A Comparison of CYK and Earley Parsing Algorithms

Te Li
Arizona State University
te.li@asu.edu

Devi Alagappan
Arizona State University
devi.alagappan@asu.edu

Abstract

Parsers for programming languages are usually based on context-free grammars. So designing efficient parsing algorithms for context-free grammars is critical. CYK algorithm and Earley algorithm are two well-known ones. We compare these two algorithms in terms of not only time but also space. The space complexity analysis in Earley's paper is not accurate as we show in the paper. Extensive experiments based on varying sizes of grammars and input strings are conducted. The results show that Earley algorithm is almost always better than CYK algorithm with regard to both time and space.

1 Introduction

We chose problem 2, parsing context-free languages, since we think the problem is very interesting and challenging. The CFG parsing problem has been studied for more than 4 decades. It would be interesting to compare some of the prominent algorithms such as CYK algorithm [3] and Earley algorithm [2]. We compare the two algorithms on two dimensions: time and space.

Our major contributions are:

1. Implementation of CYK algorithm and Earley algorithm
2. Comparison of the space complexity of the two algorithms
3. Extensive experiments based on varying grammar sizes and input sizes

The rest of the paper is organized as follows. We analyze the time complexity and the space complexity of both CYK algorithm and Earley algorithm in Section 2. Then implementation details are given in Section 3. The experimental results are explained in Section 4. Section 5 concludes the paper.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage.

2 Analysis

We first compare the time complexity of the two algorithms and then compare their space complexity.

2.1 Time Complexity

The time complexity for both algorithms is $\mathcal{O}(n^3)$ as given in [3, 2]. However, $\mathcal{O}(n^3)$ is only an upper bound of Earley and on most grammars Earley does better than $\mathcal{O}(n^3)$.

2.2 Space Complexity

The space complexity for CYK is $\mathcal{O}(n^2)$ since there are $\mathcal{O}(n^2)$ entries in the table, while the space complexity for Earley is $\mathcal{O}(n)$, not $\mathcal{O}(n^2)$ as analyzed in [2].

To analyze the space complexity of Earley algorithm, we only consider the major data structure, i.e. the states of all the state sets. The total number of states that could be added into the state sets is bounded by $(b+1)(r+1)(n+2)$, where b is the maximum length of the body of a rule in the grammar, r is the number of rules, and n is the length of the input string. To make a state different from the others, there must be at least one different element in the 4-tuple state. There can be at most $(b+1)$ different j's (as defined in [2]) for each rule. There are $(r+1)$ rules since a new rule $\phi \rightarrow S \dashv$ is added in the first state set, where S is the start symbol of the grammar. And there can be $(n+2)$ different pointer f's (as defined in [2]) for a given input string of length n. Thus, the space complexity of Earley is $\mathcal{O}(n)$.

3 Implementation

The system is implemented using Eclipse 3.2 and JRE 5.0 update 6. In this section, we introduce the architecture of the system first, then the implementation details about the two algorithms, and finally our approach to measuring time and space utilized by them.

3.1 Architecture

There are 3 major components in the system: Grammar Builder, CYK Parser, and Earley Parser. Grammar Builder takes the grammar input in the form spec-

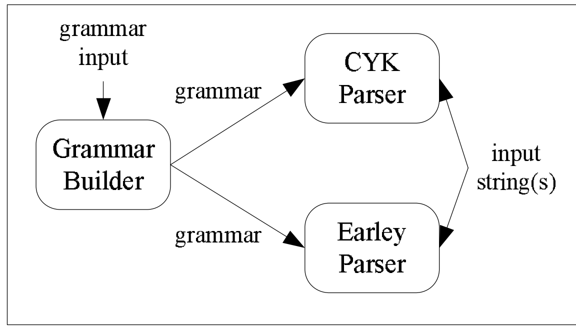


Figure 1: Architecture of the System

ified in the assignment, and builds grammar objects for both CYK Parser and Earley Parser, using HashMap class for the rules. The rules in the grammar object for CYK use body as the key since CYK builds the table in a bottom-up fashion. For example, CYK computes each entry in the first row of the table by looking up the rules which have as the body the corresponding symbol of the input string. The rules in the grammar object for Earley use head as the key since Earley uses a top-down approach to construct the state sets, i.e. creating new states by replacing a variable with its bodies.

CYK Parser and Earley Parser take as inputs the corresponding grammar object from the Grammar Builder and an input string and output true if the string is in the language of the grammar and false if not.

3.2 CYK Implementation

The flowchart of the CYK algorithm implementation is shown in Figure 2. Our implementation of CYK algorithm works with a larger class of context-free grammars, called *CNF-like grammars*. CNF-like grammars are similar to CNF since the body of each rule is either one symbol or two symbols, and empty rule for the start variable is allowed. Formally,

Definition 1 (CNF-like Grammar)**.** A context-free grammar is a CNF-like grammar if every rule is of the form

$$A \to xy$$
$$A \to z$$

where A is any variable and x, y and z are either a terminal or a variable, but not the start variable. In addition, the rule $S \to \epsilon$ is allowed, where S is the start variable.

Obviously, CNF grammars are CNF-like grammars. The reason that CYK works with CNF-like grammars is that the process for computing X_{ij}'s in the case of CNF grammars still applies to those CNF-like grammars which are not CNF grammars. The proof is straightforward and omitted.

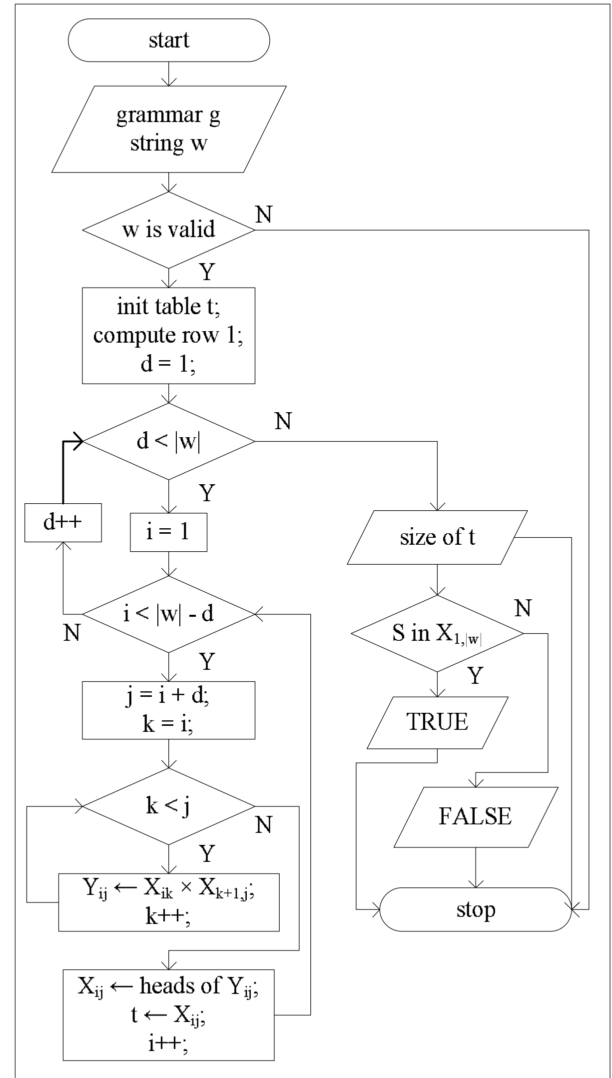


Figure 2: Flowchart of CYK Algorithm

Given a CNF-like grammar g and a string w, the parser checks the validity of w. If w is "z", the epsilon symbol, or contains only terminals, then w is valid; otherwise, the parser throws an exception and exits. Then the CYK table t is initialized, and its first row is computed as follows: the elements of the ith entry where $i = 1, 2, \ldots, |w|$ are the heads of the ith symbol of w.

Let i be the row number, j the column number, and d the difference between i and j. So $j = i + d$. The parser computes row $(d + 1)$ for $d = 1, 2, \ldots, |w| - 1$. X_{ij} is computed as follows. Let Y_{ijk} be the cross product of the sets X_{ik} and $X_{k+1,j}$, where $i \leq k < j$. Y_{ij} is the union of all the sets Y_{ijk}. Find the heads of each element in Y_{ij} and add them to X_{ij}.

After the whole table is filled in, compute and output the space occupied by the table using the method described later in section 3.4. If $X_{i,|w|}$ contains the start symbol S, then the parser returns true; other-

wise, it returns false.

3.3 Earley Implementation

The Earley algorithm implementation is given in Algorithm 1. The inputs are a CFG g and a string $w = X_1X_2 \cdots X_n$, where n is the length of w. At line 2, a space is added to the beginning of w to ensure the index i is the same as that in [2], and a $\dashv$ is added to the end of w to mark the end. S_i denotes the ith state set. The loop from lines 6 to 29 is to iteratively process each state set. Lines 7 to 25 is to process every state in S_i. S_{ix} represents the xth state in S_i. A state S_{ix} is a 5–tuple $\langle head, bodyID, j, f, \alpha\rangle$, where *head* is the head of a rule, *bodyID* is the ID of the body of a rule, and j and f are the same as defined in [2]. Lines 8 to 13 corresponds to the completer operation. Lines 14 to 22 corresponds to the predictor operation. Lines 23 to 25 corresponds to the scanner operation. The accept state S_{accept} at line 28 is $\langle \phi, 0, 2, 0, \text{“ ”}\rangle$.

We used empty string as lookahead. The most important reason for this is that CYK doesn't use lookahead. Another reason is it is not clear from the Earley paper what would be the lookahead for the states where the current symbol after the point is not followed by a terminal. Some other reasons are also explained in [1].

3.4 Time and Space Measurement

For both algorithms, we measure time taken to parse the input strings and memory space occupied by the major data structures, namely, table in CYK algorithm and state sets in Earley algorithm.

The *nanoTime* method in *System* class is used to get the current time before and after parsing. The difference between the two values is reported as the execution time.

The serialization feature of Java is used to measure the space. An *ObjectOutputStream o* is wrapped in a *ByteArrayOutputStream b*. Before exiting the parsing methods, the major data structures are written into o. Then the size of b is output as the space.

Algorithm 1 Earley(g, w)

Require: w is valid.

1: $n \Leftarrow |w|$
2: $w \Leftarrow$ “␣” $+\, w +$ “$\dashv$”
3: initialize $S_0, \ldots, S_n$
4: add $\phi \rightarrow \cdot E \dashv$ to S_0
5: add $\phi \rightarrow E \dashv$ to $g.rules$
6: **for** $i = 0$ to n **do**
7:   **for** $x = 1$ to $|S_i|$ **do**
8:     **if** $S_{ix}.j = p$ **then**
9:       **if** $S_{ix}.\alpha = X_{i+1} \ldots X_{i+g.k}$ **then**
10:         {case: completer}
11:         **for** $m = 1$ to $|S_f|$ **do**
12:           **if** $S_{fm}.symbol = S_{ix}.head$ **then**
13:             add S_{fm} with $S_{fm}.j$ ++ to S_i
14:     **else** {S_{ix} is not final}
15:       **if** $S_{ix}.symbol$ is nonterminal **then**
16:         {case: predictor}
17:         $bodies \Leftarrow$ bodies of $S_{ix}.symbol$
18:         **for** $m = 0$ to $|bodies|$ **do**
19:           $remain \Leftarrow bodies[m].substr(S_{ix}.j + 2)$
20:           $\Lambda \Leftarrow H(remain + S_{ix}.\alpha)$
21:           **for all** $\alpha \in \Lambda$ **do**
22:             add $\langle S_{ix}.symbol, m, 0, i, \alpha\rangle$ to S_i
23:       **else** {case: scanner}
24:         **if** $X_{ix}.symbol = X_{i+1}$ **then**
25:           add S_{ix} with $S_{ix}.j$ ++ to S_{i+1}
26:   **if** S_{i+1} is empty **then**
27:     **return** false
28:   **if** $i = n$ and $S_{i+1} = \{S_{accept}\}$ **then**
29:     **return** true
30: **return** false

4 Experimentation

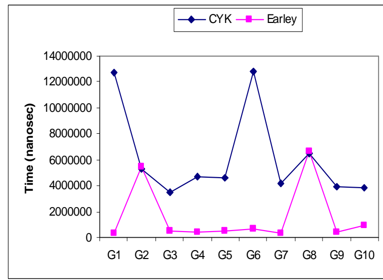
We have programmed the two algorithms in Java and conducted the experiments on a 1 GHz Pentium III with 512 MB RAM running Windows 2000. Each experiment was repeated 10 times. The maximum and minimum values were taken away and the rest 8 results were averaged to ensure accuracy.

We compare the algorithms on 10 grammars [Appendix A] of varying sizes (number of rules) ranging from 1 to 10. G_2 and G_8 are ambiguous grammars and the rest are unambiguous. All the grammars are CNF-like grammars [see Definition 1]. Some of the grammars (G_1, G_2, G_7, and G_8) are CNF grammars and some are non-CNF grammars. In addition, the input strings are randomly generated for the experiments by randomly choosing terminals of the grammar. The lengths of the strings are $10, 20, \ldots, 100$.

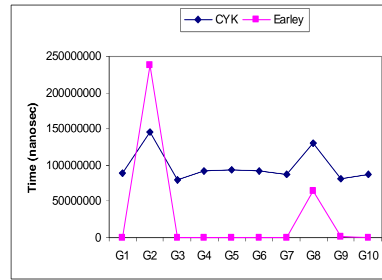
The graphs are plotted by fixing either the size of a grammar or the length of the string, and varying the other.

4.1 Time Comparison

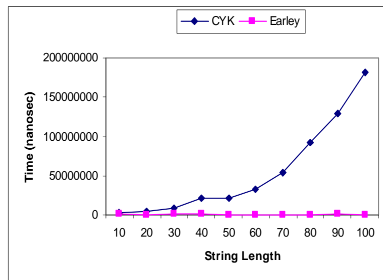
In Figure 3(a), CYK always performs worse than Earley except for G_2 and G_8 which are ambiguous grammars. In Figure 3(b), the case is almost the same as in (a), but CYK is better for G_2 and worse for G_8. This pattern occurred since the two grammars are ambiguous, the execution time of the two algorithms depends on the actual input strings. In Figure 3(c), for both algorithms, the parsing time increases as the length of strings increases. No matter what the length of the string is, Earley outperforms CYK substantially. Actually, the curve for Earley is almost linear to the length of the string. Even for ambiguous grammars, the same conclusion holds as shown in Figure 3(d).



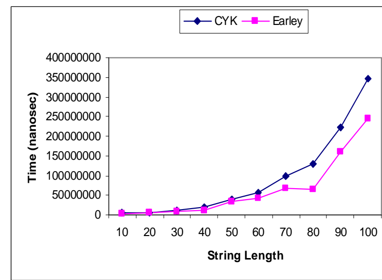
(a) Time, $|w| = 20$



(b) Time, $|w| = 80$

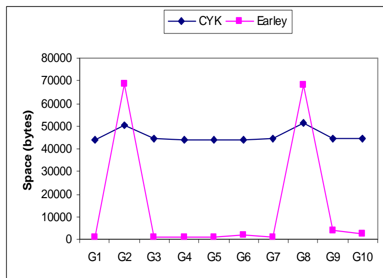


(c) Time, G_4

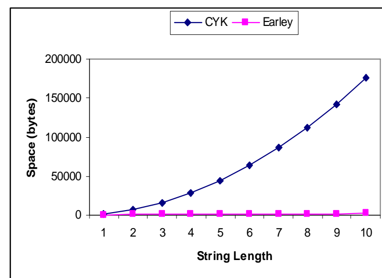


(d) Time, G_8

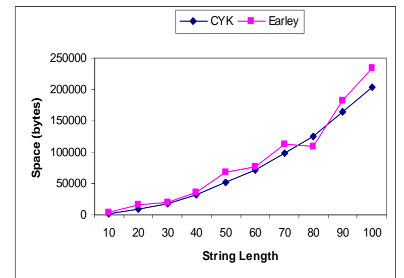
Figure 3: Execution Time of CYK and Earley



(a) Space, $|w| = 50$



(b) Space, G_4



(c) Space, G_8

Figure 4: Space Utilization of CYK and Earley

4.2 Space Comparison

In Figure 4(a), given an input string, the curve of both algorithms is constant for unambiguous grammars. For ambiguous grammars (G_2 and G_8), the space is also constant but bigger than in the case of unambiguous grammars. Earley always used much smaller space than CYK except for ambiguous grammars. As can be seen from figures 4(b) and 4(c), the space used by Earley is much smaller than CYK if the grammars are unambiguous; however, in the case of ambiguous grammars, the two algorithms utilized almost the same amount of space. Another observation from Figure 4(c) is that the space used by both algorithms is proportional to the length of the input string.

5 Conclusion

To conclude, all the experimental results show that Earley always outperforms CYK in terms of both time and space, if the grammars are unambiguous. The parsing time of Earley in case of ambiguous grammars is unpredictable since it depends on the actual input strings. Given an ambiguous grammar, the two algorithms use almost the same space irrespective of the length of the input strings. For a given string, Earley uses more space in case of ambiguous grammars.

References

[1] J. Aycock and N. Horspool. Practical Earley Parsing. *The Computer Journal*, 45(6):620–630, 2002.

[2] J. Earley. An Efficient Context-Free Parsing Algorithm. *Commun. ACM*, 13(2):94–102, 1970.

[3] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, And Computation*, chapter 7, pages 228–302. Addison-Wesley, 2001.

Appendix

A List of Grammars

The size of G_i is i, where $i = 1, 2, \ldots, 10$.

1. $\mathbf{G_1}$:
   $A \to a$

2. $\mathbf{G_2}$:
   $A \to a$
   $A \to AA$

3. $\mathbf{G_3}$:
   $S \to Ab$
   $A \to a$
   $A \to Ab$

4. $\mathbf{G_4}$:
   $S \to A$
   $A \to BC$
   $B \to b$
   $C \to a$

5. $\mathbf{G_5}$:
   $S \to \epsilon$
   $S \to A$
   $A \to BC$
   $B \to b$
   $C \to a$

6. $\mathbf{G_6}$:
   $S \to \epsilon$
   $S \to A$
   $A \to BC$
   $B \to b$
   $B \to AA$
   $C \to a$

7. $\mathbf{G_7}$:
   $S \to AB$
   $A \to a$
   $A \to SC$
   $B \to b$
   $B \to BB$
   $C \to c$
   $C \to CA$

8. $\mathbf{G_8}$:
   $S \to AB$
   $S \to BC$
   $A \to BA$
   $A \to a$
   $B \to CC$
   $B \to b$
   $C \to AB$
   $C \to a$

9. $\mathbf{G_9}$:
   $S \to \epsilon$
   $S \to A$
   $A \to BC$
   $A \to AC$
   $A \to aB$
   $B \to b$
   $B \to AA$
   $C \to c$
   $C \to BA$

10. $\mathbf{G_{10}}$:
   $S \to \epsilon$
   $S \to A$
   $S \to bb$
   $A \to BC$
   $A \to AC$
   $A \to aB$
   $B \to b$
   $B \to AA$
   $C \to c$
   $C \to BA$