A Parallel Minimum Distance Error–Correcting Context–Free Parser

Giovanni Pighizzini *
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
Via Comelico, 39 – 20135 Milano – ITALY
pighizzini@hermes.mc.dsi.unimi.it

Abstract

In this paper we present the first parallel minimum distance error–correcting parser for context–free languages. The error correction is performed considering three kinds of syntax errors: the mutation of a symbol in another incorrect symbol, the insertion of an extraneous symbol and the deletion of a symbol.

Our algorithm has the same parallel complexity of other well–known non–correcting context–free parsers such as developed in [Ruz80, GR88]. Further, with respect to these algorithms it has the advantage of working with every context–free grammar and not just with grammars in Chomsky Normal Form.

The algorithm is developed using dynamic programming techniques.

1 Introduction

A main problem in compiler design is that of recovery from syntax errors. Usually, this problem is solved using an error–handling routine. This routine is called when an error is found during the analysis of an input string x and, under its control, the compiler skips the input characters following the character which has generated the error, until it find in the input string a character that, appended to the portion of x preceding the error, produces a new "legal" input.

This empirical approach presents many disadvantage: errors contained in the skipped part of the input are not found and often many more errors than actually exist are discovered by the parser after a recovering action.

In order to avoid this kind of problems, a different approach to the recovery has been proposed by several authors, (e.g., [AP72, Iro63, Lyo74]), introducing minimum distance criteria. Three kinds of syntax errors are considered: the replacement of a symbol with an incorrect symbol, the insertion of an extraneous symbol and the deletion of a symbol. Given an input string, a minimum distance error–correcting parser produces a representation, e.g., a parse tree, of a "legal" string which can be generated from the input with the minimum possible number (called *edit distance*) of syntax errors. We point out

*This work has been supported in part by the ESPRIT Basic Research Action No. 3166: "Algebraic and Syntactic Methods in Computer Science (ASMICS)" and by MURST.

that in this approach the parser is designed for minimizing edit distances between inputs and the given language; thus no specific error–handling routines are required.

Two sequential algorithms implementing error–correcting context–free parsers were independently discovered from Aho and Peterson [AP72] and from Lyon [Lyo74]. Even if both algorithms are based on the Earley's parser [Ear70], they are quite different. In [AP72], the grammar of the language is extended adding new productions representing syntax errors. In [Lyo74], an error counter is associated to every state of Earley's algorithm and the original grammar is not modified at all. Both algorithms are obtained using dynamic programming techniques.

In this paper, we afford the problem of finding a *parallel* minimum distance error–correcting parser for context–free languages. We are motivated in this research by the increasing role that parallel computation is going to assume in many areas of computer science and by recent results on the parallel parsing of context–free languages.

We briefly recall that the first parallel recognizer for context–free languages was proposed by Ruzzo [Ruz80]. This recognizer, however, as that presented in [GR88], works only for grammars in Chomsky Normal Form. This restriction makes hard to apply the algorithm in compiler design since grammars in Chomsky Normal Form do not reflect the semantic structure of languages (see [GHR80] for a discussion on disadvantage of grammars in Chomsky Normal Form). This problem has been recently solved by Bruschi and Pighizzini [BP92], that proposed the first parallel parser working with every context–free grammar.

In this paper we show how the algorithm presented in [BP92] can be further extend so that a parallel minimum distance error–correcting context–free parser can be obtained. The most important characteristics of such an algorithm are that it works with any context–free grammar, it has the same parallel complexity of non correcting algorithms given in [Ruz80, GR88, BP92], and, as the sequential algorithm contained in [Lyo74], it does not require any transformation on the grammar. The algorithm is obtained using dynamic programming techniques.

The paper is organized as follows. In Section 2, we introduce notions and notations used throughout the paper. In Section 3 we state the basis for our algorithm studying weighted recognition trees. The algorithm is presented in Section 4.

2 Preliminary definitions

Let Σ be a finite alphabet. On the monoid Σ^* of strings over Σ, we define the binary relation $\vdash$, representing the three kinds of syntax errors considered in this paper. For x and y in Σ^*, $x \vdash y$ if and only if there exist $v, w \in \Sigma^*$, $a, b \in \Sigma$ such that $x = vaw$, $y = vbw$ and $a \neq b$ (x is obtained changing a symbol of y), or $y = vaw$ and $x = vw$ (x is obtained deleting a symbol from y), or $y = vw$ and $x = vaw$ (x is obtained inserting an extraneous symbol in y). Observe that $\vdash$ is symmetric. By $\stackrel{k}{\vdash}$ we denote the composition of $\vdash$ with itself k times. Then $x \stackrel{k}{\vdash} y$ if and only if y can be obtained from x with k errors. The *edit distance* between two strings $x, y \in \Sigma^*$, denoted by $d(x, y)$, is defined as the minimum integer k such that $x \stackrel{k}{\vdash} y$.

For example given $\Sigma = \{a, b, +, *\}$, we have:

$$a + a * a \vdash aa + a * a \vdash aa + ba * a \vdash aa + bb * a \vdash aa + bb * .$$

Then $a + a * a \vdash^{4} aa + bb*$. Moreover, it is possible to verify that this is optimal, i.e., $d(aa + bb*, a + a * a) = 4$.

The following properties of edit distance can be easily verified.

- $\forall u, v, w, z \in \Sigma^*$: $d(uv, wz) \leq d(u, w) + d(v, z)$;
- $\forall x, y', y'' \in \Sigma^*\ \exists x', x'' \in \Sigma^*$ s.t. $x = x'x''$ and $d(x, y'y'') = d(x', y') + d(x'', y'')$.

The last property is related to the *principle of optimality* of dynamic programming [Bel57]: *in an optimal sequence of decisions or choices, each subsequence must also be optimal.* For a discussion of the use of dynamic programming in the realization of a sequential minimum distance error–correcting parser the reader is referred to [Lyo74].

The notion of edit distance can be extended to languages defining, for every nonempty language $L \subseteq \Sigma^*$ and for every string $x \in \Sigma^*$, the edit distance $d(x, L)$ between x and L as the minimum of the set of distances between x and strings in L, i.e., $d(x, L) = \min\{d(x, y) \mid y \in L\}$. We point out that $d(x, L)$ is at most $|x| + m$, where m denotes the length of the shortest string in L. In fact, we can assume that all $|x|$ input characters of x are insertions and all m characters of the shortest string of L have been deleted. From this fact and the trivial inequality $d(x, y) \geq |(|x| - |y|)|$, it turns out that if $y \in L$ is a string with minimal distance from x, i.e., $d(x, L) = d(x, y) = \min\{d(x, z) \mid z \in L\}$, then the length of y is at most linear in the length of x, i.e. $|y| = O(|x|)$.

A context–free grammar G will be denoted as (V, Σ, P, S), where V is the set of variables or nonterminal symbols, Σ is the set of terminal symbols, P is the finite set of productions, and S is the start symbol. We will suppose that the sets V and Σ are disjoint. We will assume that the language generated by G, denoted by $L(G)$ or by L when G is clear from the context, is not empty. Moreover, for sake of simplicity, we will assume that the grammar G does not contain useless nonterminal symbols. Throughout the paper, variables from V will be denoted by the capital letters $A, B, C, \ldots$, terminal symbols by the lower case letters $a, b, c, \ldots$, strings from $(V \cup \Sigma)^*$ by the greek letters $\alpha, \beta, \gamma, \ldots$. ϵ will denote the empty string. Given a production $(A \rightarrow \alpha\beta) \in P$, a *dotted rule* is a string $A \rightarrow \alpha \cdot \beta$, where $\cdot \notin V \cup \Sigma$. The only dotted rule corresponding to the ϵ–production $A \rightarrow \epsilon$ will be denoted as $A \rightarrow \cdot$.

A *minimum distance recognizer* is a procedure that computes the distance between the input string and $L(G)$. A *minimum distance error–correcting parser* is a recognizer which additionally outputs a parse tree of a string with minimal distance from the input. It is convenient to fix $n \geq 1$, and to fix the input string to be parsed as $x = x_1 \ldots x_n$, $x_i \in \Sigma$, $i = 1, \ldots, n$.

Our models of parallel computation are parallel random access machines with both concurrent read and concurrent write (CRCW PRAM), and with concurrent read and exclusive write (CREW PRAM) [KR90]. Such machines consist of several independent sequential processors, each with its own private memory, communicating with one another through a global memory. In one unit of time, each processor can read one global or local memory location, execute a single RAM instruction, and write into one global or local memory location. In CRCW PRAM simultaneous write are allowed, and write conflicts are solved assuming that all processors writing into the same location must write the same value (the COMMON model). In [SV84] it was proved that CRCW PRAM are equivalent to unbounded fan–in circuits.

3 Weighted recognition trees

In order to better understanding how the algorithm we will propose works and for introducing the notion of weighted recognition tree, we briefly present a strategy for recognizing context–free languages. This strategy is based on the Earley's algorithm. It is used by the parallel parser introduced in [BP92].

Given an input string $x \in \Sigma^*$ of length n, an *item* $\mathcal{I}$ is a triple (γ, i, j), where i, j are two integers, $0 \leq i \leq j \leq n$, delimiting the substring $x_{i+1} \dots x_j$ of x, and γ is either a terminal symbol or a dotted rule. In the former case the item $\mathcal{I}$ is said to be a *terminal item*, while in the latter case it is called *production item*.

The recognition of the string x is performed identifying *realizable items*. A terminal item (a, i, j) is said to be realizable if $0 < i + 1 = j \leq n$ and $x_{i+1} = a$, while a production item $(A \to \alpha \cdot \beta, i, j)$ is realizable if and only if $\alpha \overset{*}{\Rightarrow} x_{i+1} \dots x_j$. Intuitively, the dot in a realizable item indicates the portion of the right part of the production recognized so far. Then, the input is accepted if and only if there is a production $S \to \alpha$, for some $\alpha \in (V \cup \Sigma)^*$, such that the item $(S \to \alpha \cdot, 0, n)$ is realizable.

It is easy to observe that an item of the form $(A \to \alpha B \cdot \beta, i, j)$, $B \in V$, is realizable if and only if there is an index k, $i \leq k \leq j$ and a production $B \to \gamma$ such that both items $(A \to \alpha \cdot B\beta, i, k)$ and $(B \to \gamma \cdot, k, j)$ are realizable, while an item of the form $(A \to \alpha a \cdot \beta, i, j)$, $a \in \Sigma$, is realizable if and only if there exists an index k, $i \leq k \leq j$, such that both items $(A \to \alpha \cdot a\beta, i, k)$ and (a, k, j) are realizable.

From this discussion, it turns out that the membership of the string x to the language $L(G)$ can be witnessed by the existence of a binary tree whose nodes are labeled with realizable items. A tree of this kind, formally defined in [BP92], is called *recognition tree*.

For example, a recognition tree associated to the string $x = a + a * a \in L(G)$ for the grammar G, with variables E, T, P, terminal symbols $a, b, +, *$ and productions $E \to T$, $E \to E + T$, $T \to P$, $T \to T * P$, $P \to a$ is represented in the Fig. 1.

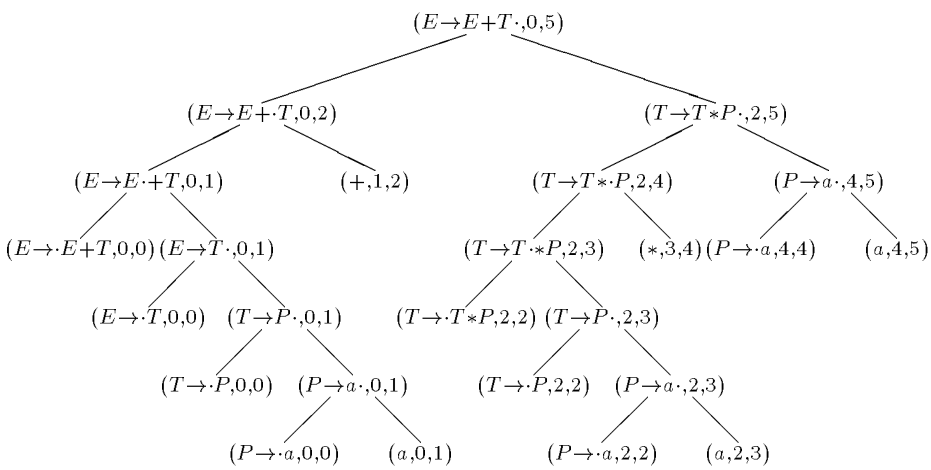

Figure 1: a recognition tree of the string $a + a * a$

We recall that while parse trees describe the syntactical structure of sentences of a language $L(G)$ with respect to the grammar G, recognition trees are related to the recognition process above sketched. For minimum distance recognition, we will consider the above defined set of items and we will associate weights to nodes of recognition trees.

Formally, we introduce the following notions.

Definition 3.1 *Given a context-free grammar $G = (V, \Sigma, P, S)$ and an input string $x = x_1 \dots x_n$, $x_i \in \Sigma$, a* partial recognition tree *T with respect to x is a binary tree, whose nodes are labeled with items, such that:*

1. *the label of a* leaf *is a terminal item or a production item of the form $(A \to \cdot\beta, i, j)$;*
2. *the label of an* internal node *is an item $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$, with $\alpha \neq \epsilon$, while the labels of its sons are respectively $\mathcal{J}$ and $\mathcal{K}$, where:*
   - *if $\alpha = \alpha' B$, with $B \in V$, $\alpha' \in (V \cup \Sigma)^*$, then $\mathcal{J} = (A \to \alpha' \cdot B\beta, i, k)$ and $\mathcal{K} = (B \to \gamma \cdot, k, j)$, for some k, $i \leq k \leq j$, and production $B \to \gamma$;*
   - *if $\alpha = \alpha' a$, with $a \in \Sigma$, $\alpha' \in (V \cup \Sigma)^*$, then $\mathcal{J} = (A \to \alpha' \cdot a\beta, i, k)$ and $\mathcal{K} = (a, k, j)$, for some k, $i \leq k \leq j$.*

The string of $L(G)$ generated *by the partial recognition tree T, denoted by $\phi(T)$, is the string obtained concatenating the first components of terminal items labeling leaves of T, from left to right. T will also said to be a partial recognition tree of the string $\phi(T)$ with respect to x.*
A recognition tree *with respect to x is a partial recognition tree whose root is labeled with an item $(S \to \alpha \cdot, 0, n)$, for some production $S \to \alpha$, where S denotes the initial symbol of G.*

It is not difficult to verify that if T is a partial recognition tree whose root is labeled with the item $(A \to \alpha \cdot \beta, i, j)$ then $\alpha \stackrel{*}{\Rightarrow} \phi(T)$.

In associating a weight to a recognition tree, we will compare every item (γ, i, j) and the string generated by the corresponding subtree, with the input $x_{i+1} \dots x_j$. More precisely, for each terminal item (a, i, j), the terminal a is compared with the string $x_{i+1} \dots x_j$. If such a string is empty, i.e., $i = j$, then a deletion occurs in x; otherwise the string $x_{i+1} \dots x_j$ contains $j - i + 1$ insertions with respect to the terminal a and, if $a \notin \{x_{i+1}, \dots, x_j\}$ then it contains also one mutation. For every production item $\mathcal{I}$ of the form $(A \to \alpha \cdot \beta, i, j)$, a terminal string generated by α is compared with the input $x_{i+1} \dots x_j$. When α is empty, i.e., when the item $\mathcal{I}$ marks a leaf of the tree, there are $j - i$ insertions. When α is not empty, the weight is the sum of weights associated to sons of the node labeled $\mathcal{I}$.

Formally, we get the following definition:

Definition 3.2 *Given a partial recognition tree T, the* weight *of T, denoted by ω_T, is the function associating to every node v of T an integer number, so defined:*

- *if v is a leaf labeled with a terminal item (a, i, j) then:*

$$\omega_T(v) = \begin{cases} 1 & \text{if } i = j \\ j - i & \text{if } i < j \text{ and } a \notin \{x_{i+1}, \dots, x_j\} \\ j - i - 1 & \text{if } i < j \text{ and } a \in \{x_{i+1}, \dots, x_j\}; \end{cases}$$

- *if v is a leaf labeled with a production item $(A \to \cdot\beta, i, j)$ then $\omega_T(v) = j - i$;*

- *if v is an internal node with sons v' and v'' then $\omega_T(v) = \omega_T(v') + \omega_T(v'')$.*

The weight of T, denoted as ω_T, is the weight of its root.

Using a simple induction, it is not difficult to verify that for every node v labeled with a production item $(A \to \alpha \cdot \beta, i, j)$, it holds $\phi(T_v) \vdash^{\omega_T(v)} x_{i+1} \dots x_j$, where T_v denotes the subtree of T rooted at v and $\alpha \stackrel{*}{\Rightarrow} \phi(T_v)$.

Figure 2 represents a recognition tree T, for the grammar of previous example, with respect to the input string $x = aa{+}bb{*}$. The string generated by this tree is $\phi(T) = a{+}a{*}a$. The number after every item denotes its weight.

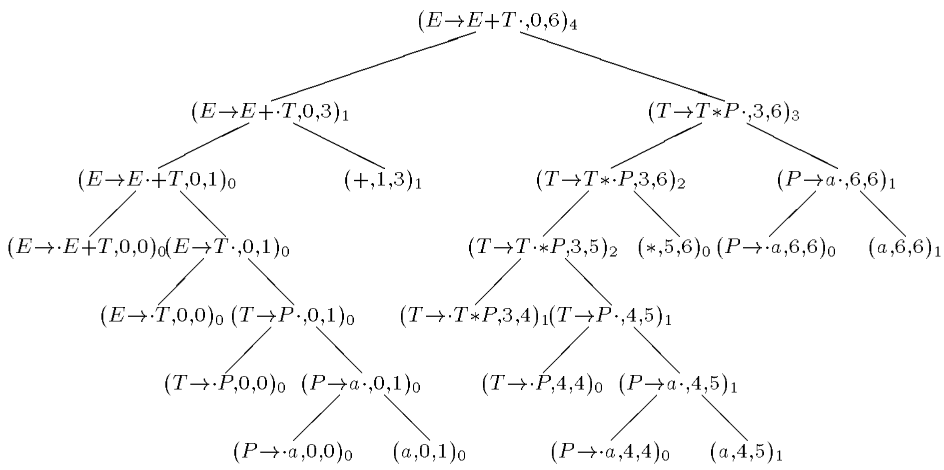

Figure 2: a recognition tree of the string $a + a * a$ with respect to $aa + bb*$

The following theorem states that for every input x, there exists a recognition tree T whose weight equals $d(x, L)$ and whose size is linearly bounded by the length of x.

Theorem 3.1 *Given a context-free grammar $G = (V, \Sigma, P, S)$, and a string $x = x_1 \dots x_n$, $x_i \in \Sigma$, $i = 1, \dots, n$, for every production item $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$ there exists a recognition tree T with respect to x whose weight is $d(x_{i+1} \dots x_j, L_\alpha)$, where $L_\alpha = \{y \in \Sigma^* \mid \alpha \stackrel{*}{\Rightarrow} y\}$, whose number of leaves is $O(n)$ and s.t. $\phi(T)$ is a string in L_α with minimal distance from $x_{i+1} \dots x_j$, i.e., $d(x_{i+1} \dots x_j, L_\alpha) = d(x_{i+1} \dots x_j, \phi(T))$.*

Proof. (outline) For every terminal item $\mathcal{I} = (a, i, j)$, let be $y_\mathcal{I} = a$, while for every production item $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$, let $y_\mathcal{I}$ be a string of terminal symbols generated by α with minimal distance form $x_{i+1} \dots x_j$. We define $rank(\mathcal{I}) = \tau(y_\mathcal{I}) + |y_\mathcal{I}| + 1$, where $\tau(y_\mathcal{I})$ denotes the length of a shortest derivation $\alpha \stackrel{*}{\Rightarrow} y_\mathcal{I}$.

By induction on $rank(\mathcal{I})$, it can be proved that there exists a partial recognition tree $T_\mathcal{I}$ with respect to x, whose root is labeled $\mathcal{I}$, s.t. $\phi(T_\mathcal{I}) = y_\mathcal{I}$, whose number of leaves is at most $rank(\mathcal{I})$ and whose weight $\omega_{T_\mathcal{I}}$ equals $d(x_{i+1} \dots x_j, L_\alpha)$ when $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$ and $d(x_{i+1} \dots x_j, a)$ when $\mathcal{I} = (a, i, j)$. Since $\tau(y_\mathcal{I}) = O(|y_\mathcal{I}|)$ [HU79] and $|y_\mathcal{I}| = O(j - i)$, using the bounds obtained in Section 2, we can conclude that $T_\mathcal{I}$ is at most $O(j - i)$. ∎

We define the weight $\omega(\mathcal{I})$ of an item $\mathcal{I}$ (with respect to x), as:

$$\omega(\mathcal{I}) = \min\{\omega_T(v) \mid v \text{ is a node of the recognition tree } T \text{ whose label is } \mathcal{I}\}.$$

As a consequence of Theorem 3.1, it turns out that for every production item $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$ it holds $\omega(\mathcal{I}) = d(x_{i+1} \dots x_j, \{y \in \Sigma^* \mid \alpha \stackrel{*}{\Rightarrow} y\})$.

We point out that for every node v of the tree T obtained from Theorem 3.1 it holds $\omega_T(v) = \omega(\mathcal{I})$, where $\mathcal{I}$ is the item labeling the node v. Then, every subtree of T is optimal (i.e., of minimal weight) for the item labeling its root.

Using Theorem 3.1 and the principle of optimality of dynamic programming, it is not difficult to see that the weight of a production item $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$, $\alpha \neq \epsilon$, is equal to the minimum sum $\omega(\mathcal{I}) + \omega(\mathcal{J})$, where $\mathcal{J}, \mathcal{K}$ are the labels of a pair of possible sons of a node labeled $\mathcal{I}$ in some recognition tree.

4 The parallel minimum distance recognizer

In this section we present our parallel minimum distance recognizer, we prove its correctness and we evaluate its complexity.

The parallel algorithm is obtained using a generalization of the technique presented in [Ryt88] for the efficient solution of some dynamic programming problems. For more details the reader is referred to that paper.

The following brief example illustrates the main idea of this technique. For computing the weight of the node $v_{\mathcal{I}}$ labeled with the item $\mathcal{I} = (T \to T * P\cdot, 3, 6)$ in the tree represented in Fig. 1, we have to consider six different levels of nodes. If in every step, we evaluate all nodes whose sons have been evaluated, we need of five parallel steps. We can reduce the number of these steps considering partial trees: we split the tree with root $v_{\mathcal{I}}$ in the trees T_1 and T_2 represented in Fig. 3, and we compute in parallel the weights of nodes of T_1 and T_2, assigning weight 0 to the leaf $v_{\mathcal{J}}$ of T_2 whose label is the item $\mathcal{J} = (T \to T \cdot *P, 3, 5)$. So, after three steps, the weight $\omega_{T_1}(v)$ of every node v of T_1 and the *partial weight* $\rho\omega_{T_2}(v, v_{\mathcal{J}})$ of every node v of T_2 are computed. Of course, the weight of a node v of T_2 is $\omega_{T_2}(v) = \rho\omega_{T_2}(v, v_{\mathcal{J}}) + \omega_{T_1}(v_{\mathcal{J}})$.

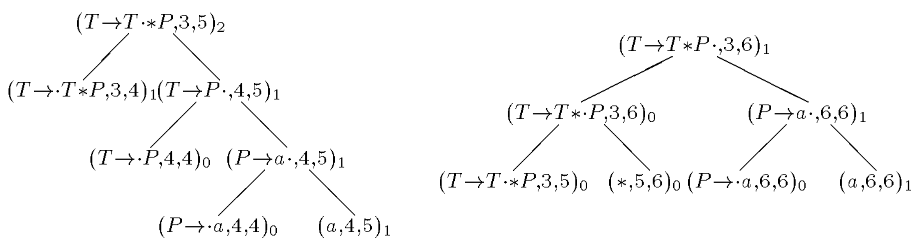

Figure 3: trees T_1 and T_2

The tree T_2 is called *tree with gap* and the node $v_{\mathcal{J}}$ is called *gap*. The technique we will use is based on the parallel evaluation of trees with gap.

We denote by $\rho\omega(\mathcal{I}, \mathcal{J})$ the minimal weight of a tree with root labeled $\mathcal{I}$ and gap labeled $\mathcal{J}$. If such a tree does not exist, e.g., if $\mathcal{I} = (A \rightarrow \alpha \cdot \beta, i, j)$, $\mathcal{J} = (C \rightarrow \gamma \cdot \delta, k, h)$ and $i < k < j < h$, we set $\rho\omega(\mathcal{I}, \mathcal{J}) = \infty$. Then, it is possible to prove the following relationships:

1. for every item $\mathcal{I}$ and for every item $\mathcal{J}$ such that $\mathcal{J}$ can be the label of a son of a node with label $\mathcal{I}$: $\rho\omega(\mathcal{I}, \mathcal{J}) = \min_{\mathcal{K}} \omega(\mathcal{K})$, where the minimum is computed on the set of all items $\mathcal{K}$ such that $(\mathcal{J}, \mathcal{K})$ or $(\mathcal{K}, \mathcal{J})$ is a pair of possible labels of sons of a node labeled $\mathcal{I}$ in some recognition tree.
2. for all $\mathcal{I}, \mathcal{J}$: $\rho\omega(\mathcal{I}, \mathcal{J}) = \min_{\mathcal{K}} (\rho\omega(\mathcal{I}, \mathcal{K}) + \rho\omega(\mathcal{K}, \mathcal{J}))$;
3. for all $\mathcal{I}$: $\rho\omega(\mathcal{I}) = \min_{\mathcal{J}} (\rho\omega(\mathcal{I}, \mathcal{J}) + \omega(\mathcal{J}))$.

The last three relationships correspond to fundamentals operations *(activateL;activateR)*, *square1*, and *pebble1* of our algorithm. The algorithm uses two arrays of integers w and pw, containing approximations of ω and $\rho\omega$. At the start of the computation all components of these arrays contain the value ∞.

Algorithm 1
begin
(1) *initialize1*
(2) *initialize2*
repeat $O(\log n)$ **times**
    begin
(3)   *activateL;activateR*
(4)   *square1*
(5)   *pebble1*
    end
    output := $\min \{w(S \rightarrow \alpha\cdot, 0, n) \mid (S \rightarrow \alpha) \text{ is a production}\}$
end

The operations contained in the algorithm are formally defined in the following way.

- *initialize1*
  for all $(A \rightarrow \cdot\beta, i, j)$ **in parallel do** $w(A \rightarrow \cdot\beta, i, j) := j - i$
- *initialize2*
  for all $(A \rightarrow \alpha a \cdot \beta, i, j)$ **and for** all k, $i \leq k \leq j$ **in parallel do**
  $$pw((A \rightarrow \alpha a \cdot \beta, i, j), (A \rightarrow \alpha \cdot a\beta, i, k)) := \begin{cases} 1 & \textbf{if } k = j \\ j - k & \textbf{if } k < j \textbf{ and} \\ & \quad a \notin \{x_{k+1}, \ldots, x_j\} \\ j - k - 1 & \textbf{if } k < j \textbf{ and} \\ & \quad a \in \{x_{k+1}, \ldots, x_j\} \end{cases}$$
- *activateL*
  for all $(A \rightarrow \alpha \cdot B\beta, i, k)$ **and for** all $(B \rightarrow \gamma\cdot, k, j)$ **in parallel do**
  $pw((A \rightarrow \alpha B \cdot \beta, i, j), (B \rightarrow \gamma\cdot, k, j)) := w(A \rightarrow \alpha \cdot B\beta, i, k)$

- *activateR*
  for all $(A \to \alpha \cdot B\beta, i, k)$ **and for** all j, $k \le j \le n$ **in parallel do**
  $$pw((A \to \alpha B \cdot \beta, i, j), (A \to \alpha \cdot B\beta, i, k)) := \min_{(B \to \gamma) \in P} w(B \to \gamma \cdot, k, j)$$

- *square1*
  for all items $\mathcal{I}, \mathcal{J}$ **in parallel do** $pw(\mathcal{I}, \mathcal{J}) := \min_{\mathcal{K}} (pw(\mathcal{I}, \mathcal{K}) + pw(\mathcal{K}, \mathcal{J}))$

- *pebble1*
  for all items $\mathcal{I}, \mathcal{J}$ **in parallel do** $w(\mathcal{I}) := \min_{\mathcal{J}} (pw(\mathcal{I}, \mathcal{J}) + w(\mathcal{J}))$

Now, we prove the correctness of Algorithm 1.

Theorem 4.1 *The output of Algorithm 1 is the edit distance $d(x, L)$ between the input string x and the language L.*

Proof. (outline) First, as an immediate consequence of the following lemma, whose proof is omitted, we can prove that $d(x, L) \le \min\{w(S \to \alpha \cdot, 0, n) \mid (S \to \alpha) \in P\}$.

Lemma 4.1 *For all items $\mathcal{I} = (A \to \alpha \cdot \beta, i, j)$, $\mathcal{J} = (C \to \gamma \cdot \delta, p, q)$, after every step of Algorithm 1 the following assertions hold:*

(i) $pw(\mathcal{I}, \mathcal{J}) < \infty$ implies $i \le p \le q \le j$ and $\alpha \stackrel{*}{\Rightarrow} y\gamma z$ for some strings $y, z \in \Sigma^*$ such that $y \stackrel{m_1}{\vdash} x_{i+1} \ldots x_p$, $z \stackrel{m_2}{\vdash} x_{q+1} \ldots x_j$ and $m_1 + m_2 = pw(\mathcal{I}, \mathcal{J})$;

(ii) $w(\mathcal{I}) < \infty$ implies $\alpha \stackrel{*}{\Rightarrow} y$, where $y \in \Sigma^*$ and $y \stackrel{w(\mathcal{I})}{\vdash} x_{i+1} \ldots x_j$.

To complete the proof, we have to show that there is an item of the form $(S \to \alpha \cdot, 0, n)$ such that at the end of the computation $w(S \to \alpha \cdot, 0, n) = d(x, L)$. The proof of this fact can be obtained using a *pebble game* on binary trees introduced in [Ryt85], that we briefly recall. Given a binary tree T, the game uses an array *cond* of nodes. At the start of the computation $cond(v) = v$ for every node and only the leaves of T are pebbled. The game uses the following operations:

- *activate*
  for all non–leaf nodes v (with sons v', v'') **in parallel do**
  begin
  if $cond(v) = v$ **and** v' is pebbled **then** $cond(v) := v''$
  if $cond(v) = v$ **and** v'' is pebbled **then** $cond(v) := v'$
  end

- *square*
  for all nodes v **in parallel do** $cond(v) := cond(cond(v))$

- *pebble*
  for all nodes v **in parallel do if** $cond(v)$ is pebbled **then** pebble v

In [Ryt85] it is proved that after $O(\log n)$ iterations of the sequence (*activate*, *square*, *pebble*), the root of T becomes pebbled.

Given a string $x \in \Sigma^*$, let $y \in L$ be such that $d(x,y) = d(x,L)$. By Theorem 3.1, there exists a recognition tree T with an most $O(n)$ leaves, and $\phi(T) = y$. We substitute in Algorithm 1 instruction (3), (4) and (5) with the following: (3') *activateL; activateR; activate*, (4') *square1; square*, (5') *pebble1; pebble*, where *activate, square*, and *pebble* works on the tree T. Then, it is possible to prove that after the execution of every instruction (3'), (4'), (5'), for all items $\mathcal{I}, \mathcal{J}$, the following invariant holds:

(if $\mathcal{I}$ is pebbled then $w(\mathcal{I}) = \omega(\mathcal{I})$) and if ($cond(\mathcal{I}) = \mathcal{J} \neq \mathcal{I}$ then $pw(\mathcal{I},\mathcal{J}) = \rho\omega(\mathcal{I},\mathcal{J})$).

Since after $O(\log n)$ iterations of $(activate, square, pebble)$ the root of T is pebbled, using previous invariant it turns out that after $O(\log n)$ iterations Algorithm 1 outputs the edit distance between the input x and the language L. ∎

Now, we study the parallel complexity of Algorithm 1.

Theorem 4.2 *Algorithm 1 computes $d(x,L)$, $|x| = n$, in time $O(\log n)$ using $O(n^8)$ processors on a CRCW PRAM and in time $O(\log^2 n)$ using $O(n^6/\log n)$ processors on a CREW PRAM.*

Proof. It is immediate to observe that the more expensive step is $square1$, where in parallel $O(n^4)$ pairs of items are considered. For each pair we have to compute the minimum among $O(n^2)$ elements. This computation requires time $O(1)$ and $O(n^4)$ processors in the CRCW model, while it requires time $O(\log n)$ and $O(n^2/\log n)$ processors in the CREW model [SV81, KR90]. ∎

5 Conclusion

In previous sections we have shown how to build a parallel minimum distance context–free recognizer. To obtain a parser from this recognizer, it is possible to use the same technique proposed in [BP92]. It consists in two phases. In the first phase, using the array w computed by Algorithm 1, a recognition tree of a string $y \in L(G)$ with minimal distance from the input x is recovered. The strategy used in this phase is the same of [Ryt86].

In the second phase, the recognition tree is transformed in a derivation tree deleting all nodes labeled with items of the form $(A \to \alpha \cdot \beta, i, j)$, with $\beta \neq \epsilon$, rearranging the pointers in a suitable way and substituting every label of the form $(A \to \alpha \cdot \beta, i, j)$ with the label A and every label of the form (a, i, j) with the label a.

The above outlined construction of the parse tree from the array w computed by the recognizer requires time $O(\log n)$ and $O(n^6)$ processors in the CRCW model. Then, for this model, the parser works in time $O(\log n)$ with $O(n^8)$ processors. Moreover, from simulation results between different variant of PRAMs [KR90], it turns out that the complexity of the parser, for the CREW model is $O(\log^2 n)$ in time and $O(n^6)$ in number of processors.

In a recent paper [LR92], a new technique for obtaining parallel algorithms for dynamic programming was introduced. This technique permits of reducing, with respect to techniques used in this paper, the total work, that is the product of the time for the number of processors. However, the algorithms obtained applying it are in general only sublinear and not polylogarithmic. Then, they are not "fast" in the usual sense. Thus, this new approach is quite different from the classical one. It should not be difficult to obtain, applying the technique of [LR92], a different version of our minimum distance recognizer with a "small" total work.

Acknowledgment I am indebted with Danilo Bruschi who suggested me this problem.

References

[AP72] A. Aho and T. Peterson, "A minimum distance error–correcting parser for context–free languages," *SIAM J* **1**(4): 305–312, 1972.

[Bel57] R. Bellman, *Dynamic Programming,* Princeton Univ. Press, 1957.

[BP92] D. Bruschi and G. Pighizzini, "A parallel general context–free parser", Tech. Report, Univ. of Milan, 1992.

[Ear70] J. Earley, "An efficient context–free parsing algorithm," *Comm. ACM* **13**(2): 94–102, 1970.

[GHR80] S. Graham, M. Harrison and W. Ruzzo,"An improved context–free recognizer," *ACM Transactions* **2**(3) : 415–462, 1980.

[GR88] A. Gibbons and W. Rytter, *Efficient parallel algorithms,* Cambridge Univ. Press, 1988.

[HU79] J. Hopcroft and J. Ullman, *Introduction to automata theory, languages and computation,* Addison–Wesley, 1979.

[Iro63] E. Irons, "An error–correcting parse algorithm," *Comm. ACM* **6**(11): 669–673, 1963.

[KR90] R. Karp and V. Ramachandran, "Parallel algorithms for shared–memory machines," in *Handbook of Theoretical Computer Science,* J.V. Leeuwen ed., Elsevier, 1990.

[LR92] L. Larmore and W. Rytter, "Efficient sublinear time parallel algorithms for dynamic programming and context–free recognition," *Proc. STACS 92*, Lecture Notes in Computer Science 577, pp. 121–132, 1992.

[Lyo74] G. Lyon, "Syntaxt–directed least–errors analysis for context–free languages; a practical approach," *Comm. ACM* **17**(1): 3–14, 1974.

[Ruz80] W. Ruzzo, "Tree–size bounded alternation," *JCSS* **21**: 218–235, 1980.

[Ryt85] W. Rytter, "The complexity of two way pushdown automata and recursive programs,"in *Combinatorial Algorithms on Words*, A. Apostolico and Z. Galil, ed.s, NATO ASI Series F:12, Springer, 1985.

[Ryt86] W. Rytter, "On the complexity of parallel parsing of general context–free languages," *TCS* **47**: 315–321, 1986.

[Ryt88] W. Rytter, "On efficient parallel computations for dynamic programming problems," *TCS* **59**: 297–307, 1988.

[SV81] Y. Shiloach and U. Vishkin, "Finding the maximum, merging and sorting in a parallel computation model," *J. Algorithms* **2**:88–102,1981.

[SV84] L. Stockmeyer and U. Vishkin, "Simulation of parallel random access machines by circuits," *SIAM J* **13**: 409–422, 1984.