UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Renan Albuquerque Marks

DNA Strand Displacement (DSD) as a Hardware Substrate for Digital, Analog and Machine Learning Systems

Belo Horizonte
2023

Renan Albuquerque Marks

DNA Strand Displacement (DSD) as a Hardware Substrate for Digital, Analog and Machine Learning Systems

Final Version

Dissertation presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Prof. Dr. Omar Paranaiba Vilela Neto

Belo Horizonte
2023

© 2023, Renan Albuquerque Marks.
Todos os direitos reservados

Marks, Renan Albuquerque.

M346d DNA Strand Displacement (DSD) as a hardware substrate for digital, analog and machine learning systems [recurso eletrônico] / Renan Albuquerque Marks. 2023.
1 recurso online (114 f. il, color.): pdf.

Orientador: Omar Paranaiba Vilela Neto.
Tese (Doutorado) - Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação.
Referências: f. 96 -108.

1. Computação – Teses. 2. Nanocomputação – Teses. 3. Computação em DNA – Teses. I. Vilela Neto, Omar Paranaiba. II. Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação. IV. Título.

CDU 519.6*73(043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa
CRB 6/1510 - Universidade Federal de Minas Gerais - ICEx

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

DNA STRAND DISPLACEMENT (DSD) AS A HARDWARE SUBSTRATE FOR DIGITAL, ANALOG AND MACHINE LEARNING SYSTEMS

RENAN ALBUQUERQUE MARKS

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores(a):

Prof. Omar Paranaiba Vilela Neto - Orientador
Departamento de Ciência da Computação - UFMG

Profa. Gisele Lobo Pappa
Departamento de Ciência da Computação - UFMG

Prof. Daniel Fernandes Macedo
Departamento de Ciência da Computação - UFMG

Prof. Ricardo dos Santos Ferreira
Departamento de Informática - UFV

Prof. Nalvo Franco de Almeida Júnior
Faculdade de Computação - UFMS

Doutor Marcos Viero Guterres
Departamento de Química - UFLA

Belo Horizonte, 06 de outubro de 2023.

Documento assinado eletronicamente por **Omar Paranaiba Vilela Neto**, **Coordenador(a) de curso de pós-graduação**, em 10/10/2023, às 09:04, conforme horário oficial de Brasília, com

fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Daniel Fernandes Macedo**, **Professor do Magistério Superior**, em 18/10/2023, às 13:54, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Ricardo dos Santos Ferreira**, **Usuário Externo**, em 23/10/2023, às 09:57, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Marcos Viero Guterres**, **Usuário Externo**, em 24/10/2023, às 13:15, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Nalvo Franco de Almeida Junior**, **Usuário Externo**, em 21/11/2023, às 10:03, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

Documento assinado eletronicamente por **Gisele Lobo Pappa**, **Professora do Magistério Superior**, em 21/11/2023, às 11:13, conforme horário oficial de Brasília, com fundamento no art. 5º do Decreto nº 10.543, de 13 de novembro de 2020.

A autenticidade deste documento pode ser conferida no site https://sei.ufmg.br/sei/controlador_externo.php?acao=documento_conferir&id_orgao_acesso_externo=0, informando o código verificador **2689170** e o código CRC **91AF8572**.

Referência: Processo nº 23072.261037/2023-69 SEI nº 2689170

To my mother, who taught me everything about what matters in life and always encouraged me. To students and researchers who work to better understand the universe's mysteries, improving everyone's lives along the way.

Acknowledgments

This research project has been an integral part of my routine. It would not have been possible without the guidance and support of people who were always there to celebrate and encourage every move made. I want to thank my parents for showing me the value of education from an early age. I am grateful for the fellowship of colleagues who have become friends and accompanied me on this journey. I thank all the people in the role of professors and researchers who taught me the most important lessons. I want to thank all the health professionals who helped me recover after my accident. I would like to thank my advisor for his zeal, attention, and professionalism during the development of this work. So, I thank and give my respects to all who were involved throughout this academic work. Finally, I would like to acknowledge the funding agencies CNPq/CAPES and UFMS (Universidade Federal de Mato Grosso do Sul) for the support.

Thank you all!

Resumo

Houve um tempo em que a evolução tecnológica de sistemas computacionais era constante e célere. A alta velocidade no desenvolvimento de novas técnicas de miniaturização significava que métodos de fabricação menores e mais eficientes ficavam disponíveis após alguns meses de pesquisa e eram empregados no desenvolvimento de novos circuitos integrados. Embora isso fosse verdade nas últimas décadas, este cenário mudou recentemente devido aos limites impostos pela física. Cada nova geração de dispositivos enfrenta novos desafios de miniaturização dos transistores, principalmente por causa do consumo de energia e dissipação de calor devido à natureza da física quântica nesta escala. À medida que a inteligência artificial se torna mais prevalente devido à necessidade de processar uma enorme quantidade de dados rapidamente em paralelo, o hardware será um gargalo. Várias soluções para este problema de hardware estão sendo propostas e testadas. Algumas delas envolvem o uso de novos paradigmas para construir sistemas computacionais além do silício, tais como o uso de luz ou moléculas orgânicas como substrato para realizar a computação. Uma das alternativas é usar o DNA como hardware "úmido" como um complemento aos computadores tradicionais de silício, acelerando alguns cálculos específicos *in vitro* ou *in vivo*. As capacidades de processamento massivamente paralelo e baixo consumo de energia do DNA o tornam a alternativa perfeita para a implementação de novas aplicações biológicas, que vão desde o diagnóstico de doenças a novos medicamentos para o tratamento de várias doenças. Esta tese propõe que a tecnologia de deslocamento de fitas de DNA (DNA Strand Displacement, DSD) é passível de ser utilizada como um substrado de hardware para desenvolvimento de circuitos moleculares modulares digitais, analógicos e de aprendizado de máquina. Assim, para atingir este objetivo, esta tese apresentará a contribuição de novas ferramentas de *Computer Aided Design* (CAD) desenvolvidas que amparam a modularização, composição, projeto, simulação e validação dos circuitos moleculares baseados em DSD.

Palavras-chave: Nanocomputação, Computação Molecular, Circuitos em DNA

Abstract

There was a time when the evolution of computer systems technology was fast and steady. The development rate of new miniaturization techniques was high, meaning that smaller and more efficient fabrication methods became available and employed to develop new integrated circuits after a few months. Although this was true in the past decades, this is no longer true due to physics limits. Each new generation of devices faces transistor miniaturization challenges mainly because of power consumption and heat dissipation due to the nature of quantum physics at this scale. As artificial intelligence becomes more prevalent due to the need to process an enormous amount of data in parallel and fast, the hardware will be a bottleneck. Several solutions to this hardware problem are being proposed and tested. Some of them involve trying new paradigms to build computer systems beyond silicon, such as using light or organic molecules as a substrate to carry out the computation. One of the alternatives is using DNA as "wet" hardware to complement the traditional silicon computers, accelerating some specific computations *in vitro* or *in vivo*. DNA's massively parallel processing capabilities and low power consumption make it the perfect alternative to implementing new medical applications, ranging from diagnostic of diseases to new drugs to treat various illnesses. This thesis proposes that DNA Strand Displacement (DSD) technology can be a hardware substrate for developing digital, analog, and machine-learning modular molecular circuits. Thus, to achieve this goal, this thesis will present the contribution of new *Computer Aided Design* (CAD) tools developed that support the modularization, composition, design, simulation, and validation of molecular circuits based on DSD.

Keywords: Nanocomputing, Molecular Computing, DNA Circuits

List of Figures

1.1 Main Antikythera mechanism fragment. The mechanism consists of a complex system of 30 wheels and plates. The machine is dated around 89 B.C. and comes from the wreck found off the island of Antikythera. National Archaeological Museum, Athens, No. 15987. (59) 17

2.1 Graphical representation of data from Table 2.1. Adapted from (90) 28
2.2 Simulation results for the equation 2.14. Both concentration and time axis are in arbitrary units. 30
2.3 The structure of DNA showing with detail the structure of the four bases, adenine, cytosine, guanine and thymine, and the location of the major and minor groove (11). 32
2.4 Example of three forms of representation of DNA molecules (104). The numbers denote domains; stared numbers denote complementary domains. 32
2.5 A DSD network (figure produced with (49)) with dynamic behavior analog to the unimolecular chemical reaction of the Equation 2.16. 33
2.6 Behavior comparison between a formal unimolecular chemical reaction and its equivalent implementation in DSD mechanism shown in Equation 2.16. 34
2.7 Four SWAP iterations of K-medoids algorithm. 41
2.8 Design flow of DSD circuits. Step (I) is the design of generic CRN; Step (II) is the compilation to a specific encoding of DSD CRN; Step (III) is the behavioral simulation of the reactions; Step (IV) is the syntesis of DNA strands. 42

4.1 Circuit for a half-adder containing an XOR gate generating the Sum signal and AND gate generating the Carry-Out signal. 55
4.2 Simulation result of code from Algorithm 5; The respective concentrations of output signals Carry-out and Sum are shown (*complementary representation*). 56
4.3 Simulation result of majority gate; The respective concentrations of outputs signals are shown (*complementary representation*). 58
4.4 Full-adder circuit schema with only Majority Gates. 60
4.5 Simulation result of a full-adder using majority gates; The respective concentrations of output signals, carry-out (Cout) and Sum, are shown in *complementary representation*. 61
4.6 Circuit to build a JK-Flip-Flop and a T-Flip-Flop from D-Flip-Flop. 62

4.7 Simulation results of the JK-Flip-Flop. Each vertical black bar, from left to right, indicates the moment when the JK-Flip-Flop (output value Q) is being set (J=1, K=0), reset (J=0, K=1), toggled (J=1, K=1) and in hold state (J=0, K=0), respectively. The axis are shown in arbitrary units (a.u.). 64
4.8 Full Adder using Majority Gate with generate and propagate output signals. . 65
4.9 4-bit Adder with Carry Look-ahead circuit. 66
4.10 Simulation result of output signal S of each individual full-adder when sums 3 + 7 resulting in 10 (1010 - binary coded). The axis are shown in arbitrary units (a.u.). 67

5.1 This figure shows the hybridization between strands. A line represents the presence of a hydrogen bond between bases; a dot otherwise represents the lack of a hydrogen bond. Please note that strands S_2 and S_4 are reversed when compared with Table 5.1, because of the opposite 5' to 3' hybridization. 72
5.2 Example circuit for one iteration of clustering algorithm with four species S_1, S_2, S_3 and S_4. Two of them (S_1 and S_3) are used as medoids to classify species S_2 and S_4. Due to the complementarity ratio of both S_2 and S_4 to S_1 and S_3 being different, S_2 and S_4 will react with the Counter Gates (C) at different speeds. This will generate different concentrations of count signals. When reacting with the Join Gate (J), these signals will produce the Mistrust Signal, used in classification. The greater difference between input concentrations of the Join Gate, the higher its output concentration is. 73
5.3 Species S_1 and S_3 will be used as medoids, so when activating their respective Counter Builder (CB) gates in yellow, they generate their respective Counters (C_{S_1} and C_{S_3}). Species S_1 and S_3 are inactivated by Counter Builder gates producing species I_{S_1} and I_{S_3}. 74
5.4 The remaining input species S_2 and S_4 react with the counters gates C_{S_1} and C_{S_3}. The input species S_2 are consumed by C_{S_1} and C_{S_3} (dashed lines) being converted into its inactive state I_{S_2}. The same happens with input species S_4 consumed by C_{S_1} and C_{S_3} (dotted lines) being converted into its inactive state I_{S_4}. The velocity of consumption of both S_2 and S_4 are defined by their "affinity" with counters C_{S_1} and C_{S_3}, as explained in Section 5.1.1. Both counter gates C_{S_1} and C_{S_3} produces intermediate species "inClassS_1" and "inClassS_3" which concentration encodes the "affinity" of input species S_2 and S_4 to species S_1 and S_3. 75

5.5 Both counter gates C_{S_1} and C_{S_3} produces intermediate species "inClassS_1" and "inClassS_3" which concentration encodes the "affinity" of input species S_2 and S_4 to species S_1 and S_3. For example: the Join (J) gate will pair each copy of the intermediate species "S_2inClassS_1" and "S_2inClassS_3" to generate the output signal "Mistrust$_{S_2}$" which high concentration levels means high mistrust in classification of species S_2. 77
5.6 At the end of one iteration, the Restorer (R) gate is responsible for "cleaning" the solution from the intermediate species (such as "inClass" and "Mistrust" signals) and reverting the inactivation of the input species. In our example, the species "I_{S_1}", "I_{S_2}", "I_{S_3}" and "I_{S_4}" are restored to their previous state, i.e., the active species "S_1", "S_2", "S_3" and "S_4", respectively. These active species are, then, ready to be used in a new iteration that will soon start. 78
5.7 Heatmap showing reaction rates for each pair of strands of Table 5.1 calculated by Multistrand. 83
5.8 Mistrust signals from the first iteration using the species S_1 and S_3 as medoids. 84
5.9 InClass signals from the first iteration using the species S_1 and S_3 as medoids. 84
5.10 Mistrust signals from the last iteration using the species S_3 and S_4 as medoids. 85
5.11 InClass signals from the last iteration using the species S_3 and S_4 as medoids. 85
5.12 The evolution of simplified silhouette cost through the iterations. 86
5.13 Sample of digits from MNIST dataset. 87
5.14 The five steps, from (a) to (e), needed to encode a picture containing hand-drawn digit five from MNIST dataset. 88
5.15 Reaction rates between sixteen samples of MNIST dataset. 88
5.16 Final clusters of digits from Figure 5.15a. 89
5.17 Courier 11 bitmap font downloaded from Monobit website. 90
5.18 Generated dataset of glyphs 1, 3, 7, and 8 with respective mutations generated by applying 10% of "salt & pepper" noise. 91
5.19 Reaction rates between sixteen samples of generated Courier 11 dataset. . . . 91
5.20 Resulting clusters after runing the molecular K-Medoids algorithm on dataset from Figure 5.19. 92

List of Tables

2.1 Experimental data collected from an elementary hipotetical $A \longrightarrow B$ reaction. Time in hours and concentrations in mole (M). Adapted from (90). 27

3.1 Table showing the comparison between the Machine Learning algorithms present in the literature and our proposal. 50

4.1 List of digital circuit elements implemented in DNAr-Logic. Specifically, the NOT gate does not use species nor reactions because an invert operation is "free" in the dual-rail (*complementary representation*) encoding. Previously proposed the gates are signaled by * (42). 53

5.1 Example DNA species. 71

5.2 Average results of seven experiments that were run with six cost functions. Each line shows the experiments optimized by the respective cost function named and each column shows the calculated costs for each experiment. The color grading were done by columns, i.e., on each column the best cost is in green and the worst cost in red. 81

5.3 Clusters from the final iteration of K-Medoids algorithm. 86

A.1 Table describing each species acronyms used in Equation A.1 for Counter Builder (CB) Gate. 110

A.2 Table describing each species acronyms used in Equation A.2 for Counter (C) Gate. 110

A.3 Table describing each species acronyms used in Equation A.3 for Join (J) Gate. 111

A.4 Table describing each species acronyms used in Equation A.4 for Restorer (R) Gate. 112

List of Algorithms

1 K-means algorithm. 37
2 Partition Around Medoids BUILD phase algorithm (Adaptated from (43)). 38
3 Partition Around Medoids SWAP phase algorithm (Adaptated from (43)). . 39
4 DNAr code to simulate the unimolecular reaction $A \to B$. 43

5 Code example in R using DNAr-Logic to build an half-adder circuit. 55
6 Code example of DNAr-Logic implementation of majority gate. 59
7 Code example to build an full-adder circuit using majority gates. 60
8 Code example to build an JK-Flip-Flop circuit using basic logic gates and a D-Flip-Flop. 63

Acronyms

API Application Programming Interface.

CAD Computer Aided Design.

CLA Carry Look-Ahead.

CRN Chemical Reaction Network.

DNA Deoxyribonucleic acid.

DSD DNA Strand Displacement.

mRNA Messenger RNA.

ODE Ordinary Differential Equation.

PCR Polymerase Chain Reaction.

RC Reservoir Computing.

RNA Ribonucleic acid.

Contents

1 Introduction **17**

1.1 Historical Background: What is a computer? . . . 17

1.1.1 "End" of Moore's Law . . . 19

1.1.2 DNA as a Hardware Substrate . . . 20

1.2 Goal, Contributions, Organization . . . 21

1.2.1 Goal . . . 21

1.2.2 Contributions . . . 21

1.2.3 Text Organization . . . 22

2 Background **23**

2.1 Chemical Reaction Networks . . . 23

2.1.1 Chemical Reactions . . . 23

2.1.2 Stoichiometric Coefficients . . . 24

2.1.3 Reversible Reactions . . . 25

2.1.4 Equilibrium Constant . . . 25

2.1.5 Concentration of Solutions . . . 26

2.1.6 Chemical Reaction Rate . . . 26

2.1.7 Chemical Reaction Network . . . 29

2.2 DNA Strand Displacement Mechanism . . . 30

2.2.1 Desoxyribonucleic Acid . . . 31

2.2.2 Strand Displacement Mechanism . . . 31

2.3 Machine Learning . . . 34

2.3.1 Supervised Learning . . . 35

2.3.2 Unsupervised Learning . . . 35

2.3.2.1 K-Means . . . 36

2.3.2.2 K-Medoids . . . 37

2.4 DNA Computing Design Automation Tools . . . 40

3 Related works and Historical Background **45**

3.1 DNA Computing . . . 45

3.2 CRN Machine Learning . . . 47

4 Design of Digital and Analog DNA Systems **51**

4.1 DNAr-Logic . . . 51

4.1.1 Methodology . . . 52
4.1.2 Design . . . 53
4.1.3 Examples, Simulations and Results . . . 54
4.1.3.1 Half-Adder . . . 54
4.1.3.2 Logic Majority Gate . . . 56
4.1.3.3 Full Adder with Majority Gates . . . 59
4.1.3.4 JK-Flip-Flop . . . 62
4.1.3.5 4-bit Adder with Carry Look-ahead . . . 65
4.1.4 Discussion . . . 67
4.2 DNAr-Analog . . . 68

5 Implementation of DNA Clustering 69
5.1 K-medoids in DNA . . . 69
5.1.1 Clustering Based on DNA Domain Similarity . . . 70
5.1.2 Algorithm implementation in CRN . . . 72
5.1.2.1 Counter Builder Gate . . . 73
5.1.2.2 Counter Gate . . . 75
5.1.2.3 Join Gate . . . 76
5.1.2.4 Restorer Gate . . . 78
5.1.2.5 Iteration Setup . . . 79
5.2 Simulations and Results . . . 79
5.2.1 Simulation Framework . . . 80
5.2.2 Convergence Criteria . . . 80
5.2.3 Simulation Results . . . 82
5.2.3.1 Case Study: Four strands, Nine bases . . . 82
5.2.3.2 Case Study: Digit Classification . . . 86
5.2.4 Discussion . . . 92

6 Conclusions and Future Works 93
6.1 Conclusions . . . 93
6.2 Future Works . . . 95

References 96

Appendix A Gate Implementation Details 109
A.1 Counter Builder Gate . . . 109
A.2 Counter Gate . . . 109
A.3 Join Gate . . . 111
A.4 Restorer Gate . . . 112

Appendix B Publications 113

Chapter 1

Introduction

1.1 Historical Background: What is a computer?

The term "computation" can have different definitions. Here, we define "computation" as a process that begins with initial conditions (such as input) and gives an output after following a definite set of rules. In other words, a computation transforms an input into output by applying a sequence of operations. This sequence of strict instructions, typically used to perform a computation, is called an "algorithm." So, a computation (and an algorithm) *per se* are not directly related to the physical substrate used to perform it.

Figure 1.1: Main Antikythera mechanism fragment. The mechanism consists of a complex system of 30 wheels and plates. The machine is dated around 89 B.C. and comes from the wreck found off the island of Antikythera. National Archaeological Museum, Athens, No. 15987. (59)

Throughout time, humans have seen the necessity of doing complex repetitive tasks with quality and ease. Figure 1.1 shows the Antikythera mechanism. It was the first known and considered an analog mechanical computer (34). Analog mechanical computers were made from mechanical components like gears, levers, and other devices that can carry

computation. The Antikythera mechanism's purpose, being a mechanical computer, is now understood to track and calculate the cycles of the Solar System. It was found in 1901 on the Greek island of Antikythera and has been dated to approximately 100 BC. It incorporates a sophistication that would not reappear until a thousand years later.

Another example is the loom machine that facilitates the fabrication of cloth and tapestry: the Jacquard Machine. Joseph Marie Jacquard patented this machine in 1804 (44), and it is considered a significant step in the history of computing hardware because of its use of punched cards to automate the weaving of complex patterns in fabric and tapestry. The cards controlled the machine weaving ability and could be simply changed at any time to alter the patterns weaved into the textile. Computer pioneer Charles Babbage used the same medium later in 1837, i.e., punch cards, to store programs that would run in his Analytical Engine (62). Finally, at the dusk of the 19th century, Herman Hollerith invented the punched card tabulating machine, patented in 1884, used for the 1890 US Census (38). The data processing industry that used punched-card technology significantly developed at the beginning of the 20th century. Punched cards remained in use in computing up until the mid-1980s.

The work of Shannon (81) helped accelerate the aforementioned remarkable development of computers in the first half of the 20th century by introducing the idea of using electrical switches to build digital logic circuits. Nevertheless, in the early twenty century, other mechanical computers were still built using mechanical components, like the Mark I Fire Control Computer, deployed by the United States Navy during World War II (36). Another mechanical computer was MONIAC, used to calculate the workings of the economy (12).

In 1945, an incomplete 101-page document written by John von Neumann contained the first published description of the logical design of a computer using the stored-program concept, which has come to be known as the von Neumann architecture. It was known as "The First Draft of a Report on the EDVAC" (commonly shortened to First Draft) and was distributed by Herman Goldstine, the security officer on the classified ENIAC project (64). Next came the electromechanical computers made from switches and relay logic, faster and capable of working with digital data (95).

In recent decades, with the help of the Computer Science field, the electronic industry has increased its influence in the life of people, providing unimaginable technological devices which can assist us in many practical problems of our lives and jobs. Nowadays, electronic digital computers practically run all of the world's infrastructure. Due to this incredible success, people with no expertise in areas of theory of computation tend to think of computers as something intrinsic to electronic devices.

1.1.1 "End" of Moore's Law

Nowadays, electronic computers that dominate the market rely on the "transistor" to work: a small electric switch with no moving parts. A transistor is like an electrical bridge that operates in two states: when it is on, it allows the flow of electric current; when it is off, it blocks the passage of electric current. What controls this behavior is a small electric potential (also known as voltage) applied to one of the transistor terminals. Different logic gates used in electronic computers use various transistors in their implementation. Four and two transistors in different circuit configurations can function as the universal NAND (not-and) and NOR(not-or) logic gates, respectively. With these universal logic gates, it is possible to create any logic circuit in computers.

Since 1970, engineers and scientists have worked to reduce the size of the fabricated transistor. This size reduction brought many benefits, such as a reduction in power consumption and a higher switching speed. In addition, with each transistor occupying less space, more of them could be used to create better-performing computation circuits. This size reduction led to the famous "Moore's Law," in which Gordon Moore observed that "the number of transistors in a dense integrated circuit (IC) doubles about every two years" (61). So, the big transistors of the past quickly reached millimeters, micrometers, and nanometers in size in the following decades.

Nevertheless, since the middle of the 2000s, a downside to this heavy reduction in size was discovered: the smaller the transistor, the greater the leakage current it possesses. This leakage current implies that tiny transistors consume power even when off. Also, modern computers have microprocessors with billions of transistors, all switching in Gigahertz frequency. Therefore, a high concentration of transistors simultaneously working in high frequencies has high power consumption, which leads to heat damaging the processor. This undesirable effect led to the end of "Dennard Scaling," which noted in 1974 that as transistors become smaller, their power density stays constant (28, 27, 13).

Also, as transistors get smaller and smaller, another problem arises: the inherent limitations of the size of atoms. Even if technological advances allow us to fabricate transistors of one atom in length, after that, they cannot be reduced anymore. This physical barrier inflicted by nature imposes technical challenges confronting the electronic chip industry due to the miniaturization of the electrical circuits to the nanometer scale (18, 80). These challenges brought to focus that the physical limits of the lithography process and electronic components will be soon reached, a technical issue that has motivated the development of new methods and paradigms for computation.

So, in recent years this knowledge of the inevitable end of fast progress in electronic computers has received a renewed interest from the scientific community. Researchers are looking for new properties in the physical and chemical systems that could be appropriate

to promote innovations in the concept of logical and programmable computer devices.

1.1.2 DNA as a Hardware Substrate

Maybe, one of the most intriguing novelties that surged in the last years is the use of the Deoxyribonucleic acid (DNA) molecule as a physical substrate to compute. DNA as a biological entity is a molecule composed of two polymeric strands of nucleotides containing the nitrogenous bases Adenine, Thymine, Guanine, and Cytosine (A, T, G, and C) in a unique sequence for each. However, DNA molecules also can be obtained in the laboratory by chemical synthesis. The sequence patterning of bases in a strand of DNA can store codified information, and specific molecules can manipulate these sequences, named enzymes, that cut and link strands of DNA in a programmed way. It was the chemical system used by Adleman to prove that DNA molecules can be used to perform computation, where he solved the Travel's Salesman problem (1).

Further advances in the manipulation of DNA molecule have made it possible to process the information without using enzymes. DNA Strand Displacement (DSD) reaction is an enzyme-free molecular technique to exchange a strand of DNA duplex (DNA complex formed by two strands bonded by their base pairs) with another single strand (input) of DNA. It is based on the hybridization of two complementary strands of DNA via Watson-Crick base pairing (A-T and C-G) and makes use of the branch migration process (82).

The DSD technique is incredibly powerful and energy-efficient. It is powerful because it can be described as a Chemical Reaction Network (CRN), which is Turing Complete (55, 31), meaning it has the same computation capability as all computers. Energy-efficient because it can resolve challenging computing problems known as NP-Complete and NP-Hard using a small fraction of energy and time compared with electronic computers (26). It is excellent to perform computations at an energy-efficient level because it opens doors to many possible applications. From a strictly computational standpoint, it allows a highly parallel biological computer to solve instances of NP-Complete problems faster than traditional computers. From a health perspective, it enables the development of intelligent drugs (8) and diagnostics such as sensing and actuation on diseases to release drugs in a controlled way (2, 9).

DNA Strand Displacement (DSD) has already been used for the construction of autonomous molecular motors (100), DNA nanostructures (103) and molecular logic gates (72). It can also implement generic CRNs (83, 71, 21).

1.2 Goal, Contributions, Organization

1.2.1 Goal

The main goal of this work is to explore the proven computing capabilities of DNA Strand Displacement (DSD) technology as a hardware substrate for developing digital, analog, and machine-learning modular molecular circuits. Thus, to achieve this goal, this thesis will work towards applying the principles of abstraction and scaling of design.

The abstraction is obtained by suppressing specific implementation details from the designer. A tool will automatically handle these details to allow the designer to scale the design up, i.e., to focus on creating more complex molecular circuits due to worrying less about specific molecular implementation details.

1.2.2 Contributions

Given the objectives presented above, this work focused on providing ways and methods to make feasible and quickly iterate the development of digital, analog, and machine learning molecular circuits based on DSD. The contributions of this work are listed:

Constructive Methodology for Molecular Circuits: This thesis proposed and developed a constructive methodology to implement software tools capable of abstracting the construction of molecular circuits and handling the specific implementation details for the designer;

Digital Molecular Circuit Software Tool: with the previous methodology, a molecular circuit design automation tool named DNAr-Logic was developed (57) and showed its capabilities in combinational and sequential circuits (56). The library allows modularization and programmatically assists the development of digital molecular circuits modeled as generic CRNs.

Analog Molecular Circuit Software Tool: The development of DNAr-Logic led to contribute (69, 67) to the initial development of another software tool named DNAr-Analog, inspired by DNAr-Logic and focused on analog molecular circuits (68).

Unsupervised Learning CRN Algorithm: also based on previous methodology, the K-medoids Unsupervised Algorithm is implemented as a CRN DSD circuit able to cluster similar DNA strands by their complementarity. For the implementation, reusable CRN gates that are not destroyed (i.e., unreversible inactivated) after use were developed and can be reused throughout iterations' runs.

1.2.3 Text Organization

This thesis is organized in 5 more chapters as follows:

Chapter 2: Presents some multidisciplinary concepts needed to comprehend the proposed work, such as chemical reactions, DSD mechanism, artificial intelligence, and machine learning fields.

Chapter 3: Presents the literature review of the most important works about computing systems made of biological components and works done to bring machine learning to Chemical Reaction Networks (CRNs).

Chapter 4: Presents part of the contributions developed by this work, such as the Constructive Methodology for Molecular Circuits and DNAr-Logic software package.

Chapter 5: Presents another part of the contributions developed by this work, such as the implementation of the K-medoids unsupervised learning algorithm in Chemical Reaction Networks (CRNs).

Chapter 6: Finally, the Conclusions and suggestions for future works are presented.

Chapter 2

Background

This chapter will lay the foundation of some multidisciplinary concepts needed to comprehend the work proposed here. In the first two Sections, 2.1 and 2.2, we will, respectively, introduce concepts of chemical reactions and DNA strand displacement mechanism. Then, in Section 2.3, introduces concepts of artificial intelligence and machine learning fields, such as supervised and unsupervised learning. Finally, Section 2.4 enumerate some tools used in design automation of DNA circuits.

2.1 Chemical Reaction Networks

In this section, we describe the basic concepts of chemical reactions. They are essential to understand how it is possible to use the language of chemical kinetics as a formal language to program a "wet" hardware made of chemical compounds.

2.1.1 Chemical Reactions

A chemical reaction is a process in which one or more substances, the reactants, are converted to one or more different substances, the products (87).

All chemical reactions are defined by a set of species (substances) and a rate constant (described later). One reaction with a set of species $S = \{A, B\}$, which A transforms in B, and a chemical reaction rate constant k is described using the following notation (also called "chemical equation"):

$$A \xrightarrow{k} B\,. \tag{2.1}$$

The species(substances) can be categorized as either reactants or products. Reactants are the species that will react and transform into the products, which are species

produced by the chemical transformation. For example, in the reaction (2.1), the left side of the arrow contains the reactants, and the right side of the arrow contains the products, which are A and B, respectively.

Equation 2.2 shows another chemical reaction of two different reactants A and B interacting with sufficient energy to create, in this process, a new product species C.

$$A + B \xrightarrow{k} C\,. \tag{2.2}$$

A fundamental aspect of the chemical reaction notation is that one must not confuse the "+" symbol between reactants as a sum: it is simply read as "and." So, Equation 2.2 is read as "A and B produces C."

The following reaction (2.3) shows an example of how to represent reactions that produce more than one product: in this case, reactant A creates two products: B, and C.

$$A \xrightarrow{k} B + C\,. \tag{2.3}$$

It is important to emphasize, as noted by (87), that chemical reactions must be distinguished from physical changes. Physical changes include changes of state, such as ice melting into water and water evaporating to vapor. If a physical change occurs, the physical properties of a substance will change, but its chemical identity will remain the same.

2.1.2 Stoichiometric Coefficients

Stoichiometric coefficients are numeric values that precedes species' names. For example, in reaction (2.4):

$$2A \xrightarrow{k} B\,. \tag{2.4}$$

the value "2" is the stoichiometric coefficient of A and "1" (implicit, omitted) is of B, which means that two molecules of A react to produce one molecule of B. These coefficients are used to balance the number of atoms of elements (molecules, for example) of reactants with the number of atoms of elements of the products (following the law of mass conservation (86)).

2.1.3 Reversible Reactions

It is possible to have a system of chemical reactions that work in the following way:

$$A + B \xrightarrow{k_1} C + D\,; \tag{2.5a}$$

$$C + D \xrightarrow{k_2} A + B\,. \tag{2.5b}$$

That means one reaction has products that are reactants of the other, and *vice-versa.* This system can be represented in a more compact way:

$$A\ +\ B \underset{k_2}{\overset{k_1}{\rightleftharpoons}} C\ +\ D\,. \tag{2.6}$$

Reaction (2.6) is called *reversible chemical reaction*, because A transforms into B, but B also can transforms into A.

2.1.4 Equilibrium Constant

Suppose a system that contains two reactions that interact competitively, such as reactions (2.5). In that case, they will share a moment in time that A will produce B at the same rate that B produces A (such as A and B are reactants and products of one reversible reaction, respectively). This means that the reversible reaction will reach equilibrium, i.e., the concentrations of both A and B will stay constant.

The relation between the concentration of products and reactants is defined by what is called "equilibrium constant", described by the following equation: $K = \dfrac{[B]}{[A]}$, where $[A]$ and $[B]$ are the concentrations of A and B at the equilibrium, respectively.

The equilibrium constant for the generic reaction $aA + bB \underset{k_2}{\overset{k_1}{\rightleftharpoons}} cC + dD$ can be expanded to:

$$K = \frac{[C]^c[D]^d}{[A]^a[B]^b}\,. \tag{2.7}$$

When $K > 1$ and in equilibrium, there is a more concentration of products than of the reactants. When $K < 1$, the concentration of reactants is more significant than that of the products. Finally, when $K = 1$, reactants and products can still have different concentrations, but the product of the reactants' concentrations is equal to the product of the concentrations of the reaction products. (51).

2.1.5 Concentration of Solutions

The mathematical model that expresses the behavior of chemical reactions is fully linked to the amount of substance in the system. So, it is fundamental to understand how to measure each amount in a system. First of all, a "system" is an environment in which the species (substances) are, and the reactions are occurring (17). Usually, substances are placed in a solution, i.e., a mixture of two or more substances, to make a chemical reaction occur (30). As the solution is the system, an amount (or quantity) of substance is expressed by its concentration, the relation between its amount and the solution volume.

The molar concentration (or molarity) is the amount of substance of the solute per volume of solution, and its symbol is mol/L (25). The work of (58) defines "mole" as:

> "The mole, symbol mol, is the SI unit of amount of substance. One mole contains exactly $6.02214076 \times 10^{23}$ elementary entities. This number is the fixed numerical value of the Avogadro constant, N_A, when expressed in mol^{-1}, and is called the Avogadro number. The amount of substance, symbol n, of a system is a measure of the number of specified elementary entities. An elementary entity may be an atom, a molecule, an ion, an electron, or any other particle or specified group of particles."

In other words: one mol from substance A is equivalent to $6.02214076 \times 10^{23}$ molecules of A. [1]

2.1.6 Chemical Reaction Rate

Usually, a chemical reaction comprises a sequence of elementary reactions that occur in a single step. This sequence of elementary reactions composes the mechanism, i.e., the "reaction path," of a chemical reaction. Because an elementary reaction does not involve the formation of an intermediate substance, the products must be formed directly from the reactants.

An elementary reaction's molecularity is the number of reactant molecules involved in the chemical reaction. Elementary reactions involving one, two, and three reactant molecules are called unimolecular, bimolecular, and termolecular reactions. Termolecular

[1] Another important fact is that the mole is used to define a relationship between the number of elements of a substance and its mass. For example: the ^{12}C carbon isotope has 12 grams per mol (12g on $6.02214076 \times 10^{23}$ atoms).

Table 2.1: Experimental data collected from an elementary hipotetical $A \longrightarrow B$ reaction. Time in hours and concentrations in mole (M). Adapted from (90).

Time (h)	$[A]$	$[B]$
0.0	1.000e-05	0.000e+00
0.5	6.016e-06	3.984e-06
1.0	3.652e-06	6.348e-06
1.5	2.236e-06	7.764e-06
2.0	1.368e-06	8.632e-06
2.5	8.352e-07	9.165e-06
3.0	5.096e-07	9.490e-06
3.5	3.111e-07	9.689e-06
4.0	1.900e-07	9.810e-06
4.5	1.160e-07	9.884e-06
5.0	7.083e-08	9.929e-06

reactions are infrequent, and reactions with higher molecularity have never been observed experimentally.

Regardless of the molecularity of the reaction, it needs time to occur. The speed at which the reaction needs to happen is known as the "rate law." The "global rate law" is an experimentally obtained rate for a chemical equation. It is used to propose a reaction mechanism consisting of a sequence of elementary unimolecular and bimolecular reactions. Combining these elementary reactions, we can arrive at the experimental rate law. Theoretically, building any complex reaction (or network of reactions) from elementary reactions is possible.

For example, the elementary chemical reaction $A \longrightarrow B$ expresses that A transforms into B during some time. However, that is not enough; the behavior of this transformation must be defined. "Chemical Kinetics", also known as reaction kinetics, is the branch of physical chemistry that is concerned with understanding the rates of chemical reactions (45).

In order to study the speed of the elementary $A \longrightarrow B$ reaction (also called the rate), in a practical situation, a chemist performs a lab experiment with this reaction. As the experiment proceeds, the concentrations of A and B (denoted by $[A]$ and $[B]$) are measured at different points in time. Suppose, for illustrative purposes, that the concentration measurements resulted in the values shown Table 2.1. All the values present in the Table 2.1 are ploted in graph at Figure 2.1. Because experimental work needs to measure and write down the concentration values, we need a compact and clear form to represent the reaction behavior at any time.

The behavior can be mathematically modeled based in law of mass-action. It affirms that the reaction rate of a chemical reaction in constant temperature is directly proportional to reactants concentrations (52, 19). Using this law, it is possible to define a mathematical expression to rate of reaction $A \xrightarrow{k} B$ as:

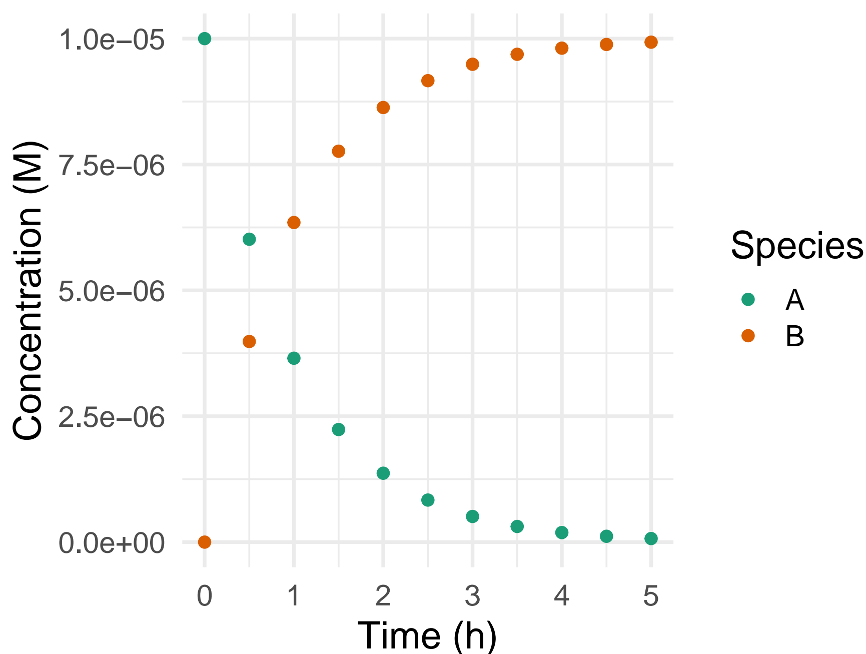


Figure 2.1: Graphical representation of data from Table 2.1. Adapted from (90)

$$r(t) = k[A]_t^n; \qquad k, n, \in \mathbb{R}; \qquad t \in \mathbb{N}; \tag{2.8}$$

where $r(t)$ expresses the time rate t and $[A]_t$ is the concentration of species A at time t. A more generic reaction $aA + bB + \ldots \xrightarrow{k}$ products can be expressed as:

$$r(t) = k \times [A]_t^n \times [B]_t^m \times \ldots; \qquad k, n, m \in \mathbb{R}; \qquad t \in \mathbb{N}; \tag{2.9}$$

The rate function of a reaction is called reaction kinetics. The kinetics of a reaction network is the association of a rate function for each reaction in the network (33).

Remark that k (also called the rate constant), used in the notation for reaction 2.1 (at the top of the arrow), is the same as used in the rate equation. This constant determines the proportionality between reactants and products. It is defined based on reaction's characteristics, such as temperature and reactants, for example (19).

The m and n coefficients define the order of the reaction. They express how much the concentration of A and B affect the reaction rate, respectively. It is essential to highlight that these values are not necessarily equal to the species' stoichiometric coefficients (a and b). In practice, these values are obtained experimentally. The reaction order is defined as the sum of the reactants' order, which is $m + n$. Since velocity is nothing more than the variation of species concentration over time, we can also rewrite the velocity function for the reaction $A \xrightarrow{k} B$ as being:

$$r(t) = -\frac{\Delta[A]_t}{\Delta t} = \frac{\Delta[B]_t}{\Delta t}, \tag{2.10}$$

where $\Delta[A]_t$ denotes the concentration variation of $[A]$ from time $t-1$ to t and $\Delta t = t-(t-1)$. We need to note that the reaction velocity is defined by the negative concentration of $[A]$, because $[A]$ is consumed in the reaction, whereas $[B]$ is being produced. With this is mind, we can merge equations 2.8 and 2.10 as:

$$-\frac{\Delta[A]_t}{\Delta t} = \frac{\Delta[B]_t}{\Delta t} = r(t) = k[A]_t^n \,. \tag{2.11}$$

An equation that expresses reaction rate in terms of the species' concentration variation is called "differential rate law" (19). Setting $\Delta t \to 0$, we have:

$$-\frac{d[A]_t}{dt} = \frac{d[B]_t}{dt} = k[A]_t^n \,. \tag{2.12}$$

The expression 2.12 can be rewriten as:

$$\begin{cases} \dfrac{d[A]}{dt} & = -k[A]_t^n \\ \dfrac{d[B]}{dt} & = k[A]_t^n \end{cases} . \tag{2.13}$$

The ordinary differential equations (ODE) system in 2.13 is one of many forms in expressing the behavior of one or many CRNs mathematically.

2.1.7 Chemical Reaction Network

A Chemical Reaction Network (CRN) is a set of coupled chemical reactions. For example, the CRN in equation 2.14 is composed of three bimolecular coupled chemical reactions (42). The chemical reactions in equations (2.14b) and (2.14c) are autocatalytic, since each one produces three copies of the reactants species X_0 and X_1, respectively.

$$X_0 + X_1 \xrightarrow{k_1} S_X; \tag{2.14a}$$

$$S_X + X_0 \xrightarrow{k_2} X_0 + X_0 + X_0; \tag{2.14b}$$

$$S_X + X_1 \xrightarrow{k_3} X_1 + X_1 + X_1. \tag{2.14c}$$

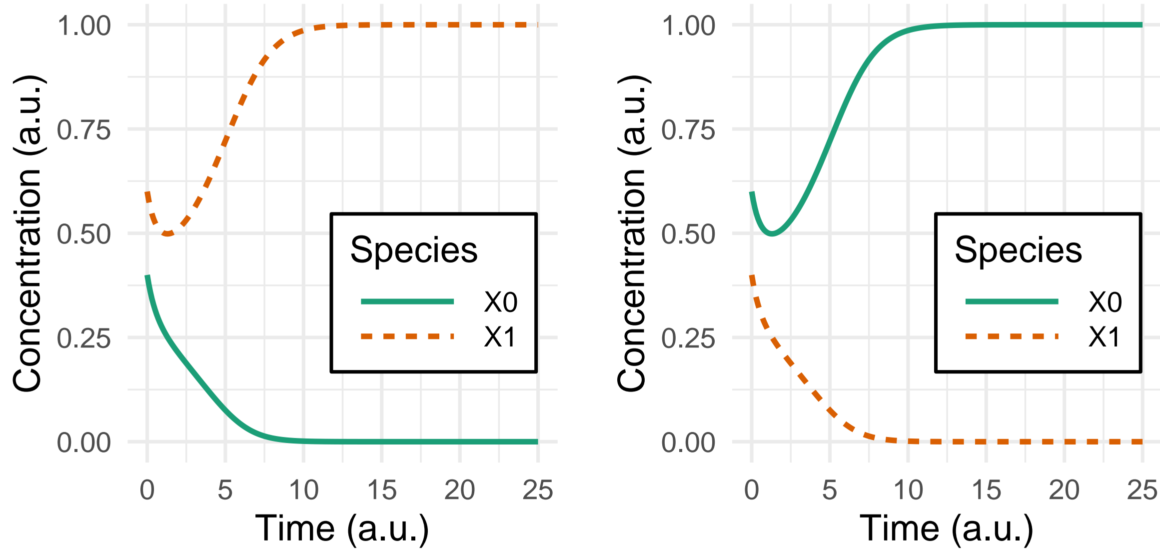
The mathematical model that describes the dynamical behavior of the species of the CRN in equation 2.14 is:

$$\frac{d[S_x]}{dt} = k_1[X_0][X_1] - k_2[S_x][X_0] - k_3[S_x][X_1]; \tag{2.15a}$$

$$\frac{d[X_0]}{dt} = -k_1[X_0][X_1] + 2k_2[S_x][X_0]; \tag{2.15b}$$

$$\frac{d[X_1]}{dt} = -k_1[X_0][X_1] + 2k_3[S_x][X_1]. \tag{2.15c}$$

Figure 2.2 shows the dynamic behavior of the species X_0 and X_1 obtained by computational simulation of the system of ordinary differential equations (2.15). The two graphics correspond to the simulations made with different initial conditions for the concentration of the species X_0 and X_1.



(a) Initial concentration of species X_1 is higher than X_0.

(b) Initial concentration of species X_0 is higher than X_1.

Figure 2.2: Simulation results for the equation 2.14. Both concentration and time axis are in arbitrary units.

Figure 2.2a shows the result of simulation when the initial concentration of $X_1 > X_0$, while Figure 2.2b shows the inverse case $X_1 < X_0$. Both the axis are in arbitrary units to show the behavior of the curves defined by its differential equations. In both cases, the concentrations of the species change in function of time until they reach the chemical equilibrium. Still, the species that have the higher initial concentration evolves to the chemical equilibrium value equal to 1, while the other one drops to zero.

2.2 DNA Strand Displacement Mechanism

This section will define some DNA concepts needed to understand its utility in implementing computation and how it is related to CRNs. Section 2.2.1 will introduce basic concepts about DNA, such as its basic blocks and how it is organized. After, at Section 2.2.2, introduces the DNA Strand Displacement (DSD) mechanism and give some examples of how it can be used to construct chemical circuits using DNA.

2.2.1 Desoxyribonucleic Acid

The Deoxyribonucleic acid (DNA), and Ribonucleic acid (RNA) are organic molecules present in all living organisms. The DNA primary purpose is to store information. Its structure is built around two strands. Each strand is composed of a sequence of nucleotides (3), that repeat as subunits. Each nucleotide is composed of three molecules: one of four nitrogenous bases available to use ((A)denine, (T)imine, (G)uanine, (C)ytosine), a sugar called deoxyribose, and a phosphate group. On RNA, (U)racile is used instead of (T)imine. These are built from chemical elements such as carbon, hydrogen, oxygen, phosphorus, and nitrogen, having slightly different structures.

One nucleotide is joined to another in a chain by covalent bonds between the phosphate of one and the sugar of the next. This structure results in an alternating phosphate-sugar polymer. The nitrogenous bases of both strands are bound together through "hydrogen bonds" of complementary Watson-Crick base pairing (A-T and C-G), which makes the double helix DNA shown in Figure 2.3.

2.2.2 Strand Displacement Mechanism

The DNA Strand Displacement (DSD) mechanism is an experimental technique to program the interaction of synthetic and small strands of DNA molecules (83). It is essential to note that all of this work is based on the use of synthetic DNA strands. Different graphical representations can be used to simplify the description of DNA. Examples of these are shown in Figure 2.4.

When a sequence of bases in synthetic DNA are grouped together, they form a *domain*. In addition, this sequence has a *codomain*, i.e., a complimentary domain that bounds to it. In Figure 2.4, domains are number named, and codomains are stared number named. The 5' (read "five-prime end") and 3'(read "three-prime end") at both ends of every strand of DNA are direction indicators of strands. In a DNA double helix, the strands run in opposite directions to permit base pairing between them.

The dynamic process starts when the toehold domain 1 of single-strand A hybridize (through complementary Watson-Crick base pairing) with the complementary toehold domain 1^* of strand G_i, following a branch migration and displacement of single-strand O_i, which in turn, interacts with double-strand T_i through complementary toehold 3^* to displace the single-strand B from the double strand T_i.

As a whole, the set of molecules in the Figure 2.5 forms a coupled DSD network,

Figure 2.3: The structure of DNA showing with detail the structure of the four bases, adenine, cytosine, guanine and thymine, and the location of the major and minor groove (11).

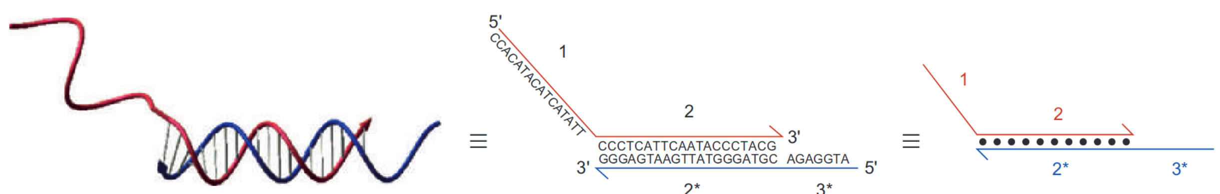


Figure 2.4: Example of three forms of representation of DNA molecules (104). The numbers denote domains; stared numbers denote complementary domains.

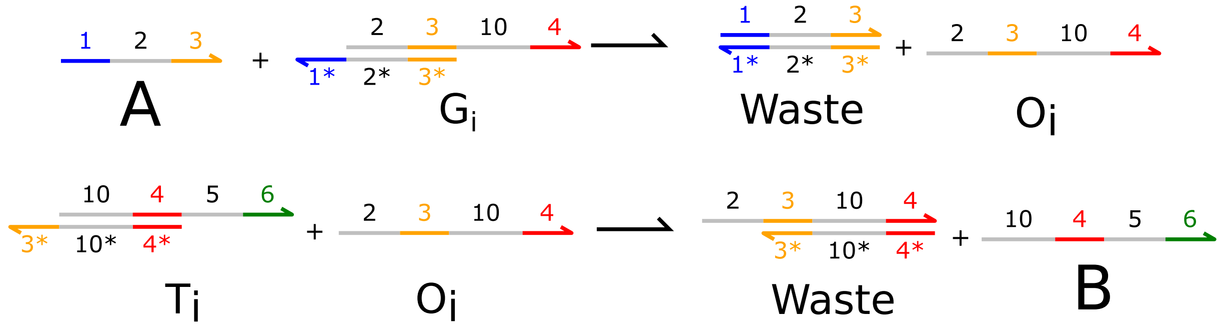


Figure 2.5: A DNA Strand Displacement (DSD) network (figure produced with (49)) with dynamic behavior analog to the unimolecular chemical reaction of the Equation 2.16.

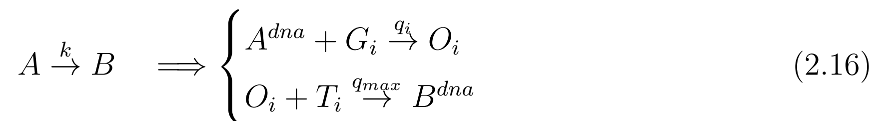
and its dynamic behavior can be modeled by the law of mass action.

It has been proposed that it is possible to design a coupled DSD network with dynamic behavior analog to the dynamics of elementary uni or bimolecular chemical reactions (83, 85). For example, the dynamic behavior of the arbitrary formal unimolecular chemical reaction, represented on the left side of the brace in Equation 2.16, can be approximated by the dynamic behavior of coupled DSD network described in the right side of the brace.

The A^{dna} and B^{dna} from the right side of Equation 2.16 are the single DNA strands A and B in Figure 2.5. These strands represent the species A and B in the arbitrary unimolecular chemical reaction from the left side of Equation 2.16. The DNA double strands G_i and T_i are the auxiliary DNA species gate and translator, respectively. The species O_i is a DNA single-strand that couples the two DSD processes.

To approximate the dynamic of the unimolecular chemical reaction, there are some constraints on the constants q_i, q_{max}, and on the concentration of auxiliary species that need to be met (83). The method also applies to emulate a generic biomolecular elementary chemical reactions using a proper design for the coupled DSD network.

Combining DSD networks that emulate the dynamic behaviors of uni and bimolecular elementary reactions, any arbitrary CRN can be physically simulated in the test tube (85).



$$A \xrightarrow{k} B \quad \Longrightarrow \begin{cases} A^{dna} + G_i \xrightarrow{q_i} O_i \\ O_i + T_i \xrightarrow{q_{max}} B^{dna} \end{cases} \tag{2.16}$$

The system of Ordinary Differential Equations (ODEs) that models the CRN $A \xrightarrow{k} B$ is described in Equation 2.13. The ODEs that model the DSD reactions present on the right side of the arrow in Equation 2.16 is described in Equation 2.17.

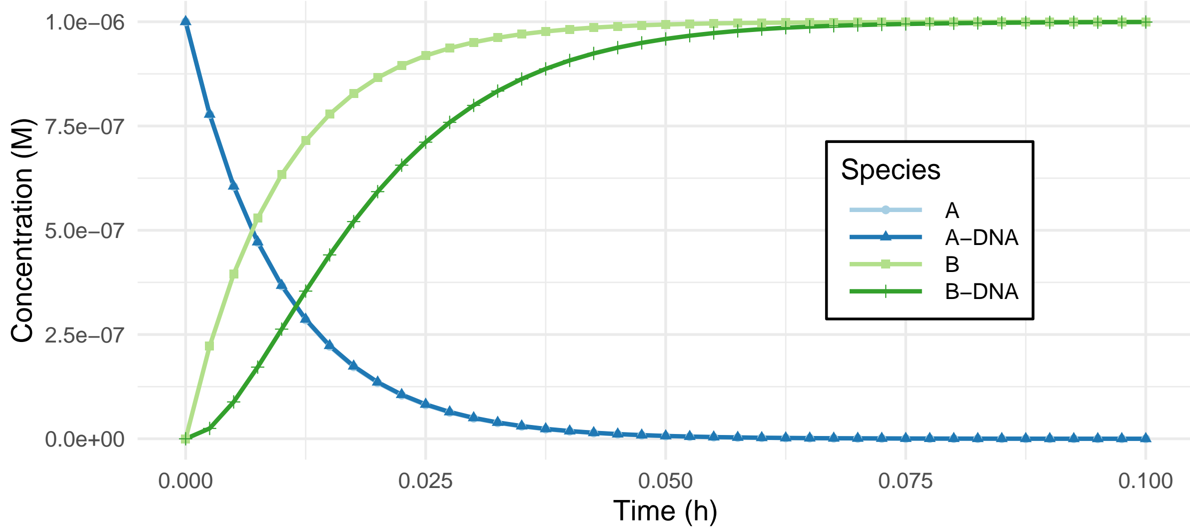


Figure 2.6: Behavior comparison between a formal unimolecular chemical reaction and its equivalent implementation in DSD mechanism shown in Equation 2.16.

$$\begin{cases} \dfrac{d[A]}{dt} &= -q_{\max}[A]_t[G_i]_t \\ \dfrac{d[B]}{dt} &= q_{\max}[O_i]_t[T_i]_t \\ \dfrac{d[G_i]}{dt} &= -q_{\max}[A]_t[G_i]_t \\ \dfrac{d[O_i]}{dt} &= q_{\max}[A]_t[G_i]_t - q_{\max}[O_i]_t[T_i]_t \\ \dfrac{d[T_i]}{dt} &= -q_{\max}[O_i]_t[T_i]_t \end{cases} . \tag{2.17}$$

Figure 2.6 shows that the behavior of DSD reactions approximate the behavior of the unimolecular chemical reaction present in Equation 2.16.

2.3 Machine Learning

Machine learning (ML) is the study of computer algorithms that can improve automatically through experience and by the use of data (60). It is seen as a part of artificial intelligence field. These algorithms use sample data to build a model to make predictions (or decisions) without being explicitly programmed by a human. There are various approaches to machine learning. Section 2.3.2.2 will cover the unsupervised learning algorithm K-Medoids used in this work. Before that, Sections 2.3.1 and 2.3.2, respectively, explain the differences between supervised and unsupervised learning algorithms.

2.3.1 Supervised Learning

Supervised learning is a category of ML algorithms that learn a function with the help of input-output examples (76). These examples are called *training data*. This training data consists of pairs of input and output values. A pair map the input to its desired output value. The supervised learning algorithm uses the training data to adjust (or "to learn", "to train") the function behavior. After "training", the function can now infer outputs to never previously seen inputs.

When considering a perfect scenario, the algorithm learns a function that can correctly determine outputs for unseen instances. This situation shows that the learning algorithm could generalize the training data enough to infer unseen input values correctly in a "reasonable" way.

Many supervised learning algorithms exist, but some issues are common in all of them, i.e., the necessity of providing two sets of data: the training set and the test set. Humans must gather real-world representative data (either from human experts or measurements). Then, the training set should contain some of the gathered data that will be used in the training phase of the learning algorithm. The remaining data is used as the test data to verify the quality of the trained function.

These two sets, if improperly chosen, can cause various degrees of problems, such as bias-variance tradeoff, not enough training data to learn the function complexity, redundancy, or high heterogeneity in the data.

2.3.2 Unsupervised Learning

Unlike the approach used by supervised learning algorithms, no previously labeled training set is provided to unsupervised learning algorithms (37). Thus, the algorithms must themselves group the data into categories with similar features. This is a significant advantage over supervised learning algorithms because it minimizes the human labor to prepare and review the training set. Furthermore, the algorithm has more freedom to identify and use previously undetected features in the data that human experts may not have discerned.

This advantage comes with a cost. Unsupervised learning algorithms require more data in the training set. Also, they take more time to converge the learning function and require greater computing power and memory storage to accomplish the initial investigative process. In addition, in some situations, the unsupervised learning algorithm can see

artifacts or anomalies in the training data as irrelevant or assigned undue importance. These artifacts and anomalies, in some cases, are even erroneous when seen by a human, but the algorithm may show greater sensibility of them.

Unsupervised learning algorithms can be organized into four major categories: clustering, anomaly detection, neural networks, and latent variable models (98, 43). The most known algorithms are in the clustering category. Some examples include hierarchical clustering (98), K-Means, K-Medoids (43), mixture models, DBSCAN (29) and OPTICS algorithm.

This work is focused in designing a chemical reaction network that implements the K-Medoids algorithm. Because K-Medoids is an variation of K-Means algorithm, in Section 2.3.2.1 we will present how K-Means algorithm works. After, in Section 2.3.2.2, we show the behavior of K-Medoids algorithm.

2.3.2.1 K-Means

K-means is one of the most popular clustering algorithms (98). It aims to partition data points into clusters, in which each data point belongs to the cluster with the nearest calculated mean, named *centroid*.

The clustering problem which both heuristics try to resolve is NP-hard, i.e., it is computationally difficult (40, 4); nevertheless, a local optimum can be obtained quickly by efficient heuristic algorithms. Unfortunately, although helpful, the k-means clustering algorithm is not very robust to noise and outliers. It generally relies on minimizing squared euclidean distances between points and uses centroids that are not necessarily present in the input points set.

The standard k-means algorithm, also known as “naive k-means”, is described in Algorithm 1. It contains two steps: the assignment step and the update step. In the assignment step, each point is assigned to the cluster of nearest mean. In the update step, the means (centroids) from each cluster are recalculated to match the possible updated clusters in previous step (54). The notation $S_i^{(t)}$ present in Equation 2.18 refers to nearest cluster set S_i in iteration t. Accordingly, $m_i^{(t+1)}$ in Equation 2.19 refers to mean (centroid) m_i to be used in next iteration $t+1$.

Algorithm 1: K-means algorithm.

Input: $X = \{x_0, x_1, ..., x_{n-1}\}, n \in \mathbb{N}$ data points and $K \in \mathbb{N}$ cluster sets.

Initialization: Given a initial array M of K means $M = [m_0, m_1, m_2, ..., m_{K-1}]$ for K-partitions (randomly chosen or based on some prior knowledge);

Assignment step: Assign each point $x_p \in X$ to the nearest cluster set S_i:

$$S_i^{(t)} = \left\{ x_p : \left\| x_p - m_i^{(t)} \right\|^2 \leq \left\| x_p - m_j^{(t)} \right\|^2 \mid \forall j, 0 \leq j \leq K-1 \right\} \tag{2.18}$$

for $i, j \in \{0, 1, 2, ..., K-1\}$ and $t \in \mathbb{N}$ iterations.

Update step: Recalculate means (centroids) for each cluster:

$$m_i^{(t+1)} = \frac{1}{\left|S_i^{(t)}\right|} \sum_{x_j \in S_i^{(t)}} x_j \tag{2.19}$$

Stop condition: Repeat both steps until the assignments do not change.

2.3.2.2 K-Medoids

K-Medoids main difference to K-Means resides in that it does not use calculated centroids as the representative of a cluster. Instead, it chooses actual data points as centers; they are called "medoids". Another significant difference is that K-Medoids can use arbitrary dissimilarity criteria instead of Euclidean distances. This change in measuring criteria causes the algorithm to minimize a sum of pairwise dissimilarities instead of a sum of squared Euclidean distances, making it more robust to noise and outliers than K-Means (41).

K-Medoids is a partitioning algorithm (such as K-Means) that groups n points into K clusters by similarity. The problem of clustering is NP-hard, i.e., grouping data by similarity is difficult to solve exactly (40, 4). The Partitioning Around Medoids (PAM, also known as K-Medoids) algorithm (43) was the first heuristic proposed to solve the partitioning problem. Other heuristics based on PAM were subsequently proposed, with various degrees of optimization in speed and quality of results, such as CLARA, CLARANS, and FastCLARA (43, 79).

The PAM algorithm uses a greedy search heuristic. This heuristic may not find the optimum solution but is fast and can find some good local optima. The PAM algorithm works in two distinct phases: the BUILD phase and the SWAP phase. These phases are shown in Algorithms 2 and 3, respectively.

In BUILD phase, an initial clustering is found by selecting representative points

Algorithm 2: Partition Around Medoids BUILD phase algorithm (Adaptated from (43)).

1. Initialize S with a point for which the sum of the dissimilarities to all other points is minimal;
2. Consider a point $x_i \in U$ as a candidate for including into the set S;
3. For a point $x_j \in U - \{x_i\}$, compute D_j;
4. If $D_j > d(x_i, x_j)$ then point x_j will contribute to the decision of selecting point x_i:

$$C_{ji} = \max\big(D_j - d(x_j, x_i), 0\big) \tag{2.20}$$

5. Compute total gain obtained by adding x_i to S:

$$g_i = \sum_{x_j \in U} C_{ji} \tag{2.21}$$

6. Choose point x_i that maximizes g_i, i.e.: $S = S \cup \{x_i\}$ and $U = U - \{x_i\}$;
7. Repeat all steps from step 2 until K points have been selected;

successively until K points are found. The first point is the one for which the sum of the dissimilarities to all other points is as small as possible. This point is the most centrally located in the set of points. After that, at each step, another point is picked. This point is the one that minimizes the objective function as much as possible (43).

To explain both phases in detail, some assumptions are needed. As our convention, data points are placed in set X. Points that are also medoids are placed into a set S of selected points. Non-medoid points are placed in set U of unselected points. So, $U = X - S$. For each point $x_p \in X$ we need to maintain two values: D_p and E_p. The D_p is the dissimilarity value between x_p and the closest point in S, i.e., the first nearest medoid. The E_p is the dissimilarity value between x_p and the second closest point in S, i.e., the second nearest medoid. So, $D_p \leq E_p$ and $x_p \in S$ if and only if $D_p = 0$. Also, both D_p and E_p must be updated every time when the sets S and U change. With these assumptions, the BUILD phase is described in Algorithm 2

The SWAP phase is where the heart of the PAM algorithm resides. In this phase, the heuristic tries to improve the set of representative points and, consequently, improve the clustering generated by this set. First, the heuristic considers all pairs of points (x_i, x_h) for $x_i \in S$ and $x_h \in U$. Then, we must determine what effect is obtained on the clustering value when a swap is carried out, i.e., when the point x_i is no longer a medoid but point x_h is. Mathematically, this is equivalent to transferring x_i from S to U and x_h from U to S. It is important to highlight that the clustering value is defined as the sum of dissimilarities between each point and the most similar representative point.

Deciding swap x_i and x_h involves computing the swap total contribution T_{ih}. This total contribution needs to consider each contribution K_{jih} of each point $x_j \in U - \{x_h\}$ to the swap of x_i and x_h. It is important to highlight that we can have only $d(x_j, x_i) > D_j$ or

Algorithm 3: Partition Around Medoids SWAP phase algorithm (Adaptated from (43)).

```
1  repeat
2      Iterate ⟵ false;
3      // For each non-medoid
4      for x_h ∈ U do
5          // For each medoid
6          for x_i ∈ S do
7              // For each non-medoid minus x_h
8              for x_j ∈ U − {x_h} do
9                  // D_j is the dissimilarity between x_j and the first closest medoid
10                 if d(x_j, x_i) > D_j then
11                     if d(x_j, x_h) ≥ D_j then
12                         K_jih ⟵ 0;
13                     else
14                         K_jih ⟵ d(x_j, x_h) − D_j;
15                 else if d(x_j, x_i) = D_j then
16                     // E_j is the dissimilarity between x_j and the second closest
                        medoid
17                     if d(x_j, x_h) < E_j then
18                         K_jih ⟵ d(x_j, x_h) − D_j;
19                         // K_jih can be either positive or negative
20                     else
21                         K_jih ⟵ E_j − D_j;
22                         // In this sub-case, K_jih > 0.
23             // Compute total swap contribution
24             T_ih = ∑ {K_jih | j ∈ U};
25     Select a pair (x_i, x_h) ∈ S × U that minimizes T_ih;
26     if T_ih < 0 then
27         S ⟵ (S − x_i) ∪ x_h;
28         Iterate ⟵ true;
29 until Iterate = false;
30 Exit, because the objective cannot be decreased anymore;
31 //This happens when all values of T_ih are positive and it is the halting condition of
    the algorithm
```

$d(x_j, x_i) = D_j$. In other words: or x_j belongs to a cluster more distant to x_i or x_j belongs to the cluster of x_i. Algorithm 3 shows how to calulate the effect of a swap between x_i and x_h, i.e., when $x_i \in U$ and $x_h \in S$.

After the total contribution value T_{ih} is obtained, the algorithm decides if the swap can be carried on. If $T_{ih} < 0$ the swap is made and another iteration begins. Otherwise, the algorithm halts. An example of k-medoids running is shown in Figure 2.7.

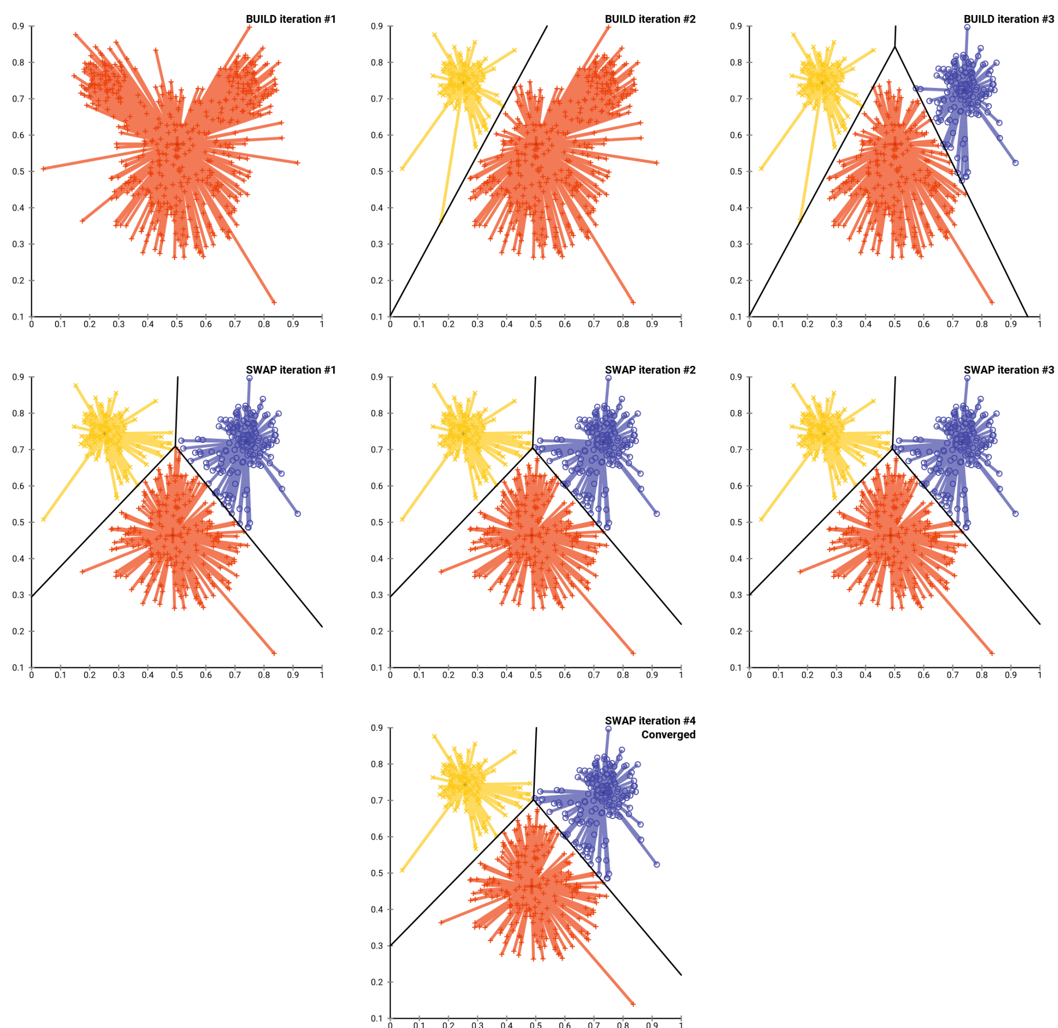
In the next Chapter, we will show some related works that builds upon the base concepts of DNA Computing and Machine Learning applied in the context of Chemical Reaction Networks.

2.4 DNA Computing Design Automation Tools

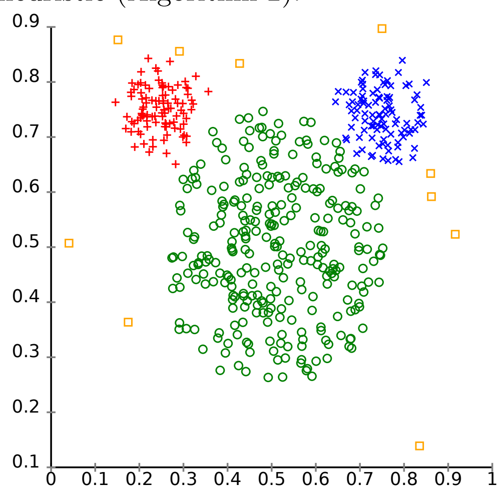
Designing and simulating DNA Strand Displacement (DSD) circuits is complex and prone to errors. Depending on what level of abstraction the DSD circuit design starts, it can be top-down or bottom-up. Top-down designs usually start from the higher levels of abstraction, such as generic Chemical Reaction Networks (CRNs) describing the dynamic behavior of a DSD chemical reaction without the specific details of DSD encoding (such as 4-Domain scheme and others (83, 85)) and work towards the compilation to DNA strands in lower levels of abstraction, such as domain level and molecular level. Bottom-up designs work the other way around, starting the design at the domain level and going up to more abstract levels. In both ways, the development cycle of new circuits and designs needs many person-hours of work. Also, due to the complexity, many errors could be introduced due to manual fatigue and mistakes. Because of these difficulties, various tools were and are being developed to address those issues, making the design of DSD circuits faster, easier, and more correct.

Figure 2.8 illustrates that we can organize the workflow development of DSD circuits in four principal steps, top-down, i.e., from the more abstract to the more specific: (I) the generic CRN design; (II) the CRN-to-DSD compilation; (III) the DSD simulation and analysis; and (IV) the DSD compilation to nucleotides. For each step, a particular software tool facilitates developing circuits in DNA. It is essential to highlight that it is possible to simulate the behavior of the molecular circuit in the design steps, so step (III) is a verification of the (I), (II), and (IV) steps.

The first step is to express the desired behavior for the DNA circuit, comprising the generic CRN design. Copasi (39) and Surface CRN (24) tools help model, simulate and visualize the dynamical behavior of biochemical systems from the description of generic CRNs (5, 53).



(a) Convergence of k-medoids clustering on an artificial dataset from an starting position chosen by the BUILD heuristic (Algorithm 2).



(b) Groundtruth clustering from "mouse" data set. The distant orange squares are outliers. Adaptated from (23).

Figure 2.7: Four SWAP iterations of K-medoids algorithm.

In the second step, we must convert (or compile) such CRN into DSD systems. Thus, we must map each species of the CRN to DNA domains. These domains form small fragments to allow the DSD technique's implementation. Nuskell (7) software focuses on converting generic CRN into various schemes (also called encodings) of DSD reactions. Then, the dynamic behavior of these DSD reactions is simulated and analyzed accordingly.

Also, in this step, the DNAr (89) is a software package developed for R language that permits the analysis and simulation of molecular circuits at the abstraction level of CRN. Similarly to Nuskell, the DNAr package can convert generic CRNs into DSD reactions and simulate them. Additionally, this package provides various quality-of-life features to its users, such as offering functions to visualize and compare simulation results from both abstraction levels (CRN and DSD), export the DSD reactions to Visual DSD (49), and promote developers to build new extensions on top of DNAr to generate and simulate large-scale complex CRNs. DNAr simulate a CRN with data received as a set of vectors. These vectors are organized in a data-oriented design, i.e., each vector contains only one type of data, such as species names, concentrations, reactions, rates, or others. Each value is associated with its index in vectors. For example, in Algorithm 4, we see how to invoke DNAr and simulate a simple unimolecular CRN.

The species names and their initial concentrations are defined on lines 5 and 6. Note that each line initializes the `species` and `ci` parameters with lists. Each position in those lists refers to one species. In other words, `species[0]` (value `'A'`) and `ci[0]` (value `1e-5`) are associated by their index 0. The same happens with `reactions[0]` (value `'A -> B'`) and `ki[0]` (value `5e-5`), which are the reaction and its rate. The last parameter `t` is a list containing the sequence of ten time points from 0 to 72000 used to simulate and plot the

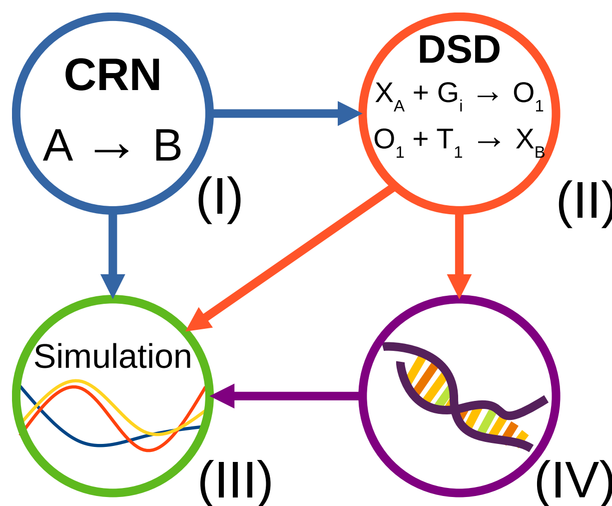


Figure 2.8: Design flow of DNA Strand Displacement (DSD) circuits. Step (I) is the design of generic Chemical Reaction Network (CRN); Step (II) is the compilation to a specific encoding of DSD CRN; Step (III) is the behavioral simulation of the reactions; Step (IV) is the syntesis of DNA strands.

Algorithm 4: DNAr code to simulate the unimolecular reaction $A \to B$.

```
library(DNAr)  # Loading the library

# Simulating
b <- react(
    species   = c('A', 'B'),
    ci        = c(1e-5, 0),
    reactions = c('A -> B'),
    ki        = c(5e-5),
    t         = seq(0, 72000, length.out = 10)  # Using 10 time points
)
```

graph.

DNAr, as in this previous example, provides the conversion from this data to ODEs and solves them to generate a graph showing the species' concentrations varying over time.

When building more complex CRNs, managing these lists is entirely manual and cumbersome. It leads to various problems, such as laboriousness from creating unique species' names to incorrectly positioning the data according to their respective indices in lists. This tribulation led to the necessity to create a tool that could lift this responsibility from the shoulders of designers and abstract it in a way that allowed them to focus on what matters: designing the circuits.

DNAr-Logic (57, 56) and DNAr-Analog (68) are two extensions built on top of DNAr that completely abstract the chemical and biological nature of DSD circuits. They solve this problem by exposing a simple Application Programming Interface (API) to developers to construct their complex circuits programmatically without worrying about these nuisances. The package's responsibility is to correctly name and reference all the data from the species' names, initial concentrations, reactions, and rates without distracting the users from their real focus: iterating through designing, testing, and validating their molecular circuits. They will be described in more details on Chapter 4.

The third step is to study the circuit at a lower level of abstraction, i.e., DSD systems. In this step, we can analyze the DSD reactions from various perspectives. The Multistrand (78) software provides a more detailed perspective. It is built on top of NUPACK (101), and can simulate nucleic acid kinetic interactions on a thermodynamic energy model. It also supports multiple interacting strands and various usage modes to study kinetic trajectories. In simpler terms, it operates on nucleic acid domain interactions. From a broader perspective, the KinDA (10) software framework simulates the kinetic behavior of DSD reactions on the domain level with nucleotide sequences assigned to each domain. It uses Multistrand and NUPACK software to simulate the DSD reactions and obtain results.

Similarly to KinDA, there is the Microsoft Visual DSD (49) software. The tool

operates in the nucleic acid domain level of abstraction. It defines a domain-specific programming language for programming nucleic acid circuits. This language automatically computes all possible reactions between the species, facilitating a reaction-based approach and eliminating the complication of manually constructing the reaction network. It can analyze and simulate the circuits using various methods, including stochastic simulation, approximate simulation of stochastic dynamics using moment closure techniques, integration of the chemical master equation, and satisfiability analysis for stable systems.

Finally, at the fourth and last step, the design of DNA circuits achieves the highest level of detail. At this point, it is necessary to define all nucleotides of the DNA strands involved and the specific characteristics of the solution. The NUPACK (101) is a software suite for analyzing and designing nucleic acid structures, devices, and systems. It provides a simulation on a very low-level abstraction. It deals with complex test tube ensembles containing arbitrary numbers of interacting strand species, providing meaningful tools to analyze and design the DNA intermolecular interactions. The tool provides algorithms that work on two elemental class problems: sequence analysis and sequence design. The first one analyzes the equilibrium base-pairing properties of a given set of DNA (or RNA) strands over a specified ensemble; the second one is the design of the sequences of a set of DNA or RNA strands over a specified ensemble given a set of desired equilibrium base-pairing properties. Likewise, the Piperine (85) specifies all characteristics of the candidate DNA strands for experimental implementation in the laboratory. It is an automatic and complete tool. It offers support from the specification of CRN to the definition of nucleotides of DNA molecules.

Although this is a comprehensive list of the most widely known tools, this list is far from complete. Due to the necessity of automation, verification, scalability, and efficiency in boot-starting this new technology, new tools are constantly being developed and tested by various researchers around the globe.

Chapter 3

Related works and Historical Background

The purpose of this chapter is to make a brief, but comprehensible, literature review of the most important works about computing systems made of biological components, such as DNA. Section 3.1 presents some review of the previous works done to build computing systems out of biological components. Section 3.2 will cover works done to bring machine learning to Chemical Reaction Networks (CRNs).

3.1 DNA Computing

This section will present some works that use DNA as a biological compound capable of computing. Other biological compounds, such as proteins and a-like, will be mentioned but are not the focus of this work. From this chapter onwards, we will use the terms “species” and “signal” synonymously to refer to the (single or double) strands of DNA present in an aqueous solution. Also, we need to define that the terms *in silico*, *in vitro*, and *in vivo* refer to the medium where the algorithms run. The *in silico* algorithms run implemented as a conventional program inside a conventional electronic computer. The *in vitro* algorithms are implemented as a Chemical Reaction Network (CRN) that react chemically in a laboratory but can be simulated in a conventional computer. The *in vivo* algorithms run as a drug inside living organisms but can also be simulated in a conventional computer.

The idea of computing using biological compounds is not a novel thing. Many types of substances, such as proteins, appear to process and transfer information instead of being used as metabolic intermediates or when building cellular structures. Instead, they are used in biochemical “circuits” that perform simple computational tasks such as amplification, integration, and information storage (15). Conceptually, any protein transforming an input signal into an output signal could operate as a computational or

information-carrying element. Proteins can interact with chemical constructs, such as DNA and RNA; proteins can also respond to light, temperature, mechanical forces, and voltage, all of these useful as biochemical circuits inputs (15).

Due to proteins and ribonucleic acid (DNA and RNA) present in cells, some authors declare that cells are the smallest computer in nature: they can be seen as molecular machines that process inputs through state transitions and generate outputs (77). However, the DNA and RNA inside cells were recognized as the repositories for computational programs. Because of that, early biomolecular computer research focused on human-scale electronic computers operated by humans to solve these complex problems of cells (77). More recent works focus on a different strategy: using nucleic acids such as DNA and RNA to implement the computation itself, i.e., molecular computers that allow input and output to be molecules, producing a native biomolecular system to control biological processes. For example, a molecular computer identifying and analyzing Messenger RNA (mRNA) of disease-related genes and producing a single-stranded DNA molecule modeled to work as an anticancer drug when needed (9).

Soloveichik presented one of seminal works that bootstrapped the use of nucleic acids to implement the dynamic behavior of any kind of formal chemical reactions (83). Their work showed how to construct systems of DNA molecules to closely approximate the dynamic behavior of arbitrary systems of coupled chemical reactions. Furthermore, by leveraging strand displacement reactions, they showed it is possible to compile any system of chemical equations into real chemical systems that use DNA as a universal substrate to do Turing-universal computation.

Further developing the idea of using DNA strand displacement to make computation, Quian developed a simple DNA gate that can be used in synthesizing large-scale molecular circuits (71). Using DNA strand displacement cascades, they develop DNA circuit design techniques that scale up easily, allowing the systematic creation of large molecular circuits. These large circuits are possible because the gates amplify small signals and incorporate a threshold to clean up noise and erroneous signals present along the circuit. Another characteristic of the gate is its modular design. It allows the systematic automated design of large circuits by a compiler that converts a high-level description of a circuit into a molecular implementation at the level of DNA sequences.

Although not directly related to DNA, Jiang projected basic logic gates such as AND, OR, NOR, XOR using a system of chemical reaction networks that can be implemented in DNA (42), for example, using the 4-Domain schema (83). By project, these circuits amplify weak signals and incorporate a threshold to clean up noise and erroneous signals present in the circuits. Also, the gates were used to create larger circuits, such as a square-root unit, a binary adder, and a linear feedback shift register that worked without errors.

The extensive survey brought up by Boruah showed that various solutions designed

using DNA could solve some NP-Complete problems — such as Travelling Salesman Problem (93), Boolean Satisfiability Problem, 3-Vertex Coloring Problem, and Vertex Cover Problem, for example — using the intrinsically massively parallel (20) characteristic of molecular computers (14, 96).

A more recent and extensive review of the literature compiled by Fan showed that DNA could be applied in different kinds of computation other than digital logic computation, such as multivalued logic and fuzzy logic, and also be used as "logic building blocks" that can construct new materials (32). Beyond computation, DNA has also been investigated as a very dense information storage device (91, 74).

Using DNA as a molecular "wet hardware" to implement computation is a feasible and efficient way to solve problems of combinatorial nature (22) and to interface with problems of biological nature natively (65).

3.2 CRN Machine Learning

The machine learning field applies several algorithms to teach the computer to identify similarities in the analyzed data. The learning process involves identifying similarities within data through a significant amount of calculations, which try to minimize the error to make predictions or categorize previously unseen new data.

Because these calculations can be parallelizable, the massively parallel nature of molecular reactions can help in this aspect. So, different scientific groups worldwide started to attack this problem, implementing machine learning algorithms using molecular reactions.

Qian (73) implemented a neural network algorithm using its previously designed DNA gate (71). They showed that the developed architecture allows experimental scale-up of multilayer digital circuits, transforming linear threshold circuits into DNA Strand Displacement (DSD) cascades that function as small neural networks. Also, their approach allowed the implementation of a working Hopfield associative memory. Although the memory was trained *in silico*, it remembered four single-stranded DNA patterns, recalling the most similar one even with an DNA pattern with missing or wrong bases.

Other work by Lakin proposed a biochemical circuit for learning linear functions (46). They used DNAzymes, which are DNA oligonucleotides that are capable of performing a specific chemical reaction, often but not always catalytic (16). The DNAzymes were used to catalyze the chemical reactions that implement the learning circuit designed. Their architecture achieved a novel mechanism to maintain and modify its internal state. They performed simulations to demonstrate that the circuit can learn and deal satisfactorily

with performance, scalability, and robustness to noise.

In work by Noh, they implemented molecular learning with DNA kernel machines (66). Their algorithm interprets the hybridization between DNA molecules as computing the inner product between embedded vectors in a corresponding vector space. The algorithm performs the learning of a binary classifier in this vector space. In addition to the simulation, they implemented the algorithm in vitro using various DNA operations, including hybridization, denaturing, Polymerase Chain Reaction (PCR), and gel electrophoresis. Similarly, Zhang presented learning based on support vector machines (102). Their approach consisted of doing the training *in silico* with a *in vitro* DSD implementation of a winner-takes-all algorithm.

In another work, Xiong used a full *in vitro* training and classification implementation of convolutional neural network (97). They also discovered that a cyclic physicochemical process (freeze/thaw cycling) could be a potent driver for strand displacement. When freezing, the steep increase in DNA concentrations could trigger the generation of a subpopulation of active complexes; during thawing, the steep drop in DNA concentrations could disassemble active and unproductive complexes.

Lakin, later, designed a supervised learning algorithm using adaptive DNA Strand Displacement (DSD) networks (48). By developing an adaptive amplifier new gate, they presented a circuit architecture that is generally applicable to the implementation of adaptive algorithms. The work includes a strand displacement learning circuit, a prediction subcircuit and a feedback circuit. The simulation of the DNA learning was validated with a reference implementation.

Although not a molecular reaction design, Lee used K-means algorithm in silico together with social network analysis theorem to effectively calculate the representative DNA strand of a group of single DNA strands (50). The hybridization tendency of this representative strand was empirically tested and confirmed to be the higher among the others strands, revealing a symbolic sequence representing multiple DNA strands.

Another approach to learning similarities between data is using the Reservoir Computing (RC) paradigm, where the authors implemented DNA molecular memristors to recognize the digits from the MNIST dataset (53). The molecular memristors were implemented in Chemical Reaction Network (CRN), and their behavior was mathematically validated through an Ordinary Differential Equation (ODE) model.

Table 3.1 shows a summary of recent works related to machine learning implemented partially or fully in Chemical Reaction Network (CRN) or DSD.

Following, in Chapter 4, we will introduce the first contribution of this work: the Computer Aided Design (CAD) software tool that facilitates the development of molecular circuits. After, in Chapter 5, we will present the second contribution: the design of a CRN that implements the K-Medoids unsupervised learning algorithm. This implementation, however, differs from previously mentioned works in two key aspects: the clustering

algorithm can be compiled to any chemical reactions that can implement the CRN. Also, it can be used to perform clustering of DNA strands by similarity *in vitro* without previously knowing the sequences of DNA strands.

Table 3.1: Table showing the comparison between the Machine Learning algorithms present in the literature and our proposal.

Work	**Technology**	**Type**	**Algorithm**	**In-silico**	**In-vitro**
(73)	DSD	Supervised	Neural-Network	Training	Classify
(46)	DNAzymes	Supervised	Neural-Network	—	Train and Classify
(66)	DSD	Supervised	SVM[1]	—	Train and Classify
(48)	DSD	Supervised	Neural-Network	—	Train and Classify
(102)	DSD	Supervised	SVM[1]	Training	Classify
(97)	DSD	Supervised	Neural-Network	—	Train and Classify
(53)	CRN	Supervised	Reservoir Computing	—	Train and Classify
(50)	DSD	Unsupervised	K-Means	Clustering	Classify
Our work	DSD	Unsupervised	K-Medoids	—	Cluster and Classify

[1]Support Vector Machine

Chapter 4

Design of Digital and Analog DNA Systems

This Chapter will present part of the contributions developed by this work, such as the Constructive Methodology for Molecular Circuits using the DNAr-Logic software package. This package handles the specific implementation details of molecular circuits for the designer, abstracting the tiresome work of designing the circuits at the CRN level. The DNAr-Logic software package to the R Language allows modularization and programmatically assists the development of digital molecular circuits modeled as generic CRNs (57, 56). The last Section 4.2 will present the contribution to the development of DNAr-Analog, although it was not the focus of this work.

4.1 DNAr-Logic

Although the chemical systems presented in previous Chapters are usually in the field of analogical signal processing, it is possible to achieve digital logic behavior with an appropriate set of coupled chemical reactions.

Analog molecular circuits have some advantages, such as chemical species concentrations that vary continuously over time are inherently analog signals that better interface with biological samples in a laboratory. Nevertheless, digital molecular circuits have the advantage of allowing the implementation of logical constructs and, with them, making decisions. Contrary to analog molecular circuits, the number of species and reactions needed to implement these digital logic circuits explode as soon as the circuits increase in size and complexity.

DNAr-Logic is a software package designed to programmatically abstract, simulate and test new molecular digital logic circuits in DNA Strand Displacement (DSD) (57, 56). It is built atop DNAr (89). Both DNAr and DNAr-Logic leverage the power of R: an open-source and easy-to-use programming language used by the scientific community.

DNAr-Logic abstracts the need to write Chemical Reaction Networks (CRNs) to create digital logic circuits based on DSD technology. This package implements all the basic logic gates needed to build more complex circuits. The circuits can be designed constructively and simulated without requiring chemistry or DSD mechanism knowledge.

4.1.1 Methodology

Our package implements — using some previously CRN proposed digital logic gates (42) and DNAr simulation capabilities — the molecular reactions in *complementary representation* of digital single-bit signals, combinational circuits — Majority, (N)AND, (N)OR and XOR logic gates — and sequential circuits — Latch-D and Flipflop-D (57). Table 4.1 contains all the gates implemented in DNAr-Logic with their specific resource (species and reactions) usage.

Table 4.1 lists all the available gates. The gates identified by an asterisk were proposed by other authors, while the gates without an asterisk are contributions from this work. In the Table 4.1, the columns "Species" and "Reactions" count the respective unique species and reactions each logic element uses. For example, a "signal" element in the first row represents a "wire" in a digital logic circuit. Due to all signals representing bits in dual-rail (*complementary representation*), there is the need to use three species for each signal: one species to represent the "`zero`" value, another species to represent the "`one`" value, and a third species to represent the intent to "switch" values (from zero to one and vice-versa). The number of reactions needed to implement the correct behavior of a bistable digital signal is three. A bistable digital signal is a signal that can only have two stable levels: the "`zero`" level and the "`one`" level; any other possible signal levels, such as 50% of "`zero`" and 50% of "one," are unstable and the slight imbalance tending one of those two levels make the signal "switch" to the tended stable level.

The columns "Inputs" and "Outputs" of Table 4.1 count the number of signal inputs and outputs each element have in its interface. For example, the "AND" element has two signals as inputs and one as output. In other words, the "AND" element uses $3 \times 3 = 9$ species and $3 \times 3 = 9$ reactions only to encode its input and output signals. Next, the columns "Total Species" encode all the species needed for the element to operate, which, for the "AND" element, is $9 + 1 = 10$ species and $9 + 5 = 14$ reactions.

Another specific example is the "NOT" element. This element does not use any species or reactions because an invert operation is "free" in the dual-rail (*complementary representation*) encoding: as each signal uses two independent species to represent the "`zero`" and "`one`" values, just by using the complement signal species the designer get the

Table 4.1: List of digital circuit elements implemented in DNAr-Logic. Specifically, the NOT gate does not use species nor reactions because an invert operation is "free" in the dual-rail (*complementary representation*) encoding. Previously proposed the gates are signaled by * (42).

Element	Species	Reactions	Inputs	Outputs	Total Species	Total Reactions
Signals*	3	3	0	0	3	3
AND*	1	5	2	1	10	14
OR*	1	5	2	1	10	14
XOR*	2	8	2	1	11	17
Latch-D*	2	6	2	1	11	15
FF-D*	4	22	2	1	13	31
Majority	2	10	3	1	14	22
NAND	1	5	2	1	10	14
NOR	1	5	2	1	10	14
NOT	0	0	0	0	0	0
FF-JK	0	0	3	1	43	55

"inverted" value for free.

The general design concept of DNAr-Logic package is explained in Section 4.1.2.

4.1.2 Design

The DNAr-Logic software package implements digital logic in the context of DNA Strand Displacement (DSD). To do so, it encodes bits in *complementary representation*, a synonym to *dual-rail representation* — a term often used in the electrical and computer engineering fields.

The dual-rail encoding encodes the binary values "`zero`" and "`one`" using two different signals that are complementary and mutually exclusive. Here, we define this representation by using two different DNA species to represent the values "`zero`" and "`one`" of a signal. When the species of the "`zero`" value is present in the solution, the species of the "`one`" value is absent, and vice-versa.

The DNAr-Logic software package defines three concepts: signals, gates, and circuits. Signals are binary values (in *complementary representation*) used by gates. Gates are structures that receive one or more input signals, process them, and generate one or more output signals. Circuits contain timing metadata used in simulations, all the signals, and connected gates that process information. One circuit can be part of another larger circuit, allowing hierarchical composition of designs. At low-level, these high-level

concepts are data structures that encapsulate the information needed by DNAr — DNA species' names, reactions (with their rates), and species' concentrations — to run the simulations. By exposing a simple API to developers to construct their complex circuits programmatically, the package's assumes responsibility to correctly name and reference all the data from the species' names, initial concentrations, reactions, and rates making users focus only on the development itself: iterating through designing, testing, and validating their molecular circuits.

4.1.3 Examples, Simulations and Results

The following Sections will show five examples of molecular circuits programmatically designed using DNAr-Logic's features in growing order of complexity. The objective is to show the tool's scalability with complex circuits. Section 4.1.3.1 shows the building of a half-adder using the elementary molecular logic gates present in DNAr-Logic. Section 4.1.3.2 presents the implementation of a majority gate in DNAr-Logic. Then, Section 4.1.3.3 shows a full-adder built entirely of these majority gates. Section 4.1.3.4 instructs how to build a J-K flip-flop using DNAr-Logic. Last but not least, Section 4.1.3.5 will show the construction of a 4-bit Carry-Look-Ahead Adder (by majority gates).

4.1.3.1 Half-Adder

A half-adder is the simplest logic circuit designed to produce the result of the sum of two bits. It consists of two logic gates — an AND gate and an XOR (eXclusive OR) gate — to calculate the results of carry-out and sum, respectively. Figure 4.1 show the schematic design of the half-adder circuit. At the left part of the Figure, "`A`" and "`B`" are the two input signals, each one carrying one bit of information. The top gate is the XOR gate that receives "`A`" and "`B`" as inputs and generates the "`Sum`" signal as output. The output of the XOR gate is "1" when both inputs have different values and "0" otherwise. The bottom gate is the AND gate that also receives "`A`" and "`B`" as inputs and generate the "`Cout`" (carry-out) signal as output. The carry-out signal contains the value "1" only when both inputs contain the value "1" and "0" otherwise.

Algorithm 5 contains a code example of how to construct and simulate a circuit of a half-adder, seen in Figure 4.1, using the DNAr-Logic package. At line 2, the variables `A`

and B are the binary input values of the circuit. At lines 3 and 4, an AND gate and an XOR gate are created with names "_Cout" and "_Sum", respectively. The output of AND gate is the "carry-out" value; the output of XOR gate is the "Sum" value. Both gates receive binary value '1' in input signals. In line 5, the circuit is created and configured to simulate from 0 to 15000 seconds in 200 units intervals. Lines 6 and 7 show both gates being added to the circuit. In line 8, a list is created containing the species names of the output signals from both gates. At last, in line 12, the plot_circuit_4domain function simulate the circuit using the DSD 4-Domain coding (83) and plot the concentrations only of the specified species.

Algorithm 5: Code example in R using DNAr-Logic to build an half-adder circuit.

```
1  library(DNArLogic)
2  A  <- 1; B <- 1
3  andGate <- make_and_gate('_Cout', A, B)
4  xorGate <- make_xor_gate('_Sum', A, B)
5  circuit <- make_circuit(seq(0, 15000, 200))
6  circuit <- circuit_add_gate(circuit, andGate)
7  circuit <- circuit_add_gate(circuit, xorGate)
8  speciesList <- c(andGate$species$output$value0,
9                   andGate$species$output$value1,
10                  xorGate$species$output$value0,
11                  xorGate$species$output$value1)
12 plot_circuit_4domain(circuit, speciesList)
```

Figure 4.2 presents the simulation result of the referred circuit using the 4-Domain DSD coding (83). Because signals are coded in complementary representation, a signal value is represented by two species: one species for zero value and another for one value. The signals "_Cout_AND_out_0" and "_Cout_AND_out_1" change concentration as the actuation of the AND gate happens. The concentration of species "_Cout_AND_out_0" goes to zero while "_Cout_AND_out_1" goes to one. Thus, this represents that the AND gate switched its output value from zero to one, because both of its inputs have logic value one.

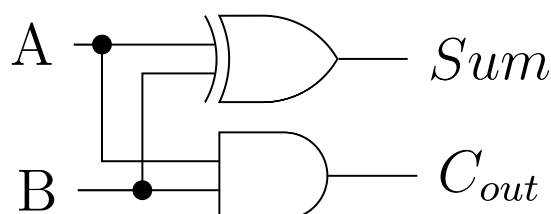


Figure 4.1: Circuit for a half-adder containing an XOR gate generating the Sum signal and AND gate generating the Carry-Out signal.

The signals "`_Sum_XOR_out_0`" and "`_Sum_XOR_out_1`" does not change concentration as the time passes. This behavior happens because both values of inputs `A` and `B` are 1, which does not trigger the change of the output logic value of the XOR gate. As the XOR gate only outputs 1 when both inputs are different (0 and 1 or vice-versa), there is no change, and the output value of the XOR gate remains zero.

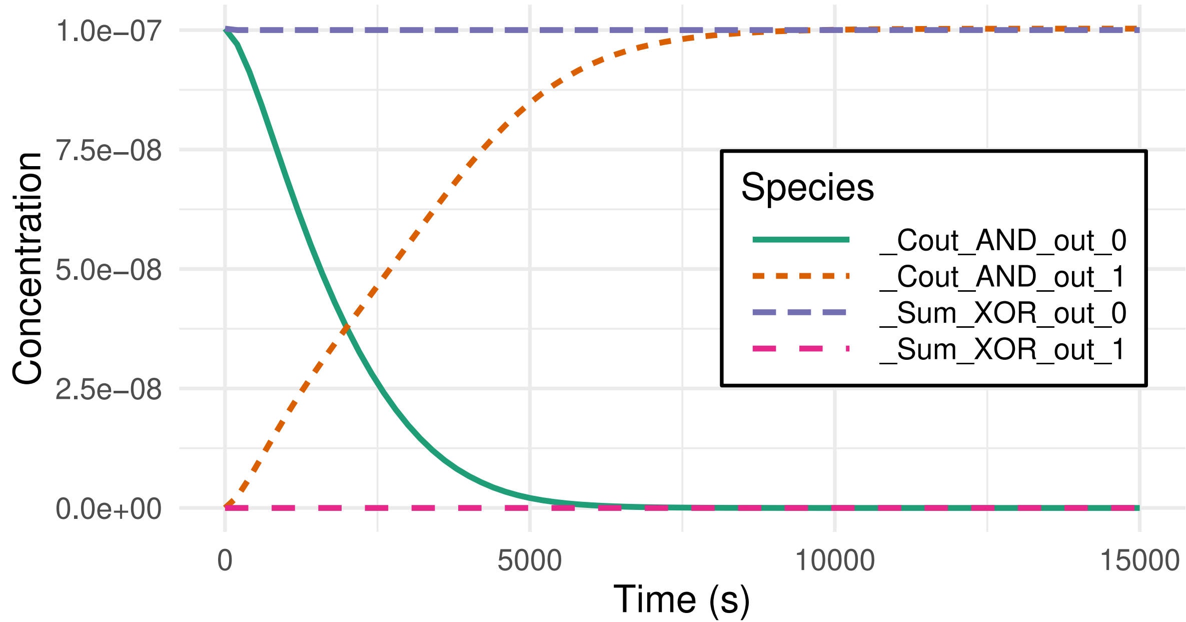


Figure 4.2: Simulation result of code from Algorithm 5; The respective concentrations of output signals Carry-out and Sum are shown (*complementary representation*).

4.1.3.2 Logic Majority Gate

The Majority Gate is a three-input logic gate that implements the majority function: a device that outputs a HIGH signal (TRUE logic value) when the majority of its signal inputs are HIGH, otherwise it outputs a LOW signal (FALSE logic value), i.e. a majority gate returns true if and only if more than 50% of its inputs are true.

Using the same convention of (42), here we define the molecular reactions for a majority gate. Suppose the inputs of the majority gate are A, B and C, and the output is Z. These signals are represented by the concentrations of A_0/A_1, B_0/B_1, C_0/C_1 and Z_0/Z_1, respectively.

By definition, $Z = 1$ when two of the three inputs are 1 (thus the "majority" of the inputs). Which means that when both A_1 and B_1, or A_1 and C_1, or B_1 and C_1 are present, Z_1 should be generated and Z_0 should be cleared out of solution. Similarly, $Z = 0$

when two of the three inputs are 0. This is implemented by the reactions:

$$A_1 + B_1 \longrightarrow A_1 + B_1 + Z_1' \tag{4.1a}$$

$$A_1 + C_1 \longrightarrow A_1 + C_1 + Z_1' \tag{4.1b}$$

$$B_1 + C_1 \longrightarrow B_1 + C_1 + Z_1' \tag{4.1c}$$

$$A_0 + B_0 \longrightarrow A_0 + B_0 + Z_0' \tag{4.1d}$$

$$B_0 + C_0 \longrightarrow B_0 + C_0 + Z_0' \tag{4.1e}$$

$$A_0 + C_0 \longrightarrow A_0 + C_0 + Z_0' \tag{4.1f}$$

In the first reaction, A_1 combines with B_1 to generate Z_1', an indicator that the value of Z should be set to 1. The concentrations of A_1 and B_1 do not change. Similarly, in the fourth reaction, A_0 combines with B_0 to generate Z_0', an indicator that the value of Z should be set to 0. All the reactions in (4.1) behave in the same manner.

The molecules Z_0' and Z_1' are transferred to an external sink denoted by $\varnothing$. These could be waste types whose concentration are not tracked. This is implemented by the reactions:

$$\begin{aligned} 2Z_0' &\longrightarrow \varnothing \\ 2Z_1' &\longrightarrow \varnothing \end{aligned} \tag{4.2}$$

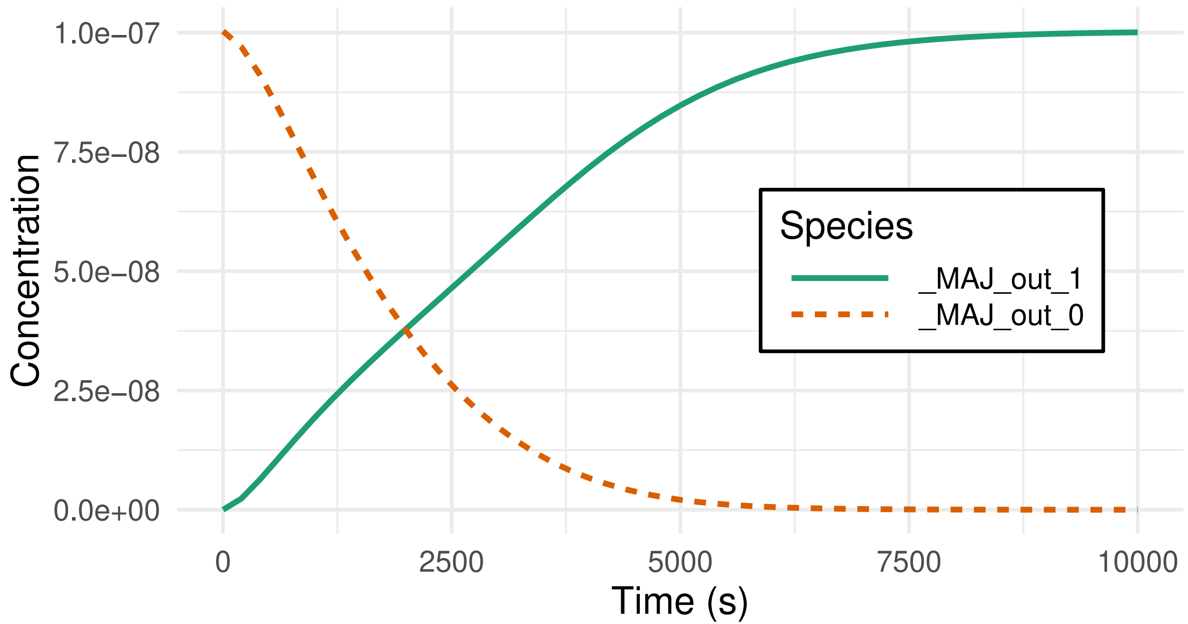
When two input molecules are both present, these reactions maintain the concentration of Z_1' (or, analogously, Z_0') at an equilibrium level. When there is only one — A_1 or B_1 or C_1 — input molecule present, the molecule Z_1' get cleared out. The same occurs when there is only one — A_0 or B_0 or C_0 — input molecule present, the molecule Z_0' get cleared out.

When the concentration of Z_1' is at an equilibrium level, Z_1' switches the molecules Z_0 to Z_1. Similarly, when the concentration of Z_0' is at an equilibrium level, Z_0' switches the molecules Z_1 to Z_0. This behavior is implemented by the reactions:

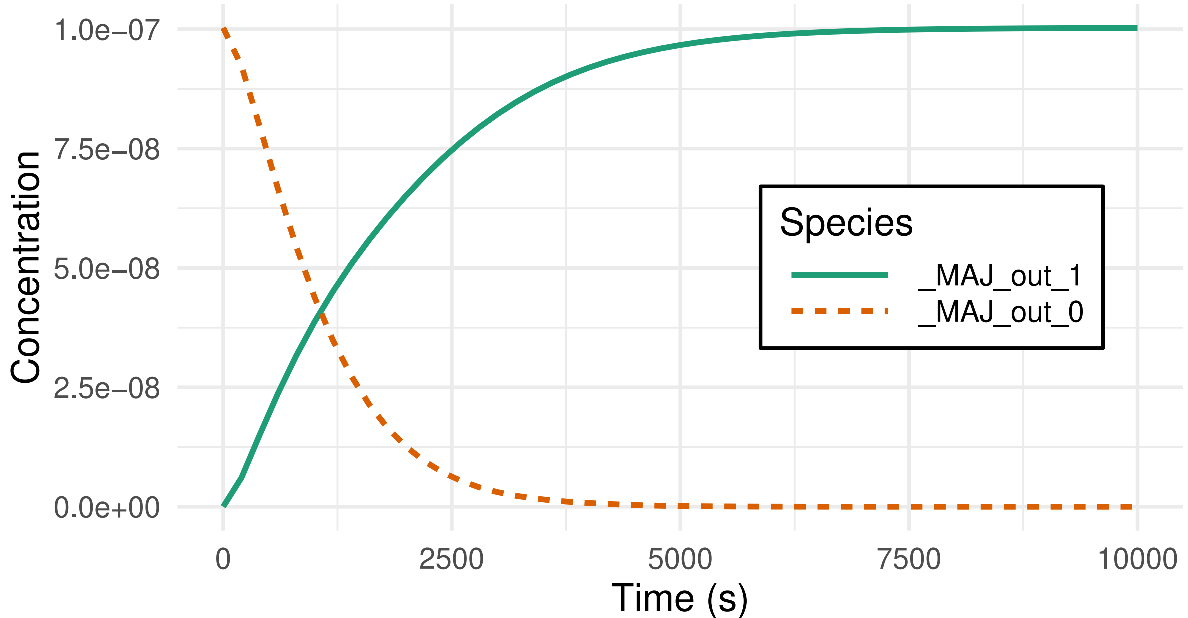
$$\begin{aligned} Z_1' + Z_0 &\longrightarrow Z_1 \\ Z_0' + Z_1 &\longrightarrow Z_0 \end{aligned} \tag{4.3}$$

Taken together, reactions (4.1), (4.2) and (4.3) implement a 3-to-1 majority gate. Figure 4.3 contains the simulation result using the 4-Domain DSD coding (83) from the majority gate defined in Section 4.1.3.2.

For the sake of similarly redudant results, only two simulation graphs are shown. However, simulations were run with all the possible combinations of input values, and all of them display the expected correct behavior. When two of the three inputs from a majority gate have TRUE values, the generated output is also TRUE (Figure 4.3a). When all the three inputs have TRUE values, the same occurs, but less time is needed to the output signal to change (Figure 4.3b). This time difference is due to the greater production of Z_1'



(a) Output species concentrations for any two inputs of value 1.



(b) Output species concentrations for all three inputs of value 1.

Figure 4.3: Simulation result of majority gate; The respective concentrations of outputs signals are shown (*complementary representation*).

molecule by all the three Reactions (4.1a), (4.1b) and (4.1c). Consequently, the change in the output signal is faster when this occurs. Besides this difference in reaction time, the behavior of the designed majority gate is correct and fully functional.

Algorithm 6 shows how the majority gate is implemented in DNAr-Logic. Lines 3 to 13 contain the setup code needed to create a new generic gate with three inputs and one output. Lines 21 to 32 contain the designed reactions from Equations 4.1, 4.2, and 4.3.

Algorithm 6: Code example of DNAr-Logic implementation of majority gate.

```
1  make_majority_gate <- function(name, input1value, input2value, input3value) {
2    # Generate species for 3 input signals and 1 output signal
3    gate <- make_generic3to1_gate(name, 'MAJ', input1value, input2value, input3value)
4
5    # Append 2 specific majority gate species
6    gate$species   <- append(gate$species, c(outTo1 = jn(name, '_MAJ_out_to_1'),
7                                             outTo0 = jn(name, '_MAJ_out_to_0')))
8
9    # Append 2 specific majority gate species' concentrations
10   gate$ci        <- append(gate$ci, list(0, 0))
11
12   # Auxiliary variables to hold species names
13   A1 <- gate$species$input1$value1; A0 <- gate$species$input1$value0
14   B1 <- gate$species$input2$value1; B0 <- gate$species$input2$value0
15   C1 <- gate$species$input3$value1; C0 <- gate$species$input3$value0
16   Z1 <- gate$species$output$value1; Z0 <- gate$species$output$value0
17   Z_1 <- gate$species$outTo1; Z_0 <- gate$species$outTo0
18
19   # Append the majority gate reactions
20   gate$reactions <- append(gate$reactions, list(
21     jn(A1, ' + ', B1, ' -> ', A1, ' + ', B1, ' + ', Z_1 ),
22     jn(A1, ' + ', C1, ' -> ', A1, ' + ', C1, ' + ', Z_1 ),
23     jn(B1, ' + ', C1, ' -> ', B1, ' + ', C1, ' + ', Z_1 ),
24     jn(A0, ' + ', B0, ' -> ', A0, ' + ', B0, ' + ', Z_0 ),
25     jn(B0, ' + ', C0, ' -> ', B0, ' + ', C0, ' + ', Z_0 ),
26     jn(A0, ' + ', C0, ' -> ', A0, ' + ', C0, ' + ', Z_0 ),
27
28     jn('2', Z_0 , ' -> 0'),
29     jn('2', Z_1 , ' -> 0'),
30
31     jn(Z_1, ' + ', Z0, ' -> ', Z1),
32     jn(Z_0, ' + ', Z1, ' -> ', Z0)
33   ))
34
35   # Append the majority gate reactions' rates
36   gate$ki <- append(gate$ki, list(1E-2,1E-2,1E-2,1E-2,1E-2,1E-2,1E-2,1E-2,1E-2,1E-2))
37
38   # Return the newly built majority gate
39   return(gate)
40 }
```

4.1.3.3 Full Adder with Majority Gates

The circuit schema of a full-adder using majority gates can be seen in Figure 4.4. Algorithm 7 shows how to implement using the features present in the DNAr-Logic. Line 1 contains the declaration of variables `A`, `B`, and `Cin`, all initialized with logic value one. All of them will be used as the binary input values of the circuit. In lines 2, 3, and 4, three instances of majority gates are created with names "`_Cout`," "`_Sum`," and "`_aux`," respectively. The third input of the "`_aux`" majority gate is the complement from input signal `Cin`. The first and last inputs from the "`_Sum`" majority gate are set to zero. The reason is that these inputs will receive the intermediate results generated by the other two majority gates. So, their "default" value is zero to not interfere with future values generated by the previously mentioned majority gates. Later in code (lines 9 and 12),

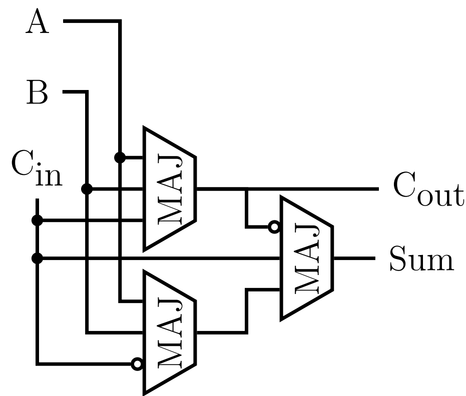


Figure 4.4: Full-adder circuit schema with only Majority Gates.

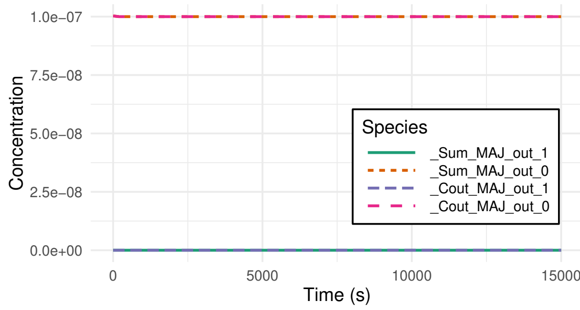
these inputs will be linked to the outputs from gates "`_Cout`" and "`_aux`," respectively. In line 5, the circuit is created and configured to simulate from 0 to 15000 seconds in 200-unit intervals. Lines 6, 7, and 8 show all gates being added to the circuit. In line 15, a list is created containing the species names of the output signals from both "`_Cout`" and "`_Sum`" gates. The `plot_circuit_4domain` function simulates the circuit using the DSD 4-Domain coding (83) and plots the concentrations only of the specified species.

Due to current limitations in DNAr, it is not possible to manually alter concentrations of DNA species after the simulation has begun, i.e., the initial concentrations must be defined before the simulation starts. Thus, we run a single simulation for each set of inputs.

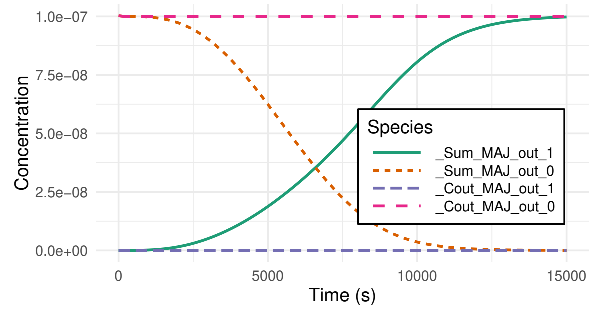
Algorithm 7: Code example to build an full-adder circuit using majority gates.

```
1  A <- 1; B <- 1; Cin <- 1
2  cout <- make_majority_gate('_Cout', A, B, Cin)
3  sum  <- make_majority_gate('_Sum', 0, Cin, 0)
4  aux  <- make_majority_gate('_aux', A, B, (1 - Cin))
5  circuit <- make_circuit(seq(0, 15000, 200))
6  circuit <- circuit_add_gate(circuit, cout)
7  circuit <- circuit_add_gate(circuit, sum)
8  circuit <- circuit_add_gate(circuit, aux)
9  circuit <- circuit_link_gate_signals_not(circuit,
10                              cout$species$output,
11                              sum$species$input1)
12 circuit <- circuit_link_gate_signals(circuit,
13                           aux$species$output,
14                           sum$species$input3)
15 species <- c(sum$species$output$value1,
16              sum$species$output$value0,
17              cout$species$output$value1,
18              cout$species$output$value0)
19 plot_circuit_4domain(circuit, species)
```

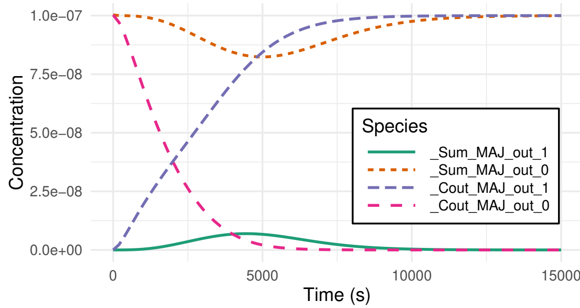
Figure 4.5 shows the simulation result using the 4-Domain DSD coding (83) for the full adder defined before. Again, for the sake of limited space, only four simulation results are shown. However, simulations were run with all the possible combinations of input values and all of them show the expected behavior of a full-adder circuit.



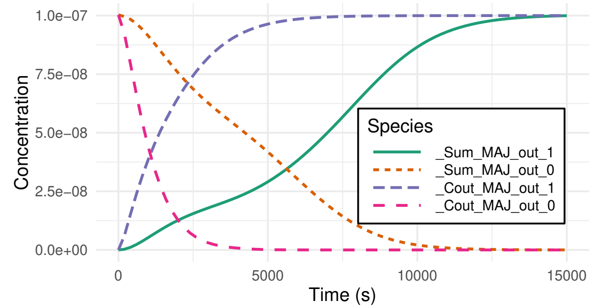
(a) Output concentrations when all inputs have value 0.



(b) Output concentrations when only input A has value 1.



(c) Output concentrations when inputs A and B have value 1.



(d) Output concentrations when all inputs have value 1.

Figure 4.5: Simulation result of a full-adder using majority gates; The respective concentrations of output signals, carry-out (Cout) and Sum, are shown in *complementary representation*.

First, Figure 4.5a shows the outputs Sum and Carry-out as zero because the three inputs (A, B and C) have value zero. Next, Figure 4.5b shows output signal Sum becoming value one and Carry-out staying at zero because only input A has value one. Next, in Figure 4.5c, two of three inputs (A and B) have value one and only the output signal Carry-Out goes to value one, besides the oscillation of Sum output signal. This oscillation is due to the latency of the propagation of signals in the chemical circuit. The Carry-out output signal has only one majority gate in its critical path to produce the result. The Sum output signal, however, has two majority gates in its critical path. One of them produces the Carry-out signal that influences the calculation of the next majority gate. Finally, Figure 4.5d shows the simulation result when all the inputs (A, B and C) have value one. Both concentrations of output signals Sum and Carry-out change as the signals propagates through the chemical circuit and stabilizes at the correct values of the sum.

4.1.3.4 JK-Flip-Flop

A Flip-Flop is a sequential circuit, i.e., it is a memory element that holds a binary value. It can be used in biological applications as a memory that retains information of presence (or absence) of a DNA strand in the solution, for example. By design, the Flip-Flops can store an arbitrary value received from its input. This Flip-Flop type is known as “Data” or “D-Flip-Flop”. Another kind is the “T-Flip-Flop”, or the “Toggle” type. It is the one capable of changing its internal state from 0 to 1 and vice-versa. The “JK-Flip-Flop” can operate as either type D or type T, just based on the value of its J and K input signals. Figure 4.6 shows how to build a Flip-Flop of JK and T types from a D type Flip-Flop, respectively drawn in dark-red, orange and blue.

In Algorithm 8 shows how to implement a JK-Flip-Flop using only DNAr-Logic’s high-level functions. Again, the writing of low-level CRN is not needed. As the JK-Flip-Flop is an entity used in other circuits, its circuit does not need the timing information, as seen in line 3. So, in lines 3 to 6, the JK-Flip-Flop circuit is created and its interface is defined with three inputs and one output. The lines 9 to 12 show the creation of the three logic gates and one D-Flip-Flop used to make a JK-Flip-Flop. The code in lines 15 to 19 insert the instantiated elements into the circuit. The lines 22 to 32 shows the interconnections of signals between the circuit elements. Finally, line 35 shows the compilation of the circuit, i.e., the species and reactions from all the circuit elements are combined into one circuit.

The Flip-Flops are sequential circuits that need a periodic clock signal to synchronize when they change their internal memory state. The methods to implement proper “clock” signals in CRNs are beyond the scope of this work. An assortment of molecular oscillators

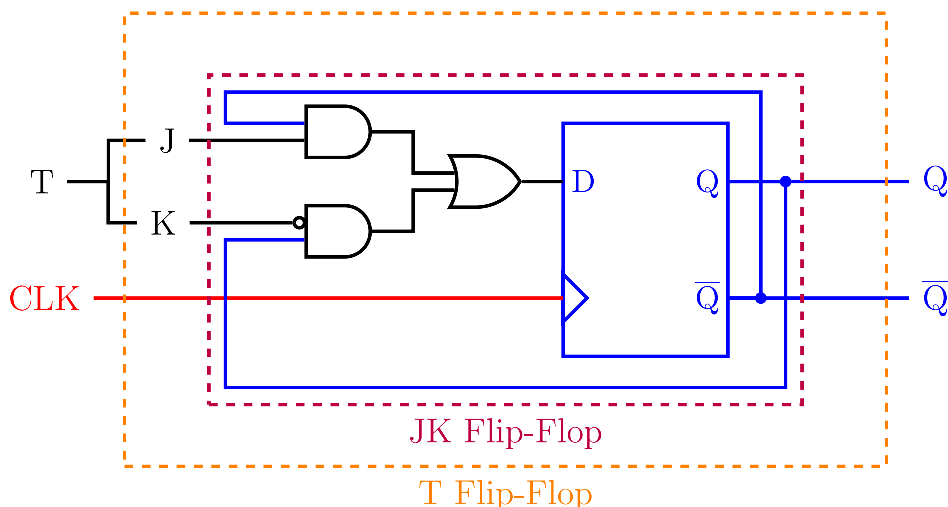


Figure 4.6: Circuit to build a JK-Flip-Flop and a T-Flip-Flop from D-Flip-Flop.

Algorithm 8: Code example to build an JK-Flip-Flop circuit using basic logic gates and a D-Flip-Flop.

```
1  make_jkff <- function(name, jValue, kValue, eValue) {
2    # JK-Flip-Flop circuit interface
3    entity <- make_circuit(0)
4    interface <- make_generic3to1_gate(name, 'JKFF', jValue, kValue, eValue)
5    entity$inputs <- list(interface$species$input1, interface$species$input2,
     ↪  interface$species$input3)
6    entity$output <- c(interface$species$output)
7
8    # Instantiate the sub-entities
9    jand <- make_and_gate(jn(name, '_jand'), 0, 0)
10   kand <- make_and_gate(jn(name, '_kand'), 0, 0)
11   jkor <- make_or_gate(jn(name, '_jkor'), 0, 0)
12   ffd  <- make_flipflopd(jn(name, '_bit'), 0, 0)
13
14   # Insert sub-entities into the circuit
15   entity <- circuit_insert_gate(entity, interface)
16   entity <- circuit_insert_gate(entity, jand)
17   entity <- circuit_insert_gate(entity, kand)
18   entity <- circuit_insert_gate(entity, jkor)
19   entity <- circuit_insert_gate(entity, ffd)
20
21   # Insert interconnections into the circuit
22   entity <- circuit_insert_gate(entity, make_link_gate(interface$species$input1,
     ↪  jand$species$input2))
23   entity <- circuit_insert_gate(entity, make_link_gate(interface$species$input2,
     ↪  kand$species$input1, negated = TRUE))
24   entity <- circuit_insert_gate(entity, make_link_gate(interface$species$input3,
     ↪  ffd$inputs[[2]]))
25
26   entity <- circuit_insert_gate(entity, make_link_gate(jand$species$output,
     ↪  jkor$species$input1))
27   entity <- circuit_insert_gate(entity, make_link_gate(kand$species$output,
     ↪  jkor$species$input2))
28   entity <- circuit_insert_gate(entity, make_link_gate(jkor$species$output,
     ↪  ffd$inputs[[1]]))
29
30   entity <- circuit_insert_gate(entity, make_link_gate(ffd$output,
     ↪  interface$species$output))
31   entity <- circuit_insert_gate(entity, make_link_gate(ffd$output, jand$species$input1,
     ↪  negated = TRUE))
32   entity <- circuit_insert_gate(entity, make_link_gate(ffd$output, kand$species$input2))
33
34   # Combine all the sub-entities in the final entity
35   entity <- circuit_compile(entity)
36
37   return(entity)
38 }
```

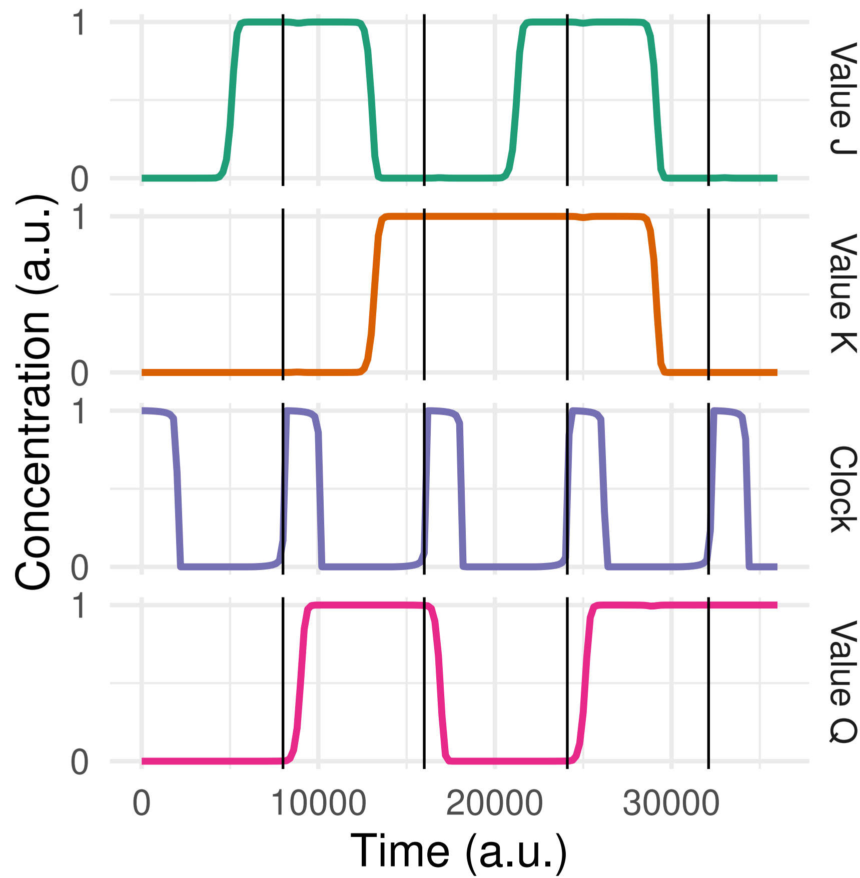


Figure 4.7: Simulation results of the JK-Flip-Flop. Each vertical black bar, from left to right, indicates the moment when the JK-Flip-Flop (output value Q) is being set (J=1, K=0), reset (J=0, K=1), toggled (J=1, K=1) and in hold state (J=0, K=0), respectively. The axis are shown in arbitrary units (a.u.).

has been proposed in the literature that can be used in this context (105, 94, 35).

Figure 4.7 shows the simulation result from the code showed in Algorithm 8. Simulations were run with all the possible combinations of input values (and all of them presented the expected behavior). For the sake of limited space, only one simulation graph that contains all the possible input combinations are shown. Figure 4.7 presents groups of four vertical black bars intersecting the time axis on the four line graphs. These vertical lines mark the clock rising edge. At each rising edge of the clock, the signals J and K have a different value. From left to right, respectively, on the first bar shows that the JK-Flip-Flop is being set (J=1, K=0), i.e., output signal Q changes from 0 to 1; in the second bar, it is being reset (J=0, K=1), i.e., output signal Q changes from 1 to 0; the third bar, it is being toggled (J=1, K=1), i.e., output signal Q changes from 0 to 1; and the fourth bar, the JK-Flip-Flop is put in hold state (J=0, K=0), i.e., output signal Q does not change.

4.1.3.5 4-bit Adder with Carry Look-ahead

An n-bit Adder circuit is a prevalent combinational logic circuit capable of adding two binary numbers with n bits each. It uses n 1-bit full adders to sum each bit. Each full adder receives three inputs A, B, and C_{in}. The inputs A and B are single bits coming from two binary numbers. The input C_{in} is the previous sum's carry-out bit. After adding A, B, and C_{in} together, the resulting sum is placed in the respective S output bit. In Figure 4.8, we can see a full adder circuit built using majority gates.

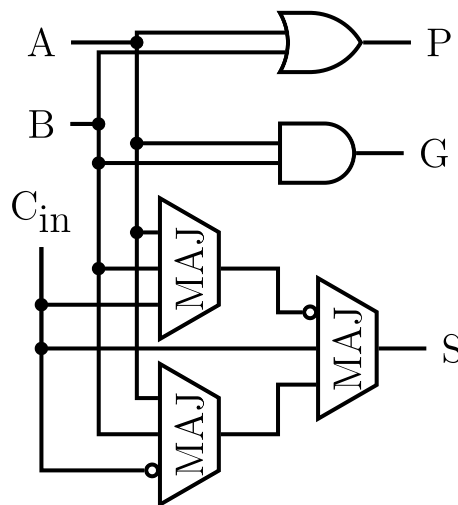


Figure 4.8: Full Adder using Majority Gate with generate and propagate output signals.

The full adder also outputs the G (generated) and P (propagated) signals. They are used to speed up the calculation for all the carry bits. The G output signal means that the sum of A and B input signals generate a carry-out if the addition will always carry, regardless of whether there is an C_{in}. In other words, it generates the carry if and only if both of A and B are 1. Equation 4.4 shows the formalization of the G output signal, where i is the index of i-th bit's sum. The dot signal denotes a logical AND function.

$$G_i = G(A_i, B_i) = A_i \cdot B_i \tag{4.4}$$

Differently from the G output signal, the P output signal means that the sum of A and B input signals will propagate a carry out if there is an C_{in}. In other words, it propagates the carry if and only if at least one of A or B is 1. Equation 4.5 shows the formalization of the P output signal, where i is the index of i-th bit's sum. The plus signal denotes a logical OR function.

$$P_i = P(A_i, B_i) = A_i + B_i \tag{4.5}$$

We need to emphasize that propagate and generate signals only depend on a single bit of the A and B inputs and do not rely on any other bit in the sum.

Each sum's carry-out bit is produced in parallel by a circuit called Carry Look-Ahead (CLA) unit. In Figure 4.9, a 4-bit Adder with the CLA circuit is shown. Its purpose is to speed up the carry-outs calculation by doing it in parallel with the adders' sum calculation. All the carry-outs are produced in the CLA unit only by looking at G (generated) and P (propagated) output signals from all full adders. The respective carry signal is calculated as seen in Equation 4.6.

$$C_i = G_i + P_i \cdot C_{i-1} \tag{4.6}$$

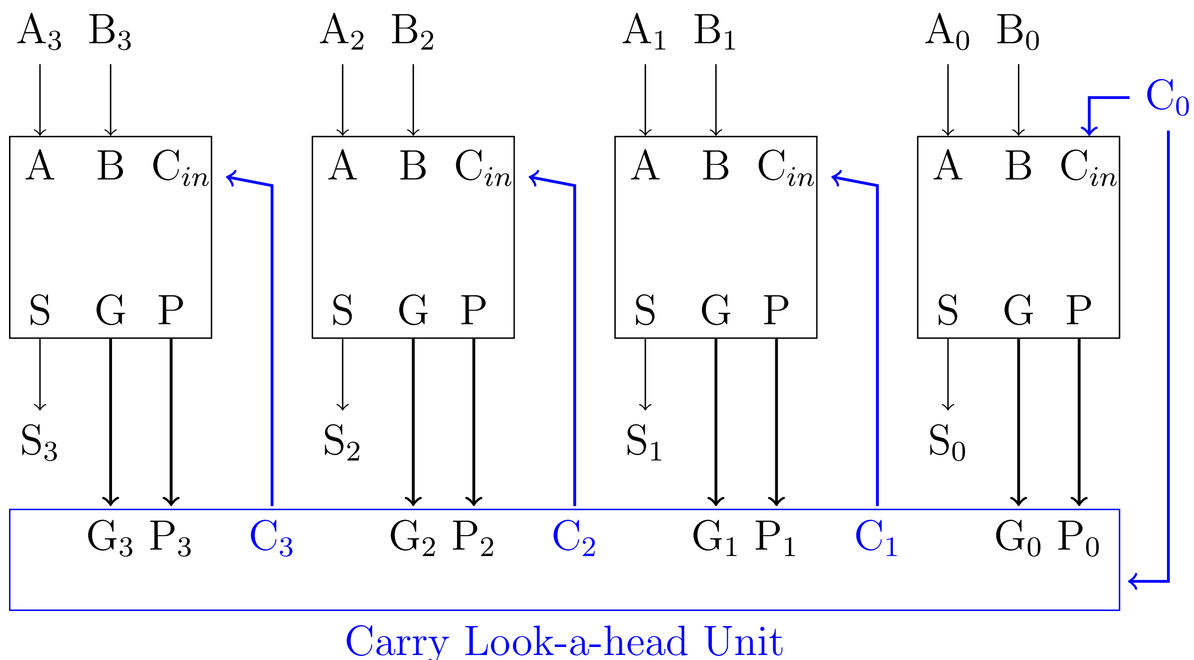


Figure 4.9: 4-bit Adder with Carry Look-ahead circuit.

Figure 4.10 shows the simulation results from the circuit. Although simulations were performed with all the possible combinations of input values (and all of them presented the expected behavior), for the sake of limited space, only one simulation graph are shown. In the example, the values of input numbers A and B are set as 3 (0011_2) and 7(0111_2), respectively. After a certain amount of time, the result converge to value 10(1010_2).

Due to space constraints, the code that implements the 4-bit Adder with Carry Look-ahead circuit using DNAr-Logic is omitted. It is present at the DNAr-Logic git repository[1].

[1] https://git.nanocomp.dcc.ufmg.br/dnacomputing/dnar-logic/-/blob/master/examples/jkff.r

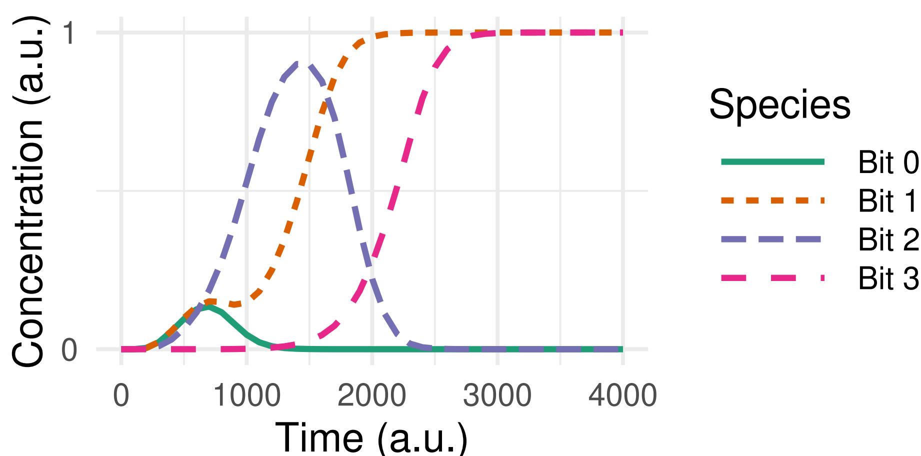


Figure 4.10: Simulation result of output signal S of each individual full-adder when sums 3 + 7 resulting in 10 (1010 - binary coded). The axis are shown in arbitrary units (a.u.).

4.1.4 Discussion

Only the simulations from the largest circuits — 4-bit CLA Adder and the JK-Flip-Flop circuits — will be discussed for the scalability argument. This choice was made because all the smaller circuits took fewer resources to build and simulate (number of species, reactions, and simulation time). The simulation of 4-bit CLA Adder and the JK-Flip-Flop circuits took around 37.1 and 73.3 seconds, respectively, in an Intel Core i5-4440 CPU at 3.10GHz, 16 GiB of RAM and Debian Linux(kernel version 5.7.15). The release of the R software suite used was 4.0.2. The actual number of solved reactions for the 4-bit CLA Adder was 676, while the JK-Flip-Flop CRN solved 408 reactions.

Although there are fewer reactions to simulate the JK-Flip-Flop, its simulation needed 180 points to have sufficient resolution to see the clock events and signal changes. The 4-bit CLA Adder, on the other hand, needed only 60 points. The difference of simulation points needed explains the difference in run time between the two circuits, even though the fastest simulation has a higher number of reactions. Further, the number of reactions and the convergence time of real experiments justify the development of packages such as the DNAr-Logic. In a few lines, one can describe the circuits and the package creates the necessary reactions, which would be a challenging task for a developer to do manually.

Future DNA computers will be much slower than traditional electronic machines. In our examples, simulations show that the reactions would take three to four orders of magnitude of time to converge when executing in the real world. It is important to emphasize that DNA digital systems have no intention of replacing conventional digital circuits. Still, open new niches for the application of digital logic in health treatments *in*

vitro and *in vivo*, such as smart drug delivery, diagnose the presence of pathogen's genetic information or treatment of diseases caused by malfunctioning DNA.

In this Section, we implemented, simulated, and validated the DNAr-Logic extension to the DNAr package. This extension contains the implementation in R language, using the framework provided by the DNAr package. The objective of this extension to the DNAr package is to make the construction of complex digital logic circuits in DSD technology easier and systematically correct. The proposed tool represents an advancement in the future development of molecular logic, which should positively impact the field's progress.

4.2 DNAr-Analog

The development of DNAr for simulating CRNs and DNAr-Logic to ease the development of molecular digital logic circuits showed the benefits of abstracting the low-level details of chemical intricacies from circuit developers. However, the number of species and reactions needed to implement these digital logic circuits explode as soon as the circuits increase in size and complexity. This quick escalation currently limits effective practical implementation on the laboratory bench.

When analyzing the recent literature, the proposals for analog DNA circuits have significantly increased lately (99, 84, 107, 106, 67, 69, 92). The increasing interest from researchers in analog molecular circuits has some possible explanations: 1) chemical species concentrations that vary continuously over time are inherently an analog signal, and 2) using approximate computation as a trade-off for gains obtained with ease of implementation of the circuit (67). These factors make utilizing analog circuits more feasible, less costly, and better interface with biological samples in a laboratory. Nevertheless, the works in the literature are independent, implemented in different simulation tools, and with distinctive parameters, making it difficult to build larger circuits and systems by reusing what already exists.

Due to the facts mentioned above, the development of DNAr-Analog library (68) to compile gates/units from different related works seemed a natural further development after DNAr-Logic development to the DNAr package. This new package is a project led and developed by Poliana Aparecida Corrêa de Oliveira as part of her Ph.D. work. It is still in the early stages of development and is a collaborative effort of both parties.

Chapter 5

Implementation of DNA Clustering

This chapter describes the thesis proposal. It first presents the implementation of the K-medoids unsupervised learning algorithm in Chemical Reaction Networks (CRNs), which later will be converted to DNA Strand Displacement (DSD) reactions. Section 5.1 will cover the implementation details: from the explanation on how to cluster the DNA strands by similarity, the design of the algorithm in Chemical Reaction Networks (CRNs) and to the details related to the speeds of the reactions involved, which are based on the physicochemical characteristics of the solution.

5.1 K-medoids in DNA

The K-Medoids clustering algorithm, described in Section 2.3, is an unsupervised learning algorithm. This algorithm is able to choose which elements are capable of maximizing similarity with all the other data in the set, given as input a universe set containing several elements and a similarity criterion between them. In other words, at the end of the execution (and without having received any type of previous training), the algorithm will create subsets that, within each one, contains elements which have the most significant similarity to each other.

The usefulness of an unsupervised learning algorithm in the biological context is to allow similarity analysis in the genetic code itself, be it DNA or RNA. Even when two complementary genetic sequences on different strands have small differences (due to replication errors and mutations), they can still interact with each other. When *in vivo*, the cellular apparatus itself correct these errors; in other cases, they are harmless, as they end up encoding the same proteins (in the case of messenger RNA).

However, the most viable way (26) to carry out this analysis of the similarity between genetic domains is *in silico* (50). First, it is necessary to do the genetic sequencing of several samples and store this data in databases; then, run the clustering algorithm on computers capable of working with these large volumes of data. Finally, after the execution

time, analyze the results (50). This work proposes the possibility of using the DNA (or RNA) both as hardware that runs the algorithm and as the input data itself, allowing the processing to be massively parallelized due to the numerous chemical reactions occurring simultaneously in the solution.

Thus, when using chemical reactions to function as “wet hardware”, we can obtain massive performance gains in problems that are often difficult to solve: both related to the execution time and the energy consumed in the process, as shown in (26).

5.1.1 Clustering Based on DNA Domain Similarity

As explained in Section 2.1, DNA (and RNA) strands contain bases that hybridize with their complementary bases, i.e., A-T and C-G (or A-U and C-G in RNA). A domain X is a sub-sequence of bases present in a single DNA strand. A single DNA strand may contain one or more domains. A complementary domain X* is a domain that contains complementary bases that can hybridize with bases present in domain X. So, a domain X and its complementary domain X* must have the same length, i.e., the same quantity of bases, denoted by $|X| = |X^*|$. We define in Equation 5.1 the complementarity ratio, $\delta(X, X^*)$, of domain X relative to X* as the number of bases from both domains (X and X*) that complement themselves on the corresponding positions divided by the total number of bases in domain X:

$$\delta(X, X^*) = \frac{\text{Complementary Bases}(X, X^*)}{|X|} \quad . \tag{5.1}$$

When the complementarity ratio is 1 (meaning 100%), it means all the bases in domain X can hybridize with the bases in domain X*. When the complementarity ratio is less than 1, it means that some bases in the complementary domain X* do not match with its counterpart domain X. Although it’s possible to both domains hybridize, the binding velocity is reduced and can even not occur when the dissimilarities between X and X* are high enough. Also, we need to emphasize that both the position on the strand and type of bases have influence on the reaction speed. Because DNA strands may have more than one domain, the complementarity ratio needs to consider all domains, denoted as Dom(S), present in a strand S. Equation 5.2 defines the complementarity ratio between two different DNA strands S and T, considering each of its domains, i.e., $s \in \text{Dom}(S)$ and $t \in \text{Dom}(T)$:

$$\text{Complementarity Ratio}(S, T) = \frac{\sum_{s}^{\text{Dom}(S)} \sum_{t}^{\text{Dom}(T)} \delta(s, t)}{|\text{Dom}(S)| \times |\text{Dom}(T)|} \quad . \tag{5.2}$$

Table 5.1: Example DNA species.

Name	Sequence
S_1	AAATAATTA
S_2	TAATTCGGT
S_3	ACCTACCTA
S_4	TAGGTATTT

Table 5.1 shows four DNA species named from S_1 to S_4, each one containing only one domain. We wrote each of these species in 5' to 3' direction. For example, let us calculate the Complementarity Ratio between species S_1 and S_2. Because S_1 and S_2 are constituted by only one domain each, we can use equation 5.2 as:

$$\text{Complementarity Ratio}(S_1, S_2) = \frac{\delta(S_1, S_2)}{|\text{Dom}(S_1)| \times |\text{Dom}(S_2)|} \quad . \tag{5.3}$$

Because the DNA hybridization starts between 5' from one strand and 3' from the other strand, we need to reverse one of the strands. Arbitrally, we choose S_2 to reverse. So, the last base from S_1 (adenine) will bond with the first base from reversed S_2 (thymine). Figure 5.1a shows how many complementary bases are between S_1 and S_2, which is six. Because both strands have nine bases each, we have $\delta(S_1, S_2) = 6/9 \approx 0.667$. Taking equation 5.3 and substituting the values we have:

$$\text{Complementarity Ratio}(S_1, S_2) = \frac{0.667}{1 \times 1} = 0.667 \quad . \tag{5.4}$$

As another example, when calculating the complementarity ratio between species S_1 and S_4, we obtain a value of $\delta(S_1, S_4) = 7/9 \approx 0.778$. Then, substituting in equation 5.2:

$$\text{Complementarity Ratio}(S_1, S_4) = \frac{\delta(S_1, S_4)}{|\text{Dom}(S_1)| \times |\text{Dom}(S_4)|} \tag{5.5a}$$

$$= \frac{0.778}{1 \times 1} = 0.778 \quad . \tag{5.5b}$$

From the result of equations 5.4 and 5.5, it is possible to see that species S_1 and S_4 have a higher complementarity ratio than species S_1 and S_2. This high complementarity favors a higher "affinity," in other words: favors a higher reaction rate between species S_1 and S_4 than between S_1 and S_2.

Some strands having a higher "affinity" will naturally react faster. This natural chemical behavior of "affinity" between DNA strands can be used as a "distance function" in a clustering algorithm. However, we must emphasize that the hybridization rate is dependent not only on the complementarity ratio, but also on temperature and strand concentration in solution (66).

(a) Hybridization between strands S_1 and S_2; It is possible to see six bonds between S_1 and S_2.

(b) Hybridization between strands S_1 and S_4; It is possible to see seven bonds between S_1 and S_4.

Figure 5.1: This figure shows the hybridization between strands. A line represents the presence of a hydrogen bond between bases; a dot otherwise represents the lack of a hydrogen bond. Please note that strands S_2 and S_4 are reversed when compared with Table 5.1, because of the opposite 5' to 3' hybridization.

Although the hybridization rate is an empirical measured value in a laboratory, a software developed by DNA and Natural Algorithms Group at Caltech, called Multistrand (6), tries to approximate it by simulating the chemical interactions between bases in DNA strands. It was applied as a support tool in this work to calculate all the hybridization rates.

5.1.2 Algorithm implementation in CRN

The Chemical Reaction Network (CRN) developed in this work to implement one iteration of the K-Medoids algorithm is organized into gates, each one carrying a specific step in the iteration cycle. Because the K-Medoids algorithm needs various successively iterations to converge, all the gates are designed to be reused in all iterations. To achieve this behavior, we applied the buffering technique (47): initially, all gates are buffered (i.e., inactive) in the solution. When a specific step in the iteration needs to run, the gate must be unbuffered (i.e., turned active) to proceed. Some gates convert their inputs into an inactive (buffered) state. This inactivation is needed because these species must be present in the solution when a new iteration cycle starts. So they will be restored to an active state (unbuffered) when the iteration ends and a new iteration needs to start.

Figure 5.2 illustrates the flow of interactions between species and gates: input species are white circles, inactive input species are grey circles, intermediate species are only text, and gates are colored boxes. Lines of various styles denote the route of interaction between species. For reasons of simplicity, we omit the activation/inactivation process of the gates, and the figure depicts an example with only four input species (the same species from Table 5.1). In the following sections, we describe the algorithm dataflow present in

Figure 5.2, step by step, which includes the purpose of each gate, its behavior, how this behavior can be modeled in a CRN, and how it interacts with the other components.

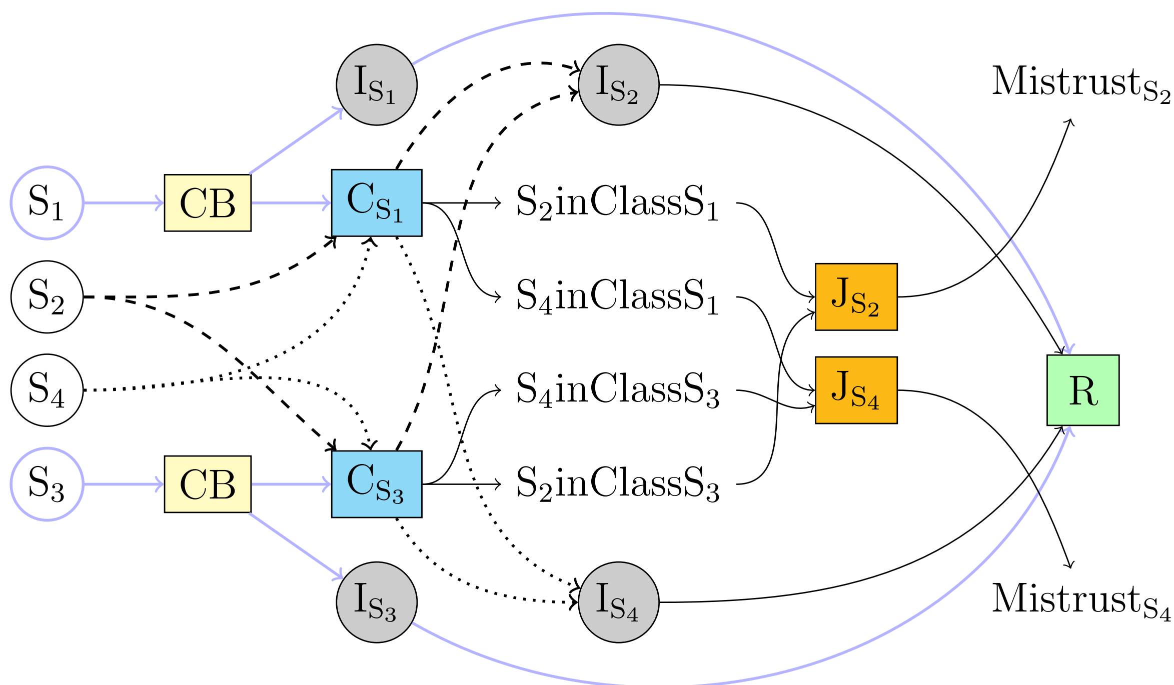


Figure 5.2: Example circuit for one iteration of clustering algorithm with four species S_1, S_2, S_3 and S_4. Two of them (S_1 and S_3) are used as medoids to classify species S_2 and S_4. Due to the complementarity ratio of both S_2 and S_4 to S_1 and S_3 being different, S_2 and S_4 will react with the Counter Gates (C) at different speeds. This will generate different concentrations of count signals. When reacting with the Join Gate (J), these signals will produce the Mistrust Signal, used in classification. The greater difference between input concentrations of the Join Gate, the higher its output concentration is.

5.1.2.1 Counter Builder Gate

The first step in the iteration is to choose the initial medoids (as seen in Section 2.3.2.2). For example, suppose we are starting the first iteration of the algorithm, and the initial medoids were randomly chosen as S_1 and S_3. These species must be absent from the solution and substituted by inactive versions I_{S_1} and I_{S_3} respectively; otherwise, they will interfere in this and subsequent iterations. This is done by the Counter Builder (CB) gates, represented as yellow boxes (a) in Figure 5.3. The CB gates consume species S_1 and S_3 producing inactive versions I_{S_1} and I_{S_3}, represented as gray circles in Figure 5.3. It also produces C_{S_1} and C_{S_3} that are the respective Counter Gates (C) for S_1 and S_3, shown as blue boxes (b) in Figure 5.3. This can be modeled as reactions shown in the following

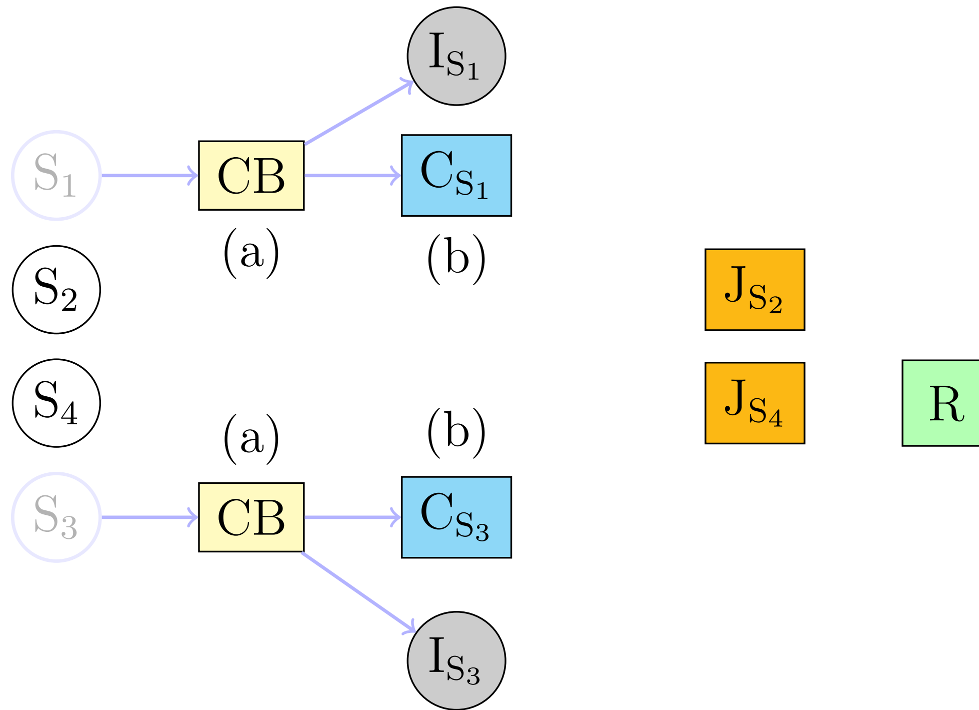


Figure 5.3: Species S_1 and S_3 will be used as medoids, so when activating their respective Counter Builder (CB) gates in yellow, they generate their respective Counters (C_{S_1} and C_{S_3}). Species S_1 and S_3 are inactivated by Counter Builder gates producing species I_{S_1} and I_{S_3}.

equations 5.6 and 5.7:

$$CB + S_1 \longrightarrow CB + I_{S_1} + C_{S_1} \tag{5.6}$$

$$CB + S_3 \longrightarrow CB + I_{S_3} + C_{S_3} \tag{5.7}$$

The Counter builder (CB) gates are present before and after the arrow because they are a chemical species that is not consumed in the process and remains present in the solution.

It is vital to say that both Counter Builder Gates (CB) and Counter Gates (C) have an active and an inactive (buffered) state. These states are necessary to correctly implement the reuse of species and gates in various iterations present in the algorithm and are toggled via species in the solution.

It is also important to emphasize that the concentration of the input species must be the same or as close as possible; that is, there must be no significant variation between the concentrations of the DNA strands that will be used as input to the algorithm, since this difference in concentration ends up influencing the speed of reactions and, consequently, invalidating the principle of reaction rate being controlled by the affinity of the DNA strands. More details of the implementation of reactions are in Appendix A.

5.1.2.2 Counter Gate

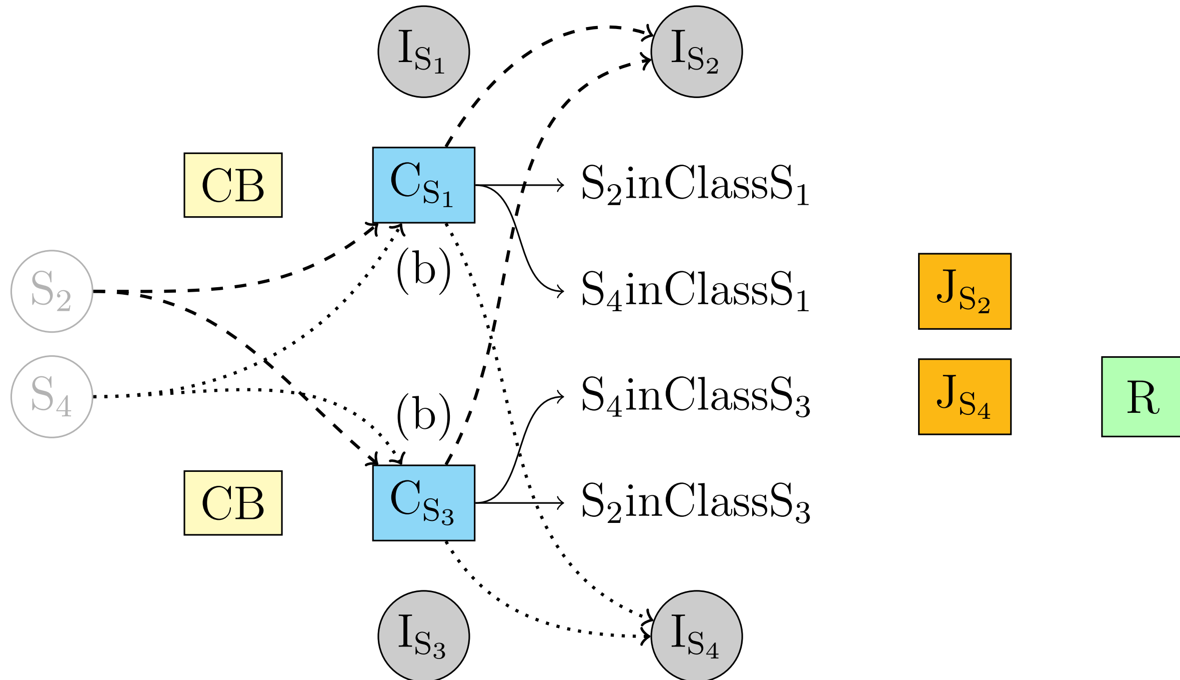


Figure 5.4: The remaining input species S_2 and S_4 react with the counters gates C_{S_1} and C_{S_3}. The input species S_2 are consumed by C_{S_1} and C_{S_3} (dashed lines) being converted into its inactive state I_{S_2}. The same happens with input species S_4 consumed by C_{S_1} and C_{S_3} (dotted lines) being converted into its inactive state I_{S_4}. The velocity of consumption of both S_2 and S_4 are defined by their "affinity" with counters C_{S_1} and C_{S_3}, as explained in Section 5.1.1. Both counter gates C_{S_1} and C_{S_3} produces intermediate species "inClassS_1" and "inClassS_3" which concentration encodes the "affinity" of input species S_2 and S_4 to species S_1 and S_3.

The next step in the iteration, after the Counter Builder (CB) gates generate the Counter (C) gates for the selected medoids (S_1 and S_3 in this example), is the "counting," i.e., the classification of the remaining non-medoid input species (S_2 and S_4 in this example). Each Counter (C) gate is represented as a blue box (b) in Figure 5.4. The dashed and dotted lines show the course of the S_2 and S_4 species in the circuit, respectively. First, the non-medoid species (S_2 or S_4) reacts with all the counter gates according to their reaction rate. The reaction rate is directly related to their "affinity," i.e., their complementarity ratio (defined in Section 5.1.1) between each input species and each counter gate.

Next, for each copy of non-medoid species (S_2 or S_4), a counter gate will produce an inactivated version of it (I_{S_2} or I_{S_4}) and an intermediate species ("inClassS_1" or "inClassS_3") which concentration encodes the "affinity" of input species S_2 and S_4 to species S_1 and S_3. Equations 5.8 and 5.9, for example, show the modeled reactions of this behavior for S_2

(but the same model applies to S_4 as well):

$$C_{S_1} + S_2 \xrightarrow{k_1} C_{S_1} + I_{S_2} + S_2\text{inClass}S_1; \tag{5.8}$$

$$C_{S_3} + S_2 \xrightarrow{k_2} C_{S_3} + I_{S_2} + S_2\text{inClass}S_3. \tag{5.9}$$

The counter gates C_{S_1} and C_{S_3} are present before and after the arrow because it is a chemical species not consumed in the process and remain present in the solution.

The higher the "affinity" between an input species and a gate, the faster the reaction. This behavior can be seen as different reaction ratios k_1 and k_2. When $k_1 = k_2$ both reactions have the same velocity. When $k_1 > k_2$ then the reaction described in equation 5.8 will produce a higher concentration of its outputs faster than reaction described in equation 5.9 (and vice-versa). This natural competition between counter gates and input species drives the final concentration level of intermediate species ("S_2inClassS_1" and "S_2inClassS_3" in this example) when the input species are fully depleted in the solution. More details of the implementation are in Appendix A.

5.1.2.3 Join Gate

After the Counter (C) gate generated the intermediate species "S_2inClassS_1", "S_2inClassS_3", "S_4inClassS_1" and "S_4inClassS_3", all the Join (J) gates can start processing these signals. Each Join (J) gate is represented as a orange box (c) in Figure 5.5 and are responsible for generating the analog Mistrust signal. The Mistrust signal represents the level of mistrust of the classification provided by the Counter gates through the intermediate signals "XinClass$Y$" which means X species in Y cluster.

The Join gates need two intermediate signals as inputs to generate the Mistrust signal. These inputs need to be two different intermediate species produced by two different counter gates that use the same species as input. For example: one join gate will have as input signals "S_2inClassS_1" and "S_2inClassS_3"; similarly, other join gate will exist to process the signals "S_4inClassS_1" and "S_4inClassS_3". If more counter gates generated more intermediate signals, there would be various join gates to cover, two-by-two, all the possible analyses of classifications.

When the trust in classification by Counter (C) Gates is high (in other words, classified species have a high "affinity" for only one medoid), then the mistrust signal concentration is low. Conversely, when the trust in classification is low, i.e., the classified species have an equal "affinity" for more than one medoid, the mistrust signal concentration is high.

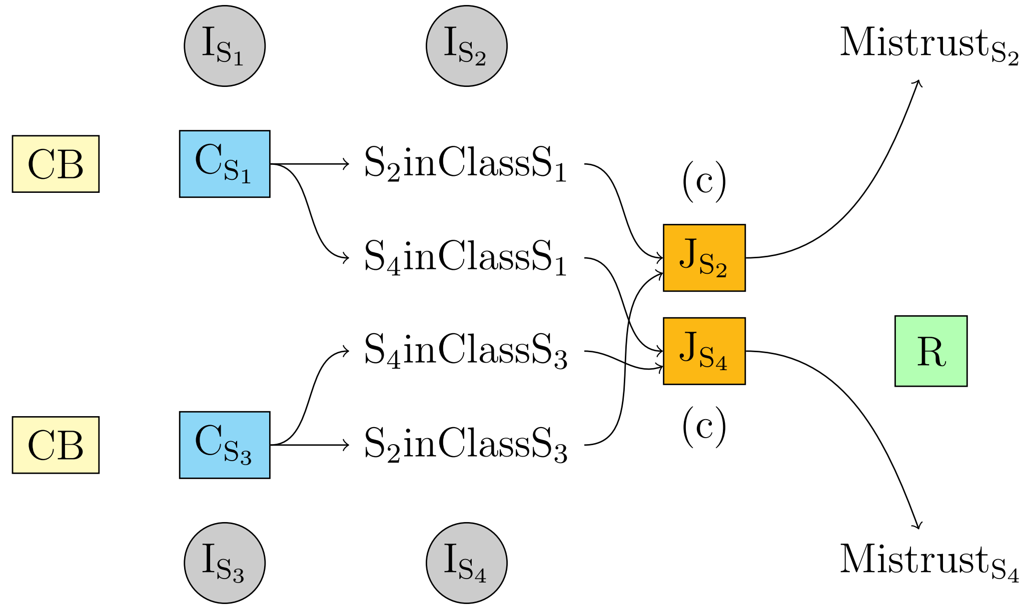
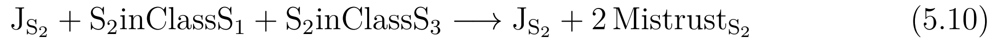
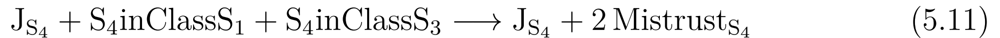


Figure 5.5: Both counter gates C_{S_1} and C_{S_3} produces intermediate species "inClassS_1" and "inClassS_3" which concentration encodes the "affinity" of input species S_2 and S_4 to species S_1 and S_3. For example: the Join (J) gate will pair each copy of the intermediate species "S_2inClassS_1" and "S_2inClassS_3" to generate the output signal "Mistrust$_{S_2}$" which high concentration levels means high mistrust in classification of species S_2.

It is possible to see that the Join gate works as an analog subtraction operation: if both inputs have the same concentration, the output is zero; if inputs have significantly different concentrations, the output concentration is high. The higher the concentration difference from the inputs, the higher the output concentration. This analog subtraction behavior can be modeled as the reactions present in Equations 5.10 and 5.11:



$$J_{S_2} + S_2inClassS_1 + S_2inClassS_3 \longrightarrow J_{S_2} + 2\,Mistrust_{S_2} \tag{5.10}$$



$$J_{S_4} + S_4inClassS_1 + S_4inClassS_3 \longrightarrow J_{S_4} + 2\,Mistrust_{S_4} \tag{5.11}$$

The join gates J_{S_2} and J_{S_2} are present before and after the arrow because they are a species not consumed in the process and remain present in the solution. Also, it is necessary to notice that the join gate generates two copies of the Mistrust signal for one copy of each "inClass" signal. This design is for correctly executing the analog subtraction of the concentration of inputs.

Same as Counter Builder Gates (CB) and Counter Gates (C), Join (J) gates too have an active and an inactive (buffered) state. These states are necessary to correctly implement the reuse of species and gates in various iterations present in the algorithm and are toggled via species in the solution.More details of the implementation are in Appendix A.

5.1.2.4 Restorer Gate

Figure 5.6: At the end of one iteration, the Restorer (R) gate is responsible for "cleaning" the solution from the intermediate species (such as "inClass" and "Mistrust" signals) and reverting the inactivation of the input species. In our example, the species "I_{S_1}", "I_{S_2}", "I_{S_3}" and "I_{S_4}" are restored to their previous state, i.e., the active species "S_1", "S_2", "S_3" and "S_4", respectively. These active species are, then, ready to be used in a new iteration that will soon start.

Last (but not least), the restoration step is done by the Restorer (R) gate. It is represented as a green box (d) in Figure 5.6. It is responsible for making the circuit ready for a new iteration, which involves reverting the inactivation from the currently running iteration's input species and removing the leftover species from the solution. In other words: at the end of every iteration, it reverts the inactivation of the input species and removes from the solution the leftover species produced from the counter gates, such as signals "inClass," and join gates output "Mistrust" signals. In our example, the species "I_{S_1}", "I_{S_2}", "I_{S_3}" and "I_{S_4}" are restored to their previous state, i.e., the active species "S_1", "S_2", "S_3" and "S_4", respectively. Also, the "inClass" and "Mistrust" signals are no longer present in the solution, which may interfere in the execution of the next iteration. The

behavior described can be modeled as the reactions present in Equations 5.12a to 5.12f.

$$R + I_{S_1} \longrightarrow S_1 + R; \quad (5.12a)$$

$$R + I_{S_2} \longrightarrow S_2 + R; \quad (5.12b)$$

$$R + I_{S_3} \longrightarrow S_3 + R; \quad (5.12c)$$

$$R + I_{S_4} \longrightarrow S_4 + R; \quad (5.12d)$$

$$R + \text{Mistrust}_{S_2} \longrightarrow R; \quad (5.12e)$$

$$R + \text{Mistrust}_{S_4} \longrightarrow R. \quad (5.12f)$$

The restorer gate R is present before and after the arrow because it is a species not consumed in the process and remain present in the solution.

Like all previous gates, Restorer (R) gates have an active and an inactive (buffered) state. Without this capability, all the gates would always be active, interfering in the computation. More details of the implementation are in Appendix A.

5.1.2.5 Iteration Setup

Because the K-Medoids algorithm needs various iterations to converge to a result fully, all the previously described gates (except for Restorer Gates) must be inactivated (buffered). Without this action, they will continue to operate and consume the inputs generating outputs. After this action, the Restorer Gates will reenable the original inputs in the solution and remove the intermediary signals that could interfere with the new iteration that will start soon.

5.2 Simulations and Results

This section will discuss the simulation results of CRN K-Medoids algorithm presented in Section 5.1. First, Section 5.2.1 presents how the simulation framework is organized. Section 5.2.2 explains how the convergence criteria of K-Medoids algorithm was chosen. Finally, Section 5.2.3 discuss the simulation results of four case studies, in growing order of complexity.

5.2.1 Simulation Framework

To simulate the circuit described in Section 5.1 we first need to calculate the reaction rates between all the species. The Multistrand software provides these reaction rates to be used in our simulation (6). We do not need to know these reaction rates in advance in the real world because they are not needed. This natural chemical behavior of "affinity" between DNA strands drives the reaction rate making them naturally hybridize faster or slower, depending on their complementarity level.

The DNAr software package was used to simulate the K-Medoids algorithm in Chemical Reaction Networks (CRNs). First, we choose (or generate) strands with some sequences of bases. Then, they will be the input of Multistrand software to calculate all the possible reaction rates between the strands based on the complementarity of the sequences themselves. After that, the reaction rates will be fed into the K-Medoids CRN algorithm. Then, it will simulate the concentration change of each species in the solution along the time, concerning all previously defined gates. Finally, the mistrust signals are checked to decide if another iteration of the algorithm will be needed or if it should stop due to the convergence criteria.

Due to a current limitation in our simulation framework, we simulate each iteration of the K-Medoids algorithm (as described in 5.1) as a CRN. We called "in vitro" this first simulation phase. Then, by the end of each iteration, the concentration values of mistrust signals are used in the "in silico" simulation phase. This phase is run in a computer that checks the algorithm's convergence and decides if another iteration needs to run.

5.2.2 Convergence Criteria

K-Medoids is a very flexible clustering algorithm: it allows the maximization (or minimization) of any cost function and distance function that fits the problem well (43). To correctly determine the better cost function for the complementarity problem, we performed an analysis of six cost functions using the value provided by the concentration of the "Mistrust" signals: maximum mistrust value (max), arithmetic mean (arith), geometric mean (geo), harmonic mean (harm), median (med) and simplified silhouette (silh) (88).

The silhouette refers to a technique of understanding and confirmation of consistency within data clusters (75). It measures how well classified a point is when compared to other clusters. Its value ranges from -1 to +1. A high value means that the point is well classified in its cluster and poorly classified in neighboring clusters. Comparing the silhouette as a

Table 5.2: Average results of seven experiments that were run with six cost functions. Each line shows the experiments optimized by the respective cost function named and each column shows the calculated costs for each experiment. The color grading were done by columns, i.e., on each column the best cost is in green and the worst cost in red.

	max	arith	geo	harm	med	silh
exp_max	0.832673	0.493056	0.37647	0.268869	0.516807	0.479445
exp_arith	0.876675	0.44686	0.339491	0.269981	0.41043	0.547633
exp_geo	0.864995	0.446916	0.339102	0.268624	0.410301	0.545678
exp_har	0.943966	0.551872	0.400473	0.235161	0.579273	0.473501
exp_med	0.891653	0.45683	0.355412	0.281212	0.377633	0.610768
exp_silh	0.909497	0.558632	0.470763	0.380096	0.537222	0.810225

cost function with other methods in a clustering algorithm (such as K-Medoids) shows it is a great way to optimize the fitness of the clusters found (88).

Table 5.2 contains the average results of seven experiments that were run with six cost functions. Each experiment comprises ten species to be clusterized with two initially selected medoids. Our objective is to compare each cost function to see which one optimizes the clustering better. Each row shows the experiments optimized by the cost function presented in the first column. For example: in the first row, "exp_max", the experiment was optimized by the "maximum value" cost function; in the second row, "exp_arith", the experiment was optimized by the "arithmetic mean" cost function; and so on. Each column shows the calculated costs for each experiment. The color grading was done by columns, i.e., on column "max" the best maximum cost is in green and the worst maximum cost in red. We need to emphasize that, in all columns, the best cost is the lowest value; the exception is the "silhouette" column, where the best cost is the highest value.

Analyzing Table 5.2, we can see that in the main diagonal, we can obtain the best values because each experiment optimizes its respective cost function. Another critical piece of information that Table shows is the silhouette value in the last column. It informs how bad the clustering process is when optimizing by minimizing maximum and harmonic

mean values. The arithmetic mean, geometric mean and median minimizations show better silhouette values but are far from the best value. As explained earlier, the silhouette with a high value means that the points are well classified in their cluster and poorly classified in neighboring clusters.

Using the simplified silhouette as a cost function (88), and maximizing its value during the K-Medoids execution, shows that we can retrieve substantial gains as better clustering of DNA species.

5.2.3 Simulation Results

First, Section 5.2.3.1 shows the first case study using the four DNA species (with the two initially selected medoids) defined in Table 5.1 in Section 5.1.1. Next, Section 5.2.3.2 shows a case study of digit classification. Both case studies run the K-Medoids clustering algorithm proposed in Section 5.1.2. Lastly, the results are discussed in Section 5.2.4.

5.2.3.1 Case Study: Four strands, Nine bases

Section 5.1 explained the implementation of the K-Medoids algorithm as a Chemical Reaction Network (CRN) that can be implemented using the DSD technique. It defined four random DNA species from S_1 to S_4 in Table 5.1 and guided the reader through the algorithm inner workings in Sections 5.1.1 and 5.1.2. This section shows the algorithm simulation results for these species using the simplified silhouette as a cost function (88). Only for the sake of remembering Table 5.1, the species are defined as follows:

Name	Sequence
S_1	AAATAATTA
S_2	TAATTCGGT
S_3	ACCTACCTA
S_4	TAGGTATTT

These DNA strands were randomly chosen because the clustering algorithm design can cluster DNA strands by their complementarity independently of their sequence of nucleic bases. The reaction rates calculated by the Multistrand software based on strand

complementarity are shown in Figure 5.7 (6). The color grading is a heatmap, i.e., the maximum rate is in dark green and the minimum in white. We need to emphasize that because the graph shows the reaction rate of all possible combinations, the resulting matrix is symmetric. Due to the matrix symmetry and to simplify the visualization, we choose to omit the elements above the main diagonal.

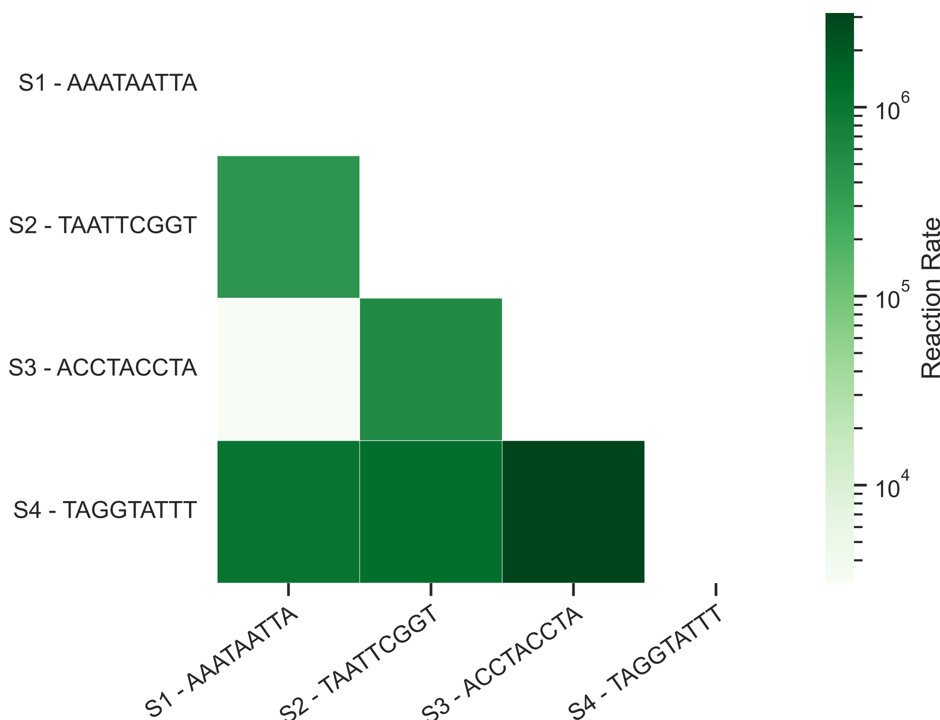


Figure 5.7: Heatmap showing reaction rates for each pair of strands of Table 5.1 calculated by Multistrand.

In this first case study, the species S_1 and S_3 were randomly chosen as initial medoids to run the algorithm. Figure 5.2 shows a visual representation of the circuit used. Figures 5.8 and 5.9 shows the concentration of output species from Mistrust and Join gates, respectively, at the end of the first iteration. Both concentration and time axis are in arbitrary units. These simulations show the dynamic behavior of one iteration of the K-Medoids algorithm (composed of species and reactions) introduced and explained in Chapter 5.

The graphic presented at Figure 5.8 shows the concentration values for the Mistrust_{S_2} and Mistrust_{S_4} species, calculated by the Join gates J_{S_2} and J_{S_4} as described in Section 5.1.2.3. Both Join gates use "inClass" species as their input, and the graph in Figure 5.9 shows their concentrations along the time.

The mistrust species indicate how well the classification was. The best scenario is for all the mistrust species to be as close to zero as possible. However, in this first iteration, the medoid configuration has yet to approach this scenario. S_2 and S_4 species have a

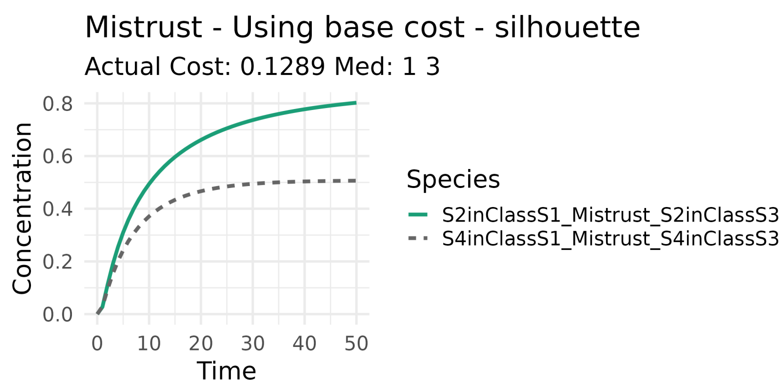


Figure 5.8: Mistrust signals from the first iteration using the species S_1 and S_3 as medoids.

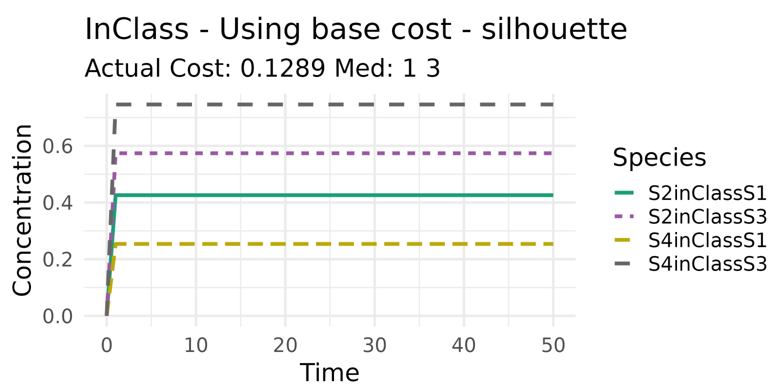


Figure 5.9: InClass signals from the first iteration using the species S_1 and S_3 as medoids.

significant level of "affinity" for the S_3 species but still compete with S_1. Nevertheless, the signals S_2inClassS_3 and S_4inClassS_3 have a higher concentration than S_2inClassS_1 and S_4inClassS_1.

Also, in this case, the S_4inClassS_1 and S_4inClassS_3 signals have a broader difference in concentration. This difference implies a better classification for S_4 and a lower concentration of Mistrust$_{S_4}$ signal. The S_2inClassS_1 and S_2inClassS_3 signals, on the other hand, generate a higher Mistrust$_{S_2}$ signal because they have a narrow difference in concentration showing a poorer S_2 classification. To sum it up: the different concentrations are explained by their different "affinity" as their reaction rate, as shown in Figure 5.7. We can denote the clustering configuration found at the end of this iteration by ($\underline{S_1}$) and (S_2, $\underline{S_3}$, S_4), where the underline marks the medoids of each cluster.

After the reactions, due to a current limitation in implementation, we must calculate *in silico* the cost function, i.e., the simplified silhouette method. The cost calculated for this configuration by the simplified silhouette method is 0.1289, close to the basal value of 0.

Figures 5.10 and 5.11 are similar to Figures 5.8 and 5.9, but the algorithm chosen medoids S_3 and S_4 when trying to minimize the clustering cost by the simplified silhouette method. Albeit S_3 and S_4 are grouped in the same cluster at the end of the first iteration,

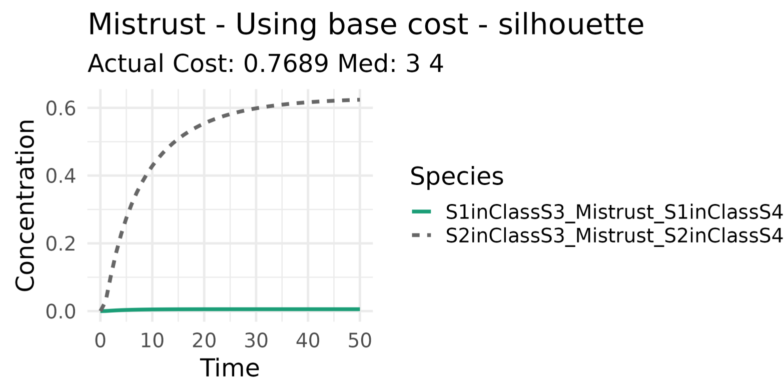


Figure 5.10: Mistrust signals from the last iteration using the species S_3 and S_4 as medoids.

S_4 is not a medoid and cannot react with S_2, nor S_3 can react with S_1. The K-Medoid algorithm works by performing a greedy search and tries to see if changing two points (one medoid by one non-medoid), i.e., S_1 by S_4, can increase cluster quality. The graphic presented by Figure 5.10 shows the concentration values for the Mistrust_{S_1} and Mistrust_{S_2} species. The graphic of Figure 5.11 shows the concentration value of the "inClass" species.

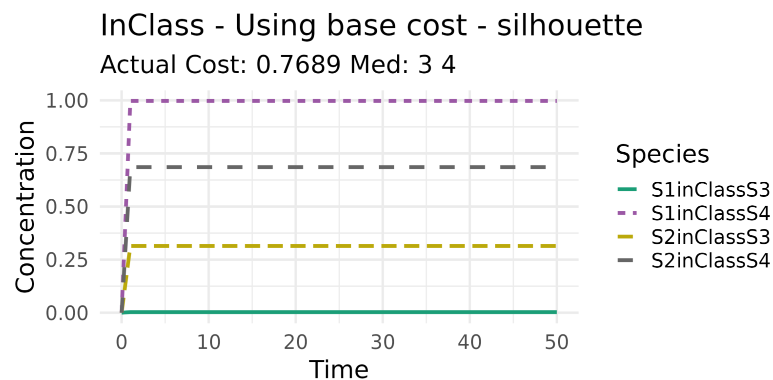


Figure 5.11: InClass signals from the last iteration using the species S_3 and S_4 as medoids.

It is possible to see that, with this new medoid configuration, the S_1 and S_2 species have more "affinity" for the S_4 species than S_3. Another exciting change in this medoid configuration is that the "affinity" of S_1 species for the S_4 is so high that the Mistrust_{S_1} signal is close to zero, meaning that S_1 signal hybridizes almost only with the S_4 medoid, as confirmed in the bottom graphic of "inClass" signals. At last, we can see that, compared with the first iteration, all the mistrust signals have a reduction in their value. Furthermore, the last iteration medoid configuration clusters the other species better, i.e., have their pairs of "inClass" signals farther apart than the initial medoid configuration seen in Figure 5.9. This better clusterization reflects the cost calculated for this configuration by the simplified silhouette method, which is 0.7689, closer to the best value of 1.

Figure 5.12 show the evolution of cost using the simplified silhouette and Table 5.3 show the final clusterization for our example.

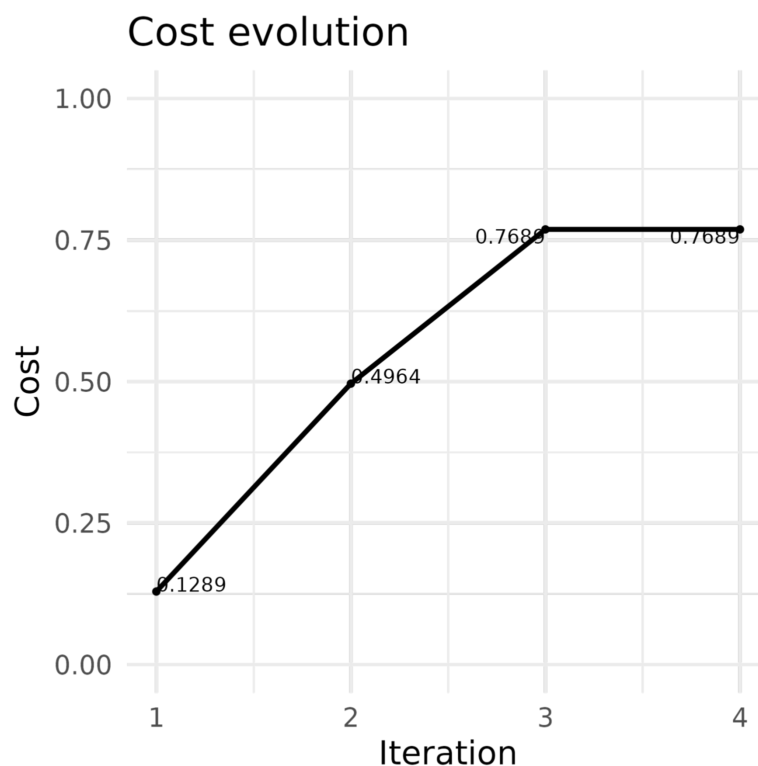


Figure 5.12: The evolution of simplified silhouette cost through the iterations.

Table 5.3: Clusters from the final iteration of K-Medoids algorithm.

Medoid	Complementary species
S_3 - `ACCTACCTA`	
S_4 - `TAGGTATTT`	S_1 - `AAATAATTA` S_2 - `TAATTCGGT`

5.2.3.2 Case Study: Digit Classification

The MNIST database (Modified National Institute of Standards and Technology database) is an extensive collection of handwritten digits. Many researchers use this database as a benchmark to evaluate machine learning algorithms' performance, particularly well-resolved with convolutional neural networks. It has a training set of 60,000 examples and a test set of 10,000 examples. Figure 5.13 shows a sample of the whole dataset. Each digit is a grayscale picture with a resolution of 28x28 pixels of a handwritten digit from 0 to 9, so each image equals 784 bytes of information.

To use these images in our molecular K-Medoids classification, first, we must encode them in DNA, choosing an encoding method among many (70). Independently of the encoding chosen, a grayscale image with 784 bytes of information will result in a long

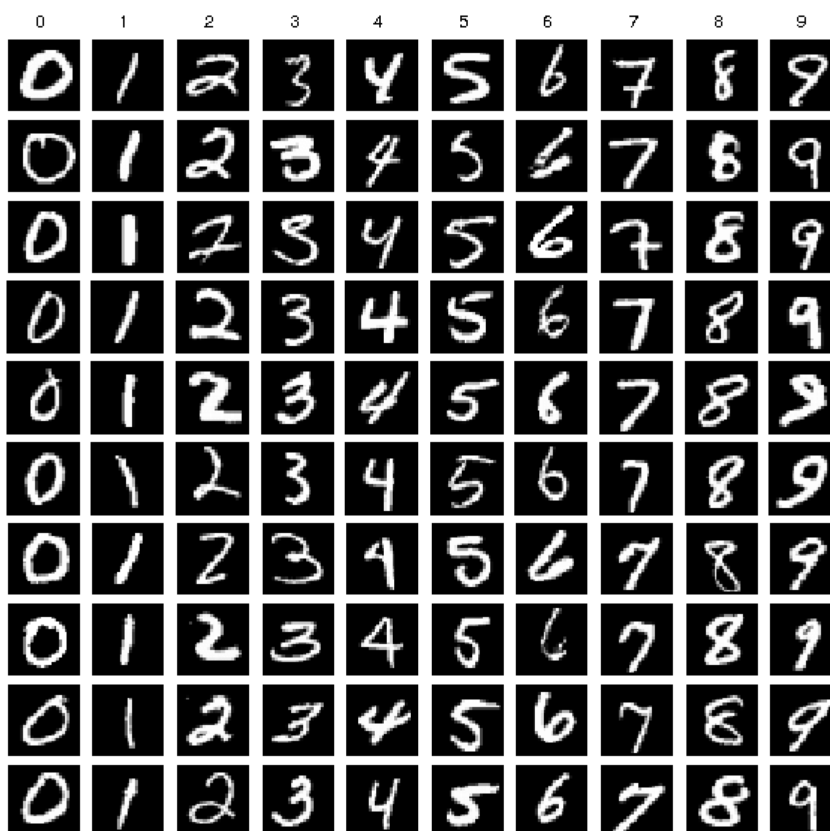


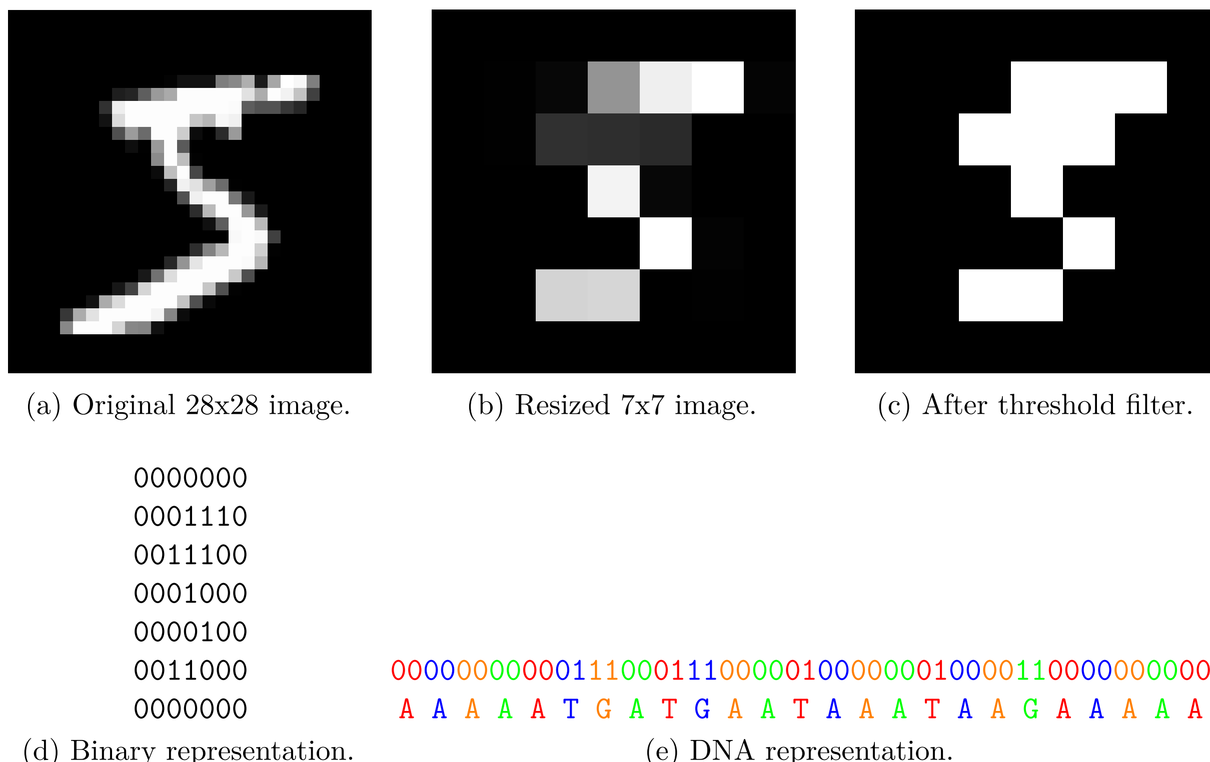
Figure 5.13: Sample of digits from MNIST dataset.

strand containing hundreds of bases, making strand displacement difficult. So, we need to reduce the information quantity to a manageable size, i.e., not going over the number of tens of bases.

The chosen method was to reduce the image resolution from 28x28 to 7x7 pixels. Information loss happens when an image's resolution is reduced because pixels are thrown away, and the resulting image becomes much more jagged. Interpolation filters exist to minimize this resizing problem by calculating an average pixel color from neighbor pixels, i.e., trying to combine information from various pixels into one. We applied the Lanczos Resampling filter for image reduction. This filter has a trade-off between the reduction of aliasing artifacts and the preservation of sharp edges.

After the reduction in resolution, the image is still grayscale. Pixels in grayscale images are unsigned 8-bit values representing from black (value 0) to white (value 255) and all the shades of gray (between 0 and 255). So, we have a reduction of 94%: from 28x28 = 784 bytes to 7x7 = 49 bytes. A significant reduction in size, but 49 bytes is still much data to evaluate a proof-of-concept algorithm. However, it is possible to reduce the amount of data even more. This resulting image goes through a threshold filter to convert the gray pixels into black or white (zero or one). This final step can encode the output image in 49 bits, just over 6 bytes.

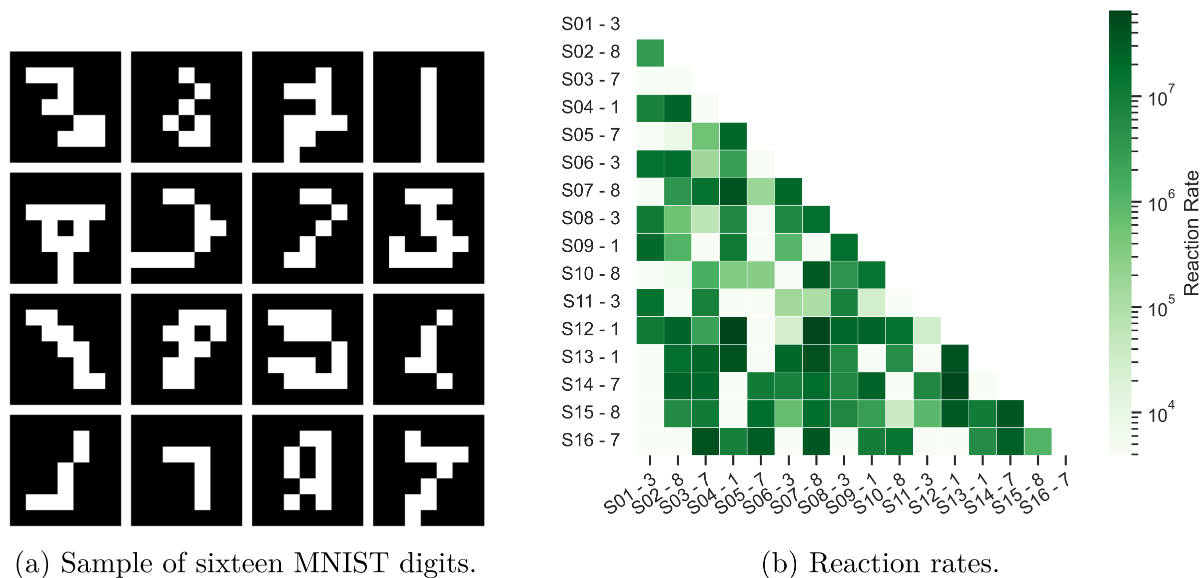
After these image-processing steps, we chose the simple transcoding method "two-



(a) Original 28x28 image. (b) Resized 7x7 image. (c) After threshold filter.

(d) Binary representation. (e) DNA representation.

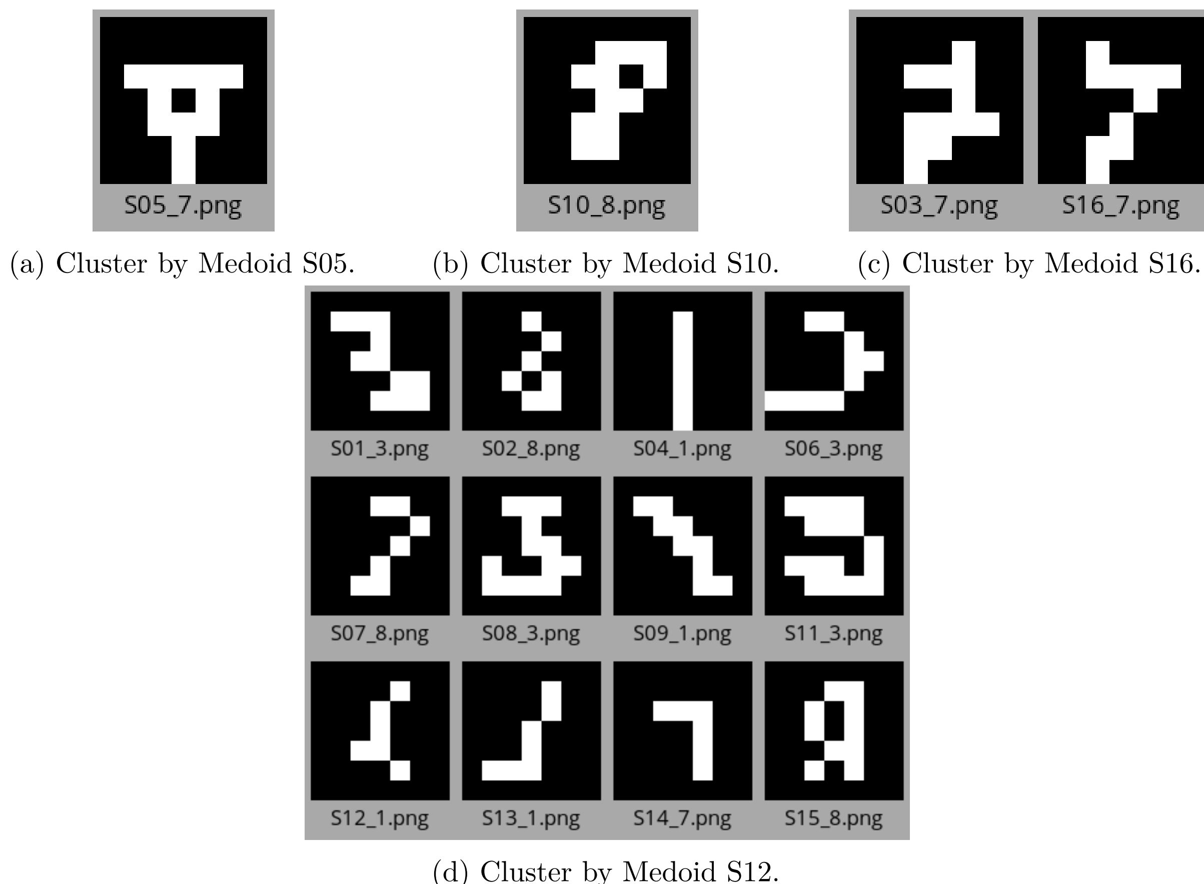
Figure 5.14: The five steps, from (a) to (e), needed to encode a picture containing hand-drawn digit five from MNIST dataset.



(a) Sample of sixteen MNIST digits. (b) Reaction rates.

Figure 5.15: Reaction rates between sixteen samples of MNIST dataset.

to-one" to convert the resulting pictures of 49 bits into DNA (70). The transcoding "two-to-one" converts bits of information into one nucleic acid base. For example, bits valued "00" become A, "01" become T, "10" become C, and "11" become G. Because 49 bits are an odd quantity, an extra zero bit was added at the end as padding to make possible encode each image into strands containing 25 bases. Figure 5.14 illustrates all the steps taken to encode each picture as DNA.



(a) Cluster by Medoid S05. (b) Cluster by Medoid S10. (c) Cluster by Medoid S16.

(d) Cluster by Medoid S12.

Figure 5.16: Final clusters of digits from Figure 5.15a.

After picking sixteen images from digits 1, 3, 7, and 8 from MNIST (4 of each) and applying the image preprocessing described before, four initial medoids from this sample were randomly selected and used as input of K-Medoids molecular algorithm presented in Chapter 5. Figure 5.15 shows the sample input with the respective reaction rates between them, pairwise.

The algorithm converged with a silhouette of 0.8731281, and the resulting four clusters are represented in Figure 5.16. It is possible to see that the hand-drawn digits depicted in the clusters are classified wrong because one cluster contains almost all the samples of data with distinctly wrong shapes. This incorrect classification is due to images containing big black pixels chunks around the edges. Also, removing so much information from the pictures in the preprocessing phase caused low-resolution images to have degraded shapes and retain only a significant amount of black regions.

Reducing the information quantity and quality through preprocessing the dataset used as input caused the wrong classification. The original hand-drawn digits have a much higher resolution of 28x28, and the whole dataset contains tens of thousands of images that provide more information to the clustering algorithms. For our purposes, we need a smaller dataset capable of exploring the potential of molecular K-Medoids in practical applications, such as similar DNA sequences with minor mutations.

For this, we resort to the old and very popular bitmap fonts used in old computer systems before PostScript, TrueType, and OpenType vector fonts were available. Bitmap fonts are low-resolution fonts designed pixel by pixel to be clear and easily identifiable on old computer screens. Figure 5.17 shows the font used in the following example: the Courrier 11 bitmap font downloaded from the Monobit website [1].

(a) All the glyphs in Courrier 11 bitmap font.

(b) Glyphs of the ten numerals from Courier 11 bitmap font.

Figure 5.17: Courier 11 bitmap font downloaded from Monobit website.

After choosing the font, we must take a subset from it. To compare to the MNIST dataset, we choose the subset of the ten numerals (Figure 5.17b) from all the glyphs in the Courier 11 bitmap font. This new dataset is a better alternative to the MNIST dataset because it has less information, as each glyph is a 5x7 = 35 bits of information, fourteen bits less than the 7x7 preprocessed images from MNIST. Besides being smaller, applying any filter in the glyphs from the bitmap font is unnecessary. So, we selected the glyphs of numbers 1, 3, 7, and 8 and applied a 10% "salt & pepper" noise to each. We generated a dataset of ten images for each number. Figure 5.18 display this dataset containing corrupted versions of the original images to simulate mutations in the original data.

Next, we selected and scrambled four glyphs from each numeral (one glyph being the original and three being the mutated ones) and gave them as input to the K-Medoids molecular algorithm. Figure 5.19b shows their respective reaction rates pairwise, and Figure 5.19a shows the glyphs used as input.

Figure 5.20 shows the result obtained by the molecular K-Medoids algorithm. Because of the use of four medoids, the algorithm tried its best to group similar images into four clusters. The first cluster shown in Figure 5.20a has the S5 species (number 1) as medoid and successfully clusters all other images containing the number 1 (albeit with imperfections). The second cluster displayed in Figure 5.20b has the S16 species (number 7) as medoid. It also shows the correct clustering of pictures containing the number seven,

[1] https://robhagemans.github.io/monobit/

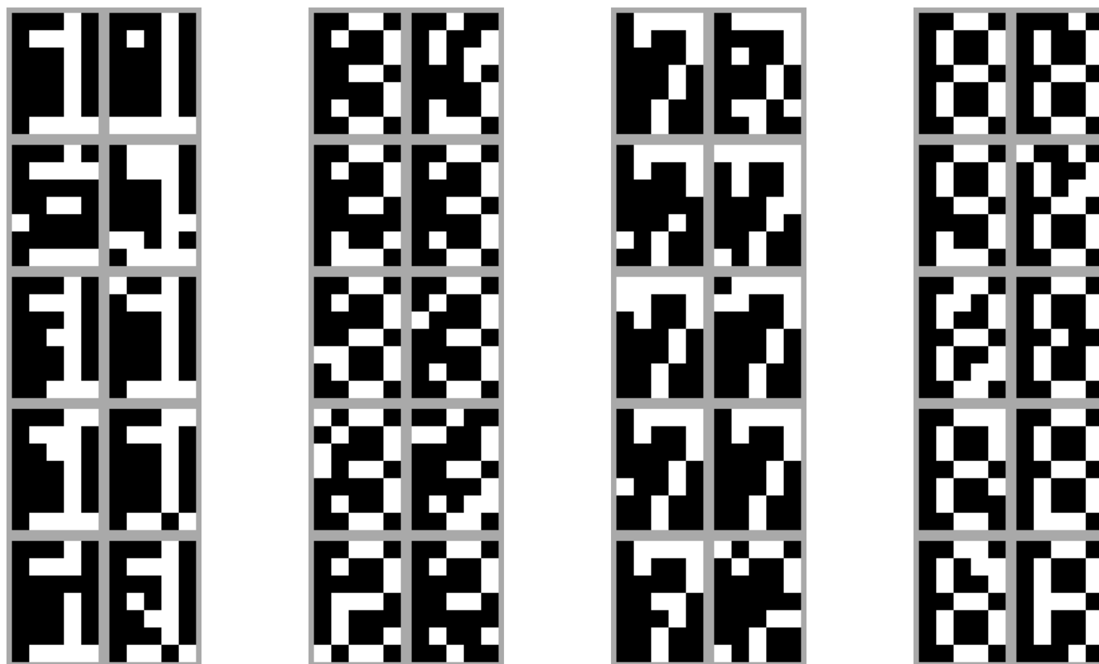
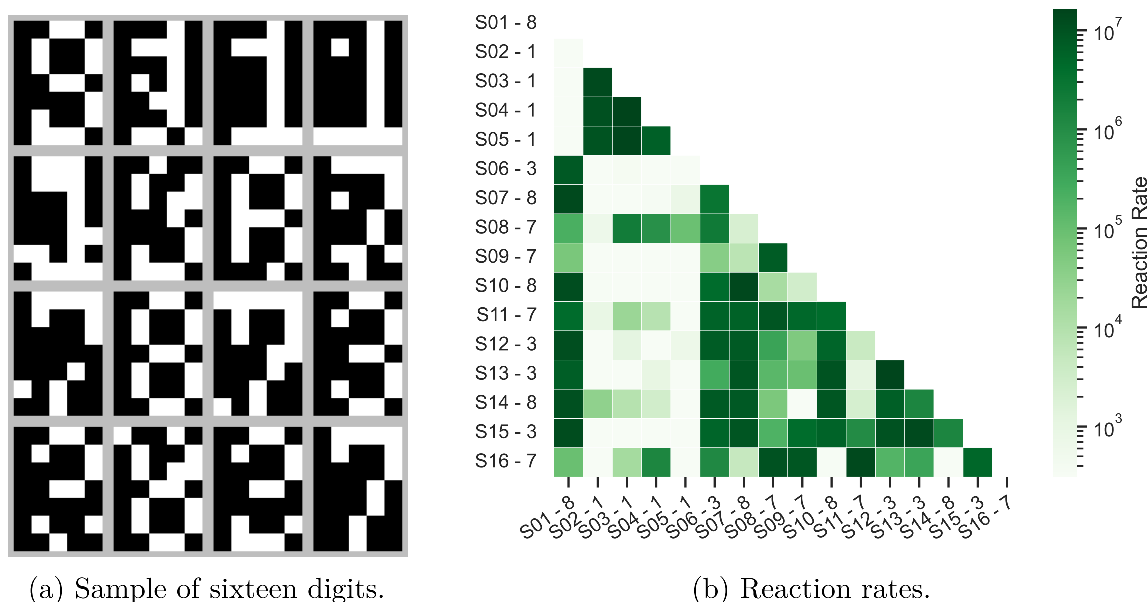


Figure 5.18: Generated dataset of glyphs 1, 3, 7, and 8 with respective mutations generated by applying 10% of “salt & pepper” noise.

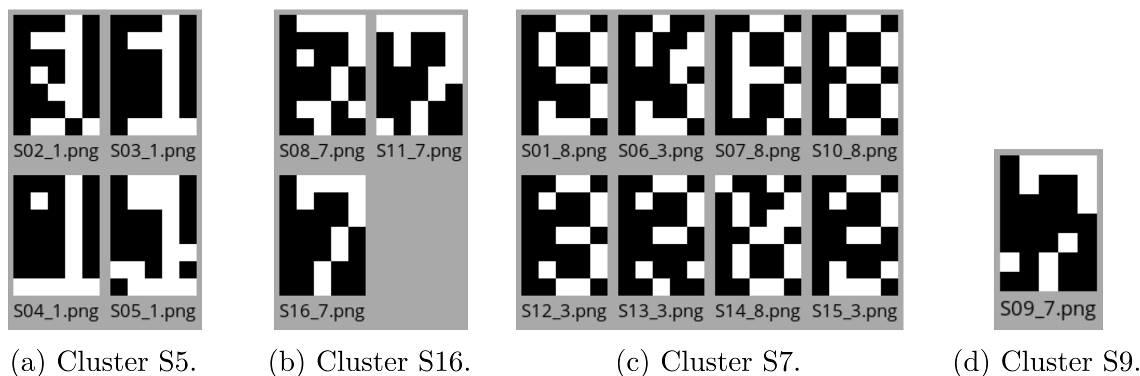


(a) Sample of sixteen digits.

(b) Reaction rates.

Figure 5.19: Reaction rates between sixteen samples of generated Courier 11 dataset.

missing only one species alone in the fourth cluster in Figure 5.20d. The third cluster depicted in Figure 5.20c has species S7 (number 8) as medoid, portraying a mixture of the correct clustering of species containing the corrupted number 8 and all the species containing the number 3. Because the difference between glyphs 3 and 8 is minimal (only 2 pixels between the non-corrupted versions), the algorithm found that the 3’s are similar to the 8’s and vice-versa. This result yields a silhouette of 0.9444282, and the clusters’ elements show significant resemblances.



(a) Cluster S5. (b) Cluster S16. (c) Cluster S7. (d) Cluster S9.

Figure 5.20: Resulting clusters after runing the molecular K-Medoids algorithm on dataset from Figure 5.19.

5.2.4 Discussion

The simulation of the K-Medoids CRN algorithm with the instance of sixteen samples of numbers presented in Section 5.2.3.2 took approximately 24 hours in an Intel Core i5-4440 CPU at 3.10GHz, 16 GiB of RAM and Debian Linux (kernel version 6.4). The release of the R software suite used was 4.3.1. The version of Multistrand software used was 2.1. The number of species and solved reactions were 968 and 648, respectively.

The 24-hour processing time can be divided into roughly two distinct phases: The initial processing phase spans 12 hours and is dedicated to Multistrand's calculations of reaction rates among the sixteen species. Meanwhile, the subsequent 12-hour phase is reserved for simulating the operation of the K-Medoids CRN. The first phase is notably time-consuming, primarily because the K-Medoids CRN algorithm requires the determination of reaction rate values for every combination of input species. However, it's important to note that this phase is exclusively necessary during simulation scenarios. In contrast, when conducting *in vitro* experiments, these reaction rates are intrinsic properties of the input species and do not require additional computation.

Although the number of reactions and species of the K-Medoids iteration is in the same order of magnitude as the CLA Adder circuit presented in Section 4.1.3.5, their simulation time differs significantly, with the K-Medoids CRN simulation taking 12 hours because of the iterative and greedy characteristics of the K-Medoids algorithm, which needs many iterations to converge to a solution. Again, the number of reactions, the iterative design process of circuits, and the actual convergence time of experiments justify the development of tools that can help alleviate low-level details from designers.

In this Section, we proposed, designed, implemented, simulated, and validated the iteration of the K-Medoids Algorithm in generic Chemical Reaction Network (CRN). This implementation shows the feasibility of constructing an unsupervised learning molecular circuit and validates its correctness as a tool to cluster DNA species by their similarity.

Chapter 6

Conclusions and Future Works

6.1 Conclusions

DNA Strand Displacement (DSD) is a promising emerging technology with great potential to be a tool used in various fields of science. Due to its complex and multidisciplinary nature, scientists need tools that provide a high level of abstraction to make designing and experimenting more straightforward, consequently popularizing its use.

This work proposed to explore the proven computing capabilities of DSD technology as a hardware substrate for developing digital, analog, and machine-learning modular molecular circuits.

The first contribution of this work is the DNAr-Logic extension (57, 56) to help the efficient and scalable design of digital molecular circuits, focusing on applying the principles of abstraction and scaling of molecular circuits. The objective of this extension to the DNAr package is to make the construction of digital logic circuits in DSD technology easier and systematically correct. Using DNAr-Logic one can design digital logic circuits without in-depth knowledge of chemical reaction networks and DSD technology. The proposed tool is robust, i.e., it can deal with complex designs programmatically and can abstract and simulate complex digital logic circuits with hundreds of reactions (56), abstracting the user from writing all the chemical reaction networks by hand, i.e., hides all the specific chemical implementation details from the designer.

After the development of DNAr-Logic, the inception DNAr-Analog extension was a natural further development (68). It is a new software package in its initial stages of development that, using the same framework created for DNAr-Logic, compiles analog circuits containing analog gates/units from different works in the literature (67, 69). Analog molecular circuits, compared to their digital counterparts, are more feasible, less costly, and better interface with biological samples in a laboratory. Both packages will automatically handle the specific implementation details to allow the designer to scale the design, i.e., to focus on creating more complex molecular circuits. They are an advancement in the future development of molecular logic, which should impact the development of new technologies.

Finally, this work presented a modular approach to designing a molecular circuit that implements one iteration of the K-Medoids clustering algorithm. This implementation was designed to use nucleic acids (DNA or RNA) both as input of the algorithm and as hardware substrate to perform computation. The implementation also provided a way to control each algorithm phase to perform the computation correctly by using gates that can be turned on or off when needed. Also, the design of the restorer gate to restore the inputs to their initial state to allow a new iteration provides an effective way to reuse the same strands over various iterations. The CRN K-Medoids clustering algorithm can cluster generic nucleic acid strands by its complementarity ratio, signaling that the strands have similar sequences of bases. This correct clustering behavior was demonstrated when, for example, corrupted pictures of numbers encoded in DNA sequences were clustered by their level of similarity. Although the simulation period was very time-consuming, this implementation demonstrates the feasibility of designing modular molecular computers that can iteratively process information using nucleic acid strands both as a hardware substrate and input values.

In conclusion, molecular computing and DNA Strand Displacement (DSD) represent groundbreaking fields at the intersection of biology, chemistry, and computer science, offering immense potential for revolutionizing information processing and diagnostics. These innovative approaches harness DNA molecules' remarkable programmability and specificity to perform complex computational tasks and act as nanoscale switches. The ability to design and precisely manipulate DNA sequences has opened up new avenues for creating highly efficient and parallelized computing systems. Moreover, the inherent parallelism and low energy requirements of molecular computing hold promise for addressing some of the scalability and energy efficiency challenges faced by traditional silicon-based technologies. As researchers continue to refine and expand the capabilities of DNA-based computing, we anticipate exciting advancements that may eventually lead to practical applications in fields such as medicine, biotechnology, and data storage.

However, it is necessary to acknowledge that molecular computing and DSD are still nascent, facing numerous technical hurdles and limitations. Issues like leaks, error rates, scalability, and integrating molecular systems with conventional electronics must be addressed to realize their potential fully. Additionally, ethical considerations surrounding the potential misuse of these technologies and the need for responsible research practices should remain at the forefront of discussions in these fields. Despite these challenges, the progress made thus far underscores the immense promise of molecular computing and DSD as transformative tools in pursuing more efficient and sustainable computing paradigms, offering a glimpse into a future where biology and technology seamlessly merge for the betterment of society.

6.2 Future Works

As molecular computing continues to evolve, researchers are poised to embark on a journey filled with exciting possibilities and challenges. Building upon the foundational knowledge and achievements discussed earlier, the following list outlines some key areas and avenues of future works that hold immense promise for advancing the frontiers of molecular computing and DNA Strand Displacement (DSD):

- Implement the entire K-Medoids algorithm as Chemical Reaction Network (CRN) to run *in vitro*, turning the implementation fully autonomous and independent from electronic computers;
- Develop new technologies to accelerate the simulation of molecular circuits in various levels of abstractions;
- Develop new algorithms and tools to assist the development of molecular computing;
- Publish papers in respected journals with findings;
- Continue collaborative work with the Nanocomp research group;

The number of applications and research projects is uncountable, only subject to the specialists' knowledge and creativity. These endeavors have the potential to not only refine our understanding of these fields but also pave the way for groundbreaking applications that could transform industries and drive innovation in science and technology.

References

1 ADLEMAN, Leonard M. Molecular computation of solutions to combinatorial problems. **Nature**, v. 369, p. 40, 1994.

2 AKYILDIZ, I.; PIEROBON, M.; BALASUBRAMANIAM, S.; KOUCHERYAVY, Y. The internet of Bio-Nano things. **IEEE Communications Magazine**, Institute of Electrical and Electronics Engineers (IEEE), v. 53, n. 3, p. 32–40, Mar. 2015. DOI: 10.1109/mcom.2015.7060516.

3 ALBERTS, B.; BRAY, D.; LEWIS, J.; RAFF, M.; ROBERTS, K.; WATSON, J.D. **Molecular Biology of the Cell**. 4th. [S.l.]: Garland, 2002.

4 ALOISE, D.; DESHPANDE, A.; HANSEN, P.; POPAT, P. NP-hardness of Euclidean sum-of-squares clustering. In: MACHINE learning. [S.l.: s.n.], 2009. P. 245–248.

5 ARVIDSSON, Jakob. **Simulated molecular adder circuits on a surface of DNA: Studying the scalability of surface chemical reaction network digital logic circuits**. 2023. PhD thesis – KTH Royal Institute of Technology.

6 BADELT, Stefan; GRUN, Casey; SARMA, Karthik V.; WOLFE, Brian; SHIN, Seung Woo; WINFREE, Erik. A domain-level DNA strand displacement reaction enumerator allowing arbitrary non-pseudoknotted secondary structures. **Journal of The Royal Society Interface**, v. 17, n. 167, p. 20190866, 2020. DOI: 10.1098/rsif.2019.0866. eprint: https://royalsocietypublishing.org/doi/pdf/10.1098/rsif.2019.0866. Available from: <https://royalsocietypublishing.org/doi/abs/10.1098/rsif.2019.0866>.

7 BADELT, Stefan; SHIN, Seung Woo; JOHNSON, Robert F.; DONG, Qing; THACHUK, Chris; WINFREE, Erik. A General-Purpose CRN-to-DSD Compiler with Formal Verification, Optimization, and Simulation Capabilities. In: LECTURE Notes in Computer Science. [S.l.]: Springer International Publishing, 2017. P. 232–248. DOI: 10.1007/978-3-319-66799-7_15. Available from: <https://doi.org/10.1007/978-3-319-66799-7_15>.

8 BENENSON, Y. Bio molecular computing systems: Principles, progress and potential. **Nat. Rev. Genet.**, v. 7, p. 455–468, 2012. DOI: 10.1038/nrg3197.

9 BENENSON, Yaakov; GIL, Binyamin; BEN-DOR, Uri; ADAR, Rivka; SHAPIRO, Ehud. An autonomous molecular computer for logical control of gene expression. **Nature**, Springer Nature, v. 429, n. 6990, p. 423–429, Apr. 2004. DOI: 10.1038/nature02551.

10 BERLEANT, Joseph; BERLIND, Christopher; BADELT, Stefan; DANNENBERG, Frits; SCHAEFFER, Joseph; WINFREE, Erik. Automated sequence-level analysis of kinetics and thermodynamics for domain-level DNA strand-displacement systems. **Journal of The Royal Society Interface**, The Royal Society, v. 15, n. 149, p. 20180107, Dec. 2018. DOI: 10.1098/rsif.2018.0107. Available from: <https://doi.org/10.1098/rsif.2018.0107>.

11 BETTS, J. Gordon; YOUNG, Kelly A.; WISE, James A.; JOHNSON, Eddie; POE, Brandon; KRUSE, Dean H.; KOROL, Oksana; JOHNSON, Jody E.; WOMBLE, Mark; DESAIX, Peter. **Anatomy and Physiology**. [S.l.: s.n.], 2013. https://openstax.org/books/anatomy-and-physiology/pages/1-introduction. Accessed 26 October 2023. Available from: <https://openstax.org/books/anatomy-and-physiology/pages/1-introduction>.

12 BISSELL, Chris. Historical perspectives - The Moniac A Hydromechanical Analog Computer of the 1950s. **IEEE Control Systems**, Institute of Electrical and Electronics Engineers (IEEE), v. 27, p. 69–74, Feb. 2007. DOI: 10.1109/mcs.2007.284511. Available from: <https://doi.org/10.1109/mcs.2007.284511>.

13 BOHR, M. A 30 Year Retrospective on Dennards MOSFET Scaling Paper. **Solid-State Circuits Society Newsletter, IEEE**, v. 12, n. 1, p. 11–13, 2007. ISSN 1098-4232. DOI: 10.1109/N-SSC.2007.4785534.

14 BORUAH, Kuntala; DUTTA, Jiten Ch. Twenty years of DNA computing: From complex combinatorial problems to the Boolean circuits. In: 2015 International Conference on Electronic Design, Computer Networks & Automated Verification (EDCAV). [S.l.]: IEEE, Jan. 2015. DOI: 10.1109/edcav.2015.7060538.

15 BRAY, D. Protein molecules as computational elements in living cells. **Nature**, v. 376, p. 307–312, 1995. DOI: 10.1038/376307a0.

16 BREAKER, Ronald R. DNA enzymes. **Nature Biotechnology**, Springer Science and Business Media LLC, v. 15, n. 5, p. 427–431, May 1997. DOI: 10.1038/nbt0597-427.

17 BROWN; LEMAY; BUSTEN; MURPHY; WOODWARD. **Reactions in Aqueous Solution**. [S.l.: s.n.], 8 Sept. 2020. https://chem.libretexts.org/@go/page/91163. [Online; accessed 2021-10-11].

18 CAVIN, Ralph K; LUGLI, Paolo; ZHIRNOV, Victor V. Science and engineering beyond Moore's law. **Proceedings of the IEEE**, IEEE, v. 100, Special Centennial Issue, p. 1720–1749, 2012.

19 CHELLABOINA, Vijaysekhar; BHAT, Sanjay P.; HADDAD, Wassim M.; BERNSTEIN, Dennis S. Modeling and analysis of mass-action kinetics. **IEEE Control Systems Magazine**, v. 29, n. 4, p. 60–78, 2009. DOI: 10.1109/MCS.2009.932926.

20 CHEN, Xin; LIU, Xinyu; WANG, Fang; LI, Sirui; CHEN, Congzhou; QIANG, Xiaoli; SHI, Xiaolong. Massively Parallel DNA Computing Based on Domino DNA Strand Displacement Logic Gates. **ACS Synthetic Biology**, American Chemical Society (ACS), v. 11, n. 7, p. 2504–2512, June 2022. DOI: 10.1021/acssynbio.2c00270. Available from: <https://doi.org/10.1021/acssynbio.2c00270>.

21 CHEN, Yuan-Jyue; DALCHAU, Neil; SRINIVAS, Niranjan; PHILLIPS, Andrew; CARDELLI, Luca; SOLOVEICHIK, David; SEELIG, Georg. Programmable chemical controllers made from DNA. **Nature nanotechnology**, Nature Publishing Group, v. 8, n. 10, p. 755, 2013.

22 CHERRY, Kevin M.; QIAN, Lulu. Scaling up molecular pattern recognition with DNA-based winner-take-all neural networks. **Nature**, Springer Nature, July 2018. DOI: 10.1038/s41586-018-0289-6.

23 CHIRE. **Cluster analysis performed on an artificial dataset.** [S.l.: s.n.], 2010. https://commons.wikimedia.org/wiki/File:ClusterAnalysis_Mouse.svg. Accessed 10 October 2021. Available from: <https://commons.wikimedia.org/wiki/File:ClusterAnalysis_Mouse.svg>.

24 CLAMONS, Samuel; QIAN, Lulu; WINFREE, Erik. Programming and simulating chemical reaction networks on a surface. **Journal of the Royal Society Interface**, The Royal Society, v. 17, n. 166, p. 20190790, 2020.

25 COHEN, E.R.; CVITAS, T.; FREY, Jeremy; HOLMSTROM, B.; KUCHITSU, K.; MARQUARDT, R.; MILLS, I.; PAVESE, Franco; QUACK, Martin; STOHNER, Jürgen; STRAUSS, Herbert; TAKAMI, M.; THOR, A.J. **Quantities, Units and Symbols in Physical Chemistry, IUPAC Green Book**. [S.l.: s.n.], Jan. 2007. ISBN 978-0-85404-433-7.

26 COURBET, Alexis; MOLINA, Franck; AMAR, Patrick. Computing with Synthetic Protocells. English. **Acta Biotheoretica**, Springer Nature, v. 63, n. 3, p. 309–323, May 2015. ISSN 0001-5342. DOI: 10.1007/s10441-015-9258-8.

27 DENNARD, R.H.; GAENSSLEN, F.H.; YU, Hwa-Nien; RIDEOUT, V.L.; BASSOUS, E.; LEBLANC, A.R. Design Of Ion-implanted MOSFET's with Very Small Physical Dimensions. **Proceedings of the IEEE**, Institute of Electrical and Electronics Engineers (IEEE), v. 87, n. 4, p. 668–678, Apr. 1999. DOI: 10.1109/jproc.1999.752522. Available from: <https://doi.org/10.1109/jproc.1999.752522>.

28 DENNARD, Robert H.; GAENSSLEN, Fritz H.; YU, Hwa-nien; RIDEOUT, V. Leo; BASSOUS, Ernest; ANDRE; LEBLANC, R. Design of ion-implanted MOSFETs with very small physical dimensions. **IEEE J. Solid-State Circuits**, p. 256, 1974.

29 ESTER, Martin; KRIEGEL, Hans-Peter; SANDER, Jörg; XU, Xiaowei. A density-based algorithm for discovering clusters in large spatial databases with noise. In: p. 226–231.

30 EWING, M. B.; LILLEY, T. H.; OLOFSSON, G. M.; RATZSCH, M. T.; SOMSEN, G. Standard quantities in chemical thermodynamics. Fugacities, activities and equilibrium constants for pure and mixed phases (IUPAC Recommendations 1994). **Pure and Applied Chemistry**, Walter de Gruyter GmbH, v. 66, n. 3, p. 533–552, Jan. 1994. DOI: 10.1351/pac199466030533. Available from: <https://doi.org/10.1351/pac199466030533>.

31 FAGES, François; GULUDEC, Guillaume Le; BOURNEZ, Olivier; POULY, Amaury. Strong Turing Completeness of Continuous Chemical Reaction Networks and Compilation of Mixed Analog-Digital Programs. In: COMPUTATIONAL Methods in Systems Biology. [S.l.]: Springer International Publishing, 2017. P. 108–127. DOI: 10.1007/978-3-319-67471-1_7.

32 FAN, Daoqing; WANG, Juan; WANG, Erkang; DONG, Shaojun. Propelling DNA Computing with Materials' Power: Recent Advancements in Innovative DNA Logic Computing Systems and Smart Bio-Applications. **Advanced Science**, Wiley, v. 7, n. 24, p. 2001766, Nov. 2020. DOI: 10.1002/advs.202001766. Available from: <https://doi.org/10.1002/advs.202001766>.

33 FEINBERG, M. **Lectures on chemical reaction networks. Notes of lectures given at the Mathematics Research Center, University of Wisconsin**, 1979.

34 FREETH, T.; BITSAKIS, Y.; MOUSSAS, X.; SEIRADAKIS, J. H.; TSELIKAS, A.; MANGOU, H.; ZAFEIROPOULOU, M.; HADLAND, R.; BATE, D.; RAMSEY, A.; ALLEN, M.; CRAWLEY, A.; HOCKLEY, P.; MALZBENDER, T.; GELB, D.; AMBRISCO, W.; EDMUNDS, M. G. Decoding the ancient Greek astronomical calculator known as the Antikythera Mechanism. **Nature**, Springer Science and Business Media LLC, v. 444, n. 7119, p. 587–591, Nov. 2006. DOI:

10.1038/nature05357. Available from:
<https://doi.org/10.1038/nature05357>.

35 GE, Lulu; ZHANG, Chuan; ZHONG, Zhiwei; YOU, Xiaohu. A formal design methodology for synthesizing a clock signal with an arbitrary duty cycle of M/N. In: 2015 IEEE Workshop on Signal Processing Systems (SiPS). [S.l.]: IEEE, Oct. 2015. DOI: 10.1109/sips.2015.7344975. Available from:
<https://doi.org/10.1109/sips.2015.7344975>.

36 HIGGINS, W. H. C.; HOLBROOK, B. D.; EMLING, J. W. Electrical Computers for Fire Control. In: ANNALS of the History of Computing. [S.l.: s.n.], July 1982. v. 4. P. 218–246.

37 HINTON, Geoffrey; SEJNOWSKI, Terrence. **Unsupervised Learning: Foundations of Neural Computation**. [S.l.]: MIT Press, 1999. ISBN 978-0262581684.

38 HOLLERITH, Herman. **US patent 395782 - Art of compiling statistics**. [S.l.: s.n.], 8 Jan. 1889.
https://worldwide.espacenet.com/patent/search?q=pn%3DUS395782. [Online; accessed 2022-06-25].

39 HOOPS, Stefan; SAHLE, Sven; GAUGES, Ralph; LEE, Christine; PAHLE, Jürgen; SIMUS, Natalia; SINGHAL, Mudita; XU, Liang; MENDES, Pedro; KUMMER, Ursula. COPASI–a complex pathway simulator. **Bioinformatics**, Oxford University Press, v. 22, n. 24, p. 3067–3074, 2006.

40 HSU, W.; NEMHAUSER., G. Easy and hard bottleneck location problems. In: DISCRETE Applied Mathematics. [S.l.: s.n.], 1979. P. 209–215.

41 JAIN, A. Data clustering: 50 years beyond k-means. In: PATTERN recognition letters. [S.l.: s.n.], 2010. P. 651–666.

42 JIANG, Hua; RIEDEL, Marc D.; PARHI, Keshab K. Digital logic with molecular reactions. In: 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). [S.l.]: IEEE, Nov. 2013. DOI: 10.1109/iccad.2013.6691194.

43 KAUFMAN, Leonard; ROUSSEEUW, Peter J. **Finding Groups in Data - An Introduction to Cluster Analysis**. [S.l.]: John Wiley and Sons, 1990. ISBN 0471735787.

44 KERANEN, Rachel. **Inventions in Computing: From the Abacus to Personal Computers**. [S.l.]: Cavendish Square Publishing, LLC, 2016. pp 41-43.

45 LAIDLER, Keith J. **Chemical kinetics, Encyclopedia Britannica**. [S.l.: s.n.], 2021. https://www.britannica.com/science/chemical-kinetics. Accessed 10 October 2021. Available from:
<https://www.britannica.com/science/chemical-kinetics>.

46 LAKIN, M. R.; MINNICH, A.; LANE, T.; STEFANOVIC, D. Design of a biochemical circuit motif for learning linear functions. **Journal of The Royal Society Interface**, The Royal Society, v. 11, n. 101, p. 20140902–20140902, Oct. 2014. DOI: 10.1098/rsif.2014.0902.

47 LAKIN, M. R.; YOUSSEF, S.; CARDELLI, L.; PHILLIPS, A. Abstractions for DNA circuit design. **Journal of The Royal Society Interface**, The Royal Society, v. 9, n. 68, p. 470–486, July 2011. DOI: 10.1098/rsif.2011.0343.

48 LAKIN, Matthew R.; STEFANOVIC, Darko. Supervised Learning in Adaptive DNA Strand Displacement Networks. **ACS Synthetic Biology**, American Chemical Society (ACS), v. 5, n. 8, p. 885–897, May 2016. DOI: 10.1021/acssynbio.6b00009.

49 LAKIN, Matthew R.; YOUSSEF, Simon; POLO, Filippo; EMMOTT, Stephen; PHILLIPS, Andrew. Visual DSD: a design and analysis tool for DNA strand displacement systems. **Bioinformatics**, v. 27, n. 22, p. 3211–3213, Oct. 2011. ISSN 1367-4803. DOI: 10.1093/bioinformatics/btr543. eprint: https://academic.oup.com/bioinformatics/article-pdf/27/22/3211/490674/btr543.pdf. Available from: <https://doi.org/10.1093/bioinformatics/btr543>.

50 LEE, Byoungsang; AHN, So Yeon; PARK, Charles; MOON, James J.; LEE, Jung Heon; LUO, Dan; UM, Soong Ho; SHIN, Seung Won. Revealing the Presence of a Symbolic Sequence Representing Multiple Nucleotides Based on K-Means Clustering of Oligonucleotides. **Molecules**, MDPI AG, v. 24, n. 2, p. 348, Jan. 2019. DOI: 10.3390/molecules24020348.

51 LIBRETEXTS, Chemistry. **The Equilibrium Constant**. [S.l.: s.n.], 2021. https://chem.libretexts.org/@go/page/1362. Accessed 11 October 2021. Available from: <https://chem.libretexts.org/@go/page/1362>.

52 ________. **The Rate Law**. [S.l.: s.n.], 8 Sept. 2020. https://chem.libretexts.org/@go/page/1423. [Online; accessed 2021-10-11].

53 LIU, Xingyi; PARHI, Keshab K. DNA Memristors and Their Application to Reservoir Computing. **ACS Synthetic Biology**, American Chemical Society (ACS), v. 11, n. 6, p. 2202–2213, May 2022. DOI: 10.1021/acssynbio.2c00184. Available from: <https://doi.org/10.1021/acssynbio.2c00184>.

54 MACKAY, David. **Information Theory, Inference and Learning Algorithms**. [S.l.]: Cambridge University Press, 2003. Chapter 20. An Example Inference Task: Clustering, p. 284–292. ISBN 978-0-521-64298-9.

55 MAGNASCO, Marcelo O. Chemical Kinetics is Turing Universal. **Physical Review Letters**, American Physical Society (APS), v. 78, n. 6, p. 1190–1193, Feb. 1997. DOI: `10.1103/physrevlett.78.1190`. Available from: <`https://doi.org/10.1103/physrevlett.78.1190`>.

56 MARKS, Renan A.; VIEIRA, Daniel K. S.; GUTERRES, Marcos V.; OLIVEIRA, Poliana A. C.; BOA, Maria C. O. Fonte; NETO, Omar P. Vilela. Design and Test of Digital Logic DNA Systems. **IEEE Design & Test**, Institute of Electrical and Electronics Engineers (IEEE), v. 38, n. 4, p. 94–101, Aug. 2021. DOI: `10.1109/mdat.2021.3069369`.

57 MARKS, Renan A.; VIEIRA, Daniel K. S.; GUTERRES, Marcos V.; OLIVEIRA, Poliana A. C.; NETO, Omar P. Vilela. DNAr-logic: a constructive DNA logic circuit designlibrary in R language for molecular computing. In: 32ND Symposium on Integrated Circuits and Systems Design - SBCCI '19. [S.l.]: ACM Press, 2019. DOI: `10.1145/3338852.3339854`.

58 MARQUARDT, Roberto; MEIJA, Juris; MESTER, Zoltán; TOWNS, Marcy; WEIR, Ron; DAVIS, Richard; STOHNER, Jürgen. Definition of the mole (IUPAC Recommendation 2017). **Pure and Applied Chemistry**, Walter de Gruyter GmbH, v. 90, n. 1, p. 175–180, Jan. 2018. DOI: `10.1515/pac-2017-0106`. Available from: <`https://doi.org/10.1515/pac-2017-0106`>.

59 MEDIAWIKI CONTRIBUTOR. **Antikythera Mechanism.** [S.l.: s.n.], 19 Dec. 2005. `https://commons.wikimedia.org/wiki/File:NAMA_Machine_d%27Anticyth%C3%A8re_1.jpg`. [Online; accessed 2022-06-25].

60 MITCHELL, T. M. **Machine Learning**. [S.l.: s.n.], 1997.

61 MOORE, G. Cramming more components onto integrated circuits. v. 38, n. 289, p. 114–117, 1965.

62 MUSEUM, Computer History. **The Babbage Engine: The Engines.** [S.l.: s.n.], 25 June 2022. `https://www.computerhistory.org/babbage/engines/`. [Online; accessed 2022-06-25].

63 NETO, Omar P. Vilela; OLIVEIRA, Poliana A. C. de; MARKS, Renan A.; NOVY, Gabriel; VIEIRA, Maria; LUZ, Laysson Oliveira; SILVA, Pedro Arthur; FERREIRA, Ricardo; RAMIREZ, Jhonattan; NACIF, José Augusto; CAPORALI, Guilherme. Design Automation for Emerging Technologies. **Journal of Integrated Circuits and Systems**, Journal of Integrated Circuits and Systems, v. 17, n. 3, p. 1–11, Dec. 2022. DOI: `10.29292/jics.v17i3.652`. Available from: <`https://doi.org/10.29292/jics.v17i3.652`>.

64 NEUMANN, J. von. First draft of a report on the EDVAC. **IEEE Annals of the History of Computing**, Institute of Electrical and Electronics Engineers (IEEE), v. 15, n. 4, p. 27–75, 1993. DOI: 10.1109/85.238389. Available from: <https://doi.org/10.1109/85.238389>.

65 NGAMDEE, Tatchanun; YIN, Lee Su; VONGPUNSAWAD, Sompong; POOVORAWAN, Yong; SURAREUNGCHAI, Werasak; LERTANANTAWONG, Benchaporn. Target Induced-DNA strand displacement reaction using gold nanoparticle labeling for hepatitis E virus detection. **Analytica Chimica Acta**, Elsevier BV, v. 1134, p. 10–17, Oct. 2020. DOI: 10.1016/j.aca.2020.08.018.

66 NOH, Yung-Kyun; LEE, Daniel D.; YANG, Kyung-Ae; KIM, Cheongtag; ZHANG, Byoung-Tak. Molecular learning with DNA kernel machines. **Biosystems**, Elsevier BV, v. 137, p. 73–83, Nov. 2015. DOI: 10.1016/j.biosystems.2015.06.007.

67 OLIVEIRA, Poliana A. C.; BOA, Maria C. O. Fonte; MARKS, Renan A.; GUTERRES, Marcos V.; NETO, Omar P. Vilela. Analysis of single-module and cascade molecular analog circuits for approximate computing based on DNA Strand Displacement. In: IEEE. 2020 33rd Symposium on Integrated Circuits and Systems Design (SBCCI). [S.l.: s.n.], 2020. P. 1–6.

68 OLIVEIRA, Poliana A. C.; MARKS, Renan A.; TEIXEIRA, João V. C.; GUTERRES, Marcos V.; NETO, Omar P. Vilela. DNAr-Analog: A Library With a Multiplexer to Easily Design, Program, and Simulate Dsd Analog Circuits. In: 2023 IEEE 14th Latin America Symposium on Circuits and Systems (LASCAS). [S.l.]: IEEE, Feb. 2023. DOI: 10.1109/lascas56464.2023.10108135. Available from: <https://doi.org/10.1109/lascas56464.2023.10108135>.

69 OLIVEIRA, Poliana A. C.; TEIXEIRA, João V. C.; MARKS, Renan A.; GUTERRES, Marcos V.; NETO, Omar P. Vilela. Evaluating analog arithmetic circuit for approximate computing with DNA strand displacement. **Analog Integrated Circuits and Signal Processing**, Springer Science and Business Media LLC, v. 108, n. 3, p. 485–493, May 2021. DOI: 10.1007/s10470-021-01828-9. Available from: <https://doi.org/10.1007/s10470-021-01828-9>.

70 PING, Zhi; MA, Dongzhao; HUANG, Xiaoluo; CHEN, Shihong; LIU, Longying; GUO, Fei; ZHU, Sha Joe; SHEN, Yue. Carbon-based archiving: current progress and future prospects of DNA-based data storage. **GigaScience**, Oxford University Press (OUP), v. 8, n. 6, June 2019. DOI: 10.1093/gigascience/giz075. Available from: <https://doi.org/10.1093/gigascience/giz075>.

71 QIAN, L.; WINFREE, E. A simple DNA gate motif for synthesizing large-scale circuits. **Journal of The Royal Society Interface**, The Royal Society, v. 8, n. 62, p. 1281–1297, Feb. 2011. DOI: 10.1098/rsif.2010.0729.

72 QIAN, Lulu; WINFREE, Erik. Scaling Up Digital Circuit Computation with DNA Strand Displacement Cascades. **Science**, American Association for the Advancement of Science (AAAS), v. 332, n. 6034, p. 1196–1201, June 2011. DOI: 10.1126/science.1200520.

73 QIAN, Lulu; WINFREE, Erik; BRUCK, Jehoshua. Neural network computation with DNA strand displacement cascades. **Nature**, Springer Nature, v. 475, n. 7356, p. 368–372, July 2011. DOI: 10.1038/nature10262.

74 REN, Yubin; ZHANG, Yi; LIU, Yawei; WU, Qinglin; SU, Juanjuan; WANG, Fan; CHEN, Dong; FAN, Chunhai; LIU, Kai; ZHANG, Hongjie. DNA-Based Concatenated Encoding System for High-Reliability and High-Density Data Storage. **Small Methods**, Wiley, v. 6, n. 4, p. 2101335, Feb. 2022. DOI: 10.1002/smtd.202101335. Available from: <https://doi.org/10.1002/smtd.202101335>.

75 ROUSSEEUW, Peter J. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. **Journal of Computational and Applied Mathematics**, v. 20, p. 53–65, 1987. ISSN 0377-0427. DOI: https://doi.org/10.1016/0377-0427(87)90125-7. Available from: <https://www.sciencedirect.com/science/article/pii/0377042787901257>.

76 RUSSELL, Stuart J.; NORVIG, Peter. **Artificial Intelligence: A Modern Approach**. [S.l.]: Prentice Hall, 2010. ISBN 9780136042594.

77 S., Ji. The cell as the smallest DNA-based molecular computer. **Biosystems**, v. 52, p. 123–133, 1999. DOI: 10.1016/s0303-2647(99)00039-8.

78 SCHAEFFER, Joseph Malcolm; THACHUK, Chris; WINFREE, Erik. Stochastic Simulation of the Kinetics of Multiple Interacting Nucleic Acid Strands. In: LECTURE Notes in Computer Science. [S.l.]: Springer International Publishing, 2015. P. 194–211. DOI: 10.1007/978-3-319-21999-8_13. Available from: <https://doi.org/10.1007/978-3-319-21999-8_13>.

79 SCHUBERT, Erich; ROUSSEEUW, Peter J. Faster k-Medoids Clustering: Improving the PAM, CLARA, and CLARANS Algorithms. In: [s.l.]: Springer International Publishing, 2019. P. 171–187. DOI: 10.1007/978-3-030-32047-8_16. Available from: <https://doi.org/10.1007/978-3-030-32047-8_16>.

80 SHALF, John M; LELAND, Robert. Computing beyond Moore's Law. **Computer**, Institute of Electrical and Electronics Engineers (IEEE), v. 48, n. 12, p. 14–23, Dec. 2015. DOI: 10.1109/mc.2015.374.

81 SHANNON, Claude E. A symbolic analysis of relay and switching circuits. **Electrical Engineering**, IEEE, v. 57, n. 12, p. 713–723, 1938.

82 SIMMEL, Friedrich C; YURKE, Bernard; SINGH, Hari R. Principles and Applications of Nucleic Acid Strand Displacement Reactions. **Chemical reviews**, ACS Publications, 2019.

83 SOLOVEICHIK, D.; SEELIG, G.; WINFREE, E. DNA as a universal substrate for chemical kinetics. **Proceedings of the National Academy of Sciences**, Proceedings of the National Academy of Sciences, v. 107, n. 12, p. 5393–5398, Mar. 2010. DOI: 10.1073/pnas.0909380107.

84 SONG, Tianqi; GARG, Sudhanshu; MOKHTAR, Reem; BUI, Hieu; REIF, John. Analog computation by DNA strand displacement circuits. **ACS synthetic biology**, ACS Publications, v. 5, n. 8, p. 898–912, 2016.

85 SRINIVAS, Niranjan; PARKIN, James; SEELIG, Georg; WINFREE, Erik; SOLOVEICHIK, David. Enzyme-free nucleic acid dynamical systems. **Science**, American Association for the Advancement of Science (AAAS), v. 358, n. 6369, eaal2052, Dec. 2017. DOI: 10.1126/science.aal2052.

86 STERNER, R. W.; SMALL, G. E.; HOOD, J. M. **The Conservation of Mass. Nature E-ducation Knowledge 3(10):20**. [S.l.: s.n.], 2011. https://www.nature.com/scitable/knowledge/library/the-conservation-of-mass-17395478/. Accessed 10 October 2021. Available from: <https://www.nature.com/scitable/knowledge/library/the-conservation-of-mass-17395478/>.

87 TREICHEL, Paul M.; KOTZ, John C. **Chemical Reaction, Encyclopedia Britannica**. [S.l.: s.n.], 2021. https://www.britannica.com/science/chemical-reaction. Accessed 10 October 2021. Available from: <https://www.britannica.com/science/chemical-reaction>.

88 VENDRAMIN, Lucas; CAMPELLO, Ricardo J. G. B.; HRUSCHKA, Eduardo R. On the Comparison of Relative Clustering Validity Criteria. In: PROCEEDINGS of the 2009 SIAM International Conference on Data Mining. [S.l.]: Society for Industrial and Applied Mathematics, Apr. 2009. DOI: 10.1137/1.9781611972795.63.

89 VIEIRA, Daniel K. S.; GUTERRES, Marcos V.; MARKS, Renan A.; OLIVEIRA, Poliana A. C.; BOA, Maria C. O. Fonte; NETO, Omar P. Vilela. DNAr: An R Package to Simulate and Analyze CRN and DSD Networks. **ACS Synthetic Biology**, American Chemical Society (ACS), Dec. 2020. DOI: 10.1021/acssynbio.0c00364.

90 VIEIRA, Daniel Kneipp de Sá. **Design de circuitos lógicos baseados em DNA visando a síntese de sistemas computacionais**. 2018. MA thesis.

91 WANG, Chenyang; MA, Guannan; WEI, Di; ZHANG, Xinru; WANG, Peihan; LI, Cuidan; XING, Jing; WEI, Zheng; DUAN, Bo; YANG, Dongxin; WANG, Pei; BU, Dongbo; CHEN, Fei. Mainstream encoding–decoding methods of DNA data storage. **CCF Transactions on High Performance Computing**, Springer Science and Business Media LLC, v. 4, n. 1, p. 23–33, Mar. 2022. DOI: 10.1007/s42514-022-00094-z. Available from: <https://doi.org/10.1007/s42514-022-00094-z>.

92 WANG, Yanfeng; MAO, Tongtong; SUN, Junwei; LIU, Peng. Exponential Function Computation Based on DNA Strand Displacement Circuits. **IEEE Transactions on Biomedical Circuits and Systems**, IEEE, v. 16, n. 3, p. 479–488, 2022.

93 WANG, Zhao-Cai; LIANG, Kun; BAO, Xiao-Guang; WU, Tun-Hua. A novel Algorithm for Solving the Prize Collecting Traveling Salesman Problem based on DNA Computing. **IEEE Transactions on NanoBioscience**, Institute of Electrical and Electronics Engineers (IEEE), p. 1–1, 2023. DOI: 10.1109/tnb.2023.3307458. Available from: <https://doi.org/10.1109/tnb.2023.3307458>.

94 WEN, Donglin; GE, Lulu; LU, Yuxiang; ZHANG, Chuan; YOU, Xiaohu. A DNA strand displacement reaction implementation-friendly clock design. In: 2017 IEEE International Conference on Communications (ICC). [S.l.]: IEEE, May 2017. DOI: 10.1109/icc.2017.7997035. Available from: <https://doi.org/10.1109/icc.2017.7997035>.

95 WOLF, J. Jay. The Office of Naval Research Relay Computer. **Mathematics of Computation**, American Mathematical Society (AMS), v. 6, n. 40, p. 207–212, 1952. DOI: 10.1090/s0025-5718-1952-0050393-0. Available from: <https://doi.org/10.1090/s0025-5718-1952-0050393-0>.

96 WU, Xian; WANG, Zhaocai; WU, Tunhua; BAO, Xiaoguang. Solving the Family Traveling Salesperson Problem in the Adleman–Lipton Model Based on DNA Computing. **IEEE Transactions on NanoBioscience**, Institute of Electrical and Electronics Engineers (IEEE), v. 21, n. 1, p. 75–85, Jan. 2022. DOI: 10.1109/tnb.2021.3109067. Available from: <https://doi.org/10.1109/tnb.2021.3109067>.

97 XIONG, Xiewei; ZHU, Tong; ZHU, Yun; CAO, Mengyao; XIAO, Jin; LI, Li; WANG, Fei; FAN, Chunhai; PEI, Hao. Molecular convolutional neural networks with DNA regulatory circuits. **Nature Machine Intelligence**, Springer Science and Business Media LLC, v. 4, n. 7, p. 625–635, July 2022. DOI:

10.1038/s42256-022-00502-7. Available from: <https://doi.org/10.1038/s42256-022-00502-7>.

98 XU, Rui; II, Donald C. Wunsch. **Clustering**. [S.l.]: John Wiley and Sons, 2009. ISBN 9780470276808.

99 YORDANOV, Boyan; KIM, Jongmin; PETERSEN, Rasmus L; SHUDY, Angelina; KULKARNI, Vishwesh V; PHILLIPS, Andrew. Computational design of nucleic acid feedback control circuits. **ACS synthetic biology**, ACS Publications, v. 3, n. 8, p. 600–616, 2014.

100 YURKE, Bernard; TURBERFIELD, Andrew J; MILLS JR, Allen P; SIMMEL, Friedrich C; NEUMANN, Jennifer L. A DNA-fuelled molecular machine made of DNA. **Nature**, Nature Publishing Group, v. 406, n. 6796, p. 605, 2000.

101 ZADEH, Joseph N.; STEENBERG, Conrad D.; BOIS, Justin S.; WOLFE, Brian R.; PIERCE, Marshall B.; KHAN, Asif R.; DIRKS, Robert M.; PIERCE, Niles A. NUPACK: Analysis and design of nucleic acid systems. **Journal of Computational Chemistry**, Wiley, v. 32, n. 1, p. 170–173, Nov. 2010. DOI: 10.1002/jcc.21596. Available from: <https://doi.org/10.1002/jcc.21596>.

102 ZHANG, Chao; ZHAO, Yumeng; XU, Xuemei; XU, Rui; LI, Haowen; TENG, Xiaoyan; DU, Yuzhen; MIAO, Yanyan; LIN, Hsiao-chu; HAN, Da. Cancer diagnosis with DNA molecular computation. **Nature Nanotechnology**, Springer Science and Business Media LLC, v. 15, n. 8, p. 709–715, May 2020. DOI: 10.1038/s41565-020-0699-0. Available from: <https://doi.org/10.1038/s41565-020-0699-0>.

103 ZHANG, David Yu; SEELIG, Georg. Dynamic DNA nanotechnology using strand-displacement reactions. **Nature chemistry**, Nature Publishing Group, v. 3, n. 2, p. 103, 2011.

104 ________. Dynamic DNA nanotechnology using strand-displacement reactions. Springer Science and Business Media LLC, v. 3, n. 2, p. 103–113, Jan. 2011. DOI: 10.1038/nchem.957. Available from: <https://doi.org/10.1038/nchem.957>.

105 ZHONG, Zhiwei; GE, Lulu; SHEN, Ziyuan; YOU, Xiaohu; ZHANG, Chuan. CRN-based design methodology for synchronous sequential logic. In: 2017 IEEE International Workshop on Signal Processing Systems (SiPS). [S.l.]: IEEE, Oct. 2017. DOI: 10.1109/sips.2017.8109979. Available from: <https://doi.org/10.1109/sips.2017.8109979>.

106 ZOU, Chengye; WEI, Xiaopeng; ZHANG, Qiang; LIU, Chanjuan; LIU, Yuan. Solution of equations based on analog DNA strand displacement circuits. **IEEE Transactions on Nanobioscience**, IEEE, v. 18, n. 2, p. 191–204, 2019.

107 ZOU, Chengye; WEI, Xiaopeng; ZHANG, Qiang; LIU, Chanjuan; ZHOU, Changjun; LIU, Yuan. Four-Analog Computation Based on DNA Strand Displacement. **ACS Omega**, American Chemical Society (ACS), v. 2, n. 8, p. 4143–4160, Aug. 2017. DOI: 10.1021/acsomega.7b00572.

Appendix A

Gate Implementation Details

A.1 Counter Builder Gate

The Counter Builder Gate (CB) is responsible for releasing inactive (i.e., buffered) Counter Gates (C) for the specific input species that will act as medoids in an iteration of the algorithm. They are represented as yellow boxes marked as (a) in Figure 5.3. In the process of releasing the Counter Gates (C), the Builder Gate (CB) inactivates (i.e., buffers) the medoid strands to prevent them from reacting to the activated Counter Gates (C). This inactivation is necessary to maintain the species in the solution to be used in the next iteration of the algorithm and prevent them from being "counted" as if they were non-medoids.

$$\mathrm{BA} + \mathrm{BCB_X} \longrightarrow \mathrm{CB_X} \tag{A.1a}$$

$$\mathrm{CB_X} + \mathrm{BDA} \longrightarrow \mathrm{BCB_X} \tag{A.1b}$$

$$\mathrm{CB_X} + \mathrm{X} \longrightarrow \mathrm{I_X} + \mathrm{CB_X} + \mathrm{BC_X} \tag{A.1c}$$

Equations A.1 show the reactions needed to implement the Counter Builder (CB) gate. Equations A.1a and A.1b show the unbuffering and buffering reactions, respectively. Equation A.1c show the iteraction between the gate (CB) with the input (X) , which inactivate it ($\mathrm{I_X}$) and release the buffered Counter (BC) Gate. Table A.1 describes each species acronyms used in Equation A.1.

A.2 Counter Gate

The Counter Gates (C) are responsible for literally "counting," i.e., reacting the non-medoid strands with the selected medoid strand, how many non-medoid strands are in the solution. They are represented as blue boxes marked as (b) in Figure 5.3. In the

Table A.1: Table describing each species acronyms used in Equation A.1 for Counter Builder (CB) Gate.

Species	Description
BA	Buffer activator (for CB gate)
BCB_X	Buffered Counter Builder
CB_X	Counter Builder
X	Input species of the gate
BC_X	Buffered(inactive) Counter Gate for X species (input)
I_X	Inactivated species X (for use in later iterations)
BDA	Buffer deactivator (for CB gate)

process, the Counter Gates inactivates (i.e., buffers) the non-medoid strands and produces report strands used as input in the Join Gates to generate the mistrust signal.

$$BAC + BC_X \longrightarrow C_X \tag{A.2a}$$

$$C_X + BDC \longrightarrow BC_X \tag{A.2b}$$

$$C_X + S \longrightarrow C_X + I_S + Rep \tag{A.2c}$$

Equation A.2 shows the reactions needed to implement the Counter (C) gate. Equations A.2a and A.2b show the unbuffering and buffering reactions, respectively. Equation A.2c show the iteraction between the gate (C) with the input (S) , which inactivate it (I_S) and release the reporter signal (Rep). Table A.2 describes each species acronyms used in Equation A.2.

Table A.2: Table describing each species acronyms used in Equation A.2 for Counter (C) Gate.

Species	Description
BAC	Buffer activator (for Counter Gate)
BDC	Buffer deactivator (for Counter Gate)
BC_X	Buffered(inactive) Counter Gate for X
C_X	Unbuffered(active) Counter Gate for X
S	Strand of DNA to be counted/clustered
I_S	Strands of DNA witch have been counted/clustered; S in "inactive" state (may be used later in other gates)
Rep	Strand of DNA only used to report counting

A.3 Join Gate

The Join Gates (J) are responsible for generating the analog Mistrust Signal. The Mistrust Signal represents the level of mistrust of the classification provided by the Counter (C) Gates. When the trust in classification by Counter (C) Gates is high (in other words, classified species have a high affinity for only one medoid), then the mistrust signal concentration is low. Conversely, when the trust in classification is low, i.e., the classified species have an equal affinity for more than one medoid, the mistrust signal concentration is high. The Join Gates are represented as orange boxes marked as (c) in Figure 5.5. The gate works as an analog subtraction operation: if both inputs have the same concentration, the output is zero; if inputs have significantly different concentrations, the output concentration is high. The higher the concentration difference from the inputs, the higher the output concentration.

$$\mathrm{BA} + \mathrm{BJ_{XY}} \longrightarrow \mathrm{J_{XY}} \tag{A.3a}$$

$$\mathrm{J_{XY}} + \mathrm{BDA} \longrightarrow \mathrm{BJ_{XY}} \tag{A.3b}$$

$$\mathrm{J_{XY}} + \mathrm{X} \longleftrightarrow \mathrm{T_{XY}} \tag{A.3c}$$

$$\mathrm{T_{XY}} + \mathrm{Y} \longrightarrow 2\,\mathrm{Mis_{XY}} \tag{A.3d}$$

Equation A.3 shows the reactions needed to implement the Join (J) gate. Equations A.3a and A.3b show the unbuffering and buffering reactions, respectively. Equation A.3c show the iteraction between the gate (J) with the input X, releasing a intermediary signal $\mathrm{T_{XY}}$. Equation A.3d show the intermediary signal $\mathrm{T_{XY}}$ interacting with the input Y which release two copies of the Mistrust signal ($\mathrm{Mis_{XY}}$). These copies denote that one strand of X and one strand of Y were present in the solution, contabilizing both strands. The bimolecular equations A.3c and A.3d are needed to allow the 4-Domain (83) encoding of the equivalent equation $\mathrm{J_{XY}} + \mathrm{X} + \mathrm{Y} \longrightarrow 2\,\mathrm{Mis_{XY}}$. Table A.3 describes each species acronyms used in Equation A.3.

Table A.3: Table describing each species acronyms used in Equation A.3 for Join (J) Gate.

Species	Description
BA	Buffer activator (for $\mathrm{J_{XY}}$ gate)
$\mathrm{BJ_{XY}}$	Buffered Join Gate for X and Y inputs
$\mathrm{J_{XY}}$	Join Gate for X and Y inputs
$\mathrm{T_{XY}}$	Auxiliary species to allow the 4-Domain encoding.
$\mathrm{Mis_{XY}}$	Analog signal for mistrust between input signals
BDA	Buffer deactivator (for $\mathrm{J_{XY}}$ gate)

A.4 Restorer Gate

The Restorer Gates (R) are responsible for reverting the inactivation (i.e., buffering) from the species of this iteration. The Restorer Gates are represented as green boxes marked as (d) in Figure 5.6. The Restorer Gates (R) take the inactivated species by Counter Builder (CB) and Counter (C) Gates and reactivate (i.e., unbuffers) the species to be used in the next iteration. The Restorer Gates (R) also clean the not used leftovers inputs from Join (J) Gate.

$$\mathrm{BAR} + \mathrm{BR} \longrightarrow \mathrm{R} \quad \text{(A.4a)}$$

$$\mathrm{R} + \mathrm{BDR} \longrightarrow \mathrm{BR} \quad \text{(A.4b)}$$

$$\mathrm{I_S} + \mathrm{R} \longrightarrow \mathrm{S} + \mathrm{R} \quad \text{(A.4c)}$$

$$\mathrm{Mistrust} + \mathrm{R} \longrightarrow \mathrm{R} \quad \text{(A.4d)}$$

Equation A.4 shows the reactions needed to implement the Restorer (R) gate. Equations A.4a and A.4b show the unbuffering and buffering reactions, respectively. Equation A.4c show the iteraction between the gate (R) with the input ($\mathrm{I_S}$), which unbuffers it releasing the activated species S. Equation A.4d show the elimination of leftovers of reporter signals generated by the Join (J) Gates. Table A.4 describes each species acronyms used in Equation A.4.

Table A.4: Table describing each species acronyms used in Equation A.4 for Restorer (R) Gate.

Species	Description
BAR	Buffer activator (for Restorer Gate)
BDR	Buffer deactivator (for Restorer Gate)
BR	Buffered(inactive) Restorer Gate
R	Unbuffered(active) Restorer Gate
S	Unbuffered Species (DNA) that was counted
$\mathrm{C_S}$	Buffered Species, i.e., counted, S in “inactive” state
Mistrust	Output reporter analog signal

Appendix B

Publications

All the papers this work has contributed to are listed below:

- MARKS, Renan A.; VIEIRA, Daniel K. S.; GUTERRES, Marcos V.; OLIVEIRA, Poliana A. C.; NETO, Omar P. Vilela. DNAr-logic: a constructive DNA logic circuit designlibrary in R language for molecular computing. In: 32ND Symposium on Integrated Circuits and Systems Design - SBCCI '19. [S.l.]: ACM Press, 2019. DOI: 10.1145/3338852.3339854

- OLIVEIRA, Poliana A. C.; BOA, Maria C. O. Fonte; MARKS, Renan A.; GUTERRES, Marcos V.; NETO, Omar P. Vilela. Analysis of single-module and cascade molecular analog circuits for approximate computing based on DNA Strand Displacement. In: IEEE. 2020 33rd Symposium on Integrated Circuits and Systems Design (SBCCI). [s.l.: s.n.], 2020. P. 1–6

- MARKS, Renan A.; VIEIRA, Daniel K. S.; GUTERRES, Marcos V.; OLIVEIRA, Poliana A. C.; BOA, Maria C. O. Fonte; NETO, Omar P. Vilela. Design and Test of Digital Logic DNA Systems. **IEEE Design & Test**, Institute of Electrical and Electronics Engineers (IEEE), v. 38, n. 4, p. 94–101, Aug. 2021. DOI: 10.1109/mdat.2021.3069369

- OLIVEIRA, Poliana A. C.; TEIXEIRA, João V. C.; MARKS, Renan A.; GUTERRES, Marcos V.; NETO, Omar P. Vilela. Evaluating analog arithmetic circuit for approximate computing with DNA strand displacement. **Analog Integrated Circuits and Signal Processing**, Springer Science and Business Media LLC, v. 108, n. 3, p. 485–493, May 2021. DOI: 10.1007/s10470-021-01828-9. Available from: <https://doi.org/10.1007/s10470-021-01828-9>

- NETO, Omar P. Vilela; OLIVEIRA, Poliana A. C. de; MARKS, Renan A.; NOVY, Gabriel; VIEIRA, Maria; LUZ, Laysson Oliveira; SILVA, Pedro Arthur; FERREIRA, Ricardo; RAMIREZ, Jhonattan; NACIF, José Augusto; CAPORALI, Guilherme. Design Automation for Emerging Technologies. **Journal of Integrated Circuits and Systems**, Journal of Integrated Circuits and Systems, v. 17, n. 3, p. 1–

11, Dec. 2022. DOI: 10.29292/jics.v17i3.652. Available from: <https://doi.org/10.29292/jics.v17i3.652>

- OLIVEIRA, Poliana A. C.; MARKS, Renan A.; TEIXEIRA, João V. C.; GUTERRES, Marcos V.; NETO, Omar P. Vilela. DNAr-Analog: A Library With a Multiplexer to Easily Design, Program, and Simulate Dsd Analog Circuits. In: 2023 IEEE 14th Latin America Symposium on Circuits and Systems (LASCAS). [s.l.]: IEEE, Feb. 2023. DOI: 10.1109/lascas56464.2023.10108135. Available from: <https://doi.org/10.1109/lascas56464.2023.10108135>