An Improved Algorithm for Parsing Context-Free Grammars

Stefan Andrei, Hikyoo Koh

No. 1, March 2011

Computer Science Department Technical Reports - Lamar University

Computer Science Department, Lamar University,
211 Red Bird Lane, Box 10056, Beaumont, TX 77710, U.S.A.
URL: http://cs.lamar.edu/tech_reports
Email: tech_reports@cs.lamar.edu
An Improved Algorithm for Parsing Context Free Grammars

Stefan Andrei
Lamar University
Computer Science Department
Beaumont, TX, USA
Stefan.Andrei@lamar.edu

Hikyoo Koh
Lamar University
Computer Science Department
Beaumont, TX, USA
Hikyoo.Koh@lamar.edu

Abstract. The best known algorithm to test membership of a word in an arbitrary context-free language takes $O(n^3)$ time, whereas the membership in a deterministic context-free language can be tested in $O(n)$ time by Knuth, where $n$ is the length of the input word. According to Aho and Ullman, identifying the productions that generate a word by an arbitrary context-free grammar defined in Chomsky normal form requires a time complexity of $O(n^2)$ and a work-space complexity is $O(n)$, where $n$ is the length of the input word. The contribution of this paper is two-fold. Firstly, we describe a refined algorithm for parsing context free grammars which has an average time complexity of $O(n \cdot \log n)$. Secondly, we show that its work-space complexity is $O(\log n)$. The experimental results indicate that although the work-space complexity has been significantly improved from $O(n)$ to $O(\log n)$, the execution time of the new parsing algorithm has decreased with about 5%.

1 Introduction

A context-free grammar is a 4-tuple consisting of a non-empty set of variables (also known as non-terminal symbols and denoted by $V$), a non-empty set of terminal symbols (denoted by $T$), a start variable (denoted by $S$), and a set of simple rules (also known as productions and denoted by $P$) [AhU72, Reg09]. A context-free rule has the general form $X \rightarrow \alpha$, where $X \in V$ and $\alpha$ is a word over $V \cup T$. These rules can be repeatedly applied in order to generate all and only valid words. For any words $u$ and $v$ over $V \cup T$, we say $u$ yields $v$, written as $u \Rightarrow v$, if there exists a rule $X \rightarrow \alpha$ and words $u_1, u_2$ over $V \cup T$ such that $u = u_1 X u_2$ and $v = u_1 \alpha u_2$. We say that $u \Rightarrow^* v$ and call it a derivation, if there exist words $u_1, \ldots, u_n \in V \cup T$ such that $u \Rightarrow u_1 \Rightarrow \ldots \Rightarrow u_n \Rightarrow v$. If we denote by $\pi$ the set of rule indexes of the previous derivation, then $u \Rightarrow^* v$ becomes $u \Rightarrow^*_\pi v$. A word obtained from the start symbol by replacing the non-terminal symbol is called a sentential form. The set of words containing only terminal symbols generated from the start symbol by applying the grammar rules becomes the grammar's language.

An equivalent accepting device for a context-free language is the so-called pushdown automata. A deterministic context-free language is one for which there exists a deterministic pushdown automata that accepts it. A context-free grammar is ambiguous if its generated language has words with two or more sequences of left-most derivations (a derivation is left-most if the next sentential form is always derived by rewriting the leftmost non-terminal symbol of the current sentential form in the sequence). A context-free language $L$ is said to be inherently ambiguous if all its grammars are ambiguous. If even one grammar for $L$ is unambiguous, then $L$ is an unambiguous language. All deterministic context-free languages can be generated by unambiguous grammars. Unfortunately, the deterministic context-free languages are not exactly equal to the subset of context-free languages that are not inherently ambiguous. For example, the palindrome language over $\{0, 1\}$ has an unambiguous context-free grammar, that is, $S \rightarrow 0S0 |
1$1 | \varepsilon$, where \( \varepsilon \) denotes the empty word. The palindrome language is not a deterministic context-free language [HMU2007]. There are many other non-deterministic context-free languages, such as \( L = \{ 0^n \ 1^n \mid n \geq 1 \} \cup \{ 0^n \ 1^{2n} \mid n \geq 1 \} \). Also, there are many languages that are inherently ambiguous, such as \( L = \{ a^n b^n c^n d^n \mid n \geq 1, m \geq 1 \} \cup \{ a^n b^m c^m d^n \mid n \geq 1, m \geq 1 \} \).

The best known algorithm to test membership of a word in an arbitrary context-free language takes \( O(n^3) \) time, whereas the membership in a deterministic context-free language can be tested in \( O(n) \) time, where \( n \) is the length of the input word. Note that this time and space complexities hold also for non-deterministic and inherent ambiguous languages.

A context-free grammar is in the Chomsky normal form if it has only productions of the form \( A \rightarrow BC \) and \( A \rightarrow a \), where \( A, B, C \in V \) and \( a \in T \). A classical result says that any context-free grammar with no empty productions can be polynomially reduced to an equivalent context-free grammar in Chomsky normal form. For example, the palindrome language above can be generated by the following context-free grammar expressed in Chomsky normal form: \( A \rightarrow 0, B \rightarrow 1, S \rightarrow \varepsilon, S \rightarrow AA, S \rightarrow BB, S \rightarrow AC, S \rightarrow BD, C \rightarrow SA, D \rightarrow SB \). According to Aho and Ullman, identifying the productions that generate a word by a context-free grammar defined in Chomsky normal form requires a time complexity of \( O(n^2) \) and a work-space complexity is \( O(n) \), where \( n \) is the length of the input word.

The algorithm presented below is related to Cocke-Younger-Kasami (CYK) Algorithm for testing the membership problem for context free grammars ([AhU72, HoU78]). The CYK Algorithm has as input a context free grammar \( G = (V_N, V_T, x_0, P) \), a word \( w = a_1 a_2 ... a_n \) over \( V_T \), and uses the sets of non-terminals \( V_y = \{ A \mid A \in V_N, A \Rightarrow^* a_i a_{i+1} ... a_{i+j-1} \} \), where \( 1 \leq i < n \) and \( 1 < j \leq n - i \), to answer the question "\( w \in L(G) \) ?". In the following, we present the classic algorithm for obtaining the derivation (that is, the set of productions) starting with \( x_0 \) and ending with the input word \( w \), in case the word \( w \) belongs to the grammar's language ([AhU72, HoU78]).

**Algorithm (ALG1):**

**Input:** \( G = (V_N, V_T, x_0, P) \) a context-free grammar (in Chomsky normal form), the sets \( V_y \) (computed by the CYK Algorithm), \( w \in V_T^+ \cap L(G) \);

**Output:** The derivation \( \pi \) such that \( x_0 \Rightarrow^* \pi \ w \).

**Method:**

First, we describe in pseudo-code the \texttt{parse()} method, followed by the main program which calls the \texttt{parse()} method. We denote by \( \#(x \rightarrow r) \) the index of the production \( x \rightarrow r \) from the set of productions \( P \). The derivation of the word \( w \) will be stored in the array \( d[] \) with the meaning \( d[i] = j \) if and only if at step \( i \), the production of index \( j \) was applied. The \texttt{parse}(i, j, A) method will save the derivation \( A \Rightarrow^* a_i a_{i+1} ... a_{i+j-1} \) into array \( d[] \).
void parse(i, j, A) {
    if (j == 1) and (m == #(A -> a_i)) {
        h++;
        d[h] = m;
    }
    else
        if \exists k = \min \{ l \mid 1 \leq l \leq j - 1, \exists A -> BC, B \in V_{lb}, C \in V_{i+t(t-1)} \} {
            h = h++;
            d[h] = #(A -> BC);
            parse(i, k, B);
            parse(i + k, j - k, C);
        }
}

The main program calling the parse() method is:

    h = 0;
    if (x_0 \in V_{ln}) {
        parse(l, n, x_0);
        println("The derivation for w is " + d[]);
    }
    else
        println("The word w is not generated by the grammar");

Theorem 1.1. (correctness and time complexity of Algorithm (ALG1), Theorem 4.8, [AhU72])
Let $G$ be a context free grammar in Chomsky normal form and $w \in V_T^*$ as input of Algorithm (ALG1). Then the algorithm gives as output the derivation $\alpha w_1 \ldots w_2 \beta$ if $w \in L(G)$ and the message "The word $w$ is not generated by the grammar" otherwise. Furthermore, the number of elementary steps of the Algorithm (ALG1) is $O(n^2)$.

Next section describes our improved parsing algorithm.

2 Our improved parsing algorithm

This section describes our contribution, namely an improved parsing algorithm. More precisely, we present the following new results:

1. a refinement of Algorithm (ALG1), called Algorithm (ALG2), and a result proving that its average time is $O(n \log n)$, where $n$ being the length of the input word $w$;
2. a proof establishing that the work-space complexity of Algorithm (ALG2) is $O(\log_2 n)$ instead of $O(n)$ as in the case of Algorithm (ALG1).

One of the differences of our algorithm and the classical Algorithm (ALG1) is that our algorithm may have not only left derivations, but also some right derivations (a derivation is right-most if the next sentential form is always derived by rewriting the rightmost non-terminal symbol of the current sentential form in the sequence. The input-space complexity of Algorithm (ALG1) is formed by the space needed for storing the context free grammar $G$ (which is the same for any input word), the space needed for storing the sets $V_y$ (which is $O(n^2)$) and the space needed for storing the input word $w$ (which is $n$). The work-space complexity is the supplementary space needed for executing the algorithm. The output-space complexity consists of the space needed for syntactic analysis (which is $O(n)$).

Regarding the work-space complexity, we remark that the stack of recursive calls of the procedure $\text{parse()}$ has the maximum length when for every input of the form $(i, j)$, we obtain $k = j - 1$ (inside the $\text{parse()}$ method). In that situation, the maximum length of the stack is $n - 1$. Therefore, for Algorithm (ALG1), the work-space complexity is $O(n)$. To refine this algorithm and get a better algorithm, we shall not store in the stack the pairs $(i + k, j - k)$, but after each call of the procedure $\text{parse()}$ we shall introduce in the stack the pair of maximum length (of course, if it has the length greater than 1), processing the other subsequence (the smaller one). Thus the recursive calls are:

\[
\text{parse}(i, k, B);
\]

\[
\text{parse}(i + k, j - k, C);
\]

Obviously, these recursive calls correspond to the derivations $B \Rightarrow^{*} a_i a_{i+1} \ldots a_{i+k-1}$ and $C \Rightarrow^{*} a_{i+k} a_{i+k+1} \ldots a_{j-1}$. Thus, the lengths of the corresponding sequences are $k$ and $j - k$, respectively. According to what we have discussed above, the refined algorithm will consider the smaller sequence to be executed first. Therefore, the above situation corresponds to the case $k \leq j - k$, otherwise we have to reverse the order of the recursive calls as follows:
parse(i + k, j - k, C);

parse(i, k, B);

Changing the order of the recursive calls may cause changes in the derivation provided as output by (ALG1). The derivation provided as output by Algorithm (ALG1) has only left derivations (that is, at every step of the derivation, the leftmost non-terminal symbol is rewritten). By switching the order of recursive calls, the final derivation will then contain some right derivations (that is, the rightmost non-terminal symbol is rewritten).

Let us discuss the problem of which non-terminal symbol will be rewritten using the grammar rules. We consider a global variable, called pos, such that if \( a = x_1 x_2 \ldots x_m \) is the current propositional form, then the corresponding rule (which has to be applied) is \( x_{\text{pos}} \rightarrow r \), where \( r \) can be a two non-terminal sequence \((BC)\) or a terminal \((a)\), and \( \text{pos} \in \{1, 2, \ldots, m\} \). The array for storing the derivation will now have two values: one would be the production index (as in ALG1), and the position in the word of the first terminal of the sub-word to be parsed.

**Algorithm (ALG2):**

**Input:** \( G = (V_N, V_T, X_0, P) \) a context-free grammar (in Chomsky normal form), the sets \( V_Y \) (computed by the CYK Algorithm), \( w \in V_T^* \cap L(G) \);

**Output:** The derivation \( \pi \) such that \( x_0 \Rightarrow^{\pi} w \). Furthermore, it indicates the position/index of the non-terminal symbol that will be rewritten.

**Method:**

As output data structure, we shall extend the array \( d[] \), which will have elements of type *class* such as:

- one data, called *val*, will have the same meaning as in (ALG1), that is, for keeping the production’s index;
- another data, called *position*, will have the meaning discussed above, i.e., \( d[h].\text{position} = \text{pos} \) if and only if the production of index \( h \) is applied in the current propositional form.

Now, we only describe in pseudo-code the method *parse()*, as the main program which calls procedure *parse*() is similar as in Algorithm (ALG1), the only difference being that Algorithm (ALG2) contains the statement \( \text{pos} = 1 \) before the call of method *parse*().

```cpp
void parse(i, j, A) {
    if (j == l) and (m == #(A -> a)) {
        h++;
        d[h].val = m;
    }
}
```
d[h].position = pos;
}
else

if \exists k = \min\{1 \leq t \leq j - 1, \exists A \to BC, B \in V_{it}, C \in V_{i+t,j-t}\} 

k \leq j - k \{
parse(i, k, B);
pos++;
parse(i + k, j - k, C);
}
else {
pos++;
parse(i + k, j - k, C);
pos--;
parse(i, k, B);
}

Example 1.1. Let us suppose that we have as input of Algorithm (ALG1) the word w of length 20. Choosing an applicable rule of form A \to B C (using the sets V_i) implies a separation by taking each second symbol in one group. Thus, the pairs of type (i, j) from the method call parse() will be at the beginning (1, 20), then (according to Algorithm (ALG1)) will be (1, 18) and (19, 20) respectively, then (1, 16) and (17, 18), and so on. Thus, the length of the semantic stack will be 10 (due to postponed recursive calls).
Considering Algorithm (ALG2), we have at the beginning (1, 20), then (1, 2) and (3, 20) respectively. After returning from the recursive call for parameters (1, 2), we can shift the rest of the word \( w \), i.e., (3, 20). This will be shifted in (3, 4) and (5, 20), and so on. Thus the length of the semantic stack using Algorithm (ALG2) is just 2 (instead of 10 in case of Algorithm (ALG1)).

The next result describes the average time complexity of Algorithm (ALG2).

**Theorem 1.2.** The average time complexity of Algorithm (ALG2) is \( O(n \cdot \log n) \), where \( n \) is the length of the input word.

**Proof.** Let us denote by \( T(n) \) the average number of operations for processing a sequence of length \( n \) using Algorithm (ALG2). Then, the two recursive calls of the procedure \( \text{parse}(\cdot) \) need \( T(k - 1) \) and \( T(n - k) \) operations, respectively. Without loss of generality, we assume that these cases have the same probability and \( k \in \{1, 2, \ldots, n\} \), we obtain the following recurrence relation:

\[
(1) \quad T(n) = n - 1 + 1/n \cdot \sum_{k=1}^{n} (T(k - 1) + T(n - k)),
\]

where \( T(0) = T(1) = 0 \). By multiplying the relation (1) by \( n \), it results that:

\[
(2) \quad n \cdot T(n) = n(n - 1) + 2[T(0) + T(1) + \ldots + T(n - 1)]
\]

We rewrite relation (2) changing \( n \) into \( n - 1 \):

\[
(3) \quad (n - 1) \cdot T(n - 1) = (n - 1) \cdot (n - 2) + 2[T(0) + T(1) + \ldots + T(n - 2)]
\]

Combining relations (2) and (3), it follows:

\[
(4) \quad n \cdot T(n) - (n - 1) \cdot T(n - 1) = 2 \cdot (n - 1) + 2 \cdot T(n - 1)
\]

This is equivalent to the following non-linear recurrence:

\[
(5) \quad n \cdot T(n) = (n + 1) \cdot T(n - 1) + 2 \cdot (n - 1)
\]

Relation (5) can be rewritten as:

\[
(6) \quad T(n)/(n + 1) = T(n - 1)/n + 2/(n + 1) - 1/n
\]

Using the recurrence relation (6) written for each and every \( n \in \mathbb{N}, n \geq 2 \), it results:

\[
(7) \quad T(n)/(n + 1) = 2/(n + 1) + \sum_{k=1}^{n} 1/k - 1/2
\]

Using the well-known inequality

\[
2/(n + 1) - \frac{1}{2} \leq 1/(n + 1), \forall n \geq 1
\]
and relation (7) above, it follows:

$$T(n)/(n+1) \leq 2 \sum_{k=1}^{n+1} (1/k)$$

Using Riemann integrals, we can approximate the above sum from relation (8) obtaining:

$$T(n)/(n+1) \leq 2 \int_{1}^{n+1} 1/x \, dx = 2 \ln (n+1)/2 = 2\ln(n+1).$$

Thus, it results that $T(n) < 2(n + 1) \ln(n + 1)$, therefore $T(n) \in O(n \cdot \ln n)$. 

The following result shows how much is the work-space complexity.

**Theorem 1.3.** The work-space complexity of the Algorithm (ALG2) is $O(\log_2 n)$.

**Proof.** Let us denote by $h(n)$ the maximum length of semantic stack. We shall show that $h(n)$ is less than $\lceil \log_2 n \rceil$. This implies that the work-space complexity is $O(\log_2 n)$. First, we show the recursive inequality:

$$h(n) \leq \begin{cases} 1 + h \left( \left\lfloor \frac{n-1}{2} \right\rfloor \right) \\ 0 \end{cases}$$

Let us consider a sequence of length $n$ which has to be processed. The call of the method `parse()` will produce two sequences $\alpha$ and $\beta$ such that $|\alpha| \leq |\beta|$. Therefore $|\alpha| \leq \lceil (n-1)/2 \rceil$. So, in the local semantic stack will be stored the corresponding indexes of sequence $\beta$ (the minimal one and the maximal one), then the execution moves to processing sequence $\alpha$. While some subsequences of $\alpha$ are processed, the sequence $\beta$ remains in semantic stack (in a lower level corresponding to recursive call); the maximum length of the stack can be obtained when $\alpha$ is maxim, i.e., $|\alpha| = \lceil (n-1)/2 \rceil$. This corresponds to $|\beta| \leq \lceil (n-1)/2 \rceil$, hence this number is $1 + h \left( \left\lfloor (n-1)/2 \right\rfloor \right)$. In the next step, the sequence $\beta$ is extracted from the semantic stack for processing, so the maximum length of the stack being given by the inequality:

$$h(n) \leq h \left( \left\lfloor \frac{n-1}{2} + 1 \right\rfloor \right) \leq h \left( \left\lfloor \frac{n-1}{2} \right\rfloor \right) + 1.$$ 

Therefore, we have proved the above recursive inequality.

In the following, we show that $h(n) \leq \lceil \log_2 n \rceil$. For that, we shall prove the induction by $m$ showing that

$$\forall n \in \mathbb{N} \text{ such that } 2^m \leq n < 2^{m+1}, \text{ then } h(n) \leq m.$$ 

**Basis:** For $m = 0$, the inequality is obvious because $\lceil \log_2 n \rceil = m$. 

**Induction Step:** \( m = 2^{m+1} \leq 2^m \leq n < 2^{m+1} \) from the induction hypothesis, we have $h(n) \leq m$. 

**Induction Hypothesis:** Assume that $h(n) \leq m$ for $2^m \leq n < 2^{m+1}$. 

**Induction Step:** From the induction hypothesis, we have $h(n) \leq m$. 

Therefore, for $m = 2^{m+1}$, we have $h(n) \leq m$. 

**Conclusion:** By the induction hypothesis, we have $h(n) \leq m$. 

Therefore, we have proved the above recursive inequality.
Inductive Step: We suppose the above inequality true for \( m \) and let \( n \) be such that \( n \in \{2^{m+1}, 2^{m+1} + 1, \ldots, 2^{m+2} - 1\} \). Then, it follows \((n-1)/2 \leq 2^{m+1}\). According to the above recursive inequality, we have:

\[
\hat{h}(n) \leq 1 + \hat{h}\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) \leq 1 + m = \lfloor \log_2 n \rfloor.
\]

In the following, we consider an example of a context-free grammar to illustrate execution of Algorithm (ALG2).

**Example 1.2.** Let \( G = \{A, B, C, D\}, \{a, b\}, A, P \) be a context-free grammar having the productions \( P \):

1) \( A \rightarrow B \ C \)
2) \( A \rightarrow C \ D \)
3) \( B \rightarrow C \ B \)
4) \( C \rightarrow D \ D \)
5) \( D \rightarrow B \ C \)
6) \( B \rightarrow a \)
7) \( C \rightarrow b \)
8) \( D \rightarrow a \)

We propose to parse the word \( w = b \ a \ a \ b \ a \) using Algorithm (ALG2). After the execution of Algorithm (ALG2), we obtain the syntactic analysis \( \pi = (2, 7, 5, 6, 4, 8, 5, 6, 7) \) (stored in \( d[]\).val) and the positions array \((1, 1, 2, 2, 3, 4, 3, 3, 4) \) (stored in \( d[]\).pos). In other words, we obtain the derivation (the non-terminal symbol underlined below represents the one which will be rewritten):

\[
A \Rightarrow 2) \ C \ D \Rightarrow 7) \ b \ D \Rightarrow 5) \ b \ B \ C \Rightarrow 6) \ b \ a \ C \Rightarrow 4) \ b \ a \ D \ D \Rightarrow 8) \ b \ a \ D \ a \Rightarrow 5)
\]

\[
\Rightarrow 5) \ b \ a \ B \ C \ a \Rightarrow 6) \ b \ a \ a \ C \ a \Rightarrow 7) \ b \ a \ a \ b \ a
\]

It is easy to see that for the propositional form \( b \ a \ D \ D \), it was essential which non-terminal symbol had to be rewritten.

**Theorem 1.4.** (correctness of Algorithm (ALG2)) Algorithm (ALG2) is correct.

**Proof.** The indexes of productions are correct based of the correctness of Algorithm (ALG1) ([AhU72, HoU78]). What we still have to prove is that the position of each non-terminal symbol is correct. We can use induction by the length of derivation, denoted by \( h \). In other words, we shall prove that for a derivation with \( h \) steps, there will be applied a production which has in the left hand side the non-terminal symbol of index \( d[]\).pos from the current propositional form.

**Basis:** \( h = 0 \). Therefore we have a derivation in one step (of the kind \( x_0 \Rightarrow a_0 \)), and in method parse() will be used the main part of the statement "if \((j = 1)\) and \((m = \#(A \rightarrow a_0))\)."

Therefore, \( d[]\).pos = 1, that is, the non-terminal symbol of the first position will be modified;
**Inductive Step:** We suppose the assertion holds for all 0, 1, \ldots, h - 1, and we prove for h. We consider an arbitrary derivation:

**Case I:** The control of method parse() is on the main part of statement “if (k ≤ j - k)”. Therefore, the left derivation subtree will be chosen. Thus, the non-terminal symbol B becomes the current one (so we do not have to change the global variable pos, obtaining d[B].pos is the same as d[A].pos). The non-terminal symbol C will become the current one on the next recursive call. So, before that recursive call, we have to increment the variable d[i].pos (obtaining d[C].pos = d[A].pos + 1).

**Case II:** The control of method parse() is on the else branch of the statement “if (k ≤ j - k)”. Therefore, the right derivation subtree will be chosen. Thus, the non-terminal symbol C becomes the current one, so we have to increment the global variable pos. Obviously, before the next recursive call (for which the root of the derivation subtree is B), we must decrement the global variable pos.

Thus, we keep in the array d[i], (i.e., in the item position), the position of each non-terminal symbol which is the leftmost symbol of the current production.

### 3 Experimental Results

We implemented (ALG1) and (ALG2) algorithms in Java programming language. We ran these two implementations on the same machine, a Pentium computer system, with 3.2GHz using 8GB of main memory. Table 1 describes the output results obtained for some context-free grammars and words.

The below languages are described in the Introduction. The corresponding Chomsky normal form grammars are the following:

1. For the palindrome language not including the empty word: S → AB, B → SA, S → CD, D → SC, S → AA, S → CC, A → 0, and C → 1;
2. For \( \{0^n1^n \cup 0^n1^{2n} \mid n \geq 1\} \) language: S → CD, S → CE, E → DD, S → CF, F → AD, S → CG, G → BE, A → CH, H → AD, A → CD, B → CI, I → BE, B → CE, C → 0, and D → 1.
3. For \( \{a^n b^n c^{m} d^{m} \cup a^n b^{m} c^{n} d^n \mid n \geq 1, m \geq 1\} \) language:

\[
S \rightarrow AS_7, S_7 \rightarrow S_2D, S \rightarrow AS_8, S_8 \rightarrow S_6D, S_5 \rightarrow AS_7, S_5 \rightarrow AS_8, S_5 \rightarrow BS_9, S_9 \rightarrow S_6C, S_6 \rightarrow BC, S \rightarrow S_1S_2, S_1 \rightarrow AS_3, S_1 \rightarrow S_1B, S_1 \rightarrow AB, S_2 \rightarrow CS_4, S_4 \rightarrow S_2D, S_4 \rightarrow CD, A \rightarrow a, B \rightarrow b, C \rightarrow c, \text{and } D \rightarrow d.
\]

Note that all three context-free languages are not deterministic context-free languages. This implies that these three languages cannot be parsed by specialized automatic tools [Reg09] which mainly have the ability to parse LR(1) languages (that is, deterministic context-free languages). In fact, the third language is inherent ambiguous. This means all its generative grammars are ambiguous.
<table>
<thead>
<tr>
<th>Grammar</th>
<th>Length of word</th>
<th>The execution time for the construction of $V_{ij}$</th>
<th>The execution time for parsing the word</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Traditional program (miliseconds)</td>
<td>Our program (miliseconds)</td>
<td>Traditional program (miliseconds)</td>
</tr>
<tr>
<td>Palindrome</td>
<td>48</td>
<td>36</td>
<td>35</td>
</tr>
<tr>
<td>Palindrome</td>
<td>96</td>
<td>453</td>
<td>452</td>
</tr>
<tr>
<td>Palindrome</td>
<td>192</td>
<td>7238</td>
<td>7067</td>
</tr>
<tr>
<td>Palindrome</td>
<td>384</td>
<td>108412</td>
<td>108529</td>
</tr>
<tr>
<td>$0^i 1^n \cup 0^n 1^{2n}$</td>
<td>48</td>
<td>70</td>
<td>68</td>
</tr>
<tr>
<td>$0^n 1^n \cup 0^n 1^{2n}$</td>
<td>96</td>
<td>842</td>
<td>858</td>
</tr>
<tr>
<td>$0^n 1^n \cup 0^n 1^{2n}$</td>
<td>192</td>
<td>44168</td>
<td>43020</td>
</tr>
<tr>
<td>$0^n 1^n \cup 0^n 1^{2n}$</td>
<td>384</td>
<td>205359</td>
<td>205062</td>
</tr>
<tr>
<td>$a^n b^m c^d m \cup a^n b^m c^d n$</td>
<td>66</td>
<td>186</td>
<td>170</td>
</tr>
<tr>
<td>$a^n b^m c^d m \cup a^n b^m c^d n$</td>
<td>132</td>
<td>3682</td>
<td>3651</td>
</tr>
<tr>
<td>$a^n b^m c^d m \cup a^n b^m c^d n$</td>
<td>324</td>
<td>133522</td>
<td>128481</td>
</tr>
<tr>
<td>$a^n b^m c^d m \cup a^n b^m c^d n$</td>
<td>452</td>
<td>482368</td>
<td>482321</td>
</tr>
</tbody>
</table>

Table 1. The results about programs implementing the traditional algorithm and our algorithm

Compared to theoretical results, the execution times of Java implementations of the traditional and the efficient parsing words as indicated in Table 1 do not differ significantly, that is, less than 5%. In addition, we observe that the construction of $V_{ij}$ sets needs a cubic time, while the execution time for parsing the word has a sub-linear behavior.

4 Conclusions

The paper presents in an informal way an improved algorithm for parsing a context free grammar (represented in Chomsky normal form) for which the work-space complexity is smaller than some previously published algorithms. As future work, the paper may be extended to designing a parallel algorithm similar to the one presented in [RaK96, IPS91].

References


