Partial Derivative Automaton by Compressing Regular Expressions*

Stavros Konstantinidis[1], António Machiavelo[2], Nelma Moreira[2], and Rogério Reis[2]

[1] Saint Mary's University, Halifax, Nova Scotia, Canada,
s.konstantinidis@smu.ca,
[2] CMUP & DM, DCC, Faculdade de Ciências da Universidade do Porto,
Rua do Campo Alegre, 4169-007 Porto, Portugal
{antonio.machiavelo,nelma.moreira,rogerio.reis}@fc.up.pt

Abstract. The partial derivative automaton ($\mathcal{A}_{\mathsf{PD}}$) is an elegant simulation of a regular expression. Although it is, in general, smaller than the position automaton ($\mathcal{A}_{\mathsf{POS}}$), the algorithms that build $\mathcal{A}_{\mathsf{PD}}$ in quadratic worst-case time, first compute $\mathcal{A}_{\mathsf{POS}}$. Asymptotically, and on average for the uniform distribution, the size of $\mathcal{A}_{\mathsf{PD}}$ is half the size of $\mathcal{A}_{\mathsf{POS}}$, being both linear on the size of the expression. We address the construction of $\mathcal{A}_{\mathsf{PD}}$ efficiently, on average, avoiding the computation of $\mathcal{A}_{\mathsf{POS}}$. The expression and the set of its partial derivatives are represented by a directed acyclic graph with shared common subexpressions. We develop an algorithm for building $\mathcal{A}_{\mathsf{PD}}$'s from expressions in strong star normal form of size n that runs in time $O\big(n^{3/2}\sqrt[4]{\log(n)}\big)$ and space $O\big(n^{3/2}/(\log n)^{3/4}\big)$, on average. Empirical results corroborate its good practical performance.

1 Introduction

The partial derivative automaton ($\mathcal{A}_{\mathsf{PD}}$) is an elegant construction to obtain nondeterministic finite automata (without ε-transitions) from regular expressions. The use of derivatives has several advantages: they are easily extended to operations other than union, concatenation, and Kleene star; word membership can be evaluated without the need to build the automaton; and the $\mathcal{A}_{\mathsf{PD}}$ is a quotient of the position (or Glushkov) automaton ($\mathcal{A}_{\mathsf{POS}}$) [6,7]. In the worst-case, for a standard regular expression of size n, both automata can have $O(n)$ states, $O(n^2)$ transitions, and can be computed in time $O(n^2)$. However, the known algorithms to build $\mathcal{A}_{\mathsf{PD}}$ in quadratic time first compute $\mathcal{A}_{\mathsf{POS}}$ and then compute a right-invariant equivalence on the states of $\mathcal{A}_{\mathsf{POS}}$ [8,16]. For practical applications, the drawbacks of these methods are the need to build a larger automaton (which is not easy to generalize for nonstandard operations) and the computation of the equivalence relation on the set of $\mathcal{A}_{\mathsf{POS}}$ states. In particular, Khorsi et al. [16] base their algorithm on the construction and minimization of two acyclic

* Research supported by NSERC (Canada) and by CMUP through FCT project UIDB/00144/2020.

deterministic finite automata, which burdens the practical performance of the algorithm, despite their linear worst-case time.

Asymptotically, and on average for the uniform distribution, the size of $\mathcal{A}_{\mathsf{PD}}$ (both in states and transitions) is half the size of $\mathcal{A}_{\mathsf{POS}}$, being both linear on the size of the expression [19,3]. Being smaller, in general, it is interesting to know if the $\mathcal{A}_{\mathsf{PD}}$ can be built efficiently without the computation of the $\mathcal{A}_{\mathsf{POS}}$. In this paper we address this problem considering regular expressions in strong star normal form (ssnf). The star normal form was first defined to construct the position automaton in time $O(n^2)$, for expressions of size n [6]. The conversion of an expression to star normal form can be done in linear time (in both the worst and average cases). This form was extended to strong star normal form (ssnf) by Gruber and Gulan [14]. The average-case complexity of conversions from ssnf expressions to other models was studied by Broda et al. [4]. Then Konstantinidis et al. [17] considered the size of partial derivatives on the average case both for standard and ssnf expressions. For the latter, asymptotically and on average, the size of the largest partial derivative is $O(n/2)$, n being the size of the expression, while one has $O(n^{3/2})$ for the standard. Any partial derivative of an expression is a concatenation of some of its subexpressions. Thus, it is interesting to estimate the number of new concatenations obtained, on average, when partial derivatives are computed. By using a tree representation of a regular expression and its set of partial derivatives, those concatenations correspond to the new nodes that are added to the initial tree. Konstantinidis et al. showed that when computing a partial derivative w.r.t. one symbol that number is asymptotically constant.

In this paper we attain asymptotic estimates for the number of new concatenations when computing the set of all partial derivatives. To represent a regular expression and the set of its partial derivatives, instead of a tree, we consider a directed acyclic graph (DAG) with shared common subexpressions. Flajolet et al. [13] showed that a tree of size n has, in this compact form, an expected size of $O\left(n \,/\, \sqrt{\log n}\right)$. We present an algorithm that computes $\mathcal{A}_{\mathsf{PD}}(\alpha)$ by constructing a DAG for α, and simultaneously builds the set of all partial derivatives by adding new concatenation nodes to the DAG. Using the asymptotic estimates mentioned above we show that for ssnf expressions the algorithm uses, on average, time $O\left(n^{3/2}\sqrt[4]{\log(n)}\right)$ and space $O\left(n^{3/2}/(\log n)^{3/4}\right)$. Experiments for uniformly randomly generated expressions, as well as for some extreme expressions, suggest that the algorithm has a good practical performance.

2 Preliminaries

A *nondeterministic finite automaton* (NFA) is a five-tuple $A = \langle Q, \Sigma, \delta, I, F\rangle$ where Q is a finite set of states, Σ is a finite alphabet, $I \subseteq Q$ is the set of initial states, $F \subseteq Q$ is the set of final states, and $\delta : Q \times \Sigma \to 2^Q$ is the transition function. The size of an NFA is its number of states plus its number of transitions. The transition function can be extended to words and to sets of states in the natural way. The *language accepted* by A is $\mathcal{L}(A) = \{w \in \Sigma^\star \mid \delta(I, w) \cap F \neq \emptyset\}$. Given an alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_k\}$ of size $k \geq 1$, the set $\mathcal{R}_k$ of (standard)

regular expressions α over Σ consists of $\emptyset$ and the expressions defined by the following context-free grammar:

$$\alpha := \varepsilon \mid \sigma_1 \mid \cdots \mid \sigma_k \mid (\alpha + \alpha) \mid (\alpha \odot \alpha) \mid (\alpha^\star), \tag{1}$$

where the symbol $\odot$ is often omitted, and represents concatenation. The *language* associated with α is denoted by $\mathcal{L}(\alpha)$ and is defined as usual. If $S \subseteq \mathcal{R}_k$, then $\mathcal{L}(S) = \bigcup_{\alpha\in S} \mathcal{L}(\alpha)$. We say that α is *nullable* if $\varepsilon \in \mathcal{L}(\alpha)$ and, in this case, define $\varepsilon(\alpha) = \varepsilon$, with $\varepsilon(\alpha) = \emptyset$, otherwise. For the *size* of a regular expression α, denoted by $\|\alpha\|$, we will consider the size of its syntactic tree, i.e. the number of symbols in α, not counting parentheses. The *alphabetic size* of α, denoted by $|\alpha|_\Sigma$, is the number of letters in α. The notions of language, nullability and of the above measures extend in a natural way to sets of expressions. The set of letters that occur in α is denoted by Σ_α. The partial derivative automaton of a regular expression was introduced independently by Mirkin [18] and Antimirov [1]. For $\alpha \in \mathcal{R}_k$, let the *linear form* (LF) of α, $\varphi(\alpha) \subseteq \Sigma \times \mathcal{R}_k$, be inductively defined by

$$\begin{aligned} \varphi(\emptyset) = \varphi(\varepsilon) = \emptyset, \qquad & \varphi(\alpha + \alpha') = \varphi(\alpha) \cup \varphi(\alpha'), \\ \varphi(\sigma) = \{(\sigma, \varepsilon)\}, \qquad & \\ \varphi(\alpha^\star) = \varphi(\alpha)\alpha^\star, \qquad & \varphi(\alpha\alpha') = \begin{cases} \varphi(\alpha)\alpha' \cup \varphi(\alpha'), \text{ if } \varepsilon(\alpha) = \varepsilon, \\ \varphi(\alpha)\alpha' \text{ otherwise,} \end{cases} \end{aligned} \tag{2}$$

where, for any $S \subseteq \Sigma \times \mathcal{R}_k$, we define $S\emptyset = \emptyset$, $S\varepsilon = S$, and $S\alpha' = \{\, (\sigma, \alpha\alpha') \mid (\sigma, \alpha) \in S \wedge \alpha \neq \varepsilon \,\} \cup \{\, (\sigma, \alpha') \mid (\sigma, \varepsilon) \in S \,\}$ for $\alpha' \neq \emptyset, \varepsilon$. For $\alpha \in \mathcal{R}_k$ and $\sigma \in \Sigma$, *the set of partial derivatives of* α *w.r.t.* σ is defined by $\partial_\sigma(\alpha) = \{\, \alpha' \mid (\sigma, \alpha') \in \varphi(\alpha) \,\}$. Partial derivatives ($\mathsf{PD}$) can be extended w.r.t. words in a natural way, as well as w.r.t languages and, both, to sets of regular expressions. We have $\mathcal{L}(\partial_w(\alpha)) = \{\, w' \mid ww' \in \mathcal{L}(\alpha) \,\}$, for $w \in \Sigma^*$. The set of all partial derivatives of α w.r.t. nonnull words is denoted by $\partial^+(\alpha)$, and satisfies the following.

Proposition 1 ([18]).

$$\begin{aligned} \partial^+(\emptyset) = \partial^+(\varepsilon) = \emptyset, \qquad & \partial^+(\alpha + \alpha') = \partial^+(\alpha) \cup \partial^+(\alpha'), \\ \partial^+(\sigma) = \{\varepsilon\}, \qquad & \partial^+(\alpha\alpha') = \partial^+(\alpha)\alpha' \cup \partial^+(\alpha'), \\ \partial^+(\alpha^\star) = \partial^+(\alpha)\alpha^\star, \qquad & \end{aligned} \tag{3}$$

where, for any $S \subseteq \mathcal{R}_k$, *we define* $S\emptyset = \emptyset$, $S\varepsilon = S$, *and* $S\alpha' = \{\, \alpha\alpha' \mid \alpha \in S \wedge \alpha \neq \varepsilon \,\} \cup \{\, \alpha' \mid \varepsilon \in S \,\}$ *for* $\alpha' \neq \emptyset, \varepsilon$.

The set of all partial derivatives of α w.r.t. words is denoted by $\mathsf{PD}(\alpha) = \partial_{\Sigma^\star}(\alpha) = \partial^+(\alpha) \cup \{\alpha\}$. The *partial derivative automaton* of α is

$$\mathcal{A}_{\mathsf{PD}}(\alpha) = \big\langle\, \mathsf{PD}(\alpha), \Sigma, \delta_{\mathsf{PD}}, \{\alpha\}, \{\, \alpha' \in \mathsf{PD}(\alpha) \mid \varepsilon(\alpha') = \varepsilon \}\big\rangle, \tag{4}$$

with $\delta_{\mathsf{PD}}(\alpha', \sigma) = \partial_\sigma(\alpha')$, for all $\alpha' \in \mathsf{PD}(\alpha)$ and $\sigma \in \Sigma$.

Proposition 2 ([1], Th. 3.4). *For any regular expression* α, $|\partial^+(\alpha)| \leq |\alpha|_\Sigma$.

Proposition 3 ([1], Th. 3.8). *Given* $\alpha \in \mathcal{R}_k$, *a partial derivative of* α *is either* ε *or a concatenation* $\alpha_1\alpha_2\cdots\alpha_n$ *such that* α_i *is a subexpression of* α *and* $n-1$ *is no greater than the number of occurrences of concatenations and stars in* α.

Corollary 1. *For $\beta \in \partial^+(\alpha)$, the size $\|\beta\|$ is $O(\|\alpha\|^2)$.*

Proposition 4 ([1,18]). *For $\alpha \in \mathcal{R}_k$, we have $|\varphi(\alpha)| \leq |\alpha|_\Sigma$ and for $(\sigma, \alpha') \in \varphi(\alpha)$, the size $\|\alpha'\|$ is $O(\|\alpha\|^2)$. Moreover, $|\delta_{\mathsf{PD}}(\alpha)|$ is $O(|\alpha|_\Sigma^2)$. If α contains no subexpression of the form $\alpha_1^\star$, then the size $\|\alpha'\|$ is $O(\|\alpha\|)$.*

Example 1. Let $\alpha_n = a_1^\star \cdots a_n^\star$, with $|\alpha|_\Sigma = n$. Then $\partial^+(\alpha_n) = \{a_i^\star \cdots a_n^\star \mid 2 \leq i \leq n\}$, and $|\varphi(\alpha_n)| = |\alpha_n|_\Sigma = n$. The largest partial derivative has size $n-1$, and $|\delta_{\mathsf{PD}}(\alpha_n)| = \sum_{i=1}^{n-1} i = \frac{n(n+1)}{2}$.

Proposition 5 ([19,2,3]). *Asymptotically in the size of the expression, and as the alphabet size grows, the average sizes $|\alpha|_\Sigma$, $|\varphi(\alpha)|$, $|\partial^+(\alpha)|$, and $|\delta_{\mathsf{PD}}(\alpha)|$ are $\frac{\|\alpha\|}{2}$, the constant 6, $\frac{\|\alpha\|}{4}$, and $\frac{\|\alpha\|}{2}$, respectively.*

3 Strong Star Normal Form and Partial Derivatives

A regular expression is in *strong star normal form* (ssnf) if for any subexpression of the form $\beta^\star$ or $\beta+\varepsilon$, β is not nullable. Introducing the operator *option*,?, with $\mathcal{L}(\beta^?) = \mathcal{L}(\beta) \cup \{\varepsilon\}$, one can define the set $\mathcal{S}_k$ of *regular expressions in* ssnf over some alphabet $\Sigma = \{\sigma_1, \ldots, \sigma_k\}$ by the following grammar:

$$\begin{aligned} \beta &:= \varepsilon \mid \emptyset \mid \beta_\varepsilon \mid \beta_{\overline{\varepsilon}}, \\ \beta_\varepsilon &:= (\beta_\varepsilon \odot \beta_\varepsilon) \mid (\beta_\varepsilon + \beta_{\overline{\varepsilon}}) \mid (\beta_{\overline{\varepsilon}} + \beta_\varepsilon) \mid (\beta_\varepsilon + \beta_\varepsilon) \mid (\beta_{\overline{\varepsilon}}^\star) \mid (\beta_{\overline{\varepsilon}}^?), \\ \beta_{\overline{\varepsilon}} &:= \sigma_1 \mid \cdots \mid \sigma_k \mid (\beta_{\overline{\varepsilon}} \odot \beta_{\overline{\varepsilon}}) \mid (\beta_{\overline{\varepsilon}} \odot \beta_\varepsilon) \mid (\beta_\varepsilon \odot \beta_{\overline{\varepsilon}}) \mid (\beta_{\overline{\varepsilon}} + \beta_{\overline{\varepsilon}}), \end{aligned} \tag{5}$$

where β_ε is for (nontrivial) nullable regular expressions, while $\beta_{\overline{\varepsilon}}$ is for the others. In the remaining of the paper we will use β to denote either of β_ε and $\beta_{\overline{\varepsilon}}$. For $\beta \in \mathcal{S}_k$, the *linear form* $\varphi(\beta)$ is defined as in (2) for the base cases and for the union. For the remaining cases we define: $\varphi(\beta_\varepsilon\beta) = \varphi(\beta_\varepsilon)\beta \cup \varphi(\beta)$, $\varphi(\beta_{\overline{\varepsilon}}\beta) = \varphi(\beta_{\overline{\varepsilon}})\beta$, $\varphi(\beta_{\overline{\varepsilon}}^\star) = \varphi(\beta_{\overline{\varepsilon}})\beta_{\overline{\varepsilon}}^\star$, and $\varphi(\beta_{\overline{\varepsilon}}^?) = \varphi(\beta_{\overline{\varepsilon}})$. The set $\partial^+(\beta)$ of all partial derivatives of $\beta \in \mathcal{S}_k$ w.r.t. nonnull words satisfies Proposition 1 except for the following cases: $\partial^+(\beta_{\overline{\varepsilon}}^\star) = \partial^+(\beta_{\overline{\varepsilon}})\beta_{\overline{\varepsilon}}^\star$ and $\partial^+(\beta_{\overline{\varepsilon}}^?) = \partial^+(\beta_{\overline{\varepsilon}})$ [17]. In the next section, using the analytic combinatorics framework, we derive asymptotic estimates for the number of new concatenations obtained when computing $\partial^+(\beta)$ for $\beta \in \mathcal{S}_k$, and thus obtaining an average-case version of Proposition 3.

3.1 The Analytic Tools

Given some measure of the objects of a combinatorial class, $\mathcal{A}$, for each $n \in \mathbb{N}_0$, let a_n be the sum of the values of this measure for all objects of size n. Now, let $A(z) = \sum_n a_n z^n$ be the corresponding generating function (*cf.* [12]). We will use the notation $[z^n]A(z)$ for a_n. The generating function $A(z)$ can be seen as a complex analytic function, and when it has a unique dominant singularity ρ, the study of the behaviour of $A(z)$ around it gives us access to the asymptotic form of its coefficients. In particular, if $A(z)$ is analytic in some indented disc neighbourhood of ρ, then one can use the following [12, Corol. VI.1, p. 392]:

Theorem 1. *The coefficients of the series expansion of the complex function* $f(z) \underset{z \to \rho}{\sim} \lambda \left(1 - \frac{z}{\rho}\right)^{\nu}$, *where* $\nu \in \mathbb{C} \setminus \mathbb{N}_0$, $\lambda \in \mathbb{C}$, *have the asymptotic approximation* $[z^n]f(z) = \frac{\lambda}{\Gamma(-\nu)}\, n^{-\nu-1}\rho^{-n} + o\left(n^{-\nu-1}\rho^{-n}\right)$. *Here* Γ *is the Euler's gamma function and the notation* $f(z) \underset{z \to z_0}{\sim} g(z)$ *means that* $\lim_{z \to z_0} \frac{f(z)}{g(z)} = 1$.

To use this, one needs to have a way to obtain ρ, ν and λ. Here we only give a high level description of how this can be done, referring the reader to Broda et al. [5,4]. First, from an unambiguous generating grammar, one obtains a set of polynomial equations involving the generating functions for the objects corresponding to the variables of the grammar, in particular the one whose coefficients we want to asymptotically estimate. Then, either using Gröbner basis or by other means [5], one gets an algebraic equation for that generating function $w = w(z)$, i.e., an equation of the form $G(z, w) = 0$, where $G(z, w) \in \mathbb{Z}[z, w]$ of which $w(z)$ is a root. Analysing the form of the curve G, and using its partial derivatives, one can find an irreducible polynomial for the singularity ρ, and, when $\lim_{z\to\rho} w(z) = a \in \mathbb{R}^+$, an irreducible polynomial for a; when $\lim_{z\to\rho} w(z) = +\infty$, the irreducible polynomial for ρ is a factor of the leading coefficient of $G(z, w)$ when seen as a polynomial in w [15, Th. 12.2.1]. After making the change of variable $s = 1 - z/\rho$, one knows that $w = w(s)$ has a Puiseux series expansion at the singularity $s = 0$, i.e., there exists a slit neighbourhood of that point in which $w(s)$ has a representation as a power series with fractional powers [15, Chap. 12],

Using the irreducible polynomial for ρ, and the one for a in the first case, while in the second case one changes variables in order to replace $+\infty$ with 0, one decides which partial derivatives of G are nonzero, and uses that information to draw a Newton polygon that yields the values of ν and λ. Then, Theorem 1 yields:

Theorem 2. *With the notations and in the conditions above described, one has*

$$[z^n]w(z) \underset{n \to \infty}{\sim} \begin{cases} \dfrac{-b_G}{\Gamma(-\nu)}\rho^{-n}n^{-\nu-1}, & \text{if } \lim_{z\to\rho} w(z) \in \mathbb{R}^+, \quad (6) \\[2ex] \dfrac{1}{c_G\,\Gamma(\nu)}\rho^{-n}n^{\nu-1}, & \text{if } \lim_{z\to\rho} w(z) = +\infty, \quad (7) \end{cases}$$

where ρ *and* ν *are as above, setting* $b_G = -\lambda$ *and* $c_G = \lambda^{-1}$.

Let the generating functions for β_ε and $\beta_{\overline{\varepsilon}}$ regular expressions be, respectively, $B_k = B_k(z) = \sum_{\beta_\varepsilon} z^{\|\beta_\varepsilon\|} = \sum_n b_n z^n$ and $\overline{B}_k = \overline{B}_k(z) = \sum_{\beta_{\overline{\varepsilon}}} z^{\|\beta_{\overline{\varepsilon}}\|} = \sum_n \overline{b}_n z^n$, where b_n and $\overline{b}_n$ are the corresponding numbers of expressions of size n. From (5), one gets $B_k = 2zB_k^2 + 2zB_k\overline{B}_k + 2z\overline{B}_k$ and $\overline{B}_k = kz + 2zB_k\overline{B}_k + 2z\overline{B}_k^2$. Using Theorem 2, Broda et al. [4] obtained that $[z^n]B_k(z) \underset{n \to \infty}{\sim} \frac{b_{B_k}}{2\sqrt{\pi}}\eta_k^{-n}n^{-\frac{3}{2}}$ and $[z^n]\overline{B}_k(z) \underset{n \to \infty}{\sim} \frac{b_{\overline{B}_k}}{2\sqrt{\pi}}\eta_k^{-n}n^{-\frac{3}{2}}$, where η_k is the unique dominant singularity

of $B_k(z)$, which happens to be the same for $\overline{B}_k(z)$. It was also shown that $\eta_k \underset{k\to\infty}{\sim} \frac{1}{\sqrt{8k}}$, $b_{B_k} \underset{k\to\infty}{\sim} \sqrt{8}$ and $b_{\overline{B}_k} \underset{k\to\infty}{\sim} \sqrt{k}$, which yields the asymptotic behaviour of the size of β_ε and $\beta_{\overline{\varepsilon}}$.

3.2 Average Number of New Concatenations in Partial Derivatives

In this section we consider the quantities $|\partial^+(\beta)|$ and $|\partial^+(\beta)|_\odot = \sum_{\alpha\in\partial^+(\beta)} |\alpha|_\odot$ which is the number of new concatenations when computing $\partial^+(\beta)$, and we estimate the average value of $|\partial^+(\beta)|_\odot$. Let $\ell(\beta)$ be the function $|\partial^+(\beta)|$ and $h(\beta)$ be the function of $|\partial^+(\beta)|_\odot$, assuming that all computed partial derivatives are distinct. Thus, ℓ and h are upper bounds for those quantities in the general case. Using the definition of ∂^+ for $\beta \in \mathcal{S}_k$, we have that those cost functions (of the expressions) satisfy

$$\begin{aligned}
h(\varepsilon) = h(\sigma) &= 0, & \ell(\varepsilon) = 0,\ \ell(\sigma) &= 1,\\
h(\beta+\beta') &= h(\beta)+h(\beta'), & \ell(\beta+\beta') &= \ell(\beta)+\ell(\beta'),\\
h(\beta\beta') &= h(\beta)+\ell(\beta)+h(\beta'), & \ell(\beta\beta') &= \ell(\beta)+\ell(\beta'),\\
h(\beta^\star_{\overline{\varepsilon}}) &= h(\beta_{\overline{\varepsilon}})+\ell(\beta_{\overline{\varepsilon}}), & \ell(\beta^\star_{\overline{\varepsilon}}) &= \ell(\beta_{\overline{\varepsilon}}),\\
h(\beta^?_{\overline{\varepsilon}}) &= h(\beta_{\overline{\varepsilon}}), & \ell(\beta^?_{\overline{\varepsilon}}) &= \ell(\beta_{\overline{\varepsilon}}).
\end{aligned}$$

In the computation of $h(\beta\beta')$, the summand $\ell(\beta)$ accounts for the number of partial derivatives of β. Similarly for $h(\beta^\star_{\overline{\varepsilon}})$. For the special case of $\beta_{\overline{\varepsilon}}$ expressions, $h(\sigma)=0$, $h(\beta_{\overline{\varepsilon}}+\beta'_{\overline{\varepsilon}}) = h(\beta_{\overline{\varepsilon}})+h(\beta'_{\overline{\varepsilon}})$, $h(\beta_{\overline{\varepsilon}}\beta_\varepsilon) = h(\beta_{\overline{\varepsilon}})+\ell(\beta_{\overline{\varepsilon}})+h(\beta_\varepsilon)$, $h(\beta\beta_{\overline{\varepsilon}}) = h(\beta)+\ell(\beta)+h(\beta_{\overline{\varepsilon}})$ and $\ell(\sigma)=1$, $\ell(\beta_{\overline{\varepsilon}}+\beta'_{\overline{\varepsilon}}) = \ell(\beta_{\overline{\varepsilon}})+\ell(\beta'_{\overline{\varepsilon}})$, $\ell(\beta_{\overline{\varepsilon}}\beta_\varepsilon) = \ell(\beta_{\overline{\varepsilon}})+\ell(\beta_\varepsilon)$, $\ell(\beta\beta_{\overline{\varepsilon}}) = \ell(\beta)+\ell(\beta_{\overline{\varepsilon}})$. Let $H_k = H_k(z)$ and $\overline{H}_k = \overline{H}_k(z)$ be the cost generating function for the measure h associated with the expressions β and $\beta_{\overline{\varepsilon}}$, respectively. Analogously, let $L_k = L_k(z)$ and $\overline{L}_k = \overline{L}_k(z)$ be the corresponding ones for ℓ. These coincide with the cost generating functions for the alphabetic size, and that was calculated in Broda et al. [4]. One has, where $T_k = B_k + \overline{B}_k$,

$$\begin{aligned}
H_k &= 4zH_kT_k + zL_kT_k + 2z\overline{H}_k + z\overline{L}_k,\\
\overline{H}_k &= 2zH_k\overline{B}_k + 2z\overline{H}_kT_k + z\overline{L}_kB_k + zL_k\overline{B}_k.\\
B_k &= 2zB_k^2 + 2zB_k\overline{B}_k + 2\overline{B}_kz,\\
\overline{B}_k &= kz + 2zB_k\overline{B}_k + 2z\overline{B}_k^2,\\
L_k - \overline{L}_k &= 4zB_k(L_k-\overline{L}_k) + 2zB_k\overline{L}_k + 2z\overline{B}_k(L_k-\overline{L}_k) + 2z\overline{L}_k,\\
\overline{L}_k &= kz + 2zB_k\overline{L}_k + 2z\overline{B}_k(L_k-\overline{L}_k) + 4z\overline{B}_k\overline{L}_k.
\end{aligned}$$

Using Gröbner basis for the equations of B_k and $\overline{B}_k$, and with the help of a symbolic manipulator, one can find a polynomial in $\mathbb{Q}(z,w)$ for which $w = H_k(z)$ is a zero, namely $16z^2\ell_k(z)^2\,w^3 + 4k\,\ell_k(z)p_4(z)\,w^2 + p_8(z)\,w + kz^2p_6(z)$, where the dominant singularity of H_k is only root in $]0,1[$ of

$$\ell_k(z) = z^3 + \frac{9}{2k+27}\,z^2 - \frac{1}{4(2k+27)}\,z - \frac{1}{k(2k+27)},$$

and p_i denotes a polynomial of degree i. Using the method described in [5], one sees that this falls in the case (7) of Theorem 2, and computing the respective constants, ρ, ν, c, one gets the following value for the asymptotic behaviour of the coefficients of $H_k(z)$:

$$[z^n]H_k(z) \underset{n \to \infty}{\sim} \frac{1}{c_{H_k}} \eta_k^{-n}, \tag{8}$$

where c_{H_k} is a function of k with an expression too cumbersome to write here, but that satisfies $c_{H_k} \underset{k \to \infty}{\sim} \frac{16\sqrt{2}}{\sqrt{k}}$. From this one now gets

Theorem 3. *The average value of the upper bound h considered above, for the number of new concatenations in all the partial derivatives of a regular expression in $\mathcal{S}_k$ is given by*

$$\frac{[z^n]H_k(z)}{[z^n](B_k(z) + \overline{B}_k(z))} \underset{n \to \infty}{\sim} \frac{2\sqrt{\pi}}{c_{G_k}(b_{B_k} + b_{\overline{B}_k})} n^{\frac{3}{2}} \underset{k \to \infty}{\sim} \sqrt{\frac{\pi}{128}}\, n^{\frac{3}{2}}. \tag{9}$$

And also, knowing that $[z^n]L_k(z) \sim \frac{\sqrt{k}}{4\sqrt{\pi}} \eta_k^{-n} n^{-1/2}$ (see [4]), one obtains

Theorem 4. *The average value of the upper bound h considered above, for the number of new concatenations per partial derivative of a regular expression in $\mathcal{S}_k$ is given by*

$$\frac{[z^n]H_k(z)}{[z^n]L_k(z)} \underset{n,\, k \to \infty}{\sim} \sqrt{\frac{\pi}{32}}\, n^{\frac{1}{2}}. \tag{10}$$

Note that in the worst case the number of new concatenations is $\Theta(\|\beta\|^2)$, as illustrated by the following example. Let $\beta_n = a_1 a_2 \cdots a_n$, for $n \geq 1$, with $\|\beta_n\| = 2n - 1$. We have $\partial^+(\beta_n) = \{\, a_i \cdots a_n \mid 2 \leq i \leq n \,\} \cup \{\varepsilon\}$ and the number of new concatenations is $|\partial^+(\beta_n)|_\odot = \sum_{i=2}^{n}(n-i) = (n^2 - 3n + 2)/2$.

4 DAG Representation and Partial Derivatives

Consider the (binary) tree representation of $\beta \in \mathcal{S}_k$. Each node v of the tree is labelled with an operator denoted by $\mathsf{lab}(v)$. Let β_v be the subexpression rooted at v. In what follows, we identify a node with its rooted subexpression. A node v is an *op*-node if $\mathsf{lab}(v) = op$, $op \in \{+, \star, ?, \odot\} \cup \Sigma$. Each node, except the root, has exactly one parent node and can have zero, one or two children. For $\beta_1 = (ab)^\star a + (ab)^\star$, its tree is depicted in Figure 1(a). One can see that there are several identical subtrees (subexpressions). The identification of all common subexpressions of β leads to a directed acyclic graph (DAG) representation of β. Let s be the number of distinct subexpressions of β. Each node of the DAG corresponds exactly to a distinct subexpression of β and can be identified by an index $i \in \{1, \ldots, s\}$. The node with index 0 corresponds to ε.

In Algorithm 1, we present an algorithm to build the DAG for a regular expression, as well as, to compute its $\mathcal{A}_{\mathsf{PD}}$. In this section we focus on the construction of the DAG without constructing partial derivatives and thus $\mathcal{A}_{\mathsf{PD}}$.

The function GETI constructs the DAG for an expression and for each type of operator calls a function that builds a node, if it does not exist, and returns its index. This function is inspired by Flajolet et al. [13], which is based on the more general algorithms presented in Downey et al. [11].

Let IND be a structure that associates each index i with a unique node (subexpression). Let last be the variable that counts the number of nodes already in the DAG. The function NODE(i, op, C) creates a node with index i, label op and children C, where C is a list of zero to two DAG indexes (which is omitted if $|C| = 0$). To construct the DAG one needs to determine if for a node i, the subtree β_i is already there. That can be decided by analysing the parents of node i and their labels, using the following functions. Let $\mathsf{star}(i)$ be the parent of i, in the case it is a $\star$-node, or Null, otherwise. Similarly define $\mathsf{option}(i)$. To uniquely identify a $\odot$-node one needs to know its left and right children. If $\mathsf{dot}(i, j)$ is not Null then a $\odot$-node with left child i and right child j exists. The same occurs for $+$-node and $\mathsf{plus}(i, j)$. Finally, $\mathsf{leaves}(\sigma)$ is not Null, if the node i is a σ-node.

Depending on the data structures used, the construction of the DAG can be achieved, in the worst-case, in quadratic time or, respectively, in linear time [11]. Using hash tables the running time is $O(n)$, on average [13, Prop. 1]. Using the result of Flajolet et al. [13], mentioned in Section 1, the expected size of the DAG of β is $O\left(\|\beta\| / \sqrt{\log \|\beta\|}\right)$. A DAG for β_1 is depicted in Figure 1(b). When building the DAG, one can also compute for each node i, the functions $\varepsilon(i)$ and $\varphi(i)$. In Algorithm 1, the function ewp is a Boolean function such that $\mathsf{ewp}(i) = \mathsf{True}$ if $\varepsilon(i) = \varepsilon$, and False otherwise. The computation of $\varphi(i)$ can lead to the creation of new $\odot$-nodes. For these nodes the computation of φ is delayed until the nodes of all subexpressions of a given expression are computed. The final DAG is shown in Figure 1(c). Note that the indexes (numbers) given to nodes are different if one computes simultaneously the DAG for β_1 with or without the partial derivatives. In the latter case, the partial derivatives are computed after the DAG for β_1 is constructed (and that is what is assumed in Figure 1). After constructing the DAG with all partial derivatives (LF), the $\mathcal{A}_{\mathsf{PD}}(\beta)$ can be easily obtained. The automaton $\mathcal{A}_{\mathsf{PD}}(\beta_1)$ is shown in Figure 1(d). In the next section, we detail the algorithm to build the $\mathcal{A}_{\mathsf{PD}}(\beta)$, as well as the overall complexity analysis.

5 Algorithm PDDAG

Given a regular expression β, we present an algorithm to compute the partial derivative automaton $\mathcal{A}_{\mathsf{PD}}(\beta)$. Although the algorithm also applies to standard regular expressions, here, we assume expressions in ssnf. In Algorithm 1, the function PDDAG implements the main procedure that constructs a DAG not only representing the expression but also all its partial derivatives. For each node, the corresponding linear form is computed and for $\star$-nodes or $\odot$-nodes special attention is needed. In both cases, the function CONCLF can add new $\odot$-nodes to the DAG. For those nodes the computation of their linear forms is delayed, as they can depend on the linear forms of the nodes that gave them origin. Hence, the

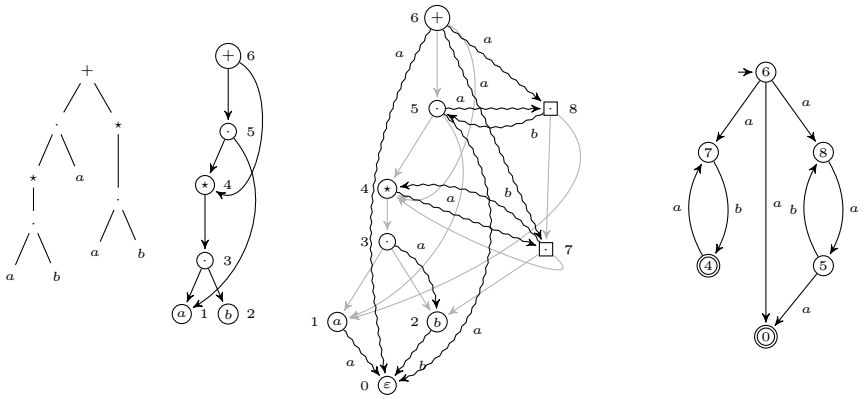


Fig. 1. For $\beta_1 = (ab)^\star a + (ab)^\star$ we show (a) a tree representation where the root corresponds to β_1, (b) the (minimal) DAG identifying common subexpressions of α_1, (c) DAG with partial derivative nodes (LF), and (d) the resulting $\mathcal{A}_{\mathsf{PD}}$. In (c), new nodes created during the computation of $\partial^+(\beta_1)$ are presented by a square. The values of $\mathsf{ewp}(i)$ are omitted. A (dark) zigzag directed edge between nodes i and j labelled by σ means that $j \in \partial_\sigma(i)$ (those accessible from the root correspond to the transitions in $\mathcal{A}_{\mathsf{PD}}$).

function ConcI is called with the *delay* parameter as True (by default is False). When the computation of the linear form of the creator node is finished, the linear forms of the delayed nodes can safely be computed. Function doDelayed computes the linear forms of the new $\odot$-nodes until no more delayed nodes exist. When all nodes of all partial derivatives have been created, the $\mathcal{A}_{\mathsf{PD}}(\beta)$ can be constructed using definition (4), starting with the root node s corresponding to β. The function makeNFA implements this construction (by space constraints, we omit its description). In the following, we discuss the complexity of the algorithm. We show that, *on average* for $\beta \in \mathcal{S}_k$, the algorithm PDDAG(β) works in time $O\big(\|\beta\|^{3/2}\sqrt[4]{\log\|\beta\|}\big)$. We make the ordinary assumption in formal language algorithms that an integer occupies space $O(1)$ and each basic integer arithmetic operation takes time $O(1)$ [10].

Lemma 1. *Given the* DAG *of β with partial derivatives, for every node v and $\sigma \in \Sigma$, $|\partial_\sigma(\beta_v)| \le |\beta|_\Sigma$.*

Theorem 5. *Algorithm* PDDAG(β) *can be implemented such that*

(i) in the average case, it uses time $O\big(\|\beta\|^{3/2}\sqrt[4]{\log\|\beta\|}\big)$;
(ii) in the worst case, it uses time $O\left(|\Sigma_\beta||\beta|_\Sigma^2\|\beta\|\log\|\beta\|\right)$;
(iii) in the average case, it uses space $O\left(\|\beta\|^{3/2}/\left(\log\|\beta\|\right)^{3/4}\right)$;
(iv) in the worst case, it uses space $O\big(|\Sigma_\beta||\beta|_\Sigma^2\|\beta\|\big)$.

Proof. As seen in Section 4, making the DAG for β can take time $\Theta(\|\beta\|)$ and the number s of initial DAG nodes is $O(\|\beta\|/\sqrt{\log\|\beta\|})$ in the average case, and $\Theta(\|\beta\|)$ in the worst case. All finite sets in the algorithm are implemented using

Algorithm 1 Partial Derivative Automaton with DAG

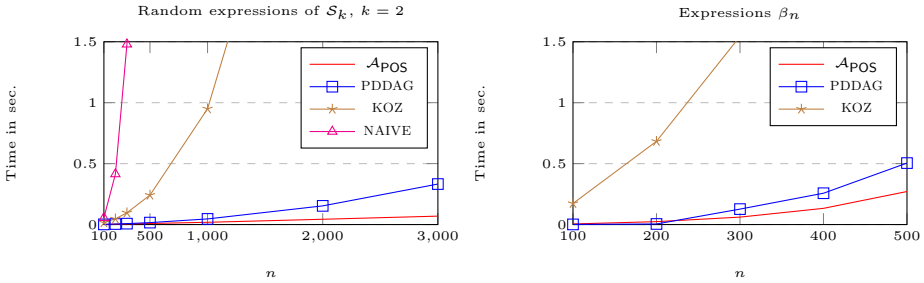
```
1: function PDDAG(α)
2:     IND[0] ←NODE(0, ε)
3:     last ← 1;Delayed ← ∅
4:     s ← GETI(α)
5:     MAKENFA()
6: function GETI(α)
7:     if α = σ then
8:         return ATOMI(σ)
9:     else if α = α_1^? then
10:        return OPTIONI(GETI(α_1))
11:    else if α = α_1^⋆ then
12:        return STARI(GETI(α_1))
13:    else if α = α_1 + α_2 then
14:        return PLUSI(GETI(α_1),GETI(α_2))
15:    else if α = α_1 ⊙ α_2 then
16:        return CONCI(GETI(α_1),GETI(α_2))
17:    else return 0
18: function ATOMI(σ)
19:    if leaves(σ) is Null then
20:        i ← last; last ← last +1
21:        IND[i] ← NODE(i, σ)
22:        ewp(i) ← False
23:        φ(i) ← (σ, 0)
24:    else
25:        i ← leaves(σ)
26:    return i
27: function STARI(i)
28:    if star(i) is Null then
29:        j ← last; last ← last +1
30:        IND[j] ← NODE(j, ⋆, i)
31:        ewp(j) ← True
32:        φ(j) ← CONCLF(i, j)
33:        DODELAYED()
34:    else
35:        j ← star(i)
36:    return j
37: function OPTIONI(i)
38:    if option(i) is Null then
39:        j ← last; last ← last +1
40:        IND[j] ← NODE(j, ?, i)
41:        ewp(j) ← True
42:        φ(j) ← φ(i)
43:    else
44:        j ← option(i)
45:    return i
46: function PLUSI(i, j)
47:    if plus(i, j) is Null then
48:        ℓ ← last; last ← last +1
49:        IND[ℓ] ← NODE(ℓ, +, i, j)
50:        ewp(ℓ) ← ewp(i) ∨ ewp(j)
51:        φ(ℓ) ← φ(i) ∪ φ(j)
52:    else
53:        ℓ ← plus(i, j)
54:    return ℓ
55: function CONCI(i, j, delay = False)
56:    if i = 0 then return j
57:    if j = 0 then return i
58:    if dot(i, j) is Null then
59:        ℓ ← last; last ← last +1
60:        IND[ℓ] ← NODE(ℓ, ⊙, i, j)
61:        ewp(ℓ) ← ewp(i) ∧ ewp(j)
62:        if delay = True then
63:            add ℓ to Delayed
64:        else
65:            φ(ℓ) ← CONCLF(i, j)
66:            if ewp(i) then
67:                φ(ℓ) ← φ(ℓ) ∪ φ(j)
68:            DODELAYED()
69:    else
70:        ℓ ← dot(i, j)
71:    return ℓ
72: function CONCLF(i, j)
73:    F ← ∅
74:    for all (σ, ℓ) ∈ φ(i) do
75:        add (σ,CONCI(ℓ, j, True)) to F
76:    return F
77: function DODELAYED( )
78:    while Delayed ≠ ∅ do
79:        i ← Delayed.pop()
80:        φ(i) ← CONCLF(left(i), right(i))
81:        if ewp(left(i)) then
82:            φ(i) ← φ(i) ∪ φ(right(i))
```

AVL-trees. The structure IND contains pairs (i, p) such that i is an index and p is the DAG node with index i. The structure DOT contains triples (i, j, ℓ) of integers such that ℓ is the index for a $\odot$-node with $i = \mathsf{left}(\ell)$ and $j = \mathsf{right}(\ell)$. The search is based on the pairs (i, j) and works as in single integer comparisons. Function $\mathsf{dot}(i, j)$ returns ℓ when the triple (i, j, ℓ) is in DOT, and Null otherwise. When $\textsc{Node}(\ell, \odot, i, j)$ is created the triple (i, j, ℓ) is added to DOT. The structure PLUS is analogous to DOT for $+$-nodes and used by the function plus. The structure Δ contains pairs (ℓ, F) such that $F = \varphi(\ell)$. Specifically, F is an AVL-tree containing pairs (σ, S) such that $S = \partial_\sigma(\ell)$. To access the set $\partial_\sigma(\ell)$, Δ is searched on ℓ to get the pair (ℓ, F), and then F is searched on σ to get the set $S = \partial_\sigma(\ell)$. Let t be the number of new nodes created when computing the linear forms. Each such node is a partial derivative of β or of a subexpression

of β. Let $\{\beta_\ell\}_{\ell=1}^{s}$ be the set of subexpressions of β, including $\beta = \beta_s$. Then $t \leq |\bigcup_{\ell=1}^{s} \partial^+(\ell)| \leq \sum_{\ell=1}^{s} |\beta_\ell|_\Sigma \leq s|\beta|_\Sigma$. By Theorem 3 we have that, on average, $t = O(s^{3/2})$. In the worst-case, $t = \Theta(s|\beta|_\Sigma)$. For each node ℓ, the set $\varphi(\ell)$ is computed and stored as $\Delta[\ell]$. Each $|\Delta[\ell]|$ is $O(|\Sigma_\beta||\beta|_\Sigma)$ in the worst case (by Lemma 1), and $O(1)$ in the average case [19,17]. Based on the above observations, the algorithm's space complexity is $O\big(|\Sigma_\beta||\beta|_\Sigma s|\beta|_\Sigma\big) = O\big(|\Sigma_\beta||\beta|_\Sigma^2\|\beta\|\big)$ in the worst case, and $O(s^{3/2}) = O\left(\|\beta\|^{3/2} / (\log\|\beta\|)^{3/4}\right)$ in the average case. We turn now to the time complexity. The task to compute the set $\Delta[\ell]$, for each node ℓ, depends on $\mathsf{lab}(\ell)$ and the children of ℓ. We only examine the time for $+$-nodes and $\odot$-nodes, as the time for the others is not more significant. If ℓ is a $+$-node, then $\Delta[\ell] = \Delta[\mathsf{left}(\ell)] \cup \Delta[\mathsf{right}(\ell)]$. Thus, the cost for each $+$-node ℓ is $O\big(|\Delta[\ell]| \log |\Delta[\ell]|\big)$. If ℓ is a $\odot$-node with children i, j then the time of ConcLF(i, j) is $O\big(|\Delta[i]| \log t + |\Delta[\ell]| \log |\Delta[\ell]|\big)$, where $\log t$ accounts for the cost of accessing DOT and $|\Delta[\ell]| \log |\Delta[\ell]|$ for making the set F (line 75). As $|\Sigma_\beta|, |\beta|_\Sigma \leq \|\beta\|$, we have that $\log t = O(\log\|\beta\|)$.

Thus, the algorithm's time complexity is $O\left(|\Sigma_\beta||\beta|_\Sigma^2\|\beta\| \log\|\beta\|\right)$ in the worst case, and $O\left(\|\beta\|^{3/2} (\log\|\beta\|)^{1/4}\right)$ in the average case. □

Fig. 2. Running times (per expression) of the simulation of expressions in ssnf by NFA using different algorithms: position ($\mathcal{A}_{\mathsf{POS}}$), partial derivatives ($\mathcal{A}_{\mathsf{PD}}$) using, respectively, PDDAG, KOZ, and a naive implementation, NAIVE.



6 Empirical Results and Conclusions

We implemented the algorithm PDDAG in Python within FAdo (`https://pypi.org/project/FAdo/`). Instead of AVL-trees we used hash tables, as those are Python's natural data structures. In the experiments we uniformly random generated expressions $\beta \in \mathcal{S}_k$, in prefix notation. For each expression size $n \in \{100, 200, 300, 500, 1000, 2000, 3000, 4000\}$, and alphabet size $k \in \{2, 5, 10, 50\}$, samples of 10000 expressions were generated. This is sufficient to ensure a 95% confidence level within a 1% error margin [9, p. 75]. In FAdo there is a naive implementation of $\mathcal{A}_{\mathsf{PD}}$ that recursively computes the linear forms with some memoization (NAIVE), as well as implementations of the position automaton $\mathcal{A}_{\mathsf{POS}}$. The algorithm for $\mathcal{A}_{\mathsf{PD}}$ by Khorsi et a. (KOZ) [16] was implemented using FAdo methods for acyclic finite automata. The tests were performed in Python 2.7 with a 2.5GHz Quad-Core i7 CPU and 16GB memory. In Figure 2, on the left, we present the average running times of the algorithms per expression of

size n, and $k = 2$. On the right, we present the running times for expressions $\beta_n = a_1^? \cdots a_n^?$, over a growing alphabet $\Sigma = \{a_1, \ldots, a_n\}$, that attain the worst-case size of $\mathcal{A}_{\mathsf{PD}}$. Both results suggest that the algorithm PDDAG has a good practical performance.

Future research is the adaptation of the tools used here to the word membership problem without computing the automaton.

References

1. Antimirov, V.M.: Partial derivatives of regular expressions and finite automaton constructions. Theoret. Comput. Sci. **155**(2), 291–319 (1996)
2. Broda, S., Machiavelo, A., Moreira, N., Reis, R.: On the average state complexity of partial derivative automata: an analytic combinatorics approach. Int. J. Found. Comput. Sci. **22**(7), 1593–1606 (2011). https://doi.org/10.1142/S0129054111008908
3. Broda, S., Machiavelo, A., Moreira, N., Reis, R.: On the average size of Glushkov and partial derivative automata. Int. J. Found. Comput. Sci. **23**(5), 969–984 (2012). https://doi.org/10.1142/S0129054112400400
4. Broda, S., Machiavelo, A., Moreira, N., Reis, R.: On average behaviour of regular expressions in strong star normal form. Int. J. Found. Comput. Sci. **30**(6-7), 899–920 (2019). https://doi.org/10.1142/S0129054119400227
5. Broda, S., Machiavelo, A., Moreira, N., Reis, R.: Analytic combinatorics and descriptional complexity of regular languages on average. ACM SIGACT News **51**(1), 38–56 (March 2020). https://doi.org/10.1145/3388392.3388400
6. Brüggemann-Klein, A.: Regular expressions into finite automata. Theoret. Comput. Sci. **48**, 197–213 (1993)
7. Champarnaud, J.M., Ziadi, D.: Canonical derivatives, partial derivatives and finite automaton constructions. Theoret. Comput. Sci. **289**, 137–163 (2002)
8. Champarnaud, J., Ziadi, D.: From c-continuations to new quadratic algorithms for automaton synthesis. Intern. Journ. of Alg. and Comp. **11**(6), 707–736 (2001)
9. Cochran, W.G.: Sampling Techniques. John Wiley and Sons, third edn. (1977)
10. Crochemore, M., Hancart, C., Lecroq, T.: Algorithms on strings. CUP (2007)
11. Downey, P.J., Sethi, R., Tarjan, R.E.: Variations on the common subexpression problem. JACM **27**(4), 758–771 (1980). https://doi.org/10.1145/322217.322228
12. Flajolet, P., R.Sedgewick: Analytic Combinatorics. CUP (2008)
13. Flajolet, P., Sipala, P., Steyaert, J.: Analytic variations on the common subexpression problem. In: Paterson, M. (ed.) Proc. 17th ICALP 90. LNCS, vol. 443, pp. 220–234. Springer (1990). https://doi.org/10.1007/BFb0032034
14. Gruber, H., Gulan, S.: Simplifying regular expressions: A quantitative perspective. Tech. rep., IFIG Research Report (2009)
15. Hille, E.: Analytic Function Theory, vol. 2. Blaisdell Publishing Company (1962)
16. Khorsi, A., Ouardi, F., Ziadi, D.: Fast equation automaton computation. J. Discrete Algorithms **6**(3), 433–448 (2008). https://doi.org/10.1016/j.jda.2007.10.003
17. Konstantinidis, S., Machiavelo, A., Moreira, N., Reis, R.: On the size of partial derivatives and the word membership problem. Acta Informatica **58**, 357–375 (2021). https://doi.org/10.1007/s00236-021-00399-6
18. Mirkin, B.G.: An algorithm for constructing a base in a language of regular expressions. Eng. Cybernetics **5**, 51—57 (1966)
19. Nicaud, C.: On the average size of Glushkov's automata. In: Dediu, A., Ionescu, A.M., Vide, C.M. (eds.) Proc. 3rd LATA. LNCS, vol. 5457, pp. 626–637. Springer (2009)