From semigroup theory to vectorization : recognizing regular languages

Claire Soyez-Martin

Université de Lille

Université de Lille

Doctoral School **MADIS**

University Department **Inria Lille**

Thesis defended by **Claire Soyez-Martin**

Defended on **14th December, 2023**

In order to become Doctor from Université de Lille

Academic Field **Computer science**

Speciality **Theoretical computer science**

Thesis Title

From semigroup theory to vectorization: recognizing regular languages

Thesis supervised by	Sylvain Salvati	Supervisor
	Michaël Hauspie	Co-Supervisor
	Charles Paperman	Co-Monitor

Committee members

Referees	Sylvain Schmitz	Professor at Université Paris Cité	
	Damien Pous	Senior Researcher at CNRS	
Examiners	Sophie Tison	Professor at Université de Lille	
	Tatiana Starikovskaya	Associate Professor at ENS Ulm	
	Pierre Senellart	Professor at ENS Ulm	Committee President
Supervisors	Sylvain Salvati	Professor at Université de Lille	
	Michaël Hauspie	HDR Associate Professor at Université de Lille	
Comonitor	Charles Paperman	Associate Professor at Université de Lille	

Université de Lille

Université de Lille

Doctoral School **MADIS**

University Department **Inria Lille**

Thesis defended by **Claire Soyez-Martin**

Defended on **14th December, 2023**

In order to become Doctor from Université de Lille

Academic Field **Computer science**

Speciality **Theoretical computer science**

Thesis Title

From semigroup theory to vectorization: recognizing regular languages

Thesis supervised by	Sylvain Salvati	Supervisor
	Michaël Hauspie	Co-Supervisor
	Charles Paperman	Co-Monitor

Committee members

Referees	Sylvain Schmitz	Professor at Université Paris Cité	
	Damien Pous	Senior Researcher at CNRS	
Examiners	Sophie Tison	Professor at Université de Lille	
	Tatiana Starikovskaya	Associate Professor at ENS Ulm	
	Pierre Senellart	Professor at ENS Ulm	Committee President
Supervisors	Sylvain Salvati	Professor at Université de Lille	
	Michaël Hauspie	HDR Associate Professor at Université de Lille	
Comonitor	Charles Paperman	Associate Professor at Université de Lille	

Université de Lille

Université de Lille

École doctorale **MADIS**

Unité de recherche **Inria Lille**

Thèse présentée par **Claire Soyez-Martin**

Soutenue le **14 décembre 2023**

En vue de l'obtention du grade de docteur de l'Université de Lille

Discipline **Informatique**

Spécialité **Informatique théorique**

Titre de la thèse

De la théorie des semigroupes à la vectorisation : valider les langages réguliers

Thèse dirigée par	Sylvain Salvati	directeur
	Michaël Hauspie	co-directeur
	Charles Paperman	co-encadrant

Composition du jury

Rapporteurs	Sylvain Schmitz	professeur à l'Université Paris Cité	
	Damien Pous	directeur de recherche au CNRS	
Examinateurs	Sophie Tison	professeure à l'Université de Lille	
	Tatiana Starikovskaya	MCF à l'ENS Ulm	
	Pierre Senellart	professeur à l'ENS Ulm	président du jury
Directeurs de thèse	Sylvain Salvati	professeur à l'Université de Lille	
	Michaël Hauspie	MCF HDR à l'Université de Lille	
Encadrant de thèse	Charles Paperman	MCF à l'Université de Lille	

There is always another secret.

Kelsier

Remerciements

Je tiens à remercier Damien Pous et Sylvain Schmitz pour avoir accepté de rapporter ma thèse. Leurs retours détaillés ont permis d'améliorer ce manuscrit, et permettront de l'améliorer encore pour la version finale qui sera diffusée. Merci également à Pierre Senellart, Tatiana Starikovskaya et Sophie Tison, qui me font l'honneur d'être membres du jury de soutenance.

Merci à mes encadrants, ne serait-ce que pour m'avoir supportée trois ans. Merci à Michaël Hauspie, avec qui il a été possible de parler technique, et qui nous a proposé d'utiliser Rust. Merci à Charles Paperman, Seigneur du Chaos et Éminence Grise, qui a proposé en moyenne une dizaine d'idées par jour (parfois il devait dormir, il paraît, ça baisse la moyenne ; mais ce n'est peut-être qu'une légende) et a fait le lien avec l'administration. Merci à Sylvain Salvati, chef d'équipe et Lambdaborn, qui a permis de créer de l'ordre à partir du chaos des idées. J'aimerais aussi remercier tous les membres de l'équipe LINKS pour leur soutien et leurs conseils.

Cette thèse n'aurait pas eu la même ambiance sans le formidable bureau des doctorants. Pour cette raison, merci à Oliver, le nouveau Chief Hapiness Manager de l'équipe, à Corentin, partenaire indispensable au duo avec Oliver, et à Nicolas, le meilleur contrepoint au chaos ambiant. Merci aussi à Paul et Momar, les anciens doctorants du labo, qui m'ont permis de prendre mes marques.

Enfin, je remercie ma famille et mes amis, à qui je dédie cette thèse. Merci à Steve pour sa présence et son soutien constants, y compris lorsque l'on était dans le même Titanic de rédaction de thèse. Merci à Timothée, Vincent, Jofrey et Simon, garder contact avec moi ce n'est pas franchement facile, ça mériterait une médaille. Mais surtout, merci de me rappeler pourquoi c'est important de continuer d'avancer. Merci à Quentin, capable de traverser régulièrement la moitié de la France pour revoir des amis. Je ne comprendrai jamais comment tu fais pour tenir. Merci à Antoine, sans qui ne ne serait pas arrivée jusque là. Et bien sûr, merci à vous qui lisez cette thèse !

From semigroup theory to vectorization: recognizing regular languages

Abstract

The pursuit of optimizing regular expression validation has been a long-standing challenge, spanning several decades. Over time, substantial progress has been made through a vast range of approaches, spanning from ingenious new algorithms to intricate low-level optimizations.
Cutting-edge tools have harnessed these optimization techniques to continually push the boundaries of efficient execution. One notable advancement is the integration of vectorization, a method that leverages low-level parallelism to process data in batches, resulting in significant performance enhancements. While there has been extensive research on designing handmade tailored algorithms for particular languages, these solutions often lack generalizability, as the underlying methodology cannot be applied indiscriminately to any regular expression, which makes it difficult to integrate to existing tools.
This thesis provides a theoretical framework in which it is possible to generate vectorized programs for regular expressions corresponding to rational expressions in a given class. To do so, we rely on the algebraic theory of automata, which provides tools to process letters in parallel. These tools also allow for a deeper understanding of the underlying regular language, which gives access to some properties that are useful when producing vectorized algorithms. The contribution of this thesis is twofold. First, it provides implementations and preliminary benchmarks to study the potential efficiency of algorithms using algebra and vectorization. Second, it gives algorithms that construct vectorized programs for languages in specific classes of rational expressions, namely the first order logic and its subset restricted to two variables.

Keywords: regular expression, algebraic theory of automata

De la théorie des semigroupes à la vectorisation : valider les langages réguliers

Résumé

L'évaluation efficace des expressions régulières constitue un défi persistant depuis de nombreuses décennies. Au fil du temps, des progrès substantiels ont été réalisés grâce à une variété d'approches, allant de nouveaux et ingénieux algorithmes à des optimisations complexes de bas niveau.
Les outils de pointe de ce domaine utilisent ces techniques d'optimisation, et repoussent constamment les limites de leur efficacité. Une avancée notoire réside dans l'intégration de la vectorisation, qui exploite une forme de parallélisme de bas niveau pour traiter l'entrée par blocs, entraînant ainsi d'importantes améliorations de performances. Malgré une recherche approfondie sur la conception d'algorithmes sur mesure pour des tâches particulières, ces solutions manquent souvent de généricité, car la méthodologie sous-jacente à ces algorithmes ne peut pas être appliquée de manière indiscriminée à n'importe quelle expression régulière, ce qui rend difficile son intégration dans les outils existants.
Cette thèse présente un cadre théorique permettant de générer des programmes vectorisés particuliers capables d'évaluer les expressions régulières correspondant aux expressions rationnelles appartenant à une classe logique donnée. L'intérêt de ces programmes vectorisés vient de l'utilisation de la théorie algébrique des automates, qui offre certains outils algébriques permettant de traiter les lettres en parallèle. Ces outils permettent également d'analyser les langages réguliers plus finement, offrent accès à des optimisations des programmes vectorisés basées sur les propriétés algébriques de ces langages.
Cette thèse apporte des contributions dans deux domaines. D'une part, nous présentons des implémentations et des benchmarks préliminaires, afin d'étudier les possibilités offertes par l'utilisation de l'algèbre et de la vectorisation dans les algorithmes d'évaluation des expressions régulières. D'autre part, nous proposons des algorithmes capables de générer des programmes vectorisés reconnaissant les langages appartenant à deux classes d'expressions rationnelles, la logique du premier ordre et sa restriction aux formules utilisant au plus deux variables.

Mots clés : expression régulière, théorie algébrique des automates

---

Contents

Introduction

In recent years, there has been a growing demand for high-performance software systems that can handle vast amounts of data efficiently. Online resources such as websites need to differentiate between normal traffic, which can be heavy, and denial-of-service (DoS) attacks. Bioinformatics deal with large DNA sequences that need to be efficiently handled in order to find patterns, search for particular properties, or compare sequences. Data crawling has become pervasive and demanding on the systems, as databases grow in size, and as larger XML or Json documents need to be parsed. One critical aspect of building such systems is data validation, ensuring that input data meets specific criteria before processing. Often, these criteria are specified using *regular expressions*. Regular expressions allow developers to specify complex patterns that the input data must adhere to, enabling the validation of email addresses, phone numbers, and other structured data formats effectively. As of today, regular expressions can be complex, using groups, backreferences and lookaheads. Because of the increasingly complex operations used by regular expressions and the increasing amounts of data to process, runtime data validation can be computationally expensive, leading to potential bottlenecks during program execution. To avoid these bottlenecks, the state-of-the-art tools for processing regular expressions use optimized programs depending on the complexity of the input expressions. The complete set of regular expressions is difficult to handle, yet a subset of expressions is directly related to rational expressions, which are a way of modelling the structure of regular languages. This link allows for a formalization of a subset of data validation using regular languages. Therefore, creating efficient algorithms that recognize regular languages is crucial for the development of softwares that need to validate large amounts of data.

Algorithms recognizing regular languages have evolved with time, going from simple (sometimes not so simple) string matching algorithms [11, 42, 18, 6, 55, 34, 54] to pattern-matching algorithms capable of recognizing any regular language [5], to approximate pattern-matching algorithms [79, 53, 78], which are able to match words against a given pattern with a bounded number of mismatches. Each of these kinds of algorithms has found application in real-world tools, such as the famous `grep`. These tools, especially `grep`, have evolved alongside the research on regular languages recognition, as both the complexity of the syntax and the efficiency of the algorithms (and their implementation) increased. These algorithms can also be split into categories depending on the data structures they use: some directly reason on the structure of the regular expressions (notably backtracking algorithms), some build a nondeterministic finite automaton then determinize it on the fly [74], others explicitly determinize that automaton to execute it in linear time. As of today, the state-of-the-art tools that perform data validation use a combination of algorithms depending on the structure of the regular expression given to them. In particular, they avoid looking at every character of the input data by skipping most characters [31]. Indeed, looking sequentially at each character is often not necessary and leads to performance issues, and so it must be avoided as much as possible. Thus, modern tools are experts at skipping most of the data to look only at the interesting parts.

How do these programs know what parts of the data are interesting to look at? The short answer is *they don't really know*. Some algorithms are used to decompose the regexes and find the sub-patterns that can be processed to skip characters, but usually these sub-patterns are just strings of characters, that can be optimized using the Boyer-Moore algorithm [11]. But, aside from strings of characters, very few sub-patterns are actually optimized in a regular expression. One reason behind that sad reality is that most tools (and the underlying algorithms) are designed to be general: they must be able to process any regular expression, that is not known in advance. No optimization is possible before knowing the regular expression to be matched against data, and thus any specific optimization must be made at runtime, which rapidly becomes too costly, as the time taken to optimize the data structures outweighs the gain from these optimized structures.

Therefore, one possible path to explore in order to produce even more efficient programs than the state of the art is to suppose that the regular expressions are known in advance. Currently, this approach is mostly used by parsers such as `lex` or `Flex`. Aside from that, extensive research has been conducted in order to produce efficient programs for given languages, such as UTF-8 encoding [40], and have led to implementations in some popular libraries (notably the SimdJSON library [45]). The programs produced take into account the properties of the studied languages and use them to optimize the search, which leads to impressive results. However, little to no research has been done on generalizing the work done for these specific languages. One of the challenges of this generalization is the fact that automatically tailoring a data validation algorithm for any given language is vastly different from tailoring a data validation algorithm for a specific chosen language. Depending on the properties of the language, different kinds of optimizations might be more adapted. This thesis attempts at following the lead of `lex` and `Flex`, by considering some of these possible optimizations.

One broad kind of optimization consists in producing *branch-free programs*, which avoid conditional statements during the execution of the program. This leads to more efficient programs, as conditional statements tend to slow them down, especially when they are mispredicted. While complete avoidance of conditional statements is not always achievable, the emphasis is placed on minimizing their usage. The exact structure of such programs varies depending on the regular expression, but the core principle of this kind of optimization is to perform computations that do not depend on the ongoing search state. One common strategy in this line of optimization consists in leveraging lookup tables to simulate automaton-like behavior without branching depending on the state of the search and the value of the data characters. Instead, the state of the search is updated using the lookup tables' entries corresponding to the current state of the search. Another optimization strategy consists in processing larger portions of the input data in one CPU cycle. The most common example is data parallelism, where the data is divided into segments, which are then concurrently processed by individual CPU cores. Thus, in one CPU cycle, as many characters as the number of cores are processed, which increases efficiency by a similar factor.

These two optimization approaches can be effectively used together within a single algorithm. One method to combine them is to use *vectorization*, a technique that relies on a particular form of parallelism called *bit-level parallelism*. This form of parallelism consists in using particular operations, that we refer to as bit-level parallel operations. In practice, these theoretical operations are implemented as *SIMD* (Single Instruction Multiple Data) *instructions*. This name denotes the ability of bit-level parallel operations to handle a *batch* of data, also called a *vector*, in one CPU cycle, contrary to usual instructions which only handle a single character per cycle. Several models have been developed to study bit-level parallelism. Notably, Pratt et al. [62] introduced a restricted model of RAM called *vector machines*, which

relies on bitwise boolean operations and shifts. This paper, along with some others [32, 69], led to the formulation of a *parallel computation thesis* by Goldschlager in his thesis [30]. When vectorization is employed for data validation, the data is divided in batches, the size of which is equal to that of a computer word. These batches are processed by performing a given sequence of vectorized operations on each batch, which computes some information, notably a vector, which is then used to update the state of the search. This approach can be seen as a generalization of lookup table concepts applied to parallelism, as it handles batches of data without conditional statements to update the state of the search.

When used for data validation, vectorized algorithms are not constructed directly from the automaton of the considered language. Indeed, an automaton is inherently sequential. There is no obvious way to accommodate this kind of sequentiality to processing data by batches. In order to construct a vectorized algorithm equivalent to a given automaton, it is necessary to go through some intermediary steps to convert the automaton into a data structure more adapted to vectorization. When considering a given fixed language, it is possible to construct a data structure tailored to the properties of the regular language to recognize. However, it is hard to generalize that method to any language, since it would require to automatically detect the properties of a given language. The strategy considered in this thesis uses *semigroups*, a tool from the algebraic theory of automata, to study some of these properties. Semigroups are linked to regular languages, as it is always possible to compute the *syntactic semigroups* of a regular language. Using this connection, we can use semigroups to study the properties of the languages. This area of research has been extensively studied, and numerous classes of semigroups have been defined using some interesting properties. Consequently, the goal of this thesis is to study the following question:

How can we leverage the algebraic theory of automata to produce, given any regular language in a fixed class, an efficient vectorized program that recognizes the language?

Semigroups are particularly useful for vectorization. Indeed, their algebraic properties are amenable to parallel computations methods such as map/reduce. We can map each letter of a word to its corresponding semigroup element, then compute in parallel pairwise products and recursively evaluate the value of the whole input. This method of computation can be implemented with SIMD instructions on each batch that is processed. Taking advantage of properties of particular semigroups, this kind of parallel computation can be further optimized with SIMD instructions. For example, a sequence of well-chosen products can then be translated into a small vectorized program. In this case, it is then possible to produce vectorized programs capable of calculating the product of an entire sequence of semigroup elements.

Such a vectorized program can be used for data validation. Indeed, for a given input word, it is possible to compute the corresponding sequence of elements of the syntactic semigroup. From there, using a vectorized program that computes the product of any sequence of elements in that semigroup, we can obtain the element to which the input word evaluates. The value of that element then determines whether the initial word belongs to the language.

Among the various semigroup classes, two classes are the focus of this thesis, because of their numerous characterizations and properties. First, the class **Ap** of aperiodic semigroups has been the main drive in this work. This class of semigroups corresponds to the class of languages FO[<], the first-order logic equipped with the order relation. It is of particular interest, as a remarkable result of Serre [67] establishes the equivalence between the class **Ap** and a class of programs characterized by a limited set of operations, all of which are bit-level parallel

operations. This result completes the one proved by Bergeron and Hamel [8], who stated that any language in **Ap** can be recognized by what they called a *vector algorithm*. However, these algorithms have some strong pre-requisites: Serre's result assumes access to Linear Temporal Logic (LTL) formulas capable of recognizing the target languages. Unfortunately, those formulas introduce some challenges, as they are hard to compute from a given regular expression, as shown by a result of Wilke [77]. As a consequence, generating programs based on these formulas becomes laborious, potentially demanding an extensive runtime.

The second class of semigroups studied in this thesis is **DA**, which is equivalent to the logical class $\mathrm{FO}_2[<]$ of first-order formulas that use at most two variables. This class is a subset of **Ap**, and so all the results in that class can be applied to **DA**. However, to find a class of programs equivalent to **DA**, it is necessary to restrict the available operations. Among the numerous characterizations of **DA**, the class has been shown to be equivalent to a class of programs called *turtle programs*, which can be recognized using bit-level parallel operations. Yet, turtle programs introduce a significative complexity, as proofs showing the equivalence with $\mathrm{FO}_2[<]$ are indirect and not constructive [19, 73]. They are of no use to actually compile automata into formulas or vectorized programs, but suggest that the size of those programs may be significantly large.

To summarize, both the classes **Ap** and **DA** have properties making them ideal to be handled with vectorized programs but, with the current results, either those vectorized programs are too large to be usable in practice, or there is no known direct algorithm to construct them. This thesis provides two general algorithms that compute vectorized programs: the first one, given a semigroup in **Ap**, produces a vectorized program that computes the product of any sequence of elements of that semigroup. The second one has a similar outcome for a given semigroup in **DA**. In order to generalize the formalization of these two algorithms, we introduce a global framework using what we call *evaluation programs*.

In order to measure the efficiency of the vectorized programs presented in this thesis, a testing framework has been developed, using the programming language Rust. This code gave some interesting results shown in the benchmarks presented in the thesis, giving insights on what the use of vectorization and semigroups can offer in terms of speedup. It initiates a practical study of the effective efficiency of those tools, and shows that efficiently encoding the programs presented in the main results of this thesis could be interesting.

Structure of the thesis. We now give an overview of the contents of this thesis. The document in divided in five chapters. Chapter 1 gives the preliminary definitions and results essential for the rest of the thesis. It encompasses automata theory, algebra and boolean circuits, which form the core areas of our research. In particular, this chapter defines the two classes of languages that are the subject of our two main results.

Chapter 2 serves as an initial benchmarking stage, offering insights into the potential enhancements that algebra can bring to existing regular expressions processing algorithms. The first section introduces regular expressions, which are used for defining languages, notably through standards such as IEEE Posix. This section provides an overview of the existing types of algorithms used to process them. The rest of the chapter is dedicated to benchmarks. The second section describes the methodology, along with the specificities of the chosen framework. The third section describes the standard algorithms that have been implemented in the benchmark's framework, and gives the results obtained for these algorithms. The fourth section describes new algorithms that leverage algebra in an attempt to improve the results of the usual algorithms. It also presents variations of those algorithms that could be more efficient. Benchmark results are presented for each of those algorithms.

Chapter 3 introduces bit-level parallelism and a circuit-based model known as vectorial circuits. The first section defines bit-level parallelism, with a focus on the specific form considered in this thesis, streaming bit-level parallelism, which was first introduced by Lamport [44]. The second section formalizes streaming bit-level parallelism as an automaton whose alphabet is a set of words of fixed size. A few implementations of regular expressions using SIMD are given, along with a small benchmark of these implementations. The third section introduces vectorial circuits as an alternative formalization of streaming bit-level parallelism, which is more adapted to represent the SIMD implementations. Two particular classes of circuits are defined in this section: ADD-vectorial circuits and Sweeping-vectorial circuits. These two classes will be the main focus of chapter 4. Before that, the fourth section of the chapter discusses some potential methods for adapting circuits to streaming. The adaptation of vectorial circuits presented in this section is inspired by an article by Murlak et al. [51].

Chapter 4 presents and proves the two theoretical contributions of this thesis. The first contribution introduces an algorithm that, given a language in the class **Ap**, constructs an ADD-vectorial circuit recognizing the language. The second contribution introduces a similar algorithm which, given a language in the class **DA**, constructs a Sweeping-vectorial circuit recognizing that language.

Chapter 1

Preliminaries

Outline of the current chapter

This thesis makes a connection between regular languages and a variation of boolean circuits, following the work of Bergeron, Hamel [8], and Serre [67]. To do so, languages are characterized through their algebraic properties, which lead to the generation of programs optimized for some classes of languages. Before detailing that work, we need to define all those terms. This chapter gives the background needed in logics and algebra to read the rest of this thesis.

We begin by talking about automata, which define regular languages. In section 1.1, we define automata, along with some other characterizations. We also introduce first-order logic, defining the class of languages on which this thesis is focused. Then, in section 1.2, we give definitions for the algebraic tools that are needed. Using the definitions from both of these sections, we give in section 1.3 the characterizations of the two classes that are used in this thesis: FO[<] and $FO_2[<]$. Finally, we define boolean circuits, which form the basis on which vectorial circuits will be defined (see section 3.3) and we present various circuit classes.

1.1 Automata theory

This thesis relies on the algebraic theory of automata in order to recognize regular languages. Indeed, some algebraic properties are useful to take into account the intricacies of the studied languages. This section presents various logical formalisms that describe classes of regular languages, notably first order logic over finite words, as all the contributions of this thesis focus on fragments of this logic.

1.1.1 Notations

For any $n \in \mathbb{N}$, we write $[n]$ for the set $\{0, \ldots, n-1\}$. Given a set S, we denote by $|S|$ its cardinality. For the sake of conciseness, a singleton is denoted by its unique element when the context makes it clear that it is a set: given an element u, the singleton $\{u\}$ is denoted by u.

1.1.2 Words

Let Σ be an alphabet, i.e. a set of elements called letters. Given an integer $n \in \mathbb{N}$, a finite *word* of size n on the alphabet Σ is a sequence $a_0 a_1 \cdots a_{n-1}$ such that, for any index $i \in [n]$, $a_i \in \Sigma$. A word $a_0 \cdots a_n$ is the *concatenation* of the letters a_0 to a_n. The concatenation of two words w_1 and w_2 can be denoted either by $w_1 w_2$ or, more explicitly, by $w_1 \cdot w_2$. We denote the empty word of Σ^* by ϵ. We use the exponent notation to denote repetition: for any letter a and any integer $n > 0$, a^n denotes the word $a \cdots a$ of size n. We also define $a^0 = \epsilon$. The union of all possible exponents of a is denoted by a^*; formally $a^* = \bigcup_{n \in \mathbb{N}} a^n$. These notations can be extended to sets. Given a set $A \subseteq \Sigma$ of letters, A^n is the set of words of size n that can be obtained by concatenating n letters of A; formally, $A^n = \{a_0 \cdots a_{n-1} \mid \forall i \in [n], a_i \in A\}$. We also set $A^0 = \{\epsilon\}$. Similarly, A^* is the set of any word that can be formed by concatenating any number of letters of A: $A^* = \bigcup_{n \in \mathbb{N}} A^n$.

A word $v = v_0 \cdots v_n$ is an *infix* (also called *factor* or *subword*) of another word $w = w_0 \cdots w_m$ if there exists an index $i < m - n$ such that the words v and $w_i \cdots w_{i+n}$ are equal. This amounts to saying that there exist words $x, y \in \Sigma^*$ such that $w = xvy$. Similarly, v is a *prefix* (resp. *suffix*) of w if there exists a word $x \in \Sigma^*$ such that $w = vx$ (resp. $w = xv$).

Example 1.1. The words *apple*, *a* and *app* are prefixes of the word *apple*. The word *le* is one of its suffixes.

1.1.3 Automata

Definition 1.1

A *language* is a subset of Σ^*, for any alphabet Σ. A language is said to be *regular* if it is recognized by a tool named an *automaton*.

An *automaton* is a quintuple $A = (\Sigma, Q, I, \delta, F)$, where

- Σ is the input alphabet, i.e. the set of letters that can be read by the automaton.
- Q is the automaton's finite set of states.
- $I \subseteq Q$ is the set of possible input states.

- $\delta : Q \times \Sigma \to 2^Q$ is the transition function.
- $F \subseteq Q$ is the set of accepting states.

The transition function δ can be partial, meaning that, for some pairs $(q, a) \in Q \times \Sigma$, $\delta(q, a)$ is not defined. For any of these pairs, we consider that the automaton rejects the word, regardless of the rest of the input. It amounts to considering a sink state q_{sink} such that $q_{\text{sink}} \notin F$, $\delta(q, a) = \{q_{\text{sink}}\}$ and, for any letter $b \in \Sigma$, $\delta(q_{\text{sink}}, b) = \{q_{\text{sink}}\}$.

In some particular cases, we allow automata to have particular transitions called *ϵ-transitions*. An ϵ-transition is a transition labeled by ϵ, in which case there exists a state $q \in Q$ such that $\delta(q, \epsilon)$ is defined. These transitions make the automaton change its current state without reading any letter of the input word.

A finite *path* in an automaton $A = (\Sigma, Q, I, \delta, F)$ is a sequence of transitions of the form $(q_0, a_0, q_1)(q_1, a_1, q_2) \cdots (q_{n-1}, a_{n-1}, q_n)$ such that, for any $i \in [n]$, $q_{i+1} \in \delta(q_i, a_i)$. The word $a_0 \dots a_{n-1}$ is the *label* of the path. An *accepting path* is a path $(q_0, a_0, q_1) \cdots (q_{n-1}, a_{n-1}, q_n)$ such that $q_0 \in I$ and $q_n \in F$.

A word w is *accepted* by the automaton A if there exists an accepting path in A such that w is its label. The set of all the words accepted by A is the *language accepted* by A. We denote it by $L(A)$.

An automaton $A = (\Sigma, Q, I, \delta, F)$ is *deterministic* if $|I| = 1$ and, for each pair $(q, a) \in Q \times \Sigma$, $|\delta(q, a)| \leq 1$. If an automaton is deterministic and has a finite set of states Q, it is called a *deterministic finite automaton* (DFA for short). If a finite automaton is not deterministic, it is called a *non-deterministic finite automaton* (NFA for short). It is always possible to consider a deterministic automaton, as every finite automaton is equivalent to a deterministic one. This property can be proved using a standard method called the powerset construction [63], introduced by Rabin and Scott in 1959.

The powerset construction consists in considering the automaton which states are all the possible sets of states in the NFA. The transitions are defined as follows: given two sets of states $q = \{q_0, \dots, q_n\}$ and $q' = \{q'_0, \dots, q'_m\}$, and a letter x in the alphabet, there is a transition from q to q' labeled by x in the powerset DFA if and only if

$$\bigcup_{q_i \in q} \delta(q_i, x) \subseteq q'$$

Then, the DFA is trimmed by keeping only the states that are reachable.

Example 1.2. Consider the regular language $(a+b)^*a(a+b)(a+b)$, which is the set of all words of the alphabet $\{a, b\}$ of size at least 3 such that the antepenultimate letter is an occurrence of a. The naive automaton recognizing this language is a simple NFA, shown in Figure 1.1.

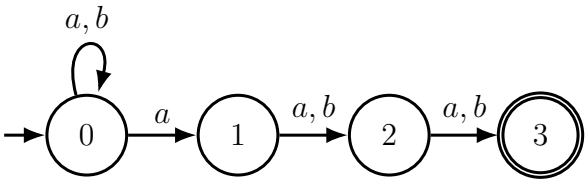


Figure 1.1 – The naive NFA recognizing $(a + b)^*a(a + b)(a + b)$

To compute the powerset automaton corresponding to that NFA while trimming the sets that are not reachable, we begin by considering the initial state, and associating a new initial

state in the DFA. Thus, we define the first state of the DFA, which corresponds to the set $\{0\}$ of states of the NFA. In the NFA, if the letter a is read from the state 0, the automaton either goes to 0 or 1, so we define the second state of the DFA, which corresponds to the set $\{0, 1\}$ of states of the NFA, and we add a transition from 0 to 1 labeled by a. If the letter b is read from the state 0 of the NFA, it stays in the state 0, so we add a loop over the state 0 of the NFA, labeled by b. Then, we consider the transitions that begin at the state $\{0, 1\}$ of the DFA: if the NFA is in state 0 or 1 and the letter a is read, it can go to any state among 0, 1 or 2. Thus, we define the third state of the DFA, which corresponds to the set $\{0, 1, 2\}$, and we add a transition from state $\{0, 1\}$ to state $\{0, 1, 2\}$, labeled by a. Using this algorithm, we obtain the DFA shown in Figure 1.2. This DFA does not contain all the possible sets of states of the NFA, as the ones not shown would not be reachable. Note that the DFA we obtained here is minimal, but that is not always the case, and it is often necessary to minimize the DFA.

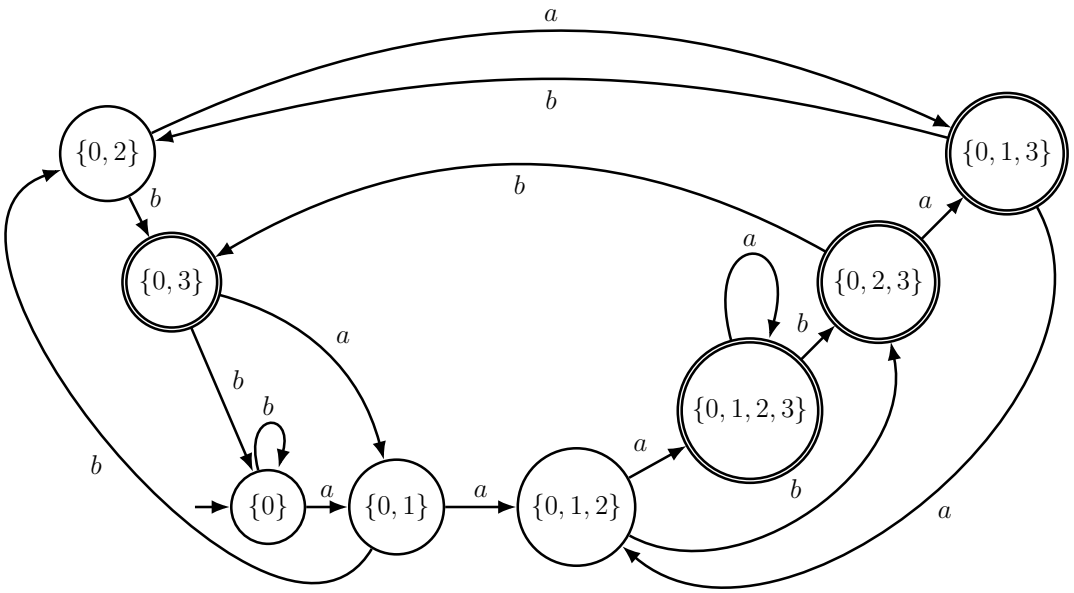


Figure 1.2 – The minimal DFA recognizing $(a + b)^*a(a + b)(a + b)$

The powerset construction has notably been used by Thompson in [74], where he gives an implicit algorithm to determinize a nondeterministic automaton. This algorithm, called *on-the-fly determinization*, takes an NFA $(\Sigma, Q, I, \delta, F)$ and executes it on the input word by maintaining a set of current states. This is done as follows: at the beginning of the algorithm, the set of states C is equal to I. Then, given a set C of current states and the next letter $a \in \Sigma$ of the input word, the next set C' is computed by considering each state q in turn and adding to C' the states q' that can be reached from q by reading a. In symbols, $C' = \{q' \in Q \mid \exists q \in C, q' \in \delta(q, a)\}$. The input word is accepted if and only if the final set of states contains an accepting state.

Example 1.3. Consider the regular language $a(baa+bcc^*)c$, which is the set of words composed of $abaac$ and all the words of the form $abcc \cdots c$, with at least two occurrences of c. This language can be translated into the equivalent NFA given in Figure 1.3. Now, consider the input text

abccc. An NFA-based algorithm on this input maintains a set of current states as follows: the initial set is $\{0\}$, as 0 is the only input state. When reading the first letter of the input, an a, the algorithm follows the unique transition labelled by a beginning at state 0. Therefore, the new set of states is $\{1\}$. Then, the algorithm reads a b. This time, there are two transitions to consider: the one from 1 to 2 and the one from 1 to 3. So, the new set of states is $\{2, 3\}$. Thus, when reading the next letter, a c, the algorithm considers the transition beginning at 2 and the one beginning at 3. There is no transition labelled by c beginning at 2, so this path fails, but there is a transition labelled by c beginning at 3, so the computation continues with the set $\{6\}$. Once more, there are two transitions beginning at 6 and labelled by c, which gives the set of states $\{6, 7\}$. The next two letters lead to the same set of states, so the accepting state 7 is in the final set of states. Thus, the input word is accepted.

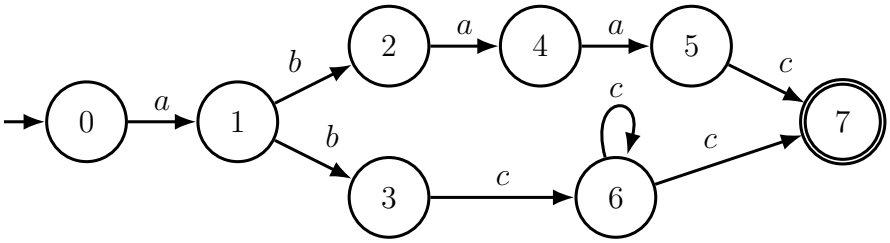


Figure 1.3 – An NFA recognizing the language $a(baa + bcc^*)c$

Considering static deterministic automata (as opposed to the DFAs constructed on the fly) instead of nondeterministic ones has its advantages and inconvenients. Indeed, the execution of a deterministic automaton can be done in linear time, as each pair (state, letter) has at most one successor state, whereas in a nondeterministic automaton such a pair may have the set Q as set of successors, and thus can only be run in time $O(|w| \times |Q|)$. On the other hand, the construction of a deterministic automaton can be costly, as the number of states can be exponential in the number of states of the nondeterministic automaton. The bound $2^{|Q|}$ is tight [50], as some languages have been proved to have no deterministic automaton with less than $2^{|Q|}$ states. As an example of simple NFA equivalent to a large DFA, one can consider the language $(a|b)^*a(a|b)\{n-1\}$, where the last group $(a|b)$ is repeated $n-1$ times: there is no deterministic automaton of size less than 2^n recognizing that language [3].

Not only is the bound $2^{|Q|}$ on the size of determinized DFAs tight, but it is also common to obtain a large DFA when using determinization, even when the minimal DFA recognizing the language is rather small. In these cases, it can be useful to *minimize* the obtained DFAs. Several algorithms exist, notably Hopcroft's algorithm [33], which is the most known. In this thesis, the algorithm used was Brzozowski's [12]. This algorithm takes an NFA and returns the minimal equivalent DFA. It consists in four steps: reversing all the transitions of the input NFA, determinizing the new NFA, reversing once more the transitions, and finally determinizing the obtained automaton.

In this thesis, the expression "deterministic automaton" always references a deterministic finite automaton (DFA), and the expression "nondeterministic automaton" always references a nondeterministic finite automaton (NFA).

1.1.4 Rational expressions

An automaton is one way to define a regular language, but there is another tool to do so. In some contexts, this tool is called a regular expression, which links it tightly to regular languages.

However, the term *regular* can be a source of confusion as it has different meanings depending on the context. For this reason, this thesis refers to this tool as its standard name, *rational expression*. Rational expressions are formal expressions defined recursively as follows: given an alphabet Σ,

- $\{\}$ and $\{\epsilon\}$ are rational expressions.
- For any letter $a \in \Sigma$, a is a rational expression.
- For any two rational expressions e and e', $e + e'$, $e \cdot e'$ and e^* are rational expressions.

A rational expression defines a language as follows: for any letter $a \in \Sigma$, the rational expression a represents the language $\{a\}$. Then, for any two rational expressions e and e' representing respectively the languages L_1 and L_2, $e + e'$ represents $L_1 \cup L_2$, $e \cdot e'$ represents $L_1 \cdot L_2 = \{w \in \Sigma^* \mid \exists w_1 \in L_1, \exists w_2 \in L_2, w = w_1 \cdot w_2\}$ and e^* represents $L_1^* = \{w^* \mid w \in L_1\}$.

There is a strong correspondence between rational expressions and automata. One fundamental result, proved by Kleene [41] in 1956, is the following:

Theorem 1.1: Kleene's theorem

The set of languages definable by rational expressions is exactly the set of languages such that there exists an automaton recognizing the language.

Example 1.4. On any alphabet Σ, the set of words of odd length can be written as $(\Sigma^2)^*\Sigma$.

On the alphabet $\Sigma = \{a, b, c\}$, the expression $(ac^*b + c)^*$ is a rational expression that represents the regular language of words that do not have an infix of the form $ac \cdots ca$ or $bc \cdots cb$ and such that an a is always followed by a b. Formally, that regular language is the one recognized by the automata in Figure 1.4.

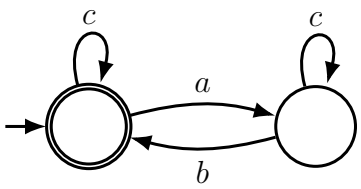


Figure 1.4 – The minimal DFA recognizing $(ac^*b + c)^*$

1.1.5 First-order logic

Among regular languages, the ones definable with *first-order logic* (FO for short) form a well-known class of languages, and the one considered in this thesis.

Syntax. Given a set $\mathcal{E}$ of functions on integers, we define the syntax of FO[$\mathcal{E}$] formulas as based on the following logical symbols: $\vee$ (or), $\wedge$ (and), $\neg$ (not), $=$ (equality test), the implication $\Rightarrow$, the symbols corresponding to the relations in $\mathcal{E}$, the existential quantifier $\exists$ and the universal quantifier $\forall$. It uses **true** and **false** as base formulas and has access to an unbounded number of variable names $(x, y, z, x_0, \ldots)$. Given an alphabet Σ, it has access to the predicate $a(x)$ for any letter $a \in \Sigma$.

We say that a variable x is *bound* in a FO[$\mathcal{E}$] formula ϕ if all the occurrences of x in ϕ are in the scope of a quantifier ($\exists$ or $\forall$). On the contrary, if there exists at least one occurrence of x in ϕ that is not in the scope of a quantifier, we say that x is a *free variable* of ϕ. A formula without free variables is called a *sentence*. With these definitions, we can define recursively the syntax of an FO[$\mathcal{E}$] formula for an alphabet Σ:

- **true** and **false** are FO[$\mathcal{E}$] formulas.
- For a letter $a \in \Sigma$, and any two variable names x and y, $a(x)$ and $x = y$ are FO[$\mathcal{E}$] formulas.
- For any n-ary function $f \in \mathcal{E}$ and any variable names $x_0, \dots, x_{n-1}$, $f(x_0, \dots, x_{n-1})$ is an FO[$\mathcal{E}$] formula.
- For any two FO[$\mathcal{E}$] formulas ϕ and ψ, $\neg\phi$, $\phi \vee \psi$, $\phi \wedge \psi$ and $\phi \Rightarrow \psi$ are FO[$\mathcal{E}$] formulas.
- For any FO[$\mathcal{E}$] formula ϕ and any variable x of ϕ, $\exists x, \phi$ and $\forall x, \phi$ are FO[$\mathcal{E}$] formulas.

Semantics. The semantics of FO[$\mathcal{E}$] associate a truth value to any sentence according to a structure, which gives a domain for the bound variables. Formally, a *structure* on a set $\mathcal{L}$ of non-logical symbols (relation, function and constant symbols) is defined by a *domain* D, which is a nonempty set, and an *assignment* which maps the symbols in $\mathcal{L}$ to relations, functions and constants: each n-ary relation symbol is mapped to an n-ary relation defined on D, each n-ary function symbol is mapped to an n-ary function defined on D, and each constant symbol is mapped to an element of D.

Given this structure, we can define the valuation of an FO[$\mathcal{E}$] formula, which will allow us to give a truth value to the sentences of FO[$\mathcal{E}$]. Given a domain D, a *valuation* is a map ν from the set of variables to the domain D. This definition can be extended to obtain a map from the set of terms over $\mathcal{L}$ to D, using the following rule: for any n-ary relation $R \in \mathcal{E}$ and any terms $t_0, \dots, t_{n-1}$, $\nu(R(t_0, \dots, t_{n-1})) := R(\nu(t_0), \dots, \nu(t_{n-1}))$. We use this valuation to define a relation "(w, ν) satisfies ϕ in the structure $\mathcal{S}$", where $w \in \Sigma^*$ is a word. To do this, we need one more notation that will allow us to assign values to bound variables: given a valuation ν and n an element of D, we define $\nu\binom{n}{x}$ to be the valuation such that $\nu\binom{n}{x}(x) = n$ and, for any variable $y \neq x$, $\nu\binom{n}{x}(y) = \nu(y)$.

Now, for an alphabet Σ, a word $w = w_0 \cdots w_{n-1} \in \Sigma^*$, an FO[$\mathcal{E}$] formula ϕ and a valuation ν on a domain $D \subseteq [n]$, we can define the expression "the pair (w, ν) satisfies the formula ϕ", written as $(w, \nu) \models \phi$, using an induction based over ϕ with the following rules:

- For any letter $a \in \Sigma$, $(w, \nu) \models a(x)$ if and only if $w_{\nu(x)}$ is the letter a.
- $(w, \nu) \models t_1 = t_2$ if and only if $\nu(t_1) = \nu(t_2)$.
- For any $f \in \mathcal{E}$, $(w, \nu) \models R(t_0, \dots, t_{n-1})$ if and only if $f(\nu(t_1), \dots, \nu(t_{n-1}))$.
- $(w, \nu) \models \neg\phi$ if and only if $(w, \nu) \not\models \phi$.
- $(w, \nu) \models \phi \vee \psi$ if and only if $(w, \nu) \models \phi$ or $(w, \nu) \models \psi$.
- $(w, \nu) \models \phi \wedge \psi$ if and only if $(w, \nu) \models \phi$ and $(w, \nu) \models \psi$.
- $(w, \nu) \models \phi \Rightarrow \psi$ if and only if $(w, \nu) \not\models \phi$ or $(w, \nu) \models \psi$.

- $(w, \nu) \models \exists x, \phi$ if and only if there exists $n \in D$ such that $(w, \nu\binom{n}{x}) \models \phi$.
- $(w, \nu) \models \forall x, \phi$ if and only if, for each $n \in D$, $(w, \nu\binom{n}{x}) \models \phi$.

Note that, according to these rules, the valuation ν affects only the value of the free variables of ϕ. Thus, the satisfiability of a sentence is independent of the chosen valuation. In this case, we may say that a word w satisfies the sentence ϕ if $(w, \nu) \models \phi$, for any valuation ν. Now, we present two fragments of the first-order logic, FO[<] and FO_2[<].

FO[<]. In this thesis, we consider only the set $\mathcal{E} = \{<\}$, where $<$ is the order on positions. In the rest of this thesis, FO[<] is simply called first-order logic.

Example 1.5. $\exists x, \forall z, x < y \Rightarrow (x = z) \vee \neg(x > z)$ is an FO[<] formula with one free variable: y. Both x and z are bound since all their occurrences are in the scope of the quantifiers over them.

The formula $\exists x, a(x) \wedge z < x \Rightarrow \forall y, z < y$ is also an FO[<] formula with one free variable, which is z. Indeed, the first occurrence of z is not in the scope of a quantifier over z.

In FO[<], the set $\mathcal{L}$ of relations on variables contains only the letter predicates and the binary relation symbol $<$. This symbol designates the usual order relation on a set of integers. The domain of this relation can be either $\mathbb{N}$, $\mathbb{Z}$ or a subset of one of these sets, and will be easy to determine depending on the context. Thus, we say that $(w, \nu) \models t_1 < t_2$ if and only if $\nu(t_1) < \nu(t_2)$. Moreover, the letter predicates are mapped to the indicator functions of their respective letters: for any letter $a \in \Sigma$, the relation symbol a is mapped to the function $\mathbb{1}_a : D \to \{true, false\}$, which returns true if and only if the input index is associated with a letter a.

Example 1.6. The word $w = aaaacba$ satisfies the sentence $\exists x, c(x) \wedge (\forall y, y < x \Rightarrow a(y))$. Indeed, if we assign the value 4 to x, we can observe that w_4 is the letter c and that all the letters before it hold a letter a.

The word $w = aaacabccbabbbbb$ and the valuation ν such that $\nu(x) = 3$ satisfy the formula $\exists y, \forall z, (z < x \Rightarrow a(z)) \wedge (y < z \Rightarrow b(z))$. Indeed, if we assign the value 9 to y, we can observe that all the letters before w_3 are occurrences of a and all the letters after w_9 are occurrences of b.

FO_2[<]. FO_2[<] is the subclass of FO[<] that uses at most two distinct variable names in its formulas. Note that this does not mean that these formulas can only consider two positions of a word, but two positions *at the same time*, which limits the possible comparisons between the positions.

Example 1.7. $\exists x, \exists y, x < y$ is an FO_2[<] sentence that describes the set of words of size at least 2.

$\exists x, a(x) \wedge \exists y, (x < y \wedge \exists x, (b(x) \wedge y < x))$ is an FO_2[<] sentence that describes the set of words of size at least 3 that have an a and a b separated by at least one position, which can hold any letter in the alphabet, including an a or a b. For example, on the alphabet $\Sigma = \{a, b, c\}$, the word $w = cabbcc$ satisfies that sentence, because we can choose $x = 1$ for the first occurrence of the variable x, $y = 2$ and $x = 3$ for the second occurrence of the variable x. Note that the two occurrences of x can have different values, since they are not bound by the same symbol $\exists$.

1.2 Algebra

Now that the basics of automata theory have been covered by section 1.1, we can move on to the main tools used in this thesis: semigroups. These tools, provided by the algebraic theory of automata, can represent a rational language, and their structure allows for a more refined analysis of the properties of the languages. This section presents semigroups, their link to rational languages, and some tools used to manipulate them.

1.2.1 Semigroups and monoids

A *semigroup* is a pair consisting of a set S and an associative binary operation $\cdot_S$ on S, called the inner operation of S. We usually write that the set S is a semigroup. A *monoid* is a triple $(M, \cdot_M, 1)$, where $(M, \cdot_M)$ is a semigroup and $1 \in M$ is an identity (or a neutral element) of $\cdot_M$. We usually write that the set M is a monoid. We only work with *finite* semigroups and monoids. We thus designate *finite semigroups* (resp. finite monoids) when we mention semigroups (resp. monoids).

Given a semigroup S, any element e of S satisfying $e \cdot_S e = e$ is called an *idempotent*. In a finite semigroup S, any element s of S admits an *idempotent power*, which is an element $\pi_S(s^n)$ (where $n > 0$ is an integer) that is idempotent, where s^n denotes the concatenation of n occurrences of s. We use the usual notation s^ω to denote the idempotent power of s (ω is the minimum integer such that, for any element $s \in S$, s^ω is the idempotent power of s). Given a semigroup S, we define $S^1 = S \cup 1$ as the monoid formed by the semigroup to which an identity is added if necessary. For any subsets X and Y of S, we denote by $X \cdot_S Y$ the set $\{x \cdot_S y \mid x \in X, y \in Y\}$. Similarly, for any $x \in S$ and $Y \subseteq S$, we write $x \cdot_S Y$ and $Y \cdot_S x$ respectively for $\{x\} \cdot_S Y$ and $Y \cdot_S \{x\}$. Given a finite set Σ, we call Σ^+ the *free semigroup over* Σ with the concatenation as the associative binary operation. This is the only infinite semigroup that we consider. Given a semigroup S, S^+ denotes the free semigroup with the underlying set S as alphabet.

Given a semigroup S and a subset $G \subseteq S$, we say that G *generates* S if the smallest sub-semigroup $T \subseteq S$ that contains G is S itself. Such a set G is called a set of *generators*, and is usually chosen to be minimal.

Canonical morphism. We denote concatenation implicitly: given two words u, v, their concatenation is written uv. For instance, taking two elements x, y of S, xy denotes a word of S^+ of length 2. This notation **must not** be confused with $x \cdot_S y$ that denotes the element of S obtained by multiplying x and y with the inner operation of S. We **never use** concatenation to mark the product within S. However, we relate words of the free semigroup S^+ to their value in S by means of *the canonical morphism*: $\pi_S \colon S^+ \to S$. It is the unique associative morphism verifying both the following properties: for every $x \in S$, $\pi_S(x) = x$ and, for every $u, v \in S^+$, $\pi_S(uv) = \pi_S(u) \cdot_S \pi_S(v)$.

Quotients and division. Let S_1 and S_2 be two semigroups. S_1 is said to be a *quotient* of S_2 if there exists a surjective morphism from S_2 to S_1. S_1 is said to *divide* S_2, and we write $S_1 \preccurlyeq S_2$, if S_1 is a quotient of a sub-semigroup of S_2.

1.2.2 Languages and semigroups

Syntactic semigroup. A link can be established between logics and semigroups by taking the *syntactic semigroup* of a language. This semigroup is defined as follows:

Definition 1.2

Given a language L on an alphabet Σ, the *syntactic congruence* of L in Σ^* is the relation $\sim_L$ defined on Σ^* such that, for any words $u, v \in \Sigma^*$, $u \sim_L v$ if and only if, for all words $x, y \in \Sigma^*$, $xuy \in L \Leftrightarrow xvy \in L$.

Definition 1.3

The *syntactic semigroup* of L is the quotient $\Sigma^+/\sim_L$, and the *syntactic monoid* of L is the quotient $\Sigma^*/\sim_L$.

When talking about the syntactic semigroup of a language, it is necessary to choose a way to represent the equivalence classes that form its elements. In this thesis, two options are used. The first one consists in representing each element by an arbitrary word in the corresponding equivalence class. The other one consists in choosing other arbitrary representatives, for example by associating numbers to the elements. This thesis favors this approach, as it is the one used in the code presented in chapter 2.

Given the syntactic semigroup S of a language L over an alphabet Σ, the set $G = \{g \in S \mid \exists a \in \Sigma, a \in g\}$ generates S, as any element of S corresponds to a word over Σ. In the rest of this thesis, the set G is used as the canonical set of generators of S, and its elements are simply called *the generators of* S.

The link between logics and semigroups has already been well studied and gave birth to very nice algebraic characterizations of some well-known classes of languages. For more information on this topic, see the survey [72] along with the books [70] and [60]. Notably, the class $FO[<]$, which is also the class of star-free languages (see section 1.3.1 for the formal definition) is equivalent to the variety **Ap** of aperiodic semigroups, the semigroups satisfying an equation of the form $x^{\omega+1} = x^\omega$, for any $x \in S$. We can also mention the class $\mathrm{FO}_2[<]$, which is equivalent to the variety **DA** of semigroups, the semigroups satisfying an equation of the form $\pi_S((xy)^\omega x(xy)^\omega) = \pi_S((xy)^\omega)$, for any $x, y \in S$.

Because of this strong link between semigroups and logics, it is important to define a notion of recognizability for semigroups.

Definition 1.4

A language L on an alphabet Σ is *recognized* by a monoid M (resp. a semigroup S) if there exists a monoid (resp. semigroup) morphism $\phi : \Sigma^* \to M$ (resp. $\phi : \Sigma^+ \to S$) and a subset $P \subseteq M$ (resp. $P \subseteq S$) such that

$$L = \phi^{-1}(P)$$

Note that, by definition, the syntactic monoid of a language L recognizes it, and its syntactic semigroup recognizes $L \setminus \{\epsilon\}$.

Transition semigroup. Another link can be established between logics and semigroups by considering the automata recognizing the languages defined by a logical fragment. The link with semigroups is then established by considering the transitions of the automata.

Definition 1.5

Let $A = (\Sigma, Q, I, \delta, F)$ be an automaton (DFA or NFA) without any ϵ-transition. We define the *transition* of a word $w \in \Sigma^*$ to be the function $\mathrm{tr}_w : Q \to 2^Q$ defined recursively as follows:

- For any letter $a \in \Sigma$, and for any state $q \in Q$ such that $\delta(q, a)$ is defined, $\mathrm{tr}_a(q) = \delta(q, a)$. Otherwise, $\mathrm{tr}_a(q) = \emptyset$.
- For any word $w \in \Sigma^+$ such that $w = w' \cdot b$, where $b \in \Sigma$ is the last letter of w then, for any state $q \in Q$, $\mathrm{tr}_w(q) = \{\mathrm{tr}_b(q') \mid q' \in \mathrm{tr}_{w'}(q)\}$.

With this definition, we can define the *transition monoid* of an automaton:

Definition 1.6

Let $A = (\Sigma, Q, I, \delta, F)$ be an automaton (DFA or NFA). The *transition monoid* of A is the monoid M_A defined as follows. First, the set of elements of the monoid is $\{\mathrm{tr}_w \mid w \in \Sigma^*\}$. Then, the inner product of the monoid is such that, for any two elements $m_1, m_2 \in M_A$ such that $m_1 = \mathrm{tr}_{w_1}$ and $m_2 = \mathrm{tr}_{w_2}$ for some words $w_1, w_2 \in \Sigma$, then $\pi_{M_A}(m_1 m_2) = \mathrm{tr}_{w_1 w_2}$.

Example 1.8. Consider the alphabet $\Sigma = \{a, b\}$ and the automata presented in Figure 1.5. The transitions of the letters are as follows:

$$\begin{array}{lcl@{\qquad}lcl} \mathrm{tr}_a(q_0) & = & \{q_0, q_1\} & \mathrm{tr}_b(q_0) & = & \{q_0\} \\ \mathrm{tr}_a(q_1) & = & \{q_2\} & \mathrm{tr}_b(q_1) & = & \{q_2\} \\ \mathrm{tr}_a(q_2) & = & \emptyset & \mathrm{tr}_b(q_2) & = & \emptyset \end{array}$$

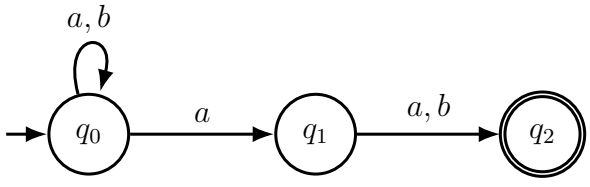


Figure 1.5 – An NFA recognizing $(a|b)^*a(a|b)$

With these transitions, we can build the transition semigroup of the automaton. To do so, we build the transitions of words of increasing size until all the words of the same size have a transition that we have already seen. First, consider the words of size 2, which are aa, ab, ba and bb. Their transitions are as follows:

$$\begin{array}{lcl@{\qquad}lcl} \mathrm{tr}_{aa}(q_0) & = & \{q_0, q_1, q_2\} & \mathrm{tr}_{ab}(q_0) & = & \{q_0, q_2\} \\ \mathrm{tr}_{aa}(q_1) & = & \emptyset & \mathrm{tr}_{ab}(q_1) & = & \emptyset \\ \mathrm{tr}_{aa}(q_2) & = & \emptyset & \mathrm{tr}_{ab}(q_2) & = & \emptyset \end{array}$$

$$\begin{array}{lcl@{\qquad}lcl} \mathrm{tr}_{ba}(q_0) & = & \{q_0, q_1\} & \mathrm{tr}_{bb}(q_0) & = & \{q_0\} \\ \mathrm{tr}_{ba}(q_1) & = & \emptyset & \mathrm{tr}_{bb}(q_1) & = & \emptyset \\ \mathrm{tr}_{ba}(q_2) & = & \emptyset & \mathrm{tr}_{bb}(q_2) & = & \emptyset \end{array}$$

All these transitions are different, and so they are distinct elements of the transition semigroup. We could go on to compute the transitions of the words of length 3. However, note that the state q_0 is necessarily in the transition of any word from the state q_0. Moreover, the transition of any word of length at least 2 from a state other than q_0 is necessarily empty. Taking these facts into account, we can observe that the transition of any word of length at least 2 is necessarily equal to one of the transitions of the words of length 2. For example, the transition of aba is equal to the transition of ba. Consequently, the transitions that we computed this far are exactly all the elements of the transition semigroup of the automaton in Figure 1.5.

To complete the computation of the transition semigroup, we need to give its inner product. By definition, we have the following:

$\pi_M(\mathrm{tr}_a \mathrm{tr}_a) = \mathrm{tr}_{aa}$
$\pi_M(\mathrm{tr}_a \mathrm{tr}_b) = \mathrm{tr}_{ab}$
$\pi_M(\mathrm{tr}_b \mathrm{tr}_a) = \mathrm{tr}_{ba}$
$\pi_M(\mathrm{tr}_b \mathrm{tr}_b) = \mathrm{tr}_{bb}$

If we compute the transitions of the words of length 3, we obtain the following:

$\pi_M(\mathrm{tr}_{aa} \mathrm{tr}_a) = \pi_M(\mathrm{tr}_a \mathrm{tr}_{aa}) = \mathrm{tr}_{aa}$
$\pi_M(\mathrm{tr}_{aa} \mathrm{tr}_b) = \pi_M(\mathrm{tr}_a \mathrm{tr}_{ab}) = \mathrm{tr}_{ab}$
$\pi_M(\mathrm{tr}_{ab} \mathrm{tr}_a) = \pi_M(\mathrm{tr}_a \mathrm{tr}_{ba}) = \mathrm{tr}_{ba}$
$\pi_M(\mathrm{tr}_{ab} \mathrm{tr}_b) = \pi_M(\mathrm{tr}_a \mathrm{tr}_{bb}) = \mathrm{tr}_{bb}$
$\pi_M(\mathrm{tr}_{ba} \mathrm{tr}_a) = \pi_M(\mathrm{tr}_b \mathrm{tr}_{aa}) = \mathrm{tr}_{aa}$
$\pi_M(\mathrm{tr}_{ba} \mathrm{tr}_b) = \pi_M(\mathrm{tr}_b \mathrm{tr}_{ab}) = \mathrm{tr}_{ab}$
$\pi_M(\mathrm{tr}_{bb} \mathrm{tr}_a) = \pi_M(\mathrm{tr}_b \mathrm{tr}_{ba}) = \mathrm{tr}_{ba}$
$\pi_M(\mathrm{tr}_{bb} \mathrm{tr}_b) = \pi_M(\mathrm{tr}_b \mathrm{tr}_{bb}) = \mathrm{tr}_{bb}$

With this, we have the entire description of the semigroup.

Note that the transition semigroup (or transition monoid) of a finite automaton is necessarily finite, as there can only be finitely many functions of the form $Q \to 2^Q$.

Remark 1.1. If we consider a DFA then, for any word $w \in \Sigma^*$, the function tr_w is actually of the more usual form $Q \to Q$. In that case, it is partial, as there may be some state q and letter a such that $\delta(q, a)$ is not defined.

Using the transition semigroup of a language is equivalent to using its syntactic semigroup, as pointed out by Proposition 1.1.

Proposition 1.1

Let L be a language. There is an isomorphism between the transition semigroup of the minimal automaton recognizing L and the syntactic semigroup of L.

For a proof of this property, see [60], chapter 1, section 4.3. The idea of this proof is to show that the syntactic semigroup necessarily divides any semigroup recognizing L, including the transition semigroup, then to show that the transition semigroup divides the syntactic semigroup.

Transition matrices. There are several equivalent definitions of the transition monoid of an automaton. One of these uses matrices to describe the transitions. This definition is the one we use in our implementation.

Definition 1.7

Let $A = (\Sigma, Q, I, \delta, F)$ be an automaton (DFA or NFA) without any ϵ-transition, and let $(q_0, \dots, q_{n-1})$ be an arbitrary ordering of Q. For any letter $a \in \Sigma$, the *transition matrix* of a is the n by n matrix $T_a = t_{0,0}, \dots, t_{0,n-1}, t_{1,0}, \dots, t_{n-1,n-1}$ such that, for any pair of integers $i, j \in [n]$, $t_{i,j} = 1$ if and only if $\delta(q_i, a)$ is defined and $q_j \in \delta(q_i, a)$. Otherwise, $t_{i,j} = 0$.

We can extend this definition to obtain a matrix equivalent to the transition of any word over Σ.

Definition 1.8

Let $A = (\Sigma, Q, I, \delta, F)$ be an automaton (DFA or NFA), and let $(q_0, \dots, q_{n-1})$ be an arbitrary ordering of Q. For any word $w = w_0 \cdots w_{n-1} \in \Sigma^n$, where $n > 0$, the *transition matrix* of w is the n by n matrix $T_w = T_{w_0} \times \cdots \times T_{w_{n-1}}$.

The transition of a word is equivalent to its transition matrix, as shown by the following lemma:

Lemma 1.1

Let $A = (\Sigma, Q, I, \delta, F)$ be an automaton (DFA or NFA), and let $(q_0, \dots, q_{n-1})$ be an arbitrary ordering of Q. For any word $w \in \Sigma^+$, if $T_w = t_{0,0}, \dots, t_{0,n-1}, t_{1,0}, \dots, t_{n-1,n-1}$ is the transition matrix of w then, for any pair of indices $i, j < n$, $t_{i,j} = 1$ if and only if $q_j \in \mathrm{tr}_w(q_i)$.

Proof. We prove the result by induction of the length of w. If $|w| = 1$, then w is composed of only one letter $a \in \Sigma$. By definition, for any pair of indices $i, j < n$, the set $\mathrm{tr}_a(q_i)$ contains the state q_j if and only if there is a transition from q_i to q_j labelled by a, or more formally, if and only if $\delta(q_i, a)$ is defined and $q_j \in \delta(q_i, a)$. Also by definition, $t_{i,j} = 1$ if and only if $\delta(q_i, a)$ is defined and $q_j \in \delta(q_i, a)$. This implies the result.

Now suppose that the result holds for any word of length at most n, and consider a word $w = w_0 \cdots w_n \in \Sigma^{n+1}$. By definition, $T_w = T_{w_0} \cdots T_{w_n} = T_{w_0 \cdots w_{n-1}} \times T_{w_n}$ and $\mathrm{tr}_w = \mathrm{tr}_{w_n} \circ \mathrm{tr}_{w_0 \cdots w_n}$. We denote by $u_{i,j}$ and $v_{i,j}$ the elements of respectively $T_{w_0 \cdots w_{n-1}}$ and T_{w_n}. Now, let $i, j \in [n+1]$ be two indices. Then, $t_{i,j} = 1$ if and only if there exists some index $k \in [n+1]$ such that $u_{i,k} = v_{k_j} = 1$. By induction hypothesis, this is equivalent to the following property: there exists an index $k \in [n+1]$ such that $q_k \in \mathrm{tr}_{w_0 \cdots w_{n-1}}(q_i)$ and $q_j \in \mathrm{tr}_{w_n}(q_k)$. By definition of the transition of a word, this is equivalent to saying that $q_j \in \mathrm{tr}_w(q_i)$, which concludes our proof. □

As a consequence of this lemma, we can construct the transition monoid of an automaton using the transition matrices of the words over Σ. To do so, we proceed similarly to what we do with the transitions of the words:

Corollary 1.1

Let $A = (\Sigma, Q, I, \delta, F)$ be an automaton (DFA or NFA). The transition monoid of A is equivalent to the monoid M defined as follows. First, the set of elements of the monoid

is $\{T_w \mid w \in \Sigma^*\}$. Then, the inner product of the monoid is such that, for any two elements $m_1, m_2 \in M$ such that $m_1 = T_{w_1}$ and $m_2 = T_{w_2}$ for some words $w_1, w_2 \in \Sigma$, then $\pi_M(m_1 m_2) = T_{w_1 w_2}$.

Remark 1.2. When the automaton is deterministic, only one element per line of the transition matrices can be set to 1. Consequently, it is possible to represent those transition matrices in a more concise way, using vectors of integers instead of matrices of bits. In that case, the transition of a word $w \in \Sigma^*$ would be the vector $v_0 \cdots v_{n-1}$ such that, for any index $i \in [n]$, $v_i = -1$ if the $(i+1)^{\text{th}}$ line of the matrix T_w contains no bit set to 1. Otherwise, $v_i = j$, where j is the unique index such that $t_{i,j} = 1$. Informally, v_i is the index of the unique state q_{v_i}, if it exists, that can be reached from q_i by reading the word w.

1.2.3 Green's relations

Consider a function $F\colon S \to \mathcal{P}(S)$, where $\mathcal{P}(S)$ denotes the powerset of S. We write $x \mathcal{F} y$ when $F(x) = F(y)$; $x \leq_{\mathcal{F}} y$ when $F(x) \subseteq F(y)$; and $x <_{\mathcal{F}} y$ when $x \leq_{\mathcal{F}} y$ and $F(x) \neq F(y)$. The relation $\mathcal{F}$ is an equivalence relation and $\leq_{\mathcal{F}}$ is a pre-order. We also write $\mathcal{F}(x) = \{y \mid y \mathcal{F} x\}$, the *$\mathcal{F}$-class of x*. We say that a semigroup is *$\mathcal{F}$-trivial* when $\mathcal{F}(x)$ is a singleton for any element $x \in S$. Green's relations are defined with the following functions: $R\colon x \mapsto x \cdot_S S^1$, $L\colon x \mapsto S^1 \cdot_S x$, $J\colon x \mapsto S^1 \cdot_S x \cdot_S S^1$, $H\colon x \mapsto R(x) \cap L(x)$. Note that the respective relations obtained from R, L, J and H are denoted by $\mathcal{R}$, $\leq_{\mathcal{R}}$, $<_{\mathcal{R}}$, $\mathcal{L}$, $\leq_{\mathcal{L}}$, $<_{\mathcal{L}}$, $\mathcal{J}$, $\leq_{\mathcal{J}}$, $<_{\mathcal{J}}$,$\mathcal{H}$, $\leq_{\mathcal{H}}$ and $<_{\mathcal{H}}$. In finite semigroups, the relation $\mathcal{J}$ is equal to the relation $\mathcal{D}$, which is the union of $\mathcal{L}$ and $\mathcal{R}$. From this relation $\mathcal{D}$ comes the name of the class **DA**, which indicates the class of semigroups whose regular $\mathcal{D}$-classes are aperiodic semigroups (see Proposition 1.2 for a characterization of aperiodic semigroups). An $\mathcal{F}$-class F, for $\mathcal{F}$ any of Green's relations, is said to be *regular* if some element $s \in F$ is idempotent, i.e. $s \cdot_S s = s$.

Example 1.9. Consider a finite semigroup S and four distinct elements $s, t, u, v \in S$ that have the following properties:

- There exist two elements $a, b \in S$ such that $s \cdot_S a = t$ and $t \cdot_S b = s$.
- There exist two elements $c, d \in S$ such that $u \cdot_S c = v$ and $v \cdot_S d = u$.
- There exist two elements $e, f \in S$ such that $e \cdot_S s = u$ and $f \cdot_S u = s$.

With these properties, we can deduce some relations between the four elements. First, we have $s\mathcal{R}t$. Indeed, the sets $R(s)$ and $R(t)$ are equal. This is due to the fact that it is possible to obtain t from s and s from t by multiplying them on the right with another element of S. Formally, this equality is shown by proving the two inclusions. We have $R(t) \subseteq R(s)$ since, for any element $t' \in R(t) = t \cdot_S S^1$, there exists an element $x \in S$ such that $t' = t \cdot_S x$. Consequently, we have $t' = s \cdot_S (a \cdot_S x)$, which implies that $t' \in R(s)$. Similarly, $R(s) \subseteq R(t)$ since, for any element $s' \in R(s) = s \cdot_S S^1$, there exists an element $y \in S$ such that $s' = s \cdot_S y$. Consequently, we have $s' = t \cdot_S (b \cdot_S y)$, which implies that $s' \in R(t)$. This proves that $R(s) = R(t)$, and so we have $s\mathcal{R}t$. The same reasoning can be applied to u and v to obtain $u\mathcal{R}v$.

Now, a similar reasoning gives us $s\mathcal{L}u$, since it is possible to obtain u from s and s from u by multiplying them on the left with another element of S. We can indeed show that $L(s) \subseteq L(u)$, and the other inclusion result follows with the same arguments. For any element $s' \in L(S) = S^1 \cdot_S s$, there exists an element $z \in S$ such that $s' = z \cdot_S s$. Consequently, we have

$s' = (z \cdot_S f) \cdot_S u$, which implies that $s' \in L(u)$. With the other inclusion obtained the same way, we have $s\mathcal{L}u$.

We showed that $s\mathcal{R}t$ and $s\mathcal{L}u$, which implies that $s\mathcal{D}t$, $s\mathcal{D}u$ and $t\mathcal{D}u$, since the relation $\mathcal{D}$ is the union of $\mathcal{R}$ and $\mathcal{L}$. Moreover, S is finite, so the relation D is equal to $\mathcal{J}$, and we have $t\mathcal{J}u$. This can be show by the same means as above, and here we show that $J(t) \subseteq J(u)$. For any element $t' \in J(t) = S^1 \cdot_S t \cdot_S S^1$, there exist two elements $x, y \in S$ such that $t' = x \cdot_S t \cdot_S y$. Consequently, we have $t' = x \cdot_S s \cdot_S a \cdot_S y = (x \cdot_S f) \cdot_S u \cdot_S (a \cdot_S y)$, which implies that $t' \in J(u)$. Consequently, we have $J(t) \subseteq J(u)$, and the other inclusion can be shown the same way.

Since $u\mathcal{R}v$, the element v belongs to the same $\mathcal{D}$-class and $\mathcal{J}$-class as u. Consequently, the elements s, t, u and v all belong to the same $\mathcal{D}$-class and $\mathcal{J}$-class.

A $\mathcal{D}$-class can be represented using an *egg-box*, where each row constitutes an $\mathcal{R}$-class and each column constitutes an $\mathcal{L}$-class. Consequently, each cell represents an $\mathcal{H}$-class. This representation is due to Green's Lemma, which establishes a multiplicative link between the elements of a given $\mathcal{D}$-class.

Lemma 1.2: Green's Lemma

Let S be a semigroup. Let $a, b \in S$ be such that $a\mathcal{R}b$. Then there exists $u, v \in S^1$ such that $au = b$ and $bv = a$. Let ρ_u and ρ_v be the right translations defined by $x\rho_u = xu$ and $x\rho_v = xv$. Then ρ_u and ρ_v preserve the $\mathcal{H}$-classes, i.e. for each pair of elements $x, y \in L(a)$ (resp. $L(b)$), $x\mathcal{H}y$ if and only if $x\rho_u\mathcal{H}y\rho_u$ (resp. $x\rho_v\mathcal{H}y\rho_v$).

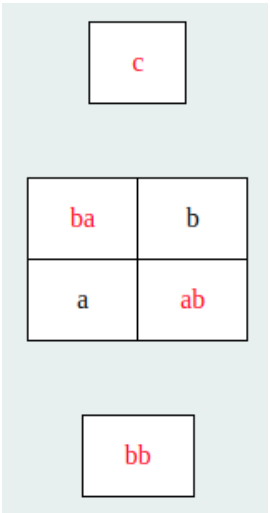


Figure 1.6 – The syntactic semigroup of $(ac^*b + c)^*$

Example 1.10. The syntactic semigroup of the language $(ac^*b + c)^*$ is represented in example 1.10. The idempotent elements are represented in red. Note that, in this semigroup, each $\mathcal{H}$-class is trivial, and so each of the boxes representing an $\mathcal{H}$-class contains only one element. In this figure, the set of elements of the semigroup is separated into three sets, that are the three $\mathcal{D}$-classes (and also the three $\mathcal{J}$-classes, since the relations $\mathcal{D}$ and $\mathcal{J}$ are equal in finite semigroups). One $\mathcal{D}$-class contains only the element c, which is a neutral element. Thus, this semigroup is also a monoid. Another $\mathcal{D}$-class contains the element corresponding to the words that cannot be extended, by adding letters at the beginning or at the end, to form a word in

the language; this element is bb, and corresponds to all the words of the form ac^*a or bc^*b. The third $\mathcal{D}$-class is more interesting, as it contains four elements:

- a represents all the words of the form $(ac^*b + c)^*ac^*$,
- b represents all the words of the form $(bc^*a + c)^*bc^*$,
- ab represents all the words of the form $(ac^*b + c)^*$,
- ba represents all the words of the form $(bc^*a + c)^*$

All these words either belong to the language or can be extended to form words in the language. More interestingly, they can all be extended to form a word represented by any of the four elements, which is why the four elements all belong to the same $\mathcal{J}$-class. These potential extensions are described in more details with the separation into $\mathcal{R}$-classes and $\mathcal{L}$-classes, respectively the lines and the columns of the box representing that D-class. For example, the elements a and ab are on the same line, meaning they belong to the same $\mathcal{R}$-class. Similarly, a and ba belong to the same $\mathcal{L}$-class. Green's Lemma then tells us that we can obtain equalities that describe the relations between these elements. For this $\mathcal{D}$-class, we obtain the following equalities: $ab \cdot_S a = a$, $ba \cdot_S b = b$, $a \cdot_S b = ab$, and $b \cdot_S a = ba$.

We can obtain respectively ab and a from the other one by multiplying them on the right with some semigroup element, and so they belong to the same $\mathcal{R}$-class. Similarly, we can obtain respectively ba and b from the other one by multiplying them on the left with some semigroup element, and so they belong to the same $\mathcal{L}$-class.

For more details on this lemma and the properties of the egg-box, see [61]. The consequences of this lemma are often used implicitly throughout this document.

The $\mathcal{J}$-depth of a semigroup. Let S be a semigroup. The *$\mathcal{J}$-depth* of a $\mathcal{J}$-class is the length of a maximal strictly decreasing sequence of $\mathcal{J}$-classes to it. Formally, given a semigroup S and a $\mathcal{J}$-class J, we say that J is of $\mathcal{J}$-depth i if there exist i $\mathcal{J}$-classes $J_1 >_\mathcal{J} J_2 \dots >_\mathcal{J} J_i$ such that $J_i = J$, but there exists no decreasing sequence $J'_1 >_\mathcal{J} J'_2 \dots >_\mathcal{J} J'_{i+1}$ such that $J'_{i+1} = J$. The $\mathcal{J}$-depth of a semigroup is the maximum $\mathcal{J}$-depth of its $\mathcal{J}$-classes. For any semigroup, there exists a unique $\mathcal{J}$-class of maximum $\mathcal{J}$-depth. Given d the $\mathcal{J}$-depth of S, for each integer i such that $1 \leq i \leq d$, we denote by $D_i(S)$ the union of all the $\mathcal{J}$-classes of depth i and we denote by $Q_i(S)$ the sub-semigroup[1] composed exactly of all the elements of S of $\mathcal{J}$-depth at least i.

Using the $\mathcal{J}$-depth, it is possible to represent a finite semigroup as its *egg-box representation*, since the relations $\mathcal{D}$ and $\mathcal{J}$ are equal for finite semigroups. This representation is composed of the egg-boxes of all the $\mathcal{D}$-classes, arranged in layers. Each layer is associated with a depth, increasing from top to bottom, and contains the egg-boxes of the $\mathcal{D}$-classes associated with that depth.

$\mathcal{J}$-constant words. Let S be a semigroup. A word $s_0 \cdots s_k$ in S^+ is *left $\mathcal{J}$-constant* if, for any index i such that $0 \leq i \leq k$, we have $\pi_S(s_0 \cdots s_i)\mathcal{J}s_0$. Symmetrically, $s_0 \cdots s_k$ is *right $\mathcal{J}$-constant* if, for any index i such that $0 \leq i \leq k$, $\pi_S(s_i \cdots s_k)\mathcal{J}s_k$. Finally, a word in S^+ is $\mathcal{J}$-constant if it is both left and right $\mathcal{J}$-constant. The latter property is equivalent to being a left $\mathcal{J}$-constant word $s_0 \cdots s_k$ such that $s_0\mathcal{J}s_k$.

[1] The notation uses the letter Q because $Q_i(S)$ is a quotient of S.

1.2.4 Cayley graphs

One possible way to represent a semigroup is through its left and right *Cayley graphs*. These two graphs have the same set of states, where each state corresponds to an element of the semigroup. The edges are labeled by the generators of the semigroup and represent a specific action of those generators on the elements of the semigroup. Namely, the left Cayley graph represents the results of the products of the form $g \cdot_S s$, and the right Cayley graph represents the results of the products of the form $s \cdot_S g$, where g is a generator of the semigroup and s is an element.

Definition 1.9

Let S be a semigroup and G a set of generators of S. The left Cayley graph of S with the generator set G is the tuple (G, Q, δ), where G is the alphabet, $Q = \{q_s \mid s \in S\}$ is the set of states, and the function $\delta : Q \times G \to Q$ is the transition function of the graph, defined such that, for any generator $g \in G$ and any element $s \in S$, $\delta(q_s, g) = q_{\pi_S(gs)}$.

Definition 1.10

Let S be a semigroup and G a set of generators of S. The right Cayley graph of S with the generator set G is the tuple (G, Q, δ), where G is the alphabet, $Q = \{q_s \mid s \in S\}$ is the set of states, and the function $\delta : Q \times G \to Q$ is the transition function of the graph, defined such that, for any generator $g \in G$ and any element $s \in S$, $\delta(q_s, g) = q_{\pi_S(sg)}$.

Example 1.11. Consider the monoid $M = \{1, a, b, c, d, e\}$ with the generators a and b, and the following inner product: $\pi_M(aa) = e$, $\pi_M(bb) = e$, $\pi_M(ab) = c$, $\pi_M(ba) = d$, $\pi_M(ac) = e$, $\pi_M(ad) = a$, $\pi_M(bc) = b$, $\pi_M(bd) = e$ and, for any element $m \in M$, $\pi_M(me) = \pi_M(em) = e$. The left Cayley graph of M is represented in Figure 1.7, and the right Cayley graph is represented in Figure 1.8.

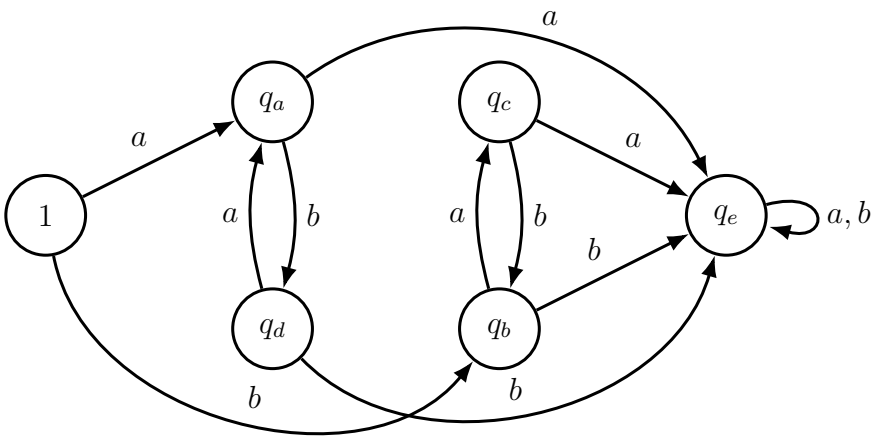


Figure 1.7 – The left Cayley graph

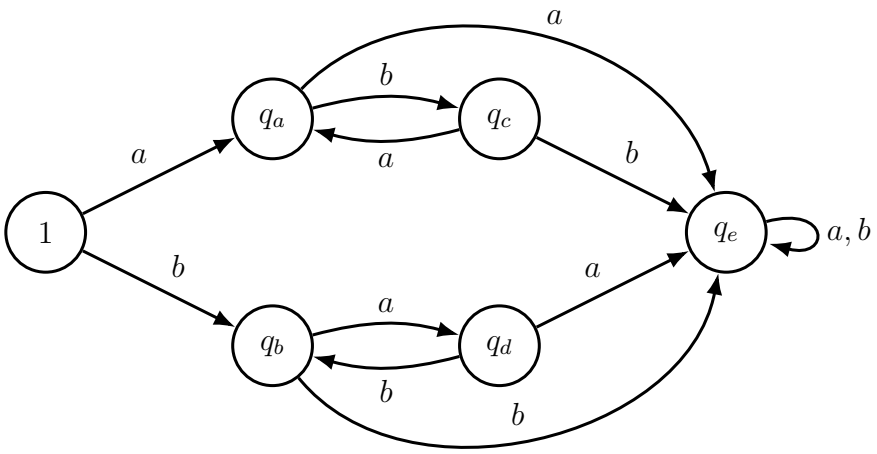


Figure 1.8 – The right Cayley graph

1.3 Characterization of some known classes of languages

This thesis focuses on two classes of rational languages: the languages in the first-order logic equipped with the order relation, and its subset composed of the languages that can be expressed using only two variables. These two classes of languages admit numerous equivalent characterizations and properties, some of which are presented in this section.

1.3.1 FO[<]

For the definition of FO[<], see section 1.1.5. The class FO[<] has been extensively studied and admits a wealth of equivalent classes. We present some of them in this section. For a full exposition of this subject, we refer to [19]. The equivalent characterizations of this class are summarized in the following theorem:

Theorem 1.2

Let Σ be an alphabet and $L \subseteq \Sigma^*$ be a language. The following propositions are equivalent:

- $L \in$ FO[<].
- L is star-free.
- $L \in$ LTL.
- The syntactic semigroup of L is aperiodic.

The star-free languages. For any finite alphabet Σ, the set of *star-free* languages of Σ^* is the smallest set $\mathcal{R} \subseteq \Sigma^*$ such that:

- $\mathcal{R}$ contains $\varnothing$, $\{\varepsilon\}$ and, for each $a \in \Sigma$, $\{a\}$.
- $\mathcal{R}$ is closed under finite union, finite product and complement.

The class of *star-free languages* is the union of the sets of the star-free languages of all possible alphabets.

By definition, every star-free language can be written as an expression using only the letters of the alphabet, the empty word ε, the empty set $\varnothing$, and the three operators union, product and complement.

In [48], McNaughton et al. showed that the set of star-free languages is exactly the class FO[<].

LTL. The *linear temporal logic* (LTL for short) is a modal temporal logic. Given an alphabet Σ, a formula in LTL is defined as follows:

- For each $a \in \Sigma$, the atomic proposition p_a is a formula.
- Given ϕ and ψ two well-formed LTL formulas, $\phi \vee \psi$, $\phi \wedge \psi$ and $\neg\phi$ are LTL formulas.
- Given ϕ and ψ two well-formed LTL formulas, $\mathrm{X}\phi$ (read as "next ϕ") and ϕ U ψ (read as "ϕ until ψ") are LTL formulas.

The semantics of LTL formulas are defined by induction on these formation rules. Given a word $w \in \Sigma^+$ and an index $i \in [|w|]$, we define the relation "(w, i) satisfies the formula ϕ", written $(w, i) \vDash \phi$ by induction on ϕ, according to the following rules:

- Given $a \in \Sigma$, $(w, i) \vDash p_a$ if the i^{th} letter of w is an a.
- Given two formulas ϕ and ψ, $(w, i) \vDash \phi \vee \psi$ (resp. $\phi \wedge \psi$, resp. $\neg\phi$), if $(w, i) \vDash \phi$ or $(w, i) \vDash \psi$ (resp. $(w, i) \vDash \phi$ and $(w, i) \vDash \psi$, resp. $\neg((w, i) \vDash \phi)$).
- Given a formula ϕ, $(w, i) \vDash \mathrm{X}\phi$ if $i + 1 < |w|$ and $(w, i + 1) \vDash \phi$.
- Given a formula ϕ, $(w, i) \vDash \phi$ U ψ if there exists an index j such that $i \leq j < |w|$, $(w, j) \vDash \psi$ and, for each index k such that $i \leq k < j$, $(w, k) \vDash \phi$

Given ϕ an LTL formula, we say that a word w *satisfies* ϕ if $(w, 0) \vDash \phi$. Then, the *language* of ϕ is defined to be the set $\{w \mid (w, 0) \vDash \phi\}$.

For readability purposes, we define additional operators:

- The logical operator true is such that, for any atomic proposition p_a, true $\equiv p_a \vee \neg p_a$.
- The operator F ("finally") is defined by $\mathrm{F}\phi \equiv$ true U ϕ.

Example 1.12. Consider the LTL formula $\phi = p_a \wedge \mathrm{X}(p_c \text{ U } (p_b \wedge \mathrm{X}\text{false}))$. This formula recognizes words that start with an occurrence of a, and then have only occurrences of c until the last letter, which is a b. This is equivalent to the rational expression ac^*b. Thus, the formula ϕ recognizes the words ab and $acccb$, but not the word $acacb$.

In his thesis, Kamp proved that LTL is as expressive as FO[<] on Dedekind-complete orderings [39]. That result was later improved by Gabbay et al. [24], who showed that the languages in LTL are exactly the languages in FO[<].

With this construction, LTL is sometimes called *Forward LTL*, as its operators use only information from the future parts of the word to give the result. Other temporal operators can be added to change that:

- Given a formula ϕ, $(w, i) \vDash \mathrm{Y}\phi$ (read as "yesterday ϕ") if $i - 1 \geq 0$ and $(w, i - 1) \vDash \phi$.

- Given a formula ϕ, $(w, i) \vDash \phi \text{ S } \psi$ (read as "ϕ since ψ") if there exists an index j such that $0 \leq j \leq i$, $(w, j) \vDash \psi$ and, for each index k such that $j < k \leq i$, $(w, k) \vDash \phi$

The temporal logic that uses only the boolean operators and the temporal operators "yesterday" and "since" is called *Past-time Linear Temporal Logic* (Past LTL for short), and the temporal logic that uses both past-time and future-time temporal operators is called TL. Note that, for Past LTL formulas, the condition to satisfy the formula must change to match the change of direction of the formula. Thus, given ϕ a Past LTL formula, we say that a word w *satisfies* ϕ if $(w, |w| - 1) \vDash \phi$. As a global rule, this acceptance condition depends on the first operator of the formula. Adding past-time operators does not increase the expressivity. Indeed, Gabbay et al. [24, Theorem 2.3] showed that any formula in TL is equivalent to a Boolean combination of formulas in the union of Past LTL and Forward LTL. This result implies that Forward LTL and TL are exactly the same class. Indeed, given such a boolean combination, we can re-write it as a boolean combination of formulas in Forward LTL. To do so, we just need to remark that each formula in Past LTL, just like the formulas in Forward LTL, introduces an order on some particular letters of the words. The difference is that a Past LTL formula uses the order from right to left, when a Forward LTL formula uses it from left to right. Thus, using that order, it is possible to write an equivalent Forward LTL formula.

Example 1.13. Consider the Forward LTL formula $\phi = p_a \wedge \text{X}(p_c \text{ U } (p_b \wedge \text{Xfalse}))$. It is equivalent to the rational expression ac^*b. That rational can also be recognized by a Past LTL formula, which is $\psi = p_b \wedge \text{Y}(p_c \text{ S } (p_a \wedge \text{Yfalse}))$.

Aperiodic semigroups. The class of *aperiodic semigroups*, called **Ap**, is the variety of semigroups satisfying an equation of the form $\pi_S(x^{\omega+1}) = \pi_S(x^\omega)$, for any $x \in S$. This variety verifies the following characterization:

Proposition 1.2

Let S be a semigroup. The following conditions are equivalent (see [65]):

- S is aperiodic.
- There exists an integer ω such that for all $s \in S$, $\pi_S(s^\omega) = \pi_S(s^{\omega+1})$.
- All $\mathcal{H}$-classes of S are trivial.

Thanks to this property, when considering an aperiodic semigroup, we only need to know the $\mathcal{H}$-class of an element to precisely characterize that element. This fact will be extensively used in the proofs of our main results.

In [65], Schützenberger proved that the rational languages which syntactic monoids are finite and aperiodic are exactly the star-free languages.

The class **Ap** admits an interesting property that is used in the proofs of section 4.2. The property states that the product of a $\mathcal{J}$-constant word is equal to the product of its first and last letters, which allows for some products to be computed quicker.

Lemma 1.3

Let S be an aperiodic semigroup. Suppose that $u = s_0 \cdots s_k \in S^+$ is a $\mathcal{J}$-constant word. Then, $\pi_S(u)$ is the unique element of $R(s_0) \cap L(s_k)$. If $k > 0$, this also implies that

$\pi_S(u) = \pi_S(s_0 \cdot s_k)$.

Proof. Since S is aperiodic, and thanks to Proposition 1.2, we know that $\pi_S(u)$ is the unique element of $R(u) \cap L(u)$, so we only have to find $R(\pi_S(u))$ and $L(\pi_S(u))$. By hypothesis, u is $\mathcal{J}$-constant, so $J(\pi_S(u)) = J(s_0) = J(s_k)$. By definition of R and L, this implies that $R(\pi_S(u)) = R(s_0)$ and $L(\pi_S(u)) = L(s_k)$. Thus we have that $\pi_S(u)$ is the unique element of $R(s_0) \cap L(s_k)$. If $k > 1$, then s_0 and s_k are separate occurrences of elements of S, so $\pi_S(u) = \pi_S(s_0 \cdot s_k)$, which is the unique element of $R(s_0) \cap L(s_k)$. □

1.3.2 $\mathrm{FO}_2[<]$

The class $\mathrm{FO}_2[<]$ has been extensively studied and admits a wealth of equivalent classes. We present some of them in this section. For a full exposition of this subject, we refer to [71]. The equivalent classes are summarized in the following theorem:

Theorem 1.3

Let Σ be an alphabet and $L \subseteq \Sigma^*$ be a language. The following propositions are equivalent:

- $L \in \mathrm{FO}_2[<]$.
- $L \in \mathrm{UTL}$.
- The syntactic semigroup of L is in **DA**.
- There exists a turtle program that recognizes L.

UTL. *Unary Temporal Logic* (UTL for short) is the fragment of LTL which uses only two operators, that are restricted versions of the Until and Since temporal operators. Given an alphabet Σ, a formula in UTL is defined as follows:

- For each $a \in \Sigma$, the atomic proposition p_a is a formula.
- Given ϕ and ψ two well-formed LTL formulas, $\phi \vee \psi$, $\phi \wedge \psi$ and $\neg\phi$ are LTL formulas.
- Given ϕ a well-formed LTL formulas, $\mathrm{F}\phi$ (read as "finally ϕ") and $\mathrm{P}\phi$ (read as "past ϕ") are LTL formulas.

The semantics of UTL formulas are defined by induction on these formation rules. The operators F and P are defined from U and S as follows: $\mathrm{F}\phi \equiv \text{true U } \phi$ and $\mathrm{P}\phi \equiv \text{true S } \phi$. Thus, given a word $w \in \Sigma^+$ and an index $i \in [|w|]$, we define the relation $(w, i) \vDash \phi$ as follows:

- Given $a \in \Sigma$, $(w, i) \vDash p_a$ if the i^{th} letter of w is an a.
- Given two formulas ϕ and ψ, $(w, i) \vDash \phi \vee \psi$ (resp. $\phi \wedge \psi$, resp. $\neg\phi$), if $(w, i) \vDash \phi$ or $(w, i) \vDash \psi$ (resp. $(w, i) \vDash \phi$ and $(w, i) \vDash \psi$, resp. $\neg((w, i) \vDash \phi)$).
- Given a formula ϕ, $(w, i) \vDash \mathrm{F}\phi$ if there exists an index j such that $i < j < |w|$ and $(w, j) \vDash \phi$.
- Given a formula ϕ, $(w, i) \vDash \mathrm{P}\phi$ if there exists an index j such that $0 \leq j < i$ and $(w, j) \vDash \phi$.

It is easy to build an $\mathrm{FO}_2[<]$ formula equivalent to a given UTL formula, as shown in example 1.14.

Example 1.14. Consider the UTL formula $\phi = (p_c \vee p_a) \wedge (\mathrm{F}(p_b) \wedge (\mathrm{P}(\neg p_d) \wedge \mathrm{F}(p_c)))$. This formula expresses the following property: the first letter must be an occurrence of either c or a, followed at some future index by an occurrence of b. That occurrence of b must be preceded by an occurrence of d, which itself must be followed by an occurrence of c. This is equivalent to the $\mathrm{FO}_2[<]$ formula $(c(0) \vee a(0)) \wedge \exists i, (0 < i \wedge d(i) \wedge \exists j, (i < j \wedge b(j)) \wedge \exists j, (i < j \wedge c(j)))$.

Etessami et al. [20, Theorems 1 and 2] showed that UTL is equivalent to $\mathrm{FO}_2[<]$ by proving that it is possible to build a UTL formula equivalent to any given $\mathrm{FO}_2[<]$ formula. However, it is important to note that they also showed that there is necessarily an exponential blow-up between $\mathrm{FO}_2[<]$ and UTL. Indeed, they gave a sequence ϕ_n of $\mathrm{FO}_2[<]$ formulas such that the shortest equivalent UTL formulas are of size $2^{\Omega(n)}$.

DA. The class **DA** is the variety of semigroups satisfying the equation $\pi_S((xyz)^\omega y (xyz)^\omega) = \pi_S((xyz)^\omega)$. This is equivalent to saying that any semigroup S is in **DA** if and only if any regular $\mathcal{D}$-class D of S is an aperiodic semigroup.

In our proofs involving semigroups in **DA**, we will rely on some classical equivalent characterizations of the variety **DA**.

Proposition 1.3

Let M be a monoid. The following conditions are equivalent:

- M is in the variety **DA**.
- If J is a regular $\mathcal{J}$-class of M, then J is an aperiodic semigroup.
- $\forall x, y, z \in M, (xyz)^\omega y (xyz)^\omega = (xyz)^\omega$ (we will refer to this as the algebraic characterization of **DA**) (see [71, Theorem 2.]).

By definition, any semigroup in **DA** is aperiodic, so we will also be able to use the characterizations presented in Proposition 1.2.

One of the interesting properties of **DA** is that, in a $\mathcal{D}$-class, if any element is idempotent, then all the elements in that class are idempotent.

Proposition 1.4

Let S be a finite semigroup in **DA** and D a $\mathcal{D}$-class of S. If D is regular then, for any element $s \in D$, s is idempotent.

Proof. To prove this property, suppose that D is regular. Let s be an element of D. Thanks to proposition 1.3, we know that D is an aperiodic semigroup. As a consequence of proposition 1.2, for some integer $\omega \geq 1$, $\pi_S(s^{\omega+1}) = \pi_S(s^\omega)$, and so $\pi_S(s^{2\omega}) = \pi_S(s^\omega)$. Thus, $\pi_S(s^\omega)$ is idempotent. Moreover, $s \cdot_S \pi_S(s^{\omega-1}) = \pi_S(s^\omega)$ and $\pi_S(s^{\omega-1}) \cdot_S s = \pi_S(s^\omega)$, so $s \geq_\mathcal{R} \pi_S(s^\omega)$ and $s \geq_\mathcal{L} \pi_S(s^\omega)$. Finally, since D is a semigroup composed of only one $\mathcal{D}$-class, s^ω is in the same $\mathcal{D}$-class as s, so $s \mathcal{R} \pi_S(s^\omega)$ and $s \mathcal{L} \pi_S(s^\omega)$, which is equivalent to $s \mathcal{H} \pi_S(s^\omega)$. This allows us to conclude, as D is an aperiodic semigroup, so proposition 1.2 implies that its $\mathcal{H}$-classes are trivial, which in turn gives us that $s = \pi_S(s^\omega)$. Since $\pi_S(s^\omega)$ is idempotent, this concludes the proof. □

Thérien and Wilke [73] showed that **DA** is equivalent to $\mathrm{FO}_2[<]$. Moreover, the class **DA** admits the following nice property that is used in the proofs of section 4.3.

Lemma 1.4

Let S be a semigroup in **DA** and R an $\mathcal{R}$-class of S. Then there exist two sets $T, K \subseteq S$ such that $S = T \uplus K$ and, for all $x \in R$, we have

- $\forall s \in T, xs\mathcal{R}x$
- $\forall s \in K, xs <_{\mathcal{J}} x$

Moreover, both T and K are sub-semigroups of S such that, if we denote by J the $\mathcal{J}$-class containing R, then $J \subseteq T$ if R is regular and $J \subseteq K$ otherwise. It follows that if S is a monoid, then T is also a monoid.

Proof. First, we prove that the sets T and K exist and are well defined. Consider $s \in S$ and $x, y \in R$. We need to prove that $xs\mathcal{R}x \Leftrightarrow ys\mathcal{R}y$. Note that, since $x\mathcal{R}y$, there exist $p, q \in S$ such that $xp = y$ and $yq = x$. Suppose that $xs\mathcal{R}x$. Then, we want to prove that $ys\mathcal{R}y$. By definition, $ys \leq_{\mathcal{R}} y$, so we only need to find some element $z \in S$ such that $ysz = y$. Since $xs\mathcal{R}x$, $\exists r \in S, xsr = x$. Thus, $yqsrp = y$ and $y(qsrp)^\omega = y$. Since S is in **DA**, we have $(qsrp)^\omega = (qsrp)^\omega s(qsrp)^\omega$, so we have $y(qsrp)^\omega s(qsrp)^\omega = y$. Since $y(qsrp)^\omega = y$, this implies that $ys(qsrp)^\omega = y$, which proves that $ys\mathcal{R}y$. Since x and y have symmetric roles, the converse follows directly. Thus, the set T exists and is well defined, and the set $K = S \setminus T$ is such that, $\forall s \in K$, $\forall x \in S$, $\neg(xs\mathcal{R}x)$. By definition of the $\mathcal{J}$-classes, we necessarily have $xs \leq_{\mathcal{J}} x$, so $xs <_{\mathcal{J}} x$, which shows that the set K has the wanted property.

Now, we prove that T is a sub-semigroup of S: let a and b be two elements of T. By definition of T, for any $x \in R$, $xa\mathcal{R}x$ and $xb\mathcal{R}x$; thus there exist $p, q \in S$ such that $xap = x$ and $xbq = x$. So, $xapbq = x$ and $x(apbq)^\omega = x$. Since S is in **DA**, it is aperiodic, so $x(apbq)^\omega = x(apbq)^{\omega+1}$. Then, we can get one letter a out of the parentheses as follows: $x(apbq)^{\omega+1} = xa(pbqa)^\omega pbq$. Now, we can isolate a letter b: since S is in **DA**, we have $xa(pbqa)^\omega pbq = xa(pbqa)^\omega b(pbqa)^\omega pbq$. To summarize our results so far, we have $x = xa(pbqa)^\omega b(pbqa)^\omega pbq$. Now, remember that $xapbq = x$. Thus, $xapbqa = xa$, and $xa(pbqa)^\omega = xa$. This way, we can get rid of the leftmost factor $(pbqa)^\omega$ in our result and obtain the following equality: $x = xab(pbqa)^\omega pbq$. This proves that $xab\mathcal{R}x$. Consequently, ab is also in T and T is a semigroup. To prove that K is a sub-semigroup of S, we remark that, for any elements $x \in R$, $s \in K$ and $t \in S$, we have $\pi_S(xst) \leq_{\mathcal{J}} \pi_S(xs) <_{\mathcal{J}} x$. Thus, the set K is an ideal of S, and so it is a sub-semigroup.

Finally, thanks to Proposition 1.4, we know that if R is regular, then all the elements of its $\mathcal{J}$-class are idempotent, so $J \subseteq T$. On the contrary, if R is not regular, then no element of J can be idempotent, so $J \subseteq K$. □

Turtle programs. Turtle programs were introduced by Schwentick et al. to characterize the expressive power of $\mathrm{FO}_2[<]$. A *turtle instruction* on an alphabet Σ is a pair (d, a), for any direction $d \in \{\rightarrow, \leftarrow\}$ and any letter $a \in \Sigma$. For readability purposes, we use the notations X_a and Y_a respectively for $(\rightarrow, a)$ and $(\leftarrow, a)$. A *turtle program* is a sequence of turtle instructions starting either at the beginning or the end of a word.

We use the symbol $\perp$ to denote an undefined position in a word. Given an alphabet Σ, a word $w \in \Sigma^*$ and two letters $a_1, a_2 \in \Sigma$, we define the effect of the turtle instructions X_{a_1} and Y_{a_2} as follows:

- For any position $i \in \{-1, \ldots, |w|-1\}$, $X_{a_1}(w, i)$ is the unique pair (w, j) such that $j > i$ or $j = \perp$ and, if $j \neq \perp$, then $w_j = a_1$ and, for each index $k \in \{i+1, \ldots, j-1\}$, $w_k \neq a_1$.
- For any position $i \in \{0, \ldots, |w|\}$, $Y_{a_2}(w, i)$ is the unique pair (w, j) such that $j \in \{0, \ldots, i-1\} \cup \{\perp\}$ and, if $j \neq \perp$, then $w_j = a_2$ and, for each index $k \in \{j+1, \ldots, i-1\}$, $w_k \neq a_2$.
- $X_{a_1}(w, \perp) = Y_{a_2}(w, \perp) = (w, \perp)$

In other words, from a position (w, i), the instruction X_{a_1} jumps to the first position (w, j) on the right such that $w_j = a_1$, or to $(w, \perp)$ if that position does not exist. Similarly, the instruction Y_{a_2} jumps to the first position (w, j) on the left such that $w_j = a_2$, or to $(w, \perp)$ if that position does not exist.

Given a turtle program $P = I_1 \cdots I_n$ and a word $w \in \Sigma^*$, we define $P(w)$ to be

- $I_n(\ldots I_1(w, -1)\ldots)$ if I_1 is of the form $(\rightarrow, a)$
- $I_n(\ldots I_1(w, |w|)\ldots)$ if I_1 is of the form $(\leftarrow, a)$

Note that this definition ensures that we start at the beginning of the word if the turtle instruction I_1 goes from left to right, and at the end of the word if it goes from right to left.

A word w is said to be *accepted* by a turtle program P if $P(w) \neq \perp$. We define $L(P)$ to be the set of all words accepted by P. In [66], Schwentick et al. showed that the set of languages which syntactic monoids are in **DA** is exactly the set of languages that can be recognized by boolean combinations of turtle programs. We denote the set of all possible boolean combinations of turtle programs as $\mathrm{TL}[X_a, Y_a]$.

1.4 Boolean circuits

The goal of this thesis is to use the algebraic theory of automata to produce vectorized algorithms, which would process input words using a form of parallelism. The vectorized algorithms presented in this thesis are inspired from boolean circuits, which are a theoretical tool used to describe algorithms, and notably to represent which parts of these algorithms may be performed in parallel.

1.4.1 Definitions

A *boolean circuit* is a directed acyclic graph, where the nodes are called *gates*. The *input gates* are the source nodes of the graph, while the *output gates* are its sink nodes. The edges of the graph are called *wires*. Each gate, except for the input gates, is associated with a *boolean function* $f : \{0, 1\}^r \to \{0, 1\}$, where r is the *fan-in* of the gate, which is the number of edges that enter the gate. Boolean functions can be represented by their *truth table*, which exhaustively list all possible inputs to the function and their corresponding outputs. In cases where the function f is not symmetric, an order on these edges is required.

Example 1.15. The circuit represented by Figure 1.9 has four input gates: b_0, b_1, b_2 and b_3. It has only one output gate, which is the one labelled $\wedge$. The boolean functions associated with the gates are as follows:

- The gate labelled by $\vee$ is of fan-in 3, and is associated with the function $f_\vee : \{0, 1\}^3 \to \{0, 1\}$ defined by $f(b_0, b_1, b_2) = b_0 \vee b_1 \vee b_2 = \begin{cases} 1 \text{ if } b_i = 1 \text{ for some } i \in \{0, 1, 2\} \\ 0 \text{ otherwise} \end{cases}$
- The gate labelled by $\neg$ is of fan-in 1, and is associated with the function $f_\neg : \{0, 1\} \to \{0, 1\}$ defined by $f(b) = \neg b = \begin{cases} 1 \text{ if } b = 0 \\ 0 \text{ otherwise} \end{cases}$
- The gate labelled by $\wedge$ is of fan-in 2, and is associated with the function $f_\wedge : \{0, 1\}^2 \to \{0, 1\}$ defined by $f(b_0, b_1) = b_0 \wedge b_1 = \begin{cases} 1 \text{ if } b_0 = b_1 = 1 \\ 0 \text{ otherwise} \end{cases}$

Note that all the boolean functions used here are symmetric.

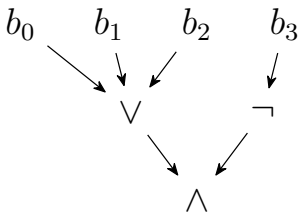


Figure 1.9 – A simple boolean circuit

The *size* of a circuit refers to the number of non-input gates it contains, while the *depth* is defined as the length of the longest oriented path within the circuit. Additionally, the *fan-out* of a gate represents the number of edges exiting that gate.

Example 1.16. The circuit in Figure 1.9 is of size 3 and of depth 2. All the gates in that circuit are of fan-out 1.

In a circuit with n input gates labeled by booleans $b_0, \ldots, b_{n-1}$, each gate g can be associated with a function $F_g : \{0, 1\}^n \to \{0, 1\}$, mapping the inputs of the circuit to the output of the gate. That function is defined by induction on the depth of the gate g. For the base case, each input gate g labelled by b_i is associated with the function F_g such that $F_g(b_0, \ldots, b_{n-1}) = b_i$. In other words, the output of the gate g is equal to the corresponding boolean input. Any other gate g of fan-in r is associated with F_g such that

$$F_g(b_0, \ldots, b_{n-1}) = f_g(F_{g_0}(b_0, \ldots, b_{n-1}), \ldots, F_{g_{r-1}}(b_0, \ldots, b_{n-1}))$$

This recursive definition enables the computation of the output of any gate within the circuit. Therefore, a circuit with n input gates and m output gates is associated with a function $F : \{0, 1\}^n \to \{0, 1\}^m$. When $m = 1$, we say that the circuit *accepts* any input $b_0 \cdots b_{n-1}$ such that $F(b_0, \ldots b_{n-1}) = 1$. Such circuits *recognize* the set of all the accepted inputs.

Example 1.17. The circuit in Figure 1.9 computes the function $F_\wedge : \{0, 1\}^4 \to \{0, 1\}$ such that $F_\wedge(b_0, b_1, b_2, b_3) = (b_0 \vee b_1 \vee b_2) \wedge \neg b_3$. Thus, that circuit recognizes the set of words $\{b_0b_1b_2b_3 \in \{0, 1\}^4 \mid (b_0 \vee b_1 \vee b_2) \wedge \neg b_3 = 1\}$, which is the set of words of length 4 such that the last bit is 0, and at least one of the first three bits is 1.

In our boolean circuits, we also authorize the use of *constant gates*. A constant gate is an input gate with a constant value assigned to it, regardless of the values of the inputs. Thus, our circuits can use the constant values 0 and 1.

Remark 1.3. Constant gates do not impact the expressivity of the circuits, as they can be obtained from any input value with a circuit of constant size. Indeed, for any boolean b, $b \vee \neg b = 1$ and $b \wedge \neg b = 0$. In boolean circuits, constant gates only provide a convenient way of drawing more readable circuits.

Extending circuits to arbitrary alphabets. It is possible to consider more general circuits. Given a fixed alphabet Σ, we say that a boolean circuit C *recognizes* a set of words in Σ^n when it has a unique output gate and there is a bijection between the letters $a_0, \dots, a_{p-1}$ of Σ and the input gates $b_0, \dots, b_{q-1}$. For example, we can represent the alphabet $\Sigma = \{a, b, c\}$ by three booleans b_a, b_b and b_c, which are equal to 1 if the letter is equal to, respectively, a, b, or c. This is called the *one-hot encoding*: a variable with n possible states is encoded on n bits, among which only one at a time can be equal to 1. We could then represent a word of size n by $3n$ booleans: $b_{a,0}, b_{b,0}, b_{c,0}, \dots, b_{a,n-1}, b_{b,n-1}, b_{c,n-1}$. As a shorthand, we write $\mathrm{enc}(u)$ for the tuple of booleans encoding the input word u, and thus $C(\mathrm{enc}(u))$ for the output of C corresponding to the input $\mathrm{enc}(u)$.

Boolean circuits can also represent functions f from Σ^n to a finite domain E. It suffices to consider circuits C which have not only a bijection between their input gates and the letters of Σ, but also a bijection between the elements $e_0, \dots, e_{r-1}$ of E and their output gates $e_0, \dots, e_{s-1}$.

1.4.2 Classes of boolean circuits

One circuit with n input gates can only accept words of length n. Thus, to recognize a given language L, we need a *family* of circuits, where no two circuits have the same number of inputs. In such a family of circuits, the circuit with n inputs recognizes the language $L \cap \{0, 1\}^n$.

Example 1.18. It is possible to generalize the circuit in Figure 1.9 to recognize the language of words of any size such that the last bit is 0 and at least one of the others is 1. In symbols, this is the language $L = \{b_0 \cdots b_{n-1} \in \{0, 1\}^+ \mid \bigvee_{i=0}^{n-2} b_i = 1 \text{ and } b_{n-1} = 0\}$. If we allow ourselves to use gates of arbitrary fan-in, this can be done easily, as it suffices to replace the gate labelled $\vee$ by another gate $\vee$ of appropriate fan-in. Then, the circuit in Figure 1.10 recognizes $L \cap \{0, 1\}^4$, the words of length 4 in the language, and the circuit in Figure 1.10 recognizes $L \cap \{0, 1\}^6$.

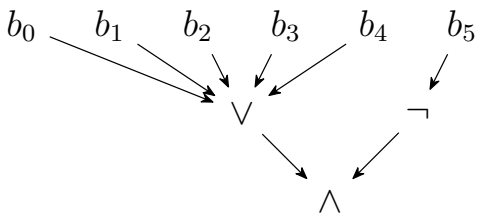


Figure 1.10 – The circuit for words in $L \cap \{0, 1\}^6$

Families of circuits can be studied through *circuit complexity*, which has been introduced by Shannon [68]. Circuit complexity categorizes families of circuits depending on the relations between the index of a circuit in a family and parameters such as the number of gates, the number of wires, or the depth. This leads to a hierarchy of circuits classes, some of which are

presented in this section. But before presenting these classes, let's talk about another feature of circuit families, which separates each class of circuits in two significantly different classes: uniformity.

Uniformity. A family of circuits $(C_n)_{n\geq 0}$ can be arbitrary. Notably, if we have a set of integers $N \subseteq \mathbb{N}$, we can recognize the language $L = \{w \in \{0,1\}^* \mid |w| \in N\}$ using the following family $(C_n)_{n\in\mathbb{N}}$:

- C_0 is the constant gate equal to 1 if $0 \in N$, and to 0 otherwise.
- For each $n \in \mathbb{N}^+$, if we denote the input by $b_0 \cdots b_{n-1}$, C_n is the circuit $b_{n-1} \vee \neg b_{n-1}$ if $n \in N$, and the circuit $b_{n-1} \wedge \neg b_{n-1}$ otherwise.

This way, we can recognize a non-recursive language using a family of circuits of constant depth. The goal of defining *uniform* circuit families is to avoid such behavior and make the complexity of those families closer to the complexity of Turing machines. The idea is that the circuits of a uniform family, including their complexity, should depend on the value of the index n, following some rule or algorithm that ensures all circuits in the family to have the same structural properties. Thus, a circuit family is *uniform* if there exists a deterministic algorithm that, for each $n \in \mathbb{N}$, provides an explicit description of the n^{th} circuit of the family.

Various notions of uniformity can be used by restricting the resources required by the algorithm generating the circuits. One classical notion requires the circuit to be generated by a deterministic Turing machine running in logarithmic time, while other definitions require the circuits' wires to be described in some logical formalisms. We refer to [75] for a complete description of the various notions of uniformity.

NC¹. NC^1 is the class of functions that can be computed by a family $(C_n)_{n\geq 0}$ of circuits following some conditions. First, all the circuits C_n are built from gates labelled only by the boolean functions $\neg$, $\vee$ and $\wedge$. The fan-in of these gates is as follows: one for $\neg$, and two for $\vee$ and $\wedge$. For each $n \geq 0$, the circuit C_n has n input boolean values. Moreover, there exists $k > 0$ such that, for any $n \geq 0$, the depth of the circuit C_n is less than $k \cdot \log_2(n)$. Finally, there exists a polynomial P such that, for any $n \geq 0$, the number of wires of the circuit C_n is less than $P(n)$.

For the particular case of functions of the form $f : \{0,1\}^* \to \{0,1\}$, this is equivalent to saying that NC^1 is the class of languages that can be recognized by a family $(C_n)_{n\geq 0}$ of circuits such that the following conditions are true. For each $n \geq 0$, the circuit C_n has n input boolean values, one output gate, and is built from gates labelled only by the boolean functions $\neg$, $\vee$ and $\wedge$. The fan-in of these gates is as follows: one for $\neg$, and two for $\vee$ and $\wedge$. Finally, there exists $k > 0$ such that, for any $n \geq 0$, the depth of the circuit C_n is less than $k \cdot \log_2(n)$.

Example 1.19. The parity function outputs 1 if the number of bits set to 1 in the input is odd, and outputs 0 otherwise. For example, for an input word $w = 10101101$, the parity function would output 1 because there are an odd number of 1s in the sequence.

To compute the parity function using circuits in NC^1, we can construct a family of circuits with $\oplus$ gates. These circuits repeatedly use $\oplus$ gates to compute the parity of adjacent pairs of bits until a single bit representing the overall parity is obtained. To do so, they operate as follows:

- Split the input sequence into pairs of adjacent bits: for example, the input 1010110 is split as $10 \cdot 10 \cdot 11 \cdot 0$.

- For each pair, use a $\oplus$ gate to compute the XOR of the two bits. This will yield a new word where each bit represents the parity of the corresponding pair of bits in the original word.
- Repeat the two first steps until there is only one bit left, which will be the parity of the entire input word. Thus, the circuit outputs 1 if and only if the input has an odd number of bits set to 1.

Since the size of the word is divided by two at each step, the depth of the circuits is logarithmic in the size of the input word. Also note that the $\oplus$ gate can be implemented using $\wedge$, $\vee$, and $\neg$ gates, which are allowed in NC^1 circuits. Therefore, the family of circuits computing the parity function is a valid example of a family of circuits in NC^1. This family recognizes the language of words that have an odd number of bits set to 1.

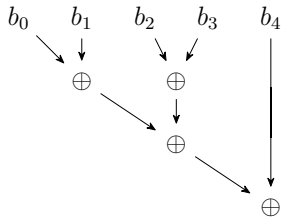


Figure 1.11 – The circuit computing the parity function for inputs of size 5

The set of languages recognized by circuits in NC^1 is characterized by the following theorem:

Theorem 1.4: Folklore

If a language is regular, then it can be recognized by a family of boolean circuits in NC^1.

This theorem is taken from folklore, and is stated by Straubing in his book [70, Theorem IX.1.1].

$\mathbf{AC^0}$. AC^0 is the class of functions that can be computed by a family $(C_n)_{n\geq 0}$ of circuits such that the following conditions are met. For each $n \geq 0$, the circuit C_n has n input boolean values and is built from gates labelled only by the boolean functions $\neg$, $\vee$ and $\wedge$. The gate $\neg$ is still of fan-in one, but $\vee$ and $\wedge$ are of unbounded fan-in. Moreover, there exists $k > 0$ such that, for any $n \geq 0$, the depth of the circuit C_n is less than k. Finally, there exists a polynomial P such that, for any $n \geq 0$, the number of wires of the circuit C_n is less than $P(n)$.

Example 1.20. For any alphabet Σ containing the letters a and c, it is possible to recognize the language of words $\Sigma^* ac^* a \Sigma^*$, i.e. the language of words that have an infix of the form $ac \cdots ca$ with a family of circuits in AC^0. To do so, we need to check each infix of length at least 2 of the input word w. The alphabet Σ is dealt with by representing the input word w with the set of booleans $l_i \mid l \in \Sigma, i \in [|w|]$, where $l_i = 1$ if and only if the letter of index i is the letter l. With this set of booleans as input, the circuit is constructed as follows: for each pair of indices (i, j) such that $0 \leq i \leq j - 1 \leq |w| - 2$, the circuit has an $\wedge$ gate of fan-in 2 connected to the booleans a_i and a_j. These $\wedge$ gates are meant to verify that we have an a both at the beginning and at the end of the infix. We also need to verify that the letters between these positions are occurrences of c, which is done using an $\wedge$ gate connected to all the booleans c_k

such that $i+1 < k < j-1$ (if there is no such position, no gate is needed). Then, we use an $\wedge$ gate connected to the results of the two other $\wedge$ gates. The result of this $\wedge$ gate is 1 if and only if the considered infix is of the form ac^*a. Thus, it suffices to add an $\vee$ gate connected to the third $\wedge$ gate for each infix. The result of that gate is 1 if and only if the input word is in the language. The circuits constructed this way are of constant depth, and have a quadratic number of gates, so the family is in AC^0.

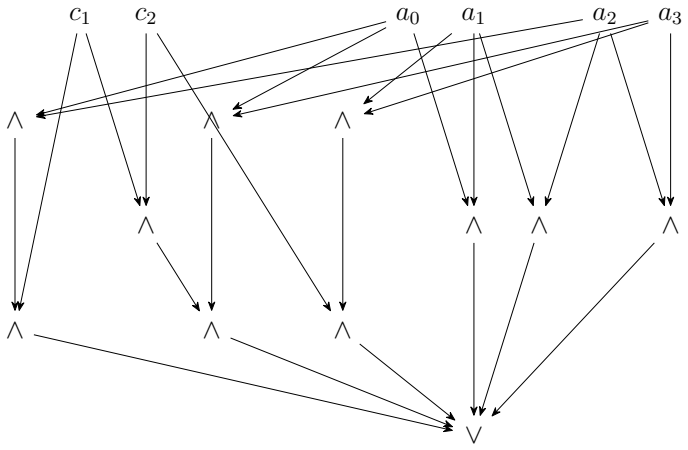


Figure 1.12 – The circuit recognizing words in $\Sigma^* ac^* a \Sigma^* \cap \Sigma^6$

Furst et al. [23] showed that it is impossible to build a family of circuits $(C_n)_{n\in\mathbb{N}^+}$ in AC^0 such that, for each $n \in \mathbb{N}^+$, the circuit C_n outputs 1 if and only if the number of boolean inputs set to 1 is even (the parity function presented in example 1.19). Later, Barrington et al. [7] gave a characterization of the regular languages in AC^0: those are exactly the languages in the class $\mathrm{FO}[\mathbf{C}]$, where $\mathbf{C}$ contains the predicates of $\mathrm{FO}[<]$ and all the unary predicates of the form C^r_d, where $C^r_d x$ is interpreted as $x \equiv r \bmod d$.

LAC^0. *LAC^0*, also called *linear AC^0* is the class of languages in AC^0 that can be recognized by a family $(C_n)_{n\geq 0}$ of circuits such that the size of C_n is $O(n)$.

Example 1.21. Given an input word of the form $a_0 \cdots a_{n-1} b_0 \cdots b_{n-1}$, we can build a circuit in LAC^0 that outputs 1 is and only if $a_0 \cdots a_{n-1} = b_0 \cdots b_{n-1}$. To do so, we use n $\oplus$ gates in parallel to check whether $a_0 = b_0$, ..., $a_{n-1} = b_{n-1}$. The result of these gates if 0 if and only if the equality is true, so we want all these results to be 0. This is checked by an $\vee$ gate on all the outputs of the $\oplus$ gates. The result of the $\vee$ gate is 0 if and only if $a_0 \cdots a_{n-1} = b_0 \cdots b_{n-1}$, so we add a $\neg$ gate to invert the result.

From this construction, we can define the family of circuits $(C_n)_{n\in\mathbb{N}}$ such that, if n is even, C_n is built as explained above, and otherwise C_n is the trivial circuit that outputs 0. This family of circuits recognizes the set of words $w = b_0 \cdots b_{2n-1}$ such that $b_0 \cdots b_{n-1} = b_n \cdots b_{2n-1}$. The circuit that processes inputs of size 6 is showed in Figure 1.13. This family of circuits is of constant depth, and the number of gates grows linearly with the size of the input words, so this family is in the class LAC^0.

Chaudhuri and Radhakrishnan [16] showed that LAC^0 is strictly included in AC^0. Then, an exact logical characterization was given when Koucký et al. [43] showed that the class of languages LAC^0 is exactly the class $\mathrm{FO}_2[\mathrm{arb}]$, the fragment of first-order logic that uses at most two variable names and has access to any arbitrary predicate.

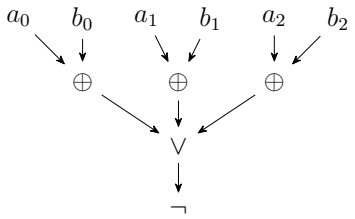


Figure 1.13 – The circuit that recognizes the words of size 6

WLAC^0. *WLAC*^{0} is the class of languages in AC^0 that can be recognized by a family $(C_n)_{n\geq 0}$ of circuits such that the number of wires of C_n is $O(n)$.

Example 1.22. It is possible to recognize the language of words of the form $101010\cdots$ using a family of circuits in WLAC^0. To do so, we take advantage of the fact that a word is in this language if and only if all the bits at even indices are set to 1 and all the others are set to 0. Thus, the circuit that recognizes the words of size n in the language is constructed as follows: it uses an $\wedge$ gate of unbounded fan-in on all the bits of even index to verify that all these bits are set to 1. An $\vee$ gate is used on all the bits of odd index to check that there is no bit set to 1 among those. Then, a $\neg$ gate is used on the result of the $\vee$ gate to obtain a 1 if and only if all the bits of odd index are set to 0. Finally, the output gate of the circuit is an $\wedge$ gate, which inputs are the outputs of the first $\wedge$ gate and the $\neg$ gate. This gate outputs 1 if and only if the bits of even index are set to 1 and the bits of odd index are set to 0, which is exactly the definition of the language. The circuit that processes inputs of size 5 is showed in Figure 1.14. This family of circuits uses a constant number of gates, and the number of wires grows linearly with the size of the input words, so this family is in the class WLAC^0.

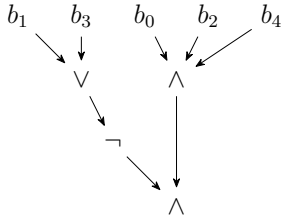


Figure 1.14 – The circuit that recognizes the word 10101

Having more gates than wires is not of much use, so WLAC$^0 \subseteq$ LAC^0. In [43], Koucký et al. gave a function that can be computed by a family of circuits in LAC^0, but cannot be computed by any family of circuits in WLAC^0, so the inclusion is strict. However, the question is still open when we restrict the classes to regular languages:

Open problem 1.1. *Do LAC*^{0} *and WLAC*^{0} *contain the same regular languages?*

The class WLAC^0 does not have a characterization as a fragment of first-order logic, but Cadilhac and Paperman [13] showed that the *regular languages* in WLAC^0 are exactly the class FO$_2$[$<$, mod], the fragment of first-order logic that uses at most two variable names and has access to the order relation $<$ and the relation $x \equiv 0 \bmod m$, for any integer $m > 0$.

Chapter 2

Experimenting on regex validation

Outline of the current chapter

As presented in the introduction, the goal of this thesis is to produce efficient programs recognizing regular languages by leveraging vectorization and the algebraic theory of automata. The challenge of recognizing regular languages can take several forms, which share a common basis. This basis is the principle: a program is said to recognize a regular language if, when given an input text and a regular language, it decides correctly whether the input text belongs to the language. Depending on the cases, the text or the language can be supposed to be known in advance, which allows for some optimizations before running the program. In this thesis, we suppose that the languages are known in advance, which places us in the domain of *online pattern matching*, as opposed to *offline pattern matching*, where the input text is known in advance and is processed using different techniques such as indexing. Consequently,

the programs presented in this thesis are built so that the regex must be given at compile-time instead of runtime, which allows for some complex optimizations during compile-time that can increase the efficiency at runtime. On the contrary, the text is given only at runtime.

In order to measure the actual efficiency of our algorithms, we implemented them (see the repositories [58, 59]). In order to make the most use of the possibilities offered by vectorization, we consider that the structures used to recognize a language – in our case, algorithms leveraging vectorization – are produced at compile-time. This allows our programs to build optimized data structures to recognize languages, which can then be optimized by a generic purpose compiler. We could produce equivalent programs at runtime by choosing the right vectorial instructions depending on the context, notably the progress already made in the validation, which can be the state reached in an automaton, the element computed in a semigroup, or the $\mathcal{R}$-class of that element. However, this dependency would force us to use conditional statements to choose the instructions to execute. This would significantly slow down our programs, as in practice conditional statements lead to branch prediction, and sometimes branch misprediction. In order to avoid this problem and produce more efficient programs, we process the automata at compile-time, where we choose the right vectorial instructions and assemble them depending on the properties of the language. The conditional statements are then used only at compile-time, allowing us to produce optimized, almost branch-free programs which can then be used as many times as needed.

Since these compiled programs can be used as many times as necessary without re-compiling them, reducing the time the code takes to run on the input text is more important to us than reducing the theoretical complexity of the algorithm that constructs the data structure (such as an automaton) to be executed on the input text. This allows us to build more complex data structures that may take more time to be compiled, but might be more efficient at runtime. Moreover, we can rely on the optimization phase of the compiler to improve the execution engine that we produce, which may increase efficiency.

This choice of compile-time processing makes it more or less meaningless to compare our implementation to the state-of-the-art tools, as they process the languages at runtime, which is desirable for these tools since the languages are not always known in advance. Consequently, though we run our benchmark on one of these state-of-the-art tools, we needed another baseline for programs which process the languages at compile time. To the best of our knowledge, only the tools lex, flex and Yacc could serve as baseline for this kind of computation. However, some experiments on flex showed that this tool is not well adapted to the needs of this thesis (see remark 2.4), and so we tried to implement our own baseline. We wrote several algorithms: one for a baseline using a classical algorithm based on an NFA, and several others to experiment on the possible uses of the algebraic theory of automata and estimate the feasibility of using algebra to produce parallel algorithms, which is a first step toward vectorization.

In our implementation, we considered *regular expressions* to represent regular languages. Regular expressions, that we present in this chapter, are a practical tool meant to manipulate texts, notably to validate that the input text is a word in a given regular language. However, they are significantly more expressive than regular languages, despite the similarities between the two names, as they include features such as backreferences. Thus, we consider only a fragment of the regular expressions, which is equivalent to the regular languages. In this chapter, we present the choices made for the implementation, such as the structure that represents an NFA. We also present the algorithms chosen as baseline to have an idea of what performance can be expected in our setting. This setting is particular, as it is chosen considering that we want to use vectorized programs, which led us to process the regular expressions at compile time to produce optimized programs that can then be run multiple times on the input texts.

2.1 Regexes: concept and usage

In section 1.1.4, we defined the set of rational expressions, which are used to define sets of words over a given alphabet. Though rational expressions are theoretical tools, a similar syntax is used in practice in various tools. These tools, which include algorithms for pattern matching and text processing, need to recognize languages in order to operate properly. Therefore, they were built using a syntax close to rational expressions, syntax which evolved with time to become more compact and more expressive, in order to meet the needs of a growing number of users. In this thesis, the formulas built using this concrete syntax will be called *regular expressions*, while the term *rational expression* will solely denotes the theoretical variant.

Regular expressions are used to describe patterns in text and thus form a domain-specific language (DSL). In other words, they are specifically designed for defining languages, i.e. sets of words. As other DSLs, they are built with their own syntax. In this section, we present a brief introduction on regular expressions and their syntax, but we focus mainly on a fragment of regular expressions which is exactly as expressive as rational expressions. We also present some of the algorithms used by existing regex manipulation tools.

2.1.1 From rational to regular expressions

It is possible to see regular expressions (regexes for short) as a extension of rational expressions (introduced in section 1.1.4) which uses more operations, including some that are more expressive than rational expressions. In this section, we define the fragment of regexes which we will focus on in the next chapters. This fragment uses more operations than rational languages, but these operations do not make it more expressive.

Note on the alphabet. Similarly to rational expressions, regexes rely on a fixed alphabet. In practice, this alphabet is often among a few concrete alphabets used in the context of text manipulation, notably ASCII and the various encodings of Unicode (among which UTF-8 and UTF-16). When considering a regex, it is crucial to carefully consider the alphabet to use. Indeed, regexes admit operations such as `.`, which recognizes any letter in the alphabet, and a complement operator which recognizes any letter outside a given set. These operators have different meanings depending on the chosen alphabet, and this can lead to sets of words of significantly different sizes. Thus, their concision highly depends on the alphabet (see example 2.1).

Example 2.1. Consider the regex `.*[^a].*` (the complement operator `^` is defined later in this section). If we denote the alphabet by Σ, then the language represented by this regex is $\bigcup_{b \in \Sigma \setminus a} \Sigma^* b \Sigma^*$. This union can be significantly large depending on the alphabet: in the case where Σ is unicode, the size may vary. As of the writing of this thesis, the last standard version of unicode is the 15.0 version, which has 149 186 characters. In the case where the chosen alphabet is ASCII, the size of the union is much more reasonable, with only 255 characters.

The operations of rational expressions. The fragment of regexes we are interested in notably uses the same operations as rational expressions, unfortunately with some shuffling of symbols. The empty set and the set containing the empty word are not represented by dedicated symbols, however they can be recognized using the tools described in the rest of this section. The letters of the chosen alphabet still represent themselves, except when one of those letters is also a metacharacter. In that case, classical escaping can be used, relying on the backslash

\, as escape symbol (see example 2.2 for an example). Furthermore, the concatenation is used but is always implicit: for two regexes r_1 and r_2, the regex r_1r_2 represents the concatenation of r_1 and r_2. Notably, the union is denoted by the metacharacter `|` instead of $+$. The star is used in the same way as in rational expressions, though it is not written as an exponent. Finally, the symbol `.` is used to denote the entire alphabet, whatever that alphabet may be, instead of Σ, which denotes an alphabet fixed in advance in the case of rational expressions. These operations define the expressivity of the fragment we consider, but this fragment admits more operations meant to improve the readability and concision of the regexes.

Operations for concision (and readability). Rational expressions can be quite tedious to write in some cases. As regexes are meant for a practical use, they include operations that are syntactic sugar, as they can be written with regexes using only the operations defined above. These operations are mainly used to denote specifications on a number of repetitions. Their syntax is as follows: given a regex `R`, the expressions `R+`, `R{i,j}` (for two integers $i \leq j$ written as their representation in base 10) and `R?` are regexes in our fragment. We call `+`, `{i,j}` and `?` *concision operators*. `R+` denotes the expression repeated one or more times and is thus equivalent to the regex `RR*`. `R{i,j}` denotes the expression `R` repeated between i and j times (both bounds are included), and is thus equivalent to

$$\underbrace{\texttt{R...R}}_{\text{i times}}\texttt{+...+}\underbrace{\texttt{R...R}}_{\text{j times}} = \texttt{R}^i(1 + \texttt{R} + \cdots + \texttt{R}^{j-i})$$

Finally, `R?` denotes the expression written 0 or 1 times. The equivalent regex is `R|^$`, using the two symbols known as *anchors*. For any regex `R` that does not contain an anchor, the expressions `^R`, `R$` and `^R$` are regexes. The anchors are metacharacters that denote the beginning of the text (`^`) and its end (`$`). They can also denote the beginning and end of a line when the regex is used in multiline mode.

Aside from repetition operators and anchors, our fragment factorizes regexes by using character classes. These are written in several possible ways. First, any string of literal characters (escaped if necessary) between square brackets denotes the union of all those characters. For instance, the regex `[cat]` is equivalent to `c|a|t`. This is useful for classes of characters which are not adjacent in the lexicographical order of the alphabet. Otherwise, another notation exists to denote such classes. Given any two literal characters $c1$ and $c2$, the regex `[c1-c2]` represents the class of all characters between $c1$ and $c2$ in the lexicographical order of the alphabet. For instance, the regex `[c-f]` is equivalent to `c|d|e|f`. To denote significantly big classes of characters, it is also possible to use the complement operator (`^`): any string of literal characters between square brackets and preceded by the complement operator denotes the complement of the class of characters denoted by that string of characters. For instance, the regex `[^aeiouy]` denotes the union of all the characters in the alphabet, except for the vowels a, e, i, o, u and y.

These operators allow for much more concise regular expressions, and therefore the rational expressions equivalent to the regular expressions written with these operators are significantly larger. For more details on the size of these rational expressions, see Lemma 2.1.

Definition 2.1

We define the *size* of a regex R, denoted by $|R|$, to be its number of characters except for the escape characters (this includes both literal characters and metacharacters).

Lemma 2.1

Let R be a regex without concision operators. Then, we have the following properties on the sizes of regexes when adding a single concision operator:

- There exists a regex without concision operators that is equivalent to `R+` and is of size $\Theta(|R|)$.
- There exists a regex without concision operators that is equivalent to `R?` and is of size $\Theta(|R|)$.
- Given two integers $i \leq j$, there exist a regex without concision operators that is equivalent to `R{i,j}` and that is of size $O((j^2 - i^2 + j)|R|)$.

Proof. As defined above, `R+` is equivalent to `RR*`, which is a regex without concision operators of size $2|R| + 1 = \Theta(|R|)$.

Similarly, `R?` is equivalent to `R|^$`, which is a regex without concision operators of size $|R| + 3 = \Theta(|R|)$.

For the last result, given the regex `R{i,j}`, by definition we already have an equivalent regex without concision operators, the regex

$$\underbrace{\texttt{R...R}}_{\text{i times}}\texttt{+...+}\underbrace{\texttt{R...R}}_{\text{j times}}$$

that is of size

$$\sum_{k=i}^{j} k|R| = \frac{(j-i+1)(i+j)}{2}|R| = \Theta((j^2 - i^2 + j)|R|)$$

□

Remark 2.1. The regex `R{i,j}` is particular as, for some values of the pattern `R`, it is equivalent to `R`. This is notably the case when `R` is of the form `S*`, for any `S`. However, there exist patterns `R` such that any regex without concision operators that are equivalent to `R{i,j}` are of size $\Omega((j^2 - i^2 + j)|R|)$, as shown in the next lemma.

Lemma 2.2

For any integer n, there exist a regex of size $\Theta(n)$ such that any equivalent rational expression is of size $\Omega(2^n)$.

Proof. Consider a fixed integer n and the corresponding integer $m = 10^n$. We claim that the smallest rational expression equivalent to the regex `a{m,m}` is

$$\underbrace{\texttt{a...a}}_{m=10^n \text{ times}}$$

which is of size 10^n. This regex can directly be translated into the rational expression $a \dots a$, with m occurrences of the letter a. To prove this, we show by induction on the structure of the expression that, for any rational expression that recognizes a regular language containing

exactly one word, it is possible to construct an equivalent rational expression of size at most equal and that does not use any operator beside the concatenation.

First, if the rational expression is a single letter, it does not use any operator and there is nothing to do.

Now, consider a rational expression R, and suppose that R represents to a language L of size 1. If R is of the form $R = R_1 + R_2$, then there are two possible cases. In the first case, one of the two languages L_1 and R_2 represented respectively by R_1 and R_2 is empty, and the other contains exactly one word. Since the union is commutative, we can suppose that R_2 is empty. Then the expression R_1 is equivalent to R, as they necessarily represent the same singleton. Consequently, it is possible to remove the union operator and obtain an equivalent rational expression of size at most equal. By induction hypothesis, we can find a rational expression R_3 equivalent to R_1, of size at most equal, that does not use either the star or the union, which concludes the proof in this case. In the second case, none of the two rational expressions recognizes the empty language. The language L contains exactly one word, so L_1 and L_2 must also contain exactly one word, and that word must be the same, as otherwise the language L would not be a singleton. Consequently, $L = L_1 = L_2$, and the expression R is equivalent to R_1. As before, since R_1 recognizes only one word, there exists an equivalent expression R_3 of size at most equal and that does not use either the star or union.

If R is of the form $R = R_1^*$, then the language L_1 represented by R_1 must be equal to $\{\epsilon\}$. Otherwise, it would contain a non-empty word w. In that case, the words $w \cdot w$ and $w \cdot w \cdot w$ would be distinct, and both would belong to the language L represented by R, which would contradict the hypothesis saying that L is a singleton. Thus, $L = L_1 = \{\epsilon\}$, and R_1 is equivalent to R. As before, R_1 represents a singleton, so it admits an equivalent rational expression R_3 of size at most equal and that does not use the union or the star.

Finally, if R is of the form $R = R_1 \cdot R_2$, then the two languages L_1 and L_2 represented respectively by R_1 and R_2 are necessarily singletons, since their concatenation is a singleton. Consequently, it is possible to apply the induction hypothesis and obtain two rational expressions R_3 and R_4 that are equivalent to the expressions R_1 and R_2, that do not use the star or the union and that are of size at most equal. This way, we obtain an expression $R' = R_3 \cdot R_4$ that is equivalent to R, of size at most equal, and that uses only the concatenation operator. This concludes the proof of this intermediary result.

This result shows that the smallest rational expression that recognizes a single word uses only the concatenation operator. This implies that the smallest rational expression equivalent to the regex `a{m,m}` is

$$\underbrace{\texttt{a}\ldots\texttt{a}}_{m=10^n \text{ times}}$$

To conclude, we only need to observe that $m = 10^n$, in decimal notation, is written with $\log_{10}(m) = n$ characters. Thus, the size of the regex `a{m,m}` is $\Theta(n)$, and the smallest equivalent rational expression is of size $10^n = \Theta(2^n)$, which proves our claim. □

Common shortcuts. Some classes of characters are so commonly used that they obtained a shortcut to denote them. For instance, the regex `\w` denotes the set of *alphanumerical characters*, which is the set composed of all the letters of the usual alphabet and all the digits. Thus, this regex is equivalent to `[a-z]|[A-Z]|[0-9]`. Other shortcuts denote a property on a position instead of representing characters or words. For instance, in the PCRE syntax, the shortcut `\b` (resp. `\B`) expresses the property "the current position is the beginning of a word" (resp. "the current position is the end of a word"), where "word" is to be taken in the linguistic

sense. These shortcuts may vary depending on the syntax used and on the chosen alphabet.

Example 2.2. The regular expression `^(0|\+33)[1-9]( [0-9]{2}){4}$` recognizes any possible french phone number. It works as follows:

- The anchors `^` and `$` ensure that the expression is executed on the entirety of the data.
- The prefix `(0|\+33)` of the expression recognizes either a 0 or the prefix +33, which allows to call french numbers internationally. It is followed by the character range `[1-9]` to complete the prefix of the number.
- The sub-expression `( [0-9]{2})` recognizes any word of three characters that begins with a space, followed by any two digits. This expression is followed by `{4}` to be matched four times. Thus, the suffix `( [0-9]{2}){4}` recognizes four pairs of digits separated by a single space, including one at the beginning.

For example, this regex recognizes the words "06 11 22 33 44" and "+337 55 66 77 88".

2.1.2 Usage of the extended regular expressions

The general set of regular expressions has a wide range of applications in various domains, including text processing, data validation, search and replace operations, and pattern matching. In this thesis, we are only interested in data validation, however the other applications played an important part in motivating the addition of more expressive operations to regular expressions, to represent more than the rational expressions.

Expressiveness beyond rational expressions: capturing and backreferences. Among the usual operations of regular expressions that are not included in our fragment are the grouping and capturing operations. These operations are meant to be used together to remember (or "capture") a part of the input text satisfying the captured part of the regex, either to output it directly, or to try and match the exact same text part elsewhere in the text. These operations work as follows: parentheses are used for grouping and capturing parts of the matched text. Then, the back-reference $\backslash n$, where n is a single digit, matches the substring previously matched by the n^{th} parenthesized subexpression of the regular expression. This is significantly more expressive than what can be done with rational expressions, as these operations allow to match a particular string seen previously in the text without knowing in advance its exact properties, as shown in example 2.3.

Example 2.3. The regex `((ac*b|c)*)(\w+)+\3\1` matches words of the form $w_0 w_1^+ w_1 w_0$, where w_0 belongs to the language defined by the rational expression $(ac^*b + c)^*$ and w_1 is composed only of alphanumerical characters. The backreferences allow us to reuse the words w_0 and w_1 even though they are not precisely defined in the regex. This is not doable with any regex equivalent to a rational expression.

Note that other advanced mechanics that will not be discussed in this thesis further enhance the expressivity of regular expressions. However, backreferences are more than enough to show the difference of expressivity, which motivates the fragment defined in section 2.1.1.

Data Validation. We saw in section 2.1.1 that regular expressions are related to rational expressions, and that a fragment of regexes is equivalent to the set of all rational expressions. Thus, it is natural that regexes can be used in a similar way to rational expressions, to recognize languages. Indeed, regexes allow the user to define rules for validating user input or data files. The goal here is to check whether the data verifies the properties expressed in the regex, without any more information. This can be related to the recognition of languages by seeing the data as a word. Checking that the data verifies some properties is equivalent to checking if the word formed by the data belongs to some language that expresses these properties. For instance, it is possible to use regexes to ensure that an email address follows a specific format, to validate a password for complexity requirements, or to check if a string matches a specific pattern before accepting it as valid input.

Example 2.4. Recall that the regular expression `^(0|\+33)[1-9]( [0-9]{2}){4}$` recognizes any possible french phone number. If we use it for data validation, it will be executed on the data and its return will only indicate whether the data matches the regex. For example, the text "+331 99 00 11 22" is matched by the regex, whereas the texts "00 11 22 44 55" and "*Phone* : 05 99 88 77 66" are not matched.

Pattern Matching. More generally, regular expressions allow the user to search for specific patterns within a given text or document. In this case, the goal is not to check whether the entire document verifies some properties, but to find all the text parts in the document that verify those properties. For example, one can use a regex to find all email addresses in a block of text, extract phone numbers from a dataset, or identify dates in a string. Formally, regexes represent any substring that matches the patterns. However, in practice, regexes do not allow for the search of every occurrence of a pattern in a text, as it is rarely what the user needs and the complexity of that operation would induce slow searches. Indeed, the occurrences of a pattern can overlap, as shown in example 2.5.

Example 2.5. Consider the regex `\w+`, which recognizes any word composed of alphanumerical characters. If matched against the text "*Tocallthepolice*, *dial*911", we usually want it to return the words "*To*", "*call*", "*the*", "*police*", "*dial*", and "911". However, each of these words leads to several matches of the regex. For example, in that text, the regex `\w+` matches the words "*all*", "*oli*" and "91", even though they are not maximal words matching the regex. In total, the text contains 56 matches of the regex, 21 of which are contained in the word "*police*".

Therefore, the search for all occurrences is reserved to spanners [21], theoretical objects meant for this application but not often used in practice because of their complexity (and because regexes suffice in practice). To avoid overlapping occurrences, when using regexes for pattern matching, the ambiguous repetition operators are used with a behavior that makes them choose what occurrences to match. This behavior is always deterministic.

There are three usual behaviors for repetition operators: *greedy*, *possessive* and *lazy*. By default, the repetition operators operate in a greedy way, matching as many characters as possible (see example 2.6) to match the entire pattern. Note that this does not mean that it will always match all the possible characters, as shown by the last example in example 2.6.

Example 2.6. The operator `+` in the regex `a+` is greedy be default. Thus, on the text "*aaaaaaa*", it will match only one occurrence, which is the entire text.

Now, consider the regex `<p>.*</p>`. If we use this regex for pattern matching in the text "* This is the first paragraph. This is the second paragraph. *", then it will

return only one match composed of the entire text, since the greedy sub-pattern `.*` can match any number of any characters, including the end tag " ". The match does not fail after the first end tag, so it continues to match the whole text.

Now, consider the regex `[aeiouy]+ei`. If we use the regex for pattern matching in the text "*siouyeia*", the greedy `+` can match "*iouyeia*", but the end of the regex does not have anything left to match. The greedy operator allows for matching less characters if needed to get a match, which leads to a single match for `[aeiouy]+`, the infix "*iouy*", as it is the only string of vowels followed by the infix "*ei*". Thus, the regex matches the infix '*iouyei*'.

This is the difference introduced by the possessive variants of the operators, that match as many characters as possible, without taking into account the rest of the pattern, which can lead to not match some occurrences (see example 2.7). This variant is, in some tools such as PCRE, indicated with a `+` after the operator: `a++` matches one or more occurrences of the letter *a* in a possessive way. As indicated by the documentation of PCRE, this behavior can be much more efficient and run faster than the greedy version, even when the two lead to equivalent regexes, as it does not have to check if backtracking in the text is needed.

Example 2.7. Let's come back to the last example of example 2.6, which is the regular expression `[aeiouy]+ei`. If we make the `+` possessive instead of greedy, we obtain the regex `[aeiouy]++ei`. If we take the same text, "*siouyeia*", the sub-pattern `[aeiouy]++` matches necessarily the infix "*iouyeia*" and cannot let go of some characters to allow for a match. Thus, the regex does not lead to a match on this text.

Finally, the lazy variant of the repetition operators, indicated with an additional `?` in most tools, is the opposite of the greedy behavior, matching as few characters as possible, as shown in example 2.8.

Example 2.8. If we consider the regex `a+?`, the operator `+?` is the lazy variant of `+`. Thus, on the text "*aaaa*", it leads to four matches, each composed of a single letter "*a*".

Let's come back to the second example of example 2.6, the regex `<p>.*</p>`. If we use the lazy variant of `*`, we obtain the regex `<p>.*?</p>`. This regex matches the text as one could naturally expect.

On the text "* Thisisthefirstparagraph. Thisisthesecondparagraph. *" it leads to two matches, made of the two paragraphs: "* This is the first paragraph. *" and "* This is the second paragraph. *".

Note that the behavior is set for only one operator at a time, so it is possible to design complex regexes that have operators with the three types of behaviors. For more details on the behaviors of repetition operators, see [4].

Text Processing. Pattern matching allows to find all occurrences of a pattern in a text. These occurrences can then be treated in order to change the text. In this context, regular expressions enable the user to perform operations such as finding and replacing specific patterns, removing unwanted characters or formatting, and extracting relevant information from unstructured text. This use of regular expressions may need to capture groups in order to modify the data.

Example 2.9. Suppose that you have a text and you want to change the phone numbers in that text to replace the spaces by hyphens. This can be done by searching for all occurrences of the pattern `(0|\+33)[1-9](( )[0-9]{2}){4}` and replace each group corresponding to a space (the third group captured by the regex) by a hyphen.

2.1.3 Existing regex processing algorithms

The history of algorithms for processing regular expressions dates back several decades and has evolved alongside the development of computer science and programming languages. The first algorithms processing regular expressions were used to match text strings, but now the entirety of the regular expressions defined in section 2.1.1 is supported in well-optimized algorithms. In this section, we will explore the various existing algorithms commonly used for processing regular expressions (regexes), along with their history.

The early algorithms: backtracking algorithms. Initially, regular expressions only represented regular languages [41] and did not include grouping, capturing, or any of the other advanced features. They were executed as-is, without pre-processing. Therefore, those that included alternations or the possibility to match an unknown number of characters required the algorithms using them to be able to go back in the text to try a different way to match the expression if the first way failed. This led to *backtracking algorithms*. Backtracking algorithms explore all possible paths in the search space to find a match. They attempt different combinations of matching patterns and backtrack when a match is not found. More formally, a backtracking algorithm without optimizations proceeds by considering sequentially each letter of the input text. Once it finds a letter that is matched by the beginning of the regex, it begins the search for a match starting at this letter, and advances both in the regex and in the input text. When it encounters an ambiguous pattern in the regex, such as `a*`, it has to make a non-deterministic choice and to remember the position in the input where it made the choice. This way, if the search fails, the algorithm goes back (*backtracks*) to the last position where it had another choice that has not yet been explored, and changes its choice there to continue the search. If no more choices are left, the entire search fails for that position in the input, and the algorithm starts again at the next position. Consequently, in the worst case where there is no match but many partial matches, the algorithm might read the entire suffix of the input text several times for each position in that text. The number of times the suffix is read is at most the number of ambiguous choices the regex can induce, which is about the number of operators among the following: `*`, `+`, `|` between patterns that have common prefixes, the repetition operator `{i,j}` if $i \neq j$, and `?`. Thus, if we consider the regex to be fixed, the complexity of backtracking is quadratic in the size of the input, which does not appear to be horrible in theory, but is an important drawback of the method in practice.

Example 2.10. Consider the regular expression `.* White|Black|Calico` that could be used to find some information about cat colors in a text. Also consider the following text:

```
- Name: Mr Whiskers, Color: Black
  Age: 2 years
  Very friendly
- Name: Willow, Color: Calico
  Age: 5 years
  Nervous cat that needs a lot of affection
```

Let's consider a naive backtracking algorithm executing the regex on this text for pattern matching. It considers the beginning of the text letter by letter until it finds a space followed by the letter W in the name of the first cat. This "W" matches the beginning of "White" so, depending on the choice it makes, the algorithm may try to match that word, which fails on the

third letter. Thus, the algorithm goes back to reading the space before the first W of the text and makes the other possible choice, matching it with the dot instead of the space. Similarly, when it gets to the actual color of the cat, it might try to match it with the dot and search for the word "White" after that, which will fail when getting to the end of the line, causing the algorithm to backtrack and finally match the color "Black".

Backtracking-based approaches offer flexibility for handling complex regex patterns but can be inefficient for certain cases due to redundant exploration. Notably, they are vulnerable to *regex bombing*, a type of DoS attack that exploits the way certain regex patterns are processed. It occurs when a crafted input causes a regex pattern to exhibit exponential time complexity, resulting in excessive CPU usage or long processing times. An attacker can take advantage of that by providing a specifically crafted input that triggers extensive backtracking, leading to the regex engine spending an inordinate amount of time evaluating the pattern and potentially causing a system or application to become unresponsive.

Example 2.11. For example, a vulnerable regex pattern that can be exploited for regex bombing might be `(a+)+b`. If an attacker supplies a string like "$aaaaaaaaaaaaaax$", the regex engine will spend an exponentially increasing amount of time exploring all possible permutations of a before realizing that there is no match. This can consume a significant amount of computational resources, causing a denial-of-service condition.

The text editor `ed`, one of the first developed parts of the First Edition Unix (1971), relied on backtracking. Later, the well-known command-line tool `grep` was created and integrated in the Fourth Edition Unix (1973). It was notably inspired from `ed`, and initially also used a backtracking algorithm. In these tools, the relatively low efficiency of backtracking was balanced by the fact that the supported regular expressions were not very expressive, including only one nondeterministic operation: the star, ignoring the alternation and parentheses [64].

NFA-based algorithms. Backtracking algorithms being non-linear, they cause some problems when using things like a cache or branch prediction. More importantly, they are vulnerable to attacks, other solutions had to be developed to improve the naive execution of the NFAs done by the backtracking algorithms. In 1968, Thompson [74] proposed a new approach that solved the issues of backtracking by using the NFAs in a deterministic manner. His algorithm considers each letter only once and sequentially, comparing it to a set of possible letters that match the regex and building the next set depending on the match found (see example 1.3). This approach essentially consists in building an NFA equivalent to the regex, then executing it deterministically on the text. This was the first *NFA-based algorithm*. NFA-based algorithms avoid being vulnerable to regex bombing by using the powerset construction: they maintain a set of current states, and thus never go back in the input. The name "NFA-based algorithm" comes from the fact that these algorithms build an NFA that they determinize on the fly, instead of building a DFA. This should not be confused with some NFA-based approaches which build an NFA and execute it using parallel algorithms that deal with the non-determinism.

Note that an algorithm to construct an NFA from a regular expression had already been proposed by Gluskkov in 1961 [29]. Though his NFAs used ϵ-transitions differently, they have been proved to be the same automata as Thompson's once those ϵ-transitions are removed. However, Thompson was the first to provide an efficient algorithm to run NFAs on words. He also proposed the first implementation of an NFA-based algorithm on the PDP-11. Later, the NFA-based approach was included in `grep` [35].

DFA-based algorithms. In practice, the NFA-based approach uses a DFA by determinizing the NFA on the fly. This avoids going back in the input when a mismatch occurs but it has a big inconvenient: at each state, which is a set of states of the NFA, it has to consider all the transitions going out of this state to compute the next one, which slows down the computation. As the NFA-based algorithms consider sets of states from the NFA, they are close to computing the powerset of the NFA and create an actual DFA. Thus, there have been some attempts at creating DFA-based algorithms, which compute a DFA equivalent to the regex, then run it on the input text. McNaughton and Yamada [49] gave in 1960 an algorithm to compute a DFA equivalent to what they called a restricted regular expression, which could not use intersection, negation, the empty word, or the empty language.

DFA-based algorithms have the advantage of running in linear time in the size of the input, regardless of the size of the DFA. However, the DFAs can be of size exponential in the size of the equivalent NFAs, and thus they can take as much time to be constructed from the regular expression. Thus, they are especially interesting if it is possible to construct a small DFA from the regex, or if the input is particularly long.

In some cases, it is possible to observe that not all the transitions in the DFA are used, or at least not used a lot. This led to an optimization which consists in avoiding to construct the entire transition table. This technique is called the *lazy transition evaluation* [3]. The idea is to compute a transition only when it is needed, and to store it in a cache to retrieve it if needed later. Once this cache is full, some transitions (preferably the less used ones) are removed from it. This technique is close to Thompson's on-the-fly determinization of the NFA, the main difference being that it does not remember all the transitions that it computes.

In the Seventh Edition Unix (1975), Aho introduced the command `egrep`, for an extended version of `grep` which included both the alternation and grouping. This command relied on a DFA, which made it twice as efficient as `grep` on simple regexes, but slower on more complex ones [35]. One of the early versions of `egrep`'s algorithm is explained in [3], section 3.9. In 1983, Aho replaced the construction of the DFA by a lazy transition evaluation.

Hybrid and optimized algorithms. Optimizations for these algorithms have been studied over the years, and gradually added to the tools using those algorithms. In 1980, Aho published an article summarizing some algorithms that could be useful for this purpose [1]. He notably talks about the Boyer-Moore algorithm [11], which searches for a string in a text and skips the parts of the text that cannot lead to a match. This algorithm can be integrated in regex processing algorithms, by considering the sub-expressions of the regexes that are strings and that are mandatory to lead to a match (the strings in alternations can be tricky to deal with). By first searching for such a string with Boyer-Moore, it is possible to narrow down the parts of the input text on which a full pattern matching algorithm is needed. As Boyer-Moore avoids looking at every character, this can be significantly beneficial. This technique has been used in `egrep` by Hume [35], improving its performance by a factor of 8.

Backtracking is not the best idea in general, but is unavoidable when advanced operators are allowed in regexes, such as backreferences. This forced people to be creative in their algorithms, to avoid backtracking as much as possible and mitigate the vulnerability when it is unavoidable, notably by looking for patterns that are likely to be used for regex bombing.

Runtime processing and compile-time processing. All the tools mentioned in this section process the regular expressions at runtime, which means that the regexes are processed dynamically during program execution. If some compiled form of the regex is produced before

processing the input text (an automaton, for example), then it is also produced during the execution, leading to an overhead that depends on the complexity of this pre-processing. In exchange, this allows for more flexibility, as the regexes can come from data produced at run-time, such as files or user input. Another approach consists in processing the regexes at compile time: all the pre-processing that needs to be done on the regex is done during the compilation of the program, which puts the overhead in the compilation instead of the execution. As the compiled program can be re-used, this can be a significant improvement, but this forces the regular expressions to be known at compile time, thus reducing the flexibility of the programs.

2.2 Compile-time optimization of automata execution

In section 2.1.1, we presented the fragment of regular expressions that we consider in this thesis. Recall that our goal here is to explore implementations that efficiently recognize regular languages, which are exactly our fragment of regexes. Consequently, from this point on, the thesis focuses on data validation, setting aside the more general problem of pattern matching. Now, we can give some details about our algorithms manipulating these regexes and the implementation we wrote. As we saw in section 2.1.3, there are many ways of matching a regular expression against an input text, each algorithm having its complexity, its advantages and inconvenients. As our goal is to leverage the algebraic theory of automata, we focus on an approach that initially uses automata to represent the regexes. With this approach, we consider the properties of these automata to produce optimized regex matching algorithms. In this section, we present our choices when implementing our algorithms, and we describe those that we use as baseline for our choice of processing regexes at compile time.

We saw in section 2.1.3 that several algorithms exist to match texts with a given regex. Here is a reminder of their complexity: backtracking algorithms are quadratic in the size of the text and vulnerable to attacks. NFA-based approaches such as Thompson's algorithm build an NFA of size linear in the size of the pattern in linear time and execute it in time $O(nm)$, where n is the size of the text and m the number of states of the NFA. DFA-based approaches build a DFA of size at most exponential in the size of the pattern in time at most exponential and execute it in time linear in the size of the input text. We could just present a new algorithm and compare its complexity to these usual algorithms. However, these are the theoretical bounds, which are often not met and that do not take into account the potentially large alphabet, or the cost of concrete representation of the automaton.

Our implementation of baseline algorithms allows us to measure the practical impact of the implementation choices such as the two examples given above. However, the main goal of this baseline is to give a point of comparison for our vectorized algorithms, which process the regexes at compile time.

As indicated above, we focus on the regexes in a fragment of regexes that is equivalent to the set of rational expressions, so we focus on automata-based approaches, using simple automata that go sequentially through the input without any memory other than a representation of the current state of the automaton. This way, we can study the regexes as automata that we want to execute as efficiently as possible.

Remark that we focus on text validation, ignoring completely the other possible uses presented in section 2.1.2. This narrowed scope allowed us to tailor our algorithms to the specific needs of text validation, optimizing their complexity accordingly.

2.2.1 Rust's specificities

The implementation is written in Rust and takes advantages of some tools the language offers.

A pre-existing parser. One of the technicalities involved when producing an implementation of regexes is the necessity to parse an input regex to get a structure that can be used to process the input text, for example a structure that represents an automaton. Rust's development provides the crate (a.k.a. package) `regex`, which gives tools to process regexes, as a module which includes an autonomous parser within a dedicated crate called `regex-syntax`.

This parser allows us to obtain a structure that represents an NFA. That structure is close to the one used by Cox in `re1` [17], as it is a sequence of instructions, among which some represent a transition labeled by a letter, some represent an intermediary state which has two ϵ-transitions leaving it, and others represent an accepting state which has no transition leaving it. However, the structure computed by `regex-syntax`'s parser is more general, as it has transitions labeled by unions of classes of letters. The structure is a sequence of instructions, each associated with its index in the sequence. The structure of the crate `regex` changed during the thesis, which resulted in two similar versions of the code. The first to be implemented used the following set of instructions:

- Char(a, goto), for any unicode character a and any valid index goto, represents a transition labeled by a. The next instruction is the one associated with the index goto.
- Ranges(classes, goto), represents a transition labeled by a set of unicode characters. That set is defined by the argument classes, which is a set of disjoint character classes. The next instruction is the one associated with the index goto.
- Bytes(start, end, goto) represents a transition labeled by a class of bytes. This class is the set of bytes whose decimal value is comprised between start and end. The next instruction is the one associated with the index goto.
- Split(goto1, goto2) represents a pair of ϵ-transitions leading to the states associated with the indices goto1 and goto2.
- Match() represents an accepting state. Note that this state has no transition leaving it. The input is recognized if and only if the NFA reaches a Match instruction exactly at the end of the input.

Remark 2.2. The structure produced by `regex-syntax`'s parser has more types of instructions meant to deal with the more expressive features of regexes, such as backreferences. Our definition is restricted to the instructions that are used in our implementation.

The second, more recent, implementation uses a similar set of instructions, with a few key changes. The most important one is that the most recent version of the crate `regex` is focused on dealing with strings of bytes instead, where the older version could handle explicitly strings of both unicode characters and bytes. Thus, instead of the instructions Char, Ranges and Bytes, the most recent version has only one instruction called ByteRange, which is equivalent to the Bytes instruction. Moreover, the Split instruction was refined in the new versions, leading to two instructions, one equivalent to Split, dealing with a non-deterministic choice between exactly two paths, and one that generalizes Split, dealing with a non-deterministic choice between an arbitrarily large number of paths. This led to a few changes between the two implementations of the code processing the regexes. Note that the most recent implementation

has been developed near the end of the thesis, which left only time to perform the benchmarks and not to analyze the differences properly.

In order to manipulate that structure without dealing with the technicalities it induces, we propose a formalization in the form of automata which transitions can be labeled by unions of ranges of characters.

Definition 2.2

Let $\Sigma = \{a_0, \ldots, a_{n-1}\}$ be an alphabet ordered by the relation $<$ such that, for any $i, j \in [n]$, $a_i < a_j$ if and only if $i < j$. An *ordered ranges automaton* is a tuple $(\Sigma, Q, I, \delta, F)$ where Q is the set of states, I is the initial state, F is the set of final states, and $\delta : Q \times (\Sigma \cup R(\Sigma)) \to 2^Q$ is the transition function, where $R(\Sigma)$ is the set of all possible *ranges* $\{a_i, a_{i+1}, \ldots, a_j\}$, where $i < j$.

With this definition, we can manipulate the structures without changing it depending on the alphabet. However, this comes with some disadvantages. First, it hides the fact that, for unicode characters, the transitions can be labeled by disjoint unions of ranges (in the Ranges instructions) whereas, for byte characters, they can be labeled by only one range of characters (for Bytes instructions). The other disadvantage is that it hides the number of Split instructions necessary to implement n transitions coming out of a state of the automaton. The following two lemmas give an idea of how to represent the structures from `regex-syntax` with ordered ranges automata.

Lemma 2.3

Given a program composed of c Char instructions, r Ranges instructions, and some Split and Match instructions, we can build an ordered ranges automaton of size $O(c + r)$ that recognizes exactly the same words over the unicode alphabet. Moreover, the number of transitions of that automaton is $O(c + rn)$, where n is the maximal size of the sets in the Ranges instructions.

The actual proof involves a lot of technical, uninteresting details, so we give only an idea of proof. Each Char instruction can be replaced by a transition labeled by the same character. That transition must be between two existing states, so it might involve creating an intermediary state, as the instructions directly after the Char instruction might not have created a state. Similarly, each Ranges instruction Ranges(I,goto) can be replaced by $|I|$ parallel transitions, each bearing one of the intervals in I. Each Match instruction is handled by making the current state an accepting state. Finally, each Split instruction is handled by creating two paths from the current state of the automaton, one for each index indicated by the instruction. These paths are only initiated here, as, at this point, we do not know what the labels of the transitions will be. For each of these paths, if the next instruction is a Char or a Ranges instruction, then the path starts by constructing a transition following the rules for that instruction. If it is a Split instruction, the path is split in two again, and those new paths are constructed recursively.

Note that the number of Split instructions does not factor at all in the size of the ordered ranges automata. This lemma implies the next one, which deals with the structures produced by the second implementation, which can be seen as automata on the alphabet of bytes.

Corollary 2.1

Given a program composed of b ByteRanges instructions and some Split and Match instructions, we can build an ordered ranges automaton of size $O(b)$ that recognizes exactly the same words over the byte alphabet. Moreover, the number of transitions of that automaton is also $O(b)$.

Processing the regexes at compile time. As explained in the introduction of this chapter, we focus on processing the regexes at compile time. This approach has already been used by existing tools, called Yacc (Yet Another Compiler Compiler) and Bison, a forward-compatible Yacc replacement which includes more features. The two tools are *parser generators*, which take as input a file containing the description of a programming language's grammar and syntax and return a parser for that programming language. The approach is very similar, as these tools take a set of rules, which can be assimilated as a regular expression, and create a program capable of interpreting a text based on these rules. We do not interpret the input text, but we validate it, which requires a similar kind of algorithm.

Rust offers many tools to execute code during compilation. The tool we are interested in is called a procedural macro. Procedural macros will not be presented in details here, but informally a procedural macro takes some code –any code– as input and executes some predefined user code –*any* user code– on it. The strength of these macros compared to the other kinds of macros is that they allow to study precisely the input code and to use parts of it in the code executed at compile time. More precisely, we use procedural macros the following way in our implementation: the input of the macro is the text of the regex. This text is processed at compile time, notably using the parser provided by `regex-syntax`, to obtain an efficient structure. This structure is then turned into code. The general look of this code is a function which can be executed on some input text to process it and return a boolean indicating whether the text belongs to the language represented by the regex.

2.2.2 Challenges brought by the framework

As explained above, Rust offers a convenient tool to build code from other parts of code at compile time. This tool, called the procedural macros, can execute any code during the compilation to return a new code to be executed at runtime, which allows to build optimized programs. However, this tool is not easy to handle, which delayed the results. The main hurdles that were encountered are detailed here, in hopes of helping some people avoid some of them in the future. Note that what is explained here about Rust is what I understood during the development: it is linked to the version of Rust at the time this thesis was written, and some tools might have helped with the problems described here, but I was not aware of their existence.

Before even starting to write a macro, it is important to think about the context in which it will be used. A macro can indeed return any kind of code, but that code must be usable. The problem with that is the fact that any function defined in a macro technically exists, but cannot be called if the user does not do something to assure Rust that the function exists. Therefore, writing a function `foo` that is exclusively defined in the macro and trying to call that function in some code that calls the macro will result in Rust not compiling and telling that the function is unknown. This influenced greatly the design of the macros in [58] and [59] which is admittedly quite unusual. In that code, if a user needs to use a macro `foo` on some

regular expression, and then see if some input word belongs to the language defined by that regex, they need to write the following code:

```
use macro_traits::CompiledU8;
let structure = foo!("This is the regex");
return structure.execute(input_word);
```

The crate `macro_traits` exists only to define *traits*, which give templates of the methods defined by the macro. It contains a few traits, including `CompiledU8`, which ensures that the structures implementing this trait has a method called `execute`, which takes as input a vector of bytes and returns a boolean. Without this trait, the call to `execute` would not compile. Consequently, whatever code returned by the macro must be some kind of structure that implements the trait `CompiledU8`. The choice that was made, in part because there seemed to be no other choice, was to return code that defined a `struct` implementing the right trait and that returned an instance of that `struct`. This allows to build a structure on which the user can call the method `execute`.

Note that, in the example of user code above, the regex is given as a literal string. This is the only way to use macros, as variables do not give their values as parameters of the macro. All a macro can see is the literal text given as parameter. It knows if a variable is given to it, but cannot identify its value. Thus, it is impossible to write a script like this one:

```
use macro_traits::CompiledU8;
let regex = "This is the regex";
let structure = fooone!(regex);
let structure = footwo!(regex);
return (structure.execute(input_word), structure.execute(input_word));
```

For each call to a macro, the regex must be explicitly written. This is why the main files used to call the macros in the repositories are so lengthy: it seems there is no other option.

Now that we covered the restrictions on the external calls to the macros, let's talk about the internal encoding of those macros. The parameter taken by any procedural macro is the literal text given inside the parenthesis, that is interpreted by Rust as Rust code. In order to use that, it is necessary to parse the structure of that code, called the Abstract Syntax Tree (AST for short). As Rust uses some complex types and structures, the AST is quite complex to understand and to parse searching for a pre-defined kind of input. Even for the use made in this thesis, in which the macros accept only literal strings, getting those strings from the AST is not obvious[1]. For that purpose, a deep dive into the documentation of the crate `syn` is required.

The last challenge encountered in the development of the code was returning the output code. When that code is simple, then it is rather straightforward, as it suffices to use the macro `quote` to turn text into an AST that can then be returned by the macro. However, if the output code is produced by parts that must be assembled, it becomes trickier. Indeed, although it is possible to combine some ASTs to create a new one, there are rules, that are hard to find. The general rule is the following: two ASTs can only be combined if Rust can guarantee that this combination produces valid Rust code. Meaning that it is impossible to directly concatenate two ASTs or insert one in another. In order for Rust to find the guarantees it needs, those ASTs

[1] A curious reader will find the function *get_regex*, constructing from the input text the string that represents the regex, in the file `regex-macros/src/lib.rs` of the repository [58].

must be parsed as more specialized parts of an AST, such as functions or variable names. Then these specialized structures must be placed in some code where their type can be expected. For example, a variable name can be placed as parameter of a function.

In summary, Rust's procedural macros are difficult to master, as each step requires a deep dive in a documentation that, although well made, is quite extensive. Maybe other languages could have been more adapted, as Rust is not the only language offering compile-time meta programming. See notably Template Haskell and OCaml's Pre-Processor eXtensions, which seem to offer tools similar to procedural macros. However, these tools seem to not be more user-friendly than procedural macros, maybe even less. Moreover, Rust has the advantage of offering tools adapted to regex processing, such as the crate `regex`. It also offers easy access to low-level instructions, contrary to OCaml, which enabled the comparison of all algorithms in the same language. For these reasons, Rust might be the best language available for measuring the efficiency of regex processing algorithms, some of which written with low-level instructions.

2.2.3 General methodology

To represent the process of running automata on words, various choices of data structures and algorithms can be made. In this section, we discuss our choices and provide justifications for their usage.

As explained in section 2.2.1, the general process followed by our implementation consists in taking a regex, giving it to `regex-syntax` to obtain a structure representing an NFA with ϵ-transitions, then compile that structure to obtain a function that can then be used to check whether some text is matched by the regex. In this section, we detail the compilation from the NFA to the matching function.

Inputs and outputs. Once the alphabet is fixed, we need to specify the kind of data structure that we call a *word*, that is the input text of our algorithm. As one of our goals is to produce algorithms suitable for streaming, one could assume that our input text is a *stream* of data, that is a sequence of elements made available sequentially at different point in time. In our case, these elements could be letters (characters or bytes) or batches of letters. However, considering a stream would require to deal with Input/Output (I/O) overhead. It introduces computations that should not be considered when measuring the efficiency of an algorithm, which takes up CPU computation power and can consume significant memory resources. In order to focus on the actual efficiency of our algorithms, we chose to use data stored in memory. Thus, in this thesis, an *input word* is a memory segment, that is a contiguous block of memory allocated for storing data. This memory segment can be used to simulate streaming by considering the elements by batches.

Thus, our algorithms will always take some memory segment as input. As we focus on text validation, the output will only be a boolean, indicating whether the input is accepted by the regex.

UTF-8 encoding and performance. When it comes to implementing an automaton, it is necessary to specify the alphabet considered by that automaton. It is particularly important when the automaton represents a regex and could be run on different kinds of inputs. The most intuitive choice would be the usual set of characters that can be represented visually, i.e. unicode.

This choice of alphabet gives a simple and immediate correspondence between the characters used by the regexes (supposing these are written with unicode characters) and the characters

of the alphabet. It also has the advantage of being independent of the encoding of those characters. However, that advantage comes with some significant drawbacks. Indeed, this approach needs the encoding of those characters to be valid, and may require to validate that encoding in addition to matching the regex. For example, if the input text is extracted from a file, we would need to validate the encoding to convert the file from bytes to unicode characters. This choice of alphabet supposes all the characters to be equivalent in terms of size in memory which, in practice, requires to allocate for each character as much memory as needed for the character which takes the most space in memory. For example, if we consider characters written as UTF-8, 32 bits are required for the largest encoding, so 32 bits would be allocated for each character, including those that can be written on only 8 bits. Thus, this choice of alphabet is not optimal in terms of memory use.

In order to address these problems, it is possible to change the alphabet to take the encoding into account. Thus, we chose to consider the alphabet composed of all possible bytes. Some of these bytes are considered incorrect, as they cannot be found in a valid UTF-8 encoding of a text, and others are only valid in certain contexts. This may require more computations to check whether some "letters" are valid, but we show later that this is not an issue in our setting. However, this multiplies the size of the alphabet by a factor of approximately 4, which can impact performance. Notably, the automata based on this alphabet can be significantly larger than the ones based on the character alphabet, as shown in example 2.12.

Example 2.12. Consider the automaton shown in Figure 2.1. This automaton uses any unicode character as letter: for example, U+1F642, whose UTF-8 encoding is four bytes long, is treated as one letter.

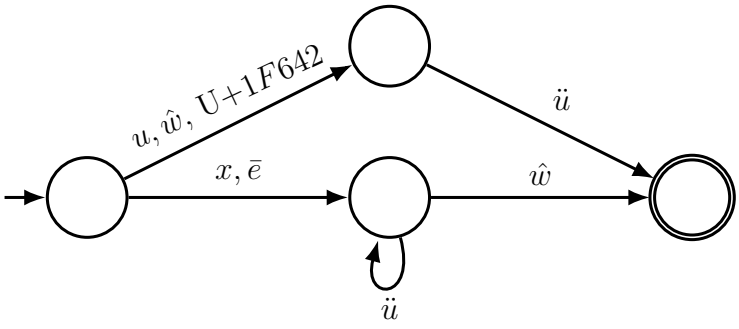


Figure 2.1 – An automaton using letters from the unicode alphabet

If the alphabet is changed to the alphabet of bytes, we obtain an automaton that must differentiate the unicode characters by the length of their UTF-8 encoding. Thus, any transition labeled by U+1F642 must be replaced by four consecutive transitions, each one bearing one of the four bytes of the encoding $\backslash x\ F0\ 9F\ 99\ 82$.The same rule applies to all the unicode characters: if their UTF-8 encoding is longer than one byte, the transition labeled by that character must be transformed into several successive transitions. Consequently, we obtain the automaton shown in Figure 2.2, which is significantly larger than the automaton in Figure 2.1.

In order to measure that impact, the benchmarks presented in this thesis include both versions: one with the intuitive alphabet of unicode characters, and one with the alphabet of all bytes. Both are obtained using tools from the crate `regex`, although some are no longer available in the most recent version, which motivated the separation of the implementations in two git repositories.

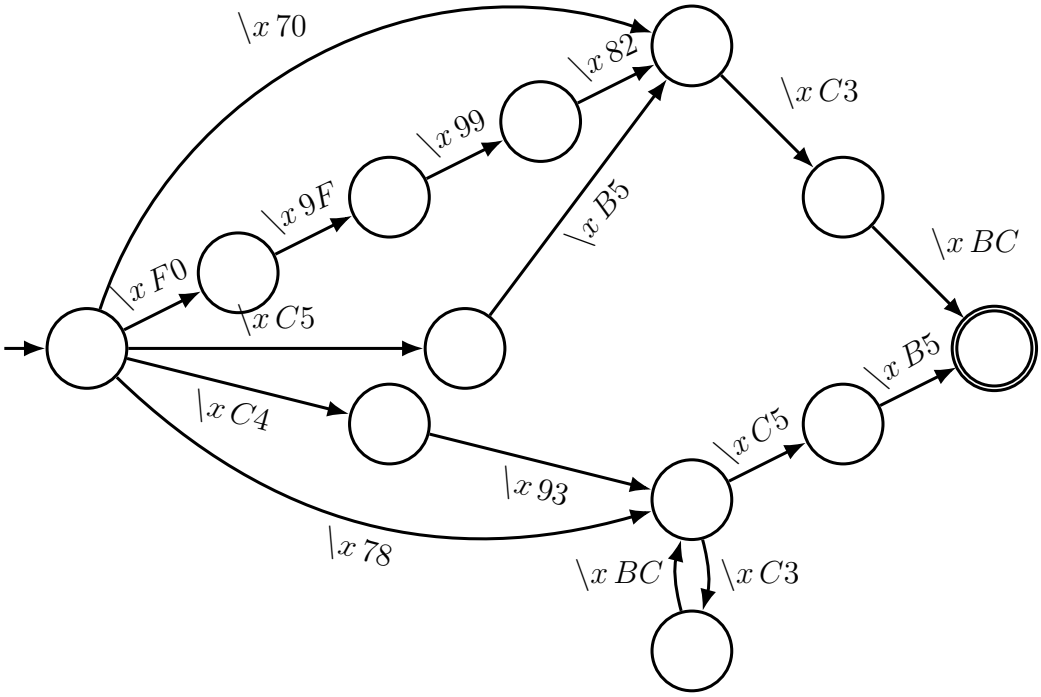


Figure 2.2 – An automaton using letters from the byte alphabet

Determinization and minimization. The NFAs produced by the crate `regex` can be large due to their implementation. Moreover, using NFAs implies that the transitions of the letters cannot be represented with boolean vectors. If we stick with boolean vectors, it is necessary to use boolean matrices, which are often mostly empty. This, in turn, can lead to a significant increase in compilation time. Although compilation time is not our primary focus, we tried to improve it by determinizing and minimizing the automata that we obtained. The structures, notably the semigroups, obtained via this method, can be quite different from the ones obtained from the NFAs, so the efficiency of each algorithm is measured for both the NFA version and the determinized version.

To minimize the NFAs, the first step taken in the code is to remove all ϵ-transitions using their transitive closure. Then, using Brzozowski's algorithm (see [12]) on the resulting NFA, the code computes the minimal DFA. This algorithm notably needs to determinize NFAs, and does so using the powerset construction. The resulting algorithms ultimately manipulate DFAs, whose transitions can be represented using vectors instead of matrices. This could lead to some performance improvements when the NFA has significantly more transitions than the minimal DFA. However, the languages considered in our benchmark do not have that property.

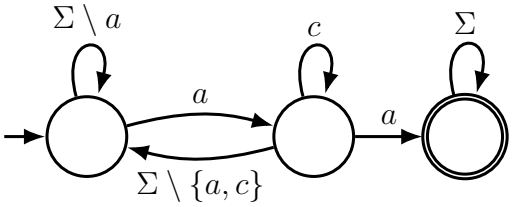
The practical baseline: the crate regex. In order to evaluate the performance of our implementation, we compare it to the crate `regex`, which handles regexes for both data validation and pattern matching. In the benchmark results, this version is called "base". One substantial difference between the regex-processing functions of `regex` and the algorithms introduced in this chapter is that it pre-processes the regex at runtime. Although this part of the computation does not count into the time measured for the benchmark results, it means that the pre-processing is limited to be tractable at runtime, which can impact performance. However, existing benchmarks [38, 25] show that the crate `regex` can perform quite well, even

though programs in C and C++ have a greater potential. The crate includes several optimizations, including literal optimizations that allow it to search for literals using bit-level parallel instructions (see section 3.1 for a definition of bit-level parallel instructions) inspired from the project Hyperscan [76], as described in [26]. These optimizations allow skipping unimportant characters, resulting in a well-optimized tool on the regular expressions used in practice. Consequently, it can make a good baseline, as comparing the algorithms presented in this thesis to `regex` gives an idea of where these algorithms can be placed in the hierarchy of existing regex-processing tools.

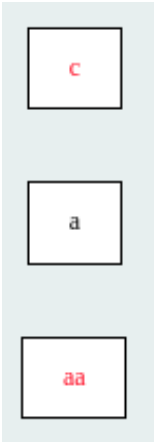
2.2.4 Inputs of the benchmarks

In our benchmarks of this chapter, we consider four regular languages. These languages and the words used to benchmark our algorithms on each of them are described below.

Languages in Ap. The language $acstara = \Sigma^*ac^*a\Sigma^*$, where Σ is the alphabet of unicode characters, is in the class **Ap**, and represents the set of words on the alphabet Σ that contain an infix of the form $ac\cdots ca$, with zero or more occurrences of c. The minimal DFA for that language is given in Figure 2.3a, and the egg-box representation of its syntactic monoid in Figure 2.3b (the elements represented in red are idempotent).



(a) Minimal DFA for $\Sigma^*ac^*a\Sigma^*$



(b) Syntactic monoid of $\Sigma^*ac^*a\Sigma^*$

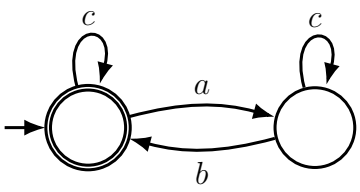
Figure 2.3 – Characterization of $\Sigma^*ac^*a\Sigma^*$

For this language, there is only one input word, named *last*, composed of 10^9 random bytes, followed by the infix aca. The word is processed so that the first 10^9 bytes do not contain any infix of the form ac^*a.

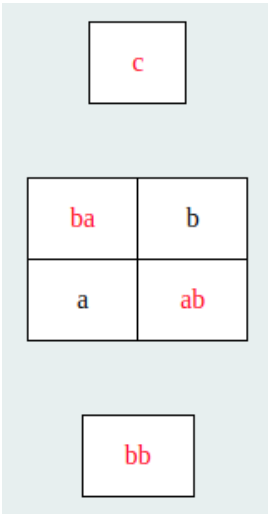
The language $abstar = (ac^*b + c)^*$, in the class **Ap**, represents the set of words on the alphabet $\Sigma = \{a, b, c\}$ such that the first letter different from c is an occurrence of a, the last letter different from c is an occurrence of b, and for any pair of occurrences of a (resp. b), there is an occurrence of b (resp. a) between the two. The minimal DFA for that language is given in Figure 2.4a, and the egg-box representation of its syntactic monoid in Figure 2.4b.

For this language, there are three input words, all belonging to the language, whose density of occurrences of a and b varies.

- The word *dense* is composed of 5.10^8 occurrences of the word ab, to which 10^7 occurrences of c have been added at random indices.



(a) Minimal DFA for $(ac^*b + c)^*$



(b) Syntactic monoid of $(ac^*b + c)^*$

Figure 2.4 – Characterization of $(ac^*b + c)^*$

- The word *average* is composed of 10^9 letters, among which 10^6 have been randomly selected to be occurrences of a and b. The letters at the selected positions alternate between a and b, the others being occurrences of c.

- The word *sparse* is built on the same algorithm as *average*, except that only 10^3 letters are occurrences of a or b.

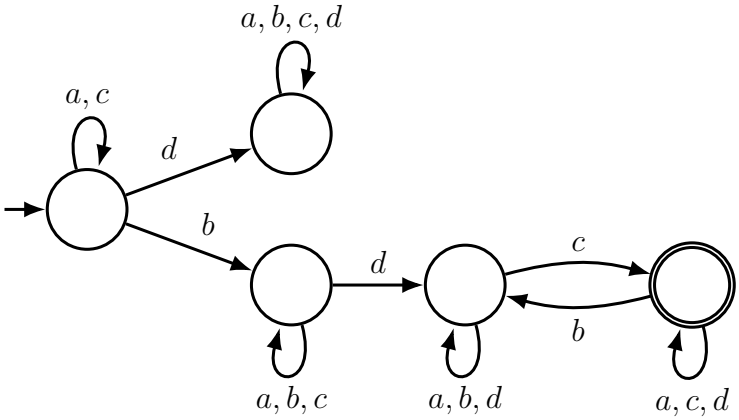
Languages in DA. $L_{\mathbf{DA}} = (a + b + c)^*b(a + b + c)^*d(a + b + c + d)^*c(a + d)^*$ is in the class **DA**, and represents the set of words in the alphabet $\Sigma = \{a, b, c, d\}$ in which there is at least one occurrence of b before the first d, which itself is before the last occurrence of c. Moreover, there must be no occurrence of b after that last occurrence of c. The minimal DFA for that language is given in Figure 2.5a, and its syntactic monoid in Figure 2.5b.

For this language, there are three input words, all belonging to the language, in which the positions of the first d and the last c vary.

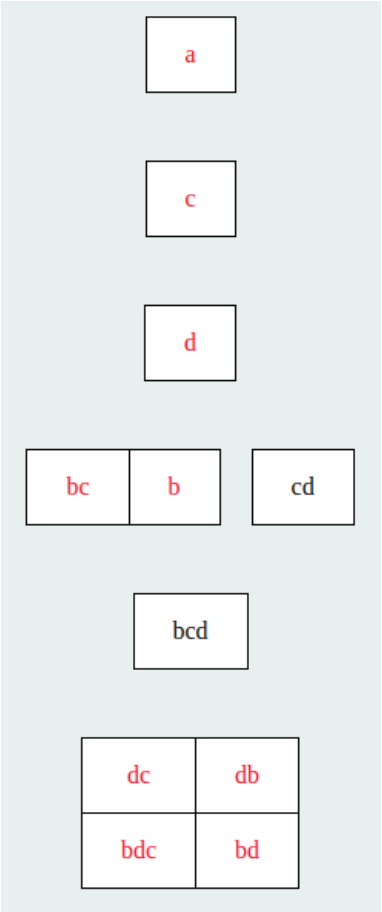
- The word *balanced* is composed of 25.10^7 letters randomly chosen among a, b or c, followed by an occurrence of b, then again 25.10^7 letters randomly chosen among a, b or c. This is followed by an occurrence of d, 25.10^7 letters randomly selected among a, b, c or d, an occurrence of c, and 25.10^7 letters randomly chosen among a or d.

- The word *increase* is composed of 10^7 letters randomly chosen among a, b or c, followed by an occurrence of b, then 2.10^7 letters randomly chosen among a, b or c. This is followed by an occurrence of d, 3.10^7 letters randomly selected among a, b, c or d, an occurrence of c, and 4.10^7 letters randomly chosen among a or d.

- The word *decrease* is composed of 4.10^7 letters randomly chosen among a, b or c, followed by an occurrence of b, then 3.10^7 letters randomly chosen among a, b or c. This is followed by an occurrence of d, 2.10^7 letters randomly selected among a, b, c or d, an occurrence of c, and 10^7 letters randomly chosen among a or d.



(a) Minimal DFA for $L_{\mathbf{DA}}$



(b) Syntactic monoid of $L_{\mathbf{DA}}$

Figure 2.5 – Characterization of $L_{\mathbf{DA}}$

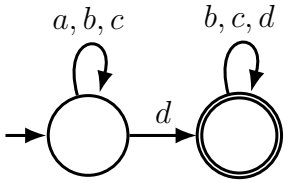
$\mathcal{R}$-trivial languages. This thesis focuses on languages in **Ap** and **DA**. However, inside **DA**, another class of languages deserves to be studied: the class of $\mathcal{R}$-trivial languages. Indeed, those languages are much simpler than other languages in **DA**, which gives us an additional tool to study the impact of the complexity of languages on the algorithms presented in this chapter.

$L_{\mathcal{R}} = (a+b+c)^*d(b+c+d)^*$ is a simple $\mathcal{R}$-trivial language, which represents the set of words in the alphabet $\Sigma = \{a, b, c, d\}$ in which there is no occurrence of a after the first occurrence of d. Its minimal DFA is given in Figure 2.6a, and its syntactic monoid in Figure 2.6b.

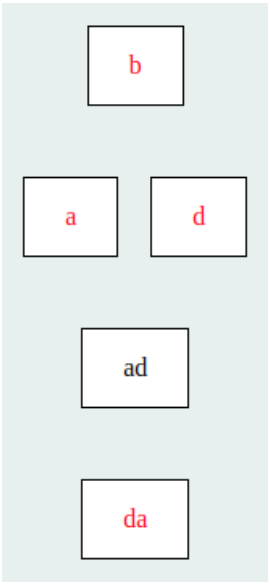
For this language, there are three input words, all belonging to the language, in which the position of the first occurrence of d varies.

- The word $first$ is composed of an occurrence of d as first letter, and 10^9 letters randomly selected among b, c and d.
- The word $last$ is composed of 10^9 letters randomly selected among a, b and c, and an occurrence of d as last letter.
- The word $middle$ is composed of 5.10^8 letters randomly selected among a, b and c, an occurrence of d in the middle, and 5.10^8 letters randomly selected among b, c and d.

Remark 2.3. The words that have been selected for the benchmarks all belong to the corresponding languages. Indeed, the goal is to measure the efficiency of the algorithms in the worst



(a) Minimal DFA for $L_{\mathcal{R}}$



(b) Syntactic monoid of $L_{\mathcal{R}}$

Figure 2.6 – Characterization of $L_{\mathcal{R}}$

case, when the computation necessarily goes through the whole word to check that it belongs to the language. The algorithms were also tested on words that do not belong to the languages and the results were not interesting, as they were either equivalent to the ones presented here, or much faster as the computation was stopped. This is due to the fact that, for some languages, the compiler notices the parts of the code that correspond to sink states (in the NFA) or elements (in the semigroup), and optimizes these parts by making the computation stop as soon as they are reached.

2.2.5 Hardware and implications

The results presented in this chapter have been obtained using grid5000, and more precisely the machine troll-3. At the time when this thesis is written, the machines troll each have two CPUs Intel Xeon Gold 5218, on the architecture x86_64. Each of these CPUs has 16 cores. The version of rustc used was rustc 1.72.1 (d5c2e9c34 2023-09-13), and the version of cargo was cargo 1.72.1 (103a7ff2e 2023-08-15). This offers the possibility to compare the algorithms on that kind of hardware, but it leaves out that hardware's impact on the efficiency. Indeed, it impacts the way the code is compiled, and thus its efficiency. Consequently, additional results on three other machines, two with Intel hardware and one with AMD hardware, are provided in appendix A.

Each of the results has been measured as a mean of twenty runs of the same command. The standard deviation was measured but was negligible in all cases. However, the algorithms' efficiency may vary depending on a surprising amount of factors. These factors notably include the hardware, the version of Rust, and the order of the instructions in some functions.[2] Consequently, comparing which algorithm is better than another with all these factors fixed is not

[2]The order of these instructions notably depends on the hashing algorithm chosen for some tools used in the code. In the repository, the hashing algorithm that was used always gives the same order for the instructions, which is useful for getting consistent results. Indeed, if another algorithm is used, the results may significantly change.

particularly significant. The results presented in this chapter are separated in four categories, depending on their throughput:

- The particularly slow runs, with a throughput below 0.1 GB/s, are not the most common case in our results but their lack of efficiency stands out.
- The average runs, with a throughput usually between 0.1 and 0.7 GB/s, include the runs of the baseline on most of the inputs used for the benchmark.
- The above average runs include some attempts of optimization presented in this chapter and have a throughput around 1 GB/s.
- The fast runs, with a throughput of at least 2 GB/s, include only runs of the parallel algorithms and one run of the baseline in a case where its optimizations can shine.

In this thesis' tables, the results are separated into two broad categories: the ones obtained in the repository [58], using algorithms on unicode characters, and the ones obtained in the repository [59], using algorithms on bytes. This distinction is important to keep in mind, as the algorithms based on the unicode alphabet manipulate unicode characters, called char, which Rust stores using four bytes, even when only one is needed.

2.3 Simple sequential algorithms

Our implementation is meant to explore new possibilities of implementation using algebraic methods. In order to evaluate these possibilities, we compare their performance to the crate `regex`, as presented in section 2.2.3. However, this crate differs greatly from our implementation. Indeed, it processes the regexes at runtime and can handle the extended syntax of regexes, including backreferences, which it can use for both data validation and pattern matching. Therefore, we added to our implementation some algorithms based on the classical approaches. Even though they are not nearly as well optimized as the state of the art, they give a baseline of what can be achieved when processing the regexes at compile time to produce efficient code. In this section, we describe our attempts at a baseline algorithm and present the results of the chosen algorithm on our benchmark.

2.3.1 A baseline for sequential execution

Executing an automaton involves a straightforward and sequential algorithm that verifies whether an input word belongs to a particular language. We implemented two algorithms which execute the automaton in two different ways. The first one relies on a backtracking approach and is not usable in practice, and the second one is based on Thompson's algorithm (see example 1.3 for an example of execution).

The naive sequential execution of the automaton. The first algorithm we implemented produced code that executes the naive sequential run of the non-deterministic automaton, using backtracking. However, the chosen framework enforces some constraints that prevented us from testing it. Indeed, the naive run is a depth-first run of the automaton, which selects one possible transition at each non-deterministic choice, and goes back if that choice does not lead to an accepting state. That requires us to be able to go back to one of the states where we made a choice. The only way to do this without a stack in our framework in a sequential program is to

use function calls, with one function per state, which rapidly results in a stack overflow when we consider long inputs.

Example 2.13. The sequence of instructions obtained from `regex-syntax`, after removing the instructions that are of no interest to us, is the following:

```
0 Split(1, 3)
1 'a'
2 'b' (goto: 0)
3 Match(0)
```

Note that this program corresponds to the minimal automaton of $(ab)^*$, which is given in Figure 2.7. The code obtained from the naive algorithm has as many helper functions as there

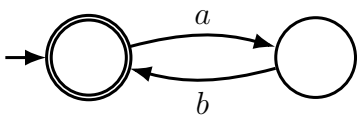


Figure 2.7 – The minimal automaton recognizing $(ab)^*$

are instructions in the program. The structure of the returned code is shown in algorithm 1.

Example 2.14. Consider the regex `(a|b)+x(a|b|c)*|(a|b|c)(a|b|c)+`. The naive algorithm will execute the naive NFA shown in Figure 2.8, and will first try to go to state 1, to match the sub-regex `(a|b)+x(a|b|c)*`. Thus, if it fails to find an x when in the state 1, it has to go back to the beginning of the word, as it is the index in which it was before leaving state 0. Thus, it may go back arbitrarily far in the input when backtracking.

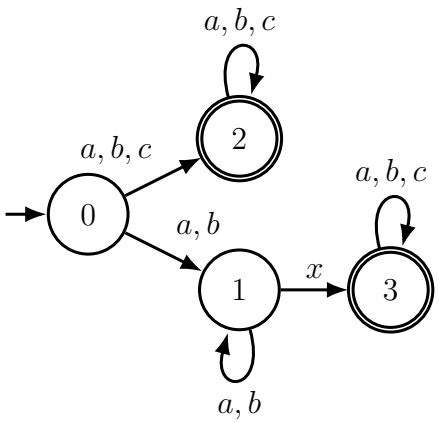


Figure 2.8 – A naive automaton for $(a+b)^+x(a+b+c)^* + (a+b+c)(a+b+c)^+$

One of our goals is to avoid backtracking, so the impossibility to test the naive algorithm

is not a grave concern.

Algorithm 1 : The code produced by the naive algorithm for $(ab)^*$

```
Function Start(word, index):
    out ← Check_a(word, index) OR Match(word, index);
    return out;
Function Check_a(word, index):
    if index < word.len() then
        if word[index] = a then
            return Check_b(word, index + 1);
        else
            return false;
        end
    else
        return false;
    end
Function Check_b(word, index):
    if index < word.len() then
        if word[index] = b then
            return Start(word, index + 1);
        else
            return false;
        end
    else
        return false;
    end
Function Match(word, index):
    return word.len() ≤ index;
Function Main(word):
    return Start(word, 0);
```

The determinized "on the fly" version. The other simple algorithm in our implementation is what we call the determinized version. Indeed, this algorithm is based on Thompson's algorithm (see section 2.1.3 for a more formal definition of the algorithm). First, it computes an NFA without ϵ-transition by taking the transitive closure of the ϵ-transition, then it produces the returned function by writing the execution of Thompson's algorithm on that automaton. This algorithm determinizes the automaton on the fly: it maintains a set of current states, and each time a letter is read, it constructs the new set of states by following all the possible transitions labeled by that letter and starting in a state from the current set. In our implementation, the new set is constructed with a series of conditional statements which compare the current letter read from the input to the intervals labelling the edges of the NFA, and enabling these edges only if the current state is their starting point.

Before determinizing the NFA on the fly, the algorithm removes the ϵ-transitions using their transitive closure (the automaton obtained for $(ab)^*$ is given in example 2.15).

Example 2.15. Consider the program shown in example 2.13, which recognizes the language $(ab)^*$. When we remove the ϵ-transitions, we remove the Split instruction, but not the Match,

as keeping the Match instructions simplifies the rest of the computation. Thus, we obtain the NFA shown in Figure 2.9. Note that the accepting state of that NFA has an acceptance condition: it accepts the input word if and only if the state is reached with the last letter of the word.

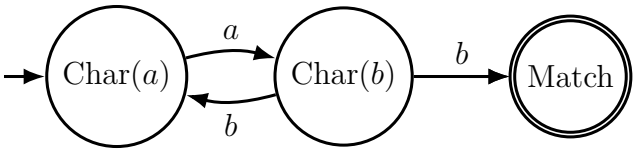


Figure 2.9 – The NFA obtained by removing the ϵ-transitions from the program corresponding to $(ab)^*$

Example 2.16. Let's consider once more the language $(ab)^*$. The program built by `regex-syntax` is given in example 2.13. In the determinized algorithm, we use a vector to keep track of the current states reached by that program. We keep the two Char instructions and the Match instruction as states: the Char instruction that checks if the current letter is an a becomes the state of index 0 in the code. The Char instruction that checks if the current letter is a b becomes the state of index 1. Finally, the Match instruction becomes the state of index 2. The code written by the determinized algorithm is given in algorithm 2. Note that the state of index 2 is useful only when the end of the word is reached.

Algorithm 2 : The code produced by the determinized algorithm for $(ab)^+$

```
Function Transition(letter, state):
    out_state ← [false, false, false];
    if state[0] AND letter = a then
        out_state[1] = true;
    end
    if state[1] AND letter = b then
        out_state[2] = true;
    end
    if state[1] AND letter = b then
        out_state[0] = true;
    end
    return out_state;
Function Main(word):
    state ← [true, false, true];
    for index ← 0 to word.len() do
        state ←Transition(word[index], state);
    end
    if state[3] then
        return true;
    end
    return false;
```

Using the determinized algorithm, we have a base version to test a sequential algorithm in our framework. However, if we want to consider more efficient algorithms, we need to

exploit some kind of parallelism, automata become unsuitable as they are inherently sequential, therefore such algorithms necessitate the use of alternative tools that are better suited for these purposes, yet capable of characterizing the same language classes. In the benchmark results, this version is called "determinize", and the variant where the automaton is explicitly determinized and minimized is called "deter-mini".

2.3.2 Benchmark results

The results are given in table 2.1. The algorithms working on unicode characters cannot be tested on the input *last* of the language $\Sigma^* ac^* a\Sigma^*$, as this input contains sequences of bytes that are invalid UTF-8. The corresponding results are marked with an X.

Most of the results presented here fall in the category of average runs. The most notable exception is the execution of the baseline algorithm (Rust's crate `regex`) on the word *last* in the language $acstara = \Sigma^* ac^* a\Sigma^*$, which is a fast run, ten times faster than on the other inputs and languages. The most probable cause of this considerable speed-up is the language itself: since Σ can match any byte, the algorithm can skip characters using simple bit-level parallel instructions to find the ones that are important to recognize the language. In that case, it searches for an a or the end of the input. Since the input is randomly generated with a uniform probability on all bytes, there are few occurrences of these important characters, and the algorithm skips most of the input, leading to this fast run. Consequently, this gives an upper bound on the throughput reachable with the baseline algorithm.

Another set of non-average runs comes from the execution of the minimized and determinized algorithm ("deter-mini") on the words that belong to $abstar = (ac^*b + c)^*$, which give above average runs with a throughput of almost 1 GB/s. Note that it is the case for both versions of the algorithm, the one on unicode characters and the one on bytes. Although we could expect this version to always be at least as fast as the non minimized variant ("determinize"), it is only the case for these inputs. There is no obvious explanation for this, and our best guess is that, in the cases where it is less efficient, the minimization changes the structure of the automaton in a way that is less efficiently handled by the compiler. This phenomenon probably also explains the fact that this algorithm is significantly more efficient of these input words.

The last set of above average runs comes from the execution of the determinized algorithm on the words that belong to the language $L_{\mathcal{R}} = (a|b|c)^* d(b|c|d)^*$. Again, the efficiency on this particular words could be due to the optimizations performed by the compiler. What's really interesting about this particular set of runs is that it is efficient only on unicode characters: on the node troll, the version on bytes is three times slower that the version on unicode characters. In this table, this is the only case that differentiate the two kinds of algorithms, as the other runs give strikingly similar results. The structure of the code is mostly the same, so we have no idea of what can cause that.

We can also compare the average results obtained on the node troll, although this might not be as relevant for the general case. It is interesting to note that the baseline (the crate `regex`) is significantly faster (with a multiplicative factor of approximately two) on bytes than on unicode characters, although each character of the input words is an ASCII character, encoded with only one byte. This is probably due to the fact that, when considering unicode characters, the baseline validates the UTF-8 encoding of the input, which requires more computation and more branching. This is not the case of the other algorithms, determinize and its minimized variant. In these algorithms, the validation of the encoding is included in the automaton. That automaton is very similar in both versions, since the only characters leading to states other

than the non-accepting sink states are ASCII characters. It would be interesting to run this benchmark on input words belonging to other languages, that accept words containing non-ASCII characters, in order to see the impact of the validation of UTF-8 on performance. With these ASCII inputs, the algorithms on bytes can be even slower than the unicode versions.

Lang.	Word	Algorithm						
		Char (Gchar/s)				Bytes (GB/s)		
		base	determinize	deter-mini	flex	base	determinize	deter-mini
acstara	*last*	*X*	*X*	*X*	*X*	7.0	0.42	0.26
abstar	*dense*	0.31	0.52	0.91	0.24	0.64	0.53	0.97
	average	0.31	0.52	0.91	0.24	0.64	0.53	0.97
	sparse	0.31	0.52	0.91	0.24	0.64	0.53	0.97
$L_{\mathbf{DA}}$	*balanced*	0.31	0.47	0.26	0.24	0.64	0.41	0.26
	decrease	0.31	0.47	0.26	0.24	0.64	0.41	0.26
	increase	0.31	0.47	0.26	0.24	0.64	0.41	0.26
$L_{\mathcal{R}}$	*first*	0.31	0.87	0.60	0.24	0.64	0.23	0.60
	last	0.31	0.87	0.60	0.24	0.64	0.32	0.60
	middle	0.31	0.87	0.60	0.24	0.64	0.26	0.60

Table 2.1 – The results of the sequential algorithms - troll

Remark 2.4. As shown in table 2.1, the tool flex performs poorly compared to Rust's `regex-syntax` crate, even though the latter processes the regex at runtime an thus cannot allow itself to optimize it as much as could be. This is one of the reasons why flex has not been chosen as baseline. It is due to the nature of the tool, which produces complex structures that are not well adapted for data validation. Notably, the regex is translated into a kind of transducer, in order to perform pattern matching on the input. The output is computed from the string that matches the regex and, in our setting, it can span the whole text. The resulting memory management dooms the performance of flex when compared to programs that only validate data and do not require unbounded memory for output handling. Indeed, although the tools provided by `regex-syntax` are meant for pattern matching, they are also optimized for data validation, which can be performed on its own, without the complex computations necessary for pattern matching. This gives `regex-syntax` a significant advantage over flex. Moreover, when we attempted to run flex on the language $acstara = \Sigma^* ac^* a \Sigma^*$, it seemed to struggle, and the computation did not finish in one full hour. Thus, it seems that this kind of regex antagonizes flex in some way, where they are especially well handled by `regex-syntax`. This makes flex less versatile, giving us one more reason to avoid choosing it as baseline.

2.4 Semigroups and parallel algorithms

In section 2.3, we presented our baseline which produces an NFA at compile time and runs it on the input at runtime using Thompson's algorithm. This algorithm executes the NFA without much more processing, and thus runs sequentially through the input text. In order to obtain a more efficient algorithm, we need to get rid of this sequential way of processing. As we saw in section 1.2, semigroups have the advantage of having an associative product and

they are directly linked to regular languages. Therefore, in this section, we try to leverage the associativity of semigroups to design efficient algorithms to recognize regular languages, while still computing the necessary data structure – in this case, semigroups and their properties – at compile time. We present several algorithms used in our implementation. Most of these algorithms are still sequential and aim at evaluating the performance cost of using semigroups instead of an automaton. One algorithm that we present is parallel and leverages as much as possible the inherent parallelism offered by the associative product. We also present some potential improvements to the sequential algorithms that use semigroups. These potential improvements are based on observation and our limited knowledge of hardware, but are still interesting to consider. Both the basic algorithms and their tweaked variants are evaluated in benchmarks given in this section.

2.4.1 Common methodology for the algorithms using semigroups

All the algorithms presented in this section have in common the conversion from an automaton to a semigroup associated with this automaton. The code produced at compile time is also of a similar shape in every algorithm, even though the specific contents differ. In this subsection, we present these common parts of the algorithms, which are computed at compile time.

The transition semigroup. As presented in section 1.2.2, there are two links between regular languages and semigroups that we can use: the transition semigroup of an automaton or the syntactic semigroup of a language. Both the syntactic and transition semigroups (or monoids) organize letters and words based on their impact on the associated automaton, but the transition semigroups are more directly linked to the structure of the automata, so we chose to focus solely on the transition semigroups, setting aside the syntactic semigroups.

Given an NFA on the unicode alphabet or the byte alphabet, each algorithm computes its associated transition semigroup by removing the ϵ-transitions using their transitive closure (the automaton obtained for $(ab)^*$ is given in example 2.15), then computing the transition semigroup of the resulting NFA as presented in the next two paragraphs.

The generators of the transition semigroup. To compute the transition semigroup, each algorithm begins by computing the generator elements of that semigroup. These generators are the elements corresponding to the letters of the chosen alphabet. In most of the algorithms, we chose to use boolean matrices for the elements of the semigroup. In these algorithms, for each letter (unicode character or byte) present in the automaton, we compute the transition matrix of that letter, as defined in definition 1.7. See example 2.17 for the computation of the generators corresponding to $(ab)^*$. Note that the automaton considered for this computation is not necessarily the minimal automaton that recognizes the language, and thus the transition matrices may be different.

Example 2.17. Consider again the program shown in example 2.13, which recognizes the language $(ab)^*$. In that program, there are only two letters explicitly used: a and b. Thus, we compute only the transition matrices of these two letters, as the others' transitions are all equal to the constant function that leads to an implicit sink state. As explained in the previous paragraph, we removed the ϵ-transitions, so there are only three states left in the automaton (see Figure 2.9), which correspond to the two Char instructions and the Match instruction. Consequently, the transition matrices we compute are of size 3×3. Now, consider the effects of the letters on that NFA: the letter a allows the program to go from the state Char(a) to

Char(b), and the letter b allows a nondeterministic choice from the state Char(b) to either Char(a) or Match. Thus, the transition matrices are as follows:

$$T_a = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \text{ and } T_b = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Note that we do not compute the transition matrix of each letter in the alphabet. This is due to the fact that letters not present in the automaton necessarily lead to the null transition matrix, which is always a non-accepting and idempotent semigroup element. This is dealt with without computing the transition matrices.

If two letters have the same transition matrix, they are equivalent in the automaton. We keep only one representative letter, but we keep the others in memory. In practice, we construct a bijection from the set of representative letters to their transition matrices, and a bijection from the set of representative letters to the sets of letters represented by each of them. Using these two bijections and the sets I and F of the initial and final states, we can construct the transition semigroup associated with the NFA without ϵ-transitions.

Constructing the transition semigroup. With the previous step completed, we already have the generators of the transition semigroup. Indeed, as a word is a sequence of letters, we can multiply the transition matrices of the letters to obtain the transition matrix of the word as defined in definition 1.8. Thus, we only need to multiply the generator elements until we cannot obtain any new element by multiplying old ones with the generators. During this process, we construct a bijection from some representative words (which letters are the representative letters) to their transition matrices. To this end, we use a queue in which we initially store all the generator matrices. As long as this queue is not empty, we remove one matrix from the queue and multiply it with each generator matrix. If one of the products is a matrix not yet seen, we add it to the bijection and to the queue. Once the queue is empty, the bijection contains all the matrices which are elements of the semigroup.

Example 2.18. In example 2.17, we showed that the generators of the transition semigroup computed by our algorithms for $(ab)^*$ are

$$T_a = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \text{ and } T_b = \begin{pmatrix} 0 & 0 & 0 \\ 1 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

With this, we compute the transition semigroup by multiplying matrices until we reach a fixed point. In this case, we initiate the queue where the first element is T_a and the second one is T_b. We start by retrieving the first element of the queue, which is T_a, and we multiply it by both T_a and T_b. We begin by computing $T_{aa} = T_a^2$, which is the null matrix of size 3×3. This matrix is not equal to T_a or T_b, so we keep it as an element of the semigroup and we add it to the queue. Then we compute $T_{ab} = T_a T_b$, and we obtain

$$T_{ab} = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

This matrix is different from the three that we have seen so far, so we keep it as a new element, which we add to the queue. We finished dealing with T_a, so we retrieve again the first element

of the queue, which is T_b, and we proceed the same way. We first compute $T_{ba} = T_b T_a$, and we obtain

$$T_{ba} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

This matrix is different from the three that we have seen so far, so we keep it as a new element, which we add to the queue. Then, we compute $T_{bb} = T_b^2$, and we obtain $T_{bb} = T_{aa}$, which is not a new element. So, we directly retrieve the next element of the queue, which is T_{aa}. We obtain $T_{aa}T_a = T_{aa}T_b = T_{aa}$, so there are no new elements to add to the queue. The next element is T_{ab}, and we obtain $T_{ab}T_a = T_a$ and $T_{ab}T_b = T_{aa}$. Again, we have no new elements. With the next element in the queue, T_{ba}, we obtain $T_{ba}T_a = T_{aa}$ and $T_{ba}T_b = T_b$. Now there are no more elements in the queue, so the semigroup is composed of five elements: T_a, T_b, T_{aa}, T_{ab} and T_{ba}.

Each product made during the construction of the semigroup is stored, which constructs the right Cayley graph of the semigroup (see section 1.2.4 for the definition of the two Cayley graphs). We will use this graph in our algorithms to compute products of elements of the semigroup, instead of re-computing the product each time. This also allows us to assign indices to the matrices and compute the product of the semigroup's elements using only these indices instead of storing possibly large boolean matrices.

Example 2.19. In example 2.18, we saw the transition semigroup obtained from the program that `regex-syntax` computes for $(ab)^*$. During the computation of that semigroup, we also computed the information necessary to build the right Cayley graph, since we computed the product of each element with each generator. Using that information, we obtain the graph shown in Figure 2.10.

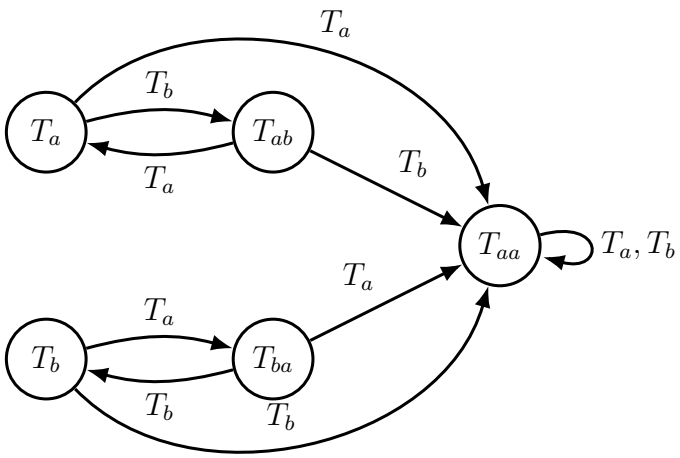


Figure 2.10 – The right Cayley graph for the transition semigroup of $(ab)^*$

The final function. Since we are working with semigroups, the function returned by our algorithms always has some key components, namely two helper functions to get the elements corresponding to the letters of the input and to multiply elements, and the main body of the function which articulates the two helper functions. The helper function which returns the element corresponding to a given letter is mostly the same in all the tested algorithms: a

series of conditional statements which compare the input letter to the ranges of letters that are generators in the semigroup. An attempt was made at replacing these conditional statements by some pattern matching using match statements, but this did not change the performance, and thus these versions are no longer present on the repository. The product, however, is handled differently depending on the chosen algorithm. The details are given in the respective descriptions of the algorithms. The articulation between the two can change a bit, but the idea stays the same: use the first helper function to convert letters into the corresponding semigroup elements, then use the second helper function to compute the product of the obtained elements. In every algorithm, if the input is empty, it is accepted if and only if the empty word is accepted by the original NFA (this piece of information is stored in the semigroup).

2.4.2 The algorithms

The properties of the semigroups lead us to consider three base algorithms, more or less naive. Two of these algorithms return a function that runs sequentially through the input word and stores the current element to multiply it with each new generator obtained from the letters. These algorithms are not meant to be used in practice, but they allow us to measure the impact of using a semigroup on the performance, and to experiment with possible improvements based on the semigroups' properties. These two algorithms have led to variations meant to improve the actual computation time, and will be detailed in section 2.4.4. The last algorithm computes the generator elements and their product using parallelism.

The naive sequential use of the semigroup. Before using the semigroups for parallelism, we produced a basic sequential algorithm, which is meant to be compared to the sequential algorithms presented in section 2.3.1. That sequential algorithm works as follows: if the input is not empty, it initializes a variable to the element corresponding to the first letter. Then, it sequentially reads each letter of the input, computes the corresponding element of the semigroup, and computes the product with the stored element to obtain the next element to store. This product is done using conditional statements which depend on the values of the stored element and the generator corresponding to the input letter.

Example 2.20. Let's get back to the language $(ab)^*$. The corresponding program built by `regex-syntax` is shown in example 2.13. We build the output code based on the transition semigroup of that program (see example 2.18) and its right Cayley graph (see example 2.19). In order to represent the elements in a concise way, we use integers instead of the matrices. The elements are numbered according to the order in which they were computed. In this example, the element T_a is represented by 0, T_b is represented by 1, T_{aa} is represented by 2, T_{ab} is represented by 3, and T_{ba} is represented by 4. The output code is composed of several functions, among which the most important is the product function presented in the function Multiply below. It is important to note that the order of the conditional statements in that function is not necessarily the one presented here.

The code produced is composed of this product function and the functions presented in algorithm 3, which are

- The ElementFromLetter function, which returns the index of the element corresponding to the current letter.
- The IsAccepted function, which returns *true* if and only if the input index indicates an element corresponding to an accepting path in the original automaton.

Fonction Multiply(*element*, *generator*), the product function produced by the naive semigroup algorithm ("stamp") for $(ab)^*$

Input : *element*, *generator*
Output : *element* × *generator*
if *element* = 0 **AND** *generator* = 0 **then**
  return 2;
end
if *element* = 0 **AND** *generator* = 1 **then**
  return 3;
end
...
if *element* = 4 **AND** *generator* = 1 **then**
  return 2;
end
return 2;

- The Main function, which articulates the three others to compute the element associated with the input word and determine if the word is accepted.

In the benchmark results, this version is called "stamp", and the variant where the automaton is explicitly determinized and minimized before the construction of the semigroup is called "stamp-mini".

Using a table for the semigroup multiplication. This version is a variation of the previous one, where the product of the stored element and the generator corresponding to the next letter is done by looking in a table instead of matching the values.

Example 2.21. Let's continue with the example of $(ab)^*$. The algorithm is mostly the same as the previous one, except for the product function, which uses a table to compute the product. This function is called Multiply, as before, and is given below.

Fonction Multiply(*element*, *generator*), the product function that uses a table for the transition semigroup of $(ab)^*$

Input : *element*, *generator*
Output : *element* × *generator*
product_table ← [2, 3, 2, 4, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 1, 2];
return *product_table*[*element* × 3 + *generator*]

The rest of the code is the same as given in algorithm 3.

This algorithm, along with those presented in section 2.4.4, allows us to measure the impact on performance of the algorithm and structures used to compute the products of elements. Using a table avoids branching that depends on the values of the elements, instead relying on lookups. This is meant to reduce the delay linked to branches and branch mispredictions, and thus increase the efficiency of the implementation.

In the benchmark results, this version is called "table", and the variant where the automaton is explicitly determinized and minimized before the construction of the semigroup is called "table-mini".

Algorithm 3 : The code produced by the naive semigroup algorithm for $(ab)^*$

```
Function ElementFromLetter(word, index):
    if word[index] = a then
        return 0;
    end
    if word[index] = b then
        return 1;
    end
    return 2;
Function IsAccepted(element):
    if element = 3 then
        return true;
    end
    return false;
Function Main(word):
    if word.len() = 0 then
        return true;
    end
    product ← ElementFromLetter(word,0);
    for index ← 1 to word.len() − 1 do
        generator ← ElementFromLetter(word,index);
        product ← Multiply(product,generator);
    end
    return IsAccepted(product);
```

Performing products in parallel. Finally, we implemented a parallel algorithm. This version computes in parallel the elements corresponding to all the letters, then computes in parallel the product of all these elements by regrouping them by pairs. This regrouping is left to a heuristic and thus is not fixed in the algorithm, as we use the Rust crate named `rayon` to perform computations in parallel (see [47] for the original idea behind `rayon`). However, the product function has to deal with the product of any two elements in the semigroup, not just the product of an element and a generator. Therefore, the number of branches in the product function is greater than in the two other versions.

In the benchmark results, this version is called "parallel", and the variant where the automaton is explicitly determinized and minimized before the construction of the semigroup is called "parallel-mini".

2.4.3 Benchmark results

The algorithms described in the previous sections have been tested on the same languages and words as the other benchmarks in this chapter (see section 2.2.4 for a detailed description).

Table 2.2 – The results of the semigroup-based algorithms - troll

Lang.	Word	Algorithm									
		Char (Gchar/s)					Bytes (GB/s)				
		base	stamp	stamp-mini	table	parallel (32 cores)	base	stamp	stamp-mini	table	table-mini
acstara	*last*	X	X	X	X	X	7.0	0.15	0.16	0.25	0.22
abstar	*dense*	0.31	0.20	0.56	0.63	12.0	0.64	0.33	0.51	0.55	0.64
	average	0.31	0.32	0.51	0.63	12.0	0.64	0.39	0.39	0.55	0.64
	sparse	0.31	0.20	0.60	0.64	13.0	0.64	0.13	0.43	0.55	0.64
$L_{\mathbf{DA}}$	*balanced*	0.31	0.048	0.087	0.55	7.9	0.64	0.038	0.076	0.15	0.17
	decrease	0.31	0.045	0.091	0.54	7.4	0.64	0.034	0.083	0.15	0.17
	increase	0.31	0.050	0.083	0.55	8.5	0.64	0.042	0.069	0.16	0.17
$L_{\mathcal{R}}$	*first*	0.31	0.11	0.18	0.54	7.5	0.64	0.20	0.20	0.26	0.24
	last	0.31	0.097	0.18	0.54	8.5	0.64	0.25	0.20	0.25	0.25
	middle	0.31	0.092	0.18	0.55	8.0	0.64	0.24	0.19	0.26	0.24

Panel A: Results of the sequential semigroup versions, and the parallel version on characters

Language	Word	Algorithm (GB/s)									
		base	stamp	parallel				parallel-mini			
				32 cores	16 cores	8 cores	4 cores	32 cores	16 cores	8 cores	4 cores
acstara	*last*	7.0	0.15	3.1	1.9	1.2	0.62	3.6	2.6	1.4	0.67
abstar	*dense*	0.64	0.33	7.5	5.4	3.1	1.7	8.0	6.6	4.1	2.1
	average	0.64	0.39	5.7	4.5	2.7	1.5	8.9	6.5	4.1	2.1
	sparse	0.64	0.13	5.6	4.3	2.6	1.3	8.4	6.3	3.6	2.0
$L_{\mathbf{DA}}$	*balanced*	0.64	0.038	2.6	1.7	0.98	0.50	2.9	1.9	0.98	0.52
	decrease	0.64	0.034	2.8	1.6	0.96	0.50	2.7	1.8	0.94	0.49
	increase	0.64	0.042	3.0	1.7	1.0	0.53	3.1	1.8	1.0	0.53
$L_{\mathcal{R}}$	*first*	0.64	0.20	4.2	2.5	1.6	0.81	3.8	2.4	1.5	0.74
	last	0.64	0.25	4.0	2.5	1.4	0.73	3.8	2.3	1.4	0.74
	middle	0.64	0.24	4.3	2.5	1.5	0.76	3.4	2.7	1.4	0.76

Panel B: Results of the parallel versions on bytes depending on the maximum number of cores used

Unicode versus byte algorithms. Recall that the results are separated into the ones using algorithms on unicode characters, and the ones using algorithms on bytes, which may impact the results, as the algorithms based on the unicode alphabet manipulate unicode characters, called char, which Rust stores using four bytes, even when only one is needed. However, in table 2.2, it seems that using the unicode alphabet does not lead to a significant decrease in throughput with respect to the algorithms using the byte alphabet. Indeed, the results of both versions fall in the same categories, or close to them. The few cases where the byte version has a greater throughput have a negligible difference with the corresponding unicode version. In fact, the algorithms on the unicode alphabet tend to be better than the ones on the byte alphabet. This is especially noticeable with the runs of the "table" algorithm on the words in $L_{\mathbf{DA}} = (a+b+c)^*b(a+b+c)^*d(a+b+c+d)^*c(a+d)^*$ and the "parallel" algorithm, in particular on the words in $L_{\mathbf{DA}}$. Indeed, even though the results fall in the same categories, the unicode versions can be almost three times faster than their byte counterparts. We have no clue as to why that is.

Another notable difference between the algorithms on unicode characters and the ones on bytes lies within the executions of the "table" algorithm. This algorithm uses a table to compute the products of successive elements of the semigroups, which removes most of the conditional branching. This should lead to results that barely depend on the language (given that the semigroups are of similar sizes). This is the case for the version on unicode characters, but not the other one, which seems to improve the basic "stamp" algorithm but still depends on the language. In order to explain this, we rely on the tool perf stat to measure the number of branches and the number of mispredicted branches. The results allow for an educated guess: both versions indeed have much less conditional branches, but the version on unicode characters mispredicts very few branches, almost none for most input words, whereas the version on bytes fails to predict a significant amount for the languages $L_{\mathbf{DA}}$ and $L_{\mathcal{R}}$ (around 12% for words in $L_{\mathbf{DA}}$ and 8% for words in $L_{\mathcal{R}}$), leading to this change of performance. In this algorithm, most of the conditional branching happens when the semigroup element corresponding to a character is computed. Thus, it is probable that the order of the instructions used for this computation depends on the version of the algorithm and heavily influences the results: in the version dealing with unicode characters, the order of instructions is probably ideal for the branch predictor, contrary to the version on bytes, which antagonizes it.

Particularly slow runs. Among these results, some fall in the category of particularly slow runs. This is notably the case of the "stamp" and "stamp-mini" algorithms on the words belonging to $L_{\mathbf{DA}}$, even if the "stamp" algorithm is not particularly efficient on the words of $L_{\mathcal{R}}$ either. In general, these two algorithms are the less efficient ones, but they are especially slow on the words in $L_{\mathbf{DA}}$. This could be due to the complexity of the language itself, though the syntactic semigroup is still rather small. One probable cause of this lack of efficiency lies within the structure of the language as, once the semigroup reaches the $\mathcal{J}$-class with the greatest $\mathcal{J}$-depth, it circles a lot between the $\mathcal{H}$-classes of bd and bdc until it reaches the last occurrence of c of the input words. This can antagonize the branch predictor and lead to poor performances. This can explain why all algorithms are less efficient on the words of $L_{\mathbf{DA}}$ than on the others.

To verify this, we used the tool perf stat. This gave us more insight, as the execution of the "stamp" and "stamp-mini" algorithms on the words in $L_{\mathbf{DA}}$, both the unicode and byte versions, has significantly more branches than on the words in $(ac^*b+c)^*$ or in $L_{\mathcal{R}}$ (approximately six times for "stamp" and three times for "stamp-mini"). More importantly, a non-negligible percentage of these branches was mispredicted: around 1% for stamp, and around 3% for

"stamp-mini", against almost none for the language $(ac^*b + c)^*$ and 5% for the language $L_{\mathcal{R}}$ in both variants of the algorithm. A larger proportion of the branches were mispredicted for $L_{\mathcal{R}}$ but, since this language leads to significantly less branches, this results in more mispredicted branches for $L_{\mathbf{DA}}$ (nearly twice as much as for $L_{\mathcal{R}}$). This helps explain the performance gaps between these languages.

Determinization. As for the results shown in table 2.1, the results of the "determinize" algorithm are not as good as one could hope. Essentially, this variant of the algorithms can slightly improve the efficiency, but does not bring significant speedups. This was to be expected since the languages used here are quite simple, and the NFAs produced by the crate `regex` are very close to DFAs. Consequently, determinization does not change significantly the structure of the automaton from which the semigroup is built.

Fast runs. Finally, the only fast runs, except for the execution of the baseline on the word of $\Sigma^*ac^*a\Sigma^*$, are the results of some execution of the "parallel" or "parallel-mini" algorithms. We can note that the throughput scales nicely between the versions on four and eight cores, but not as well afterward. This is probably due to the cost of the communication between the two CPUs of the machine.

2.4.4 Propositions of potential improvements

When experimenting with the programs using conditional statements or pattern matching to compute the products in semigroups, we observed that the order of the statements greatly impacts the performance. Thus, we tried different approaches to try to optimize the order and improve the performance.

Factorizing the code of the product. The products in the base versions are done by matching the values of the stored element and the generator corresponding to the new letter. A way to improve that is to regroup some cases that have similar properties. In this version, given G the set of generators of the semigroup, we treat the following cases in order:

- If the semigroup has an identity element e, we regroup all the products of the form $e._S g$, where g is a generator element.
- If the semigroup has an identity e which is also a generator, we regroup all the products of the form $s._S e$, where s is an element of the semigroup.
- For each element s of the semigroup, we find one element t such that $|\{g \in G \mid s._S g = t\}|$ is maximal, and we regroup all the products of the form $s._S g = t$. The statement obtained from these products is a default statement that depends only on the left element of the product, and is put at the end of the code for the product.

Example 2.22. Consider the rational language $L = (ac^*b + c)^*$ and its syntactic semigroup $S = \{0, 1, 2, 3, 4, 5\}$, where 0 corresponds to the letter a, 1 corresponds to the letter b, 2 corresponds to the letter c, 3 corresponds to ab, 4 ba, and 5 to aa (5 corresponds to the non-accepting sink state of the automaton). The inner product of S is such that $\pi_S(01) = 3$, $\pi_S(10) = 4$, $\pi_S(30) = 0$, $\pi_S(41) = 1$ and, for any element $s \in S$, $\pi_S(s2) = \pi_S(2s) = s$ and $\pi_S(s5) = \pi_S(5s) = 5$. All the other products have 5 as a result.

If we process the semigroup S with the naive semigroup algorithm, we obtain many redundant conditional statements. Instead, we can use the factorization algorithm. That algorithm first looks for an identity element, and finds 2, since, for any element $s \in S$, $\pi_S(s2) = \pi_S(2s) = s$. It regroups all the products of the form $2._S g$, where $g \in \{0, 1, 2\}$, in one conditional statement, and puts that statement at the beginning of the multiplication function. Then, it searches for an identity element which is also a generator, and finds again 2. Consequently, it regroups all the remaining products of the form $s._S 2$, where $s \in S$, in one conditional statement, and puts that statement at the beginning of the multiplication function, just after the previous one. Finally, for each element $s \in S$, it tries to regroup as many of the remaining products of the form $s._S g$ as possible to obtain one single statement that is put at the end of the function to obtain a default case which does not need to check the value of the generator used in the product. This can have multiple results depending on the way it is treated. For example, it can result in the multiplication function FactorizedMultiply.

Remark 2.5. The function FactorizedMultiply presented here could be factorized further, by regrouping the statements that return the element 5. To do this, we could add new factorization rules, which would factorize sinks and elements frequently occurring in the statements. However, it requires careful crafting, as it can interfere with the "default" cases introduced by the third rule.

Regrouping as many statements as possible diminishes the number of statements. Even though it probably does not optimize the order, it might avoid some unnecessary computations and some branch mispredictions.

Remark 2.6. Our factorization algorithm can modify the order of the statements, which in turn can impact the performance, therefore factorization does not even guarantee that the performance will not decrease. This remark applies to all the algorithms using factorization.

The factorization algorithm orders the statements by putting them in three lists: the list containing the products that leave the left element as is, the list containing the products that the algorithm cannot factorize, and the list containing the factorized products. The lists are then concatenated in the order given by the previous sentence.

In the benchmark results, this version is called "facto", and the variant where the automaton is explicitly determinized and minimized before the construction of the semigroup is called "facto-min".

Ordering the product by $\mathcal{R}$-classes. In this version, we keep track of the $\mathcal{R}$-class of the stored element and use it, through pattern matching, to call two functions meant to compute only the products of pairs of elements such that the left element is in one given $\mathcal{R}$-class. The first function computes the product if and only if the result is in the same $\mathcal{R}$-class as the left element of the pair being multiplied, and the second one, called only if the first one fails, computes the product if and only if it is not in the same $\mathcal{R}$-class as the left element.

Example 2.23. Consider again the syntactic semigroup S of $(ac^*b+c)^*$, given in example 2.22. If, instead of factorizing the product, we regroup it depending on the $\mathcal{R}$-class of the current element, then we first have to determine what the $\mathcal{R}$-classes are. There are four of them: one trivial class containing only the identity element 2 (which corresponds to the neutral letter c), one containing the non-accepting sink element 5 (which corresponds to the word aa), one containing the elements 0 and 3, and one containing 1 and 4.

Thus, we obtain eight functions, two for each of these $\mathcal{R}$-classes. The main product function, which articulates the other eight, is shown in algorithm 4. It calls helper functions depending

Fonction FactorizedMultiply(*element*, *generator*), the product function that factorizes the products in the transition semigroup of $(ac^*b + c)^*$

```
Input : element, generator
Output : element × generator
if element = 2 then
    return generator;
end
if generator = 2 then
    return element;
end
if element = 0 AND generator = 0 then
    return 5;
end
if element = 1 AND generator = 1 then
    return 5;
end
if element = 3 AND generator = 1 then
    return 5;
end
if element = 4 AND generator = 0 then
    return 5;
end
if element = 0 then
    return 3;
end
if element = 1 then
    return 4;
end
if element = 3 then
    return 0;
end
if element = 4 then
    return 1;
end
if element = 5 then
    return 5;
end
```

on the $\mathcal{R}$-class of the last computed element. For example, if that class is 1, the function tries to call ProductStaysInROne, which returns the product if and only if that product is in the $\mathcal{R}$-class 1. If that function fails, then ProductFallsOutROne is called. That helper function computes the product if and only if that product is in a different $\mathcal{R}$-class, which is necessarily the case here.

Doing this regroups the statements by $\mathcal{R}$-class. Since we only use the stored element as the left element of the products, either the product is in the same $\mathcal{R}$-class as the stored element, or its $\mathcal{J}$-depth is strictly greater (by definition of an $\mathcal{R}$-class). Thus, the same function will

Algorithm 4 : The product function produced by the semigroup algorithm regrouping the products by $\mathcal{R}$-class for $(ac^*b + c)^*$

```
Function Multiply(product,generator,class):
    if class = 0 then
        if ProductStaysInRZero(product,generator) =Some(element) then
            return (element,0);
        else
            return ProductFallsOutRZero(product,generator);
        end
    end
    ...
    if class = 3 then
        if ProductStaysInRThree(product,generator) =Some(element) then
            return (element,3);
        else
            return ProductFallsOutRThree(product,generator);
        end
    end
Function Main(word):
    if word.len() = 0 then
        return true;
    end
    (product,class) ← ElementAndClassFromLetter(word,0);
    for index ← 1 to word.len() − 1 do
        generator ← ElementFromLetter(word,index);
        (product,class) ← Multiply(product,generator,class);
    end
    return IsAccepted(product);
```

be called successively multiple times, as long as the stored product is in the corresponding $\mathcal{R}$-class. Each one of the functions that compute the product if and only if it is not in the same $\mathcal{R}$-class as the left element can be called only once. Consequently, we hoped that regrouping the products that stay in the same $\mathcal{R}$-class, and separating them from the products that increase the $\mathcal{J}$-depth, could improve the performance.

Example 2.24. Consider again the syntactic semigroup S of $(ac^*b+c)^*$, given in example 2.22, and the text *ccaccccbacbccabccccccaa*. If we associate each letter with its corresponding semigroup element, we obtain the word 2202222102122012222200. When computing the sequential product of that word, we start with the element 2, in the first $\mathcal{R}$-class. The first products also give the element 2, until we reach the first occurrence of 0. Then, we obtain the element 0, which is in the $\mathcal{R}$-class composed of the elements 0 and 3. From there, the successive products will stay in that $\mathcal{R}$-class until the last letter is reached, where the new $\mathcal{R}$-class will be the one containing only the element 5.

When the semigroup has few $\mathcal{R}$-classes with respect to the length of the input word, the series of elements obtained with sequential products necessarily stays in the same $\mathcal{R}$-class for a long time, at least once. This supports the idea of trying to regroup the products inside the

different $\mathcal{R}$-classes to accelerate the products computed in the long infixes that stay in the same $\mathcal{R}$-class.

In the benchmark results, this version is called "$\mathcal{R}$-stamp", and the variant where the automaton is explicitly determinized and minimized before the construction of the semigroup is called "$\mathcal{R}$-stamp-mini".

Ordering by $\mathcal{R}$-classes and staying as much as possible in the same small function. The above version does not use efficiently the fact that we might stay a long time inside any $\mathcal{R}$-class. As explained above, there can be only a finite number of changes of $\mathcal{R}$-class, number bounded by the depth of the semigroup, so each function computing the products that stay in the same $\mathcal{R}$-class can be called numerous times, adding an overhead in performance to search for the right product function when it is assured to often be the same. We might be able to get rid of this overhead by having one function by $\mathcal{R}$-class, in which we stay as long as the current element is in that $\mathcal{R}$-class. This avoids some function calls and we hoped that it would help the compiler. Thus, the program begins by finding the $\mathcal{R}$-class of the first letter, then calls the corresponding function. That function will in turn make the calls to the two product functions as long as the current element is in the right $\mathcal{R}$-class. If the $\mathcal{R}$-class changes, a call to the corresponding function is made.

Example 2.25. Consider again the syntactic semigroup S of $(ac^*b+c)^*$, given in example 2.22. This alternative version of the algorithm regrouping the products by $\mathcal{R}$-class uses intermediary functions, one for each $\mathcal{R}$-class. For the semigroup S, we obtain four intermediary functions, in addition to the ones that actually compute the product. These additional functions replace the pattern matching that calls the product functions depending on the current $\mathcal{R}$-class, and call only the right product functions. For example, the function dealing with the $\mathcal{R}$-class that contains 0 and 3 is the following:

Fonction ProcessClassOne(*word*, *element*, *index*), the product function that orders the $\mathcal{R}$-classes and uses intermediary functions for the transition semigroup of $(ac^*b+c)^*$

```
Input : word, element, index
Output : element × word[index]
for i ← index to word.len() − 1 do
    generator ← ElementFromLetter(word,i);
    if ProductStaysInROne(element,generator)=Some(new_el) then
        element ← new_el;
    else
        (element,class) ← ProductFallsOutROne(element,generator);
        if class = 3 then
            return ProcessClassThree(word,element,i + 1);
        end
    end
end
return element;
```

Note that, since the only other $\mathcal{R}$-class reachable from the $\mathcal{R}$-class 1 is the class 3, that class is the only one considered by the function ProcessClassOne. This way, we might be able to reduce delays due to branching by considering only the reachable classes, contrary to what is done in the previous algorithm. With these intermediary functions, we do not need to return

to the main function before reaching the end of the input word. Thus, the main function is the following:

Fonction Main(*word*), the main function of the algorithm that orders the $\mathcal{R}$-classes and uses intermediary functions

```
element ← ElementAndClassFromLetter(word,0);
if class = 0 then
    product ← ProcessClassZero(word,element,1);
else
    ...
    if class = 3 then
        product ← ProcessClassThree(word,element,1);
    end
end
return IsAccepted(product);
```

In the benchmark results, this version is called "$\mathcal{R}$-ord", and the variant where the automaton is explicitly determinized and minimized before the construction of the semigroup is called "$\mathcal{R}$-ord-mini".

Factorizing, ordering, and separating the product. This version has the same structure as the previous one: the main function calls the function associated with the $\mathcal{R}$-class of the first letter of the input, and that function only calls the two product functions that multiply an element of the $\mathcal{R}$-class with a generator, until the product is not in the $\mathcal{R}$-class anymore, in which case the function associated with the new $\mathcal{R}$-class is called. The difference with the previous version is that this one factorizes the products in the functions computing the product by $\mathcal{R}$-class, using similar factorization rules as the factorized algorithm. These rules, given a semigroup S, an $\mathcal{R}$-class R of S and a set of products which left element is in that $\mathcal{R}$-class (be it the products that stay in the class or the ones that fall out of it), are as follows:

- If there is a generator g such that, for any element $s \in R$, $\pi_S(sg) = s$, we regroup all the products of the form $s._S g$, where s is an element of the $\mathcal{R}$-class.
- For each element $s \in R$, we find one element t such that $|\{g \in G \mid s._S g = t\}|$ is maximal, and we regroup all the products of the form $s._S g = t$.

The factorization is less efficient than it can be when we factorize the complete product table, and there may be more cases to handle overall, but these cases are separated by $\mathcal{R}$-class, which can reduce delays due to branching.

Example 2.26. On the syntactic semigroup of $(ac^*b + c)^*$, this algorithm is not much more efficient than the previous one, since there is little to factorize inside the functions of the form ProductStaysInRX and ProductFallsOutRX (with X the number of the $\mathcal{R}$-class). The program is mostly the same. We can note that the function ProductStaysInRThree, which deals with the products staying in the non-accepting sink class, is reduced to one conditional statement. Indeed, since this class is a sink and contains only one element, the function ProductStaysInRThree can be factorized to return the only element of the class.

However, if we consider a semigroup with large $\mathcal{R}$-classes with a large number of products leading to the same elements, we can gain potentially a lot by using this algorithm, since it allows us to factorize these products. For it to be interesting, the semigroup needs to have

enough generators which have similar effects on some $\mathcal{R}$-class. For example, suppose that we have a semigroup S and an $\mathcal{R}$-class of S containing three elements, and three generators. Let's denote the generators by 0, 1, 2, and the elements by 3, 4, and 5. Suppose that the products in this $\mathcal{R}$-class are the following: $\pi_S(30) = 4$, $\pi_S(31) = \pi_S(32) = 5$, $\pi_S(40) = 5$, $\pi_S(41) = \pi_S(42) = 3$, $\pi_S(50) = 3$, $\pi_S(51) = \pi_S(52) = 5$. Then, the function ProductStaysInRX (where X is the identifier of the class) will factorize these products and have six conditional statements.

Fonction ProductStaysInRX(*element*, *generator*), the product function that computes the products staying in the $\mathcal{R}$-class R in the algorithm "ord-facto"

Input : *element*, *generator*
Output : *element* × *generator*
if *element* = 3 **AND** *generator* = 0 **then**
  return 4;
end
if *element* = 4 **AND** *generator* = 0 **then**
  return 5;
end
if *element* = 5 **AND** *generator* = 0 **then**
  return 3;
end
if *element* = 3 **then**
  return 5;
end
if *element* = 4 **then**
  return 3;
end
if *element* = 3 **then**
  return 5;
end

We hoped that this version would be more efficient than the previous one, as each small product function is at least as small as in the previous version of the algorithm.

In the benchmark results, this version is called "ord-facto", and the variant where the automaton is explicitly determinized and minimized before the construction of the semigroup is called "ord-facto-min".

Factorizing and ordering the product without separating. Once more, this algorithm is a variation of the previous one: it also has one main function per $\mathcal{R}$-class, which deals with the products as long as they stay in the associated class and calls the appropriate function if the class changes. The variation lies in the product itself, which is not separated into two functions as in the previous algorithm. In this version, the main function associated with an $\mathcal{R}$-class can only call one product function, which computes the product of an element and a generator and the class of that product. This function is factorized with the same algorithm as before, which might factorize more statements, as this time all the products in an $\mathcal{R}$-class are dealt with in the same function.

Example 2.27. Consider again the syntactic semigroup S of $(ac^*b+c)^*$, given in example 2.22. Once again, this algorithm regroups the products by $\mathcal{R}$-class and uses intermediary functions, one for each $\mathcal{R}$-class. The difference is that, this time, each $\mathcal{R}$-class is associated with only one product function. For example, the function dealing with the $\mathcal{R}$-class that contains 0 and 3 is the following:

Fonction ProcessClassOne(*word*, *element*, *index*), the product function that orders the $\mathcal{R}$-classes and uses intermediary functions for the transition semigroup of $(ac^*b+c)^*$ in the algorithm "assembled"

```
Input : word, element, index
Output : element × word[index]
for i ← index to word.len() − 1 do
    generator ← ElementFromLetter(word,i);
    (element,class) ← ProductFromROne(element,generator);
    if class = 3 then
        return ProcessClassThree(word,element,i + 1);
    end
end
return IsAccepted(element);
```

The function ProductFromROne is obtained by taking all possible products $s._S g$, where s is in the $\mathcal{R}$-class and g is a generator element, and factorizing them using the two rules defined above. Several orderings are possible, including the one given below.

Fonction ProductFromOne(*element*, *generator*), the factorized product function that computes the products from the $\mathcal{R}$-class of 0 and 3

```
Input : element, generator
Output : element × generator
if generator = 2 then
    return element;
end
if element = 0 AND generator = 0 then
    return 5;
end
if element = 3 AND generator = 0 then
    return 0;
end
if element = 0 then
    return 1;
end
if element = 3 then
    return 5;
end
```

In the benchmark results, this version is called "assembled", and the variant where the automaton is explicitly determinized and minimized before the construction of the semigroup is called "assembled-min".

Remark 2.7. All the algorithms presented in this section could be improved by detecting the semigroup elements from which no accepting element can be reached, and cutting the execution there. However, these versions are kept as they are in order to measure their efficiency on the totality of the inputs.

2.4.5 Benchmark results

The algorithms described in the previous sections have been tested on the same languages and words as the other benchmarks in this chapter (see section 2.2.4 for a detailed description). The results are shown in table 2.3.

The case of $\mathbf{L_{DA}}$. These results, though only preliminary, show how much room there is to improve pattern matching algorithms, and how complicated it is to optimize an algorithm for all cases. Indeed, the efficiency of the algorithms on the languages $abstar = (ac^*b + c)^*$ and $L_{\mathcal{R}} = (a+b+c)^*d(b+c+d)^*$ varies significantly, but all algorithms struggle to process the words in the language $L_{\mathbf{DA}} = (a+b+c)^*b(a+b+c)^*d(a+b+c+d)^*c(a+d)^*$. All of them manage to fall into the category of average runs, but barely, and the results do not vary significantly between algorithms. They fail to optimize the evaluation of this particular language, which shows that there is still a lot of work to be done. However, some interesting speedups can be found.

The effect of ordering the product. Although the "facto" algorithm, the "$\mathcal{R}$-stamp" algorithm and their minimized variants fail to optimize the evaluation of the inputs, the others can lead to runs that are above average. First, both the "$\mathcal{R}$-ord" algorithm and its minimized variant perform well on the words in $L_{\mathcal{R}}$. This was to be expected, as this language is $\mathcal{R}$-trivial: each $\mathcal{R}$-class contains exactly one element, which makes the product trivial as long as it stays in the same $\mathcal{R}$-class. Thus, the compiler manages to optimize it and skip most characters. This does not happen in the $\mathcal{R}$-stamp algorithm, even though it was planned to do so, since the function that computes the product takes the $\mathcal{R}$-class as parameter and it verifies its value at each new character. This unnecessary layer of conditional branching is the main difference between the two algorithms. It slows down the program, probably by confusing the compiler, which does not see the trivial products and cannot optimize the evaluation.

The effect of factorization: the "ord-facto" algorithm. The "ord-facto" algorithm and its minimized variant are based on this algorithm, and so they have similar performances in most cases. They tend to be faster, as the products inside each $\mathcal{R}$-class are factorized. However, "$\mathcal{R}$-ord" is twice as fast as "ord-facto" and on the word named *last* in the language $L_{\mathcal{R}}$. The use of perf stat shows that the number of branches and of mispredicted branches is roughly the same. However, the number of instructions per cycle is nearly multiplied by two for the execution of "$\mathcal{R}$-ord". This indicates that, during the execution of "$\mathcal{R}$-ord", the compiler executed much more instructions in parallel. It seems that, in this case, the factorization performed by "ord-facto" re-orders the instructions in a way that antagonizes the compiler, preventing it from seeing the optimizations it could perform. A similar phenomenon happens for the executions of "$\mathcal{R}$-ord" and "$\mathcal{R}$-ord-min" on the word $first$ in the language $L_{\mathcal{R}}$, although the results are less impressive: the minimization of the automaton changes the structure of the code and prevents the compiler from seeing some optimizations. On the contrary, this minimization can lead to a more efficient execution. This what happens with the execution of "ord-facto-min":

the program has a bit less branches than the one for "ord-facto", but this does not suffice to explain the gap in performance, as it executes much more executions in parallel (around 1.5 times more per cycle). In all these cases, the minimization in itself does not really influence the results, but it changes the structure of the code. This change of structure then impacts the compilation, leading to more optimizations in some cases, and less in others. More in-depth research is needed to understand the subtleties of this compilation process.

The "assembled" algorithm. Finally, the algorithm "assembled" and its mininimized variant perform much less efficiently than expected. The phenomena involved here are similar to the ones discussed above. The first one is linked to the structure of the algorithm: the idea was to have only one product function per $\mathcal{R}$-class, instead of separating the products depending on whether the result stayed in the same $\mathcal{R}$-class. This allows for more factorization, which may impact positively the results in some cases, but hinders the efficiency in others. This is the case of the words $first$ and $middle$ in the language $L_{\mathcal{R}}$. The optimization discussed previously for the algorithm "$\mathcal{R}$-ord" is not seen by the compiler, as it does not detect the fact that the product can only get out once of an $\mathcal{R}$-class. Thus, it does not skip characters like "$\mathcal{R}$-ord" does, and causes much more conditional branching, many of which are not correctly predicted. However, on the language $abstar = (ac^*b + c)^*$, an interesting phenomenon happens. If we compare the algorithms "assembled-min" and "ord-facto-min", we can note that the first one is more efficient on the words with a higher density of occurrences of a and b, and that it is the complete opposite for the second one. This due to the phenomenon discussed for "$\mathcal{R}$-ord" versus "ord-facto". Indeed, all the runs of both algorithms have the same number of instructions, branches and mispredicted branches, but the number of instructions executed in parallel is different: on the words $dense$ and $average$, "assembled-min" execute much more instructions per CPU cycle, and the opposite happens on the word $sparse$.

Lang.	Word	Algorithm (GB/s)					
		base	stamp	facto	facto-min	$\mathcal{R}$-stamp	$\mathcal{R}$-stamp-min
acstara	*last*	7.0	0.15	0.20	0.19	0.16	0.16
abstar	*dense*	0.64	0.33	0.43	0.47	0.52	0.36
	average	0.64	0.39	0.43	0.43	0.55	0.50
	sparse	0.64	0.13	0.43	0.43	0.51	0.47
$L_{\mathbf{DA}}$	*balanced*	0.64	0.038	0.14	0.15	0.13	0.12
	decrease	0.64	0.034	0.14	0.15	0.12	0.12
	increase	0.64	0.042	0.15	0.16	0.13	0.13
$L_{\mathcal{R}}$	*first*	0.64	0.20	0.24	0.27	0.20	0.19
	last	0.64	0.25	0.23	0.29	0.19	0.20
	middle	0.64	0.24	0.24	0.29	0.19	0.20

Lang.	Word	Algorithm (GB/s)					
		$\mathcal{R}$-ord	$\mathcal{R}$-ord-min	ord-facto	ord-facto-min	assembled	assembled-min
acstara	*last*	0.19	0.19	0.21	0.21	0.22	0.24
abstar	*dense*	0.54	0.73	0.81	1.3	0.81	0.80
	average	0.39	0.89	0.77	1.2	0.65	0.97
	sparse	0.72	0.54	0.64	0.78	0.77	1.3
$L_{\mathbf{DA}}$	*balanced*	0.14	0.16	0.18	0.17	0.17	0.18
	decrease	0.13	0.15	0.18	0.16	0.16	0.18
	increase	0.14	0.16	0.17	0.17	0.17	0.17
$L_{\mathcal{R}}$	*first*	1.3	0.97	1.7	1.3	0.25	0.22
	last	1.9	1.9	0.97	1.9	1.9	1.9
	middle	1.3	1.3	1.3	2.1	0.45	0.46

Table 2.3 – The results of the advanced semigroup-based algorithms - troll

Chapter 3

Vectorial circuits

Outline of the current chapter

In chapter 2, we presented a fragment of regular expressions and some algorithms to recognize the regular languages represented by these regexes. Among those algorithms, the one leveraging the algebraic theory of automata to obtain a parallel algorithm is by far the most efficient. In this chapter, we present another kind of parallelism called bit-level parallelism. Algorithms using this kind of parallelism use particular operations which process chunks of data of fixed size in parallel. Using bit-level parallelism, we can produce algorithms that are almost branch-free, which can greatly increase performance. Notably, Myers, when introducing the use of bit-level parallelism for approximate string matching [52], obtained significantly faster programs which had a great impact on stringology and DNA processing. From that point, numerous projects have been developed using bit-level parallelism, such as projects to improve parsing [45, 46], others to query JSON [37, 28, 27], or some projects that explicitly handle regular expressions. For example, Cameron et al. [14] used bit-level parallelism for regex matching and applied it to `grep`, rediscovering the possible uses of binary addition presented by Serre in [67]. Also of note is the project Hyperscan [76], which is an attempt at writing a regex engine relying on vectorization, and which has inspired the Rust `regex` crate. To study the complexity of those

bit-level algorithms, several models have been introduced [62, 9]. Among these different models, we rely on the most recent to our knowledge, which is the model of vectorial circuits. These vectorial circuits generalize boolean circuits by considering bit words instead of booleans. We focus on two distinct classes of vectorial circuits by limiting them to certain gates. Notably, in one of our classes, inspired from Serre's PTL-algorithms [67], we use the binary addition, which allows to transmit some information along the vectors. This operation's expressive power allows us to recognize languages in FO[<]. The other class that we focus on uses the prefix and suffix operations to recognize the languages in $\mathrm{FO}_2[<]$. As these circuits already process inputs by chunks, they can be used for streaming by adding a feedback loop. In this chapter, we present bit-level parallelism and its application to data validation. Then, we formalize bit-level parallel algorithms by defining vectorial circuits, and we present a method to convert vectorial circuits to adapt them to streaming.

3.1 Bit-level parallelism

The previous chapter explored some approaches that leverage the algebraic theory of automata to produce (hopefully) efficient algorithms which recognize regular languages. The goal was to produce a parallel algorithm taking advantage of the inherent parallelism of semigroups. While the results showed that this approach significantly outperforms the naive algorithms, alternative approaches also exist which can lead to even more impressive results.

One of these approaches consists in processing data chunks with operations that simultaneously manipulate all elements within a chunk. We call this form of parallelism *bit-level parallelism*, which, in practice, relies on processors equipped with the ability to efficiently handle long machine words. Among the numerous works on bit-level parallelism, there have been several propositions on integration of bit-level parallelism in the context of regular expressions processing. In this section, we present some of these propositions. However, in the rest of this thesis, we only focus on the most common approach, which consists in splitting the input words into chunks, then process each chunk sequentially using bit-level parallel operations.

3.1.1 The example of memchr

Before talking about the general case, we focus on one example taken from the standard C library, which is the function called `memchr`. This function takes a word w and a letter a, and returns the index of the first occurrence of a in w, if it exists. The underlying automaton of this function is showed in Figure 3.1. The most naive implementation for this function is a for loop over the letters of w which stops at the first letter a and returns the corresponding index.

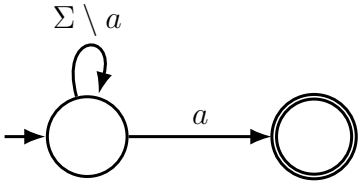


Figure 3.1 – The minimal DFA behind memchr

The program using that naive sequential algorithm is quite slow, much slower than the `memchr` function from the standard library, even when optimized by the compiler (on our machines, we obtain a factor of 4.5 between the two).

This speedup is due to the fact that the function from the standard library uses bit-level parallelism to avoid looking at every letter sequentially. More specifically, it separates the word w into chunks the size of a computer word and uses particular operations on each chunk to see if there is an a in that chunk. These operations are called bit-level parallel, as they process an entire chunk at once. If the function does not find an a, it continues to the next chunk. Otherwise, it goes sequentially through the chunk to find its exact position. Using bit-level parallel instructions on the chunks allows to obtain the information needed without looking separately at each letter, instead processing each chunk in parallel.

In order to know if there is an a in a chunk, the function compares it to another chunk of the same size containing only occurrences of a using the bitwise equality. This bit-level parallel operation is defined as follows: given two words $v = v_0 \cdots v_{n-1} \in \Sigma^*$ and $w = w_0 \cdots w_{n-1} \in \Sigma^*$, we define $v = w$ to be the binary word $b_0 \cdots b_{n-1} \in \{0, 1\}^*$ such that, for each index $i \in [n]$, $b_i = 1$ if and only if $v_i = w_i$. Using this operation, each chunk v of the input word is compared to a word $a \cdots a$ of the same size. We denote that specific operation by $\mathbb{1}_a$, which computes the indicator vector of a in its input vector (see definition 3.1 for the formal definition of an indicator vector). The binary word obtained as output contains a 1 if and only if v contains an occurrence of a, and the occurrences of 1 are at the same indices as the occurrences of a. Thus, the program checks if there is a 1 using a bit-level parallel operation. If there is no 1, it starts again on the next chunk. Otherwise, it uses the naive for loop to go through the chunk to find the exact index of the first occurrence of a.

Algorithm 5 : The code used by memchr to find the first occurrence of a letter

```
Input : word ∈ Σ*, letter ∈ Σ
index ← 0;
for chunk in word do
    indicator ← 𝟙_letter(chunk);
    if indicator ≠ 0 then
        for chunk_index ← 0 to chunk.len() − 1 do
            if indicator[chunk_index] = 1 then
                return index + chunk_index;
            end
        end
    end
    index ← index + chunk.len();
end
return −1;
```

The algorithm used by `memchr` uses bit-level parallel operations to process a word split in chunks. This kind of algorithm is said to rely on *streaming bit-level parallelism*, which is the most frequent application of bit-level parallelism.

3.1.2 Streaming bit-level parallelism

As presented above, streaming bit-level parallelism consists in splitting the input into chunks, then processing them with bit-level parallel operations, which are operations processing an entire chunk at once. The operations used in streaming bit-level parallelism operate at bit-level and often process the bits of the chunk separately. Therefore, we define a *chunk of data* to be

a binary word. In practice, the chunks of data can be more complex, but it is always possible to reduce the problem to binary words. Thus, the base operations on chunks of data are the *bitwise* logical operations, such as $\vee$, $\wedge$ and $\neg$, which process each bit independently from the others. Streaming bit-level parallelism was first introduced by Lamport [44], who used *bit words* to represent the data, and processed them using *vectorial operations*. The programs considered in this thesis are similar to Lamport's, as they also mostly deal with bit words and use vectorial operations. In this part, we introduce *indicator vectors*, which allow to obtain bit words from words on any alphabet, and we define the vectorial operations that are used in this thesis.

Indicator vectors. Lamport's paper considers bit words, but tools such as the `memchr` function deal with larger alphabets than $\{0, 1\}$. To do so, they compare the chunks to other chunks of the same size containing only occurrences of one given letter. We formalize this using the *vectorial indicator functions* $\mathbb{1}_a$, defined as follows:

Definition 3.1

Given an alphabet Σ and a letter $a \in \Sigma$, we define the *vectorial indicator function* $\mathbb{1}_a$ to be the function $\mathbb{1}_a : \Sigma^* \to \{0, 1\}^*$ such that, for any input word $w = w_0 \cdots w_{n-1} \in \Sigma^n$, $\mathbb{1}_a(w) = b_0 \cdots b_{n-1}$, with $b_i = 1$ if and only if $w_i = a$, for any index $i \in [n]$. We call the binary word $\mathbb{1}_a(w)$ the *indicator vector* of w.

We can extend the domain of that definition to sub-alphabets of Σ:

Definition 3.2

Given an alphabet Σ and a set $B \subseteq \Sigma$, we define the *vectorial indicator function* $\mathbb{1}_B$ to be the function $\mathbb{1}_B : \Sigma^* \to \{0, 1\}^*$ such that, for any input word $w = w_0 \cdots w_{n-1} \in \Sigma^n$, $\mathbb{1}_a(w) = b_0 \cdots b_{n-1}$, with $b_i = 1$ if and only if $w_i \in B$, for any index $i \in [n]$. We call the binary word $\mathbb{1}_B(w)$ the *indicator vector* of w.

This definition is useful for the proofs of this thesis, and is justified by the existing SIMD operations. Indeed, SIMD operations do not handle only the equality between bytes, but also the order ($>$). With the order, the equality, and the negation, we can compute the indicator vector of any interval of characters in the byte alphabet. By taking boolean combinations of these intervals, we can compute the indicator vector of any sub-alphabet of the byte alphabet, and thus of any alphabet it represents.

Example 3.1. Consider the word *babcaadaca*. The indicator vectors of the letters a and c are respectively $\mathbb{1}_a = 0100110101$ and $\mathbb{1}_c = 0001000010$. From these vectors, we can build the indicator vector of the set $B = \{a, c\}$ by computing the bitwise OR of the two vectors.
$\mathbb{1}_B = \mathbb{1}_a \vee \mathbb{1}_c = 0101110111$

Vectorial operations. The following vectorial operations are used by Lamport in his paper. Some of them constitute the core of the set of operations used in this thesis.

- The unary negation $\neg = (\neg_n)_{n \in \mathbb{N}^+}$ is such that, for each $n \in \mathbb{N}^+$ and any binary word $v = v_0 \cdots v_{n-1}$ of size n, the binary word $\neg_n v = w_0 \cdots w_{n-1}$ is such that, for any index $i \in [n]$, $w_i = \neg v_i = 1 - v_i$.

- The binary disjunction $\vee = (\vee_n)_{n\in\mathbb{N}^+}$ is such that, for each $n \in \mathbb{N}^+$ and any two binary words $v = v_0 \cdots v_{n-1}$ and $u = u_0 \cdots u_{n-1}$ of size n, the binary word $v \vee_n u = w_0 \cdots w_{n-1}$ is such that, for any index $i \in [n]$, $w_i = v_i \vee u_i = \max(0, \min(v_i + u_i, 1))$.

- The binary conjunction $\wedge = (\wedge_n)_{n\in\mathbb{N}^+}$ is such that, for each $n \in \mathbb{N}^+$ and any two binary words $v = v_0 \cdots v_{n-1}$ and $u = u_0 \cdots u_{n-1}$ of size n, the binary word $v \wedge_n u = w_0 \cdots w_{n-1}$ is such that, for any index $i \in [n]$, $w_i = v_i \wedge u_i = \max(0, v_i + u_i - 1)$.

- The binary addition $+ = (+_n)_{n\in\mathbb{N}^+}$ is such that, for each $n \in \mathbb{N}^+$ and two binary words $v = v_0 \cdots v_{n-1}$ and $u = u_0 \cdots u_{n-1}$ of size n, the binary word $u +_n v = w_0 \cdots w_{n-1}$ is such that $w_0 = (u_0 \wedge \neg v_0) \vee (\neg u_0 \wedge v_0)$ and, for any index $i \in [n - l]$,

$$w_i = ((u_i \oplus v_i) \wedge \neg\mathrm{carry}_i) \vee (((u_i \wedge v_i) \vee (\neg u_i \wedge \neg v_i)) \wedge \mathrm{carry}_i)$$

  where $a \oplus b = (a \wedge \neg b) \vee (\neg a \wedge b)$, and the binary word $\mathrm{carry} = \mathrm{carry}_0 \cdots \mathrm{carry}_{n-1}$ is such that $\mathrm{carry}_0 = 0$ and, for any index i such that $1 \leq i < n$,

$$\mathrm{carry}_i = \bigvee_{j=0}^{i-1} (u_j \wedge v_j \wedge \bigwedge_{k=j+1}^{i-1} (u_k \vee b_k))$$

- The l-bit left shift $\mathrm{lshift}_l = (\mathrm{lshift}_{l,n})_{n\in\mathbb{N}^+}$ is such that, for each $n \in \mathbb{N}^+$, for any integer $l < n$ and any binary word $v = v_0 \cdots v_{n-1}$ of size n, the binary word $\mathrm{lshift}_{l,n}(v) = w_0 \cdots w_{n-1}$ is such that, for any index $i \in [n-l]$, $w_i = v_{i+l}$ and, for any index i such that $n-l \leq i < n$, $w_i = 0$. If $l \geq n$, we define $\mathrm{lshift}_{l,n}(v)$ to be the binary word $0 \cdots 0$.

- The l-bit right shift $\mathrm{rshift}_l = (\mathrm{rshift}_{l,n})_{n\in\mathbb{N}^+}$ is such that, for each $n \in \mathbb{N}^+$, for any integer $l < n$ and any binary word $v = v_0 \cdots v_{n-1}$ of size n, the binary word $\mathrm{rshift}_{l,n}(v) = w_0 \cdots w_{n-1}$ is such that, for any index i such that $l \leq i < n$, $w_i = v_{i-l}$ and, for any index $i \in [l]$, $w_i = 0$. If $l \geq n$, we define $\mathrm{rshift}_{l,n}(v)$ to be the binary word $0 \cdots 0$.

Example 3.2. If we take two binary numbers with 5 bits, the bitwise operation $\wedge$ processes the 5 indices separately and returns a binary number with 5 bits, where each of the bits is the result of the conjunction of the two corresponding bits in the inputs. The table 3.1 gives an example by showing the application of the bitwise $\wedge$ to $n_1 = 01111_2 = 15$ and $n_2 = 11001_2 = 25$.

n_1	0	1	1	1	1
n_2	1	1	0	0	1
$n_1 \wedge n_2$	0	1	0	0	1

Table 3.1 – Performing the bitwise $\wedge$ of two binary integers

The table 3.2 shows the addition of the binary words u and v, along with the carry that is produced during the computation.

Several researchers have independently rediscovered the interest of the binary addition as an operation over machine words for data processing. Myers, in his seminal paper [52], illustrated the potential applications of carry-propagation, and used it to speed up approximate regular expression matching. These applications were later rediscovered by Cameron et al. [14], who efficiently compiled regular expression of the shape A^*, with A being a subset of the alphabet.

u	1	1	0	1	1	1	1
v	0	1	0	1	0	1	1
$u+v$	1	0	1	0	0	1	1
carry	0	1	0	1	1	1	1

Table 3.2 – Example of addition

The vectorial operators presented above are now common in the context of bit-level parallelism [14, 45, 76, 40]. Our vectorial algorithms will also use most of them, as they are inspired from bit-level parallelism. More specifically, our vectorial circuits (defined in definition 3.3) use these operations to label their gates.

In his paper, Lamport gives a general method to produce a program using bit-level parallelism for any binary operator returning a boolean (for example, an equality test). This method has some common traits with our work, so we give some details about it. Lamport's method, applied to a relation p, produces a masking function $m_p : (\{0,1\}^k)^2 \to \{0,1\}^k$, where k is the chosen size of the computer words. The words are separated into small chunks of n bits, each of those corresponding to a letter in the input text. Thus, the function m_p is defined by the following: let $v = v_0 \cdots v_{k-1}$ and $w = w_0 \cdots w_{k-1}$ be two input bit masks[1], and let $x = m_p(v,w) = x_0 \cdots x_{k-1}$. For each index i which is a multiple of n, $x_i \cdots x_{i+n-1} = 1^n$ if and only if $p(v_i \cdots v_{i+n-1}, w_i \cdots w_{i+n-1}) = true$. Otherwise, $x_i \cdots x_{i+n-1} = 0^n$. This definition can notably serve for an acceptance condition as defined later in definition 3.4. More generally, such bit masks can be used to select some parts of the input, either for pattern matching or for text processing.

Example 3.3. We can write a simple program for the equality test $=$ and produce a method $m_=$ as defined by Lamport. To do so, we note that two bit-words a and b of the same size n are equal if and only if $a \oplus b = 0^n$. Thus, we can define $m_=$ to be such that, for any input binary words v and w, $m_=(v,w) = \neg(v \oplus w)$ (the negation was mentioned by Lamport as a useful but not necessary operation).

The masking functions m_p can be applied to any computer word. Thus, the idea is to separate the input text into as many computer words as necessary to obtain a sequence of words, then to apply the masking functions to obtain a sequence of masks as output.

This separation of the input text into words manageable with bit-level parallelism is the main use that is made of this kind of parallelism, and it is the one that we focus on in the rest of this thesis.

3.1.3 The shift-or algorithm

In this thesis, we focus on streaming bit-level parallelism, but other applications of bit-level parallelism exist. Notably, one of the early applications of bit-level parallelism is a pattern matching algorithm called the *shift-or* algorithm, introduced in [5]. This algorithm considers regexes using only literal characters, the symbol representing the alphabet (`.` in our case), characters classes, and complements of character classes. This is necessary for this approach, as the state of the search is represented by a finite set of bits of fixed size, and thus the regex

[1] Lamport's definition included implementation details that are not considered here

must match only words of a given size. The shift-or algorithm assembles those bits into a word and modifies them using instructions that are typical of bit-level parallelism.

More specifically, for a regex reg that matches only words of size n, the shift-or algorithm uses a word of size n to represent the state of the search. We denote this word as $q = q_0 \cdots q_{n-1}$. The search goes sequentially through the input word, without backtracking or skipping letters, so each letter is considered exactly once. At step i of the algorithm, the i^{th} letter of the input text is considered, and the state q is updated. That state is to be interpreted as follows: each bit q_j of q is equal to 0 if and only if the $n - j$ last seen letters of the input match the $n - j$ first characters or classes of the regex. Thus, $q_0 = 0$ if and only if the last n seen characters of the input match the regex.

Example 3.4. Consider the regex `baba` and the text *abacbabbabac*. The regex recognizes only infixes of size 4, so the state of the search is represented by 4 bits, in the word $q = q_0q_1q_2q_3$. After reading the fourth letter of the text, the last 4 letters are *abac*, so the state q must be equal to 1111, as the last seen letter is not equal to any letter of the pattern. After reading the seventh letter, the last 4 letters are *cbab*. The last 3 letters of this infix match the first 3 letters of the pattern, and the last letter matches the first letter of the pattern, so the state q must be equal to 1010. However, when reading the eighth letter, there is a mismatch, as the fourth letter of the pattern is an a, and the letter in the text is a b, so we do not obtain a state beginning with 0. Instead, the state q must now be equal to 1110. The next three letters match the rest of the pattern, and when reading them we must successively obtain the states 1101, 1010, and 0101, which corresponds to a match since $q_0 = 0$.

In order to update the state q, the algorithm relies on a table T which associates each letter of the alphabet to a bit word of size n, which will allow to update the entire state at once. Formally, for any letter a in the alphabet, $T[a] = t_0 \cdots t_{n-1}$ where, for each index $i < n$, $t_i = 0$ if and only if the letter a matches the $(n-i)^{\text{th}}$ character or class of the regex. Using this table, we can update the state q as follows. At each step, if the new letter is a, the state q is shifted to the left by 1 to represent the progress in the input, then the bitwise OR of that shifted word and $T[a]$. In symbols, if we denote by q' the new state:

$$q' = \text{lshift}_1(q) \vee T[a]$$

Example 3.5. If we consider the same regex as in the previous example, `baba`, and the alphabet $\Sigma = \{a, b, c\}$, we obtain the table T such that $T[a] = 0101$, $T[b] = 1010$, and $T[c] = 1111$. Now, consider again the text *abacbabbabac*. After reading the fourth letter, the state q is equal to $\text{lshift}_3(T[a]) \vee \text{lshift}_2(T[b]) \vee \text{lshift}_1(T[a]) \vee T[c] = 1111$. Then, for each new letter, the state changes as follows:

- At the index 4, $q = \text{lshift}_1(q) \vee T[b] = 1110$.
- At the index 5, $q = \text{lshift}_1(q) \vee T[a] = 1101$.
- At the index 6, $q = \text{lshift}_1(q) \vee T[b] = 1010$.
- At the index 7, $q = \text{lshift}_1(q) \vee T[b] = 1110$.
- At the index 8, $q = \text{lshift}_1(q) \vee T[a] = 1101$.
- At the index 9, $q = \text{lshift}_1(q) \vee T[b] = 1010$.
- At the index 10, $q = \text{lshift}_1(q) \vee T[a] = 0101$, which indicates a match.

- At the index 11, $q = \text{lshift}_1(q) \vee T[c] = 1111$.

This algorithm can be compared to Thompson's NFA-based algorithm, as it runs an NFA on the text without backtracking by considering a state made of several bits, which resembles the powerset used for determinization.

3.2 Validating regexes over chunks of letters

Now that we defined bit-level parallelism, we can see how to use these algorithms for data validation and how well they can perform. This section consists in a benchmark on some hand-written bit-level parallel algorithms and a formalization which allows us to study bit-level parallel algorithms recognizing regular languages by comparing them to a particular kind of automata whose transitions are labeled by words of fixed size.

3.2.1 Formalization

Bit-level parallelism allows to build efficient programs for data validation. We propose a rather natural formalization of the programs based on bit-level parallel operations. This formalization comes from the fact that the kind of bit-level parallelism introduced by Lamport processes an input word sequentially, similarly to an automaton, but by chunks of fixed size. Thus, when we apply bit-level parallelism to data validation, we obtain algorithms that process the input by chunks and return a boolean indicating whether the input is accepted. This can be represented by a particular kind of automaton, which considers a chunk of letters at a time instead of only one letter. Now, we show how to represent a bit-level parallel algorithm with an automaton $A = (\Sigma^n, Q, I, \delta, F)$, where Σ is the alphabet of the inputs and n is the size of the chunks.

Computation of the automaton. The algorithms using streaming bit-level parallel operations for data validation need some memory to propagate information from one chunk to another. For example, if we want to accept any word that does not have more than p occurrences of the letter a, we need to remember how many occurrences (up to p) have been seen in the previous chunks. In this thesis, we restrict ourselves to *constant space bit-level parallel algorithms*. Because of the constant space these algorithms require, they can be considered as automata, whose states are the possible states of the memory to propagate between chunks, and whose letters are possible values of the chunks.

Formally, given a constant space bit-level parallel algorithm that operates on chunks of size k and that uses m bits of memory $b_0, \ldots, b_{m-1}$, the automaton $A = (\Sigma^k, Q, I, \delta, F)$ representing that algorithm is constructed as follows:

- $Q = \{q_i \mid i \in [2^m]\}$, where each state q_i corresponds to the state of memory in which the bit word $b_0 \cdots b_{m-1}$ is the binary representation of the integer i.
- The set I contains only one initial state q_i, which is the state associated with the initial state of the memory.
- The set F contains all the states q_i such that, if the algorithm has the memory associated with q_i when it finishes processing the input, then it accepts it.
- $\delta : Q \times (\Sigma^k) \to Q$ is such that for any state q_i and any chunk $a_0 \cdots a_{k-1}$, $\delta(q_i, a_0 \cdots a_{k-1})$ is the state q_j associated with the memory $b'_0, \ldots, b'_{m-1}$ that the algorithm obtains if it processes the chunk $a_0 \cdots a_{k-1}$ with the memory $b_0, \ldots, b_{m-1}$ associated with q_i.

Note that the size of the input words may not always be a multiple of k. Consequently, we need a way to fill the last chunk to obtain the right size. This can be done using a padding symbol that does not belong to the alphabet Σ. In this thesis, we ignore this part of the problem and we consider that the size of the words is always a multiple of k.

The automaton's size. The automata defined above have a number of states which is exponential in the space used by the bit-parallel algorithm. However, it is difficult to give a relation between a language and the memory necessary for a bit-level parallel algorithm recognizing that language. Moreover, the number of transitions in δ depends on the exact operations used in the bit-level parallel algorithm. So, in order to study the complexity of these automata, we consider the automata on the alphabet of chunks of size k that can be constructed from the automata that recognize given regular languages.

Lemma 3.1

Given a DFA $A = (\Sigma, Q, I, \delta, F)$ and a integer k, we can build an equivalent DFA $A_k = (\Sigma^k, Q_k, I_k, \delta_k, F_k)$ that processes k letters at once. Moreover, the set $\{(q_1, w, q_2) \in Q_k \times \Sigma^k \times Q_k \mid \delta(q_1, w) = q_2\}$ is of size $O(|Q| \cdot |\Sigma|^k)$, a bound that is tight.

Proof. Since A_k processes k letters at once, we replace paths of length k in A by transitions in A_k. To do this, we can use the same set of states, so we set $Q_k = Q$, $I_k = I$ and $F_k = F$. Now, we only have to build the transition function $\delta_k : Q_k \times (\Sigma^k) \to Q_k$. To do so, we replace each path of length k in A by a transition labeled by the word formed when concatenating the labels of the path. Formally, for any state $q \in Q$ and any word $w = w_0 \cdots w_{k-1} \in \Sigma^k$, if there exist k states $q_1, \ldots, q_k \in Q$ such that $(q, w_0, q_1) \cdots (q_{k-1}, w_{k-1}, q_k)$ is a path in A, then we set $\delta_k(q, w) = q_k$. Otherwise, $\delta(q, w)$ is not defined.

A_k is equivalent to A for all inputs whose size is a multiple of k. Indeed, given a word $w = w_0 \cdots w_{lk-1} \in \Sigma^{lk}$, there is path $(I_k, w_0 \cdots w_{k-1}, q_1) \cdots (q_{l-1}, w_{l(m-1)} \cdots w_{lk-1}, q_l)$ in A_k such that $\delta(I, w) = q_l$ in A. Thus, w is accepted by A_k if and only if it is accepted by A.

The number of transitions defined by δ is the number of paths of length k in A formed by the transitions, as one transition in A_k is constructed from a path of length k in A. The size of the set $\{(q_1, w, q_2) \in Q_k \times \Sigma^k \times Q_k \mid \delta(q_1, w) = q_2\}$ then follows from the fact that there are at most $|Q| \cdot |\Sigma|^k$ paths of length k in the DFA A. That bound is tight, and it is reached even in simple examples. For instance, consider the alphabet $\Sigma = \{a, b, c\}$ and the automaton A with only one state q and the following transitions: (q, a, q), (q, b, q), (q, c, q). The automata A_k obtained from this DFA all have only one state, and exactly 3^k distinct transitions, labeled by all the words of size k formed with the alphabet Σ. □

As shown in the proof, the number of states of A_k and A is the same. However, the number of transitions can be significantly greater, as A_k has as many transitions as there are distinct paths of length k in A, which gives a bound of $O(|Q| \cdot |\Sigma|^k)$.

It is possible to implement directly such automata by using a table to represent the transitions. However, with the exponential number of transitions we can obtain when constructing these automata, the table can be too large for the program to be efficient. In these cases, we need another way of implementing the automaton. One of the possible solutions consists in using bit-level parallel operations to compute the transitions without having all of them in memory. In the next subsection, we give a few examples of programs that apply this solution, along with their performance.

Example 3.6. Take the automaton for $(a+b)^*a(a+b)^*$. Formally the automaton A_k obtained by the construction above will have two states q_0 and q_1. From the state q_0, there is one transition $\delta_k(q_0, b^k) = q_0$ and $2^k - 1$ transitions $\delta_k(q_0, u) = q_1$ for all the other words u (which all contain an a). From the state q_1, there are 2^k transitions (any chunk of size k) looping over q_1.

Note that storing all these transitions separately is not very efficient, since most of them have the same start and end states. Indeed, it is possible to describe concisely these transitions, by remarking that the transition from q_0 to q_1 should be labeled by *every word of* $(a+b)^k$ *except* b^k, and the loop over q_1 should be labeled by $(a+b)^k$. This description is of size constant with respect to k, contrarily to the description of δ_k in which we separate the effects of all the words. However, to use this concise description of the automaton, we need a more general model which can label transitions by sets of letters instead of individual ones.

3.2.2 Benchmark

In section 2.3, we gave a benchmark of some algorithms used as a baseline for the algorithms processing the regular expressions at compile time. In section 2.4, we gave a benchmark of some algorithms whose goal was to leverage the algebraic theory of semigroups to try to produce algorithms more efficient than the baseline, ultimately by producing a parallel algorithm. In section 3.1, we presented bit-level parallelism, which is a different kind of parallelism. Now, we give a benchmark for a few handwritten algorithms based on bit-level parallelism, in order to show examples of programs that implement the automata on chunks without using a table to represent the transitions. Our goal being to leverage the algebraic theory of automata to produce efficient vectorized algorithms based on bit-level parallel operations, this also serves as a baseline for handwritten programs using bit-level parallel operations.

Methodology. The implementation produced for the benchmark is written in Rust and can be found in the branch `with_bench_simd` of [59]. Contrary to the benchmarks of chapter 2, we focused on a few languages instead of trying to deal with the general case. For each one of these languages, a handwritten function has been added to the framework. These functions take a word in the form of a vector of bytes, split it in chunks, and process each chunk sequentially using SIMD instructions. Each time a chunk is processed, the functions retrieve the information previously stored, and once the chunk is processed, they store the information they need to propagate to the next one.

For each chunk, the indicator vectors (defined in definition 3.1) of some letters are computed using SIMD instructions, the exact letters depending on the language. These indicator vectors are then processed to obtain the information needed to determine if the input belongs to the language.

The languages and the corresponding algorithms. We considered the same languages as the ones presented in section 2.2.4 in order to compare the bit-parallel algorithms to the sequential ones.

$\Sigma^*\mathbf{a}\mathbf{c}^*\mathbf{a}\Sigma^*$ The language $acstata = \Sigma^*ac^*a\Sigma^*$, on the alphabet of bytes, includes all the words that contain two occurrences of a separated only by occurrences of c. This language can be recognized using the bit trick defined in the addition lemma (see Lemma 4.7): we can detect any occurrence of a that is at the end of an infix of the form ac^*a using the binary addition.

The only information stored in this algorithm is the carry of that addition, which is initially equal to 0.

When processing a chunk, the algorithm computes the indicator vectors of a and c, then performs the following binary addition:

$$(\mathbf{add\_result}, \text{carry}) = \mathbb{1}_a + (\mathbb{1}_a \vee \mathbb{1}_c) + \text{carry}$$

Then, a new vector is computed, which keeps only the bits set to 1 that indicate occurrences of a:

$$\mathbf{result} = \mathbf{add\_result} \wedge \mathbb{1}_a$$

Each bit set to 1 in the vector **result** indicates an occurrence of a which is at the end of an infix of the form ac^*a, as shown in example 3.7.

Example 3.7. Consider the word *abbaccbcabcaacaa*, and suppose that the chunk size is 4. Then, the computation proceeds as shown in the table below.

carry	0	1	0	1
$\mathbb{1}_c$	0000	1101	0010	0100
$\mathbb{1}_a$	1001	0000	1001	1011
$\mathbb{1}_a \vee \mathbb{1}_c$	1001	1101	1011	1111
$\mathbb{1}_a + (\mathbb{1}_a \vee \mathbb{1}_c) + \text{carry}$	0100	0011	0110	1011
$(\mathbb{1}_a + (\mathbb{1}_a \vee \mathbb{1}_c) + \text{carry}) \wedge \mathbb{1}_a$	0000	0000	0000	1011

When processing the last chunk, we obtain a vector that is different from **0** since there are infixes aa and aca whose second occurrence of a is in that chunk. Thus, the word is in the language.

$(\mathbf{ac^*b + c})^*$ The language $abstar = (ac^*b + c)^*$, on the alphabet of bytes, includes all the words on the alphabet $\{a, b, c\}$ that alternate occurrences of a and b, beginning with an a and ending with a b. Those words can contain occurrences of c anywhere. This language can be recognized using the bit trick defined in the addition lemma (see Lemma 4.7), similarly as for $\Sigma^*ac^*a\Sigma^*$: we can detect any occurrence of a that is at the end of an infix of the form ac^*a using the binary addition, as well as any occurrence of b that is at the end of an infix of the form bc^*b. Any word that contains such infixes cannot be in the language. All that remains is to verify that the first letter other than a c is an a and that the last letter other than a c is a b. All this can be done with a few instructions. As for the language $\Sigma^*ac^*a\Sigma^*$, the vector **add_result** and the corresponding carry are computed as follows:

$$(\mathbf{add\_result}, \text{carry}) = \mathbb{1}_a + (\mathbb{1}_a \vee \mathbb{1}_c) + \text{carry}$$

The vector **add_result** is used to detect any factor of the form ac^*a or bc^*b. For infixes of the form ac^*a, the following vector is computed:

$$\mathbf{result\_a} = \mathbf{add\_result} \wedge \mathbb{1}_a$$

This vector marks exactly each occurrence of a which is at the end of a factor ac^*a, so it must be equal to **0** for the word to belong to the language. Then, infixes of the form ac^*b are dealt with by computing the following vector:

$$\mathbf{result\_b} = \mathbf{add\_result} \wedge \mathbb{1}_b$$

This vector marks each occurrence of b that is at the end of an infix of the form ac^*b. It must mark all the occurrences of b for the word to belong to the language, so it suffices to verify that it is equal to $\mathbb{1}_b$. Note that it also verifies that the first b is after the first a. Finally, in order to verify that the last letter other than c is a b, it suffices to check, at the end of the computation, that the carry from the binary addition is null, which means that it has been absorbed by an occurrence of b. The only information stored in this algorithm is the carry of that addition, which is initially equal to 0. An example of computation is shown in example 3.8.

Example 3.8. Consider the word *abcaccbcacccccbb*, and suppose that the chunk size is 4. Then, the computation proceeds as shown in the table below.

carry	0	1	0	1
$\mathbb{1}_b$	0100	0010	0000	0011
$\mathbb{1}_a$	1001	0000	1000	0000
$\mathbb{1}_a \vee \mathbb{1}_c$	1011	1101	1111	1100
add_result	0110	0011	0000	0010
result_a	0000	0000	0000	0000
result_b	0100	0010	0000	0010

For each chunk, the vector **result_a** is null, so there are no infixes of the form ac^*a. However, when processing the last chunk, we obtain a vector **result_b** that is different from $\mathbb{1}_b$ since there is an infix bb in that chunk (and so, the second b of that infix is not marked by **result_b**). Thus, the word is not in the language.

$(\mathbf{a+b+c})^*\mathbf{d}(\mathbf{b+c+d})^*$ The language $L_{\mathcal{R}} = (a+b+c)^*d(b+c+d)^*$, on the alphabet of bytes, includes all the words on the alphabet $\{a, b, c, d\}$ such that the last occurrence of a, if it exists, is before the first occurrence of d. This language can be recognized by finding the first occurrence of d and verifying that there is no occurrence of a after it. To do so, the algorithm proceeds in two loops: the first one advances in the input until it finds an occurrence of d. The second one verifies that there is no letter a after the chunk containing that occurrence of d. Between the two loops, the algorithm verifies that there is no occurrence of a after the first d in the same chunk as that d.

$(\mathbf{a+b+c})^*\mathbf{b}(\mathbf{a+b+c})^*\mathbf{d}(\mathbf{a+b+c+d})^*\mathbf{c}(\mathbf{a+d})^*$ The language $L_{\mathbf{DA}} = (a+b+c)^*b(a+b+c)^*d(a+b+c+d)^*c(a+d)^*$, on the alphabet of bytes, includes all the words on the alphabet which have at least one occurrence of b, c and d such that the first occurrence of d is after the first occurrence of b and before the last occurrence of c, which itself is after the last occurrence of b. To recognize this language, the algorithm proceeds in two loops: the first one advances in the input until it finds an occurrence of d, and verifies that there is at least one occurrence of b before it by storing a vector which accumulates the information of the relative positions of the occurrences of b. The second loop verifies that there is no letter b after the last occurrence of c, and that this c is after the first occurrence of d. To do so, it stores a vector **acc** and uses it to propagate the necessary information. Thus, the vector **acc** is equal to $10\cdots0$ if before the current chunk there was a c, and there has been no occurrence of b since that c. Otherwise, **acc** is null.

The results have been obtained using grid5000, and more precisely the machine chiclet-4. At the time when this thesis is written, the machines chiclet each have two CPUs AMD EPYC 7301, on the architecture x86_64. Each of these CPUs has 16 cores. All the results have been measured as a mean of twenty runs of the same command.

Results. The results are given in table 3.3. As for the results of section 2.3.2, the performance of the baseline is very stable, except for the word in $\Sigma^* ac^* a \Sigma^*$, on which the baseline is about ten times faster than on the other languages. As explained before, this is due to the properties of the language, which allow the baseline algorithm to skip most characters using simple bit-level parallel instructions.

Even on the language $\Sigma^* ac^* a \Sigma^*$, the baseline is slower than the bit-level parallel algorithm. This indicates that there is room for improvement in the general tools such as this baseline, which could be improved using more bit-level parallel instructions. It is particularly clear for the languages other than $\Sigma^* ac^* a \Sigma^*$, on which the handmade bit-level parallel algorithm is more than ten times faster than the baseline, which considers each character sequentially. However, the speed-up provided by the bit-level parallel algorithms is in part due to the fact that they are designed for one language only, which allows them to be as good as possible. Consequently, these results only provide some kind of upper bound on the potential speed-ups that could be obtained by using bit-level parallelism to recognize a more general set of languages.

Language	Word	Algorithm	
		base (GB/s)	SIMD (GB/s)
abstar	*dense*	0.45	9.5
	average	0.45	9.5
	sparse	0.45	7.7
acstara	*last*	5.5	14
$L_{\mathbf{DA}}$	*balanced*	0.45	15
	decrease	0.45	15
	increase	0.45	16
$L_{\mathcal{R}}$	*first*	0.45	13
	last	0.45	14
	middle	0.45	14

Table 3.3 – The results of the SIMD algorithms compared to the baseline

Generalization. The programs presented in this section are handwritten and optimized for the corresponding languages. When seeing such programs, it is difficult to generalize them and obtain a general method to produce a program using bit-level parallel operations to recognize any given regular language. This has been studied in particular cases, such as the projects Hyperscan [76] and Parabix [46]. However, to the best of our knowledge, none of the existing projects using vectorized instructions in a general context uses it for the kind of optimization presented in section 3.2.2, which require a deep knowledge of the underlying structure of the language represented by the regex. Auto-vectorization, the most common form of vectorization in the tools currently available, focuses on optimizing simple loops, notably regular nested loop kernels operating on arrays. The project Hyperscan decomposes the regexes and uses a variant the shift-or algorithm presented in section 3.1.3, representing the state of an NFA search by a bit-vector that is updated using table lookups and SIMD instructions instead of conditional statements. Parabix is more similar to what is presented in this thesis: it processes the input by batches, and each character of the input is associated with a character class. The program computes the indicator vectors of these classes using low-level vectorial instructions. These indicator vectors are then processed with low-level vectorial instructions. However, in that first article, Parabix aims only at parsing XML documents, which means that it only handles

string matching. In a second paper [14], a primitive similar to the addition lemma presented in Lemma 4.7 is introduced. This primitive uses the binary addition, which thanks to Serre's work [67], is known to be expressive enough to recognize any first-order language, but the primitive is only used to speed up the processing of sub-regexes of the form B^*, where B is a subset of the alphabet.

In order to try and introduce a generalization of vectorized algorithms benefitting from the optimizations enabled by the structure of the language, we need to formalize these programs relying on bit-level parallelism. To do so, we chose to use vectorial circuits, which manipulate vectors. A vector can represent a chunk but, as we saw in the programs used in the benchmark, the memory used is composed only on the carries of the additions and the bits shifted out of the chunk and carried to the next one. This memory is meant to compute the bit-level parallel operations as if they operated on the entire input word. Thus, in the next section, we represent the input word by a single vector.

3.3 Classes of vectorial circuits

In the previous section, we presented a formalization of bit-level parallel algorithms using automata. Although this formalization gives us an idea of the complexity of the algorithms, it is not well adapted to represent efficient implementations. Indeed, we saw that implementing the automata using tables to represent the transitions could lead to tables of exponential size and thus to inefficient programs. Thus, we need another, more detailed formalization which represents bit-level parallel operations. In section 1.4, we presented boolean circuits, which are good candidates since they allow us to represent the parallelism inherent to a given algorithm. However, bit-level parallel algorithms process words by chunks, which are treated as atomic entities, and thus we want our circuits to manipulate entire words instead of splitting them into separate bits. Therefore, we consider *vectorial circuits*, which are a generalization of boolean circuits on bit words instead of only booleans. These circuits are designed to be independent of the size of the bit words, so that each vectorial circuit represents a family of circuits that process bit words of a given size. This way, we can consider only uniform families of circuits, which better represent the practical bit-level parallel algorithms that we want to study. In this section, we define vectorial circuits and the operations that we consider for labeling the gates of these circuits. With these operations, we define two families of vectorial circuits that we will focus on in the rest of this thesis, and we give a link between them and the usual complexity of boolean circuits. The first class of vectorial circuits, meant to recognize the languages in FO[<], has been studied in previous works, notably by Olivier Serre [67], while the second one, to the best of our knowledge, has not been formalized before.

3.3.1 Definitions

Our goal is to write vectorized algorithms that can take advantage of bit-level parallel instructions. Thus, we would like our circuits to manipulate a mathematical equivalent to vectors of booleans instead of just booleans. Informally, a *vectorial circuit* is a circuit whose gates are associated with functions that manipulate sequences of booleans.

A *boolean vector* is a word on the alphabet $\{0, 1\}$. We denote the names of the vector in bold face to distinguish the word x from the vector $\mathbf{x}$. The *dimension* of a vector $\mathbf{x}$ is its length $|\mathbf{x}|$. We let $\mathbf{1}_n$ and $\mathbf{0}_n$ respectively denote the sequence of n 1's and the sequence of n 0's. When n is irrelevant or obvious for the context, we write $\mathbf{1}$ and $\mathbf{0}$.

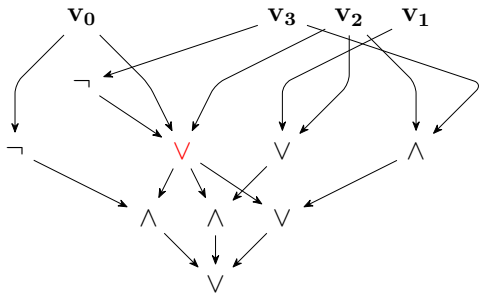
A *vectorial circuit* is a directed acyclic graph. The difference with boolean circuits is that the input gates bear boolean vectors instead of booleans, and the functions associated with the gates operate on boolean vectors of some dimension which is the same for all the vectors in a given circuit. The vocabulary for vectorial circuits is the same as for boolean circuits: the nodes are called gates, the source nodes are called input gates, the sink nodes are called output gates, the edges are called wires.

In order to recognize languages, we consider functions that manipulate vectors of any dimension. Therefore, vectorial circuits can be seen as circuit templates that, for each n, instantiate a *concrete circuit* working on vectors of dimension n. Once the dimension is fixed to n, associating vectors of dimension n to the input gates and flowing the values through the gates (where the right function operating on vectors of dimension is used) yields output values in the output gates.

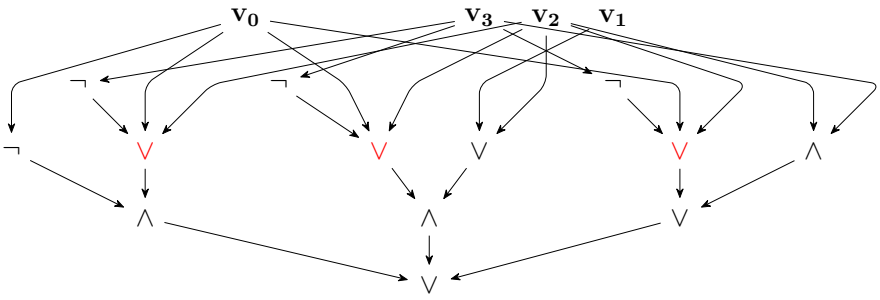
To precisely define vectorial circuits, we need to define the notion of sharing, which is essential to our results.

Sharing. In our circuits, all gates can have an arbitrary fan-out, independently of the associated function. This allows us to share the output value of a gate between several parts of a circuit without having to compute it several times. This model is closer to the actual programs that can be built from our circuits, and gives a more accurate description of their complexity.

Example 3.9. The vectorial circuit in Figure 3.2a uses three times the result of the gate in red. If we want to draw the same circuit without sharing any variable, we need to duplicate the sub-circuit that computes the result of that gate, to obtain the same result in three different gates. This gives the circuit in Figure 3.2b, where the gates in red all have the same result as the red gate in Figure 3.2a.



(a) A simple vectorial circuit



(b) The same vectorial circuit without sharing

Figure 3.2 – A vectorial circuit with and without sharing

Vectorial circuits. Now, we can define vectorial circuits formally.

Definition 3.3

A *vectorial circuit* is a directed acyclic graph whose gates are labelled by sequences of functions $(f_n)_{n\in\mathbb{N}^+}$ such that, for each $n \in \mathbb{N}^+$, f_n is of the form $f_n : (\{0,1\}^n)^r \to \{0,1\}^n$, where $r \in \mathbb{N}$ is the fan-in of the gate. The fan-out of each gate can be arbitrary, as sharing is authorized in vectorial circuits.

The basic operations in our vectorial circuits are the *bitwise boolean operations*. These are constructed from the usual boolean operations. Notably, we consider the three basic bitwise boolean operators $\vee$, $\wedge$ and $\neg$ defined in section 3.1.2. The circuits given in Figure 3.2 demonstrate how to use these sequences, with or without sharing.

We define a *class of vectorial circuits* as the maximum set of vectorial circuits that use only gates labelled by the operations in a given set.

Vectorial circuit naturally compute functions of the form $f : (\{0,1\}^+)^r \to (\{0,1\}^+)^q$. In order to recognize languages, we need to add a condition on the image of that function.

Definition 3.4

In a vectorial circuit, an *acceptance condition* is a function of the form $f : (\{0,1\}^+)^r \to \{0,1\}$.

Definition 3.5

Let C be a vectorial circuit that computes a function of the form $(\{0,1\}^+)^r \to (\{0,1\}^+)^q$, and let $f : (\{0,1\}^+)^q \to \{0,1\}$ be an acceptance condition. We say that the pair (C, f) *accepts* a word $w \in (\{0,1\}^+)^r$ is $f(C(w)) = 1$.

Now, we define our acceptance condition for the circuits in the main results of this thesis (the ADD-vectorial circuits and the Sweeping-vectorial circuits).

Definition 3.6

Given C an ADD-vectorial circuit (see definition 3.7) or a Sweeping-vectorial circuit (see definition 3.8) with output vectors $\mathbf{out_0}, \ldots, \mathbf{out_{k-1}}$, the acceptance condition of the circuit C is the function $f : (\{0,1\}^+)^k \to \{0,1\}$ such that, for any word $w \in \Sigma^*$, $f(C(w)) = 1$ if and only if the vector

$$\bigvee_{i\in[k]} \mathbf{out_i}$$

is of the form $0\cdots01\cdots1$ with at least one occurrence of 1, that is, it belongs to the language 0^*1^+.

Term notation. The circuits with exactly one output gate can be advantageously denoted by terms built with operations (respecting their arity) and input gates. For example, the term $(\mathbf{v_1} \wedge \neg\mathbf{v_2}) \vee (\neg\mathbf{v_3} \wedge \mathbf{v_4})$ represents the left-most circuit of Figure 3.3. Allowing gates

to have several occurrences in terms gives access to some limited kind of sharing. This is exemplified with the central and right-most circuits of Figure 3.3. So as to fully capture such sharing capabilities with the term notation, we use equations: a term t that is to be shared is associated to a gate $\mathbf{v}$ with the equation $\mathbf{v} = t$ and, when $\mathbf{v}$ is used in another term, this refers to the *shared* circuit t. For example, we write $\mathbf{v} = \neg\mathbf{v_1} \wedge \mathbf{v_2}$, $(\neg\mathbf{v} \wedge \mathbf{v_3}) \vee (\mathbf{v} \wedge \mathbf{v_4})$ to denote the third circuit in Figure 3.3.

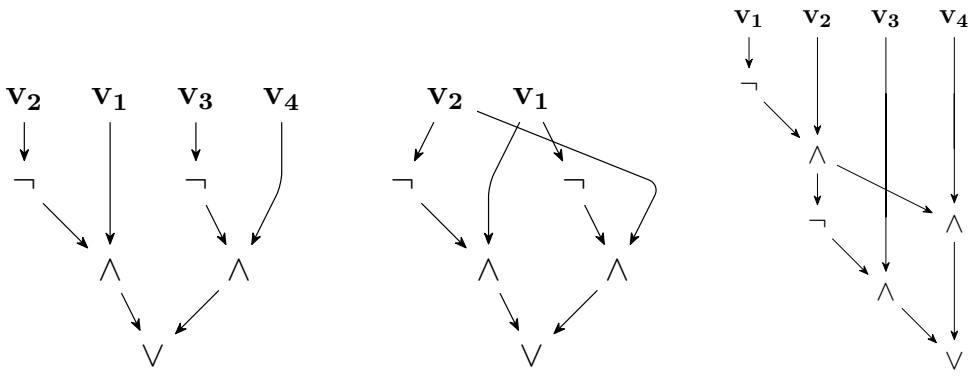


Figure 3.3 – Graph representation of terms: $(\mathbf{v_1} \wedge \neg\mathbf{v_2}) \vee (\neg\mathbf{v_3} \wedge \mathbf{v_4})$; $(\mathbf{v_1} \wedge \neg\mathbf{v_2}) \vee (\neg\mathbf{v_1} \wedge \mathbf{v_2})$; and $\mathbf{v} = \neg\mathbf{v_1} \wedge \mathbf{v_2}$, $(\neg\mathbf{v} \wedge \mathbf{v_3}) \vee (\mathbf{v} \wedge \mathbf{v_4})$

Extending circuits to arbitrary alphabets. Given a fixed alphabet Σ, we say that a vectorial circuit C *recognizes* a set of words in Σ^+ when it has a unique output gate and there is a bijection between the letters a_0, …, a_{p-1} of Σ and its input gates $\mathbf{a_0}$, …, $\mathbf{a_{p-1}}$. We consider a particular bijection: given a word u of length n, we write $\mathbb{1}_a(u)$ for the vector $\mathbf{x}$ of dimension n so that, for every i in $[n]$, $\mathbf{x}_i = 1$ if and only if $u_i = a$. We say that u is *accepted* or *recognized* by the circuit when $C(\mathbb{1}_{a_1}(u), \ldots, \mathbb{1}_{a_p}(u)) \neq \mathbf{0}$. As a shorthand, we write $\mathrm{enc}(u)$ for the tuple $(\mathbb{1}_{a_1}(u), \ldots, \mathbb{1}_{a_p}(u))$ and thus $C(\mathrm{enc}(u))$ for $C(\mathbb{1}_{a_1}(u), \ldots, \mathbb{1}_{a_p}(u))$.

Vectorial circuits can also represent functions f from Σ^+ to a finite domain E. It suffices to consider circuits C which have a bijection between their input gates and the letters of Σ, but also a bijection between the elements e_0, …, e_{r-1} of E and their output gates $\mathbf{e_0}$, …, $\mathbf{e_{r-1}}$. We say that C *represents* f when for every u, the output $(\mathbf{z_0}, \ldots, \mathbf{z_{r-1}}) = C(\mathrm{enc}(u))$ is such that, for every i in $[r]$, $\mathbf{z_i} \neq \mathbf{0}$ if and only if $f(u) = e_i$.

Link with boolean circuit classes. There is a tight link between vectorial circuits and boolean circuit families, as one vectorial circuit can be seen as a uniform family of circuits which manipulate vectors of fixed dimensions (see [36, Section 5]). If we can give boolean circuits computing all the functions used in a vectorial circuit, we then have an equivalence with a family of boolean circuits. There is a similar property for classes of vectorial circuits:

Proposition 3.1

Let P be a set of operations on vectors and $F\mathscr{C}$ a functional class of boolean circuits. Let $\mathscr{V}$ be the class of vectorial circuits parametrized by P. If any function in P can be computed using a family of boolean circuits in $F\mathscr{C}$, then all the functions computed by the class $\mathscr{V}$ are in the class $F\mathscr{C}$.

Proof. We prove the result by induction on the depth of the circuits. Consider an arbitrary vectorial circuit C in $\mathscr{V}$. Let $f : (\{0,1\}^+)^q \to (\{0,1\}^+)^r$ be the function computed by the circuit C. For the base case, suppose that C is of depth 0. Therefore, C has only one layer of input gates, some of which are output gates. That vectorial circuit can be transformed into a family $(C_n)_{n\in\mathbb{N}^+}$ of boolean circuits of depth 0 by splitting the input gates into separate booleans: each circuit C_n has $n \times q$ input gates, which corresponds to q vectors of dimension n. This family of circuits is of constant depth and has no wires, and thus is necessarily in $F\mathscr{C}$.

Now, suppose that C is of depth $n+1$, for $n \in \mathbb{N}$ and that any function computed a circuit of depth n in $\mathscr{V}$ is in $F\mathscr{C}$. We can transform the vectorial circuit C into into a family $(C_n)_{n\in\mathbb{N}^+}$ of boolean circuits in $F\mathscr{C}$. To do so, let C' be the circuit obtained from C by removing the gates of depth $n+1$. C' is of depth n, so by induction hypothesis there exists a family $(C'_n)_{n\in\mathbb{N}^+}$ of boolean circuits in $F\mathscr{C}$ that computes the same function as C'. Moreover, any function in P can be computed using a family of boolean circuits in $F\mathscr{C}$, thus we can replace each output gate by the corresponding circuit and add those to the circuits C'_n by connecting them to the output gates of the circuits. This gives the family $(C_n)_{n\in\mathbb{N}^+}$, which by construction is in the class $F\mathscr{C}$. □

What about constant gates?. The circuits that we present in this thesis use only input gates that depend solely on the input vectors. It is also possible to consider *constant gates*, as a generalization of the constants 0 and 1 in boolean circuits. In the context of vectorial circuits, we define a constant gate to be an infinite sequence of binary words $C = (\mathbf{C_n} \in \{0,1\}^n)_{n\in\mathbb{N}}$. This is equivalent to another characterization, where a constant gate is a language $L \subseteq \{0,1\}^+$ such that, for any $n \in \mathbb{N}^+$, $|L \cap \{0,1\}^n| = 1$. Note that in our case we consider only regular languages and thus, as a result of Theorem 3.2 of [22], we can suppose the language L to be regular, since a constant gate C can be seen as a numerical monadic predicate. However, in the general case, constant gates allow to define circuits that recognize non-regular languages.

Example 3.10. Consider the language of all the binary words such that a bit is set to 1 if and only if its position is a power of 2. It is not possible to recognize this language with a vectorial circuit that uses only the addition, the prefix and suffix operations, and the usual bitwise boolean operations, since these circuits are the ADD-vectorial circuits defined in section 3.3.2, and we prove that these circuits form a subclass of AC^0. However, if we define the constant gate C_{power_2} as the set of constant vectors such that any bit is set to 1 if and only if its index is a power of 2, then we can build a vectorial circuit of constant depth that outputs a vector different from $\mathbf{0}$ if and only if the input word is in the language. Moreover, we need no complex operations to do so, as we only need to compare the input vector $\mathbf{w}$ with C_{power_2}, which can be done using $\oplus$. The word belongs to the language if and only if the result after the $\oplus$ gate is $\mathbf{0}$, so we add a $\neg$ gate, which allows us to recognize the language with our acceptance condition.

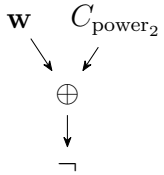


Figure 3.4 – A vectorial circuit recognizing the language of binary words where the positions set to 1 are exactly the ones that are powers of 2

As in boolean circuits, constant vectors equal to **0** and **1** can be obtained from any input vector using a circuit of constant size. Thus, these can be used anywhere as constant gates without changing the expressive power of the circuit class, as long as the basic boolean operations are available in said class. However, more complex constants cannot be obtained from any input vector. Adding these constant vectors as gates in a circuit class can drastically change its expressive power, even among the regular languages.

Example 3.11. Consider the language $L = (ab)^+$ on the alphabet $\{a, b\}$, which is the language of all words that alternate between a and b, start by a and end by b. To recognize this language, we need to verify that the word starts with the letter a, ends with b, and that no infix of the input word is equal to aa or bb. To do so with a vectorial circuit which does not use arbitrary constants, it is necessary to be able to move information at least locally, using a shift instruction or the binary addition for example, to test the infixes of length 2. However, it is possible to use the constants $\mathbf{first} = 10\cdots0$ and $\mathbf{last} = 0\cdots01$. Indeed, both can be obtained from the trivial constant $\mathbf{1} = 1\cdots1$ and the left and right shifts: we have $\mathbf{first} = \neg\text{rshift}_1(\mathbf{1})$ and $\mathbf{last} = \neg\text{lshift}_1(\mathbf{1})$. In Figure 3.5, we give the vectorial circuit using the right shift rshift_k.

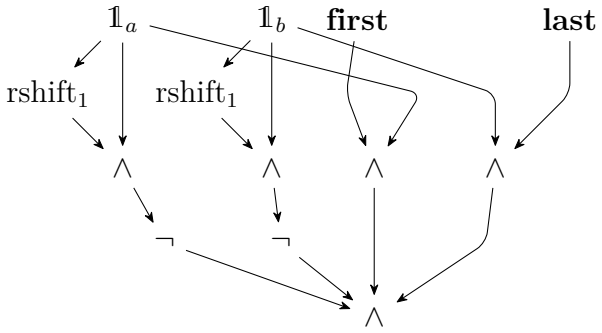


Figure 3.5 – A vectorial circuit recognizing $(ab)^+$ without constant gates

If we allow ourselves to use constant gates, we only need the basic boolean operations, as shown in Figure 3.6. Thus, adding the two constants shown in this circuit increases the expressivity of the circuit to the point where we can get rid of the shift operation, though it has a significant expressivity.

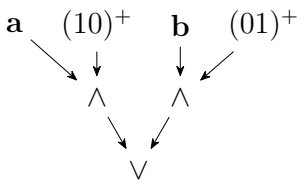


Figure 3.6 – A vectorial circuit recognizing $(ab)^+$ with constant gates

Now that we defined vectorial circuits and their general properties, let's talk about the particular choices that were made in this thesis.

The base operations labelling the gates. In this thesis, we restrict ourselves to a few operations. First, we use the bitwise boolean operations defined in section 3.1.2: the unary negation $\neg$ and the binary operations $\wedge$ and $\vee$, respectively the bitwise conjunction and disjunction.

We also use the prefix and suffix operations, which allow to mark respectively a suffix and a prefix of some word. Given a function $f : \{0,1\}^+ \to \{0,1\}$, we define the unary operation pref-f (resp. suf-f): given a vector $\mathbf{x} = b_0 \dots b_{n-1}$ of dimension n, with b_0, ..., b_{n-1} in $\{0,1\}$, pref-$f(\mathbf{x})$ (resp. suf-$f(\mathbf{x})$) is the vector $\mathbf{z} = c_0 \dots c_{n-1}$ where for each $i \in [n]$, $c_i = f(b_0 \cdots b_i)$ (resp. $c_i = f(b_i \cdots b_{n-1})$). In this thesis, we use the unary operations pref-$\vee$, suf-$\vee$, pref-$\wedge$ and suf-$\wedge$, that we call respectively prefix-or, suffix-or, prefix-and, and suffix-and.

Example 3.12. pref-$\vee$(00101110) = 00111111
suf-$\vee$(00101110) = 11111110
pref-$\wedge$(00101110) = suf-$\wedge$(00101110) = 00000000
pref-$\wedge$(11101010) = 11100000

The binary addition, defined in section 3.1.2, might be the most important operation that we use. This function performs the usual binary addition, but from left to right, and does not keep the highest bit of the result if the length exceeds the dimension of the input vectors.

We use two more incidental operations, called LSB (Least Significant Bit) and MSB (Most Significant Bit). The two unary operations replace by 0 respectively the left-most 1 and right-most 1 of their argument vector. For these two operations, when the argument vector is $\mathbf{0}_n$, the resulting vector is also $\mathbf{0}_n$. These operations can be simulated using the addition and the prefix and suffix operations, but one of the circuit classes that interests us does not have access to addition. Thus, we define them as part of our base gate functions. Defining LSB and MSB as base operations is not devoid of sense, as processors have access to instructions giving the positions of the first and last bit set to 1.

Circuit composition. We can compose vectorial circuits: given two circuits $\mathscr{C}_1$ and $\mathscr{C}_2$ associated with some functions $F_1 : (\{0,1\}^+)^n \to (\{0,1\}^+)^m$ and $F_2 : (\{0,1\}^+)^m \to (\{0,1\}^+)^p$, we can compose $\mathscr{C}_1$ and $\mathscr{C}_2$ to produce a circuit associated with the function $F_2 \circ F_1$.

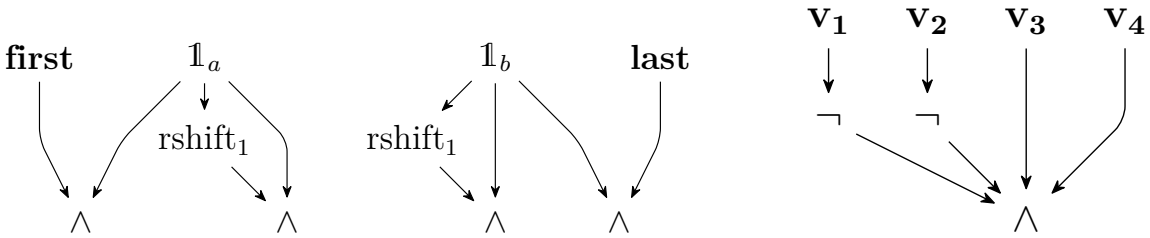
This composition translates well in term notation. We adopt a notation that denotes *parametrized circuits*: $\mathrm{c}(\mathbf{v_1}, \dots, \mathbf{v_n}) := t$ where t is a term built with the gates $\mathbf{v_1}, \dots, \mathbf{v_n}$. For circuits t_1, ..., t_n, we write $\mathrm{c}(t_1, \dots, t_n)$ for the circuit described by the term obtained by replacing $\mathbf{v_1}$, ..., $\mathbf{v_n}$ respectively with t_1, ..., t_n in t. For example, we define the bitwise exclusive-or as $\mathbf{v_1} \oplus \mathbf{v_2} := (\mathbf{v_1} \wedge \neg\mathbf{v_2}) \vee (\neg\mathbf{v_1} \wedge \mathbf{v_2})$. See example 3.13 for a more complete example of circuit composition.

Example 3.13. The circuit given in Figure 3.5 can be seen as the composition of two simpler circuits. For example, the first of these circuits could take the vectors $\mathbb{1}_a$ and $\mathbb{1}_b$ as input and output the tuple $(\mathbf{v_1}, \mathbf{v_2}, \mathbf{v_3}, \mathbf{v_4})$, where $\mathbf{v_1} = \mathbb{1}_a \wedge \mathbf{first}$, $\mathbf{v_2} = \mathbb{1}_a \wedge \mathrm{rshift}_1(\mathbb{1}_a)$, $\mathbf{v_3} = \mathbb{1}_b \wedge \mathrm{rshift}_1(\mathbb{1}_b)$ and $\mathbf{v_4} = \mathbb{1}_b \wedge \mathbf{last}$. This circuit is shown in Figure 3.7a. In that case, the second circuit, shown in Figure 3.7b, takes as input four vectors $\mathbf{v_1}$, $\mathbf{v_2}$, $\mathbf{v_3}$ and $\mathbf{v_4}$, and returns the vector equal to $\neg\mathbf{v_1} \wedge \neg\mathbf{v_2} \wedge \mathbf{v_3} \wedge \mathbf{v_4}$.

If we denote respectively by $\mathcal{C}$, $\mathcal{C}_1$ and $\mathcal{C}_2$ the circuits in Figure 3.5, Figure 3.7a, and Figure 3.7b, we denote the link between them by $\mathcal{C}(\mathbb{1}_a, \mathbb{1}_b) := \mathcal{C}_2 \circ \mathcal{C}_1(\mathbb{1}_a, \mathbb{1}_b)$.

With circuit composition, we can construct complex circuits. We use this to define some circuits that will be used a lot in our results. Notably, for our Sweeping-vectorial circuits, we will often use the operation NotZero, which returns **1** if the input has at least one bit set to 1, and **0** otherwise.

Example 3.14. NotZero(01001) = 11111
NotZero(0000000) = 0000000
NotZero(1000) = 1111



(a) The first circuit composing Figure 3.5

(b) The second circuit composing Figure 3.5

Figure 3.7 – The two circuits composing Figure 3.5

In the Sweeping-vectorial circuits, this operation is defined as follows: for any vector $\mathbf{v}$, NotZero($\mathbf{v}$) := pref-$\vee$(suf-$\vee$($\mathbf{v}$)). This notably allows the circuits to keep the value of a vector if and only if some vector is not null. In particular, we define the operation Eq such that, for any two vectors $\mathbf{u}$ and $\mathbf{v}$ of the same dimension, Eq($\mathbf{u}$, $\mathbf{v}$) is equal to $\mathbf{1}$ if and only if $\mathbf{u} = \mathbf{v}$. Otherwise, it is equal to $\mathbf{0}$. In the Sweeping-vectorial circuits, we set Eq($\mathbf{u}$, $\mathbf{v}$) := $\neg$NotZero($\mathbf{u} \oplus \mathbf{v}$). We also define the operation Thr2 such that, for any vector $\mathbf{u}$, Thr2($\mathbf{u}$) is equal to $\mathbf{1}$ if and only if $\mathbf{u}$ has at least two bits set to 1. Otherwise, it is equal to $\mathbf{0}$. In the Sweeping-vectorial circuits, we set Thr2($\mathbf{u}$) := NotZero(LSB($\mathbf{u}$)). The last operation we need to define using NotZero is IfThenElse. Given three vectors $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$ which have the same dimension, IfThenElse($\mathbf{u}, \mathbf{v}, \mathbf{w}$) is equal to $\mathbf{v}$ if and only if $\mathbf{u}$ is not null. Otherwise, it is equal to $\mathbf{w}$. In the Sweeping-vectorial circuits, we set IfThenElse($\mathbf{u}, \mathbf{v}, \mathbf{w}$) := $(\mathbf{x} \wedge \mathbf{v}) \vee (\neg\mathbf{x} \wedge \mathbf{w})$, where $\mathbf{x}$:= NotZero($\mathbf{u}$).

In the next parts, we present in more details the vectorial circuits that are used in the results shown in chapter 4, namely the ADD-vectorial circuits, used to recognize languages in FO[<], and Sweeping-vectorial languages, used to recognize languages in $FO_2[<]$.

3.3.2 ADD-vectorial circuits

In the class of vectorial circuits defined in this section, we use the constant gate $\mathbf{End} = 0 \cdots 01$. Formally, $\mathbf{End} = (\mathbf{End_n})_{n\in\mathbb{N}}$, where $\mathbf{End_n} = 0^{n-1}1$.

Definition 3.7

An ADD-*vectorial circuit* is a vectorial circuit built only with $\wedge$, $\vee$, $\neg$, $+$, and the constant gate **End**.

Example 3.15. The vectorial circuit in Figure 3.8 is an ADD-vectorial circuit and computes the term $(\mathbf{a} + \neg\mathbf{b}) \wedge \mathbf{a} \wedge \mathbf{End}$.

A logical equivalent: the class FO[<]. We can show that the ADD-vectorial circuits are as expressive as the class of first-order languages. More precisely, we show that they are equivalent to FO[<], thus adding a new tool to the list of classes equivalent to that well-known class (see Theorem 1.2 for a non-exhaustive list of these equivalent classes).

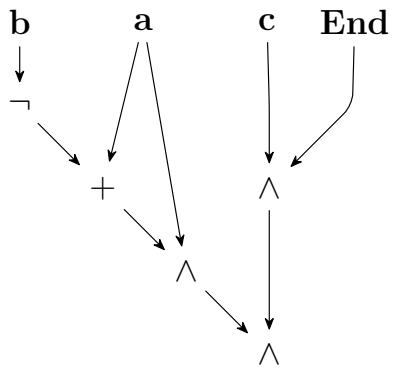


Figure 3.8 – The ADD-vectorial circuit computing $(\mathbf{a} + \neg\mathbf{b}) \wedge \mathbf{a} \wedge \mathbf{End}$

Theorem 3.1

Let Σ be an alphabet and $L \subseteq \Sigma^*$ be a language. Then, L is definable in LTL if and only if there exists an ADD-vectorial circuit that recognizes L.

Proof. First, suppose that L is definable in LTL. Since LTL is as expressive as Past LTL, there exists a Past LTL formula ϕ that recognizes L. We can construct an ADD-vectorial circuit equivalent to the formula ϕ by induction on the structure of ϕ. Given an atomic proposition p_a with $a \in \Sigma$, we replace it with the gate $\mathbb{1}_a$ bearing the indicator vector of a. For the induction step, we replace the usual boolean operators $\vee$, $\wedge$ and $\neg$ by their vectorial counterparts. If $\phi = Y\psi$, by induction hypothesis we can consider an ADD-vectorial circuit C_ψ that recognizes ψ. Then, the ADD-circuit $C_\psi + C_\psi$ recognizes ϕ. This is a consequence of the addition lemma (see Lemma 4.7), with $\mathbf{x} = \mathbf{z} = \mathbf{1}$ and $\mathbf{y} = C_\psi$. Finally, if $\phi = \psi_1 \text{ S } \psi_2$, we use the addition lemma (see Lemma 4.7): by definition of Successor, we have $C_\phi = \text{Successor}\,(C_{\psi_1} \vee C_{\psi_2}, C_{\psi_2}, \neg C_{\psi_1})$, and thus $C_\phi = (C_{\psi_2} + (C_{\psi_1} \vee C_{\psi_2})) \wedge (C_{\psi_1} \vee C_{\psi_2})$. In order to satisfy our acceptance condition, we want to obtain a vector of the form 0^*1^+ if and only if the input word belongs to L. By definition of formulas in LTL, the output vector of the circuit we built ends with a bit set to 1 if and only if the input word belongs to the language. Thus, we consider the function $f : \{0, 1\}^* \to \{0, 1\}^*$ defined by $f(C_\phi(w)) = 1$ if and only if $C_\phi \wedge \mathbf{End}$ is not null. With this acceptance condition, the circuit C_ϕ recognizes the language L.

Now, suppose that we have an ADD-vectorial circuit C_L that recognizes L. Suppose that C_L does not use sharing (by definition, it is always possible to build a circuit without sharing from a circuit that uses it). We can construct by induction on the structure of the circuit an LTL formula which recognizes L (i.e. it is true for the first position of a word w if and only if $w \in L$). We replace any gate $\mathbb{1}_a$ by an atomic proposition p_a. We replace the constant **End** by the formula $\mathrm{X}\mathit{false}$, which is true only for the last letter of a word. For the induction step, we replace the bitwise gates $\vee$, $\wedge$ and $\neg$ by their boolean counterparts. Finally, if the circuit is of the form $C_L = C_1 + C_2$, where C_1 and C_2 are two ADD-vectorial circuits equivalent to two LTL formulas ψ_{C_1} and ψ_{C_2}, then the formula $\phi_{C_L} = \psi_{C_1} \oplus \psi_{C_2} \oplus \mathrm{Y}((\psi_{C_1} \wedge \psi_{C_2} \text{ S } (\psi_{C_1} \vee \psi_{C_2})))$ is equivalent to C_L. This equivalence is proved in Coq,[2] and the code can be found in the repository [56]. □

[2]The proof uses an inverted addition, which goes from right to left, and thus states the equivalence with the formula obtained by replacing the since operator by until, and the yesterday operator by next. The formula given here is the same but reversed, going from left to right

A note on Serre's result. Serre [67] proved a similar equivalence result with a slightly different object. He considered vectorial circuits that use

- The indicator vectors $\mathbb{1}_a$ of the letters in the alphabet,
- The boolean operations $\vee$, $\wedge$ and $\neg$,
- The binary addition,
- And the right shift $\uparrow_i$, for $i \in \{0, 1\}$, such that, for any vector $\mathbf{w} = w_0 \cdots w_{n-1}$, $\uparrow_i \mathbf{w} = iw_0 \cdots w_{n-2}$

These are called *PTL-vectorial algorithms*. A PTL-vectorial algorithm *recognizes* a language L if, given a word w, it returns a binary vector of dimension $|w|$ such that the index i of that vector holds a 1 if and only if $w_0 \cdots w_i \in L$. Serre's result states that the set of PTL-vectorial algorithms recognizes exactly the languages in FO[<]. However, there is an important difference between this class and the ADD-vectorial circuits, which lies in the output vectors and the acceptance condition: our circuits recognize a language L if, given a word w, it returns a binary vector of dimension $|w|$ that is of the form 0^*1^+ if and only if $w \in L$. This difference in the acceptance condition is the reason why we introduced the vector **End**, which allows us to mark the end of the input word, where PTL-vectorial programs do not need this.

Example 3.16. Consider the language $\Sigma^* a$, where $a \in \Sigma$. As a result of Serre's paper, there exists a PTL-vectorial algorithm P which takes a word $w \in \Sigma^*$ as input and returns the binary vector of dimension $|w|$ such that the index i of that vector holds a 1 if and only if $w_0 \cdots w_i \in \Sigma^* a$. For example, if $w = youlostthegameha$, the output vector must be 0000000000010001. This vector is exactly the indicator vector of a; indeed, a word belongs to $\Sigma^* a$ if and only if its last letter is an a, and so each prefix of the form $w_0 \cdots w_{i-1} a$ is in the language. This means that the output vector must always be equal to the indicator vector of a, which gives a simple PTL-vectorial algorithm that recognizes $\Sigma^* a$.

In order to build an ADD-vectorial circuit that recognizes $\Sigma^* a$, we need to keep only the bit corresponding to the last letter, since we want the output vector to be of the form 0^*1^+ if the input is accepted, and $\mathbf{0}$ otherwise. To keep only the last bit, we use the constant vector **End**. This way, we can recognize the language $\Sigma^* a$ with the ADD-vectorial circuit $\mathbb{1}_a \wedge \mathbf{End}$: the output vector is equal to **End** if and only if the last letter is an a, and otherwise it is equal to $\mathbf{0}$.

Another difference lies in the use Serre makes of the right shifts $\uparrow_0$ and $\uparrow_1$. We do not need those operations in the ADD-vectorial circuits. Indeed, $\uparrow_0$ can be emulated using the binary addition, and $\uparrow_1$ is not useful to recognize languages in FO[<], as it does not appear in Serre's proof of the fact that any star-free language can be recognized using a PTL-vectorial algorithm. If we want our circuits to be able to perform this operation, we need to allow the use of one more constant vector, equal to $\mathbf{Start} = 10 \cdots 0$, to introduce a 1 at the beginning of the shifted vector.

Using this constant, we can prove that any PTL-vectorial algorithm can be converted into an equivalent ADD-vectorial circuit. The result is trivial for the gates bearing the boolean operations or the binary addition. We now need to construct an ADD-vectorial circuit equivalent to the right shift. We claim that, for any boolean vector $\mathbf{x}$,

$$\uparrow_0 \mathbf{x} = \mathbf{x} + \mathbf{x}$$

This is obvious when considering that boolean vectors are the binary representation of integers. As a consequence of this claim, we can deduce a circuit equivalent to $\uparrow_1$, since we only need to force the first bit of the vector to be equal to 1, which we can do using **Start**.

$$\uparrow_1 \mathbf{x} = (\mathbf{x} + \mathbf{x}) \vee \mathbf{Start}$$

In order to satisfy our acceptance condition, we compute the bitwise AND of the constant **End** and the output vector obtained with the ADD-vectorial circuit equivalent to the PTL-vectorial algorithm.

Now, we can try to prove that any ADD-vectorial circuit can be translated into an equivalent PTL-vectorial algorithm. The result is trivial for the gates bearing the boolean operations or the binary addition.

To prove the equivalence between PTL-vectorial algorithms and ADD-vectorial circuits, we would need to prove that adding the constant vectors $\mathbf{End} = 0 \cdots 01$ and $\mathbf{Start} = 10 \cdots 0$ does not increase the expressivity beyond that of PTL-vectorial algorithms. We do not give a formal proof of that fact, however we remark that both the languages 0^*1 and 10^* belong to the class FO[<], and so they cannot increase the expressivity beyond that class.

A subclass of $\mathbf{AC^0}$. Since the ADD-vectorial circuits with our acceptance condition are equivalent to the class FO[<], they form a subclass of AC^0. It is possible to show that without going through FO[<]. Indeed, the binary addition of two integers, which is the most expressive operation in our circuits, can be computed using a family of boolean circuits of constant depth, with gates of unbounded fan-in, and having a polynomial number of wires. Straubing gave an explicit construction of these circuits in his book [70]. We give a similar construction for the sake of completeness. Let $A = a_0 \cdots a_{n-1}$ and $B = b_0 \cdots b_{n-1}$ be the binary representations of the two integers to be added (the most significant bit being the rightmost one). We represent the truncated sum as $c_0 \cdots c_{n-1}$. In order to compute it, we first compute the carry at each index of the inputs: for each index i such that $1 \leq i \leq n-1$, we define the bit carry_i as follows

$$\text{carry}_i = \bigvee_{j=0}^{i-1} ((a_j \wedge b_j) \wedge \bigwedge_{k=j+1}^{i-1} (a_k \vee b_k))$$

Using this carry, we can compute the sum of the inputs using a circuit of depth 3.

$$c_0 = a_0 \oplus b_0$$

$$\text{for } 1 \leq i \leq n-1,\ c_i = ((a_i \oplus b_i) \wedge \neg\text{carry}_i) \vee ((a_i \oplus \neg b_i) \wedge \text{carry}_i)$$

Therefore, the binary addition belongs to the class AC^0. However, we can prove that no wire linear circuit in AC^0 computes the addition, by using lower bounds about *weak superconcentrators* (see Chandra et al. [15, Theorem 3.6]). Consequently, our ADD-vectorial circuits cannot be computed with an AC^0 circuit having only linearly many wires.

Finally, the constant **End** is trivially computable in AC^0 thanks to the non-uniformity of the boolean circuits. This proves that our circuits form a subclass of AC^0. The following related problem is open and stated in the seminal paper of Furst et al [23].

Open problem 3.1. *Is the binary addition computable with a circuit of linear AC^0?*

If the answer to that open question is positive, then our ADD-vectorial circuits are also in linear AC^0. Hence, to prove that addition is not in linear AC^0, it is sufficient to prove any non-linear circuit lower bound for FO[<] definable languages.

3.3.3 Sweeping-vectorial circuits

Definition 3.8

A Sweeping-*vectorial circuit* is a vectorial circuit built only with the operations $\wedge$, $\vee$, $\neg$, pref-$\vee$, pref-$\wedge$, suf-$\vee$, suf-$\wedge$, LSB and MSB.

Example 3.17. The vectorial circuit in Figure 3.9 is a Sweeping-vectorial circuit and computes the term $\mathbf{v} := \text{pref-}\vee(\text{suf-}\vee(\mathbf{v_1}))$, $(\neg\mathbf{v} \wedge \mathbf{v_2}) \vee (\mathbf{v} \wedge \mathbf{v_3})$.

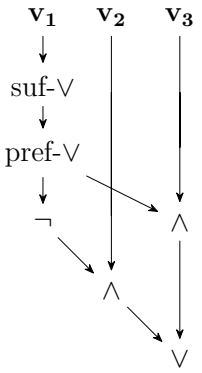


Figure 3.9 – The Sweeping-vectorial circuit computing $(\neg\mathbf{v} \wedge \mathbf{v_2}) \vee (\mathbf{v} \wedge \mathbf{v_3})$, where $\mathbf{v} := \text{pref-}\vee(\text{suf-}\vee(\mathbf{v_1}))$

A subclass of linear AC^0. We can prove that the set of Sweeping-vectorial circuits with an acceptance condition is a subclass of LAC^0. Indeed, Chandra et al. showed [15, Theorem 3.4] that the prefix-or operation can be computed with a boolean circuit of constant depth and with a linear number of wires. As a consequence of this result, the operations pref-$\vee$, pref-$\wedge$, suf-$\vee$ and suf-$\wedge$ can be performed by boolean circuits in LAC^0.

LSB also belongs to LAC^0. Indeed, it can be built from pref-$\vee$ as follows: given a vector $\mathbf{x} = b_0 \cdots b_{n-1}$, where n is the dimension of $\mathbf{x}$, and the vector $\text{pref-}\vee(\mathbf{x}) = p_0 \cdots p_{n-1}$, we have $\text{LSB}(\mathbf{x}) = l_0 \cdots l_{n-1}$, with $l_0 = 0$ and, for each index i such that $1 \leq i \leq n-1$, $l_i = b_i \wedge p_{i-1}$. Since the prefix-or operation belongs to LAC^0, so does LSB. A similar result holds for MSB. As a result of Proposition 3.1, it results that the set of Sweeping-vectorial circuits is a subclass of LAC^0.

A logical equivalent: the class $FO_2[<]$. Sweeping-vectorial circuits are not only a subclass of LAC^0. They are, in fact, equivalent to the class $FO_2[<]$, which adds a new tool to the list of classes equivalent to $FO_2[<]$ (see Theorem 1.3 for a non-exhaustive list of these equivalent classes).

Theorem 3.2

Let Σ be an alphabet and $L \subseteq \Sigma^*$ be a language. There exists a Sweeping-vectorial circuit that recognizes L if and only if $L \in \mathrm{FO}_2[<]$.

To prove this, we use another class of languages that is equivalent to $\mathrm{FO}_2[<]$: the set $\mathrm{TL}[X_a, Y_a]$ of the boolean combinations of turtle programs. As presented in section 1.3.2, this class recognizes the same languages as $\mathrm{FO}_2[<]$, which allows us to use it in our proof.

Proposition 3.2

A language is recognized by a Sweeping-vectorial circuit if and only if it is definable in $\mathrm{TL}[X_a, Y_a]$.

Proof. We rely on the equivalence between $\mathrm{FO}_2[<]$ and $\mathrm{TL}[X_a, Y_a]$, stated in section 1.3.2, to prove the result. First, we show that $\mathrm{TL}[X_a, Y_a]$ is captured by Sweeping-vectorial circuits, i.e. that any formula of $\mathrm{TL}[X_a, Y_a]$ defines a function from words to Boolean vectors that can be modeled by these circuits; here we interpret a formula of $\mathrm{TL}[X_a, Y_a]$ as a function from words to Boolean vectors, by considering the truth vector of a formula on every position of the input word. Clearly, Boolean operations are equivalent in both models. Now, suppose that we have a Sweeping-vectorial circuit C_α that is equivalent to a $\mathrm{TL}[X_a, Y_a]$ formula α, i.e. given a word $u \in \Sigma^+$ and a vector **pos** (of the same dimension as u) that has a unique 1 in some position x, we have $C_\alpha(\mathbf{pos}, \mathrm{enc}(u)) \neq \mathbf{0}$ if and only if $u, x \models \alpha$. Then, for any letter $a \in \Sigma$, the formula $\beta = X_a\alpha$ can be modeled by the circuit C_β defined as follows: for any word $u \in \Sigma^*$ and any vector **pos** having a unique 1, we begin by finding the first letter a strictly after the position marked by **pos**, by sequentially computing the following vectors: $\mathbf{authorizedPos} = \mathrm{LSB}(\text{pref-}\vee(\mathbf{pos}))$, $\mathbf{next} = \mathbf{authorizedPos} \wedge \mathbb{1}_a(u)$ and $\mathbf{nextPos} = \mathbf{next} \oplus \mathrm{LSB}(\mathbf{next})$. This gives us the new position vector to give as parameter to the new circuit. Then we can define $C_\beta(\mathbf{pos}, \mathrm{enc}(u)) := C_\alpha(\mathbf{nextPos}, \mathrm{enc}(u))$, which is equivalent to β. Finally, we can model the formula $\gamma = Y_a\alpha$ using the same formulas, except for the vector **authorizedPos**, which is computed as $\mathbf{authorizedPos} = \mathrm{MSB}(\text{suf-}\vee(\mathbf{pos}))$, in order to get the positions situated strictly before the position x marked by **pos**. Finally, Sweeping-vectorial circuits mimic the initial configurations and acceptance conditions of turtle programs. Indeed, while turtle programs need two initial configurations depending on whether they start on the left or right of the input word, it is sufficient for Sweeping-vectorial circuits to consider the entire word at the beginning of the computation. The fact that it starts at the left or at the right of the word is handled by the gates used to emulate the first instruction of the turtle program. For the acceptance condition, a word is accepted by a turtle program if and only if the computation ends without failing. This means that, when it ends, the program stopped in some valid position in the word. Thus, a word should be accepted by the equivalent Sweeping-vectorial circuit if and only if the output vector is not null, which can be done with a tailored acceptance condition.

Now, we prove that Sweeping-vectorial circuits are captured by formulas in $\mathrm{FO}_2[<]$. Since Sweeping-vectorial circuits output a truth vector indicating if some property is true for each position in the input word, we must consider $\mathrm{FO}_2[<]$ formulas which can take into account a starting position. Thus, we consider only $\mathrm{FO}_2[<]$ formulas with one free variable, which represents the position in which we begin to evaluate the formula in the input word.

Clearly, Boolean operations are equivalent in both models. We can also interpret pref-$\vee$ as an $\mathrm{FO}_2[<]$ formula: consider an $\mathrm{FO}_2[<]$ formula φ with one free variable and suppose that

it is computed by a circuit C_φ. Then, the formula $\varphi_{\text{pref-}\vee}$ with one free variable defined by $\varphi_{\text{pref-}\vee}(y) := \exists x \leq y,\ \varphi(x)$ has its truth vector equal to the output of pref-$\vee(C_\varphi)$. The other prefix and suffix operations can be dealt with in a similar way. Finally, we can interpret LSB and MSB as $\text{FO}_2[<]$ formulas. Indeed, consider an $\text{FO}_2[<]$ formula φ with one free variable and suppose that it is computed by a circuit C_φ. Then, the formula φ_{LSB} with one free variable defined by $\varphi_{\text{LSB}}(y) := \varphi(y) \wedge \exists x < y,\ \varphi(x)$ has its truth vector equal to the output of $\text{LSB}(C_\varphi)$. MSB is equivalent to the same formula where $x < y$ is replaced by $x > y$. □

Remark 3.1. We can always build a vectorial circuit of size linear in the size of the equivalent formula. Conversely, if we consider formulas with sharing of sub-formulas, we can also build an equivalent formula of size linear in the size of the circuit. However, the link between the size of an $\text{FO}_2[<]$ formula and the size of its minimal equivalent turtle program is unclear, as the proofs are not constructive, and there are no known bounds on the size of turtle programs.

3.4 Streaming with circuits

Streaming refers to the continuous and real-time processing of data as it arrives or flows in a sequential manner. It involves the analysis and extraction of patterns from a continuous stream of input data, without the need to store or access the entire dataset at once. Streaming is particularly valuable when dealing with large volumes of data that cannot be accommodated in memory or when immediate processing of incoming data is necessary. Instead of loading and processing the entire dataset, streaming algorithms and techniques are employed to process the data incrementally, typically one element at a time or in small batches.

By this definition, streaming is a natural application of bit-level parallel algorithms. We take this application into account in our formalization by presenting a construction which allows to build a vectorial circuit that processes the input word one chunk at a time and uses the data obtained from those chunk to process the next ones. Thus, even though the circuits presented in this thesis are capable of processing only one complete input word, it is possible to adapt them for streaming by using the construction presented in this section.

Although circuits are a great tool to model algorithms and study their parallel complexity, the connection to streaming is quite recent [51]. The circuits constructed by Murlak et al. are composed of a main circuit, a feedback loop, and an acceptor circuit, whose inputs are the outputs of the main circuit. The main circuit takes as input the feedback, which has initially a neutral value, and the next chunk of data. Those circuits allow to model the notion of bit-level parallelism introduced in [44], using some circuit parameters. In their work, Murlak et al. show that the circuit complexity of a regular language matches the streaming circuit complexity of the language. We proceed similarly to [51] to adapt vectorial circuits to streaming by providing for each base operation a streaming interpretation that can be composed in a streaming way.

3.4.1 Propagating information

Some of the operations we presented in section 3.1.2, such as the three usual bitwise boolean operators, do not require any other information than the input vectors. However, that is not the case for all the operations used for bit-level parallel algorithms. Some operations, such as the binary addition or the left and right shifts, require to keep in memory some information about the previous vectors that were processed. Indeed, the vectorial circuits used for streaming must be equivalent to the ones used on the entire inputs, and so they must propagate some

information between consecutive vectors to simulate the fact that they are adjacent in the input. For example, the usual binary addition requires to keep in memory the carry produced.[3] Thus, it requires one bit of memory to propagate information between the chunks, as shown in example 3.18.

Example 3.18. In table 3.4, we show an example of binary addition where the carry is propagated through the vectors, which are of dimension 5.

v_1	11	30	9
v_2	18	5	19
Boolean writing of v_1	11010	01111	10010
Boolean writing of v_2	01001	10100	11001
Boolean writing of $v_1 + v_2$	10111	11000	10111
Memory	0	1	0
$v_1 + v_2$	29	3	29

Table 3.4 – Performing the binary addition of two vectors of integers, v_1 and v_2

The usual right shift rshift_k requires to keep in memory k bits, as many as it shifts out of the vector.[4] Indeed, the k bits shifted out of the vector are to be inserted into the next one, as shown in example 3.19.

Example 3.19. In table 3.4, we show an example of right shift where the shifted bits are propagated through the vectors, which are of dimension 5.

v_1	1	10	16	27
Boolean writing of v_1	10000	01010	00001	11011
Boolean writing of $\text{rshift}_2(v_1)$	00100	00010	10000	01110
Memory	00	10	01	11
$\text{rshift}_2(v_1)$	4	8	1	14

Table 3.5 – Performing the right shift rshift_2 of the vector of integers v_1

Note that the left shift would require the same memory. However, since it applies from right to left and we read the chunks from left to right, this operation is not used in practice.

3.4.2 Adapting vectorial circuits to streaming

As showed above, some operations, such as the binary addition and the right shift, must produce more information than the vector they compute as a result. If these operations are to be executed on all vectors, this additional information must be kept and used on the subsequent vectors to obtain the same results as if the input was not split into chunks.

Thus, to adapt our circuits to streaming, we represent the memory needed for the additional information with a feedback loop. This feedback loop is made of wires that come out of the gates

[3] There are some cases in which we do not want to keep the carry, such as the shift-add algorithm presented in [2]. This is why we talk about the "usual" binary addition, which propagates its carry.

[4] The same remark applies as for the binary addition: in some cases, we do not want to keep those bits in memory.

producing additional information and get back to the beginning of the circuit by producing new inputs. In this section, we formally describe the construction for gates labeled by the binary addition and the right shift, as these are the only operation relevant to us. However, the general idea is the same no matter the operation: the gates labeled by the operation must be modified to produce two outputs: the vector that results of the operation, and a vector[5] bearing the additional information produced during the computation. The first vector is treated as in the original circuit, and used as input for the next gates. The second one, however, is used as input for the next chunk of data. More specifically, it is used to modify the input vectors of the gate in order to include the additional information in the computation.

The right shift. To adapt the right shift to streaming, we need to use the bits shifted out of the vectors to shift them in the vectors used in the computation of the next chunk. To do so, given a gate labeled by rshift_k, we modify it to produce two output vectors: the result of the shift, and the vector containing the bits shifted out of the input. Formally, the gate produced a vector $\mathbf{shifted} = b_0 \cdots b_{k-1} 0 \cdots 0$, where the k bits $b_0, \ldots, b_{k-1}$ are the last k bits of the input vector. In order to integrate this vector to the computation of the next chunk, we add an input gate labeled $\mathbf{shifted}_i$, such that the corresponding gate is the i^{th} one labeled by a right shift. This input gate is set to $\mathbf{0}$ at the beginning of the computation, and is used only to modify the output of that specific gate, by computing the bitwise OR between the two vectors in order to add the bits at the beginning of the vector.

Example 3.20. In the program written to recognize $(ab)^*$ in section 3.2.2, we used the bit-level parallel formula $(\text{rshift}_1(\mathbb{1}_a) \vee \text{carry}_a) \wedge \mathbb{1}_a$ to verify if there were two adjacent occurrences of a. If we adapt this circuit to streaming using the above construction, we obtain the circuit presented in Figure 3.10.

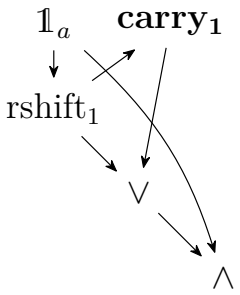


Figure 3.10 – The streaming vectorial circuit that marks infixes equal to aa

The binary addition. The binary addition produces only one bit of additional information, which contains the carry produced at the end of the computation. Thus, there are two possible approaches. The first one is the one used in practice by SIMD operations. It consists in producing only a boolean instead of a vector to propagate the additional information. However, this requires a more flexible definition of vectorial circuits to include boolean input gates in addition to the gates bearing boolean vectors. Since this does not change the theoretical complexity, we will only present the second approach, which consists in creating a vector **carry**, whose bits are all set to 0, except the first one which bears the value of the carry. Formally,

[5] With particularly complex operations, more than one vector might be needed, however one is sufficient to explain the idea of the general case.

$\mathbf{carry} = b0\cdots0$, where the bit b is the value of the carry created by the addition. In order to integrate this vector to the computation of the next chunk, we add an input gate labeled $\mathbf{carry_i}$, such that the corresponding gate is the i^{th} one labeled by the binary addition. This input gate is set to $\mathbf{0}$ at the beginning of the computation, and is used only as input of the corresponding gate, which now takes three vectors: the two input vectors, and $\mathbf{carry_i}$.

Example 3.21. In the program written to recognize $(aA^*b + A)^*$ (with $A = \Sigma \setminus \{a, b\}$) in section 3.2.2, we used the bit-level parallel formula $(\mathbb{1}_a + (\neg\mathbb{1}_b) + \text{carry}_a) \wedge \mathbb{1}_a$ to verify if there was any infix of the form $a(\Sigma \setminus \{a, b\})^*a$, i.e. if there were two occurrences of a with no b in-between. If we adapt this circuit to streaming using the above construction, we obtain the circuit presented in Figure 3.11.

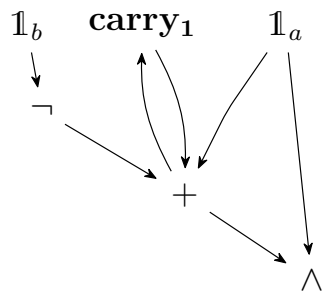


Figure 3.11 – The streaming vectorial circuit that marks infixes of the form $a(\Sigma \setminus \{a, b\})^*a$

Chapter 4

From semigroups to vectorial circuits

Outline of the current chapter

The section 1.2 defined semigroups, along with Green's relations, the tools that allow to study the properties of the semigroups. These properties were used in benchmarks presented in section 2.3 and section 2.4. The results showed that a clever use of the semigroups' properties could contribute to improve the efficiency of programs recognizing regular languages. However, the simple parallel algorithm using semigroups outperforms all the others. In order to produce efficient programs that recognize regular languages using semigroups, taking advantage of the inherent parallelism of semigroups is necessary. Moreover, the benchmark presented in section 3.2 shows that vectorized programs can achieve interesting speeds. This is due to the fact that these programs avoid conditional branching, thus helping the optimization at runtime. This last chapter makes a step toward using algebra-based vectorized programs for data validation. It introduces evaluation programs, which describe successive products of elements in a word over the elements of a semigroup. This constitutes an overlay of abstraction over the vectorized programs that compute the product of this kind of word to determine if some text belongs to a language. The chapter then proceeds to present vectorized programs for two classes of languages: FO[<] and FO_2[<]. These vectorized programs are presented as vectorial

circuits, which are defined in section 3.3, and can be easily translated into vectorized code such as the one used in section 3.2.

4.1 Evaluation programs

In this chapter, we consider words over some semigroup S. Our goal is to compute the product in S of the letters composing these words. For any word $u \in S^+$, this amounts to computing $\pi_S(u)$. This computation can be performed by vectorial circuits. Instead of directly building these circuits, we first pay attention to evaluation strategies that we call *evaluation programs*. These strategies form an overlay of abstraction over the intricacy of circuits. They are meant to modularize the construction of vectorial circuits.

4.1.1 Definition

Given a semigroup S, an evaluation program over S transforms words in S^+ by replacing some of the factors by their values (through the canonical morphism) in S. In this section, we consider a fixed semigroup S.

Definition 4.1

A *partial evaluation step* over S is a relation over S^+ denoted by $\to_S$ and defined as $uvw \to_S u\pi_S(v)w$ for any v in S^+ and $u, w \in S^*$. We denote by $\to_S^+$ the transitive closure of $\to_S$. We say that v *is a partial evaluation of* u when $u \to_S^+ v$.

Note that if v is a partial evaluation of u over S, then $\pi_S(u) = \pi_S(v)$. Usually, the context makes it clear which semigroup is considered. Thus, we generally leave the semigroup implicit and only say that v is a partial evaluation of u. Note that each word u is a partial evaluation of itself.

Definition 4.2

A *partial evaluation program* over S is a partial function $P : S^+ \to S^+$ such that, for any word $u \in S^+$ that is in the domain of P, $P(u)$ *is a partial evaluation of* u. If the domain of P is S^+, then we say that P is *total*.

The function π_S is an example of a partial evaluation (which is, in this case, total). Another example is the function $\mathrm{LProd}_S \colon S^+ \to S^+$ that performs the product of the first two elements of the input, if there are at least two elements, and otherwise returns the input word. Similarly, we define RProd_S as the partial evaluation program which performs the product of the two last elements, if there are at least two elements, and otherwise returns the input word. In symbols, these partial evaluations are defined as follows:

Definition 4.3

The operation LProd_S is defined by $\mathrm{LProd}_S(s_0) = s_0$ for any element $s_0 \in S$, and, for any word $u = s_0 \cdots s_n$ of length at least two, $\mathrm{LProd}_S(u) = \pi_S(s_0 s_1) s_2 \cdots s_n$. Similarly, the

operation RProd_S is defined by $\mathrm{RProd}_S(s_0) = s_0$ for any element $s_0 \in S$, and, for any word $u = s_0 \cdots s_n$ of length at least two, $\mathrm{RProd}_S(u) = s_0 \cdots s_{n-2}\pi_S(s_{n-1}s_n)$.

Note that evaluation programs are closed under composition (see example 4.1).

Example 4.1. The composition of two occurrences of LProd_S and one occurrence of RProd_S results in the evaluation program which takes a word and computes the products of the first three elements and the last two elements. Consider the evaluation program $P = \mathrm{LProd}_S \circ \mathrm{RProd}_S \circ \mathrm{LProd}_S$ and the words $u = u_0u_1u_2u_3u_4$ and $v = v_0v_1v_2$. If we apply P to u, the program starts by computing the product of the first two elements, which results in the word $u' = \pi_S(u_0u_1)u_2u_3u_4$. Then it computes the product of the last two elements of u', resulting in the word $u'' = \pi_S(u_0u_1)u_2\pi_S(u_3u_4)$. Finally, it computes the product of the first two elements of u'', so $P(u) = \pi_S(u_0u_1)\pi_S(u_2u_3u_4)$. If we apply P to v, we obtain successively the words $v' = \pi_S(v_0v_1)v_2$, $v'' = \pi_S(v_0v_1v_2) = \pi_S(v)$ and $P(v) = \pi_S(v)$. Note that the evaluation programs LProd_S and RProd_S do not commute, as $P(u)$ would be different if we defined P as $P = \mathrm{LProd}_S \circ \mathrm{LProd}_S \circ \mathrm{RProd}_S$.

4.1.2 Waterfall evaluation programs

In this part, we are designing a first set of specific evaluation programs. These programs work by evaluating the semigroup in a top-down fashion (with respect to the $\mathcal{J}$-order). To do so, in this section, we consider operations that can be successively applied to all the sub-semigroups $Q_i(S)$ (defined in section 1.2.3) of a given semigroup S. At each step, we detect the set of maximal infixes whose product is maximal for the $\mathcal{J}$-order of $Q_i(S)$, evaluate those infixes and multiply the results with the letters that immediately follow. Then, we remove in turn each $\mathcal{J}$-maximal element by computing the product of each occurrence with the next letter on the right. Each step computes a new word whose letters all belong to $Q_{i+1}(S)$, except for the last one, which can still belong to $D_i(S)$. We deal with this letter at the end of the computation.

Definition 4.4

A word u in S^+ is *$\mathcal{J}$-maximal falling* whenever for every $p, s \in S^*$ and $v, w \in S$ so that $u = pvws$, we have $\pi_S(vw) \in Q_2(S)$.

See example 4.2 for an example of $\mathcal{J}$-maximal falling word. Note that when u is $\mathcal{J}$-maximal falling and $|u| > 1$, $\pi_S(u) \in Q_2(S)$.

Example 4.2. Consider the syntactic semigroup of $(ab)^+$, whose egg-box is shown in example 4.2. This semigroup is composed of five elements: 0 corresponds to the letter a and all the words of the form $a(ba)^*$, 1 corresponds to the letter b and all the words of the form $b(ab)^*$, 2 corresponds to all the words of the form $(ab)^+$, 3 corresponds to all the words of the form $(ba)^+$, and 4 corresponds to all the words containing either aa or bb as an infix. As a consequence of this correspondence, the inner product of that semigroups is such that $\pi_S(00) = \pi_S(11) = 4$. The semigroup is composed of two $\mathcal{J}$-classes, the class of $\mathcal{J}$-depth 1 containing 0, 1, 2 and 3, and the class of $\mathcal{J}$-depth 2 containing only 4.

For this semigroup, the words $u = 000000$ and $v = 1111$ are $\mathcal{J}$-maximal falling, since the product of any infix of length 2 is equal to 4, which is of $\mathcal{J}$-depth 2. However, the word $w = 00001$ is not $\mathcal{J}$-maximal falling, as the product of the last two letters is $\pi_S(01) = 2$, an element of $\mathcal{J}$-depth 1.

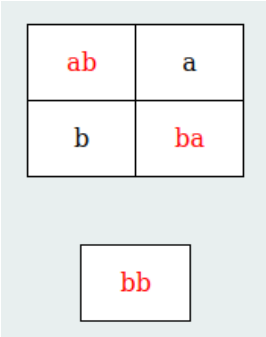


Figure 4.1 – The syntactic semigroup of $(ab)^+$

Definition 4.5: $\mathcal{J}$-maximal decomposition

Consider a semigroup S and a word $u \in S^+$. If $u \notin Q_2(S)^+$, let $t \in \mathbb{N}$ and some words $w_0, \ldots, w_{t+1} \in S^*$, $v_0, \ldots, v_t \in S^+$, that define a decomposition $u = w_0 v_0 w_1 \cdots v_t w_{t+1}$. This decomposition is called *$\mathcal{J}$-maximal* in S if the following two conditions are satisfied:

- For any integer i such that $0 \leq i \leq t+1$, w_i is a word in $Q_2(S)^*$.
- For any integer i such that $1 \leq i \leq t$, v_i is a maximal infix of u verifying $\pi_S(v_i) \in D_1(S)$. More formally, if we consider the decomposition $u = pav_ibs$, with $p, s \in S^*$, and $a, b \in S \cup \epsilon$ (a is the letter preceding v_i, if it exists, and b is the letter following v_i, if it exists) then, if $a \neq \epsilon$, $\pi_S(av_i) <_{\mathcal{J}} \pi_S(v_i)$ and, if $b \neq \epsilon$, $\pi_S(v_ib) <_{\mathcal{J}} \pi_S(v_i)$.

We call $t+1$ the size of the decomposition.

If $u \in Q_2(S)^+$, the $\mathcal{J}$-maximal decomposition of u in S is composed of only one element equal to u itself. In this case, the decomposition is unique and by convention of size 0.

Remark 4.1. The property of the (v_i)'s implies that each v_i is a word of $D_1(S)^+$. Indeed, the product of a word is at most $\mathcal{J}$-equivalent to the letter of greatest $\mathcal{J}$-depth, so all the letters must be of $\mathcal{J}$-depth 1 for the product to be in $D_1(S)$.

See example 4.3 for an example of $\mathcal{J}$-maximal decomposition.

Example 4.3. Consider again the syntactic semigroup S of $(ab)^+$, composed of five elements such that $\pi_S(01) = \pi_S(22) = 2$, $\pi_S(10) = \pi_S(33) = 3$, $\pi_S(20) = \pi_S(03) = 0$, $\pi_S(31) = \pi_S(12) = 1$ and, for any other pair $(s,t) \in S \times S$, $\pi_S(st) = 4$. Now, consider the word $u = 40100312214413201$. This word admits a $\mathcal{J}$-maximal decomposition of size 6, defined as follows: $w_0 = 4$, $v_0 = 010$, $w_1 = \epsilon$, $v_1 = 03122$, $w_2 = \epsilon$, $v_2 = 1$, $w_3 = 44$, $v_3 = 1$, $w_4 = \epsilon$, $v_4 = 3$, $w_5 = \epsilon$, $v_5 = 201$, and $w_6 = \epsilon$. If we use parentheses to represent this decomposition, we obtain $u = (4)(010)()(03122)()(1)(44)(1)()(3)()(201)()$. All the conditions for a decomposition to be $\mathcal{J}$-maximal are fulfilled. First, each word w_i is in 4^*, which is equal to $Q_2(S)^*$ for this semigroup S. Moreover, the product of each word v_i is of $\mathcal{J}$-depth 1: $\pi_S(v_0) = 0$, $\pi_S(v_1) = 2$, $\pi_S(v_2) = 1$, $\pi_S(v_3) = 1$, $\pi_S(v_4) = 3$, and $\pi_S(v_5) = 2$. Finally, each v_i is maximal, since the product with the letters to their left or right is equal to 4, which is in $Q_2(S)$.

Every word in S^+ admits a unique $\mathcal{J}$-maximal decomposition in S. This property hinges on the Localization Theorem of Clifford and Miller [61, Proposition 1.6, page 48].

Lemma 4.1: Localization Theorem

Let S be a semigroup and x, y be in S. We have $xy\mathcal{J}x$ if and only if there exists an idempotent e in $R(y) \cap L(x)$.

We will use the following useful technical lemma.

Lemma 4.2

Let S be a semigroup. Let x, y be $\mathcal{J}$-equivalent elements of S and z another element of S.

- If $\pi_S(xy)\mathcal{J}x$ and $\pi_S(zxy) <_{\mathcal{J}} x$, then $\pi_S(zx) <_{\mathcal{J}} x$.
- If $\pi_S(xy)\mathcal{J}x$ and $\pi_S(xyz) <_{\mathcal{J}} x$, then $\pi_S(yz) <_{\mathcal{J}} x$.

Proof. Both cases are symmetric, so we only prove the first one. Suppose that $\pi_S(zx)\mathcal{J}x$, $\pi_S(xy)\mathcal{J}x$ and $\pi_S(zxy) <_{\mathcal{J}} x$. Lemma 4.1 implies that there is no idempotent in $R(y) \cap L(zx)$. But, as $\pi_S(zx)\mathcal{J}x$, by definition of $\mathcal{L}$, we have $\mathcal{L}(zx) = \mathcal{L}(x)$. Therefore $R(y) \cap L(x)$ does not contain an idempotent. Finally Lemma 4.1 entails $xy <_{\mathcal{J}} x$, a contradiction. □

We are now proving the existence and uniqueness of $\mathcal{J}$-maximal decomposition. The proof works by considering a variant of $\mathcal{J}$-maximal decomposition by enforcing the maximality constraint only at the right and showing that the two variants are actually equivalent.

Lemma 4.3

Let S be a semigroup. For any finite word $u \in S^+$, there exists a unique $\mathcal{J}$-maximal decomposition of u.

Proof. We focus on proving the existence of such a decomposition. Uniqueness follows from the proof of existence. First, note that $S = D_1(S) \uplus Q_2(S)$. Thus, for any word $u \in S^+$, there necessarily exists a unique decomposition of u of the form $u = w_0 v_0 w_1 \cdots w_t v_t w_{t+1}$ such that

- $w_0, w_{t+1} \in Q_2(S)^*$.
- For each integer i such that $1 \leq i \leq t$, $w_i \in Q_2(S)^+$.
- For each $i \in [t+1]$, $v_i \in D_1(S)^+$.

Informally, this is a decomposition of u based only on the $\mathcal{J}$-depth of each letter. Thanks to remark 4.1, we know that the words w_i that are not empty in any $\mathcal{J}$-maximal decomposition of u correspond exactly to the words w_i of this decomposition. Thus, we only need to prove that any word over $D_1(S)$ can be decomposed into a unique sequence of maximal infixes whose product is in $D_1(S)$.

Consider a fixed word $u \in D_1(S)^+$. We prove that there exists a decomposition of u of the form $u = v_0 \cdots v_s$, where each word v_i is maximal in u such that $\pi_S(v_i) \in D_1(S)$.

If $\pi_S(u) \in D_1(S)$, then both existence and uniqueness follow from the definition.

Suppose now that $\pi_S(u) \in Q_2(S)$. Then we define a decomposition of u with weaker properties than the $\mathcal{J}$-maximal decompositions, and we prove that it is a $\mathcal{J}$-maximal decomposition of u (with empty words as the (w_i)'s). This decomposition is of the form $u = v_0 v_1 \cdots v_t$ for

some integer $t \in \mathbb{N}$ and is defined from left to right such that, for each integer $i \in [t+1]$, $\pi_S(v_i) \in D_1(S)$ and, for each integer $i \in [t]$, $\pi_S(v_i x_{i+1}) \in Q_2(S)$, where x_{i+1} is the first letter of v_{i+1}. By construction, this decomposition exists and is unique. We prove that this decomposition satisfies the properties of the (v_i)'s given in definition 4.5, that is, we prove that each word v_i is maximal such that $\pi_S(v_i) \in D_1(S)$. To do that, we only need to prove that, for each integer i such that $1 \leq i \leq t$, $\pi_S(y_{i-1}v_i) \in Q_2(S)$, where y_{i-1} is the last letter of the word v_{i-1}.

Thus, consider some integer $i \in [t]$. We focus on the infix $v_i v_{i+1}$ in order to prove that $\pi_S(y_i v_{i+1}) \in Q_2(S)$, where y_i is the last letter of v_i. This fact is a consequence of Lemma 4.2. Since we know that $\pi_S(v_i x_{i+1}) \in Q_2(S)$, where x_{i+1} is the first letter of v_{i+1}, and that $\pi_S(v_i) \in D_1(S)$, Lemma 4.2 implies that $\pi_S(y_i x_{i+1}) \in Q_2(S)$. Thus, $\pi_S(y_i v_{i+1}) \in Q_2(S)$. Thus, this decomposition is the unique $\mathcal{J}$-maximal decomposition of u. □

Now, we need to refine the $\mathcal{J}$-maximal decomposition. Keeping the same notation for u, and the $\mathcal{J}$-maximal decomposition $u = w_0 v_0 \cdots v_t w_{t+1}$ of its prefix in $Q_i(S)$, we split the words v_i by setting aside their first letter and defining $v_i = x_i v_i'$ ($x_i \in S$ is the first letter of v_i). Then, we compute the product of the words v_i' with the first letter of the words w_i. To do so, we define the evaluation program $\mathrm{Collapse}_{S,i}$.

Definition 4.6

Consider a semigroup S of $\mathcal{J}$-depth d, an integer $i \leq d$ and a word $u = u'a$, with $u' \in Q_i(S)^+$ and $a \in (S \setminus Q_i(S)) \cup \{\epsilon\}$. Let $w_0 v_0 \cdots v_t w_{t+1}$ be the $\mathcal{J}$-maximal decomposition of u' in $Q_i(S)$. We set, for any $0 \leq i \leq t$, $v_i = x_i v_i'$, $w_i' = w_i x_i$ and $w_i' = y_i w_i''$, so that $u' = w_0' v_0' y_1 w_1'' \cdots w_t'' v_t' w_{t+1}$. Then, the word $\mathrm{Collapse}_{S,i}(u)$ is defined as follows:

$$\mathrm{Collapse}_{S,i}(u) = w_0' \pi_S(v_0' y_1) w_1'' \cdots \pi_S(v_i' y_{i+1}) w_{i+1}'' \cdots w_t'' z$$

where z denotes either $\pi_S(v_t' a)$ if w_{t+1} is the empty word, or $\pi_S(v_t' y_{t+1}) w_{t+1}' a$.

Remark that the image of any word $u \in S^+$ by $\mathrm{Collapse}_{S,i}$ is a word $v = v'a$ such that v is a $\mathcal{J}$-maximal falling word (see definition 4.4). In order to be able to repeat the operation $\mathrm{Collapse}_{S,i}$, we need to perform more products and obtain a word in $Q_{i+1}(S)^* \times (S \cup \{\epsilon\})$. To that end, we define the partial evaluation program $\mathrm{Falling}_S(s)$ which takes a word $u = u'a \in Q_i(S)^+ \times ((S \setminus Q_i(S)) \cup \{\epsilon\})$ such that $u' \in Q_i(S)^+$ is a $\mathcal{J}$-maximal falling word, and returns a partial evaluation of u in which there is no occurrence of s, except potentially for the last letter. To that end, we first define the s-decomposition of a word u, which is the decomposition of the form $u = w_0 s^{k_0} x_0 w_1 \cdots s^{k_t} x_t w_{t+1}$ where the k_i's are strictly positive integers, the x_i's are non-s elements of S, except for x_t which can also be equal to ϵ if $w_{t+1} = \epsilon$, and the w_i's are words without any occurrence of s. The s-decomposition of a word always exists and is unique. With this, we can define the evaluation program $\mathrm{Falling}_S(s)$.

Given the s-decomposition of some word $u \in S^+$, keeping the same notation, we define

Definition 4.7

Consider a semigroup S of $\mathcal{J}$-depth d, an integer $i \leq d$, an element $s \in Q_i(S)$, and a word $u = u'a$, where $u' \in Q_i(S)^+$ is a $\mathcal{J}$-maximal falling word and $a \in (S \setminus Q_i(S)) \cup \{\epsilon\}$. Let $u' = w_0 s^{k_0} x_0 w_1 \cdots s^{k_t} x_t w_{t+1}$ be the s-decomposition of u'. The partial evaluation of u

denoted by $\mathrm{Falling}_S(s)(u)$ is defined as follows:

$$\mathrm{Falling}_S(s)(u) = w_0\pi_S(s^{k_1}x_1)\cdots\pi_S(s^{k_{t-1}}x_{t-1})w_tvw_{t+1}a'$$

where v and a' are such that

- If $x_t \neq \epsilon$, then $a' = a$ and $v = \pi_S(s^{k_t}x_t)$.
- If $x_t = \epsilon$ and $k_t > 1$, then $a' = a$ and $v = \pi_S(s^{k_t})$.
- If $x_t = \epsilon$ and $k_t = 1$, then $a' = \pi_S(sa)$ and $v = \epsilon$.

Remark 4.2. The operations $\mathrm{Collapse}_{S,i}$ and $\mathrm{Falling}_S(s)$ defined here are similar to those presented in [57]. However, they slightly differ in order to facilitate the use of vectorial circuits that use only operations that propagate information from left to right. The most important difference lies in the fact that the operation $\mathrm{Falling}_S(s)$ presented in [57] has a smaller domain, as it is defined only on the words which last letter is in $Q_{i+1}(S)$, where i is the $\mathcal{J}$-depth of the element s.

Using the programs given above, we define the *waterfall evaluation programs*:

Definition 4.8

A *waterfall evaluation program* is an evaluation program built with the operations $\mathrm{Collapse}_{S,i}$ and $\mathrm{Falling}_{Q_i(S)}(s)$ for all $1 \leq i \leq d$ and $s \in D_i(S)$.

By applying successively $\mathrm{Falling}_S(s)$ for each element $s \in Q_i(S)$, we can obtain a word in $Q_{i+1}(S)^* \times (S \cup \{\epsilon\})$. Now, we can build the evaluation program π_S using waterfall evaluation programs.

Lemma 4.4

Let S be a semigroup of $\mathcal{J}$-depth d. The evaluation program π_S is equal to a waterfall evaluation program obtained by composing $O(|S|)$ evaluation programs among $\mathrm{Collapse}_{S,i}$ and $\mathrm{Falling}_{Q_i(S)}(s)$ for all $1 \leq i \leq d$ and $s \in D_i(S)$.

Proof. For any integer i such that $1 \leq i \leq d$, we define $O_i = (s_1, \ldots, s_k)$ to be any enumeration of $D_1(Q_i(S))$. Given such an enumeration, we define as follows the operation $\mathrm{Falling}_{Q_i(S)}[O_i]$:

$$\mathrm{Falling}_{Q_i(S)}[O_i] = \mathrm{Falling}_{Q_i(S)}(s_k) \circ \cdots \circ \mathrm{Falling}_{Q_i(S)}(s_1)$$

We define an intermediate partial evaluation program f_i which is a restriction of π_S to the domain $Q_i(S)^+ \times (S \cup \{\epsilon\})$. In symbols, for any word $u \in Q_i(S)^+ \cup S$, $f_i(u) = \pi_S(u)$. Note that f_1 is equal to π_S on any word of S^+. We prove by induction over $j = d - i$ with i ranging from 0 to $d-1$ that

$$f_j = f_{j+1} \circ \mathrm{Falling}_S[O_j] \circ \mathrm{Collapse}_{S,j}$$

with the convention that f_{d+1} is the identity. The base case (f_{d+1}) being fixed, we only need to prove the result by induction on i for $0 \leq i \leq d-1$. We remark that the image of $\mathrm{Collapse}_{S,j}$ produces a word $v = v'a$, where v' is a $\mathcal{J}$-maximal falling word and $a \in (S \setminus Q_i(S)) \cup \{\epsilon\}$. Then, the application of $\mathrm{Falling}_{Q_i(S)}(s_j)$ on v produces new elements which are only in $Q_{i+1}(S)$,

except for the last letter, and removes all occurrences of the element s_j, except for the last letter. Hence, by applying $\text{Falling}_{Q_i(S)}[O_i]$ we obtain a word in $Q_{i+1}(S)^* \times (S \cup \{\epsilon\})$. Since $Q_{d+1}(S)$ is empty, the last word obtained contains only one letter, which is by definition equal to the product of the input word. □

Note that, as an incidental optimization, it is possible to replace the last sequence of calls to Falling by LProd, since the last Collapse necessarily returns either one element in S, or two elements, the first being in $D_d(S)$, and the second in S.

Remark 4.3. Waterfall evaluation programs have some resemblance with the *factorizations forest* of Simon [10]. Indeed, our programs create a factorization forest for each word they are applied to. Moreover, the proof of the factorization forests theorem uses an induction on $\mathcal{J}$-classes, as we do for our programs. However, they are not quite the same. A waterfall evaluation program can be applied to any word on the right alphabet, whereas the factorization forest theorem proves the existence of a forest of bounded depth for a fixed word. This theorem is used to prove the existence of a formula corresponding to a given semigroup in two cases: the classes $\mathcal{B}\Sigma_1$ and Σ_1. To our knowledge, there are no proofs for the class **DA** or the class of aperiodic semigroups. Lastly, such proofs amount to consider all the formulas of some quantifier depth that depends on the forest depth, a technique that resembles Wilke's proof for **DA** [77, Corollary 1] and also gives awful upper bounds.

4.1.3 Sweeping evaluation programs

Now, we introduce an evaluation program which processes words in a more lateral fashion – from left to right or right to left. In this part, S is a semigroup of $\mathcal{J}$-depth d.

In the proofs of section 4.3, we will perform evaluations that produce left (resp. right) $\mathcal{J}$-constant prefixes (resp. suffixes). We define those programs depending on the $\mathcal{J}$-depth of the semigroup that is considered. First, we introduce a *left splitting* higher order operation that applies an evaluation program over a prefix of the input word defined by some constraints over Green's relations. Formally, for any integer $i \leq d$, we define the operation $\text{LSplit}_{S,i}$ as follows. Consider a word $u = s_0 \cdots s_k \in S^+$ and an evaluation program P (see definition 4.2) defined at least on all left $\mathcal{J}$-constant words of depth i. If s_0 is not of $\mathcal{J}$-depth i, we set $\text{LSplit}_{S,i}\langle P\rangle(u) = u$. Otherwise, there exist two uniquely defined words $p \in S^+$, $s \in S^*$ such that $u = ps$, where p is left $\mathcal{J}$-constant and either s is empty or, if we denote by $x \in S$ its first letter, $\pi_S(px) <_{\mathcal{J}} \pi_S(p)$. In this case, $\text{LSplit}_{S,i}\langle P\rangle(u) = P(p)s$. We define similarly the symmetric operation $\text{RSplit}_{S,i}$. Finally, we define the partial function JProd_S, that is the restriction of π_S over words that are $\mathcal{J}$-constant. Formally, JProd_S is defined only on $\mathcal{J}$-constant words and, given $u \in S^+$ such a word, $\text{JProd}_S(u) = \pi_S(u)$.

Definition 4.9

A *sweeping evaluation program* is an evaluation program built with the following operations: LProd_S, RProd_S (see definition 4.3), the partial function JProd_S, and the higher order operations $\text{LSplit}_{S,i}$ and $\text{RSplit}_{S,i}$ for any integer i such that $1 \leq i \leq d$.

Lemma 4.5

Let S be a semigroup. There exists a sweeping evaluation program computing π_S. More-

over, there exists such a program that is equal to the composition of $O(2^d)$ operations.

To prove Lemma 4.5, we introduce an intermediate operation. Let i be an integer such that $1 \leq i \leq d$. We denote by $P_{i,l}$ the *left sweeping evaluation program* of $\mathcal{J}$-depth i, which computes π_S on the maximal prefix of $\mathcal{J}$-depth at most i (included). Formally, given a word $u = s_0 \cdots s_k \in S^+$, $P_{i,l}(u)$ is equal to u if s_0 is of $\mathcal{J}$-depth strictly greater than i. Otherwise, there exist $p \in S^+$, $q \in S^*$ such that $u = pq$, where either q is empty, or $x \in S$ is its first letter and then $\pi_S(p)$ is of $\mathcal{J}$-depth at most i and $\pi_S(px)$ is of $\mathcal{J}$-depth strictly greater than i. In this case, $P_{i,l}(u) = \pi_S(p)q$. We define symmetrically $P_{i,r}$, the *right sweeping evaluation program* of $\mathcal{J}$-depth i.

The next lemma allows to conclude the proof of Lemma 4.5 since $\pi_S = P_{d,l} = P_{d,r}$.

Lemma 4.6

For any integer i such that $1 \leq i \leq d$, there exist sweeping evaluation programs computing $P_{i,l}$ and $P_{i,r}$.

Proof. We will prove by induction on the $\mathcal{J}$-depth i that we can implement a left (resp. right) sweeping evaluation program $P_{i,l}$ (resp. $P_{i,r}$). In this proof, we consider a word $u = s_0 \cdots s_{k-1}$ over S. For the base case, we first suppose that $i = 1$, i.e. we consider maximal $\mathcal{J}$-classes. Thus, if s_0 is of $\mathcal{J}$-depth 1, we will compute the product of the unique prefix $s_0 \cdots s_p$ of u such that $\pi_S(s_0 \cdots s_p)\mathcal{J}s_0$ and $\pi_S(s_0 \cdots s_{p+1}) <_{\mathcal{J}} s_0$. If s_{p+1} does not exist, we want to compute $\pi_S(u)$. Note that $s_0 \cdots s_p$ is $\mathcal{J}$-constant, hence we can apply JProd_S to it. Thus, we can compute the base case $P_{1,l}$ using the program $\mathrm{LSplit}_{S,1}\langle \mathrm{JProd}_S \rangle$. Note that this program is well defined since JProd_S is in particular defined on all $\mathcal{J}$-constant words of depth 1, which are exactly the left $\mathcal{J}$-constant words of depth 1. Symmetrically, $P_{1,r} = \mathrm{RSplit}_{S,1}\langle \mathrm{JProd}_S \rangle$. To prove the induction case, we will rely on the following fact (see 1.2.3 for the definition of a left $\mathcal{J}$-constant word):

Fact 4.1. *For any left $\mathcal{J}$-constant word $v \in S^+$ of $\mathcal{J}$-depth i, the word $\mathrm{RProd}_S \circ P_{i-1,r}(v)$ is $\mathcal{J}$-constant.*

Proof. The result is obtained from the fact that the last element of $w = \mathrm{RProd}_S \circ P_{i-1,r}(v)$ is necessarily of $\mathcal{J}$-depth i. Indeed, by definition, the word $x = P_{i-1,r}(v)$ is such that the product of its last two elements (if there are at least two elements) is at least of $\mathcal{J}$-depth i. Since we supposed that v is left $\mathcal{J}$-constant of $\mathcal{J}$-depth i, it is guaranteed that this product is defined and is exactly of $\mathcal{J}$-depth i. Thus, both the first and last elements of w are of $\mathcal{J}$-depth i, as well as $\pi_S(w)$. Thus the product of any prefix or suffix of w will be of $\mathcal{J}$-depth i, and in the same $\mathcal{J}$-class as the first and last elements of w, which corresponds to the definition of $\mathcal{J}$-constant. □

For the induction step, we assume to have sweeping evaluation programs $P_{i,r}$ and $P_{i,l}$ for any integer $i < d$. We prove the result for $P_{i+1,r}$ and $P_{i+1,l}$. These two cases being symmetrical, we only show the result for $P_{i+1,l}$. Let $v = \mathrm{LProd}_S \circ P_{i,l}(u)$. If $|P_{i,l}(u)| \neq 1$, we have necessarily that the first letter of v is of $\mathcal{J}$-depth strictly greater than i. Otherwise $v = P_{i,l}(u) = \pi_S(u)$. We are going to split v with respect to the $\mathcal{J}$-depth i and apply the program $E = \mathrm{JProd}_S \circ \mathrm{RProd}_S \circ P_{i,r}$ to the prefix. Indeed, after the split, and thanks to the previous Fact, we can apply JProd_S over the factor $\mathrm{RProd}_S \circ P_{i,r}(p)$, where p is the prefix obtained after the split. Indeed, this factor is $\mathcal{J}$-constant. To conclude, $P_{i+1,l} = \mathrm{LSplit}_{S,i+1}\langle E \rangle \circ \mathrm{LProd}_S \circ P_{i,l}$.

Thus, each P_i is defined using $O(2^i)$ operations. □

It is unclear whether this exponential behavior is avoidable or not. In this proof, the exponential factor comes from the need to determine both the $\mathcal{R}$ and $\mathcal{L}$-classes of the last element computed before falling in another $\mathcal{J}$-class. With this line of reasoning, it is hard to avoid the exponential factor. However, it might be possible to design a class of vectorial circuits equivalent to ours and that would only require an polynomial number of gates. As of the writing of this thesis, this is still an open problem.

4.2 Compilation procedure of semigroups in Ap

In this section, we prove the following theorem by constructing circuits verifying the property:

Theorem 4.1

Let S be an aperiodic semigroup of $\mathcal{J}$-depth d. We can construct an ADD-vectorial circuit of size $O(d|S|^3)$ that computes the operation π_S.

To prove Theorem 4.1, we consider a fixed aperiodic semigroup S, and we denote by d its $\mathcal{J}$-depth. Thanks to Lemma 4.4, it is sufficient to provide, for any integer $i \leq d$, ADD-vectorial circuits computing the operations $\text{Collapse}_{S,i}$ and $\text{Falling}_S(s)$ for any $s \in Q_i(S)$ over some vectorial encoding of any partial evaluation of a word. Once we have those, we proceed as in Lemma 4.4, in which we prove that, for any integer i such that $1 \leq i \leq d$, $\pi_{Q_i(S)}$ is equal to the composition of $\pi_{Q_{i+1}(S)}$ and $O(|S|)$ operations among $\text{Collapse}_{S,i}$ and $\text{Falling}_S(s)$ (for any $s \in D_i(S)$). More precisely, the evaluation program for π_S uses $|S|$ operations of the form $\text{Falling}_S(s)$ (where $s \in S$) and d operations of the form $\text{Collapse}_{S,i}$ (recall that d is the $\mathcal{J}$-depth of S). Now, we can obtain the result by using the lemmas presented in this section.

4.2.1 Vectorial encoding of a partial evaluation of a word

Our sweeping and waterfall evaluation programs perform operations on partial evaluations of a word, so we need to provide an explanation on how we encode these partial evaluations. From now on, a partial evaluation will always designate a partial evaluation of a word, usually the initial word that needs to be processed with the evaluation program. Informally, a vectorial representation of a partial evaluation is a set of vectors, each vector corresponding to a semigroup element. The size of the vectors is equal to the size of the initial word on which we apply the evaluation program. We need the vectors we employ to always have the same size throughout the execution, so our definition needs to be more general than indicator vectors. Each bit set to one denotes the presence of the element in the partial evaluation, and the order of the bits set to one determines the order of the letters in the word.

As in the case of characteristic functions of letters, in the vector encoding of a word, vectors in an encoding do not overlap, however their union may not cover all the positions. More formally, a *vectorial encoding of a partial evaluation* is a mapping $\mathbf{c}\colon S \to \{0,1\}^n$, for some integer $n \geq 1$, such that we have the following constraint: for any $s, t \in S$ such that $s \neq t$, we have $\mathbf{c}(s) \wedge \mathbf{c}(t) = \mathbf{0}$. Note that, by definition, a word is a partial evaluation of itself, so $\text{enc}(u)$ is a vectorial encoding of a partial evaluation, with the additional property that $\bigvee_{a \in S} \mathbb{1}_a(u) = \mathbf{1}$.

Given such a function $\mathbf{c}$ outputting vectors of dimension n, we can interpret it as a word of length at most n by respecting the order of appearance of the bits. Formally, a word $s_0 \cdots s_k \in S^{\leq n}$ is represented by a vectorial encoding $\mathbf{c}$ of dimension n if, for every index $j \leq k$, there exists some integer i such that $0 \leq i < n$, $\mathbf{c}(s_j)_i = 1$ and $|\{t \in \mathbb{N} \mid t < i \text{ and } \bigvee_{s \in S} \mathbf{c}(s)_t = 1\}| = j - 1$.

We introduce two vectors that we will use extensively in our circuits: given S a semigroup and $\mathbf{c}$ a vectorial encoding of a partial evaluation, we define the universe vector $\mathbf{U} = \bigvee_{s \in S} \mathbf{c}(s)$. We also define the vector marking the end of the vectors: $\mathbf{End} = \neg\mathrm{MSB}(\mathbf{1})$.

We use only circuits of the following form: the input is a vectorial encoding of a partial evaluation $\mathbf{c}\colon S \to \{0,1\}^n$ (for some $n \in \mathbb{N}^+$) representing some word $u \in S^{\leq n}$, and the output is a vectorial encoding $\mathbf{out}\colon S \to \{0,1\}^n$ representing the partial evaluation of u obtained by applying the operation corresponding to the circuit. To prove our theorems, we construct vectorial circuits using this encoding that implement the partial evaluation functions we need.

Example 4.4. Consider the semigroup $S = \{a, b, c\}$ with the inner multiplication as follows: $\pi_S(ac) = \pi_S(ca) = c$, $\pi_S(bc) = \pi_S(cb) = c$, $\pi_S(aa) = b$. The word $w = aaabac$ has the vectorial encoding $\mathbf{a} = 111010$, $\mathbf{b} = 000100$, $\mathbf{c} = 000001$. With our circuits, if we multiply the last two letters of w using this encoding, we will obtain the encoding $\mathbf{a} = 111000$, $\mathbf{b} = 000100$, $\mathbf{c} = 000001$. As expected, it represents the word $w' = aaabc$: the first four elements of the vectors are the same and still represent the word $aaab$, the fifth element is a 0 in all vectors so it represents no letter, and the sixth element is a 1 in $\mathbf{c}$, so it represents the letter c. Since we started from a word of length 6, the length of the vectors is still 6 but some of the indices do not represent a letter anymore.

4.2.2 Addition lemma

The circuits in this section heavily rely on the binary addition. Here, we define a new operation that encapsulates the use we make of the binary addition and gives a better intuition on what our circuits do.

Let $\mathbf{x}, \mathbf{y}$ be two disjoint vectors of dimension n and $\mathbf{z}$ a vector of dimension n that contains both $\mathbf{x}$ and $\mathbf{y}$. We denote by $\mathbf{v} = \mathrm{Successor}\,(\mathbf{x}, \mathbf{y}, \mathbf{z})$ the vector such that for all $i < n$, $\mathbf{v}_i = 1$ if and only if $\mathbf{x}_i = 1$, there exists $j < i$ such that $\mathbf{y}_j = 1$ and, for all $k \in \mathbb{N}$ such that $j < k < i$, $\mathbf{z}_k = 0$. In other words, $\mathrm{Successor}\,(\mathbf{x}, \mathbf{y}, \mathbf{z})$ indicates the positions marked in $\mathbf{x}$ that follow a marked position of $\mathbf{y}$, with no other position marked by $\mathbf{z}$ in-between.

Lemma 4.7: Addition lemma

Let $\mathbf{x}, \mathbf{y}$ be two disjoint vectors of dimension n and $\mathbf{z}$ a vector of dimension n whose set of indices $\{i \mid \mathbf{z}_i = 1\}$ contains both $\{i \mid \mathbf{x}_i = 1\}$ and $\{i \mid \mathbf{y}_i = 1\}$. Then, we have $\mathrm{Successor}\,(\mathbf{x}, \mathbf{y}, \mathbf{z}) := (\mathbf{y} + (\mathbf{y} \vee \neg\,\mathbf{z})) \wedge \mathbf{x}$

This lemma has a rather tedious proof. So as to relieve the reader from checking its details, we provide a formalization and a proof of this Addition Lemma in Coq [56]. Here is a sketch of the proof: it consists of two steps. First, we show that vectorial circuits built with the Next-Until modality of LTL, aka XU, and logical gates, are equivalent to circuits built with addition and logical gates. This equivalence is proved by constructing circuits based on XU and logical gates to encode addition and circuits based on addition and logical gates to encode XU. The intuition behind this equivalence is that carry propagation and XU both propagate information sideways. Second, encoding XU in first order-logic and using the previous equivalence allows us to relate the computation of the circuit of Lemma 4.7 to the specification of $\mathrm{Successor}\,(\mathbf{x}, \mathbf{y}, \mathbf{z})$ and prove the relation correct.

Example 4.5. Consider the vectors $\mathbf{x} = 01010110101$, $\mathbf{y} = 00100100000$, and $\mathbf{z} = 11110110111$. Note that $\mathbf{z}$ contains both $\mathbf{x}$ and $\mathbf{y}$, but is different from $\mathbf{x} \vee \mathbf{y}$. The vector $\mathrm{Successor}\,(\mathbf{x}, \mathbf{y}, \mathbf{z})$ is computed as in the following table:

$\mathbf{x}$	010101101001
$\mathbf{y}$	001001000100
$\mathbf{z}$	111101101101
$\neg\mathbf{z}$	000010010010
$\mathbf{y} \vee \neg\mathbf{z}$	001011010110
$\mathbf{y} + (\mathbf{y} \vee \neg\mathbf{z})$	000110110001
carry	000100100011
$(\mathbf{y} + (\mathbf{y} \vee \neg\mathbf{z})) \wedge \mathbf{x}$	000100100001

As claimed in Lemma 4.7, the vector $(\mathbf{y} + (\mathbf{y} \vee \neg\mathbf{z})) \wedge \mathbf{x}$ has occurrences of 1 at exactly all the indices that bear a 1 in $\mathbf{x}$ after another occurrence of 1 in $\mathbf{y}$, with no occurrence of 1 in $\mathbf{z}$ strictly between the two.

Remark 4.4. The formula given in the Addition Lemma is close to the one given for the primitive MatchStar in [14]. Indeed, that primitive takes two vectors $\mathbf{M} = m_0 \cdots m_{n-1}$ and $\mathbf{C} = c_0 \cdots c_{n-1}$, and returns a vector $\mathbf{V} = v_0 \cdots v_{n-1}$ of the same dimension such that $v_i = 1$ if and only if the position i is preceded by a sequence of 1s in the vector C, sequence that begins at an index j such that $m_j = 1$. In symbols:

$$m_i = 1 \vee \exists j < i, (m_j = 1 \wedge \forall k, j \leq k < i \Rightarrow c_k = 1)$$

The formula they use for that purpose is the following:

$$\mathbf{V} = (((\mathbf{M} \wedge \mathbf{C}) + \mathbf{C}) \oplus \mathbf{C}) \vee \mathbf{M}$$

That use of the binary addition is basically the same as in the Addition Lemma, the only difference being that MatchStar has a more specific purpose. With the Addition Lemma, it is possible to obtain an equivalent formula, but that formula would be much more complex, as it has not been handcrafted like MatchStar.

4.2.3 The Collapse$_S$ operation

This subsection proves the following lemma:

Lemma 4.8

For any aperiodic semigroup S of depth d and any integer $i \leq d$, we can compute Collapse$_{S,i}$ over any vectorial encoding of a partial evaluation in its domain with an ADD-vectorial circuit of size $O(|S|^3)$.

Consider a word $u = va$, with $v \in Q_i(S)^+$ and $a \in (S \backslash Q_i(S)) \cup \{\epsilon\}$. Let $w_0 v_0 \cdots v_t w_{t+1}$ be the $\mathcal{J}$-maximal decomposition of v and let $w'_0 v'_0 y_1 w''_1 \cdots v'_j y_{j+1} w''_{j+1} \cdots v'_t w_{t+1}$ be the refined decomposition defined in definition 4.6. Let $(\mathbf{c}(s))_{s \in S}$ be a vectorial encoding of u. Our goal is to compute a vectorial encoding $(\mathbf{out}(s))_{s \in S}$ of the partial evaluation $u' = w'_0 \pi_S(v'_0 y_1) w''_1 \cdots \pi_S(v'_j y_{j+1}) w''_{j+1} \cdots w''_t z$ defined in definition 4.6 from the encoding $(\mathbf{c}(s))_{s \in S}$. To do that, we begin by finding all the letters that belong to the infixes v'_j. By definition, each v'_j is exactly a maximal block which product is in $D_i(S)$ minus its first letter (since $x_j v'_j = v_j$). Thus, we will now mark the elements of those blocks.

For each pair (s,t) of elements in $D_i(S)$ such that $\pi_S(st)$ is in $D_i(S)$, we compute the vector marking each occurrence of t that is preceded by an s:

$$\mathbf{Stays}(s,t) = \text{Successor}\,(\mathbf{c}(t), \mathbf{c}(s), \mathbf{U})$$

where $\mathbf{U} = \bigvee_{s\in S} \mathbf{c}(s)$ is the universe vector. We then aggregate those vectors to obtain, for each element $t \in D_i(S)$, the vector marking the occurrences of t which product with the previous letter in the word u is in $D_i(S)$: we define the set $E_t = \{s \in D_i(S) \mid \pi_S(st) \in D_i(S)\}$, with which we can compute the desired vector:

$$\mathbf{Stays}(t) = \bigvee_{s\in E_t} \mathbf{Stays}(s,t)$$

We also compute the union of all those vectors: we define the set $R = \{t \in S \mid \exists s \in D_i(S), \pi_S(st) \in D_i(S)\}$, with which we can compute the desired vector:

$$\mathbf{Stays} = \bigvee_{s\in R} \mathbf{Stays}(s)$$

With this, we can find the limits of the maximal infixes which product is in $D_i(S)$, without counting the first element, which does not belong to v'_j. Note that we put that letter aside in case we need to use it to make the product of the previous infix fall in $Q_{i+1}(S)$. Thus, the vector **Stays** marks exactly the elements that belong to any of the infixes v'_j. Then, we mark the first letter of each v'_j. These letters are exactly the elements of S that are marked by **Stays** and such that the previous element marked by **U** is not marked by **Stays**. Thus, for each element $s \in D_i(S)$, we compute the vector which labels the first letter of each v_j with its value if it is an s:

$$\mathbf{SecondEl}(s) = \text{Successor}\,(\mathbf{Stays}(s), \mathbf{U} \wedge \neg\mathbf{Stays}, \mathbf{U})$$

We aggregate those vectors:

$$\mathbf{SecondEl} = \bigvee_{s\in D_i(S)} \mathbf{SecondEl}(s)$$

Then we mark the letters y_{j+1}, which are the elements just after the end of the infixes v'_j, and we label them by the last element of the corresponding v'_j. For each element $s \in D_i(S)$, we compute the vector marking each y_{j+1} if and only if the last letter of v'_j is an s:

$$\mathbf{BlockFall}(s) = \text{Successor}\,(\mathbf{U} \wedge \neg\mathbf{Stays}, \mathbf{Stays}(s), \mathbf{U})$$

We aggregate those results:

$$\mathbf{BlockFall} = \bigvee_{s\in D_i(S)} \mathbf{BlockFall}(s)$$

With this, we can compute the vector which marks the limits of the infixes $v'_j y_{j+1}$:

$$\mathbf{Limits} = \mathbf{SecondEl} \vee \mathbf{BlockFall}$$

Note that, if w_{t+1} is empty, this vector marks the first letter of v'_t, but not its last letter. However, if w_{t+1} is empty and $a \neq \epsilon$, then the letter a is marked by **BlockFall** as the letter

y_{t+1}, and thus it is marked by **Limits**.

With these vectors, we can compute all the products $\pi_S(v'_j y_{j+1})$. To this end, for each element $p \in Q_i(S)$, we want to define the set of triplets $(s,t,x) \in (D_i(S))^2 \times S$ such that, for any word $v \in S^+$ such that v begins by s, ends by t and is $\mathcal{J}$-constant, the product of the word vx is p. Note that we consider all the elements $p \in Q_i(S)$ because, if $a \neq \epsilon$ is marked by **Limits**, then we also compute the product $\pi_S(v'_t a)$. Thus, for any element $p \in Q_i(S)$, we define the set

$$F_p = \{(s,t,x) \in (D_i(S))^2 \times S \mid \pi_S(\alpha \cdot x) = p, \text{ where } \alpha = \mathcal{R}(s) \cap \mathcal{L}(t)\}$$

This set is exactly what we need since, thanks to Lemma 4.2, we know that the product of the infix v is only determined by s and t. Moreover, that product is not always equal to $\pi_S(st)$. Indeed, the infix v can be of size 1, in which case $s = t$ and there is only one occurrence to consider. In the case where $\pi_S(ss) \neq s$, the product of v is equal to s and not to $\pi_S(ss)$. However, since S is aperiodic, we know thanks to Proposition 1.2 that the element $\pi_S(st)$ is uniquely characterized by its $\mathcal{H}$-class, which is equal to $\mathcal{R}(s) \cap \mathcal{L}(t)$. Thus, the set F_p defines exactly the triplets that we need. Note that each triplet $(s,t,x) \in (D_i(S))^2 \times S$ can only appear in at most one set F_p, so the size of the union of all these sets is only $O(|D_i(S)|^2 * |S|) \leq O(|S|^3)$.

Then we use this to compute the products of the infixes. For each element $p \in Q_{i+1}(S)$, we label each y_{j+1} by p if and only if there exists a triplet $(s,t,y_{j+1}) \in F_p$ such that the first letter of v'_j is an s (which can be determined using **Stays**(s)) and the last letter of v'_j is a t (and thus y_{j+1} is marked by the vector **BlockFall**(t)):

$$\mathbf{Prod}(p) = \bigvee_{(s,t,x) \in F_p} \text{Successor}\left(\mathbf{BlockFall}(t), \mathbf{SecondEl}(s), \mathbf{Limits}\right) \wedge \mathbf{c}(x)$$

Note that the union of all the vectors **Prod**(p) contains exactly all the elements obtained with the products $\pi_S(v'_j y_{j+1})$. With these vectors, we will be able to compute a vectorial encoding of the word $\text{Collapse}_{S,i}(u)$. Before computing the output vectors, we consider the case where w_{t+1} is empty and $a = \epsilon$. This case has not been treated yet since it implies that there is no letter after v'_t which allows us to compute the product as explained above. However, the first letter of v'_t has been marked in **SecondEl**. Thus, we only need to mark the last letter of the word v'_t. For each element $s \in D_i(S)$, we label by s the last element of u if and only if $w_{t+1} = a = \epsilon$ and v'_t begins with an s:

$$\mathbf{FirstLeftover}(s) = \text{Successor}\left(\mathbf{End}, \mathbf{SecondEl}(s), \mathbf{Limits} \vee \mathbf{End}\right) \wedge \neg \bigvee_{t \in S \setminus Q_i(S)} \mathbf{c}(t)$$

Then we compute the product of that suffix $(\pi_S(v'_t))$. For each element $p \in D_i(S)$, we define the set $G_p = \{(s,t) \in (D_i(S))^2 \mid \pi_S(\alpha) = p, \text{ where } \alpha = \mathcal{R}(s) \cap \mathcal{L}(t)\}$ and we compute the vector **Prod**(p) which is equal to **End** if and only if $w_{t+1} = a = \epsilon$, v'_t starts with some s and ends with some t such that (s,t) is in G_p:

$$\mathbf{Prod}(p) = \bigvee_{(s,t) \in G_p} \mathbf{FirstLeftover}(s) \wedge \mathbf{c}(t)$$

Now we have everything we need to compute the output vectors. To do this, we remove every element marked by **Stays** (the letters of every v'_j) and **BlockFall** (every y_{j+1}), and we add the

elements computed in the vectors $\mathbf{Prod}(s)$ (the elements $\pi_S(v'_j y_{j+1})$ and $\pi_S(v'_t)$ if $w_{t+1} = a = \epsilon$). For each element $s \in S$:

$$\mathbf{out}(s) = \mathbf{Prod}(s) \vee (\mathbf{c}(s) \wedge \neg(\mathbf{Stays} \vee \mathbf{BlockFall}))$$

4.2.4 The Falling$_S(s)$ operation

This subsection proves the following lemma:

Lemma 4.9

Let S be an aperiodic semigroup of $\mathcal{J}$-depth d. For any element $s \in D_1(S)$, we can compute Falling$_S(s)$ over any vectorial encoding of a partial evaluation in its domain with an ADD-vectorial circuit of size $O(d|S|)$.

Before proving lemma 4.9, we prove the following technical lemma.

Lemma 4.10

Let S be an aperiodic semigroup of $\mathcal{J}$-depth d, an integer $i \leq d$, and an element $s \in D_i(S)$. Let $u = u'a$ be a word over $Q_i(S)$. Suppose that u' is a $\mathcal{J}$-maximal falling word. Consider the s-decomposition of u, written as $w_0 s^{k_1} x_1 w_1 \cdots w_{t-1} s^{k_t} x_t w_t$. Then, we can use an ADD-vectorial circuit of size $O(|S|)$ which takes the vectorial encoding of u as input and produces a vectorial encoding of the word $w_0 s^{k_1-1} \pi_S(sx_1) \cdots s^{k_{t-1}} \pi_S(sx_{t-1}) w_t s^{k_t} \pi_S(sy) v$, where y is the first letter of the infix $w_{t+1}a$ and $yv = w_{t+1}a$.

The idea of this technical lemma is the following: by applying the lemma, we compute each product of the form st, where t is an element distinct from s, and if the last two letters are occurrences of s, we multiply them. Thus, we decrease by one the number of occurrences of s in each infix of the form $s \cdots s$.

Let $(\mathbf{c}(t))_{t \in S}$ be a vectorial encoding of u. We begin by marking the elements that are preceded by an occurrence of s and are not occurrences of s.

$$\mathbf{Last} = \text{Successor}\,(\mathbf{U} \wedge \neg\mathbf{c}(s), \mathbf{c}(s), \mathbf{U})$$

We separate these elements depending on their value: for each element $t \in S \setminus s$, we set

$$\mathbf{Last}(t) = \mathbf{Last} \wedge \mathbf{c}(t)$$

We also have to deal with the case where the last two letters of u are occurrences of s. In that case, we want to multiply them, but the last occurrence of s is not marked by **Last**. Thus, we mark the last element of u if and only if it is an occurrence of s preceded by another occurrence of s:

$$\mathbf{IsEnd} = \text{Successor}\,(\mathbf{End} \wedge \mathbf{c}(s), \mathbf{c}(s), \mathbf{U})$$

In order to remove the occurrences of s that will be used, we also mark the occurrences of s that are not preceded by another occurrence of s.

$$\mathbf{First} = \text{Successor}\,(\mathbf{c}(s), \mathbf{U} \wedge \neg\mathbf{c}(s), \mathbf{U}) \wedge \neg\mathbf{End}$$

These are not exactly the occurrences that should be removed, but each one belongs to the same block of occurrences of s as an occurrence that is used in one of the products we will perform, so removing them is sufficient to obtain the result we want. Moreover, by definition, we will still obtain a valid vectorial encoding of the partial evaluation of u that we want to obtain.

Now we can remove the occurrences of s that will be used in the products:

$$\mathbf{rm}(s) = \mathbf{c}(s) \wedge \neg\mathbf{First} \wedge \neg\mathbf{IsEnd}$$

Then we can remove the elements that are preceded by an occurrence of s: for each element $t \in S \setminus s$, we set

$$\mathbf{rm}(t) = \mathbf{c}(t) \wedge \neg\mathbf{Last}(t)$$

Finally, we perform the products: for each element $p \in S$, we define the set $P_p = \{t \in S \setminus s \mid \pi_S(st) = p\}$ of elements t such that p is equal to $\pi_S(st)$. With this, we can produce the vectors $\mathbf{out}(p)$:

$$\mathbf{out}(p) = \mathbf{rm}(p) \vee \bigvee_{t \in P_p} \mathbf{Last}(t)$$

Note that each element $t \in S \setminus s$ appears in at most one set P_p, and that consequently the computation of all the vectors $\mathbf{out}(p)$ requires only a circuit of size $O(|S|)$. Finally, we add the product of the last two elements of u if they are both occurrences of s. To do so, we consider the element $p = \pi_S(ss)$ and we update $\mathbf{out}(p)$ if necessary:

$$\mathbf{out}(p) = \mathbf{out}(p) \vee \mathbf{IsEnd}$$

This gives us a vectorial encoding of the partial evaluation we wanted for this technical lemma.

With this technical lemma, we can prove lemma 4.9. Let u be a word of S^+ and the set $(\mathbf{c}(t))_{t \in S}$ be a vectorial encoding of u. Using lemma 4.10, we can prove the result quite easily by repeating the lemma a fixed number of times. Indeed, by applying lemma 4.10, we can reduce by 1 the size of each of the blocks of occurrences of s. Moreover, the semigroup S is aperiodic, so by Proposition 1.2 there necessarily exists an integer ω_s such that $\pi_S(s^{\omega_s+1}) = \pi_S(s^{\omega_s})$ (s^{ω_s} is the idempotent power of s). Thus, if we apply lemma 4.10 ω_s times, the only occurrences of s that will be left will be either any letter s that was originally followed by at least ω_s other occurrences of s, or the last letter of the word. Since $\pi_S(s^{\omega_s+1}) = \pi_S(s^{\omega_s})$, we can just forget the occurrences that are not the last letter without changing anything else. We denote by $(\mathbf{omega}(t))_{t \in S}$ the vectorial encoding of the decomposition of u obtained after applying Lemma 4.10 sequentially ω_s times. Since the products that we could compute from now on will only remove occurrences of s without changing the other elements, we can now remove all the occurrences of s, except for the last letter, that are left in the new partial evaluation of u. Thus, we can set

$$\mathbf{out}(s) = \mathbf{omega}(s) \wedge \mathbf{End}$$

and, for each element $t \neq s$,

$$\mathbf{out}(t) = \mathbf{omega}(t)$$

then we obtain a vectorial encoding of the partial evaluation that we wanted.

Note that, for any $t \in S$, we have $\omega_t \leq d$, where d in the $\mathcal{J}$-depth of S, so we apply lemma 4.10 at most d times.

With this, we can conclude that π_S can be computed with an ADD-vectorial circuit of size

$O(d|S|^3)$.

4.3 Compilation procedure of semigroups in DA

In this section, we prove the following theorem by constructing some circuits verifying the property:

Theorem 4.2

Let S be a semigroup in **DA** of $\mathcal{J}$-depth d. We can construct a Sweeping-vectorial circuit of size $O(2^d|S|^2)$ that computes the operation π_S.

To prove Theorem 4.2, we consider a fixed semigroup $S \in \mathbf{DA}$, and we denote by d its $\mathcal{J}$-depth. Thanks to Lemma 4.5, it is sufficient to provide Sweeping-vectorial circuits computing the base operations over some vectorial encoding of any partial evaluation of a word. Those operations are LProd_S, RProd_S, JProd_S and, for each integer i such that $1 \leq i \leq d$, the operations $\text{LSplit}_{S,i}$ and $\text{RSplit}_{S,i}$. Once we have those, we proceed as in Lemma 4.5, in which we prove that π_S is equal to the composition of $O(2^d)$ operations among JProd_S, LProd_S, RProd_S and $\text{LSplit}_{S,i}\langle E\rangle$ (for any integer i such that $1 < i \leq d$ and some sweeping program E composed of the same operations, that are taken into account in the $O(2^d)$). To prove this, we have $\pi_S = P_{d,l} = P_{d,r}$ where, for each integer i such that $1 \leq i \leq d$, the program $P_{i,l}$ computes π_S on the maximal prefix of $\mathcal{J}$-depth at most i (included), and the symmetric program $P_{i,r}$ which acts on suffixes instead of prefixes. Our proof constructs Sweeping-vectorial circuits for those programs, by induction on the depth i. Now, we can obtain the result by using the lemmas presented in this section. The circuits presented in this section rely on the vectorial encodings of words presented in section 4.2.1.

4.3.1 An intermediary operation

Before proving the Lemmas necessary for Theorem 4.2, we define some intermediary operations that will allow us to decompose the functions that we want to compute. These intermediary operations will all be of the same form: for any integer $i \in \mathbb{N}$, we define the operation $\text{Value}_{i,S}$ that identifies the i^{th} semigroup element that occurs in the word represented by the input vectors. Formally, we define this operation as follows:

Definition 4.10

Let S be a semigroup in **DA** and let $(\mathbf{c}(s))_{s\in S}$ be a vectorial encoding of some word $u \in S^+$. For each element $s \in S$ and integer $i \in \mathbb{N}$, we define the vector $\text{Value}_{i,S}(s)$ as follows:

- $\text{Value}_{i,S}(s) = \mathbf{1}$ if and only if there exists an integer j such that the j^{th} element of $\mathbf{c}(s)$ is a 1 and the position j is the i^{th} position of the vector $\mathbf{U}$ to hold a 1.
- Otherwise, $\text{Value}_{i,S}(s) = \mathbf{0}$.

Example 4.6. Consider the word $w = abbacb$, where a, b and c are three distinct elements of some semigroup S. Then, $\text{Value}_{0,S}(a) = \mathbf{1}$, $\text{Value}_{2,S}(b) = \mathbf{1}$, and $\text{Value}_{3,S}(c) = \mathbf{0}$.

Lemma 4.11

For any integer $i \geq 1$, we can compute the function $\text{Value}_{i,S}$ over any vectorial encoding of a partial evaluation with a Sweeping-vectorial circuit of size $O(i + |S|)$.

Proof. Given a set of input vectors $I = (\mathbf{c}(s))_{s\in S}$, $\text{Value}_{i,S}(I)$ is a set of vectors $(\mathbf{out}(s))_{s\in S}$ such that, for each element $s \in S$, $\mathbf{out}(s)$ is computed as follows. We begin by removing the first $i-1$ bits set to 1 in the union of the inputs by defining the following vectors:

$$\mathbf{U}_0 = \mathbf{U}$$

$$\forall j < i-1, \mathbf{U}_{j+1} = \text{LSB}(\mathbf{U}_j)$$

Then, for each $x \in S$, we set to 0, in $\mathbf{c}(x)$, the $i-1$ first bits set to 1 in the union of the inputs:

$$\mathbf{rm}(x) = \mathbf{c}(x) \wedge \mathbf{U}_{i-1}$$

Now, to detect the element associated to the i^{th} bit set to 1 in $\mathbf{U}$, we only need to detect the value associated to the first bit set to 1 in $\bigvee_{x\in S} \mathbf{rm}_x$, which is done as follows: for any $s \in S$, we compute the vector $\mathbf{out}(s)$ that is full of ones if and only if the position of the first bit set to 1 in $\mathbf{U}_{i-1}$ (that is the union of the vectors $\mathbf{rm}(x)$) is set to 1 in the vector $\mathbf{rm}(s)$.

$$\mathbf{out}(s) = \neg\text{NotZero}(\text{pref-}\vee(\mathbf{rm}(s)) \oplus \text{pref-}\vee(\mathbf{U}_{i-1}))$$

□

Example 4.7. Let's go back to example 4.6: consider the word $w = abbacb$, where a, b and c are three distinct elements of some semigroup S. Suppose that the vectorial encoding of w is such that $\mathbf{c}(a) = 1000100$, $\mathbf{c}(b) = 0011001$ and $\mathbf{c}(c) = 0000010$ (the encoding includes an empty space after the first occurrence of a). Then the vector $\mathbf{U}$ is the union of these three vectors, and is equal to 1011111.

In order to compute $\text{Value}_{0,S}(a)$, we consider the vectors $\mathbf{U}_0 = \mathbf{U}$ and $\mathbf{rm}(a) = \mathbf{c}(a)$. Then, we verify that the first occurrence of 1 in $\mathbf{rm}(a)$ is in the same position as the first occurrence of 1 in $\mathbf{U}_0$, by computing the following vectors:

$\mathbf{rm}(a)$	1000100
$\mathbf{U}_0$	1011111
pref-$\vee(\mathbf{rm}(a))$	1111111
pref-$\vee(\mathbf{U}_0)$	1111111
pref-$\vee(\mathbf{rm}(a)) \oplus$ pref-$\vee(\mathbf{U}_0)$	0000000
NotZero(pref-$\vee(\mathbf{rm}(a)) \oplus$ pref-$\vee(\mathbf{U}_0)$)	0000000
$\neg$NotZero(pref-$\vee(\mathbf{rm}(a)) \oplus$ pref-$\vee(\mathbf{U}_0)$)	1111111

The last vector is the value of $\text{Value}_{0,S}(a)$, and is equal to $\mathbf{1}$, which indicates that a is the first letter of the word w. Similarly, to compute $\text{Value}_{2,S}(b)$, we consider the vectors $\mathbf{U}_2 = \text{LSB}(\text{LSB}((\mathbf{U}))) = 0001111$ and $\mathbf{rm}(b) = \mathbf{c}(b) \wedge \mathbf{U}_2 = 0001001$. Then, we verify that the first occurrence of 1 in $\mathbf{rm}(b)$ is in the same position as the first occurrence of 1 in $\mathbf{U}_2$, by computing the following vectors:

$\mathbf{rm}(b)$	0001001
$\mathbf{U}_2$	0001111
pref-$\vee(\mathbf{rm}(b))$	0001111
pref-$\vee(\mathbf{U}_2)$	0001111
pref-$\vee(\mathbf{rm}(b)) \oplus$ pref-$\vee(\mathbf{U}_2)$	0000000
NotZero(pref-$\vee(\mathbf{rm}(b)) \oplus$ pref-$\vee(\mathbf{U}_2)$)	0000000
$\neg$NotZero(pref-$\vee(\mathbf{rm}(b)) \oplus$ pref-$\vee(\mathbf{U}_2)$)	1111111

We obtain $\text{Value}_{2,S}(b) = \mathbf{1}$, which indicates that the third letter of w is a b. Finally, to compute $\text{Value}_{3,S}(c)$, we consider the vectors $\mathbf{U}_3 = \text{LSB}((\mathbf{U}_2))$ =0000111 and $\mathbf{rm}(c) = \mathbf{c}(c) \wedge \mathbf{U}_3 =$ 0000010. Then, we verify that the first occurrence of 1 in $\mathbf{rm}(c)$ is in the same position as the first occurrence of 1 in $\mathbf{U}_3$, by computing the following vectors:

$\mathbf{rm}(c)$	0000010
$\mathbf{U}_3$	0000111
pref-$\vee(\mathbf{rm}(c))$	0000011
pref-$\vee(\mathbf{U}_0)$	0000111
pref-$\vee(\mathbf{rm}(b)) \oplus$ pref-$\vee(\mathbf{U}_2)$	0000100
NotZero(pref-$\vee(\mathbf{rm}(b)) \oplus$ pref-$\vee(\mathbf{U}_2)$)	1111111
$\neg$NotZero(pref-$\vee(\mathbf{rm}(b)) \oplus$ pref-$\vee(\mathbf{U}_2)$)	0000000

We obtain $\text{Value}_{3,S}(c) = \mathbf{0}$, which indicates that the third letter of w is not a c.

4.3.2 The JProd_S operation

This part proves the following lemma:

Lemma 4.12

For any semigroup $S \in \mathbf{DA}$, we can compute JProd_S over any vectorial encoding of a partial evaluation in its domain with a Sweeping-vectorial circuit of size $O(|S|^2)$.

Let $u = u_0 \cdots u_k$ be a word of S^+ and the set $(\mathbf{c}(s))_{s \in S}$ be a vectorial encoding of u. To compute $\text{JProd}_S(u)$, we want to detect the first and last bits set to 1 in $\mathbf{U}$ in order to compute the vectors corresponding to the word composed only of the element $\pi_S(u_0 u_k)$. The first element is directly indicated by the vectors $\text{Value}_{1,S}(s)$ for each element $s \in S$. Now we detect the last element by computing the similar vectors $\mathbf{Last}(s)$ for each $s \in S$:

$$\mathbf{Last}(s) = \text{Eq}(\text{suf-}\vee(\mathbf{c}(s)), \text{suf-}\vee(\mathbf{U}))$$

To complete the product we need to set to 1 the bit at the end of the word in the right vector. To do this, we detect the position of the last bit set to 1 in the union of the inputs and store it in the vector **PosLast**, which then has a unique bit set to 1.

$$\mathbf{PosLast} = \mathbf{U} \wedge \neg\text{MSB}(\mathbf{U})$$

Now we can perform the multiplication. To do this, we define, for each element $s \in S$, the set $M_s = \{(t, p) \in S^2 \mid \mathcal{R}(t) \cap \mathcal{L}(p) = \{s\}\}$. Now, we compute the vectors $\mathbf{out}(s)$ for each element $s \in S$. These vectors contain only the product of the first and last elements, or the first element if u contains only one element. With our definition of M_s, for the same reasons as

in the proof of Lemma 4.8, we do not need to verify that there were two elements to multiply, since if there is only one element s, it is equal to the unique element of $\mathcal{R}(s) \cap \mathcal{L}(s)$ and thus will stay unchanged.

$$\mathbf{out}(s) = \mathbf{PosLast} \wedge \bigvee_{(t,p)\in M_s} \mathrm{Value}_{1,S}(t) \wedge \mathbf{Last}(p)$$

Note that, since a given pair of elements (t, p) can only belong to one set M_s, this computation has a complexity of $O(|S|^2)$.

4.3.3 The LProd_S and RProd_S operations

This part proves the following lemma:

Lemma 4.13

For any semigroup $S \in \mathbf{DA}$, we can compute LProd_S and RProd_S over any vectorial encoding of a partial evaluation in their domains with Sweeping-vectorial circuits of size $O(|S|^2)$.

The two operations are symmetrical, so we will just present the circuit for LProd_S. Let $u = u_0 \cdots u_k$ be a word of S^+ and $(\mathbf{c}(s))_{s\in S}$ be a vectorial encoding of u. We want to detect the first and second bits set to 1 in $\mathbf{U}$ in order to compute the vectors corresponding to the word composed only of the element $\pi_S(u_0u_1)$. We can use $\mathrm{Value}_{1,S}$ and $\mathrm{Value}_{2,S}$ to compute vectors that give the values of the first and second elements.

If the word u is composed of only one element, we cannot detect a second element, so the vector $\mathrm{Thr2}((\mathbf{c}(s))_{s\in S})$ can be used as a way to know if the vectors obtained after using $\mathrm{Value}_{1,S}$ and $\mathrm{Value}_{2,S}$ have a meaning. If that vector is not null, we want to compute the product of the elements u_0 and u_1. To complete the product, we need to set to 1 the bit at the position of the second element in the vector corresponding to the value of the product. To do this, we detect the position of the second element and store it in the vector **PosSec**, which has a unique 1 in the position of the second element.

$$\mathbf{AfterSec} = \text{pref-}\vee(\mathrm{LSB}(\mathbf{U}))$$

$$\mathbf{PosSec} = \mathbf{AfterSec} \wedge \neg\mathrm{LSB}(\mathbf{AfterSec})$$

Now we start to prepare the multiplication. We compute the vectors $\mathbf{rm}(x)$ (for each element $x \in S$), in which we set to 0 the bits of the first two elements of the word.

$$\mathbf{Other} = \mathrm{LSB}(\mathrm{LSB}(\mathbf{U}))$$

$$\mathbf{rm}(x) = \mathbf{c}(x) \wedge \mathbf{Other}$$

Now we only have to add the product of the two first elements in the right vector. So, for each element $s \in S$, we compute the vector $\mathbf{Prod}(s)$, in which we add the position of the second element if that second element is an s. To do that, for each element $s \in S$, we define M_s to be the set of pairs (t, p) such that $s = \pi_S(tp)$. In symbols, $M_s = \{(t, p) \in S^2 \mid s = \pi_S(tp)\}$.

$$\mathbf{Prod}(s) = \mathbf{rm}(s) \vee \bigvee_{(t,p)\in M_s} \mathrm{Value}_{1,S}(t) \wedge \mathrm{Value}_{2,S}(p) \wedge \mathbf{PosSec}$$

Note that, since a given pair of elements (t, p) can only belong to one set M_s, this computation has a complexity of $O(|S|^2)$. Finally, the vectors $\mathbf{Prod}(s)$ only have meaning if there was two elements to multiply in the first place, so the vectors $\mathbf{out}(s)$ are computed as follows:

$$\mathbf{out}(s) = \text{IfThenElse}(\text{Thr2}((\mathbf{c}(s))_{s\in S}), \mathbf{Prod}(s), \mathbf{c}(s))$$

4.3.4 The $\text{LSplit}_{S,i}$ and $\text{RSplit}_{S,i}$ operations

This part proves the following lemma:

Lemma 4.14

Let S be a semigroup in **DA** of $\mathcal{J}$-depth d, i be an integer such that $1 \leq i \leq d$, let P be a sweeping evaluation program defined at least on all left $\mathcal{J}$-constant words of depth i, and suppose that we have a Sweeping-vectorial circuit of size s_P that computes P over any vectorial encoding of a partial evaluation. Then we can compute $\text{LSplit}_{S,i}\langle P\rangle$ and $\text{RSplit}_{S,i}\langle P\rangle$ over any vectorial encoding of a partial evaluation in their respective domains with Sweeping-vectorial circuits of size $O(|S|^2 + s_P)$.

The two operations are symmetrical, so we will only present the circuit for $\text{LSplit}_{S,i}\langle P\rangle$. Let $u = u_0 \cdots u_k$ be a word of S^+ and the set $(\mathbf{c}(s))_{s\in S}$ be a vectorial encoding of u. If u_0 is of $\mathcal{J}$-depth i, we want to detect the first element u_i such that $\pi_S(u_0 \cdots u_i)$ is of $\mathcal{J}$-depth at least $i+1$ in order to replace the prefix of $u_0 \cdots u_{i-1}$ by its image through P. To do that, we begin by checking if the first element of the word is of $\mathcal{J}$-depth i. We want to have an output different from the input if and only if u_0 is of $\mathcal{J}$-depth i, i.e. if $u_0 \in D_i(S)$:

$$\mathbf{DoSmth} = \bigvee_{s\in D_i(S)} \text{Value}_{1,S}(s)$$

For now, suppose that $\mathbf{DoSmth} \neq \mathbf{0}$. We can now determine the $\mathcal{R}$-class of the first element. For each $\mathcal{R}$-class R of $\mathcal{J}$-depth i, we compute the vector $\mathbf{RClass}(R)$ that is full of ones if and only if the first element is in R:

$$\mathbf{RClass}(R) = \bigvee_{s\in R} \text{Value}_{1,S}(s)$$

Now, we want to find the first position such that the product of the prefix is of $\mathcal{J}$-depth at least $i+1$. To find that position, we use Lemma 1.4, which tells us that the set of elements that make that product fall in a $\mathcal{J}$-class of greater $\mathcal{J}$-depth depends only on the $\mathcal{R}$-class of the prefix, which is uniquely determined by the first element, since that element is necessarily of $\mathcal{J}$-depth i. We already have the vectors indicating what $\mathcal{R}$-class should be considered. Now, for each $\mathcal{R}$-class, let F_R be the set of elements of S such that for any pair $(r, s) \in R \times F_R$, $\pi_S(rs) \notin R$ (the set K defined in Lemma 1.4). In symbols, $F_R = \{s \in S \mid \forall r \in R,\ \pi_S(rs) \notin R\}$. We want to consider only the set F_R corresponding to the $\mathcal{R}$-class of the first element and to search for the first element of that set that is not the first element of the factorization. To do this, we begin by computing, for any element $a \in S$, the vector $\mathbf{Next}(a)$ which allows us to know where the first element a after the first element is:

$$\mathbf{Next}(a) = \text{pref-}\vee(\mathbf{c}(a) \wedge \text{LSB}(\mathbf{U}))$$

Then we assemble those results depending on the set F_R each element a belongs to. For each $\mathcal{R}$-class R of $\mathcal{J}$-depth i, we compute the following vector:

$$\mathbf{BeforeFall}(R) = \neg \bigvee_{a \in F_R} \mathbf{Next}(a)$$

Each vector $\mathbf{BeforeFall}(R)$ indicates all the positions strictly before the first element $s \in F_R$ that is at least the second element of the word. With this, we know that the prefix indicated by $\mathbf{BeforeFall}(R)$, where R is the $\mathcal{R}$-class of the first element, is the unique maximal prefix which product belongs to R. Thus, we want to apply the circuit corresponding to the program P to the prefix that is indicated by $\mathbf{BeforeFall}(R)$, where R is the $\mathcal{R}$-class of the first element. Now, we need to introduce a mask which will indicate the prefix considered in the rest of the split operation. This mask is constructed from the vectors $\mathbf{BeforeFall}(R)$ for each $\mathcal{R}$-class R that belongs to the set $\mathcal{R}_i$ of $\mathcal{R}$-classes of $\mathcal{J}$-depth i, and it is equal to the vector $\mathbf{BeforeFall}(R)$ corresponding to the unique vector $\mathbf{RClass}(R)$ that is not null:

$$\mathbf{mask} = \bigvee_{R \in \mathcal{R}_i} \mathbf{RClass}(R) \wedge \mathbf{BeforeFall}(R)$$

Now we can use the circuit C_P on the set of vectors $(\mathbf{c}(s) \wedge \neg\mathbf{mask})_{s \in S}$. That call returns a set of vectors $(\mathbf{r}(s))_{s \in S}$ corresponding to the application of P to the prefix. Now we only have to add back the rest of the word. For each element $s \in S$, the vector $\mathbf{out}(s)$ is computed as follows:

$$\mathbf{out}(s) = \text{IfThenElse}(\mathbf{DoSmth}, \mathbf{r}(s) \vee (\mathbf{c}(s) \wedge \neg\mathbf{mask}), \mathbf{c}(s))$$

Conclusion and future prospects

Contributions

This thesis' overall goal was to study the possibility of a general application of bit-level parallelism to data validation. To that end, the algebraic theory of automata was chosen, as it provides the syntactic semigroups of languages. The interest of semigroups is two-fold: they include an inner product, which allows for efficient parallel computations, and they are able to express some properties of languages that can be used in a vectorized algorithm.

The main theoretical contribution of this thesis consists in Theorem 4.1 and Theorem 4.2. These two theorems give an explicit construction of vectorial circuits recognizing languages in respectively $\mathrm{FO}[<]$ and $\mathrm{FO}_2[<]$. These vectorial circuits can then be translated into vectorized programs using SIMD instructions. Moreover, the constructed vectorial circuits are small compared to what can be expected given the previous state of the art on the size of equivalent formulas. Indeed, the circuits constructed in Theorem 4.1 are of size polynomial in the size of the syntactic semigroup of the language, which is at most exponential in the size of the minimal automaton recognizing the language, whereas Serre's result, using LTL formulas, only gave a doubly exponential upper bound in the size of the minimal automaton (this is due to a bound given in [77]). Note that, in both cases, there is an exponential factor due to the transformation from the automata to the semigroups. It can be removed by considering the semigroup itself as an input of the programs, which removes the grey area surrounding that transformation, which may be desirable as not all semigroups are of size exponential in the size of the automaton. As another result, the circuits constructed in Theorem 4.2 are of size exponential in the size of the syntactic semigroup of the language, where little to no explicit algorithm existed before this thesis.

These theoretical contributions give better upper bounds on the size of the vectorized programs that can be obtained using the algebraic theory of automata, and were published in an article [57]. However, the practical applicability of those vectorized programs remained to be proven. This led to the implementations in [58] and [59]. The two repositories were originally meant to host an implementation of the vectorial circuits presented in chapter 4. However, the task is challenging as it requires to produce efficient code at compile time, and a lot of research remains to be done to determine what can increase or decrease the efficiency of the code structures presented in this thesis. Thus, the repositories served as sandboxes to measure the efficiency of several algorithms to recognize languages, some of these using semigroups (mostly sequentially), some using low-level instructions on vectors. This led to the benchmarks presented in chapter 2 and chapter 3, which show that it is possible to be faster than the baseline in some cases and highlight the fact that there is a lot of room for code optimization. That optimization is not trivial and requires more research to be understood.

Future prospects

The subject of this thesis, in particular on the implementation side, shows interesting research opportunities and, as indicated before, there is room for optimizing the implementation that has yet to be understood. Here are some future prospects for research in this domain.

Efficient implementations

The various toy examples provided in this thesis to measure the efficiency of the various algorithms implemented show that there is room for optimization, even without any bit-level parallelism. The results also show that the potential optimizations are complex and linked with intricate low-level phenomena. Interestingly, the efficiency of our implementation varies greatly (with several order of magnitude) between different languages. This may be linked to the ability of the compiler to perform efficient control-flow analysis. This analysis could itself depend on some algebraic properties of the underlying automata, although this hypothesis needs to be tested.

Study more in-depth the existing algorithms. The benchmarking of the repositories [58] and [59] was delayed by the study of the results and their implications. Indeed, the results are full of oddities. Some of these are linked to implementation flaws, such as the lack of tabulation of some parts of the programs. But it is not the case for all oddities, and we suspect that they are due to some low-level phenomena for which we can provide very little insight. Notably, it seems that the order of the products in the functions computing the inner product of a semigroup can greatly impact the efficiency of the programs, but that may not be the case, or simply a part of a much more complex answer. Perhaps the compiler re-arranges by itself the order of that product in a way that is sometimes adversarial to the program's practical efficiency. Thus, an in-depth study of the implementations and their possible variations could be particularly interesting.

Implement the vectorial circuits. The benchmarks presented in this thesis showed that SIMD instructions can lead to very efficient programs for data validation, and that semigroups, if processed cleverly, can lead to sequential programs that reach throughput values close to the ones achieved by Rust's `regex` crate. The final step, which consists in combining the two, remains to be implemented. That combined implementation could maybe use the lessons learned during the benchmarks presented in this thesis, as these benchmarks showed that a naive implementation of the semigroup's inner product is not very efficient. The repository [58] contains a preliminary attempt in the form of the macro `compile_u8_circuit`, which takes a structure representing a vectorial circuit and returns a vectorized program that corresponds to that circuit. That implementation has neither been tested enough, nor been optimized, but can be used as a placeholder for a future implementation which would compute the vectorial circuit corresponding to a given language.

Extending the theory

The theoretical results presented in this thesis could be extended to obtain results that apply to larger classes of languages.

Arbitrary constant vectors. This thesis completely ignores the possible use of arbitrary constant vectors, as they can drastically change the expressive power of a class of vectorial circuits. However, it could be interesting to study what *drastically* exactly means here. Adding arbitrary constant vectors to vectorial circuits amounts to adding arbitrary monadic predicates to the equivalent class of languages. This subject of arbitrary predicates has been studied by Straubing [70], who introduced the *Straubing property*: a logical class $F[\mathcal{P}]$ has the Straubing property if all regular languages definable in $F[\mathcal{P}]$ are definable in $F[\mathcal{P} \cap \mathcal{R}eg]$, where $\mathcal{P}$ is a set of arbitrary predicates. The particular case of arbitrary monadic predicates has been studied by Fijalkow and Paperman [22], who showed that any class of regular languages in $MSO[\mathcal{P}]$, where $\mathcal{P}$ is a set of arbitrary monadic predicates, is equal to some class of languages in $MSO[\mathcal{P}']$, where $\mathcal{P}'$ is a set of regular monadic predicates. To follow the lead of these results, it would be interesting to precisely define the expressive power brought to the various classes of regular languages by the arbitrary monadic predicates.

More operations. Arbitrary constant vectors are one way to increase the expressive power of vectorial circuits. Adding operations to the possible labels of gates is another possibility. There probably are other classes of regular languages that can be computed using a different set of operations. Notably, it could be interesting to study the expressive power added by the vectorial operation pref-$\oplus$.

SIMD as a class of circuits. Finally, the goal of this thesis was to make the most of some SIMD instructions, but what about the complete set of SIMD instructions? Does that set define a well-known class of circuits? We conjecture that the class of vectorial circuits using the full extend of SIMD expressive power is equal to the class TC^0 of boolean circuits.

Appendix A

Supplementary benchmarks

Outline of the current chapter

The benchmark results presented in chapter 2 have all been measured on one particular machine, on the node troll from grid5000. However, the code was run on other nodes, with different micro-architectures, in order to get a better idea of its general efficiency. The results are presented in this appendix. As before, the version of rustc used was rustc 1.72.1 (d5c2e9c34 2023-09-13), and the version of cargo was cargo 1.72.1 (103a7ff2e 2023-08-15). All the results have been measured as a mean of twenty runs of the same command.

A.1 Benchmark on Intel processors

The results presented in chapter 2 have been measured on an Intel processor, one of the most recent available on grid5000 when the code was executed. This node is marked as exotic, which we feared might skew the results. Thus, other benchmarks were run on other nodes with Intel processors. These processors are a bit older, which impacts the results.

A.1.1 The node dahu

The results in this subsection have been obtained using the machine dahu-9. At the time when this thesis is written, the machines dahu each have two CPUs Intel Xeon Gold 6130, on the architecture x86_64. Each of these CPUs has 16 cores.

Lang.	Word	Algorithm						
		Char (Gchar/s)				Bytes (GB/s)		
		base	determinize	deter-mini	flex	base	determinize	deter-mini
acstara	*last*	*X*	*X*	*X*	*X*	6.8	0.40	0.24
abstar	*dense*	0.30	0.50	0.89	0.23	0.61	0.51	0.92
	average	0.30	0.50	0.89	0.24	0.61	0.51	0.92
	sparse	0.30	0.50	0.89	0.23	0.61	0.51	0.92
$L_{\mathbf{DA}}$	*balanced*	0.30	0.44	0.25	0.23	0.61	0.39	0.24
	decrease	0.30	0.44	0.25	0.24	0.61	0.39	0.24
	increase	0.30	0.45	0.25	0.24	0.61	0.39	0.24
$L_{\mathcal{R}}$	*first*	0.30	0.85	0.57	0.24	0.61	0.22	0.57
	last	0.30	0.85	0.57	0.23	0.61	0.30	0.57
	middle	0.30	0.85	0.57	0.23	0.61	0.25	0.57

Table A.1 – The results of the sequential algorithms - dahu

Table A.2 – The results of the semigroup-based algorithms - dahu

Lang.	Word	Algorithm									
		Char (Gchar/s)					Bytes (GB/s)				
		base	stamp	stamp-mini	table	parallel	base	stamp	stamp-mini	table	table-mini
acstara	*last*	*X*	*X*	*X*	*X*	*X*	6.8	0.14	0.16	0.23	0.21
abstar	*dense*	0.30	0.19	0.53	0.60	10	0.61	0.32	0.49	0.52	0.61
	average	0.30	0.31	0.49	0.60	11	0.61	0.37	0.37	0.52	0.61
	sparse	0.30	0.19	0.57	0.60	10	0.61	0.12	0.41	0.52	0.61
$L_{\mathbf{DA}}$	*balanced*	0.30	0.046	0.082	0.52	7.8	0.61	0.036	0.072	0.15	0.16
	decrease	0.30	0.043	0.087	0.52	7.3	0.61	0.032	0.079	0.14	0.16
	increase	0.30	0.049	0.078	0.52	8.3	0.61	0.040	0.066	0.15	0.17
$L_{\mathcal{R}}$	*first*	0.30	0.10	0.17	0.52	7.4	0.61	0.19	0.19	0.25	0.22
	last	0.30	0.092	0.17	0.52	8.3	0.61	0.24	0.19	0.24	0.24
	middle	0.30	0.088	0.17	0.52	7.8	0.61	0.23	0.18	0.24	0.23

Panel A: Results of the sequential semigroup versions

Language	Word	Algorithm (GB/s)								
		base	stamp	parallel				parallel-mini			
				32 cores	16 cores	8 cores	4 cores	32 cores	16 cores	8 cores	4 cores
acstara	*last*	6.8	0.14	3.4	2.1	1.1	0.59	3.7	2.1	1.3	0.65
abstar	*dense*	0.61	0.32	7.0	5.0	3.0	1.4	8.9	7.0	3.9	2.0
	average	0.61	0.37	6.7	4.6	2.6	1.3	7.4	6.8	3.8	2.0
	sparse	0.61	0.12	6.2	4.5	2.5	1.3	8.1	6.6	3.7	1.9
$L_{\mathbf{DA}}$	*balanced*	0.61	0.036	2.8	1.6	0.94	0.48	2.8	1.6	0.90	0.49
	decrease	0.61	0.032	2.8	1.7	0.91	0.46	2.7	1.5	0.89	0.47
	increase	0.61	0.040	3.1	1.6	0.96	0.48	3.1	1.9	0.98	0.50
$L_{\mathcal{R}}$	*first*	0.61	0.19	4.2	2.6	1.5	0.77	3.8	2.3	1.4	0.70
	last	0.61	0.24	4.1	2.2	1.3	0.69	3.2	2.7	1.4	0.72
	middle	0.61	0.23	4.4	2.3	1.4	0.72	4.0	2.3	1.4	0.72

Panel B: Results of the parallel versions depending on the maximum number of cores used

Lang.	Word	Algorithm (GB/s)					
		base	stamp	facto	facto-min	$\mathcal{R}$-stamp	$\mathcal{R}$-stamp-min
acstara	*last*	6.8	0.14	0.19	0.18	0.15	0.15
abstar	*dense*	0.61	0.32	0.41	0.44	0.49	0.34
	average	0.61	0.37	0.40	0.41	0.52	0.49
	sparse	0.61	0.12	0.41	0.41	0.48	0.45
$L_{\mathbf{DA}}$	*balanced*	0.61	0.036	0.14	0.14	0.12	0.12
	decrease	0.61	0.032	0.13	0.14	0.12	0.11
	increase	0.61	0.040	0.14	0.15	0.12	0.12
$L_{\mathcal{R}}$	*first*	0.61	0.19	0.23	0.26	0.19	0.18
	last	0.61	0.24	0.22	0.28	0.18	0.19
	middle	0.61	0.23	0.23	0.27	0.18	0.19

Lang.	Word	Algorithm (GB/s)					
		$\mathcal{R}$-ord	$\mathcal{R}$-ord-min	ord-facto	ord-facto-min	assembled	assembled-min
acstara	*last*	0.18	0.18	0.20	0.20	0.21	0.23
abstar	*dense*	0.51	0.69	0.77	1.2	0.77	0.76
	average	0.37	0.83	0.73	1.2	0.61	0.92
	sparse	0.69	0.51	0.61	0.74	0.74	1.2
$L_{\mathbf{DA}}$	*balanced*	0.13	0.15	0.17	0.16	0.16	0.17
	decrease	0.12	0.14	0.17	0.15	0.15	0.17
	increase	0.14	0.15	0.17	0.16	0.16	0.17
$L_{\mathcal{R}}$	*first*	1.2	0.92	1.6	1.2	0.24	0.21
	last	1.8	1.8	0.92	1.8	1.8	1.8
	middle	1.2	1.2	1.2	2.0	0.43	0.44

Table A.3 – The results of the advanced semigroup-based algorithms - dahu

A.1.2 The node chifflot

The results in this subsection have been obtained using the machine chifflot-8. At the time when this thesis is written, the machines chifflot each have two CPUs Intel Xeon Gold 6126, on the architecture x86_64. Each of these CPUs has 12 cores.

Lang.	Word	Algorithm						
		Char (Gchar/s)				Bytes (GB/s)		
		base	determinize	deter-mini	flex	base	determinize	deter-mini
acstara	*last*	X	X	X	X	6.9	0.37	0.29
abstar	*dense*	0.30	0.50	0.89	0.23	0.61	0.43	0.91
	average	0.30	0.50	0.89	0.23	0.61	0.43	0.91
	sparse	0.30	0.50	0.89	0.23	0.61	0.43	0.92
$L_{\mathbf{DA}}$	*balanced*	0.30	0.44	0.25	0.23	0.61	0.39	0.24
	decrease	0.30	0.44	0.25	0.24	0.60	0.39	0.24
	increase	0.30	0.44	0.25	0.23	0.60	0.39	0.24
$L_{\mathcal{R}}$	$first$	0.30	0.85	0.57	0.23	0.61	0.19	0.59
	last	0.30	0.85	0.57	0.23	0.61	0.26	0.59
	middle	0.30	0.85	0.57	0.23	0.61	0.22	0.59

Table A.4 – The results of the sequential algorithms - chifflot

Table A.5 – The results of the semigroup-based algorithms - chifflot

Lang.	Word	Algorithm									
		Char (Gchar/s)					Bytes (GB/s)				
		base	stamp	stamp-mini	table	parallel	base	stamp	stamp-mini	table	table-mini
acstara	*last*	*X*	*X*	*X*	*X*	*X*	6.9	0.12	0.16	0.20	0.22
abstar	*dense*	0.30	0.23	0.39	0.60	7.1	0.61	0.31	0.57	0.52	0.61
	average	0.30	0.41	0.36	0.60	7.9	0.61	0.36	0.44	0.52	0.61
	sparse	0.30	0.28	0.39	0.60	7.9	0.61	0.13	0.40	0.52	0.61
$L_{\mathbf{DA}}$	*balanced*	0.30	0.045	0.081	0.52	5.4	0.61	0.036	0.071	0.15	0.16
	decrease	0.30	0.042	0.085	0.52	5.2	0.60	0.032	0.079	0.14	0.16
	increase	0.30	0.049	0.078	0.52	5.6	0.60	0.040	0.065	0.15	0.16
$L_{\mathcal{R}}$	*first*	0.30	0.10	0.16	0.52	6.5	0.61	0.19	0.19	0.25	0.22
	last	0.30	0.093	0.16	0.52	7.3	0.61	0.24	0.19	0.24	0.24
	middle	0.30	0.087	0.16	0.52	6.9	0.61	0.22	0.18	0.24	0.23

Panel A: Results of the sequential semigroup versions

Language	Word	Algorithm (GB/s)								
		base	stamp	parallel				parallel-mini			
				32 cores	16 cores	8 cores	4 cores	32 cores	16 cores	8 cores	4 cores
acstara	*last*	6.9	0.12	3.8	2.0	1.1	0.54	4.0	2.5	1.3	0.65
abstar	*dense*	0.61	0.31	8.0	5.5	2.8	1.5	10	4.9	2.8	1.4
	average	0.61	0.36	8.5	4.4	2.5	1.2	9.1	4.4	2.3	1.3
	sparse	0.61	0.13	8.2	4.2	2.4	1.2	9.1	4.3	2.3	1.2
$L_{\mathbf{DA}}$	*balanced*	0.61	0.036	3.2	1.8	0.90	0.48	3.1	1.9	0.98	0.51
	decrease	0.60	0.032	3.1	1.7	0.88	0.47	3.0	1.9	0.98	0.49
	increase	0.60	0.040	3.2	1.8	0.93	0.48	3.2	2.0	1.0	0.52
$L_{\mathcal{R}}$	*first*	0.61	0.19	4.5	2.8	1.4	0.72	4.5	2.9	1.5	0.78
	last	0.61	0.24	4.4	2.6	1.4	0.68	4.5	2.9	1.6	0.80
	middle	0.61	0.22	4.5	2.7	1.4	0.71	4.5	2.9	1.6	0.79

Panel B: Results of the parallel versions depending on the maximum number of cores used

Lang.	Word	Algorithm (GB/s)					
		base	stamp	facto	facto-min	$\mathcal{R}$-stamp	$\mathcal{R}$-stamp-min
acstara	*last*	6.9	0.12	0.15	0.20	0.16	0.16
abstar	*dense*	0.61	0.31	0.40	0.57	0.40	0.39
	average	0.61	0.36	0.40	0.45	0.47	0.60
	sparse	0.61	0.13	0.40	0.52	0.44	0.48
$L_{\mathbf{DA}}$	*balanced*	0.61	0.036	0.14	0.14	0.12	0.12
	decrease	0.60	0.032	0.13	0.14	0.12	0.11
	increase	0.60	0.040	0.14	0.15	0.12	0.12
$L_{\mathcal{R}}$	*first*	0.61	0.19	0.24	0.24	0.19	0.18
	last	0.61	0.24	0.22	0.26	0.18	0.19
	middle	0.61	0.22	0.23	0.25	0.18	0.19

Lang.	Word	Algorithm (GB/s)					
		$\mathcal{R}$-ord	$\mathcal{R}$-ord-min	ord-facto	ord-facto-min	assembled	assembled-min
acstara	*last*	0.17	0.18	0.21	0.20	0.20	0.21
abstar	*dense*	0.47	0.83	0.76	0.86	0.77	0.85
	average	0.40	0.84	0.73	1.2	0.61	0.94
	sparse	0.68	0.52	0.63	0.61	0.61	0.93
$L_{\mathbf{DA}}$	*balanced*	0.13	0.15	0.17	0.16	0.16	0.17
	decrease	0.12	0.14	0.17	0.15	0.15	0.17
	increase	0.14	0.15	0.17	0.16	0.16	0.17
$L_{\mathcal{R}}$	*first*	1.2	0.91	1.6	1.2	0.24	0.21
	last	1.8	1.8	0.91	1.8	1.8	1.8
	middle	1.2	1.2	1.2	2.0	0.43	0.44

Table A.6 – The results of the advanced semigroup-based algorithms - chifflot

A.2 Benchmark on an AMD processor: the node chiclet

The results in this section have been obtained using the machine chiclet-8. At the time when this thesis is written, the machines chiclet each have two CPUs AMD EPYC 7301, on the architecture x86_64. Each of these CPUs has 16 cores.

Lang.	Word	Algorithm						
		Char (Gchar/s)				Bytes (GB/s)		
		base	determinize	deter-mini	flex	base	determinize	deter-mini
acstara	*last*	X	X	X	X	5.5	0.27	0.23
abstar	*dense*	0.13	0.33	0.65	0.16	0.45	0.35	0.66
	average	0.13	0.33	0.65	0.16	0.45	0.35	0.66
	sparse	0.13	0.33	0.65	0.16	0.45	0.35	0.66
$L_{\mathbf{DA}}$	*balanced*	0.13	0.32	0.17	0.16	0.45	0.32	0.18
	decrease	0.13	0.32	0.17	0.16	0.45	0.32	0.18
	increase	0.13	0.32	0.17	0.16	0.45	0.32	0.18
$L_{\mathcal{R}}$	*first*	0.13	0.60	0.31	0.16	0.45	0.18	0.31
	last	0.13	0.60	0.31	0.16	0.45	0.47	0.31
	middle	0.13	0.6	0.31	0.16	0.45	0.26	0.31

Table A.7 – The results of the sequential algorithms - chiclet

Table A.8 – The results of the semigroup-based algorithms - chiclet

Lang.	Word	Algorithm									
		Char (Gchar/s)					Bytes (GB/s)				
		base	stamp	stamp-mini	table	parallel	base	stamp	stamp-mini	table	table-mini
acstara	*last*	*X*	*X*	*X*	*X*	*X*	5.5	0.12	0.15	0.18	0.19
abstar	*dense*	0.13	0.18	0.38	0.38	2.9	0.45	0.23	0.36	0.34	0.38
	average	0.13	0.38	0.49	0.38	2.9	0.45	0.53	0.37	0.33	0.38
	sparse	0.13	0.27	0.51	0.38	3.0	0.45	0.14	0.33	0.33	0.38
$L_{\mathbf{DA}}$	*balanced*	0.13	0.037	0.067	0.33	2.8	0.45	0.026	0.080	0.14	0.14
	decrease	0.13	0.034	0.069	0.33	2.7	0.45	0.024	0.084	0.14	0.14
	increase	0.13	0.039	0.065	0.33	2.7	0.45	0.029	0.076	0.14	0.14
$L_{\mathcal{R}}$	*first*	0.13	0.081	0.12	0.33	3.0	0.45	0.17	0.19	0.21	0.22
	last	0.13	0.080	0.12	0.33	3.1	0.45	0.17	0.17	0.21	0.22
	middle	0.13	0.076	0.12	0.33	3.1	0.45	0.18	0.18	0.21	0.21

Panel A: Results of the sequential semigroup versions

Language	Word	Algorithm (GB/s)								
		base	stamp	parallel				parallel-mini			
				32 cores	16 cores	8 cores	4 cores	32 cores	16 cores	8 cores	4 cores
acstara	*last*	5.5	0.12	3.6	1.9	0.94	0.47	4.0	2.1	1.1	0.53
abstar	*dense*	0.45	0.23	6.7	3.6	1.8	0.92	6.9	3.7	1.9	0.99
	average	0.45	0.53	8.7	4.7	2.5	1.2	8.5	4.5	2.4	1.1
	sparse	0.45	0.14	7.9	4.1	2.3	1.1	7.4	3.8	1.9	0.99
$L_{\mathbf{DA}}$	*balanced*	0.45	0.026	2.8	1.5	0.74	0.37	3.1	1.6	0.80	0.40
	decrease	0.45	0.024	2.8	1.4	0.73	0.36	3.1	1.6	0.78	0.39
	increase	0.45	0.029	2.9	1.5	0.75	0.38	3.2	1.6	0.82	0.40
$L_{\mathcal{R}}$	*first*	0.45	0.17	4.1	2.1	1.1	0.56	4.2	2.2	1.1	0.54
	last	0.45	0.17	4.2	2.2	1.1	0.55	4.3	2.2	1.1	0.55
	middle	0.45	0.18	4.1	2.1	1.1	0.55	4.2	2.2	1.1	0.54

Panel B: Results of the parallel versions depending on the maximum number of cores used

Lang.	Word	Algorithm (GB/s)					
		base	stamp	facto	facto-min	$\mathcal{R}$-stamp	$\mathcal{R}$-stamp-min
acstara	*last*	5.5	0.12	0.13	0.14	0.13	0.13
abstar	*dense*	0.45	0.23	0.32	0.39	0.27	0.25
	average	0.45	0.53	0.39	0.33	0.35	0.38
	sparse	0.45	0.14	0.43	0.32	0.36	0.38
$L_{\mathbf{DA}}$	*balanced*	0.45	0.026	0.10	0.12	0.11	0.11
	decrease	0.45	0.024	0.099	0.11	0.11	0.11
	increase	0.45	0.029	0.10	0.12	0.11	0.11
$L_{\mathcal{R}}$	*first*	0.45	0.17	0.18	0.20	0.16	0.15
	last	0.45	0.17	0.16	0.21	0.13	0.16
	middle	0.45	0.18	0.17	0.20	0.14	0.15

Lang.	Word	Algorithm (GB/s)					
		$\mathcal{R}$-ord	$\mathcal{R}$-ord-min	ord-facto	ord-facto-min	assembled	assembled-min
acstara	*last*	0.14	0.17	0.17	0.17	0.18	0.18
abstar	*dense*	0.38	0.53	0.94	0.87	0.71	0.63
	average	0.54	0.54	0.83	0.79	0.89	0.89
	sparse	0.40	0.51	0.68	0.88	0.69	0.89
$L_{\mathbf{DA}}$	*balanced*	0.12	0.11	0.12	0.13	0.13	0.13
	decrease	0.11	0.10	0.12	0.13	0.12	0.13
	increase	0.12	0.12	0.13	0.14	0.13	0.14
$L_{\mathcal{R}}$	*first*	1.2	1.3	1.2	1.3	0.19	0.20
	last	1.8	1.3	1.3	1.3	1.3	1.8
	middle	1.5	1.3	1.3	1.3	0.34	0.36

Table A.9 – The results of the advanced semigroup-based algorithms - chiclet

Bibliography

[1] Alfred V Aho. "Pattern matching in strings". In: *Formal Language Theory*. Elsevier, 1980, pp. 325–347.

[2] Alfred V Aho and Margaret J Corasick. "Efficient string matching: an aid to bibliographic search". In: *Communications of the ACM* 18.6 (1975), pp. 333–340.

[3] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: principles, techniques, and tools*. Vol. 1. Addison-wesley Reading, 1988.

[4] Anonymus. *Mastering Quantifiers*. URL: https://web.archive.org/web/20230819223759/https://www.rexegg.com/regex-quantifiers.html.

[5] Ricardo Baeza-Yates and Gaston H Gonnet. "A new approach to text searching". In: *Communications of the ACM* 35.10 (1992), pp. 74–82.

[6] Ricardo Baeza-Yates, Berthier Ribeiro-Neto, et al. *Modern information retrieval*. Vol. 463. 1999. ACM press New York, 1999.

[7] David A Mix Barrington et al. "Regular languages in NC1". In: *Journal of Computer and System Sciences* 44.3 (1992), pp. 478–499.

[8] Anne Bergeron and Sylvie Hamel. "Cascade decompositions are bit-vector algorithms". In: *Implementation and Application of Automata: 6th International Conference, CIAA 2001 Pretoria, South Africa, July 23–25, 2001 Revised Papers 6*. Springer. 2002, pp. 13–26.

[9] Guy E Blelloch. *Vector models for data-parallel computing*. Vol. 2. Citeseer, 1990.

[10] Mikołaj Bojańczyk. "Factorization forests". In: *International Conference on Developments in Language Theory*. Springer. 2009, pp. 1–17.

[11] Robert S Boyer and J Strother Moore. "A fast string searching algorithm". In: *Communications of the ACM* 20.10 (1977), pp. 762–772.

[12] Janusz A Brzozowski. "Canonical regular expressions and minimal state graphs for definite events". In: *Proc. Symposium of Mathematical Theory of Automata*. 1962, pp. 529–561.

[13] Michaël Cadilhac and Charles Paperman. "The regular languages of wire linear AC 0". In: *Acta Informatica* 59.4 (2022), pp. 321–336.

[14] Robert D Cameron et al. "Bitwise data parallelism in regular expression matching". In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 2014, pp. 139–150.

[15] Ashok K Chandra, Steven Fortune, and Richard Lipton. "Lower bounds for constant depth circuits for prefix problems". In: *Automata, Languages and Programming: 10th Colloquium Barcelona, Spain, July 18–22, 1983 10*. Springer. 1983, pp. 109–117.

[16] Shiva Chaudhuri and Jaikumar Radhakrishnan. "Deterministic restrictions in circuit complexity". In: *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*. 1996, pp. 30–36.

[17] Russ Cox. "Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...)" In: *URL: http://swtch. com/rsc/regexp/regexp1. html* (2007), p. 94.

[18] Maxime Crochemore and Wojciech Rytter. *Text algorithms*. Maxime Crochemore, 1994.

[19] Volker Diekert, Paul Gastin, and Manfred Kufleitner. "A survey on small fragments of first-order logic over finite words". In: *International Journal of Foundations of Computer Science* 19.03 (2008), pp. 513–548.

[20] Kousha Etessami, Moshe Y Vardi, and Thomas Wilke. "First-order logic with two variables and unary temporal logic". In: *Proceedings of Twelfth Annual IEEE Symposium on Logic in Computer Science*. IEEE. 1997, pp. 228–235.

[21] Ronald Fagin et al. "Document spanners: A formal approach to information extraction". In: *Journal of the ACM (JACM)* 62.2 (2015), pp. 1–51.

[22] Nathanaël Fijalkow and Charles Paperman. "Monadic second-order logic with arbitrary monadic predicates". In: *ACM Transactions on Computational Logic (TOCL)* 18.3 (2017), pp. 1–17.

[23] Merrick Furst, James B Saxe, and Michael Sipser. "Parity, circuits, and the polynomial-time hierarchy". In: *Mathematical systems theory* 17.1 (1984), pp. 13–27.

[24] Dov Gabbay et al. "On the temporal analysis of fairness". In: *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1980, pp. 163–173.

[25] Andrew Gallant. *Rebar*. https://web.archive.org/web/20230911123902/https://github.com/BurntSushi/rebar. 2023.

[26] Andrew Gallant. *Regex engine internals as a library*. 2023. URL: https://web.archive.org/web/20230903144141/https://blog.burntsushi.net/regex-internals/.

[27] Mateusz Gienieczko. "Fast execution of JSONPath queries". Master thesis. University of Warsaw, 2022.

[28] Mateusz Gienieczko, Charles Paperman, and Filip Murlak. *rsonpath*. https://web.archive.org/web/20230411022735/https://github.com/V0ldek/rsonpath. 2022.

[29] Victor Mikhaylovich Glushkov. "The abstract theory of automata". In: *Russian Mathematical Surveys* 16.5 (1961), p. 1.

[30] Leslie Michael Goldschlager. *Synchronous parallel computation*. 1978.

[31] Mike Haertel. *Why GNU grep is fast*. 2010. URL: https://web.archive.org/web/20230709032219/https://lists.freebsd.org/pipermail/freebsd-current/2010-August/019310.html (visited on 08/09/2023).

[32] Juris Hartmanis and Janos Simon. "On the power of multiplication in random access machines". In: *15th Annual Symposium on Switching and Automata Theory (swat 1974)*. IEEE. 1974, pp. 13–23.

[33] John Hopcroft. "An n log n algorithm for minimizing states in a finite automaton". In: *Theory of machines and computations*. Elsevier, 1971, pp. 189–196.

[34] R Nigel Horspool. "Practical fast searching in strings". In: *Software: Practice and Experience* 10.6 (1980), pp. 501–506.

[35] Andrew Hume. "A tale of two greps". In: *Software: Practice and Experience* 18.11 (1988), pp. 1063–1072.

[36] Neil Immerman. "Descriptive and computational complexity". In: *Computational Complexity Theory, ed. J. Hartmanis, Proc. Symp. in Applied Math.* Vol. 38. 1989, pp. 75–91.

[37] Lin Jiang and Zhijia Zhao. "JSONSki: streaming semi-structured data with bit-parallel fast-forwarding". In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2022, pp. 200–211.

[38] Mario Juárez. *Regex benchmark*. https://web.archive.org/web/20230414164714/http://github.com/mariomka/regex-benchmark. 2021.

[39] Johan Anthony Wilem Kamp. *Tense logic and the theory of linear order*. University of California, Los Angeles, 1968.

[40] John Keiser and Daniel Lemire. "Validating UTF-8 in less than one instruction per byte". In: *Software: Practice and Experience* 51.5 (2021), pp. 950–964.

[41] Stephen C Kleene et al. "Representation of events in nerve nets and finite automata". In: *Automata studies* 34 (1956), pp. 3–41.

[42] Donald E Knuth, James H Morris Jr, and Vaughan R Pratt. "Fast pattern matching in strings". In: *SIAM journal on computing* 6.2 (1977), pp. 323–350.

[43] Michal Koucký, Pavel Pudlák, and Denis Thérien. "Bounded-depth circuits: separating wires from gates". In: *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*. 2005, pp. 257–265.

[44] Leslie Lamport. "Multiple byte processing with full-word instructions". In: *Communications of the ACM* 18.8 (1975), pp. 471–475.

[45] Geoff Langdale and Daniel Lemire. "Parsing gigabytes of JSON per second". In: *The VLDB Journal* 28.6 (2019), pp. 941–960.

[46] Dan Lin et al. "Parabix: Boosting the efficiency of text processing on commodity processors". In: *IEEE International Symposium on High-Performance Comp Architecture*. IEEE. 2012, pp. 1–12.

[47] Niko Matsakis. *Rayon: data parallelism in Rust*. 2015. URL: https://web.archive.org/web/20230527040204/http://smallcultfollowing.com/babysteps/blog/2015/12/18/rayon-data-parallelism-in-rust/.

[48] Robert McNaughton and Seymour A Papert. *Counter-Free Automata (MIT research monograph no. 65)*. The MIT Press, 1971.

[49] Robert McNaughton and Hisao Yamada. "Regular expressions and state graphs for automata". In: *IRE transactions on Electronic Computers* 1 (1960), pp. 39–47.

[50] Albert R Meyer and Michael J Fischer. "Economy of description by automata, grammars, and formal systems". In: *12th annual symposium on switching and automata theory (swat 1971)*. IEEE Computer Society. 1971, pp. 188–191.

[51] Filip Murlak, Charles Paperman, and Michał Pilipczuk. "Schema validation via streaming circuits". In: *Proceedings of the 35th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 2016, pp. 237–249.

[52] Gene Myers. “A fast bit-vector algorithm for approximate string matching based on dynamic programming”. In: *Journal of the ACM (JACM)* 46.3 (1999), pp. 395–415.

[53] Gonzalo Navarro. “NR-grep: a fast and flexible pattern-matching tool”. In: *Software: Practice and Experience* 31.13 (2001), pp. 1265–1312.

[54] Gonzalo Navarro and Mathieu Raffinot. “A bit-parallel approach to suffix automata: Fast extended string matching”. In: *Annual Symposium on Combinatorial Pattern Matching*. Springer. 1998, pp. 14–33.

[55] Gonzalo Navarro and Mathieu Raffinot. *Flexible pattern matching in strings: practical on-line search algorithms for texts and biological sequences*. Cambridge university press, 2002.

[56] Charles Paperman, Sylvain Salvati, and Claire Soyez-Martin. *Addition Lemma*. Sept. 2022. URL: https://hal.archives-ouvertes.fr/hal-03787033.

[57] Charles Paperman, Sylvain Salvati, and Claire Soyez-Martin. “An Algebraic Approach to Vectorial Programs”. In: *40th International Symposium on Theoretical Aspects of Computer Science*. 2023.

[58] Charles Paperman and Claire Soyez-Martin. *Workspace macros*. 2023. URL: https://gitlab.univ-lille.fr/claire.soyez-martin/workspace-macros.

[59] Charles Paperman and Claire Soyez-Martin. *Workspace macros u8*. 2023. URL: https://gitlab.univ-lille.fr/claire.soyez-martin/workspace-macros-u8.

[60] J.E. Pin and European Mathematical Society Publishing House ETH-Zentrum SEW A27. *Handbook of Automata Theory: Volume I: Theoretical Foundations; Volume II: Automata in Mathematics and Selected Applications*. EMS Press, 2021. ISBN: 9783985470068.

[61] Jean Eric Pin. *Varieties of formal languages*. Vol. 184. Springer, 1986.

[62] Vaughan R Pratt, Michael O Rabin, and Larry J Stockmeyer. “A characterization of the power of vector machines”. In: *Proceedings of the sixth annual ACM symposium on Theory of computing*. 1974, pp. 122–134.

[63] Michael O Rabin and Dana Scott. “Finite automata and their decision problems”. In: *IBM journal of research and development* 3.2 (1959), pp. 114–125.

[64] Dennis Ritchie. “An incomplete history of the QED text editor”. In: *Report, Computing Sciences Research Center, Bell Laboratories, Murray Hill, NJ, USA, 19xx. URL http://plan9. bell-labs. com/who/dmr/qed. html. Robbins* (1995).

[65] M. P. Schützenberger. “On finite monoids having only trivial subgroups”. In: *Information and control* 8 (1965), pp. 190–194.

[66] Thomas Schwentick, Denis Thérien, and Heribert Vollmer. “Partially-ordered two-way automata: A new characterization of DA”. In: *Developments in Language Theory: 5th International Conference, DLT 2001 Wien, Austria, July 16–21, 2001 Revised Papers 5*. Springer. 2002, pp. 239–250.

[67] Olivier Serre. “Vectorial languages and linear temporal logic”. In: *Theoretical computer science* 310.1-3 (2004), pp. 79–116.

[68] Claude E Shannon. “A symbolic analysis of relay and switching circuits”. In: *Electrical Engineering* 57.12 (1938), pp. 713–723.

[69] Janos Simon. “On feasible numbers (preliminary version)”. In: *Proceedings of the ninth annual ACM symposium on Theory of computing*. 1977, pp. 195–207.

[70] Howard Straubing. *Finite automata, formal logic, and circuit complexity*. Springer Science & Business Media, 1994.

[71] Pascal Tesson and Denis Thérien. “Diamonds are forever: The variety DA”. In: *Semigroups, algorithms, automata and languages*. World Scientific, 2002, pp. 475–499.

[72] Denis Therien and Pascal Tesson. “Logic meets algebra: the case of regular languages”. In: *Logical Methods in Computer Science* 3 (2007).

[73] Denis Thérien and Thomas Wilke. “Over words, two variables are as powerful as one quantifier alternation”. In: *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*. 1998, pp. 234–240.

[74] Ken Thompson. “Programming techniques: Regular expression search algorithm”. In: *Communications of the ACM* 11.6 (1968), pp. 419–422.

[75] Heribert Vollmer. *Introduction to circuit complexity: a uniform approach*. Springer Science & Business Media, 1999.

[76] Xiang Wang et al. “Hyperscan: A fast multi-pattern regex matcher for modern {CPUs}”. In: *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 2019, pp. 631–648.

[77] Thomas Wilke. “Classifying discrete temporal properties”. In: *Annual symposium on theoretical aspects of computer science*. Springer. 1999, pp. 32–46.

[78] Sun Wu and Udi Manber. “Agrep–a fast approximate pattern-matching tool”. In: *Usenix Winter 1992 Technical Conference*. 1992, pp. 153–162.

[79] Sun Wu and Udi Manber. “Fast text searching: allowing errors”. In: *Communications of the ACM* 35.10 (1992), pp. 83–91.

Contents

From semigroup theory to vectorization: recognizing regular languages

Abstract

The pursuit of optimizing regular expression validation has been a long-standing challenge, spanning several decades. Over time, substantial progress has been made through a vast range of approaches, spanning from ingenious new algorithms to intricate low-level optimizations.
Cutting-edge tools have harnessed these optimization techniques to continually push the boundaries of efficient execution. One notable advancement is the integration of vectorization, a method that leverages low-level parallelism to process data in batches, resulting in significant performance enhancements. While there has been extensive research on designing handmade tailored algorithms for particular languages, these solutions often lack generalizability, as the underlying methodology cannot be applied indiscriminately to any regular expression, which makes it difficult to integrate to existing tools.
This thesis provides a theoretical framework in which it is possible to generate vectorized programs for regular expressions corresponding to rational expressions in a given class. To do so, we rely on the algebraic theory of automata, which provides tools to process letters in parallel. These tools also allow for a deeper understanding of the underlying regular language, which gives access to some properties that are useful when producing vectorized algorithms. The contribution of this thesis is twofold. First, it provides implementations and preliminary benchmarks to study the potential efficiency of algorithms using algebra and vectorization. Second, it gives algorithms that construct vectorized programs for languages in specific classes of rational expressions, namely the first order logic and its subset restricted to two variables.

Keywords: regular expression, algebraic theory of automata

De la théorie des semigroupes à la vectorisation : valider les langages réguliers

Résumé

L'évaluation efficace des expressions régulières constitue un défi persistant depuis de nombreuses décennies. Au fil du temps, des progrès substantiels ont été réalisés grâce à une variété d'approches, allant de nouveaux et ingénieux algorithmes à des optimisations complexes de bas niveau.
Les outils de pointe de ce domaine utilisent ces techniques d'optimisation, et repoussent constamment les limites de leur efficacité. Une avancée notoire réside dans l'intégration de la vectorisation, qui exploite une forme de parallélisme de bas niveau pour traiter l'entrée par blocs, entraînant ainsi d'importantes améliorations de performances. Malgré une recherche approfondie sur la conception d'algorithmes sur mesure pour des tâches particulières, ces solutions manquent souvent de généricité, car la méthodologie sous-jacente à ces algorithmes ne peut pas être appliquée de manière indiscriminée à n'importe quelle expression régulière, ce qui rend difficile son intégration dans les outils existants.
Cette thèse présente un cadre théorique permettant de générer des programmes vectorisés particuliers capables d'évaluer les expressions régulières correspondant aux expressions rationnelles appartenant à une classe logique donnée. L'intérêt de ces programmes vectorisés vient de l'utilisation de la théorie algébrique des automates, qui offre certains outils algébriques permettant de traiter les lettres en parallèle. Ces outils permettent également d'analyser les langages réguliers plus finement, offrent accès à des optimisations des programmes vectorisés basées sur les propriétés algébriques de ces langages.
Cette thèse apporte des contributions dans deux domaines. D'une part, nous présentons des implémentations et des benchmarks préliminaires, afin d'étudier les possibilités offertes par l'utilisation de l'algèbre et de la vectorisation dans les algorithmes d'évaluation des expressions régulières. D'autre part, nous proposons des algorithmes capables de générer des programmes vectorisés reconnaissant les langages appartenant à deux classes d'expressions rationnelles, la logique du premier ordre et sa restriction aux formules utilisant au plus deux variables.

Mots clés : expression régulière, théorie algébrique des automates

---

Inria Lille