Bottom-Up Dynamic Programming Approach in Cocke-Younger-Kasami Algorithm for Efficient English Language Grammar Checker

Genta Indra Winata (13511094)¹
Computer Science/Informatics
School of Electrical Engineering and Informatics
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
¹13511094@std.stei.itb.ac.id

Abstract—Parsing is a fundamental process in Computer Science especially in natural language processing. It is the process of analyzing string of symbols according to the rules of a formal grammar which is context-free grammar. There are multiple ways to parse strings and one of them is Cocke-Younger-Kasami Algorithm. English Language has a rich and complex grammar with varies of tenses and different ways of implementation between the pronouns, quantifiers, adjectives, verbs, adverbs, article, etc. The complexity of English Language Grammar creates an issue where people may easily get grammar error when writing their papers. Therefore, this issue need to be minimized by using grammar checking algorithm. One of the option is Cocke-Younger-Kasami Algorithm (CYK). CYK Algorithm employs bottom-up parsing and dynamic programming. CYK operates on context-free grammars in given Chomsky normal form (CNF). The algorithm has a high efficiency in parsing with the most efficient parsing algorithm in terms of worst case running time O(n³|G|) where n is the length of the parsed string and |G| is the size of the CNF grammar.

Index Terms— parsing, grammar, algorithm, dynamic programming

I. INTRODUCTION

Recently, natural language processing has been a popular topic in Computer Science. One of the application is text parsing. In this paper, the writer will discuss about dynamic programming approach to one of the application in the Context-free Grammar (CFG) parsing. CYK is one of the example. In order, to operate a standard CYK algorithm, the grammar in the production rules must be in Chomsky Normal Form (CNF). The implementation of CYK applies production rules and comprises two type symbols such as nonterminal and terminal symbols.

Today, English Language is widely used in many countries and many essays and papers are written in English Language. But recently a problematic issue overcomes, related to the wrong grammar usage. The correctness of the sentences is very essential to many academic and business purposes. It is hard to determine all words by humans’ eyes and certainly, human makes mistake(s) in grammar checking. Therefore, automated error checking is needed to help human in detecting the grammar errors.

English language has a rich and diverse of grammar. The grammar usage used many parameters to determine whether a sentence is a valid grammar or not. The parameters are pronouns, determiners and quantifiers, possessives, adjectives, adverbials, verbs, nouns, clause, phrases and sentence. The permutation of two or more of above parameters creates grammar rules. The grammar rules will be used as a base to resolve the validity of a sentence. The validity of a sentence is very important in One of the approach to determine the validity of sentence of a grammar is by using CYK algorithm. This algorithm using dynamic programming bottom-up approach by collects all nonterminal and terminals symbols from the rules given. This algorithm is highly efficient and has the most efficient worst-case asymptotic complexity with O(n³|G|) where n is the length of the parsed string and |G| is the size of the CNF grammar.

II. FUNDAMENTAL THEORIES

2.1 Dynamic Programming

Dynamic Programming solves problems by combining the solutions to subproblems.¹ When developing this algorithm, we may follow four steps:¹

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution, typically in a bottom-up fashion
4. Construct an optimal solution from computed information.

2.2 Bottom-up Dynamic Programming Approach

Bottom-up technique uses table in the computation of dynamic programming algorithm. This is actually the ‘true form’ of dynamic programming as it was originally known as ‘tabular method’². There are steps to build this approach:²

1. Determine the required set of parameters that uniquely describe the problem (the state).
2. If there are N parameters required to represent the state, prepare N dimensional Dynamic Programming Table.

3. Now, with the base-case states in the Dynamic Programming table already filled, determine the state that can be filled next. Repeat this process until the Dynamic Programming table is complete.

This technique usually can be done using loops rather than recursive method. For instance, we are using bottom-up approach in solving fibonacci problem:

```plaintext
1. Fibonacci(n)
2. Declare a table of integer fib[n]
3. Let fib[0] and fib[1] be 1
4. For each i from 2 to n do:
5.     Let fib[i] be fib[i-1] + fib[i-2].
6. End for
7. return fib[n].
```

The above algorithm shows the implementation of getting fibonacci number by bottom-up approach. At line 2, we firstly have to declare a table of integer as a place for putting all computations that will be done at line 4 to 6. At line 3, we assign the zero's and first index with 1 as the base. Later in line 4-6, every value of fib[i] is assigned with the addition of the two consecutive previous elements. Finally at line 7, the function returns the nth fibonacci number.

For the illustration, we can take a look the figure below:

![Dynamic Programming Table Diagram]

Now, with the base-case states in the Dynamic Programming table already filled, determine the state that can be filled next. Repeat this process until the Dynamic Programming table is complete.

This technique usually can be done using loops rather than recursive method. For instance, we are using bottom-up approach in solving fibonacci problem:

```
1. Fibonacci(n)
2. Declare a table of integer fib[n]
3. Let fib[0] and fib[1] be 1
4. For each i from 2 to n do:
5.     Let fib[i] be fib[i-1] + fib[i-2].
6. End for
7. return fib[n].
```

The above algorithm shows the implementation of getting fibonacci number by bottom-up approach. At line 2, we firstly have to declare a table of integer as a place for putting all computations that will be done at line 4 to 6. At line 3, we assign the zero's and first index with 1 as the base. Later in line 4-6, every value of fib[i] is assigned with the addition of the two consecutive previous elements. Finally at line 7, the function returns the nth fibonacci number.

For the illustration, we can take a look the figure below:

![Dynamic Programming Table Diagram]

### Formal Grammar

A formal grammar comprises a set of production rules for strings in a formal language. The rules shows how to form a string from the valid character of the language according to the language's syntax.

A grammar is a tuple G = (V, T, S, P) where
- V is a finite, non-empty set of symbols called variables (or not-terminals or syntactic categories)
- T is an alphabet of symbols called terminals
- S ∈ V is the start symbol of the grammar
- P is a finite set of production α → β where α ∈ (V ∪ T)⁺ and β ∈ (V ∪ T)*

For example:

```
V = {Sentence, Subject, Verb, Object}
T = {I, You}
S = {Sentence}
P = {Sentence -> Subject Verb Object,
Verb->eat,
Object->orange}
```

A valid sentence for above example is “I eat orange”. Sentence “I eat orange” is valid because it obeys the production rules.

```
Sentence -> Subject Verb Object
Subject -> I
Verb -> eat
Object -> orange
```

### Context-free Grammar

Context-free Grammar (CFG) is a formal grammar with a set of recursive rewriting rules or productions used to generate patterns of strings. CFG has production rule in the form of

```
V → w
```

where V is a single nonterminal symbol and w is a string of terminal/nonterminal (can be empty).

A CFG consists of several components such as terminal symbols, nonterminal symbols, productions and a start symbol.

a. Terminal Symbols
   Symbols which are the characters of the alphabet appear in the string generated by the grammar.

b. Nonterminal Symbols
   Symbols which are placeholders for pattern of terminal symbols and can be generated by nonterminal symbol.

c. Productions
   Rules for replacing nonterminal symbols or terminal symbols.

d. Start Symbol
   A special nonterminal symbol that appears in the initial string generated by the grammar.

For example:
```
S → aSa, 
S → bSb, 
S → ε
```
A typical derivation in this grammar is:
\[ S \rightarrow aSa \rightarrow aaSaa \rightarrow aabSbaa \rightarrow aabbaa \]

2.5 Cocke-Young-Kasami Algorithm

Cocke-Young-Kasami (CYK) is a parsing algorithm for context-free grammars. This algorithm’s name came from three inventors, John Cocke, Daniel Younger and Tadao Kasami and it employs bottom-up dynamic programming approach. Here are the pseudo code of CYK algorithm:

1. Let the input be a string \( S \) consisting of \( n \) characters: \( a_1 \ldots a_n \).
2. Let the grammar contain \( r \) nonterminal symbols \( R_1 \ldots R_r \).
3. This grammar contains the subset \( R_S \) which is the set of start symbols.
4. Let \( P[n,n,r] \) be an array of booleans. Initialize all elements of \( P \) to false.
5. For each \( i = 1 \) to \( n \)
6. For each unit production \( R_j \rightarrow a_i \)
7. Set \( P[i,i,j] = \text{true} \)
8. For each \( i = 2 \) to \( n \) -- Length of span
9. For each \( j = 1 \) to \( n-i+1 \) -- Start of span
10. For each \( k = 1 \) to \( i-1 \) -- Partition of span
11. For each production \( R_A \rightarrow R_B R_C \)
12. If \( P[j,k,B] \) and \( P[j+k-i,k,C] \) then
13. Set \( P[j,i,A] = \text{true} \)
14. If any of \( P[1,n,x] \) is true (\( x \) is iterated over the set \( s \), where \( s \) are all the indices for \( R_S \)) then
15. \( S \) is member of language
16. Else
17. \( S \) is not member of language

This algorithm considers every possible subsequence of the sequence of words and sets \( P[i,j,k] \) to be true starting from 1 of length \( j \) can be generated from \( R_k \). It has considered subsequences of length 1 and goes to greater lengths. It considers every possible partition of the subsequence of two parts and check if there is a production \( P \rightarrow QR \). If so, it records \( P \) as matching the whole sequence. Once the process is completed, the sentence is recognized by the grammar.

In the CYK algorithm, the production rules are saved in the form Chomsky Normal Form (CNF). There are three forms of CNF:

- \( A \rightarrow BC \) or \( A \rightarrow \epsilon \)
- \( A \rightarrow \alpha \)
- \( S \rightarrow \epsilon \)

where \( A, B \) and \( C \) are nonterminal symbols, \( \alpha \) is a terminal symbol, \( S \) is the start symbol, and \( \epsilon \) is the empty string.

This is one of the example grammar:

\[
\begin{align*}
S & \rightarrow NP VP \\
VP & \rightarrow VP PP \\
P & \rightarrow V NP \\
V & \rightarrow drinks \\
N & \rightarrow juice \\
\text{Det} & \rightarrow a \\
\text{Det} & \rightarrow he \\
\end{align*}
\]

From above grammar, we can form a table where in each row has the increment of number of words:
In the production rules above are written in Chomsky Normal Form (CNF). From above rules we can have many word combination that can be formed into sentences by combining subjects, verbs, nouns, adjectives, pronouns and also considering the time which the action takes place. “S” will be the start state and the state points to the next state based on the rules. For example, state S has a production rule,

\[
S \rightarrow S \text{ PRESENT\_CONJUNCTION1}
\]

S will recursively go back to state S and afterwards go to state PRESENT\_CONJUNCTION1 for the next string on the right of the recursive process. Then in PRESENT\_CONJUNCTION1 state will parse either C\_AND S or C\_AND PRESENTS\_COM1 or C\_OR S or C\_OR PRESENT\_COM1 or C\_BUT S or C\_BUT PRESENT\_COM1. From these options, there are varies of sentence combinations. For instances,

**I LOVE ORANGE AND EAT ORANGE**

“**I LOVE ORANGE AND EAT ORANGE**” comes from several production rules such as

\[
S \rightarrow S \text{ PRESENT\_CONJUNCTION1} \rightarrow C\_AND \text{ PRESENTS\_COM1}
\]

**IV. IMPLEMENTATION COCKE-YOUNGER-KASAMI ALGORITHM IN ENGLISH LANGUAGE GRAMMAR CHECKER**

4.1 Cocke-Younger-Kasami Algorithm Implementation

From the grammar stated before, we can implement them by using Cocke-Younger-Kasami Algorithm. We firstly begin with state S as the start state. Then, we continue to initialize all the table’s elements into false and determine nonterminal symbols. Then, all the nonterminal symbols are included in the grammar and for each unit production with length 1 is set to true and continue for length 2, 3 and so on. After we achieve the max length, we can determine the validity of the sentence. For this case, the nonterminals symbols are noun, pronouns, verbs, to be, etc.

Here are the interface of the application that has been built by the writer that used to check strings’ grammar:

**Picture 3** – Application Graphic User Interface with Valid Sentence

Picture 3 shows the input sentence and the result after the validation process. Sentence “**TODAY I LOVE ORANGE AND EAT ORANGE**” is valid. The above test case spends 79 milliseconds. Here are the CYK Table from Picture 3 test case for a better illustration.
This is the example with invalid sentence by using sentence “TODAY IS BIG ORANGE”.

Here are the CYK Table from Picture 4 test case for a better illustration.

The above table built from sentence “TODAY IS A BIG ORANGE” and stopped at the third row when trying to concatenate 2 strings “TODAY” and “IS A BIG ORANGE”, therefore the sentence is invalid. There is no combination word “today” and “is a big orange” based on the production rules in the grammar that has been stated before.

4.2 Algorithm Analysis and Testing

This algorithm has a polynomial time and has the most efficient worst-case asymptotic complexity with $O(n^3|G|)$. Here are the statistics taken by experiment with 10 samples:

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Sentence</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JAMES IS HAPPY</td>
<td>20 ms</td>
</tr>
<tr>
<td>2</td>
<td>HE IS SAD</td>
<td>8   ms</td>
</tr>
<tr>
<td>3</td>
<td>TODAY I LOVE ORANGE AND EAT ORANGE</td>
<td>35 ms</td>
</tr>
<tr>
<td>4</td>
<td>HABIBIE IS COOL</td>
<td>39 ms</td>
</tr>
<tr>
<td>5</td>
<td>WE TASTE SYRUP AND DRINK JUICE</td>
<td>29 ms</td>
</tr>
<tr>
<td>6</td>
<td>SONNY LOVES MOUSE</td>
<td>14 ms</td>
</tr>
<tr>
<td>7</td>
<td>JAMES WAS GOOD</td>
<td>8   ms</td>
</tr>
<tr>
<td>8</td>
<td>TODAY CH IS GOOD</td>
<td>20 ms</td>
</tr>
<tr>
<td>9</td>
<td>THEY EAT CHICKEN AND DRINK JUICE</td>
<td>31 ms</td>
</tr>
<tr>
<td>10</td>
<td>THEY EAT CHICKEN</td>
<td>8   ms</td>
</tr>
</tbody>
</table>

Table 1 – Experiment Result

Table 1 shows a progressive increment as the words’ length increase. For a more tangible result, there is a bar chart below:

Picture 6 shows the differentiation of time spent by number of words. The increment of time between the number of words is doubled. The implementation of Cocke-Younger-Kasami Algorithm in the English grammar checker shows a good sign that this algorithm forms an excellent result. The overall average of time spent is 21.0 milliseconds from 5.1 characters. Moreover, we can conclude that this algorithm has a pattern to be a polynomial time algorithm.
<table>
<thead>
<tr>
<th>Length of Word</th>
<th>Average Time (ms)</th>
<th>Checking if the O(n³), with length of word 3 character and avg time 16 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>21</td>
</tr>
<tr>
<td>6</td>
<td>30</td>
<td>32</td>
</tr>
<tr>
<td>7</td>
<td>35</td>
<td>37</td>
</tr>
</tbody>
</table>

Table 2 – Approximation of Algorithm’s Worst Time

The above table emphasizes that the algorithm is polynomial time algorithm with the approximation with O(n³) algorithm. Furthermore, the algorithm will be an option in English Language Grammar Checking.

V. CONCLUSION

Cocke-Younger-Kasami Algorithm is an efficient algorithm in language grammar parsing with worst time complexity O(n³|G|) where n is the length of the parsed string and |G| is the size of the CNF grammar. It is also known well as the algorithm with the best in worst-case asymptotic complexity. The subjects, verbs, nouns, adjectives, pronoun, article, and time reference are known as the nonterminals and value of each component becomes the terminals of the grammar.

The implementation of Cocke-Younger-Kasami Algorithm is very useful in checking English Language Grammar. This algorithm would able to check whether a sentence obey the grammar rules or not by building the production rules within the grammar.

VII. ACKNOWLEDGMENT

First of all, I would thank to God for His perfect providence while preparing, writing, and editing the paper. Then, I deeply express my thanks to my advisors, Dr. Ir. Rinaldi Munir and Dr. Masayu Leylia Khodra, whose help, advice, and support. I also thank to my parents and friends for the prayers and valuable courage given to me. Finally, I would thank to Institut Teknologi Bandung for the hospitality and support during the completion of the paper.

REFERENCES


PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 20 Desember 2013

Genta Indra Winata