Deutsch-Japanischer Workshop '92, Saarbrücken, 0.1.1
(German-Japanese Workshop '92)

A Family of Generalized LR Parsing Algorithms Using Ancestors Table

Hozumi TANAKA K. G. SURESH Kouiti YAMADA
Department of Computer Science, Tokyo Institute of Technology
2-12-1 Ôokayama Meguro-ku Tokyo 152, Japan
Email : {tanaka, suresh}@cs.titech.ac.jp

Abstract

A family of new generalized LR parsing algorithms are proposed which make use of a set of ancestors tables introduced by Kipps [5]. As Kipps's algorithm does not give us a method to extract any parsing results, his algorithm is not considered as a practical parsing algorithm but as a recognition algorithm [8]. In this paper, we will propose a few of methods to extract all parsing 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 Tomita's parsing algorithm can exceed $O(n^3)$ for some context-free grammars (CFGs), the time complexity of our parsing algorithm is in the order of n^3 for any CFGs, since our algorithm is based on the Kipps's recognition algorithm. In order to extract a parsing tree from a set of ancestors tables, it takes time in the order of n^2. However, by making small modifications in the ancestors table, it is possible to extract a parsing tree in the order of n. A preliminary experiment suggests our parsing algorithm seems to be very promising.

1 Introduction

The LR(k) parsing algorithm [6] can parse deterministically and efficiently any input sentences generated by a LR(k) grammar which is a subset of context-free grammar (CFG). Tomita extended the LR(k) parsing algorithm to handle a general CFG not limited to Chomsky normal form [12]. The extended algorithm is called Tomita's parsing algorithm which is known as one of the most efficient generalized LR (GLR) parsing algorithms. Empirically, Tomita's algorithm is faster than Earley's algorithm, but there are some CFGs for which the time complexity of Tomita's algorithm is worse than that of Earley's [3] and for general CFGs, the order of parsing time crosses over n^3 for the input sentence of length n [5]. This is because during the reduce actions on graph-structured stack (GSS), in order to get 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 during the reduce action on GSS, Kipps introduced an ancestors table in which the ancestor vertices are stored [5]. 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 GSS, and thus can avoid duplicated traversals of a same edge and the duplicated access of the same ancestors. As a result, Kipps algorithm can give the time complexity in the order of n^3 for any CFGs.

However, as Kipps's algorithm does not give us a way to extract any parsing results, it is not considered as a parsing algorithm but as a *recognition algorithm* [8]. In this paper, we propose a family of GLR algorithms which can get all parse trees from ancestors table without traversing any edge in GSS, and whose time complexity gives the same order of n^3 as that of Kipps recognition algorithm. In order to extract a parsing tree from a set of ancestors tables in the top vertices of GSS, which has been stored during shift and reduce actions, it takes time in the order of n^2, but by making small modifications in the ancestors table, it is possible to extract a parsing tree in the order of n. For the family of GLR parsing algorithms, when the parsing result is highly ambiguous, the experiments confirms the possibility of tremendous speed up in the parsing time.

Following Kipps [5], we briefly explain Tomita's and Kipps's recognition algorithms in section 2. Section 3 explains a family of the new GLR parsing algorithms using ancestors table. Section 4 gives an experimental evaluation of the family of GLR parsing algorithms which reveals the facts that our GLR parsing algorithm is more efficient than Tomita's parsing algorithm. Finally in section 5 we give our conclusion and future works.

2 An Overlook of Tomita and Kipps Recognition Algorithms

In this section after a brief explanation of Tomita's algorithm as a recognition algorithm, we will explain Kipps recognition algorithm. The position number used in these algorithms is defined as follows.

Let T^+ is a set of a sequence of terminal symbols (words) more than 1 and let $W = w_1\ w_2\ ....\ w_n \in T^+$ be an input sentence. The number i between the word w_i and w_{i+1} is called the position number of the word w_i. The special position number *0* and *n* represents the left and right hand side position of w_1 and w_n respectively.

2.1 Tomita's Method As a Recognizer

Following Kipps, we will explain Tomita's algorithm as a recognition algorithm [5]. Each vertex shifted to a stack will have a state and an edge towards its parents. In Tomita's method, several stacks are combined and packed into a graph structured stack

(GSS). GSS is constructed from a set of vertices and edges. Since GSS forms edges from a vertex towards its parents, it becomes a DAG. There may be several leaves in a GSS. Leaves represents top-of-stack (top vertices of GSS) and the state of each leaves represents currently active states. The leaves of GSS grows in stages. Each stage U_i corresponds during the processing of the i-th word w_i of the input sentence with the next look-ahead word w_{i+1}.

Figure 1 shows a schematic example of GSS. Here v_i represents a vertex and w_i represents i-th input word. Thus the vertex v_6 in stage U_6 covers w_6 and w_5 w_6. v_5 in stage U_5 covers w_4w_5. In the same way, v_4 in stage U_4 covers w_4 and w_3w_4. In real situation, each vertex of the GSS in U_i is a triple $< i, s, L >$, where s and L represents the parse state and a set of parent vertices respectively.

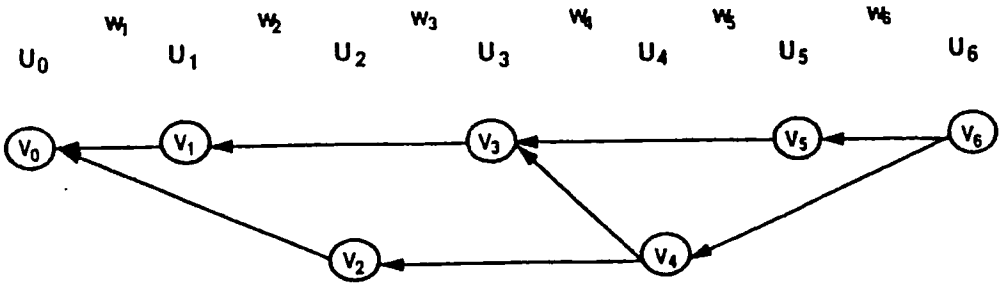


Fig 1: An Example of GSS

The Tomita's recognition algorithm works as follows. A GSS is initialized by pushing $< 0, s0, \{\} >$ in stage U_0, which becomes the root of the GSS with w_1 as a look-ahead word (scanning word), whose preterminal will be used to determine the parsing actions in the LR table. The input sentence is parsed stage by stage for each word from left to right thus changing the GSS.

Upon scanning w_{i+1} as a look-ahead word, the recognition algorithm carries out the following four actions in stage U_i. What kinds of actions (shift; reduce; accept; error) are to be carried out is determined by the leaves in U_i, LR table, and the preterminal of the scanning word w_{i+1}.

1. Reduce : The recognizer pops the number of vertices (corresponding to the right hand side (rhs) of the production rule specified by the reduce action) from the top of the stack and then creates a new leaf in U_i which becomes active and the state of which will be determined by the Goto field.

2. Shift : A new leaf corresponding to a look-ahead word w_{i+1} is pushed in U_{i+1}. The state of the leaf is determined by the shift action. Note that the newly created leaf in U_{i+1} is not active until there is no active leaf remained unprocessing in U_i and the look-ahead word becomes w_{i+2}.

3. Error : The leaf with error action will be truncated.

4. Accept : Recognition process will end with success.

Only after all the leaves in the stage U_i has been processed, the recognizer proceeds to the stage U_{i+1} scanning the next word w_{i+2}.

In case of 1 and 2, a new leaf is added in U_i and U_{i+1} respectively and edges are formed from the new leaf to its parents. If there exists a leaf with the same state as that of a newly created leaf in U_i in case of 1, or U_{i+1} in case of 2, then they will be merged into one. The leaf after merge will have several parents. Merging of leaves with same state avoids the duplicated processing of the input sentence and also it makes sure the number of vertices in each stage to be within the number of total states in LR table. Hence the order of the number of vertices in each stage becomes constant.

Let us focus more on a reduce action in the stage U_i. A reduce action pops the number of vertices (say q) equal to the number of nonterminal and preterminal symbols in rhs of the rule used in the reduce action. Then the ancestor vertices at a distance of q will tentatively become the top of the stack, and using the Goto field of LR table, a new leaf is pushed in the stage U_i for each ancestor. At the same time new edges are formed from the leaves to the ancestors and the vertices in the distant stages becomes the parents of the leaves. In consequence, a vertex $< i, s, L >$ in U_i has at most c*i parents where c is the number of total states and is constant. We can conclude that the number of ancestors of each vertex is in the order of i.

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 to them. As the number of parents is in the order of i, the number of edges between the leaf and the ancestors at a distance of q becomes at most i^q. Therefore, the time of Tomita's recognition algorithm is in the order of $n^{1+\rho}$ ($= \sum_{i=1}^{n+1} i^\rho$, where ρ is the number of nonterminal and preterminal symbols in the rhs of the longest production). If $\rho > 2$, the order crosses over n^3. For the grammars in Chomsky normal form $\rho = 2$ and hence the order of recognition time becomes n^3.

2.2 Kipps Recognizer

Due to the time consumed in getting the ancestor vertices, the time complexity of Tomita's recognition algorithm crosses over the order of n^3 for general CFGs. As explained in 2.1, although the number of ancestors at a distance of q from a leaf in the stage U_i is in the order of i, the time needed to extract them is in the order of i^q. The reason is that, to get the ancestor vertices, an edge once traversed might be again traversed repeatedly and an ancestor vertex once accessed might be again accessed repeatedly.

For example, in the GSS shown in figure 1, in order to pick up an ancestor vertex, say v_3, at a distance 2 from the top-of-stack v_6, we have to traverse two paths from v_6 to v_3, namely v_6–v_5–v_3 and v_6–v_4–v_3, resulting in accessing the same one ancestor v_3 two times. This means that in case of Tomita's algorithm, the same ancestors and/or the same edges might be accessed many times.

If the access to the same ancestors more than once is avoided, the time to get the ancestors can be reduced. For this purpose, Kipps changed the data structure of the

vertex as $< i, s, A >$ (see fig.2). Here i represents the stage number, s the state and A is the ancestors table which consists of a set of tuples such that $\{< k, L_k > | k = 1, 2, \cdots, \rho\}$ where L_k is a set of ancestors at a distance of k from the vertex $< i, s, A >$. From the above discussion, we know that the ancestors table is formed by at most ρ (see 2.1) tuples and the number of ancestors in L_k is in the order of i.

In figure 2, we elaborate the figure 1 when $\rho = 3$, by showing the contents of each vertex along with the contents of ancestors table.

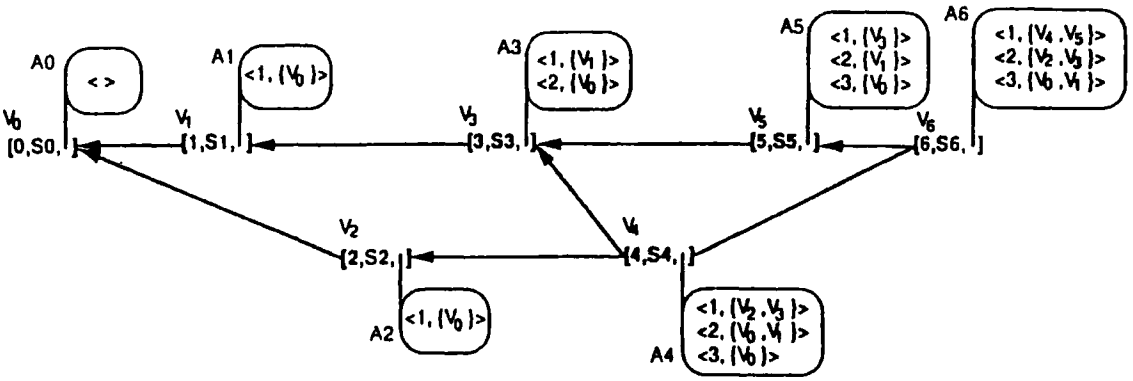


Fig 2 : An Example of GSS showing Ancestors Table

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 (see fig. 2). According to Kipps, time taken to fill all entries in an ancestors table is in the order of i^2. Once an entry in an ancestors table is filled, the time to reget that entry is constant thereafter. Therefore it is possible to get ancestor vertices from the entry in constant time. From the above arguments, for a sentence of length n, it is proved that the time complexity of Kipps recognition algorithm will become in the order of n^3 $(= \sum_{i=1}^{n} i^2)$.

3 A Family of Generalized LR Parsing Algorithms Using Ancestors Tables

At first, we will introduce Drit parsing algorithm which is one of a family of GLR parsing algorithms using ancestors table and then we will introduce two other parsing algorithms, called Ancestors table based GLR (AGLR) parsing algorithm, which are faster than Drit parsing algorithm. The most important feature of Drit and AGLR parsing algorithms is that, the partial parsing results can be obtained from ancestors tables in the top vertices alone. Thus during reduce actions, as well as Kipps recognition algorithm, the traversal of edges in GSS is completely avoided. Due to this feature,

the time complexity for parsing limits to n^3 for any CFG.

3.1 Drit Parser

During shift and reduce actions, from the ancestors table in the leaves alone, it is possible to create dot reverse items (drit) [10] which is dual to that of the Earley's items. By modifying Kipps recognition algorithm we propose a parsing algorithm called Drit parsing algorithm which creates drits during shift and reduce actions.

The meaning of a drit $[A \rightarrow \alpha \cdot \beta, j]$ in a drit set R_i is as follows. The position number just after the β is j and that of dot is i. Thus β represents the portion of the input sentence from w_{i+1} to w_j which have been processed. In case of Earley's items, α is the portion processed. The drit $[A \rightarrow \cdot\gamma, j]$ 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.

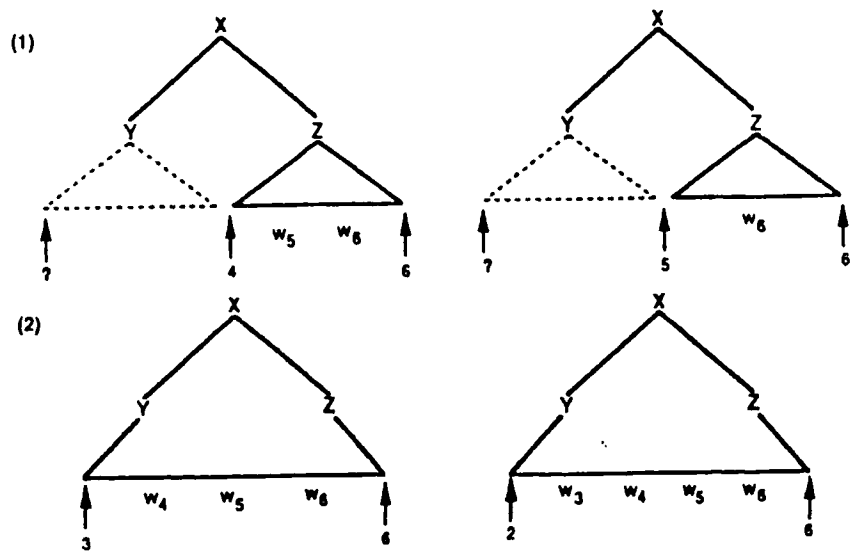


Fig 3 : Example of Input sentence covered by grammar rule X⟶Y Z for V_6in fig. 2

The reasons for creating drits instead of Earley's items are explained by using figure 2 and figure 3. Suppose the reduce action X → Y Z is applied to the top-of-stack (leaf) v_6, namely <6,s_6,A6>.

Where A6 = {<1, {v_4,v_5}>, <2, {v_2,v_3}>, <3, {v_0,v_1}>},

v_5 = <5,s5,A5>, v_4 = <4,s4,A4>, v_3 = <3,s3,A3>, v_2 = <2,s2,A2>, ⋯.

From the ancestors table A6 in v_6 alone, we can know the following facts.

(1) Vertex v_6 corresponds to Z which covers w_5w_6 and w_6, since the vertices at a distance 1 from v_6 are v_5 and v_4 whose position numbers are 5 and 4 respectively.

(2) As the vertices at a distance of 2 from v_6 are v_3 and v_2, Y Z covers $w_4w_5w_6$ and $w_3w_4w_5w_6$ respectively, since the position number of v_3 and v_2 is 3 and 2 respectively.

Note that from the ancestors table A6 of the top-of-stack alone, it is impossible to

know whether there is an edge connecting v_2 with v_5 and v_4, or an edge connecting v_3 with v_5 and v_4. In other words, the only thing we know from A6 is that there exists some vertices at a distance 1, 2, etc from the top-of-stack v_6. Even though we are able to know the existence of proper edges between the vertices if we traverse through the GSS, we do not want to do so, because it leads to the same inefficiency problem as Tomita's algorithm. In case of Earley's items we have to know the exact portion of input sentence covered by Y, but from the ancestors table A6 alone, we are unable to know it. From the above considerations, we conclude that from A6 alone, it is not guaranteed to create necessary and sufficient Earley's items.

On the contrary, we can create the following drits using A6 alone, because in drits the exact portion of input sentence covered by Y is not necessary to know.

Drits from (1):

$$R_5 \ni [\, X \rightarrow Y \cdot Z,\ 6], \quad R_4 \ni [\, X \rightarrow Y \cdot Z,\ 6],$$

Drits from (2):

$$R_3 \ni [\, X \rightarrow \cdot\ Y\ Z,\ 6], \quad R_2 \ni [\, X \rightarrow \cdot\ Y\ Z,\ 6]$$

The reason why we can create necessary and sufficient drits is that GLR parsing is based on the right-most derivations which drits reflects. Another bonus in using drits is the localization of duplication checks for newly created drit. The position 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 within that stage.

Let us consider that the parser is going to enter in the stage U_{i+1} from the stage U_i by shifting a look-ahead word w_{i+1}. If we assume C be the preterminal of the word w_{i+1}, 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 of word w_{i+1}, the active leaves have the position number i. Only after shifting the word w_{i+1} for all the leaves, the top position number will be incremented by one and the processing enters the new stage U_{i+1}.

In summary, drit parser creates a set of drits during shift and reduce actions. By considering the duality of a drit and an Earley's items, from a set of drits it is possible to generate all the possible parsing trees using an algorithm similar to that of Earley's tree generation algorithm. The time taken to generate one parse tree is in the order of n^2.

Let us consider the computational complexity of a Drit parser. As drit parsing algorithm is based on Kipps recognition algorithm, the time complexity of drit parser is in the order of n^3, since the creation of drits does not affect the time complexity. The only thing which affects the time complexity of Drit parser is the time consumed by filling up ancestors table. As mentioned before in 2.2, in stage i, it takes in the order of i^2. Thus summing up i from 1 to n, the time complexity of the Drit parser becomes n^3.

To find the space complexity of Drit parsing algorithm, we have to consider the

memory space consumed by GSS and by the total number of drits created. It is obvious that the space consumed by GSS is in the order of n^2. The total number of drits created is in the order n^3. Since each vertex of the GSS can have at most i parents, the number of drits created at one reduce action is $\rho * i$ where ρ is the length of the longest production. For a stage U_i, reduce actions will occur for i times and hence the number of drits created at one stage U_i is $\rho * i * i$, which is in the order of i^2. Summing from $i = 0, 1, \cdots, n$ gives the order as n^3. Thus the space used by Drit parsing algorithm becomes in the order of n^3.

3.1.1 An Example of Drit Parsing

In this section we give a trace of Drit parsing algorithm using the grammar and the LR table in figure 4 and 5 [12].

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

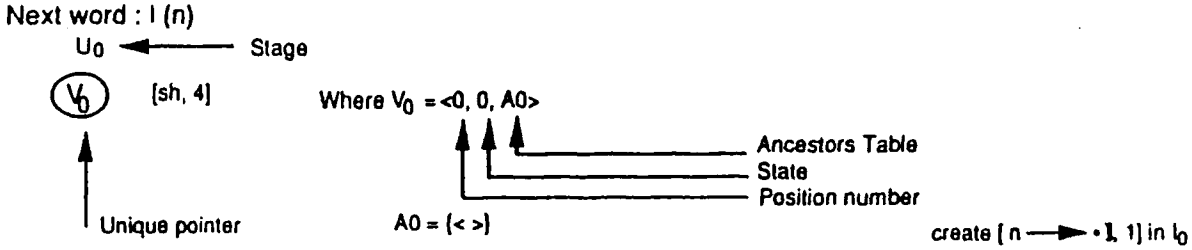
Fig. 4 : Simple English grammar

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. 5 : LR table of the grammar in fig. 4

Each step in the trace is shown below. Note that an ancestors table has two entries because, the rhs of rules in fig.4 has atmost two non-terminal/preterminal symbols.

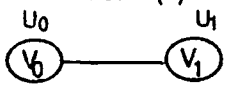
At the beginning, the GSS contains only one vertex labelled v_0 in the stage U_0. By looking at the action table, the next action "shift 4 [sh, 4]" is determined from the LR table given in fig. 5, and a drit corresponding to the shift action is created.



On shifting the word "I", the parser enters into the stage U_1 and pushes a vertex v_1 with position number 1, the state 4 and an ancestors table A1. The next action "reduce 3 [re, 3]" is determined from the action part of the LR table, because the state

of v_1 is 4 and the preterminal of the next word is "v".

Next word : saw (v)

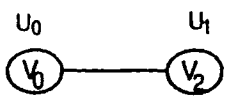


[re, 3]

Where V_1 = <1, 4, A1>
A1 = {<1, {V_0}>}

create [NP ⟶ •n, 1] in t_0

Before reducing, drits corresponding to the reduce actions are created from the ancestors table A1 of the top vertex v_1. As [re, 3] is performed using the rule number 3, NP → n, whose rhs has only one symbol and so, the ancestors table A1 is looked for the parent vertex at distance 1 to get v_0. Thus during the reduce action we are not actually traverse through GSS, instead we look for the ancestors table A1 of the top vertex v_1 alone. The parser looks for the Goto part of the LR table and a new vertex labelled with v_2 with state 2 is pushed into the stage U_1 of GSS. For the top vertex v_2, "shift 7" has been determined as the next action.



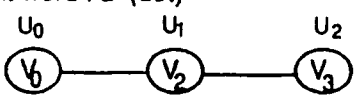
[sh, 7]

Where V_2 = <1, 2, A2>
A2 = {<1, {V_0}>}

create [v ⟶ •saw, 2] in t_1

After executing "shift 7" we have the GSS as shown below with the top vertex v_3.

Next word : a (det)



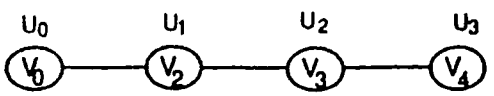
[sh, 3]

Where V_3 = <2, 7, A3>
A3 = {<1, {V_2}>,
<2,{V_0}>}

create [det ⟶ •a, 3] in t_2

After executing "shift 3" we have the GSS as shown below.

Next word : man (n)



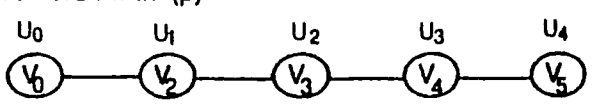
[sh, 10]

Where V_4 = <3, 3, A4>
A4 = {<1, {V_3}>,
<2, {V_2}>}

create [n ⟶ •man, 4] in t_3

After executing "shift 10" we have the GSS as shown below. Note that the ancestors table A4 has two entries, which corresponds to the number of non-terminal and preterminal symbols of the rhs of the longest production rule given in fig.4

Next word : with (p)

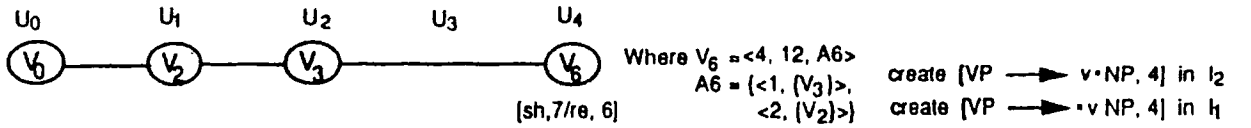


[re, 4]

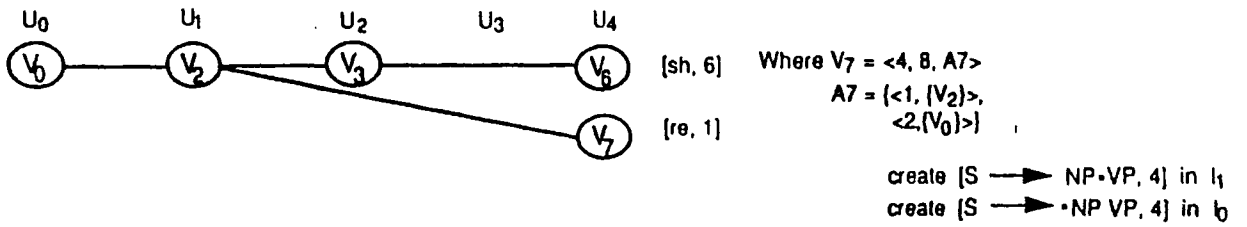
Where V_5 = <4, 10, A5>
A5 = {<1, {V_4}>, create [NP ⟶ det •n, 4] in t_3
<2, {V_3}>} create [NP ⟶ •det n, 4] in t_2

The next action is "reduce 4" and the drits using the rule number 4, NP → det n, are created from the ancestors table A5. The GSS is reduced and the new vertex v_6 is

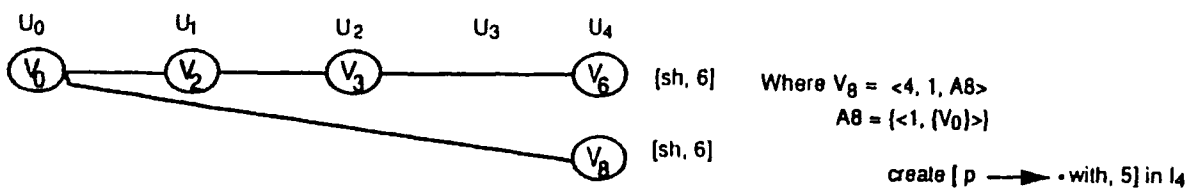
added in the stage U_4.



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_7 is created and the GSS is as shown below. The top vertex v_6 is still active since the action "shift 6" is not yet executed. Thus at this point, we have two active vertices v_6 and v_7.

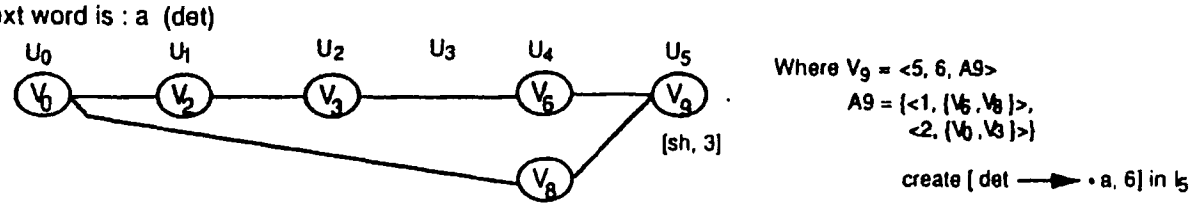


The action "reduce 1" associated with the top vertex v_7 is executed and the resultant GSS is shown below.



Now there are two "shift 6" actions for both top vertex v_6 and v_8 with the same word "in". A merged vertex v_9 with state 6 is pushed into the GSS, where the first entry of the ancestors table A9 of v_9 will have two parents (v_6 and v_8) and, the second entry at a distance 2 is formed by merging the first entry of the ancestors table A6 and A8 as shown below.

Next word is : a (det)



The above trace shows all the actions carried out by the Drit parser, such as shift, reduce, merge, creating drits. During "error" process, the corresponding branch of GSS will be terminated as an error and during "accept" process it will be terminated by accepting the sentence.

3.2 AGLR Parser

In this section let us consider the other GLR parsing algorithm called Ancestors table based GLR (AGLR) parsing algorithm.

3.2.1 AGLR Parser - Version 1 (AGLR 1)

Drit parsing algorithm creates a set of drits during shift and reduce actions from which partially parsed trees can be generated. Since it is possible to create a set of drits from ancestors table, during reduce action we can think about simply storing the ancestors table of the leaf vertex as it is. If we do so, the duplication check of drits can be avoided. For instance, suppose the reduce action X → Y Z is applied to the top vertex v_6 in fig. 2,

from A6 = {<1, {v_4,v_5}>, <2, {v_2,v_3}>, ··· }

we store the following information as

[{X → Y Z}, {<0,{6}>, <1, {4,5}>, <2, {2,3}>}].

Note that the rule(s) used for the reduce action and position numbers of the ancestors vertices along with their respective distances are stored. The information <0,{6}> is stored in order to know the right most position of Z. It is obvious that a set of drits can be obtained from the informations stored above. Since this GLR parsing algorithm is totally based on ancestors table in top vertices, from now on we call this GLR parsing algorithm as Ancestor table based GLR parsing algorithm, AGLR in short. This method also has the time complexity the same as that of Drit parsing algorithm, in the order of n^3. The time to extract a parse tree from the informations stored as above, is also the same as that of Drit parser, namely n^2.

3.2.2 AGLR Parser - Version 2 (AGLR 2)

In order to extract a parse tree after Drit and AGLR1 parsing has completed, it takes time in the order of n^2. With a small modification of the contents of ancestors table, it is possible to reduce this order to n. To do so, at first when a new leaf v is formed, the ancestors table will record <0, {v(L)}>. Here, L is a set of position number of parent vertices of v. For example in case of fig. 2, the ancestors table of the leaf vertex v_6, <6, s, A6> becomes as shown below.

A6 = {<0, {v_6(5,4)}>,
<1, {v_4(2),v_5(3)}>,
<2, {v_2(0),v_3(1)}>,
<3, ...¿}

If the reduce action on the leaf v_6 specifies X → Y Z, then the above ancestors table will be stored with a small modification along with a set of rules used by reduce actions on the leaf.

[{X → Y Z}, {<0,{4(6),5(6)}>, <1, {2(4),3(5)}>, <2, {0(2),1(3)}>}]

In storing the above information, we unpack the set of position numbers in A and reverse the order. This type of data structure will give us a clear and simple image in finding the complexity of tree generation. Here for instance,

(1) <0,{4(6),5(6)}> indicates that, a sequence of words w_5w_6 (the position number between 4 and 6) and a word w_6 (the position number between 5 and 6) are covered by Z.

(2) <1, {2(4),3(5)}> indicates that, w_3w_4 (the position number between 2 and 4) and a sequence of words w_4w_5 (the position number between 3 and 5) are covered by Y.

(3) From (1) and (2) : $w_4w_5w_6$ and $w_3w_4w_5w_6$ is covered by Y Z and thus X.

Note that in the case of ancestor tables in leaves, as described in section 3.1, it is unable to decide the exact portion of input sentence which is covered by Y. This causes the time complexity of tree generation to be in the order of n^2. However, (2) teaches just the portions covered by Y. From the facts, we are able to extract the parse trees in a deterministic way. Accordingly, from the modified ancestors table as mentioned above, to extract one parse tree, the time needed is in the order of n. The parsing time complexity still remains the same as n^3. But the space complexity increases to n^3 because, according to the modification of the contents of ancestors tables it takes i^2 order in the stage U_i.

4 Experimental Results

In this section, we will examine the parsing algorithms discussed so far and compare them. Two types of grammars were used in our experiment. Gram-1 consists of 3 grammar rules [3]. This grammar has high dense ambiguity. Gram-2 is the second type of grammar used, consisting of 123 grammar rules. For this grammar, the input sentence used for testing has PP attachments as, "I open the door with a key with a key ...." . We compare Drit parser and AGLR 1 parser with Tomita's parser.

Machine used : Sony News work station (20MIPS).

Programming Language used : C.

Grammar Used (Gram-1) :

$$S \rightarrow S\ S\ S\ S$$
$$S \rightarrow S\ S$$
$$S \rightarrow a$$

Input sentence : aaaaaaaaaa.....

Input Length	*Parsing Time in msec*		
	Drit	AGLR 1	Tomita
10	49	34	140
11	68	45	239
12	94	60	399
13	133	83	698
14	178	98	1084
15	231	111	1420
..	...	...	...
..	...	...	...
20	...	269	12562
25	...	530	64163
30	...	980	246130

Grammar used (Gram-2) : English grammar with 123 rules.
Input sentence : I open the door with a key with a .....

Input Length	Ambiguity	*Parsing Time in msec*		
		Drit	AGLR 1	Tomita
16	84	48	48	61
19	264	65	69	90
22	858	95	83	116
25	2860	122	105	156
28	9724	167	128	190
31	33592	220	166	226
..	....	...	...	...
..	....	...	...	...
49	....	...	430	681
52	....	...	511	802

5 Conclusion

For certain CFG it was found that, the time complexity of Tomita's GLR parsing algorithm is more than that of Earley's algorithm [11,12]. Kipps gave a recognition algorithm in which he made small modifications in Tomita's algorithm. The time complexity of the modified recognition algorithm is the same as that of Earley's (n^3 where n is the length of the input sentence) for any CFG [5]. However, Kipps algorithm only recognizes the input sentence as grammatically acceptable or not and it does not produce any parsing results such as partially parsed trees or items. For this reason, Kipps algorithm can not be taken as a practical parsing algorithm [8].

In this paper, using ancestors table introduced by Kipps, we propose a method to extract the parsing trees. We explain a family of parsing algorithm based on ancestors

tables, which confirms the fast performance of compared to Tomita's algorithm. Since the family of parsing algorithms, Drit, AGLR1 and AGLR2, are based on Kipps algorithm, their parsing time complexity is in the order of n^3 and space complexity is in the order of n^2 and n^3 as summerizing in fig. 6.

	Tomita	Drit	AGLR-I	AGLR-II
time	n^m	n^3	n^3	n^3
GSS space	n^2	n^2	n^2	n^3
Tree Extraction	n	n^2	n^2	n

where $m \geq 3$

The followings are our future research works. (1) Further detailed evaluation of AGLR parsing algorithm is needed along with experimental comparison to Tomita's method, (2) Developing a parallel algorithm, (3) A parallel algorithm for tree generation, (4) Developing a tool for natural language processing based on AGLR parsing algorithm.

References

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

[2] Earley, J. :
An Efficient Augmented-Context-Free Parsing Algorithm, Comm. of ACM, 13, 1-2, 95-102 (1970).

[3] Johnson, M. :
The Computational Complexity of Tomita's Algorithm, Generalized LR Parsing, Kluwer Academic Publishers, pp.35-42 (1991).

[4] Kay, M. :
Algorithm Schemata and Data Structures in Syntactic Processing, Readings in Natural Language Processing, Morgan Kaufmann Publishers, Inc. pp.35-70

[5] Kipps, J.R. :
Analysis of Tomita's Algorithm for General Context-Free Parsing, Generalized LR Parsing, Kluwer Academic Publishers, pp.43-59 (1991).

[6] Kunth, D.E. :
On the Translation of Languages Left to Right, Information and control, 8(6), pp.607-639 (1965).

[7] Numazaki, H. and Tanaka, H. :
A New Parallel Algorithm for Generalized LR Parsing, COLING'90 , Vol.2, pp.305-310 (1990).

[8] Schabes, Y. :
Polynomial Time and Space Shift-Reduce Parsing of Arbitrary Context-free Grammars, Proc. of 29th ACL, 106-115 (1991).

[9] Suresh, K.G. and Tanaka, H. :
Implementation and Evaluation of Yet Another Generalized LR Parsing Algorithm, Proc.of the Indian Computing Congress, Tata McGraw-Hill, 506-515 (1991).

[10] Tanaka, H and Suresh, K.G. :
YAGLR : Yet Another Generalized LR Parser, Proceedings of ROCLING IV, Republic of China, 21-31 (1991).

[11] Tanaka, H and Numazaki, H. :
Parallel GLR Parsing Based on Logic Programming, in Tomita, M ed. : Generalized LR Parsing, Kluwer Academic Publishers, 77-91 (1991).

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

[13] Tomita, M. Ed. :
Generalized LR parsing, Kluwer Academic Publishers (1991).