. If L is closed under $\bowtie_*$, L is a base for itself. Otherwise, if L is not closed under $\bowtie_*$, it does not have a base. Indeed, if it had a base B, we would have that $L = B^{\bowtie_*} = (B^{\bowtie_*})^{\bowtie_*} = L^{\bowtie_*}$, which is a contradiction.

A language L is said to be an *iterated blending language* if it can be generated by iterated blending, that is, if there exists a base B, such that $L = B^{\bowtie_*}$. Some languages are iterated blending languages and some are not, as shown by the following examples.

Let $\Sigma = \{0, 1\}$. The language $L_1 = \Sigma^*0$, which consists of the binary representations of all even natural numbers, is an iterated blending language because it can be generated by applying iterated blending to the base $B_1 = L_1$ or to the finite base $B_2 = \{110, 100, 010, 000\}$. On the other hand, the language L_2, consisting of the binary representations of all prime numbers, is not an iterated blending language because it is not closed under $\bowtie_*$.

Proposition 3.35 and Proposition 3.36 can now be rephrased as follows: The question of whether or not a language L is an iterated blending language (has a base) is decidable if L is regular, and it is undecidable if L is context free. The next proposition answers the analogous question regarding the existence of a finite base.

Proposition 3.37. *The question of whether or not a language L has a finite base with respect to $\bowtie_*$ is decidable if L is regular, and it is undecidable if L is context free.*

Proof. The proof is the same as that of the question of whether or not a language L can be generated by a simple splicing system, which was proved to be decidable if L is regular and undecidable if L is context free [184]. $\square$

A similar proof idea can also be used to prove the following result, showing that a language has a base if and only if it has a finite base.

Proposition 3.38. *Let $L \subseteq \Sigma^*$ be an arbitrary language. There exists a language $R \subseteq \Sigma^*$ such that $L = R^{\bowtie_*}$ if and only if $L = B^{\bowtie_*}$, where $B = L \cap \{w \in \Sigma^* \mid |w|_a \leq 2$ for all $a \in \Sigma\}$.*

Proof. It is clear that if $L = B^{\bowtie_*}$, then there exists a language $R \subseteq \Sigma^*$ such that $L = R^{\bowtie_*}$. For the other implication, assume that there exists a language $R \subseteq \Sigma^*$ such that $L = R^{\bowtie_*}$. Since $L = R^{\bowtie_*}$, by definition, R is a base for L with respect to $\bowtie_*$. Therefore, L is closed under $\bowtie_*$, and $L^{\bowtie_*} = L = R^{\bowtie_*}$.

Let $w \in L$ such that there exists $a \in \Sigma$ with $|w|_a \geq 3$, and let $S_w^0 = \{w\}$. Since $|w|_a \geq 3$, the word w can be decomposed as $w = w_1 a w_2 a w_3 a w_4$, where $w_1 \in \Sigma^*$ and $w_2, w_3, w_4 \in (\Sigma \setminus \{a\})^*$. Consider the words $w' = w_1 a w_2 a w_4$ and $w'' = w_1 a w_3 a w_4$. It is clear that $w \in w' \bowtie w''$, that $w' \in w \bowtie w \subseteq L$, and that $w'' \in w \bowtie w \subseteq L$.

Consider now the set $S_w^1 = (S_w^0 \setminus \{w\}) \cup \{w', w''\}$. We have that $w \in (S_w^1)^{\bowtie_*}$, $(S_w^1)^{\bowtie_*} \subseteq L^{\bowtie_*}$, and $|w'|_a = |w''|_a = |w|_a - 1$. If there exists a word in S_w^1 wherein the number of occurrences of a is at least 3, then we repeat the process for this word to obtain a new set; otherwise, choose another letter $b \in \Sigma \setminus \{a\}$, and repeat the process. By repeating this process, after a finite number of steps, we obtain a set S_w such that $S_w \subseteq B$, where $B = L \cap \{w \mid |w|_a \leq 2$ for all $a \in \Sigma\}$. Moreover, $w \in S_w^{\bowtie_*}$, $S_w^{\bowtie_*} \subseteq B^{\bowtie_*}$, and $S_w^{\bowtie_*} \subseteq L$. It is clear that

$$\bigcup_{w \in L \setminus B} S_w \cup \{w \in L \mid |w|_a \leq 2 \text{ for all } a \in \Sigma\} = B.$$

We claim that $B^{\bowtie_*} = L$. Indeed, since $B \subseteq L$ and L is closed under $\bowtie_*$, we have that $B^{\bowtie_*} \subseteq L$. For the other inclusion, if $w \in L$ and $|w|_a \leq 2$ for all $a \in \Sigma$, then $w \in B \subseteq B^{\bowtie_*}$. Otherwise, that is, if $w \in L$ but w has at least three occurrences of a for some $a \in \Sigma$, we have that $w \in S_w^{\bowtie_*} \subseteq B^{\bowtie_*}$. This proves that $L \subseteq B^{\bowtie_*}$. $\square$

Corollary 3.39. *Given a language $L \subseteq \Sigma^*$, the following are equivalent:*

- *L is closed under iterated blending.*
- *L is an iterated blending language.*
- *L has a base R with respect to iterated blending, i.e., such that $R^{\bowtie_*} = L$.*
- *L has a finite base B with respect to iterated blending, i.e., such that $B^{\bowtie_*} = L$.*

Corollary 3.40. *For an arbitrary language $R \subseteq \Sigma^*$, $R^{\bowtie_*}$ is regular. All the families of languages in the Chomsky hierarchy are closed under $\bowtie_*$.*

Corollary 3.41. *Given an alphabet Σ, there are finitely many iterated blending languages over Σ.*

Proof. Consider an alphabet Σ of cardinality n. Every word of length at least $2n+1$ must have a letter repeated at least 3 times, so there are at most $\sum_{i=0}^{2n} n^i$ different words containing each letter at most twice. Therefore, by Proposition 3.38, there are at most $2^{\sum_{i=0}^{2n} n^i}$ different bases, and thus there are finitely many iterated blending languages. $\square$

3.4.3 State complexity

By Proposition 3.16, the family of regular languages is closed under blending. Thus, we can consider the state complexity of blending on two regular languages. Recall from Proposition 3.16 that the blending of two languages can be expressed as a series of union, catenation, and quotient operations. While the state complexity of each of these operations is known, the state complexity of a combination of operations is not necessarily the same as the composition of the state complexities of the operations [245].

We construct a DFA that recognizes the language of the blending of the two languages L_m and L_n, recognized respectively by DFAs $A_m = (Q_m, \Sigma, s_m, F_m, \delta_m)$ and $A_n = (Q_n, \Sigma, s_n, F_n, \delta_n)$. We construct a DFA $A' = (Q', \Sigma, s', F', \delta')$, where

- $Q' = Q_m \times 2^{Q_n}$;
- $s' = (s_m, \emptyset)$;
- $F' = \{(q, P) \in Q_m \times 2^{Q_n} \mid P \cap F_n \neq \emptyset\}$; and

- $\delta'((q, P), a) = (\delta_m(q, a), P')$ for $a \in \Sigma$, where

$$P' = \begin{cases} \bigcup_{p \in P} \delta_n(p, a) & \text{if } \delta_m(q, a) \text{ is the sink state,} \\ \bigcup_{p \in Q_n} \delta_n(p, a) & \text{otherwise.} \end{cases}$$

Each state of A' is a pair consisting of a state of A_m and a subset of states of A_n. Informally, we can divide the computation of a word into two phases. In the first phase, states of the form (q, P) are reached where q is not the sink state of A_m. Here, the set P is determined solely by the input symbol as the machine tries to guess the symbol on which the blending occurs. In the second phase, the machine reaches states $(q_\emptyset, P)$, where $q_\emptyset$ is the sink state of A_m. The second phase only occurs when the blending occurs, and the input that has been read is no longer a prefix of a word recognized by A_m. In this phase, the set P is determined by the transition function of A_n. It follows from this construction that A' recognizes the language $L_m \bowtie L_n$.

A simple count shows that the number of states in the state set of A' is $m \cdot 2^n$. We will show that, depending on the size of the alphabet, not all of these states are necessarily reachable. First, we consider the case where the alphabet is unary.

Proposition 3.42. *Let L_m and L_n be regular languages defined over a unary alphabet such that L_m is recognized by an m-state DFA, and L_n is recognized by an n-state DFA. Then the state complexity of $L_m \bowtie L_n$ is $m + n - 2$ if both L_m and L_n are finite or 2 otherwise. Furthermore, this bound is reachable.*

Proof. Recall that by Proposition 3.16, $L_m \bowtie L_n = (L_m(a^+)^{-1_r})a((a^+)^{-1_l}L_n)$. If either L_m or L_n is infinitely large, then we have $L_m \bowtie L_n = a^+$, in which case the state complexity of $L_m \bowtie L_n$ is 2. If both L_m and L_n are finite, then it is easy to see that the state complexity of $L_m \bowtie L_n$ is $m + n - 2$. □

Now, we consider the state complexity when the languages are defined over alphabets of a size greater than 1.

Lemma 3.43. *The DFA A' requires at most $(m - 1) \cdot (k - 1) + 2^n + 1$ states, where $k = |\Sigma| \leq 2^n$.*

Lemma 3.44. *Let $m, n \geq 3$ and $4 \leq k < 2^n$. There exist families of DFAs A_m with m states and B_n with n states defined over an alphabet with k letters such that a DFA recognizing $A_m \bowtie B_n$ requires at least $(m - 1) \cdot (k - 1) + 2^n + 1$ states.*

Proof. Let $\Sigma = \{a_1, \ldots, a_{k-2}, b, c\}$. We define the DFAs A_m and B_n over Σ.

Let $A_m = (Q_m, \Sigma, s_m, F_m, \delta_m)$, where $Q_m = \{0, \ldots, m-1\}$, $s_m = 0$, and $F_m = \{m-2\}$. We define the transition function δ_m by

- $\delta_m(p, a_i) = p$ for all $0 \leq p \leq m-2$ and $1 \leq i \leq k-2$;
- $\delta_m(p, b) = p + 1$ for $0 \leq p \leq m-2$;
- $\delta_m(p, c) = m - 1$ for $0 \leq p \leq m-2$; and
- $\delta_m(m-1, a) = m - 1$ for all $a \in \Sigma$.

The DFA A_m is shown in Figure 3.4.

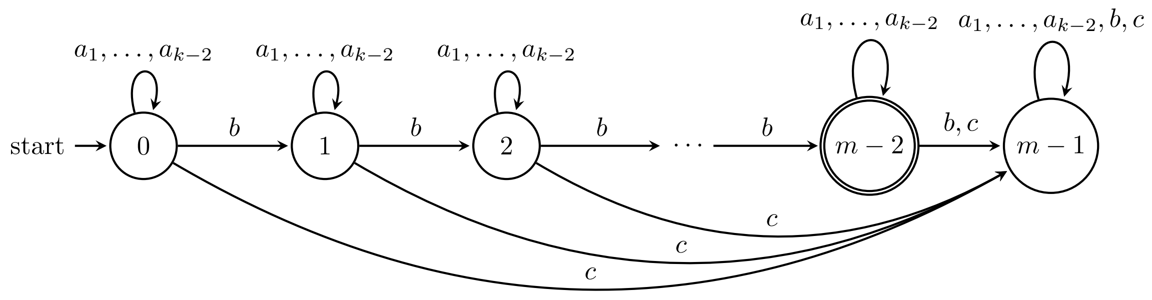


Figure 3.4: The DFA A_m.

Let $B_n = (Q_n, \Sigma, s_n, F_n, \delta_n)$, where $Q_n = \{0, \ldots, n-1\}$, $s_n = 0$, and $F_n = \{n-1\}$. We define the transition function δ_n by

- $\delta_n(q, b) = q + 1 \bmod n$ for $0 \leq q \leq n-1$; and
- $\delta_n(q, c) = q$ for $0 \leq q \leq n-1$.

For transitions on symbols a_i with $1 \leq i \leq k-2$, we define an enumeration of the subsets of Q_n and let $Q_n[i]$ be the i^{th} subset of Q_n. Every arbitrary enumeration of the subsets of Q_n suffices for this proof subject to the condition that:

- for $0 \leq i \leq k-2$, each i corresponds to a distinct subset of Q_n (that is, $Q_n[i] \neq Q_n[j]$ if and only if $i \neq j$ for $0 \leq i, j \leq k-2$); and

- we reserve the following: $Q_n[0] = Q_n$, $Q_n[1] = \{0, 1, \dots, n-2\}$, $Q_n[2] = \{0\}$.

Also note that while we have defined $Q_n[0]$, there are no symbols a_0. We will show later that, by our definitions, the role of a_0 will be played by b. If $k > 2^n$, then this property cannot hold, but it is clear that we can enumerate all 2^n subsets of Q_n.

Then we define transitions on $a_i \in \Sigma$ by

$$\delta_n(q, a_i) = \begin{cases} q & \text{if } q \in Q_n[i], \\ (q + \min_{(q+j \bmod n) \in Q_n[i]} j) \bmod n & \text{if } q \notin Q_n[i]. \end{cases}$$

In other words, for each state $q \in Q_n$, the transition on the symbol a_i goes to the "next" state that is in $Q_n[i]$. If $q \in Q_n[i]$, then that q itself is the "next" state.

The DFA B_3 is shown in Figure 3.5, with $Q_3[i]$ defined for $0 \leq i \leq 6$ as follows:

$$\begin{array}{lll} Q_3[0] = \{0,1,2\}, & & \\ Q_3[1] = \{0,1\}, & Q_3[2] = \{0\}, & Q_3[3] = \{1\}, \\ Q_3[4] = \{2\}, & Q_3[5] = \{0,2\}, & Q_3[6] = \{1,2\}. \end{array}$$

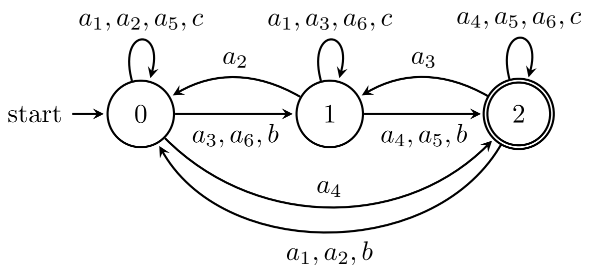


Figure 3.5: The DFA B_3.

We will show that A' contains $(m-1) \cdot (k-1) + 2^n + 1$ reachable and distinguishable states.

First, to show that the states are reachable, we note that $s' = (s_m, \emptyset)$ is clearly reachable as the initial state. Then, we observe that for $1 \leq i \leq k-2$, the state $(q, Q_n[i])$ with $q \in Q_m \setminus \{m-1\}$ is reachable on the word $b^q a_i$, and $(q, Q_n[0])$ is reachable on the word b^q. Since the only symbol not used here is c, this gives us $(m-1) \cdot (k-1)$ states.

Now we consider states of the form $(m-1, P)$, where $P \subseteq Q_n$. Observe that $(m-1, Q_n)$ can be reached on the word b^{m-1}. Also, note that $(m-1, \emptyset)$ can be reached on the letter c from $(0, \emptyset)$.

Next, we will show that all states of the form $(m-1, P)$, where $P = Q_n \setminus T$, for some $T \subseteq Q_n$, are reachable from $(m-1, Q_n)$ by induction on $|T|$. First, consider $|T| = 1$, $T = \{t\}$, $0 \leq t \leq n-1$. Then, we have $(m-1, Q_n) \xrightarrow{a_1 b^{t+1}} (m-1, Q_n \setminus \{t\})$.

Assume that all states $(m-1, Q_n \setminus T')$ are reachable from $(m-1, Q_n)$, where $u = |T'| \geq 1$. We will show that all states $(m-1, Q_n \setminus T)$ are reachable from $(m-1, Q_n)$, where $|T| = u+1 < |Q_n|$. Let $P = Q_n \setminus T = \{t_1, t_2, \ldots, t_{l-1}\}$, where elements in P are in ascending order. If $t_1 = 0$, $t_{l-1} \neq n-1$, then we have

$$(m-1, \{0, t_2, \ldots, t_{l-1}, n-1\}) \xrightarrow{a_1} (m-1, \{0, t_2, \ldots, t_{l-1}\}) = (m-1, P). \tag{3.1}$$

Thus, $(m-1, P)$ is reachable from the state $(m-1, P \cup \{n-1\})$, which is reachable from $(m-1, Q_n)$ by assumption.

If $t_1 = 0$ and $t_{l-1} = n-1$, there exists a largest integer $v \notin P$, where $1 \leq v < n-1$, then we have

$$(m-1, \{w + n - v - 1 \bmod n \mid w \in P\}) \xrightarrow{b^{v+1}} (m-1, P).$$

Thus, $(m-1, P)$ is reachable from $(m-1, Q_n)$ by Equation 3.1.

If $t_1 > 0$, then we have

$$(m-1, \{0, t_2 - t_1, \ldots, t_{l-1} - t_1\}) \xrightarrow{b^{t_1}} (m-1, \{t_1, t_2, \ldots, t_{l-1}\}).$$

That is, $(m-1, P)$ is reachable from $(m-1, Q_n)$ by Equation 3.1. Thus, all states $(m-1, Q_n \setminus T)$ with $|T| = u+1$ are reachable.

Thus, we have an additional 2^n reachable states of the form $(m-1, P)$, giving us a total of $(m-1) \cdot (k-1) + 2^n + 1$ reachable states.

Next, we will show that these states are pairwise distinguishable. Consider two states (q, P) and (q', P'). First, we consider when $P \neq P'$. In this case, reading c takes the state (q, P) to $(m-1, P)$ and (q', P') to $(m-1, P')$. Without loss of generality, there exists an element $t \in P$ such that $t \notin P'$. Then these states are distinguished by the word b^{n-1-t}.

Now, fix $P = P'$ and assume without loss of generality that $q > q'$. First, suppose $q < m-1$. Then we have

$$(q, P) \xrightarrow{b^{m-1-q}} (m-1, Q_n[0]) \xrightarrow{a_2} (m-1, Q_n[2]) \xrightarrow{a_1} (m-1, \{0\}).$$

Recall that we had defined $Q_n[2] = \{0\}$. Now, since $q > q'$, we have $m-1-q+q' < m-1$. Let $q'' = m-1-q+q'$, and we have

$$(q', P') \xrightarrow{b^{m-1-q}} (q'', Q_n[0]) \xrightarrow{a_2} (q'', Q_n[2]) \xrightarrow{a_1} (q'', Q_n[1]).$$

Since $Q_n[1] \neq \{0\}$, the two states are now distinguishable by the prior case.

Now, suppose $q = m-1$. Then we have

$$(q, P) = (m-1, P) \xrightarrow{a_2} (m-1, Q_n[2]) = (m-1, \{0\}) \xrightarrow{a_1} (m-1, \{0\})$$

and

$$(q', P') = (q', P) \xrightarrow{a_2} (q', Q_n[2]) \xrightarrow{a_1} (q', Q_n[1]).$$

Again, since $Q_n[1] \neq \{0\}$, the two states are now distinguishable by the prior case.

Thus, we have shown that all $(m-1)\cdot(k-1)+2^n+1$ states are reachable and pairwise distinguishable. □

These results together give us the following theorem.

Theorem 3.45. *Let A_m be a DFA with m states recognizing the language L_m and let A_n be a DFA with n states recognizing the language L_n, where L_m and L_n are defined over an alphabet Σ of size k, and $m, n \geq 3$. Then the state complexity of $L_m \bowtie L_n$ is $(m-1)\cdot(k-1)+2^n+1$ if $4 \leq k < 2^n$, and $(m-1)\cdot(2^n-1)+2^n+1$ if $k \geq 2^n$, and this bound can be reached in the worst case.*

3.5 Conjugate word blending

As shown in Definition 3.14, word blending allows γ_1 and γ_2 to be different strings. As a step towards a formal model that is closer to the XPCR process experimentally observed in [83], we now require that $\gamma_1 = \gamma_2$, $\gamma_1 \neq \lambda$, and $\gamma_2 \neq \lambda$, and the *conjugate word blending* of two words is defined as follows.

Definition 3.46. *Given two words x and y over an alphabet Σ, the conjugate word blending of x with y is defined as*

$$x \,\underline{\bowtie}\, y = \{\alpha w \beta \mid x = \alpha w \gamma, y = \gamma w \beta, \alpha, \beta \in \Sigma^*, \gamma, w \in \Sigma^+\}.$$

The term "conjugate word blending" alludes to the fact that the common segments of the operands, $w\gamma$ and γw, are conjugate words (a word u is a conjugate of a word v if u can be obtained from v by cyclically shifting its letters [239]). We can extend this word operation to languages in the natural way:

$$L_1 \,\underline{\bowtie}\, L_2 = \bigcup_{x \in L_1, y \in L_2} (x \,\underline{\bowtie}\, y).$$

As an example, if $u = 1010101$ and $v = 10101$, then $u \,\underline{\bowtie}\, v = \{10101\underline{0}101, 101\underline{010}1, 101\underline{0}1\}$ (the underlined subwords are the corresponding overlaps w). If Σ is an alphabet, and $a \in \Sigma$, then $\Sigma^* \,\underline{\bowtie}\, \Sigma^* = \Sigma^+$, $\Sigma^* \,\underline{\bowtie}\, \{aa\} = \Sigma^* a$, and $\{aa\} \,\underline{\bowtie}\, \Sigma^* = a\Sigma^*$. As this example shows, the conjugate word blending operation is not commutative.

As with the original version of word blending in Section 3.4, we can express the conjugate word blending operation as a splicing scheme. The connection between splicing and word blending was shown in Section 3.4, where it was proved that word blending $\bowtie$ can be expressed as one step of the splicing scheme consisting of the rules

$$R_{\bowtie} = \{a\#\lambda\$a\#\lambda \mid a \in \Sigma\}.$$

It is not difficult to see that the conjugate word blending operation cannot be expressed in the same way. Conjugate word blending can be thought of as a single step of a splicing scheme with the following set of rules

$$R_{\underline{\bowtie}} = \{w\#\gamma\$\gamma w\#\lambda \mid w, \gamma \in \Sigma^+\}.$$

However, observe that this splicing scheme is not regular. In fact, it is context sensitive. If we apply a morphism φ that erases $\#$ and $\$$, we get

$$\varphi(R_{\underline{\bowtie}}) = \{w\gamma\gamma w \mid w, \gamma \in \Sigma^+\},$$

which is not context free. This suggests that the closure properties of the conjugate word blending operation may be different from the original version of the word blending operation.

We now investigate the closure properties of the main Chomsky families of languages under conjugate word blending. We first show that we can construct a nondeterministic finite automaton that recognizes the conjugate word blending of two regular languages.

Proposition 3.47. *Given two NFAs A and B, we can effectively construct an NFA C such that $L(C) = L(A) \,\underline{\bowtie}\, L(B)$.*

Proof. Consider states $p \in Q_A$ and $q \in Q_B$ and NFAs $A = (Q_A, \Sigma, s_A, F_A, \delta_A)$ and $B = (Q_B, \Sigma, s_B, F_B, \delta_B)$. We define the following two languages:

$$L(A_p) = \{w \in \Sigma^+ \mid \delta_A(p, w) \cap F_A \neq \emptyset\} \text{ and}$$

$$L({}_qB) = \{w \in \Sigma^+ \mid q \in \delta_B(s_B, w)\}.$$

We can now construct the NFA $C = (Q', \Sigma, s', F', \delta')$ that accepts exactly the language $L(A) \underline{\bowtie} L(B)$, as follows. The sets of states of the NFA C is $Q' = Q_A \cup (Q_A \times Q_B \times Q_B) \cup Q_B$, the initial state is $s' = s_A$, and the set of final states is $F' = F_B$. The transition function $\delta' : Q' \times \Sigma \cup \{\lambda\} \to 2^{Q'}$ is constructed as follows:

1. $\delta'(p, a) = \delta_A(p, a) \cup \{\langle p', q', r' \rangle \mid p' \in \delta_A(p, a), q' \in \delta_B(r', a), r' \in Q_B\}$ for all $p \in Q_A$, and $a \in \Sigma$;

2. $\delta'(\langle p, q, r \rangle, a) = \{\langle p', q', r \rangle \mid p' \in \delta_A(p, a), q' \in \delta_B(q, a)\}$ for all $p \in Q_A$, $q, r \in Q_B$, and $a \in \Sigma$;

3. $\delta'(\langle p, q, r \rangle, \lambda) = \{q\}$ for all $p \in Q_A$ and $q, r \in Q_B$ for which $L(A_p) \cap L({}_rB) \cap \Sigma^+ \neq \emptyset$; and

4. $\delta'(q, a) = \delta_B(q, a)$ for all $q \in Q_B$ and $a \in \Sigma$.

The idea behind this construction is that for a word $\alpha w \beta \in \alpha w \gamma \underline{\bowtie} \gamma w \beta$, the states in Q_A are used for the derivation of α, the states in Q_B are used for the derivation of β, and the states in $Q_A \times Q_B \times Q_B$ are used for the derivation of w, as follows. If the NFA C is in a state from Q_A and reads a letter, it nondeterministically decides the letter is in α or the letter is the first letter of w by transitions of type 1. If the state $\langle p, q, r \rangle$ is reached after a transition of type 1, we assume that the state of B after reading γ is r, and the state of A (respectively B) after reading the first letter of w is p (respectively q). Transitions of type 2 simulate the simultaneous processing, by both A and B, of letters from w. By transitions of type 3, the NFA C checks if the non-empty subword γ exists, and, if it does, it continues with the derivation of β according to transitions of type 4.

Let us now prove that $L(A) \underline{\bowtie} L(B) \subseteq L(C)$. Consider a word $z \in L(A) \underline{\bowtie} L(B)$ with $z = \alpha w \beta$, where $x = \alpha w \gamma \in L(A)$, $y = \gamma w \beta \in L(B)$, and $\gamma \in \Sigma^+$. Now, write $w = aw'$ for $a \in \Sigma$ and $w' \in \Sigma^*$. Since $x \in L(A)$, there must be a path in A:

$$s_A \xrightarrow{\alpha} p_1 \xrightarrow{a} p_2 \xrightarrow{w'} p_3 \xrightarrow{\gamma} p_4, \text{where } p_4 \in F_A.$$

Similarly, since $y \in L(B)$, there must be a path in B:

$$s_B \xrightarrow{\gamma} q_1 \xrightarrow{a} q_2 \xrightarrow{w'} q_3 \xrightarrow{\beta} q_4, \text{where } q_4 \in F_B.$$

From this, we will show that there exists an accepting computation path for z in C:

$$s' = s_A \xrightarrow{\alpha} p_1 \xrightarrow{a} \langle p_2, q_2, q_1 \rangle \xrightarrow{w'} \langle p_3, q_3, q_1 \rangle \xrightarrow{\lambda} q_3 \xrightarrow{\beta} q_4,$$

where $q_4 \in F_B = F'$.

More precisely, we observe that at the beginning of the blending on a, we have that $\langle p_2, q_2, q_1 \rangle \in \delta'(p_1, a)$ since $q_2 \in \delta_B(q_1, a)$. Since $p_4 \in \delta_A(p_3, \gamma)$, $p_4 \in F_A$, $q_1 \in \delta_B(s_B, \gamma)$, and $\gamma \in \Sigma^+$, we have $\gamma \in L(A_{p_3})$ and $\gamma \in L({}_{q_1}B)$, so we have $\gamma \in L(A_{p_3}) \cap L({}_{q_1}B) \cap \Sigma^+$. Therefore, $q_3 \in \delta'(\langle p_3, q_3, q_1 \rangle, \lambda)$. Thus, we have shown that $z \in L(C)$, and consequently $L(A) \bowtie L(B) \subseteq L(C)$.

Now we show that $L(C) \subseteq L(A) \bowtie L(B)$. Let $z \in L(C)$. Then there exists a path on z in C from s_A to a state in F_B. Recall that there are three types of states in C: states of A from Q_A; triples of states $\langle p, q, r \rangle$, where $p \in Q_A$ and $q, r \in Q_B$; and states of B from Q_B. The definition of C implies that every accepting computation of a word w in C must contain all three types of states, in this order. Then we can consider an accepting path for $z = \alpha a w' \beta$, where $\alpha, \beta, w' \in \Sigma^*$ and $a \in \Sigma$, by

$$s' = s_A \xrightarrow{\alpha} p_1 \xrightarrow{a} \langle p_2, q_2, q_1 \rangle \xrightarrow{w'} \langle p_3, q_3, q_1 \rangle \xrightarrow{\lambda} q_3 \xrightarrow{\beta} q_4 \in F_B = F'.$$

In this path, p_1 is the last state of A that occurs, $\langle p_2, q_2, q_1 \rangle$ is the first triple that occurs, $\langle p_3, q_3, q_1 \rangle$ is the final triple that occurs, q_3 is the first state of B that occurs, and q_4 is an accepting state of C, which by definition is an accepting state of B.

From the definition of the transition function, it is clear that the words α, αa, and $\alpha a w'$ are all prefixes of a word in $L(A)$ and the words β and $w'\beta$ are all suffixes of a word in $L(B)$.

The final observation is that we need to consider transitions from $\langle p_3, q_3, q_1 \rangle$ to q_3 on the empty word λ. Such a transition can only occur if there exists a non-empty word $\gamma \in L(A_{p_3}) \cap L({}_{q_1}B) \cap \Sigma^+$. From this, we have $\alpha a w' \gamma \in L(A)$ since $p_3 \in \delta_A(s_A, \alpha a w')$ and $\delta_A(p_3, \gamma) \cap F_A \neq \emptyset$ by definition of $L(A_{p_3})$. We also have $\gamma a w' \beta \in L(B)$ since by definition of $L({}_{q_1}B)$, we have $q_1 \in \delta_B(s_B, \gamma)$. By the definition of δ', we have $\langle p_2, q_2, q_1 \rangle \in \delta'(p_1, a)$ if $q_2 \in \delta_B(q_1, a)$. Therefore, there is a path in B:

$$s_B \xrightarrow{\gamma} q_1 \xrightarrow{a} q_2 \xrightarrow{w'} q_3 \xrightarrow{\beta} q_4 \in F_B.$$

Taking $w = aw'$, we can now write $z = \alpha w\beta \in L(C)$, with $x = \alpha w\gamma \in L(A)$ and $y = \gamma w\beta \in L(B)$. By the definition of conjugate word blending, this implies $z \in x \underline{\bowtie} y$ and thus $z \in L(A) \underline{\bowtie} L(B)$.

Therefore, $L(C) \subseteq L(A) \underline{\bowtie} L(B)$, and we can conclude that $L(C) = L(A) \underline{\bowtie} L(B)$. $\square$

Corollary 3.48. *The class of regular languages is closed under conjugate word blending.*

Next, we will show that unlike word blending, the family of context-free languages is not closed under conjugate word blending.

Proposition 3.49. *The class of context-free languages is not closed under conjugate word blending.*

Proof. This can be proved by a counterexample. Consider two context-free languages $L_1 = \{a^n\$b^n\$\# \mid n \in \mathbb{N}\}$ and $L_2 = \{\#\$b^m\$a^m \mid m \in \mathbb{N}\}$, we have that $(L_1 \underline{\bowtie} L_2) \cap a^*\$b^*\$a^* = \{a^n\$b^n\$a^n \mid n \in \mathbb{N}\}$, which is not context free. Thus, since the class of context-free languages is closed under intersection with regular languages, it is not closed under conjugate word blending. $\square$

Proposition 3.50. *The class of context-sensitive languages is not closed under conjugate word blending.*

Proof. Assume that the class of context-sensitive languages is closed under conjugate word blending. Let L_0 be a recursively enumerable but not context-sensitive language over an alphabet Σ and let $a, b \notin \Sigma$. Then, there is a context-sensitive language L such that L consists of words of the form wba^i, where $i \geq 0$ and $w \in L_0$, and for every word w in L_0 there is an $i \geq 0$ such that $wba^i \in L$. We have that $(La \underline{\bowtie} a^+b) \cap \Sigma^*b = L_0b$. If the class of context-sensitive languages were closed under conjugate word blending, then L_0b would be context sensitive, which is a contradiction. $\square$

We will now show that the class of recursively enumerable languages is closed under conjugate word blending. Recall that *sequential deletion* was defined in [139] as the binary language operation $L_1 \to L_2 = \bigcup_{u \in L_1, v \in L_2} (u \to v)$, where $u \to v = \{w \in \Sigma^* \mid u = w_1vw_2, w = w_1w_2,\ w_1, w_2 \in \Sigma\}$. We begin with the following lemma.

Lemma 3.51. *Consider two languages L_1, L_2 over an alphabet Σ, two symbols $\#, \$ \notin \Sigma$, and a homomorphism $h(a) = a$, for $a \in \Sigma$, and $h(\#) = h(\$) = \lambda$. Conjugate word blending can be expressed as*

$$L_1 \underline{\bowtie} L_2 = (L \cap L') \to (\#\Sigma^+\$\Sigma^+\$),$$

where
$$L = (h^{-1}(L_1) \cap (\Sigma^*\Sigma^+\#\Sigma^+\$)) \,\overline{\odot}\, \left(h^{-1}(L_2) \cap \left(\#\Sigma^+\$\Sigma^+\$\Sigma^*\right)\right),$$
$\rightarrow$ *is the sequential deletion operation,* $\overline{\odot}$ *is the overlap assembly operation, and* $L' = \bigcup_{w\in\Sigma^*} \Sigma^* w\#\Sigma^+\$w\$\Sigma^*$*.*

Proof. Consider a word $z \in L_1 \bowtie L_2$, where there exists a decomposition $z = \alpha w\beta$ where $x = \alpha w\gamma \in L_1$, $y = \gamma w\beta \in L_2$, and $\gamma \in \Sigma^+$. We have that

$$\begin{aligned} z &= \alpha w\beta \\ &\in \alpha w\#\gamma\$w\$\beta \rightarrow (\#\Sigma^+\$\Sigma^+\$) \\ &= (\alpha w\#\gamma\$w\$\beta \cap L') \rightarrow (\#\Sigma^+\$\Sigma^+\$) \\ &\subseteq (L \cap L') \rightarrow (\#\Sigma^+\$\Sigma^+\$). \end{aligned}$$

Next, consider a word $z \in (L \cap L') \rightarrow (\#\Sigma^+\$\Sigma^+\$)$. There exist words $\alpha, \beta \in \Sigma^*$ and $w, \gamma \in \Sigma^+$ such that $z = \alpha w\beta$ and $z' = \alpha w\#\gamma\$w\$\beta \in (L \cap L') \subseteq L$. Thus, there exist words $x' = \alpha w\#\gamma\$ \in (h^{-1}(L_1) \cap (\Sigma^*\Sigma^+\#\Sigma^+\$))$ and $y' = \#\gamma\$w\$\beta \in (h^{-1}(L_2) \cap (\#\Sigma^+\$\Sigma^+\$\Sigma^*))$ such that $z' \in x' \,\overline{\odot}\, y'$, $x = \alpha w\gamma \in L_1$, and $y = \gamma w\beta \in L_2$. Thus, we have that $z \in L_1 \bowtie L_2$. □

Proposition 3.52. *The class of recursively enumerable languages is closed under conjugate word blending.*

Proof. This follows from Lemma 3.51, since the class of recursively enumerable languages is closed under overlap assembly [73], inverse homomorphism, intersection [243], and sequential deletion [139]. □

In summary, the results of this section show that the classes of regular and recursively enumerable languages are closed under conjugate word blending, while the classes of context-free and context-sensitive languages are not. As conjectured earlier, these closure properties are different from those of word blending, the difference being that the class of context-free languages is closed under word blending but not under conjugate word blending.

3.6 DNA implementation of conjugate word blending

In this section, we describe the wet lab experiments that motivated and implemented the conjugate word blending operation. Section 3.6.1 introduces some basic notions of molecular biology. Section 3.6.2 outlines the initial experimental evidence that led to the definition of conjugate word blending operation. Section 3.6.3 reports the experiments that confirmed and validated the XPCR-based implementation of the conjugate word blending. Note that some preliminary work for these experiments was developed in [14, 83] with the aim of generating a DNA library of operons (i.e., permutations of genes) able to optimize the PAH degradation work of Burkholderia fungorum DBT1.

3.6.1 Molecular biology preliminaries

In the remainder of this chapter, we will denote the union of the sets U and V by $U + V$. Often we will use as synonyms the terms strand, word, string, fragment, and molecule. The term *amplicon* is used to denote a fragment of DNA that is the product of molecular amplification (i.e., replication).

XPCR is a PCR-based protocol that realizes what in the context of splicing systems is called a null-context splicing rule [95], which is a particular splicing rule $u_1\#u_2\$u_3\#u_4$ having $u_2u_4 = \lambda$ and $u_1 = u_3$. In its general form, XPCR takes as input sequences $\alpha X_1\gamma Y_2\beta + \alpha Y_1\gamma X_2\beta$, where X_1, Y_1, Y_2, and X_2 are genes, and α, β, and γ are primer sequences, and produces as an output the *chimeric* sequences (a chimeric sequence is a sequence formed from the prefix of one sequence and the suffix of another sequence joined together) $\alpha X_1\gamma X_2\beta$ and $\alpha Y_1\gamma Y_2\beta$—this corresponds to the application of a null-context splicing rule with $u_1 = u_3 = \gamma$. The essential feature of this process (e.g., the recombination between $\alpha X_1\gamma$ and $\gamma X_2\beta$ that produces $\alpha X_1\gamma X_2\beta$) can also be formalized as the overlap assembly operation between two strings xy and yz, resulting in the string xyz. Figure 3.2 illustrates the overlap assembly between xy and yz, where $x = \alpha X$, $y = \gamma$, and $z = Y\beta$. If $X = \mathrm{A}$ and $Y = \mathrm{D}$ are the genes introduced in Section 3.3, then the expected length of the chimeric amplicon $\alpha\mathrm{A}\gamma\mathrm{D}\beta$ is 1,600 bp, due to the primer and gene length ($|A| = 1{,}019$ bp, $|D| = 518$ bp, and $|\alpha| = |\beta| = |\gamma| = 21$ bp).

All experiments of DNA strand amplification were performed in double sampling (that is, on two test tubes in parallel), with negative controls (test tubes with the same contents, except with no DNA templates), under different experimental conditions (including temperature, concentration, gene, and length variations), and repeated with two different polymerase enzymes, *Taq* polymerase and *Pfu* polymerase. To ensure higher duplication fidelity, Pfu DNA polymerase was chosen over the routinely used Taq DNA polymerase for initial reactions (gene extraction from the original genome) due to its proofreading capabilities and thermal resistance.

3.6.2 The initial experimental evidence

Concatenation of two different genes by XPCR was successfully implemented, even under interference, as illustrated in Figure 3.6, where a third input template γBγ was added (to favour the formation of additional longer molecules αAγBγDβ), apt to perturb the expected two-genes amplification. This was a way to prove the stability and robustness of XPCR, namely, its reliability under perturbation.

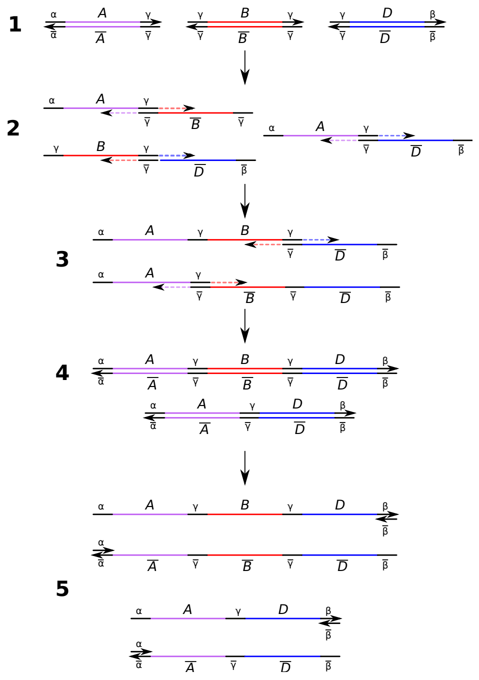


Figure 3.6: XPCR-based two-gene concatenation (genes A and D), from input templates (αAγ, γBγ, and γDβ) and primers (α and $\overline{\beta}$) [83]. Row 1: Input templates. Row 2: Partially double-stranded DNA molecules that are two-gene recombinants generated by hybridization of DNA single strands from the input templates. Row 3: Partially double-stranded DNA molecules that are three-gene recombinants generated by the hybridization of DNA single strands from the input templates and the DNA double strands that were created from the recombinants in Row 2 after extension by polymerase. Only sequences that were exponentially amplified are illustrated. Row 4: Two DNA double strands that were exponentially amplified are αAγBγDβ and αAγBβ. The amplification of the longer formation αAγBγDβ was produced in an insignificant quantity, as illustrated in Figure 3.7. Row 5: These two DNA double strands were amplified using primers (α and $\overline{\beta}$), like in PCR.

In fact, also in some experiments reported in this section, an *interference molecule* $\gamma X \gamma$ was added (with $X \in \{\text{A,B,D}\}$), at higher concentrations than the other input molecules, to see whether it would interfere with the amplification of molecules containing γ as a prefix or suffix by forming longer concatenations (of three genes, as in Figure 3.6).

XPCR did not behave as expected when attempts were made to concatenate copies of the same gene using the method illustrated in Figure 3.6. In [14], several experiments were carried out with the aim of concatenating two (or more) copies of the same gene, using primers (α and $\overline{\beta}$) and templates $\alpha X \gamma$ and $\gamma X \beta$ (or templates $\alpha X \gamma$, $\gamma X \gamma$, and $\gamma X \beta$), where X is a gene. The output of these experiments was, unexpectedly, $\alpha X \beta$ rather than $\alpha X \gamma X \beta$ (or $\alpha X \gamma X \gamma X \beta$, respectively). These results were observed in presence of the interference molecules $\gamma X \gamma$ at different concentration ratios.

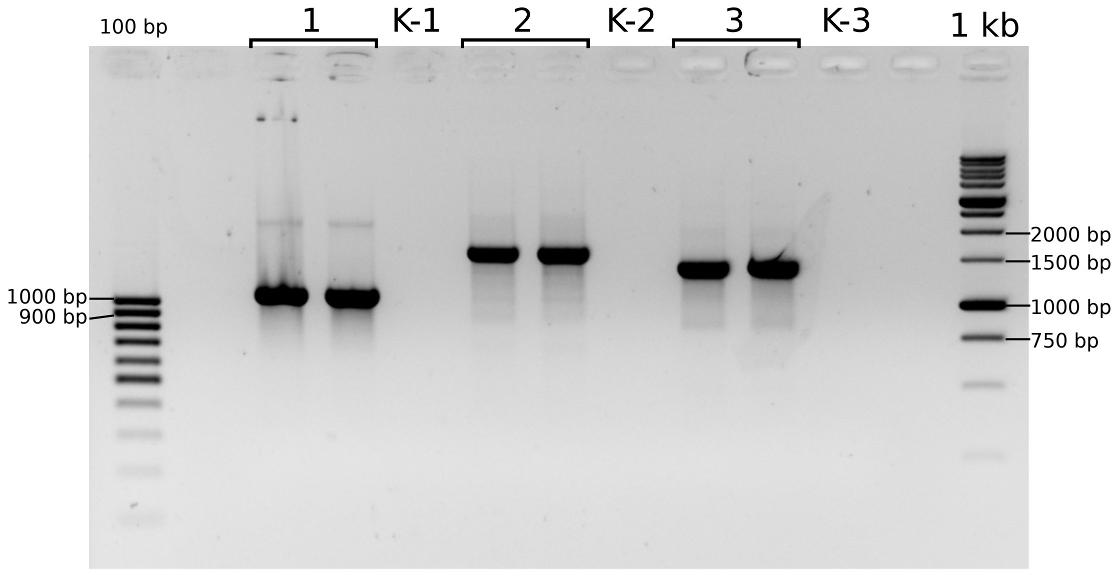


Figure 3.7: XPCR with templates containing the same gene and different genes, respectively [14]. Amplifications with primers (α and $\overline{\beta}$) and Taq polymerase. Lane 1: XPCR with templates $\alpha\text{A}\gamma$, $\gamma\text{A}\gamma$, and $\gamma\text{A}\beta$ exhibited a main product of about 1,000 bp ($\alpha\text{A}\beta$, a dark band) and a secondary product of about 2,000 bp ($\alpha\text{A}\gamma\text{A}\beta$, a faint grey band, in the same lane). Lane 2: XPCR with templates $\alpha\text{A}\gamma$, $\gamma\text{B}\gamma$, and $\gamma\text{D}\beta$ exhibited an amplification product of about 1,600 bp, which corresponded to the output $\alpha\text{A}\gamma\text{D}\beta$. Lane 3: XPCR with templates $\alpha\text{B}\gamma$, $\gamma\text{D}\gamma$, and $\gamma\text{A}\beta$ with an output amplification product of about 1,400 bp, corresponding to $\alpha\text{B}\gamma\text{A}\beta$. Lanes K-1, K-2, and K-3: Negative controls without templates for the reactions in Lanes 1, 2, and 3, respectively.

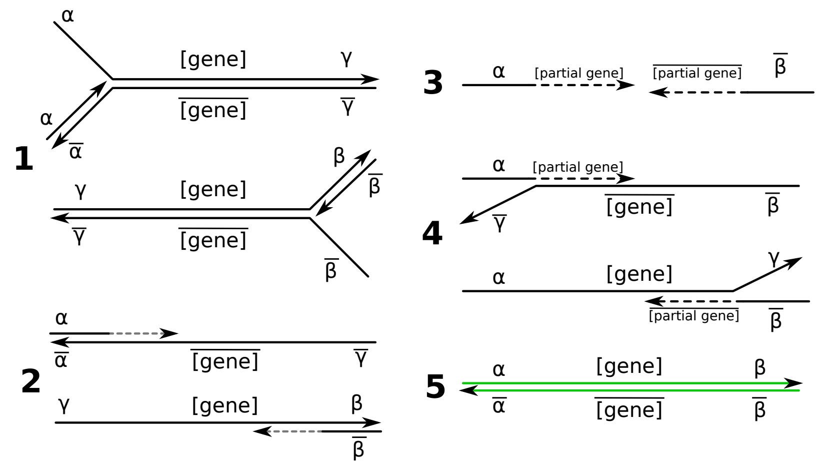


Figure 3.8: A possible explanation for the formation of the conjugate blending operation output. (The implicit assumption is that there always exists one template out of millions for which the described premature detachment occurs, and that this is enough to generate an exponential amplification of $\alpha X \beta$, with gene X and primers α and $\overline{\beta}$, in the next PCR cycles.) Subgraph 1: Both primers anneal. Subgraph 2: Primer polymerase extension occurs along single templates. Over long segments X, this process takes a long time and may be interrupted by the high denaturation temperature expected in next step of PCR. This causes a premature detachment of the polymerase enzyme. Subgraph 3: Incomplete template copies generated. Subgraph 4: In the next PCR cycle, the resulting incomplete strands may anneal to each other and also to the other template and then generate (by polymerase extension) single strands $\alpha X \beta$ and $\overline{\alpha X \beta}$. Subgraph 5: These generated single strands will work as templates, where they will be exponentially amplified due to primer annealing. The single strands containing γ or $\overline{\gamma}$ anneal with neither of the primers and are not amplified.

As exemplification of these phenomena, in Figure 3.7, we report experimental results that exhibit as outputs both the concatenation of two different genes described in Figure 3.6 (in the presence of a long interference molecule containing a different gene) and the unexpected amplicon $\alpha\text{A}\beta$, when two copies of the same gene A were present in the templates. More precisely, amplification of an input composed of three different templates, $\alpha\text{A}\gamma$, $\gamma\text{B}\gamma$, and $\gamma\text{D}\beta$ ($\alpha\text{B}\gamma$, $\gamma\text{D}\gamma$, and $\gamma\text{A}\beta$, respectively) produced as an output $\alpha\text{A}\gamma\text{D}\beta$ ($\alpha\text{B}\gamma\text{A}\beta$, respectively)), as seen in Lane 2 (Lane 3, respectively) of Figure 3.7. On the other hand, amplification of an input composed of three different templates all containing the gene A, that is, $\alpha\text{A}\gamma$, $\gamma\text{A}\gamma$, and $\gamma\text{A}\beta$, produced as an output only the sequence $\alpha\text{A}\beta$, as seen in Lane 1 of Figure 3.7. In all cases, we amplified three different templates, present in equal concentrations, by PCR reactions under identical experimental conditions, with basic Taq polymerase.

These results provided experimental evidence of a limitation of the XPCR protocol, which indeed cannot be used to concatenate multiple occurrences of the same gene in a significant quantity [83].

In the case of multiple occurrences of the same gene, the unexpected outcome of XPCR might be due to phenomena similar to those observed in [111], which altered the normal amplification of DNA strands sharing long fragments. In particular, when we perform PCR with primers (α and $\overline{\beta}$) on templates with the same gene X, such as $\alpha X\gamma$ and $\gamma X\beta$, it results in the biased production of the shortest amplicon $\alpha X\beta$, as depicted in Figure 3.8, up to the point where longer fragments are not detectable. On the electrophoresis gel, this leads to faint or indistinguishable bands for the longer products and a strong signal for the short product. In other words, once the shortest sequence $\alpha X\beta$ has been formed, it is amplified faster than the longer strand $\alpha X\gamma X\beta$, probably due to the higher annealing efficiency of primers on shorter sequences.

3.6.3 Conjugate word blending: Experimental results

In this section, we report the details of additional wet lab DNA experiments that motivated and validated the notion of conjugate word blending explored in this paper. Below is a summary of all PCR experiments that demonstrate the conjugate word blending operation in action. The primers used are α and $\overline{\beta}$, and these experiments confirmed the amplified production of $\alpha X\beta$ sequences, as detailed below. Note that, based on the length of primers α, γ, and β (21 bp) and genes A (1,019 bp), B (311 bp), and D (518 bp), the expected lengths of the amplicons $\alpha X\beta$ are 1,061 bp (for $X = \text{A}$), 353 bp (for $X = \text{B}$), and 560 bp (for $X = \text{D}$).

1. XPCR with two different templates containing gene D, namely $\alpha\text{D}\gamma$ and $\gamma\text{D}\beta$, output $\alpha\text{D}\beta$. The reaction, performed with Pfu polymerase, has the output amplicon reported in Lane 1 on the left panel of Figure 3.9.

2. The main output of XPCR with three different templates containing gene D, namely, $\alpha\text{D}\gamma$, $\gamma\text{D}\gamma$, and $\gamma\text{D}\beta$, was $\alpha\text{D}\beta$. The DNA double strand $\gamma\text{D}\gamma$ was the interference molecule. The template concentration ratio (of $\alpha\text{D}\gamma$, $\gamma\text{D}\gamma$, and $\gamma\text{D}\beta$) was 1: 10: 1

to favour the amplification of longer amplicons. Reactions were carried out with Pfu polymerase and five different annealing temperatures that corresponds to the amplicons visible in Lanes 1.1, 1.2, 1.3, 1.4, and 1.5 on the right panel of Figure 3.10.

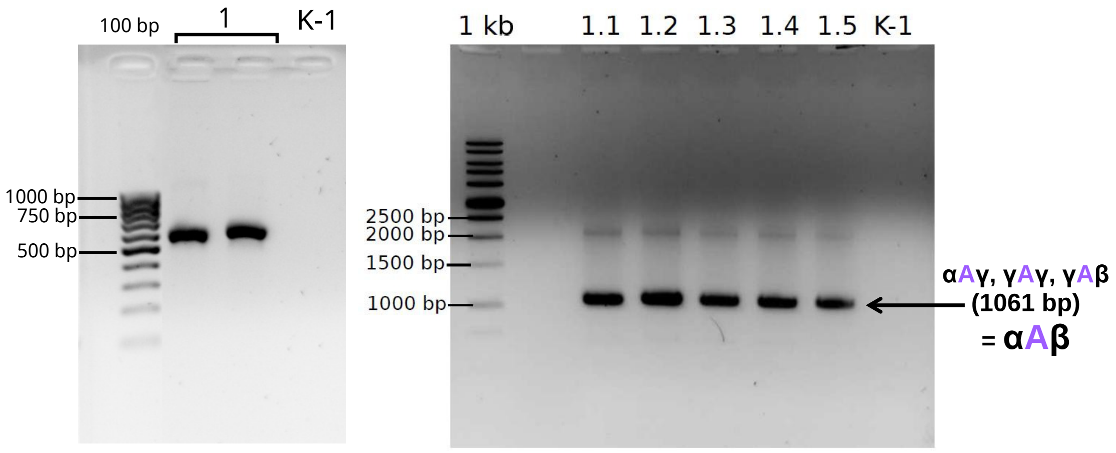


Figure 3.9: XPCR with templates containing gene D with no interference molecules (left panel) and XPCR with templates containing gene A with a high concentration of interference molecules (right panel) [83]. These reactions were performed with with primers (α and $\overline{\beta}$) and Pfu polymerase. Left panel, Lane 1: XPCR with templates αDγ and γDβ exhibited a product of about 500 bp (αDβ). Left panel, Lane K-1: Negative control without templates for the reaction in Lane 1. Right panel, Lanes 1.1, 1.2, 1.3, 1,4, and 1.5: XPCR with templates αAγ, γAγ, and γAβ (with concentration ratio 1: 10: 1) and different annealing temperatures exhibited a main product of about 1,000 bp (αAβ) and a secondary product of about 2,000 bp (αAγAβ). Right panel, Lane K-1: Negative control without templates for the reactions in Lanes 1.1, 1.2, 1.3, 1,4, and 1.5.

3. The output of XPCR with three different templates containing gene B, namely αBγ, γBγ, and γBβ with the concentration ratio 1: 2: 1, was αBβ. Reactions were carried out with Taq polymerase, and corresponding products are visible in Lane 1.1 in the left panel of Figure 3.10.

4. The output of XPCR with three different templates containing gene A, namely αAγ, γAγ, and γAβ, was αAβ. Reactions were carried out with Pfu polymerase and with different concentrations for the interference molecule γAγ with respect to the other two templates αAγ and γAβ. The experimental results of XPCR with template concentration ratios (of αAγ, γAγ, and γAβ) of 1: 2: 1 and 1: 5: 1 are shown in

Figure 3.11, and those of 1: 10: 1 are shown in Figure 3.9. Each of these three experiments, corresponding to the different concentration ratios, was performed at five different annealing temperatures.

We now describe in more detail a typical reaction, such as those shown in Figure 3.9, where the templates are αDγ and γDβ (or αAγ, γAγ, and γAβ, with concentration ratios 1: 10: 1, respectively). The left panel of Figure 3.9 exhibits the outcome of XPCR with templates αDγ and γDβ in the form of a band that confirms the presence of a product of about 500 bp. Sequencing showed that this product was indeed the amplicon αDβ. On the right panel of Figure 3.9, in all five reactions with different temperatures, the main products of about 1,000 bp (αAβ) are evident as the result of an XPCR over templates αAγ, γAγ, and γAβ. A faint band of about 2,000 bp is visible as well, possibly containing expected concatenations αAγAβ (2,101 bp), while concatenations αAγAγAβ (3,141 bp) were not formed in observable quantities.

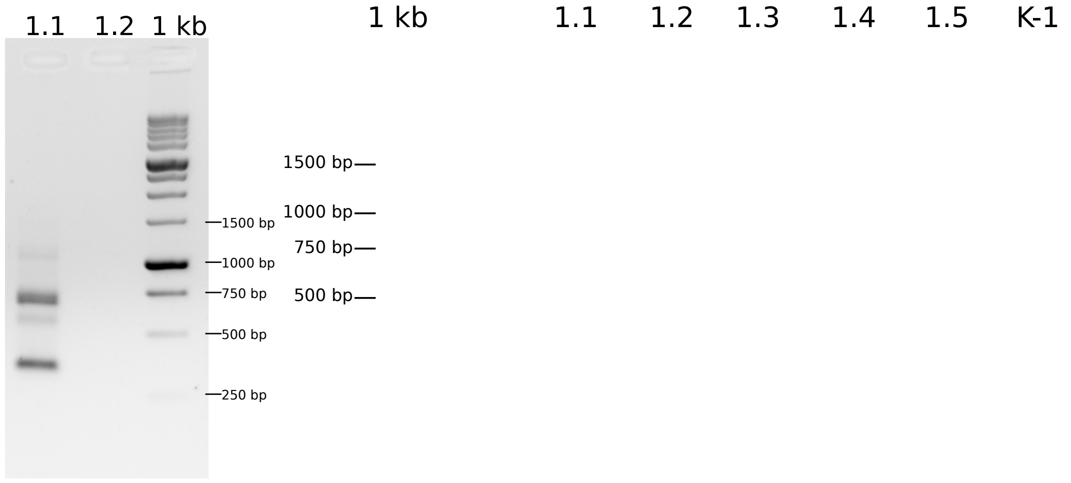


Figure 3.10: XPCR with templates containing gene B and XPCR with templates containing gene D [14]. Left panel, Lane 1.1: XPCR was performed using Taq polymerase and templates αBγ, γBγ, and γBβ (with concentration ratio 1: 2: 1). It exhibited a main product of about 300 bp (αBβ) and a secondary product of about 650 bp (αBγBβ). Left panel, Lane 1.2: Negative control without templates for the reaction in Lane 1.1. Right panel, Lanes 1.1, 1.2, 1.3, 1,4, and 1.5: XPCR was performed with Pfu polymerase, templates αDγ, γDγ, and γDβ (with concentration ratio 1: 10: 1), and different annealing temperatures. A main product of about 500 bp (αDβ) and a secondary product of about 1,100 bp (αDγDβ) were exhibited. Right panel, Lanes K-1: Negative control without templates for the reactions in Lanes 1.1, 1.2, 1.3, 1.4, and 1.5.

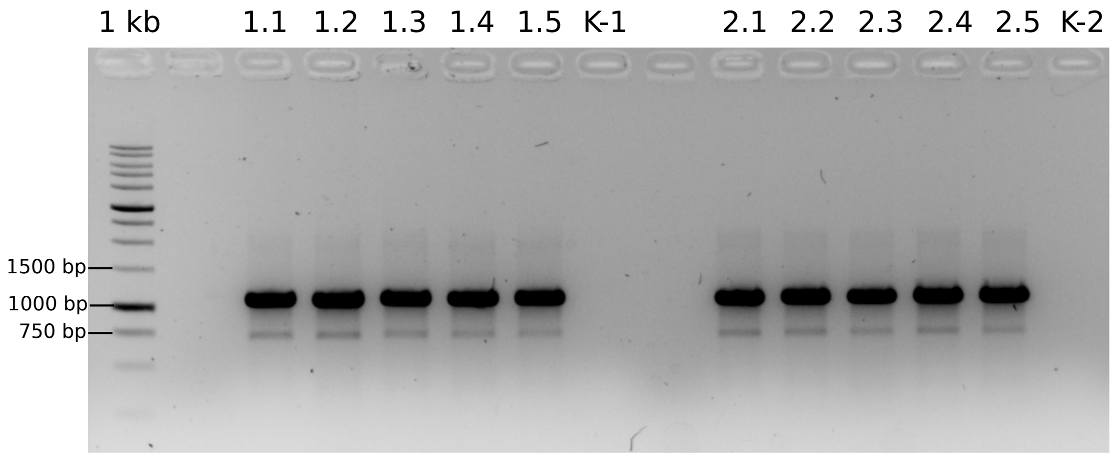


Figure 3.11: XPCR with templates containing gene A at two different ratios of molecular interference [14]. These reactions were performed with Pfu polymerase and templates $\alpha A\gamma$, $\gamma A\gamma$, and $\gamma A\beta$. Lanes 1.1, 1.2, 1.3, 1.4, and 1.5: The molecular concentration ratio was $1:2:1$. A different annealing temperature was used for every lane. Lanes 2.1, 2.2, 2.3, 2.4, and 2.5: The molecular concentration ratio was $1:5:1$. A different annealing temperature was used for every lane. All aforementioned lanes exhibited a main product of about 1,061 bp (corresponding to $\alpha A\beta$) and other very faint bands of biased products. Lanes K-1 and K-2: Negative controls without templates for the reactions in all aforementioned lanes.

The experiments illustrated in Figure 3.9 and Figure 3.11 demonstrate that XPCR with templates containing the gene A is robust, as the same outcome was obtained under different annealing temperatures and different interference molecule concentration ratios. Similar experiments were repeated with genes B and D with different DNA polymerases, different annealing temperatures, and different interference molecule concentration ratios, as illustrated in Figure 3.9 and Figure 3.10. This suggests that this process has no restrictions on the length of the gene.

3.7 Conclusion

In Section 3.1, we reviewed some related aspects of formal language studies. Section 3.2 listed some biologically inspired word operations. Section 3.3 discussed XPCR details and the previous defined word operation, called overlap assembly, inspired by it.

In Section 3.4, we first defined a word blending operation motivated by an experimentally observed outcome of XPCR on DNA strings. We studied closure properties, decision problems, and the state complexity for this operation, and we showed that given an alphabet, there are finitely many languages that can be the result from the iterated word blending applied to an initial language.

Section 3.5 introduced and studied conjugate word blending, a binary string operation that models the unexpected outcome of the wet lab XPCR procedure under a specific set up, namely when used to attempt concatenating two copies of the same gene. We investigated computational properties of this operation and proved that the classes of regular and recursively enumerable languages are closed under conjugate word blending, while the classes of context-free and context-sensitive languages are not.

In Section 3.6, we reviewed the wet lab experiments that the conjugate word blending operation is modelled upon, with three bacterial genes of different lengths, and verify its outcome under several experimental conditions, such as using different DNA polymerase enzymes (Taq and Pfu), different primer annealing temperatures, and in the presence of a so-called interference molecule, at various concentration ratios to the template molecules. In [83], it was hypothesized that the unexpected behaviour of XPCR under these specific conditions is caused by strand displacement (template switching and/or hydrolysis of competing strands [136]). Another explanation was proposed for this phenomenon, as illustrated in Figure 3.8. Further experimental work is needed to validate this explanation and the exact mechanisms of the observed molecular biology phenomenon on which the conjugate word blending is based.

As future work, we will develop theoretical investigations of its variants to more closely model the experimental reality of DNA string computation implemented by the XPCR wet lab procedure. Another direction could be to consider the complexity of the decision problems studied in Section 3.4.2 to find the exact number of different iterated blending languages in Corollary 3.41 and to characterize the class of languages generated by iterated blending. Finally, the question of determining the state complexity of iterated word blending remains open. Possible approaches could potentially make use of the connections between iterated blending and simple splicing systems.

Chapter 4

Involutive Fibonacci Words

The sequence of Fibonacci strings is an infinite sequence of strings obtained from two initial letters, $f_1 = a$ and $f_2 = b$, by the recursive definition $f_{n+2} = f_{n+1}f_n$ for all $n \geq 1$ [159]. We first propose a unified terminology to identify the different types of Fibonacci words in the extensive literature on the topic. Motivated by ideas stemming from theoretical studies of DNA computing, we then define and explore involutive Fibonacci words (ϕ-Fibonacci words and indexed ϕ-Fibonacci words, where ϕ denotes either a morphic or an antimorphic involution). As mentioned in Section 2.2, information-encoding DNA single strands can form intramolecular structures, such as hairpins, that are undesirable for some DNA computing experiments. Therefore, it is useful to find a simple and recursive method to design DNA single strands that can avoid those undesired intramolecular structures. We study borderedness and primitivity of involutive Fibonacci words in our search for a method that generates arbitrarily long DNA words that do not have undesired self-binding properties. We also study other properties of involutive Fibonacci words, such as structures and relationships.

In Section 4.1, a unified terminology is proposed, and different types of Fibonacci words are grouped according to this definition. Section 4.2 gives some preliminaries used in the following sections. In Section 4.3, we define and study several generalizations of Fibonacci words, called involutive Fibonacci words, inspired by DNA Watson-Crick complementarity. Section 4.4 proposes a generalization of indexed Fibonacci words to indexed ϕ-Fibonacci words and discusses their interrelationships. Section 4.5 explores borders and ϕ-borders of involutive Fibonacci words. In Section 4.6 and Section 4.7, the primitivity of various involutive Fibonacci words with singleton letters as the initial words is studied. Section 4.8 summarizes the results and discusses future work.

Section 4.1, Section 4.3, Section 4.4, and parts of Section 4.2 are adapted from a paper I co-authored [145] titled "Involutive Fibonacci words."

Section 4.6, Section 4.7, and parts of Section 4.2 are adapted from a paper I co-authored [149] titled "Primitivity of atom Watson-Crick Fibonacci words."

4.1 Introduction

Fibonacci words or Fibonacci strings were introduced as word counterparts of the Fibonacci numbers defined by $F_0 = 0$, $F_1 = 1$, and the recursion $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. "Fibonacci strings" were first defined by Knuth in his "The Art of Computer Programming" (volume 1, section 1.2.8, exercise 36, [159]) as being an infinite sequence of strings obtained from two initial letters, $f_1 = a$ and $f_2 = b$, by the recursive definition $f_{n+2} = f_{n+1}f_n$ for all $n \geq 1$. Various other definitions of Fibonacci words have been proposed since, as detailed in the sequel. In this section, we propose a unified terminology for the purpose of clarifying and comparing the multiple variants of the definition of Fibonacci words that exist in the literature.

The following definition proposes a uniform and intuitive terminology for the various types of Fibonacci words studied in the literature.

Definition 4.1. *Let Σ be an alphabet with $|\Sigma| \geq 2$ and let $u, v \in \Sigma^+$. The* n^{th} standard Fibonacci words *are defined recursively as:*

$$f_1(u,v) = u, f_2(u,v) = v, \text{ and}$$

$$f_n(u,v) = f_{n-1}(u,v)f_{n-2}(u,v) \text{ for } n \geq 3.$$

The sequence of standard Fibonacci words *is defined as $F(u,v) = \{f_n(u,v)\}_{n\geq 1}$, that is, $F(u,v) = \{u, v, vu, vuv, vuvvu, vuvvuvuv, vuvvuvuvvuvvu, \ldots\}$.*

Similarly, the n^{th} reverse Fibonacci words *are defined recursively as:*

$$f'_1(u,v) = u, f'_2(u,v) = v, \text{ and}$$

$$f'_n(u,v) = f'_{n-2}(u,v)f'_{n-1}(u,v) \text{ for } n \geq 3.$$

The sequence of reverse Fibonacci words *is defined as $F'(u,v) = \{f'_n(u,v)\}_{n\geq 1}$, that is, $F'(u,v) = \{u, v, uv, vuv, uvvuv, vuvuvvuv, uvvuvvuvuvvuv, \ldots\}$.*

If the initial words u and v are singleton letters, the resulting words will be called atom standard Fibonacci words *and* atom reverse Fibonacci words*, respectively.*

Note that the length of the n^{th} atom standard/reverse Fibonacci word f_n is in fact the Fibonacci number F_n for $n \geq 1$.

Below is a summary—not necessarily exhaustive—of papers in the literature, grouped by the types of Fibonacci words they study, according to Definition 4.1:

- Atom standard Fibonacci words: [19, 24, 50, 51, 60, 61, 68, 78, 79, 81, 158–160, 188, 192, 198, 208, 219, 220, 240, 241, 247, 248, 251, 256, 274];
- Atom reverse Fibonacci words: [51, 78, 104, 138, 248, 267, 268, 281];
- (Non-atomic) standard Fibonacci words: [49, 78, 281]; and
- (Non-atomic) reverse Fibonacci words: [49, 78, 267, 268, 281].

Indeed, according to Definition 4.1, the strings defined in [159] are atom standard Fibonacci words. They were extensively studied in such examples as [19, 60, 61, 160, 240], with some slight modifications of either changing the initial letters or slightly changing the indices. For examples, see [19,61,240,241]. Some properties involving the structure of such atom standard Fibonacci words were studied in [68, 81].

It was noted in [78] that every standard (reverse) Fibonacci sequence $F(u, v)$ ($F'(u, v)$, respectively) is a homomorphic image of the atom standard (atom reverse, respectively) Fibonacci sequence $F(a, b)$ ($F'(a, b)$, respectively) via the homomorphism $h(a) = u$ and $h(b) = v$, where $a \neq b$. Thus, properties of atom Fibonacci words are especially important.

In the remainder of this chapter, if the first two Fibonacci words are obvious from the context, the argument (u, v) will be omitted, and we will write the n^{th} standard Fibonacci word as f_n, the n^{th} reverse Fibonacci word as f'_n, the standard Fibonacci sequence as F, and the reverse Fibonacci sequence as F'.

An equivalent definition of the sequence of atom standard Fibonacci words, using the iteration of a morphism, was given in such studies as [19, 188], where a morphism $\nu : \Sigma^* \to \Sigma^*$ is defined by $\nu(b) = ba$, $\nu(a) = b$, $f_1 = a$, and $f_{n+1} = \nu^n(a)$ for all $n \geq 1$, which determines the sequence $a, b, ba, bab, babba, babbabab, babbababbabba, \ldots$. Properties of atom standard Fibonacci words and sequences generated by iterating the morphism ν were studied in [24,188,208,219,220,247,274]. It was shown in [219] that the length of the word f_n, defined by such a morphism, is the n^{th} Fibonacci number. In [251], it was shown that the infinite atom standard Fibonacci sequence is an automatic sequence (a sequence computed by a deterministic finite automaton with output), and Mousavi, Schaeffer, and Shallit [192] used the software Walnut to study properties of Fibonacci words using such automata.

Yet another alternative definition of the sequence of atom standard Fibonacci words is based on the "golden mean" $\Phi = (1+\sqrt{5})/2$ [256], whereby $f_n = c_1c_2\cdots c_n$, with $c_i = a$ if $i \in \{\lfloor k\Phi \rfloor \mid k \geq 1\}$, and $c_i = b$ otherwise, for all $1 \leq i \leq n$.

The atom reverse Fibonacci words of Definition 4.1 were defined in [138] using an iterative morphism and discussed by Higgins [104] under the name *papal sequence*. Their properties were further studied in [248].

A generalization of atom standard and reverse Fibonacci words to the standard and reverse Fibonacci words of Definition 4.1 (wherein the first two words are non-atomic) was discussed in [49, 78, 267, 268, 281].

Another generalization was introduced in [49], which defined what we herein call *indexed Fibonacci words*. Under this definition, every indexed Fibonacci word is associated with a binary sequence whose last digit is 0 if the word was obtained by the standard concatenation order of the previous two words, and 1 if it was obtained by the reverse concatenation order. Properties of indexed Fibonacci words were studied in [50, 51].

In this chapter, we propose several generalizations of standard, reverse, and indexed Fibonacci words, motivated by an idea first advanced and studied in the context of DNA computing [142, 212], whereby the Watson-Crick DNA complementarity is formalized as an antimorphic involution function θ on Δ^*, where Δ is the DNA alphabet defined as $\Delta = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$. Indeed, a DNA strand can be viewed as a word over Δ, wherein A is Watson-Crick complementary to T and C to G, that is, $\theta(\mathrm{A}) = \mathrm{T}$ and $\theta(\mathrm{C}) = \mathrm{G}$. Two complementary DNA single strands of opposite orientation bind together to form a DNA double strand (intermolecular structure). Also, if non-overlapping subwords of a DNA strand are complementary, the strand may bind to itself to form intramolecular structures such as stem-loops, known more commonly as hairpins (Figure 2.6).

As such, hairpins tend to interfere with DNA computations, and therefore are usually explicitly avoided by DNA computing experimentalists when encoding information as DNA strands. See [3, 9, 132, 133, 225] about this problem and about some of the "good" designs of DNA strands that are free of hairpins. However, hairpins are not a structure to always be avoided in DNA computations. For example, hairpins are the main component of "hairpin engine" DNA computing techniques [92, 237, 242]. In [135, 261, 262], hairpins serve as a binary information medium for DNA-based random-access memory. Last but not least, hairpins are the basic components of some DNA-based programmable "smart drugs" [17]. As it turns out, several of the generalizations of the Fibonacci words defined in this paper are guaranteed to form hairpins, which makes them a good candidate for encodings in hairpin-based DNA computations.

Note that other extensions of the atom Fibonacci sequence have been proposed and investigated in the literature, such as:

- The *mapped shuffled Fibonacci languages* is defined as $F_{(u,v)} = \{h(w) \mid w \in F_{(a,b)}\}$, whereby $h(a) = u$, $h(b) = v$, and $F_{(a,b)} = \bigcup_{i\geq 1} F_i$. The languages F_i, $i \geq 1$, are obtained from $F_1 = \{a\}$ and $F_2 = \{b\}$ by the recursive definition $F_{n+2} = F_n \diamond F_{n+1}$ for all $n \geq 1$, where $\diamond$ is the shuffle operation, as shown in [115].

- The sequence $\{s_n\}_{n\geq -1}$ is defined by $s_{-1} = 1$, $s_0 = 0$, and $s_n = s_{n-1}^{d_n} s_{n-2}$ for $n \geq 1$, where $d_1 \geq 0$ and $d_n > 0$ for $n > 1$ [21].

- The *k-Fibonacci words*, whereby $f_{k,0} = 0$, $f_{k,1} = 0^{k-1}1$, and $f_{k,n} = f_{k,n-1}^{k} f_{k,n-2}$ for $n \geq 2$ and $k \geq 1$ [230].

- The *(n,i)-Fibonacci words* whereby $f_0^{[i]} = 0$, $f_1^{[i]} = 0^{i-1}1$, and $f_n^{[i]} = f_{n-1}^{[i]} f_{n-2}^{[i]}$ for $n \geq 2$ and $i \geq 1$ [231, 232].

- The *m-bonacci words*, whereby $f_i = \phi_m^i(0)$, $\phi_m(m-1) = 0$, and $\phi_m(i) = 0(i+1)$, for all $0 \leq i \leq m-2$ [28].

These and other extensions of Fibonacci words, such as that in [69], are out of the scope of this thesis.

4.2 Preliminaries

An *involution* is a function f that is its own inverse, that is, for all x in the domain of f, we have $f(f(x)) = x$. A function $h : \Sigma^* \to \Sigma^*$ is called a *morphism on* Σ^* if $h(\lambda) = \lambda$, and we have that $h(uv) = h(u)h(v)$ for all $u, v \in \Sigma^*$. A function $h : \Sigma^* \to \Sigma^*$ is called an *antimorphism on* Σ^* if $h(\lambda) = \lambda$, and we have that $h(uv) = h(v)h(u)$ for all $u, v \in \Sigma^*$. Note that if h is a morphism on the language Σ^*, then $h(a_1 a_2 \cdots a_n) = h(a_1)h(a_2) \cdots h(a_n)$, and, if h is an antimorphism on Σ^*, then $h(a_1 a_2 \cdots a_n) = h(a_n) \cdots h(a_2)h(a_1)$ for all $a_i \in \Sigma$ and $1 \leq i \leq n$. A function $\phi : \Sigma^* \to \Sigma^*$ is called a *morphic involution on* Σ^* (an *antimorphic involution on* Σ^*) if it is an involution on Σ extended to a morphism on Σ^* (to an antimorphism on Σ^*, respectively). For convenience, in the remainder of this chapter, we use the convention that the letter ϕ denotes an involution that is either morphic or antimorphic (such a function will be termed *[anti]morphic involution*), that the letter θ denotes an antimorphic involution, and that the letter μ denotes a morphic involution.

A word $x \in \Sigma^*$ is called a *border* of $w \in \Sigma^+$ if $w = x\alpha = \beta x$ for some $\alpha, \beta \in \Sigma^*$. A border x of w is said to be proper if $|x| \neq |w|$. A word $x \in \Sigma^*$ is called a *ϕ-border* of $w \in \Sigma^+$ if $w = x\alpha = \beta\phi(x)$ for some $\alpha, \beta \in \Sigma^*$, and a proper ϕ-border if, in addition, $|x| \neq |w|$, see [147]. The empty word λ is a ϕ-border of every $w \in \Sigma^+$. A non-empty word is said to be ϕ-bordered if it has a proper non-empty ϕ-border, and ϕ-unbordered otherwise. If ϕ is the identity on Σ extended to a morphism on Σ^*, then the ϕ-bordered words coincide with the classical bordered words, and the ϕ-unbordered words coincide with the classical unbordered words [280]. We recall:

Lemma 4.2. *[147] Let θ be an antimorphic involution on Σ^*. Then, for all $x \in \Sigma^+$, we have that x is θ-bordered if and only if $x = ay\theta(a)$ for some $a \in \Sigma$ and $y \in \Sigma^*$.*

A word $u \in \Sigma^*$ is called a *conjugate* of $v \in \Sigma^*$ if there exists a word $w \in \Sigma^*$ such that $uw = wv$, or, equivalently, if $u = xy$ and $v = yx$ for $x, y \in \Sigma^*$. In [148] the concept of the conjugacy of words was extended to incorporate the notion of an (anti)morphic involution: A word $u \in \Sigma^*$ is a *ϕ-conjugate* of $v \in \Sigma^*$ if there exists a word $w \in \Sigma^*$ such that $uw = \phi(w)v$. If ϕ is the identity on Σ extended to a morphism on Σ^*, this notion becomes the classic conjugacy on words.

A word $w \in \Sigma^*$ is called a *palindrome* if $w = w^r$. A word $w \in \Sigma^*$ is called a *ϕ-palindrome* if $w = \phi(w)$, and the set of all ϕ-palindromes is denoted by P_ϕ. If $\phi = \mu$ is a morphic involution on Σ^*, then the only μ-palindromes are the words over Σ', where $\Sigma' \subseteq \Sigma$ and μ is the identity on Σ'. Lastly, if $\phi = \theta$ is the identity function on Σ extended to an antimorphism on Σ^*, then a θ-palindrome is a classical palindrome, while, if $\phi = \mu$ is the identity function on Σ extended to a morphism on Σ^*, then every word is a μ-palindrome.

A word $w \in \Sigma^+$ is said to be *primitive* if $w = u^i$ implies $w = u$ and $i = 1$. Let Q denote the set of all primitive words. For every word $w \in \Sigma^+$, there exists a unique word $\rho(w) \in \Sigma^+$ called the *primitive root* of w such that $\rho(w) \in Q$ and $w = \rho(w)^n$ for some $n \geq 1$. A word $w \in \Sigma^+$ is said to be *ϕ-primitive* if $w = \{u, \phi(u)\}^i$ implies $w \in \{u, \phi(u)\}$ and $i = 1$. If ϕ is the identity on Σ extended to a morphism on Σ^*, this notion becomes the classic primitivity on words.

The following observations will be used in the remainder of the chapter.

Lemma 4.3. *For $n, m \geq 1$, the following hold.*

- $\gcd(n, n+1) = 1$.
- *For m even, if* $\gcd(n, m) = 1$, *then* $\gcd(n, \frac{m}{2}) = 1$.

- $\gcd(n, n+2)$ *is* 1 *if* n *is odd and* 2 *if* n *is even.*

We will also make use of the following identities on the Fibonacci numbers F_n, which can be proved using Lemma 4.3 and induction.

Lemma 4.4. *For all* $n \geq 1$*, the following identities hold.*

- $\gcd(F_n, F_{n+1}) = 1$.
- $\gcd(F_n, \frac{F_{n+1}}{2}) = 1$ *for* F_{n+1} *even.*
- $\gcd(\frac{F_n}{2}, F_{n+1}) = 1$ *for* F_n *even.*
- $\gcd(F_n - 1, F_n + 1)$ *is* 1 *if* F_n *is even and* 2 *if* F_n *is odd.*
- $\gcd(\frac{F_n-1}{2}, \frac{F_n+1}{2}) = 1$ *if* F_n *is odd.*

Given a set $\Sigma_4 = \{x_1, x_2, x_3, x_4\}$, out of all possible permutations of Σ_4 of the form

$$\phi = \begin{pmatrix} x_1 & x_2 & x_3 & x_4 \\ \phi(x_1) & \phi(x_2) & \phi(x_3) & \phi(x_4) \end{pmatrix},$$

there are 10 mappings that are involutions on Σ_4. We denote them by ϕ_i, $1 \leq i \leq 10$, and they are listed in Table 4.1.

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8	ϕ_9	ϕ_{10}
x_1	x_1	x_2	x_3	x_4	x_1	x_1	x_1	x_2	x_3	x_4
x_2	x_2	x_1	x_2	x_2	x_3	x_4	x_2	x_1	x_4	x_3
x_3	x_3	x_3	x_1	x_3	x_2	x_3	x_4	x_4	x_1	x_2
x_4	x_4	x_4	x_4	x_1	x_4	x_2	x_3	x_3	x_2	x_1

Table 4.1: List of all possible involutions over the set $\Sigma_4 = \{x_1, x_2, x_3, x_4\}$. If the letter in the cell of the column of ϕ_i and row of x_j is x_k, where $1 \leq i \leq 10$ and $1 \leq j$, $k \leq 4$, this denotes that $\phi_i(x_j) = x_k$. For example, $\phi_3(x_3) = x_1$.

Note that the mapping ϕ_1 is the involution whereby all letters are mapped to themselves (the identity on Σ_4). The mappings ϕ_i, $2 \leq i \leq 7$, are the involutions whereby two of the letters are mapped to each other, and the other two are mapped to themselves. The mappings ϕ_8, ϕ_9, and ϕ_{10} are the only involutions whereby two of the letters are mapped to each other, and the other two letters are also mapped to each other.

Throughout this chapter, we use the convention that for a sequence (x_1, x_2, x_3, x_4) of letters from Σ_4, we have $x_i \neq x_j$ whenever $i \neq j$. In the particular case of the DNA alphabet, that is, where $\Sigma_4 = \Delta = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$, there are a total of $4! = 24$ possibilities for the choice of (x_1, x_2, x_3, x_4) for $x_i \neq x_j$ and $i \neq j$. For each such choice of (x_1, x_2, x_3, x_4), the Watson-Crick involution θ_{DNA} will coincide on Δ^* with one of ϕ_8, ϕ_9, and ϕ_{10}. For example, if we fix $(x_1, x_2, x_3, x_4) = (\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T})$, then ϕ_{10} coincides with θ_{DNA} on Δ^*, whereas, if we fix $(x_1, x_2, x_3, x_4) = (\mathrm{C}, \mathrm{G}, \mathrm{A}, \mathrm{T})$, then ϕ_8 coincides with θ_{DNA} on Δ^*, and, if we fix $(x_1, x_2, x_3, x_4) = (\mathrm{A}, \mathrm{C}, \mathrm{T}, \mathrm{G})$, then ϕ_9 coincides with θ_{DNA} on Δ^*.

Table 4.2 illustrates a particular case from Table 4.1, where we have $(x_1, x_2, x_3, x_4) = (\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T})$. For this example, we list all possible mappings ϕ on Δ that can be extended to an (anti)morphic involution on Δ^*.

	ϕ_1	ϕ_2	ϕ_3	ϕ_4	ϕ_5	ϕ_6	ϕ_7	ϕ_8	ϕ_9	ϕ_{10}
A	A	C	G	T	A	A	A	C	G	T
C	C	A	C	C	G	T	C	A	T	G
G	G	G	A	G	C	G	T	T	A	C
T	T	T	T	A	T	C	G	G	C	A

Table 4.2: List of all possible involutions over the DNA alphabet $\Delta = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$.

In the remainder of the chapter, the mapping ϕ_i on Δ extended to a morphic involution on Δ^* will be denoted by μ_i for $1 \leq i \leq 10$, and similarly, the mapping ϕ_i on Δ extended to an antimorphic involution on Δ^* will be denoted by θ_i for $1 \leq i \leq 10$. Note that for $(x_1, x_2, x_3, x_4) = (\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T})$, the morphic involution μ_1 is the identity on Δ^*, the antimorphic involution θ_1 is the mirror image, and the antimorphic involution $\theta_{10} = \theta_{\mathrm{DNA}}$ formalizes the Watson-Crick complementarity of DNA strings in Δ^*, as shown in [142, 212].

We recall the following from [60, 172, 276].

Theorem 4.5. *[60] For $n \geq 1$, the atom Fibonacci word f_n is primitive.*

Lemma 4.6. *[172] Let $x, y \in \Sigma^+$ be two non-empty words.*

- *If $xy = p^i$, $p \in Q$, and $i \geq 1$, then $yx = q^i$ for some $q \in Q$.*
- *If $xy = yx$, then $\rho(x) = \rho(y)$.*

Proposition 4.7. *[276] Let p and q be primitive and $d = \gcd(|p|, |q|)$. If $p^m = qx$, for $m \geq 2$ with $q = xy$, $y \in \Sigma^+$, and $|x| \geq |p| - d$, then $p = q$.*

Lastly, in Section 4.6 and Section 4.7, we will make an extensive use of the following result.

Lemma 4.8. *For a word x over an alphabet Σ, we have that if $\gcd(|x|_a, |x|_b) = 1$ for at least a pair of letters $a, b \in \Sigma$, then x is primitive.*

Proof. We prove the contrapositive. By definition, if x is not primitive, then it can be written as $x = p^i$, where $p \in Q$ and $i > 1$. For all pairs of letters $a, b \in \mathrm{Alph}(x)$, we have $|x|_a = i \cdot |p|_a$ and $|x|_b = i \cdot |p|_b$. Therefore, $\gcd(|x|_a, |x|_b)$ is a multiple of i, thus, $\gcd(|x|_a, |x|_b) \neq 1$. For all pairs of letters $a, b \in \Sigma$, if one of the letters is not in $\mathrm{Alph}(x)$, then $\gcd(|x|_a, |x|_b) \neq 1$. $\square$

4.3 Involutive Fibonacci words

In this section, we generalize Definition 4.1 to define the n^{th} standard and reverse ϕ-Fibonacci words/sequences in several ways, where ϕ is an (anti)morphic involution. We consider special cases of such ϕ-Fibonacci words where the initial two words u and v have various properties: u and v are both palindromes, u and v are both ϕ-palindromes, or $u = \phi(v)$.

Definition 4.9. *Let Σ be an alphabet with $|\Sigma| \geq 2$, let ϕ be an (anti)morphic involution on Σ^*, and $u, v \in \Sigma^+$. If the first two ϕ-Fibonacci words are u and v, respectively, three types of n^{th}* standard ϕ-Fibonacci words*, $g_n^\phi(u,v)$, $w_n^\phi(u,v)$, and $z_n^\phi(u,v)$, $n \geq 3$, are defined recursively as follows:*

$$\begin{aligned}
g_n^\phi(u,v) &= \phi(g_{n-1}^\phi(u,v)) \cdot g_{n-2}^\phi(u,v) && \textit{(alternating)};\\
w_n^\phi(u,v) &= \phi(w_{n-1}^\phi(u,v)) \cdot \phi(w_{n-2}^\phi(u,v)) && \textit{(palindromic); and}\\
z_n^\phi(u,v) &= z_{n-1}^\phi(u,v) \cdot \phi(z_{n-2}^\phi(u,v)) && \textit{(hairpin).}
\end{aligned}$$

Similarly, three types of n^{th} reverse ϕ-Fibonacci words*, $[g_n^\phi(u,v)]'$, $[w_n^\phi(u,v)]'$, and $[z_n^\phi(u,v)]'$, $n \geq 3$, are defined recursively as follows:*

$$\begin{aligned}
[g_n^\phi(u,v)]' &= [g_{n-2}^\phi(u,v)]' \cdot \phi([g_{n-1}^\phi(u,v)]') && \textit{(alternating)};\\
[w_n^\phi(u,v)]' &= \phi([w_{n-2}^\phi(u,v)]') \cdot \phi([w_{n-1}^\phi(u,v)]') && \textit{(palindromic); and}\\
[z_n^\phi(u,v)]' &= \phi([z_{n-2}^\phi(u,v)]') \cdot [z_{n-1}^\phi(u,v)]' && \textit{(hairpin).}
\end{aligned}$$

The sequence of standard alternating ϕ-Fibonacci words $G(u,v)$ can now be defined as $G(u,v) = \{g_n^\phi(u,v)\}_{n\geq 1}$, and the sequences $W(u,v)$, $Z(u,v)$, $G'(u,v)$, $W'(u,v)$, and $Z'(u,v)$ can be similarly defined.

If the first two ϕ-Fibonacci words in the sequence are singleton letters in Σ, the respective ϕ-Fibonacci words will be called atom (standard or reverse) ϕ-Fibonacci words*. If the involution ϕ is clear from the context, we will sometimes call ϕ-Fibonacci words simply* involutive Fibonacci words*.*

In the remainder of this chapter, when the particular (anti)morphic involution ϕ involved in the Fibonacci recursion needs to be emphasized, we will use the notation $g_n^\phi(u,v)$, $w_n^\phi(u,v)$, or $z_n^\phi(u,v)$ to denote the corresponding ϕ-Fibonacci words for $n \geq 1$. However, if either the initial words u and v or the mapping ϕ is clear from the context (as is the case in Definition 4.9), they will sometimes be omitted.

Note that, in the particular case when ϕ is the identity function on Σ extended to a morphism on Σ^*, the words $g_n = w_n = z_n$ all coincide with the standard Fibonacci words f_n, while $g'_n = w'_n = z'_n$ all coincide with f'_n, for all $n \geq 1$. Thus, f_n and f'_n can also be called standard and reverse ϕ-Fibonacci words, respectively, for all $n \geq 1$, with ϕ being the identify function extended to a morphism.

We now illustrate the definitions with the following examples. Consider the DNA alphabet $\Delta = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$, $\phi(\mathrm{A}) = \mathrm{T}$, $\phi(\mathrm{G}) = \mathrm{C}$, and vice versa. Assume that the first two ϕ-Fibonacci words are A and C, respectively. Table 4.3 describes the first of the various atom standard and reverse ϕ-Fibonacci words, where ϕ is either a morphic involution (MI) or an antimorphic involution (AMI) on Δ^*. Note that, since Δ denotes the DNA alphabet, if the involution $\phi = \theta$ defined as above on Δ is extended to an antimorphism on Δ^*, then it models the DNA Watson-Crick complementarity of DNA strands. In this case, the standard palindromic θ-Fibonacci words w_n and the reverse palindromic θ-Fibonacci words w'_n form hairpin structures with partially double-stranded stems (Figure 2.6 depicts the word w_8), while the standard and reverse hairpin θ-Fibonacci words z_n and z'_n form hairpins with fully double-stranded stems.

It was first shown in [60] that the prefix of the atom standard Fibonacci word f_n of length $|f_n| - 2$ is a palindrome, for all $n \geq 3$. Later in [78], the authors proved that the prefix of length $|f_n| - 2$ of f_n is also the suffix of the atom reverse Fibonacci word f'_n, for all $n \geq 3$. These results from [60, 78] can be combined in the following lemma.

Lemma 4.10. *Let $\Sigma = \{a, b\}$, with $a \neq b$, and let $f_1 = a$ and $f_2 = b$. Then, for $n \geq 3$, we have that $f_n = s_n d_n$ and $f'_n = d'_n s_n$, where $d'_n = d_n^r$ such that s_n is a palindrome, and $d_n = ab$ if n is even, while $d_n = ba$ if n is odd.*

	$g_n = \phi(g_{n-1})g_{n-2}$		$g'_n = g'_{n-2}\phi(g'_{n-1})$	
n	MI	AMI	MI	AMI
3	GA	GA	AG	AG
4	CTC	TCC	CTC	CCT
5	GAGGA	GGAGA	AGGAG	AGAGG
6	CTCCTCTC	TCTCCTCC	CTCTCCTC	CCTCCTCT

	$w_n = \phi(w_{n-1})\phi(w_{n-2})$		$w'_n = \phi(w'_{n-2})\phi(w'_{n-1})$	
n	MI	AMI	MI	AMI
3	GT	GT	TG	TG
4	CAG	ACG	GAC	GCA
5	GTCCA	CGTAC	ACCTG	CATGC
6	CAGGTGTC	GTACGCGT	CTGTGGAC	TGCGCATG

	$z_n = z_{n-1}\phi(z_{n-2})$		$z'_n = \phi(z'_{n-2})z'_{n-1}$	
n	MI	AMI	MI	AMI
3	CT	CT	TC	TC
4	CTG	CTG	GTC	GTC
5	CTGGA	CTGAG	AGGTC	GAGTC
6	CTGGAGAC	CTGAGCAG	CAGAGGTC	GACGAGTC

Table 4.3: The n^{th} atom ϕ-Fibonacci words g_n, g'_n, w_n, w'_n, z_n, and z'_n with initial words A and C, $3 \leq n \leq 6$, where $\phi(\text{A}) = \text{T}$, $\phi(\text{C}) = \text{G}$, is an involution extended to either a morphism (MI) or an antimorphism (AMI).

One can easily observe from Lemma 4.10 that $s_n = s_n^r$, for all $n \geq 3$. Hence, for $n \geq 3$, we have that $d'_n s_n = f'_n = f_n^r = d_n^r s_n^r$. Thus, we conclude the following.

Lemma 4.11. *Let $\Sigma = \{a, b\}$, with $a \neq b$, and let $f_1 = f'_1 = a$, $f_2 = f'_2 = b$. Then we have $f'_n = f_n^r$, for all $n \geq 1$.*

One can easily show that a result similar to Lemma 4.11 also holds for atom standard and atom reverse ϕ-Fibonacci words.

In the remainder of this chapter, we will often have to make statements that hold for several types of ϕ-Fibonacci words. For brevity, we will use the notational convention that a statement of the type "$\alpha_n \in \{f_n, g_n, w_n, z_n\}$ for all $n \geq 1$" means that either we have $\alpha_n = f_n$ for all $n \geq 1$, or that $\alpha_n = g_n$ for all $n \geq 1$, or that $\alpha_n = w_n$ for all $n \geq 1$, or that $\alpha_n = z_n$ for all $n \geq 1$. Moreover, we will use the notational convention that a statement of the type "$\alpha_n = g_n$ for all $n \geq 1$" also implies the statement "$\alpha'_n = g'_n$ for all $n \geq 1$."

Lemma 4.12. *Let $\Sigma = \{a, b, c, d\}$, let ϕ be an (anti)morphic involution on Σ^*, and let the initial ϕ-Fibonacci words be $\alpha_1 = \alpha'_1 = a$, $\alpha_2 = \alpha'_2 = b$. If $\alpha_n \in \{f_n, g_n, w_n, z_n\}$ for all $n \geq 1$, then $\alpha'_n = \alpha^r_n$ for all $n \geq 1$.*

Proof. By strong induction on n. □

Note that Lemma 4.11 and Lemma 4.12 justify our terminology, calling f_n, g_n, w_n, and z_n "standard" ϕ-Fibonacci words, while calling f'_n, g'_n, w'_n, and z'_n "reverse" ϕ-Fibonacci words. Lemma 4.11, which holds for atom Fibonacci words, can now be generalized to Fibonacci words with $f_1 = u$ and $f_2 = v$, provided that u and v are (non-empty) palindromes. Indeed, the following result is a direct corollary of Theorem 4 in [49], which stated that $f_n(u, v)$ and $f'_n(u, v)$ are conjugates for every $u, v \in \Sigma^+$.

Corollary 4.13. *Let $f_1 = f'_1 = u$ and $f_2 = f'_2 = v$, such that u and v in Σ^+ are palindromes. Then, for all $n \geq 3$, there exist non-empty palindromes $x_n, y_n \in \Sigma^+$ such that $f_n = x_n y_n$ and $f'_n = y_n x_n$, and hence $f'_n = f^r_n$.*

Note that, in general, the decomposition of f_n as a product of two non-empty palindromes x_n and y_n (Corollary 4.13) is not necessarily unique. For example, if we have $f_1 = bab$ and $f_2 = aba$, then $f_3 = ababab = (a)(babab) = (ababa)(b) = (aba)(bab)$ has three different decompositions into a product of two palindromes.

In the following, we will prove a result similar to Corollary 4.13, for the ϕ-Fibonacci words g_n and g'_n, for both morphic as well as antimorphic involutions. The following result is a generalization of Lemma 4.10, to the case of g_n and g'_n.

Proposition 4.14. *Let ϕ be an (anti)morphic involution on Σ^*, and let $g_1 = g'_1 = u$ and $g_2 = g'_2 = v$, where $u, v \in \Sigma^+$. Then, for $n \geq 3$, we have:*

$$g_n = \begin{cases} s_n xy : n \bmod 2 = 0, \\ s_n pq : n \bmod 2 = 1, \end{cases} \qquad g'_n = \begin{cases} yxs'_n : n \bmod 2 = 0, \\ qps'_n : n \bmod 2 = 1, \end{cases}$$

where $s_3 = s'_3 = \lambda$, and

- *If $\phi = \mu$ is a morphic involution, then $x = \mu(u)$, $y = v$, $p = \mu(v)$, $q = u$, and for all $n \geq 4$, $s_n = s'_n = \mu(s_{n-1})g'_{n-2} = g_{n-2}\mu(s_{n-1})$. In addition, for $n \geq 4$, there exists word y_n such that $y_n g_n = g'_n y_n$ and $y_n = g'_{n-2}\mu(u)v$ when n is even and $y_n = g'_{n-2}\mu(v)u$ otherwise.*

- *If $\phi = \theta$ is an antimorphic involution, then $x = y = v$, $p = \theta(v)$, $q = u$ and for all $n \geq 4$, $s_n = \theta(y_{n-1})\theta(s'_{n-1})$ and $s'_n = \theta(s_{n-1})\theta(y_{n-1})$. In addition, for $n \geq 4$, there exists a word y_n such that $y_n g_n = g'_n y_n$ and $y_n = y_{n-2}g_{n-2}$, where $y_3 = u$ and $y_2 = v$.*

Proposition 4.14 can now be used to prove Corollary 4.15, which generalizes Theorem 4 in [49] and Corollary 4.13 to the case of g_n and g'_n.

Corollary 4.15. *Let ϕ be an (anti)morphic involution on Σ^*, and let the two initial words $g_1 = g'_1 = u$ and $g_2 = g'_2 = v$ be words in Σ^+. Then the words g_n and g'_n are conjugates for all $n \geq 1$. If in addition, u and v are palindromes, then for all $n \geq 3$, there exists palindromes x_n and y_n such that $g_n = x_n y_n$, $g'_n = y_n x_n$, and hence also $g'_n = g^r_n$.*

Given an (anti)morphic involution ϕ and initial words g_1 and g_2, the decomposition of g_n into palindromes is not necessarily unique. Consider for example the (anti)morphic involution ϕ such that $\phi(a) = b$ and $\phi(b) = a$. If $g_1 = bab$ and $g_2 = aba$, then $g_4 = abaabaaba$, which can be expressed as $g_4 = (aba)(abaaba) = (abaaba)(aba)$.

Theorem 4 of [49] showed that the Fibonacci words f_n and f'_n are conjugates of each other. Similarly, Corollary 4.15 shows that the ϕ-Fibonacci words g_n and g'_n are conjugates of each other, for both morphic and antimorphic ϕ. This does not hold for ϕ-Fibonacci words z_n and w_n, as shown by checking the case $n = 4$ in Table 4.3.

Even though we cannot prove conjugacy of z_n and z'_n (w_n and w'_n, respectively) for all $n \geq 1$ in the general case, the following proposition holds, implying that if the first two Fibonacci words are palindromes, then $z'_n = z^r_n$ and $w'_n = w^r_n$ (Corollary 4.17).

Proposition 4.16. *Let ϕ be an (anti)morphic involution on Σ^*, and let $\alpha_1 = \alpha'_1 = u$ and $\alpha_2 = \alpha'_2 = v$, where $u, v \in \Sigma^+$. If $\alpha_n \in \{z_n, w_n\}$ for all $n \geq 1$, then, for all $n \geq 3$, we have that $\alpha_n = s_n d_n$ and $\alpha'_n = d'_n s'_n$, where:*

- *If $\alpha_n = z_n$ and $\phi = \mu$ is a morphic involution, then*

$$d_n = \begin{cases} \mu(u)\mu(v) : n \bmod 4 = 0, \\ \mu(v)u : n \bmod 4 = 1, \\ uv : n \bmod 4 = 2, \\ v\mu(u) : n \bmod 4 = 3, \end{cases} \qquad d'_n = \begin{cases} \mu(v)\mu(u) : n \bmod 4 = 0, \\ u\mu(v) : n \bmod 4 = 1, \\ vu : n \bmod 4 = 2, \\ \mu(u)v : n \bmod 4 = 3. \end{cases}$$

- *If $\alpha_n = z_n$ and $\phi = \theta$ is an antimorphic involution, then we have $d_n = u\theta(v)$ and $d'_n = \theta(v)u$ for all $n \geq 5$.*

- *If $\alpha_n = w_n$ and $\phi = \mu$ is a morphic involution, then*

$$d_n = \begin{cases} u\mu(v) : n \bmod 4 = 0, \\ vu : n \bmod 4 = 1, \\ \mu(u)v : n \bmod 4 = 2, \\ \mu(v)\mu(u) : n \bmod 4 = 3, \end{cases} \qquad d'_n = \begin{cases} \mu(v)u : n \bmod 4 = 0, \\ uv : n \bmod 4 = 1, \\ v\mu(u) : n \bmod 4 = 2, \\ \mu(u)\mu(v) : n \bmod 4 = 3. \end{cases}$$

- *If $\alpha_n = w_n$ and $\phi = \theta$ is an antimorphic involution, then*

$$d_n = \begin{cases} uv : n \bmod 3 = 1, \\ \theta(v)\theta(u) : n \bmod 3 = 2, \\ v\theta(v) : n \bmod 3 = 0, \end{cases} \qquad d'_n = \begin{cases} vu : n \bmod 3 = 1, \\ \theta(u)\theta(v) : n \bmod 3 = 2, \\ \theta(v)v : n \bmod 3 = 0. \end{cases}$$

Proof. We only prove the case when $\alpha_n = w_n$ for all $n \geq 1$ and $\phi = \mu$ is a morphic involution by strong induction on n. The base cases for $n \in \{4, 5, 6, 7\}$ can be verified directly where $\mu(u) = u'$ and $\mu(v) = v'$. Assume the statement is true for all $k \leq n$, and consider w_{n+1}. If $n + 1 \bmod 4 = 0$, then we have that

$$w_{n+1} = \mu(w_n)\mu(w_{n-1}) = \mu(s_n v' u' s_{n-1} u' v) = \mu(s_n) vu\mu(s_{n-1}) uv' = s_{n+1} uv',$$

where $s_{n+1} = \mu(s_n)vu\mu(s_{n-1})$, and at the same time

$$w'_{n+1} = \mu(w'_{n-1})\mu(w'_n) = \mu(vu's'_{n-1})\mu(u'v's'_n) = v'u\mu(s'_{n-1})uv\mu(s'_n) = v'us'_{n+1},$$

where $s'_{n+1} = \mu(s'_{n-1})uv\mu(s'_n)$. If $u = u^r$ and $v = v^r$, then we have a decomposition whereby $s'_{n+1} = \mu(s^r_{n-1})uv\mu(s^r_n) = s^r_{n+1}$. Also, note that $w'_{n+1} = w^r_{n+1}$. Cases where $n + 1 \bmod 4 \in \{1, 2, 3\}$ can be proved similarly. $\square$

Corollary 4.17. *Let ϕ be an (anti)morphic involution on Σ^*, let $u, v \in \Sigma^+$, and let $\alpha_1 = \alpha'_1 = u$ and $\alpha_2 = \alpha'_2 = v$. If $\alpha_n \in \{z_n, w_n\}$ for all $n \geq 1$, then for all $n \geq 3$, we have that $\alpha_n = s_n d_n$ and $\alpha'_n = d'_n s'_n$ as in Proposition 4.16. If in addition u and v are palindromes, then for all $n \geq 5$, we have that $\alpha'_n = \alpha^r_n$, $s'_n = s^r_n$, and $d'_n = d^r_n$.*

Corollary 4.13, Corollary 4.15, and Corollary 4.17 show that in the special case of u and v being non-empty palindromes, if $\alpha_n \in \{f_n, g_n, w_n, z_n\}$ for all $n \geq 1$, then α'_n is the reverse of α_n for all $n \geq 5$. We now consider other special cases by imposing other constraints on the initial words u and v. We first consider the case when the initial words u and v are (non-empty) ϕ-palindromes. If $\phi = \mu$ is a morphic involution, then μ is the identity mapping on $\text{Alph}(u) \cup \text{Alph}(v)$, and a newly defined ϕ-Fibonacci word g_n, w_n, or z_n coincides with f_n for all $n \geq 1$ (the case of f_n was discussed in Corollary 4.13). The following proposition considers the case when $\phi = \theta$ is an antimorphic involution.

Theorem 4.18. *Let $f_i = f'_i$, $g_i = g'_i$, $w_i = w'_i$, and $z_i = z'_i$ for $i = 1, 2$, and let θ be an antimorphic involution on Σ^*. If f_i, g_i, w_i, and z_i are non-empty θ-palindromes for $i = 1, 2$, then:*

- *For all $n \geq 1$, $f_n = \theta(f'_n)$, $g_n = \theta(g'_n)$, $w_n = \theta(w'_n)$, and $z_n = \theta(z'_n)$.*
- *For all $n \geq 5$, g_n and g'_n are conjugates of each other, namely $g_n = x_n y_n$ and $g'_n = y_n x_n$, where x_n, y_n are θ-palindromes. In addition, x_{2n} can be decomposed as $x_{2n} = y_{2n-1} = g_1 g_3 \cdots g_{2n-5} g_{2n-3}$, while x_{2n+1} can be decomposed as $x_{2n+1} = y_{2n} = g_2^2 g_4 g_6 \cdots g_{2n-4} g_{2n-2}$.*
- *If $n \bmod 3 \neq 0$, we have that $z_n = z'_n$. Otherwise, we have that z_n and z'_n are conjugates, that is, $z_n = x_n y_n$ and $z'_n = y_n x_n$, where both $x_n = z_{n-1}$ and $y_n = z_{n-2}$ are θ-palindromes.*
- *For all $n \geq 5$, w_n and w'_n are θ-conjugates, that is, we have $w_n = x_n y_n$ and $w'_n = \theta(y_n) x_n$, where x_n is a θ-palindrome, and we have decompositions whereby $x_n = w_{n-3} w'_{n-3}$ and $y_n = w'_{n-4} w'_{n-2}$.*

Proof. By strong induction on n. □

It was observed in [104] that the length of the atom reverse Fibonacci word f'_n is the Fibonacci number F_n. For other ϕ-Fibonacci words $\alpha_n \in \{g_n, z_n, w_n, g'_n, z'_n, w'_n\}$ for all $n \geq 1$, and if either $\alpha_1 = \alpha_2$ or $\phi(\alpha_1) = \alpha_2$, then $\alpha_n = u_1 u_2 u_3 \cdots u_k$, where, for $1 \leq i \leq k$, $u_i \in \{\alpha_1, \phi(\alpha_1)\}$, and therefore the length of the nth ϕ-Fibonacci word equals $|\alpha_1| \times F_n$.

We end this section by considering another special case of ϕ-Fibonacci words, where the initial words u and v satisfy the condition $\phi(u) = v$.

Proposition 4.19. *Let $u, v \in \Sigma^+$, let $g_1 = w_1 = z_1 = g'_1 = w'_1 = z'_1 = u$, and let $g_2 = w_2 = z_2 = g'_2 = w'_2 = z'_2 = v$. Let ϕ be an (anti)morphic involution on Σ^* such that $\phi(u) = v$. Then, for all $n \geq 3$, we have:*

1. *If n is odd, $g_n = g'_n = u^{F_n}$, and if n is even, $g_n = g'_n = v^{F_n}$.*
2. *If $\phi = \mu$ is a morphic involution and u and v are palindromes, then*

$$\begin{cases} w_n = x_n y_n, w'_n = y_n x_n^r : n \bmod 2 = 0, \\ w_n = x_n y_n, w'_n = y_n^r x_n : n \bmod 2 = 1. \end{cases}$$

3. *If $\phi = \theta$ is an antimorphic involution and u and v are palindromes, then*

$$\begin{cases} w_n = (uv)^i, w'_n = (vu)^i : n \bmod 3 = 0, i = \frac{|F_n|}{2}, \\ w_n = w'_n : n \bmod 3 \neq 0. \end{cases}$$

Proof. 1. Follows from the definition of g_n, g'_n, and the hypothesis that $\phi(u) = v$.

2. Follows by induction on n, from the definition of w_n, w'_n, and the assumptions that $\mu(u) = v$ and that u and v are palindromes.

3. Proof similar to that of (2).

□

4.3.1 Relations between Fibonacci words and μ-Fibonacci words

In this section, we find relationships that exist between alternating μ-Fibonacci words g_n and standard Fibonacci words f_n for morphic involutions μ (see Theorem 4.21). We namely prove that, as suggested by Table 4.3, in the case of a morphic involution μ, the words in the sequence of μ-Fibonacci words $\{g_n(u,v)\}_{n\geq 1}$ can be obtained by alternating the words from two different sequences of standard Fibonacci words, $\{f_n(u,\mu(v))\}_{n\geq 1}$ and $\{f_n(\mu(u),v)\}_{n\geq 1}$, as follows: $g_n(u,v)$ coincides with $f_n(u,\mu(v))$ for odd n and with $f_n(\mu(u),v)$ for even n. This property was the rationale for calling g_n "alternating Fibonacci words." A similar relationship holds between sequences $\{g'_n\}_{n\geq 1}$ and $\{f'_n\}_{n\geq 1}$, and between sequences $\{w_n\}_{n\geq 1}$ (respectively $\{w'_n\}_{n\geq 1}$) and $\{z_n\}_{n\geq 1}$ (respectively $\{z'_n\}_{n\geq 1}$). The following lemma is used to prove these relationships.

Lemma 4.20. *Let $\phi = \mu_b$ be a morphic involution on Σ^*, let μ_a be a morphic involution on Σ^* such that $\mu_a\mu_b = \mu_b\mu_a$, and let $u, v \in \Sigma^+$. If $\alpha_n \in \{f_n, g_n, w_n, z_n\}$ for all $n \geq 1$ are μ_b-Fibonacci words then $\mu_a(\alpha_n(u,v)) = \alpha_n(\mu_a(u),\mu_a(v))$ and $\mu_a(\alpha'_n(u,v)) = \alpha'_n(\mu_a(u),\mu_a(v))$ for all $n \geq 1$.*

Proof. We consider the standard μ_b-Fibonacci words g_n. The proof is by strong induction on n. By definition, we have $\mu_a(g_1(u,v)) = \mu_a(u) = g_1(\mu_a(u),\mu_a(v))$, and $\mu_a(g_2(u,v)) = \mu_a(v) = g_2(\mu_a(u),\mu_a(v))$, so the base case holds. Assume $\mu_a(g_i(u,v)) = g_i(\mu_a(u),\mu_a(v))$ for all $1 \leq i \leq k$. Using the definition of g_n for $n \geq 1$, the fact that μ_a is a morphism, and the induction hypothesis, we have that

$$\mu_a(g_{k+1}(u,v)) = \mu_a(\mu_b(g_k(u,v)) \cdot g_{k-1}(u,v))$$

$$\begin{aligned}&= \mu_b(\mu_a(g_k(u,v))) \cdot \mu_a(g_{k-1}(u,v)) \\ &= \mu_b(g_k(\mu_a(u),\mu_a(v))) \cdot g_{k-1}(\mu_a(u),\mu_a(v)) \\ &= g_{k+1}(\mu_a(u),\mu_a(v)).\end{aligned}$$

The proofs for other μ_b-Fibonacci words are similar. □

Lemma 4.20 can be used to prove the following result.

Theorem 4.21. *Let μ be a morphic involution on Σ^*, let $u, v \in \Sigma^+$, and let*

$$(\alpha_n, \beta_n) \in \{(f_n, g_n), (g_n, f_n), (z_n, w_n), (w_n, z_n), (f'_n, g'_n), (g'_n, f'_n), (z'_n, w'_n), (w'_n, z'_n)\},$$

for all $n \geq 1$. The following relations hold for all $n \geq 1$:

$$\alpha_n(u,v) = \begin{cases} \beta_n(\mu(u), v) : n \bmod 2 = 0, \\ \beta_n(u, \mu(v)) : n \bmod 2 = 1. \end{cases}$$

Proof. We prove the pair (g_n, f_n) using strong induction on n. By definition, we have $g_1(u,v) = u = f_1(u, \mu(v))$ and $g_2(u,v) = v = f_2(\mu(u), v)$.

Assume now that for $1 \leq i \leq k$, $g_i(u,v) = f_i(u,\mu(v))$ if i is odd and $g_i(u,v) = f_i(\mu(u), v)$ if i is even.

If $k+1$ is odd (the case of $k+1$ even is similar), then per the definition of g_{k+1} and f_{k+1}, the induction hypothesis, and Lemma 4.20 with $\mu_a = \mu_b = \mu$, we have that:

$$\begin{aligned} g_{k+1}(u,v) &= \mu(g_k(u,v)) \cdot g_{k-1}(u,v) \\ &= \mu(f_k(\mu(u),v)) \cdot f_{k-1}(u,\mu(v)) \\ &= f_k(u,\mu(v)) \cdot f_{k-1}(u,\mu(v)) \\ &= f_{k+1}(u,\mu(v)). \end{aligned}$$

The proofs for other cases are similar. □

In the case of an antimorphic involution, a relation like that of Theorem 4.21 does not hold. Indeed, Table 4.3 shows that

$$g_4(\mathrm{A},\mathrm{C}) = \mathrm{TCC} \neq \mathrm{CTC} = f_4(\theta(\mathrm{A}),\mathrm{C}) = f_4(\mathrm{T},\mathrm{C}).$$

We will end this subsection with some observations on iterated morphisms generating certain types of involutive Fibonacci words. Let $\Delta = \{\mathrm{A},\mathrm{C},\mathrm{G},\mathrm{T}\}$ be the DNA alphabet and θ be the Watson-Crick antimorphic involution on Δ^* that maps A to T and C to G. Then, assuming that A and C are the first two Fibonacci words, we have that:

- The word g_n can be obtained by iterating on A the morphism h_g defined as $h_g(\mathrm{A}) = \mathrm{C}$, $h_g(\mathrm{C}) = \mathrm{GA}$, $h_g(\mathrm{G}) = \mathrm{TC}$, and $h_g(\mathrm{T}) = \mathrm{G}$.
- The word w_n can be obtained by iterating on A the morphism h_w defined as $h_w(\mathrm{A}) = \mathrm{C}$, $h_w(\mathrm{C}) = \mathrm{GT}$, $h_w(\mathrm{T}) = \mathrm{G}$, and $h_w(\mathrm{G}) = \mathrm{AC}$.
- The word z_n can be obtained by iterating on A the morphism h_z defined as $h_z(\mathrm{A}) = \mathrm{C}$, $h_z(\mathrm{C}) = \mathrm{CT}$, $h_z(\mathrm{T}) = \mathrm{G}$, and $h_z(\mathrm{G}) = \mathrm{AG}$.

4.4 Indexed involutive Fibonacci words

In this section we show that in the case of both a morphic and an antimorphic involution, the ϕ-Fibonacci words are connected to the indexed Fibonacci words defined and studied in [49]. We also define *indexed ϕ-Fibonacci words* (Definition 4.28), which are a generalization of indexed Fibonacci words, and find relationships between various types of such words.

We first show the relations between the θ-Fibonacci words g_n and indexed Fibonacci words. Recall the notion of *indexed Fibonacci words*, defined and investigated in [49] (note that [49] used a different notation):

Definition 4.22. *Let Σ be an alphabet, and let $u, v \in \Sigma^+$. The indexed Fibonacci words are defined recursively as*

$$f^0(u,v) = u, f^{00}(u,v) = v,$$

and, for all $n \geq 2$,

$$f^{r_1r_2\cdots r_n0}(u,v) = f^{r_1r_2\cdots r_n}(u,v)f^{r_1r_2\cdots r_{n-1}}(u,v) \text{ and}$$

$$f^{r_1r_2\cdots r_n1}(u,v) = f^{r_1r_2\cdots r_{n-1}}(u,v)f^{r_1r_2\cdots r_n}(u,v),$$

where $r_1 = r_2 = 0$ and $r_i \in \{0,1\}$ for all $3 \leq i \leq n$.

Informally, in the construction of an indexed Fibonacci word $f^{00r_3r_4\cdots r_nr_{n+1}}(u,v)$, we use the digit $r_{n+1} = 0$ to denote concatenating the last word with the second last word in the sequence (according to the standard Fibonacci concatenation order), and digit $r_{n+1} = 1$ to denote concatenating the second last word with the last word (according to the reverse Fibonacci concatenation order). Note that the standard (respectively reverse) Fibonacci words now become particular cases of indexed Fibonacci words, in the construction of which the standard Fibonacci concatenation order (respectively reverse Fibonacci concatenation order) is always used, that is, $r_n = 0$ for all $n \geq 3$ (respectively $r_n = 1$ for all $n \geq 3$), as follows:

$$f_1(u,v) = f'_1(u,v) = f^0(u,v) = u, f_2(u,v) = f'_2(u,v) = f^{00}(u,v) = v, \text{ and}$$

$$\text{for } n \geq 3, f_n(u,v) = f^{00\,0^{n-2}}(u,v) \text{ and } f'_n(u,v) = f^{00\,1^{n-2}}(u,v).$$

As before, when the initial words u, v are clear for the context, the argument (u,v) will be omitted.

The derivation of a sequence of indexed Fibonacci words can be represented by a path from the root (f^0 and f^{00}) to a leaf $f^{00r_3\cdots r_n}$, $n \geq 3$, in a tree-like structure as follows:

$$f^0 = u, f^{00} = v \to \begin{cases} f^{000} = vu \to \begin{cases} f^{0000} = vuv \to \cdots \\ f^{0001} = vvu \to \cdots \end{cases} \\ f^{001} = uv \to \begin{cases} f^{0010} = uvv \to \cdots \\ f^{0011} = vuv \to \cdots \end{cases} \end{cases}$$

We now recall a result from [49].

Proposition 4.23. *If u and v are non-empty palindromes, then for all $n \geq 3$, we have that $f^{00r_3r_4\cdots r_n}(u,v) = (f^{00s_3s_4\cdots s_n}(u,v))^r$, where $s_j = 1 - r_j$ for $3 \leq j \leq n$.*

The following Lemma generalizes the above result and will aid in the proof of Proposition 4.25.

Lemma 4.24. *Let ϕ be an (anti)morphic involution on Σ^* and $u, v \in \Sigma^+$. Then, for all $n \geq 3$, we have that $\phi(f^{00r_3\cdots r_n}(u,v)) = f^{00s_3\cdots s_n}(\phi(u), \phi(v))$, where $r_i \in \{0,1\}$, and for all $3 \leq i \leq n$*

$$\begin{cases} s_i = r_i : \phi \text{ is a morphic involution,} \\ s_i = 1 - r_i : \phi \text{ is an antimorphic involution.} \end{cases}$$

Proof. We only prove for the case when $\phi = \theta$ is an antimorphic involution by strong induction on n. The base case ($n = 3$) follows by the definition of indexed Fibonacci words and the fact that θ is antimorphic. Indeed, we have that

$$\theta(f^{000}(u,v)) = \theta(vu) = \theta(u)\theta(v) = f^{001}(\theta(u), \theta(v)),$$

$$\theta(f^{001}(u,v)) = \theta(uv) = \theta(v)\theta(u) = f^{000}(\theta(u), \theta(v)).$$

The case $n = 4$ can be proved similarly.

Assume now that $\theta(f^{00r_3\cdots r_j}(u,v)) = f^{00s_3\cdots s_j}(\theta(u),\theta(v))$, where $r_i \in \{1,0\}$ and $s_i = 1 - r_i$, $3 \le i \le j$, for all $3 \le j \le k$ and $k \ge 4$. Consider $f^{00r_3\cdots r_k r_{k+1}}(u,v)$ and assume that $r_{k+1} = 0$. Using the definition of indexed Fibonacci words and the induction hypothesis, we have:

$$\begin{aligned}\theta(f^{00r_3\cdots r_k 0}(u,v)) &= \theta(f^{00r_3\cdots r_k}(u,v)f^{00r_3\cdots r_{k-1}}(u,v))\\ &= \theta(f^{00r_3\cdots r_{k-1}}(u,v))\theta(f^{00r_3\cdots r_k}(u,v))\\ &= f^{00s_3\cdots s_{k-1}}(\theta(u),\theta(v))f^{00s_3\cdots s_k}(\theta(u),\theta(v))\\ &= f^{00s_3\cdots s_{k-1}s_k 1}(\theta(u),\theta(v)).\end{aligned}$$

The case $r_{k+1} = 1$ can be proved similarly. Thus, the inductive step and the proof hold. □

The next result shows that for an antimorphic involution θ, the sequence of alternating θ-Fibonacci words g_n consists of interleaving words from two sequences: If n is odd, it takes the word from the sequence of indexed Fibonacci words $f^{000101010101\cdots}$ (which alternates between the standard and reverse concatenation in its construction). If n is even, it takes the word from the sequence of indexed Fibonacci words $f^{0010101010\cdots}$ (which alternates between the reverse and standard concatenation in its construction).

Proposition 4.25. *Let θ be an antimorphic involution on Σ^*, let u, v be two words in Σ^+, and let $\alpha_n \in \{g_n, g'_n\}$ for all $n \ge 1$. The following relations hold for all $n \ge 3$:*

$$\alpha_n(u,v) = \begin{cases} f^{00\{sr\}^i}(\theta(u),v) : n \bmod 2 = 0, i = \frac{n-2}{2},\\ f^{00r\{sr\}^i}(u,\theta(v)) : n \bmod 2 = 1, i = \frac{n-3}{2},\end{cases}$$

where $r = 0, s = 1$ if $\alpha_n = g_n$, and $r = 1, s = 0$ if $\alpha_n = g'_n$.

Proof. It follows by strong induction on n, using Lemma 4.24. □

A relationship similar to that of Proposition 4.25 can be obtained for the case of morphic involutions as stated in Proposition 4.27, the immediate proof of which uses Lemma 4.24 and Lemma 4.26.

Lemma 4.26. *Let u, v be two words in Σ^+. Then, $f^{\gamma 00}(u,v) = f^{\gamma 11}(u,v)$ for all words $\gamma = r_1 r_2 \cdots r_n$, $n \ge 2$, where $r_1 = r_2 = 0$, $r_i \in \{0,1\}$ for $3 \le i \le n$.*

Proof. It follows $f^{\gamma 00}(u,v) = f^{\gamma}(u,v)f^{r_1 r_2 \cdots r_{n-1}}(u,v)f^{\gamma}(u,v) = f^{\gamma 11}(u,v)$. □

Proposition 4.27. *Let μ be a morphic involution on Σ^*, let u, v be two words in Σ^+, and let $\alpha_n \in \{g_n, g'_n\}$ for all $n \geq 1$. Then, $\alpha_3(u,v) = f^{00r}(u, \mu(v))$ and the following relations hold for $n \geq 4$:*

$$\alpha_n(u,v) = \begin{cases} f^{00r^i ss}(\mu(u), v) : i = n-4, n \bmod 2 = 0, \\ f^{00r^i ss}(u, \mu(v)) : i = n-4, n \bmod 2 = 1, \end{cases}$$

where $r = 0, s = 1$ if $\alpha_n = g_n$, and $r = 1, s = 0$ if $\alpha_n = g'_n$.

We now extend the concept of indexed Fibonacci words defined and studied in [49] to indexed ϕ-Fibonacci words.

Definition 4.28. *Let ϕ be an (anti)morphic involution on Σ^*, let $u, v \in \Sigma^+$, and let $\alpha \in \{f, g, w, z\}$. The* indexed ϕ-Fibonacci words *are defined recursively as*

$$\alpha^0(u,v) = u, \alpha^{00}(u,v) = v,$$

and for all $n \geq 2$ we have that $r_1 = r_2 = 0$, $r_i \in \{0,1\}$ for $3 \leq i \leq n$, and:

$$\alpha^{r_1 r_2 \cdots r_n 0}(u,v) = \begin{cases} \alpha^{r_1 r_2 \cdots r_n}(u,v) \cdot \alpha^{r_1 r_2 \cdots r_{n-1}}(u,v) & : \alpha = f, \\ \phi(\alpha^{r_1 r_2 \cdots r_n}(u,v)) \cdot \alpha^{r_1 r_2 \cdots r_{n-1}}(u,v) & : \alpha = g, \\ \phi(\alpha^{r_1 r_2 \cdots r_n}(u,v)) \cdot \phi(\alpha^{r_1 r_2 \cdots r_{n-1}}(u,v)) & : \alpha = w, \\ \alpha^{r_1 r_2 \cdots r_n}(u,v) \cdot \phi(\alpha^{r_1 r_2 \cdots r_{n-1}}(u,v)) & : \alpha = z, \end{cases}$$

$$\alpha^{r_1 r_2 \cdots r_n 1}(u,v) = \begin{cases} \alpha^{r_1 r_2 \cdots r_{n-1}}(u,v) \cdot \alpha^{r_1 r_2 \cdots r_n}(u,v) & : \alpha = f, \\ \alpha^{r_1 r_2 \cdots r_{n-1}}(u,v) \cdot \phi(\alpha^{r_1 r_2 \cdots r_n}(u,v)) & : \alpha = g, \\ \phi(\alpha^{r_1 r_2 \cdots r_{n-1}}(u,v)) \cdot \phi(\alpha^{r_1 r_2 \cdots r_n}(u,v)) & : \alpha = w, \\ \phi(\alpha^{r_1 r_2 \cdots r_{n-1}}(u,v)) \cdot \alpha^{r_1 r_2 \cdots r_n}(u,v) & : \alpha = z. \end{cases}$$

Note that for a morphic involution μ, the results in Lemma 4.26 hold also for the indexed μ-Fibonacci words g, while Proposition 4.27 also holds if the roles of f and g are swapped. However, one can easily verify that Lemma 4.26 does not hold for the indexed μ-Fibonacci words z or w, and Proposition 4.27 does not hold in the case where $\alpha_n \in \{z_n, z'_n\}$ for all $n \geq 1$ and f is the indexed μ-Fibonacci word w, or in the case where $\alpha_n = \{w_n, w'_n\}$ for all $n \geq 1$ and f is the indexed μ-Fibonacci word z.

However, the following results hold, which extends Theorem 4.21 to the case of indexed Fibonacci words.

Proposition 4.29. *Let μ be a morphic involution on Σ^*, let $u, v \in \Sigma^+$, and let $(\alpha_n, \beta) \in \{(f_n, g), (g_n, f), (z_n, w), (w_n, z), (f'_n, g'), (g'_n, f'), (z'_n, w'), (w'_n, z')\}$ for all $n \geq 1$. Then the following relations hold for all $n \geq 1$:*

$$\alpha_n(u,v) = \begin{cases} \beta^{00r^{n-2}}(\mu(u), v) : n \bmod 2 = 0, \\ \beta^{00r^{n-2}}(u, \mu(v)) : n \bmod 2 = 1, \end{cases}$$

where $r = 0$ if $\alpha_n \in \{g_n, f_n, z_n, w_n\}$, and $r = 1$ if $\alpha_n \in \{g'_n, f'_n, z'_n, w'_n\}$.

Proof. It follows by strong induction on n. □

We now generalize Lemma 4.24 to indexed ϕ-Fibonacci words as follows.

Lemma 4.30. *Let $\phi = \phi_2$ be an (anti)morphic involution on Σ^*, and let ϕ_1 be an (anti)morphic involution on Σ^* such that $\phi_1\phi_2 = \phi_2\phi_1$. Let u, v be two words in Σ^+, and let $\alpha \in \{f, g, w, z\}$ be constructed using the (anti)morphic involution ϕ_2. Then, for all $n \geq 1$, we have that $\phi_1(\alpha^{r_1r_2r_3\cdots r_n}(u,v)) = \alpha^{s_1s_2s_3\cdots s_n}(\phi_1(u), \phi_1(v))$, where $r_1 = r_2 = s_1 = s_2 = 0$, and for all $3 \leq i \leq n$, we have:*

$$\begin{cases} s_i = r_i, \text{ if } \phi_1 \text{ is a morphic involution}, \\ s_i = 1 - r_i, \text{ if } \phi_1 \text{ is an antimorphic involution}. \end{cases}$$

Proof. It follows by induction on n. □

Using Lemma 4.30, one can now show relations between various indexed θ-Fibonacci words, similar to those of Proposition 4.25.

Proposition 4.31. *Let θ be an antimorphic involution on Σ^*, let $u, v \in \Sigma^+$, and let $(\alpha_n, \beta) \in \{(f_n, g), (g_n, f), (z_n, w), (w_n, z), (f'_n, g'), (g'_n, f'), (z'_n, w'), (w'_n, z')\}$ for all $n \geq 1$. The following relations hold for all $n \geq 3$:*

$$\alpha_n(u,v) = \begin{cases} \beta^{00\{sr\}^i}(\theta(u), v) : i = \frac{n-2}{2}, n \geq 4, n \bmod 2 = 0, \\ \beta^{00r\{sr\}^i}(u, \theta(v)) : i = \frac{n-3}{2}, n \geq 3, n \bmod 2 = 1, \end{cases}$$

where $r = 0, s = 1$ if $\alpha_n \in \{g_n, f_n, z_n, w_n\}$, and $r = 1, s = 0$ if $\alpha_n \in \{g'_n, f'_n, z'_n, w'_n\}$.

Proof. The statement follows from Lemma 4.30 by induction on n. □

4.5 Borders and ϕ-borders of ϕ-Fibonacci words

It is well known that both the standard and reverse Fibonacci words are bordered for all $n \geq 3$. In this section, we investigate the borderedness and ϕ-borderedness of ϕ-Fibonacci words. As seen in the next example, the borderedness of ϕ-Fibonacci words depends on the two initial Fibonacci words, as well as on the involution under consideration.

Example 4.32. Let ϕ be an (anti)morphic involution on Δ^*, where $\Delta = \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}$, defined as $\phi(\mathrm{A}) = \mathrm{T}$, $\phi(\mathrm{C}) = \mathrm{G}$, and vice versa, and let $g_1 = w_1 = z_1 = \mathrm{AC}$ and $g_2 = w_2 = z_2 = \mathrm{T}$. Consider the ϕ-Fibonacci words w_n, g_n, z_n, for $n \geq 3$.

If $\phi = \theta$ is an antimorphic involution:

- The first standard palindromic θ-Fibonacci words are

  $$w_3 = \mathrm{AGT}, w_4 = \mathrm{ACTA}, w_5 = \mathrm{TAGTACT}, w_6 = \mathrm{AGTACTATAGT},$$

  and $w_7 = \mathrm{ACTATAGTACTAGTACTA}$. Thus, the word w_6 is bordered as well as θ-bordered, but the word w_7 is not θ-bordered.

- The first standard alternating θ-Fibonacci words are

  $$g_3 = \mathrm{AAC}, g_4 = \mathrm{GTTT}, \text{ and } g_5 = \mathrm{AAACAAC}.$$

  Note that the words g_i, for $3 \leq i \leq 5$, are neither bordered nor θ-bordered.

- The first standard hairpin θ-Fibonacci words are

  $$z_3 = \mathrm{TGT}, z_4 = \mathrm{TGTA}, z_5 = \mathrm{TGTAACA}, \text{ and } z_6 = \mathrm{TGTAACATACA}.$$

  Note that z_3 is bordered, but z_4, z_5, and z_6 are not bordered. Also, z_4, z_5, and z_6 are θ-bordered.

If, on the other hand, $\phi = \mu$ is a morphic involution:

- The first standard palindromic μ-Fibonacci words are

  $$w_3 = \mathrm{ATG}, w_4 = \mathrm{TACA}, w_5 = \mathrm{ATGTTAC}, \text{ and } w_6 = \mathrm{TACAATGATGT}.$$

  Thus, w_6 is both bordered and μ-bordered.

- The first standard alternating μ-Fibonacci words are

$$g_3 = \text{AAC}, g_4 = \text{TTGT}, g_5 = \text{AACAAAC}, \text{ and } g_6 = \text{TTGTTTGTTGT}.$$

  Note that the words g_5 and g_6 are bordered but not μ-bordered.

- The first standard hairpin μ-Fibonacci words are

$$z_3 = \text{TTG}, z_4 = \text{TTGA}, z_5 = \text{TTGAAAC}, \text{ and } z_6 = \text{TTGAAACAACT}.$$

  Note that z_4, z_5, and z_6 are μ-bordered, but z_4 and z_5 are not bordered.

Example 4.32 suggests the following result.

Theorem 4.33. *Let ϕ be an (anti)morphic involution on Σ^*. Then, for all $n \geq 6$, we have:*

- *If $\phi = \mu$, then the μ-Fibonacci words g_n, g'_n, w_n, w'_n, z_n, and z'_n are bordered.*
- *If $\phi = \mu$, then the μ-Fibonacci words w_n and w'_n are μ-bordered.*
- *If $\phi = \theta$, then the θ-Fibonacci words w_n and w'_n are bordered.*
- *The ϕ-Fibonacci words z_n and z'_n are ϕ-bordered.*

Proof. The statement follows from Definition 4.9, as one can easily infer the following. For a morphic involution μ, we have:

- For all $n \geq 4$, $g_n = g_{n-2}\mu(g_{n-3})g_{n-2}$ and $g'_n = g'_{n-2}\mu(g'_{n-3})g'_{n-2}$.
- For all $n \geq 4$, $w_n = w_{n-2}w_{n-3}\mu(w_{n-2})$ and $w'_n = \mu(w'_{n-2})w'_{n-3}w'_{n-2}$.
- For all $n \geq 6$, $z_n = z_{n-4}\mu(z_{n-5})\mu(z_{n-4})\mu(z_{n-3})\mu(z_{n-3})z_{n-4}$, and similarly, for all $n \geq 6$, $z'_n = z'_{n-4}\mu(z'_{n-3})\mu(z'_{n-3})\mu(z'_{n-4})\mu(z'_{n-5})z'_{n-4}$.
- For all $n \geq 6$, $w_n = w_{n-4}w_{n-5}\mu(w_{n-4})w_{n-3}w_{n-3}w_{n-4}$, and similarly, for all $n \geq 6$, $w'_n = w'_{n-4}w'_{n-3}w'_{n-3}\mu(w'_{n-4})w'_{n-5}w'_{n-4}$.

For an antimorphic involution θ, we have $w_n = w_{n-3}w_{n-2}w_{n-4}w_{n-3}$ and similarly $w'_n = w'_{n-3}w'_{n-4}w'_{n-2}w'_{n-3}$ for $n \geq 5$. For an (anti)morphic involution ϕ, we have $z_n = z_{n-2}\phi(z_{n-3})\phi(z_{n-2})$ and $z'_n = \phi(z'_{n-2})\phi(z'_{n-3})z'_{n-2}$ for $n \geq 4$. □

	$g_n = \phi(g_{n-1})g_{n-2}$		$z_n = z_{n-1}\phi(z_{n-2})$		$w_n = \phi(w_{n-1})\phi(w_{n-2})$	
	MI	AMI	MI	AMI	MI	AMI
bordered	True	False	True	False	True	True
ϕ-bordered	False	False	True	True	True	False

Table 4.4: The ϕ-borderedness of ϕ-Fibonacci words where ϕ is an involution extended to either a morphism (MI) or an antimorphism (AMI).

The results are summarized in Table 4.4.

From Theorem 4.33, we see that z_n is ϕ-bordered. In fact, for the case of an antimorphic involution θ, the following relations hold for all $n \geq 6$: the θ-borders of z_n are longer than z_{n-2} and $z_n = A_n z_r \theta(A_n)$ when n is odd, while $z_n = A_n \theta(z_r)\theta(A_n)$ if n is even, where $A_n = z_{i_1} z_{i_2} \cdots z_{i_{k-1}} z_{i_k}$, with $r = (n \bmod 3) + 3$, $i_1 = n - 2$, $i_t = i_{t-1} - 3$, and $i_k = r + 1$. A similar decomposition holds for z'_n. When θ is the Watson-Crick antimorphic involution on DNA strings, the property of θ-borderedness results in the DNA strings binding to themselves and forming hairpin structures (with fully double-stranded stems) [242], so we call z_n and z'_n "hairpin ϕ-Fibonacci words."

In cases where the borderedness or ϕ-borderedness does not hold in general, the following examples suggest that placing additional constraints on the initial words may ensure the borderedness or ϕ-borderedness of ϕ-Fibonacci words.

Example 4.34. Let ϕ be defined as in Example 4.32. We have the following:

- If $z_1 = \text{C}$, $z_2 = \text{AT}$, and $\phi = \theta$, then $z_3 = \text{ATG}$, $z_4 = \text{ATGAT}$, $z_5 = \text{ATGATCAT}$, and $z_6 = \text{ATGATCATATCAT}$. Note that z_4, z_5, and z_6 are bordered. The condition is $\text{pref}'(z_2) \cap \text{suff}'(\theta(z_2)) \neq \emptyset$.

- If $g_1 = \text{A}$, $g_2 = \text{CT}$, and $\phi = \theta$, then $g_3 = \text{AGA}$, $g_4 = \text{TCTCT}$, $g_5 = \text{AGAGAAGA}$, and $g_6 = \text{TCTTCTCTTCTCT}$. Note that g_3, g_4, g_5, and g_6 are bordered. The condition is $\text{pref}'(\theta(g_1)) \cap \text{suff}'(g_2) \neq \emptyset$.

- If $g_1 = \text{A}$, $g_2 = \text{CA}$, and $\phi = \theta$, then $g_3 = \text{TGA}$, $g_4 = \text{TCACA}$, $g_5 = \text{TGTGATGA}$, and $g_6 = \text{TCATCACATCACA}$. Note that g_3, g_4, g_5, and g_6 are θ-bordered. The condition is $\text{suff}'(g_1) \cap \text{suff}'(g_2) \neq \emptyset$.

- If $g_1 = \text{AT}$, $g_2 = \text{TGA}$, and $\phi = \mu$, then $g_3 = \text{ACTAT}$, $g_4 = \text{TGATATGA}$, $g_5 = \text{ACTATACTACTAT}$, and $g_6 = \text{TGATATGATGATATGATATGA}$. Note that g_2, g_3, g_4, g_5, and g_6 are μ-bordered. The condition is $\text{suff}'(g_1) \cap \text{pref}'(g_2) \neq \emptyset$.

- If $w_1 = \text{CA}$, $w_2 = \text{CTG}$, and $\phi = \theta$, then

  $$w_3 = \text{CAGTG}, w_4 = \text{CACTGCAG}, w_5 = \text{CTGCAGTGCACTG},$$

  and $w_6 = \text{CAGTGCACTGCAGCTGCAGTG}$. Note that w_2, w_3, w_4, w_5, and w_6 are θ-bordered. Note that the language $\text{pref}'(w_2) \cap \text{suff}'(\theta(w_1))$ is non-empty, and so are $\text{pref}'(w_1) \cap \text{pref}'(w_2)$ and $\text{pref}'(\theta(w_1)) \cap \text{suff}'(w_2)$.

We will now prove that the observations inferred from Example 4.34 hold in general (Proposition 4.36). We use the following Lemma.

Lemma 4.35. *Let u and v be two words in Σ^+, let $g_1 = g'_1 = w_1 = w'_1 = z_1 = z'_1 = u$, and let $g_2 = g'_2 = w_2 = w'_2 = z_2 = z'_2 = v$. If $\phi = \theta$ is an antimorphic involution on Σ^*, then the following relations hold for all $n \geq 3$ and some $t_n, t'_n \in \Sigma^*$:*

$$g_n = \begin{cases} \theta(u)t_n v : n \bmod 2 = 0, \\ \theta(v)t_n u : n \bmod 2 = 1, \end{cases} \qquad g'_n = \begin{cases} vt'_n\theta(u) : n \bmod 2 = 0, \\ ut'_n\theta(v) : n \bmod 2 = 1, \end{cases}$$

$$w_n = \begin{cases} \theta(v)t_n\theta(u) : n \bmod 3 = 0, \\ ut_n\theta(v) : n \bmod 3 = 1, \\ vt_n v : n \bmod 3 = 2, \end{cases} \qquad w'_n = \begin{cases} \theta(u)t'_n\theta(v) : n \bmod 3 = 0, \\ \theta(v)t'_n u : n \bmod 3 = 1, \\ vt'_n v : n \bmod 3 = 2, \end{cases}$$

$$z_n = vt_n\theta(v), \qquad z'_n = \theta(v)t'_n v : n \geq 4.$$

If, on the other hand, $\phi = \mu$ is a morphic involution on Σ^*, then the following relations hold for all $n \geq 3$ and some $t_n, t'_n \in \Sigma^*$:

$$g_n = \begin{cases} vt_n v : n \bmod 2 = 0, \\ \mu(v)t_n u : n \bmod 2 = 1, \end{cases} \qquad g'_n = \begin{cases} vt'_n v : n \bmod 2 = 0, \\ ut'_n\mu(v) : n \bmod 2 = 1. \end{cases}$$

Proof. We only prove for the θ-Fibonacci word g_n for n odd. Note that the first standard alternating θ-Fibonacci words are $g_1 = u$, $g_2 = v$, $g_3 = \theta(v)u$, and $g_4 = \theta(u)vv$. Assume true for all $k \leq n$. Consider the word g_{n+1} for $n+1$ an even number. Then by strong induction, we have

$$g_{n+1} = \theta(g_n)g_{n-1} = \theta(\theta(v)t_n u)\theta(u)t_{n-1}v = \theta(u)\theta(t_n)v\theta(u)t_{n-1}v = \theta(u)t_{n+1}v,$$

hence the result. The other cases can be proved in a similar fashion. □

It is clear from Theorem 4 in [49] that if $f_1 = u$ and $f_2 = v$, then the Fibonacci words f_n are of the form vut_nvu when n is odd and of the form vs_nv when n is even, and hence are bordered for all $n \geq 4$. One can clearly see that if ϕ is an (anti)morphic involution such that $\phi(\text{pref}'(y)) \cap \text{suff}'(x) \neq \emptyset$ and $\phi(\text{pref}'(y)) \cap \text{suff}'(y) \neq \emptyset$, then each f_n is ϕ-bordered.

We now use Lemma 4.35 to provide conditions on the initial words, which are sufficient to ensure that the θ-Fibonacci words g_n and w_n are θ-bordered, that the μ-Fibonacci words g_n are μ-bordered, and that the θ-Fibonacci words z_n and g_n are bordered for all $n \geq 4$.

Proposition 4.36. *Let $\phi = \theta$ be an antimorphic involution on Σ^*, let $u, v \in \Sigma^+$, and let $g_1 = w_1 = z_1 = u$ and $g_2 = w_2 = z_2 = v$. The following relations hold for all $n \geq 3$:*

- *If $\text{pref}'(v) \cap \text{suff}'(\theta(v)) \neq \emptyset$, then z_n is bordered.*
- *If $\text{pref}'(\theta(u)) \cap \text{suff}'(v) \neq \emptyset$, then g_n is bordered.*
- *If $\text{suff}'(u) \cap \text{suff}'(v) \neq \emptyset$, then g_n is θ-bordered.*
- *If $\text{pref}'(\theta(u)) \cap \text{suff}'(v) \neq \emptyset$, $\text{pref}'(v) \cap \text{suff}'(\theta(u)) \neq \emptyset$, $\text{pref}'(u) \cap \text{pref}'(v) \neq \emptyset$, then w_n is θ-bordered.*

If, on the other hand, $\phi = \mu$ is a morphic involution on Σ^* and $\text{suff}'(u) \cap \text{pref}'(v) \neq \emptyset$ and v is μ-bordered, then g_n is μ-bordered for $n \geq 4$.

Proof. It follows directly from Lemma 4.35 and the definition of ϕ-borders. □

Note that the initial words α_1 and α_2 given in Example 4.34 satisfy the conditions in Proposition 4.36. Also, Proposition 4.36 implies the following, for the case of the atom θ-Fibonacci words.

Corollary 4.37. *Let $\phi = \theta$ be an antimorphic involution on Σ^*, let a, b be two letters in Σ, and let $g_1 = z_1 = w_1 = a$ and $g_2 = z_2 = w_2 = b$. Then for all $n \geq 4$, we have:*

- *If $b = \theta(b)$, then z_n is θ-bordered.*
- *If $\theta(a) = b$, then g_n is bordered.*
- *If $a = b$, then g_n is θ-bordered, and in addition if $\theta(a) = b$, then w_n is θ-bordered.*

Lastly, we present a property of w_n for the case of an antimorphic involution, which justifies their being called "palindromic Fibonacci words."

Definition 4.38. *Let θ be an antimorphic involution on Σ^*, and define*

$$P_\theta = \{w \in \Sigma^+ \mid w = \theta(w)\}, \text{ and } P_{2\theta} = \{w\theta(w) \mid w \in \Sigma^+\}.$$

We call a string x a θ-palstar if it belongs to $P_{2\theta}^*$. A non-empty θ-palstar is said to be prime θ-palstar if it cannot be written as a concatenation of two or more θ-palstars.

Note that in the particular case when θ is the mirror image, the definition above becomes the well-known definitions of palstar and prime palstar that were introduced in [160]. Note that each string in $P_{2\theta}$ is a θ-palindrome of even length, and conversely, $P_{2\theta} = \{x \in P_\theta \mid 2k = |x| \text{ for some } k\}$. By repeated decompositions, one can show that every θ-palstar is expressible as a concatenation of prime θ-palstars. One can also prove that such a decomposition of a θ-palstar into prime θ-palstars is unique, which is a consequence of the following Lemma.

Lemma 4.39. *For an antimorphic involution θ on Σ^*, a prime θ-palstar cannot begin with another prime θ-palstar.*

Proof. It can be proved similarly to the known fact that a prime palstar cannot begin with another prime palstar [160]. □

The following result shows that θ-Fibonacci words w_n and w'_n can be expressed in two different ways as a θ-palstar concatenated with a θ-Fibonacci word w_i (respectively w'_i). This justifies the ϕ-Fibonacci words w_n being called "palindromic-Fibonacci words."

Proposition 4.40. *Given an antimorphic involution $\phi = \theta$ on Σ^*, for all $n \geq 6$, the word w_n can be decomposed as $w_n = A_n w_r = w_r B_n$, where A_n, B_n are the θ-palstars*

$$A_n = p_{i_1} q_{j_1} p_{i_2} q_{j_2} \cdots p_{i_{k-1}} q_{j_{k-1}} p_{i_k} q_{j_k}, B_n = p_{s_1} p_{s_2} \cdots p_{s_{l-1}} p_{s_l},$$

such that $p_i = w_i\theta(w_i)$, $q_i = \theta(w_i)w_i$, $j_t = i_t - 1$, $i_t = i_{t-1} - 3$, $i_1 = n-3$, $i_k = r$, $s_1 = r+1$, $s_l = n-2$, $s_t = s_{t-1} + 3$, and $r = (n \bmod 3) + 3$.

Proof. We prove the result by induction on n. For the base case, let $n = 6$, which gives us $w_6 = \theta(w_5)\theta(w_4) = w_3 w_4 w_2 w_3 = w_3\theta(w_3)\theta(w_2)w_2w_3$. Assume that,

$$w_k = p_{k-3}q_{k-4}p_{k-6}q_{k-7}\cdots p_{r+3}q_{r+2}p_r q_{r-1} w_r,$$

for $r = (k \bmod 3) + 3$ and for all $k \leq n$. Consider w_{k+1}. Then, by definition, we have

$$w_{k+1} = \theta(w_k)\theta(w_{k-1}) = w_{k-2}w_{k-1}w_{k-3}w_{k-2} = w_{k-2}\theta(w_{k-2})\theta(w_{k-3})w_{k-3}w_{k-2}.$$

Hence, we have by induction that $w_{k+1} = p_{k-2}q_{k-3}p_{k-5}q_{k-6}\cdots p_r q_{r-1} w_r$. The proof for the other equality is similar. □

For the reverse palindromic θ-Fibonacci words, the following result, similar to Proposition 4.40, holds.

Proposition 4.41. *Let $\phi = \theta$ be an antimorphic involution on Σ^*. Then, for all $n \geq 6$, the word w'_n can be decomposed as $w'_n = C_n w'_r = w'_r D_n$, where C_n, D_n are the θ-palstars*

$$C_n = q'_{i_1} q'_{i_2} \cdots q'_{i_{k-1}} q'_{i_k}, D_n = p'_{j_1} q'_{s_1} p'_{j_2} q'_{s_2} \cdots p'_{j_{k-1}} q'_{s_{k-1}} p'_{j_k} q'_{s_k},$$

such that $p'_i = w'_i \theta(w'_i), q'_i = \theta(w'_i) w'_i$, $i_1 = n-2$, $i_t = i_{t-1} - 3$, $i_k = r+1$, and $j_1 = r-1$, $j_k = n-3$, $s_t = j_t + 1$, $j_t = j_{t-1} + 3$, and $r = (n \bmod 3) + 3$.

Proof. Similar to that of Proposition 4.40, by induction on n. □

4.6 Primitivity of atom ϕ-Fibonacci words with different initial letters

In this section, we discuss the primitivity of atom ϕ-Fibonacci words $\alpha_n^{\phi}(a, b)$ with different initial letters $a, b \in \Sigma_4$, for all $n \geq 1$, where ϕ is an (anti)morphic involution on Σ_4^* and $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \geq 1$. We first show that if we have an alphabet $\Sigma_4 = \{x_1, x_2, x_3, x_4\}$ and a sequence (x_1, x_2, x_3, x_4), if we choose x_1 and x_2 as the two initial letters of the ϕ-Fibonacci sequence $\{\alpha_n^{\phi}(x_1, x_2)\}_{n \geq 1}$, it is enough to discuss the primitivity properties of atom ϕ-Fibonacci words for the mappings ϕ_1, ϕ_2, ϕ_4, ϕ_5, and ϕ_{10}.

Note that if $\phi = \mu_1$, the identity function on Σ_4^*, and $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \geq 1$, the atom ϕ-Fibonacci words $\alpha_n^{\mu_1}(x_1, x_2)$ coincide with the classical Fibonacci words $f_n(x_1, x_2)$, which are primitive when $x_1, x_2 \in \Sigma_4$ for all $n \geq 1$ as shown in [60].

Lastly, note that the proofs for the results of this section hold for every choice of (x_1, x_2, x_3, x_4) and with x_1 and x_2 as the initial letters of the ϕ-Fibonacci sequence. This justifies stating/proving the subsequent results for only one of the cases (usually the sequence [A, C, G, T] with A and C as the two initial letters).

In the sequel, we let $[x]_{a \to b}$ denote the word obtained from x by replacing all occurrences of a in x by b and let $[x]_{a \rightleftarrows b}$ denote the word obtained from x by replacing all occurrences of a by b and all occurrences of b by a. For example, if $x = abbab$ then $[x]_{a \to b} = [abbab]_{a \to b} = bbbbb$ and $[x]_{a \rightleftarrows b} = [abbab]_{a \rightleftarrows b} = baaba$. We first observe the following.

Lemma 4.42. *Let $a, b \in \Sigma$ and $x \in \Sigma^*$. We have that:*

- $[x]_{a \rightleftarrows b} = [x]_{b \rightleftarrows a}$.
- *If* $b \notin \mathrm{Alph}(x)$*, we have that* $[x]_{a \to b} = [x]_{a \rightleftarrows b}$.
- *If* $b \notin \mathrm{Alph}(x)$*, we have* $[[x]_{a \to b}]_{b \to a} = x$.

Lemma 4.43. *Let* $a, b \in \Sigma$, $x \in \Sigma^*$, *and* $i \geq 0$. *We have that* $[x^i]_{a \to b} = ([x]_{a \to b})^i$ *and* $[x^i]_{a \rightleftarrows b} = ([x]_{a \rightleftarrows b})^i$.

Lemma 4.44. *Let* $a, b \in \Sigma$ *and* $x \in \Sigma^*$. *We have that:*

1. *If* $b \notin \mathrm{Alph}(x)$*, then* x *is primitive if and only if* $[x]_{a \to b}$ *is primitive.*
2. *If* $a, b \in \mathrm{Alph}(x)$*, then* x *is primitive if and only if* $[x]_{a \rightleftarrows b}$ *is primitive.*

Proof. For statement (1), if $a \notin \mathrm{Alph}(x)$, then $x = [x]_{a \to b}$, so x is primitive if and only if $[x]_{a \to b}$ is primitive. Let $a \in \mathrm{Alph}(x)$. Assume x is not primitive but $[x]_{a \to b}$ is primitive. Since x is not primitive, we have $x = q^i$, where $q \in Q$ and $i \geq 2$. By Lemma 4.43, we have $[x]_{a \to b} = [q^i]_{a \to b} = ([q]_{a \to b})^i$, which is a contradiction. The case where x is primitive but $[x]_{a \to b}$ is not primitive can be proved similarly. Therefore, x is primitive if and only if $[x]_{a \to b}$ is primitive. The statement (2) can be proved similarly using Lemma 4.43. □

Theorem 4.45. *Let* $\phi_i \in \{\mu_i, \theta_i\}$, $1 \leq i \leq 10$, *be an (anti)morphic involution on* Σ_4^* *and* $\alpha_n \in \{g_n, w_n, z_n\}$ *for* $n \geq 1$. *For* $n \geq 1$, *the following statements hold for* ϕ_i*-Fibonacci words* $\alpha_n^{\phi_i}(x_1, x_2)$ *:*

1. $[\alpha_n^{\phi_4}(x_1, x_2)]_{x_4 \to x_3}$ *is primitive if and only if* $\alpha_n^{\phi_4}(x_1, x_2)$ *is primitive.*
2. $[\alpha_n^{\phi_5}(x_1, x_2)]_{x_3 \to x_4}$ *is primitive if and only if* $\alpha_n^{\phi_5}(x_1, x_2)$ *is primitive.*
3. $[\alpha_n^{\phi_{10}}(x_1, x_2)]_{x_4 \rightleftarrows x_3}$ *is primitive if and only if* $\alpha_n^{\phi_{10}}(x_1, x_2)$ *is primitive.*

Proof.

1. Let $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \geq 1$. One can easily prove by induction that, for all $n \geq 1$, we have $x_3 \notin \mathrm{Alph}(\alpha_n^{\phi_4}(x_1, x_2))$. Hence by Lemma 4.44, the statement holds.
2. Let $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \geq 1$. For all $n \geq 1$, one can prove by induction that $x_4 \notin \mathrm{Alph}(\alpha_n^{\phi_5}(x_1, x_2))$. Hence by Lemma 4.44, the statement holds.

3. We have two cases. The first one is when $\alpha_n \in \{w_n, z_n\}$ for all $n \geq 1$. In this case, if $1 \leq n \leq 4$, the statement can be easily verified. If $n \geq 5$, then one can easily prove by induction that $x_3, x_4 \in \mathrm{Alph}(\alpha_n^{\phi_{10}}(x_1, x_2))$. Thus, the statement holds by Lemma 4.44.

   The second case is when $\alpha_n = g_n$ for all $n \geq 1$. In this case, one can first prove by induction that, for all $n \geq 1$, we have $x_3 \notin \mathrm{Alph}(g_n^{\phi_{10}}(x_1, x_2))$ if n is even and $x_4 \notin \mathrm{Alph}(g_n^{\phi_{10}}(x_1, x_2))$ if n is odd. Using this fact, if n is even then by Lemma 4.42, we have $[g_n^{\phi_{10}}(x_1, x_2)]_{x_4 \rightleftarrows x_3} = [g_n^{\phi_{10}}(x_1, x_2)]_{x_3 \rightleftarrows x_4} = [g_n^{\phi_{10}}(x_1, x_2)]_{x_3 \to x_4}$. Also if n is even, then by Lemma 4.44, we have that $[g_n^{\phi_{10}}(x_1, x_2)]_{x_4 \rightleftarrows x_3} = [g_n^{\phi_{10}}(x_1, x_2)]_{x_3 \to x_4}$ is primitive if and only if $g_n^{\phi_{10}}(x_1, x_2)$ is primitive. The case where n is odd can be proved similarly.

□

We now show the equivalence of various ϕ-Fibonacci words $\alpha_n^{\phi}(x_1, x_2)$ for certain values of $\phi \in \{\phi_i \mid 1 \leq i \leq 10\}$. We use the following lemma, which can be proved easily by induction on n.

Lemma 4.46. *Let $\phi_i \in \{\mu_i, \theta_i\}$, $1 \leq i \leq 10$, be an (anti)morphic involution on Σ_4^* and $\alpha_n \in \{g_n, w_n, z_n\}$ for $n \geq 1$. The following equalities hold for ϕ_i-Fibonacci words $\alpha_n^{\phi_i}(x_1, x_2)$ and for all $n \geq 1$:*

1. $\phi_1(\alpha_n^{\phi_1}(x_1, x_2)) = \phi_7(\alpha_n^{\phi_7}(x_1, x_2))$ *if and only if* $\alpha_n^{\phi_1}(x_1, x_2) = \alpha_n^{\phi_7}(x_1, x_2)$.
2. $\phi_2(\alpha_n^{\phi_2}(x_1, x_2)) = \phi_8(\alpha_n^{\phi_8}(x_1, x_2))$ *if and only if* $\alpha_n^{\phi_2}(x_1, x_2) = \alpha_n^{\phi_8}(x_1, x_2)$.
3. $[\phi_4(\alpha_n^{\phi_4}(x_1, x_2))]_{x_4 \to x_3} = \phi_3(\alpha_n^{\phi_3}(x_1, x_2))$ *if and only if* $[\alpha_n^{\phi_4}(x_1, x_2)]_{x_4 \to x_3} = \alpha_n^{\phi_3}(x_1, x_2)$.
4. $[\phi_5(\alpha_n^{\phi_5}(x_1, x_2))]_{x_3 \to x_4} = \phi_6(\alpha_n^{\phi_6}(x_1, x_2))$ *if and only if* $[\alpha_n^{\phi_5}(x_1, x_2)]_{x_3 \to x_4} = \alpha_n^{\phi_6}(x_1, x_2)$.
5. $[\phi_{10}(\alpha_n^{\phi_{10}}(x_1, x_2))]_{x_4 \rightleftarrows x_3} = \phi_9(\alpha_n^{\phi_9}(x_1, x_2))$ *if and only if*
$$[\alpha_n^{\phi_{10}}(x_1, x_2)]_{x_4 \rightleftarrows x_3} = \alpha_n^{\phi_9}(x_1, x_2).$$

Proof. We only prove statement (1) by induction, and it is sufficient to prove it for one of $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \geq 1$ since the other cases are similar. Let ϕ_1 be a morphic involution, and, without loss of generality, let $\alpha_n = z_n$ for all $n \geq 1$. By Definition 4.9 and Table 4.1, the result holds for $n = 1$ and $n = 2$. For the inductive step, assume that $\phi_1(z_i^{\phi_1}(x_1, x_2)) = \phi_7(z_i^{\phi_7}(x_1, x_2))$ if and only if $z_i^{\phi_1}(x_1, x_2) = z_i^{\phi_7}(x_1, x_2)$ for all $3 \leq i < k$. We now have to prove that the equivalence holds for k.

For the direct implication, assume $\phi_1(z_k^{\phi_1}(x_1,x_2)) = \phi_7(z_k^{\phi_7}(x_1,x_2))$. Therefore, we have $\phi_1(z_k^{\phi_1}(x_1,x_2)) = \phi_1(z_{k-1}^{\phi_1}(x_1,x_2)\phi_1(z_{k-2}^{\phi_1}(x_1,x_2))) = \phi_1(z_{k-1}^{\phi_1}(x_1,x_2))z_{k-2}^{\phi_1}(x_1,x_2)$, and similarly we have $\phi_7(z_k^{\phi_7}(x_1,x_2)) = \phi_7(z_{k-1}^{\phi_7}(x_1,x_2))z_{k-2}^{\phi_7}(x_1,x_2)$. By $\phi_1(z_k^{\phi_1}(x_1,x_2)) = \phi_7(z_k^{\phi_7}(x_1,x_2))$, and the fact that $|z_{k-2}^{\phi_7}(x_1,x_2)| = |z_{k-2}^{\phi_1}(x_1,x_2)|$, we have $z_{k-2}^{\phi_7}(x_1,x_2) = z_{k-2}^{\phi_1}(x_1,x_2)$ and $\phi_7(z_{k-1}^{\phi_7}(x_1,x_2)) = \phi_1(z_{k-1}^{\phi_1}(x_1,x_2))$. Therefore, we have $z_k^{\phi_1}(x_1,x_2) = z_{k-1}^{\phi_1}(x_1,x_2)\phi_1(z_{k-2}^{\phi_1}(x_1,x_2)) = z_{k-1}^{\phi_7}(x_1,x_2)\phi_7(z_{k-2}^{\phi_7}(x_1,x_2)) = z_k^{\phi_7}(x_1,x_2)$ by the inductive hypothesis.

For the converse implication, assume $z_k^{\phi_1}(x_1,x_2) = z_k^{\phi_7}(x_1,x_2)$. Therefore, we have $z_k^{\phi_1}(x_1,x_2) = z_{k-1}^{\phi_1}(x_1,x_2)\phi_1(z_{k-2}^{\phi_1}(x_1,x_2))$, and similarly we have

$$z_k^{\phi_7}(x_1,x_2) = z_{k-1}^{\phi_7}(x_1,x_2)\phi_7(z_{k-2}^{\phi_7}(x_1,x_2)).$$

By $z_k^{\phi_1}(x_1,x_2) = z_k^{\phi_7}(x_1,x_2)$, and the fact that $|z_{k-1}^{\phi_7}(x_1,x_2)| = |z_{k-1}^{\phi_1}(x_1,x_2)|$, we have $z_{k-1}^{\phi_7}(x_1,x_2) = z_{k-1}^{\phi_1}(x_1,x_2)$ and $\phi_7(z_{k-2}^{\phi_7}(x_1,x_2)) = \phi_1(z_{k-2}^{\phi_1}(x_1,x_2))$. Therefore, we have

$$\phi_1(z_k^{\phi_1}(x_1,x_2)) = \phi_1(z_{k-1}^{\phi_1}(x_1,x_2)\phi_1(z_{k-2}^{\phi_1}(x_1,x_2))) = \phi_1(z_{k-1}^{\phi_1}(x_1,x_2))z_{k-2}^{\phi_1}(x_1,x_2)$$

$$= \phi_7(z_{k-1}^{\phi_7}(x_1,x_2))z_{k-2}^{\phi_7}(x_1,x_2) = \phi_7(z_{k-1}^{\phi_7}(x_1,x_2)\phi_7(z_{k-2}^{\phi_1}(x_1,x_2))) = \phi_7(z_k^{\phi_7}(x_1,x_2))$$

by the inductive hypothesis.

The case when ϕ_1 is an antimorphic involution is similar. □

We now prove the following.

Lemma 4.47. *Let $\phi_i \in \{\mu_i, \theta_i\}$, $1 \le i \le 10$, be an (anti)morphic involution on Σ_4^* and $\alpha_n \in \{g_n, w_n, z_n\}$ for $n \ge 1$. For all $n \ge 1$, the following equalities hold for ϕ_i-Fibonacci words $\alpha_n^{\phi_i}(x_1,x_2)$:*

1. $\alpha_n^{\phi_1}(x_1,x_2) = \alpha_n^{\phi_7}(x_1,x_2)$.
2. $\alpha_n^{\phi_2}(x_1,x_2) = \alpha_n^{\phi_8}(x_1,x_2)$.
3. $[\alpha_n^{\phi_4}(x_1,x_2)]_{x_4\to x_3} = \alpha_n^{\phi_3}(x_1,x_2)$.
4. $[\alpha_n^{\phi_5}(x_1,x_2)]_{x_3\to x_4} = \alpha_n^{\phi_6}(x_1,x_2)$.
5. $[\alpha_n^{\phi_{10}}(x_1,x_2)]_{x_4\rightleftarrows x_3} = \alpha_n^{\phi_9}(x_1,x_2)$.

Proof. We only prove statement (1) by induction, as the other cases are similar. By Definition 4.9, the result holds for $n = 1$ and $n = 2$. We assume $\alpha_i^{\phi_1}(x_1, x_2) = \alpha_i^{\phi_7}(x_1, x_2)$ holds for $3 \le i < k$. It is enough to prove for one of $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \ge 1$. Without loss of generality, let $\alpha_n = z_n$ for all $n \ge 1$. Then, $\alpha_k^{\phi_1}(x_1, x_2) = z_k^{\phi_1}(x_1, x_2) = z_{k-1}^{\phi_1}(x_1, x_2)\phi_1(z_{k-2}^{\phi_1}(x_1, x_2))$. By the inductive hypothesis and by Lemma 4.46 we have

$$z_k^{\phi_1}(x_1, x_2) = z_{k-1}^{\phi_7}(x_1, x_2)\phi_7(z_{k-2}^{\phi_7}(x_1, x_2)) = z_k^{\phi_7}(x_1, x_2).$$

Hence, the result. □

From Corollary 4.13, Corollary 4.15, Corollary 4.17, and the fact that words of length 1 are palindromes, we have the following observation.

Lemma 4.48. *Let ϕ be an (anti)morphic involution on Σ^*, $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \ge 1$, and $a, b \in \Sigma$. For $n \ge 1$, the atom standard ϕ-Fibonacci word $\alpha_n^{\phi}(a, b)$ is primitive if and only if the atom reverse ϕ-Fibonacci word $[\alpha_n^{\phi}(a, b)]'$ is primitive.*

As a consequence of Theorem 4.45, Lemma 4.47, and Lemma 4.48, we only need to study the primitivity of atom ϕ-Fibonacci words $\alpha_n^{\phi}(x_1, x_2)$ for all $n \ge 1$, where $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \ge 1$ and $\phi \in \{\theta_1, \theta_2, \theta_4, \theta_5, \theta_{10}, \mu_2, \mu_4, \mu_5, \mu_{10}\}$.

Note that the results obtained above hold for every choice of (x_1, x_2, x_3, x_4) and with x_1 and x_2 as the initial letters of the ϕ-Fibonacci sequence. Therefore, in the remainder of this section, we will only prove primitivity results about one of the cases, namely the sequence $(\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T})$ over the DNA alphabet Δ, with A and C as the two initial letters.

4.6.1 Atom alternating ϕ-Fibonacci words

We first discuss the primitivity of atom alternating ϕ-Fibonacci words g_n for $n \ge 1$. In Table 4.5, we give the first few values of the sequences $\{g_n^{\phi}(\mathrm{A}, \mathrm{C})\}_{n \ge 1}$ for (anti)morphic involutions $\phi \in \{\mu_1, \mu_2, \mu_4, \mu_5, \mu_{10}, \theta_1, \theta_2, \theta_4, \theta_5, \theta_{10}\}$.

Using Lemma 4.10, we prove the following theorem. Note that the cyclic shift by one position from the front of a word $x \in \Sigma^*$ to the end of it can be represented by a composition of (left) derivative, concatenation, and finite union, that is, by $\bigcup_{a \in \Sigma}(\partial_a x)a$.

Theorem 4.49. *Let $\phi = \theta_1$. For all $n \ge 1$, the atom ϕ-Fibonacci word $g_n^{\theta_1}(\mathrm{A}, \mathrm{C})$ is a conjugate of $f_n'(\mathrm{A}, \mathrm{C})$. More precisely,*

$$g_n^{\theta_1} = \begin{cases} (\partial_{\mathrm{C}} f_n')\mathrm{C} : n \bmod 2 = 0, \\ (\partial_{\mathrm{A}} f_n')\mathrm{A} : n \bmod 2 = 1. \end{cases}$$

ϕ	$g_3^{\phi}(\mathrm{A},\mathrm{C})$	$g_4^{\phi}(\mathrm{A},\mathrm{C})$	$g_5^{\phi}(\mathrm{A},\mathrm{C})$	$g_6^{\phi}(\mathrm{A},\mathrm{C})$	$g_7^{\phi}(\mathrm{A},\mathrm{C})$
μ_1	CA	CAC	CACCA	CACCACAC	CACCACACCACCA
μ_2	AA	CCC	AAAAA	CCCCCCCC	AAAAAAAAAAAAA
μ_4	CA	CTC	CACCA	CTCCTCTC	CACCACACCACCA
μ_5	GA	CAC	GAGGA	CACCACAC	GAGGAGAGGAGGA
μ_{10}	GA	CTC	GAGGA	CTCCTCTC	GAGGAGAGGAGGA
θ_1	CA	ACC	CCACA	ACACCACC	CCACCACACCACA
θ_2	AA	CCC	AAAAA	CCCCCCCC	AAAAAAAAAAAAA
θ_4	CA	TCC	CCACA	TCTCCTCC	CCACCACACCACA
θ_5	GA	ACC	GGAGA	ACACCACC	GGAGGAGAGGAGA
θ_{10}	GA	TCC	GGAGA	TCTCCTCC	GGAGGAGAGGAGA

Table 4.5: List of words $g_n^{\phi}(\mathrm{A},\mathrm{C})$, where $3 \leq n \leq 7$ and $\phi \in \{\mu_1, \mu_2, \mu_4, \mu_5, \mu_{10}, \theta_1, \theta_2, \theta_4, \theta_5, \theta_{10}\}$.

Proof. For $1 \leq n \leq 7$, this can be easily checked from Table 4.5. Assume the statement holds for $g_i^{\theta_1}$, where $7 \leq i < k$. We now prove this for g_k. Without loss of generality, let k be even. Then, by definition and Lemma 4.10, we have

$$\begin{aligned}
g_k^{\theta_1} &= \theta_1(g_{k-1}^{\theta_1})g_{k-2}^{\theta_1} = (g_{k-1}^{\theta_1})^r g_{k-2}^{\theta_1} = ((\partial_{\mathrm{A}} f'_{k-1})\mathrm{A})^r \cdot ((\partial_{\mathrm{C}} f'_{k-2})\mathrm{C}) \\
&= (((\partial_{\mathrm{A}} f'_{k-3} f'_{k-2}))\mathrm{A})^r \cdot ((\partial_{\mathrm{C}} f'_{k-2})\mathrm{C}) = \mathrm{A}(\mathrm{C}s_{k-3}\mathrm{CA}s_{k-2})^r \mathrm{A}s_{k-2}\mathrm{C} \\
&= \mathrm{A}s_{k-2}\mathrm{AC}s_{k-3}\mathrm{CA}s_{k-2}\mathrm{C} = \mathrm{A}s_{k-2}(f'_{k-3}f'_{k-2})\mathrm{C} = \mathrm{A}s_{k-2}f'_{k-1}\mathrm{C} \\
&= (\partial_{\mathrm{C}}\mathrm{CA}s_{k-2}f'_{k-1})\mathrm{C} = (\partial_{\mathrm{C}} f'_{k-2}f'_{k-1})\mathrm{C} = (\partial_{\mathrm{C}} f'_k)\mathrm{C}.
\end{aligned}$$

□

By Theorem 4.5 and Lemma 4.6, we have the following corollary.

Corollary 4.50. *Let $\phi = \theta_1$. For all $n \geq 1$, the atom ϕ-Fibonacci word $g_n^{\theta_1}(\mathrm{A},\mathrm{C})$ is primitive.*

We now use the following lemma which is a generalized version of Lemma 4.20 when ϕ is a morphic involution. We show that the result also holds when ϕ is an antimorphic involution.

Lemma 4.51. *Let ϕ be an (anti)-morphic involution on Σ^*, let μ_a be a morphic involution on Σ^* such that $\mu_a\phi = \phi\mu_a$, and let $u, v \in \Sigma^+$. If $\alpha_n \in \{f_n, g_n, w_n, z_n\}$ for all $n \geq 1$, then for all $n \geq 1$, we have $\mu_a(\alpha_n^{\phi}(u,v)) = \alpha_n^{\phi}(\mu_a(u), \mu_a(v))$, and $\mu_a([\alpha_n^{\phi}(u,v)]') = [\alpha_n^{\phi}(\mu_a(u), \mu_a(v))]'$.*

Proof. We consider the standard ϕ-Fibonacci words g_n. The proof is by induction on n. By definition of g_n (Definition 4.9), we have $\mu_a(g_1(u,v)) = \mu_a(u) = g_1(\mu_a(u), \mu_a(v))$, and $\mu_a(g_2(u,v)) = \mu_a(v) = g_2(\mu_a(u), \mu_a(v))$, so the base case holds. Assume that $\mu_a(g_i(u,v)) = g_i(\mu_a(u), \mu_a(v))$ for all $1 \le i \le k$. Using the definition of g_n (Definition 4.9), the fact that μ_a is a morphism, and the induction hypothesis, we have

$$\begin{aligned}\mu_a(g_{k+1}(u,v)) &= \mu_a(\phi(g_k(u,v)) \cdot g_{k-1}(u,v)) \\ &= \phi(\mu_a(g_k(u,v))) \cdot \mu_a(g_{k-1}(u,v)) \\ &= \phi(g_k(\mu_a(u), \mu_a(v))) \cdot g_{k-1}(\mu_a(u), \mu_a(v)) \\ &= g_{k+1}(\mu_a(u), \mu_a(v)).\end{aligned}$$

The proofs for other ϕ-Fibonacci words are similar. □

The following result can be proved by induction and Lemma 4.51. We first observe that $\theta_i = \theta_1\mu_i = \mu_i\theta_1$ for $1 \le i \le 10$.

Lemma 4.52. *For $i \in \{4, 5, 10\}$ and for all $n \ge 1$, the following relations between the atom alternating ϕ-Fibonacci words $g_n^{\theta_1}$ and the atom alternating ϕ-Fibonacci words $g_n^{\theta_i}(\mathrm{A},\mathrm{C})$ hold.*

1. *If $i = 10$, for $n \ge 1$, we have that*

$$g_n^{\theta_{10}}(\mathrm{A},\mathrm{C}) = \begin{cases} g_n^{\theta_1}(\mathrm{T},\mathrm{C}) : n \bmod 2 = 0, \\ g_n^{\theta_1}(\mathrm{A},\mathrm{G}) : n \bmod 2 = 1. \end{cases}$$

2. *If $i = 5$, for $n \ge 1$, we have that*

$$g_n^{\theta_5}(\mathrm{A},\mathrm{C}) = \begin{cases} g_n^{\theta_1}(\mathrm{A},\mathrm{C}) : n \bmod 2 = 0 \\ g_n^{\theta_{10}}(\mathrm{A},\mathrm{C}) : n \bmod 2 = 1 \end{cases} = \begin{cases} g_n^{\theta_1}(\mathrm{A},\mathrm{C}) : n \bmod 2 = 0, \\ g_n^{\theta_1}(\mathrm{A},\mathrm{G}) : n \bmod 2 = 1. \end{cases}$$

3. *If $i = 4$, for $n \ge 1$, we have that*

$$g_n^{\theta_4}(\mathrm{A},\mathrm{C}) = \begin{cases} g_n^{\theta_{10}}(\mathrm{A},\mathrm{C}) : n \bmod 2 = 0 \\ g_n^{\theta_1}(\mathrm{A},\mathrm{C}) : n \bmod 2 = 1 \end{cases} = \begin{cases} g_n^{\theta_1}(\mathrm{T},\mathrm{C}) : n \bmod 2 = 0, \\ g_n^{\theta_1}(\mathrm{A},\mathrm{C}) : n \bmod 2 = 1. \end{cases}$$

Proof. We only prove statement (1) by induction on n, as the other cases are similar. By definition of g_n (Definition 4.9), the result holds for $n = 1$ and $n = 2$. Assume the statement to be true for $g_i^{\theta_{10}}(\mathrm{A},\mathrm{C})$, where $3 \leq i < k$. If k is even, then $k \bmod 2 = 0$ and by the inductive hypothesis,

$$\begin{aligned} g_k^{\theta_{10}}(\mathrm{A},\mathrm{C}) &= \theta_{10}(g_{k-1}^{\theta_{10}}(\mathrm{A},\mathrm{C}))g_{k-2}^{\theta_{10}}(\mathrm{A},\mathrm{C}) = \theta_{10}(g_{k-1}^{\theta_1}(\mathrm{A},\mathrm{G}))g_{k-2}^{\theta_1}(\mathrm{T},\mathrm{C}) \\ &= \theta_1(\mu_{10}(g_{k-1}^{\theta_1}(\mathrm{A},\mathrm{G})))g_{k-2}^{\theta_1}(\mathrm{T},\mathrm{C}). \end{aligned}$$

Then by Lemma 4.51, we have

$$g_k^{\theta_{10}}(\mathrm{A},\mathrm{C}) = \theta_1(g_{k-1}^{\theta_1}(\mu_{10}(\mathrm{A}),\mu_{10}(\mathrm{G})))g_{k-2}^{\theta_1}(\mathrm{T},\mathrm{C}) = \theta_1(g_{k-1}^{\theta_1}(\mathrm{T},\mathrm{C}))g_{k-2}^{\theta_1}(\mathrm{T},\mathrm{C}) = g_k^{\theta_1}(\mathrm{T},\mathrm{C}).$$

The case when k is odd is similar. □

Hence, we conclude the following.

Theorem 4.53. *For all $n \geq 1$ and $\phi \in \{\theta_4, \theta_5, \theta_{10}\}$, the atom ϕ-Fibonacci word $g_n^{\phi}(\mathrm{A},\mathrm{C})$ is primitive.*

Proof. It is clear from Corollary 4.50 that the atom ϕ-Fibonacci words $g_n^{\theta_1}(\mathrm{T},\mathrm{C})$, $g_n^{\theta_1}(\mathrm{A},\mathrm{G})$, and $g_n^{\theta_1}(\mathrm{A},\mathrm{C})$ are primitive for $n \geq 1$. Hence, by Lemma 4.52, we conclude that $g_n^{\phi}(\mathrm{A},\mathrm{C})$ are primitive for all $n \geq 1$ and $\phi \in \{\theta_4, \theta_5, \theta_{10}\}$. □

We now show (Theorem 4.54) that for all $\mu = \mu_i$ and $i \in \{4, 5, 10\}$, the atom μ-Fibonacci words g_n are primitive for all $n \geq 1$.

Theorem 4.54. *Let $\phi \in \{\mu_4, \mu_5, \mu_{10}\}$. For all $n \geq 1$, the atom ϕ-Fibonacci word $g_n^{\phi}(\mathrm{A},\mathrm{C})$ is primitive.*

Proof. By Theorem 4.21, we have $g_n^{\phi}(\mathrm{A},\mathrm{C}) = f_n(\mathrm{A},\phi(\mathrm{C}))$ if n is odd and $g_n^{\phi}(\mathrm{A},\mathrm{C}) = f_n(\phi(\mathrm{A}),\mathrm{C})$ if n is even. Note that $\phi(\mathrm{A}) \neq \mathrm{C}$ and $\phi(\mathrm{C}) \neq \mathrm{A}$ for $\phi \in \{\mu_4, \mu_5, \mu_{10}\}$. Then, by Theorem 4.5, the word $g_n^{\phi}(\mathrm{A},\mathrm{C})$ is primitive for $\phi \in \{\mu_4, \mu_5, \mu_{10}\}$ and $n \geq 1$. □

We have the following theorem.

Theorem 4.55. *Let $\phi \in \{\mu_2, \theta_2\}$. For all $n \geq 3$, the atom ϕ-Fibonacci word $g_n^{\phi}(\mathrm{A},\mathrm{C})$ is not primitive.*

Proof. Note that for $\phi \in \{\mu_2, \theta_2\}$, ϕ maps A to C and vice versa, so, by Proposition 4.19, $g_n^{\phi} = \mathrm{A}^{F_n}$ if n is odd and $g_n^{\phi} = \mathrm{C}^{F_n}$ if n is even. Thus, g_n^{ϕ} is not primitive for $\phi \in \{\mu_2, \theta_2\}$ for $n \geq 3$. □

4.6.2 Atom palindromic ϕ-Fibonacci words

In this subsection, we discuss the primitivity of atom palindromic ϕ-Fibonacci words. In Table 4.6, we give the first few values of the sequences $\{w_n^\phi(\mathrm{A},\mathrm{C})\}_{n\geq 1}$ for (anti)morphic involutions $\phi \in \{\mu_1, \mu_2, \mu_4, \mu_5, \mu_{10}, \theta_1, \theta_2, \theta_4, \theta_5, \theta_{10}\}$.

ϕ	$w_3^\phi(\mathrm{A},\mathrm{C})$	$w_4^\phi(\mathrm{A},\mathrm{C})$	$w_5^\phi(\mathrm{A},\mathrm{C})$	$w_6^\phi(\mathrm{A},\mathrm{C})$	$w_7^\phi(\mathrm{A},\mathrm{C})$
μ_1	AC	CAC	CACCA	CACCACAC	CACCACACCACCA
μ_2	AC	CAA	ACCCA	CAAACACC	ACCCACAACAAAC
μ_4	CT	CAC	CTCCA	CACCTCTC	CTCCACACCACCT
μ_5	GA	CAG	GACCA	CAGGAGAC	GACCACAGCAGGA
μ_{10}	GT	CAG	GTCCA	CAGGTGTC	GTCCACAGCAGGT
θ_1	CA	ACC	CCAAC	CAACCCCA	ACCCCAACCAACC
θ_2	AC	ACA	CACAC	ACACACAC	ACACACACACACA
θ_4	CT	ACC	CCTAC	CTACCCCT	ACCCCTACCTACC
θ_5	GA	ACG	CGAAC	GAACGCGA	ACGCGAACGAACG
θ_{10}	GT	ACG	CGTAC	GTACGCGT	ACGCGTACGTACG

Table 4.6: List of words $w_n^\phi(\mathrm{A},\mathrm{C})$, where $3 \leq n \leq 7$ and $\phi \in \{\mu_1, \mu_2, \mu_4, \mu_5, \mu_{10}, \theta_1, \theta_2, \theta_4, \theta_5, \theta_{10}\}$.

Using the definition of ϕ and w_n^ϕ and induction, we calculate the number of occurrences of letters in the words $w_n^\phi(\mathrm{A},\mathrm{C})$ for $n \geq 3$ and an (anti)morphic involution $\phi \in \{\phi_1, \phi_4, \phi_5, \phi_{10}\}$. These values are summarized in Table 4.7.

We now discuss the primitivity of $w_n^{\mu_2}(\mathrm{A},\mathrm{C})$ for all $n \geq 1$, and we use the following lemma.

Lemma 4.56. *Let Σ be an alphabet and $a, b \in \Sigma$ be letters. The following hold.*

1. *For all $n \geq 1$, $F_n \bmod 2 = 0$ if and only if $n \bmod 3 = 0$.*
2. *For $\phi \in \{\mu_2, \mu_8\}$ and for all $n \geq 2$, $w_{3n}^\phi(a,b) \neq x\phi(x)$ where $x \in \Sigma^+$.*

Proof. Statement (1) can be proved easily by induction and using properties of modulo operation. We now prove statement (2) for the case $\phi = \mu_2$ (the case when $\phi = \mu_8$ is similar). One can easily verify from Table 4.6 that the statement holds for $n = 2$. Assume now that $w_{3i} \neq x\mu_2(x)$, with $x \in \Sigma^+$, for all $3 \leq i < k$. Then, by definition of w_n (Definition 4.9) and by the inductive hypothesis, we have $w_{3k} = \mu_2(w_{3k-1})\mu_2(w_{3k-2}) = w_{3k-2}w_{3k-3}\mu_2(w_{3k-2}) \neq w_{3k-2}x\mu_2(x)\mu_2(w_{3k-2})$. Hence, the result holds. ☐

	$i=1$	$i=2$			$i=5$		
n mod 3	≥ 0	0	1	2	0	1	2
$\lvert w_n^{\phi_i}(\mathrm{A},\mathrm{C})\rvert_\mathrm{A}$	F_{n-2}	$\frac{F_n}{2}$	$\frac{F_n+1}{2}$	$\frac{F_n-1}{2}$	F_{n-2}		
$\lvert w_n^{\phi_i}(\mathrm{A},\mathrm{C})\rvert_\mathrm{C}$	F_{n-1}		$\frac{F_n-1}{2}$	$\frac{F_n+1}{2}$	$\frac{F_{n-1}-1}{2}$	$\frac{F_{n-1}}{2}$	$\frac{F_{n-1}+1}{2}$
$\lvert w_n^{\phi_i}(\mathrm{A},\mathrm{C})\rvert_\mathrm{G}$	$-$	$-$			$\frac{F_{n-1}+1}{2}$		$\frac{F_{n-1}-1}{2}$

	$i=4$			$i=10$		
n mod 3	0	1	2	0	1	2
$\lvert w_n^{\phi_i}(\mathrm{A},\mathrm{C})\rvert_\mathrm{A}$	$\frac{F_{n-2}-1}{2}$	$\frac{F_{n-2}+1}{2}$	$\frac{F_{n-2}}{2}$	$\frac{F_{n-2}-1}{2}$	$\frac{F_{n-2}+1}{2}$	$\frac{F_{n-2}}{2}$
$\lvert w_n^{\phi_i}(\mathrm{A},\mathrm{C})\rvert_\mathrm{C}$	F_{n-1}			$\frac{F_{n-1}-1}{2}$	$\frac{F_{n-1}}{2}$	$\frac{F_{n-1}+1}{2}$
$\lvert w_n^{\phi_i}(\mathrm{A},\mathrm{C})\rvert_\mathrm{G}$	$-$			$\frac{F_{n-1}+1}{2}$		$\frac{F_{n-1}-1}{2}$
$\lvert w_n^{\phi_i}(\mathrm{A},\mathrm{C})\rvert_\mathrm{T}$	$\frac{F_{n-2}+1}{2}$	$\frac{F_{n-2}-1}{2}$	$\frac{F_{n-2}}{2}$	$\frac{F_{n-2}+1}{2}$	$\frac{F_{n-2}-1}{2}$	$\frac{F_{n-2}}{2}$

Table 4.7: The numbers of occurrences of letters in the atom palindromic ϕ_i-Fibonacci words $w_n^{\phi_i}(\mathrm{A},\mathrm{C})$ for $\phi_i \in \{\theta_i, \mu_i\}$, $i \in \{1, 2, 4, 5, 10\}$ and $n \geq 3$.

Proposition 4.57. *Let $\phi = \mu_2$. For all $n \geq 1$ such that $n \bmod 3 = 0$, the atom palindromic ϕ-Fibonacci word $w_n^{\mu_2}(\mathrm{A},\mathrm{C})$ is primitive.*

Proof. By Table 4.6, w_n is primitive for $1 \leq n \leq 7$. Suppose w_n is not primitive for $n > 7$ and $n \bmod 3 = 0$, then we have $w_n = p^j$, where $j \geq 2$ and $p \in Q$. By definition of w_n (Definition 4.9), the word w_n can be decomposed as

$$w_n = \mu_2(w_{n-1})\mu_2(w_{n-2}) = w_{n-2}w_{n-3}w_{n-3}w_{n-4} = w_{n-4}w_{n-5}\mu_2(w_{n-4})w_{n-3}w_{n-3}w_{n-4}.$$

Let $y = w_{n-4}w_{n-5}\mu_2(w_{n-4})w_{n-3}w_{n-3}$ such that $w_n = yx$ and $x = w_{n-4}$. Note that $(n-4) \bmod 3 = 2$ and by Table 4.7, we have $|w_n|_\mathrm{A} = |w_n|_\mathrm{C}$ and $|x|_\mathrm{A}+1 = |x|_\mathrm{C}$, so $|y|_\mathrm{A} = |y|_\mathrm{C}+1$. Hence, by Lemma 4.4 and Lemma 4.8, both y and x are primitive. Since $x = w_{n-4}$, we have $|x| = F_{n-4} \geq \frac{7F_{n-4}-F_{n-8}}{7} - 1 = \frac{F_n}{7} - 1$. For $j \geq 7$, we get $|x| \geq \frac{F_n}{j} - 1 \geq \frac{F_n}{j} - \gcd(|p|, |y|) = |p| - \gcd(|p|, |y|)$, as $\gcd(|p|, |y|) \geq 1$. Then, by Proposition 4.7, we have $p = y$, which is impossible. Hence, $w_n \neq p^j$ for $j \geq 7$ and $p \in Q$.

We now consider the cases when $2 \leq j \leq 6$. We split it into three cases when j is even, $j = 3$, and $j = 5$.

- If j is even, then w_n can be written as $w_n = pp$, where $|p| = \frac{F_n}{2} > 4$. By Lemma 4.56, $|w_{n-3}|$ is even, so there exist $x, y \in \Delta^+$ such that $|x| = |y|$ and $w_{n-3} = xy$. Then, by definition $w_n = w_{n-2}w_{n-3}\mu_2(w_{n-2})$, we have $p = w_{n-2}x = y\mu_2(w_{n-2})$ for $w_{n-3} = xy$. Thus, $p = w_{n-2}x = \mu_2(w_{n-3})\mu_2(w_{n-4})x = \mu_2(x)\mu_2(y)\mu_2(w_{n-4})x = y\mu_2(w_{n-2})$ implies $y = \mu_2(x)$ and $w_{n-3} = x\mu_2(x)$, which contradicts Lemma 4.56, so j cannot be even.

- If $j = 3$, then by induction, we have $F_n \bmod 3 = 0$ if and only if $n \bmod 4 = 0$. Therefore, if $n \bmod 4 \neq 0$, then w_n cannot be written as $w_n = q^3$, where $q \in Q$, so we only need to consider the case where $n \bmod 12 = 0$. Note that the word w_n can be written as

$$\begin{aligned} w_n &= \mu_2(w_{n-3})\mu_2(w_{n-4})w_{n-3}w_{n-3}w_{n-4} \\ &= w_{n-4}w_{n-5}\mu_2(w_{n-4})\mu_2(w_{n-4})\mu_2(w_{n-5})\mu_2(w_{n-4})\mu_2(w_{n-5})w_{n-4} = q^3 \end{aligned}$$

  and $|F_{n-4}|$ is divisible by 3. Then, there exist $x, y, r \in \Delta^+$ such that $|x| = |y| = |r|$ and $w_{n-4} = xyr$. Since $w_n = q^3$, we have

$$q = xyrw_{n-5}\mu_2(xy) = \mu_2(r)\mu_2(xyr)\mu_2(w_{n-5})\mu_2(x) = \mu_2(yr)\mu_2(w_{n-5})xyr.$$

  Thus, we have $x = \mu_2(y) = \mu_2(r) = y$, which is a contradiction because μ_2 is not the identity mapping. Hence, $j \neq 3$.

- If $j = 5$, by induction, we have $F_n \bmod 5 = 0$ if and only if $n \bmod 5 = 0$. Therefore, if $n \bmod 5 \neq 0$, w_n cannot be written as $w_n = q^5$, where $q \in Q$, so we only need to consider the case where $n \bmod 15 = 0$. Note that the word w_n can be written as $w_n = w_{n-4}w_{n-5}\mu_2(w_{n-4})\mu_2(w_{n-4})\mu_2(w_{n-5})\mu_2(w_{n-4})\mu_2(w_{n-5})w_{n-4} = q^5$ and $|F_{n-5}|$ is divisible by 5. Then, there exist $x, y, r, s, t \in \Delta^+$ such that $|x| = |y| = |r| = |s| = |t|$ and $w_{n-5} = xyrst$. Since $w_n = q^5$, we have $q = w_{n-4}xyr = st\mu_2(w_{n-4})s' = t'\mu_2(xyrs) = \mu_2(t)\mu_2(w_{n-4})\mu_2(xy) = \mu_2(rst)w_{n-4}$, where $\mu_2(w_{n-4}) = s't'$ for some $s', t' \in \Delta^+$. Then, we have $r = \mu_2(r)$, which is a contradiction because μ_2 is not the identity mapping. Hence, $j \neq 5$.

□

Theorem 4.58. *Let $\phi \in \{\theta_1, \theta_2, \theta_4, \theta_5, \theta_{10}, \mu_2, \mu_4, \mu_5, \mu_{10}\}$. The primitivity properties of the atom palindromic ϕ-Fibonacci words $w_n^{\phi}(\mathrm{A}, \mathrm{C})$ for $n \geq 3$, are as follows:*

1. *For $\phi = \theta_2$, the atom ϕ-Fibonacci word $w_n^{\theta_2}(\mathrm{A}, \mathrm{C})$ is primitive if and only if $n \bmod 3 \neq 0$.*

2. *If $\phi \in \{\theta_1, \theta_4, \theta_5, \theta_{10}, \mu_2, \mu_4, \mu_5, \mu_{10}\}$, the atom ϕ-Fibonacci word $w_n^{\phi}(\mathrm{A}, \mathrm{C})$ is primitive.*

Proof. 1. Given that $\phi = \theta_2$. From Table 4.6, we have $w_3^{\theta_2} = \mathrm{AC}$, which is primitive. By Proposition 4.19, the words $w_n^{\theta_2}$ are not primitive for $n > 3$ and $n \bmod 3 = 0$. Conversely, if $n \bmod 3 \neq 0$, then by Table 4.7, $\gcd(|w_n^{\theta_2}|_{\mathrm{A}}, |w_n^{\theta_2}|_{\mathrm{C}}) = 1$ for all $n \geq 3$. Therefore, by Lemma 4.8, the words $w_n^{\theta_2}$ are primitive for $n \geq 3$ and $n \bmod 3 \neq 0$.

2. For $\phi = \mu_2$, by Proposition 4.57, the words $w_n^{\mu_2}$ are primitive for $n \bmod 3 = 0$. By Table 4.7, for the converse and for all other cases of ϕ, there exist two letters $a, b \in \Delta$, where $\gcd(|w_n^\phi|_a, |w_n^\phi|_b) = 1$, for all $n \geq 3$. Therefore, by Lemma 4.4 and Lemma 4.8, the words w_n^ϕ are primitive for all $n \geq 3$.

□

4.6.3 Atom hairpin ϕ-Fibonacci words

In Table 4.8, we begin by giving the first few values of the sequences $\{z_n^\phi(\mathrm{A},\mathrm{C})\}_{n\geq 1}$ for (anti)morphic involutions $\phi \in \{\mu_1, \mu_2, \mu_4, \mu_5, \mu_{10}, \theta_1, \theta_2, \theta_4, \theta_5, \theta_{10}\}$.

ϕ	$z_3^\phi(\mathrm{A},\mathrm{C})$	$z_4^\phi(\mathrm{A},\mathrm{C})$	$z_5^\phi(\mathrm{A},\mathrm{C})$	$z_6^\phi(\mathrm{A},\mathrm{C})$	$z_7^\phi(\mathrm{A},\mathrm{C})$
μ_1	CA	CAC	CACCA	CACCACAC	CACCACACCACCA
μ_2	CC	CCA	CCAAA	CCAAAAAC	CCAAAAACAACCC
μ_4	CT	CTC	CTCCA	CTCCACAC	CTCCACACCACCT
μ_5	CA	CAG	CAGGA	CAGGAGAC	CAGGAGACGACCA
μ_{10}	CT	CTG	CTGGA	CTGGAGAC	CTGGAGACGACCT
θ_1	CA	CAC	CACAC	CACACCAC	CACACCACCACAC
θ_2	CC	CCA	CCAAA	CCAAACAA	CCAAACAACCCAA
θ_4	CT	CTC	CTCAC	CTCACCAC	CTCACCACCTCAC
θ_5	CA	CAG	CAGAG	CAGAGCAG	CAGAGCAGCACAG
θ_{10}	CT	CTG	CTGAG	CTGAGCAG	CTGAGCAGCTCAG

Table 4.8: List of words $z_n^\phi(\mathrm{A},\mathrm{C})$, where $3 \leq n \leq 7$ and $\phi \in \{\mu_1, \mu_2, \mu_4, \mu_5, \mu_{10}, \theta_1, \theta_2, \theta_4, \theta_5, \theta_{10}\}$.

Similar to that of Table 4.7, we calculate the number of occurrences of letters in the words $z_n^\phi(\mathrm{A},\mathrm{C})$, for all $n \geq 3$ and $\phi \in \{\phi_1, \phi_4, \phi_5, \phi_{10}\}$, as summarized in Table 4.9.

Lemma 4.59. *Let $\phi \in \{\theta_2, \mu_2\}$. For all $n > 3$, the atom hairpin ϕ-Fibonacci word $z_n^\phi(\mathrm{A},\mathrm{C})$ cannot be a square.*

Proof. First, we consider the case where $\phi = \theta_2$. By Table 4.8, the statement is true for $4 \leq n \leq 7$. Assuming that the statement holds for $z_i^\phi(\mathrm{A},\mathrm{C})$, where $7 \leq i < k$, we now prove it for $z_k^\phi(\mathrm{A},\mathrm{C})$. We only need to consider the condition where $k \bmod 3 = 0$, since by induction, we have $F_k \bmod 2 = 0$ if and only if $k \bmod 3 = 0$. By definition of z_n (Definition 4.9), we have $z_k = z_{k-2}\theta_2(z_{k-3})\theta_2(z_{k-2}) = pp$. Since $(k-3) \bmod 3 =$

	$i=1$	$i=2$				$i=5$		
n mod 6	≥ 0	0	$1,5$	$2,4$	3	$0,5$	$1,4$	$2,3$
$\lvert z_n^{\phi_i}(\mathrm{A},\mathrm{C})\rvert_{\mathrm{A}}$	F_{n-2}	$\frac{F_n}{2}+1$	$\frac{F_n+1}{2}$	$\frac{F_n-1}{2}$	$\frac{F_n}{2}-1$	F_{n-2}		
$\lvert z_n^{\phi_i}(\mathrm{A},\mathrm{C})\rvert_{\mathrm{C}}$	F_{n-1}	$\frac{F_n}{2}-1$	$\frac{F_n-1}{2}$	$\frac{F_n+1}{2}$	$\frac{F_n}{2}+1$	$\frac{F_{n-1}-1}{2}$	$\frac{F_{n-1}}{2}$	$\frac{F_{n-1}+1}{2}$
$\lvert z_n^{\phi_i}(\mathrm{A},\mathrm{C})\rvert_{\mathrm{G}}$	–	–				$\frac{F_{n-1}+1}{2}$	$\frac{F_{n-1}}{2}$	$\frac{F_{n-1}-1}{2}$

	$i=4$			$i=10$					
n mod 6	$0,1$	$2,5$	$3,4$	0	1	2	3	4	5
$\lvert z_n^{\phi_i}(\mathrm{A},\mathrm{C})\rvert_{\mathrm{A}}$	$\frac{F_{n-2}+1}{2}$	$\frac{F_{n-2}}{2}$	$\frac{F_{n-2}-1}{2}$	$\frac{F_{n-2}+1}{2}$		$\frac{F_{n-2}}{2}$	$\frac{F_{n-2}-1}{2}$		$\frac{F_{n-2}}{2}$
$\lvert z_n^{\phi_i}(\mathrm{A},\mathrm{C})\rvert_{\mathrm{C}}$	F_{n-1}			$\frac{F_{n-1}-1}{2}$	$\frac{F_{n-1}}{2}$	$\frac{F_{n-1}+1}{2}$	$\frac{F_{n-1}}{2}$		$\frac{F_{n-1}-1}{2}$
$\lvert z_n^{\phi_i}(\mathrm{A},\mathrm{C})\rvert_{\mathrm{G}}$	–			$\frac{F_{n-1}+1}{2}$		$\frac{F_{n-1}-1}{2}$			$\frac{F_{n-1}+1}{2}$
$\lvert z_n^{\phi_i}(\mathrm{A},\mathrm{C})\rvert_{\mathrm{T}}$	$\frac{F_{n-2}-1}{2}$	$\frac{F_{n-2}}{2}$	$\frac{F_{n-2}+1}{2}$	$\frac{F_{n-2}-1}{2}$		$\frac{F_{n-2}}{2}$	$\frac{F_{n-2}+1}{2}$		$\frac{F_{n-2}}{2}$

Table 4.9: The numbers of occurrences of letters in the atom hairpin ϕ_i-Fibonacci words $z_n^{\phi_i}(\mathrm{A},\mathrm{C})$ for $\phi_i \in \{\theta_i, \mu_i\}$, $i \in \{1,2,4,5,10\}$ and $n \geq 3$.

$F_{k-3} \bmod 2 = 0$, there exist $x, y \in \Delta^+$ such that $z_{k-3} = xy$ and $|x| = |y|$. Here, $p = z_{k-2}\theta_2(y)$ and $p = \theta_2(x)\theta_2(z_{k-2}) = \theta_2(x)z_{k-4}\theta_2(z_{k-3}) = \theta_2(x)z_{k-4}\theta_2(y)\theta_2(x)$. Therefore, $\theta_2(x) = \theta_2(y)$, so $z_{k-3} = x^2$, which contradicts the inductive hypothesis.

Next, we consider the case where $\phi = \mu_2$. By Table 4.8, the statement is true for $4 \leq n \leq 7$. Assuming the statement holds for $z_i^{\phi}(A,C)$, where $7 \leq i < k$, we now prove it for $z_k^{\phi}(A,C)$. We only need to consider the condition where $k \bmod 3 = 0$, since by induction, we have $F_k \bmod 2 = 0$ if and only if $k \bmod 3 = 0$. By definition of z_n (Definition 4.9), we have $z_k = z_{k-2}\mu_2(z_{k-3})\mu_2(z_{k-2}) = pp$. Since $(k-3) \bmod 3 = F_{k-3} \bmod 2 = 0$, so there exist $x, y \in \Delta^+$ such that $z_{k-3} = xy$ and $|x| = |y|$. We now have $p = z_{k-2}\mu_2(x) = z_{k-3}\mu_2(z_{k-4})\mu_2(x) = xy\mu_2(z_{k-4})\mu_2(x)$, and $p = \mu_2(y)\mu_2(z_{k-2}) = \mu_2(y)\mu_2(z_{k-3})z_{k-4} = \mu_2(yxy)z_{k-4}$. This implies $x = \mu_2(y)$ and $\mu_2(z_{k-4})\mu_2(x) = xz_{k-4}$. Note that $z_{k-3} = z_{k-4}\mu_2(z_{k-5}) = xy$, so $z_{k-4} = x\mu_2(y_1)$ for some $y = y_1y_2$. Thus, $xz_{k-4} = xx\mu_2(y_1) = \mu_2(z_{k-4})\mu_2(x) = \mu_2(xy_1)\mu_2(x)$, which implies $x = \mu_2(x)$, so $z_{k-3} = x^2$, which is a contradiction. □

Theorem 4.60. *Let $\phi \in \{\theta_1, \theta_2, \theta_4, \theta_5, \theta_{10}, \mu_2, \mu_4, \mu_5, \mu_{10}\}$. For all $n \geq 1$, we have:*

1. *If $\phi \in \{\theta_2, \mu_2\}$ and $n \neq 3$, then the atom hairpin ϕ-Fibonacci word $z_n^{\phi}(\mathrm{A},\mathrm{C})$ is primitive.*

2. *If $\phi \in \{\theta_1, \theta_4, \theta_5, \theta_{10}, \mu_4, \mu_5, \mu_{10}\}$, then the atom hairpin ϕ-Fibonacci word $z_n^{\phi}(\mathrm{A},\mathrm{C})$ is primitive.*

Proof. 1. Let $\phi \in \{\theta_2, \mu_2\}$. Note that $z_3 = \mathrm{CC}$ is not primitive. By Table 4.9, for $n \bmod 6 \in \{1,2,4,5\}$, we have $\gcd(|z_n^\phi|_{\mathrm{A}}, |z_n^\phi|_{\mathrm{C}}) = 1$, so by Lemma 4.8, the word z_n^ϕ is primitive in these cases. For $n \bmod 6 \in \{0,3\}$, the number of occurrences of A and C are even or odd integers that are consecutive and positive. If they are consecutive positive odd integers, by Lemma 4.8, the word z_n^ϕ is primitive. If they are consecutive positive even integers, the only non-trivial common divisor is 2, and we can prove that z_n^ϕ is primitive by contradiction. Assume z_n^ϕ is not primitive in this case, which means that $z_n^\phi = p^i$ (refer to proof of Lemma 4.8 to show that $i = 2$), where $p \in Q$, and this contradicts Lemma 4.59.

2. By Table 4.9, for all other cases of ϕ, there exist two letters $a, b \in \Delta$, where $\gcd(|z_n^\phi|_a, |z_n^\phi|_b) = 1$ for all $n \geq 3$. Therefore, by Lemma 4.8, the word z_n^ϕ is primitive for all $n \geq 3$.

□

Let $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \geq 1$ and let $\phi \in \{\theta_1, \theta_2, \theta_4, \theta_5, \theta_{10}, \mu_2, \mu_4, \mu_5, \mu_{10}\}$. Based on Corollary 4.50, Theorem 4.53, Theorem 4.54, Theorem 4.55, Theorem 4.58, and Theorem 4.60, we conclude that the words $g_n^{\theta_2}$ and $g_n^{\mu_2}$ are not primitive for all $n \geq 3$, the word $w_n^{\mu_2}$ is primitive if $n \bmod 3 \neq 0$ and $n \geq 3$, and for all other cases, the word α_n^ϕ is primitive. These are summarized in Table 4.10.

	$i \in \{1,7\}$	$i \in \{2,8\}$	$i \in \{3,4\}$	$i \in \{5,6\}$	$i \in \{9,10\}$
$g_n^{\theta_i}(A,C)$	✓	✗ (except $n = 1,2$)	✓	✓	✓
$g_n^{\mu_i}(A,C)$	✓	✗ (except $n = 1,2$)	✓	✓	✓
$w_n^{\theta_i}(A,C)$	✓	✓ (except n mod $3 = 0, n > 3$)	✓	✓	✓
$w_n^{\mu_i}(A,C)$	✓	✓	✓	✓	✓
$z_n^{\theta_i}(A,C)$	✓	✓(except $n = 3$)	✓	✓	✓
$z_n^{\mu_i}(A,C)$	✓	✓(except $n = 3$)	✓	✓	✓

Table 4.10: Primitivity of atom ϕ-Fibonacci words $\alpha_n^\phi(\mathrm{A},\mathrm{C})$ for all $n \geq 1$, with different initial letters $\mathrm{A}, \mathrm{C} \in \Delta$, where $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \geq 1$, and $\phi \in \{\theta_i, \mu_i \mid 1 \leq i \leq 10\}$ (here, ✓ means that the words are primitive, and ✗ means that they are not primitive).

4.7 Primitivity of atom ϕ-Fibonacci words with identical initial letters

In this section, we discuss the primitivity of atom ϕ-Fibonacci words $\alpha_n^{\phi}(a,a)$ with identical initial letters for all $n \geq 1$, where $a \in \Sigma_4$, $\phi \in \{\theta_i \mid 1 \leq i \leq 10\} \cup \{\mu_i \mid 1 \leq i \leq 10\}$, and $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \geq 1$. The primitivity results of this section hold for every choice of (x_1, x_2, x_3, x_4) and with x_1 and x_1 as the initial letters of the ϕ-Fibonacci sequence. Therefore, we only prove primitivity results for one of the cases, the sequence (A, C, G, T) over the DNA alphabet Δ, with A and A as the first two initial letters.

We can classify ϕ into two categories, where $\phi(a) = a$ and $\phi(a) \neq a$. If $\phi(a) = a$, then for all $n \geq 1$ we have that $\alpha_n^{\phi}(a,a) = a^{F_n}$, where $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \geq 1$. Hence, $\alpha_n^{\phi}(a,a)$ is not primitive for all $n \geq 3$. Therefore, we only need to consider the case where $\phi(a) \neq a$. The set of all (anti)morphic involutions ϕ for the sequence (x_1, x_2, x_3, x_4) with $x_1 = a$ and $\phi_i(a) \neq a$ is $\{\phi_i \mid i = 2, 3, 4, 8, 9, 10\}$. It is enough if we discuss the primitivity for one of such ϕ_i, say ϕ_2. We first show that it is sufficient to study the primitivity only for $\alpha_n^{\phi_2}(a,a)$, where $n \geq 1$.

We use the following lemma.

Lemma 4.61. *Let $\phi_i \in \{\mu_i, \theta_i\}$, $i \in \{2, 3, 4, 8, 9, 10\}$, be an (anti)morphic involution on Σ_4^* and $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \geq 1$. For all $n \geq 1$, the following equalities hold regarding ϕ_i-Fibonacci words $\alpha_n^{\phi_i}(x_1, x_1)$:*

- $\phi_2(\alpha_n^{\phi_2}(x_1, x_1)) = \phi_8(\alpha_n^{\phi_8}(x_1, x_1))$ *if and only if* $\alpha_n^{\phi_2}(x_1, x_1) = \alpha_n^{\phi_8}(x_1, x_1)$.
- $\phi_3(\alpha_n^{\phi_3}(x_1, x_1)) = \phi_9(\alpha_n^{\phi_9}(x_1, x_1))$ *if and only if* $\alpha_n^{\phi_3}(x_1, x_1) = \alpha_n^{\phi_9}(x_1, x_1)$.
- $\phi_4(\alpha_n^{\phi_4}(x_1, x_1)) = \phi_{10}(\alpha_n^{\phi_{10}}(x_1, x_1))$ *if and only if* $\alpha_n^{\phi_4}(x_1, x_1) = \alpha_n^{\phi_{10}}(x_1, x_1)$.
- $[\phi_2(\alpha_n^{\phi_2}(x_1, x_2))]_{x_2 \to x_3} = \phi_3(\alpha_n^{\phi_3}(x_1, x_2))$ *if and only if* $[\alpha_n^{\phi_2}(x_1, x_2)]_{x_2 \to x_3} = \alpha_n^{\phi_3}(x_1, x_2)$.
- $[\phi_2(\alpha_n^{\phi_2}(x_1, x_2))]_{x_2 \to x_4} = \phi_4(\alpha_n^{\phi_4}(x_1, x_2))$ *if and only if* $[\alpha_n^{\phi_2}(x_1, x_2)]_{x_2 \to x_4} = \alpha_n^{\phi_4}(x_1, x_2)$.

Proof. The proof, by induction on n, is similar to that of Lemma 4.46. □

Based on Lemma 4.61, we have the following result.

Lemma 4.62. *Let $\phi_i \in \{\theta_i, \mu_i\}$, $i \in \{2, 3, 4, 8, 9, 10\}$, be an (anti)morphic involution on Σ_4^*, and let $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \geq 1$. For all $n \geq 1$, the following equalities regarding ϕ_i-Fibonacci words $\alpha_n^{\phi_i}(x_1, x_1)$ hold:*

- $\alpha_n^{\phi_2}(x_1, x_1) = \alpha_n^{\phi_8}(x_1, x_1)$.
- $[\alpha_n^{\phi_2}(x_1, x_1)]_{x_2 \to x_3} = \alpha_n^{\phi_3}(x_1, x_1) = \alpha_n^{\phi_9}(x_1, x_1)$.
- $[\alpha_n^{\phi_2}(x_1, x_1)]_{x_2 \to x_4} = \alpha_n^{\phi_4}(x_1, x_1) = \alpha_n^{\phi_{10}}(x_1, x_1)$.

Proof. The proof by induction on n and using Lemma 4.61 is similar to that of Lemma 4.47. □

One can easily observe that for $\alpha_n^{\phi}(a, a)$, we have $|\mathrm{Alph}(\alpha_n^{\phi}(a, a))| \leq 2$ for all $n \geq 3$, where ϕ is an (anti)morphic involution and $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \geq 3$. Thus, by Lemma 4.62 and Lemma 4.48, it is enough to discuss the primitivity for $\alpha_n^{\phi_2}(x_1, x_1)$, $n \geq 3$. As the choice of initial letter does not matter, we choose for convenience A and A to be the two initial letters. We now study the primitivity of the words $\alpha_n^{\phi_2}(\mathrm{A}, \mathrm{A})$ for all $n \geq 3$, where $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \geq 3$.

4.7.1 Atom alternating ϕ-Fibonacci words

The first few values of the sequence $\{g_n^{\phi}(\mathrm{A}, \mathrm{A})\}_{n \geq 1}$ for $\phi \in \{\mu_2, \theta_2\}$ are given in Table 4.11.

ϕ	$g_3^{\phi}(\mathrm{A}, \mathrm{A})$	$g_4^{\phi}(\mathrm{A}, \mathrm{A})$	$g_5^{\phi}(\mathrm{A}, \mathrm{A})$	$g_6^{\phi}(\mathrm{A}, \mathrm{A})$	$g_7^{\phi}(\mathrm{A}, \mathrm{A})$
μ_2	CA	ACA	CACCA	ACAACACA	CACCACACCACCA
θ_2	CA	CAA	CCACA	CACAACAA	CCACCACACCACA

Table 4.11: List of words $g_n^{\phi}(\mathrm{A}, \mathrm{A})$, where $3 \leq n \leq 7$ and $\phi \in \{\mu_2, \theta_2\}$.

We first show that for a morphic involution μ_2, the μ_2-Fibonacci words g_n with identical initial letters are primitive. The proof is similar to that of Theorem 4.54.

Theorem 4.63. *Let $\phi = \mu_2$. For all $n \geq 1$, the atom alternating ϕ-Fibonacci word $g_n^{\mu_2}(\mathrm{A}, \mathrm{A})$ is primitive.*

Proof. By Theorem 4.21, we have $g_n^{\phi}(\mathrm{A}, \mathrm{A}) = f_n(\mathrm{A}, \phi(\mathrm{A}))$ if n is odd and $g_n^{\phi}(\mathrm{A}, \mathrm{A}) = f_n(\phi(\mathrm{A}), \mathrm{A})$if n is even. Note that $\phi(\mathrm{A}) \neq \mathrm{A}$ for $\phi = \mu_2$. Then, by Theorem 4.5, the word $g_n^{\phi}(\mathrm{A}, \mathrm{A})$ is primitive for $\phi = \mu_2$ and $n \geq 1$. □

We now discuss the primitivity of θ_2-Fibonacci words $g_n^{\theta_2}(\mathrm{A}, \mathrm{A})$, $n \geq 3$. The following result can be proved by induction, and we omit the proof.

Theorem 4.64. *Let $\phi \in \{\theta_1, \theta_2\}$. For all $n \geq 1$, the atom alternating ϕ-Fibonacci word $g_n^{\theta_2}(\mathrm{A}, \mathrm{A})$ can be represented by atom alternating ϕ-Fibonacci words $g_n^{\theta_1}(\mathrm{A}, \mathrm{C})$ and $g_n^{\theta_1}(\mathrm{C}, \mathrm{A})$ as follows:*

$$g_n^{\theta_2}(\mathrm{A}, \mathrm{A}) = \begin{cases} g_n^{\theta_1}(\mathrm{C}, \mathrm{A}) : n \bmod 2 = 0, \\ g_n^{\theta_1}(\mathrm{A}, \mathrm{C}) : n \bmod 2 = 1. \end{cases}$$

Proof. The proof by induction on n and using Lemma 4.51 is similar to that of Lemma 4.52. $\square$

Using Theorem 4.64 and Corollary 4.50, we have the following corollary.

Corollary 4.65. *Let $\phi = \theta_2$. For all $n \geq 1$, the atom alternating ϕ-Fibonacci word $g_n^{\theta_2}(\mathrm{A}, \mathrm{A})$ is primitive.*

4.7.2 Atom palindromic ϕ-Fibonacci words

The first few values of the sequences $\{w_n^{\phi}(\mathrm{A}, \mathrm{A})\}_{n \geq 1}$ for $\phi \in \{\mu_2, \theta_2\}$ are given in Table 4.12.

ϕ	$w_3^{\phi}(\mathrm{A}, \mathrm{A})$	$w_4^{\phi}(\mathrm{A}, \mathrm{A})$	$w_5^{\phi}(\mathrm{A}, \mathrm{A})$	$w_6^{\phi}(\mathrm{A}, \mathrm{A})$	$w_7^{\phi}(\mathrm{A}, \mathrm{A})$
μ_2	CC	AAC	CCAAA	AACCCCCA	CCAAAAACAACCC
θ_2	CC	AAC	ACCAA	CCAACACC	AACACCAACCAAC

Table 4.12: List of words $w_n^{\phi}(\mathrm{A}, \mathrm{A})$, where $3 \leq n \leq 7$ and $\phi \in \{\mu_2, \theta_2\}$.

Therefore, we have the following theorem.

Theorem 4.66. *Let $\phi = \mu_2$. For all $n \geq 1$, the atom palindromic ϕ-Fibonacci word $w_n^{\mu_2}(\mathrm{A}, \mathrm{A})$ is primitive.*

Proof. We have by Theorem 4.21, $w_n^{\mu_2}(\mathrm{A}, \mathrm{A}) = z_n^{\mu_2}(\mathrm{A}, \mathrm{C})$ if n is odd and $w_n^{\mu_2}(\mathrm{A}, \mathrm{A}) = z_n^{\mu_2}(\mathrm{C}, \mathrm{A})$ if n is even. Therefore, by Theorem 4.60, $w_n^{\mu_2}(\mathrm{A}, \mathrm{A})$ is primitive for $n \geq 1$. $\square$

We now count the number of occurrences of the letters A and C in the atom palindromic ϕ_2-Fibonacci words.

Lemma 4.67. *Let $\phi \in \{\mu_2, \theta_2\}$. For all $n \geq 1$, the numbers of occurrences of letters in the atom palindromic ϕ-Fibonacci word $w_n^{\phi}(\mathrm{A}, \mathrm{A})$ satisfy:*

$$|w_n^{\phi}|_{\mathrm{A}} = \begin{cases} \frac{F_n}{2} - 1 : n \bmod 3 = 0, \\ \frac{F_n + 1}{2} : n \bmod 3 \neq 0, \end{cases} \quad |w_n^{\phi}|_{\mathrm{C}} = \begin{cases} \frac{F_n}{2} + 1 : n \bmod 3 = 0, \\ \frac{F_n - 1}{2} : n \bmod 3 \neq 0. \end{cases}$$

Proof. The proof uses the fact that $|w^{\phi}_{n+2}|_{\mathrm{A}} = |w^{\phi}_{n+1}|_{\mathrm{C}} + |w^{\phi}_{n}|_{\mathrm{C}}, |w^{\phi}_{n+2}|_{\mathrm{C}} = |w^{\phi}_{n+1}|_{\mathrm{A}} + |w^{\phi}_{n}|_{\mathrm{A}}$ and is by induction on n. □

By induction on n, we have the following lemma.

Lemma 4.68. *Let $\phi = \theta_2$. For all $n > 3$, the atom palindromic ϕ-Fibonacci word $w^{\theta_2}_n(\mathrm{A}, \mathrm{A})$ cannot be a square.*

Proof. The proof is by induction on n. One can easily check that the statement is true for $4 \leq n \leq 10$. We now assume that $w^{\theta_2}_i(\mathrm{A}, \mathrm{A})$ is not a square for all $10 \leq i < k$. We prove that this is true for $w^{\theta_2}_k(\mathrm{A}, \mathrm{A})$. We only need to consider the condition where $k \bmod 3 = 0$, since by induction, we have $F_k \bmod 2 = 0$ if and only if $k \bmod 3 = 0$. By definition of w_n (Definition 4.9), we have $w_k = \theta_2(w_{k-1})\theta_2(w_{k-2}) = w_{k-3}w_{k-2}w_{k-4}w_{k-3} = w_{k-3}\theta_2(w_{k-3})\theta_2(w_{k-4})w_{k-4}w_{k-3} = w_{k-3}w_{k-5}w_{k-4}w_{k-6}w_{k-5}w_{k-4}w_{k-3} = pp$. Since $(k-6) \bmod 3 = F_{k-6} \bmod 2 = 0$, there exist $x, y \in \Delta^+$ such that $w_{k-6} = xy$ and $|x| = |y|$. We now have

$$p = w_{k-3}w_{k-5}w_{k-4}x = \theta_2(w_{k-4})\theta_2(w_{k-5})w_{k-5}w_{k-4}x = w_{k-6}w_{k-5}\theta_2(w_{k-5})w_{k-5}w_{k-4}x$$

$$= xyw_{k-5}\theta_2(w_{k-5})w_{k-5}w_{k-4}x,$$

and $p = yw_{k-5}w_{k-4}w_{k-3}$. Therefore, we have $x = y$, so $w_{k-6} = x^2$, which contradicts the inductive hypothesis. □

Theorem 4.69. *Let $\phi = \theta_2$. For $n \geq 1$, the atom palindromic ϕ-Fibonacci word $w^{\theta_2}_n(\mathrm{A}, \mathrm{A})$ is primitive if and only if $n \neq 3$.*

Proof. The proof is similar to part (1) of the proof of Theorem 4.60 and uses Lemma 4.68. □

4.7.3 Atom hairpin ϕ-Fibonacci words

In Table 4.13, we give the first few values of the sequences $\{z^{\phi}_n(\mathrm{A}, \mathrm{A})\}_{n\geq 1}$ for $\phi \in \{\mu_2, \theta_2\}$.

Theorem 4.70. *Let $\phi = \mu_2$. For all $n \geq 1$, the atom hairpin ϕ-Fibonacci word $z^{\mu_2}_n(\mathrm{A}, \mathrm{A})$ is primitive.*

Proof. We have by Theorem 4.21, $z^{\mu_2}_n(\mathrm{A}, \mathrm{A}) = w^{\mu_2}_n(\mathrm{A}, \mathrm{C})$ if n is odd and $z^{\mu_2}_n(\mathrm{A}, \mathrm{A}) = w^{\mu_2}_n(\mathrm{C}, \mathrm{A})$ if n is even. Therefore, by Theorem 4.58, $z^{\mu_2}_n(\mathrm{A}, \mathrm{A})$ is primitive for all $n \geq 1$. □

ϕ	$z_3^{\phi}(\mathrm{A},\mathrm{A})$	$z_4^{\phi}(\mathrm{A},\mathrm{A})$	$z_5^{\phi}(\mathrm{A},\mathrm{A})$	$z_6^{\phi}(\mathrm{A},\mathrm{A})$	$z_7^{\phi}(\mathrm{A},\mathrm{A})$
μ_2	AC	ACC	ACCCA	ACCCACAA	ACCCACAACAAAC
θ_2	AC	ACC	ACCAC	ACCACAAC	ACCACAACACAAC

Table 4.13: List of words $z_n^{\phi}(\mathrm{A},\mathrm{A})$, where $3 \le n \le 7$ and $\phi \in \{\mu_2, \theta_2\}$.

We now count the number of occurrences of the letters A and C in the atom hairpin ϕ_2-Fibonacci words. The proof is by induction, and we omit it.

Lemma 4.71. *Let $\phi \in \{\mu_2, \theta_2\}$. For all $n \ge 1$, the numbers of occurrences of letters in the atom hairpin ϕ-Fibonacci word $z_n^{\phi}(\mathrm{A},\mathrm{A})$ satisfy:*

$$|z_n^{\phi}|_{\mathrm{A}} = \begin{cases} \frac{F_n}{2} : n \bmod 6 = 0, 3, \\ \frac{F_n+1}{2} : n \bmod 6 = 1, 2, \\ \frac{F_n-1}{2} : n \bmod 6 = 4, 5, \end{cases} \quad |z_n^{\phi}|_{\mathrm{C}} = \begin{cases} \frac{F_n}{2} : n \bmod 6 = 0, 3, \\ \frac{F_n-1}{2} : n \bmod 6 = 1, 2, \\ \frac{F_n+1}{2} : n \bmod 6 = 4, 5. \end{cases}$$

Proof. The proof is by induction on n and uses the fact that for all $n \ge 1$, $|z_{n+2}^{\phi}|_{\mathrm{A}} = |z_{n+1}^{\phi}|_{\mathrm{A}} + |z_n^{\phi}|_{\mathrm{C}}, |z_{n+2}^{\phi}|_{\mathrm{C}} = |z_{n+1}^{\phi}|_{\mathrm{C}} + |z_n^{\phi}|_{\mathrm{A}}$, where $|z_1^{\phi}|_{\mathrm{A}} = |z_2^{\phi}|_{\mathrm{A}} = 1$ and $|z_1^{\phi}|_{\mathrm{C}} = |z_2^{\phi}|_{\mathrm{C}} = 0$. Hence, using inductive hypothesis, one can obtain the result for $|z_k^{\phi}|_{\mathrm{A}}$ and $|z_k^{\phi}|_{\mathrm{C}}$. □

We next show that the atom hairpin μ_2-Fibonacci words $z_n^{\mu_2}(\mathrm{A},\mathrm{A})$ are primitive for $n \ge 3$. We need the following lemma, which can be proved by induction on n.

Lemma 4.72. *Let $\phi = \theta_2$. The following hold for all $n \ge 1$:*

1. *If $n \bmod 3 = 0$, then $z_n^{\theta_2}(\mathrm{A},\mathrm{A}) \ne q^2$ for every θ_2-palindrome q.*
2. *If $n \bmod 4 = 0$, then $z_n^{\theta_2}(\mathrm{A},\mathrm{A}) \ne q^3$ for every θ_2-palindrome q.*
3. *If $n \bmod 5 = 0$, then $z_n^{\theta_2}(\mathrm{A},\mathrm{A}) \ne q^5$ for every θ_2-palindrome q.*

Proof. We only prove statement (1). Given that $n \bmod 3 = 0$. Then, $F_n \bmod 2 = 0$, and there exist $x, y \in \Delta^+$ such that $|z_n| = xy$ and $|x| = |y|$. One can easily verify the statement for $n = 3$ and $n = 6$. Assume the statement to be true for $z_i^{\theta_2}(\mathrm{A},\mathrm{A})$, where $i \bmod 3 = 0$ and $3 \le i < k$. Let k be a number such that $k \bmod 3 = 0$ and $k > n$. Suppose $z_k = z_{k-1}\theta_2(z_{k-2}) = p^2$, where p is a θ_2-palindrome. Then, $z_k = z_{k-1}\theta_2(z_{k-2}) = z_{k-2}\theta_2(z_{k-3})\theta_2(z_{k-2})$ and $p = z_{k-2}x = y\theta_2(z_{k-2})$ for $\theta_2(z_{k-3}) = xy$. Since p is a θ_2-palindrome, $x = \theta_2(y)$ and $z_{k-2} = z_{k-3}\theta_2(z_{k-4}) = x\theta_2(x)\theta_2(z_{k-4})$. This implies that $x = y = \theta_2(x)$ and hence $z_{k-3} = x^2$, which contradicts to our induction hypothesis. Hence, the result. □

The following result uses Lemma 4.72 and has a proof similar to that of Proposition 4.57.

Theorem 4.73. *Let $\phi = \theta_2$. For all $n \geq 1$, the atom hairpin ϕ-Fibonacci word $z_n^{\theta_2}(\mathrm{A}, \mathrm{A})$ is primitive.*

Based on Corollary 4.65 and Theorem 4.63, Theorem 4.66, Theorem 4.69, Theorem 4.70, and Theorem 4.73, the results can be generalized to non-trivial ϕ-Fibonacci words with the same initial letters. The primitivity properties of the n^{th} atom ϕ-Fibonacci word with the same two initial letters, for all $n \geq 1$, are summarized in Table 4.14.

	$i \in \{2, 3, 4, 8, 9, 10\}$	$i \in \{1, 5, 6, 7\}$
$g_n^{\phi_i}(\mathrm{A}, \mathrm{A})$	✓	✗
$w_n^{\phi_i}(\mathrm{A}, \mathrm{A})$	✓ (except $n = 3$)	✗
$z_n^{\phi_i}(\mathrm{A}, \mathrm{A})$	✓	✗

Table 4.14: Primitivity of atom ϕ-Fibonacci words $\alpha_n^{\phi}(\mathrm{A}, \mathrm{A})$ for all $n \geq 1$, with identical initial letters $A \in \Delta$, where $\alpha_n \in \{g_n, w_n, z_n\}$ for all $n \geq 1$, and $\phi \in \{\theta_i, \mu_i \mid 1 \leq i \leq 10\}$ (here, ✓ means that the words are primitive, and ✗ means that they are not primitive).

4.8 Conclusion

This chapter proposed a unified terminology (Definition 4.1) for the various definitions of Fibonacci words that exist in the literature. It also defined and investigated two generalizations of Fibonacci words, namely ϕ-Fibonacci words (Definition 4.9) and indexed ϕ-Fibonacci words (Definition 4.28), where ϕ is a morphic or an antimorphic involution on Σ^*.

The results about (θ-)borderedness and primitivity of various involutive Fibonacci words are important since they could lead to a method that generates arbitrarily long DNA words avoiding intramolecular structures. These results are summarized as follows.

According to Theorem 4.33, the standard and reverse hairpin θ-Fibonacci words z_n and z'_n are θ-bordered for $n \geq 6$, and thus form hairpins with fully double-stranded stems. One can easily observe that the standard palindromic θ-Fibonacci words w_n and the reverse palindromic θ-Fibonacci words w'_n form hairpin structures with partially double-stranded stems, as illustrated in Figure 2.6. In addition, according to Proposition 4.36, if the initial θ-Fibonacci words u and v satisfy some additional conditions, then the standard alternating θ-Fibonacci words $g_n(u, v)$ and the standard palindromic θ-Fibonacci words

$w_n(u, v)$ are θ-bordered for $n \geq 3$, and thus they can form hairpins with fully double-stranded stems. Lastly, according to Proposition 4.40 and Proposition 4.41, the standard and reverse palindromic θ-Fibonacci words w_n and w'_n contain θ-palstars for $n \geq 6$, and thus they can self-assemble into DNA secondary structures containing multiple hairpin structures.

According to Section 4.6 and Section 4.7, the primitivity properties of atom involutive Fibonacci words over a four-letter alphabet indicate that, for some (anti)morphic involutions, some initial letters, and some indices n, the n^{th} ϕ-Fibonacci word is primitive, while for some others, it is not. In the particular case of the Watson-Crick complementarity involution θ_{DNA} over the DNA alphabet $\Delta = \{\text{A}, \text{C}, \text{G}, \text{T}\}$, our results imply that regardless of the initial two letters in the Fibonacci recursion (different, or the same), the n^{th} atom Watson-Crick Fibonacci word is primitive for all $n > 3$. Although primitivity does not ensure the avoidance of DNA self-binding, this is an initial step towards the study of θ-primitivity, where θ is an antimorphic involution.

Future topics of research include investigating relations between various types of ϕ-Fibonacci words for the antimorphic case (similar to the results obtained in Section 4.3.1 for the morphic case) and the study of the properties of ϕ-Fibonacci words z_n in the special case when the first two ϕ-Fibonacci words satisfy the relation $\phi(u) = v$. Other areas of investigation include the ϕ-primitivity of ϕ-Fibonacci words, as well as the combinatorial properties of ϕ-Fibonacci words (counting their distinct factors, squares, ϕ-squares, cubes, ϕ-cubes, palindromes, ϕ-palindromes, etc.).

Chapter 5

Complexity Estimation of DNA Computing Algorithms on the Subset Sum Problem

This chapter gives a scaling comparison of two implemented natural computing algorithms (DNA computing using DNA strands and network biocomputing using biotic motile agents) and an electronic computing algorithm to solve the benchmark NP-complete problem, the subset sum problem (SSP). Section 5.1 gives a brief introduction to this chapter. In Section 5.2, details about a DNA computing procedure for SSP experimentally demonstrated in [102] are given. Section 5.3 gives a summary of a network biocomputing method solving SSP experimentally implemented in [202]. In Section 5.5, the pre-computing costs, the volume needed for the computation, the run time, and the energy costs of these three different implementations of algorithms that solve SSP are compared. Section 5.6 concludes this chapter.

This chapter is adapted from the co-authored paper [215] titled "As good as it gets: A scaling comparison of DNA computing, network biocomputing, and electronic computing approaches to an NP-complete problem."

The co-authors of [215] include an experimental team who designed and implemented the compared network biocomputing method for the subset sum problem. The actual implementation was published and discussed in [202, 269].

5.1 Introduction

NP-complete problems are the mathematical representations of many and very diverse real-life applications, such as protein design [217] and folding [80], data clustering in networks [27], circuit verification [196], optimal routing [113], and formal contextual reasoning [183]. Because of the exponential increase of solution space with the problem input size, electronic computers, including high-end supercomputers, cannot solve large instances of NP-complete problems since they operate with limited parallelism at best. Electronic computers face additional difficulties, such as reaching their fundamental physical limits of the gate size, which are already in the couple of nanometers, and engineering-related limits (for example, required energy and heat dissipation). All these difficulties challenge the sustainability of Moore's law [157, 190], which predicts the doubling of computing power approximately every two years. Consequently, alternative computational paradigms capable of massively parallel operations have been proposed since the mid 1990s [3, 4]. Inspired by the natural massive parallelism of biological systems, many of these alternative computational paradigms use biological entities, from molecular to cellular, *in vitro* or *in vivo*, to solve computational problems. The large variety of biocomputation approaches (see Table 5.1 for a summary), combined with the fact that they are not yet standardized, precludes a direct comparison of their respective scalability, as well as a comparison with the "classical" electronic computing approach.

This chapter compares the key performance parameters of three fundamentally different computing approaches: two massively parallel biocomputation approaches (the DNA computing [DNA-C] [3] and the more recently reported network biocomputing using motile agents [NB-C] [202,269]) and the classical electronic computing (E-C) approach. To achieve a fair analysis, all these computing approaches were employed to solve the same benchmark NP-complete problem, the subset sum problem (SSP).

In comparing the DNA-C, NB-C, and E-C approaches for solving SSP, the key operating parameters surveyed comprised the pre-computing costs (Section 5.5.1), the volume needed for computation (Section 5.5.2), the run time (Section 5.5.3), and the energy costs (Section 5.5.4). All were calculated using the experimental parameters and estimation methodology previously reported in [102, 202, 269].

Before we go further, a detailed definition of SSP is discussed. As mentioned in Section 2.3, SSP (see [254]) asks the following yes/no question: "Given a finite set $S \subset \mathbb{N}$ of positive integers and a positive integer target number $t \in \mathbb{N}$, does there exist a subset $S' \subseteq S$ such that $t = \Sigma_{s \in S'} s$?" Note that other variants of SSP allow the input set S to be a multiset, which allows repetitions of its elements. An instance of SSP consists of a particular set S and a particular number t, and the size of the instance is defined as the cardinality of the input set S. Note that each instance of SSP has a yes/no answer. For example, for the SSP instance of size 3 with input set $S = \{2, 3, 9\}$ and target number $t = 5$, the answer is "yes", and the subset of numbers that add up to $t = 5$ is $S' = \{2, 3\}$ (the "yes" answer is the solution to this instance of the problem). In computer science, solving SSP means finding an algorithm that outputs the correct solution (yes/no answer) for all instances. In the case of a "yes" answer, there is the optional step to identify the subsets that fulfill the required condition.

Agent	Problem	Reference
DNA	Hamiltonian path problem	[3]
	Maximal independent set	[97]
	Minimal dominating set	[103]
	Knapsack/subset sum problem	[7,102,257]
	Satisfiability	[25,26,169,242,260,273]
	Maximal clique problem	[207]
	Maze	[39]
Motor proteins	Subset sum problem	[202]
Bacteria	Hamiltonian path problem	[13]
	Maze	[209,214]
Slime mold	Steiner tree problem	[168,194]
	Travelling salesman problem	[131]
	Maze	[195]
Fungi	Maze	[94,98]
C. elegans	Maze	[228]
Bees	Travelling salesman problem	[32]
	Maze	[285]
Ants	Travelling salesman problem	[67]
Other multi-cellular organisms	Maze	[23,205,216,236,264]

Table 5.1: Natural computing approaches to solving instances of NP-complete problems and their related optimization problems. Biological agents used range from DNA molecules to multi-cellular macroscopic organisms. Only biocomputations that have been experimentally implemented are included.

5.2 DNA computing procedure for solving SSP

DNA computing overview was given in Section 2.2. Existing DNA-C procedures for solving SSP can be categorized as either theoretical DNA algorithms with no experimental implementation [38,213] or DNA algorithms with a wet lab experimental implementation [7,102]. The latter are based on the idea of expressing each subset S' of the input set S as a unique path in a special type of directed weighted graph, as illustrated in Figure 5.1.

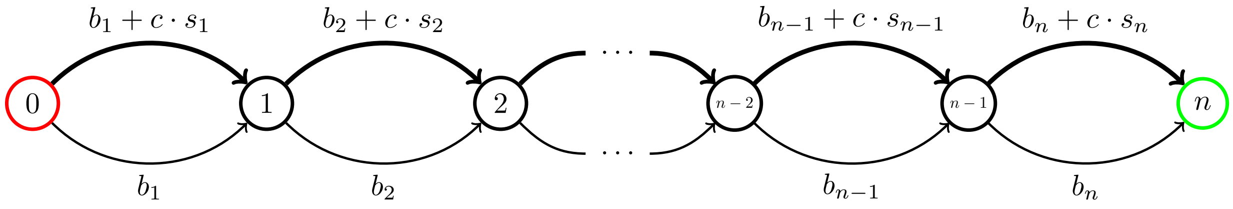


Figure 5.1: An SSP instance of size n with input set $S = \{s_i \mid 1 \leq i \leq n\}$ represented by a directed weighted graph with $(n+1)$ nodes and designated start and end vertices. In this graph, each subset of S corresponds to a unique path from the start node (red) to the end node (green), and vice versa. Note that every such path has to pass through all the nodes of the graph. If, when connecting node $(j-1)$ with node j, $1 \leq j \leq n$, the path traverses the top edge (thick line), this indicates that the subset it represents contains the number s_j. If, instead, the path traverses the bottom edge (thin line), this indicates that the subset it represents does not contain the number s_j. The edges are weighted, with the weights of the top and bottom edges being $(b_i + c \cdot s_i)$, and respectively b_i, for $1 \leq i \leq n$.

If the input set is $S = \{s_1, \ldots, s_n\}$, the corresponding graph has $n+1$ nodes, a designated start node labelled 0, and a designated end node labelled n. For each $1 \leq j \leq n$, the node labelled j corresponds to the number s_j in S, and two consecutive nodes are connected by exactly two directed edges. Each path between node 0 and node n uniquely represents a subset S' of S, as follows. The presence or absence of a number s_j in S' is indicated by the path representing S' traversing either one, or the other (but not both), of the two edges that connect node $(j-1)$ to node j: If s_j is in S', the path traverses the "top" edge, and if s_j is not in S' then the path traverses the "bottom" edge instead (see Figure 5.1). In addition, the weights of the two edges that connect consecutive nodes are different, so that the length of each start-to-end path (the sum of the weights of its edges) corresponds to the sum of the numbers in the subset that it represents.

This conceptual design can be physically implemented by first encoding each directed weighted edge as a DNA strand of a certain length. This is followed by the generation of a combinatorial library of DNA strands encoding all possible start-to-end paths (concatenations of edges/strands) through the graph, which in turn represent all possible subsets of S. The next step is the physical extraction, from this library, of the DNA strands of the desired length representing the solutions to the SSP instance. The length of such an SSP-solution-representing DNA strand is determined based on the target number t specified by that SSP instance. Note that, in practice, experimental considerations dictate the introduction of additional parameters c and $b_1, \ldots, b_n$, whereby the weight of the edge signaling the presence of s_i in a subset is $(b_i + c \cdot s_i)$, while the weight of the edge signalling the absence of s_i in a subset is b_i. For example, the set S itself is represented by the start-to-end path that traverses all the "top" edges in the graph, and the DNA strand that encodes it has the expected length $\sum_{1 \leq i \leq n}(b_i + c \cdot s_i)$. In contrast, the empty set $\emptyset$ is represented by the start-to-end path that traverses all the "bottom" edges in the graph, and the (much shorter) DNA strand that encodes it has the expected length $\sum_{1 \leq i \leq n} b_i$. For a target number t, a subset S' of S whose elements sum up to t exists, if and only if there exists a start-to-end path in the graph encoded by a DNA strand of length $\sum_{1 \leq i \leq n} b_i + c \cdot t$.

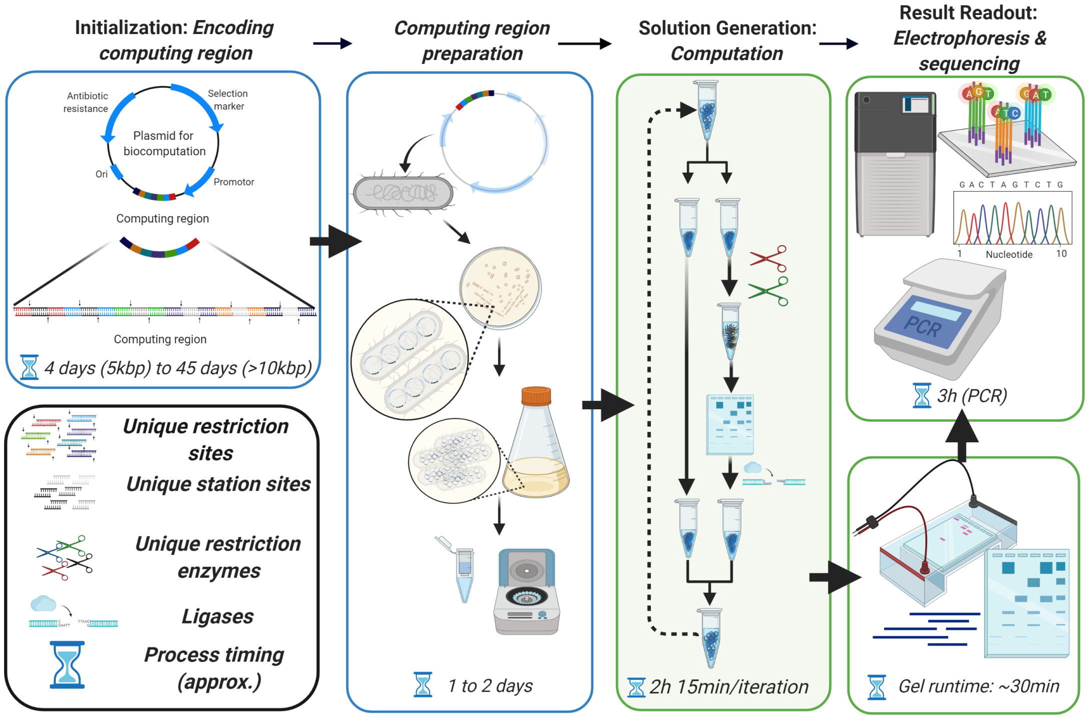
Both [102] and [7] are based on the concept above, with [7] solving an instance of SSP of size $n = 3$ with input set $S = \{2, 3, 4\}$, target number $t = 5$, and parameters $c = 10$, $b_1 = 25$, and $b_2 = b_3 = 20$, and with [102] solving two instances of SSP of size $n = 8$ and input set $S = \{21, 45, 36, 51, 36, 36, 36, 36\}$, one with target number $t = 104$ and another with target number $t = 174$, and parameters $c = 1$ and $b_i = 6$ for $1 \leq i \leq 8$.

The wet lab experimental details of the more recent of the two DNA-C procedures for SSP in [102] are described further. For a direct comparison with NB-C, this DNA-C procedure has to be applicable also to SSP instances with small numbers in the input set (in particular, numbers smaller than the length of the restriction sites of the enzymes used in the DNA computation). To this end, in the description below, the parameters $c = b_i = k$, where $1 \leq i \leq n$, are used, where k is the length of the restrictions sites of all enzymes employed in the experiment.

The DNA-C procedure for solving SSP comprises three steps: the pre-computing step, the solution generation step, and the result readout step.

In the pre-computing step, the natural numbers comprising input set $S = \{s_1, \ldots, s_n\}$ of an SSP instance of size n are encoded into a so-called "computing region of S" that is inserted into a plasmid (a circular DNA double strand), as follows (Figure 5.2, left panel, top). To each number s_i, $1 \leq i \leq n$, one associates a restriction enzyme e_i with restriction site r_i of length k. In addition, two unique restriction enzymes e_{start} and e_{end} are used, with restrictions sites r_{start} and r_{end} (also of length k) as delimiters of the computing region of S

as defined later on. The enzymes are chosen so that all restriction sites are different from each other. Each restriction enzyme e_i, $i \in \{1, 2, \ldots, n\} \cup \{\text{start}, \text{end}\}$, cuts according to a specific pattern, and its restriction point (the location of its cut) is after the k_i^{th} base of its restriction sequence from the $5'$ end. Also, in this experiment, all restrictions sites are palindromic, so k_i uniquely describes the cuts on both strands.



Initialization: Encoding computing region
Computing region preparation
Solution Generation: Computation
Result Readout: Electrophoresis & sequencing
Antibiotic resistance
Selection marker
Plasmid for biocomputation
Ori
Promotor
Computing region
Computing region
4 days (5kbp) to 45 days (>10kbp)
Unique restriction sites
Unique station sites
Unique restriction enzymes
Ligases
Process timing (approx.)
1 to 2 days
2h 15min/iteration
PCR
3h (PCR)
Nucleotide
Gel runtime: ~30min

Figure 5.2: The DNA-C procedure in [102] for solving an instance of SSP. Left panel, bottom: Visual figure legend. Left panel, top: The natural numbers comprising the input set $S = \{s_1, \ldots, s_n\}$, of an SSP instance of size n, are encoded into a "computing region of S" inserted into a plasmid. Centre-left panel: The plasmid is amplified by cloning. Centre-right panel: The pool comprising all DNA strands representing candidate SSP solutions is generated, by a succession of split-and-merge substeps, using restriction enzyme digestions, purifications by gel electrophoresis, and ligations. Right panel, bottom: The DNA strands representing the candidate SSP solutions are length-separated by gel electrophoresis. Right panel, top: The DNA sequences of the desired length are extracted, amplified by PCR (optionally), and sequenced.

Next, for each number s_i, $1 \leq i \leq n$, a DNA strand of length $k + (k \cdot s_i - k) + k = k \cdot (s_i + 1)$, called a *station*, is designed, consisting of the concatenation of three sequences: the recognition site r_i, a specially designed "middle sequence" of length $k \cdot s_i - k$, and a second copy of r_i.

In the DNA-C procedure for SSP in [102], the enzyme e_{start} is *XbaI* and the enzyme e_{end} is *HindIII*, with the respective restriction sites shown in Figure 5.3, with $k_{\text{start}} = k_{\text{end}} = 1$.

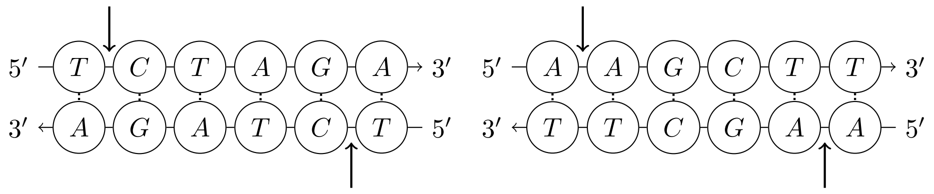


Figure 5.3: Restriction sites of restriction enzymes XbaI (left) and HindIII (right).

The input set S is encoded by a DNA double strand called the *computing region of* S, which consists of the concatenation of the recognition site r_{start}, one station for each of the input numbers s_i, $1 \leq i \leq n$, and the recognition site r_{end}. Note that the recognition sites and middle sequences are carefully designed so that r_{start} and r_{end} occur exactly once (at the beginning and end of the computing region of S, respectively), and each r_i, $1 \leq i \leq n$, occurs exactly twice in the computing region of S (once at the beginning and once at the end of the station for s_i).

Using this encoding scheme, a DNA strand representing the computing region of the input set S is synthesized and inserted into a base plasmid containing one copy of the recognition site r_{start} and one copy of the recognition site r_{end} (with no overlap), using the respective restriction enzyme digestions and ligations. Care must be taken with the encodings and choice of plasmid, so the restriction sites occur only in the designated places. As shown in the centre-left panel of Figure 5.2, this base plasmid is then inserted into bacteria, such as E. coli, and amplified (exponentially multiplied) to the amount necessary for the ensuing DNA computation. The generated plasmids are then extracted from bacteria and transferred to a test tube.

In the solution generation step, the goal is to generate the space of all potential solutions for this SSP instance by creating all different plasmids with a computing region representing a subset of the input set. The *computing region of a subset* S' of S consists of the concatenation of the restrictions site r_{start}, followed by the ordered concatenation of n DNA strands, each representing the presence or absence of a number s_i in S', and followed by the restriction site r_{end}. More precisely, if the number s_i belongs to S' then the station of s_i (which includes two copies of r_i) is present at the i^{th} location in the computing region of S', while if s_i does not belong to S' then a single copy of r_i is present at the i^{th} location in the computing region of S'. The length of the DNA double strand encoding the computing region of a subset S' is:

$$L_{DNA}^{S'}(n) = k + k \cdot (n - |S'|) + \sum_{s_j \in S'} k(s_j + 1) + k \,(\text{bp}).$$

The generation of the combinatorial library of solution candidates can now proceed by a succession of split-and-merge substeps, as follows (Figure 5.2, centre-right panel). The content of the initial test tube T, containing the plasmid with the computing region of S, is evenly split into two tubes T_1 and T_2. Next, while tube T_2 remains intact, the plasmids in tube T_1 are digested by the restriction enzyme e_1 in order to cut out the station representing the number s_1. The resulting linear DNA molecules are ligated back into circular DNA molecules (after purification by gel electrophoresis to separate and extract the DNA strands encoding the station representing s_1). Following this, the contents of the new T_1 and T_2 are merged into a tube T', which now comprises plasmids of two types: half of the plasmids have a computing region of subsets containing s_1, and the other half of the plasmids have computing regions of subsets that do not contain s_1. This succession of split-and-merge substeps is repeated for each of the remaining numbers in S, resulting in a final tube that contains a combinatorial library of plasmids with computing regions spanning all possible subsets of S.

In the result readout step, the possible solution candidates are separated by high resolution electrophoresis (e.g., non-denaturing polyacrylamide gel electrophoresis, PAGE, or 2% agarose gel electrophoresis), and the existence of an SSP solution is determined by the presence or absence of a band on the gel corresponding to the desired DNA strand length (Figure 5.2, right panel, bottom). Optionally, the DNA strands in the band indicating a solution can then be sequenced (Figure 5.2, right panel, top).

More precisely, the plasmids in the final test tube are first cut by the restriction enzymes e_{start} and e_{end}, resulting in linear DNA molecules representing all possible subset sum combinations for the set S. If a solution to the given SSP instance exists, that is, if there exists a subset S' of numbers in S that contains numbers adding up to the target number t (that is, $\sum_{s_j \in S'} s_j = t$), then a band with DNA strands of length

$$L_{\text{DNA}}^{S}(n) - k_{start} - k_{end} - \sum_{s_j \in S \setminus S'} ks_j = L_{\text{DNA}}^{S}(n) - k_{start} - k_{end} - [(\sum_{s_i \in S} ks_i) - kt] \,(\text{bp})$$

will be detected on the gel, and the answer to this instance of SSP is "yes." In this case one can, optionally, sequence each DNA strand representing an SSP solution, to identify the corresponding subset of numbers in S that sum up to t (there can be several such subsets). To determine such a subset S', for every number s_i, $1 \leq i \leq n$, its presence or absence in S' can be determined by checking the number of occurrences of the recognition site r_i in the DNA sequence representing S' (two occurrences of r_i if $s_i \in S'$, and one occurrence of r_i if $s_i \notin S'$).

Alternatively, if there is no band on the gel corresponding to the aforementioned expected length, then the answer to this instance of SSP is "no."

5.3 Network biocomputing (NB-C) for SSP

Various implementations of massively parallel computation using motile agents were proposed to solve NP-complete problems. Agents can be abiotic, such as photons [204, 277] and beads [114, 124]. NB-C uses biological agents such as cytoskeletal filaments (actin filaments or microtubules) propelled by protein molecular motors (myosin or kinesin) [200, 202] and microorganisms [98, 197, 214, 270]. Exploring computational networks requires that the motile agents are self-propelled, independent of each other, easily visible, reasonably small, and able to move at a high velocity. The large variety of motile biological agents offers the opportunity of selecting those that fulfil these requirements.

Unlike electronic computers, which perform operations *in silico*, NB-C uses motile biological agents that explore and perform "operations" in a physical space, designed according to the problem to be solved. These special features of NB-C have interesting consequences for the scaling of computation.

The following discussion focuses on the details of the implementation of NB-C to solve SSP [202, 269]. It comprises the pre-computing step, the solution generation step (where motile biological agents explore the network), and the result readout step.

In the pre-computing step, the input information is encoded as a physical network. The design of the network that encodes SSP instances was first reported in [201] and refined in [202], where the input is encoded by a triangular shape network made of junctions. There are two types of junctions used: pass junction and split junction, as illustrated in Figure 5.4.

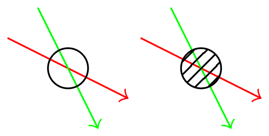


Figure 5.4: Two types of junctions used in the network encoding of the subset sum problem. There are two routes, illustrated by a red arrow and a green arrow, through a junction. Left: A pass junction is represented by an empty circle. If a motile agent enters the junction from a route, it will exit following the same route. Right: A split junction is represented by a shaded circle. If a motile agent enters the junction from a route, it has a 50 % chance of exiting via the same route and a 50 % chance of exiting via the other route.

If the input set is $S = \{s_1, \ldots, s_n\}$, the corresponding network has $1 + \sum_{1 \le i \le n} s_i$ rows of junctions, and the i^{th} row has i junctions. In addition, all the junctions in the i^{th} row are split junctions if and only if there exists $0 \le j \le n$ such that $i = 1 + \sum_{1 \le l \le j} s_l$, and the split junctions in the i^{th} row correspond to the input number s_{j+1}. The motile agents enter the network by the split junction in the first row, and they exit the network by the split junctions in the last row. Each split junction in the last row corresponds to a number between 0 and $\sum_{1 \le i \le n} s_i$. The design of the network ensures that agents exit a split junction in the last row if and only if there exists a subset of the input set that sums to the number corresponding to the junction. Each path from the entrance junction to an exit junction represents a subset of the input set, and the sum of this subset is the number that corresponds to the exit junction. This path includes a red route out of a split junction if and only if the corresponding number from the input set is included in the subset. For example, the network encoding of an instance of SSP with input set $S = \{2, 5, 9\}$ used in [202] is illustrated in Figure 5.5.

The dimensions and optimizations of the designed network are related to the particularities of the biological agents used. The size of a junction in the network for different biological agents is described in a previous study [269]. For example, if protein molecular motor-propelled cytoskeletal filaments are chosen, the network channel dimensions and surface modifications will be tailored to accommodate agents to freely traverse the network [202]. The fabrication of the networks can be achieved by PDMS-based soft lithography, low resolution SU-8-based optical lithography, and e-beam lithography, where a higher resolution is required for the networks designed for smaller agents, such as cytoskeletal filaments. Furthermore, for bacterial-driven NB-C, the networks need to be able to provide nutrients for bacteria, while for cytoskeletal filaments-driven NB-C, the networks need to be able to provide ATP.

In the solution generation step, all possible subsets are generated by biological agents traversing all different paths from the entrance junction to an exit junction. For example, in Figure 5.5, the subset sum numbers associated with an exit junction that an agent can traverse to from the entrance junction is colored blue. In addition, a path representing the subset $\{5\}$ is shown in Figure 5.5, where it takes the green routes from the first and third split junctions (2 and 9 are not included) and the red route from the second split junction (5 is included), and it exits the network from the junction representing the subset sum 5.

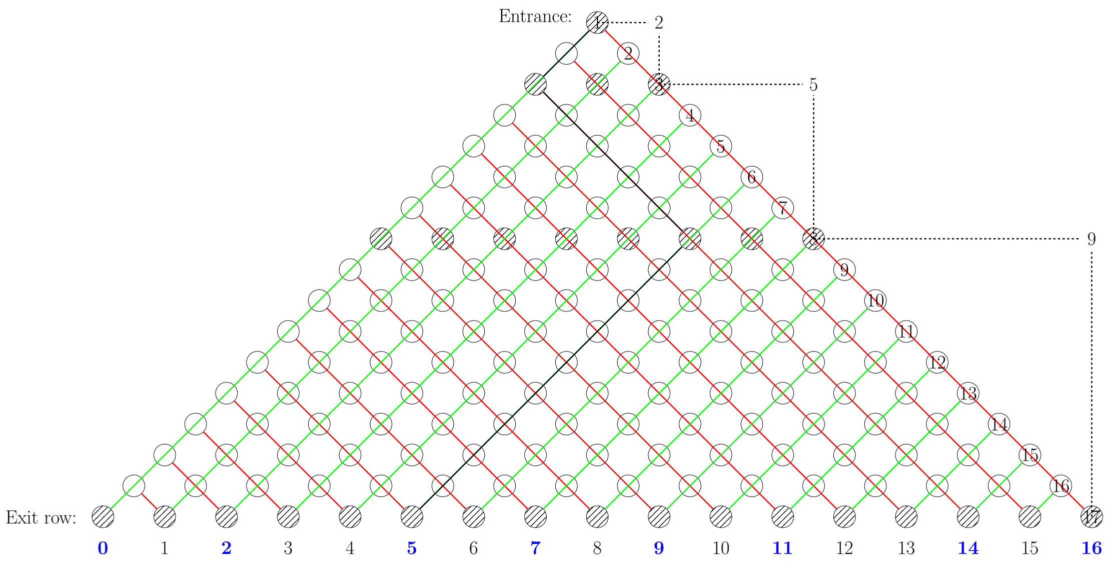


Figure 5.5: Network encoding of an instance of the subset sum problem with the input set $S = \{2, 5, 9\}$. Split junctions are represented by shaded circles, and pass junctions are represented by empty circles. The red route of a junction is directional from top left to bottom right, and the green route of a junction is directional from top right to bottom left. The parts of routes that are not connected to other junctions are omitted. There are 17 rows, and the i^{th} row from the top contains exactly i junctions. All the junctions in the 3rd $(1 + 2)$, 8th $(1 + 2 + 5)$, and 17th $(1 + 2 + 5 + 9)$ rows are split junctions. The split junction in the 1st row is the entrance of the network, and the split junctions in the last row are the exits of the network, each of which represents a potential subset sum of the input set. The possible subset sums are labelled by blue numbers, and there will be agents exiting the corresponding split junctions in the last row. A path that represents the subset $\{5\}$ with sum 5 is represented by black lines.

The network is called a *class II* network if there is a one-to-one correspondence between subset sums and paths from the entrance junction to an exit junction, and it is called a *class I* network otherwise [269]. For example, the network in Figure 5.5 is class I, but the network encoding the input set $S = \{2, 3, 5\}$ is class II because the subset that sums to 5 can be either $\{2, 3\}$ or $\{5\}$.

In the above model, it is assumed that the biological agents go through pass and split junctions perfectly according to the design. However, in experiments, physical factors like biased turn preferences and interactions with the boundaries of the channels can influence the binary decision. For instance, when bacteria explore a network, their ability to choose a route is modulated by chemotactic cues, nutrient foraging, flagellum/a architecture, and interactions with surrounding walls [98, 99, 214].

In the result readout step, the existence of an SSP solution can be verified by checking whether there are any agents exiting the exit junction that represents the target number. In addition, because images of the whole network were taken during the solution generation step, the composition of the subsets that represent SSP solutions can (optionally) be obtained by analyzing the traces of the agents. The trajectories of the agents in the network can be recorded using an optical interface, such as a microscope. The various limits of the optical readout interfaces were reviewed recently, specifically the various limits of the field of view on NB-C [269]. A timestamped image recording, necessary to obtain a reliable readout from agent-based computation [197, 214], comprises image acquisition, image post-processing, and agent tracking. Image processing tools, such as ImageJ Fiji [1, 263], and tracking plugins, such as Track Mate (semi-automatics) [186, 263] and MTrackJ(manual) [186, 249], will process the recorded frames of agent movement in the network. The deciphered tracks and the density maps encode the solutions, and the readout can be achieved using the following approaches.

- Density maps and backtracking: The sum of all the agent trajectories is projected as a heat map, i.e., spatially distributed frequencies of agent locations. Depending on the color scheme used, the most taken paths (correct solutions) are brightly colored or of the highest saturation, while the least taken paths (incorrect solutions) are dark or of minimum saturation (alternatively, the heat maps can be interrogated for their numerical values too). These density maps are used as a qualitative measure and a form of quick parallel readout to arrive at the solutions for a particular set. Although quick, this methodology only delivers the target sums, i.e., the existence of a solution to SSP. The combination of the target sum and the multiple routes for the same exit for complexity class II type SSP can be derived by backtracking.

- Agent counting at the exits of the network: Another methodology is based on counting the number of agents at the exits. This methodology, demonstrated for cytoskeletal filaments-driven NB-C [202], translates to a bar chart with a distribution of agents with the highest relative count for correct exits versus the lowest relative counts for agents exiting incorrect exits.

- Continuous tracking of the agents from the entry to the exit: Another methodology could be based on single-particle continuous tracking, either by manual, semi-automatic, or automatic methods, which can be adapted to track the agent movement in the computation network, thus recording the visited junctions encoding a particular solution.

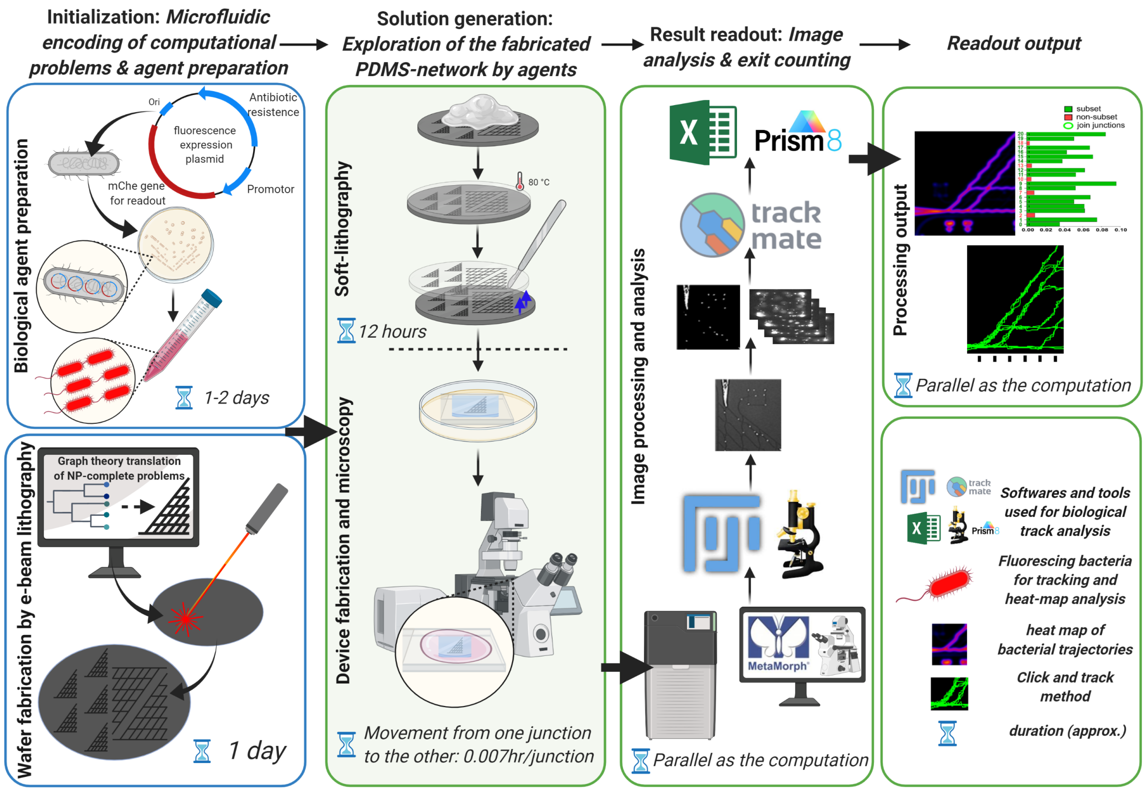

Initialization: Microfluidic encoding of computational problems & agent preparation
Solution generation: Exploration of the fabricated PDMS-network by agents
Result readout: Image analysis & exit counting
Readout output
Biological agent preparation
Ori
Antibiotic resistence
fluorescence expression plasmid
mChe gene for readout
Promotor
1-2 days
Wafer fabrication by e-beam lithography
Graph theory translation of NP-complete problems
1 day
Soft-lithography
80 °C
12 hours
Device fabrication and microscopy
Movement from one junction to the other: 0.007hr/junction
Image processing and analysis
Prism8
track mate
MetaMorph
Parallel as the computation
Processing output
subset
non-subset
join junctions
Parallel as the computation
Softwares and tools used for biological track analysis
Fluorescing bacteria for tracking and heat-map analysis
heat map of bacterial trajectories
Click and track method
duration (approx.)

Figure 5.6: The model flow chart of agent-based NB-C and the associated stages. The figure here gives a bird's-eye view of the computational operations like exploring the network by bacteria and pre- and post- computational steps like network design and fabrication, bacterial preparation and culture for experimentation, results readout, image analysis, and the associated required duration.

In addition to these already demonstrated methodologies, several more elaborate approaches have been proposed. Examples include the switchable tagging of biological agents "on the fly," which would allow the computational trajectories being stored in the agents as transient or permanent memory [269].

A comprehensive list of experimental procedures and the time taken to solve each stage of the agent-based computation is represented in Figure 5.6. Numerical calculations, such as the total number of agents required to solve SSP instances of different sizes (cardinality), and the network sizes comparison were reproduced, with some modifications, as reported in [269]. The applied modifications to previous scaling analysis stem from considering different input sets (i.e., unit, prime, Fibonacci, and exponential), as introduced in Section 5.5.

5.4 Electronic computing (E-C)

A review of the electronic computing is beyond the scope of this contribution. However, a balanced comparison of the projected capacities of DNA-C and NB-C to solve NP-complete problems will benefit from benchmarking it against the equivalent projected performance of electronic computers.

Electronic computers are essentially sequential machines (multi-core computers feature bounded parallelism, at best), and therefore they are inherently challenged by the exponential increase of the number of possible solutions with the size of the NP-complete problem. However, electronic computers do present important advantages in tackling NP-complete problems. First and foremost, the speed of computation is presently running in the billions of operations per second. Presently, AMD Threadripper 3990X, one of the fastest commercial chips, performs approximately 2.3 million MIPS (million instruction per second) at 4.35 GHz [249]. Second, after more than half a century of technology development following Moore's law [190, 191], electronic computer chips benefit from a deep, large, and fully functional ecosystem, including manufacturing and performance standards, business networks, and last but not least, a large body of elaborate mathematical algorithms and information processing protocols. Third, despite reports of a slow-down in technology development and of the "end of Moore's law" [70, 123], the semiconductor industry continues to find ways for dramatic improvements.

To ensure a correct comparison between the performance of DNA-C, NB-C, and the benchmark electronic computers with regard to their performance in solving SSP, the latter must perform the calculations by brute force, similar to the former two massively parallel computation approaches. The basic methodology to solve SSP by various generations of electronic computer chips was described in [269]. Briefly, a computer program to solve SSP was developed in the C programming language to enable low-level memory access, efficient mapping to machine instructions, and flexibility. Out of several algorithms to solve SSP, a naïve, brute force approach was adopted. The SSP algorithm was designed to explore all $\Sigma_{0 \leq k \leq n} \binom{n}{k} = 2^n$ subsets, each of which contains at most n elements. Thus, the running time is of the order $O(n \cdot 2^n)$.

Due to random access memory (RAM) and clock speed being the major factors affecting CPU speed, we replicated the computing resources of Intel 286, Intel 386, Intel 486, and Intel Pentium Pro by simulating part of their computer hardware with virtual machines. However, these calculations were dependent on chip power, and consequently the fastest chip could not solve an SSP instance larger than $n = 50$, for the prime input set. Fortunately, the computing time was found to be in a near perfect relationship with the technical parameters of the chips, i.e., MIPS and clock frequency, and this allowed the extrapolation of the computing time for solving SSP at higher cardinalities and for more advanced computer chips (AMD Threadripper 3990X).

5.5 Scaling comparison of the DNA-C, NB-C, and E-C methods solving SSP

This section comprises a detailed scaling comparison of three qualitatively different methods for solving SSP: the DNA-C procedure of [102] described in Section 5.2, the NB-C method of [202, 269] described in Section 5.3, and the classical E-C implementation of the (sequential) exhaustive search algorithm for SSP. As comparison benchmarks, four different types of input sets are considered, drawn from the following sequences of positive integers. The unit sequence is $\{a_i\}_{i \geq 0}$, where $a_i = 1$ for all $i \in \mathbb{N}$, the prime sequence $\{p_i\}_{i \geq 0}$ consists of all the prime numbers in ascending order, the Fibonacci sequence is $\{f_i\}_{i \geq 0}$, where $f_0 = f_1 = 1$ and $f_{i+2} = f_{i+1} + f_i$ for all $i \in \mathbb{N}$, and the two-exponential sequence is $\{\exp_i\}_{i \geq 0}$, where $\exp_i = 2^i$ for all $i \in \mathbb{N}$. The *unit set* of cardinality n is now defined as comprising the first n elements of the unit sequence, and the *prime set*, the *Fibonacci set*, and the *two-exponential set* (or, simply, *exponential set*) of cardinality n are similarly defined. For example, for cardinality $n = 7$, the unit set is $\{1, 1, 1, 1, 1, 1, 1\}$, the prime set is $\{2, 3, 5, 7, 11, 13, 17\}$, the Fibonacci set is $\{1, 1, 2, 3, 5, 8, 13\}$, and the exponential set is $\{1, 2, 4, 8, 16, 32, 64\}$. Using these sets as inputs for instances of SSP, the DNA-C and NB-C methods for solving the SSP problem are now compared, at different cardinalities, as follows. Section 5.5.1 discusses the pre-computing step for both methods: the synthesis and length of DNA strands utilized in the case of DNA-C method, and the network fabrication details in the case of the NB-C method. Section 5.5.2 compares the physical volume of the computation for each method, while Section 5.5.4 compares the energy needed for computation, and Section 5.5.3 compares the running time of both biocomputations.

5.5.1 Pre-computing

Before considering the "core" operational parameters of the three computational procedures considered here, it must be observed that the classical electronic computers are, following decades of development and standardization, "ready-made hardware" in that no substantial pre-computing procedures need to be performed. Furthermore, in most instances electronic computers utilize pre-loaded algorithms, possibly including those solving special cases of NP-complete problems.

In contrast, non-classical computing methods such as DNA-C and NB-C are characterized by a lack of standardization and usually offer *ad-hoc*, problem-dependent, solutions (possibly involving the fabrication of new "hardware" for each problem). Consequently, while this state of affairs is expected to improve with further development of non-classical computing, a thorough scaling comparison and analysis cannot ignore the necessary pre-computing step for the DNA-C and NB-C methods.

In the case of DNA-C procedure, the implementation of the pre-computing step described in Section 5.2 comprises the synthesis of the DNA strands representing the computing region of the input set S, followed by their insertion into plasmids, and by the amplification of the plasmids containing the computing region of S. As detailed in Section 5.2, for an SSP instance of size n with input set $S = \{s_1, s_2, \ldots, s_n\}$, the length of the DNA strand representing the computing region of S is $L^S_{\text{DNA}}(n)$. Figure 5.7 illustrates the lengths of computing regions of the unit, prime, Fibonacci, and exponential input sets of various cardinalities, if the length of all restriction enzyme recognition sites is $k = 6$, as chosen in [102]. Using these calculations, and taking 10 kbp as the maximum length of synthesizable DNA strands (see Section 2.2), it follows that the largest input sets that can be encoded using this DNA-C procedure and current technology are input sets of cardinality $n = 1{,}665$ for unit sets, $n = 30$ for prime sets, $n = 15$ for Fibonacci sets, and $n = 9$ for exponential sets.

The possible rate-limiting factors for DNA-C include:

- Despite the high-fidelity DNA-polymerase based amplification, random errors still occur [11, 171].
- DNA fragment hybridization mismatches [8, 87].

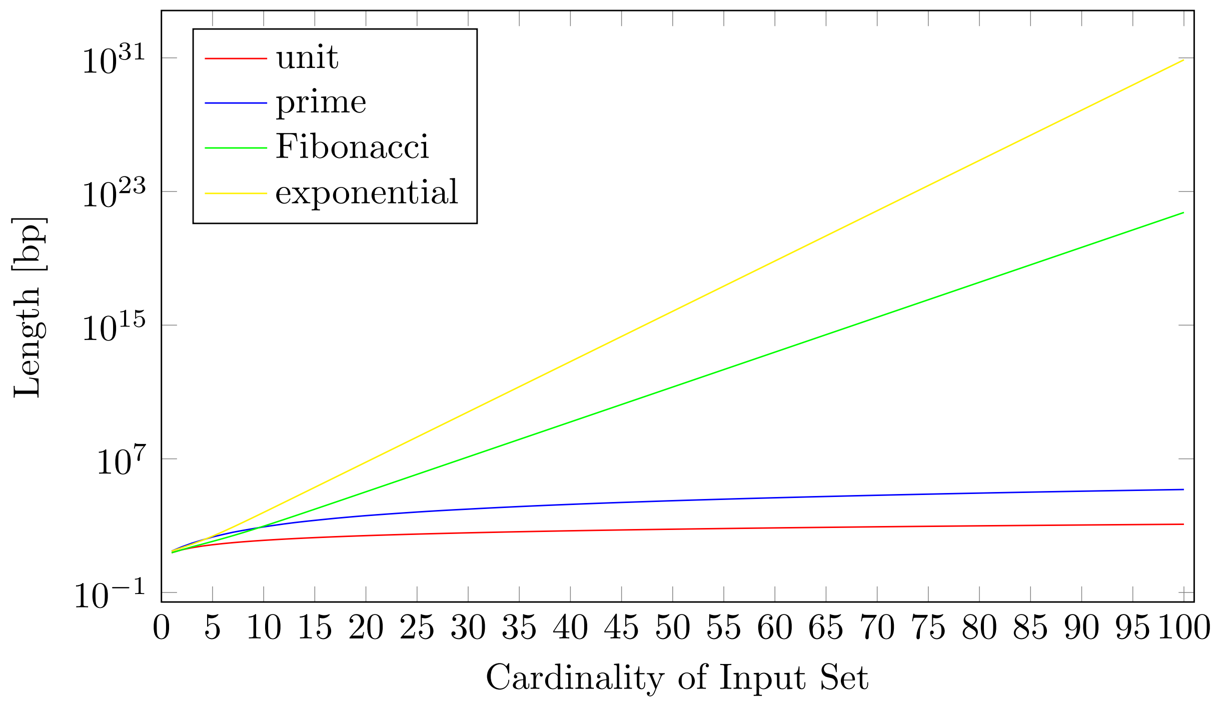


Figure 5.7: Logarithmic scale graph illustrating the length of the DNA strand that encodes the computing region of the input set in the DNA-C procedure in [102], for the unit, prime, Fibonacci, and exponential input sets of different cardinalities, from $n = 1$ to $n = 100$. Note that the length of the computing region of the input set grows exponentially (linear growth, logarithmic scale) for exponential and Fibonacci input sets and grows linearly for unit input sets (quasi-constant growth, logarithmic scale).

- Metastable DNA hybrid structures, e.g., hairpin loops [77, 170, 284].

However, these errors can be mitigated with optimized protocols, e.g., variation of temperature and use of specific enzymes relaxing the hairpin loops and other metastable structures [12]. Additionally, the use of New Generation Sequencing can help the quantification of errors and provide feedback loops for protocol optimization. Consequently, considering these achievable optimization paths, the scaling estimations regarding DNA-C procedures consider only the optimal properties of the DNA strands.

Besides the length of the computing region of the input set, which determines an upper limit on the size of the SSP instance solvable by this DNA-C procedure, another limitation on the pre-computing step is the number of restriction enzymes available. This is because encoding of an input set of cardinality n requires $(n + 2)$ restriction enzymes (one for each number in the input set, and one for each end of the computing region of S). Each enzyme must have a distinct recognition site, and all restriction sites have to be of the same length k. If $k = 6$, as chosen in [102], the number of known enzymes with different recognition sites of this length is at most 47 (see [235]), and this becomes an additional upper limit for the maximum size of SSP instance solvable by this DNA-C procedure. Note the constraints on the design of viable number-encoding stations, and the limitations on the length of DNA strands that can be synthesized (discussed in Section 2.2), which could lead to further limiting the maximum size of the SSP instance that can be solved by this technique.

In the case of the NB-C method, the scaling analysis will use the bacteria-operated version because it offers more opportunities for insight, especially due to the possibility of agent multiplication. In general, NB-C also needs two separate pre-computing modules: (i) micro- or nano-fabrication of the computational network using photolitography or electron-beam lithography, followed by semiconductor manufacturing proper for the computational device for cytoskeletal filaments-driven NB-C, or for the master mold for microorganisms-driven NB-C, followed in the latter case by PDMS-based soft lithography [100, 202]; (ii) genetic engineering of the bacterial strain with fluorescence expressing plasmids [197, 214]. The split and pass junctions have an average diagonal length of 106.3 µm for a bacterium agent of an average size of 0.5 µm · 1.0 µm. The diagonal length may vary depending on the bacterial or the motile agents used for computation. The fabrication time for the network is computed based on the total number of junctions in each input set. A nanometer-precise fabrication is necessary for finer cytoskeletal filaments version [200, 202], but for e-beam lithography, a higher resolution, coupled with larger channel dimensions, results in a longer fabrication time per unit cell. Another limiting factor is the largest wafer size available to accommodate the computational network, with the most used wafer diameter being 8 inches. Figure 5.8 presents the number of unit cells required for a specific cardinality; the fabrication time is proportional to this number. As for NB-C analysis, the pre-computation processes, such as network fabrication and the mass production of bacteria, are not included in the assessment of the computing time needed to solve SSP for various cardinalities. The preparation of bacteria for NB-C proceeds once per computation, with a duration allowing several of these preliminary procedures within 12 hours. Because the NB-C network encodes a brute force mathematical algorithm for solving SSP, once the key parameters of the geometry are acquired, e.g., number, type, position of the nodes, length and widths of the channels, templates for post-processing rectifiers, and "ghost lanes" [270], the actual design of the planar layout of the biological agents-driven computer can progress entirely automatically, e.g., using lithography design software.

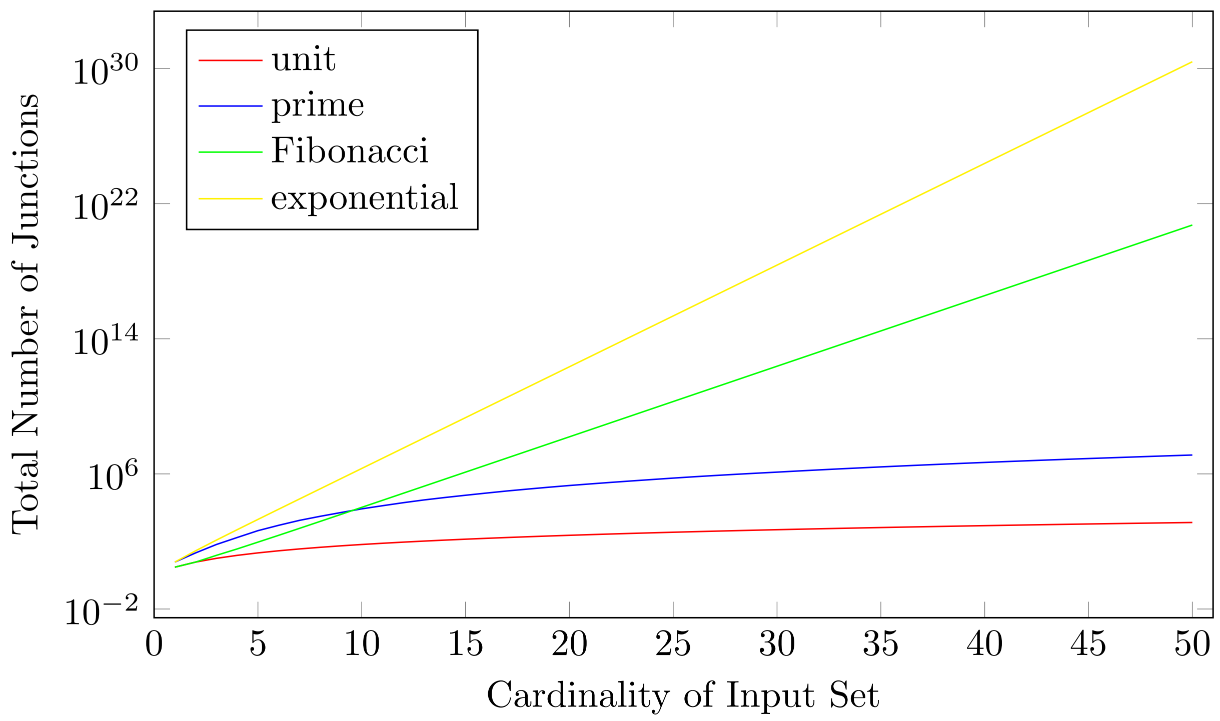


Figure 5.8: Logarithmic scale graph illustrating the total number of junctions that are used in the network in the NB-C method of [269] for the unit, prime, Fibonacci, and exponential input sets of different cardinalities, from $n = 1$ to $n = 50$. Note that the total number of junctions grows exponentially for exponential and Fibonacci input sets and grows linearly for unit input sets.

To further clarify the design procedure: in the SSP network used, a set of a problem with total sum size ς is encoded in a modular system, i.e., a lattice build from two isomorphic unit cells. One unit cell type contains a pass junction, where agent traffic lines cross without interaction, and the other unit cell type has additional split junctions, allowing a change of traffic direction [202, 269, 270]. The elements s_i of the set are represented by one split-junctions-containing unit cell followed by $(s_i - 1)$ pass-junction-only unit cells. All elements s_i of the set represented this way are sequentially ordered to obtain a row of ς unit cells. This row forms the basis of the network, which is copied $(\varsigma - 1)$ times and stacked in the direction perpendicular to the row. The concept of this design was presented in [270]. Note that adding an element in the set results in diagonal upward traffic, whereas skipping an element results in horizontal traffic. As a consequence, only a triangular part of the rectangular lattice, starting in the bottom-left corner, is needed. This translation of a set into a triangular network can be fully automated; the time needed scales with ς^2.

Note that in case the split lanes in the split junctions unit cell can be optionally blocked. One type of unit cell can be employed in the whole lattice, allowing all sets of total sum size ς to be calculated by the same network [202, 269]; this would considerably reduce the design and fabrication costs for this type of calculation network.

To summarize, the limitations on the pre-computing step of this DNA-C procedure are the length of synthesizable DNA strands and the number of available restriction enzymes, while the limitations on the pre-computing step of the NB-C method are the fabrication resolution and the size of the silicon wafer to be fabricated. Thus, the pre-computing step limits the size of an SSP instance that can be solved by the DNA-C procedure to at most $n = 45$ for unit sets, $n = 30$ for prime sets, $n = 15$ for Fibonacci sets, and $n = 9$ for exponential sets. Similarly, for the NB-C method used in this comparison, which uses bacteria as the agents and uses e-beam fabrication, the pre-computing step limits the size of an SSP instance that can be solved by the NB-C method to at most $n = 2000$ for unit sets, $n = 37$ for prime sets, $n = 16$ for Fibonacci sets, and $n = 10$ for exponential sets.

While serious challenges remain, it is important to note that none relate to a current technological limitation. Even with the present state-of-the-art technologies, there are several ways to improve performance to scale the computation at a cost. However, for a fair comparison of the E-C computing time with the computing time of alternative computing approaches, the time cost for pre-computing and readout procedures is omitted. For this scaling analysis, we only use information from the experimentally implemented procedures for solving SSP reported in [102] for DNA-C and [202, 269] for NB-C, and the performance specifications of typical electronic computers for E-C.

5.5.2 Volume comparison

In this section, we compare the maximum physical volume required by the DNA-C, NB-C, and E-C methods at a given time during the computation of a solution to an SSP instance: the maximum volume of DNA molecules for the DNA-C procedure, the maximum volume of biological agents (bacteria) for the NB-C method, and the volume of the computer for E-C.

For the DNA-C method, the maximum volume of the computation is proportional to the maximum number of DNA molecules, which is reached during the pre-computing step and remains constant afterwards. Indeed, the DNA-C procedure generates all 2^n potential solutions to an SSP instance of input size n by selectively removing number-representing "stations" from the computing region of the input set. For this volume calculation, we make the assumption that each SSP-solution-encoding DNA molecule is present in the same number of copies, and that $w_{\min}$ denotes the weight of the smallest amount of DNA detectable on a gel (with current technology, $w_{\min}$ is 1 ng [90,129]). With these assumptions, the maximum number of molecules that are required during a DNA-C computation can now be computed, based on the maximum number of the shortest SSP-solution-encoding DNA molecules that have a total weight of $w_{\min}$.

To calculate the number of base pairs of a partially double-stranded DNA molecule with sticky ends, we assume that each base in a sticky end counts as $\frac{1}{2}$ base pair. With this assumption, the length of the shortest computing region of a subset (the empty set) at the result readout step is $L^{\emptyset}_{\mathrm{DNA}}(n) - k$ (bp). To calculate how many of these shortest molecules fit into $w_{\min}$, we use the fact that the weight of linear double-stranded DNA molecules of length x (bp) is

$$W_{\mathrm{ds}}(x) = 617.96 \cdot (x - 2) + 2 \cdot (617.96 + 18.02) = (617.96 \cdot x + 36.04) \text{ (g/mol)},$$

where the average molecular weight of one internal base pair is 617.96 g/mol, and the average molecular weight of the two bases at an extremity is (617.96 + 18.02) g/mol (this includes the weight of the additional -OH and -H groups at the ends), see [218]. If the DNA double strands are circular, then their weight is $W_{\mathrm{c}}(x) = 617.96 \cdot x$ g/mol.

Given that the number of all potential solutions to a given SSP instance of size n is 2^n, it follows that the maximum number $N^{\max}_{\mathrm{DNA}}(n)$ of linear DNA molecules encoding the computing region of the input set is:

$$N^{\max}_{\mathrm{DNA}}(n) = \frac{2^n \cdot w_{\min}}{W_{\mathrm{ds}}(L^{\emptyset}_{\mathrm{DNA}}(n) - k)} \text{(mol)}.$$

It follows that the maximum volume required during the DNA-C procedure equals the total number $N^{\max}_{\mathrm{DNA}}(n)$ of plasmids, including the computing regions of the input set S, multiplied by the weight of one such "fully-stuffed" plasmid and divided by the density of DNA in the solution:

$$V^{\max}_{\mathrm{DNA}}(n) = \frac{N^{\max}_{\mathrm{DNA}}(n) \cdot W_{\mathrm{c}}(b + L^{S}_{\mathrm{DNA}}(n))}{d} \text{(mL)},$$

where d is the density of the DNA solution in g/mL, and b is the number of base pairs in the part of the base plasmid that is used. Taking $b = 2{,}174$ bp, $k = 6$ bp, $w_{\min} = 1 \cdot 10^{-9}$ g, and $d = 1 \cdot 10^{-3}$ g/mL (see [90]), the estimated volumes of DNA molecules in the DNA-C procedure for SSP with unit, prime, Fibonacci, and exponential input sets of different sizes are shown in Figure 5.9. If we take a 5 L as the maximum size of container that can be handled in a lab setting, the maximum size of an SSP instance that is solvable with the DNA-C procedure is $n = 28$ for unit sets, $n = 26$ for prime sets, $n = 21$ for Fibonacci sets, and $n = 17$ for exponential sets. Note that this is a theoretical upper limit, and that with a larger volume additional experimental, difficulties could arise at every step.

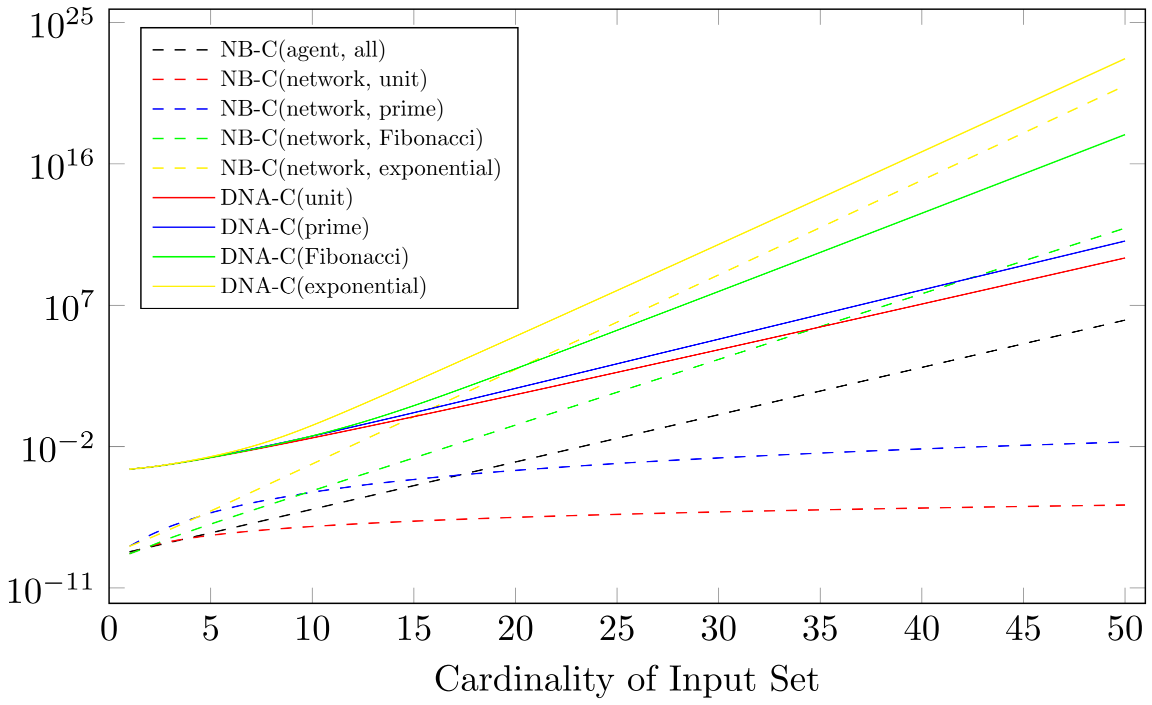


Figure 5.9: The estimated volumes of biological agents (solution of DNA molecules and bacteria, respectively) used by the DNA-C and NB-C methods and network channel volumes used by NB-C to solve an instance of SSP with unit, prime, Fibonacci, and exponential input sets of different cardinalities, from $n = 1$ to $n = 50$ (logarithmic scale). Note that the agent volume in NB-C is equal for all various SSP instances of the same cardinality.

For NB-C, in the bacterial-driven version, the volume of the computational agents required for solving a particular cardinality of SSP is dependent on the minimum number of bacteria needed for solving a specific cardinality. The minimum number of agents (bacteria) required to solve an SSP instance of cardinality n can be calculated using the

Euler coupon collector relation [253], with the Euler-Mascheroni constant $\gamma \approx 0.577\,21$, as:

$$M_{\text{NB-C}}(n) = 2 \cdot (2^n \cdot \ln(2^n) + \gamma \cdot 2^n + \frac{1}{2}).$$

A more conservative estimation will use a multiplier of the number provided by Euler's relationship for the minimum number of agents to explore the SSP network for a given cardinality. Furthermore, the higher the error rates in the junctions, the larger the number of agents needed to solve SSP. According to experimental reports of bacterial errors, i.e., 0.5 % at the pass junctions of SSP networks, a doubling of the value provided by the Euler relationship appears as conservatively justified [161, 253, 269]. The graph in Figure 5.9 presents the minimum volume of the required bacteria. A similar calculation can be made for cytoskeletal filaments, leading to smaller volume of the computational agents. At the other end of the spectrum, similar calculations can be performed for larger motile species, e.g., from the *Euglena* genus, leading to a considerably larger volume of agents. The volume of agents required during the NB-C approach is

$$V_{\text{NB-C}}(n) = \frac{M_{\text{NB-C}}(n)}{d} (\text{mL}),$$

where d is the density of the agents in the solution. Taking E. coli as the agent, which has a density of $2 \cdot 10^9$ cells in 1 mL of culture solution [193], the estimated minimum volumes of agents required by the NB-C approach for solving SSP instances with unit, prime, Fibonacci, and exponential input sets of different sizes are shown in Figure 5.9. This value is equal for solving SSP instances with unit, prime, Fibonacci, and exponential input sets of the same cardinality.

For NB-C however, apart from the total (bacterial) agent volume needed, there is the need to also consider the volume of the network channels. This volume is calculated using the track lengths in all junction unit cells of a network design, multiplied by the track width and height (for E. coli networks being 2 µm and 4 µm, respectively). For the compact complexity class II networks, with unit, prime, and Fibonacci input sets, the network channel volume is expanding moderately with cardinality (Figure 5.9), as many channels (and exits) will be visited multiple times. However, for the (unary coded) two-exponential network, (complexity class I, in which sets can only be reached by one route), the network volume will rapidly increase with cardinality.

For electronic computers, it can be arguably assumed that at the limit, the computing agents are the electrons. In addition to their very small size, given their sequential processing of electronic computers, there is a very small need for "agents," thus making the discussion regarding the volume of agents superfluous.

To summarize, the maximum physical volume required by the DNA-C and the (bacterial) agents and network volumes required by the NB-C methods to solve an SSP instance grows exponentially with the size of the input set. The volume required by the DNA-C algorithm also grows proportionally to the sum of the numbers in the input set. Compared with the NB-C method, the DNA-C method requires orders of magnitudes larger volumes even for unit input sets. For example, the NB-C method to solve an SSP instance with an input set of cardinality of $n = 30$ requires approximately 1.07 mL of bacteria, while for the same cardinality the DNA-C method requires 14.7 L for unit sets, 68.8 L for the prime set, $7.54 \cdot 10^4$ L for the Fibonacci set, and $7.44 \cdot 10^7$ L for the exponential set.

5.5.3 Run time comparison

Perhaps the most important performance criterion for comparing various computing methods for solving NP-complete problems is the computing time required to solve a benchmark problem. Indeed, one of the goals of the research into alternative computing methods was to exploit their massive parallelism to shorten the computing time required to solve NP-complete problems.

The time complexity of a bio-algorithm comprises three parts: the time required for the pre-computing, the time required by the actual computation step (potential solutions' generation), and the time needed to read out the results. The time comparison in this section refers only to the time required during the actual computation step, called thereafter *run time*, and excludes the time spent in the pre-computing step (creating the plasmids in the case of DNA-C, and fabricating the network and culturing the bacteria in the case of NB-C). Similarly, the time spent for read-out analysis was not included either, because:

- The process is simple if only the existence of an SSP solution is considered.
- The process of finding the actual solutions could in principle be parallelized and thus achievable with present technologies, depending on cost considerations only.

For the solution generation step of the DNA-C approach, mentioned in Section 5.2, the DNA-C procedure for solving an SSP instance of size n comprises n split-and-merge steps. Each such step starts with the digestion of half of the plasmid population by restriction enzymes to cut out a specific number-encoding "station." This is followed by purification by agarose gel electrophoresis to separate and remove the number-encoding "station" short DNA strands. This is followed, in turn, by the re-ligation of the longer linear strands, which include the computing region of some subset of the input set, back into circular plasmids.

Thus, if t_e is the time required for one restriction digestion, t_p the time required for one plasmid purification, and t_l the time required for one ligation, then the estimated run time for a DNA-C procedure solving a size n instance of SSP is:

$$T^{\max}_{\mathrm{DNA}}(n) = (t_e + t_p + t_l) \cdot n \text{ (s)}.$$

Taking $t_e = 0.5\,\mathrm{h}$ (1,800 s), $t_p = 1.5\,\mathrm{h}$ (5,400 s), and $t_l = 0.25\,\mathrm{h}$ (900 s), see [90], this results in a run time complexity of $T^{\max}_{\mathrm{DNA}}(n) = 8{,}100 \cdot n$ (s), as illustrated in Figure 5.10, for various values of n. Note that the time complexity is always linear, but that larger restriction digestion time t_e for some enzymes could increase the constant in its formula.

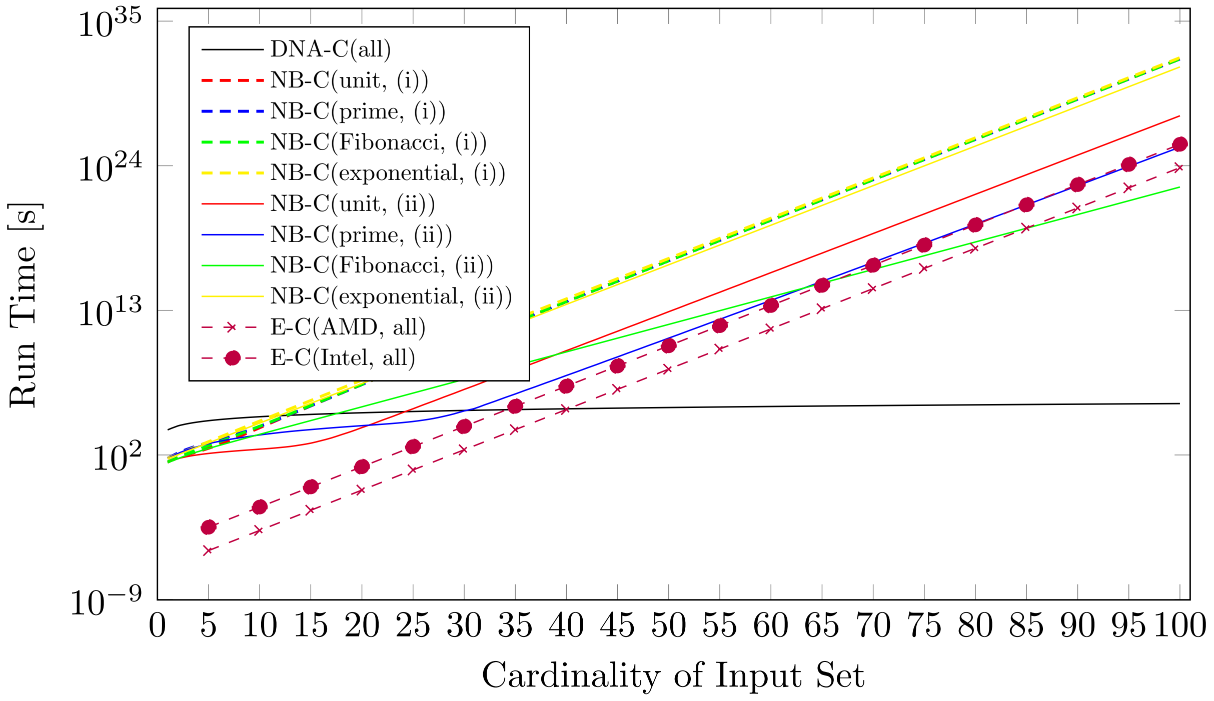


Figure 5.10: The estimated run time required by the DNA-C, NB-C, and E-C methods to solve an instance of SSP with input sets of different cardinalities, from $n = 1$ to $n = 100$ (logarithmic scale). The real run time of NB-C will be in between the curves depicting scenario (i) (combinatorial run mode) and scenario (ii) (multiplication run mode). Note that the run time of the DNA-C method grows linearly in the cardinality of the input set, while the run time of the NB-C and the E-C methods grows exponentially.

The total NB-C run time for solving SSP, as a function of cardinality of the input set, depends on many parameters but can be modelled as a mixture of two extreme run modes between which the "real world" operation takes place:

(i) The combinatorial run mode [269]: agents enter the network at the starting point with a fixed booting frequency (determined by the effective agent length in a queue and the average agent speed) and proceed on their routes to the exits. The total run time is the time needed to boot all 2^n agents (times the coupon collectors correction for stochastic multiple identical variable combinations [253]), plus the time needed for a single agent to run from start to exit. For low cardinality problems, the single agent run time is important; for high cardinality problems, this value is eclipsed by the booting time needed for all agents to enter the network.

(ii) The optimum multiplication run mode [269]: a single agent enters at the starting point, and at every split junction there is agent multiplication (cell division). In case there is only one route to every legal SSP exit (complexity class I network), the total run time equals the (longest) one-agent run time from start to exit. This would be the case for the strongly expanding exponential set. More compact networks show multiple routes to the same exits (complexity class II network). As a consequence, there will be a rise in traffic density farther down the network; this may lead to traffic jams. In the "optimistic" scenario here, the agents simply wait until there is space available to proceed. The total run time is the run time of one agent plus the time needed to "de-boot" the agents that need to leave the (most-visited) middle exits of the network, with a fixed "de-booting" frequency (likewise determined by the effective agent length and average agent speed). Again, at low cardinality, the single agent run time is important, but for high cardinality problems, this value is eclipsed by the "de-booting" time needed for the agents to leave the exits in the middle part of the network.

In the "real world," cell division will probably not occur at every split junction, and may slow down at higher traffic density. Regardless, the run time versus cardinality plot for a given network will be in between the curves of scenarios (i) and (ii) described above, also shown in Figure 5.10.

To summarize, the run time complexity of the DNA-C procedure, $T_{\text{DNA}}^{\max}(n)$, is linear in the problem size n, in contrast with the exponential run time complexity for both the NB-C and the E-C methods for solving SSP. In addition, the run time required by the NB-C method also grows linearly in the sum of the numbers in the input set. Overall, the linear run time complexity of the DNA-C procedure for SSP is its most competitive feature compared with both NB-C and E-C.

5.5.4 Energy comparison

Another key performance criterion, especially for high performance computing, is energy consumption. The following analysis takes into consideration the pre-computing procedures or readout energy consumption for none of the three computing methods considered.

In the computation step of the DNA-C procedure, two operations are employed: cutting by restriction enzymes and pasting by ligase enzymes, as described in Section 2.2. The restriction enzyme digestion is inexpensive in terms of energy consumption, as no external energy is needed [272]. However, ligation is energy-intensive, as ligases consume one ATP (6.3 kcal/mol = 26,359.2 J/mol of energy [63]) per backbone nick-sealing event. Ligating two DNA double strands together entails two such nick-sealing events, one for each single strand. Recall that the DNA-C procedure comprises n split-and-merge substeps, and each substep entails re-ligating half of the plasmids (namely those from which one number-representing station was cut out). Hence, the energy consumption of the computation step of the DNA-C procedure for an SSP instance of size n is:

$$E_{\text{DNA}}(n) = n \cdot (\frac{1}{2} N_{\text{DNA}}^{\max}(n)) \cdot (2 \cdot 26{,}359.2) \text{ (J)}.$$

Note that the energy cost for the computation stage of the DNA-C procedure grows exponentially in the input size n, due to exponential growth of the number of strands $N_{\text{DNA}}^{\max}(n)$, as illustrated in Figure 5.11. Note that, while the total energy cost grows exponentially in the input size n, the DNA-C procedure is very energy efficient when we consider the average electrical power in watts (energy divided by the run time, analyzed in Section 5.5.3). Indeed, even though the wattage also grows exponentially with n, for an input set size as large as $n = 50$ the wattage needed is only about 20 W, similar to the energy consumption of an LED light bulb. Note that in order to keep the computation in an optimal reacting environment (temperature, concentration, etc.), additional energy sources may be required.

For the NB-C energy computation, at a given cardinality we know how many agents are needed to finish the calculation (including the coupon collector's correction [253]). We know the locomotion energy needed for passing one node [269]. Hence, we can easily calculate the total locomotion energy needed to perform the calculation for the combinatorial run mode (no cell division, scenario (a) in Section 5.5.3), as shown in Figure 5.11. In the multiplication run mode (with cell division, scenario (b) in Section 5.5.3), on the one hand, we are over-calculating the locomotion energy because many agents originate somewhere in the network and need to run only part of the total track. On the other hand, we fully ignore the energy needed for agent multiplication (cell division), which is likely to cost much more than just running several nodes. At present, we do not yet have a proper energy estimate for the multiplication run mode.

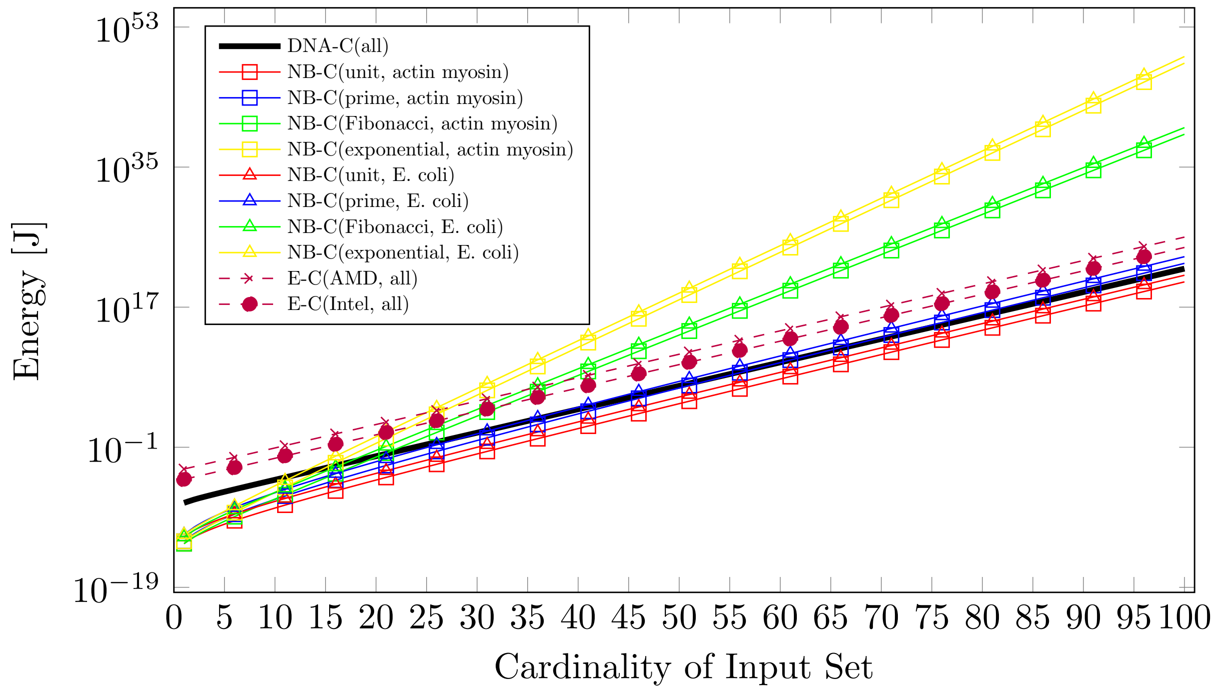


Figure 5.11: The energy consumed in the computation step of DNA-C, NB-C, and E-C methods to solve an instance of SSP with unit, prime, Fibonacci, and exponential input sets of different cardinalities, from $n = 1$ to $n = 100$ (logarithmic scale). Note that the energy cost of the DNA-C and E-C methods is independent of the type of input set and depends only on its cardinality, while the energy cost of the NB-C method depends on the cardinality, the sum of the numbers of the input set, and the type of agent (here E. coli or actin myosin).

Finally, the energy required by an E-C method to solve SSP can be easily calculated as the product between the number of operations required by the E-C procedure for solving a particular instance of SSP and the energy cost per operation from the chip specifications.

To summarize, the energy cost of the computation step for DNA-C, NB-C, and E-C grows exponentially with the size of the input set. Moreover, the energy cost in the computation step of the NB-C method also increases linearly in the sum of the numbers in the input set. Overall, NB-C appears to be more energy efficient than DNA-C in some cases (unit and prime input sets). In other cases (Fibonacci and exponential input sets), NB-C consumes more energy than DNA-C, increasingly so with the increase of the input set cardinality and regardless of the agents used. However, despite the energy cost being high for NB-C in some cases, we note that bacteria can self-produce the ATP from cheaply available nutrient sources like beef extract, yeast extract, and tryptone, while for DNA-C the ATP has to be externally supplied as a purified additive. Hence, in a way, bacterial-based NB-C is more self-sustainable [116], despite its sometimes high energy cost compared to DNA-C. Finally, E-C appears to be an order of magnitude less energy efficient than DNA-C.

5.6 Conclusion

Our analysis showed that the sequentiality of E-C translated in a very small volume compared to that required by DNA-C and NB-C, at the cost of the E-C computing time being outperformed first by DNA-C (linear run time), and then by NB-C. Finally, NB-C appears to be more energy efficient than DNA-C for some types of input sets, while being less energy efficient than DNA-C for other types of input sets, while E-C is always an order of magnitude less energy efficient than DNA-C.

This scaling study suggests that currently none of these computing approaches won, even theoretically, for all three key performance criteria, and that they all required breakthroughs to overcome their limitations.

Chapter 6

Conclusions

We conclude this thesis by reviewing its contributions and suggesting future research topics.

This thesis investigated three inter-related topics of DNA computing: formal language models of two DNA computing bio-operations (Chapter 3), concepts in combinatorics of words that capture the properties of information encoded by DNA strands (Chapter 4), and a comparison between the time, space, and energy complexities of experimental implementations of DNA computing, network biocomputing, and electronic computing algorithms for solving the subset sum problem (Chapter 5).

In Chapter 3, we defined the operation called word blending, where the input words are $\alpha w \gamma_1$ and $\gamma_2 w \beta$ sharing a non-empty overlap w, and the output word is $\alpha w \beta$. This operation was motivated by an unexpected outcome of a wet lab experiment that attempted to recombine two DNA molecules containing the same gene. We studied closure properties of the Chomsky families of languages under this operation and its iterated version, the decidability of the existence of a solution to equations involving this operation, and the state complexity of this operation. We then defined and studied conjugate word blending, which is more closely related to XPCR experimental details, where γ_1 and γ_2 are required to be identical and non-empty. In the future, we can consider variations of word blending where a length restriction on parts of the input words is imposed.

In Chapter 4, a unified terminology for various definitions of Fibonacci words in the literature was proposed. It was then generalized to involutive Fibonacci words motivated by Watson-Crick complementarity. We have shown the borderedness and ϕ-borderedness of involutive Fibonacci words under various conditions and the primitivity of atom involutive Fibonacci words defined with different (anti)morphic involutions. Future research topics include the ϕ-primitivity of involutive Fibonacci words and their combinatorial properties, such as the number of distinct factors and squares.

In Chapter 5, the experimental implementations of DNA computing (DNA-C), network biocomputing (NB-C), and electronic computing (E-C) algorithms for solving the subset sum problem were reviewed and compared. Our scaling analysis showed that none of DNA-C, NB-C, and E-C approaches won for all of the time, space, and energy complexities, and they all had their limitations and trade-offs. In the future, a similar analysis can be applied to other NP-complete problems, and we can work on a hybrid computation model that combines the advantages of different approaches.

References

[1] M. D. Abràmoff, P. J. Magalhães, and S. J. Ram. Image processing with ImageJ. *Biophotonics International*, 11(7):36–42, 2004.

[2] R. Adar, Y. Benenson, G. Linshiz, A. Rosner, N. Tishby, and E. Shapiro. Stochastic computing with biomolecular automata. *Proceedings of the National Academy of Sciences*, 101(27):9960–9965, 2004.

[3] L. M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021–1024, 1994.

[4] L. M. Adleman. Computing with DNA. *Scientific American*, 279(2):54–61, 1998.

[5] M. Agrawal, N. Kayal, and N. Saxena. PRIMES is in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[6] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[7] Y. Aoi, T. Yoshinobu, K. Tanizawa, K. Kinoshita, and H. Iwasaki. Solution of the knapsack problem by deoxyribonucleic acid computing. *Japanese Journal of Applied Physics*, 37(10R):5839–5841, 1998.

[8] Y. Aoi, T. Yoshinobu, K. Tanizawa, K. Kinoshita, and H. Iwasaki. Ligation errors in DNA computing. *Biosystems*, 52(1):181–187, 1999.

[9] M. Arita and S. Kobayashi. DNA sequence design using templates. *New Generation Computing*, 20(3):263–277, 2002.

[10] S. A. Babu and P. K. Pandya. Chop expressions and discrete duration calculus. In D. D'Souza and P. Shankar, editors, *Modern Applications of Automata Theory*, volume 2 of *IISc Research Monographs Series*, pages 229–256. World Scientific, 2012.

[11] W. M. Barnes. PCR amplification of up to 35-kb DNA with high fidelity and high yield from lambda bacteriophage templates. *Proceedings of the National Academy of Sciences*, 91(6):2216–2220, 1994.

[12] A. D. Bates and A. Maxwell. *DNA Topology.* Oxford University Press, 2005.

[13] J. Baumgardner et al. Solving a Hamiltonian path problem with a bacterial computer. *Journal of Biological Engineering*, 3(11):1–11, 2009.

[14] F. Bellamoli. Production of Gene Libraries by Multiple XPCR. Master's thesis, Department of Biotechnology, University of Verona, 2013.

[15] F. Bellamoli, G. Franco, L. Kari, S. Lampis, T. Ng, and Z. Wang. Conjugate word blending: Formal model and experimental implementation by XPCR. *Natural Computing*, 20(4):647–658, 2021.

[16] Y. Benenson, R. Adar, T. Paz-Elizur, Z. Livneh, and E. Shapiro. DNA molecule provides a computing machine with both data and fuel. *Proceedings of the National Academy of Sciences*, 100(5):2191–2196, 2003.

[17] Y. Benenson, B. Gil, U. Ben-Dor, R. Adar, and E. Shapiro. An autonomous molecular computer for logical control of gene expression. *Nature*, 429(6990):423–429, 2004.

[18] Y. Benenson, T. Paz-Elizur, R. Adar, E. Keinan, Z. Livneh, and E. Shapiro. Programmable and autonomous computing machine made of biomolecules. *Nature*, 414(6862):430–444, 2001.

[19] J. Berstel. Mots de Fibonacci. In *Séminaire d'Informatique Théorique*, pages 57–78. Rapport L.I.T.P., 1980.

[20] J. Berstel. *Axel Thue's Papers on Repetitions in Words: A Translation.* Départements de mathématiques et d'informatique, Université du Québec à Montréal, 1995.

[21] J. Berstel. On the index of Sturmian words. In J. Karhumäki, H. Maurer, et al., editors, *Jewels are Forever: Contributions on Theoretical Computer Science in Honor of Arto Salomaa*, pages 287–294. Springer, 1999.

[22] S. Bhat, D. Bialy, J. E. Sealy, J.-R. Sadeyen, P. Chang, and M. Iqbal. A ligation and restriction enzyme independent cloning technique: An alternative to conventional methods for cloning hard-to-clone gene segments in the influenza reverse genetics system. *Virology Journal*, 17(82):1–9, 2020.

[23] R. A. Bierley, G. J. Rixen, A. I. Tröster, and W. W. Beatty. Preserved spatial memory in old rats survives 10 months without training. *Behavioral and Neural Biology*, 45(2):223–229, 1986.

[24] E. Bombieri and J. E. Taylor. Which distributions of matter diffract? An initial investigation. *Journal de Physique Colloques*, 47(C3):19–28, 1986.

[25] R. S. Braich, N. Chelyapov, C. Johnson, P. W. K. Rothemund, and L. M. Adleman. Solution of a 20-variable 3-SAT problem on a DNA computer. *Science*, 296(5567):499–502, 2002.

[26] R. S. Braich, C. Johnson, P. W. K. Rothemund, D. Hwang, N. Chelyapov, and L. M. Adleman. Solution of a satisfiability problem on a gel-based DNA computer. In A. Condon and G. Rozenberg, editors, *Proceedings of 6th International Workshop on DNA-Based Computers (DNA 2000)*, volume 2054 of *LNCS*, pages 27–42. Springer, 2001.

[27] U. Brandes, D. Delling, M. Gaertler, R. Gorke, M. Hoefer, Z. Nikoloski, and D. Wagner. On modularity clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20(2):172–188, 2008.

[28] K. Břinda, E. Pelantová, and O. Turek. Balances of m-bonacci words. *Fundamenta Informaticae*, 132(1):33–61, 2014.

[29] L. Brody. Nucleotide. *National Human Genome Research Institute*, 2022. Retrieved August 2, 2022, from https://www.genome.gov/genetics-glossary/Nucleotide.

[30] J. A. Brzozowski. In search of most complex regular languages. *International Journal of Foundations of Computer Science*, 24(6):691–708, 2013.

[31] J. A. Brzozowski, L. Kari, B. Li, and M. Szykuła. State complexity of overlap assembly. *International Journal of Foundations of Computer Science*, 31(8):1113–1132, 2020.

[32] A. Buatois and M. Lihoreau. Evidence of trapline foraging in honeybees. *Journal of Experimental Biology*, 219(16):2426–2429, 2016.

[33] C. Câmpeanu, K. Culik, K. Salomaa, and S. Yu. State complexity of basic operations on finite languages. In O. Boldt and H. Jürgensen, editors, *Proceedings of 4th International Workshop on Implementing Automata (WIA 99)*, volume 2214 of *LNCS*, pages 60–70. Springer, 2001.

[34] C. Câmpeanu and W. H. Ho. The maximum state complexity for finite languages. *Journal of Automata, Languages and Combinatorics*, 9(2–3):189–202, 2004.

[35] A. Carausu and G. Păun. String intersection and short concatenation. *Revue Roumaine de Mathématiques Pures et Appliquées*, 26(5):713–726, 1981.

[36] R. Ceterchi. An algebraic characterization of semi-simple splicing. *Fundamenta Informaticae*, 72(1–2):19–25, 2006.

[37] S. Chandak et al. Overcoming high nanopore basecaller error rates for DNA storage via basecaller-decoder integration and convolutional codes. In *Proceedings of 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP 2020)*, pages 8822–8826. IEEE, 2020.

[38] W.-L. Chang, M. S.-H. Ho, and M. Guo. Molecular solutions for the subset-sum problem on DNA-based supercomputing. *Biosystems*, 73(2):117–130, 2004.

[39] J. Chao et al. Solving mazes with single-molecule DNA navigators. *Nature Materials*, 18(3):273–279, 2019.

[40] G. Chatterjee, N. Dalchau, R. A. Muscat, A. Phillips, and G. Seelig. A spatially localized architecture for fast and modular DNA computing. *Nature Nanotechnology*, 12(9):920–927, 2017.

[41] D. Cheptea, C. Martın-Vide, and V. Mitrana. A new operation on words suggested by DNA biochemistry: Hairpin completion. In J.-G. Dumas, editor, *Proceedings of Transgressive Computing 2006 (TC 2006)*, pages 105–114, 2006.

[42] K. M. Cherry and L. Qian. Scaling up molecular pattern recognition with DNA-based winner-take-all neural networks. *Nature*, 559(7714):370–376, 2018.

[43] D.-J. Cho, Y.-S. Han, S.-D. Kang, H. Kim, S.-K. Ko, and K. Salomaa. Pseudo-inversion: Closure properties and decidability. *Natural Computing*, 15(1):31–39, 2016.

[44] D.-J. Cho, Y.-S. Han, H. Kim, A. Palioudakis, and K. Salomaa. Duplications and pseudo-duplications. In C. S. Calude and M. J. Dinneen, editors, *Proceedings of 14th International Conference on Unconventional Computation and Natural Computation (UCNC 2015)*, volume 9252 of *LNCS*, pages 157–168. Springer, 2015.

[45] D.-J. Cho, Y.-S. Han, H. Kim, and K. Salomaa. Site-directed deletion. In M. Hoshi and S. Seki, editors, *Proceedings of 22nd International Conference on Developments*

in Language Theory (DLT 2018), volume 11088 of *LNCS*, pages 219–230. Springer, 2018.

[46] D.-J. Cho, Y.-S. Han, T. Ng, and K. Salomaa. Outfix-guided insertion. *Theoretical Computer Science*, 701:70–84, 2017.

[47] D.-J. Cho, Y.-S. Han, K. Salomaa, and T. J. Smith. Site-directed insertion: Language equations and decision problems. *Theoretical Computer Science*, 798:40–51, 2019.

[48] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, 1959.

[49] W.-F. Chuan. Fibonacci words. *Fibonacci Quarterly*, 30(1):68–76, 1992.

[50] W.-F. Chuan. Symmetric Fibonacci words. *Fibonacci Quarterly*, 31(3):251–255, 1993.

[51] W.-F. Chuan. Generating Fibonacci words. *Fibonacci Quarterly*, 33(2):104–112, 1995.

[52] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[53] C. T. Clelland, V. Risca, and C. Bancroft. Hiding messages in DNA microdots. *Nature*, 399(6736):533–534, 1999.

[54] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT press, 4th edition, 2022.

[55] E. Csuhaj-Varjú, I. Petre, and G. Vaszil. Self-assembly of strings and languages. *Theoretical Computer Science*, 374(1–3):74–81, 2007.

[56] M. Daley, O. H. Ibarra, and L. Kari. Closure and decidability properties of some language classes with respect to ciliate bio-operations. *Theoretical Computer Science*, 306(1–3):19–38, 2003.

[57] J. Dassow. A ciliate bio-operation and language families. In C. S. Calude, E. Calude, et al., editors, *Proceedings of 8th International Conference on Developments in Language Theory (DLT 2004)*, volume 3340 of *LNCS*, pages 151–162. Springer, 2004.

[58] J. Dassow, V. Mitrana, and G. Păun. On the regularity of duplication closure. *Bulletin of the European Association for Theoretical Computer Science*, 69:133–136, 1999.

[59] J. Dassow, V. Mitrana, and A. Salomaa. Operations and language generating devices suggested by the genome evolution. *Theoretical Computer Science*, 270(1):701–738, 2002.

[60] A. de Luca. A combinatorial property of the Fibonacci words. *Information Processing Letters*, 12(4):193–195, 1981.

[61] A. de Luca. A division property of the Fibonacci word. *Information Processing Letters*, 54(6):307–312, 1995.

[62] S. Di Gregorio, C. Zocca, S. Sidler, A. Toffanin, D. Lizzari, and G. Vallini. Identification of two new sets of genes for dibenzothiophene transformation in Burkholderia sp. DBT1. *Biodegradation*, 15:111–123, 2004.

[63] K. S. Dickson, C. M. Burns, and J. P. Richardson. Determination of the free-energy change for repair of a DNA phosphodiester bond. *Journal of Biological Chemistry*, 275(21):15828–15831, 2000.

[64] C. W. Dieffenbach and G. S. Dveksler, editors. *PCR Primer: A Laboratory Manual.* Cold Spring Harbor Laboratory Press, 2nd edition, 2003.

[65] M. Domaratzki. Semantic shuffle on and deletion along trajectories. In C. S. Calude, E. Calude, et al., editors, *Proceedings of 8th International Conference on Developments in Language Theory (DLT 2004)*, volume 3340 of *LNCS*, pages 163–174. Springer, 2005.

[66] M. Domaratzki. Minimality in template-guided recombination. *Information and Computation*, 207(11):1209–1220, 2009.

[67] M. Dorigo and L. M. Gambardella. Ant colony system: A cooperative learning approach to the traveling salesman problem. *IEEE Transactions on Evolutionary Computation*, 1(1):53–66, 1997.

[68] X. Droubay. Palindromes in the Fibonacci word. *Information Processing Letters*, 55(4):217–221, 1995.

[69] C. F. Du, H. Mousavi, E. Rowland, L. Schaeffer, and J. O. Shallit. Decision algorithms for Fibonacci-automatic words, II: Related sequences and avoidability. *Theoretical Computer Science*, 657:146–162, 2017.

[70] L. Eeckhout. Is Moore's law slowing down? What's next? *IEEE Micro*, 37(4):4–5, 2017.

[71] K. Ellul. Description Complexity Measures of Regular Languages. Master's thesis, David R. Cheriton School of Computer Science, University of Waterloo, 2003.

[72] S. K. Enaganti, O. H. Ibarra, L. Kari, and S. Kopecki. Further remarks on DNA overlap assembly. *Information and Computation*, 253:143–154, 2017.

[73] S. K. Enaganti, O. H. Ibarra, L. Kari, and S. Kopecki. On the overlap assembly of strings and languages. *Natural Computing*, 16:175–185, 2017.

[74] S. K. Enaganti, L. Kari, and S. Kopecki. A formal language model of DNA polymerase enzymatic activity. *Fundamenta Informaticae*, 138(1-2):179–192, 2015.

[75] S. K. Enaganti, L. Kari, T. Ng, and Z. Wang. Word blending in formal languages: The Brangelina effect. In S. Stepney and S. Verlan, editors, *Proceedings of 17th International Conference on Unconventional Computation and Natural Computation (UCNC 2018)*, volume 10867 of *LNCS*, pages 72–85. Springer, 2018.

[76] S. K. Enaganti, L. Kari, T. Ng, and Z. Wang. Word blending in formal languages. *Fundamenta Informaticae*, 171(1-4):151–173, 2020.

[77] Z. Ezziane. DNA computing: Applications and challenges. *Nanotechnology*, 17(2):R27–39, 2005.

[78] C.-M. Fan and H. J. Shyr. Some properties of Fibonacci languages. *Tamkang Journal of Mathematics*, 27(2):165–182, 1996.

[79] G. Fici. Factorizations of the Fibonacci infinite word. *Journal of Integer Sequences*, 18(9):1–14, 2015.

[80] A. S. Fraenkel. Complexity of protein folding. *Bulletin of Mathematical Biology*, 55(6):1199–1210, 1993.

[81] A. S. Fraenkel and J. Simpson. The exact number of squares in Fibonacci words. *Theoretical Computer Science*, 218(1):95–106, 1999.

[82] G. Franco. A polymerase based algorithm for SAT. In M. Coppo, E. Lodi, et al., editors, *Proceedings of 9th Italian Conference on Theoretical Computer Science (ICTCS 2005)*, volume 3701 of *LNCS*, pages 237–250. Springer, 2005.

[83] G. Franco, F. Bellamoli, and S. Lampis. Experimental analysis of XPCR-based protocols. *arXiv preprint arXiv:1712.05182*, 2017.

[84] G. Franco, C. Giagulli, C. Laudanna, and V. Manca. DNA extraction by XPCR. In C. Ferretti, G. Mauri, et al., editors, *Proceedings of 10th International Workshop on DNA Computing (DNA 10)*, volume 3384 of *LNCS*, pages 104–112, 2005.

[85] G. Franco and V. Manca. Algorithmic applications of XPCR. *Natural Computing*, 10(2):805–819, 2011.

[86] G. Franco, V. Manca, C. Giagulli, and C. Laudanna. DNA recombination by XPCR. In A. Carbone and N. A. Pierce, editors, *Proceedings of 11th International Workshop on DNA Computing (DNA 11)*, volume 3892 of *LNCS*, pages 55–66. Springer, 2006.

[87] M. Garzon, P. Neathery, R. Deaton, R. C. Murphy, D. R. Franceschetti, and S. Stevens Jr. A new metric for DNA computing. In J. R. Koza, K. Deb, et al., editors, *Proceedings of 2nd Annual Conference on Genetic Programming*, volume 32, pages 472–478. Morgan Kaufman, 1997.

[88] S. Ginsburg. *Algebraic and Automata-Theoretic Properties of Formal Languages.* Elsevier, 1975.

[89] J. S. Golan. *The Theory of Semirings with Applications in Mathematics and Theoretical Computer Science.* Addison-Wesley, 1992.

[90] M. R. Green and J. Sambrook. *Molecular Cloning: A Laboratory Manual.* Cold Spring Harbor Laboratory Press, 4th edition, 2012.

[91] S. T. Gries. Shouldn't it be breakfunch? A quantitative analysis of blend structure in English. *Linguistics*, 42(3):639–667, 2004.

[92] M. Hagiya, M. Arita, D. Kiga, K. Sakamoto, and S. Yokoyama. Towards parallel evaluation and learning of boolean μ-formulas with molecules. In H. Rubin and D. H. Wood, editors, *Proceedings of DIMACS Workshop DNA Based Computers III*, volume 48 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 57–72. American Mathematical Society, 1999.

[93] Y.-S. Han and K. Salomaa. State complexity of union and intersection of finite languages. *International Journal of Foundations of Computer Science*, 19(3):581–595, 2008.

[94] K. L. Hanson, D. V. Nicolau Jr., L. Filipponi, L. Wang, A. P. Lee, and D. V. Nicolau. Fungi use efficient algorithms for the exploration of microfluidic networks. *Small*, 2(10):1212–1220, 2006.

[95] T. Head. Formal language theory and DNA: An analysis of the generative capacity of specific recombinant behaviors. *Bulletin of Mathematical Biology*, 49(6):737–759, 1987.

[96] T. Head, G. Păun, and D. Pixton. Language theory and molecular genetics: Generative mechanisms suggested by DNA recombination. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 2, pages 295–360. Springer, 1997.

[97] T. Head, G. Rozenberg, R. S. Bladergroen, C. K. D. Breek, P. H. M. Lommerse, and H. P. Spaink. Computing with DNA by operating on plasmids. *Biosystems*, 57(2):87–93, 2000.

[98] M. Held, M. Binz, C. Edwards, and D. V. Nicolau. Dynamic behaviour of fungi in microfluidics: A comparative study. In D. L. Farkas, D. V. Nicolau, et al., editors, *Proceedings of the Imaging, Manipulation, and Analysis of Biomolecules, Cells, and Tissues VII (SPIE 7182)*, page 718213. SPIE, 2009.

[99] M. Held, C. Edwards, and D. V. Nicolau. Probing the growth dynamics of Neurospora crassa with microfluidic structures. *Fungal Biology*, 115(6):493–505, 2011.

[100] M. Held, O. Kašpar, C. Edwards, and D. V. Nicolau. Intracellular mechanisms of fungal space searching in microenvironments. *Proceedings of the National Academy of Sciences*, 116(27):13543–13552, 2019.

[101] R. B. Helling, H. M. Goodman, and H. W. Boyer. Analysis of endonuclease R·EcoRI fragments of DNA from lambdoid bacteriophages and other viruses by agarose-gel electrophoresis. *Journal of Virology*, 14(5):1235–1244, 1974.

[102] C. V. Henkel, T. Bäck, J. N. Kok, G. Rozenberg, and H. P. Spaink. DNA computing of solutions to knapsack problems. *Biosystems*, 88(1–2):156–162, 2007.

[103] C. V. Henkel, R. S. Bladergroen, C. I. A. Balog, A. M. Deelder, T. Head, G. Rozenberg, and H. P. Spaink. Protein output for DNA computing. *Natural Computing*, 4(1):1–10, 2005.

[104] P. M. Higgins. The naming of Popes and a Fibonacci sequence in two noncommuting indeterminates. *Fibonacci Quarterly*, 25(1):57–61, 1987.

[105] O. Ho-Shing, K. H. Lau, W. Vernon, T. T. Eckdahl, and A. M. Campbell. Assembly of standardized DNA parts using BioBrick ends in E. coli. In J. Peccoud, editor, *Gene Synthesis: Methods and Protocols*, volume 852 of *Methods in Molecular Biology*, pages 61–76. Humana Press, 2012.

[106] M. Holzer and S. Jakobi. Chop operations and expressions: Descriptional complexity considerations. In G. Mauri and A. Leporati, editors, *Proceedings of 15th International Conference on Developments in Language Theory (DLT 2011)*, volume 6795 of *LNCS*, pages 264–275. Springer, 2011.

[107] M. Holzer and S. Jakobi. State complexity of chop operations on unary and finite languages. In M. Kutrib, N. Moreira, et al., editors, *Proceedings of 14th International Workshop on Descriptional Complexity of Formal Systems (DCFS 2012)*, volume 7386 of *LNCS*, pages 169–182. Springer, 2012.

[108] M. Holzer, S. Jakobi, and M. Kutrib. The chop of languages. *Theoretical Computer Science*, 682:122–137, 2017.

[109] M. Holzer and M. Kutrib. State complexity of basic operations on nondeterministic finite automata. In J.-M. Champarnaud and D. Maurel, editors, *Proceedings of 7th International Conference on Implementation and Application of Automata (CIAA 2002)*, volume 2608 of *LNCS*, pages 148–157. Springer, 2003.

[110] M. Holzer and M. Kutrib. Unary language operations and their nondeterministic state complexity. In M. Ito and M. Toyama, editors, *Proceedings of 6th International Conference on Developments in Language Theory (DLT 2002)*, volume 2450 of *LNCS*, pages 162–172. Springer, 2003.

[111] C. M. Hommelsheim, L. Frantzeskakis, M. Huang, and B. Ülker. PCR amplification of repetitive DNA: A limitation to genome editing technologies and many other applications. *Scientific Reports*, 4(5052):1–13, 2015.

[112] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 3rd edition, 2006.

[113] J. J. Hopfield and D. W. Tank. "Neural" computation of decisions in optimization problems. *Biological Cybernetics*, 52(3):141–152, 1985.

[114] J. R. Howse, R. A. L. Jones, A. J. Ryan, T. Gough, R. Vafabakhsh, and R. Golestanian. Self-motile colloidal particles: From directed propulsion to random walk. *Physical Review Letters*, 99(4):048102, 2007.

[115] H.-K. Hsiao and S.-S. Yu. Mapped shuffled Fibonacci languages. *Fibonacci Quarterly*, 41(5):421–430, 2003.

[116] P. Hunter. Can bacteria save the planet? New developments in systems biology and biotechnology to harness bacteria for renewable energy and environmental regeneration. *EMBO Reports*, 11(4):266–269, 2010.

[117] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on computing*, 17(5):935–938, 1988.

[118] M. Ito. *Algebraic Theory of Automata and Languages*. World Scientific, 2004.

[119] M. Ito, L. Kari, Z. Kincaid, and S. Seki. Duplication in DNA sequences. In A. Condon, D. Harel, et al., editors, *Algorithmic Bioprocesses*, Natural Computing Series, pages 43–61. Springer, 2009.

[120] M. Ito, P. Leupold, F. Manea, and V. Mitrana. Bounded hairpin completion. *Information and Computation*, 209(3):471–485, 2011.

[121] M. Ito, P. Leupold, and K. Shikishima-Tsuji. Closure of language classes under bounded duplication. In O. H. Ibarra and Z. Dang, editors, *Proceedings of 10th International Conference on Developments in Language Theory (DLT 2006)*, volume 4036 of *LNCS*, pages 238–247. Springer, 2006.

[122] M. Ito and G. Lischke. Generalized periodicity and primitivity for words. *Mathematical Logic Quarterly*, 53(1):91–106, 2007.

[123] H. Iwai. End of the scaling theory and Moore's law. In Y.-L. Jiang, X.-P. Qu, et al., editors, *Proceedings of 16th International Workshop on Junction Technology (IWJT 2016)*, pages 1–4. IEEE, 2016.

[124] S. Jiang and S. Granick. *Janus Particle Synthesis, Self-Assembly and Applications*. Royal Society of Chemistry, 2012.

[125] J. Jirásek, G. Jirásková, and A. Szabzri. State complexity of concatenation and complementation. *International Journal of Foundations of Computer Science*, 16(3):511–529, 2005.

[126] G. Jirásková. State complexity of some operations on binary regular languages. *Theoretical Computer Science*, 330(2):287–298, 2005.

[127] G. Jirásková. On the state complexity of complements, stars, and reversals of regular languages. In M. Ito and M. Toyama, editors, *Proceedings of 12th International Conference on Developments in Language Theory (DLT 2008)*, volume 5257 of *LNCS*, pages 431–442. Springer, 2008.

[128] G. Jirásková. Concatenation of regular languages and descriptional complexity. *Theory of Computing Systems*, 49:306–318, 2011.

[129] B. G. Johansson. Agarose gel electrophoresis. *Scandinavian Journal of Clinical and Laboratory Investigation*, 29(sup124):7–19, 1972.

[130] C. R. Johnson. Automating the DNA computer: Solving n-variable 3-SAT problems. In C. Mao and T. Yokomori, editors, *Proceedings of 12th International Meeting on DNA Computing (DNA 2006)*, volume 4287 of *LNCS*, pages 360–373. Springer, 2006.

[131] J. Jones and A. Adamatzky. Computation of the travelling salesman problem by a shrinking blob. *Natural Computing*, 13(1):1–16, 2014.

[132] N. Jonoska, D. Kephart, and K. Mahalingam. Generating codes for DNA computing. *Congressus Numerantium*, 156:99–110, 2002.

[133] N. Jonoska and K. Mahalingam. Languages of DNA based code words. In J. Chen and J. Reif, editors, *Proceedings of 9th International Workshop on DNA Based Computers (DNA 9)*, volume 2943 of *LNCS*, pages 61–73. Springer, 2004.

[134] E. Kalle, M. Kubista, and C. Rensing. Multi-template polymerase chain reaction. *Biomolecular Detection and Quantification*, 2:11–29, 2014.

[135] A. Kameda, M. Yamamoto, A. Ohuchi, S. Yaegashi, and M. Hagiya. Unravel four hairpins! *Natural Computing*, 7:287–298, 2008.

[136] T. Kanagawa. Bias and artifacts in multitemplate polymerase chain reactions (PCR). *Journal of Bioscience and Bioengineering*, 96(4):317–323, 2003.

[137] P. D. Kaplan, Q. Ouyang, D. S. Thaler, and A. Libchaber. Parallel overlap assembly for the construction of computational DNA libraries. *Journal of Theoretical Biology*, 188(3):333–341, 1997.

[138] J. Karhumäki. On cube-free ω-words generated by binary morphisms. *Discrete Applied Mathematics*, 5(3):279–297, 1983.

[139] L. Kari. *On Insertion and Deletion in Formal Languages.* PhD thesis, University of Turku, 1991.

[140] L. Kari. Deletion operations: Closure properties. *International Journal of Computer Mathematics*, 52(1-2):23–42, 1994.

[141] L. Kari. On language equations with invertible operations. *Theoretical Computer Science*, 132(1-2):129–150, 1994.

[142] L. Kari, R. Kitto, and G. Thierrin. Codes, involutions, and DNA encodings. In W. Brauer, H. Ehrig, et al., editors, *Formal and Natural Computing: Essays Dedicated to Grzegorz Rozenberg*, volume 2300 of *LNCS*, pages 376–393. Springer, 2002.

[143] L. Kari and S. Konstantinidis. Language equations, maximality and error-detection. *Journal of Computer and System Sciences*, 70(1):157–178, 2005.

[144] L. Kari, S. Konstantinidis, and P. Sosík. Bond-free languages: Formalizations, maximality and construction methods. *International Journal of Foundations of Computer Science*, 16(5):1039–1070, 2005.

[145] L. Kari, M. S. Kulkarni, K. Mahalingam, and Z. Wang. Involutive Fibonacci words. *Journal of Automata, Languages and Combinatorics*, 26(3–4):255–280, 2021.

[146] L. Kari and E. Losseva. Block substitutions and their properties. *Fundamenta Informaticae*, 73(1):165–178, 2006.

[147] L. Kari and K. Mahalingam. Involutively bordered words. *International Journal of Foundations of Computer Science*, 18(5):1089–1106, 2007.

[148] L. Kari and K. Mahalingam. Watson-Crick conjugate and commutative words. In M. H. Garzon and H. Yan, editors, *Proceedings of 13th International Meeting on DNA computing (DNA 13)*, volume 4848 of *LNCS*, pages 273–283. Springer, 2008.

[149] L. Kari, K. Mahalingam, P. Pandoh, and Z. Wang. Primitivity of atom Watson-Crick Fibonacci words. *Journal of Automata, Languages and Combinatorics*, 27(1–3):151–178, 2022.

[150] L. Kari, G. Păun, G. Rozenberg, A. Salomaa, and S. Yu. DNA computing, sticker systems, and universality. *Acta Informatica*, 35(5):401–420, 1998.

[151] L. Kari, G. Păun, G. Thierrin, and S. Yu. At the crossroads of DNA computing and formal languages: Characterizing recursively enumerable languages using insertion-deletion systems. In H. Rubin and D. H. Wood, editors, *Proceedings of DIMACS Workshop DNA Based Computers III*, volume 48 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 329–346, 1999.

[152] L. Kari and P. Sosík. On the weight of universal insertion grammars. *Theoretical Computer Science*, 396(1):264–270, 2008.

[153] L. Kari and G. Thierrin. Contextual insertions/deletions and computability. *Information and Computation*, 131(1):47–61, 1996.

[154] L. Kari and G. Thierrin. Maximal and minimal solutions to language equations. *Journal of Computer and System Sciences*, 53(3):487–496, 1996.

[155] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller, J. W. Thatcher, et al., editors, *Proceedings of a Symposium on the Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer, 1972.

[156] H. Kim and Y.-S. Han. Non-overlapping inversion on strings and languages. *Theoretical Computer Science*, 592:9–22, 2015.

[157] L. B. Kish. Moore's law and the energy requirement of computing versus performance. *IEE Proceedings—Circuits, Devices and Systems*, 151(2):190–194, 2004.

[158] B. Kjos-Hanssen. Automatic complexity of Fibonacci and Tribonacci words. *Discrete Applied Mathematics*, 289:446–454, 2021.

[159] D. E. Knuth. *The Art of Computer Programming (Volume 1: Fundamental Algorithms).* Addison-Wesley, 1st edition, 1968.

[160] D. E. Knuth, J. H. Morris, Jr., and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[161] J. E. Kobza, S. H. Jacobson, and D. E. Vaughan. A survey of the coupon collector's problem with random sample sizes. *Methodology and Computing in Applied Probability*, 9(4):573–584, 2007.

[162] S. Kopecki. On iterated hairpin completion. *Theoretical Computer Science*, 412(29):3629–3638, 2011.

[163] S. Kosuri and G. M. Church. Large-scale de novo DNA synthesis: Technologies and applications. *Nature Methods*, 11(5):499–507, 2014.

[164] S. Kuroda. Classes of languages and linear-bounded automata. *Information and Control*, 7(2):207–223, 1964.

[165] L. F. Landweber and L. Kari. Universal molecular computation in ciliates. In L. F. Landweber and E. Winfree, editors, *Proceedings of DIMACS Workshop Evolution as Computation*, Natural Computing Series, pages 257–274. Springer, 2002.

[166] P. S. Landweber. Three theorems on phrase structure grammars of type 1. *Information and Control*, 6(2):131–136, 1963.

[167] P. Y. Lee, J. Costumbrado, C.-Y. Hsu, and Y. H. Kim. Agarose gel electrophoresis for the separation of DNA fragments. *Journal of Visualized Experiments*, 62(3923):1–5, 2012.

[168] L. Liu, Y. Song, H. Zhang, H. Ma, and A. V. Vasilakos. Physarum optimization: A biology-inspired algorithm for the steiner tree problem in networks. *IEEE Transactions on Computers*, 64(3):818–831, 2015.

[169] Q. Liu, L. Wang, A. G. Frutos, A. E. Condon, R. M. Corn, and L. M. Smith. DNA computing on surfaces. *Nature*, 403(6766):175–179, 2000.

[170] Z. Liu, R. W. Deibler, H. S. Chan, and L. Zechiedrich. The why and how of DNA unlinking. *Nucleic Acids Research*, 37(3):661–671, 2009.

[171] K. S. Lundberg, D. D. Shoemaker, M. W. Adams, J. M. Short, J. A. Sorge, and E. J. Mathur. High-fidelity amplification using a thermostable DNA polymerase isolated from Pyrococcus furiosus. *Gene*, 108(1):1–6, 1991.

[172] R. C. Lyndon and M. P. Schützenberger. The equation $a^M = b^N c^P$ in a free group. *Michigan Mathematical Journal*, 9:289–298, 1962.

[173] O. A. S. Lyon and K. Salomaa. Nondeterministic state complexity of site-directed deletion. In P. Caron and L. Mignot, editors, *Proceedings of 26th International Conference on Implementation and Application of Automata (CIAA 2022)*, volume 13266 of *LNCS*, pages 189–199. Springer, 2022.

[174] K. Mahalingam, A. Maity, P. Pandoh, and R. Raghavan. Block reversal on finite words. *Theoretical Computer Science*, 894:135–151, 2021.

[175] V. Manca and G. Franco. Computing by polymerase chain reaction. *Mathematical Biosciences*, 211(2):282–298, 2008.

[176] F. Manea, C. Martín-Vide, and V. Mitrana. Hairpin lengthening: Language theoretic and algorithmic results. *Journal of Logic and Computation*, 25(4):987–1009, 2015.

[177] F. Manea, R. Mercas, and V. Mitrana. Hairpin lengthening and shortening of regular languages. In H. Bordihn, M. Kutrib, et al., editors, *Languages Alive: Essays Dedicated to Jürgen Dassow on the Occasion of His 65th Birthday*, volume 7300 of *LNCS*, pages 145–159. Springer, 2012.

[178] F. Manea and V. Mitrana. Hairpin completion versus hairpin reduction. In S. B. Cooper, B. Löwe, et al., editors, *Proceedings of 3rd Conference on Computability in Europe (CiE 2007)*, volume 4497 of *LNCS*, pages 532–541. Springer, 2007.

[179] F. Manea, V. Mitrana, and T. Yokomori. Two complementary operations inspired by the DNA hairpin formation: Completion and reduction. *Theoretical Computer Science*, 410(4):417–425, 2009.

[180] C. Mao, T. H. LaBean, J. H. Reif, and N. C. Seeman. Logical computation using algorithmic self-assembly of DNA triple-crossover molecules. *Nature*, 407(6803):493–496, 2000.

[181] A. Marathe, A. E. Condon, and R. M. Corn. On combinatorial DNA word design. *Journal of Computational Biology*, 8(3):201–219, 2001.

[182] A. N. Maslov. Cyclic shift operation for languages. *Problemy Peredachi Informatsii*, 9(4):81–87, 1973.

[183] F. Massacci. Contextual reasoning is NP-complete. In B. Clancey and D. Welds, editors, *Proceedings of 13th National Conference on Artificial intelligence (AAAI 96)*, pages 621–626. AAAI Press, 1996.

[184] A. Mateescu, G. Păun, G. Rozenberg, and A. Salomaa. Simple splicing systems. *Discrete Applied Mathematics*, 84(1–3):145–163, 1998.

[185] G. Mauri and C. Ferretti. Word design for molecular computing: A survey. In J. Chen and J. Reif, editors, *Proceedings of 9th International Workshop on DNA Based Computers (DNA 9)*, volume 2943 of *LNCS*, pages 37–47. Springer, 2004.

[186] E. Meijering, O. Dzyubachyk, and I. Smal. Chapter nine—methods for cell and particle tracking. In P. M. Conn, editor, *Imaging and Spectroscopic Analysis of Living Cells: Optical and Spectroscopic Techniques*, volume 504 of *Methods in Enzymology*, pages 183–200. Academic Press, 2012.

[187] F. Mera and G. Pighizzini. Complementing unary nondeterministic automata. *Theoretical Computer Science*, 330(2):349–360, 2005.

[188] F. Mignosi and G. Pirillo. Repetitions in the Fibonacci infinite word. *RAIRO Theoretical Informatics and Applications*, 26(3):199–204, 1992.

[189] B. G. Mirkin. On dual automata. *Cybernetics*, 2(1):6–9, 1966.

[190] G. E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[191] G. E. Moore. Moore's law at 40. In D. C. Brock, editor, *Understanding Moore's Law: Four Decades of Innovation*, pages 67–84. Chemical Heritage Press, 2006.

[192] H. Mousavi, L. Schaeffer, and J. O. Shallit. Decision algorithms for Fibonacci-automatic words, I: Basic results. *RAIRO Theoretical Informatics and Applications*, 50(1):39–66, 2016.

[193] J. A. Myers, B. S. Curtis, and W. R. Curtis. Improving accuracy of cell and chromophore concentration measurements using optical density. *BMC Biophysics*, 6(1):4, 2013.

[194] T. Nakagaki, R. Kobayashi, Y. Nishiura, and T. Ueda. Obtaining multiple separate food sources: Behavioural intelligence in the physarum plasmodium. *Proceedings of the Royal Society of London Series B: Biological Sciences*, 271(1554):2305–2310, 2004.

[195] T. Nakagaki, H. Yamada, and Á. Tóth. Maze-solving by an amoeboid organism. *Nature*, 407(6803):470, 2000.

[196] G.-J. Nam, K. A. Sakallah, and R. A. Rutenbar. A new FPGA detailed routing approach via search-based boolean satisfiability. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(6):674–684, 2002.

[197] M. Nayak, A. S. Perumal, D. V. Nicolau, and F. C. M. J. M. van Delft. Bacterial motility behaviour in sub-ten micron wide geometries. In *Proceedings of 16th IEEE International New Circuits and Systems Conference (NEWCAS 2018)*, pages 382–384. IEEE, 2018.

[198] L. Németh. Fibonacci words in hyperbolic Pascal triangles. *Acta Universitatis Sapientiae, Mathematica*, 9(2):336–347, 2017.

[199] A. Nerode. Linear automaton transformations. *Proceedings of the American Mathematical Society*, 9(4):541–544, 1958.

[200] D. V. Nicolau, D. V. Nicolau Jr., G. Solana, K. L. Hanson, L. Filipponi, L. Wang, and A. P. Lee. Molecular motors-based micro- and nano-biocomputation devices. *Microelectronic Engineering*, 83(4–9):1582–1588, 2006.

[201] D. V. Nicolau Jr., K. Burrage, and D. V. Nicolau. Computing with motile bio-agents. In D. V. Nicolau, editor, *Biomedical Applications of Micro- and Nanoengineering III*, volume 6416, pages 220–228. SPIE, 2006.

[202] D. V. Nicolau Jr. et al. Parallel computation with molecular-motor-propelled agents in nanofabricated networks. *Proceedings of the National Academy of Sciences*, 113(10):2591–2596, 2016.

[203] S. Ogasawara and K. Fujimoto. Solution of a SAT problem on a photochemical DNA computer. *Chemistry Letters*, 34(3):378–379, 2005.

[204] M. Oltean and O. Muntean. Solving the subset-sum problem with a light-based device. *Natural Computing*, 8(2):321–331, 2009.

[205] D. S. Olton, C. Collison, and M. A. Werz. Spatial memory and radial arm maze performance of rats. *Learning and Motivation*, 8(3):289–314, 1977.

[206] T. Oshiba. Closure property of the family of context-free languages under the cyclic shift operation. *Institute of Electronics, Information and Communication Engineers*, 55(4):119–122, 1972.

[207] Q. Ouyang, P. D. Kaplan, S. Liu, and A. Libchaber. DNA solution of the maximal clique problem. *Science*, 278(5337):446–449, 1997.

[208] J.-J. Pansiot. Mots infinis de Fibonacci et morphismes itérés. *RAIRO Theoretical Informatics and Applications*, 17(2):131–135, 1983.

[209] S. Park, P. M. Wolanin, E. A. Yuzbashyan, P. Silberzan, J. B. Stock, and R. H. Austin. Motion to form a quorum. *Science*, 301(5630):188, 2003.

[210] G. Păun. Regular extended H systems are computationally universal. *Journal of Automata, Languages and Combinatorics*, 1(1):27–36, 1996.

[211] G. Păun, M. J. Pérez-Jiménez, and T. Yokomori. Representations and characterizations of languages in Chomsky hierarchy by means of insertion-deletion systems. *International Journal of Foundations of Computer Science*, 19(4):859–871, 2008.

[212] G. Păun, G. Rozenberg, and A. Salomaa. *DNA Computing: New Computing Paradigms*. Texts in Theoretical Computer Science. Springer, 1998.

[213] M. J. Pérez-Jiménez and F. Sancho-Caparrini. Solving knapsack problems in a sticker based model. In N. Jonoska and N. C. Seeman, editors, *Proceedings of 7th International Workshop on DNA-Based Computers (DNA 7)*, volume 2340 of *LNCS*, pages 161–171. Springer, 2002.

[214] A. S. Perumal, M. Nayak, V. Tokárová, O. Kašpar, and D. V. Nicolau. Space partitioning and maze solving by bacteria. In A. Compagnoni, W. Casey, et al., editors, *Proceedings of 11th EAI International Conference on Bio-inspired Information and Communication Technologies (BICT 2019)*, volume 289 of *LNICST*, pages 175–180. Springer, 2019.

[215] A. S. Perumal, Z. Wang, G. Ippoliti, F. C. M. J. M. van Delft, L. Kari, and D. V. Nicolau. As good as it gets: A scaling comparison of DNA computing, network biocomputing, and electronic computing approaches to an NP-complete problem. *New Journal of Physics*, 23(12):125001, 2021.

[216] C. G. Pick and J. Yanai. Eight arm maze for mice. *International journal of Neuroscience*, 21(1–2):63–66, 1983.

[217] N. A. Pierce and E. Winfree. Protein design is NP-hard. *Protein Engineering, Design and Selection*, 15(10):779–782, 2002.

[218] S. Pilo, G. Zizelski Valenci, M. Rubinstein, L. Pichadze, Y. Scharf, Z. Dveyrin, E. Rorman, and I. Nissan. High-resolution multilocus sequence typing for Chlamydia trachomatis: Improved results for clinical samples with low amounts of C. trachomatis DNA. *BMC Microbiology*, 21(1):28, 2021.

[219] G. Pirillo. On a combinatorial property of Fibonacci semigroup. *Discrete Mathematics*, 122(1):263–267, 1993.

[220] G. Pirillo. Fibonacci numbers and words. *Discrete Mathematics*, 173(1):197–207, 1997.

[221] D. Pixton. Regularity of splicing languages. *Discrete Applied Mathematics*, 69(1–2):101–124, 1996.

[222] D. Pixton. Splicing in abstract families of languages. *Theoretical Computer Science*, 234(1–2):135–166, 2000.

[223] G. Păun. On the splicing operation. *Discrete Applied Mathematics*, 70(1):57–79, 1996.

[224] G. Păun. DNA computing based on splicing: Universality results. *Theoretical Computer Science*, 231(2):275–296, 2000.

[225] G. Păun, G. Rozenberg, and T. Yokomori. Hairpin languages. *International Journal of Foundations of Computer Science*, 12(6):837–847, 2001.

[226] L. Qian and E. Winfree. Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, 332(6034):1196–1201, 2011.

[227] L. Qian, E. Winfree, and J. Bruck. Neural network computation with DNA strand displacement cascades. *Nature*, 475(7356):368–372, 2011.

[228] J. Qin and A. R. Wheeler. Maze exploration and learning in C. elegans. *Lab on a Chip*, 7(2):186–192, 2007.

[229] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal of Research and Development*, 3(2):114–125, 1959.

[230] J. L. Ramírez and G. N. Rubiano. On the k-Fibonacci words. *Acta Universitatis Sapientiae, Informatica*, 5(2):212–226, 2013.

[231] J. L. Ramírez and G. N. Rubiano. Properties and generalizations of the Fibonacci word fractal, exploring fractal curves. *The Mathematica Journal*, 16:1–25, 2014.

[232] J. L. Ramírez, G. N. Rubiano, and R. D. Castro. A generalization of the Fibonacci word fractal and the Fibonacci snowflake. *Theoretical Computer Science*, 528:40–56, 2014.

[233] J. A. Reuter, D. V. Spacek, and M. P. Snyder. High-throughput sequencing technologies. *Molecular Cell*, 58(4):586–597, 2015.

[234] E. Rich. *Automata, Computability and Complexity: Theory and Applications.* Pearson, 3rd edition, 2008.

[235] R. J. Roberts, T. Vincze, J. Posfai, and D. Macelis. REBASE—a database for DNA restriction and modification: Enzymes, genes and genomes. *Nucleic Acids Research*, 43(D1):D298–D299, 2014.

[236] W. A. Roberts and N. Van Veldhuizen. Spatial memory in pigeons on the radial maze. *Journal of Experimental Psychology: Animal Behavior Processes*, 11(2):241–260, 1985.

[237] J. A. Rose, K. Komiya, S. Yaegashi, and M. Hagiya. Displacement whiplash PCR: Optimized architecture and experimental validation. In C. Mao and T. Yokomori, editors, *Proceedings of 12th International Meeting on DNA Computing (DNA 12)*, volume 4287 of *LNCS*, pages 393–403. Springer, 2006.

[238] P. W. K. Rothemund, N. Papadakis, and E. Winfree. Algorithmic self-assembly of DNA Sierpinski triangles. *PLOS Biology*, 2(12):2041–2053, 2004.

[239] G. Rozenberg and A. Salomaa, editors. *Handbook of Formal Languages, Vol. 1: Word, Language, Grammar.* Springer, 1997.

[240] W. Rytter. The structure of subword graphs and suffix trees of Fibonacci words. *Theoretical Computer Science*, 363(2):211–223, 2006.

[241] K. Saari. Periods of factors of the Fibonacci word. In *Proceedings of 6th International Conference on Words (WORDS 07)*, pages 273–279. Institut de Mathématiques de Luminy, 2007.

[242] K. Sakamoto, H. Gouzu, K. Komiya, D. Kiga, S. Yokoyama, T. Yokomori, and M. Hagiya. Molecular computation by DNA hairpin formation. *Science*, 288(5469):1223–1226, 2000.

[243] A. Salomaa. *Formal Languages.* ACM Monograph Series. Academic Press, 1973.

[244] A. Salomaa, K. Salomaa, and S. Yu. Undecidability of the state complexity of composed regular operations. In A.-H. Dediu, S. Inenaga, et al., editors, *Proceedings of 5th International Conference on Language and Automata Theory and Applications (LATA 2011)*, volume 6638 of *LNCS*, pages 489–498. Springer, 2011.

[245] K. Salomaa and S. Yu. On the state complexity of combined operations and their estimation. *International Journal of Foundations of Computer Science*, 18(4):683–698, 2007.

[246] S. Scheinberg. Note on the boolean properties of context free languages. *Information and Control*, 3(4):372–375, 1960.

[247] P. Séébold. Sequences generated by infinitely iterated morphisms. *Discrete Applied Mathematics*, 11(3):255–264, 1985.

[248] P. Séébold. Fibonacci morphisms and Sturmian words. *Theoretical Computer Science*, 88(2):365–384, 1991.

[249] A. Semenov. Distributed computing based on container-component model. In A. Bogdan, editor, *Proceedings of 8th International Conference on Applied Mathematics and Mechanics in the Aerospace Industry (AMMAI 2020)*, volume 927 of *IOP Conference Series: Materials Science and Engineering*, page 012070. IOP Publishing, 2020.

[250] S. Shah, J. Wee, T. Song, L. Ceze, K. Strauss, Y.-J. Chen, and J. Reif. Using strand displacing polymerase to program chemical reaction networks. *Journal of the American Chemical Society*, 142(21):9587–9593, 2020.

[251] J. O. Shallit. A generalization of automatic sequences. *Theoretical Computer Science*, 61(1):1–16, 1988.

[252] J. O. Shallit. *A Second Course in Formal Languages and Automata Theory*. Cambridge University Press, 2009.

[253] S. Shioda. Coupon subset collection problem with quotas. *Methodology and Computing in Applied Probability*, pages 1–33, 2020.

[254] M. Sipser. *Introduction to the Theory of Computation*. Cengage Learning, 3rd edition, 2012.

[255] T. Song, A. Eshra, S. Shah, H. Bui, D. Fu, M. Yang, R. Mokhtar, and J. Reif. Fast and compact DNA logic circuits based on single-stranded gates using strand-displacing polymerase. *Nature Nanotechnology*, 14(11):1075–1081, 2019.

[256] K. B. Stolarsky. Beatty sequences, continued fractions, and certain shift operators. *Canadian Mathematical Bulletin*, 19(4):473–482, 1976.

[257] E. Stoschek, M. Sturm, and T. Hinze. DNA-Computing—ein funktionales Modell im laborpraktischen Experiment. *Informatik Forschung und Entwicklung*, 16(1):35–52, 2001.

[258] H. Su, J. Xu, Q. Wang, F. Wang, and X. Zhou. High-efficiency and integrable DNA arithmetic and logic system based on strand displacement synthesis. *Nature Communications*, 10(1):5390, 2019.

[259] A. Takahara and T. Yokomori. On the computational power of insertion-deletion systems. In M. Hagiya and A. Ohuchi, editors, *Proceedings of 8th International Workshop on DNA Based Computers (DNA 8)*, volume 2568 of *LNCS*, pages 269–280. Springer, 2002.

[260] Y. Takenaka and A. Hashimoto. DNA computing by competitive hybridization for maximum satisfiability problem. In *Proceedings of the 2002 Congress on Evolutionary Computation (CEC 02)*, volume 1, pages 472–476. IEEE, 2002.

[261] M. Takinoue and A. Suyama. Molecular reactions for a molecular memory based on hairpin DNA. *Chem-Bio Informatics Journal*, 4(3):93–100, 2004.

[262] M. Takinoue and A. Suyama. Hairpin-DNA memory using molecular addressing. *Small*, 2(11):1244–1247, 2006.

[263] J.-Y. Tinevez, N. Perry, J. Schindelin, G. M. Hoopes, G. D. Reynolds, E. Laplantine, S. Y. Bednarek, S. L. Shorte, and K. W. Eliceiri. TrackMate: An open and extensible platform for single-particle tracking. *Methods*, 115:80–90, 2017.

[264] E. C. Tolman and C. H. Honzik. Introduction and removal of reward, and maze performance in rats. *University of California Publications in Psychology*, 4(17):257–275, 1930.

[265] H. Tseng. DNA cloning without restriction enzyme and ligase. *Biotechniques*, 27(6):1240–1244, 1999.

[266] D. C. Tulpan, H. H. Hoos, and A. E. Condon. Stochastic local search algorithms for DNA word design. In M. Hagiya and A. Ohuchi, editors, *Proceedings of 8th International Workshop on DNA Based Computers (DNA 8)*, volume 2568 of *LNCS*, pages 229–241. Springer, 2003.

[267] J. C. Turner. Fibonacci word patterns and binary sequences. *Fibonacci Quarterly*, 26(3):233–246, 1988.

[268] J. C. Turner. The alpha and the omega of the Wythoff pairs. *Fibonacci Quarterly*, 27(1):76–86, 1989.

[269] F. C. M. J. M. van Delft, G. Ipolitti, D. V. Nicolau Jr., A. Sudalaiyadum Perumal, O. Kašpar, S. Kheireddine, S. Wachsmann-Hogiu, and D. V. Nicolau. Something has to give: Scaling combinatorial computing by biological agents exploring physical networks encoding NP-complete problems. *Interface Focus*, 8(6):20180034, 2018.

[270] F. C. M. J. M. van Delft, A. S. Perumal, A. van Langen-Suurling, C. de Boer, O. Kašpar, V. Tokárová, F. W. A. Dirne, and D. V. Nicolau. Design and fabrication of networks for bacterial computing. *New Journal of Physics*, 23(8):085009, 2021.

[271] J. C. Venter et al. The sequence of the human genome. *Science*, 291(5507):1304–1351, 2001.

[272] T. Vincze, J. Posfai, and R. J. Roberts. NEBcutter: A program to cleave DNA with restriction enzymes. *Nucleic Acids Research*, 31(13):3688–3691, 2003.

[273] X. Wang, Z. Bao, J. Hu, S. Wang, and A. Zhan. Solving the SAT problem using a DNA computing algorithm based on ligase chain reaction. *Biosystems*, 91(1):117–125, 2008.

[274] Z.-X. Wen and Z.-Y. Wen. Some properties of the singular words of the Fibonacci word. *European Journal of Combinatorics*, 15(6):587–598, 1994.

[275] D. Woods, D. Doty, C. Myhrvold, J. Hui, F. Zhou, P. Yin, and E. Winfree. Diverse and robust molecular algorithms using reprogrammable DNA self-assembly. *Nature*, 567(7748):366–372, 2019.

[276] H. L. Wu. On the Properties of Primitive Words. Master's thesis, Institute of Applied Mathematics, Chung-Yuan Christian University, 1992.

[277] X.-Y. Xu, X.-L. Huang, Z.-M. Li, J. Gao, Z.-Q. Jiao, Y. Wang, R.-J. Ren, H. Zhang, and X.-M. Jin. A scalable photonic computer solving the subset sum problem. *Science Advances*, 6(5):eaay5853, 2020.

[278] S. Yu. State complexity of regular languages. *Journal of Automata, Languages and Combinatorics*, 6(2):221–234, 2001.

[279] S. Yu, Q. Zhuang, and K. Salomaa. The state complexities of some basic operations on regular languages. *Theoretical Computer Science*, 125(2):315–328, 1994.

[280] S.-S. Yu. *Languages and Codes.* Tsang Hai Book Publishing, 2005.

[281] S.-S. Yu and Y.-K. Zhao. Properties of Fibonacci languages. *Discrete Mathematics*, 224(1):215–223, 2000.

[282] F. Zeng, Z. Hao, P. Li, Y. Meng, J. Dong, and Y. Lin. A restriction-free method for gene reconstitution using two single-primer PCRs in parallel to generate compatible cohesive ends. *BMC Biotechnology*, 17(1):32, 2017.

[283] D. Y. Zhang, A. J. Turberfield, B. Yurke, and E. Winfree. Engineering entropy-driven reactions and networks catalyzed by DNA. *Science*, 318(5853):1121–1125, 2007.

[284] F. Zhang, J. Nangreave, Y. Liu, and H. Yan. Structural DNA nanotechnology: State of the art and future perspective. *Journal of the American Chemical Society*, 136(32):11198–11211, 2014.

[285] S. W. Zhang, K. Bartsch, and M. V. Srinivasan. Maze learning by honeybees. *Neurobiology of Learning and Memory*, 66(3):267–282, 1996.