Antimirov and Mosses's rewrite system revisited

Marco Almeida

Nelma Moreira

Rogério Reis

LIACC, Faculdade de Ciências, Universidade do Porto
Departamento de Ciência de Computadores
Rua do Campo Alegre, 1021/1055
4169-007 Porto, Portugal

{*mfa,nam,rvr*}*@ncc.up.pt*

Received (Day Month Year)
Accepted (Day Month Year)
Communicated by (xxxxxxxxxx)

Antimirov and Mosses proposed a rewrite system for deciding the equivalence of two (extended) regular expressions. They argued that this method could lead to a better average-case algorithm than those based on the comparison of the equivalent minimal deterministic finite automata. In this paper we present a functional approach to that method, prove its correctness, and give some experimental comparative results. Besides an improved functional version of Antimirov and Mosses's algorithm, we present an alternative one using partial derivatives. Our preliminary results lead to the conclusion that, indeed, these methods are feasible and, most of the time, faster than the classical methods.

Keywords: regular languages, regular expressions, derivatives, partial derivatives, regular expression equivalence, minimal automata, rewriting systems

1. Introduction

Although, for efficiency reasons, finite automata are normally used for regular language manipulation, regular expressions provide a particularly good notation for the representation of this class of languages. The problem of deciding whether two regular expressions are equivalent is, however, PSPACE-complete [$SM73$]. This decision problem is normally solved by transforming each regular expression into an equivalent NFA, converting those automata to equivalent deterministic ones, and finally minimizing both DFAs, and decide if the resulting automata are isomorphic. In the worst case, the complexity of the automata determinization process is exponential in the number of states.

Antimirov and Mosses [$AM94$] presented a rewrite system for deciding the equivalence of extended regular expressions based on a new complete axiomatization of the extended algebra of regular sets. This axiomatization, or any other classical

complete axiomatization of the standard algebra of regular sets, can be used to construct an algorithm for deciding the equivalence of two regular expressions, but these deduction systems tend to be quite inefficient. This rewrite system acts as a refutation method that normalizes regular expressions in such a way that testing their equivalence corresponds to an iterated process of testing the equivalence of their derivatives. Termination is assured because the set of derivatives to be considered is finite and possible cycles are detected using *memoization*. Antimirov and Mosses suggested that their method could lead to a better average-case algorithm than those based on the comparison of the equivalent minimal DFAs. In this paper we present a functional approach to the Antimirov-Mosses method, prove its correctness, and give some experimental comparative results. Besides an improved functional version of Antimirov and Mosses's algorithm, we present an alternative one using partial derivatives. Our preliminary results lead to the conclusion that indeed these methods are feasible and, most of the time, faster than the classical methods.

The paper is organized as follows. Section 2 contains several basic definitions and facts concerning regular languages and regular expressions. In Section 3 we present our variant of Antimirov and Mosses's method for testing the equivalence of two regular expressions. An improved version using partial derivatives is also presented. Section 4 gives some experimental comparative results between classical methods and the one presented in Section 3. Finally, in Section 5 we discuss some open problems, as ongoing and future work.

2. Regular expressions and automata

Here we recall some definitions and facts concerning regular languages, regular expressions and finite automata. For further details we refer the reader to the works of Hopcroft *et al.* [$HMU00$], Kozen [$Koz97$], and Kuich and Salomaa [$KS86$].

Let Σ be an alphabet and $\Sigma^\star$ be the set of all *words* over Σ. The *empty word* is denoted by ϵ and the length of a word w is denoted by $|w|$. A language is a subset of $\Sigma^\star$, and if L_1 and L_2 are two languages, then $L_1 \cdot L_2 = \{xy \mid x \in L_1 \text{ and } y \in L_2\}$. A regular expression α over Σ represents a (regular) language $L(\alpha) \subseteq \Sigma^\star$ and is inductively defined as follows: $\emptyset$ is a regular expression and $L(\emptyset) = \emptyset$; ϵ is a regular expression and $L(\epsilon) = \{\epsilon\}$; $a \in \Sigma$ is a regular expression and $L(a) = \{a\}$; if α and β are regular expressions, $(\alpha+\beta)$, $(\alpha\cdot\beta)$ and $(\alpha)^\star$ are regular expressions, respectively with $L((\alpha+\beta)) = L(\alpha) \cup L(\beta)$, $L((\alpha\cdot\beta)) = L(\alpha)\cdot L(\beta)$ and $L((\alpha)^\star) = L(\alpha)^\star$. The operator $\cdot$ is often omitted. We adopt the usual convention that $\star$ has precedence over $\cdot$, which has higher precedence than $+$. Let RE be the set of regular expressions over Σ. The size of α is denoted by $|\alpha|$ and represents the number of symbols, operators, and parentheses in α. We denote by $|\alpha|_\Sigma$ the number of symbols in α. We define the *constant part* of α as $\varepsilon(\alpha) = \epsilon$ if $\epsilon \in L(\alpha)$, and $\varepsilon(\alpha) = \emptyset$ otherwise. In the first case we say that α possesses the *empty word property*. Two regular expressions α and β are *equivalent*, and we write $\alpha \sim \beta$, if $L(\alpha) = L(\beta)$. The algebraic structure $(RE, +, \cdot, \emptyset, \epsilon)$, constitutes an idempotent semi-ring, and, with

the unary operator $\star$, a Kleene algebra. There are several well-known complete (non purely equational) axiomatizations of Kleene algebras [$Sal66, Koz94$], but we will essentially consider Salomaa's axiom system F_1 which, besides the usual axioms for an idempotent semi-ring, contains the following two axioms for the $\star$ operator:

$$\alpha^\star \sim \epsilon + \alpha\alpha^\star;\ \ \alpha^\star \sim (\epsilon + \alpha)^\star.$$

As for rules of inference, system F_1 has the usual rule of substitution and the following rule of *solution of equations*:

$$\frac{\alpha \sim \beta\alpha + \gamma, \quad \varepsilon(\beta) = \emptyset}{\alpha \sim \beta^\star\gamma} \tag{1}$$

A *nondeterministic finite automaton* (NFA) A is a tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is the finite set of states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times Q$ the transition relation, q_0 the initial state and $F \subseteq Q$ the set of final states. An NFA without ϵ-transitions is *deterministic* (DFA) if, for each pair $(q, a) \in Q \times \Sigma$ there exists at most one q' such that $(q, a, q') \in \delta$. Two NFA are *equivalent* if they accept the same language. A DFA is called *minimal* if there is no equivalent DFA with fewer states. Minimal DFAs are unique up to isomorphism. DFAs, NFAs, and regular expressions represent the same set of languages, *i.e.*, regular languages.

2.1. *Succinct regular expressions*

Equivalent regular expressions do not need to have the same size. *Irreducible* regular expressions as defined by Ellul *et.al* [$EKSW05$] have no redundant occurrences of $\emptyset$, ϵ, $\star$, and parentheses. A regular expression α is *uncollapsible* if **none** of the following conditions hold:

- α contains the proper sub-expression $\emptyset$, and $|\alpha| > 1$;
- α contains a sub-expression of the form $\beta\gamma$ or $\gamma\beta$ where $L(\beta) = \{\epsilon\}$;
- α contains a sub-expression of the form $\beta + \gamma$ or $\gamma + \beta$ where $L(\beta) = \{\epsilon\}$ and $\epsilon \in L(\gamma)$.

A regular expression α is *irreducible* if it is uncollapsible and **both** the following conditions are true:

- α does not contain superfluous parentheses (we adopt the usual operator precedence conventions and omit outer parentheses);
- α does not contain a sub-expression of the form $\beta^{\star^\star}$.

The previous reductions rely on considering regular expressions *modulo* some algebraic properties: identity elements of $+$ and $\cdot$, annihilator element for $\cdot$, and idempotence of $\star$.

Let ACI be the set of axioms that includes the associativity, commutativity and idempotence of disjunction and let $ACIA$ be the set ACI plus the associativity of concatenation. In this work, besides where otherwise stated, we consider irreducible regular expressions modulo $ACIA$ (and denote them by RE).

This allows a more succinct representation of regular expressions, and it is essential for ensuring the termination of the algorithms described in the next section.

2.2. *Implementation*

As already stated, throughout this paper we always consider irreducible regular expressions modulo $ACIA$. Our implementation aims to ensure these properties in a simple and efficient way. We used an object-oriented approach such that, for each operation, there is a data structure which enforces the $ACIA$ properties and simplifies the algorithms used to assure that the regular expressions are kept irreducible.

2.2.1. *Disjunctions*

A disjunction is represented as a set of regular expressions. This gives us a natural way to enforce the ACI properties.

In order to have an irreducible disjunction, the sub-expression $\emptyset$ may not occur, and, if any of the arguments possesses the empty word property, ϵ is not allowed as a sub-expression. The set representation allows for an algorithm which performs these checks in linear time on the number of arguments. As an example, consider the regular expression $\alpha + \emptyset + \beta^\star\gamma + \alpha + \epsilon$, where $\epsilon \notin L(\gamma)$. It would be represented by the set $\{\epsilon, \alpha, \beta^\star\gamma\}$.

In order to reduce the complexity of the algorithm that checks the constant part of a given sub-expression α, we memoize the results. As such, when a regular expression β is to be constructed, we need only to compute the disjunction (if β is a sum) or conjunction (if β is a concatenation) of the already computed constant part of each argument (the constant part of a starred expression will always be ϵ). Again, we save this value as a property of β in order to avoid future recursive calls.

2.2.2. *Concatenation*

Concatenations of regular expressions are kept in a list. This allows us to take advantage of the associative property and easily apply transformations to any pair of adjacent regular expressions. This representation also simplifies the application of the following transformations which are necessary to make the regular expressions irreducible, as we can simply go through the list and remove each occurrence of ϵ in linear time:

$$\alpha \cdot \epsilon \to \alpha, \quad \epsilon \cdot \alpha \to \alpha, \quad \alpha \cdot \emptyset \to \emptyset, \quad \emptyset \cdot \alpha \to \emptyset.$$

If $\emptyset$ is found, we simply return it as the equivalent irreducible regular expression. These transformations are better illustrated by the following examples:

$$\alpha \cdot \epsilon \cdot \beta \to [\alpha, \beta]; \quad \alpha \cdot \emptyset \cdot \beta \to \emptyset; \quad (\alpha \cdot \beta^\star) \cdot \gamma \to [\alpha, \beta^\star, \gamma]; \quad \alpha \cdot (\beta^\star \cdot \gamma) \to [\alpha, \beta^\star, \gamma];$$

2.2.3. *Kleene star*

As for the Kleene star, we use a class to represent the $\star$ operator.

In order to keep it irreducible, the constructor of the class does not create regular expressions of the form $\alpha^{\star\star}$. This is achieved by checking, in constant time, if the regular expression passed as an argument is already of the same type. If this is the case, only the argument is kept, thus avoiding the double star. We also added the following two simplifications to the star operator representation:

$$\emptyset^\star \to \epsilon, \quad \epsilon^\star \to \epsilon,$$

which can be very useful to the system described in Section 3.

2.3. *Linear regular expressions*

A regular expression α is *linear* if it is of the form $a_1\alpha_1 + \cdots + a_n\alpha_n$ for $a_i \in \Sigma$ and $\alpha_i \in RE$. The set of all the linear regular expressions is denoted by RE_{lin}, and can be defined by the following context-free grammar G_1, where A is the initial symbol, $L(C) = \Sigma$, and $L(B) = RE - \{\epsilon, \emptyset\}$:

$$A \to C \mid C \cdot B \mid A + A. \tag{G_1}$$

We say that an expression $a\beta$ has *head* $a \in \Sigma$ and *tail* β. We denote by $\mathrm{head}(\alpha)$ and $\mathrm{tail}(\alpha)$, respectively, the multiset of all heads and the multiset of all tails in a linear regular expression α. A linear regular expression α is *deterministic* if no element of $\mathrm{head}(\alpha)$ occurs more than once. We denote the set of all deterministic linear regular expressions by RE_{det}. Every regular expression α can be written as a disjunction of its constant part and a (deterministic) linear regular expression [*Sal*66]. A regular expression is said to *pre-linear* if it belongs to the language generated by the following context-free grammar G_2 with initial symbol A', and A and B are as in G_1:

$$\begin{aligned} A' &\to \emptyset \mid D \\ D &\to A \mid D \cdot B \mid D + D. \end{aligned} \tag{G_2}$$

The set of all pre-linear regular expressions is denoted by RE_{plin}.

2.4. *Derivatives*

The *derivative* [*Brz*64] of a regular expression α with respect to a *symbol* $a \in \Sigma$, denoted $a^{-1}(\alpha)$, is defined recursively on the structure of α as follows:

$$a^{-1}(\emptyset) = \emptyset; \qquad a^{-1}(\alpha + \beta) = a^{-1}(\alpha) + a^{-1}(\beta);$$

$$a^{-1}(\epsilon) = \emptyset; \qquad a^{-1}(\alpha\beta) = a^{-1}(\alpha)\beta + \varepsilon(\alpha)a^{-1}(\beta);$$

$$a^{-1}(b) = \begin{cases} \epsilon, & \text{if } b = a; \\ \emptyset, & \text{otherwise;} \end{cases} \qquad a^{-1}(\alpha^\star) = a^{-1}(\alpha)\alpha^\star.$$

If α is a deterministic linear regular expression, we have:

$$a^{-1}(\alpha) = \begin{cases} \beta, & \text{if } a \cdot \beta \text{ is a sub-expression of } \alpha; \\ \epsilon, & \text{if } \alpha = a; \\ \emptyset, & \text{otherwise.} \end{cases}$$

The *derivative* of a regular expression α with respect to the *word* $w \in \Sigma^\star$, denoted $w^{-1}(\alpha)$, is defined recursively on the structure of w:

$$\epsilon^{-1}(\alpha) = \alpha; \quad (ua)^{-1}(\alpha) = a^{-1}(u^{-1}(\alpha)), \text{ for any } u \in \Sigma^\star.$$

Considering regular expressions modulo the ACI axioms, Brzozowski [$Brz64$] proved that, for every regular expression α, the set of its derivatives with respect to any word w is finite.

3. Regular expression equivalence

The classical approach to the problem of comparing two regular expressions α and β, *i.e.*, deciding if $L(\alpha) = L(\beta)$, typically consists of transforming each regular expression into an equivalent NFA, convert those automata to equivalent deterministic ones, and minimize both DFAs. Because, for a given regular language, the minimal DFA is unique up to isomorphism, these can be compared using a canonical representation [$AMR07b$], and thus checked if $L(\alpha) = L(\beta)$. In this section, we present two methods to verify the equivalence of two regular expressions. The first method is a variant of the rewrite system presented by Antimirov and Mosses [$AM94$], which provides an algebraic calculus for testing the equivalence of two regular expressions without the construction of the canonical minimal automata. It is a functional approach on which we always consider the regular expressions to be irreducible and not extended (with intersection). The use of irreducible regular expressions allows us to avoid the simplification step of Antimirov and Mosses's system with little overhead. The second method improves this first one by using the notion of partial derivative.

3.1. *Linearization of regular expressions*

Let $a \in \Sigma$, and α, β, γ be arbitrary regular expression. We define the functions $\mathrm{lin} = \mathrm{lin}_2 \circ \mathrm{lin}_1$, and det as follows:

$$
\begin{aligned}
\mathrm{lin}_1 : RE &\to RE_{plin} \\
\mathrm{lin}_1(\emptyset) &= \emptyset; \\
\mathrm{lin}_1(\epsilon) &= \emptyset; \\
\mathrm{lin}_1(a) &= a; \\
\mathrm{lin}_1(\alpha+\beta) &= \mathrm{lin}_1(\alpha) + \mathrm{lin}_1(\beta); \\
\mathrm{lin}_1(\alpha^\star) &= \mathrm{lin}_1(\alpha)\alpha^\star; \\
\mathrm{lin}_1(a\alpha) &= a\alpha; \\
\mathrm{lin}_1((\alpha+\beta)\gamma) &= \mathrm{lin}_1(\alpha\gamma) + \mathrm{lin}_1(\beta\gamma); \\
\mathrm{lin}_1(\alpha^\star\beta) &= \mathrm{lin}_1(\alpha)\alpha^\star\beta + \mathrm{lin}_1(\beta).
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{lin}_2 : RE_{plin} &\to RE_{lin} \cup \{\emptyset\} \\
\mathrm{lin}_2(\alpha+\beta) &= \mathrm{lin}_2(\alpha) + \mathrm{lin}_2(\beta); \\
\mathrm{lin}_2((\alpha+\beta)\gamma) &= \mathrm{lin}_2(\alpha\gamma) + \mathrm{lin}_2(\beta\gamma); \\
\mathrm{lin}_2(\alpha) &= \alpha. \quad \text{(Otherwise)}
\end{aligned}
$$

$$
\begin{aligned}
\det : RE_{lin} \cup \{\emptyset\} &\to RE_{det} \cup \{\emptyset\} \\
\det(a\alpha + a\beta + \gamma) &= \det(a(\alpha+\beta) + \gamma); \\
\det(a\alpha + a\beta) &= a(\alpha+\beta); \\
\det(a\alpha + a) &= a(\alpha+\epsilon); \\
\det(\alpha) &= \alpha. \qquad \text{(Otherwise)}
\end{aligned}
$$

The function lin linearizes regular expressions. Function lin_1 corresponds to the function f of the original rewrite system which, contrary to what is claimed by Antimirov and Mosses, returns a pre-linear regular expression, and not a linear one.

We use the function lin for efficiency reasons because a single call makes all the derivatives with regard to *any* symbol of the alphabet readily available.

To show that $\mathrm{lin}(\alpha)$ returns either the linear part of α or $\emptyset$, it is enough to observe the following facts, of which we present only the proof for the first. The remaining proofs can be found in an extended version of the present paper (along with all other missing proofs) [$AMR07a$].

- The function lin_1 is well defined.
- For $\alpha \in RE$, $\mathrm{lin}_1(\alpha) \in L(G_2)$.
- For $\alpha \in RE_{plin}$, $\alpha \sim \mathrm{lin}_2(\alpha)$.
- For $\alpha \in RE$, $\mathrm{lin}(\alpha) \in L(G_1) \cup \{\emptyset\}$.
- For $\alpha \in RE_{lin} \cup \{\emptyset\}$, $\det(\alpha) \in RE_{det}$ and $\alpha \sim \det(\alpha)$.
- For $\alpha \in RE$, $L(\mathrm{lin}(\alpha)) = \begin{cases} L(\alpha), & \text{if } \epsilon \notin L(\alpha); \\ L(\alpha) - \{\epsilon\}, & \text{if } \epsilon \in L(\alpha). \end{cases}$

Lemma 1. *The function* lin_1 *is well defined.*

Proof. Let $a \in \Sigma$ and α, β, γ be arbitrary regular expressions. We proceed by induction on the structure of the regular expressions.

It is clear that for $\emptyset$, ϵ, a, $\alpha = \beta + \gamma$, and $\alpha = \beta^\star$ the function $\mathrm{lin}_1(\alpha)$ is well defined. We need only to show that $\mathrm{lin}_1(\alpha)$ is also well defined when α is a concatenation of regular expressions. These are all the possible cases:

$$\emptyset \cdot \alpha; \qquad \alpha \cdot \emptyset; \qquad \epsilon \cdot \alpha; \qquad \alpha \cdot \epsilon; \tag{6}$$

$$a \cdot \alpha; \qquad (\alpha+\beta) \cdot \gamma; \qquad \alpha^\star \cdot \beta. \tag{7}$$

Because we are dealing with irreducible regular expressions modulo $ACIA$, $\emptyset \cdot \alpha \sim \alpha \cdot \emptyset \sim \emptyset$ and $\text{lin}_1(\emptyset)$ is well defined. For the same reason, $\epsilon \cdot \alpha \sim \alpha \cdot \epsilon \sim \alpha$, so we do not have to consider concatenations with ϵ. This leaves us with the cases in (7), all of which are explicitly considered by the function $\text{lin}_1(\alpha)$. □

Thus we have:

Theorem 2. *For any regular expression α, $\alpha \sim \varepsilon(\alpha) + \text{lin}(\alpha)$, and $\alpha \sim \varepsilon(\alpha) + \det(\text{lin}(\alpha))$.*

Considering the definition of derivative (Subsection 2.4), we also have:

Theorem 3. *Let $a \in \Sigma$ and $\alpha \in RE$, then $a^{-1}(\alpha) = a^{-1}(\det(\text{lin}(\alpha)))$.*

3.2. *Regular expression equivalence*

We now present the function *equiv* that implements the comparison method which pseudocode is listed as Algorithm 1. The auxiliar function *derivatives* computes the set of the derivatives of a pair of deterministic linear regular expressions (α, β), with respect to each symbol of the alphabet. It is enough to consider only the symbols in $\text{head}(\alpha) \cup \text{head}(\beta)$, and we do that for efficiency reasons. The function is defined as follows:

$$\begin{aligned} \text{derivatives} &: (RE_{det} \cup \{\emptyset\}) \times (RE_{det} \cup \{\emptyset\}) \to \mathcal{P}(RE \times RE) \\ \text{derivatives}(\alpha, \beta) &= \{\, (a^{-1}(\alpha), a^{-1}(\beta)) \mid a \in \text{head}(\alpha) \cup \text{head}(\beta) \,\}. \end{aligned}$$

The function *equiv*, applied to two regular expressions α and β, returns $True$ if and only if $\alpha \sim \beta$. It is defined in the following way:

$$\begin{aligned} \text{equiv} &: \mathcal{P}(RE^2) \times \mathcal{P}(RE^2) \to \{True, False\} \\ \text{equiv}(\emptyset, H) &= True; \\ \text{equiv}(\{(\alpha, \beta)\} \cup S, H) &= \begin{cases} False, & \text{if } \varepsilon(\alpha) \neq \varepsilon(\beta); \\ \text{equiv}(S \cup S', H'), & \text{otherwise;} \end{cases} \end{aligned}$$

where

$$\begin{aligned} \alpha' &= \det(\text{lin}(\alpha)); \qquad & S' &= \{\, p \mid p \in \text{derivatives}(\alpha', \beta'),\ p \notin H' \,\}; \\ \beta' &= \det(\text{lin}(\beta)); \qquad & H' &= \{\, (\alpha, \beta) \,\} \cup H. \end{aligned}$$

At each step the function *equiv* proceeds by replacing in a set S each pair of regular expressions by the pairs of its corresponding derivatives. When either a disagreement pair is found, *i.e.*, a pair of derivatives such that their constant parts are different, or the set S is empty, the function returns. If $\alpha \sim \beta$ the call $\text{equiv}(\{(\alpha, \beta)\})$ returns the value $True$, otherwise returns $False$. Comparing with Antimirov and Mosses's rewrite system TR, we note that in each call to $\text{equiv}(S, H)$, the set S contains only pairs of regular expressions which are not in H, thus rendering the rule (IND) of TR unnecessary. On the other hand, our data structures

avoid the need of the rule (SIM) by assuring that the regular expressions are always irreducible.

```
def equiv(S,H):
    if S = ∅:
        return True
    S = S \ {(α, β)}
    if ε(α) ≠ ε(β):
        return False
    H′ = {(α, β)} ∪ H
    α′ = det(lin((α)))
    β′ = det(lin((β)))
    S′ = derivatives(α′, β′) \ H′
    return equiv(S ∪ S′,H′)
```

Algorithm 1. An improved functional version of Antimirov and Mosses's rewrite system.

Theorem 4. *The function* equiv *is terminating.*

Proof. It is clear that the function terminates when its first argument, the set S, is empty. Each call to the function removes one element from S, and appends the set of the derivatives which have not yet been calculated, S', to S. By Theorem 3, we know that the linearization process does not affect the derivation process. This sequence of derivatives with respect to a symbol is equivalent to a single derivative with respect to a word w. As we proceed by lexicographical order, all different derivatives will be considered. Because all operations are performed modulo $ACIA$, there is only a finite number of derivatives with respect to any word [$Brz64$], and, from a given point on, $S \cup S' = S$. As each call to equiv removes one element from S, after a finite number of function evaluations, $S = \emptyset$ and the function terminates. In order to assure that the derivatives of the same pair of regular expressions are not computed more than once, and thus prevent a possible infinite loop, we keep in the set H the history of all the already processed pairs of regular expressions. □

Lemma 5. *Given* $\alpha, \beta \in RE_{det} \cup \{\emptyset\}$,

$$\alpha \sim \beta \Rightarrow \forall(\alpha', \beta') \in \text{derivatives}(\alpha, \beta),\ \alpha' \sim \beta'.$$

Lemma 6. *Given two regular expressions,* α *and* β*, such that* $\alpha \sim \beta$*,*

$$\text{equiv}(\{(\alpha, \beta)\}, \emptyset) = True.$$

Proof. If $\alpha = \beta = \emptyset$,

$$\text{equiv}(\{(\emptyset, \emptyset)\}, H) = \text{equiv}(\emptyset, H \cup \{(\emptyset, \emptyset)\}) = True.$$

If $\alpha \sim \beta$ we know, by Lemma 5, that

$$\varepsilon(\alpha') = \varepsilon(\beta') \qquad \forall(\alpha', \beta') \in \text{derivatives}(\alpha, \beta)$$

and thus, by iteration, the call to equiv$(\{(\alpha, \beta)\}, \emptyset)$ will never return $False$. □

Lemma 7. *Given two regular expressions, α and β, such that $\alpha \nsim \beta$,*

$$\text{equiv}(\{(\alpha, \beta)\}, \emptyset) = False.$$

Proof. As Brzozowski shows, if $w \in L(\alpha)$, $\varepsilon(w^{-1}(\alpha)) = \epsilon$. If $\alpha \nsim \beta$, either

$$\exists w \in L(\alpha) : \varepsilon(w^{-1}(\beta)) \neq \epsilon \qquad \text{or} \qquad \exists w \in L(\beta) : \varepsilon(w^{-1}(\alpha)) \neq \epsilon.$$

Without loss of generality, suppose the first case is true. We have that

$$\text{equiv}(\{(w^{-1}(\alpha), w^{-1}(\beta))\} \cup S, H) = False$$

and we know that the call to equiv$(\{(\alpha, \beta)\}, \emptyset)$, must call equiv$(\{(\alpha', \beta')\} \cup S, H)$ such that α' is $w^{-1}(\alpha)$ and β' is $w^{-1}(\beta)$. □

Theorem 8. *The call* equiv$(\{(\alpha, \beta)\}, \emptyset)$ *returns True if and only if $\alpha \sim \beta$.*

Proof. By direct application of Lemmas 6 and 7. □

3.3. *Improved equivalence method using partial derivatives*

Antimirov [*Ant*96] introduced the notion of the partial derivatives set of a regular expression α and proved that its cardinality is bounded by the number of alphabetic symbols that occurs in α. He showed that this set can be obtained directly from a new linearization process of α. This new process can be easily implemented in our approach, as a variant of the linearization function, as we already consider disjunctions as sets. We now briefly review this notion and show how it can be used to improve the equiv algorithm.

3.3.1. *Linear forms*

Let $\Sigma \times RE$ be the set of *monomials* over an alphabet Σ. Let $\mathcal{P}_{fin}(A)$ be the set of all finite parts of the set A. A linear regular expression $a_1\alpha_1 + \cdots + a_n\alpha_n$ can be represented by a finite set of monomials $l \in \mathcal{P}_{fin}(\Sigma \times RE)$, named *linear form*, and such that $l = \{(a_1, \alpha_1), \ldots, (a_n, \alpha_n)\}$. We define a function $\sigma : \mathcal{P}_{fin}(\Sigma \times RE) \to RE_{lin}$ by $\sigma(l) = a_1\alpha_1 + \cdots + a_n\alpha_n$.

Concatenation of a linear form l with a regular expression β is defined by $l\beta = \{(a_1, \alpha_1\beta), \ldots, (a_n, \alpha_n\beta)\}$.

The linearization of a regular expression α is then defined as follows:

$$\begin{aligned}
&\text{lf} : RE \to \mathcal{P}_{fin}(\Sigma \times RE)\\
&\text{lf}(\emptyset) = \emptyset; && \text{lf}(\alpha^\star) = \text{lf}(\alpha) \cdot \alpha^\star;\\
&\text{lf}(\epsilon) = \emptyset; && \text{lf}(a \cdot \alpha) = \{(a, \alpha)\};\\
&\text{lf}(a) = \{(a, \epsilon)\}; && \text{lf}((\alpha + \beta) \cdot \gamma) = \text{lf}(\alpha \cdot \gamma) \cup \text{lf}(\beta \cdot \gamma);\\
&\text{lf}(\alpha + \beta) = \text{lf}(\alpha) \cup \text{lf}(\beta); && \text{lf}(\alpha^\star \cdot \beta) = \text{lf}(\alpha) \cdot \alpha^\star \cdot \beta \cup \text{lf}(\beta).
\end{aligned}$$

The following theorem relates the method of linearization presented in Section 2.3 with linear forms.

Theorem 9. *For any regular expression* α, $\mathrm{lin}(\alpha) = \sigma(lf(\alpha))$.

3.3.2. *Partial derivatives*

Given a regular expression α and a symbol $a \in \Sigma$, a *partial derivative* of α with respect to a is a regular expression ρ such that $(a, \rho) \in \mathrm{lf}(\alpha)$. The set of partial derivatives of α with respect to a is denoted by $\partial_a(\alpha)$. The notion of partial derivative of α can be extended to words $w \in \Sigma^\star$, sets of regular expressions $R \subseteq RE$, and sets of words $W \subseteq \Sigma^\star$, as follows:

$$\partial_\epsilon(\alpha) = \{\alpha\}; \qquad \partial_w(R) = \bigcup_{\alpha \in R} \partial_w(\alpha);$$

$$\partial_{ua}(\alpha) = \partial_a(\partial_u(\alpha)), \text{ for any } u \in \Sigma^\star; \qquad \partial_W(\alpha) = \bigcup_{w \in W} \partial_w(\alpha).$$

There is a strong connection between the sets of partial derivatives and the derivatives of a regular expression. Trivially extending the notion of language represented by a regular expression to sets of regular expressions, we have that $L(\partial_w(\alpha)) = L(w^{-1}(\alpha))$, for any $w \in \Sigma^\star$, $\alpha \in RE$. One of the advantages of using partial derivatives is that for any $\alpha \in RE$, $|PD(\alpha) = \partial_{\Sigma^\star}(\alpha)| \leq |\alpha|_\Sigma$, where $PD(\alpha)$ stands for the set of all the syntactically different partial derivatives.

3.3.3. *Improving* equiv *by using partial derivatives*

Let us now consider a determinization process for linear forms. We say that a linear form is deterministic if, for each symbol $a \in \Sigma$, there is at most one element of the form (a, α). Let lfX be an extended version of the linearization function lf, defined as follows:

$$\mathrm{lfX}(\alpha) = \{(a, \sum_{(a,\alpha') \in \mathrm{lf}(\alpha)} \alpha') \mid a \in \Sigma\}.$$

We can replace the function composition $det \circ lin$ with the deterministic linear form obtained with lfX. This new extended linear form allows us to use the previously defined *equiv* function with only two slight modifications. We redefine the *derivatives* function as follows:

$$\begin{aligned} \text{derivatives} \quad &: \quad \mathcal{P}^{det}_{fin}(\Sigma \times RE) \times \mathcal{P}^{det}_{fin}(\Sigma \times RE) \to \mathcal{P}(RE \times RE) \\ \text{derivatives}(l_\alpha, l_\beta) &= \{(\alpha', \beta') \mid (a, \alpha') \in l_\alpha, (a, \beta') \in l_\beta\}. \end{aligned}$$

By making $l_\alpha = \mathrm{lfX}(\alpha)$, $l_\beta = \mathrm{lfX}(\beta)$, and

$$S' = \{p \mid p \in \text{derivatives}(l_\alpha, l_\beta), p \notin H'\}, \quad H' = \{(\alpha, \beta)\} \cup H$$

we obtain a new version of the equiv function, which will be called equivP.

3.3.4. *Complexity issues*

The worst-case time complexity of the lf function, can be estimated considering that the associated recurrence is $T(n) = 2T(2n/3) + n$, where n is the size of the regular expression given as argument. Then by direct application of the master theorem [$CLRS03$], we conclude that $T(n)$ is bounded by $\Theta(n^{1.7})$. As for the equivP function, let $n = \max(|\alpha|_\Sigma, |\beta|_\Sigma)$. It proceeds by comparing subsets of $PD(\alpha)$ with $PD(\beta)$, whose size is bounded by n. This leads to 2^n possible comparisons, in the worst case.

4. Experimental results

We will now present some experimental results. These are the running times for the two methods for checking the equivalence of regular expressions. One uses the equivalent minimal DFA, the other is the direct regular expression comparison method, as described in Section 3. All tests were performed on batches of $10,000$ pairs of uniformly random generated regular expressions, and the running times do not include the time necessary to parse each regular expression. Each batch contains regular expressions of size 10, 50 or 100, with either 2, 5 or 10 symbols. For the uniform generation of random regular expressions we implemented the method described by Mairson [$Mai94$] for the generation of context-free languages. We used a grammar for almost irreducible regular expressions presented by Shallit [$LS05$]. As the data sets were obtained with a uniform random generator, the size of each sample is sufficient to ensure a 95% confidence level within a 1% error margin. It is calculated with the formula $n = (\frac{z}{2\epsilon})^2$, where z is obtained from the normal distribution table such that $P(-z < Z < z)) = \gamma$, ϵ is the error margin, and γ is the desired confidence level.

We tested the equivalence of each pair of regular expressions using both the classical approach and the direct comparison method. We used Glushkov's algorithm to obtain the NFAs from the regular expressions, and the well-known subset construction to make each NFA deterministic. As for the DFA minimization process, we applied two different algorithms: Hopcroft and Brzozowski's. On one hand, Hopcroft's algorithm has the best known worst-case running time complexity analysis, $O(kn \log n)$. On the other, it is pointed out by Almeida *et. al* [$AMR07c$] that when minimizing NFAs, Brzozowski's algorithm has a better practical performance. As for the direct comparison method, we compared both the original rewriting system (**AM**) and our variant of the algorithm both with (**equivP**) and without partial derivatives (**equiv**).

As shown in Figure 1 (a), when comparing randomly generated regular expressions, any of the direct methods is always faster. Note also that Hopcroft's algorithm never achieves shorter running times than Brzozowski's. Because both Antimirov and Mosses's algorithm and our variation try to compute a refutation, we performed a set of tests for testing the equivalence of two syntactically equal regular expressions. Again, we used batches of $10,000$ pairs of regular expressions. Figure 1 (b)

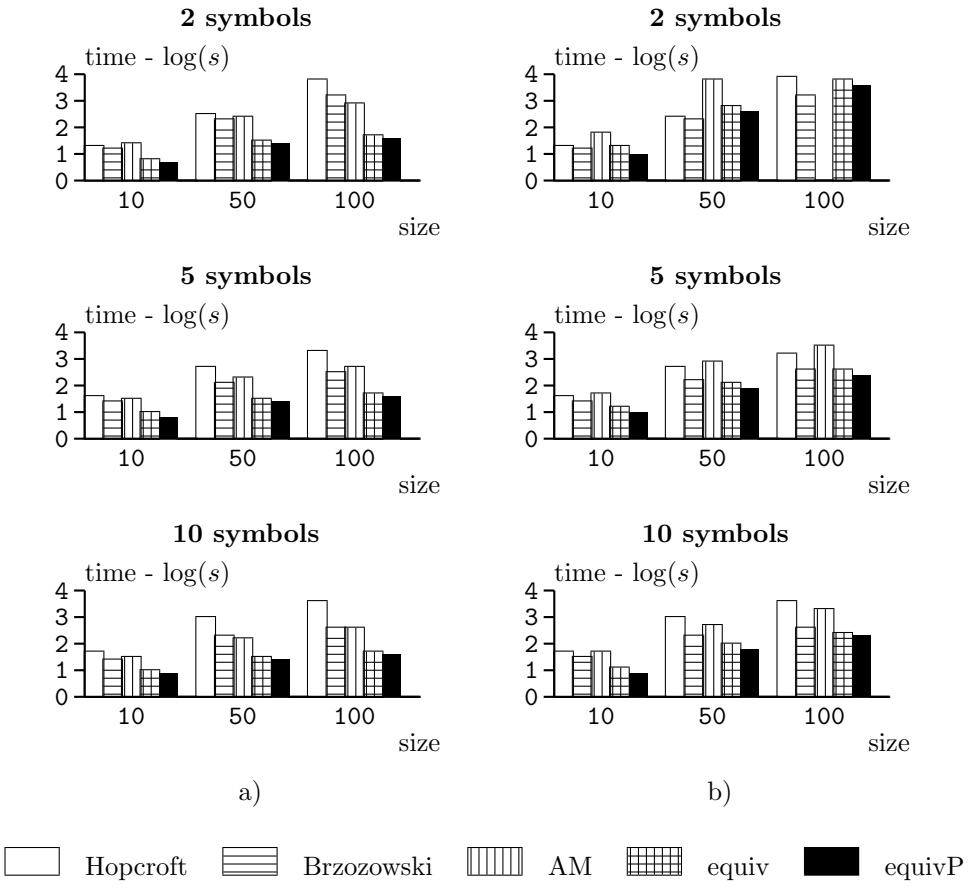

Fig. 1. Running times of three different methods for checking the equivalence of regular expressions. a) 10,000 pairs of random regular expressions; b) 10,000 pairs of syntactically equal random regular expressions. The missing column corresponds to a larger than reasonable observed runtime.

shows the results of the application of each algorithm to pairs of syntactically equal random regular expressions. Except for the samples of regular expressions with size 50 or 100, over an alphabet of 2 symbols, the direct regular expressions comparison methods are still the fastest. Again, Brzozowki's algorithm always presents better running times than Hopcroft's. Among the direct comparison methods, **equivP** always performs better, with a speedup of $20\% - 30\%$. It is important to note, however, that by using partial derivatives we memoize intermediate results and thus avoid the recomputation of some elements of the partial derivatives set.

4.1. *More experimental data*

We analyzed the time spent on each step of the minimal DFA construction method, *i.e.*, on the construction of the NFAs (**NFA**), the determinization process (**DFA**),

and Hopcroft's minimization algorithm (**mDFA**). The running time for each of these steps is presented in Table 1. It is clear that, asymptotically, the bottleneck is the minimization algorithm, which always takes over 50% of the total amount of time when the size of the regular expressions and/or the alphabet grows. The

$k = 2$	$n = 10$		$n = 50$		$n = 100$	
	$\|Q\|$	time	$\|Q\|$	time	$\|Q\|$	time
NFA	$(7.35, 7.33)$	6.76	$(29.02, 29.01)$	60.11	$(56.22, 56.15)$	177.81
DFA	$(6.56, 6.55)$	3.82	$(34.82, 34.89)$	47.88	$(115.79, 114.34)$	312.8
mDFA	$(6.49, 6.45)$	9.29	$(24.70, 24.78)$	203.15	$(63.12, 61.50)$	6473.63

$k = 5$	$n = 10$		$n = 50$		$n = 100$	
	$\|Q\|$	time	$\|Q\|$	time	$\|Q\|$	time
NFA	$(8.89, 8.90)$	8.46	$(36.45, 36.51)$	66.12	$(70.84, 70.93)$	191.42
DFA	$(8.60, 8.62)$	6.2	$(35.15, 35.20)$	36.08	$(69.07, 69.16)$	83.99
mDFA	$(8.42, 8.45)$	25.74	$(29.35, 29.39)$	406.04	$(54.33, 54.54)$	1700.31

$k = 10$	$n = 10$		$n = 50$		$n = 100$	
	$\|Q\|$	time	$\|Q\|$	time	$\|Q\|$	time
NFA	$(9.65, 9.63)$	8.25	$(40.87, 40.87)$	68.07	$(79.62, 79.64)$	191.19
DFA	$(9.52, 9.51)$	8.69	$(40.05, 40.04)$	52	$(77.91, 77.93)$	116.47
mDFA	$(9.54, 9.50)$	41.3	$(35.10, 35.10)$	981.91	$(66.03, 66.04)$	4341.69

Table 1. Running times (seconds) for each step of the regular expressions comparison with Hopcroft's algorithm.

k	2			5			10		
n	10	50	100	10	50	100	10	50	100
AM	2.71	3.54	3.73	5.74	9.82	12.45	8.22	14.27	17.97
equiv	2.45	3.20	3.40	4.43	7.01	8.80	6.36	10.42	12.78
equivP	2.44	3.23	3.46	4.43	7.03	8.83	6.36	10.40	12.81

Table 2. Average number of functions calls to **AM**, **equiv**, and **equivP**.

average number of states for the NFAs, the equivalent DFAs and the minimal DFAs is also presented in Table 1. The average number of recursive calls to **AM**, **equiv**, and **equivP** is presented in Table 2. The numbers are similar, so the difference on the running time of the algorithms cannot be justified by the amount of function calls. To ensure the fairness of the comparison for the methods using NFAs we tried two other algorithms for computing NFAs from regular expressions. Some statistical data about the performance of Thompson [$Tho68$], Glushkov [$Glu61, Yu97$], and Ilie and Yu's [$IY03$] algorithms for computing NFAs from regular expressions is given in Table 3. Glushkov's algorithm is always the fastest, and produces quite small NFAs, both in terms of number of states and transitions. As expected, the NFAs produced

$k=2$	$n=10$			$n=50$			$n=100$		
Alg.	$\|Q\|$	$\|\delta\|$	time	$\|Q\|$	$\|\delta\|$	time	$\|Q\|$	$\|\delta\|$	time
Thomp.	18.35	17.46	8.79	83.82	82.90	110.47	165.72	164.80	411.42
Glush.	7.45	7.37	4.48	29.45	35.96	33.83	57.05	72.90	98.02
Follow	5.91	6.10	26.75	20.30	25.36	813.64	35.93	46.52	5135.32

$k=5$	$n=10$			$n=50$			$n=100$		
Alg.	$\|Q\|$	$\|\delta\|$	time	$\|Q\|$	$\|\delta\|$	time	$\|Q\|$	$\|\delta\|$	time
Thomp.	19.54	18.60	10.79	90.38	89.42	133.31	178.38	177.42	484.02
Glush.	8.91	9.19	5.16	36.53	45.83	36.18	70.98	93.55	101.55
Follow	7.54	7.98	41.71	28.63	35.80	1723.54	53.15	68.55	12413.52

$k=10$	$n=10$			$n=50$			$n=100$		
Alg.	$\|Q\|$	$\|\delta\|$	time	$\|Q\|$	$\|\delta\|$	time	$\|Q\|$	$\|\delta\|$	time
Thomp.	19.85	18.90	11.69	94.03	93.05	153.43	185.64	184.66	554.47
Glush.	9.65	9.70	5.34	40.88	48.03	36.84	79.64	97.13	101.14
Follow	8.60	8.72	51.99	34.67	40.48	2585.59	66.18	78.84	21086.45

Table 3. Running time (seconds), average number of states and transitions for three types of NFAs obtained from random expressions.

with Thompson's algorithm are the ones with the highest number of states.

5. Conclusion

We presented a variant method based on a rewrite system for testing the equivalence of two regular expressions, that attempts to refute its equivalence by finding a pair of derivatives that disagree in their constant parts. While a good behaviour was expected for some non-equivalent regular expressions, experimental results point to a good average-case performance for this method, even when feeded with equivalent regular expressions. Some improvement was also achieved by using partial derivatives. Given the spread of multi-cores and grid computer systems, a parallel execution of the classic method and our direct comparison method can lead to an optimized framework for testing regular expressions equivalence. A better theoretical understanding of relationships between the two approaches would be helpful towards the characterization of their average-case complexity. We have related this method with the one by Hopcroft and Karp [$HK71$] for testing the equivalence of DFAs without minimization [$AMR09$]. Preliminary results show, however, that any of the direct methods we presented is likely to perform better.

6. Acknowledgements

We thanks the referees for their remarks that helped to improve this paper. This work was partially funded by Fundação para a Ciência e Tecnologia (FCT) and Program POSI, and by project ASA (PTDC/MAT/65481/2006). Marco Almeida is funded by FCT grant SFRH/BD/27726/2006.

References

[AM94] V. M. Antimirov and P. D. Mosses. Rewriting extended regular expressions. In G. Rozenberg and A. Salomaa, editors, *Developments in Language Theory*, pages 195 – 209. World Scientific, 1994.

[AMR07a] M. Almeida, M. Moreira, and R. Reis. Testing the equivalence of regular expressions. Technical Report DCC-2007-07, DCC - FC & LIACC, Universidade do Porto, November 2007.

[AMR07b] M. Almeida, N. Moreira, and R. Reis. Enumeration and generation with a string automata representation. *Theoret. Comput. Sci.*, 387(2):93–102, 2007.

[AMR07c] M. Almeida, N. Moreira, and R. Reis. On the performance of automata minimization algorithms. Technical Report DCC-2007-03, DCC - FC & LIACC, Universidade do Porto, June 2007.

[AMR09] M. Almeida, N. Moreira, and R.Reis. Testing regular languages equivalence, 2009. Submitted.

[Ant96] V. M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theoret. Comput. Sci.*, 155(2):291–319, 1996.

[Brz64] J. A. Brzozowski. Derivatives of regular expressions. *JACM*, 11(4):481–494, October 1964.

[CLRS03] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT, 2003.

[EKSW05] K. Ellul, B. Krawetz, J. Shallit, and M. Wang. Regular expressions: New results and open problems. *Journal of Automata, Languages and Combinatorics*, 10(4):407–437, 2005.

[Glu61] V. M. Glushkov. The abstract theory of automata. *Russian Math. Surveys*, 16:1–53, 1961.

[HK71] J. Hopcroft and R. M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 71-114, University of California, 1971.

[HMU00] J. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley, 2000.

[IY03] L. Ilie and S. Yu. Follow automata. *Inf. Comput.*, 186(1):140–162, 2003.

[Koz94] D. C. Kozen. A completeness theorem for Kleene algebras and the algebra of regular events. *Infor. and Comput.*, 110(2):366–390, May 1994.

[Koz97] D. C. Kozen. *Automata and Computability*. Undergrad. Texts in Computer Science. Springer-Verlag, 1997.

[KS86] W. Kuich and A. Salomaa. *Semirings, Automata, Languages*, volume 5. Springer–Verlag, 1986.

[LS05] J. Lee and J. Shallit. Enumerating regular expressions and their languages. In *9th International Conference on Implementation and Application of Automata, CIAA 2004*, volume 3314 of *LNCS*, pages 2–22. Springer-Verlag, 2005.

[Mai94] H. G. Mairson. Generating words in a context-free language uniformly at random. *Information Processing Letters*, 49:95–99, 1994.

[Sal66] A. Salomaa. Two complete axiom systems for the algebra of regular events. *Journal of the Association for Computing Machinery*, 13(1):158–169, 1966.

[SM73] L.J. Stockmeyer and A. R. Meyer. Word problems requiring exponential time: Preliminary report. In *Conf. Record of 5th Annual ACM Symposium on Theory of Computing, Austin, Texas, USA*, pages 1–9. ACM, 1973.

[Tho68] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):410–422, 1968.

[Yu97] S. Yu. Regular languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 1. Springer-Verlag, 1997.