PAPER Special Issue on Natural Language Processing and Understanding

A Family of Generalized LR Parsing Algorithms Using Ancestors Table

Hozumi TANAKA†, *Member*, K.G. SURESH† *and* Koichi YAMADA†, *Nonmembers*

SUMMARY A family of new generalized LR parsing algorithms are proposed which make use of a set of ancestors tables introduced by Kipps [4]. As Kipps's algorithm does not give us a method to extract any parsing results, his algorithm is not considered as a practical parser but as a recognizer. In this paper, we will propose two methods to extract all parse trees from a set of ancestors tables in the top vertices of a graph-structured stack. For an input sentence of length n, while the time complexity of the Tomita parser can exceed $O(n^3)$ for some context-free grammars (CFGs), the time complexity of our parser is $O(n^3)$ for any CFGs, since our algorithm is based on the Kipps's recognizer. In order to extract a parse tree from a set of ancestors tables, it takes time in order n^2. Some preliminary experimental results are given to show the efficiency of our parsers over Tomita parser.

key words: *GLR parsing, Graph structure stack, Kipps method, Ancestors table, Complexity*

1. Introduction

The LR(k) parser [5] can parse deterministically and efficiently any input sentences generated by a LR(k) grammar. LR(k) grammars are a subset of context-free grammars (CFGs). Tomita extended the LR(k) parser to handle general CFGs not limited to the Chomsky normal form [12]. The extended algorithm is called the Tomita parser, and is known as one of the most efficient generalized LR (GLR) parsers. Empirically, Tomita's algorithm is faster than Earley's algorithm [2], but there are some CFGs [3] for which the time complexity of Tomita's algorithm is worse than that of Earley's and for general CFGs, the parsing time crosses over $O(n^3)$ for the input sentence of length n [4]. This is because using Tomita's data structure, the graph-structured stack (GSS), during the reduce actions of the LR table, in order to retrieve a set of ancestors vertices, duplicated traversal of the same edges and the access of the same ancestors occur many times.

To avoid the above problem, Kipps introduced a data structure called an ancestors table in which the ancestor vertices are stored [4]. Using only the ancestors tables in the *top* vertices (leaves) of GSS, Kipps algorithm can generate a set of ancestors vertices in constant time without traversing any edge in the GSS, and thus can avoid duplicated traversals of the same edge and the duplicated access of the same ancestors. As a result, Kipps algorithm can give $O(n^3)$ time complexity for any CFGs.

However, as Kipps's algorithm does not give us a way to extract any parse results, it is not considered as a practical parser [7] but as a *recognizer*. In this paper, we propose a family of GLR parsing algorithms (Drit parser and AGLR parser) which can get all parse trees from ancestors table without traversing any edge in GSS, and whose time complexity gives the same n^3 order as that of Kipps algorithm. In order to extract a parse tree from partially parsed informations (a set of drits in case of Drit parser; a set of ancestors tables in case of AGLR), which has been stored during shift and reduce actions, it takes $O(n^2)$ time. For the family of GLR parsing algorithms, when the result of parsing is highly ambiguous, the experiments confirm the possibility of tremendous speed up in the parsing time.

Following Kipps [4], we briefly explain Kipps recognizer in Sect. 2. Section 3 explains the family of new GLR parsing algorithms. Section 4 gives an experimental evaluation of the family of GLR parsing algorithms showing evidence that our GLR parser is efficient than Tomita parser. In Sect. 5, we give the tree generation algorithm for AGLR parser, and we conclude with Sect. 6.

2. An Overview of Kipps Recognition Algorithm

Figure 1 shows a schematic example of a GSS. Here v_i represents a vertex (the vertex v_a is the root of GSS and v_g is a leaf or top vertex) and w_i represents i-th input word. The leaves of a GSS grows in stages. At each stage U_i the i-th word w_i of the input sentence is processed with the help of the next look-ahead word w_{i+1}. The vertex v_g in stage U_6 covers w_6 and w_5 w_6, v_f in stage U_5 covers $w_4 w_5$. In the same way, v_e in stage U_4 covers w_4 and $w_3 w_4$.

In Tomita's algorithm*, the same ancestors and/or the same edges might be accessed many times. For example, in the GSS shown in Fig. 1, in order to retrieve an ancestor vertex, say v_d, at a distance 2 from the top-of-stack v_g, we have to traverse two paths from v_g to v_d, namely v_g–v_f–v_d and v_g–v_e–v_d, resulting in accessing the same one ancestor v_d two times. In general, the ancestors at a distance of q from a leaf in the stage U_i will be obtained by traversing every edge from the leaf

Manuscript received November 30, 1992.
Manuscript revised August 1, 1993.

†The authors are with the Faculty of Engineering, Tokyo Institute of Technology, Tokyo, 152 Japan.

*We assume the familiarity of Tomita parser.

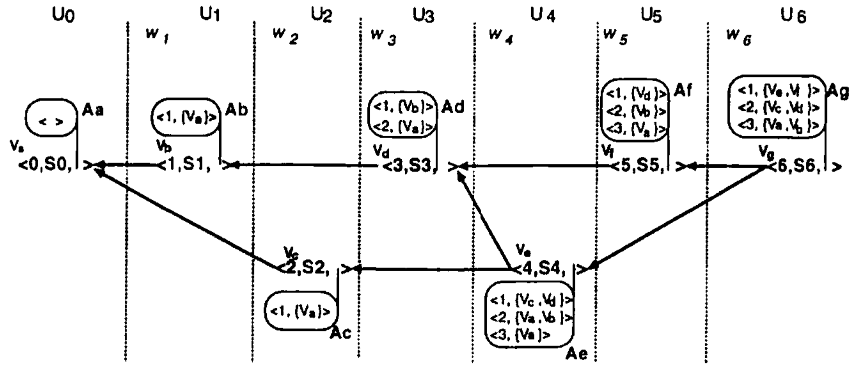


Fig. 1 An example of GSS showing ancestors table.

to them. As the number of parents of each vertex is in the order of i, the number of paths between the leaf and the ancestors at a distance of q becomes at most i^q. In general, i^p, where p is the number of nonterminal and preterminal symbols in the right hand side (rhs) of the longest production.

Due to the time consumed in retrieving the ancestor vertices, the recognition time of Tomita's algorithm becomes $O(n^{1+p})$ for general CFGs, and thus for Chomsky normal form, the recognition time of Tomita's algorithm becomes $O(n^3)$. If the access to the same ancestors and/or same edges more than once is avoided, the time to retrieve the ancestors can be reduced. For this purpose, Kipps changed the data structure of the vertex to $< i, s, A >$ (see Fig. 1). Here i represents the stage number, s the state and A is the ancestors table which consists of a set of tuples such that $\{< k, L_k > | k = 1, 2, \cdots, p\}$ where L_k is a set of ancestors at a distance of k from the vertex $< i, s, A >$. The ancestors table is formed by at most p tuples and the number of ancestors in L_k is in $O(i)$. Figure 1 shows the contents of each vertex along with the contents of ancestors table, here $p = 3$.

When a new leaf is created during shift and reduce actions, each ancestors table can be formed in a constructive way by using the ancestors tables formed in the past. Concretely, on using the ancestors table A$'$ of the parent vertex of a leaf, the tuple $< k, L'_k >$ in A$'$ can be used to form the tuple $< k+1, L_{k+1} >$ of the ancestors table A of the leaf. The time taken to fill all the ancestors tables in stage U_i is in $O(i^2)$. Once an entry in an ancestors table is filled, the time to retrieve that entry is constant thereafter. In other words, only looking for an entry $< q, L_q >$ in the ancestors table of a leaf, it is possible to get a set of ancestors $(=L_q)$ at a distance q from the leaf. From the above arguments, it is clear that the time complexity of Kipps recognizer will become $O(n^3)$ (i.e, $\sum_{i=1}^{n} i^2$). The algorithm to fill an ancestors table can be found in Ref. [4].

3. A Family of Generalized LR Parsing Algorithms using Ancestors Table

At first, we will introduce the Drit parser and then a slightly different parser, called the Ancestors table based GLR (AGLR) parser. The most important feature of the Drit and AGLR parsers is that the partial parse results can be obtained from ancestors tables in the GSS's top vertices alone. Thus during reduce actions, as with the Kipps algorithm, the traversal of edges in GSS is completely avoided. Due to this feature, the time complexity for parsing is limit to $O(n^3)$ for any CFG.

3.1 Drit Parser

From the ancestors table in the leaves alone, it is possible to create dot reverse items (drits) [9] during shift and reduce actions. Drits are dual to Earley's items created in the Earley parser [2]. By modifying Kipps recognizer, we propose a parser called a Drit parser.

In a drit $[\mathrm{A} \rightarrow \alpha \cdot \beta, j]$ in R_i, j represents the stage number just after β and i represents the stage number of the position where the dot appears (in this case the stage number just before β). Thus β represents the portion of the input sentence from w_{i+1} to w_j which has been processed. In case of Earley's items, α is the portion processed. The drit $[\mathrm{A} \rightarrow \cdot\gamma, j]$ in the drit set R_i represents the part of the input sentence from w_{i+1} to w_j which is analyzed as γ and then recognized as A.

Some readers may wonder why we create drits instead of Earley's items. Clearly, it is not possible to create Earley's items directly with these parsing algorithms which do rightmost derivations. Earley's data structure were based on particular parsing style, so we have to make suitable modifications. For clarity, we will give an example of creating drits using the ancestors table of v_g (refer Figs. 1 and 2). Through the example we show that the process is not guaranteed to create necessary and sufficient Earley's items.

Suppose the reduce action X $\rightarrow$ Y Z is applied to the top-of-stack (leaf) v_g, namely <6,s6,Ag>, where Ag $= \{<1, \{\mathrm{v}_e, \mathrm{v}_f\}>, <2, \{\mathrm{v}_c, \mathrm{v}_d\}>, <3, \{\mathrm{v}_a, \mathrm{v}_b\}>\}$, and

v_f = <5,s5,Af>, v_e = <4,s4,Ae>, v_d = <3,s3,Ad>, v_c = <2,s2,Ac>, $\cdots$.

From the ancestors table Ag in v_g alone, we know the following facts (refer Fig. 2).

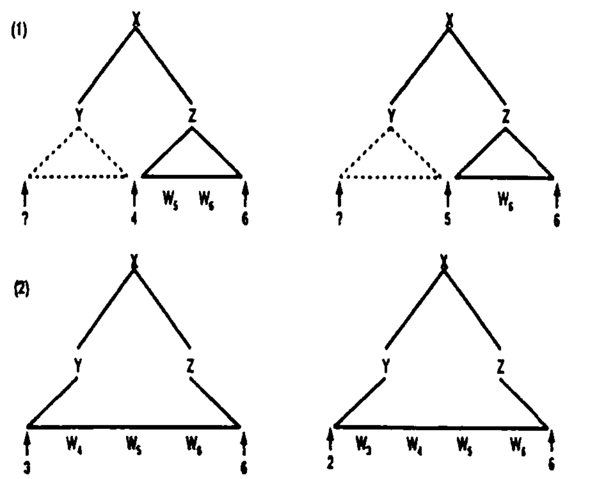


Fig. 2 Input sentence covered by grammar rule X → Y Z for v_g in Fig. 1.

(1) Vertex v_g corresponds to Z which covers w_5w_6 and w_6, since the vertices at a distance 1 from v_g are v_e and v_f whose stage numbers are 4 and 5 respectively.

(2) As the vertices at a distance of 2 from v_g are v_d and v_c, Y Z covers $w_4w_5w_6$ and $w_3w_4w_5w_6$ respectively, since the stage numbers of v_d and v_c are 3 and 2 respectively.

In case of creating Earley's items we have to know the exact portion of the input sentence covered by Y, but from the ancestors table Ag alone, (1) and (2) suggest that we are unable to know it. Using Drit parser's data structures, to get exact portion of the input sentence covered by Y, we have to traverse through the GSS, but we do not want to do so, because it leads to the same inefficiency problem as Tomita's algorithm. This is the reason why, from Ag alone, the creation of necessary and sufficient Earley's items is not guaranteed.

In contrast, we can create the following drits using Ag alone, because in drits it is not necessary to know the exact portion of the input sentence covered by Y.
Drits from (1):
$R_5 \ni [\, X \rightarrow Y \cdot Z, 6]$, $R_4 \ni [\, X \rightarrow Y \cdot Z, 6]$,
Drits from (2):
$R_3 \ni [\, X \rightarrow \cdot\, Y\, Z, 6]$, $R_2 \ni [\, X \rightarrow \cdot\, Y\, Z, 6]$

The reason why we can create necessary and sufficient drits is that GLR parsing is based on right-most derivations which drits reflects. Another bonus in using drits is the localization of duplication checks for newly created drits. The stage number inside the drits will remain the same throughout the processing of a stage. This enables us to limit the scope of duplication check of drits to within that stage.

Now we will give an algorithm for creating drits during the reduce action. Let us consider the production rule used during the reduce action in stage U_i as

$$D_p \rightarrow C_{p1}\ C_{p2} \cdots C_{pq-k}\ C_{pq-k+1} \cdots C_{pq}$$

In this case, we can create drits from the algorithm given below.

for k from q to 1
 for $\forall j'$ s.t. $< j', s', A' > \in$ ANCESTORS(v, k)
  let $R_{j'} := R_{j'} \cup \{[\, D_p \rightarrow C_{p1}\ C_{p2} \cdots \cdots C_{pq-k} \cdot C_{pq-k+1} \cdots C_{pq}, i]\}$

This algorithm of creating drits should be added at the beginning of the reduce procedure given by Kipps in Ref. [4].

Let us consider the case where the parser is going to enter the stage U_{i+1} from the stage U_i by shifting a lookahead word w_{i+1}. If we assume C be the preterminal of the word w_{i+1}, then during the shift action a drit $[C \rightarrow \cdot\, w_{i+1}, i+1]$ is created in R_i.

$$R_i := R_i \cup \{[C \rightarrow \cdot\, w_{i+1}, i+1]\}$$

The reason for including the newly created drit in the set R_i is that, at the time just before shifting word w_{i+1}, the active leaves have the stage number i. After shifting the word w_{i+1} for all the leaves, the top stage number will be incremented by one and when no actions remain in the stage U_i, the processing will enter the new stage U_{i+1}. This step of creating drits during shift action should be added at the beginning of the shift procedure given by Kipps in Ref. [4].

Let us consider the computational complexity of a Drit parser. As the drit parser is based on Kipps recognizer, and as the creation of drits does not affect the filling of ancestors table, the time consumed in filling up ancestors table will remain as the same as mentioned in Sect.2. That is, in stage i, it takes $O(i^2)$ time. Then in Drit parser, the factor which is to be worried is the time consumed in creating drits.

According to Lemma 1 in Sect. 3.1.2, the number of drits created in a stage U_i is in $O(|G|i)$. The time to create the drits in this stage will also become $O(|G|i)$. Thus creation of drits will consume $O(|G|n^2)$ time for a sentence of length n. This shows that creation of drits does not affect the order of parsing time complexity. In this way, drit parser can parse a sentence of length n in $O(n^3)$ time.

To find the space complexity of Drit parser, we have to consider the memory space consumed by GSS and the total number of drits created. It is obvious that the space consumed by GSS is in $O(n^2)$. From Lemma 1, we know that the total number of drits created in a stage U_i is in $O(|G|i)$. For an input of length n, this becomes $O(|G|n^2)$. Thus the space used by GSS in the Drit parser is $O(n^2)$ and, including the space consumed by total number of drits, becomes $O(|G|n^2)$.

In summary, the drit parser creates a set of drits using only the ancestors table of each leaf during the shift and reduce actions. By considering the duality of a drit and an Earley's item, from a set of drits we can generate all the possible parse trees using an algorithm similar to that of Earley's tree generation algorithm, which consumes $O(n^2)$ time to generate a parse tree [1].

3.1.1 An Example of Drit Parsing

In this section we give an example of the Drit parser using the grammar and the LR table in Figs. 3 and 4 [12]. The input sentence used is "I saw a man with a telescope". In this example we give only necessary steps and skip the rest, and note that the ancestors table has two entries because, in Fig. 3 $\rho = 2$.

At the beginning, the GSS has only one vertex labeled v_a in the stage U_0 as shown in Fig. 5(a). By looking at the action table, the next action "shift 4 [sh, 4]" is determined from the LR table given in Fig. 4, and a drit corresponding to the shift action is created.

On shifting the word "I", the parser enters into the stage U_1 and pushes a vertex v_b with stage number 1, the state 4 and an ancestors table Ab. From the state 4 of v_b and the preterminal v of 'saw', the next action "reduce 3 [re, 3]" is determined. Before reducing, drits corresponding to the reduce actions are created from the ancestors table Ab of the top vertex v_b. This is shown in Fig. 5(b).

The action[re, 3] is performed using the rule number 3, NP $\rightarrow$ n, whose rhs has only one symbol and so, the ancestors table Ab is looked for the parent vertex at distance 1 to get v_a. Thus during the reduce action traverse through GSS is avoided. The parser looks for the Goto part of the LR table and a new vertex labeled v_c with state 2 is pushed into the stage U_1 of GSS. For the top vertex v_c, "shift 7" has been determined as the next action. This is shown in Fig. 5(c).

Continuing in this fashion, after some 3 steps, the GSS becomes as shown in Fig. 5(d).

At this point, a conflict with "reduce 7" and "shift 6" occurs and both should be executed. After executing "reduce 7", the new vertex v_h is created and the GSS is as shown below. The top vertex v_g is still active since the action "shift 6" is not yet executed. Thus at this point, we have two active vertices v_g and v_h, as shown in Fig. 5(e).

The top vertex after executing "reduce 1" will also have a "shift 6" action. Now each of the top vertices have a "shift 6" action with the same preterminal 'p' of the word "in". So, a merged vertex v_j with state 6 is pushed into the GSS, where the first entry of the

(1)	S	$\rightarrow$	NP,VP
(2)	S	$\rightarrow$	S,PP
(3)	NP	$\rightarrow$	n
(4)	NP	$\rightarrow$	det,n
(5)	NP	$\rightarrow$	NP,PP
(6)	PP	$\rightarrow$	p,NP
(7)	VP	$\rightarrow$	v,NP

Fig. 3 An English grammar.

State	Action field					Goto field			
	det	n	v	p	$	NP	PP	VP	S
0	sh3	sh4				2			1
1				sh6	acc		5		
2			sh7	sh6			9	8	
3		sh10							
4			re3	re3	re3				
5				re2	re2				
6	sh3	sh4				11			
7	sh3	sh4				12			
8				re1	re1				
9			re5	re5	re5				
10			re4	re4	re4				
11			re6	re6/sh6	re6		9		
12				re7/sh6	re7		9		

Fig. 4 LR table of the grammar in Fig. 3.

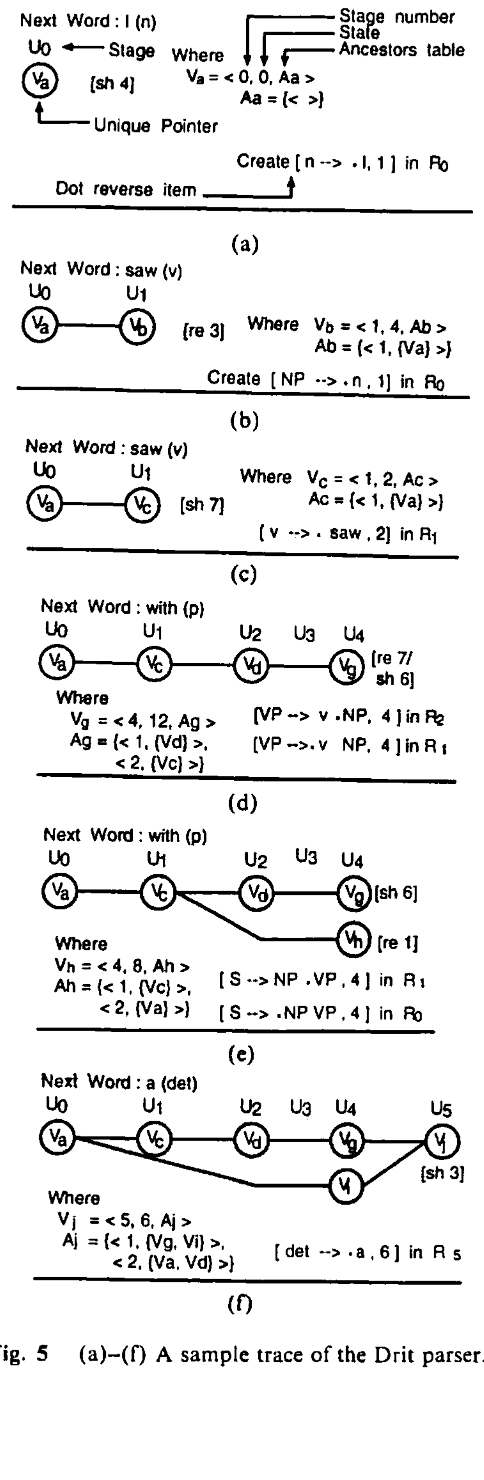


Fig. 5 (a)–(f) A sample trace of the Drit parser.

ancestors table Aj of v_j will have two parents (v_g and v_i, where v_i = <4,1,Ai> and Ai = {<1,{v_a}>}) and, the second entry at a distance 2 is formed by merging the first entry of the ancestors table Ag and Ai as shown in Fig. 5(f).

The rest of the parsing continues in the same fashion. During an "error" process, the corresponding branch of GSS will be terminated as an error and during an "accept" process it will be terminated by accepting the sentence.

3.1.2 A Theoretical Result of the Drit Parser

Lemma 1: The number of drits in a drit set is in $O(|G|i)$ in stage U_i.

Proof: Using a grammar G, during the reduce action, the number of drits created will be equivalent to $|G|$, where $|G|$ is the total number of terminal and preterminal symbols in the rhs of all the rules in G. That is, $|G|$ is computed as: $|G| = \sum_{A \to \alpha \in P} |\alpha|$, where $|\alpha|$ is the length of α and P is the set of production in G.

In stage U_i, the reduce action will be called for at most i times. This is due to the condition on recursive calls, that the reduce action will be called no more than once for each parent of a vertex in U_i, where the number of parents is proportional to i. Hence the number of drits created in stage U_i will become $|G| * i$.

Since there are a bounded number of vertices (say c) in a stage, the above reduce action will occur a bounded number of times in a stage. Thus the number of drits created will become $|G| * i * c = O(|G|i)$. □

3.2 AGLR Parser

In this section we will consider another GLR parser called the Ancestors table based GLR (AGLR) parser. We give a naive version of AGLR parser in the following.

The Drit parser creates a set of drits during shift and reduce actions. Since a set of drits can be created from an ancestors table, during reduce and shift actions we can simply store the ancestors table of the leaf vertex along with the rule used in the reduce action. And we add to each vertices, a link to their parent, and then store the vertices in a two dimensional array called a vertex table [14].

In AGLR, when a new vertex v is first formed, the ancestors table of v will record its own history at 0-th distance, as <0, {v}>. The reason for adding its own history is to know the rightmost position of the rule applied in the reduce action, which can be used during tree generation process.

In case of Fig. 1, for example, the ancestors table of the leaf vertex v_g, <6, s6, Ag> is modified as shown below.

Ag = {<0, {v_g}>, <1, {v_e, v_f}>,
<2, {v_c, v_d}>, <3, {....}>}

If the reduce action on the leaf v_g specifies X → Y Z, then the above ancestors table will be stored along with the rule used by reduce actions on the leaf.

[{X → Y Z}, {<0, {vg}>, <1, {ve, vf}>}]

We call this information an *ancestor item*. In general we represent an ancestor item as [{X → β}, A]. In the ancestor item we store the rule used for the reduce action and the ancestors vertices along with their respective distances. In the above ancestor item v_g is in stage 6 and, since the parent of v_e and v_f are in the stage 2 and 3 respectively, it accomplishes X from 2 to 6 and from 3 to 6. This information is stored in an ancestor item table. For further detail refer [14].

The vertices in the ancestor item points to the vertex table. For each vertex v, the vertex table will enter <(v, i), PL>. Here i is the stage number in which v appear, PL (a set of parent link) is represented as {(Pv, j)} which means that Pv is the parent of v in stage j. In case of Fig. 1, the vertex table becomes:

[<(v_a, 0),{()}>, <(v_b, 1),{(v_a, 0)}>, <(v_c, 2),{(v_a, 0)}>, <(v_d, 3),{(v_b, 1)}>, <(v_e, 4),{(v_c, 2), (v_d, 3)}>, <(v_f, 5),{(v_d, 3)}>, <(v_g, 6),{(v_e, 4), (v_f, 5)}>]

Using the informations in the above ancestor item and the vertex table, we know the following.

(1) <0, {v_g}> in the ancestor item and < (v_g, 6), {(v_e, 4), (v_f, 5)}> in the vertex table indicates that, a sequence of words w_5w_6 (the stage number between 4 of v_e and 6 of v_g) and a word w_6 (the stage number between 5 of v_f and 6 of v_g) are covered by Z (refer Fig. 6(1)).

(2) <1, {v_e, v_f}> in the ancestor item and < (v_e, 4), {(v_c, 2), (v_d, 3)}>, < (v_f, 5), {(v_d, 3)}> in the vertex table indicates that, w_3w_4 (the stage number between 2 of v_c and 4 of v_e), the word w_4 (the stage number between 3 of v_d and 4 of v_e), and w_4w_5 (the stage number between 3 of v_d and 5 of v_f) are covered by Y (refer Fig. 6(2)).

(3) From (1) and (2) : w_3w_4&w_5w_6, w_4&w_5w_6 and w_4w_5&w_6 are covered by Y Z and thus X (refer Fig. 6(3)).

Note† that, (2) in the above instance, teaches just the portions covered by Y.

The time taken to fill an ancestors table is $O(i)$ in stage U_i. Since an ancestors table is filled after every reduce and shift action, it takes $O(i^2)$ time in stage U_i. For a sentence of length n, the time complexity to fill the ancestors table becomes $O(n^3)$, which is the same as Kipps and Drit parsers.

With an efficient representation for the vertices using the vertex table, the GSS space complexity of AGLR

†Careful reader will find out that it is possible to extract a set of Earley's items as well as drits from the modified ancestors table in the top vertex. However it is not necessary to do so because, it is enough to store the ancestors table of the top vertex as it is in order to generate any parse tree.

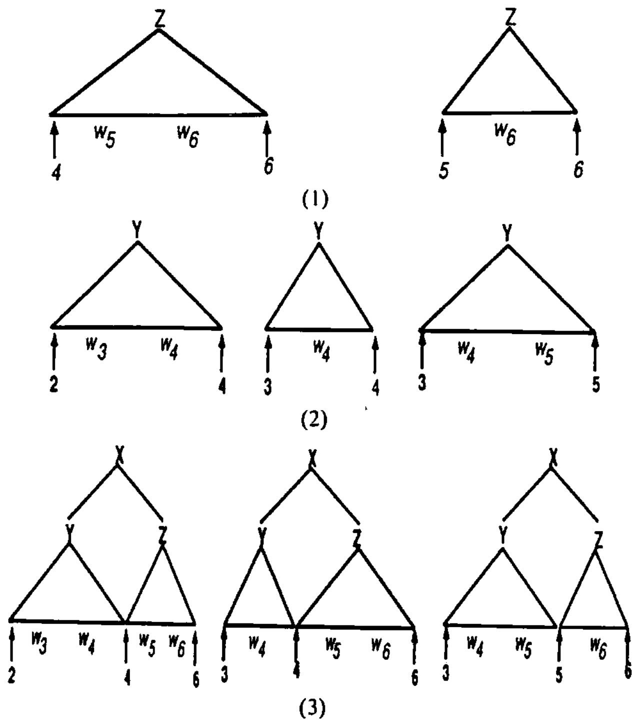


Fig. 6 (1) Input covered by Z in X → Y Z. (2) Input covered by Y in X → Y Z. (3) Input covered by X → Y Z.

is restricted to $O(n^2)$. The array representation of the vertex table enable us to access any vertex in O(1) time. The ancestors table will have only the vertex pointers. Thus the space consumed by the ancestors table becomes $O(i)$ in a stage U_i. Since there will be only $O(n)$ vertices in the GSS, the space consumed by the vertex table becomes $O(n^2)$. Eventually, the total space consumed by vertex table along with the ancestors table becomes $O(n^2)$. The introduction of vertex table will not affect the time complexity and the tree generation process, the details of which is given in [14].

However, the ancestor items stored will consume $O(n^3)$ space. In this way, both the parsing time and the space complexity of AGLR becomes $O(n^3)$.

4. Experimental Results

4.1 The Environment

In this section, we will examine the Drit and AGLR parsers and compare them with Tomita parser. In P.Shann [8], experimental comparisons with Chart parser and the Tomita parser has been shown that the Tomita parser performs faster than Chart parser. Through experiments we will show that our parsers are faster than the Tomita parser, also satisfying our theoretical expectations.

We used the same grammars and sentence sets appeared in Ref. [12]. In this paper we will consider one such grammar which is frequently used in natural language processing. This grammar, say grammar G (which is same as grammar-IV in Ref. [12]), consists of 394 grammar rules. This grammar was originally written by Takakura [10]. The inputs to this grammar are:

1. Normal sentences used in text books (call sentence set I). A sample sentence is shown below. The complete sentence set I will be found in Ref. [12].

   In looking at language as a cognitive process, we deal with issues that have been the focus of linguistic study for many years, and this book includes insights gained from these studies.

2. PP attachment sentences (call sentence set II), which has a pattern

   n v det n (p det n)^m, $m \geq 0$.

The experiments were done on Sony News work station (20 MIPS) using C programming language for implementing Tomita, Drit and AGLR parsers.

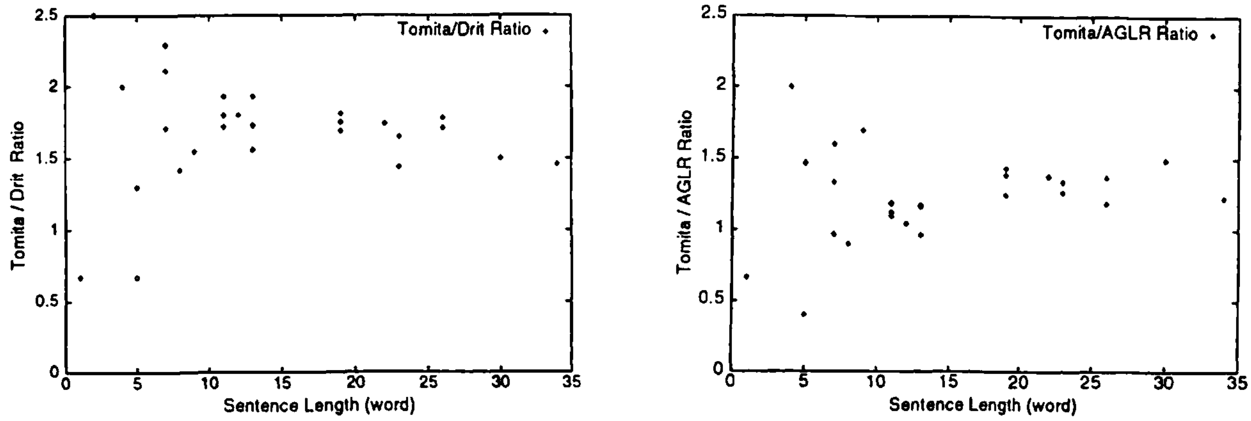
4.2 The Evaluation

The results of parsing sentence set I using grammar G is shown in Figs. 7(a) and 7(b) and that of sentence set II is shown in Figs. 8(a) and 8(b). Figures 7(a) and 8(a) indicates the ratio of Tomita/Drit parsing against length of sentences in sentence set I and sentence set II respectively. These graphs shows that Drit parser is considerably faster than Tomita parser.

The Figs.7(b) and 8(b) indicates the ratio of Tomita/AGLR parsing against length of sentences in sentence set I and sentence set II respectively. These graphs also shows that AGLR parser is faster than Tomita parser in most of the cases.

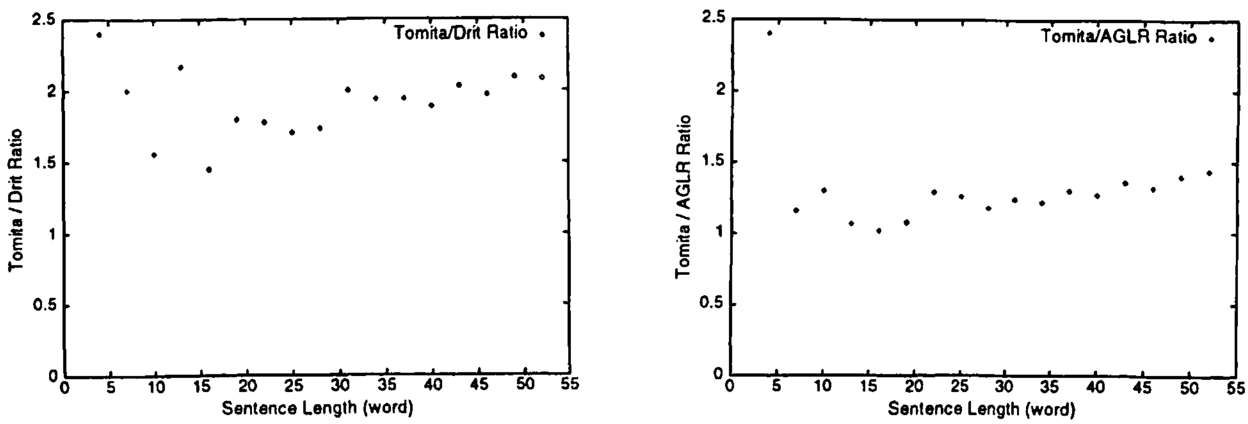
Careful examination of grammar G reveals that it contains more rules in Chomsky normal form. The average length of rhs of the rules used in this grammar is 2.75, (i.e, $\rho = 2.75$). Since Tomita parser's time complexity depends on the value of ρ, using this grammar, the practical performance of AGLR and Drit parsers are better than Tomita parser. If we use grammars in Chomsky normal form (for which $\rho = 2$), both Drit and AGLR parser will give the same performance with Tomita parser.

Next in Figs. 9(a) and 9(b), we will give the comparison of memory space used by the Tomita and Drit parsers. The memory space is used mainly by GSS, packed forest (in case of Tomita), and ancestors table, drits (in case of Drit parser). Note that in Drit parser, we use the ancestors table and store the drits as mentioned in Sect. 3.1. This is the reason why Drit parser consumes less space compared with Tomita parser, even after using the ancestors table. The other details of the practical evaluation can be found in Refs. [11] and [6].



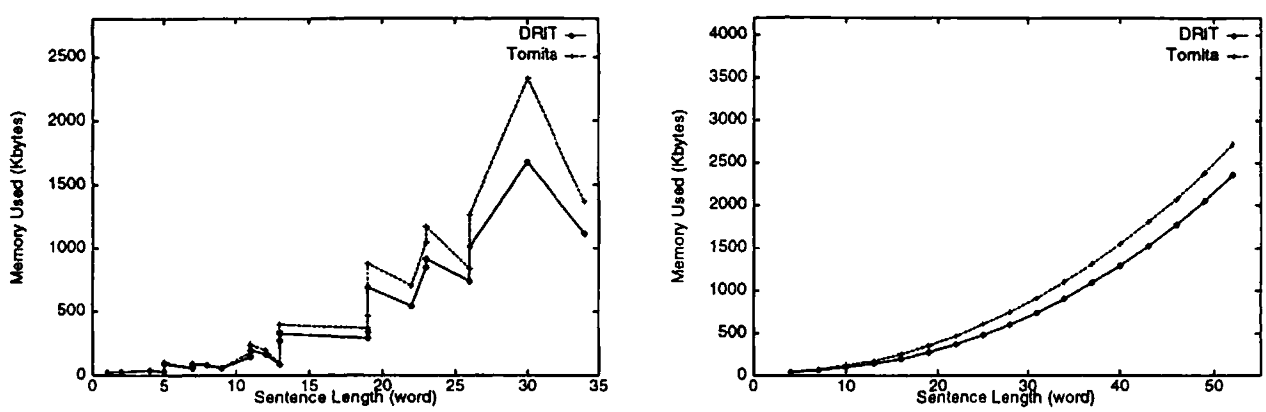
(a) (b)

Fig. 7 (a) Tomita/Drit ratio for sentence set I. (b) Tomita/AGLR ratio for sentence set I.



(a) (b)

Fig. 8 (a) Tomita/Drit ratio for sentence set II. (b) Tomita/AGLR ratio for sentence set II.



(a) (b)

Fig. 9 (a) Memory consumed for sentence set I. (b) Memory consumed for sentence set II.

5. Generating a Tree from a Set of Ancestors Table

For the drits created in Drit parser, the tree generation algorithm of Earley as given in Ref. [1] can be modified to generate all the parse trees. Here we will give the algorithm for constructing a parse tree using the ancestor items and the vertex table. The algorithm for AGLR produces a left parse. The following algorithm constructs a parse tree from a set of ancestor items stored in the ancestor item table (AIT) which is obtained during the parsing process. In an ancestor item $[\{X \rightarrow \beta\}, A]$, A represents a set of ancestors in the form $< D_0, E_0 >, < D_1, E_1 >, \cdots, < D_m, E_m >, \cdots, < D_\rho, E_\rho >$, where $D_0, D_1, \cdots D_\rho$ represents distances and $E_0, E_1, \cdots E_\rho$ represents ancestors at corresponding distances. ρ is the length of the longest production. In the algorithm the vertex table is represented by VT.

ALGORITHM :

Construction of a left parse from a unique set of ancestor items in AIT.

Input : A CFG, $G = (N,T, P, S)$, an input sentence $w = w_1\ w_2 \ldots\ w_n \in T^*$, a set of ancestors item, VT, and AIT.

Output : A left parse for w, or a "error" message.

Method : If $(0, n, Ai) \notin$ AIT (s.t. $Ai = [\{S \rightarrow \alpha\}, At]$) then w is not in L(G), so emit "error" and halt. Otherwise execute the routine $O([0, n, \{S \rightarrow \alpha\}, At])$, the routine O is defined as follows.

Routine $O([i, j, \{A \rightarrow \beta\}, At])$:

(1) If $\beta = X_1\ X_2 \cdots X_{m-1}\ X_m$,
    set $k = 1, l = m\text{-}1, r = i$.

(2) (a) If $X_k \in T$, add 1 to k and r, subtract 1 from l.
    (b) If $X_k \in N$ then for $< D_l, E_l > \in At$, and for $v_h \in E_l$, find $<(v_h, q), PL> \in VT$ s.t. $(Pv_h, r) \in PL$ (where Pv_h is the parent of v_h) then, find an ancestor item $(r, q, \{X_k \rightarrow \gamma\}, At') \in$ AIT.
    Then execute $O([r, q, \{X_k \rightarrow \gamma\}, At'])$.
    Add 1 to k, subtract 1 from l, set $r = q$.

(3) Repeat step (2) until $k = m+1$. Halt.

Note that in an ancestors table, at each distance $D_0, D_1, \cdots, D_\rho$, there may be more than one ancestor. If we want to generate a particular tree, in the worst case, all the possibilities will be considered one by one in step 2(b) to determine a correct path.

6. Conclusion

For certain CFGs it was found that the time complexity of Tomita's GLR parser is more than that of Earley's parser [3], [4], [12], [13]. Kipps gave a recognizer in which he made small modifications to Tomita's algorithm. The time complexity of the modified recognizer is the same as that of Earley's $O(n^3)$ for any CFG [4]. However, Kipps algorithm only recognizes the input sentence as grammatically acceptable or not and it does not produce any parsing results such as partial parse trees or items. For this reason, Kipps algorithm can not be taken as a practical parser.

In this paper, using ancestors table introduced by Kipps, we proposed a family of parsing algorithms, Drit and AGLR. Using their ancestors tables we show a method to extract parse trees. Experiments supported theoretical results showing that these algorithms perform faster than Tomita's algorithm. Since they are based on Kipps algorithm, their parsing time complexity is in $O(n^3)$. Our theoretical results on complexity are summarized in Table 1.

Table 1 Complexity table.

Complexity Factors	Tomita	Kipps	Drit	AGLR
Parsing Time	$n^{1+\rho}$	n^3	n^3	n^3
GSS space	n^2	n^2	n^2	n^2
Tree Extraction	n	-	n^2	n^2

References

[1] Aho, A.V. and Ullman, J.D., *The Theory of Parsing and Compiling*, vol. 1. Prentice-Hall, New Jersey, 1972.

[2] Earley, J., "An efficient augmented-context-free parsing algorithm," *Comm. of ACM*, vol.13, no.1-2, pp.95–102, 1970.

[3] Johnson, M., *The Computational Complexity of GLR Parsing*, pp.35–42, In Generalized LR Parsing, Edited by M. Tomita, Kluwer Academic Publishers, 1991.

[4] Kipps, J.R., *GLR Parsing in Time $O(n^3)$*, pp.43–59. Generalized LR Parsing, Kluwer Academic Publishers, 1991.

[5] Knuth, D.E., "On the translation of languages left to right ", *Information and Control*, vol.8, no.6, pp.607–639, 1965.

[6] Yamada, K., Suresh, K.G., Chosa, M., Numazaki, H. and Tanaka, H., "Implementation and evaluation of generalized LR parsing algorithms using ancestors table," In *SIG. NLP 92-4, Information Processing Society of Japan*, pp.25–32, 1992.

[7] Schabes, Y., "Polynomial time and space shift-reduce parsing of arbitrary context-free grammars," In *Proc. of 29th ACL*, pp.106–115, 1991.

[8] Shann, P., *Experiments with GLR and Chart Parsing*, pp.17–34, Generalized LR Parsing, Kluwer Academic Publishers, 1991.

[9] Suresh, K.G. and Tanaka, H., "Implementation and evaluation of yet another generalized LR parsing algorithm," *Journal of Information Science and Technology*, vol.1, no.4, pp.262–279, Jul. 1992. Also as technical report 92TR-0012, Dept. of Computer Sc., Tokyo Institute of Technology, Aug. 1992.

[10] Takakura, S. and Tanaka, H., *Bottom-Up Parsing For English Based on Prolog*, Dept. of Computer Science, Tokyo Institute of Technology, 1984.

[11] Tanaka, H. and Suresh, K.G., "A family of generalized LR parsing algorithms using ancestors table". Technical Report 92TR-0019, Dept. of Computer Science, Tokyo Institute of Technology, 1992.

[12] Tomita, M., *Efficient Parsing for Natural Language*, Kluwer, Boston, Mass, 1986.

[13] Tomita, M., *Generalized LR Parsing*, Kluwer Academic Publishers, 1991.

[14] Suresh, K.G., "Improved GLR Parsing Algorithms for Natural Language Processing," Doctor Thesis, Comput. Sci. Dept., Tokyo Institute of Technology, Tokyo, Japan 1994.

Hozumi Tanaka received the B.S. and M.S. degrees in faculty of science and engineering from Tokyo Institute of Technology in 1964 and 1966 respectively. In 1966 he joined in the Electro Technical Laboratories, Tsukuba. He received his Doctor of Engineering in 1980. In 1983 he joined as an associate professor in the faculty of Information Engineering in Tokyo Institute of Technology and he became professor in 1986. He has been engaged in Artificial Intelligence and Natural Language Processing research. He is member of the Information Processing Society of Japan, etc.

K.G. Suresh was born in 1967 in Madras, India. He received his Bachelor degree in Physics from the Madras Univ. in 1987, and Master degree in Computer Science from the Bharadhidasan Univ. in 1989. He is currently in Tokyo Institute of Technology, pursuing his studies toward the Doctor of Engineering programme under prof. H. Tanaka. His field of interest is in Natural language processing, Machine translation, complexity theory and software engineering. He received best paper award in 1991 from the Institute of Chartered Computer Professional of India.

Koichi Yamada received his B.S. and M.S. degrees in faculty of engineering from Tokyo Institute of Technology in 1991 and 1993 respectively. At present he is working for Mitubishi Electrical Company, Japan.