A Layered Implementation Framework
for Regular Languages

Baudouin Le Charlier

Université catholique de Louvain, Belgium

baudouin.lecharlier@uclouvain.be

September 24, 2025

Abstract

I present the most fundamental features of an implemented system designed to manipulate representations of regular languages. Some functionalities of the system have been presented in previous papers without describing the low level data structures and algorithms ensuring its efficiency. This latter point is the main subject of the present paper. The system is structured into two layers, allowing regular languages to be represented in an increasingly compact, efficient, and integrated way. Both layers are first presented at a high level, adequate to design and prove the correctness of abstract algorithms. Then, their low-level implementations are described meticulously. Algorithms using the system must be written at this level.

At the high level, the first layer offers a notion of normalized regular expressions ensuring that the set of all syntactic derivatives of an expression is finite. This is convenient to design high-level algorithms to compute automata from expressions, compare expressions for inclusion or equivalence, or even simplify expressions. At the low level, normalized expressions are uniquely represented by identifiers, i.e. by standard integers so that checking syntactic equality boils down to checking equality of integers. High-level operations on normalized expressions are implemented by algorithms working on integers. High-level algorithms working on normalized expressions can therefore be straightforwardly translated into programs manipulating integers. The contribution of this layer is the simplification of existing algorithms, at the high level, and a significant increase in efficiency, at the low level.

The second layer, called the background, introduces additional notions to record, integrate, and simplify things computed within the first layer, or even outside the system. Therefore, results of previous computations can be reused for solving new tasks more simply and quickly. At the high level, normalized expressions denoting the same regular language can be unified by grouping them into equivalence classes. One shortest expression is chosen in each class as its representative, which can be used to form equations relating expressions to their derivatives. Sets of equations can be used to represent deterministic finite automata (DFAs). Equations are also used as equality constraints over the regular languages denoted by expressions. Solving such constraints may reduce the number of equivalence classes, and, therefore, the sizes of DFAs. At the low level, equations are uniquely represented by integers, similarly to normalized expressions.

In this paper, I focus on describing the low-level data structures and the main algorithms of the two layers without describing applicative algorithms. Some of the latter have been described in previous papers; others are topics for future work. Nevertheless, this paper also presents extensive experimental results to demonstrate the usefulness of the proposed framework and, in particular, the fact that it makes it possible to represent large sets of regular languages in a unified way where distinct identifiers designate different languages, represented by both a small expression and a minimal deteministic automaton (MDFA). It is also shown that such large sets of regular languages can provide interesting statistical information on the way large expressions can be minimized or simplified.

1

Contents

List of Tables

List of Figures

1 Introduction

According to Set theory [16], all mathematical objects are sets. So, for mathematicians, sets are a ubiquitous notion, sufficient to define and reason about anything we can imagine. Nevertheless, to write down definitions and reasonings, we also need symbols and formulas. But these are not always distinguished from the sets themselves, in common practice and elementary textbooks [23]. Some mathematicians even consider that the sets *are* the formulas, actually (see, for instance, [15], Section 0.2. *Le langage réel des Mathématiques*).

In this paper, we consider regular languages. They are possibly infinite sets of strings of letters, which cannot be actually written on paper by enumeration. Instead, we can use finite symbol groupings called regular expressions. Also in this area, some authors confuse the expressions and the sets. This was especially true in the early days of computer science (for example, in [7, 13]), when relatively simple regular expressions were used to help designing electronic circuits by hand [7]. In [13], Conway freely uses regular languages as states of an automaton recognizing a regular language. Although defined as sets of strings, the states can be represented on paper by simple bullets ([13], page 42), confirming that large sets can be viewed as atomic objects. To compute the sets represented by the bullets, it is not practical to reason on their set-theoretical definition but rules are set out, making it possible to compute expressions representing them. Sets are represented by symbols in the rules. This method is completely similar to the derivation rules given in [23] for mathematical functions. Conway writes that *the implied algorithm is extremely efficient and, in many cases, gives the minimal machine directly to anyone skilled in input differentiation* ([13], page 43). The method presents a number of difficulties however: Although it is proven in [13] that the number of derivatives of a regular language is finite, as sets, a mechanical application of the rules may generate infinitely many expressions; many of them represent the same regular language but it can be difficult to check which ones. As for mathematical functions, it is at least necessary to apply simplification rules, otherwise bigger and bigger expressions are generated, making the method impractical.

The methods proposed in [7, 13] may be considered applicable only to relatively small examples and by people "skilled in input differentiation", i.e., primarily, skilled in simplifying expressions. A major improvement to these approaches was proposed later in [4], which lays the foundations for an efficient automation of the method. Here, we want to go further by introducing an implemented framework in which regular languages really are "as simple as bullets", which can be easily and efficiently manipulated for solving problems such as simplifying regular expressions [10, 18, 28], computing deterministic finite automata [1, 7, 9], checking inclusion or equivalence of regular languages [3, 6], or studying statistical properties of regular languages [20].

In more technical terms, the key idea behind the work presented in this paper is to use *unique identifiers* to stand for any kind of objects we want to represent in our system. Identifiers actually are *integers*, which makes it possible to design efficient data structures to relate the objects to their identifiers. Describing these data structures and demonstrating their usefulness is one of the main goals of this work. The framework that we introduce is made of two layers of increasing power and complexity:

1. The first layer is concerned with the notion of *normalized regular expressions*. A high-level definition of these expressions is given as well as operations to work on them. Plain regular expressions can be efficiently translated to normalized expressions and, above all, many equivalent expressions are translated to the same syntactically unique normalized expression (see Theorem 2). At the implementation level, unique identifiers are assigned to normalized expressions, using hashing techniques, so that high-level operations are implemented by algo-

rithms working on integers. This allows an efficient implementation of high-level algorithms working on normalized expressions. As an example, the algorithm of [7] can be implemented, without checking sufficient conditions for the equivalence of expressions as in [7], by testing the equality of their identifiers.

2. The second layer, called *the background*, maintains an ideally large set of normalized expressions partitioned into classes of equivalent ones, i.e. expressions denoting the same regular language. In every equivalence class, a best *representative* is chosen. Additionally, the background may contain *equations* relating some representatives to the representatives of their direct derivatives. These equations are used as constraints on the equivalence classes and they can be grouped to form deterministic automata (DFA) recognizing the regular languages represented in the background. More information can be added to the background by merging equivalence classes and reducing the set of equations when equivalent expressions are discovered by any algorithm working with the objects in the background. On the other hand, the information contained in the background can be used to simplify the task of algorithms applied to objects newly added to it. Equivalence classes are implemented using the Union-Find method [14] applied to identifiers while equations are given identifiers that are useful to use them efficiently as constraints on the equivalence classes.

The rest of this paper is organized as follows. Sections 2 and 3 respectively deal with normalized expressions and the background, presented at both a high level and an implementation level, where the usefulness of identifiers and low-level data structures is stressed. Section 4 describes experiments. Since the goal of this paper is to describe the foundations of the approach independently of any specific application, the experiments concentrate on the background itself, i.e. on what it provides "for free". For instance, it is shown that the equivalence classes of the background can be refined, i.e. merged, in such a way that expressions in two different classes denote different regular languages. In addition, the representatives of classes are small, sometimes minimal, expressions, obtained using the sole fact that two expressions denoting the same regular language belong to the same equivalence class. This method, applied to large sets of randomly generated expressions, gives us precise statistical information on the distribution of regular expressions with respect to their minimal size.

2 First layer: normalized expressions

We first give a high-level definition of normalized expressions and their operations in Section 2.1. Section 2.2 describes their low-level implementation with a complexity analysis and a short experimental evaluation.

2.1 High-level description of normalized expressions and their operations

2.1.1 Regular expressions and normalized expressions

Basic concepts We assume a finite set of *letters*. In practice, in this paper, only lowercase letters, i.e. a, b, ..., z are used. (For the examples, mainly a and b.) A string of letters, denoted by t, u, or w, is a finite sequence $x_1 x_2 \ldots x_n$ where $x_1, x_2, \ldots, x_n$ are letters (n is the length of the string). The empty string ($n = 0$) is denoted by 1. We identify every letter x with the string of length 1 containing x only. The concatenation of two strings u and w, denoted by $u \cdot w$ or, more simply, by $u\,w$, is the string $y_1 y_2 \ldots y_m x_1 x_2 \ldots x_n$ where $u = y_1 y_2 \ldots y_m$ and $w = x_1 x_2 \ldots x_n$. Let

S, S_1, and S_2 be sets of strings. The concatenation of S_1 and S_2, denoted by $S_1 \cdot S_2$, is the set of strings $\{w_1 \cdot w_2 \mid w_1 \in S_1 \ \& \ w_2 \in S_2\}$. The iteration of S, denoted by S^*, is the set of strings $\{w_1 \cdot w_2 \cdot \ldots \cdot w_n \mid w_1, w_2, \ldots, w_n \in S \ (n \geq 0)\}$. The derivative of S with respect to the string w is the set $S_w = \{u \mid w \cdot u \in S\}$. If w is a letter x, we say that S_x is the direct derivative of S with respect to x.

Definition 1. Regular expressions, normalized expressions

1. A (plain) regular expression E is either the symbol 0, or the symbol 1, or a letter, or a *union*, of the form $E_1 + E_2$, or a *concatenation*, of the form $E_1 \cdot E_2$, or an *iteration*, of the form E^*, where E, E_1 and E_2 are (simpler) regular expressions. A regular expression denotes a set of strings in the "obvious way" (see, e.g., [1, 13, 28]). This set is denoted by $\mathcal{L}(E)$ and is called a regular set.

2. A *normalized* regular expression is either the symbol 0, or the symbol 1, or a letter, or a *union*, of the form $E_1 + E_2 + \ldots + E_n$, where $n \geq 2$ and E_1, E_2, $\ldots$, E_n are syntactically different normalized expressions that are neither unions nor equal to 0, or a *concatenation*, of the form $E_1 \cdot E_2$, where E_1 and E_2 are not equal to 0 or 1 and E_1 is not a concatenation, or an *iteration*, of the form E^*, where E is not equal to 0 or 1 and is not an iteration.

   We say that E_1, E_2, $\ldots$, E_n, E_1 and E_2, and E respectively are *the direct subexpressions* of the normalized expressions $E_1 + E_2 + \ldots + E_n$, $E_1 \cdot E_2$, and E^*. The symbols 0, 1, and the letters have an empty set of subexpressions. If E' is a direct subexpression of E, we also say that E is a *superexpression* of E'.

   We sometimes write meta-expressions such as "$E_1 + E_2 + \ldots + E_n$, where $n \geq 0$". This means that the intended expression either is 0 (if $n = 0$), or is neither a union nor 0 (if $n = 1$), or is a union (if $n \geq 2$).

   We assume a total ordering on the set of normalized expressions.[1] Using this ordering, we impose the additional constraint that, in a union $E_1 + E_2 + \ldots + E_n$, the sequence E_1, E_2, $\ldots$, E_n is strictly sorted, in ascending order, with respect to this ordering.

   (End of definition)

Note My implemented system[10] also considers *extended* regular expressions and *normalized extended* regular expressions, which are of the form $E_1 \, \omega \, E_2$, where E_1 and E_2 are regular or extended regular expressions (normalized if appropriate) and ω is a set operator such as $\cap$, $\setminus$, and $\triangle$. Expressions of the form $!E$, where $!$ stands for the complement, are also considered. For simplicity, extended expressions are not considered in this paper.

The set of normalized expressions can be seen as a subset of the set of all regular expressions by identifying every union $E_1 + \ldots + E_n$ to the plain regular expression $E_1 + (E_2 + \ldots + E_n)$ (using right associativity). Thus, the language defined by a normalized expression E is equal to $\mathcal{L}(E')$, where E' is the regular expression to which E is identified. Explicit rules defining $\mathcal{L}(E)$ are

---

[1] In practice, this ordering is fixed by the implementation (see Section 2.2): At the low level, representations of normalized expressions are created one by one in some order, which progressively defines the ordering on expressions. It will become clear later that this choice is the most efficient for implementing the basic operations on normalized expressions.

Figure 1: Set of strings $\mathcal{L}(E)$ denoted by a normalized expression

$$
\begin{aligned}
\mathcal{L}(0) &= \{\} \\
\mathcal{L}(1) &= \{1\} \\
\mathcal{L}(x) &= \{x\} \\
\mathcal{L}(E_1 + \ldots + E_n) &= \mathcal{L}(E_1) \cup \ldots \cup \mathcal{L}(E_n) \quad (n \geq 2) \\
\mathcal{L}(E_1 \cdot E_2) &= \mathcal{L}(E_1) \cdot \mathcal{L}(E_2) \\
\mathcal{L}(E^*) &= (\mathcal{L}(E))^*
\end{aligned}
$$

given in Figure 1. When we write normalized expressions on paper, we write them as the plain expressions to which they are identified and we can drop parentheses as is usually done for plain regular expressions. As a typical example, the expression

$$b(a + b(1 + a + b^*b))((a + b)a^*)^*$$

has to be parsed as $E_1 (E_2 E_3)$ where $E_1 = b$, $E_2 = a + b(1 + a + b^*b)$, and $E_3 = ((a + b)a^*)^*$.

2.1.2   Operations on normalized expressions

We introduce three operations on normalized expressions. They can be used, at a high level, for describing algorithms working on normalized expressions. Since 0, 1, and the letters already are normalized, normalizing plain expressions can be done recursively, using these operations, by induction on the structure of plain expressions.

   The operations *union*, *concat*, and *iter* take one or two normalized expressions as argument(s), and return a uniquely defined normalized expression denoting the union, the concatenation, or the iteration of the language(s) denoted by their argument(s). For the sake of beauty, we usually write these operations as the infix operators $\oplus$ (*union*) and $\odot$ (*concat*), and the postfix operator $\star$ (*iter*).

Definition 2. Operations on normalized regular expressions

Let E, E_1, and E_2 be normalized expressions.

- The operation *union* ($\oplus$)

   1. $union(0, E) = union(E, 0) = E$
   2. Assume that E_1 and E_2 are different from 0.
      For $i = 1, 2$, let $S_i = \{E_{i\,1}, \ldots, E_{i\,n_i}\}$ where $E_i = E_{i\,1} + \ldots + E_{i\,n_i}$, if E_i is a union, and let $S_i = \{E_i\}$, otherwise. Let $F_1, \ldots, F_m$ be the strictly ordered sequence of normalized expressions such that $S_1 \cup S_2 = \{F_1, \ldots, F_m\}$. Then, by definition, $union(E_1, E_2)$ is the normalized expression $F_1 + \ldots + F_m$, if $m \geq 2$. Note that we can have $m = 1$. In that case, the result simply is F_1, which is not a union by definition of normalized expressions.

- The operation *concat* ($\odot$)

   1. $concat(0, E) = concat(E, 0) = 0$

2. $concat(1, E) = concat(E, 1) = E$

3. Assume that E_1 and E_2 are different from 0 and 1.

   If E_1 is not a concatenation, then $concat(E_1, E_2)$ is the concatenation $E_1 . E_2$. Otherwise, E_1 can be written as $F_1 . F_2$, where F_1 is not a concatenation. In that case, $concat(E_1, E_2) = F_1 . G$ where $G = concat(F_2, E_2)$.

- The operation *iter* ($\star$)

  1. $iter(0) = iter(1) = 1$

  2. If E is an iteration, $iter(E) = E$.

  3. If E is not an iteration and is different from 0 and 1, $iter(E) = E^*$.

(End of definition)

We conclude this section by two theorems (proven in Appendix A) stating the main mathematical properties of normalized expressions and their operations. It is important to understand that the symbol =, used between two meta-expressions denoting normalized expressions, denotes strict syntactic equality of these normalized expressions.

Theorem 1 *Properties of the operations* $\oplus$, $\odot$, *and* $\star$

1. *The operation union is associative, commutative, and idempotent. (So, we can freely write $E_1 \oplus E_2 \ldots \oplus E_n$ without paying attention to the position of the expressions E_i in the whole expression. Moreover, it does not matter if we repeat the same subexpression several times.)*

2. *The operation concat is associative. So, for any normalized expressions E_1, E_2, E_3, we can write:*
$$E_1 \odot E_2 \odot E_3 = (E_1 \odot E_2) \odot E_3 = E_1 \odot (E_2 \odot E_3).$$

3. *Let E, E_1, and E_2 be normalized expressions. The following equalities hold:*

$$\mathcal{L}(E_1 \oplus E_2) = \mathcal{L} E_1 \cup \mathcal{L} E_2 \qquad \mathcal{L}(E_1 \odot E_2) = (\mathcal{L} E_1).(\mathcal{L} E_2) \qquad \mathcal{L}(E\star) = (\mathcal{L}(E))^*$$

Definition 3. A congruence relation on regular expressions

We define the relation $\cong$ as the least congruence on the set of plain regular expressions that contains the following equivalences, for any regular expressions E, E_1, E_2, and E_3:

$$E + E \cong 0 + E \cong E \cong E + 0 \qquad 0 \cdot E \cong 0 \cong E \cdot 0 \qquad 1 \cdot E \cong E \cong E \cdot 1$$

$$E_1 + (E_2 + E_3) \cong (E_1 + E_2) + E_3 \qquad E_1 + E_2 \cong E_2 + E_1$$

$$E_1 \cdot (E_2 \cdot E_3) \cong (E_1 \cdot E_2) \cdot E_3 \qquad 0^* \cong 1^* \cong 1 \qquad (E^*)^* \cong E^*$$

(End of definition)

Theorem 2 *Plain regular expressions that are equivalent with respect to the relation $\cong$ are normalized to the same expression.*

2.2 Implementation of normalized expressions and their operations

The mathematical properties of the operations $\oplus$, $\odot$, and $\star$ ensure that we can compute a deterministic finite automaton from any normalized expression by computing its syntactic derivatives. An algorithm for doing so is described and proven correct in [9]. This algorithm can be seen as a variant, and an improvement, of the methods proposed in [4, 7, 13]. The main improvement is the fact that syntactic equality provides a sufficient condition for the equivalence of expressions that ensures that the set of syntactic derivatives is finite. As a matter of fact, we can go further by providing an implementation of normalized expressions in which syntactic equality can be checked in $O(1)$ time, by associating a unique integer identifier to any expression represented in the system. This identifier is attributed on demand and may change from one run of the system to the next. This issue is the subject of this section. Notice also that, in this section, I write expression instead of normalized expression, for simplicity.

2.2.1 Main implementation choices

As mentioned above, I choose to identify each expression represented in the system by a unique integer. The number of available identifiers is decided at the start of the system. We denote it by the letter M. The available identifiers are then: $0, 1, \ldots, M-1$.

It is also worth mentioning in passing that the system is implemented in Java, as this programming language offers a very flexible notion of array, which is used extensively. In contrast, we make no use of more advanced features of Java, so that the system could easily be reimplemented in most other imperative programming languages, such as C. Since we make relatively little use of objects, except of course arrays, the system also makes relatively little use of the Java garbage collector. In the following, we denote an array of n elements by $t[n]$, where t is its name. Its elements are denoted by $t[i]$, where i denotes an integer such that $0 \leq i < n$. We use the notation $\{v_0, \ldots, v_{n-1}\}$ for describing an array of n values.

2.2.2 Low-level data structures

Before explaining how expressions are represented, we need to describe some low-level data structures that are extensively used in the system. The most useful of them is called `MultiList`. A single object of type `MultiList` is specified by two positive integers n and m, and it implements n *disjoint* lists $list(0)$, $list(1)$, $\ldots$, $list(n-1)$ of integers greater or equal to 0 and less than m. Most of the time $m = M$. When an object `MultiList`(n, m) is created, all its lists are empty. It is possible to add a number into a list, if it does not belong to any list beforehand. It is also possible to remove a number from a list to which it belongs beforehand. These operations are executed in constant time. For convenience, it is allowed to make an attempt to add a number to a list even if it already belongs to it or to another list of the same `MultiList`. In that case, no list is modified. Similarly, an attempt to remove a value not belonging to any list does not change anything. We can also traverse a list to get all its elements in $O(\ell)$ time, where ℓ is the number of elements in the list. Adding and removing elements can be done during the traversal of one or possibly several lists. A newly added element becomes the first of the list. Removing an element does not change the ordering of the others. It is also possible to check in $O(1)$ time whether a number belongs to some unspecified list.

An object `MultiList`(n, m) is implemented thanks to three arrays of integers `first`$[n]$, `pred`$[m]$, and `succ`$[m]$. The values in these arrays are determined by the following rules: `first`$[i]$ is the first element in $list(i)$, if the list is not empty; it is equal to -1, if the list is empty. If an integer i belongs

to some list, `pred`[i] is the number preceding i in the list, if i is not the first element in the list, and `pred`[i] $= -1$, otherwise. Similarly, `succ`[i] is the number following i in the list, if i is not the last element in the list, and `succ`[i] $= -1$, otherwise. If an integer i such that $0 \leq i < m$ does not belong to any list, the equality `pred`[i] $= 0 =$ `succ`[i] holds. On the basis of these rules, it is easy to implement the operations specified above within the announced complexity constraints.

The implementation of expressions also uses two other kinds of objects. An object `OneList`(m) is equivalent to an object `MultiList`($1, m$) except that the operations need one argument less, since the list number implicitly is 0. On the other hand, we need an object called `TwoLists`(m), which is used as a pool of identifiers of expressions (with $m = M$). It is implemented with an object `MultiList`($2, m$). The list $list(0)$ contains the identifiers currently in use, while $list(1)$ contains the identifiers that are free to be used. We can choose an identifier to use, either explicitly, if it is not in use, or implicitly as the first free identifier. We can also return a no longer needed identifier to the free list. This is needed to implement a hand-crafted garbage collector for the system (see Appendix B.2).

2.2.3   Internal representation of expressions

Let us assume a finite set of expressions that we call "expressions currently represented in the system". It is required that all their subexpressions also belong to this set. Moreover, we assume that each expression E has received an identifier, i.e. an integer iE such that $0 \leq iE < M$, different for different expressions. Such a set is implemented thanks to the following objects.

`tabNexpr`[M][] is an array of integer arrays. Let iE be the identifier of an expression E, currently represented in the system. Then, `tabNexpr`[iE] is an array of integers containing the identifiers of the direct subexpressions of E.

`type`[M] is an array of small integers representing the types of the expressions. Types are the following constant values: ZERO, ONE, LETTER, UNION, CONCAT, STAR.

`tabCode`[M] is an array of integers. With the same conventions as above, the integer `tabCode`[iE] is the hash code of the expression E. Hash codes are computed as integer functions of the identifiers of the direct subexpressions of expressions, and of the integer representing their type (except that, for letters, the identifier of the letter is used; see below).

`iExprList`(M) is a `TwoLists`(M) object, determining the set of identifiers of the expressions currently represented in the system, and, complementarily, the set of free identifiers. The atomic expressions 0, 1, a, ..., z are given the identifiers 0, 1, 2, ..., 27, respectively. Those values are added to `iExprList` before creating any other expression.

`hashTable`(n, M) is a `MultiList`(n, M) object used as a hash table to determine if an expression currently is represented in the system. The value of n is chosen big enough to ensure that checking if an identifier is contained in the hash table can be done in $O(1)$ time, on the average.

2.2.4   Implementation of the operations on expressions

We have already said how atomic expressions are represented. Their identifiers are the same in all executions of the system. Now, let us explain how the operations $\oplus$, $\odot$, and $\star$ are implemented. The algorithms are low-level implementations of the abstract algorithms of Definition 2. The fundamental change is the fact that the arguments and result are identifiers of expressions instead of expressions.

It is also necessary to check if the result of an operation already exists or if it has to be created, which is meaningless in the abstract case.

The first step of each operation consists of determining the type of the resulting expression and computing an array containing the identifiers of its direct subexpressions. This is done as in Definition 2 but pairs of expressions and strictly ordered sequences of expressions are represented by arrays of integers. The total order on expressions, left unspecified in Definition 1, is thus materialized by the usual order on integers. This implementation choice is the most efficient possible.[2] Having the type of the result and the array containing the identifiers of its subexpressions, its hash code h is computed, and used to traverse the list $list(i)$ of the hash table, where $i = \text{mod}(h, n)$ for some appropriate function mod ensuring that $0 \leq i < n$. For all identifiers iE in $list(i)$, the value $\texttt{type}[iE]$ and possibly the array $\texttt{tabNexpr}[iE]$ are respectively compared to the type and to the array freshly computed. In case of equality, the result already exists so that the identifier iE is returned by the operation. If no comparison succeeds, a new identifier iE is chosen from $\texttt{iExprList}$, and the elements $\texttt{tabNexpr}[iE]$, $\texttt{type}[iE]$, and $\texttt{tabCode}[iE]$ are initialized with the newly computed array, type, and hash code. Also, the identifier iE is added to $list(i)$ and, finally, returned as the result of the operation.

2.2.5 Complexity of normalization

In this paper, normalized expressions are proposed as an efficient representation of plain regular expressions for solving problems such as computing DFAs ([9]) and simplifying expressions ([10]). They constitute the first layer of a more expressive data structure, described in the next section. In any case, as a first step, it is interesting to provide arguments about its efficiency as a stand-alone data structure. Therefore, we first discuss the time and space complexity of executing a single operation $\oplus$, $\odot$, or $\star$. Next, we consider the overall complexity of algorithms based on these operations. Finally, statistics on concrete experiments are presented.

Let us call the *terms* of a normalized expression E the expressions $E_1, \ldots, E_n$ such that E is syntactically equal to $E_1 + E_2 + \ldots + E_n$ ($n \geq 0$) (see Definition 1). Similarly, let us call the *factors* of a normalized expression E, the expressions $E_1, \ldots, E_n$ such that E is syntactically equal to $E_1 \cdot (E_2 \cdot (\ldots E_n))$ ($n \geq 0$). (The case $n = 0$ holds when $E = 1$, and the case $n = 1$ takes place when E is not a concatenation.) With this terminology, the time and space complexity of computing $E_1 \oplus E_2$ is $O(n_1 + n_2)$, where n_1 and n_2 are the numbers of terms in E_1 and E_2. Similarly, the time and space complexity of computing $E_1 \odot E_2$ is $O(n_1)$, where n_1 is the number of factors in E_1. Also, the complexity of computing $E\star$ is $O(1)$. In all cases, the complexity is due to building new arrays $\texttt{tabNexpr}[iE]$, computing hash codes, and traversing lists in the hash table. The results hold on the average provided that the lists contain $O(1)$ elements.

Now, let us discuss the complexity of algorithms working on normalized expressions and making intensive use of the operations $\oplus$, $\odot$, and $\star$. In fact, it is not possible to draw any general conclusion since the kind of expressions used may differ from an application to another (see, for instance, [9, 10]). But, as a first approach, we can consider the simple problem of normalizing a plain regular expression chosen at random. For such expressions, the probability that a subexpression is normalized to an expression containing many terms or many factors is low. Therefore, normalizing a plain expression has an average complexity close to that of constructing a copy of the plain expression, i.e. it is close to linear in the size of the expression. However, there are a few worst cases where the complexity is quadratic. An example is the expression $(C_1 + (C_2 + \ldots + (C_{n-1} + C_n) \ldots))$, where the C_i

---

[2]As a consequence of this choice, the same expression can be represented differently in different runs of the system, depending on what expressions have been created beforehand.

Table 1: Complexity of normalization

$size$	t_r	t_T	t_N	$\lvert sub \rvert$	$\sharp iE$	$\sharp empty$	$\sharp nempty$	$\sharp iter$	$\sharp dejaVu$
8	0.17	0.40	0.44	1.84	$2,022$	$1,994$	$30,542$	$30,542$	$30,542$
16	0.20	0.85	0.89	2.01	$20,234$	$20,203$	$50,577$	$50,577$	$50,574$
32	0.30	1.61	1.91	2.10	$61,333$	$60,848$	$89,799$	$90,015$	$89,342$
64	0.45	2.85	3.77	2.14	$137,937$	$135,991$	$175,819$	$176,881$	$173,901$
128	0.79	5.91	7.52	2.17	$286,821$	$281,059$	$352,482$	$378,982$	$346,748$
256	1.36	11.2	15.7	2.18	$578,190$	$561,321$	$716,449$	$722,859$	$699,608$
512	2.62	21.9	33.2	2.19	$1,147{\times}10^3$	$1,092{\times}10^3$	$1,476{\times}10^3$	$1,517{\times}10^3$	$1,420{\times}10^3$
1K	5.24	45.4	70.4	2.19	$2,260{\times}10^3$	$2,067{\times}10^3$	$3,078{\times}10^3$	$3,464{\times}10^3$	$2,885{\times}10^3$
2K	10.2	88.4	152	2.20	$4,435{\times}10^3$	$3,751{\times}10^3$	$6,548{\times}10^3$	$7,810{\times}10^3$	$5,864{\times}10^3$
4K	21.5	169	349	2.20	$8,709{\times}10^3$	$6,379{\times}10^3$	$14,224{\times}10^3$	$17,582{\times}10^3$	$11,894{\times}10^3$
8K	51	365	783	2.20	$17,086{\times}10^3$	$11,602{\times}10^3$	$29,620{\times}10^3$	$39,757{\times}10^3$	$24,136{\times}10^3$

are small expressions that are not unions. For such examples, it is better to first build an array $\{iC_1',\ iC_2',\ \ldots,\ iC_n'\}$ of identifiers by normalizing $C_1, C_2, \ldots, C_n$, independently. This array can be sorted in $O(n \log n)$ time, which allows us to normalize the whole expression at worst in $O(s \log s)$ time, where s is the size of the expression. A similar optimization can be done for concatenation with a large number of factors, but, in this case, it is sufficient to apply the operator $\odot$ from right to left, to ensure a linear complexity.

The discussion above suggests that it is useful to implement two n-ary versions of the operations $union$ and $concat$. The space and time complexity of computing $union(iE_1, \ldots, iE_n)$ is $O((m_1 + \ldots + m_n) \log n)$, where m_i is the number of terms of E_i $(1 \leq i \leq n)$. The complexity of $concat(iE_1, \ldots, iE_n)$ is linear in the total number of factors of $E_1, \ldots, E_n$.

Experiments carried out to support the previous considerations are presented in Tables 1 and 2. Large files of $10,000$ plain expressions of sizes 8, 16, $\ldots$, $8K (= 8,192)$ using at most two letters have been normalized. (Additional information about the test environment is given in Section 4.) Expressions in the files are written in infix notation on a single line. They are read one after the other and parsed to give a plain regular expression represented in the form of a binary tree. Then, the tree is traversed recursively to normalize the expression. All normalized expressions generated by this process are kept in memory. The following values are collected in the tables: the size of expressions in the file ($size$), the average times needed to read an expression from the file (t_r), parse the line and build the binary tree (t_T), and normalize the expression (t_N). Times are given in microseconds. The column $\lvert sub \rvert$ gives the average number of direct subexpressions of an expression, i.e. the number of elements of the arrays `tabNexpr`$[iE]$. The column $\sharp iE$ contains the total number of identifiers in use at the end of the normalization process. The last four columns provide information about the hash table usage: the columns $\sharp empty$ and $\sharp nempty$ respectively give the number of times that an empty or non empty list was searched for an existing identifier. The column $\sharp iter$ gives the number of iterations in the lists of the hash table. Finally, the column $\sharp dejaVu$ provides the number of times that a newly normalized expression was found syntactically equal to an existing one, so that no new identifier was needed. Table 1 corresponds to the case where normalization is done

Table 2: Complexity of normalization (opt)

$size$	t_{r}	t_T	t_N	$\lvert sub \rvert$	$\sharp iE$	$\sharp empty$	$\sharp nempty$	$\sharp iter$	$\sharp dejaVu$
8	0.16	0.40	0.61	1.84	$2,022$	$1,994$	$27,209$	$27,209$	$27,209$
16	0.21	0.82	1.33	1.99	$18,494$	$18,460$	$40,361$	$40,361$	$40,355$
32	0.30	1.60	2.54	2.06	$50,697$	$50,427$	$69,580$	$70,471$	$69,338$
64	0.44	2.99	4.97	2.09	$108,839$	$107,648$	$135,461$	$135,825$	$134,298$
128	0.74	5.72	10	2.10	$220,628$	$217,027$	$270,320$	$272,447$	$266,747$
256	1.29	11.7	20.5	2.11	$440,462$	$429,868$	$547,967$	$556,454$	$537,401$
512	2.59	22	41.7	2.11	$867,830$	$836,295$	$1,121 \times 10^3$	$1,197 \times 10^3$	$1,090 \times 10^3$
$1K$	4.90	42.7	86.5	2.12	$1,705 \times 10^3$	$1,595 \times 10^3$	$2,323 \times 10^3$	$2,443 \times 10^3$	$2,214 \times 10^3$
$2K$	10.1	84.7	176	2.12	$3,338 \times 10^3$	$2,946 \times 10^3$	$4,891 \times 10^3$	$5,479 \times 10^3$	$4,500 \times 10^3$
$4K$	21.1	166	382	2.13	$6,549 \times 10^3$	$5,181 \times 10^3$	$10,499 \times 10^3$	$12,307 \times 10^3$	$9,130 \times 10^3$
$8K$	88.9	358	982	2.13	$12,835 \times 10^3$	$8,261 \times 10^3$	$23,098 \times 10^3$	$31,419 \times 10^3$	$18,523 \times 10^3$

by a simple recursion on the structure of plain expressions, while Table 2 uses the optimization, explained in Section 2.2.5, allowing us to avoid a quadratic complexity in worst cases.

An examination of the results in the tables confirms that normalization is an efficient and useful process. It also shows that the assumptions stated to justify the average complexity of the method actually hold for expressions chosen at random. For instance, the time needed for normalization is less than two times the time needed to build a plain regular expression, except for large expressions. The times needed for normalization grow almost linearly with the sizes of expressions. They are even better without the optimization.[3] This is greatly explained by the values in the column $\lvert sub \rvert$, which show that the number of terms in expressions is small, on the average. In fact, a more interesting benefit of the optimization is that fewer normalized expressions (mainly unions) are constructed, saving memory space (compare the columns $\sharp iE$). Finally, the figures in the last four columns fully support the claim that the time complexity of using the hash table is $O(1)$ on the average: The identifiers are almost perfectly distributed in the lists and most iterations in the lists are due to the normalization of a plain expression into an existing normalized expression.

3   Second layer: equivalence classes and equations

Here, we go a step further toward our goal of building a "world" where regular languages have a best representation, both simple and unique. To reach this goal, regular languages are given an integer identifier corresponding to both a short normalized expression and the initial state of a DFA for the language. Original techniques used to make it possible are described here but, to ensure that different regular languages are given different identifiers, the implemented system also uses well-known algorithms for minimizing DFAs [1, 17, 25]. In this section, we focus solely on the new techniques. The new and old methods are used together in Section 4, devoted to experiments, showing that the overall goal is achievable.

---

[3]This fact suggests that the best method could be dynamically chosen depending on the number of terms in unions. It would also be more efficient to use an insertion sort for computing the union of a short list of expressions.

A key idea of the approach described below is to maintain a large set of normalized expressions that are systematically simplified and linked to a DFA. This set constitutes a kind of database of regular languages from which useful information can be extracted to efficiently simplify other expressions and compute DFAs for them. In the following, we call this database the *background*. In Section 3.1, the background is described at a high level, with the main operations required to extend it and maintain its consistency. Section 3.2 deals with its implementation, which is efficient and constitutes the most original and challenging contribution of this document.

3.1   High-level description of the background

3.1.1   Content of the background: expressions, equivalence classes, and equations

We define the background as a "mutable" abstract object containing at each moment a finite set of normalized expressions grouped into equivalence classes. The background also contains equations relating expressions in it.

Expressions belonging to the same equivalence class must denote the same regular language. The converse does not hold, in general, although it can be achieved, as shown later in Section 4.2. Morever, each equivalence class contains a unique best expression, called the *representative* of the class. If E is an expression in the background, we denote the representative of its equivalence class by $rep(E)$. The representatives are chosen in such a way that $size(rep(E)) \leq size(E)$. The size of an expression can be defined recursively in an obvious way but several variations can be imagined. That is why no strict definition is given here; it will be for the experiments, in Section 4.

Equations are constructs of the form $E = o + \ldots + x \cdot E_x + \ldots$, where $o \in \{0, 1\}$, the x are distinct letters, and E and the E_x are expressions in the background. We say that E is the *left part* of the equation, while $o + \ldots + x \cdot E_x + \ldots$ is its *right part*. Notice that, technically, a right part is *not* a normalized expression but another kind of formal object built with 0 or 1, letters, and normalized expressions. For every equation in the background, it is required that

1. $E = rep(E)$ and $E_x = rep(E_x)$, for all x.

2. $\mathcal{L}(E) = \mathcal{L}(o) \cup \ldots \cup \{x\} . \mathcal{L}(E_x) \cup \ldots$.

We call these conditions the *invariant* of the background. See Section 3.1.3 for examples. We can see that, intuitively, each equation relates an expression to its direct derivatives but the expressions E_x are not necessarily the exact syntactic derivatives of E, with respect to x, as defined in [9]. Let E be an expression belonging to the background. We say that E has an equation in the background if $rep(E)$ is the left part of an equation in the background. However, it is not necessarily the case that an expression in the background has an equation. Thus, in general, the number of equations in the background is smaller than the number of expressions. However, this is not always true because some equations may overlap: We say that two different equations in the background *overlap* if either their left parts or their right parts are equal (i.e. syntactically identical). A background containing overlapping equations can be refined by merging some equivalence classes. Otherwise, we say that the background is *reduced*. This desirable property should be enforced as soon as possible but it can be locally violated in some operations, as explained in the next subsection.

Sets of equations in the background can be used to represent deterministic finite automata, as suggested in [7, 9, 13]. We say that a set of equations is *complete* if every expression E_x used in the right part of an equation is the left part of an equation in this set. A complete set of equations determines a DFA for all expressions that are the left part of an equation of this set: Left parts are the states and right parts define transition functions to next states and indicate if the corresponding

left parts are accepting states or not. If E is the left part of an equation belonging to a complete set of equations, we say that the smallest complete set of equations containing this equation is *the* DFA of E in the background. If E' is another expression such that $E = rep(E')$, we say that the DFA of E is *a* DFA for E'.

3.1.2 Operations on the background

The background can be extended with new expressions and new equations. On the other hand, it can be made more informative by merging equivalence classes and refining equations accordingly.

New expressions can be added to the background by applying the operations $\oplus$, $\odot$, and $\star$ to expressions in the background. A newly added expression defines a new equivalence class containing this expression only. Similarly, new equations can be added to background by specifying a left part E and a right part $o + \ldots + x \cdot E_x + \ldots$. They must respect the invariant specified in Section 3.1.1. The first condition of the invariant can be easily enforced, but the second condition is under the responsibility of the user of the system. It can be ensured by using the algorithm to compute derivatives described in [9]. A reduced background may lose this property after the addition of a new equation.

Two equivalence classes can be unified to a single one by using the operation *unify*. Let E_1 and E_2 be two expressions belonging to two different equivalence classes of the background. The operation $unify(E_1, E_2)$ unites the two classes into one, as follows: Assuming that $rep(E_1)$ is better than $rep(E_2)$,[4] $rep(E_1)$ becomes the new representative of the new equivalence class. Moreover, $rep(E_2)$ is replaced by $rep(E_1)$ in every equation using it. The old equations are removed from the background. It is the user's responsability to make sure that the precondition $\mathcal{L}(E_1) = \mathcal{L}(E_2)$ holds, since the invariant of the background can be violated otherwise.

Unifying equivalence classes and adding equations may result in a non reduced background. Then, a reduced background can be obtained by applying the operation *reduce* to it: If the background is not reduced, the operation selects two overlapping equations $E_i = o_i + \ldots + x \cdot E_{i\,x} + \ldots$ ($i = 1, 2$). If $E_1 \neq E_2$, it unifies E_1 and E_2; otherwise, it selects two expressions $E_{1\,x}$ and $E_{2\,x}$ that are not equal, and it unifies them. Afterwards, it iterates the same process until a reduced background is obtained. The execution of the operation *reduce* terminates since the number of equivalence classes decreases by one at each iteration. Moreover, the invariant of the background is maintained, since it is maintained after each execution of *unify*. In addition, the final set of equations is made of all equations $rep(E) = o + \ldots + x \cdot rep(E_x) + \ldots$ such that an equation $E = o + \ldots + x \cdot E_x + \ldots$ was in the initial background. Consequently, each complete set of equations is replaced by a new complete set of equations, representing a possibly smaller DFA for the same expressions. It can also be proven that the final set of equivalence classes produced by the operation *reduce* does not depend on what particular pairs of overlapping equations are chosen at each iteration of the algorithm (see Section B.1). This confluence property is nice but not really essential for most applications of the work presented in this paper.

3.1.3 Example

I give an example to show how the background evolves when it is extended with new expressions and equations, and subsequently refined by using the operations *unify* and *reduce*.

---

[4]I intentionally leave undefined the predicate "is better than" here because many definitions make sense. See the discussion at the end of Section 5 and also Appendix C.3. A desirable definition could be "is easier to understand" but it is not formalizable.

Let us start with a minimal background, only containing the atomic expressions 0, 1, a, ..., z spread into 27 equivalence classes. Let us add to the background the expression $E = (1 + a)(ab^*)^*$. Now let us compute the derivatives of E and of its subexpressions using the algorithm of [9].[5] The principle of this algorithm boils down to unfolding expressions to the left until they start with a letter. Expressions starting with the same letter are then grouped to form the derivatives. For the expression $F = (ab^*)^*$, we get:

$$\begin{aligned} F &= 1 + ab^*(ab^*)^* \\ &= 1 + ab^*F \\ &= 1 + a \cdot G \qquad (1) \end{aligned}$$

$$\begin{aligned} G &= b^*F \\ &= F + bb^*F \\ &= 1 + a \cdot G + b \cdot G \quad (2) \end{aligned}$$

The equations (1) and (2) may thus be added to the background. They constitute a complete set of equations, i.e. a DFA, and even an MDFA, for F.

Note The above calculation of equations for F and G makes use of the equality symbol ($=$) in an intuitive and "loose" way. It certainly does not denote syntactic equality as in Definition 1 but we can see that the left and right parts of equations denote the same regular language, which intuitively justifies the conclusion. The algorithm in [9] obtains the same result, by working on normalized expressions using the operations $\oplus$, $\odot$, $\star$, and a set of derivation rules for normalized expressions.

Let us continue by computing derivatives and equations for the expression E:

$$\begin{aligned} E &= (1 + a)F \\ &= F + aF \\ &= 1 + ab^*F + aF \\ &= 1 + a \cdot (F + b^*F) \\ &= 1 + a \cdot H \qquad (3) \end{aligned}$$

$$\begin{aligned} H &= F + b^*F \\ &= 1 + ab^*F + bb^*F \\ &= 1 + a \cdot G + b \cdot G \quad (4) \end{aligned}$$

At this point, we have a DFA for E, made of the three equations (3), (4), and (2). However, the equations (2) and (4) overlap (as defined in Section 3.1.1). Since $size(G) < size(H)$, we replace H by G in all equations. Therefore, equation 4 is removed from the background, and equation 3 is replaced by

$$E = 1 + a \cdot G \quad (3')$$

Now the equations (1) and ($3'$) overlap. Thus, the expressions E and F are unified, which means that E is simplified to F. Equation ($3'$) is removed from the background and we are left with equations (1) and (2), constituting an MDFA for E and F. We also see that the expressions E, F, G, and H are distributed into two equivalence classes $\{E, F\}$ and $\{G, H\}$, of which F and G are the representatives. Notice that exactly the same classes and equations would have been obtained

---

[5]To focus on the important things, we do not develop the computation of the derivatives of 1, a, b, $1 + a$, and b^*.

if the derivatives of E were computed before those of F. But computing the derivatives of only E would have failed to simplify E into F because F is not a derivative of E.

Finally, let us add the expression $U = (a + b)^*$ to the background and compute its derivatives. We get a single new equation:

$$\begin{aligned} U &= 1 + (a + b)\,U \\ &= 1 + a \cdot U + b \cdot U \quad (5) \end{aligned}$$

Although $\mathcal{L}(U) = \mathcal{L}(G)$, the expressions U and E are kept in different equivalence classes because equations (2) and (5) do not overlap. However, we can apply a minimization algorithm [17, 25] to the set of all equations to conclude that $\mathcal{L}(U) = \mathcal{L}(G)$, and thus to safely execute $unify(U, E)$, resulting into the final set of equations:

$$\begin{aligned} F &= 1 + a \cdot U &\quad (1') \\ U &= 1 + a \cdot U + b \cdot U &\quad (5) \end{aligned}$$

and the equivalence classes $\{E, F\}$ and $\{G, H, U\}$, of which F and U are the representatives.

3.2 Implementation of the background

The high-level description of the background, as a tool to represent regular languages in a unified way, looks promising but the reader may doubt that it can be implemented efficiently enough to be usable in practice. In the rest of this section, I describe an efficient implementation. The main issues to be solved and the principles of the techniques used to do so are discussed in Section 3.2.1. The used data structures are presented in Section 3.2.2. Then, the main algorithms are precisely described in Section 3.2.3. Finally, Section 3.2.4 shows that the example of Section 3.1.3 is correctly carried out at the implementation level.

3.2.1 Principles

In summary, we need to optimally implement the operations *unify* and *reduce*. Two subproblems need a special attention: Choosing pairs of overlapping equations and replacing an expression identifier by another, everywhere in the set of all equations. In our implementation, the first subproblem is solved in time $O(1)$, and the second one in time $O(nEq)$ where nEq is the initial number of equations containing the replaced identifier, in the set of equations.[6]

Equations can be represented as pairs made of an identifier iE for the left part, and an array `tIE` of identifiers for the right part. However, we want to represent them uniquely, as done for expressions, to safely reason on sets of equations. Thus, we decide to assign an identifier iEq to every equation represented in the background. Moreover, we also choose to assign an identifier iR to every right part of an equation, because this is very convenient to detect that two equations overlap. All in all, we represent an equation by a pair $\langle iE, iR \rangle$ of identifiers, to which we assign a "top-level" identifier iEq. Now, it can be explained how detecting and choosing a pair of overlapping equations is done. For every identifier iE and every identifier iR, we maintain a list of all identifiers iEq of equations, i.e. pairs $\langle iE, iR \rangle$, using it. When the background is reduced, all those lists contain at most one element. We also provide a list of the identifiers iE corresponding to expressions E that are the left part of at least two equations, and a similar list for the identifiers iR of right parts. Using all this, we can detect in $O(1)$ time that at least two equations overlap and retrieve their identifiers as the first two elements of a list.

---

[6] We use the fact that the number of letters in the alphabet is bounded.

Now, let us explain how to solve the problem of replacing an identifier iE_2 by another identifier iE_1 in all equations. We need to quickly find all identifiers iEq of an equation of which iE_2 is the left part or having a right part containing iE_2. For the left part, we already have a list of all equations using iE_2. To find equations with right parts containing iE_2, we provide, for every letter x and every identifier iE, a list of all identifiers iEq of equations with a right part containing an expression E_x of which iE is the identifier. Using those lists, we can retrieve all equations using iE_2 in a number of steps equal to the number of these equations. Each time a pair $\langle iE, iR \rangle$ is retrieved, it is removed from the background and its identifier is removed from all the lists containing it. It is replaced by another pair $\langle iE', iR' \rangle$ using iE_1 instead of iE_2,[7] with a new identifier, unless this other pair already exists in the background. All necessary operations can be done in $O(1)$ time if we consider that the number of letters is bounded.

Finally, we also need a way to represent equivalence classes of expressions with an efficient method to merge them. Unsurprisingly, we use the well-known Union-Find method [19], which only requires a single array of M integers.

3.2.2 Data structures

`tabIE`$[M]$, `tabIR`$[M]$, and `tabTIE`$[M][]$ are arrays representing the equations in the background. With the notation of Section 3.2.1 the following equalities must hold:

$$\texttt{tabIE}[iEq] = iE, \quad \texttt{tabIR}[iEq] = iR, \quad \text{and} \quad \texttt{tabTIE}[iR] = \texttt{tabIE},$$

where iE is the identifier of the left part of an equation, `tabIE` is an array of integers representing its right part, and iR is the identifier of `tabIE`. The array `tabIE` represents the right part of the equation as follows: `tabIE`$[0] = o$,[8] `tabIE`$[1] = iE_a$, `tabIE`$[2] = iE_b$, ..., where iE_x stands for the identifier of E_x.

`hashEq`(n, M), and `hashTIE`(n, M) are `MultiList`(n, M) objects serving as hash tables for the identifiers of equations and right parts of equations. The hash codes are computed similarly as for expressions, based on the identifiers of expressions in the equations. The value of n is chosen as for expressions (see Section 2.2.3).

`nextIR`(M) and `nextIEq`(M) are `TwoLists`(M) objects determining the sets of identifiers of right parts and equations currently represented in the system. They are used similarly to `iExprList`(M) for expressions (again, see Section 2.2.3).

`list_IEQ_IE`(M, M) and `list_IEQ_IR`(M, M) are `MultiList`(M, M) objects containing the lists of identifiers of equations corresponding to each identifier iE of a left part, and to each identifier iR of a right part.

`list_IEQ_IE_x`$[nl](M, M)$ is an array of nl objects `MultiList`(M, M) providing, for every letter x and every expression identifier iE, the list of identifiers iEq of equations such that iE is the identifier of E_x in the equation. The integer nl is the number of different letters allowed in expressions and it is assumed that those are the first nl letters in the alphabet.

---

[7]We set $iE' := iE_1$, if $iE = iE_2$, and $iE' := iE$, otherwise. Moreover, iR' is the identifier of a right part obtained by replacing E_2 by E_1 in the right part identified by iR. Thus, $iR' = iR$, if this right part does not use E_2.

[8]We identify the symbols 0 and 1 with their identifiers, i.e. the integers 0 and 1.

`list_IE_2IEq`(M) and `list_IR_2IEq`(M) are `OneList`(M) objects respectively providing the list of all identifiers iE of expressions that are the left part of at least two equations and the list of all identifiers iR of right parts used by at least two equations.

`tree`[M] is an array of integers representing the equivalence classes of the background. Identifiers corresponding to a single equivalence class are grouped to form a tree of which the representative of the class is the root. If iE is such an identifier, the condition `tree`[iE] < 0 must hold. For another identifier, it is required that `tree`[iE] $= iE'$ where iE' is above iE in the tree. When merging two classes, we choose the smallest one as the new representative. This is not the optimal way to maintain a minimal depth of the tree. In practice however, this method is acceptable and the depth seldom is greater than 1 since we redirect nodes to the representative every time it is computed.

`size`[M] is an array of long integers such that `size`[iE] $= size(E)$, where iE is the identifier of E. It is wise to use long integers because syntactic derivatives of expressions can be very large.

3.2.3   Algorithms

We are now in a position to provide a description of the algorithms implementing the operations *unify* and *reduce* as well as some subproblems helping their implementation. Ordinary language is prefered to pseudocode but, to show that it is sufficently precise, we provide an example of Java code in Appendix B.3.

findIEq(iE, iR) and *findIR*(`tabIE`) respectively check if a pair $\langle iE, iR \rangle$ representing an equation or an array `tabIE` representing the right part of an equation, are represented in the background. Their identifier is returned when the check is positive. The integer -1 is returned, otherwise.

A hash code h is first computed for $\langle iE, iR \rangle$ or `tabIE`. Then, it is reduced to $i = \mod(h, n)$. Afterwards, the list $list(i)$ of the object `hashEq`(n, M) or `hashTIE`(n, M) is traversed, depending on the case. Each iteration delivers an integer id that is used to check if $\langle iE, iR \rangle$ or `tabIE` already exists in the background by checking if `tabIE`[id] $= iE$ and `tabIR`[id] $= iR$, or if `tabTIE`[id] $=$ `tabIE`. As soon as the check succeeds, the identifier id is returned. The integer -1 is returned if the check never succeeds.

addEq(iE, `tabIE`) adds an equation to the background, unless it already belongs to it. The integer iE is the identifier of the left part of the equation, and `tabIE` is an array representing its right part, as explained in Section 3.2.2. It is assumed that all expressions used in the equation are representatives.

1. We call *findIR*(`tabIE`) to get iR. If $iR \neq -1$, we call *findIEq*(iE, iR) to get iEq; then, if $iEq \neq -1$, we stop since the equation already exists in the background.

2. If $iR = -1$, we choose a new identifier in `nextIR` as the actual value of iR. We set `tabTIE`[iR] := `tabIE`; and we add iR to the list $list(i)$ of `hashTIE`, where $i = \mod(h, n)$ and h is the hash code of `tabIE`.

3. We call *findIEq*(iE, iR) to get iEq. If $iEq = -1$, we choose a new identifier in `nextIEq` as the actual value of iEq, we set `tabIE`[iEq] := iE and `tabIR`[iEq] := iR, and we add iEq to the list $list(i')$ of `hashEq`, where $i' = \mod(h', n)$ and h' is the hash code of $\langle iE, iR \rangle$.

4. We add iEq to the list $list(iE)$ of `list_IEQ_IE` and to the list $list(iR)$ of `list_IEQ_IR`.

5. If the list $list(iE)$ of `list_IEQ_IE` contains two elements or more, we add iE to the list `list_IE_2IEq`. Similarly, we add iR to `list_IR_2IEq` if the list $list(iR)$ of `list_IEQ_IR` contains at least two elements.

6. We add iEq to the list $list(\mathtt{tabIE}[i+1])$ of every `MultiList list_IEQ_IE_x`$[i]$ where $0 \le i < nl$.

removeEq(iEq) removes the equation identified by iEq from the background.

1. We retrieve the pair $\langle iE, iR \rangle$ and the array `tabIE`, representing the equation by setting $iE := \mathtt{tabIE}[iEq]$, $iR := \mathtt{tabIR}[iEq]$, and $\mathtt{tabIE} := \mathtt{tabTIE}[iR]$.

2. We remove iEq from `hashEq`, `list_IEQ_IE`, and `list_IEQ_IR` (details are left to the reader).

3. We remove iEq from the list $list(\mathtt{tabIE}[i+1])$ of every `MultiList list_IEQ_IE_x`$[i]$ where $0 \le i < nl$.

4. We return iEq to the list of free identifiers in `nextIEq`.

5. If the list $list(iR)$ of `list_IEQ_IR` is empty, we remove iR from `hashTIE` and we return iR to the list of free identifiers in `nextIR`.

6. If the list $list(iE)$ of `list_IEQ_IE` no longer contains two elements or more, we remove iE from the list `list_IE_2IEq`. Similarly, we remove iR from `list_IR_2IEq` if the list $list(iR)$ of `list_IEQ_IR` does not contain more than one element anymore.

substitute(iE_1, iE_2) replaces all occurrences of iE_2 by iE_1 in all equations of the background. To be precise, we should say that, after executing the operation, the data structures implementing the background correcty implement the "abstract" background obtained by replacing all occurrences of the expression E_2 by E_1 in the initial "abstract" background. It is assumed that, initially, E_1 and E_2 are representatives of their equivalence classes.

1. We traverse the list $list(iE_2)$ of `list_IEQ_IE` and, for every identifier iEq in the list, we do the following:
   (a) We set $iR := \mathtt{tabIR}[iEq]$ and $\mathtt{tabIE} := \mathtt{tabTIE}[iR]$.
   (b) We execute *removeEq(iEq)*.
   (c) We create a new array $\mathtt{tabIE}^{new}$ in which we copy the corresponding elements of `tabIE` except the elements equal to iE_2, which are replaced by iE_1.
   (d) We execute *addEq($iE_1, \mathtt{tabIE}^{new}$)*.

2. For all i such that $0 \le i < nl$, we traverse the list $list(iE_2)$ of `list_IEQ_IE_x`$[i]$ and, for every identifier iEq in the list, we do the same operations as above with the change that iE_1 is replaced by iE, where $iE = \mathtt{tabIE}[iEq]$, in the call to *addEq*.

Remark The attentive reader will have noticed that, in step 1, E_2 necessarily is the left part of all equations retrieved by the loop, so that *addEq* must be called with iE_1 as first argument. To the contrary, in step 2, E_2 cannot be the left part of any retrieved equation because all such equations have already been retrieved and modified in step 1. Therefore, it is correct to call *addEq* with iE. It can possibly be the case that $iE = iE_1$. Moreover, no equation modified in step 1 can be retrieved again in step 2.

reduce implements the abstract operation *reduce* specified in Section 3.1.2 (see also Appendix B.1).

While at least one of the two lists `list_IE_2IEq` and `list_IR_2IEq` is not empty, we do the following:

1. We select two expression identifiers iE_1 and iE_2 as follows:

    (a) If `list_IE_2IEq` is not empty, let iE be its first element, and let iEq_1 and iEq_2 be the first two elements of the list $list(iE)$ of `list_IEQ_IE`; we compare the corresponding elements of the arrays `tabTIE`$[iEq_1]$ and `tabTIE`$[iEq_2]$ until two different elements are found; we choose them as the values of iE_1 and iE_2.

    (b) Otherwise, the list `list_IR_2IEq` must be non empty: Let iR be its first element, and let iEq_1 and iEq_2 be the first two elements of the list $list(iR)$ of `list_IEQ_IR`; we set $iE_1 := $ `tabIE`$[iEq_1]$ and $iE_2 := $ `tabIE`$[iEq_2]$.

2. We assign to iF_1 and iF_2 two different values such that `size`$[iF_1] \leq$ `size`$[iF_2]$, chosen among iE_1 and iE_2.

3. We execute $substitute(iF_1, iF_2)$.

4. We set `tree`$[iF_2] := iF_1$.

$rep(iE)$ implements the operation $rep(E)$.

1. We set $iB := iE$.

2. While `tree`$[iB] \geq 0$, we set $iB := $ `tree`$[iB]$.

3. While `tree`$[iE] \neq iB$, we set $iF := $ `tree`$[iE]$; `tree`$[iE] := iB$; $iE := iF$.

4. We return iB.

$unify(iE_1, iE_2)$ implements the abstract operation $unify(E_1, E_2)$ specified in Section 3.1.2.

1. We set $iE_1 := rep(iE_1)$ and $iE_2 := rep(iE_2)$. If $iE_1 = iE_2$, we stop here.

2. We assign to iF_1 and iF_2 two different values such that `size`$[iF_1] \leq$ `size`$[iF_2]$, chosen among iE_1 and iE_2.

3. We execute $substitute(iF_1, iF_2)$.

4. We set `tree`$[iF_2] := iF_1$.

5. We execute $reduce$.

Note The correctness of the algorithms presented above can be checked by symbolic execution, based on the high-level description of the background in Section 3.1 and its representation by low-level data structures made precise in 3.2.2. It would be somewhat tedious to write down the details of this check explicitly, however, but, as an exercise, the reader could, for instance, check that no explicit update of the objects `list_IEQ_IE`, `list_IEQ_IR`, `list_IR_2IEq`, and `list_IE_2IEq` is needed in the description of the algorithm $reduce$ because all necessary changes are performed by the algorithm $substitute$, through the use of $addEq$ and $removeEq$. It could also be reassuring to look at the example below.

3.2.4 Example (continued)

I complement the example of Section 3.1.3 by illustrating that everything works fine at the low level. I focus on the execution of $reduce$ and $unify$, leaving aside the computation of derivatives, which falls within the scope of [9]. Executing the algorithms step by step would be tedious and anything but enlightening for most people. So, I only raise the tricky points and show that they are correctly solved.

Figure 2: Low-level description of the first situation

$$
\begin{array}{lcl lcl lcl}
\texttt{tabIE}[iEq1] & = & iF & \texttt{tabIR}[iEq1] & = & iR1 & \texttt{tabTIE}[iR1] & = & \{1,\, iG,\, 0\} \\
\texttt{tabIE}[iEq2] & = & iG & \texttt{tabIR}[iEq2] & = & iR2 & \texttt{tabTIE}[iR2] & = & \{1,\, iG,\, iG\} \\
\texttt{tabIE}[iEq3] & = & iE & \texttt{tabIR}[iEq3] & = & iR3 & \texttt{tabTIE}[iR3] & = & \{1,\, iH,\, 0\} \\
\texttt{tabIE}[iEq4] & = & iH & \texttt{tabIR}[iEq4] & = & iR2 & & &
\end{array}
$$

$$
\begin{array}{lcll}
list(iE) & = & (iEq3) & \text{in } \texttt{list\_IEQ\_IE} \\
list(iF) & = & (iEq1) & \text{in } \texttt{list\_IEQ\_IE} \\
list(iG) & = & (iEq2) & \text{in } \texttt{list\_IEQ\_IE} \\
list(iH) & = & (iEq4) & \text{in } \texttt{list\_IEQ\_IE} \\[4pt]
list(iR1) & = & (iEq1) & \text{in } \texttt{list\_IEQ\_IR} \\
list(iR2) & = & (iEq4,\, iEq2) & \text{in } \texttt{list\_IEQ\_IR} \\
list(iR3) & = & (iEq3) & \text{in } \texttt{list\_IEQ\_IR} \\[4pt]
list(iG) & = & (iEq4,\, iEq2,\, iEq1) & \text{in } \texttt{list\_IEQ\_IE\_x}[0] \\
list(iH) & = & (iEq3) & \text{in } \texttt{list\_IEQ\_IE\_x}[0] \\[4pt]
list(iG) & = & (iEq4,\, iEq2) & \text{in } \texttt{list\_IEQ\_IE\_x}[1]
\end{array}
$$

$$
\texttt{list\_IR\_2IEq} \quad = \quad (iR2)
$$

In fact, every high-level situation presented in Section 3.1.3 can be readily translated to a low-level description without requiring any explicit hand simulation of the algorithms of Section 3.2.3, because the translation is completely determined by the representation rules given in Section 3.2.1. Thus, I *a priori* provide the low-level translations of three key high-level situations and I discuss separately the pitfalls to be avoided in moving from one to the other.

The three situations that we consider are 1) the content of the background after the computation of the equation (4), 2) the content of the background after the execution of *reduce*, and 3) the content of the background after executing *unify*(U, G). Their translations are depicted in Figures 2, 3, and 4. In these figures, expressions, equations, and right parts of equations are replaced, i.e. represented, by identifiers that we consistently denote by iE, iF, ..., for expressions, $iEq1$, $iEq2$, ..., for equations, and $iR1$, $iR2$, ..., for right parts. Identifiers are chosen in the free lists of the objects iExprList, nextIEq, nextIR, respectively. The figures only depict the non empty lists. Moreover, the hash tables hashEq and hashTIE respectively contain all identifiers of equations and right parts that are currently in use.

Let us consider the passing from the first to the second situation, done by the operation *reduce*. The fact that equations (2) and (4) overlap is correctly identified by the fact that list_IR_2IEq = $(iR2)$. Thus, H must be replaced by G in equation (4), which must be removed from the background. But the resulting equation is identical to equation (2). There is a risk to create a copy of (2) here, but the operation *reduce* correctly avoids doing so: A copy of the array $\{1, iG, iG\}$ is created and its hash code is computed to look in the table hashTIE whether an identifier exists for it. It must be noticed that $iR2$, still belongs to this hash table because, after removing the equation (4), $list(iR2) = (iEq2)$ in list_IEQ_IR. Thus, the identifier $iR2$ is returned. Afterwards, an identifier for the new equation is searched in the hash table hashEq, based on the pair of identifiers $\langle iG,\ iR2 \rangle$. The identifier $iEq2$ is returned, showing that the "new" equation already exists. No duplicate is created. Now, after considering the case of equation (4), the operation *reduce* looks at the list $list(H) = (iEq3)$

Figure 3: Low-level description of the second situation

$$\texttt{tabIE}[iEq1] = iF \quad \texttt{tabIR}[iEq1] = iR1 \quad \texttt{tabTIE}[iR1] = \{1,\ iG,\ 0\}$$
$$\texttt{tabIE}[iEq2] = iG \quad \texttt{tabIR}[iEq2] = iR2 \quad \texttt{tabTIE}[iR2] = \{1,\ iG,\ iG\}$$

$$
\begin{array}{lll}
list(iF) = (iEq1) & \text{in} & \texttt{list\_IEQ\_IE} \\
list(iG) = (iEq2) & \text{in} & \texttt{list\_IEQ\_IE}
\end{array}
$$

$$
\begin{array}{lll}
list(iR1) = (iEq1) & \text{in} & \texttt{list\_IEQ\_IR} \\
list(iR2) = (iEq2) & \text{in} & \texttt{list\_IEQ\_IR}
\end{array}
$$

$$
\begin{array}{lll}
list(iG) = (iEq2,\ iEq1) & \text{in} & \texttt{list\_IEQ\_IE\_x}[0] \\
list(iG) = (iEq2) & \text{in} & \texttt{list\_IEQ\_IE\_x}[1]
\end{array}
$$

$$\texttt{tree}[iE] = iF \quad \texttt{tree}[iH] = iG$$

in $\texttt{list\_IEQ\_IE\_x}[0]$ to detect that the right part of equation (3) contains an occurrence of H. A modified copy of $\texttt{tabTIE}[iR3]$, equal to $\{1,\ iG,\ 0\}$, is created and its identifier $iR1$ is retrieved from $\texttt{hashTIE}$. Then, it is checked that no identifier exists for $\langle iE, iR1 \rangle$. So a new identifier $iEq3'$ is chosen in $\texttt{nextIEq}$ and added everywhere it should be. At this point, $\texttt{list\_IR\_2IEq} = (iR1)$; the identifiers $iEq1$ and $iEq3'$ are selected in the list $list(iR1)$ of $\texttt{list\_IEQ\_IR}$. So, E is replaced by F in the equation (3$'$). The identifier $iEq3'$ is removed from all lists to which it belongs and returned to the free list of $\texttt{nextIEq}$. The equation (3$'$), modified, in fact is (1). This is detected as before for (2) and (4). Consequently, the execution of *reduce* stops, giving the situation in Figure 3.

Now, let us consider the passing from the second situation to the third. New identifiers iU, $iEq5$, and $iR5$ are chosen and a new array $\{1,\ iU,\ iU\}$ is created to represent the equation (5). Then, the operation $unify(iU, iG)$ is executed after detecting that $\mathcal{L}(U) = \mathcal{L}(G)$, by the minimization

Figure 4: Low-level description of the third situation

$$\texttt{tabIE}[iEq1'] = iF \quad \texttt{tabIR}[iEq1'] = iR1' \quad \texttt{tabTIE}[iR1'] = \{1,\ iU,\ 0\}$$
$$\texttt{tabIE}[iEq5] = iU \quad \texttt{tabIR}[iEq5] = iR5 \quad \texttt{tabTIE}[iR5] = \{1,\ iU,\ iU\}$$

$$
\begin{array}{lll}
list(iF) = (iEq1') & \text{in} & \texttt{list\_IEQ\_IE} \\
list(iU) = (iEq5) & \text{in} & \texttt{list\_IEQ\_IE}
\end{array}
$$

$$
\begin{array}{lll}
list(iR1') = (iEq1') & \text{in} & \texttt{list\_IEQ\_IR} \\
list(iR5) = (iEq5) & \text{in} & \texttt{list\_IEQ\_IR}
\end{array}
$$

$$
\begin{array}{lll}
list(iU) = (iEq1',\ iEq5) & \text{in} & \texttt{list\_IEQ\_IE\_x}[0] \\
list(iU) = (iEq5) & \text{in} & \texttt{list\_IEQ\_IE\_x}[1]
\end{array}
$$

$$\texttt{tree}[iE] = iF \quad \texttt{tree}[iH] = iG \quad \texttt{tree}[iG] = iU$$

algorithm (from [25] or [17], not to be described here). Since $\text{size}[iU] < \text{size}[iG]$, the equations 1 and (2) are removed from the background and the equation (1) is replaced by $(1')$ giving raise to the situation of Figure 4.

4   Experimental evaluation

Now, I report on experiments conducted to illustrate the power of the implementation framework described in this paper, as well as its computational efficiency. Remember that the first layer of the framework efficiently implements normalized expressions by associating a unique identifier to each of them, which is enough to ensure that the set of their syntactic derivatives is finite (see [9]). This improves the methods proposed in [7, 13] and can be seen as an optimal implementation of the method proposed in [4]. The algorithm to compute the DFA corresponding to an expression is defined and proven correct in [9]. Therefore, it is not reexplained here. However, the implementation of normalized expressions is not explained in [9], nor is the implementation of the background. Those issues are the subject of the present paper. The second layer of the framework, i.e. the background, has also already been used in another paper [10] dedicated to the simplification of regular expressions, using different kinds of elaborate simplification rules. That paper also fails to explain how the background is implemented. This shortcoming is once again remedied here.

To be consistent with the explanations above, we are not going to repeat the experiments previously conducted in [9, 10]. Instead, we focus on experiments that allow us to emphasize the intrinsic strength of the system, namely the fact that expressions and DFAs are intimately integrated thanks to their common unique identifier, which makes it possible to build a "world" (a background) where every represented regular *language* has a unique identifier which is, at the same time, the identifier of an MDFA and a small (often minimal) expression for this language. In Section 4.1, experiments are conducted to show that building DFAs from expressions very much benefits from simplifying expressions beforehand. In Section 4.2, we consider a straightforward simplification algorithm exploiting the fact that, given a background containing sufficiently many randomly generated expressions, most expressions represented in the background are equivalent to another, small and often even minimal expression, also in the background. The integrated structure of the background makes it possible to detect all such equivalences. We show that this method also is quite efficient, on the average, provided that a few simplification rules are applied beforehand. In Section 4.3, we analyze the structure of the background created by applying the algorithm of Section 4.2 to a relatively large set of randomly chosen expressions. This study further explains why the algorithm celebrated here is accurate and fast, on the average. Icing on the cake, it gives us precise information on the statistical distribution of expressions with respect to their simplified if not minimal size.

Remember that the system is programmed in Java. The tests are run on an old MacBook Pro (early 2015) with an Intel Core i5 dual-core running at 2.7 GHz with 8GB of memory. Java version 1.8.0_131 is used. We also fix the number of identifiers available for normalized expressions to $5,000,000$ in all tests. Times are wall-clock times, as they are measured using the method System.nanoTime().

The expressions used in these experiments contain at most two different letters. This restrictive choice makes the experimental results more striking and interesting to analyze. In particular, no bounds are imposed on the size of expressions (mostly derivatives) created by the algorithms, in spite of the fact that the problems are PSPACE-complete [24]. Other experimental results, for expressions using up to eight letters, are given in Appendix C.

Table 3: Computing DFAs from normalized expressions (sizes)

size	$\sharp D_E$	$\sharp D_B$	$\sharp D_O$	$\sharp C_O$	$\sharp Eq_B$	$\sharp Eq_M$	Max_E	Max_B	Max_M	ma_E	ma_B	vu_E	vu_B
8	1	1	0	2	3	3	7	7	7	63%	100%	83%	86%
16	5	4	2	2	4	4	18	17	17	40%	94%	9%	17%
32	8	8	6	1	6	6	34	32	30	15%	71%	4	9
64	15	15	14	1	10	8	74	66	66	2%	34%	0	0
128	32	32	31	1	20	9	273	195	188	5	6%	0	0
256	85	85	84	1	50	11	1,875	1,458	1,336	0	13	0	0
512	309	307	307	1	182	11	$115e^2$	6,527	2,939	0	0	0	0
1K	1,773	1,767	1,767	1	1,055	11	$910e^2$	$645e^2$	2,608	0	0	0	0
2K	$206e^2$	$206e^2$	$206e^2$	1	$126e^2$	16	$228e^4$	$147e^4$	7,207	0	0	0	0
4K	$480e^3$	$572e^3$	$572e^3$	0	$297e^3$	8	$432e^4$	$252e^4$	86	0	0	0	0

4.1 Computing a DFA from an expression

We address the problem of building a DFA or an MDFA from a given expression. Remember that DFAs are represented by complete sets of equations of the form $E = o + \ldots + x \cdot E_x + \ldots$, where the x are letters. In this case, $x \in \{a, b\}$. Roughly speaking, the E_x "are" the derivatives of E with respect to the letter x. Actually, two variants are considered according to the layer of the framework that is used. The fundamental algorithm only requires the first layer and it computes the E_x as the *syntactic derivatives* of E, as defined in [9]. These syntactic derivatives are uniquely defined normalized expressions. This algorithm is called E below. A second algorithm, called B, needs to use the second layer of the framework, i.e. the background: Every equation computed by the fundamental algorithm is rewritten as $rep(E) = o + \ldots + x \cdot rep(E_x) + \ldots$ to be compatible with the existing equations of the background. The possibly modified equation is added to the background, which is then reduced. The set of equations obtained by this second algorithm generally is smaller than the set of derivatives computed by algorithm E, but it is not guaranteed to be minimal. Thus, in order to compute minimal sets of equations (MDFAs), we consider a third algorithm, called M, which computes the equivalence classes of the expressions, used in the equations, denoting the same regular language (see [1], pages $124 - 128$, for the description of such an algorithm). Expressions in the same equivalence class are unified using the *unify* operation of the background, resulting in a minimal set of equations.

To assess the efficiency of these three algorithms, we have created files of $10,000$ randomly generated expressions of equal sizes 8, 16, ..., 8192 ($8K$). The algorithms have been applied to every file, as follows: The expressions are read and normalized one by one; their DFA is then computed (their MDFA, for algorithm M). Importantly, the previously read expressions are kept in the system and their computed DFAs as well. Thus, the results of previous computations can be reused, if applicable. Statistics about these experiments are depicted in Tables 3 and 4. Table 3 provides information about the size of the computed DFAs and the reusability of previous results. The column *size* gives the size of the input expressions before normalization. The columns $\sharp D_E$ and $\sharp D_B$ give the average number of iterations executed by algorithms E and B (each iteration computes a new derivative). The column $\sharp D_O$ gives the same information for a variant of algorithm

Table 4: Computing DFAs from normalized expressions (times)

$size$	t_E^{dfa}	t_B^{dfa}	t_O^{dfa}	t_M^{dfa}	t_E^{der}	t_B^{der}	t_O^{der}	t_M^{der}	t_B^{red}	t_O^{red}	t_M^{red}	t_M^{min}
8	$11\,\mu$	$5.0\,\mu$	$3.7\,\mu$	$22\,\mu$	$1.2\,\mu$	$1.4\,\mu$	$858\,\nu$	$882\,\nu$	$1.4\,\mu$	$844\,\nu$	$1.1\,\mu$	$1.2\,\mu$
16	$19\,\mu$	$16\,\mu$	$13\,\mu$	$13\,\mu$	$9.0\,\mu$	$5.5\,\mu$	$4.9\,\mu$	$5.3\,\mu$	$3.8\,\mu$	$2.9\,\mu$	$3.4\,\mu$	$4.3\,\mu$
32	$28\,\mu$	$51\,\mu$	$31\,\mu$	$46\,\mu$	$17\,\mu$	$17\,\mu$	$15\,\mu$	$14\,\mu$	$7.6\,\mu$	$6.9\,\mu$	$8.3\,\mu$	$8.3\,\mu$
64	$66\,\mu$	$111\,\mu$	$87\,\mu$	$102\,\mu$	$54\,\mu$	$64\,\mu$	$40\,\mu$	$56\,\mu$	$14\,\mu$	$14\,\mu$	$21\,\mu$	$16\,\mu$
128	$174\,\mu$	$193\,\mu$	$190\,\mu$	$158\,\mu$	$160\,\mu$	$146\,\mu$	$142\,\mu$	$116\,\mu$	$22\,\mu$	$23\,\mu$	$35\,\mu$	$26\,\mu$
256	$597\,\mu$	$640\,\mu$	$622\,\mu$	$576\,\mu$	$580\,\mu$	$505\,\mu$	$505\,\mu$	$488\,\mu$	$49\,\mu$	$52\,\mu$	$93\,\mu$	$65\,\mu$
512	$3.4\,m$	$3.8\,m$	$3.6\,m$	$3.6\,m$	$3.3\,m$	$3.4\,m$	$3.2\,m$	$3.2\,m$	$192\,\mu$	$193\,\mu$	$415\,\mu$	$296\,\mu$
1K	$38\,m$	$40\,m$	$39\,m$	$39\,m$	$37\,m$	$37\,m$	$36\,m$	$36\,m$	$1.6\,m$	$1.5\,m$	$4.1\,m$	$2.7\,m$
2K	$1.0\,s$	$1.3\,s$	$1.2\,s$	$1.3\,s$	$998\,m$	$1.2\,s$	$1.1\,s$	$1.2\,s$	$54\,m$	$53\,m$	$141\,m$	$90\,m$
4K	$91\,s$	$126\,s$	$116\,s$	$109\,s$	$91\,s$	$118\,s$	$109\,s$	$104\,s$	$3.9\,s$	$3.6\,s$	$6.9\,s$	$4.2\,s$

B that detects the fact that a computed derivative already has an equation in the background, i.e. the fact that its representative is the left part of an equation. The column $\sharp C_O$ is the average number of times that this check saves an iteration in the algorithm. The columns $\sharp Eq_B$ and $\sharp Eq_M$ provide the average number of equations computed by algorithms B and M. (For algorithm E, this number is equal to $\sharp D_E$.) The values in Max_E, Max_B, and Max_M are the maxima of these numbers. The columns ma_E and ma_B are the number of times that algorithms E and B produce a minimal set of equations (or derivatives, for E), on the average, in percent or in actual value, when appropriate. Finally, vu_E and vu_B are the number of times that a normalized expression has been seen previously. But the corresponding input expressions may be different. In such a case, no computation is performed for the newly read expression, except, of course, the normalization. All values are rounded, and a value such as $432e^4$ is an abbreviation for 432×10^4.

Some comments are worth making. We see that the number of iterations made by the algorithms grows very fast (exponentially), and the number of equations computed by algorithm B as well, but this is not the case after minimization by algorithm M, for which this number stabilizes at 11.[9] This suggests that many input expressions are equivalent, and can possibly be simplified to much smaller expressions, as will be shown in Section 4.2 . We can also note that algorithm B often provides minimal sets of equations when the size of expressions is not too large.

Table 4 provides information about the running time of the same algorithms as above. Depending on the case, times are specified in nanoseconds (ν), microseconds (μ), milliseconds (m), or seconds (s). The columns t_E^{dfa}, t_B^{dfa}, t_O^{dfa}, and t_M^{dfa} provide the average running times of algorithms E, B, O, and M (without the time spent in minimization of the DFA). The columns t_E^{der}, t_B^{der}, t_O^{der}, and t_M^{der} are the times spent in computing syntactic derivatives. The columns t_B^{red}, t_O^{red}, and t_M^{red} are the times spent in algorithms $reduce$ and $unify$. Finally, the column t_M^{min} gives the time spent in algorithm M to minimize the DFA computed by algorithm O. This includes the time spent to compute the sets of equivalent expressions and the time spent to unify them in the background. It must be noted that the algorithm $reduce$ is used by algorithms B and M to reduce the background after adding

---

[9]Since the algorithms need too much space and take too much time when $size$ goes beyond $1K$, we have limited the number of input expressions to 1000 for $size = 2K$, and to 100 for $size = 4K$. No attempt has been made to apply the algorithms to expressions such that $size = 8K$.

Table 5: Computing DFAs from lifted expressions (sizes)

$size$	$\sharp D_E$	$\sharp D_B$	$\sharp D_O$	$\sharp C_O$	$\sharp Eq_B$	$\sharp Eq_M$	Max_E	Max_B	Max_M	ma_E	ma_B	vu_E	vu_B
8	0	0	0	2	3	3	7	7	7	92%	100%	90%	90%
16	4	3	1	2	4	4	18	17	17	73%	98%	37%	40%
32	6	6	4	2	6	6	34	32	30	48%	87%	25%	25%
64	10	10	9	2	9	8	74	66	66	36%	68%	30%	30%
128	18	18	17	1	13	9	270	195	188	36%	54%	32%	32%
256	32	32	31	1	23	11	$1,874$	$1,391$	$1,336$	36%	50%	33%	33%
512	57	57	56	1	38	11	$114e^2$	$6,527$	$2,939$	37%	50%	34%	34%
$1K$	111	111	109	1	72	11	$250e^2$	$180e^2$	$2,608$	37%	50%	34%	34%
$2K$	197	197	196	1	131	11	$362e^3$	$268e^3$	$7,207$	37%	50%	34%	34%
$4K$	206	205	205	1	133	11	$123e^3$	$853e^2$	$1,767$	38%	51%	35%	35%
$8K$	238	238	237	1	159	10	$303e^3$	$212e^3$	$2,126$	38%	51%	35%	35%

an equation to it. In addition, algorithm M uses *unify* to put equivalent expressions into the same equivalence class and compute a best representative for them.

We can see that all times grow exponentially but this is only due to the computation of the syntactic derivatives: Their number grows exponentially but also their size. The times spent in reducing or minimizing sets of equations remain ten times smaller.

Although the exponential behavior observed in Tables 3 and 4 is unavoidable in the worst case, we show in the next experiment that this is not true, on the average, if the randomly chosen expressions are properly simplified beforehand. We first apply a straightforward simplification algorithm, called *lifting*, inspired by the lifting method from [18] and similar to the still simpler algorithm from [20]. This algorithm is linear in the size of the expression. It is best to use it before normalization of the expressions, i.e. outside the background, to save identifiers needed for normalized expressions. As will be shown below, the simplification operated by lifting is sufficient to greatly improve the efficiency of the algorithms analyzed previously. More information about the lifting simplification is given in Appendix D.

Now, we analyze the performance of the previous algorithms modified as follows: After reading an expression, and before normalizing it, we simplify it by lifting. The corresponding results are presented in Tables 5 and 6. It must be stressed that, this time, the algorithms can be, and are, applied to sets of $10,000$ expressions even for sizes equal to $2K$, $4K$, and $8K$.

Looking at Table 5, we observe that the values in the columns $\sharp D_E$, $\sharp D_B$, and $\sharp D_O$ grow less than linearly. This is because the average size of lifted expressions tends to a limit (see [20]). This is also the case for DFAs. But the later phenomenon is independent of lifting: Lifting just makes it cheaper to actually compute the DFAs. The last four columns show that, after lifting, about 50% of the DFAs computed by algorithm B are minimal (38% for E), and that much more lifted expressions are recognized as "déja vu".

Table 6 confirms the conclusions drawn from Table 5: the average time needed to compute DFAs and MDFAs from expressions of virtually any size is quite small. As an example, for $size = 4K$, all times are reduced by more than 4 orders of magnitude. For huge expressions, most of the time is spent in lifting, since the algorithm is linear while the average size of lifted expressions remains approximately the same (70). However, we must not forget that those are average values for perfectly

Table 6: Computing DFAs from lifted expressions (times)

size	t_E^{dfa}	t_B^{dfa}	t_O^{dfa}	t_M^{dfa}	t_E^{der}	t_B^{der}	t_O^{der}	t_M^{der}	t_B^{red}	t_O^{red}	t_M^{red}	t_M^{min}
8	$10\,\mu$	$24\,\mu$	$4.3\,\mu$	$22\,\mu$	$784\,\nu$	$839\,\nu$	$1.1\,\mu$	$974\,\nu$	$872\,\nu$	$734\,\nu$	$724\,\nu$	$1.7\,\mu$
16	$9.2\,\mu$	$13\,\mu$	$13\,\mu$	$25\,\mu$	$4.4\,\mu$	$4.5\,\mu$	$3.9\,\mu$	$4.2\,\mu$	$2.6\,\mu$	$2.9\,\mu$	$3.2\,\mu$	$5.5\,\mu$
32	$20\,\mu$	$42\,\mu$	$30\,\mu$	$31\,\mu$	$9.7\,\mu$	$8.5\,\mu$	$13\,\mu$	$8.7\,\mu$	$5.4\,\mu$	$6.1\,\mu$	$5.6\,\mu$	$7.9\,\mu$
64	$36\,\mu$	$61\,\mu$	$61\,\mu$	$60\,\mu$	$25\,\mu$	$38\,\mu$	$21\,\mu$	$19\,\mu$	$9.0\,\mu$	$9.5\,\mu$	$12\,\mu$	$13\,\mu$
128	$69\,\mu$	$95\,\mu$	$123\,\mu$	$88\,\mu$	$56\,\mu$	$47\,\mu$	$69\,\mu$	$50\,\mu$	$14\,\mu$	$16\,\mu$	$25\,\mu$	$23\,\mu$
256	$132\,\mu$	$182\,\mu$	$187\,\mu$	$188\,\mu$	$124\,\mu$	$131\,\mu$	$121\,\mu$	$125\,\mu$	$22\,\mu$	$22\,\mu$	$39\,\mu$	$33\,\mu$
512	$292\,\mu$	$380\,\mu$	$359\,\mu$	$344\,\mu$	$276\,\mu$	$284\,\mu$	$274\,\mu$	$252\,\mu$	$37\,\mu$	$38\,\mu$	$81\,\mu$	$68\,\mu$
1K	$849\,\mu$	$969\,\mu$	$967\,\mu$	$914\,\mu$	$828\,\mu$	$809\,\mu$	$812\,\mu$	$745\,\mu$	$84\,\mu$	$72\,\mu$	$186\,\mu$	$152\,\mu$
2K	$2.1\,m$	$2.5\,m$	$2.5\,m$	$2.4\,m$	$2.1\,m$	$2.1\,m$	$2.2\,m$	$2.1\,m$	$146\,\mu$	$146\,\mu$	$513\,\mu$	$401\,\mu$
4K	$2.5\,m$	$3.0\,m$	$2.8\,m$	$2.8\,m$	$2.5\,m$	$2.6\,m$	$2.5\,m$	$2.5\,m$	$173\,\mu$	$151\,\mu$	$461\,\mu$	$349\,\mu$
8K	$3.6\,m$	$4.1\,m$	$3.8\,m$	$4.0\,m$	$3.6\,m$	$3.6\,m$	$3.4\,m$	$3.5\,m$	$229\,\mu$	$176\,\mu$	$688\,\mu$	$527\,\mu$

randomly chosen expressions. Worst cases still remain and become harder and harder when the size grows, as suggested by the columns Max_E, and Max_B of Table 5. Note that the column Max_M seems to suggest that even the worst case for MDFAs size tends towards a limit.

4.2 The simplifying power of the background

In this section, the power of the framework for simplifying expressions is both logically explained and experimentally illustrated. Importantly, the studied algorithms do not implement explicit simplification rules as in [10, 18, 28]. Such rules have actually been implemented in the system [10] but the goal here is to show that, in a way, simplification of expressions is "provided for free" by the background being properly used. The key idea is to maintain a state of the background where all expressions denoting the same regular language have the same representative. Such a representative can be viewed as *the* simplification of all expressions in its equivalence class. The power of the method is due to the observed fact that, inside a large set of randomly chosen expressions, including all their subexpressions, many expressions have an equivalent short one, which often is even minimal. This is further discussed in Section 4.3.

4.2.1 The fundamental algorithm *PU*

Let us assume a large set of randomly chosen expressions. In our experiments, these are sets of $10,000$ expressions of fixed sizes: $8, 16, \ldots, 8K$ (the same as in Section 4.1). We add the expressions to the background one by one, while normalizing them. In fact, it is much more efficient to first simplify them by lifting, as explained in Section 4.1. Then we apply the following steps to every subexpression of the newly added expression:

1. We simplify it by *propagation*, that is: We (possibly) create a new equivalent expression by replacing its direct subexpressions by their representatives, using operators $\oplus$, $\odot$, and $\star$.

2. We build the MDFA of the simplified subexpression, as explained in Section 4.1.

3. We unify the newly created MDFA with the global MDFA consisting of the union of the MDFAs of all expressions to which the three steps have already been applied. This can be done reasonably efficiently by associating a hashcode to the MDFA of every processed subexpression. This hashcode only depends on the "shape" of the MDFA, not on the actual values of its derivatives, except for those equal to 0, 1, or a letter, since their identifiers are fixed in the system. For every hashcode h, a list of identifiers of the corresponding expressions is maintained. So we go through the list corresponding to the hashcode of the new subexpression and we check the equivalence of the new subexpression with the expressions in the list. If an equivalent expression is found, the two expressions are unified and a single one is kept in the list. Otherwise, the new subexpression is added to the list. The same treatment is applied to all expressions that are left parts of the equations constituting the MDFA, unless it is detected that these expressions already belong to the global MDFA.

Additional explanations are needed to understand how the method is implemented and why it is efficient. Each subexpression encountered in the process need not be considered more than once. This can speed up the process a lot. Moreover, we should not consider an expression before looking at all its subexpressions, because a lot of time can be saved in computing the MDFA of the expression if it is presimplified by propagation. To ensure both behaviors, we use a handmade data structure that maintains a dependency graph between every encountered expression and its direct superexpressions. All expressions have a counter of their not yet processed subexpressions. When this counter is set to 0, the expression is put into a list of all expressions "ready to be processed". The next expression to be processed is arbitrarily chosen in this list. The implementation of this data structure is similar to the data structure presented in [19], pages $258 - 268$, for implementing a topological sort.

Table 7 gives experimental results about the method just explained, applied to the same test data as in the previous subsection. The algorithm is called PU, where P recalls that propagation is used while U recalls that unification of MDFAs is applied. Since normalization (N) and lifting (L) already can provide a substantial simplification, we also consider them as a basis for comparison. The three columns *ssize (arith)* provide the average size of the simplified expressions, i.e. their arithmetic mean. We see that normalization reduces the size by 28% on the average, independently of the input size. The behaviour of lifting is very different since the average size converges to a limit below 70. Similarly, the average size for algorithm PU tends to a value below 25.5, which is much better. The two colums *ssize (geo)* provide the geometric means, which are much smaller, indicating that many simplified expressions have a size well below the arithmetic mean.[10] The next three columns t report on the average execution times. For normalization, they grow linearly, although the theoretical time complexity of normalization is $O(s \log s)$, where s is the input size. For lifting, the times are similar for small expressions but they become almost three times faster for the largest expressions. They grow less than linearly, while the theoretical complexity of lifting is linear. For algorithm PU, the times grow linearly at the beginning but seem to converge to approximately $4.5\,m$, for large expressions. This can be explained by the fact that PU is applied to lifted expressions, the average size of which tends to a constant (70). Note that we do not include the lifting time in the time for PU. We can observe that the times for PU are not much bigger than the times for computing DFAs, reported in Table 6, in spite of the fact that algorithm PU does much more work, and, in particular, computes MDFAs for all subexpressions of the input expressions. This can be explained, on the one hand, by the fact that a lot of the work made for previous expressions can be reused

---

[10]This also allows the reader to compare the results with the figures reported in [18]. For input expressions where $size = 8$, the geometric mean is equal to 0 because an expression is simplified to 1, the size of which is equal to 0.

Table 7: Simplification of expressions (1)

size	ssize (arith)			ssize (geo)		t			t^{der}	$\sharp min$		
	N	L	PU	L	PU	N	L	PU	PU	N	L	PU
8	71%	4.78	4.49	0.00	0.00	$12\,\mu$	$12\,\mu$	$7.9\,\mu$	$727\,\nu$	43%	84%	99%
16	70%	8.86	7.66	7.96	6.86	$18\,\mu$	$35\,\mu$	$34\,\mu$	$4.3\,\mu$	14%	51%	88%
32	71%	15.49	12.44	12.97	10.28	$26\,\mu$	$29\,\mu$	$98\,\mu$	$11\,\mu$	9	27%	55%
64	71%	24.62	18.00	17.43	12.64	$65\,\mu$	$48\,\mu$	$244\,\mu$	$32\,\mu$	0	29%	50%
128	71%	35.15	21.78	20.42	12.93	$58\,\mu$	$53\,\mu$	$589\,\mu$	$62\,\mu$	0	31%	55%
256	72%	46.66	24.12	22.97	12.99	$125\,\mu$	$99\,\mu$	$1.2\,m$	$160\,\mu$	0	32%	57%
512	72%	56.47	25.18	24.40	12.92	$159\,\mu$	$125\,\mu$	$2.0\,m$	$315\,\mu$	0	33%	59%
1K	72%	63.66	25.53	25.25	12.84	$287\,\mu$	$154\,\mu$	$3.5\,m$	$637\,\mu$	0	33%	59%
2K	72%	67.10	25.11	25.12	12.54	$523\,\mu$	$251\,\mu$	$4.1\,m$	$925\,\mu$	0	33%	60%
4K	72%	69.62	25.31	25.23	12.47	$1.0\,m$	$387\,\mu$	$4.0\,m$	$764\,\mu$	0	34%	60%
8K	72%	69.23	25.23	24.75	12.48	$2.0\,m$	$659\,\mu$	$4.4\,m$	$1.1\,m$	0	34%	60%

for new ones, and, on the other hand, by the fact that propagation often reduces very much the size of expressions before their DFA is computed. This claim is supported by the figures in the column t^{der}. If we compare them with the figures in the corresponding columns in Table 6, we see that they are more than three times lower. Moreover, they constitute only a small part of the total time, contrary to what we see in Figure 6. The last three columns of Table 7 indicate the number of input expressions that are minimized by the algorithms, either in percent or in actual value for small numbers. The method to determine these numbers is described later on, in Section 4.2.3. We see that normalization can minimize only small expressions while lifting and PU minimize up to 34% and 60% of the expressions, respectively. Once again, these proportions tend to a limit when the size of the input expressions grows.

4.2.2 Three weaker variants of the fundamental algorithm

To better understand why algorithm PU is effective, it is useful to compare it to three other algorithms, which implement only some of its functionalities. The algorithm PU has been previously described by three steps 1, 2, and 3, executed for all subexpressions. Here, we consider algorithms M, which only applies the step 2, P, which applies steps 1 and 2, and U, which applies steps 2 and 3. Thus, all algorithms compute MDFAs for all subexpressions, but P also concentrates on propagation while U concentrates on unifying MDFAs. Experimental results are given in Table 8. As far as size reduction is concerned, we see that all three algorithms are much less effective than PU. Moreover, they also are much less efficient, spending a lot of time in computing derivatives. However, algorithm P is three times more efficient than the two others, for large expressions, confirming the role of propagation to speed up the computation of DFAs. Conversely, algorithm U is much better than the two others to minimize expressions, confirming that the test data contain many minimal subexpressions that are equivalent to other large input expressions.

Table 8: Simplification of expressions (2)

size	ssize (arith) M	P	U	ssize (geo) M	P	U	t M	P	U	t^{der} M	P	U	$\sharp min$ M	P	U
8	4.6	4.6	4.5	0.0	0.0	0.0	$4.8\,\mu$	$4.4\,\mu$	$6.2\,\mu$	$750\,\nu$	$556\,\nu$	$704\,\nu$	97%	97%	99%
16	8.2	8.2	7.7	7.4	7.3	6.9	$14\,\mu$	$25\,\mu$	$31\,\mu$	$3.5\,\mu$	$12\,\mu$	$3.4\,\mu$	71%	73%	85%
32	14.6	14.1	13.2	12.1	11.8	10.8	$52\,\mu$	$41\,\mu$	$76\,\mu$	$22\,\mu$	$9.1\,\mu$	$12\,\mu$	33%	35%	49%
64	23.3	22.0	20.6	16.4	15.6	13.9	$89\,\mu$	$103\,\mu$	$206\,\mu$	$33\,\mu$	$35\,\mu$	$28\,\mu$	32%	33%	47%
128	33.0	30.2	27.6	18.9	17.7	14.9	$206\,\mu$	$212\,\mu$	$563\,\mu$	$99\,\mu$	$102\,\mu$	$84\,\mu$	36%	37%	52%
256	43.6	38.1	34.9	21.0	19.2	15.8	$500\,\mu$	$462\,\mu$	$1.3\,m$	$275\,\mu$	$221\,\mu$	$244\,\mu$	37%	38%	54%
512	52.8	45.2	41.6	22.3	20.2	16.3	$1.2\,m$	$952\,\mu$	$2.8\,m$	$832\,\mu$	$591\,\mu$	$882\,\mu$	38%	39%	55%
1K	59.3	49.3	45.4	23.0	20.7	16.5	$3.7\,m$	$2.3\,m$	$6.2\,m$	$2.8\,m$	$1.7\,m$	$2.5\,m$	38%	39%	56%
2K	62.4	51.3	47.4	22.9	20.5	16.2	$8.6\,m$	$4.4\,m$	$12\,m$	$6.5\,m$	$3.2\,m$	$6.5\,m$	38%	39%	57%
4K	64.8	52.4	48.9	22.9	20.4	16.1	$11\,m$	$6.0\,m$	$15\,m$	$8.6\,m$	$4.6\,m$	$8.9\,m$	39%	39%	56%
8K	64.9	52.7	50.0	22.8	20.3	16.0	$29\,m$	$8.7\,m$	$35\,m$	$22\,m$	$6.8\,m$	$24\,m$	38%	39%	57%

Table 9: Number of regular languages with a minimal expression of $size \leq 15$ ($nl = 2$)

size	≤ 4	5	6	7	8	9	10	11	12	13	14	15
$\sharp min$	36	41	132	353	836	2.9K	6.9K	22K	63K	185K	572K	1.7M

4.2.3 Two improvements: computing minimal expressions and iterating

We now go a step further, following two possible ways of improving the simplifying power of algorithm PU. First, we consider the problem of minimizing expressions, i.e. finding a shortest equivalent expression for a given expression. Let L be a regular *language*. We define the size of L, denoted by $size(L)$, as the smallest integer s such that there exists an expression E such that $size(E) = s$ and $\mathcal{L}(E) = L$. Then, we may say that an expression E is *minimal* if $size(E) = size(\mathcal{L}(E))$. It is possible to compute a single minimal expression for all regular languages using a given number of letters nl, up to a certain size. For $nl = 2$, this is reasonably feasible, on the computer used for these experiments, up to $size = 15$. Table 9 provides the number of regular languages with a minimal expression of a given size s, for $s \leq 15$ and $nl = 2$. (Moreover, Table 17, in Appendix C.3, displays a selection of such minimal expressions.)

To compute minimal expressions, we use two sets of lists of identifiers (i.e. two objects `MultiList`). The first set is made of lists $list_1(s)$, where $0 \leq s \leq 15$. They are to be filled with identifiers of minimal expressions of size s. The second set contains a larger set of lists $list_2(h)$ containing minimal expressions having an MDFA with a hash code equal to h. The lists $list_1(s)$ are filled bottom up for $s = 0, 1, \ldots, 15$, by computing expressions $E_1 \oplus E_2$ and $E_1 \odot E_2$, where $E_i \in list_1(s_i)(i = 1, 2)$ with $s_1 + s_2 + 1 = s$, and expressions $E\star$, where $E \in list_1(s-1)$. To save computation time, some

Table 10: Simplification of expressions (3)

size	ssize (arith)			ssize (geo)			t			t^{der}			$\sharp min$		
	R	RI	PUI	R	RI	PUI	R	RI	PUI	R	RI	PUI	R	RI	PUI
8	4.49	4.49	4.49	0.0	0.0	0.0	$6.8\,\mu$	$461\,\nu$	$702\,\nu$	$806\,\nu$	$0\,\nu$	$0\,\nu$	100%	100%	100%
16	7.44	7.44	7.59	6.7	6.7	6.8	$59\,\mu$	$917\,\nu$	$2.1\,\mu$	$19\,\mu$	$27\,\nu$	$84\,\nu$	100%	100%	91%
32	11.50	11.49	12.22	9.7	9.7	10.1	$249\,\mu$	$18\,\mu$	$20\,\mu$	$56\,\mu$	$236\,\nu$	$533\,\nu$	73%	73%	59%
64	16.70	16.68	17.67	12.0	11.9	12.5	$623\,\mu$	$5.4\,\mu$	$14\,\mu$	$92\,\mu$	$382\,\nu$	$1.9\,\mu$	59%	59%	52%
128	20.21	20.16	21.29	12.3	12.3	12.7	$1.3\,m$	$39\,\mu$	$54\,\mu$	$136\,\mu$	$1.3\,\mu$	$14\,\mu$	62%	62%	57%
256	22.37	22.23	23.41	12.4	12.4	12.8	$2.4\,m$	$117\,\mu$	$256\,\mu$	$236\,\mu$	$4.1\,\mu$	$22\,\mu$	63%	63%	59%
512	23.48	23.34	24.37	12.4	12.3	12.7	$3.5\,m$	$24\,\mu$	$242\,\mu$	$402\,\mu$	$5.4\,\mu$	$135\,\mu$	64%	64%	60%
1K	23.91	23.71	24.72	12.3	12.3	12.6	$5.3\,m$	$273\,\mu$	$344\,\mu$	$758\,\mu$	$188\,\mu$	$95\,\mu$	64%	64%	61%
2K	23.57	23.36	24.34	12.1	12.0	12.3	$6.0\,m$	$411\,\mu$	$669\,\mu$	$1.1\,m$	$186\,\mu$	$211\,\mu$	65%	65%	61%
4K	23.63	23.38	24.38	12.0	11.9	12.2	$6.1\,m$	$285\,\mu$	$302\,\mu$	$900\,\mu$	$49\,\mu$	$108\,\mu$	66%	66%	62%
8K	23.52	23.31	24.43	12.0	11.9	12.2	$5.9\,m$	$65\,\mu$	$482\,\mu$	$935\,\mu$	$22\,\mu$	$293\,\mu$	65%	65%	62%

checks are applied to avoid creating several times the same expression. Then, the MDFA of the expression is computed together with its hash code h. Afterwards, the list $list_2(h)$ is traversed to check if an expression equivalent to the new one already exists. If it is so, the new expression is ignored, otherwise, it is added to both lists $list_1(s)$ and $list_2(h)$. In the end, all minimal expressions can be stored in an external file.

It is now possible to "improve" algorithm PU by reading all minimal expressions from the external file and adding them to the background before doing anything else.[11] We use the letter R to designate the improved algorithm. Experimental results about algorithm R are shown in Table 10. We see that it is slightly more accurate than PS (by approximately 7%), but less efficient by 50%. It also calculates minimal expressions in 5% more cases. Note that, in algorithm R, minimality of expressions is deduced from their equivalence with an expression in the external file. For the other algorithms, we compared the size of the simplified expressions with the size of the corresponding expressions, as computed by algorithm R.

A second improvement we can think of is to apply algorithms more than once to the input expressions, in order to take advantage of a richer state of the background. In fact, an expression considered at the beginning may be found equivalent to a smaller expression later, but this possible simplification may be missed if we output the simplified expression immediately. The same phenomenon can reduce propagation accuracy, and, in this case, is even more difficult to detect. To get better results, a simple solution consists of re-executing the algorithms with the state of the background obtained at the end of the first run. We designate such an algorithm by adding the letter I after the name of the non iterating version. Experimental results are given for algorithms RI and PUI, in Table 10. The times for these algorithms correspond to the reexecution phase only. They are much smaller than for the first execution. We observe that the simplified expressions are a bit smaller. Also, PUI detects 2% more minimal expressions. Note that reexecution may be

---

[11]To make the method efficient, we do not copy the minimal expressions into the file, but a binary image of their representation in the background. For the sake of brevity, we avoid giving further details.

Table 11: Distribution of expressions with respect to simplification (algorithm PU)

$size$	$ssize$	0:2	3:4	5:8	9:16	17:32	33:64	65:128	129:256	257:512
8	$\sharp expr$	2,391	3,414	4,195	0	0	0	0	0	0
	$\sharp diff$	7	28	565	0	0	0	0	0	0
16	$\sharp expr$	452	1,964	3,781	3,803	0	0	0	0	0
	$\sharp diff$	5	21	664	3,287	0	0	0	0	0
32	$\sharp expr$	22	1,913	2,079	2,771	3,215	0	0	0	0
	$\sharp diff$	2	10	240	2,412	3,211	0	0	0	0
64	$\sharp expr$	0	2,318	1,759	1,454	2,410	2,059	0	0	0
	$\sharp diff$	0	2	91	1,038	2,403	2,059	0	0	0
128	$\sharp expr$	0	2,642	1,787	1,610	1,440	1,768	753	0	0
	$\sharp diff$	0	1	74	1,107	1,424	1,768	753	0	0
256	$\sharp expr$	0	2,696	1,853	1,636	1,393	1,343	960	119	0
	$\sharp diff$	0	1	71	1,090	1,387	1,342	960	119	0
512	$\sharp expr$	0	2,822	1,818	1,651	1,282	1,323	879	222	3
	$\sharp diff$	0	1	72	1,094	1,276	1,323	879	222	3
1K	$\sharp expr$	0	2,798	1,883	1,632	1,331	1,242	840	258	16
	$\sharp diff$	0	1	72	1,054	1,328	1,242	840	258	16
2K	$\sharp expr$	0	2,866	1,917	1,607	1,353	1,213	772	246	26
	$\sharp diff$	0	1	73	1,051	1,346	1,212	772	246	26
4K	$\sharp expr$	0	2,931	1,884	1,630	1,328	1,150	760	292	25
	$\sharp diff$	0	1	71	1,073	1,312	1,150	760	292	25
8K	$\sharp expr$	0	2,919	1,917	1,594	1,280	1,234	781	248	27
	$\sharp diff$	0	1	69	1,044	1,279	1,234	781	248	27

done in several ways. Here, we store the output of the initial execution in a file and we re-apply the algorithms to this file. Moreover, a few additional simplifications can be obtained by iterating the algorithms more than once, but this possibility is not evaluated here. Finally, it is possible to perform the reexecution without computing new derivatives, using only propagation. This method is faster but slightly less accurate.

4.3 On the distribution of expressions with respect to their simplified size

In these final experiments, we focus on two issues:

1. We further elaborate on the "simplifying power" of the algorithms presented in the previous subsection. We concentrate on the two most significant algorithms, PU and R. (Algorithms PUI and RI are slightly more accurate but more difficult to analyze.)

2. We use statistics on our experimental results to help understand how regular expressions are distributed according to their minimum size. For expressions with a small minimum size, indisputable conclusions can be drawn. These expressions make up the majority.

In Tables 11 and 12, statistics are given about the number of input expressions that are simplified to a given size by algorithms PU and R. The number contained in the cell of a column $i:j$ belonging

Table 12: Distribution of expressions with respect to simplification (algorithm R)

$size$	$ssize$	0:2	3:4	5:8	9:16	17:32	33:64	65:128	129:256	257:512
8	$\sharp expr$	2,391	3,414	4,195	0	0	0	0	0	0
	$\sharp diff$	7	28	565	0	0	0	0	0	0
16	$\sharp expr$	452	1,964	3,839	3,745	0	0	0	0	0
	$\sharp diff$	5	21	702	3,249	0	0	0	0	0
32	$\sharp expr$	22	1,913	2,088	3,600	2,377	0	0	0	0
	$\sharp diff$	2	10	246	3,240	2,377	0	0	0	0
64	$\sharp expr$	0	2,318	1,760	1,895	2,392	1,635	0	0	0
	$\sharp diff$	0	2	92	1,472	2,392	1,635	0	0	0
128	$\sharp expr$	0	2,642	1,815	1,880	1,368	1,728	567	0	0
	$\sharp diff$	0	1	75	1,392	1,364	1,728	567	0	0
256	$\sharp expr$	0	2,696	1,853	1,900	1,331	1,281	853	86	0
	$\sharp diff$	0	1	71	1,350	1,328	1,281	853	86	0
512	$\sharp expr$	0	2,822	1,818	1,875	1,215	1,288	796	185	1
	$\sharp diff$	0	1	72	1,314	1,213	1,288	796	185	1
1K	$\sharp expr$	0	2,798	1,883	1,857	1,264	1,192	779	218	9
	$\sharp diff$	0	1	72	1,276	1,264	1,192	779	218	9
2K	$\sharp expr$	0	2,866	1,917	1,835	1,258	1,176	713	213	22
	$\sharp diff$	0	1	73	1,275	1,254	1,176	713	213	22
4K	$\sharp expr$	0	2,931	1,884	1,856	1,219	1,152	698	240	20
	$\sharp diff$	0	1	71	1,285	1,217	1,152	698	240	20
8K	$\sharp expr$	0	2,919	1,920	1,792	1,238	1,184	710	222	15
	$\sharp diff$	0	1	70	1,244	1,237	1,184	710	222	15

to a row $\sharp expr$ where $size = s$, is the number of input expressions of size s that are simplified to an expression with a size between i and j, inclusive. The corresponding number in row $\sharp diff$, is the number of corresponding simplified expressions that are different. A logarithmic scale is used for the simplified sizes, which makes the statistics fairly easy to read.

Several interesting observations can be made.

- In both Tables 11 and 12, the figures for input expressions of size ≥ 128 are very close, which means that the distribution of random expressions with respect to their simplified size is almost independent of their size, if this size is large enough.

- The figures in Tables 11 and 12 are very similar. It means that initializing the background with minimal expressions of size ≤ 15 does not bring a big improvement.

- For both algorithms PU and R, at least 65% of the input expressions are simplified into an expression such that size ≤ 16. Moreover, besides the fact that almost 50% of the input expressions are simplified to a size ≤ 8, the expressions to which they are simplified are very few, only about 70. Even more astonishingly, for an input size ≥ 128, all (more than $2,600$) expressions simplified to a size ≤ 4 are simplified to the same expression (in fact: $(a + b)^*$).

Table 13: Distribution of possibly minimal expressions (algorithm PU)

$size$	$ssize$	0:2	3:4	5:8	9:16	17:32	33:64	65:128	129:256	257:512
8	$\sharp pmin$	2,391	3,414	4,190	0	0	0	0	0	0
16	$\sharp pmin$	452	1,964	3,767	3,507	0	0	0	0	0
32	$\sharp pmin$	22	1,913	2,060	1,665	2,385	0	0	0	0
64	$\sharp pmin$	0	2,318	1,756	855	791	1,031	0	0	0
128	$\sharp pmin$	0	2,642	1,786	1,053	407	328	147	0	0
256	$\sharp pmin$	0	2,696	1,853	1,153	354	204	71	9	0
512	$\sharp pmin$	0	2,822	1,816	1,130	331	209	54	5	0
1K	$\sharp pmin$	0	2,798	1,883	1,123	345	188	47	5	0
2K	$\sharp pmin$	0	2,866	1,917	1,131	335	164	24	2	0
4K	$\sharp pmin$	0	2,931	1,884	1,164	321	160	31	6	0
8K	$\sharp pmin$	0	2,919	1,917	1,141	339	164	41	4	0

To the contrary, almost all input expressions simplified to size > 16 are simplified to different expressions.

Although the statistics presented above only are results about the simplifying power of algorithms PU and R, which are not able to minimize a regular expression in all cases, it can be argued that they give a rather good upper approximation of the number of expressions minimized to a given size. As a matter of fact, algorithm R provides optimal results for expressions with size ≤ 16, and its statistics (the figures in Table 12) are very close to the statistics for algorithm PU. For expressions with a greater simplified size, nothing precise can be said but the results "seem so good" that we can be satisfied with them as an upper approximation. More information about the actual minimal expressions into which input expressions of size $8K$ are simplified is given in Table 17 of Appendix C.3.

Remember that algorithm PU basically simplifies expressions in two ways: propagation and unification of MDFAs. The algorithm R proceeds in the same way but, in addition, detects that certain expressions are equivalent to a minimal expression prerecorded in the background. Let us now concentrate on the relative importance of propagation and unification. Propagation often simplifies subexpressions but this simplification is primarily useful to speed up the computation of a DFA for the subexpression, which is then minimized and unified with the global MDFA of the background. Unification is extremely powerful because it ensures that the subexpression is simplified to the smallest equivalent expression in the background, and also because of two facts:

1. Since the input expressions are perfectly randomly chosen, many small expressions are registered in the background, as subexpressions of an input expression.

2. Generally speaking, most expressions are equivalent to a small expression.

In Tables 13 and 14, statistics are given about the number (noted $\sharp pmin$) of input expressions that are simplified by unification to a subexpression of a lifted input expression. We call such subexpressions "possibly minimal expressions" since no shorter subexpression exists in the set of input expressions.

It can be seen in Table 14 that almost all of them actually are minimal when their size is less than or equal to 8. More than half of them are minimal for a size between 9 and 15. The rest of

Table 14: Distribution of possibly minimal expressions (algorithm R)

$size$	$ssize$	0:2	3:4	5:8	9:16	17:32	33:64	65:128	129:256	257:512
8	$\sharp pmin$	2,391	3,414	4,186	0	0	0	0	0	0
	$\sharp min$	2,391	3,414	4,195	0	0	0	0	0	0
16	$\sharp pmin$	452	1,964	3,778	2,397	0	0	0	0	0
	$\sharp min$	452	1,964	3,839	3,710	0	0	0	0	0
32	$\sharp pmin$	22	1,913	2,068	1,052	1,144	0	0	0	0
	$\sharp min$	22	1,913	2,088	3,326	0	0	0	0	0
64	$\sharp pmin$	0	2,318	1,756	719	436	511	0	0	0
	$\sharp min$	0	2,318	1,760	1,772	0	0	0	0	0
128	$\sharp pmin$	0	2,642	1,814	894	235	161	48	0	0
	$\sharp min$	0	2,642	1,815	1,781	0	0	0	0	0
256	$\sharp pmin$	0	2,696	1,853	1,040	202	91	23	4	0
	$\sharp min$	0	2,696	1,853	1,794	0	0	0	0	0
512	$\sharp pmin$	0	2,822	1,816	1,050	214	117	15	0	0
	$\sharp min$	0	2,822	1,818	1,772	0	0	0	0	0
1K	$\sharp pmin$	0	2,798	1,883	1,034	212	86	15	1	0
	$\sharp min$	0	2,798	1,883	1,760	0	0	0	0	0
2K	$\sharp pmin$	0	2,866	1,917	1,055	187	94	9	0	0
	$\sharp min$	0	2,866	1,917	1,713	0	0	0	0	0
4K	$\sharp pmin$	0	2,931	1,884	1,084	202	85	15	0	0
	$\sharp min$	0	2,931	1,884	1,744	0	0	0	0	0
8K	$\sharp pmin$	0	2,919	1,920	1,065	199	95	14	0	0
	$\sharp min$	0	2,919	1,920	1,687	0	0	0	0	0

those possibly minimal expressions are few and they are possibly unified to a minimal expression by algorithm R. Other simplified expressions, which are significantly less numerous than the possibly minimal expressions, are simplified either by propagation or by unification to an expression simplified by propagation. Yet such simplified expressions can be minimal, sometimes. For instance, we can guess that many possibly minimal expressions with size > 16 in Table 14 actually are minimal. They are significantly fewer than in Table 13, suggesting than the additional ones in Table 13 are minimized by algorithm R. As a matter of fact, we can observe that the number of possibly minimal expressions in Table 13 (6, 525 for $size = 8K$) is almost equal to the number of minimal expressions in Table 14 (6, 526 for $size = 8K$).

To conclude, a few words can be said about possible ways to improve algorithms PU and R. First, we can notice that algorithm R loses a lot of efficiency by prerecording all minimal expressions of size less than 16 (see Table 9): Half of the identifiers available in the background are attributed to these expressions, while few of them are actually used to minimize a subexpression of an input expression. We could experimentally take note of the minimal expressions that are used and keep them in a file much smaller than the file currently used. Second, we can enhance both algorithms with explicit simplification rules similar to those of [10, 18, 28]. Propagation already is such a

simplification rule, which is cheap and effective for randomly generated expressions. More elaborate simplification rules should be applied after propagation.

5   Related work

The work presented in this paper, and especially its implementation aspects, has many similarities to our previous work on what I called *collections of structures* [5, 8, 11, 12]. A collection of structures is a data structure representing all terms of a congruence relation defined by a finite set of equations between ground terms [21, 26]. The terms are partitioned into equivalence classes in such a way that a term t can be minimized in time $O(size(t))$ and that the equivalence of two terms t_1 and t_2 can be decided in time $O(size(t_1) + size(t_2))$. The method can be extended to theories defined by equations using variables, provided that the number of their equivalence classes is finite (and not too big), such as boolean formulas using a small number of letters (atomic propositional formulas). A complete study of collections of structures, including a theoretical complexity analysis, an experimental evaluation, and a comparison with related work can be found in [5]. I have attempted to extend the applicability of collections of structures to regular expressions, based on [22, 27], but the experimental results were disappointing. In this paper, collections of structures are abandoned in favor of the use of normalized expressions and the simpler Union-Find method to represent equivalence classes, which facilitates the integration of the two worlds of expressions and finite automata.

As said before, the specific purpose of this paper is not to describe and discuss methods to simplify regular expressions. It is only to exactly explain how the framework works and to give evidence that it provides an integrated representation of regular languages that can be useful to solve various problems. Nevertheless, other works mainly dedicated to the simplification of regular expressions present aspects comparable to specific points developed in this paper. We now discuss these issues.

The work in [18] is explicitly devoted to the problem of simplifying regular expressions. We compare it to our work from several viewpoints:

1. The implementation framework. In fact, this is the main issue to be addressed, as the implementation aspects are at the heart of this document.

2. The applicability of our framework to the implementation of the simplification methods presented in [18].

3. Some subtle differencies in the nature of the regular expressions that are considered and in the way they are randomly generated. This allows us to explain some differences in the experimental results.

As for the first point, let us note that the work in [18] is implemented within the functional language Haskell, which makes it difficult and sometimes almost impossible to implement the low-level data structures extensively used in our work (see e.g. Section 2.2.2). Instead, classical tree structures used to represent expressions are enhanced with so-called tags providing information on the denoted language and the simplification context. The fact that integers are used as identifiers in our system also speeds up some operations common to both systems, such as sorting terms in unions. In [18], a linear order on expressions is used to compare expressions, while we compare integers. The

fact that we use identifiers for expressions implies some overhead when new expressions are built but this overhead is low as explained in Section 2.2.5 and has the benefit that the same expression is never built two times, which would often arise as shown in Tables 1 and 2. This can speed up simplification since the same expression need not be simplified two times. Derivatives of regular expressions are also used in [18] to check language equivalence and to compute a linear order on regular languages. This requires us to check the equivalence of expressions with respect to the relation $\cong$ of Definition 3. In [18], this is done by checking the syntactic equality of *standardized* expressions, which correspond to normalized expressions in our terminology. In our work, however, this check amounts to checking the equality of two integers, which clearly is much more efficient. It can also be observed that, in the framework described here, much information about work previously done is recorded in the background and can be reused to simplify and speed up further work. This does not seem to be the case in [18] where expressions are simplified one by one. However, there is a proviso for "catalogues" mapping some expressions, or some languages represented by compacted automata inspired from [2], to equivalent minimal expressions. The same effect can be achieved by the background as shown in Section 4.2 with algorithm R. Moreover, comparable results can be obtained at a lower cost by simplifying expressions to subexpressions contained in the background, as shown by the analysis of algorithm PU. Another difference is our use of a hashcode to search for equivalent expressions in a `MultiList` object. This is probably more efficient than using a linear order on regular languages as in [18] if the hashing function is well chosen, i.e. sufficiently uniform. An advantage is that we do not have to keep a representation of the MDFAs of all minimal languages. An integer is enough.

Concerning the second point, it is quite clear that most of the simplification techniques presented in [18] can be accommodated in our framework, including the tags giving information on the denoted languages and the simplification context: Such tags can be associated with identifiers using a few global arrays. The only problematic issue is their idea of renaming expressions to avoid representing several (or many) expressions equivalent modulo renaming inside the catalogues. It is not clear at this moment how this idea could be integrated in what is proposed here but it would probably entail a lot of additional algorithmic work and data structures, and a big loss in efficiency.

Regarding the third point, notice that the work in [18] uses a different syntax of (plain) regular expressions, where expressions of the form $1 + E$ or $E + 1$ are replaced by so-called queries of the form $E?$. This implies a very different set of randomly generated expressions. Moreover, they "pre-standardize" unions and concatenation when generating random expressions (S. Kahrs, personal communication, October 10, 2024). Such differences with the work presented here probably explain why some of their experimental results are very different from the measures in this paper. For instance, they report that 42.8% of input expressions of size $2,050$ using two letters are simplified to $(a + b)^*$ by lifting, while we only get 26% here. In fact, those figures measure intrinsic properties of the chosen expressions, not the value of the simplification techniques (which are almost optimal for solving this problem).

An elaborate method to simplify regular expressions is described in [28], pages $92 - 120$. It is implemented in Standard ML and made available in [29]. The approach is quite different from here and from [18] because it is *mainly pedagogic, with no claims of either efficiency or completeness* (A. Stoughton, personal communication, October 14, 2022). It uses a notion of regular expression complexity where the star-height, i.e. the maximum number of nested iterations in an expression, is the main parameter to minimize. For instance, the expression $1 + a(a + b)^*$ is considered simpler than $(ab^*)^*$, although its size is greater. A notion of weakly simplified expression, similar to standardized expressions in [18], is used as well as a set of 26 simplification rules reducing the complexity of

expressions. The rules use a partial decision algorithm to check the inclusion of regular languages. This algorithm proceeds by induction on the structure of expressions. Thus, the conceptual link with DFAs is not taken into account in that work. The simplification algorithm computes a sequence of weakly simplified expressions of strictly decreasing complexity. At each step, structural rules are applied to the current expression in an attempt to find one or several matching simplification rules. The application of structural rules potentially creates an exponential number of no longer weakly simplified variants of the current expression. Thus, the expression is weakly simplified again at the end of each step.

It would be possible to use the complexity measure and to adapt the simplification rules of [28] to normalized expressions, inside an enhanced version of algorithm PU. This would be more efficient. Moreover, it is the case that several of the simplification rules proposed in [28] simplify an expression to one of its subexpression. Such simplifications are automatically and implicitly done by the algorithm PU.

Finally, it is interesting to relate the results of Section 4.3 to the theoretical work in [20]. Using purely mathematical methods, they compute that the simplified lifting method (see Appendix D) simplifies expressions to an average size of 78, 496, 2,518, and 11,684, respectively for $nl = 2, 3, 4, 5$, when the input size of the expression tends to infinity. These figures can be compared to the average simplified sizes obtained by algorithms L and PU for a size equal to $8K$. They respectively are equal to 69, 439, 1,508 and 3,243 for L and to 25, 334, 1,400, and 3,138, for PU. The values for L and $nl = 2, 3$ are close to the figures in [20], indicating that $8K$ is sufficiently large to get good statistical results on expressions with 2 or 3 letters. Moreover, this shows that algorithm PU simplifies such expressions much better than lifting, arguably giving interesting statistical information about minimization, in these cases. The fact that the values for $nl = 4, 5$ are well below 2,518, and 11,684 suggests that it would be necessary to consider input expressions much longer than $8K$ to get interesting statistical information in these cases. This analysis is also confirmed by the fact that, in [20], the average proportion of expressions minimized to the universal one is computed as at least equal to 31%, 13%, 6.2%, and 2.8%, respectively for $nl = 2, 3, 4, 5$, and that we get only slightly lower values for $nl = 2, 3$ but much lower ones for $nl = 4, 5$ (see Table 15).

6 Conclusion and future work

I have presented an implementation framework for regular languages in which a large collection of regular languages can be represented as both regular expressions and DFAs, in an integrated way. The main contribution consists of a meticulous description of low-level data structures and algorithms designed to efficiently implement higher-level notions such as normalized expressions and equations between expressions and derivatives. In fact, this paper constitutes a prequel to my previous papers on simplifying regular expressions [10] and computing derivatives of regular expressions to build DFAs [9]. For lack of space, I was unable to describe the framework in a convincing way in those papers, which made their contribution difficult to assess. I guess that this is now possible.

Another contribution lies in the experimental evaluation. It shows that the framework allows us to partition a large collection of regular languages in such a way that non equal regular languages belong to different equivalence classes; moreover, every represented language is given an identifier designating a small expression and an MDFA for the language, at the same time. Also, the collection of represented languages can be analyzed to give statistical information on the distribution of expressions with respect to their simplified size. These experiments are new with respect to those of [9, 10] as they assess the framework itself, independently of specific applications.

The work in this paper and in [9, 10] leaves open many research avenues such as relating the sizes of minimal expressions to the sizes of MDFAs, with the aim of optimally tuning the system, inventing new simplification rules for difficult expressions, and building a "final" well-documented version of the system. The experiments conducted in this paper suggest that the system could used to build a kind of simplification server that would improve itself over time by recording more and more "interesting" simplified expressions.

Artefact

The Java code of the algorithms presented in this paper can be found in the dropbox folder `System Java Code`, with some test data and explanations about running the test programs.

Author contribution and funding declaration

Acknowledgements

References

[1] A.V. Aho and J.D. Ullman. *The Theory of Parsing, Translation, and Compiling: Parsing.* Prentice-Hall series in automatic computation. Prentice-Hall, 1972.

[2] Marco Almeida, Nelma Moreira, and Rogério Reis. Testing the equivalence of regular languages. In Jürgen Dassow, Giovanni Pighizzini, and Bianca Truthe, editors, *Proceedings Eleventh International Workshop on Descriptional Complexity of Formal Systems, DCFS 2009*, volume 3 of *EPTCS*, pages 47–57, 2009.

[3] Valentin M. Antimirov. Rewriting regular inequalities (extended abstract). In Horst Reichel, editor, *Fundamentals of Computation Theory, 10th International Symposium, FCT '95*, volume 965 of *Lecture Notes in Computer Science*, pages 116–125. Springer, 1995.

[4] Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2):291–319, 1996.

[5] Mêton Mêton Atindehou. *Une structure de données pour représenter de grands ensembles de termes égaux : application à une méthode générale de simplification d'expressions.* PhD thesis, UCL – SST/ICTM/INGI – Pôle en ingénierie informatique, Belgium, 2018.

[6] Filippo Bonchi and Damien Pous. Checking NFA equivalence with bisimulations up to congruence. In Roberto Giacobazzi and Radhia Cousot, editors, *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13*, pages 457–468. ACM, 2013.

[7] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, 1964.

[8] Baudouin Le Charlier. Experimental evaluation of a method to simplify expressions. *CoRR*, abs/2003.06203, 2020. Available at https://arxiv.org/abs/2003.06203.

[9] Baudouin Le Charlier. Another look at derivatives of regular expressions, with an experimental evaluation. 2023. Available at SSRN: https://ssrn.com/abstract=4592136.

[10] Baudouin Le Charlier. A program that simplifies regular expressions (tool paper). *CoRR*, abs/2307.06436, 2023. Available at https://arxiv.org/abs/2307.06436.

[11] Baudouin Le Charlier and Mêton Mêton Atindehou. A method to simplify expressions: Intuition and preliminary experimental results. In Boris Konev, Stephan Schulz, and Laurent Simon, editors, *IWIL@LPAR 2015, 11th International Workshop on the Implementation of Logics, Suva, Fiji, November 23, 2015*, volume 40 of *EPiC Series in Computing*, pages 37–51. EasyChair, 2015.

[12] Baudouin Le Charlier and Mêton Mêton Atindehou. A data structure to handle large sets of equal terms. In James H. Davenport and Fadoua Ghourabi, editors, *7th International Symposium on Symbolic Computation in Software Science, SCSS 2016, Tokyo, Japan, March 28-31, 2016*, volume 39 of *EPiC Series in Computing*, pages 81–94. EasyChair, 2016.

[13] J.H. Conway. *Regular Algebra and Finite Machines.* Chapman and Hall mathematics series, London. Dover Publications, Incorporated, 1971.

[14] Bernard A. Galler and Michael J. Fischer. An improved equivalence algorithm. *Commun. ACM*, 7(5):301–303, 1964.

[15] Roger Godement. *Cours d'algèbre*. Editions Hermann, 1973.

[16] Paul R Halmos. *Naive Set Theory*. Undergraduate texts in mathematics. Springer-Verlag, New York, 1974.

[17] John E. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[18] Stefan Kahrs and Colin Runciman. Simplifying regular expressions further. *J. Symb. Comput.*, 109:124–143, 2022.

[19] Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms*. Addison-Wesley, 1968.

[20] Florent Koechlin and Pablo Rotondo. Analysis of an efficient reduction algorithm for random regular expressions based on universality detection. In Rahul Santhanam and Daniil Musatov, editors, *16th International Computer Science Symposium in Russia, CSR 2021*, volume 12730 of *Lecture Notes in Computer Science*, pages 206–222. Springer, 2021.

[21] Dexter Kozen. Complexity of finitely presented algebras. In *Proceedings of the 9th Annual ACM Symposium on Theory of Computing, May 4-6, 1977, Boulder, Colorado, USA*, pages 164–177, 1977.

[22] Dexter Kozen. A completeness theorem for kleene algebras and the algebra of regular events. In *Proceedings of the Sixth Annual Symposium on Logic in Computer Science (LICS '91), Amsterdam, The Netherlands, July 15-18, 1991*, pages 214–225, 1991.

[23] Elliott Mendelson. *Schaum's Outline of Calculus*. McGraw-Hill Education, New York, 2022.

[24] Albert R. Meyer and Larry J. Stockmeyer. The equivalence problem for regular expressions with squaring requires exponential space. In *13th Annual Symposium on Switching and Automata Theory, College Park, Maryland, USA, October 25-27, 1972*, pages 125–129. IEEE Computer Society, 1972.

[25] Edward F. Moore. Gedanken-experiments on sequential machines. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, USA, 1956.

[26] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[27] Arto Salomaa. Two complete axiom systems for the algebra of regular events. *J. ACM*, 13(1):158–169, 1966.

[28] Alley Stoughton. *Formal Language Theory: Integrating Experimentation and Proof*. https://alleystoughton.us/forlan/book.pdf, 2003–2025. Open source book.

[29] Alley Stoughton. The Forlan Project. https://alleystoughton.us/forlan/, 2003–2025. Open source software.

A   Complements on normalized expressions

A.1   Proof of Theorem 1

1. The first part of the theorem is a consequence of the fact that the operation $\cup$ is associative, commutative, and idempotent. Let E_1, E_2, and E_3 be normalized expressions and let S_1, S_2, and S_3 be the corresponding sets of expressions according to Definition 2. The value of $E_1 \oplus E_1$ is determined by $S_1 \cup S_1$. Since $S_1 \cup S_1 = S_1$, it is equal to E_1. Similarly, $E_1 \oplus E_2 = E_2 \oplus E_1$ because $S_1 \cup S_2 = S_2 \cup S_1$, and $E_1 \oplus (E_2 \oplus E_3) = (E_1 \oplus E_2) \oplus E_3$ because $S_1 \cup (S_2 \cup S_3) = (S_1 \cup S_2) \cup S_3$.

2. The second part of the theorem is immediate if one of the expressions E_1, E_2, and E_3 is equal to 0 or 1. Otherwise the proof can be done by induction on the syntactic structure of E_1:

   (a) Let us first suppose that E_1 is not a concatenation. Then,

   $$
   \begin{aligned}
   (E_1 \odot E_2) \odot E_3 &= (E_1 \cdot E_2) \odot E_3 \\
   &= E_1 \cdot (E_2 \odot E_3) \\
   &= E_1 \odot (E_2 \odot E_3)
   \end{aligned}
   $$

   All equalities are immediate consequences of the definition of $\odot$.

   (b) Now, let us suppose that $E_1 = F_1 \cdot F_2$ where F_1 is not a concatenation. We have:

   $$
   \begin{aligned}
   (E_1 \odot E_2) \odot E_3 &= ((F_1 \cdot F_2) \odot E_2) \odot E_3 \\
   &= (F_1 \cdot (F_2 \odot E_2)) \odot E_3 \\
   &= (F_1 \odot (F_2 \odot E_2)) \odot E_3 \\
   &= F_1 \odot ((F_2 \odot E_2) \odot E_3) \\
   &= F_1 \odot (F_2 \odot (E_2 \odot E_3)) \\
   &= F_1 \cdot (F_2 \odot (E_2 \odot E_3)) \\
   &= (F_1 \cdot F_2) \odot (E_2 \odot E_3) \\
   &= E_1 \odot (E_2 \odot E_3)
   \end{aligned}
   $$

   All equalities are immediate consequences of the definition except the fourth and the fifth ones, which use the induction on E_1 (first applied to F_1, and then to F_2).

3. The proofs are direct consequences of the definitions.
   Let use prove, for instance, that $\mathcal{L}(E_1 \oplus E_2) = \mathcal{L}(E_1) \cup \mathcal{L}(E_2)$. All terms (see Section 2.2.5) of $E_1 \oplus E_2$ are terms of E_1 or E_2, and conversely. Let $F_1, \ldots, F_n$ be these terms ($n \geq 0$). By definition of $\mathcal{L}$, in Figure 1, $\mathcal{L}(E_1 \oplus E_2) = \mathcal{L}(F_1) \cup \ldots \cup \mathcal{L}(F_n) = \mathcal{L}(E_1) \cup \mathcal{L}(E_2)$.

A.2   Proof of Theorem 2

Since the relation $\cong$ is a congruence, it is sufficient to show that all pairs of plain expressions respecting an equivalence of Definition 3 are normalized to the same normalized expression. Let us denote the normalization of the plain regular expression E by $\overline{E}$. All proofs are similar and direct consequences of the definitions and of Theorem 1. As an example let us prove that

$$
\overline{E_1 + (E_2 + E_3)} = \overline{(E_1 + E_2) + E_3}.
$$

Clearly:

$$
\begin{aligned}
\overline{E_1 + (E_2 + E_3)} \ &= \ \overline{E_1} \oplus \overline{E_2 + E_3} && (\text{Definition of } \overline{E_1 + (E_2 + E_3)}\,) \\
&= \ \overline{E_1} \oplus (\overline{E_2} \oplus \overline{E_3}) && (\text{Definition of } \overline{E_2 + E_3}) \\
&= \ (\overline{E_1} \oplus \overline{E_2}) \oplus \overline{E_3} && (\text{Theorem 1 }) \\
&= \ (\overline{E_1 + E_2}) \oplus \overline{E_3} && (\text{Definition of } \overline{E_1 + E_2}) \\
&= \ \overline{(E_1 + E_2) + E_3} && (\text{Definition of } \overline{(E_1 + E_2) + E_3}\,)
\end{aligned}
$$

B Complements on the background

B.1 Confluence of the operation *reduce*

The operation *reduce* modifies the background by merging some of its equivalence classes of expressions and reducing its set of equations by replacing every expression occurring in equations by its representative in the new equivalence classes. Our goal here is to prove that the new set of equivalence classes is the same for all possible executions of *reduce*, independently of what pair of overlapping equations is selected at each step. Let Eq_0 be the set of equations in the background before an execution of *reduce*, and let $\mathcal{E}$ be the set of expressions occurring in these equations. The execution of *reduce* computes an equivalence relation $\sim_q$ on $\mathcal{E}$ from which the new set of equivalence classes of the background can be deduced, and it is simpler to prove that $\sim_q$ is the same in all possible executions than to reason on the complete set of equivalence classes. So we focus on the unicity of $\sim_q$. The proof is organized in four steps, as follows:

1. We give an explicit characterization of the (non deterministic) sequence

$$
\langle \sim_0,\ Eq_0 \rangle \ \longrightarrow \ \langle \sim_1,\ Eq_1 \rangle \ \longrightarrow \ \dots \ \longrightarrow \ \langle \sim_q,\ Eq_q \rangle
$$

   computed by the operation *reduce*.

2. We define an equivalence relation $\sim$, in a declarative, i.e. non algorithmic, way.

3. We prove that $\quad \sim_0 \ \subseteq \ \sim_1 \ \subseteq \ \dots \ \subseteq \ \sim_q \ \subseteq \ \sim .$

4. We observe that $\quad \sim \ \subseteq \ \sim_q .$

So, we can conclude that all possible executions of *reduce* compute the same equivalence relation $\sim$.

Step 1 Let us start with some notation. We abbreviate the construction $o + \dots + x \cdot E_x + \dots$ by *rpart*. So equations can be written as $E = rpart$, $E' = rpart'$, and the like. We say that E_x and E'_x are corresponding expressions in *rpart* and *rpart'*. If ρ is a relation on $\mathcal{E}$, we write $rpart\ \rho\ rpart'$ to mean that $E_x\ \rho\ E'_x$ for all corresponding expressions of *rpart* and *rpart'*.

   Now, we proceed to characterize every pair $\langle \sim_i,\ Eq_i \rangle$ ($0 \le i \le q$), by recurrence on i. We remember that, for all i, a unique representative is chosen for every expression in its equivalence class. Moreover, the set Eq_i consists of all equations $E = rpart$ only using representatives of expressions in $\sim_i$ such that $E \sim_i E'$ and $rpart \sim_i rpart'$, for some equation $E' = rpart'$ in Eq_0.

1. For $i = 0$, the relation $\sim_0$ simply is the set of all pairs $\langle E, E \rangle$ such that $E \in \mathcal{E}$. Moreover, for every equation $E = rpart$ in Eq_0, this equation itself is such that $E \sim_0 E$ and $rpart \sim_0 rpart$, and it uses representatives only.

2. For any i such that $i < q$, let F_1 and F_2 be the expressions on which the chosen overlapping equations $E_1 = rpart_1$ and $E_2 = rpart_2$ disagree. Let us assume that the operation *unify* chooses to replace F_2 by F_1 in the equations of Eq_i. Then:

   (a) The relation $\sim_{i+1}$ is the smallest equivalence relation such that $\sim_i \subseteq \sim_{i+1}$ and $\langle F_1, F_2 \rangle \in \sim_{i+1}$. The equivalence classes of F_1 and F_2 are merged in $\sim_{i+1}$ and F_1 becomes the representative of all expressions of the new class. For other classes, the representative remains the same.

   (b) Every equation $E = rpart$ in Eq_i is modified into an equation $E'' = rpart''$ in Eq_{i+1}, by replacing all occurrences of F_2 by F_1 in the first equation. Since Eq_0 contains an equation $E' = rpart'$ such that $E' \sim_i E$ and $rpart' \sim_i rpart$, the same equation is such that $E' \sim_{i+1} E''$ and $rpart' \sim_{i+1} rpart''$, by definition of $\sim_{i+1}$.

Step 2 Now, let us define the relation $\sim$. We define it as the smallest equivalence relation on $\mathcal{E}$ such that

$$E \sim E' \iff rpart \sim rpart'$$

for all pairs of equations $E = rpart$ and $E' = rpart'$ in Eq_0.

Step 3 Next, let us prove that $\sim_0 \subseteq \sim_1 \subseteq \ldots \subseteq \sim_q \subseteq \sim$. It is clear that $\sim_0 \subseteq \sim$ since $\sim_0$ is the smallest equivalence relation on $\mathcal{E}$. Now, let us prove that $\sim_i \subseteq \sim$ implies $\sim_{i+1} \subseteq \sim$ for $i < q$. Using the same notation as above, the proof boils down to showing that $F_1 \sim F_2$ since, then, $\sim_i \subseteq \sim$ and $\langle F_1, F_2 \rangle \in \sim_{i+1}$ implies that $\sim_{i+1} \subseteq \sim$. Two cases must be considered: Either $E_1 = E_2$, or $rpart_1 = rpart_2$. In the first case, the set Eq_0 contains two equations $E_1'' = rpart_1''$ and $E_2'' = rpart_2''$ such that $E_1'' \sim_i E_1 = E_2 \sim_i E_2''$. Thus $E_1'' \sim E_2''$, which implies that $rpart_1'' \sim rpart_2''$, by definition of $\sim$. Finally, the latter equivalence implies that $F_1 \sim F_2$, since F_1 and F_2 are equivalent modulo $\sim_i$ to two corresponding expressions in $rpart_1''$ and $rpart_2''$. The case $rpart_1 = rpart_2$ is similar since this equality successively implies $rpart_1'' \sim_i rpart_2''$, $rpart_1'' \sim rpart_2''$, $E_1 \sim E_2$, and $F_1 \sim F_2$.

Step 4 Finally, to prove that $\sim \subseteq \sim_q$, we can observe that the relation $\sim_q$ satisfies the definition of $\sim$, except possibly minimality, i.e. that we have:

$$E_1 \sim_q E_2 \iff rpart_1 \sim_q rpart_2$$

for all pairs of equations $E_1 = rpart_1$ and $E_2 = rpart_2$ in Eq_0. Consider two such equations and let $E_1' = rpart_1'$ and $E_2' = rpart_2'$ be their representatives in Eq_q. Let us assume first that $E_1 \sim_q E_2$. This implies that $E_1' = E_2'$ since equivalent expressions have the same representative. But, since no distinct equations overlap in Eq_q, we must have $rpart_1' = rpart_2'$, which implies $rpart_1 \sim_q rpart_2$ because $rpart_1 \sim_q rpart_1'$ and $rpart_2 \sim_q rpart_2'$. The proof in the other direction is similar.

Note 1 Using the same notation as above, the set of equivalence classes of the background after an execution of the operation *reduce* can be constructed by removing, from the initial set, all classes containing an expression in $\mathcal{E}$ and by adding to this set unions of these classes containing expressions equivalent with respect to $\sim$. In symbols:

$$\mathcal{C} = (\mathcal{C}_0 \setminus \{C \in \mathcal{C}_0 \mid \mathcal{E} \cap C \neq \{\}\}) \cup \left\{ \bigcup_{C' \cap C \neq \{\}} C \;\middle|\; C' \in \mathcal{C}_\sim \;\&\; C \in \mathcal{C}_0 \right\}$$

where $\mathcal{C}_0$, $\mathcal{C}$, and $\mathcal{C}_\sim$ respectively are the initial and final set of equivalence classes of the background, and the set of equivalence classes of the relation $\sim$.

Note 2 When two equations $E = rpart$ and $E' = rpart'$ overlap in the background, it is normally the case that $o = o'$, as a consequence of the invariant of the background stated in Section 3.1.1. However, it is the user's responsibility to ensure that $\mathcal{L}(E) = \mathcal{L}(E')$ when two expressions E and E' are unified. In case of a mistake, there is a high probability that the algorithm *reduce* will find overlapping equations such that $o \neq o'$. This is even unavoidable if the MDFAs of E and E' are unified as in algorithm PU. Therefore, it is useful to add a check that $\texttt{tabTIE}[iEq_1][0] = \texttt{tabTIE}[iEq_2][0]$ at Step $1(a)$ of the algorithm *reduce* (see Section 3.2.3). If the equality does not hold, it means that the user's program contains an error. This may be the case for erroneous simplification rules, for example.

B.2 Garbage collection

When the object $\texttt{iExprList}(M)$ no longer contains any free identifier (see Section 2.2.3), it is often possible to return certain identifiers to it, in order to continue the process underway. To do so, the background is equipped with an algorithm working as follows: It receives as inputs some data structures containing all identifiers needed to proceed. All identifiers reachable from these identifiers are marked. Afterwards, all non marked identifiers are returned to the list $\texttt{iExprList}(M)$. Finally, the current process is resumed.

However, as we use a high-level programming language, it is neither really possible to automatically return to the exact position where the program was interrupted, nor to automatically determine the list of identifiers used by the current process. Thus, the following method is used: When no more free identifiers are available, an exception is thrown. The current process must catch the exception, call the garbage collector passing the information about the identifiers to be saved, and finally resume its work appropriately. In some cases, such as the computation of all derivatives of an expression, a big part of the interrupted process must often be resumed. Notice that it is the user's responsibility to decide which data in the current background are best to continue what is underway. For instance, in the experiments of Section 4, the algorithms attempt to keep all the identifiers used in the global MDFA: Representatives of equivalence classes are kept and other identifiers are freed unless they are needed by the current process. In some cases, it may make more sense to retain only some of the equations, or none at all. Sadly and obviously, it can always be the case that a given algorithm cannot be completed even by reinitializing all data structures from scratch.

B.3 An example of Java code

Figure 5 depicts the Java code of the algorithm *substitute* described at page 20. This Java method is part of the class $\texttt{Background}$, which contains the declarations of the arrays $\texttt{tabIE}[M]$, $\texttt{tabIR}[M]$, $\texttt{tabTIE}[M][]$, and the objects $\texttt{list\_IEQ\_IE}(M, M)$ and $\texttt{list\_IEQ\_IE\_x}[nl](M, M)$. It can be observed that the translation from ordinary language to Java is straightforward.

C Additional experimental results

I provide new experimental results, complementary to those of Sections 4.2 and 4.3. We analyze the performance of algorithms N, L, and PU applied to expressions using up to 3, 4, 5, and 8 letters. As previously the algorithms are applied to large files of $10,000$ randomly generated expressions of sizes ranging from 8 to $8K$. Since expressions become less and less simplifiable when their number nl of letters grows, an upper bound of 512 has been imposed on the size of subexpressions for which an

Figure 5: Java code of the method *substitute*

```java
void substitute(int iE1, int iE2)
{
  while (! list_IEQ_IE.isEmpty(iE2))
  {
    int iEq = list_IEQ_IE.choose(iE2) ;
    int iR = tabIR[iEq] ;
    int [] tabIE = tabTIE[iR] ;

    removeEq(iEq) ;

    int [] tabN = substitute(tabIE, iE1, iE2) ;

    addEq(iE1, tabN) ;
  }

  int x = 1 ;
  while (x != list_IEQ_IE_x.length)
  {
    while (! list_IEQ_IE_x[x].isEmpty(iE2))
    {
      int iEq = list_IEQ_IE_x[x].choose(iE2) ;
      int iE = tabIE[iEq] ;
      int iR = tabIR[iEq] ;
      int [] tabIE = tabTIE[iR] ;

      removeEq(iEq) ;

      int [] tabN = substitute(tabIE, iE1, iE2) ;

      addEq(iE, tabN) ;
    }
    x ++ ;
  }
}
```

MDFA is computed after simplification by propagation. Otherwise, the time spent to compute the derivatives of a subexpression can soar, and the number of derivatives can even exceed the memory size. However, propagation is still applied to all subexpressions.

C.1 Simplification of expressions

Table 15 shows the arithmetic and geometric means of the sizes of expressions simplified by the algorithms N, L, and PU, for various input sizes, and for allowed numbers of letters $nl = 2, 3, 4, 5, 8$.

It also depicts the numbers of expressions found equivalent to the universal one: $(a + b + \ldots + \ell)^*$, where ℓ is the nl-th letter in the alphabet. Lastly, it provides information about the mean execution times. Sizes of the simplified expressions are given in percent of the size of the input expressions if their ratio is greater than or equal to $\frac{2}{3}$; their actual values are given otherwise (rounded).

We can observe that the sizes of the simplified expressions quickly grow when the value of nl increases, which most probably indicates that the sizes of minimal expressions grow almost as quickly. As for the number of expressions found equivalent to the universal one, it decreases very quickly, and we can guess that the values found by algorithm PU are close to the actual average proportion of expressions that are equivalent to the universal expression. Concerning execution times, we see that they grow linearly for algorithms N and L with respect to both $size$ and nl. For algorithm PU, it is insightful to first look at the case $nl = 3$: The execution times grow exponentially until $size = 1K$ but less than linearly afterwards. For $size = 8K$, they are greater by two orders of magnitude than the times for $nl = 2$. This can be explained as follows: The algorithm PU applies to expressions first simplified by algorithm L. The average size of these lifted expressions stabilizes below 70 for $nl = 2$; so, the execution times for PU also stabilize at $4.1\,m$. To the contrary, for $nl = 3$, the average size of lifted expressions does not stabilize, although it would for larger sizes of input expressions. Therefore, the times for PU continue to grow but not so fast in the end due to the 512 bound limiting the use of computing MDFAs. Nevertheless, since the average size of lifted expressions remains below 512, the MDFAs of many subexpressions are computed. This can explain the fact that the sizes of simplified expressions still are much better for PU than for L. Now, the results for $nl = 4, 5, 8$ can be explained by the fact that many lifted expressions remain longer that 512, on the average, which speeds up the execution times of PU but limits its simplification power, as can be observed in Table 15. Increasing the 512 bound could improve the simplification power of PU but at too high a cost in many cases.

C.2   Distribution of expressions with respect to their simplified size

Although algorithm PU gives rather precise information about the distribution of expressions with respect to their simplified size for $nl = 2$, this is no longer the case for higher values of nl. Notice that we are mainly interested by the distribution when the size of expressions tends to infinity because this distribution has an objective value and applies to input expressions of any size, except small ones. The main reason why precise information can be obtained for $nl = 2$, but not beyond, is that algorithm PU applies to lifted expressions whose average size stabilizes at 70 by $size = 1K$. Therefore, any large set of randomly chosen expressions of size at least $1K$ can be used to approximate the distribution. It is not so for higher values of nl since the average size of lifted expressions does not stabilize for sizes less than or equal to $8K$ (see Table 15). Moreover, we cannot go beyond, since we have already imposed a bound of 512 on the size of subexpressions for which an MDFA is computed, which is the second cause of inaccuracy of the modified algorithm PU, analyzed here.

Anyway, the results provided by algorithm PU, for $nl = 3$, are presented in Table 16. We see that, compared to the case $nl = 2$ (see Tables 11 and 13), much fewer expressions are simplified into a "possibly minimal" expression, i.e. into a subexpression of the input set. This suggests that, to be more precise, we should consider (much) larger input sets.

Table 15: Simplification of expressions ($nl = 2, 3, 4, 5, 8$)

	nl	2			3			4			5			8		
	size	N	L	PU	N	L	PU	N	L	PU	N	L	PU	N	L	PU
	8	75%	5	4	75%	75%	5	88%	75%	75%	88%	75%	75%	88%	88%	88%
a	16	69%	9	8	75%	69%	10	81%	75%	75%	88%	81%	75%	88%	88%	88%
r	32	72%	15	12	78%	21	19	81%	75%	72%	84%	78%	78%	91%	88%	84%
i	64	72%	25	18	78%	38	35	81%	72%	69%	84%	78%	77%	89%	86%	86%
t	128	71%	35	22	78%	66	57	82%	68%	82	85%	77%	74%	90%	87%	86%
h	256	71%	47	24	78%	110	90	82%	158	148	85%	74%	71%	89%	86%	86%
	512	72%	56	25	78%	170	133	82%	279	258	85%	69%	67%	89%	86%	85%
m	1K	72%	64	25	78%	246	186	82%	475	439	85%	655	634	90%	85%	84%
e	2K	72%	67	25	78%	312	238	82%	753	699	85%	1,165	1,127	90%	84%	83%
a	4K	72%	70	25	78%	380	289	82%	1,109	1,029	85%	1,987	1,922	90%	83%	82%
n	8K	72%	69	25	78%	439	334	82%	1,508	1,400	85%	3,243	3,138	90%	80%	80%
	8	0	0	0	0	0	0	0	0	0	0	0	0	88%	88%	88%
g	16	69%	8	7	75%	10	9	81%	75%	69%	81%	75%	75%	88%	88%	88%
e	32	69%	13	10	78%	20	18	81%	72%	69%	84%	78%	75%	91%	88%	84%
o	64	70%	17	13	78%	33	30	81%	69%	42	84%	77%	75%	89%	86%	86%
m	128	71%	20	13	78%	50	42	82%	77	73	85%	73%	71%	89%	86%	85%
	256	71%	23	13	78%	72	56	82%	127	117	85%	68%	166	89%	86%	85%
m	512	71%	24	13	78%	95	70	82%	200	180	85%	303	290	89%	85%	84%
e	1K	72%	25	13	78%	115	83	82%	300	271	85%	519	498	90%	83%	82%
a	2K	72%	25	13	78%	125	91	82%	415	375	85%	833	801	90%	81%	80%
n	4K	72%	25	12	78%	135	99	82%	521	475	85%	1,260	1,211	90%	78%	77%
	8K	72%	25	12	78%	143	105	82%	613	558	85%	1,817	1,747	90%	74%	73%
	8	171	719	721	41	98	98	4	4	4	0	0	0	0	0	0
♯	16	11	1,244	1,264	4	290	292	3	57	57	0	5	5	0	0	0
u	32	0	1,814	1,871	0	560	560	0	197	198	0	55	55	0	0	0
n	64	0	2,190	2,317	0	821	827	0	272	273	0	110	110	0	6	6
i	128	0	2,435	2,642	0	1,009	1,024	0	388	389	0	139	139	0	6	6
v	256	0	2,478	2,696	0	993	1,014	0	463	464	0	186	187	0	15	15
e	512	0	2,561	2,822	0	1,008	1,024	0	476	480	0	197	197	0	19	19
r	1K	0	2,555	2,798	0	1,058	1,090	0	471	473	0	194	194	0	21	21
s	2K	0	2,612	2,866	0	1,126	1,155	0	466	469	0	219	219	0	17	20
a	4K	0	2,653	2,931	0	1,109	1,136	0	492	492	0	169	220	0	7	19
l	8K	0	2,666	2,919	0	1,138	1,156	0	464	464	0	104	203	0	4	19
	8	15 μ	12 μ	8.1 μ	22 μ	25 μ	17 μ	13 μ	14 μ	16 μ	11 μ	13 μ	19 μ	9.3 μ	34 μ	46 μ
m	16	16 μ	20 μ	32 μ	31 μ	34 μ	45 μ	15 μ	28 μ	63 μ	14 μ	44 μ	54 μ	12 μ	15 μ	74 μ
e	32	25 μ	46 μ	103 μ	20 μ	35 μ	121 μ	39 μ	29 μ	139 μ	44 μ	30 μ	116 μ	44 μ	26 μ	169 μ
a	64	62 μ	40 μ	264 μ	50 μ	52 μ	344 μ	37 μ	48 μ	405 μ	67 μ	62 μ	382 μ	33 μ	45 μ	364 μ
n	128	83 μ	56 μ	623 μ	64 μ	85 μ	1.2 m	102 μ	96 μ	1.0 m	96 μ	110 μ	1.1 m	58 μ	103 μ	852 μ
	256	122 μ	92 μ	1.3 m	103 μ	112 μ	6.0 m	119 μ	141 μ	6.5 m	139 μ	115 μ	5.0 m	108 μ	133 μ	4.2 m
t	512	230 μ	138 μ	2.0 m	184 μ	170 μ	23 m	195 μ	209 μ	42 m	234 μ	254 μ	27 m	182 μ	254 μ	14 m
i	1K	299 μ	177 μ	3.4 m	254 μ	232 μ	142 m	335 μ	288 μ	156 m	413 μ	360 μ	55 m	404 μ	462 μ	20 m
m	2K	581 μ	235 μ	4.1 m	715 μ	320 μ	225 m	720 μ	483 μ	154 m	674 μ	617 μ	81 m	657 μ	853 μ	33 m
e	4K	1.2 m	402 μ	4.1 m	1.2 m	510 μ	401 m	1.4 m	746 μ	323 m	1.3 m	1.2 m	131 m	1.4 m	1.7 m	62 m
	8K	2.4 m	734 μ	4.1 m	2.3 m	832 μ	535 m	2.9 m	1.2 m	289 m	2.5 m	1.9 m	206 m	2.6 m	2.7 m	123 m

Table 16: Distribution of expressions with respect to simplification ($nl = 3$)

size	ssize	6	12	24	48	96	192	384	768	1.5K	3K	6K
8	♯simp	7,482	2,518	0	0	0	0	0	0	0	0	0
	♯pmin	7,475	2,501	0	0	0	0	0	0	0	0	0
16	♯simp	1,862	5,525	2,613	0	0	0	0	0	0	0	0
	♯pmin	1,856	4,884	2,512	0	0	0	0	0	0	0	0
32	♯simp	691	1,134	6,035	2,140	0	0	0	0	0	0	0
	♯pmin	691	815	4,233	1,912	0	0	0	0	0	0	0
64	♯simp	828	649	851	6,352	1,320	0	0	0	0	0	0
	♯pmin	828	542	390	3,184	983	0	0	0	0	0	0
128	♯simp	1,024	752	723	1,121	5,869	511	0	0	0	0	0
	♯pmin	1,024	676	341	410	1,647	257	0	0	0	0	0
256	♯simp	1,014	796	739	968	1,608	4,779	96	0	0	0	0
	♯pmin	1,014	731	341	354	351	501	24	0	0	0	0
512	♯simp	1,024	791	721	945	1,295	1,979	3,240	5	0	0	0
	♯pmin	1,024	716	338	351	283	147	40	0	0	0	0
1K	♯simp	1,090	793	671	833	1,130	1,557	2,177	1,749	0	0	0
	♯pmin	1,090	692	334	310	251	115	28	2	0	0	0
2K	♯simp	1,155	735	683	809	1,145	1,372	1,714	1,720	667	0	0
	♯pmin	1,155	649	342	298	221	98	17	2	0	0	0
4K	♯simp	1,136	765	686	795	1,072	1,307	1,525	1,549	1,065	100	0
	♯pmin	1,136	682	346	290	234	112	17	0	0	0	0
8K	♯simp	1,156	749	661	836	1,010	1,242	1,431	1,468	1,138	307	2
	♯pmin	1,156	658	330	329	257	91	18	0	0	0	0

C.3 Examples of most frequent simplified expressions

Table 17 depicts the set of expressions into which the expressions of size $8K$ from our test data are the most frequently simplified (for $nl = 2$). To keep the table reasonably small, only expressions selected at least 3 times for simplification are collected. Since these expressions all have a size ≤ 15, they are minimal since they are computed by algorithm R. Every expression is depicted with the number of input expressions simplified into it. For instance, 104 expressions found in the input file are simplified (in fact, minimized) to (a + b)(a + b)*. Statistically, this means that any expression of size $8K$ has a 1% chance of being equivalent to (a + b)(a + b)*, and therefore of being simplified into it. In addition, we see that many expressions are "similar". If we define similar by the fact that two expressions are identical modulo a permutation of the letters and/or by reversing factors in concatenations, we see that all minimal expressions of size 5 or 6 are similar. Similar expressions have the same probability of being the simplification of an expression of any given input size. This similarity notion can be used to reduce the size of Table 17. For example, we could indicate that 443 expressions, i.e. 4.43% of the input expressions, are simplified into an expression similar to (ab*)*. We can also observe that some expressions in the table, although minimal, are not really illuminating as a description of the regular language they denote. For instance, the expression (a*ba*)*, of size 8, denotes the set of strings that are empty or contain at least one occurrence of the letter b. A clearer description is given by 1 + (a + b)*b(a + b)*, of size 12, or by 1 + a*b(a + b)*, of size 10. This illustrates the value of the star-height metric used by [28].

Table 17: Frequent simplified expressions ($nl = 2$)

size = 4	
2,919	(a + b)*

size = 5	
117	(b*a)*
116	(a*b)*
110	(ab*)*
100	(ba*)*

size = 6	
192	b(a + b)*
178	(a + b)*b
168	a(a + b)*
162	(a + b)*a

size = 7	
31	b + (ab*)*
29	a + (ba*)*
28	b + (b*a)*
23	a + (a*b)*
18	b(a*b)*
10	a(b*a)*
10	b(b*a)*
9	(b*a)*a
7	b(ab*)*
6	b(ba*)*
6	a(ba*)*
6	(ba*)*a
6	(a*b)*b
5	(b*a)*b
4	a(a*b)*
4	(a*b)*a
3	(ab*)*b

size = 8	
104	(a + b)(a + b)*
33	(a*ba*)*
32	(b*ab*)*
27	b + a(a + b)*
24	b + (a + b)*a
24	a* + (ba*)*
24	b* + (ab*)*
22	(a + b)*aa
21	b*(b*a)*

size = 8	
20	a*(a*b)*
18	a(a + b)*b
17	a + (a + b)*b
17	ab(a + b)*
17	(a + b)*ab
15	b(a + b)*a
15	ba(a + b)*
13	b(a + b)*b
13	bb(a + b)*
12	a + b(a + b)*
12	(a + b)*bb
10	1 + a(b*a)*
10	aa(a + b)*
9	(a + b)*ba
8	a(a + b)*a
6	1 + (b*a)*a
6	1 + b(a*b)*
6	(a*b(1 + a))*
5	1 + (a*b)*b
3	1 + a(ab*)*
3	(1 + b)(ab*)*
3	(b*a)*(1 + b)
3	((1 + b)ab*)*
3	((1 + a)ba*)*
3	(a + (a + b)b)*

size = 9	
52	a*b(a + b)*
47	b*a(a + b)*
10	1 + ba(a + b)*
8	1 + ab(a + b)*
7	1 + a(a + b)*b
6	1 + (a + b)*ba
6	1 + bb(a + b)*
6	1 + b(a + b)*a
5	1 + (a + b)*ab
5	1 + a(a + b)*a
5	1 + b(a + b)*b
5	(1 + b)a(a + b)*
5	(a + b)*a(1 + b)
4	1 + (a + b)*aa

size = 9	
4	b + a(b*a)*
4	(1 + a)b(a + b)*
4	(a + b)*b(1 + a)
3	1 + aa(a + b)*
3	(b + (ab*)*)b
3	((a + bb)(1 + b))*
3	(ba + ab*)*

size = 10	
16	a + (a*ba*)*
12	(a + b)(a + b)*a
12	(a + b)(a + b)*b
11	b + (b*ab*)*
11	b(a + b)(a + b)*
11	a(a + b)(a + b)*
6	b(a* + (ba*)*)
4	a(b*ab*)*
4	(ab*)*b*a
4	(ab* + b*a)*
3	1 + a(b + (b*a)*)
3	1 + b + a(ab*)*
3	b(b* + (ab*)*)
3	ba*(a*b)*
3	ba(a + b)*a
3	bba(a + b)*
3	bb(a + b)*b
3	a(a*ba*)*
3	a(a + b)*bb
3	a(a + b)*ba
3	a* + (a*b)*b
3	b* + (b*a)*b
3	(a + b)*(b + aa)
3	ab* + (ba*)*
3	(a + (a*b)*)(1 + a)
3	(a + (a + b)(a + b))*

size = 11	
11	a*b(a + b)*b
8	a + a*b(a + b)*
8	(ba*)*(b*a)*
8	(ab*)*(a*b)*
7	bb*a(a + b)*

size = 11	
6	aa*b(a + b)*
6	a*b(a + b)*a
5	1 + a(a + b)(a + b)*
5	((1 + a)(b + (a + b)a))*
4	ba*b(a + b)*
4	ab*a(a + b)*
3	1 + (a + b)*b(a + b)
3	1 + b(1 + (a*b)*b)
3	b + b*a(a + b)*
3	a(a + b)*a(1 + b)
3	a*b*(b*a)*
3	b*(b + (a + b)*a)
3	(1 + a)b(a + b)*b
3	(ab*)*b*a*
3	((1 + b)(a + (a + b)b))*
3	(a + (a + b)*bb)*
3	(b + (a + b)*aa)*

size = 12	
4	(a + b)(a + b)(a + b)*
4	(a + b)*(a + (a + b)b)
4	(aa + (1 + a)b(1 + a))*
3	1 + b*a(a + b)*b
3	1 + a*b(a + b)*a
3	a + b(a + b)(a + b)*
3	a* + ab + (ba*)*
3	(a + b)*a(1 + b(1 + b))
3	((b + ab*a)(1 + a))*
3	((1 + b)(a + ba*b))*

size = 13	
6	(a + b)a*b(a + b)*
5	(a + b)*ba(a + b)*
4	1 + (a + b)(a + b)(a + b)*
3	(a + b)*bb(a + b)*

size = 14	
3	1 + (a + b)a*b(a + b)*
3	b + ((a + ba*b)(1 + b))*
3	a* + (ba*)*(a*b)*
3	b* + (ab*)*(b*a)*

size = 15	
3	ab + (ba*)*(b*a)*

D More information on lifting

The lifting method was introduced in [18]. A related but simplified method is described and theoretically studied in [20]. The lifting algorithm, used in Section 4.2, has intermediate power and complexity. As said before, it is used as a pre-processing step in most of the algorithms studied in Section 4, i.e. it is applied "outside the system" to simplify an input expression before it is normalized and added to the set of expressions represented in the system. This ensures a more economical use of identifiers. Note that our lifting has a linear complexity, as does the algorithm of [20], while the algorithm of [18] is quadratic because union $(+)$ is an n-ary operator in their work.

The goal of this appendix is to describe the simplification rules that are used by our lifting algorithm. There are basically three rules. Let E be a (plain) regular expression. Let us denote the set of letters occurring in E by $\mathcal{A}(E)$ and let us denote the set of letters x such that $x \in \mathcal{L}(E)$ by $\mathcal{A}_1(E)$. Let x_1, x_2, ..., x_ℓ be letters. The rules are the following:

1. If $\mathcal{A}(E) = \mathcal{A}_1(E)$, then the expression E^* can be simplified to $(x_1 + x_2 + \ldots + x_\ell)^*$ where $\mathcal{A}_1(E) = \{x_1, x_2, \ldots, x_\ell\}$.

2. If $1 \in \mathcal{L}(E)$ and $\mathcal{A}(E) \subseteq \{x_1, x_2, \ldots, x_\ell\}$ then $E \cdot (x_1 + x_2 + \ldots + x_\ell)^*$ and $(x_1 + x_2 + \ldots + x_\ell)^* \cdot E$ both simplify to $(x_1 + x_2 + \ldots + x_\ell)^*$.

3. If $\mathcal{A}(E) \subseteq \{x_1, x_2, \ldots, x_\ell\}$ then $E + (x_1 + x_2 + \ldots + x_\ell)^*$ and $(x_1 + x_2 + \ldots + x_\ell)^* + E$ both simplify to $(x_1 + x_2 + \ldots + x_\ell)^*$.

The rules can be applied bottom-up on subexpressions, since the sets $\mathcal{A}(E)$, $\mathcal{A}_1(E)$ and the fact that $1 \in \mathcal{L}(E)$ can be computed bottom-up, as we go along. Since we use a limited number of letters, the sets can be represented by a single binary word and computed using logical operators on words. They are more general than the rules used in [20] where a single "universal" expression $(x_1 + x_2 + \ldots + x_\ell)^*$ using all letters possibly used in expressions is considered.

The fact that the rules in [20] are effective for randomly generated expressions, especially for expressions using few letters, is a statistical property of regular expressions theoretically proved in [20]. The greater generality of our rules does not bring much improvement in practice for such expressions.