ANALYSIS OF TOMITA'S ALGORITHM FOR GENERAL CONTEXT-FREE PARSING[1]

JAMES R. KIPPS (KIPPS@RAND-UNIX.ARPA)
The RAND Corporation, Santa Monica, CA 90406

Abstract. A variation on Tomita's algorithm is analyzed in regards to its time and space complexity. It is shown to have a general time bound of $O(n^{\bar{\rho}+1})$, where n is the length of the input string and $\bar{\rho}$ is the length of the longest production. A modified algorithm is presented in which the time bound is reduced to $O(n^3)$. The space complexity of Tomita's algorithm is shown to be proportional to n^2 in the worst case and is changed by at most a constant factor with the modification. Empirical results are used to illustrate the trade off between time and space on a simple example. A discussion of two subclasses of context-free grammars that can be recognized in $O(n^2)$ and $O(n)$ is also included.

1. INTRODUCTION

Algorithms for general context-free (CF) parsing, e.g., Earley's algorithm (Earley, 1968) and the Cocke-Younger-Kasami algorithm (Younger, 1967), are necessarily less efficient than algorithms for restricted CF parsing, e.g., the LL, operator precedence, and LR algorithms (Aho and Ullman, 1972), because they must simulate a multi-path, nondeterministic pass over their inputs using some form of search, typically, goal-driven. While many of the general algorithms can be shown to theoretically perform as well as the restricted algorithms on a large subclass of CF grammars, due to the inefficiency of goal expansion the general algorithms have not been widely used as practical parsers for programming languages.

A basic characteristic shared by many of the best known general algorithms is that they are top-down parsers. Recently, Tomita (1985) introduced an algorithm for general CF parsing defined as a variation on standard LR parsing, i.e., a table-driven, bottom-up parsing algorithm. The benefit of this approach is that it eliminates the need to expand alternatives of a nonterminal at parse time (what Earley refers to as the predictor operation). For Earley's algorithm, the predictor operation is one of two $O(n^2)$ components. While eliminating this operation would not change the algorithm's time bound of $O(n^3)$, it could be significant to practical parsing. It is of interest to analyze the complexity of Tomita's algorithm and see how it compares.

Upon examination, Tomita's algorithm is found to have a general time complexity of $O(n^{\bar{\rho}+1})$, where n is as before and $\bar{\rho}$ is the length of the longest production in the source grammar. Thus, this algorithm achieves $O(n^3)$ for grammars in Chomsky normal form (Chomsky, 1959) but has potential for being worse when productions are of unrestricted lengths. In this paper, I present a modification of Tomita's algorithm that allows it to run in time proportional to n^3 for grammars with productions of arbitrary lengths.

2. TOMITA'S ALGORITHM

The following is an informal description of Tomita's algorithm as a recognizer; familiarity with standard LR parsing is assumed. Tomita views his algorithm as a variation on standard LR parsing. The algorithm takes a shift-reduce approach, using an extended LR parse table to guide its actions. The extended parse table records shift/reduce and reduce/reduce conflicts as multiple action entries, so the parse table can no longer be used for strictly deterministic parsing. The algorithm simulates a nondeterministic parse with pseudo-parallelism. It scans an input string $x_1 \cdots x_n$ from left to right, following all paths in a breath-first manner and merging like subpaths when possible to avoid redundant computations.

[1] This work was supported by the Defense Advanced Research Projects Agency, under contract number MDA-903-85-C-0030.

The algorithm operates by maintaining a number of parsing processes in parallel. Each *process* has a stack, scans the input string from left-to-right, and behaves basically the same as the single parsing process in standard LR parsing. Each stack element is labeled with a parse state and points to its parent, i.e., the previous element on a process's stack. The top-of-stack is the *current state* of a process.

Each process does not actually maintain its own separate stack. Rather, these "multiple" stacks are represented using a single directed acyclic (but reentrant) graph called a *graph-structured stack*. Each stack element corresponds to a vertex of the graph. Each leaf of the graph acts as a distinct top-of-stack to a process. The root of the graph acts as a common bottom-of-stack. The edge between a vertex and its parent is directed toward the parent. Because of the reentrant nature of the graph (as explained below), a vertex may have more than one parent.

The leaves of the graph grow in stages. Each stage U_i corresponds to the ith symbol x_i from the input string. After x_i is scanned, the leaves in stage U_i are in a one-to-one correspondence with the algorithm's *active* processes, where each process references a distinct leaf of the graph and treats that leaf as its current state. Upon scanning x_{i+1}, an active process can either (1) add an additional leaf to U_i, or (2) add a leaf to U_{i+1}. Only processes that have added leaves to U_{i+1} will be active when x_{i+2} is scanned.

In general, a process behaves in the following manner. On x_i, each active process (corresponding to the leaves in U_{i-1}) executes the entries in the action table for x_i given its current state. When a process encounters multiple actions, it *splits* into several processes (one for each action), each sharing a common top-of-stack. When a process encounters an error entry, the process is discarded (i.e., its top-of-stack vertex sprouts no leaves into U_i by way of that process). All processes are synchronized, scanning the same symbol at the same time. After a process shifts on x_i into U_i, it waits until there are no other processes that can act on x_i before scanning x_{i+1}.

The Shift Action. A process (with top-of-stack vertex v) shifts on x_i from its current state s to some successor state s' by

(1) creating a new leaf v' in U_i labeled s';

(2) placing an edge from v' to its top-of-stack v (directed towards v); and

(3) making v' its new top-of-stack vertex (in this way changing its current state).

Any successive process shifting to the same state s' in U_i is *merged* with the existing process to form a single process whose top-of-stack vertex has multiple parents, i.e., by placing an additional edge from the top-of-stack vertex of the existing process in U_i to the top-of-stack vertex of the shifting process. The merge is done because, individually, these processes would behave in exactly the same manner until a reduce action removed the vertices labeled s' from their stacks. Thus, merging avoids redundant computation. Merging also ensures that each leaf in any U_i will be labeled with a distinct parse state, which puts a finite upper-bound on the possible number of active processes and, thus, limits the size of the graph-structured stack.

The Reduce Action. A process executes a reduce action on a production p by following the chain of parent links down from its top-of-stack vertex v to the ancestor vertex from which the process began scanning for p earlier, essentially "popping" intervening vertices off its stack. Since merging means a vertex can have multiple parents, the reduce operation can lead back to multiple ancestors. When this happens, the process is again split into separate processes (one for each ancestor). The ancestors will correspond to the set of vertices at a distance $\bar{p}$ from v, where $\bar{p}$ equals the number of symbols in the right-hand side of the pth production. Once reduced to an ancestor, a process shifts to the state s' indicated in the goto table for D_p (the nonterminal on the left-hand side of the pth production) given the ancestor's state. A process shifts on a nonterminal much as it does a terminal, with the exception that the new leaf is added to U_{i-1} rather than U_i; a process can only enter U_i by shifting on x_i.

The algorithm begins with a single initial process whose top-of-stack vertex is the root of the graph-structured stack. It then follows the general procedure outlined above for each symbol in the input string, continuing until there are either no leaves added to U_i (i.e., no more active processes), which denotes *rejection*, or a process executes the accept action on scanning the $n+1$st input symbol '$\dashv$,' which denotes *acceptance*.

3. ANALYSIS OF TOMITA'S ALGORITHM

In this section, a formal definition of Tomita's algorithm is presented as a recognizer for input string $x_1 \cdots x_n$. This definition is understood to be with respect to an extended LR parse table (with start state S_0) constructed from a source grammar G.

Notation. The productions of G are numbered arbitrarily $1, \cdots, d$, where each production is of the form $D_p \rightarrow C_{p1} \cdots C_{p\bar{p}}$ $(1 \leq p \leq d)$ and where $\bar{p}$ is the number of symbols on the right-hand side of the pth production.

Definition. The entries of the extended LR parse table are accessed with the functions ACTIONS and GOTO.

- ACTIONS(s,x) returns a set of actions from the action table along the row of state s under the column labeled x. This set will contain no more than one of a shift action **sh***s'* (shift to state s) *or* an accept action **acc**; it may contain any number of reduce actions **re***p* (reduce using production p). An empty action set corresponds to an error.
- GOTO(s,D_p) returns a state s' from the goto table along the row of state s under the column labeled with nonterminal D_p.

Definition. Each *vertex* of the graph-structured stack is a triple $\langle i, s, l \rangle$, where i is an integer corresponding to the ith input symbol scanned (at which point the vertex was created as a leaf), s is a parse state (corresponding to a row of the parse table), and l is a set of parent vertices. The *processes* described in the last section are represented implicitly by the vertices in successive U_i's. The root of the graph-structured stack, and hence the initial process, is the vertex $\langle 0, S_0, \emptyset \rangle$.

The Recognizer. The recognizer is a function of one argument REC($x_1 \cdots x_n$). It calls upon the functions SHIFT(v,s) and REDUCE(v,p). SHIFT(v,s) either adds a new leaf to U_i labeled with parse state s whose parent is vertex v or merges vertex v with the parents of an existing leaf. REDUCE(v,p) executes a reduce action from vertex v using production p. REDUCE calls upon the function ANCESTORS(v,$\bar{p}$), which returns the set of all ancestor vertices a distance of $\bar{p}$ from vertex v. These functions, which vary somewhat from the formal definition given in Tomita (1985),[2] are defined in Figure 3.1.

In REC, [1] adds the end-of-sentence symbol '$\dashv$' to the end of the input string; [2] initializes the root of the graph-structured stack; [3] iterates through the symbols of the input string. On each symbol x_i, [4] processes the vertices (denoting the active processes) of successive U_{i-1}'s, adding each vertex to P to signify that it has been processed. On each vertex v, [5] executes the shift, reduce, and accept actions from the action table according to v's state s. After processing the vertices in U_{i-1}, [6] checks whether a vertex was added to U_i, ensuring that at least one process is still active before scanning x_{i+1}.

In SHIFT, [7] shifts a process into state s by adding a vertex to U_i labeled s. If a vertex labeled s already exists, v is added to its parents, merging processes; otherwise, a new vertex is created with a single parent v.

---

[2] Tomita's functions REDUCE and REDUCE-E have been collapsed into a single REDUCE function; also added were the ANCESTORS function and the concept of a "clone" vertex. While these changes do not alter Tomita's algorithm significantly, they were helpful in developing ideas about its complexity.

```
    REC(x_1 ··· x_n)
[1]   let x_{n+1} := ⊣
      let U_i := [ ]   (0 ≤ i ≤ n)
[2]   let U_0 := [⟨0, S_0, ∅⟩]
[3]   for i from 1 to n + 1
        let P := [ ]
[4]     for ∀v = ⟨i − 1, s, l⟩ s.t. v ∈ U_{i−1}
          let P := P ∘ [v]
[5]       if ∃'sh s'' ∈ ACTIONS(s, x_i), SHIFT(v, s')
          for ∀'re p' ∈ ACTIONS(s, x_i), REDUCE(v, p)
          if 'acc' ∈ ACTIONS(s, x_i), accept
[6]     if U_i is empty, reject
    SHIFT(v, s)
[7]   if ∃v' = ⟨i, s, l⟩ s.t. v' ∈ U_i
        let l := l ∪ {v}
      else
        let U_i := U_i ∘ [⟨i, s, {v}⟩]
    REDUCE(v, p)
[8]   for ∀v_1' = ⟨j', s', l_1'⟩ s.t. v_1' ∈ ANCESTORS(v, p̄)
        let s'' := GOTO(s', D_p)
[9]     if ∃v'' = ⟨i − 1, s'', l''⟩ s.t. v'' ∈ U_{i−1}
[10]      if v_1' ∈ l''
            do nothing (ambiguous)
          else
[11]      if ∃v_2' = ⟨j', s', l_2'⟩ s.t. v_2' ∈ l''
            let v_c'' := ⟨i − 1, s'', {v_1'}⟩
            for ∀'re p' ∈ ACTIONS(s'', x_i), REDUCE(v_c'', p)
          else
[12]        let l'' := l'' ∪ {v_1'}
[13]        if v'' ∈ P
              let v_c'' := ⟨i − 1, s'', {v_1'}⟩
              for ∀'re p' ∈ ACTIONS(s'', x_i), REDUCE(v_c'', p)
        else
[14]      let U_{i−1} := U_{i−1} ∘ [⟨i − 1, s'', {v_1'}⟩]
    ANCESTORS(v = ⟨j, s, l⟩, k)
[15]  if k = 0
        return({v})
      else
        return(⋃_{v'∈l} ANCESTORS(v', k − 1))
```

Fig. 3.1—Tomita's Algorithm

In REDUCE, [8] iterates through the ancestor vertices a distance of $\bar{p}$ from v, setting s'' to the state indicated in the goto table under D_p given the ancestor's state s'. Each ancestor vertex v_1' is shifted into U_{i-1} on s''. [9] checks whether such a vertex v'' already exists. (If not, [14] adds a vertex labeled s'' to U_{i-1}.) If v'' does already exist, [10] checks that a shift from the current ancestor v_1' has not already been made. (If it has, then some segment of the input string has been recognized as an instance of the same nonterminal D_p in two different ways, and the current derivation can be discarded as ambiguous; otherwise, v_1' is merged with the parents of the existing vertex.) Before merging, [11] checks whether v_1' is a "clone" vertex, created by [13] in an earlier call to REDUCE (as described below). If v_1' is not a clone, [12] adds it to the parents of v'', merging processes. [13] checks if v'' has already been processed. If so, then it missed any reductions through v_1'. To correct this, v'' is "cloned" into v_c'' (i.e., a variant on v'' with a single parent v_1'), and all reduce actions executed on v'' are now executed on v_c''.

Returning to [11], when reducing on a null production, ANCESTORS will return a clone vertex as the ancestor of itself. If a variant v_2' of v_1' already exists in the parents of v'', then v_1' is a clone of v_2'. At this point v'' has already been processed, meaning that there could still be reductions that have not gone through the single parent of v_1'. To correct this, v'' is again cloned, and all reduce actions executed on v'' are executed on the new clone v_c''.

Finally, in ANCESTORS, [15] recursively descends the chain of parents of vertex v, returning the set of vertices a distance of k from v.

The General Case. Tomita's algorithm is an $O(n^{\bar{\rho}+1})$ recognizer in general, where $\bar{\rho}$ is the greatest $\bar{p}$ in G. The reasons for this are as follows:

(a) Since each vertex in U_i must be labeled with a distinct parse state, the number of vertices in any U_i is bounded by the number of parse states;

(b) The number of parents l of a vertex $v = \langle i, s, l \rangle$ in U_i is proportional to i. Because processes could have begun scanning for some production p in each U_j $(j \leq i)$, a process in U_i could reduce using p and split into $\sim i$ processes (one for each ancestor in a distinct U_j). Then each process could shift on D_p to the same state in U_i and, thus, that vertex could have $\sim i$ parents;

(c) For each x_{i+1}, SHIFT will be called a bounded number of times (at most once for each vertex in U_i). SHIFT executes in a bounded number of steps.

(d) For each x_{i+1} and production p, REDUCE(v,p) will be called a bounded number of times in REC, and REDUCE(v_c'',p) (the recursive call to REDUCE) will be called no more than $\sim i$ times. The reason for the former is the same as in (c). The latter is due to the conditions on the recursive call, which maintain that it can be called no more than once for each parent of a vertex in U_i, of which there are at most proportional to i;

(e) REDUCE(v,p), because at most $\sim i$ vertices can be returned by ANCESTORS, executes in $\sim i$ steps plus the steps needed to execute ANCESTORS.

(f) ANCESTORS(v,$\bar{p}$) executes in $\sim i^{\bar{p}}$ steps in the worst case. While at most $\sim i$ processes could have begun scanning for p, the number of paths by which any single process could reach v in U_i is bounded by the number of ways the intervening input symbols can be partitioned among the $\bar{p}$ vocabulary symbols in the right-hand side of production p. For a process that started from U_j $(j \leq i)$, the number of paths to v in U_i in the recognition of p can be proportional to

$$\sum_{m_1=j}^{0} \sum_{m_2=m_1}^{0} \cdots \sum_{m_{\bar{p}-1}=m_{\bar{p}-2}}^{0} 1.$$

Summing from $j = 0, \cdots, i$ gives a closed form proportional to $i^{\bar{p}}$. ANCESTORS(v_c'',$\bar{p}$), where $v_c'' = \langle i, s\{v'\} \rangle$, executes in $\sim i^{\bar{p}-1}$ steps because there is that many ways $\sim i$ ancestor vertices could reach v' and only one way v' could reach v_c'';

(g) The worst case time bound is dominated by the time spent in ANCESTORS, which can be added to the time spent in REDUCE. Since REDUCE(v,p), with a bound $\sim i^{\bar{p}}$, is called only a bounded number of times, and REDUCE(v_c'',p), with a time bound of $\sim i^{\bar{p}-1}$, is called at most $\sim i$ times, the worst case time to process any x_i is $\sim i^{\bar{p}}$, for each $i = 0, \cdots, n+1$ and longest production ρ;

(h) Summing from $i = 0, \cdots, n+1$ gives REC a general time bound proportional to $n^{\bar{\rho}+1}$.

As a result, this bound indicates that Tomita's algorithm only belongs to complexity class $O(n^3)$ when applied to grammars in Chomsky normal form (CNF)[3] or some other equally truncated notation.

[3] In CNF, productions can have one of two forms, A → BC or A → a; thus, the length of the longest production is at most 2.

Although any CF grammar can be automatically converted to CNF (Hopcraft and Ullman, 1979), extracting useful information from derivation trees produced by such grammars would be time consuming at best (if possible at all).

4. MODIFYING TOMITA'S ALGORITHM FOR N^3 TIME

In this section, Tomita's algorithm is made an $O(n^3)$ recognizer for CF grammars with productions of arbitrary length. Essentially, the modifications are to the ANCESTORS function. ANCESTORS is the only function that forces us to use $i^{\bar{p}}$ steps. It is interesting to note that ANCESTORS can take this many steps even though it returns at most $\sim i$ ancestor vertices and even though there are at most $\sim i$ intervening vertices and edges between a vertex in U_i and its ancestors. This indicates that ANCESTORS can recurse down the same subpaths more than once. The efficiency of ANCESTORS and Tomita's algorithm can be improved by eliminating this redundancy.

The modification described here turns ANCESTORS into a table look-up function. Assume there is a two-dimensional "ancestors" table. One dimension is indexed on the vertices in the graph-structured stack, and the other is indexed on integers $k = 1, \cdots, \bar{p}$, where $\bar{p}$ equals the greatest $\bar{p}$. Each entry (v, k) is the set of ancestor vertices a distance of k from vertex v. Then, ANCESTORS(v,k) returns the (at most) $\sim i$ ancestor at (v, k) in ~ 1 steps. Of course, the table must be filled dynamically during the recognition process, so the time expended in this task must also be determined.

In Figure 4.1, ANCESTORS is defined as a table look-up function that dynamically generates table entries the first time they are requested. In this definition, the ancestor table is represented by changing the parent field l of a vertex $v = \langle i, s, l \rangle$ from a set of parent vertices to an ancestor field a. For a vertex $v = \langle i, s, a \rangle$, a consists of a set of tuples $\langle k, l_k \rangle$, such that l_k is the set of ancestor vertices a distance of k from v.

Figure 4.1 illustrates the necessary modifications made to the definitions of Figure 3.1; the function REC is unchanged. In SHIFT, [1] adds a vertex to U_i labeled s. If such a vertex does not already exist, one is created whose ancestor field records that v is the ancestor vertex at a distance of 1; otherwise, v is added to the other distance-1 ancestors.

In REDUCE, [2] iterates through the ancestor vertices a distance of $\bar{p}$ from v, setting s'' to the state indicated in the goto table under D_p given the ancestor's state s'. Each ancestor vertex v_1' is shifted into U_{i-1} on s''. [3] checks whether such a vertex v'' already exists. (If not, [10] will add a vertex labeled s'' to U_{i-1}.) If v'' does already exist, [4] checks that a shift from the current ancestor v_1' has not already been made. If it has, then v_1' can be discarded as ambiguous; if not, then v_1' can be merged with the other ancestors a distance of 1 from v''. Before merging, [5] checks whether v_1' is a clone vertex as described in Section 3. If v_1' is a clone (the result of being reduced on a null production), v'' is again cloned, and all reduce actions executed on v'' are executed on the new clone v_c''. After the application of REDUCE, [6] updates the ancestor table stored in v'' to record entries made in the ancestor field a_c'' of the clone when $k \geq 2$. Otherwise, if v_1' is not a clone, [7] adds it to the distance-1 ancestors of v'', merging processes. [8] checks if v'' has already been processed. If so, then it missed any reductions through v_1', so v'' is cloned into v_c'' and all reduce actions executed on v'' are now executed on v_c''. After reducing v_c'', [9] updates the ancestor table stored in v'' to record entries made in the ancestor field a_c'' of the clone when $k \geq 2$.

In ANCESTORS, [11] searches a (the portion of the ancestor table stored with v) for ancestor vertices at a distance of k from v. If an entry exists, those vertices are returned; if not, [12] calls ANCESTORS recursively to generated those vertices and, before returning the generated vertices, records them in the ancestor field of v.

The question now becomes how much time is spent filling the ancestor table. For ANCESTORS(v,$\bar{p}$), time is bounded in the worst case by $\sim i^2$ steps, while for ANCESTORS(v_c'',$\bar{p}$), it is bounded by $\sim i$ steps. In general, ANCESTORS(v,k), where $v = \langle i, s, a \rangle$, will take $\sim i$ steps to execute the first time it is called (one for each recursive call to ANCESTORS(v',$k-1$), where

```
    SHIFT(v,s)
[1]   if ∃v' = ⟨i,s,a⟩ s.t. v' ∈ U_i ∧ ⟨1,l⟩ ∈ a,
        let l := l ∪ {v}
      else
        let U_i := U_i ∘ [⟨i,s,[⟨1,{v}⟩]⟩]

    REDUCE(v,p)
[2]   for ∀v_1' = ⟨j',s',a_1'⟩ s.t. v_1' ∈ ANCESTORS(v,p̄)
        let s'' := GOTO(s',D_p)
[3]     if ∃v'' = ⟨i-1,s'',a''⟩ s.t. v'' ∈ U_{i-1} ∧ ⟨1,l''⟩ ∈ a''
[4]       if v_1' ∈ l''
            do nothing (ambiguous)
          else
[5]       if ∃v_2' = ⟨j',s',a_2'⟩ s.t. v_2' ∈ l''
            let v_c'' := ⟨i-1,s'',a_c''⟩ s.t. a_c'' = [⟨1,{v_1'}⟩]
            for ∀'re p' ∈ ACTIONS(s'',x_i), REDUCE(v_c'',p)
[6]         let l_{k_1} := l_{k_1} ∪ l_{k_2} s.t. ⟨k,l_{k_1}⟩ ∈ a'' ∧ ⟨k,l_{k_2}⟩ ∈ a_c'' (k≥2)
          else
[7]         let l'' := l'' ∪ {v_1'}
[8]         if v'' ∈ P
              let v_c'' := ⟨i-1,s'',a_c''⟩ s.t. a_c'' = [⟨1,{v_1'}⟩]
              for ∀'re p' ∈ ACTIONS(s'',x_i), REDUCE(v_c'',p)
[9]           let l_{k_1} := l_{k_1} ∪ l_{k_2} s.t. ⟨k,l_{k_1}⟩ ∈ a'' ∧ ⟨k,l_{k_2}⟩ ∈ a_c'' (k≥2)
        else
[10]      let U_{i-1} := U_{i-1} ∘ [⟨i-1,s'',{v_1'}⟩]

    ANCESTORS(v = ⟨j,s,a⟩,k)
[11]  if k = 0,
        return({v})
      else
      if ∃⟨k,l_k⟩ ∈ a,
        return(l_k)
      else
[12]    let l_k := ⋃_{v'∈l_1|⟨1,l_1⟩∈a} ANCESTORS(v',k-1)
        let a := a ∪ {⟨k,l_k⟩}
        return(l_k)
```

Fig. 4.1—Modified Algorithm

$v' \in l_1$ and $\langle 1, l_1 \rangle \in a$) and ~ 1 steps thereafter. When ANCESTORS(v,$\bar{p}$) is executed, there are $\sim i$ such "virgin" vertices between v and its ancestors, and so this call can execute $\sim i^2$ steps in the worst case. ANCESTORS(v_c'',$\bar{p}$) is called only after the call to ANCESTORS(v,$\bar{p}$) has been made, where v_c'' is a clone of v. This means that $\sim i$ of the vertices between v' and the ancestor vertices have been processed, so the call to ANCESTORS(v',$\bar{p}-1$) could take at most proportional to i steps for each of a bounded number of intervening vertices.

Given this, the upper bound on the number of steps that can be executed by the total calls on REDUCE for a given x_i is proportional to i^2. Summing from $i = 0, \cdots, n+1$ gives $\sim n^3$ steps as the worst case upper bound on the execution time of the modified algorithm.

5. SPACE BOUNDS

The space complexity of Tomita's algorithm as it appears in Section 3 is proportional to n^2 in the worst case. This is because the space requirements of the algorithm are bounded by the requirements of the graph-structured stack. There are a bounded number of vertices in each U_i of the graph-structured stack, and each vertex can have at most $\sim i$ parents. Summing again from $i = 0, \cdots, n+1$ gives $\sim n^2$ as the worst case space requirement for the graph-structured stack.

With the modification of Section 4, the space requirements of the graph-structured stack are increased by at most a constant factor of n^2. This is because the modification replaces the $\sim i$ parents of a vertex in U_i with at most $\sim \bar{\rho} i$ entries in the ancestors field. So, for a vertex $v = \langle i, s, a \rangle$ *s.t.* $v \in U_i$, the ancestors field a will be a subset of $\{(c, l_c) | 1 \leq c \leq \bar{\rho}\}$ where $|l_c| \simeq i$. Summing from $i = 0, \cdots, n+1$ gives $\sim \bar{\rho} n^2$ or $\sim n^2$ still as a worst case upper bound on space.

6. EMPIRICAL RESULTS

The variation on Tomita's algorithm presented in Section 3 and the modified algorithm presented in Section 4 have both been implemented in C. The graphs in figures 6.1 and 6.2 show empirical results comparing the time and space requirements of both implementations. Each time/space graph set corresponds to the grammars, $G1$, $G2$, and $G3$, which are dominated by productions of length 2, 3 and 4.

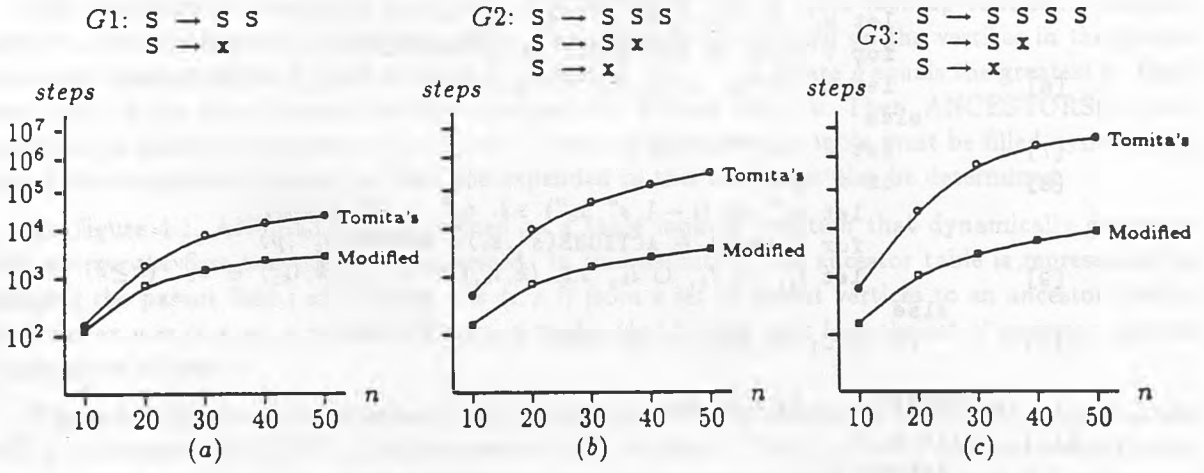

Fig. 6.1—Comparison of Time Complexity

The time graphs in Figure 6.1 measure the number of calls to SHIFT, REDUCE, and ANCESTORS. The input sentences are strings of x's of length 10 to 50. Our analysis of time complexity predicts that the modified algorithm will take roughly the same number of steps for each grammar, while the steps taken by Tomita's algorithm will increase as a function of the length of the dominant production. The empirical data gathered from our two implementations agrees with this prediction. When $n = 50$, the modified algorithm took $\sim$ 7000 steps for grammar $G1$ in Figure 6.1 (a), $\sim$ 6000 for $G2$ in Figure 6.1 (b), and $\sim$ 10000 for $G3$ in Figure 6.1 (c); Tomita's algorithm took $\sim$ 44,000 steps for grammar $G1$, $\sim$ 660,000 for $G2$, and $\sim$ 7,300,000 for $G3$.

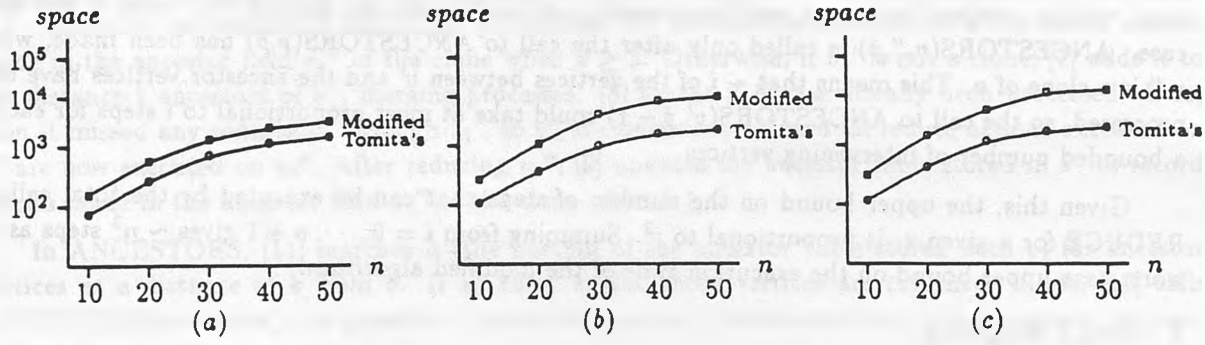

Fig. 6.2—Comparison of Space Complexity

The space graphs in Figure 6.2 measure the number of edges required by the graph-structured stack (in Tomita's algorithm) and the length of entries in the ancestors table (in the modified algorithm). The number of vertices required is the same for both algorithms and is not counted; space that can be reclaimed before scanning successive x_i's is also not counted. Our analysis of space complexity

predicts that Tomita's algorithm will require $\sim n^2$ space and that the modified algorithm will require at most a factor of n^2 additional space. The empirical evidence also agrees with this prediction. The space requirements of the modified algorithm differs from Tomita's algorithm by a factor of ~ 2.1 for grammar $G1$ in Figure 6.2 (a), ~ 3.9 for $G2$ in Figure 6.2 (b), and ~ 4.7 for $G3$ in Figure 6.2 (c).

7. LESS THAN N^3 TIME

Several of the better known general CF algorithms have been shown to run in less than $O(n^3)$ time for certain subclasses of grammars. Therefore, it is of interest to ask if Tomita's algorithm, as well as the modified version presented here, can also recognize some subclasses of CF grammars in less than $O(n^3)$ time. In this section, I informally describe two such subclasses that can be recognized in $O(n^2)$ and $O(n)$ time, respectively. The arguments for their existence parallel those given by Earley (1968), where they are formally specified.

Time $O(n^2)$ Grammars. ANCESTORS is the only function that forces us to use $\sim i^\delta$ steps in Tomita's algorithm and $\sim i^2$ steps in the modified algorithm. We determined that this could happen when a ancestor vertex v' from U_j ($j \leq i$) reached the reducing vertex v in U_i by more than a single path, i.e., the symbols $x_j \cdots x_i$ were derived from a nonterminal D_p in more than one way, indicating that grammar G is ambiguous. If G were unambiguous, then there would be at most one path from a given v' to v. This means that the bounded calls to ANCESTORS(v,$\bar{p}$) can take at most $\sim i$ steps and that ANCESTORS(v_c'',$\bar{p}$) can take at most a bounded number of steps. The first observation is due to the fact that there are $\sim i$ ancestor vertices that can be reached in only one way. Similarly, the second observation is due to the fact that if ANCESTORS(v_c'',$\bar{p}$) took $\sim i$ steps, returning $\sim i$ ancestors, and was called $\sim i$ times, then some ancestor vertices must have shifted into U_i in more than one way, which would be a contradiction, meaning grammar G must be ambiguous. So, if the grammar is unambiguous, then the total time spent in REDUCE for any x_i is $\sim i$ and the worst case time bound for the Tomita's algorithm is $O(n^2)$. A similar result is true for the modified algorithm.

Time $O(n)$ Grammars. In his thesis, Earley (1968) points out that " ... *for some grammars the number of states in a state set can grow indefinitely with the length of the string being recognized. For some others there is a fixed bound on the size of any state set. We call the latter grammars bounded state grammars.*" While Earley's "states" have a different meaning than states in Tomita's algorithm, a similar phenomena occurs, i.e., for the bounded state grammars there is a fixed bound on the number of parents any vertex can have. In Tomita's algorithm, bounded state grammars can be recognized in time $O(n)$ for the following reason. No vertex can have more than a bounded number of ancestors (if otherwise, then $\sim i$ vertices could be added to the parents of some vertex in U_i, proving by contradiction that the grammar is not bounded state). This means that the ANCESTORS function can execute in a bounded number of steps. Likewise, REDUCE can only be called a bounded number of times. Summing over the x_i gives us an upper bound $\sim n$. Again, a similar result is true for the modified algorithm. Interestingly enough, Earley states that almost all LR(k) grammars are bounded state, as well, which suggests that Tomita's algorithm, given k-symbol look ahead, should perform with little loss of efficiency as compared to a standard LR(k) algorithm when the grammar is "close" to LR(k). Earley also points out that not all bounded state grammars are unambiguous; thus, there are non-LR(k) grammars for which Tomita's algorithm can perform with LR(k) efficiency.

8. CONCLUSION

The results in this paper support in part Tomita's claim (1985) of efficiency for his algorithm. With the modification introduced here, Tomita's algorithm is shown to be in the same complexity class as existing general CF algorithms. These results also give support to his claim that his algorithm should run with near LR(k) efficiency for near LR(k) grammars.

It should be noted that while the modification to Tomita's algorithm has theoretic interest it would detract from a practical parser. Realistic grammars are constrained by the fact that they must be human-readable. Since human-readable grammars should never realize the worst-case $O(n^{\delta+1})$ time

bound of Tomita's algorithm, the benefits of the ancestors table in the modified algorithm would not balance out its overhead cost. In this regard, the modified algorithm should not be viewed as an "improvement" over Tomita's algorithm but as a means of illustrating its place among other general CF algorithms.

The variation on Tomita's algorithm described in this paper, as well as the modified algorithm, have been implemented in both LISP and C at The RAND Corporation. The LISP implementation (Kipps, 1988) is distributed with ROSIE (Kipps et al., 1987), a language for applications in artificial intelligence with a highly ambiguous English-like syntax. The C implementation is part of the RAND Translator-Generator project, which is developing a "next generation" YACC[4] for non-LR(k) languages.

REFERENCES

Aho, A.V., J.D. Ullman, *The Theory of Parsing, Translation and Compiling*, Prentice-Hall, Englewood Cliffs, NJ, 1972.

Chomsky, N., "On Certain Formal Properties of Grammars," in *Information and Control*, vol. 2, no. 2, pp. 137-167, 1959.

Earley, J., *An Efficient Context-Free Parsing Algorithm*, Ph.D. Thesis, Computer Science Dept., Carnegie-Mellon University, Pittsburg, PA, 1968.

Hopcraft, J.E., J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

Kipps, J.R., B. Florman, H.A. Sowizral, *The New ROSIE Reference Manual and User's Guide*, R-3448-DARPA, The RAND Corporation, 1987.

Kipps, J.R., "A Table-Driven Approach to Fast Context-Free Parsing," N-2841-DARPA, The RAND Corporation, 1988.

Knuth, D.E., "On the Translation of Languages from Left to Right," *Information and Control*, vol. 8, pp. 607-639, 1965.

Johnson, S.C., "YACC—Yet Another Compiler Compiler," CSTR 32, Bell Laboratories, Murray Hill, NJ, 1975.

Tomita, M., *An Efficient Context-Free Parsing Algorithm for Natural Languages and Its Applications*, Ph.D. Thesis, Computer Science Dept., Carnegie-Mellon University, Pittsburg, PA, 1985.

Younger, D.H., "Recognition and Parsing of Context-Free Languages in Time n^3," in *Information and Control*, vol. 10, no. 2, pp. 189-208, 1967.

[4] YACC (Johnson, 1975) is a parser-generator for LALR(1) languages.