

Computing the follow automaton of an expression

J.-M. Champarnaud¹, F. Nicart^{1,2}, D. Ziadi¹

¹ L.I.F.A.R, Université de Rouen,
{jmc,nicart,ziadi}@dir.univ-rouen.fr

² XRCE, Xerox Research Center Europe, 38240 Meylan
florent.nicart@xrce.xerox.com

Abstract. Small nondeterministic recognizers are very useful in practical applications based on regular expression searching. The follow automaton, recently introduced by Ilie and Yu, is such a small recognizer, since it is a quotient of the position automaton. The aim of this paper is to present an efficient computation of this quotient, based on specific properties of the *ZPC*-structure of the expression. The motivation is twofold. Since this structure is already a basic tool for computing the position automaton, Antimirov's automaton and Hromkovic's automaton, the design of an algorithm for computing the follow automaton via this structure makes it easier to compare all these small recognizers. Secondly such an algorithm provides a straightforward alternative to the rather sophisticated handling of ε -transitions used in the original algorithm.

1 Introduction

Regular expressions are a very convenient formalism used in a wide range of applications like regular expression searching, text processing or natural language processing. Since they are fully equivalent, finite automata are their natural implementation. Simple and very efficient, regular expressions and their finite automata are integrated into many computer science applications such as `grep`, `perl`, `flex`, etc. Thus, computing small finite state automata from regular expressions is a challenging problem.

The position automaton of a regular expression [8, 14] is of particular interest. Let $|E|$ be the size of the expression E , i.e. the number of nodes of its syntax tree and let $\|E\|$ be the number of occurrences of symbols in E . The position automaton of E has $\|E\| + 1$ states. It can be built in $O(|E|^3)$ time by a naive algorithm and in $O(\|E\|^2)$ time by optimized implementations based on expression transformation [2], lazy evaluation [7], or implicit structure computation [17, 15]. Moreover, it has been proved that quotients of this automaton can be computed with the same quadratic time complexity. Champarnaud and Ziadi have shown that Antimirov's automaton [1] is such a quotient; they have used the notion of canonical derivative to design a quadratic algorithm to compute it [4]. More recently, Ilie and Yu [12] have introduced the follow automaton of a regular expression E . There exists an equivalence relation over the set of positions of E , called the follow relation and denoted by \equiv_f , such that the follow automaton is the quotient of the position automaton by the relation \equiv_f .

Our aim is to design an efficient computation of this quotient, based on specific properties of the \mathcal{ZPC} -structure of the expression [17, 15]. The motivation is twofold. First this structure is already a basic tool for computing not only the position automaton and Antimirov's automaton, but also Hromkovic's automaton [10, 9] which focuses on the reduction of the number of transitions. An algorithm based on the \mathcal{ZPC} -structure for computing the follow automaton makes it easier to compare all these small recognizers. Secondly such an algorithm provides a straightforward alternative to the rather sophisticated handling of ε -transitions used in the original algorithm.

In our approach, the expression is first normalized; the normalization that we consider includes size reduction (elimination of redundant ε 's, \emptyset 's and $*$'s) as well as Star Normal Form transformation [2] and its time complexity is linear w.r.t. the size $|E|$ of the expression. Then its \mathcal{ZPC} -structure is built in linear time and space w.r.t. $|E|$. We prove that, as far as the expression is a normalized one, the set of follow links of the \mathcal{ZPC} -structure has specific properties. It allows us to compute the follow relation via a simple marking of the nodes of the \mathcal{ZPC} -structure, hence with a linear time complexity w.r.t. $|E|$. Finally we present a new algorithm to compute the follow automaton of a regular expression, with an $O(c \times |E|)$ time complexity, where c is the index of the relation \equiv_f . It turns out that this algorithm is about three times faster than the original one.

Next section gathers definitions concerning expressions and automata and a short description of classical constructions (position automaton, \mathcal{ZPC} -structure and follow automaton). Section 3 presents the specific properties of the \mathcal{ZPC} -structure of a normalized expression and the new algorithm to build the follow automaton. Experimental tests are reported in Section 4.

2 Preliminaries

In this section we first recall some basic definitions and properties about regular expressions and finite automata. For more details, we refer to [11] and [16].

2.1 Regular expressions and finite automata

Let Σ be a non-empty finite set of symbols, called an *alphabet*. The set of all the words over Σ is denoted by Σ^* . The *empty word* is denoted by ε . A *language* over Σ is a subset of Σ^* . *Regular expressions* over an alphabet Σ are inductively defined as usually. We will write $Sym(E) = \{+\}$ (resp. $\{., *\}$) if $E = F + G$ (resp. $E = FG$, $E = F^*$). We call *alphabetic width* of E , denoted by $\|E\|$, the number of occurrences of symbols of Σ in E whereas we call *size* of E , denoted by $|E|$, the number of nodes of the syntax tree of E . For all integer j , $1 \leq j \leq \|E\|$, if x is the j^{th} alphabetic symbol in E , the pair (x, j) (written x_j) is called a *position* of E . The set of all the positions of E is denoted by $Pos(E)$. An expression E is said to be *linear* over Σ if and only if every symbol of Σ occurs at most one time in E . The *linearized version* of E is the expression \bar{E} deduced from E by replacing the symbol x in position j by x_j , for all j , $1 \leq j \leq \|E\|$. We denote by h the

mapping from $Pos(E)$ to Σ induced by the linearization of E . An expression E is said to be *nullable* if and only if $\varepsilon \in L(E)$. We set $Null(E) = \{\varepsilon\}$ if $\varepsilon \in L(E)$ and \emptyset otherwise.

In practical applications, it is usual to preprocess the input expression in order to reduce its size and to make its size proportional to its alphabetic width. We will consider the following definition [12]:

Definition 1 *Given a regular expression E , an equivalent reduced expression can be computed in linear time by applying the following rules to every node ν of the syntax tree of E :*

- a) \emptyset -reduction: *if $L(\nu) = \emptyset$, the subtree rooted by ν is replaced by \emptyset . At the end, E contains no \emptyset or equals to \emptyset .*
- b) ε -reduction: *if $L(\nu) = \{\varepsilon\}$, the subtree rooted by ν is replaced by ε . Then, if the parent node is labelled by ‘.’, it is replaced by the other child. If it is labelled by ‘*’, it is replaced by the child. If it is labelled by ‘+’ and if ε belongs to the language of the other child, then the parent is replaced by its child.*
- c) *-reduction: *every vertex labelled with ‘*’ such that parent node is also labelled by ‘*’ is replaced by the child.*

An automaton is a quintuple $\mathcal{A} = (Q, \Sigma, \delta, I, F)$ where Q is a finite set of states, Σ is the alphabet, $\delta \subseteq Q \times \Sigma \times Q$ is the transition mapping, $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final states. An ε -automaton is an automaton with $\delta \subseteq Q \times (\Sigma \cup \varepsilon) \times Q$.

2.2 Classical constructions

The position automaton The position automaton [8, 14] of a regular expression E , denoted by \mathcal{P}_E , is related to specific subsets of $Pos(E)$. If E is linear, the following subsets of Σ are computed: $First(E)$ (resp. $Last(E)$), the set of symbols that match the first (resp. last) symbol of some word in $L(E)$, and, for all x in Σ , $Follow(E, x)$, the set of symbols that follow the symbol x in some word of $L(E)$. The functions $First$, $Last$ and $Follow$ can be inductively computed. The set of states of the position automaton of E is $Pos(E)$ added with a specific position denoted by 0. The following notation will be used: $Pos_0(E) = Pos(E) \cup \{0\}$; the set $Last_0(E)$ is equal to $Last(E)$ if $Null(E) = \emptyset$ and to $Last(E) \cup \{0\}$ otherwise; the set $Follow_0(E, x)$ is equal to $Follow(E, x)$ if $x \in Pos(E)$ and to $First(E)$ if $x = 0$. It is easy to see that the position automaton of a regular expression E is such that $\mathcal{P}_E = (Pos_0(E), \Sigma, \delta, \{0\}, Last_0(E))$, with $\delta(x, a) = \{y \mid y \in Follow_0(E, x) \text{ and } h(y) = a\}, \forall x \in Pos_0(E), \forall a \in \Sigma$.

Remark 1 *We consider the expression $E_0 = \$E$ where $\$ \notin \Sigma$. Then $Pos(E_0) = Pos_0(E)$, $Last(E_0) = Last_0(E)$ and, for all $x \in Pos(E_0)$, $Follow(E_0, x) = Follow_0(E, x)$. Hence an equivalent definition of the position automaton of E : $\mathcal{P}_E = (Pos(E_0), \Sigma, \delta, \{0\}, Last(E_0))$, with $\delta(x, a) = \{y \mid y \in Follow(E_0, x) \text{ and } h(y) = a\}, \forall x \in Pos(E_0), \forall a \in \Sigma$.*

Definition 2 *Two positions x and y of a regular expression E are said to be Last-equivalent in E ($x \sim_E y$) if and only if $x \in Last(E) \Leftrightarrow y \in Last(E)$.*

Definition 3 A regular expression E is said to be in Star Normal Form (E is in SNF) if and only if for every expression F such that F^* is a subexpression of E , the following property holds: $\forall x \in \text{Last}(F), \text{Follow}(F, x) \cap \text{First}(F) = \emptyset$.

It was shown in [2] that any regular expression can be turned into Star Normal Form in linear time.

The \mathcal{ZPC} -structure The \mathcal{ZPC} -structure of a regular expression [17, 15] is a linear space and time representation of the position automaton that is based on two state forests connected by a set of links. Let us briefly recall how to convert a regular expression into its \mathcal{ZPC} -structure (see Figure 1). More details can be found in [3, 6].

1. We perform a depth-first traversal of the syntax tree $T(\overline{E})$ of \overline{E} in order to add specific links from each leaf to its successor and from each node to its leftmost leaf and to its rightmost leaf. These specific links allow us to directly access to the set of positions of each node.
2. We create two copies of the tree $T(\overline{E})$, respectively denoted by $\text{Lasts}(E)$ and $\text{Firsts}(E)$.
3. For each node $A = B \cdot C$ of $\text{Lasts}(E)$, if C is not nullable, we disable the connection to B , and we update the leftmost leaf pointer of A .
4. For each node $A = B \cdot C$ of $\text{Firsts}(E)$, if B is not nullable, we disable the connection to C , and we update the rightmost leaf pointer of A .
5. For every node $A = B \cdot C$, we create a *follow link* from B in $\text{Lasts}(E)$ to C in $\text{Firsts}(E)$. It encodes the set of transitions associated with $\text{Last}(B) \times \text{First}(C)$.
6. For every node $A = B^*$, we create a follow link from B in $\text{Lasts}(E)$ to A in $\text{Firsts}(E)$. It encodes the set of transitions associated with $\text{Last}(B) \times \text{First}(B)$.

It has been shown in [17, 15] that the \mathcal{ZPC} -structure of a regular expression requires $O(|E|)$ space, can be built in $O(|E|)$ time and converted into a position automaton in $O(|E|^2)$ time. A follow link is said to be *redundant* if and only if the set of transitions it encodes is included into the set of transitions encoded by another follow link. Redundant follow links may be eliminated in linear time w.r.t. $|E|$.

The follow automaton The inductive construction of the ε -automaton $\mathcal{A}_f^\varepsilon(E)$ of a regular expression E is defined by Ilie and Yu in [12]. Its computation is in $O(|E|)$ time; see Section 5 for more details. The *follow automaton* $\mathcal{A}_f(E)$ (see Figure 2) is produced by eliminating ε -transitions from $\mathcal{A}_f^\varepsilon(E)$, which can be performed in $O(|E|^2)$ time. Let $\equiv_f \subseteq (\text{Pos}_0(E))^2$ be the equivalence relation defined by

$$x \equiv_f y \Leftrightarrow \text{Follow}_0(E, x) = \text{Follow}_0(E, y) \text{ and } x \in \text{Last}_0(E) \Leftrightarrow y \in \text{Last}_0(E)$$

Then $\mathcal{A}_f(E)$ is a quotient of the position automaton: $\mathcal{A}_f(E) \simeq \mathcal{P}_E / \equiv_f$.

Following Definition 2 and Remark 1, we will rather consider the relation $\equiv_f \subseteq (\text{Pos}(E_0))^2$ such that $x \equiv_f y \Leftrightarrow \text{Follow}(E_0, x) = \text{Follow}(E_0, y) \wedge x \sim_{E_0} y$. Indeed, this definition is more convenient when working on a \mathcal{ZPC} -structure since it allows us to express properties directly on the basic sets of the expression E_0 .

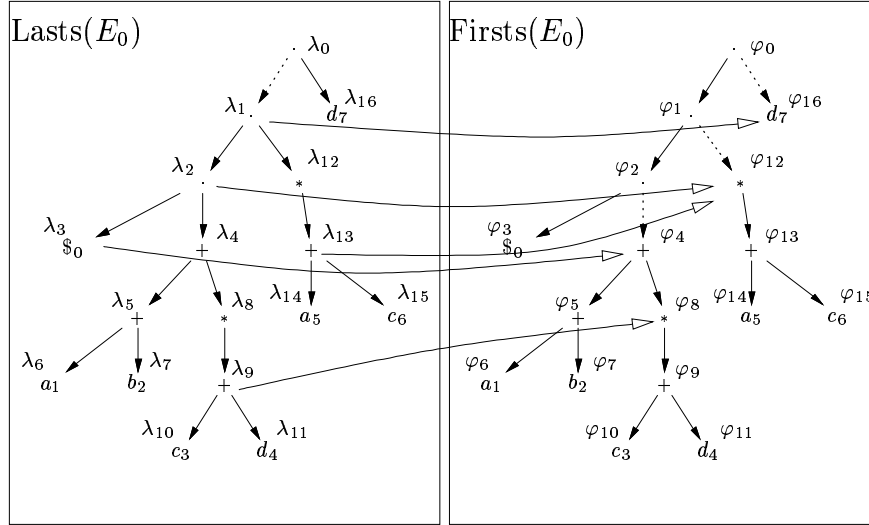


Fig. 1. The \mathcal{ZPC} -structure of $E_0 = \$ \cdot ((a + b) + (c + d)^*) \cdot (a + c)^* \cdot d$.

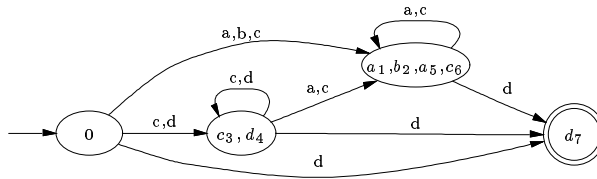


Fig. 2. The follow automaton of $E = ((a + b) + (c + d)^*) \cdot (a + c)^* \cdot d$.

3 From a \mathcal{ZPC} -structure to a follow automaton

Our aim is to provide an efficient algorithm to compute a follow automaton. We first introduce normalized expressions and prove some of their properties. Then we describe an efficient computation of the relation \equiv_f over the \mathcal{ZPC} -structure of a normalized expression. Finally we show how the transition function of the follow automaton can be deduced from the \mathcal{ZPC} -structure.

3.1 Normalized expressions

Definition 4 A regular expression E is said to be normalized if the following conditions hold:

1. The expression E is a reduced one (according to Definition 1).
2. The operation ‘ \cdot ’ is assumed to be left associative when building the syntax tree of the expression.
3. The expression E is in SNF.

Here is a detailed list of properties of normalized expressions. Let H be a subexpression of a normalized expression E . Then the following properties hold:

- a. If $H = F + G$, then $F \neq \emptyset$ and $G \neq \emptyset$.
- b. If $H = F + \varepsilon$, then $Null(F) = \emptyset$.
- c. If $H = F \cdot G$, then $F \neq \emptyset$, $G \neq \emptyset$, $F \neq \varepsilon$ and $G \neq \varepsilon$.
- d. If $H = F \cdot G$, then $Sym(G) \neq \cdot$.
- e. If H^* is a subexpression of E , then for all $x \in Last(H)$, $Follow(H, x) \cap First(H) = \emptyset$.

Properties (a), (b) and (c) come from the fact E is reduced. Property (d) comes from the fact the operation \cdot is left associative. Property (e) is a consequence of the fact E is in SNF. It is straightforward to check that given a regular expression E' of size s' , it is possible to construct an equivalent normalized expression E in $O(s')$ time and space. Therefore we can assume that all regular expressions are normalized. We now state two useful propositions. The first one addresses arbitrary regular expressions; the second one addresses normalized expressions.

Proposition 1 *Let x and y be two Last-equivalent positions of a regular expression E . If E is in SNF, for all H such that H^* is a subexpression of E , it holds: $Follow(F^*, x) = Follow(F^*, y) \Leftrightarrow Follow(F, x) = Follow(F, y)$.*

Proposition 2 *Let E be a normalized expression such that $E = F \cdot G$ and let x and y be two positions of E such that $x \in Last(F)$, $y \in Last(G)$ and $Follow(E, x) = Follow(E, y)$. Then G is such that $G = H^*$.*

3.2 Computation of \equiv_f over the \mathcal{ZPC} -structure

Let E be a regular expression and let us consider its \mathcal{ZPC} -structure. We denote by λ_F (resp. φ_F) the root of the tree associated with the subexpression F in $Lasts(E)$ (resp. $Firsts(E)$). Let x be a position of E . In order to shorten notation, the node associated with x in $Lasts(E)$ is denoted by x too. In the tree that contains x there is a unique path from the root to x . We consider the reverse path $\pi(x) = \lambda_{i_1} \lambda_{i_2} \dots \lambda_{i_p}$, with $\lambda_{i_1} = x$. We will denote $Colast(E) = Pos(E) \setminus Last(E)$.

Definition 5 *Let E be a regular expression and x a position of E . We denote by $\Delta_E(x)$ the set of nodes of $Firsts(E)$ that are head of a follow link whose tail belongs to the path $\pi(x)$. We have:*

$$\Delta_E(x) = \{follow(\lambda) \mid \lambda \in \pi(x) \text{ and } (\lambda, follow(\lambda)) \text{ is a follow link}\}$$

Let x and y be two positions of an arbitrary regular expression. It is easy to check that $\Delta_E(x) = \Delta_E(y) \Rightarrow x \equiv_f y$. We are going to show that, as far as normalized expressions are concerned, the inverse also holds, i.e.: $x \equiv_f y \Rightarrow \Delta_E(x) = \Delta_E(y)$.

Theorem 1 *Let x and y be two positions of a normalized expression E . It holds: $(Follow(E, x) = Follow(E, y) \wedge x \sim_E y) \Leftrightarrow \Delta_E(x) = \Delta_E(y)$.*

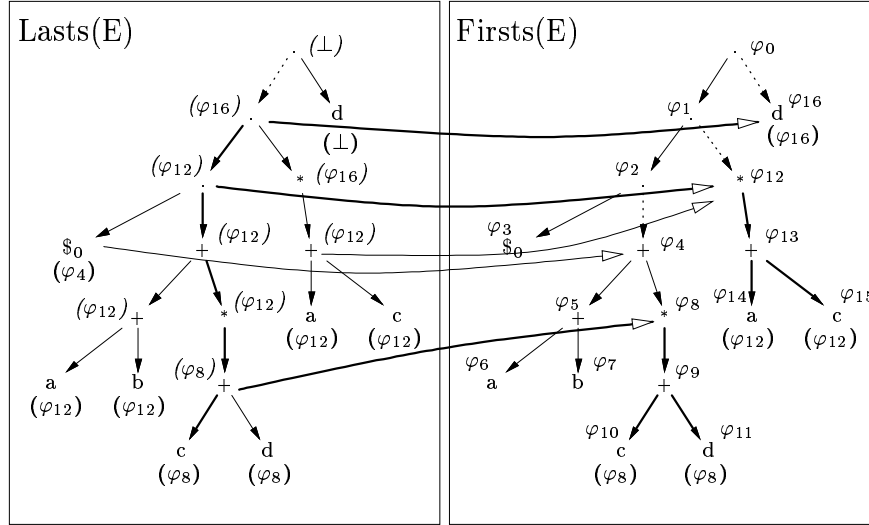


Fig. 4. Computation of the transitions of the state (φ_8, c_3) according to the Algorithm 2.

Let E be a normalized regular expression and $E_0 = \$E$. Theorem 2 leads to the Algorithm 1 that computes the relation \equiv_f of E via the \mathcal{ZPC} -structure of $E_0 = \$E$, and hence the set Q_f of states of the follow automaton of E . On the other hand, Algorithm 2 computes the set of transitions of the follow automaton.

Algorithm 1 Since the expression E is normalized, the expression E_0 is normalized too, as far as the syntax tree of E_0 satisfies left associativity of ‘.’ operation. Thus, according to Theorem 2, it comes: $x \equiv_f y \Leftrightarrow D_{E_0}(x) = D_{E_0}(y)$.

The Algorithm 1 is based on a marking of the positions of $Lasts(E_0)$ (see Figure 3). The set Q_f is initialized to \emptyset and the call $Marking(\lambda_E, \phi_0)$ is performed. Every position x is marked with the head of the lower follow link whose tail is on the path $\pi(x)$. Two positions in E_0 are \equiv_f -equivalent iff they have an identical marking. The predicate $broken(\lambda)$ is true iff λ is not connected to its parent in $Lasts(E_0)$. The procedure $Marking$ performs a prefix traversal of $Lasts(E_0)$. The marking $D(\lambda)$ of the node λ is equal to $\varphi = follow(\lambda)$ if there exists a follow link (λ, φ) and to the marking of its parent otherwise.

The set of classes of \equiv_f is the set of markings of the positions of E_0 . The computation of the set of transitions is facilitated by the use of a class representative, such as the least leaf (w.r.t. the order of the traversal) with a given marking. It can be achieved by a marking of the nodes of $Firsts(E_0)$ that is not detailed here. Finally we get: $Q_f = \{(\varphi, x) \mid \varphi \in Firsts(E_0), x \in Pos(E_0), x \text{ is the least position s.t. } D(x) = \varphi\}$. At the end of the execution, every position in $Lasts(E_0)$ is marked with the head of its associated lower follow link. There-

Algorithm 1 Computes the set Q_f of states of the follow automaton.

Procedure *Marking*(λ : node, parent_mark: node)

```
if follow( $\lambda$ ) =  $\perp$  then
   $D(\lambda) \leftarrow$  parent_mark
else
   $D(\lambda) \leftarrow$  follow( $\lambda$ )
end if
 $leftson \leftarrow$  left( $\lambda$ )
if  $leftson \neq \perp$  then
  if broken( $leftson$ ) then
    Marking( $leftson, \perp$ )
  else
    Marking( $leftson, D(\lambda)$ )
  end if
end if
 $rightson \leftarrow$  right( $\lambda$ )
if  $rightson \neq \perp$  then
  Marking( $rightson, D(\lambda)$ )
end if
if  $leftson = \perp \wedge rightson = \perp$  then {case of a node}
  if  $(D(\lambda), \cdot) \notin Q_f$  then
     $Q_f \leftarrow Q_f \cup (D(\lambda), \lambda)$ 
  end if
end if
```

Algorithm 2 Computes the set δ_f of transitions of the follow automaton.

Procedure *Transitions*()

```
for all  $(D(x), x) \in Q_f$  do
  for all  $y \in$  Targets( $x$ ) do
     $\delta_f \leftarrow \delta_f \cup \{(D(x), x), h(y), (D(y), \cdot))\}$ 
  end for
end for
```

Function *Targets*(λ : node)

```
 $T \leftarrow \emptyset$ 
repeat
   $\phi =$  follow( $\lambda$ )
  if  $\phi \neq \perp$  then
     $T \leftarrow T \cup$  First( $\phi$ )
  end if
  if broken( $\lambda$ ) then
     $\lambda \leftarrow \perp$ 
  else
     $\lambda \leftarrow$  parent( $\lambda$ )
  end if
until  $\lambda = \perp$ 
return  $T$ 
```

fore two positions in E_0 are equivalent iff they get an identical marking by the Algorithm 1. Moreover the Algorithm 1 has an $O(|E|)$ time complexity.

Algorithm 2 The Algorithm 2 computes the set of transitions of the follow automaton of E (see Figure 4). There exists a transition $((\varphi, x), h(y), (\varphi', x'))$ from (φ, x) to (φ', x') in Q_f iff $(x, h(y), y)$ is a transition of the position automaton and $D(y) = \varphi'$. The function *Targets* computes the set T of targets of the transitions coming from the state (φ, x) . For each follow link $(\lambda_1, \varphi_1 = \text{follow}(\lambda_1))$ such that λ_1 belongs to the path $\pi(x)$, the sets $\text{First}(\varphi_1)$ and T are merged in linear time. Moreover, since the expression E_0 is in SNF, the successive $\text{First}(\varphi_1)$ sets are disjoint. Therefore, for each state (φ, x) , the computation of T is linear. Hence the Algorithm 2 has an $O(c \times |E|)$ time complexity. Finally we get the following proposition:

Proposition 4 *The follow automaton of a normalized regular expression can be computed from its ZPC-structure by the Algorithms 1 and 2, with an $O(c \times |E|)$ time complexity, where c is the index of the relation \equiv_f .*

4 Experimental results

The two algorithms have been coded in *C++* using the *STL* (Standard Template Library) and the general design is object oriented. Both of the algorithms benefit from the same implementation of automata. Automata are represented by a data structure that allows fast insertion of states and transitions and a variant has been designed for the epsilon follow automaton that carries some optimizations.

The modus operandi is the following. For every regular expression, the two algorithms have been run one thousand times in order to have measurable times. The function `clock()` as been used since it provides the real CPU time of a process. All the tests have been run under Linux on a Pentium II 300 Mhz computer with 192 MB memory. The Figure 5 gives the running time (in seconds) versus the length of the expressions.

1. Randomly generated regular expressions: we used an home made random expression generator to produce 1000 expressions of length 30 to 240 with a step of 30. See Figure 5.(a) for the results.
2. Families of regular expressions: we have tested some families of regular expressions proposed by Ilie and Yu [13].
 - Family 1: we consider the expressions inductively defined by $E_1 = (a_1 + \epsilon)^*$ and $E_{i+1} = (E_i + F_i)^*$ where F_i is obtained from E_i by replacing each a_j by $a_{j+|E_i|}$. See Figure 5.(b) for the results.
 - Family 2: we consider the expressions of the form $E_n = a_1 \cdot (b_1 + \dots + b_n)^* + a_2 \cdot (b_1 + \dots + b_n)^* + \dots + a_n \cdot (b_1 + \dots + b_n)^*$. We generate expressions up to $n = 30$. See Figure 5.(c) for the results.

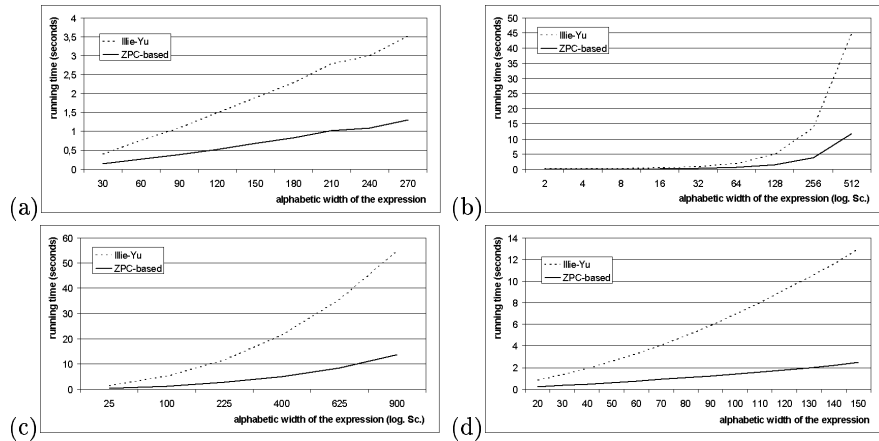


Fig. 5. Running time of the two algorithms: (a) Randomly generated expressions, (b) Family 1, (c) Family 2 and (d) Family 3.

- Family 3: we consider the expressions of the form $E_{n,m} = (a_1 + a_2 + \dots + a_n) \cdot (a_1 + a_2 + \dots + a_n + b_1 + \dots + b_m)^*$. We generate a set of regular expressions for length from 20 to 150 by step of 10 and for each length l , we consider all the possible values of n and m such that $l = 2n + m$. See Figure 5.(d) for the results.

5 Conclusion

Experimental tests show that the ε -free algorithm for computing the follow automaton is about three times faster than the original one. Moreover it is quite easy to implement it from the \mathcal{ZPC} -structure. This new construction should facilitate the study of the properties of the follow automaton and its comparison with other small NFAs such as Antimirov's automaton and Hromkovic's automaton.

References

1. Antimirov V., *Partial derivatives of regular expressions and finite automaton constructions*, Theoret. Comput. Sci., 155, 2917–319, 1996.
2. Brüggemann-Klein A., *Regular Expressions into Finite Automata*, Theoret. Comput. Sci., 120, 197–213, 1993.
3. Champarnaud J.-M., *Subset Construction Complexity for Homogeneous Automata, Position Automata and ZPC-Structures*, Theoret. Comput. Sci., 267, 17–34, 2001.
4. Champarnaud J.-M. and D. Ziadi, *Computing the Equation Automaton of a Regular Expression in $O(s^2)$ Space and Time*, in CPM'2001, Lecture Notes in Computer Science, A. Amir and G. M. Landau eds., Springer-Verlag, 2089(2001), 157–168.

5. Champarnaud J.-M. and D. Ziadi, *From c-continuations to new quadratic algorithms for automaton synthesis*, Intern. Journ. of Alg. and Comp., 11-6, 707–735, 2001.
6. Champarnaud J.-M., *Evaluation of three implicit structures to implement nondeterministic automata from regular expressions*, Intern. J. of Foundations of Comp. Sc., 13-1, 99-113, 2002.
7. Chang C.-H. and Paige R., *From Regular Expressions to DFA's Using Compressed NFA's*, Theoret. Comput. Sci., 178, 1–36, 1997.
8. Glushkov V.M., *The Abstract Theory of Automata*, Russian Math. Surveys, 16, 1–53, 1961.
9. Hagenah C. and Muscholl A., *Computing ε -free NFAs from regular expressions in $O(n \log^2(n))$ time*, Theoret. Inform. Appl. 34(4), 2000, 257–277.
10. Hromkovic J., Seibert S., Wilke T., *Translating regular expressions into small ε -free nondeterministic finite automata*, J. Comput. System Sci, 62(4), 2001, 565–588.
11. J.E. Hopcroft and J.D. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA, 1979.
12. Ilie L. and Yu S., *Constructing NFAs by optimal use of positions in regular expressions*, in: A. Apostolico, M. Takeda, eds., Proc. of CPM'02, Lecture Notes in Computer Science 2373, Springer-Verlag, 279–288, 2002.
13. Ilie L. and Yu S., *Algorithms for Computing Small NFAs*, in: K. Diks, W. Ritter, eds., Proc. of MFCS'02, Lecture Notes in Computer Science 2420, Springer-Verlag, 328–340, 2002.
14. McNaughton R., Yamada H., *Regular Expressions and State Graphs For Automata*, IEEE Trans. on Electronic Computers, 9-1, 39–47, 1960.
15. Ponty J.-L., Ziadi D. and Champarnaud J.-M., *A new Quadratic Algorithm to Convert a Regular Expression into an Automaton*, in: D. Raymond, D. Wood, S. Yu eds., Proc. of WIA'96, Lecture Notes in Computer Science 1260, Springer-Verlag, 109-110, 1997.
16. S. Yu, *Regular languages*, in: G. Rozenberg, A. Salomaa, Handbook of Formal Languages, Vol. I, Springer-Verlag, Berlin, 41–110, 1997.
17. Ziadi D., Ponty J.-L. and Champarnaud J.-M., *Passage d'une expression rationnelle à un automate fini non-déterministe*, Journées Montoises (1995), Bull. Belg. Math. Soc. 4, 177-203, 1997.