Journal of Computer and System Sciences **56**, 83–89 (1998)
Article No. SS971537

Sanguthevar Rajasekaran

Department of CISE, University of Florida, Gainesville, Florida 32611
E-mail: raj@cise.ufl.edu

and

Shibu Yooseph

Department of CIS, University of Pennsylvania, Philadelphia, Pennsylvania 19104
E-mail: yooseph@gradient.cis.upenn.edu

Received February 10, 1995; revised January 17, 1997

We propose an $O(M(n^2))$ time algorithm for the recognition of tree adjoining languages (TALs), where n is the size of the input string and $M(k)$ is the time needed to multiply two $k \times k$ boolean matrices. Tree adjoining grammars are formalisms suitable for natural language processing and have received enormous attention in the past among not only natural language processing researchers but also algorithms designers. The first polynomial time algorithm for TAL parsing was proposed in 1986 and had a run time of $O(n^6)$. Quite recently, an $O(n^3 M(n))$ algorithm has been proposed. The algorithm presented in this paper improves the run time of the recent result using an entirely different approach.

1. INTRODUCTION

The tree adjoining grammar (TAG) formalism was introduced by Joshi, Levy, and Takahashi [4]. TAGs are tree generating systems and are strictly more powerful than context-free grammars. They belong to the class of *mildly context sensitive grammars* [5]. They have been found to be good grammatical systems for natural languages [6]. The first polynomial time parsing algorithm for TALs was given by Vijay-Shanker and Joshi [15], which had a run time of $O(n^6)$, for an input of size n. Their algorithm had a flavor similar to the Cocke–Younger–Kasami (CYK) algorithm for context-free grammars.

An Earley-type parsing algorithm has been given by Schabes and Joshi [12]. An optimal linear time parallel parsing algorithm for TALs was given by Palis, Shende, and Wei [7]. In a recent paper, Rajasekaran [9] shows how TALs can be parsed in time $O(n^3 M(n))$.

In this paper, we propose an $O(M(n^2))$ time recognition algorithm for TALs, where $M(k)$ is the time needed to multiply two $k \times k$ boolean matrices. The best known value for $M(k)$ is $O(n^{2.376})$ [1]. Although our algorithm is similar in flavor to those of Graham, Harrison, and Ruzzo [2], and Valiant [13] (which were algorithms proposed for recognition of context-free languages (CFLs)), there are crucial differences. As such, the techniques of [2, 13] do not seem to extend to TALs [10, 11].

* This research was supported in part by an NSF Research Initiation Award CCR-92-09260 and an NSF Grant CCR-95-03007.

2. TREE ADJOINING GRAMMARS

DEFINITION 1. A *tree adjoining grammar* (*TAG*) G consists of a quintuple $(N, \Sigma \cup \{\varepsilon\}, I, A, S)$, where

N is a finite set of nonterminal symbols,

Σ is a finite set of terminal symbols disjoint from N,

ε is the empty terminal string not in Σ,

I is a finite set of labelled initial trees,

A is a finite set of auxiliary trees,

$S \in N$ is the distinguished start symbol.

The trees in $I \cup A$ are called *elementary trees*. All internal nodes of elementary trees are labelled with nonterminal symbols. Also, every initial tree is labelled at the root by the start symbol S and has leaf nodes labelled with symbols from $\Sigma \cup \{\varepsilon\}$. An auxiliary tree has both its root and exactly one leaf (called the *foot node*) labelled with the *same* nonterminal symbol. All other leaf nodes are labelled with symbols in $\Sigma \cup \{\varepsilon\}$, at least one of which has a label strictly in Σ. An example of a TAG is given in Fig. 1.

For our purpose, we will assume that every internal node in an elementary tree has exactly two children. Each node in a tree is represented by a tuple $\langle tree, node, index, label \rangle$. For brevity, we will refer to a node with a single variable m wherever there is no confusion. A good introduction to TAGs can be found in [3, 8].

0022-0000/98 $25.00

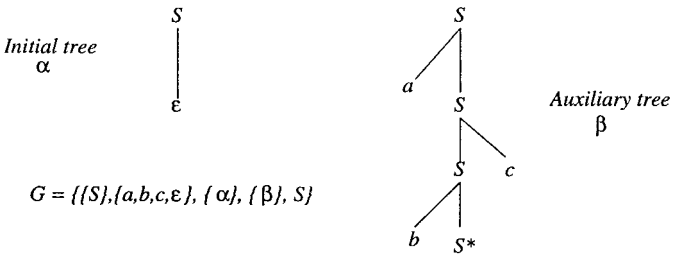

FIG. 1. Example of a TAG.

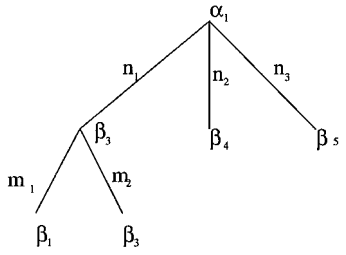

FIG. 3. Derivation tree.

3. THE ADJUNCTION OPERATOR

A tree built from an operation involving two other trees is called a *derived tree*. The operation involved is called *adjunction*.

DEFINITION 2. *Adjunction* is an operator which builds a new tree γ, from an auxiliary tree β and another tree α (α is any tree—initial, auxiliary, or derived). Let α contain an internal node m labelled X and let β be the auxiliary tree with root node also labelled X. The resulting tree γ, obtained by *adjoining* β onto α at node m is built as follows (Fig. 2):

1. The subtree of α rooted at m, call it t, is excised, leaving a copy of m behind.

2. The auxiliary tree β is attached at the copy of m and its root node is identified with the copy of m.

3. The subtree t is attached to the foot node of β and the root node of t (i.e., m) is identified with the foot node of β.

This definition can be extended to include adjunction constraints at nodes in a tree. The constraints include selective, null, and obligatory adjunction constraints [3]. The algorithm we present can be modified to include such constraints.

Although the above definition of adjunction involves an auxiliary tree and another possibly complex structure (i.e., derived tree), we have defined it as such only for convenience. Strictly speaking, adjunction is done only at a node in an elementary tree. We will use the convention that more than one auxiliary tree can be adjoined to an elementary tree provided that each auxiliary tree is adjoined at a distinct node in the elementary tree. The sequence of adjunctions can be described using what is called a *derivation tree*. Fig. 3 shows a derivation tree for the sequence of adjunctions using some TAG G. Informally, each edge in this derivation tree is labelled with the address of the node where the auxiliary tree, indicated by the label on the lower vertex of this edge, is adjoined. Note that no two edges on the same level of the derivation tree share a common label, thus indicating what we mentioned earlier. In Fig. 3, β_3 is adjoined at node n_1 of the elementary tree α_1. Then auxiliary trees β_1 and β_3 are adjoined at the indicated nodes of β_3. The other adjunctions are explained similarly. For more details regarding derivation trees, we refer to [14].

We briefly discuss an alternate view of adjunction. From Fig. 3, it is useful to think of first obtaining the derived tree γ got by adjoining β_1 and β_3 on the tree β_3 at the nodes indicated in the figure. γ can then be adjoined at node n_1 of the elementary tree α_1.

Note that γ is obtained as a result of adjunctions involving only auxiliary trees. Our final adjunction involved γ and α_1. We call γ a *derived auxiliary tree.*

DEFINITION 3. A *derived auxiliary tree* is a tree derived by a series of adjunction operations (as described in Definition 2) involving only auxiliary trees.

This extended view of adjunction, where we allow for one of the trees to be either an auxiliary tree or a derived auxiliary tree (as opposed to our initial definition of adjunction where we insisted that one tree always be an auxiliary tree), will be useful in our recognition algorithm.

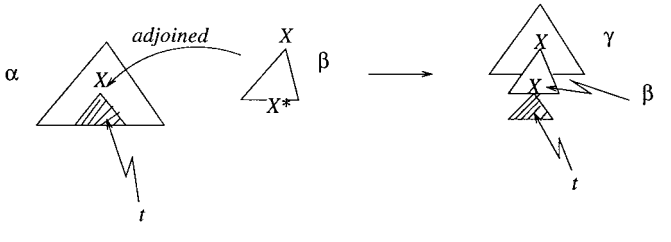

FIG. 2. Adjunction operation.

4. RECOGNITION ALGORITHM

The data structure we use is a four-dimensional matrix $A_{(n+1)\times(n+1)\times(n+1)\times(n+1)}$, where n is the size of the input string. The entries in A are nodes of the trees of G.

4.1. *Entries of Matrix A*

Let the input string be $a_1 a_2 \cdots a_n$. Let m be a node in an elementary tree η. Let γ be a derived tree which is obtained as a result of a sequence of adjunctions involving auxiliary trees (or derived auxiliary trees) and some of the nodes of η.

Case I. Suppose m is an ancestor of the footnote (labelled X) of η. Then $m \in A(i, j, k, l)$ iff the labels at the leaves (in left-to-right order) of the subtree of γ, rooted at m, is given by $a_{i+1} \cdots a_j X a_{k+1} \cdots a_l$.

Case II. Suppose m is not an ancestor of the footnote of η. Then $m \in A(i, j, j, l)$, for some j, iff the labels at the leaves (in left-to-right order) of the subtree of γ, rooted at m, is given by $a_{i+1} \cdots a_l$.

DEFINITION 4. A node m *spans a subtree* (i, j, k, l) if $m \in A(i, j, k, l)$.

4.2. *Algorithm*

The idea behind our algorithm is the following: Suppose the input string is generated by the TAG G. Let γ be a derived tree for this string. The algorithm works in a bottom-up fashion figuring out the nodes in γ. This process of figuring out if a node m spans a subtree (i, j, k, l) works by checking if an adjunction was performed at node m by an auxiliary tree or a derived auxiliary tree. This process is done efficiently using matrix multiplication if the string is not generated by the TAG then this is discovered in the process.

Initialization:

- $A(i, i+1, i+1, i+1)$ with all nodes m such that m has the same label as the input a_{i+1} for $0 \leqslant i \leqslant (n-1)$.
- $A(i, i, j, j)$ for $0 \leqslant i \leqslant j \leqslant n$ with all foot nodes.
- $A(i, i, i, i)$ for $0 \leqslant i \leqslant n$ with all nodes m such that m has label ε.

The following is the recognition algorithm, *Compute Nodes*, which is initially called with the sequence $\langle 0, 1, 2, ..., n \rangle$ corresponding to the input string $a_1 \cdots a_n$, where the element i of the sequence corresponds to the space between the ith and $(i+1)$th symbols of the input string. (Here element 0 of the sequence corresponds to the space before a_1). In the process of dealing with subproblems, *Compute Nodes* deals with sequences of the form $\langle r_1, r_2, ..., r_p \rangle$, where $r_1 < r_2 < \cdots < r_p$ and r_{i+1} need not be equal to $r_i + 1$ (i.e., the elements in the sequence need not be consecutive). We will refer to the sequence $\langle r_1, r_2, ..., r_p \rangle$ as a sequence of symbol positions. *Compute Nodes* identifies all nodes spanning subtrees (i, j, k, l), with $\{i, l\} \subseteq \{r_1, r_2, ..., r_p\}$ and $\{j, k\} \subseteq \{r_1, r_2 + 1, r_1 + 2, ..., r_p\}$.

ALGORITHM. *Compute Nodes* $(r_1 r_2 ... r_p)$.

Begin

1. If $p = 2$ and $r_2 = r_1 + 1$, then
do{
1a. For each node $m \in$ TAG G (*Comment*: *Check for adjunction*)
do{
If m spans a subtree (l_1, j, k, l_2)
[where $r_1 \leqslant l_1 \leqslant j \leqslant k \leqslant l_2 \leqslant r_2$], and
if there is a node m_1 spanning a subtree (r_1, l_1, l_2, r_2) such that m_1 has the same label as m and is the root of an auxiliary tree, then add m to $A(r_1, j, k, r_2)$.
}enddo
1b. Return.
}enddo
2. If $p = 2$ and $r_2 \neq r_1 + 1$, then return, else
do{
2a. *Compute Nodes* $(r_1 r_2 \cdots r_{2p/3})$.
2b. *Compute Nodes* $(r_{1+p/3} \cdots r_p)$.
2c. For each node $m \in$ TAG G,
do{
If m has its left and right child spanning subtrees (i, j, k, s) and (s, r, r, l), respectively,
[where $i \in \{r_1, r_2, ..., r_{p/3}\}$,
$s \in \{r_{1+p/3}, ..., r_{2p/3}\}$,
$l \in \{r_{1+2p/3}, ..., r_p\}$], or,
if m has its left and right child spanning subtrees (i, r, r, s) and (s, j, k, l), respectively,
[where $i \in \{r_1, r_2, ..., r_{p/3}\}$,
$s \in \{r_{1+p/3}, ..., r_{2p/3}\}$,
$l \in \{r_{1+2p/3}, ..., r_p\}$], then add m to $A(i, j, k, l)$.
}enddo
2d. For each node $m \in$ TAG G (*Comment*: *We now check for adjunction*)
do{
If m spans a subtree (q, j, k, r), and there is a node m_1 spanning a subtree (i, q, r, l)
[where $i \in \{r_1, r_2, ..., r_{p/3}\}$,
$l \in \{r_{1+2p/3}, ..., r_p\}$], such that m_1 has the same label as m and is the root of an auxiliary tree, then add m to $A(i, j, k, l)$.
}enddo
2e. *Compute Nodes* $(r_1 r_2 \cdots r_{p/3} r_{1+(2p/3)} \cdots r_p)$
2f. Return.
}enddo
End

Output. Yes (i.e., the string is in the language) iff there exists a root of an initial tree (labelled S) in some $A(0, j, j, n)$.

In the above algorithm, for clarity, we have omitted a small technical detail regarding the empty string ε. Suppose we have just decided that a node m, from an elementary tree β,

is in $A(i, j, k, l)$. Suppose in β, the sibling m', of m, yields ε at its frontier (i.e., subtree rooted at m' yields the empty string). Then we need to add the parent m^* of m to $A(i, j, k, l)$ also. In fact, we might need to put the parent m^{**}, of m^*, in $A(i, j, k, l)$ also if the sibling of m^* yields ε. This updating procedure can be done efficiently by preprocessing G so as to identity, for every node m, the list of nodes associated with m, so that this list is also added to the same entry that m is added to. For a node m, we shall call the list associated with it as $ASSOC\ LIST(m)$. For each node m, this list can be computed using the Procedure *MAKE LIST* which we now describe. We assume that the initialization part of *Compute Nodes* and every step adding a node to an entry in A in *Compute Nodes* also adds that node's *ASSOC LIST* to the same entry.

Initialize $ASSOC\ LIST(m) = \phi\ \forall m$ and then call Procedure *MAKE LIST* on each elementary tree, in a top down fashion starting with the root node.

PROCEDURE *MAKE LIST*(m).
Begin
If m is non-leaf, then

1. If m has children m_1 and m_2 both yielding the empty string at their frontiers (i.e. m spans a subtree yielding ε) then

$$ASSOC\ LIST(m_1) = ASSOC\ LIST(m) \cup \{m\}$$
$$ASSOC\ LIST(m_2) = ASSOC\ LIST(m) \cup \{m\}$$

2. If m has children m_1 and m_2, with only m_2 yielding the empty string at its frontier, then

$$ASSOC\ LIST(m_1) = ASSOC\ LIST(m) \cup \{m\}$$

End

Since the size of the grammar is constant, the above procedure can be done in constant time. Also, note that in the above procedure, a foot node is considered as yielding ε also. (This is clear for the same reason as for initializing all $A(i, i, j, j)$, where $0 \leqslant i \leqslant j \leqslant n$, with all the foot nodes.)

4.3. *Implementation Details of Compute Nodes*

Step 2c can be carried out in the following manner. The two cases are handled as follows.

Case 1. If node m_1 in a derived tree is the ancestor of the foot node, and node m_2 is its right sibling, such that $m_1 \in A(i, j, k, l)$ and $m_2 \in A(l, r, r, s)$, then their parent, say node m should belong to $A(i, j, k, s)$. This checking can be accomplished efficiently using boolean matrix multiplication in the following way (we use a technique similar to the one used in [9]): Construct two boolean matrices B_1, of size $((n+1)^2\ p/3) \times p/3)$ and B_2, of size $p/3 \times p/3$,

$$B_1(ijk, l) = 1 \quad \text{iff } m_1 \in A(i, j, k, l)$$
$$\text{and } i \in \{r_1, ..., r_{p/3}\}$$
$$\text{and } l \in \{r_{1+p/3}, ..., r_{2p/3}\}$$
$$= 0 \quad \text{otherwise}$$

Note that in B_1, $0 \leqslant j \leqslant k \leqslant n$,

$$B_2(l, s) = 1 \quad \text{iff } m_2 \in A(l, r, r, s)$$
$$\text{and } l \in \{r_{1+p/3}, ..., r_{2p/3}\}$$
$$\text{and } s \in \{r_{1+2p/3}, ..., r_p\}$$
$$= 0 \quad \text{otherwise.}$$

Clearly the dot product of the ijkth row of B_1 with the sth column of B_2 is a 1 iff $m \in A(i, j, k, s)$. Thus, update $A(i, j, k, s)$ with $\{m\} \cup ASSOC\ LIST(m)$.

Case 2. If node m_2 in a derived tree is the ancestor of the foot node, and node m_1 is its left sibling, such that $m_1 \in A(i, j, j, l)$ and $m_2 \in A(l, p, q, r)$, then their parent, say node m should belong to $A(i, p, q, s)$. This can also be handled similar to the manner described for Case 1. Update $A(i, p, q, s)$ with $\{m\} \cup ASSOC\ LIST(m)$.

Checking for adjunction. (Steps 1a and 2d). We know that if a node $m \in A(i, j, k, l)$, and the root m_1 of an auxiliary tree is in $A(r, i, l, s)$, then adjoining the tree η, rooted at m_1, onto the node m, results in the node m spanning a subtree (r, j, k, s), i.e. $m \in A(r, j, k, s)$.

We can essentially use the previous technique of reducing to boolean matrix multiplication. We describe how Step 2d is carried out (Step 1a can be done in a similar fashion). Construct two matrices C_1 and C_2 of sizes $(p^2/9) \times (n+1)^2$ and $(n+1)^2 \times (n+1)^2$, respectively, as follows:

$$C_1(il, jk) = 1 \quad \text{iff } \exists m_1, \text{ root of an auxiliary tree}$$
$$\in A(i, j, k, l), \text{ with same label as } m$$
$$= 0 \quad \text{otherwise.}$$

Note that in C_1, $i \in \{r_1, ..., r_{p/3}\}$, $l \in \{r_{1+2p/3}, ..., r_p\}$, and $0 \leqslant j \leqslant k \leqslant n$:

$$C_1(qt, rs) = 1 \quad \text{iff } m \in A(q, r, s, t)$$
$$= 0 \quad \text{otherwise.}$$

Note that in C_2, $0 \leqslant q \leqslant r \leqslant s \leqslant t \leqslant n$.

Clearly the dot product of the ilth row of C_1 with the rsth column of C_2 is a 1 iff $m \in A(i, r, s, l)$. Thus, update $A(i, r, s, l)$ with $\{m\} \cup ASSOC\ LIST(m)$.

5. COMPLEXITY

Step **2c** can be computed in $O(n^2 M(p))$.

Steps **1a** and **2d** can be computed in $O((n^2/p^2)^2\, M(p^2))$.

If $T(p)$ is the time taken by Procedure *Compute Nodes* for an input of size p then

$$T(p) = 3T\left(\frac{2p}{3}\right) + O(n^2 M(p)) + O\left(\left(\frac{n^2}{p^2}\right)^2 M(p^2)\right),$$

where n is the initial size of the input string.

Solving the recurrence relation, we get $T(n) = O(M(n^2))$.

6. PROOF OF CORRECTNESS

We will show the proof of correctness of the algorithm by induction on p, the length of the sequence $\langle r_1, r_2, ..., r_p\rangle$. To help in our endeavour, we will make use of the concept if a *minimal node* w.r.t. two symbol positions r_s, r_t, where $r_t > r_s + 1$.

DEFINITION 5. Given two elements r_s, r_t from the sequence $\langle r_1, r_2, ..., r_p\rangle$ such that $r_t > r_s + 1$, we say that a node m (from an elementary tree α) is *minimal* w.r.t. (r_s, r_t) if it spans a subtree (i, j, k, l) with $i \leqslant r_s$ and $l \geqslant r_t$, such that either of the following properties holds:

1. The left child and right child of m in α, namely m_1 and m_2, are such that they span subtrees (i, q, q, h_1) and (h_1, j, k, l), respectively, or they span subtrees (i, j, k, h_1) and (h_1, q, q, l), respectively, where $r_s < h_1 < r_t$.

2. m spans the subtree (i, j, k, l) as a result of adjoining an auxiliary tree (or a derived auxiliary tree) with root m^*, spanning subtree (i, q_1, q_2, l), on the node m which initially spanned the subtree (q_1, j, k, q_2), such that node m^* has children satisfying property 1 or m^* belongs to the *ASSOC LIST* of a node satisfying property 1.

3. m belongs to the *ASSOC LIST* of a node satisfying properties 1 or 2.

DEFINITION 6. Given a sequence $\langle r_1, r_2, ..., r_p\rangle$, we call (r_i, r_{i+1}) a *gap* if $r_{i+1} > r_i + 1$.

Given the sequence $\langle 0, 1, 2, ..., n\rangle$ as input, the algorithm *Compute Nodes* proceeds by working with subproblems of the form $\langle r_1, r_2, ..., r_p\rangle$, where these sequences can have gaps in them. We will now show that the algorithm identifies all minimal nodes w.r.t. every new gap created (in each recursive call) and that this is enough to ensure its correctness.

LEMMA 1. *Given a sequence* $\langle r_1, r_2, ..., r_p\rangle$ *of symbol positions and given*

a. $\forall$ *gaps*(r_1, r_{q+1}), *all nodes spanning subtrees* (i, j, k, l) *with* $r_q < i \leqslant j \leqslant k \leqslant l < r_{q+1}$

b. $\forall$ *gaps*(r_q, r_{q+1}), *all nodes spanning subtrees* (i, j, k, l) *such that either* $r_q < i < r_{q+1}$ *or* $r_q < l < r_{q+1}$

c. $\forall$ *gaps*(r_q, r_{q+1}), *all the minimal nodes for the gap such that these nodes span subtrees* (i, j, k, l) *with* $\{i, l\} \subseteq \{r_1, r_2, ..., r_p\}$ *and* $i \leqslant l$

d. *the initialization information*

the algorithm Compute Nodes identifies all the nodes spanning subtrees (i, j, k, l) *with* $\{i, l\} \subseteq \{r_1, r_2, ..., r_p\}$ *and* $i \leqslant j \leqslant k \leqslant l$.

Proof.

Base case ($p = 2$). There are two cases to consider:

Case 1. $r_2 > r_1 + 1$, which implies that (r_1, r_2) is a gap. Since every node spanning a subtree (r_1, j, k, r_2) is a minimal node w.r.t. (r_1, r_2) and since this information is already known, it follows that Compute Nodes trivially returns the set of all minimal nodes spanning subtrees (i, j, k, l) with $\{i, l\} \subseteq \{r_1, r_2\}$.

Case 2. $r_2 = r_1 + 1$. Recall that the initialization information trivially identifies all leaf nodes in the elementary trees having the same label as the symbol a_{r_2}. Further, the initialization also adds the *ASSOC LIST* of every node to the same entry that the node is added to. Also recall that at least one leaf node in every auxiliary tree has a label strictly in Σ. Thus the only possible nodes yet to be identified as spanning (r_1, r_2, r_2, r_2) are those nodes at which an adjunction was performed by an auxiliary tree (containing exactly one leaf node having the label from Σ and this label being the same as a_{r_2}). This is taken care of in Step 1a of *Compute Nodes*. Thus *Compute Nodes* correctly returns all nodes spanning subtrees (i, j, k, l) with $\{i, l\} \subseteq \{r_1, r_2\}$.

Induction hypothesis. $\forall$ sequences $\langle r_1, r_2, ..., r_{p'}\rangle$ of symbol positions of length $\leqslant p$ (i.e., $p' \leqslant p$), *Compute Nodes*, given the information as mentioned in the statement of the lemma, identifies nodes spanning subtrees (i, j, k, l) such that $\{i, l\} \subseteq \{r_1, r_2, ..., r_{p'}\}$ and $i \leqslant j \leqslant k \leqslant l$.

For the induction step, suppose *Compute Nodes* is given the sequence $\langle r_1, r_2, ..., r_{p+1}\rangle$, together with the information mentioned in the lemma, as input.

Now, by the induction hypothesis, *Compute Nodes* correctly identifies all nodes spanning subtrees (i, j, k, l) with $\{i, l\} \subseteq \{r_1, r_2, ..., r_{2(p+1)/3}\}$ and also all nodes spanning subtrees (i, j, k, l) with $\{i, l\} \subseteq \{r_{(p+1)/3+1}, ..., r_{p+1}\}$.

It then gets rid of the middle $\frac{1}{3}$ of the sequence and works with the sequence $\langle r_1, r_2, ..., r_{(p+1)/3}, r_{(2(p+1)/3)+1}, ..., r_{p+1}\rangle$. Note that this sequence has a new gap, namely $(r_{(p+1)/3}, r_{(2(p+1)/3)+1})$. So, before we can apply the induction hypothesis to this sequence, we need to ensure that *Compute Nodes* has identified all the minimal nodes w.r.t. this new gap.

A minimal node m w.r.t. the gap $(r_{(p+1)_3}, r_{(2(p+1)/3)+1})$ can span a subtree (i, j, k, l), with $i \leqslant r_{(p+1)/3}$ and $l \geqslant r_{2(p+1)/3}$, where one of the following holds:

1. Either i or l is in a gap in the sequence $\langle r_1, r_2, ..., r_{p+1}\rangle$, i.e. either i or l is strictly between r_q and r_{q+1} for some q, where $r_{q+1} > r_q + 1$.

2. $\{i, l\} \subseteq \{r_1, r_2, ..., r_{p+1}\}$ and the h_1 value of node m (recall Definition 5 for h_1 value reference) is not in $\{r_{(p+1)/2+1}, r_{((p+1)/3)+2}, ..., r_{2(p+1)/3}\}$.

3. $\{i, l\} \subseteq \{r_1, r_2, ..., r_{p+1}\}$ and the h_1 value of node m (recall Definition 5 for h_1 value reference) is in $\{r_{((p+1)/3)+1}, r_{((p+1)/3)+2}, ..., r_{2(p+1)/3}\}$.

All nodes having property 1 or 2 are either already known or are identified by *Compute Nodes* (using the hypothesis). Only nodes having property 3 have to be identified.

It can be seen that Steps 2c and 2d of *Compute Nodes* identifies all such nodes. Thus, we can now apply the induction hypothesis on the sequence $\langle r_1, r_2, ..., r_{(p+1)/3}, r_{2(p+1)/3+1}, ..., r_{p+1}\rangle$ and infer that *Compute Nodes* correctly identifies all the nodes spanning subtrees (i, j, k, l) with $\{i, l\} \subseteq \{r_1, r_2, ..., r_{p+1}\}$. ▮

We now state a lemma relating the recognition of a string and the entries in the matrix A. The proof is a straightforward consequence of the definition of the entries of A.

LEMMA 2. *The input string $a_1 a_2 \cdots a_n$ is generated by the TAG G iff there is a root of an initial tree (labelled S) in some $A(0, j, j, n)$.*

LEMMA 3. *The root m of an initial tree (labelled S) is in $A(0, j, j, n)$, for some j, iff m is a minimal node w.r.t. the gap $(0, n)$.*

Proof. For any node $m' \in A(r_1, i_1, i_2, r_2)$, clearly m' is minimal w.r.t. gap (r_1, r_2). Thus, if $m \in A(0, j, j, n)$, then m is minimal w.r.t. $(0, n)$.

For the other direction, suppose m is a minimal node w.r.t. the gap $(0, n)$. Then, since m does not dominate a foot node, it follows that $m \in A(0, j, j, n)$ for some j. ▮

Thus we have the following theorem.

THEOREM 1. *Compute Nodes correctly determines if the given string is generated the TAG G.*

Proof. Using Lemma 1, we see that at each stage of the recursion, *Compute Nodes* correctly identifies all minimal nodes w.r.t. new gaps created. In particular, for the last subproblem, namely the sequence $\langle 0, n\rangle$, *Compute Nodes* correctly identifies all the minimal nodes w.r.t. $(0, n)$. This, along with Lemmas 2 and 3, imply that *Compute Nodes* correctly determines if the given string is in the TAL generated by G. ▮

7. HANDLING ADJUNCTION CONSTRAINTS

In this section we briefly discuss how Procedure *Compute Nodes* can be modified to handle adjunction constraints. In particular, we discuss how to handle Selective Adjunction constraints. The other constraints can be handled similarly. Selective adjunction constraints at a node indicate the subset of auxiliary trees that can be adjoined at that node.

The entries of matrix A are modified so that each node $m \in A(r, j, k, s)$ also has a list associated with it. Each element in this list is the root of an auxiliary tree (or a derived auxiliary tree) which when adjoined at node m, results in $m \in A(r, j, k, s)$. The list is initially empty.

Recall the implementation details of checking for adjunction from Section 4.3. Steps 1a and 2d are modified as follows to handle selective adjunction constraints at a node m. Instead of checking for all possible adjunctions at node m in one multiplication involving C_1 and C_2, we will check for adjunction involving the root of each auxiliary tree and node m separately. Since the grammar size is constant, the run time analysis does not change. For each m_1, such that m_1 is the root of an auxiliary tree with same label as node m, we do the following.

Construct two matrices C_1 and C_2 of sizes $(p^2/9) \times (n+1)^2$ and $(n+1)^2 \times (n+1)^2$, respectively, as

$$C_1(il, jk) = 1 \quad \text{if } m_1 \in A(i, j, k, l).$$

$$= 0 \quad \text{otherwise.}$$

Note that in C_1, $i \in \{r_1, ..., r_{p/3}\}$, $l \in \{r_{1+2p/3}, ..., r_p\}$, and $0 \leqslant j \leqslant k \leqslant n$:

$C_2(qt, rs) = 1$ if $m \in A(q, r, s, t)$, and either the list attached to m in $A(q, r, s, t)$ contains a node m' such that the auxiliary tree with root m, is allowed to be adjoined at m' or, the list is empty and the auxiliary tree with root m_1 is allowed to be adjoined at m

0 otherwise.

Note that in C_2, $0 \leqslant q \leqslant r \leqslant s \leqslant t \leqslant n$.

Clearly the dot product of the ilth row of C_1 with the rsth column of C_2 is a 1 iff $m \in A(i, r, s, l)$. If m is identified to be in $A(i, r, s, l)$, then add m_1 to m's list in $A(i, r, s, l)$. Also update $A(i, r, s, l)$ with $\{m\} \cup ASSOC\ LIST(m)$.

8. IMPLEMENTATION

The *TAL* recognizer given in this paper was implemented in *Scheme* on a SPARC station-10/30. Theoretical results in this paper and those in [9] clearly demonstrate that

asymptotically fast algorithms can be obtained for TAL parsing with the help of matrix multiplication algorithms. The main objective of the implementation was to check if matrix multiplication techniques help in practice also to obtain efficient parsing algorithms.

The recognizer implemented two different algorithms for matrix multiplication, namely the trivial cubic time algorithm and an algorithm that exploits the sparsity of the matrices. The TAL recognizer that uses the cubic time algorithm has a run time comparable to that of Vijay-Shanker and Joshi's algorithm [15].

Below is given a sample of a grammar tested and also the speedup using the sparse version over the ordinary version. The grammar generated the TAL $\{a^n b^n c^n \mid n \geqslant 0\}$. The initial and auxiliary trees of this grammar are shown in Fig. 1. The adjunction constraint on the nodes of the auxiliary tree β is the following—adjunction is allowed only at the child (labelled S) of the root of β.

Interestingly, the sparse version is an order of magnitude faster than the ordinary version for strings of length greater than 7:

String	Answer	Speedup
abc	*Yes*	3.1
aabbcc	*Yes*	6.1
aabcabc	*No*	8.0
abacabac	*No*	11.7
aaabbbccc	*Yes*	11.4

The speedups obtained in the above implementation results suggest that, even in practice, better parsing algorithms can be obtained through the use of matrix multiplication techniques.

9. CONCLUSIONS

In this paper we have presented an $O(M(n^2))$ time algorithm for parsing TALs, n being the length of the input string. We have also demonstrated with our implementation work that matrix multiplication techniques can help us obtain efficient parsing algorithms. We also note here that *Compute Nodes* can easily be modified to handle adjunction constraints.

REFERENCES

1. D. Coppersmith and S. Winograd, Matrix multiplication via arithmetic progressions, *in* "Proc. 19th Annual ACM Symposium on Theory of Computing, 1987," pp. 1–6; *J. Symbolic Comput.* **9** (1990), 251.
2. S. L. Graham, M. A. Harrison, and W. L. Ruzzo, On line context free language recognition in less than cubic time, *in* "Proc. ACM Symposium on Theory of Computing, 1976," pp. 112–120.
3. A. K. Joshi, An introduction to tree adjoining grammars, *in* "Mathematics of language" (A. Manaster-Ramer, Ed.), Benjamins, Amsterdam, 1987.
4. A. K. Joshi, L. S. Levy, and M. Takahashi, Tree adjunct grammars, *J. Comput. System Sci.* **10**, No. 1 (1975).
5. A. K. Joshi, K. Vijay-Shanker, and D. Weir, The convergence of mildly context-sensitive grammar formalisms, *in* "Foundational Issues of Natural Language Processing" pp. 31–81, MIT Press, Cambridge, MA, 1991.
6. A. Kroch and A. K. Joshi, "Linguistic Relevance of Tree Adjoining Grammars," Technical Report MS-CS-85-18, Department of Computer and Information Science, University of Pennsylvania, 1985.
7. M. Palis, S. Shende, and D. S. L. Wei, An optimal linear time parallel parser for tree adjoining languages, *SIAM J. Comput.* (1990).
8. B. H. Partee, A. Ter Meulen, and R. E. Wall, "Studies in Linguistics and Philosophy," Vol. 30, Kluwer Academic, Amsterdam, 1990.
9. S. Rajasekaran, Tree adjoining language parsing in $o(n^6)$ time, *SIAM J. Comput.* **25**, No. 4 (1996), 862–873.
10. G. Satta, Tree adjoining grammar parsing and Boolean matrix multiplication, *in* "31st Meeting of the Association for Computational Linguistics, 1993."
11. G. Satta, personal communication, September 1993.
12. Y. Schabes and A. K. Joshi, An Earley-type parsing algorithm for tree adjoining grammars, *in* "Proc. 26th Meeting of the Association for Computational Linguistics, 1986."
13. L. G. Valiant, General context-free recognition in less than cubic time, *J. Comput. System Sci.* **10** (1975), 308–315.
14. K. Vijay-Shanker, "A Study of Tree Adjoining Grammars," Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania.
15. K. Vijay-Shanker and A. K. Joshi, Some computational properties of tree adjoining grammars, *in* "Proc. 11th Meeting of the Association for Computational Linguistics, 1986."