GLR PARSING ALGORITHM WITH AN IMPROVEMENT

K.G.SURESH Hozumi TANAKA

Department of Computer Science, Tokyo Institute of Technology
2-12-1 Ôokayama Meguro-ku Tokyo 152, Japan
Phone : 81 3 3726 1111 Extn. 4188
Email : {suresh, tanaka}@cs.titech.ac.jp

Abstract

The generalized LR (GLR) parser devised by Tomita is very efficient and has been used for the natural language processing systems such as machine translation and speech understanding systems. Although the time complexity of Tomita's parser exceeds O(n^3) for the general context-free grammars (CFGs), empirically it is proved to be very efficient in practical natural language processing. Kipps modified Tomita's algorithm and proposed an algorithm with O(n^3) time complexity for general CFGs. But this modified algorithm can not be considered as a practical parser due to its unability to generate parse trees. However, with some modifications to the Kipps algorithm, we propose an algorithm to generate all the parse trees and complete the Kipps algorithm as a practical parser. The parsing time and the space complexity of our improved algorithm remains the same as O(n^3) and O(n^2) respectively for general CFGs. We proved the complexity results practically as well as theoretically. We show a preliminary experiment giving support to our theoretical expectations.

1 Introduction

The generalized LR parser devised by Tomita has been used for the natural language processing systems such as machine translation systems and speech understanding systems [10, 11]. The reasons are as follows: At first, GLR parser such as Tomita's, is very efficient compared to other parsers even though the time complexity of Tomita's parser exceeds O(n^3) for general CFGs, where through out this paper n stands for the length of the input sentence. Secondly, parsing proceeds incrementally from left to right of the input sentence enabling real time processing.

The time complexity of Tomita's GLR (TGLR) algorithm becomes O($n^{1+\rho}$) for general context-free grammars (CFGs), where ρ is the length of the longest production in the grammar. Thus TGLR needs its grammars to be in Chomsky normal form to parse in O(n^3) time. Kipps [5] pointed out that this is due to the duplicated traversal of the same edges and the access of the same ancestors repeatedly during reduce action on the graph-structured stack (GSS). To avoid this problem, Kipps himself gave a modified algorithm which can perform in O(n^3) time complexity for any CFG, by introducing a data structure called ancestors table. But this modified algorithm can not be considered as a practical parser [5, 6]. The reason may be its unabilitiness to generate parse trees.

However, with some modifications to the Kipps method, we propose a method to generate all the parse trees and complete Kipps method as a practical parser. The parsing time of our improved parser remains the same as O(n^3) for any CFG, even after modifying the important data structure called ancestors table and we proved the complexity results both theoretically and practically. The most important feature of our method is that, the partial parse informations can be obtained from the ancestors table of the top vertex alone. (Thus avoiding the traversal of GSS). In this paper, we call our improved GLR parser as AGLR (Ancestors table based GLR) parser. In order to extract a parse tree from the partial parse informations of AGLR, it takes $O(n^2)$ time.

In this paper we assume the familiarity of Tomita's algorithm. In the rest of this paper we give a brief introduction to Kipps algorithm in section 2. In section 3 we outline AGLR parsing process. In this section, we give some theoretical results and tree generation algorithm of AC ., and a note on the property of ancestors table. In section 4, we give practical results with discussions. Finally in section 5, we give our conclusion.

2 The Kipps Recognizer

Figure 2.1 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 help of the next look-ahead word w_{i+1}. For example, the vertex v_g in stage U_6 covers w_6 and w_5 w_6, v_f in stage U_5 covers $w_4 w_5$, and so on.

Tomita's algorithm the same ancestors and/or the same edges might be accessed many times. For example, in the GSS shown in figure 2.1, 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 to them. As the number of parents[1] 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^ρ, where ρ is the number of nonterminal and preterminal symbols in the right hand side (rhs) of the longest production.

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, because this is the factor which makes the recognition time of TGLR to $O(n^{1+\rho})$ for general CFGs. For this purpose, Kipps changed the data structure of the vertex to $< i, s, A >$ (see fig.2.1). Here i represents the stage number, s the state and A is the ancestors table which consists of a set of tuples such that $\{< k, L_k > | k = 1, 2, \cdots, \rho\}$ 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 ρ tuples and the number of ancestors in L_k is in $O(i)$. Figure 2.1 shows the contents of each vertex along with the contents of ancestors table, here $\rho = 3$.

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 entries in an ancestors table is in $O(i^2)$ in stage U_i. 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, $\sum_{i=1}^{n} i^2$).

[1]In the following descriptions, the term 'parent' of v stands for all the vertices immediately left of v. That is, all the vertices at distance 1 from v. The term 'ancestors' of v stands for all the vertices including the parent vertices on the left of v. That is, all the vertices at distance ≥ 1.

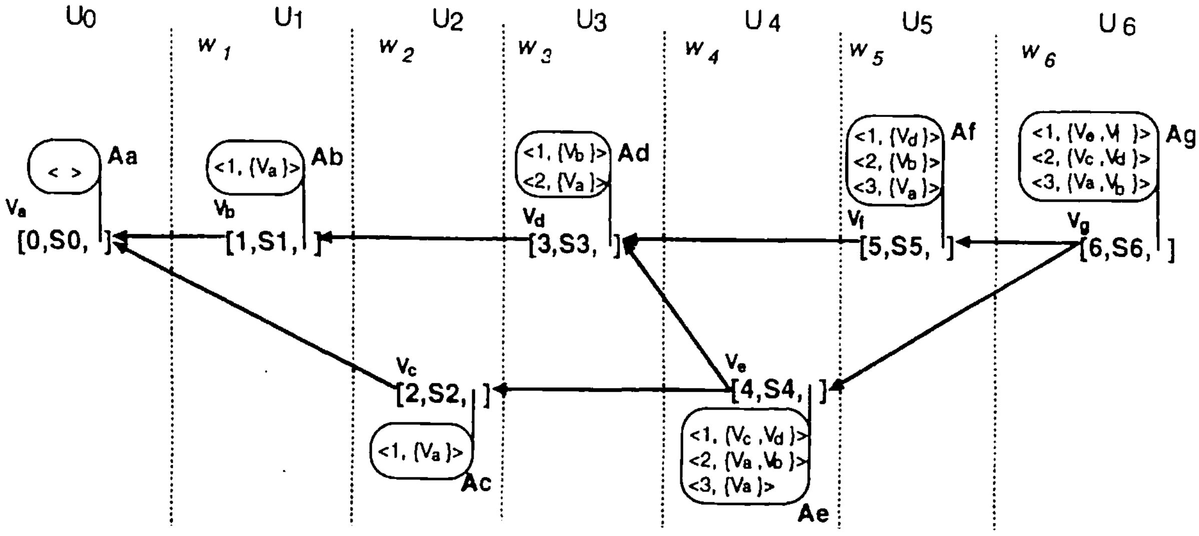


Figure 2.1: An example of GSS showing ancestors table

3 The AGLR Parsing Algorithm

The Kipps algorithm described above does not give us a way to extract any parse results and can be considered only as a recognizer. In this section we modify Kipps algorithm so as to generate all the parse trees. The modification is that we add to each vertices, a link to their parent in the form of stage numbers of the parent, and then store the vertices along with the links in a table called the *vertex table*. We call these links as *parent links*. Using the vertex table and the modified ancestors table, AGLR generates all the possible parse trees. We give a naive version of the AGLR parsing, which does this task.

3.1 The Ancestors Table of AGLR

In AGLR, the one addition in the ancestors table is that, when a new leaf v is 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. The one modification is that the ancestors in the ancestors table of AGLR points to the vertices with parent links in the vertex table.

In case of fig.2.1, 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 ancestors table of AGLR will be stored along with the rule used by reduce actions on the leaf.

$$[\{X \rightarrow Y\ Z\}, \{<0,\{v_g\}>, <1, \{v_e, v_f\}>\}]$$

We call this information an *ancestor item*. In the ancestor item we store the rule used for the reduce action along with the vertices at the corresponding distances. Here note that the ancestor item has only the first two distances of the ancestors table. This is because the rule used for the reduce action has only two symbols in its rhs.

During shift action, for example, let us consider the parser enters into the stage U_7 from the stage U_6 by shifting a look-ahead word w_7 and, if we assume C be the preterminal of the word w_7, the following information is stored.

$$[\{C \rightarrow w_7\}, \{<0,\{v_h\}> \}]$$

and in vertex table, $< v_h : 7(6)>$, (as explained in sec. 3.2) which indicates that, the word w_7 (the word between 6 and 7) is covered by C.

As the AGLR parser is based on Kipps algorithm, and as the addition of 0-th field does not affect the filling of ancestors table, the time consumed in filling up ancestors table will remain the same, i.e, $O(i)$ to fill an ancestors table 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 an input sentence of length n, the time complexity to fill the ancestors table becomes $O(n^3)$, which is same as that of Kipps algorithm.

3.2 The Vertex Table

In AGLR, when a new leaf v is formed during a shift (due to a shift or a goto) action, the vertex table will record <v, $i(PL)$>. Here i is the stage number in which v occurs and PL is a set of ge numbers of the parent vertices of v, and is called the parent links. In case of fig.2.1, the vertex table is created as shown below.

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

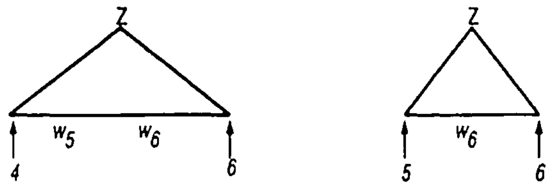
A vertex table is represented using an array with vertices as the pointers and parent links as their contents. Like GSS, the vertex table should be updated after every action. The vertex table differs from the GSS in a sense that, the vertices recorded in the vertex table are not poped. The array representation enables us to access any vertex in the vertex table in $O(1)$ time. Since the number of states in a stage U_i is .stant, say c, (which is equivalent to the number of states in the LR table), and each vertex may have at most $O(\cdot)$ parent links in this stage (because each vertex may have at most $O(i)$ parents), the space consumed by the vertex table in a stage becomes $c * O(i) = O(i)$. For an input sentence of length n, the total space consumed by the vertex table becomes $O(n^2)$.

The time consumed to fill the parent links of a vertex in stage U_i is also in $O(i)$. Since there are $O(n)$ vertices in the vertex table, the time consume in filling the parent links of all the vertices becomes $O(n^2)$. Thus addition of vertex table does not affect the time and space complexity of AGLR.

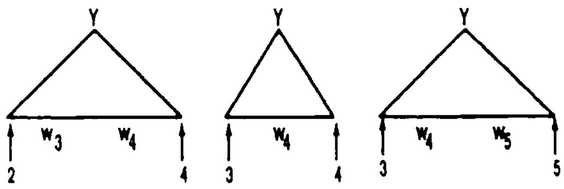
3.3 Representing A Tree using Parent Links

Using the ancestor item created during reduce action in section 3.1 and the vertex table in 3.2, we give an instance of tree generation in AGLR.

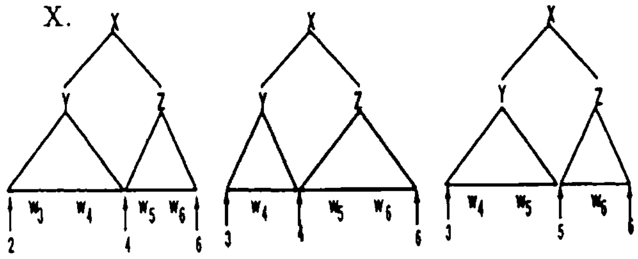
(1) <0,{v_g}> in the ancestor item and <v_g : 6(4, 5)> in the vertex table indicates that, a sequence of words w_5w_6 (the stage number between 4 and 6) and a word w_6 (the stage number between 5 and 6) are covered by Z.



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



(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.



Note that (2) in the above instance, teaches just the portions covered by Y. In this way it is possible for us to extract the partial tree information from the vertex table in co-ordination with the ancestor item.

3.4 Algorithm for Tree Generation

Here we will give the algorithm for constructing a parse tree using the informations asserted from the ancestors table of the leaves and the vertex table. The algorithm for AGLR produces a right parse. In the following algorithm which constructs a parse tree from a set of ancestor items, an ancestor item is represented as $< D_0, A_0 >, < D_1, A_1 >, \cdots, < D_m, A_m >, \cdots, < D_\rho, A_\rho >$, where $D_0, D_1, \cdots D_\rho$ represents distances and $A_0, A_1, \cdots A_\rho$ represents ancestors at corresponding distances. In the algorithm, Ai represents ancestors of the ancestor item and the parent links of each ancestor is represented by PL.

ALGORITHM :

Construction of a right parse from a unique set of ancestor items.

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

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

Method : If no ancestors item of the form $[n, \{S \rightarrow \alpha\}, Ai]$, s.t. $\alpha = X_1\ X_2 \cdots X_m$ and if $< m, \{v_a\} > \in Ai \wedge < v_a, 0(\ ) > \notin VT$, then w is not in L(G), so emit "error" and halt. Otherwise execute the routine $O([n, \{S \rightarrow \alpha\}, Ai])$, the routine O is defined as follows.

```
Routine $O([i, \{A \rightarrow \beta\}, Ai])$ :
(1) If $\beta = X_1\ X_2 \cdots X_{m-1}\ X_m$ then,
        set $k = m$, $l = 0$, $r = i$.
(2) (a) If $X_k \in T$, subtract 1 from $k$ and $r$,
                add 1 to $l$.
    (b) If $X_k \in N$ then for $< D_l, E_l > \in Ai$,
        find $v_h \in E_l$ s.t. $< v_h, r(PL) > \in VT$
                and $e \in PL$ then,
        find an ancestor item
                $[r, \{X_k \rightarrow \gamma\}, Ai']$.
        Then execute $O([r, \{X_k \rightarrow \gamma\}, Ai'])$.
        Subtract 1 from $k$, add 1 to $l$,
            set $r = e$.
(3) Repeat step (2) until $k = 0$. Halt.
```

It is also possible to produce a left parse with a simple modification in this algorithm. Note that in an ancestors table, there may be more than one ancestors grouped (as explained in sec.3.5) at each distance $D_0, D_1, \cdots, D_\rho$, which represents more than one way of parsing the input. To generate a tree deterministically, in the worst case, all the possibilities have to be considered during the successive backtrack. This operation makes us to use $O(n^2)$ time to generate a tree deterministically.

3.5 A Property of the Ancestors Table

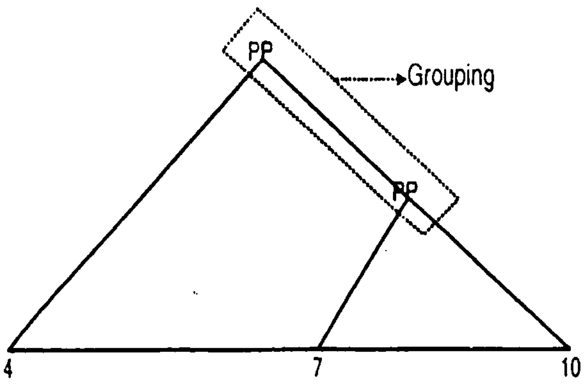
Here we state about a property of the ancestors table. When two or more vertices happens to be the ancestors of a vertex (say v_a), provided they are <u>at the same distance</u> from v_a, then those ancestor vertices are grouped into the ancestors table of v_a at the corresponding distance. In contrast, in the packed forest of TGLR, if there are two or more vertices which are parent to a vertex, they can be packed if and only if the parents are at the same stage.

This is one of the important properties of the ancestors table. We call this property as *grouping*, to differentiate between packing. Like packing in TGLR, grouping also represents the ambiguities of the parsed sentence. The ancestor item along with the vertex table in AGLR and the packed forest in Tomita, both represents partial parse informations.

The following figure shows the grouping phenomenon for PP. In the figure, the dotted line shows the grouped vertices, which are represented in the ancestor item and in the vertex table given below.

$[\{PP \rightarrow p\ NP\}, \{<0, \{v_h\}>, <1, \{v_f, v_g\}>, <2, \{v_d, v_e\}>\}]$

[$\cdots\cdots$, <v_d : 4(...)>, <v_e : 7(...)>, <v_f : 5(4)>, <v_g : 8(7)>, <v_h : 10(5,8)>, $\cdots\cdots$]



In the example, the vertices v_f and v_g at distanc are grouped. The vertex v_f is in stage U_5 and v_g is in U_8. Similarly, v_c and v_d at distance 2 are grouped, which are in U_7 and U_4 respectively. Thus in the above figure, the PP between 4 and 10, and 7 and 10 are grouped into one ancestor table, even though they are at different levels.

Even though grouping does not affect other complexities, the only disadvantage with grouping is that it makes the tree generation process to cost more time. In case of packed forest representation, a tree can be generated in $O(n)$ time, where as in our case, because of grouping it takes $O(n^2)$ time to generate a tree using the ancestor item and the vertex table.

4 ractical Results

In this section we show some practical results giving support to our theoretical results. We use a real grammar which is frequently applied in the natural language processing. This grammar consists of 394 rules and is same as the grammar-IV used in [9]. The original version of this grammar was developed by Takakura [7]. Two types of sentences were used. Sentence set I consists of natural sentences appearing in text books (refer [9]). The sentence set II consists of PP-attachment sentence which has the pattern n v det n (p det n)^m, where $m \geq 0$. Since AGLR is a parser, it is unfair to compare it with Kipps algorithm, which is a recognizer. For this reason, we compare our practical results with Tomita parser. Both these parsers were implemented in C. For the results of other grammars, refer [8].

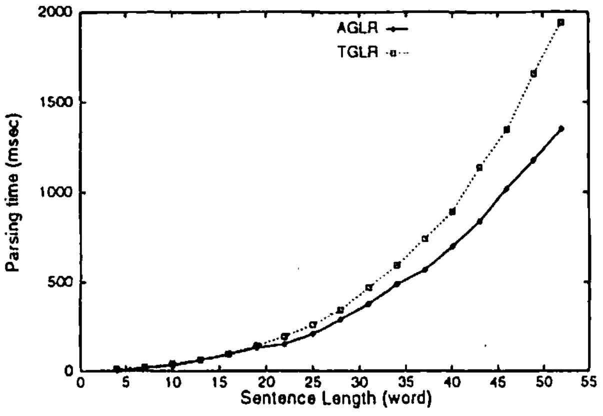


Figure 3.1(a): For sentence set II.

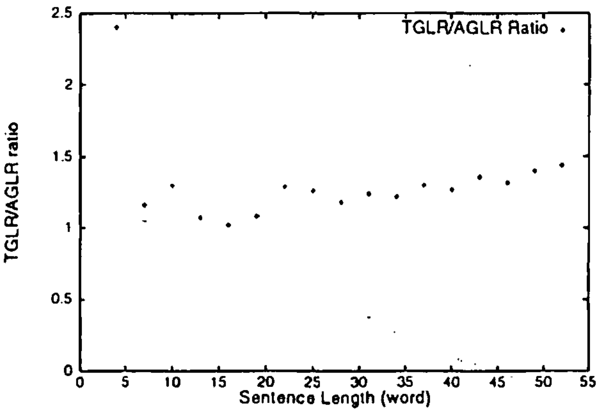


Figure 3.1(b): For sentence set II

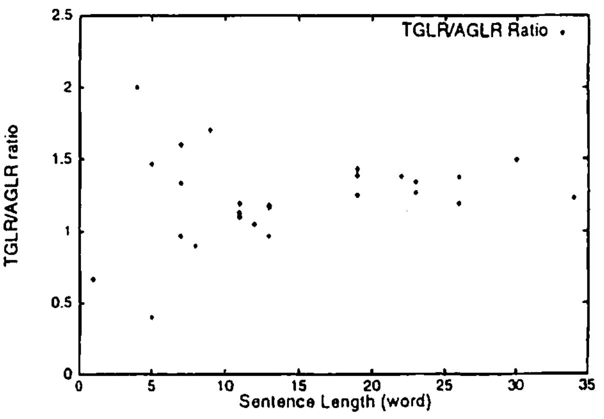


Fig 3.1(c):TGLR/AGLR ratio for sentence set I

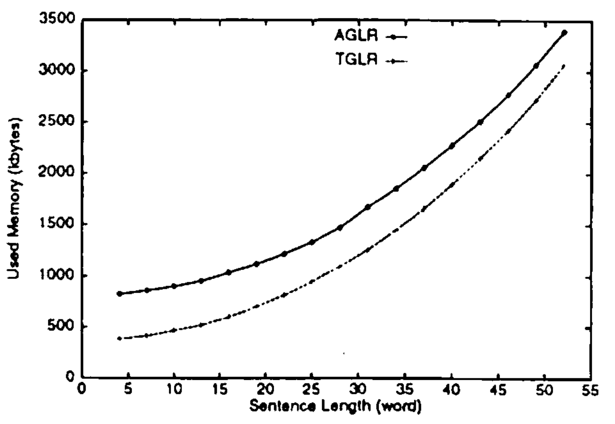


Fig 3.2 Memory consumed for sentence set II.

Figure 3.1(a) and 3.1(b) shows the result of parsing sentence set II using the grammar. Figure 3.1(a) shows that, AGLR performs faster as the input length and the ambiguity increases. The TGLR/AGLR ratio in fig. 3.1(b) makes this point clear. On average the AGLR parser is 1.31 times faster than Tomita parser.

Figure 3.1(c) gives the result of parsing sentence set I. This figure shows that the TGLR/AGLR ratio of parsing time is greater than 1 in most of the cases, and on average AGLR is 1.26 times faster than Tomita parser.

It should be noted that most of the rules (about 54.1%) in this grammar is not in Chomsky normal form. The average length on rhs of grammar rules in this grammar is 2.75. This is the reason why in the above graphs, AGLR performs faster than Tomita. If we use grammars with Chomsky normal form, or whose average rhs length becomes near to 2, both Tomita and AGLR will give the same practical performance as the theoretical time complexity becomes O(n^3).

When using high dense ambiguous grammars (S $\rightarrow$ a, S $\rightarrow$ S S, S $\rightarrow$ S S S S) and the inputs like the one given in [4], when the input length ranges from 20 to 30, AGLR performs 50 to 150 times faster than Tomita parser. For this grammar in [4], the time complexity of TGLR becomes O(n^5), where as AGLR is O(n^3), which agrees with the theoretical results.

In figure 3.2, we give the comparison of memory space consumed by the Tomita and AGLR parsers. The memory space is used mainly by GSS, packed forest in case of Tomita, and ancestors table and vertex table in case of AGLR.

Since we use ancestors table and vertex table in AGLR, the memory space used by AGLR parser is increased by a constant factor compared with Tomita. This is because the ancestors table consumes an extra O(n^2) space.

Our theoretical results on complexity are summarized in table 1.

Complexity Resources	TGLR	Kipps	AGLR
Parsing Time	$n^{1+\rho}$	n^3	n^3
GSS space	n^2	n^2	n^2
Partial parse info.	n^3	-	n^3
Tree Extraction	n	-	n^2

Table 1 : Complexity Table

5 Conclusion

In this paper we improved Kipps algorithm and complete it to generate all the parse trees. This is achieved by introducing a vertex table, which records the parent links of each vertices. Using the vertex table and the ancestors table, we presented a method to generate a parse tree in O(n^2) time. We proved that the time and space complexity of AGLR as O(n^3) and O(n^2) respectively, both theoretically and practically.

It should be noted that as mentioned in sec.3.4, even though we could able to generate all the parse trees, it takes O(n^2) time to generate a parse tree, which is same as Earley's [1, 3]. Thus we would like to conclude that our improved GLR parser improves Kipps algorithm as a practical parser by generating all the possible parse trees and we would like to acknowledge that, the benefits of the vertex table and the ancestors table are nullified, because in case of Tomita's packed forest, it takes only O(n) time to generate a parse tree.

Reference

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

2. Billot, S and Lang, B. : *The Structure of Shared Forests in Ambiguous Parsing*, Proc. of ACL'91, pp.143-151, 1991.

3. Earley, J. : *An Efficient Augmented-Context-Free Parsing Algorithm*, Comm. of ACM, 13, 1-2, pp.95-102, 1970.

4. Johnson, M. : *The Computational Complexity of Tomita's Algorithm*, Generalized LR Parsing, Kluwer Academic Publishers, ·.35-42, 1991.

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. Schabes, Y. : *Polynomial Time and Space Shift-Reduce Parsing of Arbitrary Context-free Grammars*, Proc. of 29th ACL, pp.106-115, 1991.

7. Takakura and Tanaka, H. : *B.Tech thesis*, Dept. of Computer Science, Tokyo Institute of Technology, 1984.

8. Tanaka, H, Suresh, K.G and Yamada, K. : *A Family of Generalized LR Parsing Algorithms Using Ancestors Table*, in Technical . port, 92TR-0019 Department of Computer Science, Tokyo Institute of Technology, 1992.

9. Tomita, M. : *Efficient Parsing for Natural Language*, Kluwer, Boston, Mass, 1986.

10. Toshiyuki Hanazawa, Kita, K., Nakamura, S., Kawabata, T. and Shikano, K. : *ATR HMM-LR Continuous Speech Recognition System*, Proceedings of 1990 International Conference on Acoustics, Speech, and Signal Processing, S2.4, pp.53-56, Albuquerque, 1990.

11. Jaime G. Carbonell and Masaru Tomita. : *Knowledge-based machine translation, the CMU approach*, in Machine Translation - Theoretical and methodological issues, Ed. Sergei Nirenburg, pp.68-89, 1987.