SLOW AND FAST PARALLEL RECOGNITION

Hans de Vreught, Job Honig*
Delft University of Technology
Faculty of Technical Mathematics and Informatics
Section Theoretical Computer Science
Julianalaan 132, 2628 BL Delft, The Netherlands
e-mail: hdev@dutiae.tudelft.nl, joho@dutiae.tudelft.nl

ABSTRACT

In the first part of this paper a slow parallel recognizer is described for general CFG's. The recognizer runs in $\Theta(n^3/p(n))$ time with $p(n) = O(n^2)$ processors. It generalizes the items of the Earley algorithm to double dotted items, which are more suited to parallel parsing. In the second part a fast parallel recognizer is given for general CFG's. The recognizer runs in $O(\log\ n)$ time using $\Theta(n^6)$ processors. It is a generalisation of the Gibbons and Rytter algorithm for grammars in CNF.

1 INTRODUCTION

The subject of context-free parsing is well studied, e.g. see (Aho and Ullman, 1972; 1973; Harrison, 1978). Nowadays, research on the subject has shifted to parallel context-free parsing (op den Akker, Alblas, Nijholt, and Oude Luttighuis, 1989). Two areas of interest can be distinguished: slow and fast parallel parsing. We call a parallel algorithm fast when it does its job in polylogarithmic time. This is in contrast to the sequential case, in which algorithms are called fast when they run in polynomial time. Obtaining a fast parallel algorithm is often quite simple: when the fast sequential algorithm is highly parallelizable, using an exponential number of processors is sufficient. This is not very realistic, however.

A parallel algorithm is called feasible only when it uses a polynomial number of processors. Note that when a feasible slow parallel algorithm runs in polynomial time, it can be simulated by a fast sequential algorithm. Therefore in practice we often see that slow parallel is fast enough; fast parallel algorithms often achieve their speed because of their huge number of processors and large amounts of storage.

Several authors have studied algorithms for slow parallel recognition. Most of these algorithms are variants of the Cocke-Younger-Kasami (CYK) algorithm and the Earley algorithm. In the first part of this paper another slow parallel recognizer is given (de Vreught and Honig, 1989; 1990b). Its new feature is that it uses double dotted items, which are more natural for parallel parsing; these items make it easy to do error determination, a feature that is shared with most parallel bottom up algorithms. Although there are some similarities between the three algorithms, they should not be regarded as variants of each other since they all fill their respective matrices with different 'items' and for entirely different reasons.

When compared to a parallel version of the Earley algorithm, which would have to be bottom up, our algorithm generates far less items on the principal diagonal of the recognition matrix. A detailed comparison of the items required by the given algorithm and the Earley algorithm will be necessary to show the strengths or weaknesses of both approaches to parallel parsing. The sizes of the item sets in relation to particular classes of grammars is still under research.

The subject of fast parallel parsing is relatively new. Amongst the first to give a fast parallel recognizer were Gibbons and Rytter (1988). Their recognizer requires a grammar in CNF; it can be regarded as the fast par-

*Using initials: J.P.M. de Vreught and H.J. Honig.

allel version of the slow parallel CYK algorithm. The speeded up version is obtained by also examining the consequences of incomplete items. When an incomplete item gets completed, we can also complete the consequences immediately. The reason for the algorithm being fast is based on the fact that for every skewed tree (with n internal nodes) of height $O(n)$ describing the composition of a certain item, there exists a reasonably well balanced one of height $O(\log n)$ that uses both complete and incomplete items.

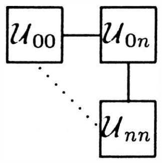
In the second part of the paper a fast parallel recognizer for general CFG's is given (de Vreught and Honig, 1990a). In spite of the fact that any CFG can be transformed into CNF in $O(1)$ time, using CNF is undesirable in practice (especially in natural language processing). The fast parallel recognizer does not need to transform the grammar. The fast parallel recognizer can be regarded as the fast parallel version of the slow parallel recognizer described in the first part. The fast parallel recognizer is based on the Gibbons and Rytter algorithm for grammars in CNF (Gibbons and Rytter, 1988). The paper is concluded with some final remarks.

2 THE SLOW PARALLEL RECOGNIZER

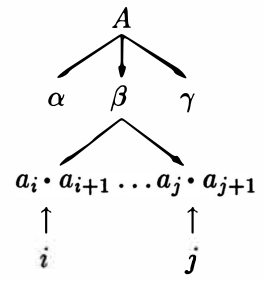
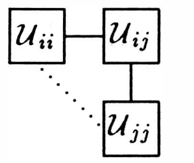
We start by sketching the ideas behind the slow parallel recognizer. Then we will give an inductive relation which plays a central role in our algorithms. Finally we will present the slow parallel recognizer.

2.1 INFORMAL DESCRIPTION

Let $a_1 \ldots a_n$ be the string to be recognized. We are going to build an upper triangular matrix $\mathcal{U}$ as shown below.

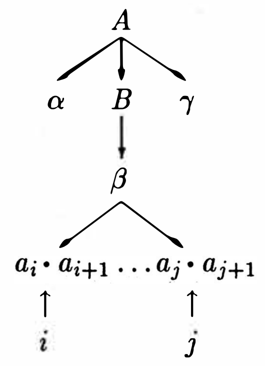
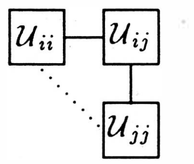


In each cell $\mathcal{U}_{ij}$ we enter items of the form $A \rightarrow \alpha\bullet\beta\bullet\gamma$ such that $A \rightarrow \alpha\beta\gamma$ is a production and $\beta \Rightarrow^* a_{i+1} \ldots a_j$. We will also insist that if $\beta = \lambda$ then $\alpha\gamma = \lambda$:

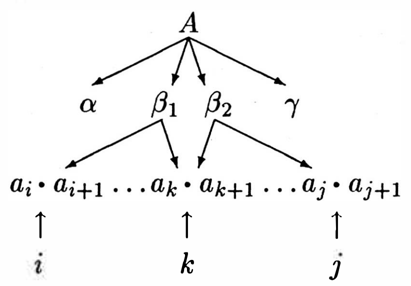
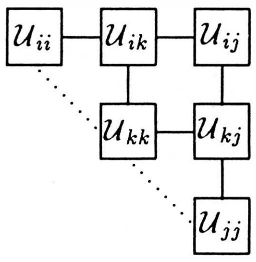
$A \rightarrow \alpha\bullet\beta\bullet\gamma \in \mathcal{U}_{ij}$

Suppose $B \rightarrow \bullet\beta\bullet \in \mathcal{U}_{ij}$ and let $A \rightarrow \alpha B\gamma$ be a production. In that case we can assert that $A \rightarrow \alpha\bullet B\bullet\gamma \in \mathcal{U}_{ij}$:

$A \rightarrow \alpha\bullet B\bullet\gamma \in \mathcal{U}_{ij}$

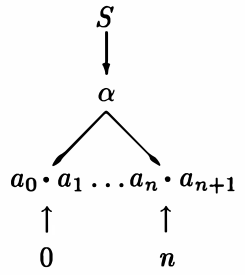
This assertion follows from the application of the inclusion operation to $B \rightarrow \bullet\beta\bullet \in \mathcal{U}_{ij}$. Another operation is concatenation. If $A \rightarrow \alpha\bullet\beta_1\bullet\beta_2\gamma \in \mathcal{U}_{ik}$ and $A \rightarrow \alpha\beta_1\bullet\beta_2\bullet\gamma \in \mathcal{U}_{kj}$ then we can assert that $A \rightarrow \alpha\bullet\beta_1\beta_2\bullet\gamma \in \mathcal{U}_{ij}$, applying the concatenation operation:

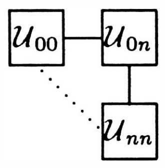



$A \rightarrow \alpha\bullet\beta_1\beta_2\bullet\gamma \in \mathcal{U}_{ij}$

When the entries of matrix $\mathcal{U}$ are closed with respect to the operations, we look for an item

$S \rightarrow \cdot\alpha\cdot \in \mathcal{U}_{0n}$ where S is the start symbol of the grammar:





$S \rightarrow \cdot\alpha\cdot \in \mathcal{U}_{0n}$

In the following, we will give a relation **U** defining the item sets constructed during the recognition process. We do this by identifying the matrix $\mathcal{U}$ with the relation **U** such that $A \rightarrow \alpha\cdot\beta\cdot\gamma \in \mathcal{U}_{ij}$ iff $(i,j,A \rightarrow \alpha\cdot\beta\cdot\gamma) \in \mathbf{U}$.

2.2 THE RELATION

Let $G = (V, \Sigma, P, S)$ be the CFG in question and let $x = a_1 \ldots a_n$ the string to be recognized. Furthermore, let $\cdot \notin V$ and let λ be the empty string. Finally, let $\mathcal{J} = \{0, \ldots, n\}^2 \times \{A \rightarrow \alpha\cdot\beta\cdot\gamma \mid A \rightarrow \alpha\beta\gamma \in P\}$.

Definition 2.2.1 $\mathbf{U} = \{(i,j,A \rightarrow \alpha\cdot\beta\cdot\gamma) \in \mathcal{J} \mid A \Rightarrow \alpha\beta\gamma$ and $\beta \Rightarrow^* a_{i+1} \ldots a_j$ and if $\beta = \lambda$ then $\alpha\gamma = \lambda\}$

In (de Vreught and Honig, 1989) some variants of **U** are examined; for instance, one of them takes context into account. The disadvantage of definition 2.2.1 is that it is not immediately clear how to determine whether or not an item is in the relation. For this purpose we need an inductive definition.

Definition 2.2.2 The relation $\mathbf{U}'$ over $\mathcal{J}$ is defined as follows:

- If $A \rightarrow \lambda \in P$ then $(j,j,A \rightarrow \cdot\cdot) \in \mathbf{U}'$ for any $j \in \{0, \ldots, n\}$.
  This item is a base item.
- If $A \rightarrow \alpha a_j \gamma \in P$ then $(j-1,j,A \rightarrow \alpha\cdot a_j\cdot\gamma) \in \mathbf{U}'$ for any $j \in \{1, \ldots, n\}$.
  This item is a base item.
- If $A \rightarrow \alpha B \gamma \in P$ and $(i,j,B \rightarrow \cdot\beta\cdot) \in \mathbf{U}'$ then $(i,j,A \rightarrow \alpha\cdot B\cdot\gamma) \in \mathbf{U}'$.
  This operation is called inclusion.
- If $(i,k,A \rightarrow \alpha\cdot\beta_1\cdot\beta_2\gamma) \in \mathbf{U}'$ and $(k,j,A \rightarrow \alpha\beta_1\cdot\beta_2\cdot\gamma) \in \mathbf{U}'$ then $(i,j,A \rightarrow \alpha\cdot\beta_1\beta_2\cdot\gamma) \in \mathbf{U}'$.
  This operation is called concatenation.
- Nothing is in $\mathbf{U}'$ except those elements which must be in $\mathbf{U}'$ by applying the preceding rules finitely often.

It can be proved that $\mathbf{U} = \mathbf{U}'$.

2.3 THE RECOGNIZER

We will now present the recognition algorithm (de Vreught and Honig, 1989). In the algorithm **mode** is either **sequence** or **parallel**.

```
Recognizer($n$):
    for $i := 0$ to $n$ do
        for $j := 0$ to $n - i$ in parallel do
            case $i$
                $= 0$:
                    Empty($j + i$)
                $= 1$:
                    Symbol($j + i$)
                $> 1$:
                    $\mathcal{U}_{j,j+i} := \emptyset$
                    for $k := 1$ to $i - 1$
                    in mode do
                        Concatenator
                            ($j, j + k, j + i$)
            while $\mathcal{U}_{j,j+i}$ still changes do
                Concatenator($j, j, j + i$)
                Concatenator($j, j + i, j + i$)
                Includer($j, j + i$)
    return Test($n$)

Empty($j$):
    $\mathcal{U}_{jj} := \{A \rightarrow \cdot\cdot \mid A \rightarrow \lambda \in P\}$

Symbol($j$):
    $\mathcal{U}_{j-1,j} := \{A \rightarrow \alpha\cdot a_j\cdot\gamma \mid A \rightarrow \alpha a_j \gamma \in P\}$

Includer($i, j$):
    for all $B \rightarrow \cdot\beta\cdot \in \mathcal{U}_{ij}$ do
        $\mathcal{U}_{ij} := \mathcal{U}_{ij} \cup$
            $\{A \rightarrow \alpha\cdot B\cdot\gamma \mid A \rightarrow \alpha B \gamma \in P\}$

Concatenator($i, k, j$):
    for all $A \rightarrow \alpha\cdot\beta_1\cdot\beta_2\gamma \in \mathcal{U}_{ik}$
    with $|\beta_1| = 1$ do
        for all $A \rightarrow \alpha\beta_1\cdot\beta_2\cdot\gamma \in \mathcal{U}_{kj}$ do
            $\mathcal{U}_{ij} := \mathcal{U}_{ij} \cup \{A \rightarrow \alpha\cdot\beta_1\beta_2\cdot\gamma\}$
```

```
Test($n$):
    accept := False
    for all $S \rightarrow \alpha \in P$ do
        if $S \rightarrow \bullet\alpha\bullet \in \mathcal{U}_{0n}$ then
            accept := True
    return accept
```

Although the algorithm fills the matrix diagonal by diagonal, there are many other filling orders for the matrix (de Vreught and Honig, 1989). Note that all cells on a diagonal can be filled independently of each other. When **mode** = **sequence**, it can be shown that a CREW-PRAM (Concurrent Read Exclusive Write - Parallel RAM) (Quinn, 1987) with $p(n) = O(n)$ processors can fill the matrix in $T(n) = \Theta(n^3/p(n))$ time.

The concatenations done in the loop over k in Recognizer can also be done independently of each other. However, in that case the architecture must allow parallel writing in cell $\mathcal{U}_{j,j+i}$. Thus when **mode** = **parallel**, it can be shown that a CRCW-PRAM (Concurrent Read Concurrent Write - Parallel RAM) (Quinn, 1987) with $p(n) = O(n^2)$ processors can fill the matrix in $T(n) = \Theta(n^3/p(n))$ time. In both cases the space complexity is dominated by the matrix: $S(n) = O(n^2)$ space.

Example 2.3.1 Consider the string *aabcc* and the CFG $G = (V, \Sigma, P, A)$:

- $V = \{A, B\} \cup \Sigma$
- $\Sigma = \{a, b, c\}$
- P contains the following productions:
  - $A \rightarrow aB$
  - $B \rightarrow Acc$
  - $B \rightarrow b$

Notice that G is λ-free (this simplifies the example). From the given grammar and string, the following matrix (see figure 1) can be obtained.

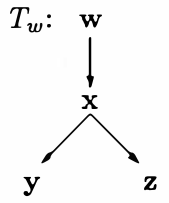
3 THE FAST PARALLEL RECOGNIZER

In this section we will sketch the ideas behind the fast algorithm. The proof that the recognizer is fast uses a pebble game, described in (Gibbons and Rytter, 1988), and critically depends on the fact that the 'minimal composition trees' are linear in size (with respect to the length of the string to be recognized). Instead of determining **U** directly we will compute its extension $\tilde{\mathbf{U}}$, on which the fast parallel recognizer is based. Finally we will describe the recognizer for a general CFG. The algorithms is based on the fast parallel Gibbons and Rytter recognizer for CFG's in CNF (Gibbons and Rytter, 1988).

3.1 COMPOSITION TREES

Definition 2.2.2 offers a way of justifying the presence of an item x in **U**. A justification is a sequence of rules corresponding to a proof showing why $x \in$ **U**. Sometimes an item x can be justified in more than one way. We will consider justifications one at a time. A complete justification of an item x in **U** will be called a composition for x; such a composition can be represented by a composition tree T_x. The nodes in T_x are labelled with the items mentioned in the antecedents of the rules of definition 2.2.2 that are applied; the root is labelled x.

Example 3.1.1 Suppose w is the result of an inclusion of x, x is the result of a concatenation of y and z, and y and z are base items. The composition tree T_w for w is as given below.



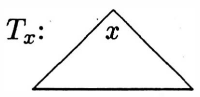
3.2 INFORMAL DESCRIPTION

We will speed up the slow parallel algorithm that computes relation **U** to a fast parallel algorithm computing **U** by using a relation denoted by $\tilde{\mathbf{U}}$ (given in section 3.4). The presence of each item x in **U** can be justified by means of a composition tree T_x. In T_x all

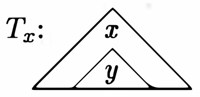
	$A \rightarrow \bullet a \bullet B$				$A \rightarrow \bullet aB \bullet$ $B \rightarrow \bullet A \bullet cc$
		$A \rightarrow \bullet a \bullet B$	$A \rightarrow \bullet aB \bullet$ $B \rightarrow \bullet A \bullet cc$	$B \rightarrow \bullet Ac \bullet c$	$B \rightarrow \bullet Acc \bullet$ $A \rightarrow a \bullet B \bullet$
			$B \rightarrow \bullet b \bullet$ $A \rightarrow a \bullet B \bullet$		
				$B \rightarrow A \bullet c \bullet c$ $B \rightarrow Ac \bullet c \bullet$	$B \rightarrow A \bullet cc \bullet$
					$B \rightarrow A \bullet c \bullet c$ $B \rightarrow Ac \bullet c \bullet$

Figure 1. Matrix $\mathcal{U}$ for *aabcc*

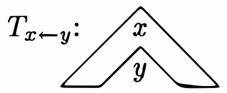
nodes are labelled with items in **U**. The root is labelled x. The other nodes are labelled by the items mentioned in the antecedents of the rules of the inductive definition of **U**. As an immediate consequence we have that each subtree of T_x is a composition tree too. We will represent T_x (or to be more exact: the existence of T_x) as in the figure below:



Suppose T_y exists. Thus we assume $y \in$ **U**. Let us see what the consequences of this assumption are. Suppose we can derive T_x for item x from T_y:

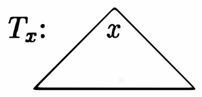


Assume we don't now wether or not y actually is in **U**. Instead of saying that we have determined T_x, we say that we have determined T_x except for the part T_y: we have the partial composition tree $T_{x \leftarrow y}$ (or better: its existence) represented as given below:

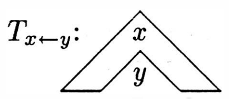


Note that $T_{x \leftarrow y}$ might exist whilst T_y does not (because $y \notin$ **U**). By using these partial composition trees, we draw conclusions from facts yet to be established. This makes the algorithm for the recognizer fast; the proof of this is based on Rytter's pebble game (Gibbons and Rytter, 1988).

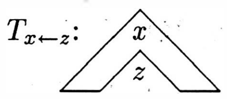
For each base item x (in **U**), we can assert the existence of a composition tree T_x:



Suppose x can be obtained from y by means of an inclusion operation. In that case we can assert the partial composition tree $T_{x \leftarrow y}$:

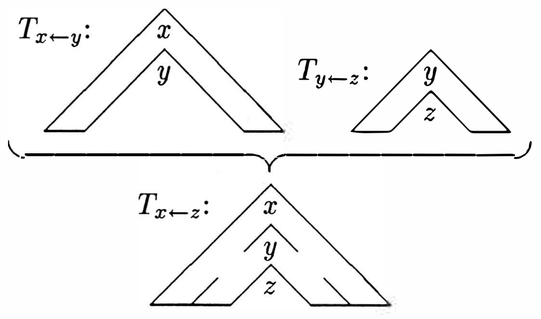


Now suppose that x can be obtained from y and z by means of a concatenation operation and assume that T_y exists (the case that T_z exists, is handled analogously). In that case we can assert the partial composition tree $T_{x \leftarrow z}$:

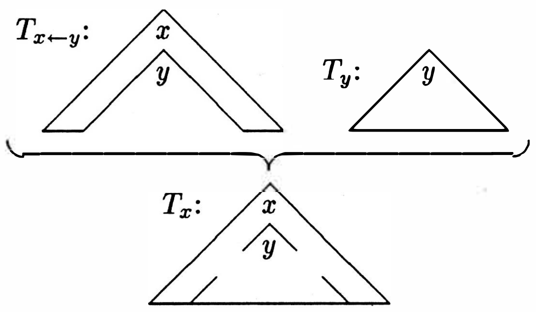


The rules for the inclusion and concatenation operations are called activation rules (the names of all rules are borrowed from the pebble game).

The square rule (a misnomer) merges two partial composition trees $T_{x \leftarrow y}$ and $T_{y \leftarrow z}$ to obtain the partial composition tree $T_{x \leftarrow z}$:



The final rule is the pebble rule, which merges a partial composition tree $T_{x \leftarrow y}$ and a composition tree T_y to obtain the composition tree T_x:



When we would define a composition tree for $\tilde{\mathbf{U}}$ in the same way as we did for $\mathbf{U}$, we would find that for an arbitrary $\mathbf{U}$-composition tree T_x there exists a reasonably well balanced $\tilde{\mathbf{U}}$-composition tree $\tilde{T}_x$, which also asserts that the item x is in $\mathbf{U}$. It can be shown that if the activation rule, the square rule, and the pebble rule are iterated $O(\log\ n)$ times, we have found the existence of at least one composition tree T_x for every x in $\mathbf{U}$ (and for only those). Therefore we can say that we can compute $\mathbf{U}$ in $O(\log\ n)$ time.

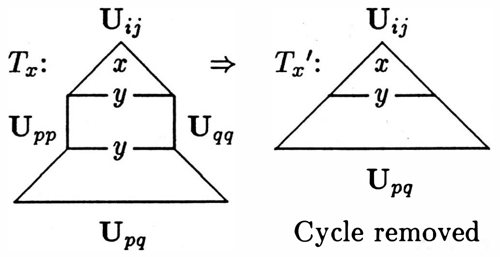
3.3 THE MINIMAL COMPOSITION SIZE

As a notational shortcut we will speak of an item x in $\mathbf{U}_{ij}$, by which we mean that $x \in \mathbf{U}$ and that x is of the form $(i, j, A \rightarrow \alpha \cdot \beta \cdot \gamma)$. The composition size will be defined as the number of operations in the composition tree. We call a composition tree minimal iff its composition size is minimal. In this section we will argue why the minimal composition size for item x in $\mathbf{U}_{ij}$ is linear in $j - i + 1$. There are two cases to consider:

- A composition tree which has an item appearing twice as a label on a path (such a tree is called a 'cyclic'[1] composition tree) is not minimal.
- An 'acyclic' composition tree has a linear composition size.

[1]A misnomer on our part.

Assume that for item x in $\mathbf{U}_{ij}$ we have found a cyclic composition tree T_x. So on a certain path in T_x we must have a certain item y in $\mathbf{U}_{pq}$ that appears twice as a label (the non-trivial path between those nodes is called a 'cycle'):



It is clear that when the part between the upper y and the lower y is removed from T_x, the number of operations in $T_x{}'$ is less than the number in T_x. So after removing a cycle, we allways get a smaller composition tree. Thus the minimal composition tree is a member of the set of the acyclic composition trees.

We will now argue that any acyclic composition tree has a composition size bounded by a function linear in the length of the string to be recognized. Since we don't need a tight upper bound, we will not use an actual composition. Instead, we will assume that in every step on our way the worst case occurs. This may lead to a 'case' that is worse than the actual worst case.

We will assume that every internal node is the result of a concatenation. Suppose x is the result of an inclusion of y: in that case T_x contains one more operation than T_y. But when x is the result of a concatenation of y and z, then T_x contains one more operation than T_y and T_z together. Thus a concatenation can only lead to more (and never to fewer) operations than an inclusion. We will assume that the compositions are acyclic.

We define $M = |\{A \rightarrow \alpha \cdot \beta \cdot \gamma \mid A \rightarrow \alpha\beta\gamma \in P\}|$; M is an upper bound for the number of items in any $\mathbf{U}_{ij}$. Let us focus on an item x in $\mathbf{U}_{jj}$, see figure 2(a). We know that item x has an acyclic composition, so T_x is bounded in height by $O(M)$. Since a completely balanced tree has the maximum number of operations, we have an exponential number of

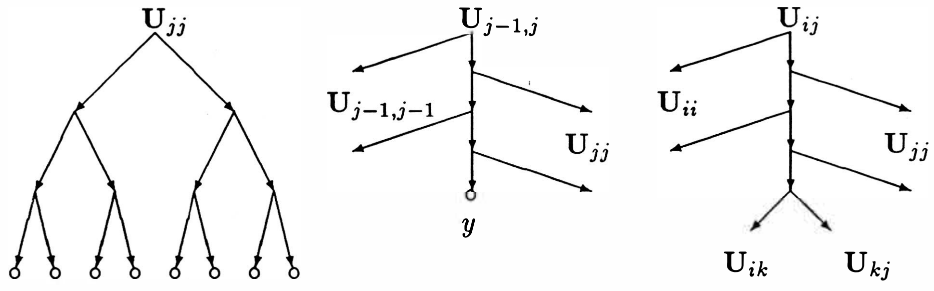


Figure 2. A simplified partial subtree of an acyclic T_x

operations in M. However, this number is independent of n. Thus there exist only $O(1)$ many operations in such a composition.

The next case is an item x in $\mathbf{U}_{j-1,j}$, see figure 2(b). We know that there must exist a path from x to a base item y in $\mathbf{U}_{j-1,j}$. All nodes on that path are in $\mathbf{U}_{j-1,j}$ and the path is bounded in length by $O(M)$. Any internal node on that path has one son in $\mathbf{U}_{j-1,j}$ and one son in either $\mathbf{U}_{j-1,j-1}$ or $\mathbf{U}_{jj}$ (if the node corresponds to an inclusion, this last son does not exist). Here too, it can be shown that only $O(1)$ operations are possible for item x.

The last case will be item x in $\mathbf{U}_{ij}$ with $i+1<j$, see figure 2(c). This is essentially like the previous case, but y is not a base item anymore. In this case y is the result of a concatenation of an item in $\mathbf{U}_{ik}$ and an item in $\mathbf{U}_{kj}$ with $i<k<j$. So instead we get $O(1)$ operations plus the number of operations needed for the item in $\mathbf{U}_{ik}$ and the item in $\mathbf{U}_{kj}$. These considerations lead to a difference equation, the solution of which shows that the number of operations for x is $O(n)$, see (de Vreught and Honig, 1990a).

3.4 THE EXTENDED RELATION

Definition 3.4.1 The relation $\tilde{\mathbf{U}}$ over $\mathcal{J} \cup \mathcal{J}^2$ is defined as follows:

- If $A \to \lambda \in P$ then $(j, j, A \to \cdot\cdot) \in \tilde{\mathbf{U}}$ for any $j \in \{0, \ldots, n\}$.
  This rule is used for the initialization.
- If $A \to \alpha a_j \gamma \in P$ then $(j-1, j, A \to \alpha \cdot a_j \cdot \gamma) \in \tilde{\mathbf{U}}$ for any $j \in \{1, \ldots, n\}$.
  This rule is used for the initialization.
- If $A \to \alpha B \gamma \in P$ and $B \to \beta \in P$ then $(i, j, A \to \alpha \cdot B \cdot \gamma) \leftarrow (i, j, B \to \cdot\beta\cdot) \in \tilde{\mathbf{U}}$ with $0 \le i \le j \le n$.
  This rule is called the activation rule for the inclusion operation.
- If $(i, k, A \to \alpha \cdot \beta_1 \cdot \beta_2 \gamma) \in \tilde{\mathbf{U}}$ then $(i, j, A \to \alpha \cdot \beta_1 \beta_2 \cdot \gamma) \leftarrow (k, j, A \to \alpha \beta_1 \cdot \beta_2 \cdot \gamma) \in \tilde{\mathbf{U}}$ with $k \le j \le n$.
  This rule is called an activation rule for the concatenation operation.
- If $(k, j, A \to \alpha \beta_1 \cdot \beta_2 \cdot \gamma) \in \tilde{\mathbf{U}}$ then $(i, j, A \to \alpha \cdot \beta_1 \beta_2 \cdot \gamma) \leftarrow (i, k, A \to \alpha \cdot \beta_1 \cdot \beta_2 \gamma) \in \tilde{\mathbf{U}}$ with $0 \le i \le k$.
  This rule is called an activation rule for the concatenation operation.
- If $x \leftarrow y \in \tilde{\mathbf{U}}$ and $y \leftarrow z \in \tilde{\mathbf{U}}$ then $x \leftarrow z \in \tilde{\mathbf{U}}$.
  This rule is called the square rule.
- If $x \leftarrow y \in \tilde{\mathbf{U}}$ and $y \in \tilde{\mathbf{U}}$ then $x \in \tilde{\mathbf{U}}$.
  This rule is called the pebble rule.
- Nothing is in $\tilde{\mathbf{U}}$ except those elements which must be in $\tilde{\mathbf{U}}$ by applying the preceding rules finitely often.

It can be shown that $\mathbf{U} = \tilde{\mathbf{U}} \cap \mathcal{J}$.

3.5 THE RECOGNIZER

We will present the fast parallel recognizer (de Vreught and Honig, 1990a).

```
Recognizer($n$):
    $\tilde{\mathbf{U}} := \emptyset$
    for all $i_1, i_2$ such that $0 \leq i_1 \leq i_2 \leq n$
    in parallel do
        Initialization($i_1$)
        ActivateInclusion($i_1, i_2$)
    while $\tilde{\mathbf{U}}$ still changes do
        for all $i_1, \ldots, i_6$ such that
        $0 \leq i_1 \leq \ldots \leq i_6 \leq n$ in parallel do
            ActivateConcatenation($i_1, \ldots, i_3$)
            Square($i_1, \ldots, i_6$)
            Square($i_1, \ldots, i_6$)
            Pebble($i_1, \ldots, i_4$)
    return Test($n$)

Initialization($j$):
    $\tilde{\mathbf{U}} := \tilde{\mathbf{U}} \cup$
        $\{(j, j, A \to \cdot\cdot) \in \mathcal{J} \mid A \to \lambda \in P\}$
    $\tilde{\mathbf{U}} := \tilde{\mathbf{U}} \cup$
        $\{(j-1, j, A \to \alpha \cdot a_j \cdot \gamma) \in \mathcal{J} \mid$
        $A \to \alpha a_j \gamma \in P\}$

ActivateInclusion($i, j$):
    $\tilde{\mathbf{U}} := \tilde{\mathbf{U}} \cup$
        $\{(i, j, A \to \alpha \cdot B \cdot \gamma) \leftarrow$
        $(i, j, B \to \cdot\beta\cdot) \in \mathcal{J}^2 \mid$
        $A \to \alpha B \gamma \in P$ and $B \to \beta \in P\}$

ActivateConcatenation($i, k, j$):
    for all $A \to \alpha\beta_1\beta_2\gamma \in P$ do
        if $(i, k, A \to \alpha \cdot \beta_1 \cdot \beta_2 \gamma) \in \tilde{\mathbf{U}}$ then
            $\tilde{\mathbf{U}} := \tilde{\mathbf{U}} \cup$
                $\{(i, j, A \to \alpha \cdot \beta_1\beta_2 \cdot \gamma) \leftarrow$
                $(k, j, A \to \alpha\beta_1 \cdot \beta_2 \cdot \gamma)\}$
            if $(k, j, A \to \alpha\beta_1 \cdot \beta_2 \cdot \gamma) \in \tilde{\mathbf{U}}$ then
                $\tilde{\mathbf{U}} := \tilde{\mathbf{U}} \cup$
                    $\{(i, j, A \to \alpha \cdot \beta_1\beta_2 \cdot \gamma) \leftarrow$
                    $(i, k, A \to \alpha \cdot \beta_1 \cdot \beta_2 \gamma)\}$

Square($i_1, i_2, i_3, j_3, j_2, j_1$):
    for all $A_1 \to \alpha_1\beta_1\gamma_1$, $A_2 \to \alpha_2\beta_2\gamma_2$,
    $A_3 \to \alpha_3\beta_3\gamma_3 \in P$ do
        if $(i_1, j_1, A_1 \to \alpha_1 \cdot \beta_1 \cdot \gamma_1) \leftarrow$
        $(i_2, j_2, A_2 \to \alpha_2 \cdot \beta_2 \cdot \gamma_2) \in \tilde{\mathbf{U}}$
        and $(i_2, j_2, A_2 \to \alpha_2 \cdot \beta_2 \cdot \gamma_2) \leftarrow$
        $(i_3, j_3, A_3 \to \alpha_3 \cdot \beta_3 \cdot \gamma_3) \in \tilde{\mathbf{U}}$ then
            $\tilde{\mathbf{U}} := \tilde{\mathbf{U}} \cup$
                $\{(i_1, j_1, A_1 \to \alpha_1 \cdot \beta_1 \cdot \gamma_1) \leftarrow$
                $(i_3, j_3, A_3 \to \alpha_3 \cdot \beta_3 \cdot \gamma_3)\}$

Pebble($i_1, i_2, j_2, j_1$):
    for all $A_1 \to \alpha_1\beta_1\gamma_1$, $A_2 \to \alpha_2\beta_2\gamma_2 \in P$ do
        if $(i_1, j_1, A_1 \to \alpha_1 \cdot \beta_1 \cdot \gamma_1) \leftarrow$
        $(i_2, j_2, A_2 \to \alpha_2 \cdot \beta_2 \cdot \gamma_2) \in \tilde{\mathbf{U}}$
        and $(i_2, j_2, A_2 \to \alpha_2 \cdot \beta_2 \cdot \gamma_2) \in \tilde{\mathbf{U}}$ then
            $\tilde{\mathbf{U}} := \tilde{\mathbf{U}} \cup$
                $\{(i_1, j_1, A_1 \to \alpha_1 \cdot \beta_1 \cdot \gamma_1)\}$

Test($n$):
    accept := False
    for all $S \to \alpha \in P$ do
        if $(0, n, S \to \cdot\alpha\cdot) \in \tilde{\mathbf{U}}$ then
            accept := True
    return accept
```

With the pebble game described in (Gibbons and Rytter, 1988), and the fact that the minimal composition size of an item is linear, we can show that any item can be constructed in $O(\log\ n)$ time. Thus $\mathbf{U} = \tilde{\mathbf{U}} \cap \mathcal{J}$ can be computed in $O(\log\ n)$ time. It can be shown that closure of $\tilde{\mathbf{U}}$ requires an extra $O(\log\ n)$ time following the computation of the completion of $\mathbf{U} \subseteq \tilde{\mathbf{U}}$ (detection of the closure of $\tilde{\mathbf{U}}$ is easy, whilst detection of the completion of $\mathbf{U} \subseteq \tilde{\mathbf{U}}$ is not).

It can be shown (de Vreught and Honig, 1990a) that the algorithm will compute the relation $\mathbf{U}$ on a CRCW-PRAM with $p(n) = \Theta(n^6)$ processors in $T(n) = O(\log\ n)$ time using $S(n) = O(n^4)$ space.

4 FINAL REMARKS

The slow parallel recognizer is based on a relatively simple idea. In spite of several similarities, it is not a variant of the Cocke-Younger-Kasami (CYK) algorithm or the Earley algorithm (Aho and Ullman, 1972; Harrison, 1978; Earley, 1970); the algebraic definitions specifying the algorithms all differ considerably, and therefore these algorithms all enter their 'items' into their respective matrices for different reasons. Just as for the given algorithm, there exist slow parallel versions of the CYK algorithm and of the Earley algorithm (Nijholt, 1990; Chiang and Fu, 1984).

The topic of fast parallel recognizing and parsing is still young and little research on the subject has been conducted. One of the first publications of a fast parallel recognizer is (Brent and Goldschlager, 1984). Far better known are the results of Gibbons and Rytter. They have described a fast parallel recognizer and parser for grammars in CNF (Gibbons and Rytter, 1988). Unfortunately, CNF is undesirable for many purposes. This is why we have developed a new fast parallel recognizer that leaves the grammar unchanged. Another recognizer with the same property can be found in (Sikkel and Nijholt, 1991).

Although not given in this paper there also exist parallel parsers which can be used in conjunction with the parallel recognizers. For the slow parallel recognizer there exists a slow parallel parser that can do its job with $\Theta(n)$ processors in $O(n \log\ n)$ time (de Vreught and Honig, 1990b). When the grammar is acyclic, there exists a fast parallel parser running with $\Theta(n^6)$ processors in $O(\log\ n)$ time (de Vreught and Honig, 1990a).

Since the subject of fast parallel parsing is so young, there are many open questions, some of which will probably be solved in the near future. For instance, at this moment it is not yet known whether or not fast parsing of general CFG's is possible without transforming the grammar (we suspect that it is). In addition, determining the behaviour of the algorithms for unambiguous grammars is an interesting research problem.

REFERENCES

Aho, Alfred V. and Ullman, Jeffrey D. 1972 *The Theory of Parsing, Translation and Compiling, Volume I: Parsing*, Prentice Hall, Englewood Cliffs, NJ.

Aho, Alfred V. and Ullman, Jeffrey D. 1973 *The Theory of Parsing, Translation and Compiling, Volume II: Compiling*, Prentice Hall, Englewood Cliffs, NJ.

Akker, Rieks op den; Alblas, Henk; Nijholt, Anton; and Oude Luttighuis, Paul 1989 An Annoted Bibliography on Parallel Parsing, Memoranda Informatica 89-67, University of Twente, Enschede.

Brent, Richard P. and Goldschlager, Leslie M. 1984 A Parallel Algorithm for Context-Free Parsing, *Austral. Comput. Sci. Comm.* 6: 7-1 - 7-10.

Chiang, Y.T. and Fu, King S. 1984 Parallel Parsing Algorithms and VLSI Implementations for Syntactic Pattern Recognition, *IEEE Trans. Pattern Anal. Mach. Intell.* 6: 302-314.

Earley, Jay 1970 An efficient Context-Free Parsing Algorithm, *Commun. ACM* 13(2): 94-102.

Gibbons, Alan and Rytter, Wojciech 1988 *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, MA.

Harrison, Michael A. 1978 *Introduction to Formal Language Theory*, Addison Wesley, Reading, MA.

Nijholt, Anton 1990 The CYK-Approach to Serial and Parallel Parsing, Memoranda Informatica 90-13, University of Twente, Enschede.

Quinn, Michael J. 1987 *Designing Efficient Algorithms for Parallel Computers*, McGraw-Hill, New York, NY.

Sikkel, Klaas and Nijholt, Anton 1991 An Efficient Connectionist Context-Free Parser, In *2nd Int. Workshop on Parsing Technologies 1991* (these proceedings).

Vreught, Hans de and Honig, Job 1989 A Tabular Bottom Up Recognizer, Reports of the Faculty of Technical Mathematics and Informatics 89-78, Delft University of Technology, Delft.

Vreught, Hans de and Honig, Job 1990a A Fast Parallel Recognizer, Reports of the Faculty of Technical Mathematics and Informatics 90-16, Delft University of Technology, Delft.

Vreught, Hans de and Honig, Job 1990b General Context-Free Parsing, Reports of the Faculty of Technical Mathematics and Informatics 90-31, Delft University of Technology, Delft.