

An optimal parallel algorithm to convert a regular expression into its Glushkov automaton

Djelloul Ziadi and Jean-Marc Champarnaud
Laboratoire d'Informatique de Rouen
Pl. E. Blondel, 76821 Mont-Saint-Aignan, Cédex
emails : ziadi@dir.univ-rouen.fr jmc@dir.univ-rouen.fr

Abstract

The aim of this paper is to describe a CREW-PRAM optimal algorithm which converts a regular expression of size s into its Glushkov automaton in $O(\log s)$ time using $O(s^2/\log s)$ processors. This algorithm makes use of the star normal form of an expression as defined by Brüggemann-Klein and is based on the sequential algorithm due to Ziadi, Ponty and Champarnaud, which computes an original representation of Glushkov automaton in $O(s)$ time.

1 Introduction

PRAM model is a general framework used for describing and analyzing parallel algorithms. It consists of n processors working synchronously and exchanging data through a shared memory unit. We shall suppose here that concurrent reads are allowed, but concurrent writes are not (CREW-PRAM). A PRAM algorithm is said to be efficient if it works in a polylogarithmic time using a polynomial number of processors. It is said to be optimal if its sequential time (the product of its parallel time by the number of processors) is equal to the computation time of the fastest known sequential algorithm solving the problem. Our aim is to produce optimal PRAM algorithms in automata domain [13]; this paper describes such an algorithm, for converting a regular expression into an automaton.

There exist a lot of different sequential algorithms to convert a regular expression into an automaton. Watson's taxonomy [11] is an excellent reference for such a topic. Up to now, Thompson's approach [10], which yields a non-deterministic automaton with ε -transitions, is the only one to have been parallelized. Rytter algorithms [8] are based on adaptations of Thompson's method due to Hopcroft and Ullman [6], and to Sedgewick [9]. They work on a CREW-PRAM model and are optimal; they convert a regular expression of size s in $O(\log s)$ time using $O(s/\log s)$ processors.

Building an automaton from a regular expression is currently performed in order to test whether a word belongs to a given language or not. If the automaton has ε -transitions, they must be first eliminated, which is in $O(s^2)$ sequential time. This elimination is based on the computing of a transitive closure, for which there is no optimal PRAM algorithm. It is one of the reasons why we concentrate our efforts on computing a result without ε -transitions.

This paper describes a CREW-PRAM parallelization of Glushkov approach [5][7] which yields a non-deterministic automaton without ε -transitions. Our parallel algorithm is based on a new sequential algorithm described by Ziadi, Ponty and Champarnaud in [12] and [?]. This sequential algorithm (named ZPC algorithm) first transforms a regular expression of size s into an original representation of its Glushkov automaton in $O(s)$ time. Our parallelization is based on the following results :

- 1) an optimal algorithm which builds the ZPC representation in $O(\log s)$ time using $O(s/\log s)$ processors, as far as the expression is in star normal form (notion due to Brüggemann-Klein [2]).
- 2) an efficient algorithm which computes the star normal form of a regular expression in $O(\log s)$ time using $O(s)$ processors.
- 3) an optimal algorithm which converts the ZPC representation into a table of transitions in $O(\log s)$ time using $O(s^2/\log s)$ processors.

Combining 1 and 2 we get an efficient algorithm to compute the ZPC representation. Combining 1, 2 and 3 we get an optimal algorithm to convert a regular expression into its Glushkov automaton.

Section 2 presents terminology and writing conventions, and recalls Glushkov construction. Section 3 briefly describes ZPC sequential algorithm and introduces the notion of star normal form. Section 4 presents an optimal algorithm which computes ZPC representation for a regular expression in star normal form. Section 5 describes an efficient algorithm which constructs the star normal form of a regular expression. Section 6 provides an optimal algorithm which converts the ZPC representation into a table of transitions

in $O(\log s)$ time using $O(s^2/\log s)$ processors.

2 Definitions and writing conventions

In this section, we first introduce the terminology and the notations used in this paper. With a few exceptions, these notations can be found in [12].

2.1 Regular expressions and languages

Let Σ be a non-empty finite set of symbols, called the alphabet. Σ^* represents the set of all words over Σ . The empty word is denoted by ε . The symbols in Σ are represented by the first lower-case letters such as a, b, c, \dots . Union (+), product (\cdot), and Kleene star ($*$) are the classical regular operations over the subsets of Σ^* . Upper-case letters such as E, F and G represent regular expressions. In order to specify the position of the symbols in the expression, the symbols are subscripted following the order of reading. For example, starting from $E = (a + \varepsilon)ba$ we obtain the subscripted expression $\overline{E} = (a_1 + \varepsilon)b_2a_3$. Subscripts are called positions and are represented by the last lower-case letters of the alphabet, such as x, y, z . The set of positions of a regular expression E is denoted by $Pos(E)$. χ is the application which maps each position in $Pos(E)$ to the symbol of Σ which appears at this position in E . We denote by σ ($\sigma = \{\lambda_1, \lambda_2, \dots, \lambda_n\}$, where $|Pos(E)| = n$) the subscripted alphabet. We denote by $L(E)$ the language generated by the regular expression E . We denote by $T(E)$ the syntax tree associated with E . If ν is a node in $T(E)$, $symbol(\nu)$ is the operator or the operand associated with ν . We write ν_l (resp. ν_r) the left (resp. right) son of ν . If ν has a single son (it is the case if ν is labeled ' $*$ '), this son is written ν_s . By E_ν we denote the subexpression rooted at ν . The size $|E|$ of a regular expression E is the length of its postfix form. Under the assumption that a node of $T(E)$ and its son cannot be both labeled by Kleene star, it is easy to prove that $|E| = O(|Pos(E)|)$. Therefore we will express our complexity results as functions of $|E|$.

2.2 Glushkov automaton

In order to construct a non-deterministic finite automaton recognizing $L(E)$, Glushkov [5] has introduced four functions :

- $Null(E)$ is equal to $\{\varepsilon\}$ if $\varepsilon \in L(E)$ and \emptyset otherwise.

- $First(E)$ is the set of initial positions of words of the language $L(E)$.
- $Last(E)$ is the set of final positions of words of the language $L(E)$.
- $Follow(E, x)$ is the set of positions which follow immediately the position x in the expression E .

Example 1 For the expression $E = (a + b)^*ab$, we have :

- $\overline{E} = (a_1 + b_2)^*a_3b_4$,
- $Null(E) = \emptyset$,
- $First(E) = \{1, 2, 3\}$,
- $Last(E) = \{4\}$,
- $Follow(E, 1) = Follow(E, 2) = \{1, 2, 3\}$,
- $Follow(E, 3) = \{4\}$, $Follow(E, 4) = \emptyset$.

These four functions can be defined over σ in the following way :

- $Null(E) = \text{if } \varepsilon \in L(\overline{E}) \text{ then } \{\varepsilon\} \text{ else } \emptyset$
- $First(E) = \{x \in Pos(E) \mid \exists u \in \sigma^* : \lambda_x u \in L(\overline{E})\}$
- $Last(E) = \{x \in Pos(E) \mid \exists u \in \sigma^* : u \lambda_x \in L(\overline{E})\}$
- $Follow(E, x) = \{y \in Pos(E) \mid \exists v \in \sigma^*, \exists w \in \sigma^* : v \lambda_x \lambda_y w \in L(\overline{E})\}$

We shall use the following notation : for each set X , we note \mathcal{I}_X the function which is equal to $\{\varepsilon\}$ for all $x \in X$ and \emptyset otherwise.

Proposition 1 $Null(E)$ can be inductively computed as follows :

$$\begin{aligned}
Null(\{\varepsilon\}) &= \{\varepsilon\} \\
Null(\emptyset) &= \emptyset \\
Null(a) &= \emptyset \\
Null(F + G) &= Null(F) \cup Null(G) \\
Null(F.G) &= Null(F) \cap Null(G) \\
Null(F^*) &= \{\varepsilon\}
\end{aligned}$$

Proposition 2 *First can be inductively computed as follows :*

$$\begin{aligned}
First(\epsilon) &= \emptyset \\
First(\emptyset) &= \emptyset \\
First(x) &= \{x\} \\
First(F + G) &= First(F) \cup First(G) \\
First(F.G) &= First(F) \cup Null(F).First(G) \\
First(F^*) &= First(F)
\end{aligned}$$

Proposition 3 *Last can be inductively computed as follows :*

$$\begin{aligned}
Last(\epsilon) &= \emptyset \\
Last(\emptyset) &= \emptyset \\
Last(x) &= \{x\} \\
Last(F + G) &= Last(F) \cup Last(G) \\
Last(F.G) &= Last(G) \cup Null(G).Last(F) \\
Last(F^*) &= Last(F)
\end{aligned}$$

Proposition 4 *Follow(E, x) can be inductively computed as follows :*

$$\begin{aligned}
Follow(\epsilon, x) &= \emptyset \\
Follow(\emptyset, x) &= \emptyset \\
Follow(a, x) &= \emptyset \\
Follow(F + G, x) &= \mathcal{I}_{Pos(F)}(x).Follow(F, x) \cup \mathcal{I}_{Pos(G)}(x).Follow(G, x) \\
Follow(F.G, x) &= \mathcal{I}_{Pos(F)}(x).Follow(F, x) \cup \mathcal{I}_{Pos(G)}(x).Follow(G, x) \\
&\quad \cup \mathcal{I}_{Last(F)}(x).First(G) \\
Follow(F^*, x) &= \mathcal{I}_{Pos(F)}(x).Follow(F, x) \cup \mathcal{I}_{Last(F)}(x).First(F)
\end{aligned}$$

Definition 1 *Glushkov automaton $M_E = (Q_E, \Sigma, \delta_E, s_I, T_E, \chi)$ of the expression E is defined as follows :*

- $Q_E = Pos(E) \cup \{s_I\}$
- $\forall a \in \Sigma, \delta_E(s_I, a) = \{y \mid y \in First(E) \text{ and } \chi(y) = a\}$
- $\forall a \in \Sigma, \forall x \in Pos(E), \delta_E(x, a) = \{y \mid y \in Follow(E, x) \text{ and } \chi(y) = a\}$
- $F_E = Last(E) \cup Null(E). \{s_I\}$

Theorem 1 [12] *Let E be a regular expression and M_E its Glushkov automaton, then $L(E) = L(M_E)$. \square*

3 ZPC sequential algorithm

Let E be a regular expression of size s . A naive implementation of Glushkov algorithm leads to a $O(s^3)$ time complexity. Brüggemann-Klein [2], Chang and Paige [3], and Ziadi, Ponty and Champarnaud [12] have proposed algorithms with an $O(s^2)$ time complexity. The latter one (named ZPC algorithm) is the base of our parallelization. We briefly describe it now.

3.1 Computation of First and Last

We consider the syntax tree $T(E)$ and for each node ν in $T(E)$, we denote by $Set(\nu)$ the set of leaves in the subtree whose root is ν . This set will be represented by a list. In order to have an $O(1)$ access to $Set(\nu)$, ν points to its leftmost leaf and to its rightmost leaf. The forest $TF(E)$ which maps each node ν to $First(E_\nu)$ is computed in the following way according to the Proposition 2 :

1. Initialize $TF(E)$ by $T(E)$.
2. For every node ν labeled “.”, cut the link to its right son ν_r if $Null(E_{\nu_l}) = \emptyset$, with respect to the statement $First(F \cdot G) = First(F) \cup Null(F) \cdot First(G)$.
3. For each node, update its pointers to its leftmost and rightmost leaves, and link the rightmost leaf of its left son to the leftmost leaf of its right son.

The forest $TL(E)$ which maps each node ν to $Last(E_\nu)$ is computed in a dual way with respect to the statement $Last(F \cdot G) = Last(G) \cup Null(G) \cdot Last(F)$ of Proposition 3 (see figure 2).

3.2 Computation of $\delta(M_E)$

Let Δ_ν be the set of edges induced by the node ν . Δ_ν is defined as follows:

$$\Delta_\nu = \begin{cases} Last(E_{\nu_l}) \times First(E_{\nu_r}) & \text{if } \nu \text{ is labeled } \cdot \\ Last(E_{\nu_s}) \times First(E_{\nu_s}) & \text{if } \nu \text{ is labeled } * \\ \emptyset & \text{otherwise} \end{cases}$$

On the data structure we use, cartesian products involved by Δ_ν calculus are implemented by a pointer from ν_l in $TL(E)$ to ν_r in $TF(E)$ if ν is labeled by “.”, or from ν_s in $TL(E)$ to ν_s in $TF(E)$ if ν is labeled by “*” (see figure 2). These pointers are called links “follow”.

Proposition 5 [12] $\delta(M_E) = \left(\bigcup_{\nu \in T(E)} \Delta_\nu \right) \cup (\{s_I\} \times \text{First}(E))$

$\delta(M_E)$ is not necessarily a disjoint union. ZPC algorithm eliminates redundant Δ_ν sets in order to get a disjoint union. The parallel algorithm we describe in the next section makes use of the fact that the star normal form of the expression E also yields a disjoint union.

3.3 Star normal form

According to Brüggemann-Klein [2], a regular expression E is said to be in star normal form (SNF) if for each H such that H^* is a subexpression of E , we have

$$\forall x \in \text{Last}(H), \text{Follow}(H, x) \cap \text{First}(H) = \emptyset$$

This condition is called *SNF* condition.

Theorem 2 [2] *For each regular expression E , there is a regular expression E^\bullet called the star normal form of E such that :*

1. E^\bullet satisfies the SNF condition
2. $M_{E^\bullet} = M_E$
3. M_{E^\bullet} is computed in $O(s^2)$ time, where $s = |E|$.

In order to compute E^\bullet , Brüggemann-Klein introduces the expression E° verifying the following conditions:

1. E° satisfies the SNF condition,
2. $M_{E^\circ*} = M_{E^*}$.

By recursively substituting each subexpression H^* of E with $H^{\circ*}$, we obtain E^\bullet .

Proposition 6 [2] *E^\bullet can be computed by the following inductive rules :*

$$\begin{array}{ll} [E = \varepsilon \text{ or } \emptyset] & E^\bullet = E \\ [E = a] & E^\bullet = E \\ [E = F + G] & E^\bullet = F^\bullet + G^\bullet \\ [E = FG] & E^\bullet = F^\bullet G^\bullet \\ [E = F^*] & E^\bullet = F^{\circ**} \end{array}$$

Proposition 7 $E^{\circ\bullet}$ can be computed by the following inductive rules :

$$\begin{array}{ll}
[E = \varepsilon \text{ or } \emptyset] & E^{\circ\bullet} = \emptyset \\
[E = a] & E^{\circ\bullet} = E \\
[E = F + G] & E^{\circ\bullet} = F^{\circ\bullet} + G^{\circ\bullet} \\
[E = FG] & E^{\circ\bullet} = \begin{cases} F^{\bullet}G^{\bullet} & \text{if } \text{Null}(F) = \emptyset \text{ and } \text{Null}(G) = \emptyset \\ F^{\circ\bullet}G^{\bullet} & \text{if } \text{Null}(F) = \emptyset \text{ and } \text{Null}(G) = \{\varepsilon\} \\ F^{\bullet}G^{\circ\bullet} & \text{if } \text{Null}(F) = \{\varepsilon\} \text{ and } \text{Null}(G) = \emptyset \\ F^{\circ\bullet} + G^{\circ\bullet} & \text{if } \text{Null}(F) = \{\varepsilon\} \text{ and } \text{Null}(G) = \{\varepsilon\} \end{cases} \\
[E = F^*] & E^{\circ\bullet} = F^{\circ\bullet}
\end{array}$$

Example 2 Computation of E^{\bullet} for $E = (a^*b^*)^*ab$:

$$\begin{aligned}
E^{\bullet} &= ((a^*b^*)^*ab)^{\bullet} \\
&= ((a^*b^*)^*a)^{\bullet}b^{\bullet} \\
&= ((a^*b^*)^*)^{\bullet}a^{\bullet}b \\
&= (a^*b^*)^{\circ\bullet*}ab \\
&= (a^{*\circ\bullet} + b^{*\circ\bullet})^*ab \\
&= (a^{\circ\bullet} + b^{\circ\bullet})^*ab \\
&= (a + b)^*ab
\end{aligned}$$

The relation between the star normal form and the elimination of redundant links in ZPC algorithm is resumed by the following Proposition :

Proposition 8 [12] *If E is in star normal form then :*

$$\delta(M_E) = \left(\bigsqcup_{\nu \in T(E)} \Delta_{\nu} \right) \sqcup (\{s_I\} \times \text{First}(E))$$

3.4 Modified ZPC algorithm

In order to parallelize ZPC algorithm we modify it as follows :

- 1.a Construct the tree $T(E)$
- 1.b Compute the star normal form E^{\bullet} of E
2. For each node ν in the tree $T(E^{\bullet})$ compute $\text{Null}(E_{\nu}^{\bullet})$
3. Construct the forests $TL(E^{\bullet})$ and $TF(E^{\bullet})$
4. For each node ν in $T(E^{\bullet})$ compute the links “follow” representing Δ_{ν}

4 Parallelization of modified ZPC algorithm

In this section we suppose that the expression E verifies *SNF* condition. Let s be the size of E . We show that in this case modified ZPC algorithm can be parallelized in $O(\log s)$ time using $O(s/\log s)$ processors, which is an optimal result. Steps 1.a, 2, 3 and 4 of modified ZPC algorithm are parallelized in the following way.

4.1 Step 1.a : $T(E)$ construction

We make use of the optimal parallel algorithm due to Bar-On and Vishkin [1], which computes the syntax tree of an arithmetic expression of size s in $O(\log s)$ time using $O(s/\log s)$ processors. This algorithm works on a completely bracketed expression; it means that each subexpression is enclosed between a left and a right bracket (these brackets form a pair). Bar-On and Vishkin claim that an expression can be converted to an equivalent completely bracketed expression in $O(\log s)$ optimal time using $O(s/\log s)$ processors. The syntax tree $T(E)$ is constructed from the sequence P_E of brackets of the completely bracketed expression E . Each node ν of $T(E)$ is associated with the pair of brackets enclosing the subexpression E_ν (see figure 4). In order to construct $T(E)$, Bar-On and Vishkin define the function *match* which associates to every position i in P_E the position j such that the brackets $P_E(i)$ and $P_E(j)$ form a pair. For example, in the figure 4, we have $match(4) = 17$ and $match(5) = 10$. *match* function is computed in time $O(\log s)$ with $O(s/\log s)$ processors and so $T(E)$ construction is achieved with the same complexity.

4.2 Step 2 : *Null* Computation

We adopt the parallel algorithm that Gibbons and Rytter [4] give for evaluation of an algebraic expression. This algorithm is optimal : its time complexity is $O(\log s)$ using $O(s/\log s)$ processors.

4.3 Step 3 : $TL(E)$ and $TF(E)$ construction

As mentioned in previous section, $TL(E)$ is deduced from $T(E)$ by deleting every link between a node ν labeled ‘.’ to its left son ν_l , if $Null(E_{\nu_r}) = \emptyset$. Dually, $TF(E)$ is deduced from $T(E)$ by deleting every link between a node ν labeled ‘.’ to its right son ν_r , if $Null(E_{\nu_l}) = \emptyset$. Links deletion is easily performed in $O(1)$ time using $O(s)$ processors. In both forests, every node

ν points to the leftmost leaf (*leftmost*(ν) pointer) and to the rightmost leaf (*rightmost*(ν) pointer) in the tree rooted at ν . The computation of *leftmost* and *rightmost* pointers requires a particular attention. We present an optimal algorithm for achieving this step. We describe it in the case of the computation of *leftmost* pointers in $TL(E)$ and we outline adjustments for computation of *rightmost* pointers and for computation in $TF(E)$. Let us first illustrate it on an example.

Example 3 Let $E = ((a)(((a)(b)) + ((c)(d))))$. We compute $T(E)$ and $TL(E)$. It appears that *leftmost* pointers cannot be easily computed from a standard traversal of $TL(E)$. So, we make a copy $T'(E)$ of $T(E)$ in which we permute the subtrees rooted at ν_l and ν_r every time a link from a node ν to its left son ν_l is deleted in $TL(E)$. Now, we can use the trace of the top-down right suffix traversal of $T'(E)$ to draw *leftmost* pointers. In our example, that trace is equal to $a_1c_4d_5 \cdot a_2b_3 \cdot + \cdot$. The last leaf appearing before a node ν in the trace is the *leftmost* leaf of ν .

| | | | | | | | | | |
|-----------------|-------|-------|-------|---------|-------|-------|---------|-------|---------|
| trace | a_1 | c_4 | d_5 | \cdot | a_2 | b_3 | \cdot | $+$ | \cdot |
| <i>leftmost</i> | a_1 | c_4 | d_5 | d_5 | a_2 | b_3 | b_3 | b_3 | b_3 |

We now detail the two sub-steps of this algorithm (computation of *trace*, computation of *leftmost* pointers).

4.3.1 Sub-step 1: computation of *trace*

Subtrees permutations are not physically realized; they are rather performed via a marking of the opening brackets of the sequence P_E . For a node ν , if its link to ν_l is deleted in $TL(E)$, the opening bracket of ν_l (resp. ν_r) is marked by r (resp. l); otherwise the opening bracket of ν_l (resp. ν_r) is marked by l (resp. r). By convention, the opening bracket of the root is marked by l .

In our example we have $P_E = ((((((l)())())())())())$ and after subtrees permutations we get $P_E = (l(r)(l(l(r)(l)(r(r)(l))))$.

Let $rank(\nu)$ be the rank of the node ν in a top-down right suffix traversal of $T'(E)$. Our aim is to compute $rank(\nu)$ without constructing P'_E nor $T'(E)$. Let $A[i]$ be the position of the opening bracket $P_E[i]$ in the sequence of opening brackets of $T'(E)$. We first compute $A[i]$ with the help of *brother* function defined as follows :

$$brother(i) = \begin{cases} match(match(i)+1)+1 & \text{if } P_E[match(i) + 1] = 'r' \\ match(i)+1 & \text{otherwise} \end{cases}$$

Then we get $rank(\nu)$ by computation of prefix sums of the array A . We finally use $rank()$ to compute the *trace* of a top-down right suffix traversal of $T'(E)$. The resulting procedure is :

```

begin
  Computation of positions in the sequence of opening brackets  $P'_E$ 
  forall  $1 \leq i \leq |P_E|$  pardo
    case  $P_E[i]$  of
      ( $l$ : begin  $A[i]:=1$ ;  $A[match[i]]:=-A[i]$ ; end;
       ( $r$ : begin  $A[i]:=(match[brother[i]] - brother[i] + 1)/2$ ;  $A[match[i]]:=-A[i]$ ; end;
      end;
    Computation of ranks in  $T'(E)$ 
  prefix-sums of  $A$ 
  Collecting trace
  forall  $1 \leq i \leq |P_E|$  pardo  $trace[i]:=node[i]$ ;
end

```

where $node[i]$ is the symbol associated with brackets $P[i]$ and $P[match[i]]$.

Example 4 We consider the expression $E = a(ab + cd)$. Prefix sums on A are denoted by $\Sigma_p(A)$.

| | | | | | | | | | | | | | | | | | | |
|---------------|-------|-------|-------|-------|-------|-------|-------|-------|-------|----|-------|-------|-------|-------|-------|----|----|----|
| i | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| P_E | (l | (r |) | (l | (l | (r |) | (l |) |) | (r | (r |) | (l |) |) |) |) |
| node | . | a_1 | a_1 | + | . | a_2 | a_2 | b_3 | b_3 | . | . | c_4 | c_4 | d_5 | d_5 | . | + | . |
| A | 1 | 8 | -8 | 1 | 1 | 2 | -2 | 1 | -1 | -1 | 4 | 2 | -2 | 1 | -1 | -4 | -1 | -1 |
| $\Sigma_p(A)$ | 1 | 9 | 1 | 2 | 3 | 5 | 3 | 4 | 3 | 2 | 6 | 8 | 6 | 7 | 6 | 2 | 1 | 0 |
| trace | a_1 | c_4 | d_5 | . | a_2 | b_3 | . | + | . | | | | | | | | | |

Our claim is that this procedure correctly computes the trace of a top-down right suffix traversal of $T'(E)$. The proof comes from Lemma 1 and Lemma 2.

Lemma 1 $\forall i, 1 \leq i \leq |P_E|, \sum_{k=i}^{match(i)} A[k] = 0 \quad \square$

Lemma 2 Let ν be a node and i be the position of the opening bracket in P_E associated with the node ν in $T'(E)$. We have : $rank(\nu) = \sum_{k=1}^i A[k]$.

Proof . We shall note $r(i) = \sum_{k=1}^i A[k]$. Lemma 2 is verified for $T'(E)$ root since $rank(\nu) = r(1) = 1$. Consider node ν at position i and let us assume that Lemma 2 holds for nodes whose positions are less or equal to i . Let g and d be positions in P_E of opening brackets respectively associated with ν_l and ν_r in $T'(E)$ ($g = brother[d]$). We are going to show that Lemma 2 holds for ν_r and ν_l . There are two possibilities :

Case 1 : $g \leq d$

In this case $g = i + 1$. By induction $r(i) = rank(\nu)$. $T'(E)$ traversal is such that $rank(\nu_l) = rank(\nu) + 1$. So we get $rank(\nu_l) = r(i) + 1$. As $A[g] = 1$ (by initialization) and $g = i + 1$, we can write $rank(\nu_l) = \sum_{k=1}^g A[k]$.

On the other hand, $T'(E)$ traversal is such that $rank(\nu_r) = rank(\nu) + t(\nu_l) + 1$, where $t(\nu_l)$ is the number of nodes in the subtree rooted at ν_l . As $t(\nu_l)$ is equal to the number of pairs of brackets inside the pair associated to ν_l (including this pair), it is easy to verify that we have $t(\nu_l) = (match[g] - g + 1)/2$. Therefore it comes $rank(\nu_r) = rank(\nu) + (match[g] - g + 1)/2 + 1$.

Let us consider now $r(d) = \sum_{k=1}^d A[k]$. We have
 $r(d) = (\sum_{k=1}^i A[k]) + (\sum_{k=i+1}^{d-1} A[k]) + A[d]$.

As $g = i + 1$ and $match[g] = d - 1$, Lemma 1 implies $\sum_{k=i+1}^{d-1} A[k] = 0$. Moreover, by initialization $A[d] = 1 + (match[g] - g + 1)/2$. So we have $r(d) = r(i) + (match[g] - g + 1)/2 + 1$.

By induction $rank(\nu) = r(i)$, so we have $rank(\nu_l) = r(d)$.

Case 2 : $g > d$

The demonstration is similar as in the first case. \square

4.3.2 Substep 2 : computation of *leftmost* pointers

Computation of *leftmost* pointers is achieved via a marking of *trace* based on the following lemma :

Lemma 3 *Let T be a syntax tree and M eb a marking of the trace of a right suffix traversal of T . We shall assume T is not void. Suppose internal nodes are initially marked by 0 and leaves by 1. Then prefix sums of M give the same mark to nodes having the same leftmost leaf.*

Proof . The initial value of M can be seen as a word of the language $(10^*)^+$. For each subexpression 10^* of such a word, nodes marked by 0 (if any) have the same leftmost leaf, which is the node marked by 1. All of these nodes (and only them) will be identically marked by prefix sums of

$M.$ \square

We associate a processor to each element of *trace*. Each processor determines its leftmost leaf by performing the following sequence :

```
begin
  Initialization of marking B
  forall  $1 \leq i \leq s$  pardo
    if trace[ $i$ ] is a letter then  $B[i]:=1$  else  $B[i]:=0$ ;
  Computation of marking B
  prefix-sums of B
  Collecting leftmost pointers
  forall  $1 \leq i \leq s$  pardo
    if trace[ $i$ ] is a letter then  $leaf[B[i]]:=trace[i]$ ;
  forall  $1 \leq i \leq s$  pardo  $leftmost[i]:=leaf[B[i]]$ ;
end
```

We shall complete step 3 description by the following remarks :

1. The computation of the *rightmost* pointers can be done in a similar way by calculating the order of the opening brackets in a prefix traversal of $T(E)$ and by achieving the prefix sums in B from right to left.
2. The computation of *leftmost* and *rightmost* pointers in the forest $TF(E)$ is deduced from the construction presented on $TL(E)$.
3. The computation of linking of the leaves inside a same tree, works as follows. Associate a processor to each node ν in $T(E)$ which performs the following sequence :

```
begin
  join  $rightmost(\nu_l)$  to  $leftmost(\nu_r)$  in  $TL(E)$ 
  join  $rightmost(\nu_l)$  to  $leftmost(\nu_r)$  in  $TF(E)$ 
end
```

Conclusion of step 3 : the computation of $TL(E)$ and $TF(E)$ (*leftmost* and *rightmost* pointers, leaves linking) can be achieved in $O(\log s)$ time using $O(s/\log s)$ processors (same complexity as for prefix-sums).

4.4 Step 4 : computation of the links “follow” representing Δ_ν

In order to create the links representing Δ_ν , to each node ν we associate a processor which performs the following sequence :

```

begin
  case  $\nu$  of
    · : join  $\nu_l$  in  $TL(E)$  to  $\nu_r$  in  $TF(E)$ 
    * : join  $\nu_s$  in  $TL(E)$  to  $\nu_s$  in  $TF(E)$ 
  end
end
end

```

This sequence is achieved in constant time using $O(s)$ processors.

4.5 Conclusion

The result of these successive steps is illustrated by figure 6. With respect to the complexity of each step, we can state the following Theorem:

Theorem 3 *Let E be a regular expression of size s , verifying SNF condition. The Glushkov automaton of E represented by the forest of Lasts, the forest of Firsts and the links follow can be computed by an optimal parallel algorithm of time complexity $O(\log s)$ using $O(s/\log s)$ processors on a CREW-PRAM.*

5 Computation of the star normal form

Let E be a regular expression and $T(E)$ its syntax tree. We assume that the function $father()$ has been computed on $T(E)$. The problem is the following : given $T(E)$, build $T(E^\bullet)$, the syntax tree of the star normal form E^\bullet of E . We consider the forests $F(E^\bullet)$ and $F(E^{\circ\bullet})$ associated to the expressions E^\bullet and $E^{\circ\bullet}$ (cf section 3) and constructed as follows :

- a) $F(E^\bullet)$ and $F(E^{\circ\bullet})$ are initialized by a copy of $T(E)$.
- b) The computation of E^\bullet according to Theorem 3 partitions $F(E^\bullet)$ (resp. $F(E^{\circ\bullet})$) into subtrees inside which internal nodes are evaluated without jumping in $F(E^{\circ\bullet})$ (resp. $F(E^\bullet)$). We modify the function $father()$ in both forests in order to represent these partitions. Moreover, in $F(E^{\circ\bullet})$, we replace the operator “.” by the operator “+” with respect to the definition of $(G \cdot H)^{\circ\bullet}$ and we replace the operator “*” by the identity operator “ \diamond ”

with respect to the Definition of $(E^*)^{\circ\bullet}$. Figure 7 gives the representation of the function $father()$ in the forests $F(E^\bullet)$ and $F(E^{\circ\bullet})$ for the expression $E = (a^*b^*)^*ab$.

We denote by $\nu(E^\bullet)$ (resp. $\nu(E^{\circ\bullet})$) the node of $F(E^\bullet)$ (resp. $F(E^{\circ\bullet})$) corresponding to the node ν of $T(E)$. We associate a processor to every node ν in $T(E)$ and each processor performs the following sequence :

```

begin
  case symbol( $\nu$ ) of
    · : case (Null( $E_{\nu_l}$ ), (Null( $E_{\nu_r}$ ))) of
      ( $\emptyset, \emptyset$ )      : begin father( $\nu_l(E^{\circ\bullet})$ ):= nil; father( $\nu_r(E^{\circ\bullet})$ ):= nil; end;
      ( $\{\varepsilon\}, \emptyset$ )   : father( $\nu_l(E^{\circ\bullet})$ ):= nil;
      ( $\emptyset, \{\varepsilon\}$ ) : father( $\nu_r(E^{\circ\bullet})$ ):= nil;
    end
    * : begin father( $\nu_s(E^\bullet)$ ):= nil; symbol( $\nu(E^{\circ\bullet})$ ):=  $\nu$  end;
  end;
end

```

This construction is achieved in constant time using $O(s)$ processors.

The problem is now to deduce $T(E^\bullet)$ from $F(E^\bullet)$ and $F(E^{\circ\bullet})$. We denote by ν^\bullet the node of $T(E^\bullet)$ corresponding to the node ν of $T(E)$. We associate a processor to each node ν of $T(E)$. Each processor must decide whether $\nu^\bullet = \nu(E^\bullet)$ or $\nu^\bullet = \nu(E^{\circ\bullet})$. A sequential solution to this problem would be typically recursive. We need some technique to make a local decision. This technique is illustrated on a simplified case where trees are replaced by lists, as shown in figure 8. Let $L_a = (a_0, a_1, \dots, a_n)$ and $L_b = (b_0, b_1, \dots, b_n)$ be two lists. We arbitrary suppress some links in each list, with respect to the condition :

$$\forall i, 0 \leq i < n, (next(a_i) = nil) \wedge (next(b_i) = nil) = false$$

L_a and L_b are now collections of lists. Our aim is to compute the list $L = (c_0, c_1, \dots, c_n)$ from L_a and L_b such that :

$$\begin{aligned}
& - c_0 = a_0 \\
& - \forall i, 0 \leq i < n, \text{ if } c_i = a_i \\
& \quad \text{then } c_{i+1} = \begin{cases} a_{i+1} & \text{if } next(a_i) \neq nil \\ b_{i+1} & \text{otherwise} \end{cases} \quad \text{else } c_{i+1} = \begin{cases} b_{i+1} & \text{if } next(b_i) \neq nil \\ a_{i+1} & \text{otherwise} \end{cases}
\end{aligned}$$

Let x be an element of list. We call rank of x the distance $d(x)$ of x to the head of the list. The assignment of a_i or b_i to c_i can be locally decided from $d(a_i)$ and $d(b_i)$, using the property : $d(a_i) \geq d(b_i) \Leftrightarrow c_i = a_i$.

This technique extends to the case of forests (see figure 7); $d(\nu)$ is the distance of node ν to the root of the tree it belongs to; $d(\nu)$ is computed by doubling technique, in $O(\log s)$ time using $O(s)$ processors. Each processor performs the following sequence :

```

begin
  if  $d(\nu(E^{\circ\bullet})) > d(\nu(E^\bullet))$ 
  then  $symbol(\nu^\bullet) := symbol(\nu(E^{\circ\bullet}))$ 
  else  $symbol(\nu^\bullet) := symbol(\nu(E^\bullet))$ ;
end

```

Finally we can state the following Theorem :

Theorem 4 $T(E^\bullet)$ can be computed from $T(E)$ in time $O(\log s)$ using $O(s)$ processors on a CREW-PRAM.

6 Computation of the sets $Follow(E, x)$

In this section we assume that E is in star normal form. We show that, under this assumption, it is possible to compute the set $Follow(E, x)$, for $x \in Pos(E)$, as a linked list L_x of states in time $O(\log s)$ using $O(s/\log s)$ processors in a CREW-PRAM model. The position x is also a leaf of $TL(E)$. We consider the nodes $\lambda_1, \lambda_2, \dots, \lambda_m$ which are ancestors of x in the tree containing x , and which are heads of a link *follow*. Let us denote $I_x = \{\phi_1, \phi_2, \dots, \phi_m\}$ the set of associated tails. The following algorithm computes the list L_x .

```

begin
1 Construction of the list  $I_x = \{\phi_1, \phi_2, \dots, \phi_m\}$ 
2 forall  $1 \leq i \leq m - 1$  pardo
3   join  $leftmost(\phi_i)$  to  $leftmost(\phi_{i+1})$ 
4 join  $L_x$  to  $leftmost(\phi_1)$ 
end

```

Let us analyze the complexity of this algorithm. First we show that step 1 can be achieved in time $O(\log s)$ using $O(s/\log s)$ processors. We shall make use of P_E , the sequence of brackets associated to E . Let i be the position in P_E of the opening bracket associated to the node x , and r be the position of the opening bracket associated to the root of the tree of $TL(E)$ containing x . We shall assume that every opening bracket is labeled with the

rank of the corresponding node in $T(E)$. Let us consider the subsequence $S = P_E[r] \dots P_E[i]$. The reduced sequence [4] S' of S can be obtained by deleting all the brackets $P_E[j]$ such that $j \in [r, i]$ and $match[j] \in [r, i]$. For example, if $S = =)())(($, then $S' = =)()$. S' can be computed in time $O(\log s)$ using $O(s/\log s)$ processors [4]. Let us remark that S' is exactly the sequence of opening brackets associated with the ancestors of x . Among these ancestors we eliminate those which are not heads of links *follows*. and then we compute I_x , in time $O(\log s)$ using $O(s/\log s)$ processors.

It is easy to see that Steps 2-4 can be achieved in time $O(\log s)$ using $O(s/\log s)$ processors. So the computation of all the sets $Follow(E, x)$ can be done in time $O(\log s)$ using $O(s^2/\log s)$ processors in a CREW-PRAM.

Theorem 5 *Let E be a regular expression of size s . Glushkov automaton associated to E can be computed in time $O(\log s)$ using $O(s^2/\log s)$ processors in a CREW-PRAM. \square*

7 Conclusion

We have described an optimal parallel algorithm to compute ZPC representation of the star normal form of an expression and an efficient algorithm to compute the star normal form of an expression. They combine in an efficient algorithm ($O(\log s)$ time using $O(s)$ processors) to compute the ZPC representation of an expression. We do not know whether there exists an optimal algorithm to compute the star normal form of an expression, but we provide an optimal algorithm to convert the ZPC representation of an expression verifying *SNF* condition into a table of transitions ($O(\log s)$ time using $O(s^2/\log s)$ processors). Thus we finally get an optimal algorithm to convert a regular expression into its Glushkov automaton, in $O(\log s)$ time using $O(s^2/\log s)$ processors.

References

- [1] I. Bar-On and U. Vishkin, "Optimal parallel generation of a computation tree form", *ACM Transactions on Programming Languages and Systems*, 1985, 348-57.
- [2] A. Brüggemann-Klein, "Regular Expressions into Finite Automata", *Theoretical Computer Science* **120**, 1993, 197-213.

- [3] C.-H. Chang and R. Paige, “From regular expressions to DFAs using compressed NFAs”, in Apostolico. Crochemore. Galil. and Manber. editors. LNCS **644**, *Combinatorial Pattern Matching Proceedings*, Springer Verlag, 1992, 88-108.
- [4] A.M. Gibbons and W. Rytter, “Efficient Parallel algorithms”, *Cambridge University Press*, 1988.
- [5] V.M. Glushkov, “The abstract theory of automata”, *Russian Mathematical Surveys*, **16**, 1961, 1-53.
- [6] J.E. Hopcroft and J. Ullman, “Introduction to Automata Theory, Languages and Computation”, *Addison-Wesley*, 1979.
- [7] R. McNaughton and H. Yamada, “Regular Expression and State Graphs for Automata”, *IRA Trans. on Electronic Computers* EC-9, **1**, 1960, 39-47.
- [8] W. Rytter, “A note on parallel transformations of regular expressions to nondeterministic finite automata”, *Information Processing Letters*, **31**, 1989, 103-109.
- [9] R. Sedgewick, “Algorithms”, *Addison-Wesley*, 1983.
- [10] K. Thompson, “Regular expression search algorithms”, *C. ACM*, **11**(6), 1968, 419-422.
- [11] B. Watson, “Taxonomies and Toolkits of Regular Language Algorithms”, *CIP-DATA Koninklijke Bibliotheek*, Den Haag, Ph. D, Eindhoven University of Technology, 1995.
- [12] D. Ziadi, J.-L. Ponty et J.-M. Champarnaud, “Passage d’une Expression Rationnelle à un Automate Fini Non-déterministe”, *Rapport L.I.R.95.05*, à paraître dans *Bulletin of the Belgian Mathematical Society - Simon Stevin*.
- [13] D. Ziadi, “Algorithmique parallèle et séquentielle des automates”, Thèse de doctorat, Université de Rouen, 1996.

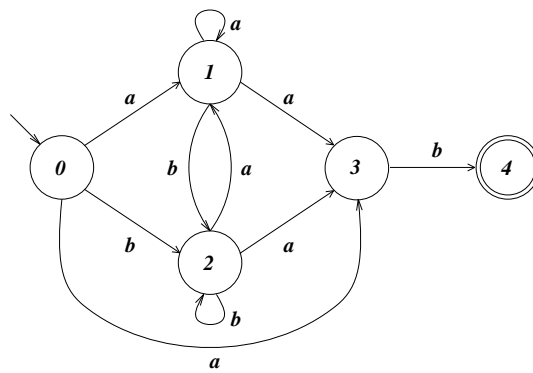
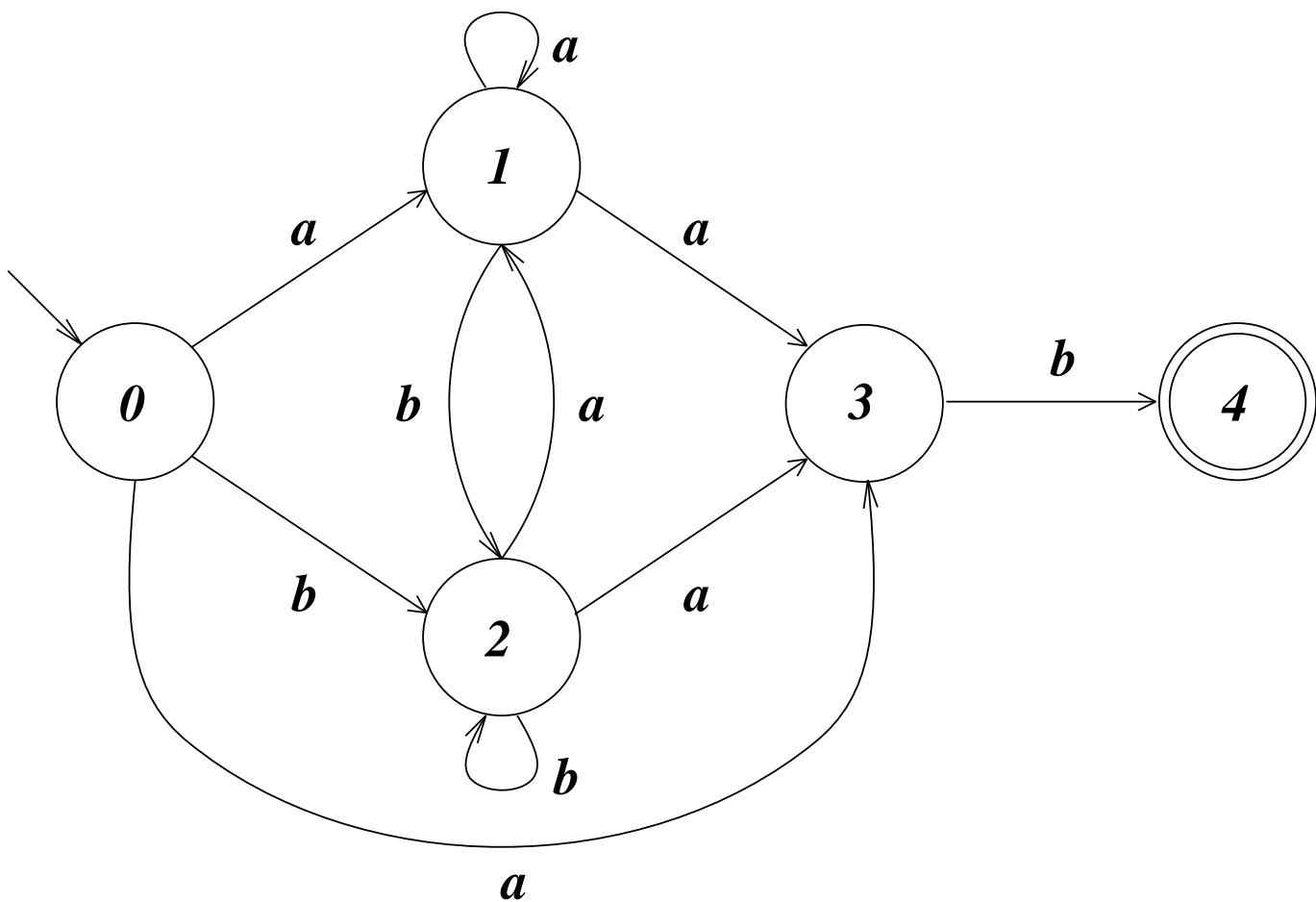
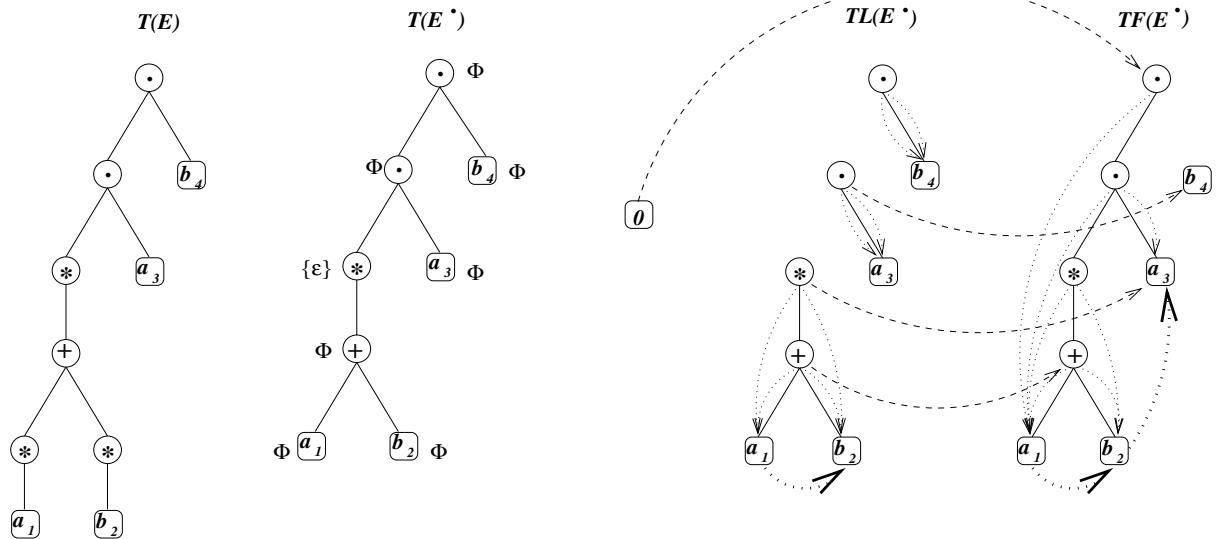


Figure 1: Glushkov automaton of the expression $E = (a + b)^* \cdot a \cdot b$



| ν_i | $Last(\nu_i)$ | $First(\nu_i)$ | Δ_{ν_i} |
|---------|---------------|----------------|--|
| ν_5 | {1} | {1} | $= \emptyset$ |
| ν_4 | {1} | {1} | $Last(\nu_5) \times First(\nu_5) = \{1\} \times \{1\}$ $= \{(1,1)\}$ |
| ν_6 | {2} | {2} | $= \emptyset$ |
| ν_7 | {2} | {2} | $Last(\nu_6) \times First(\nu_6) = \{2\} \times \{2\}$ $= \{(2,2)\}$ |
| ν_3 | {1,2} | {1,2} | $Last(\nu_4) \times First(\nu_7) = \{1\} \times \{2\}$ $= \{(1,2)\}$ |
| ν_2 | {1,2} | {1,2} | $Last(\nu_3) \times First(\nu_3) = \{1,2\} \times \{1,2\}$ $= \{(1,1), (1,2), (2,1), (2,2)\}$ |
| ν_8 | {3} | {3} | $= \emptyset$ |
| ν_1 | {3} | {1,2,3} | $Last(\nu_2) \times First(\nu_8) = \{1,2\} \times \{3\}$ $= \{(1,3), (2,3)\}$ |
| ν_9 | {3} | {3} | $= \emptyset$ |
| ν_0 | {4} | {1,2,3} | $Last(\nu_1) \times First(\nu_9) = \{3\} \times \{4\}$ $= \{(3,4)\}$ |

Figure 2: ZPC algorithm for the expression $E = (a_1^*b_2^*)^*.a_3.b_4$



Links follow

..... Leftmost and rightmost pointers

..... Linking of the leaves

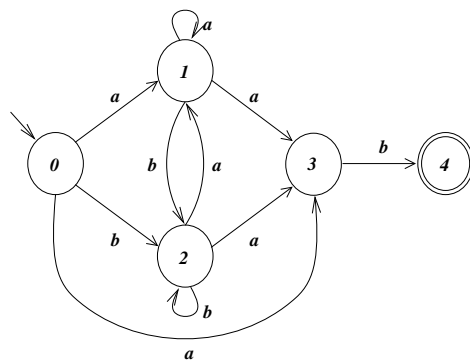


Figure 3: Modified ZPC algorithm for the expression $E = (a^* + b^*) \cdot a \cdot b$.

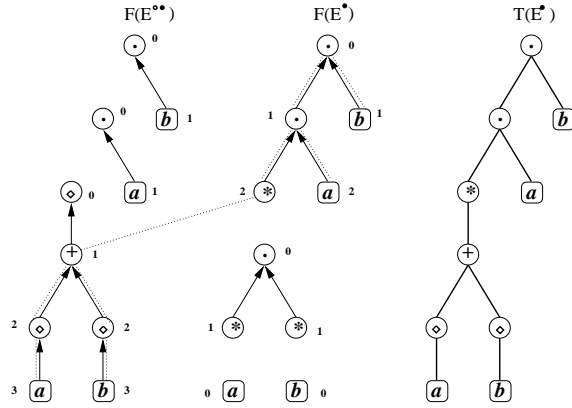


Figure 7: The forests associated to $(a^*b^*)^*ab$.

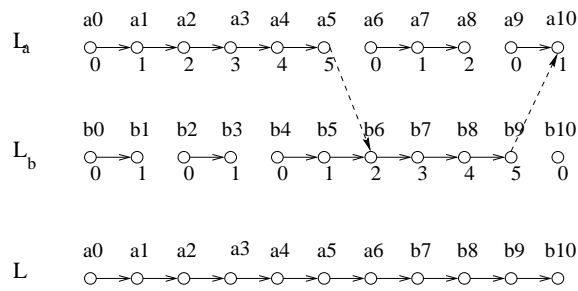


Figure 8: "lists stitching".