Dynamic Programming Algorithm To Determine How Context-Free Grammar Can Generate A String

Samuel Simon

Program Studi Teknik Informatika, Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung
Alamat: Jalan Ganeca No. 10 Bandung
Email: if16032@students.if.itb.ac.id

Abstract

In formal language theory, a context-free grammar (CFG) is a grammar in which every production rule is of the form *V* → *w*, where *V* is a single nonterminal symbol and *w* is a string of terminals and/or nonterminals (possibly empty). The term "context-free" expresses the fact that nonterminals can be rewritten without regard to the context in which they occur. Context-free grammars play a central role in the description and design of programming languages and compilers. They are also used for analyzing the syntax of natural languages. For that reason, algorithms to determine whether a string can be generated by a given context-free grammar and, if so, how it can be generated are important. Every that kind algorithm is in the class of polynomial time. The best known such algorithm is called the Cocke-Younger-Kasami (CYK) algorithm, and uses dynamic programming to build a collection of derivations of substrings from grammar symbols.

Keywords: dynamic programming, CYK, context-free grammar.

1. INTRODUCTION

A Context-Free Grammar (CFG) is a set of recursive rewriting rules (or production) used to generate patterns of strings. A CFG consists of the following components:

- A set of terminal symbols, which are the characters of the alphabet that appear in the strings generated by the grammar.
- A set of non-terminal symbols, which are placeholders for patterns of terminal symbols that can be generated by the non-terminal symbols.
- A set of production, which are rules for replacing (or rewriting) non-terminal symbols (on the left side of the production) in a string with other non-terminal or terminal symbols (on the right side of the production).
- A start symbol, which is a special non-terminal symbol that appears in the initial string generated by the grammar.

The formal definition of CFG *G* is a 4-turple: G = (V, ∑, R, S), where V is nonterminal, ∑ is a terminal, R is a set of rules or productions of the grammar. S is the start variable, used to represent the whole sentence (or program).

To generate a string of terminal symbols from a CFG, we:

- Begin with a string consisting of the start symbol;
- Apply one of the productions with the start symbol on the left hand side, replacing the start symbol with the right hand side of the production;
- Repeat the process of selecting non-terminal symbols in the string, and replacing them with the right hand side of some corresponding production, until all non-terminal have been replaced by terminal symbols.

A CFG provides a simple and precise mechanism for describing the methods by which phrases in some natural language are built from smaller blocks, capturing the block structure of sentences in a natural way. The block structure aspect that CFG capture is so fundamental to grammar that the terms syntax and grammar are often identified with CFG rules, especially in computer science. Formal constraints not captured by the grammar are considered to be part of the semantic of the language.

Some examples of CFG:

- Example 1

  S → a
  S → aS
  S → bS

  The terminals here are *a* and *b*, while the only non-terminal is S. The language described is all nonempty strings of *a*s and *b*s that end in *a*. This grammar is regular: no rule has more than one nonterminal in its right-hand side, and each of these nonterminals is at hte same end of the right-hand side.

  Every regular grammar corresponds directly to a nondeteministic finite automaton, so we know that this is a regular language.

It is common to list all right-hand sides for the same left-hand side on the same line, using | to seperate them like this:

$S \rightarrow a \mid aS \mid bS$

Technically, this is the same grammar as above.

- Example 2

  In a CFG, we can pair up characters the way we do with brackets. The simplest example:
  $S \rightarrow aSb$
  $S \rightarrow ab$

  This grammar generates the language **$\{a^n b^n \; ; \; n \geq 1\}$**, which is not regular.

  The special character ε stands for the empty string. By changing the above grammar to: $S \rightarrow aSb \mid \varepsilon$, we obtain a grammar generating the language **$\{a^n b^n \; ; \; n \geq 0\}$** instead. This differs only in that it contains the empty string while the original grammar did not.

2. DYNAMIC PROGRAMMING ALGORITHM

In mathematics and computer science, dynamic programming is a method of solving problems exhibiting the properties of overlapping subproblems and optimal substructure that takes much less time than naive methods.

In general, we can solve a problem with optimal substructure using a three-step process:
1. Break the problem into smaller problems.
2. Solve these problems optimally using this three-step process recursively.
3. Use these optimal solutions to construct an optimal solution for the original problem.

The subproblems are, themselves, solved by dividing them into sub-subproblems, and so on, until we reach some simple case that is solvable in constant time. To say that a problem has overlapping subproblems is to say that the same subproblems are used to solve many different larger problems. For example, in the Fibonacci sequence, $F_3 = F_1 + F_2$ and $F_4 = F_2 + F_3$, computing each number involves computing F_2. Because both F_3 and F_4 are needed to compute F_5,a naive approach to computing F_5 may end up computing F_2 twice or more. This applies whenever overlapping subproblems are present: a naive approach may waste time recomputing optimal solutions to subproblems it has already solved.

In order to avoid this, we instead save the solutions to problems we have already solved. Then, if we need to solve the same problem later, we can retrieve and reuse our already-computed solution. This approach is called memoizaton (not memorization). If we are sure we would not need a particular solution anymore, we can throw it away to save space. In some cases, we can even compute the solutions to subproblems we know that we will need in advance.

3. DYNAMIC PROGRAMMING IN CFG:

3.1. Cocke-Younger-Kasami

The best know algorithm to determine whether a string can be generated by a given CFG and how it can be generated is Cocke-Younger-Kasami (CYK). This algorithm is an example of dynamic programming and known as parsing the string.

The standard version of CYK recognizes languages defined by CFG written in Chomsky normal form (CNF). Since any CFG can be converted to CNF without too much difficulty, CYK can be used to recognize any context-free language. It is also possible to extend the CYK algorithm to handle some CFG which are not written in CNF; this may be done to improve performance, although at the cost of making the algorithm harder to understand.

The worst case asymptotic time complexity of CYK is $\Theta(n^3)$, where n is the length of the parsed string. This makes it one of the most efficient (in those terms) algorithms for recognizing any context-free language. However, there are other algorithms that will perform better for certain subsets of the context-free languages.

3.2. The Algorithm of CYK

The CYK algorithm is a bottom up algorithm and is important theoretically, since it can be used to constructively prove that the membership problem for context-free languages is decidable.

The CYK algorithm for the membership problem is as follows:

- Let the input string be a sequence of n letters $a_1 \ldots a_n$.
- Let the grammar contain r terminal and nonterminal symbols $R_1 \ldots R_r$, and let R_1 be the start symbol.
- Let P[n, n, r] be an array of booleans. Initialize all elements of P to false.
- For each i = 1 to n
  For each unit production $R_j \rightarrow a_i$, set P[i,1,j] = true.
- For each i = 2 to n (length of span)
  For each j = 1 to n-i+1 (start of span)

```
        For each k = 1 to i-1 (partition of span)
            For each production $R_A \rightarrow R_B R_C$
                If P[j, k, B] and P[j+k, i-k, C],
                then set P[j, i, A] = true
▪ If P[1,n,1] is true
    Then string is member of language
    Else string is not member of language
```

It is simple to extend the above algorithm to not only determine if a sentence is in a language, but to also construct a parse tree, by storing parse tree nodes as elements of the array, instead of booleans. Since the grammars being recognized can be ambiguous, it is necessary to store a list of nodes (unless one wishes to only pick one possible parse tree); the end result is then a forest of possible parse trees.

3.3 Sample Usage of CYK

Given CFG G = (V, ∑, R, S) where

V = {S} ∪ ∑ ∑ = { (,) }

R: $S \rightarrow SS$

$S \rightarrow (S)$

$S \rightarrow \varepsilon$

G = (V, ∑, R, S) in Chomsky Normal Form (CNF) becomes

V = {S} ∪ ∑ ∑ = { (,) }

R: $S \rightarrow SS$

$S \rightarrow (S_1$

$S_1 \rightarrow S)$

$S \rightarrow ( )$

We try to detemine whether $w = x_1\ x_2\ \dots\ x_n,\ n \geq 2$ be generated by G?

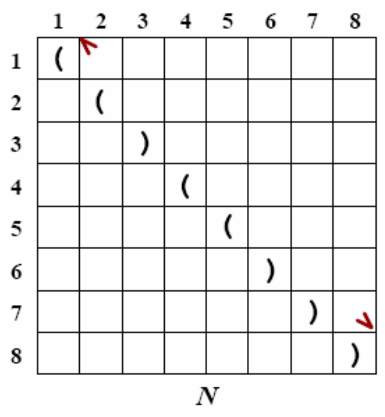
Consider the string w = (( ) (( )))

R: $S \rightarrow SS$ $N_{i,i+s}$

$S \rightarrow (S_1$

$S_1 \rightarrow S)$ $= \{A \in V : A \rightarrow^* x_i \dots x_{i+s}\}$

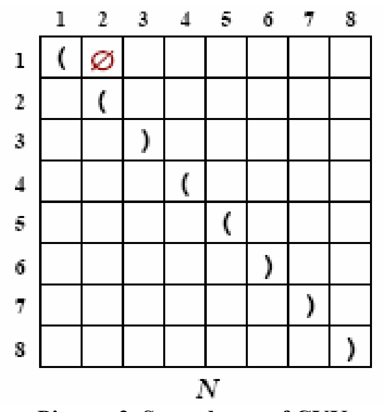
$S \rightarrow ( )$



	1	2	3	4	5	6	7	8
1	(	↖						
2		(						
3			)					
4				(				
5					(			
6						)		
7							)	↘
8								)

N

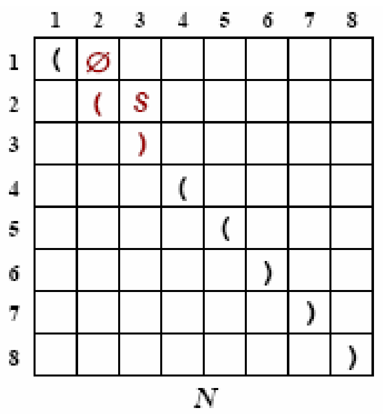
Picture 1. First step of CYK



	1	2	3	4	5	6	7	8
1	(	∅						
2		(						
3			)					
4				(				
5					(			
6						)		
7							)	
8								)

N

Picture 2. Second step of CYK



	1	2	3	4	5	6	7	8
1	(	∅						
2		(	S					
3			)					
4				(				
5					(			
6						)		
7							)	
8								)

N

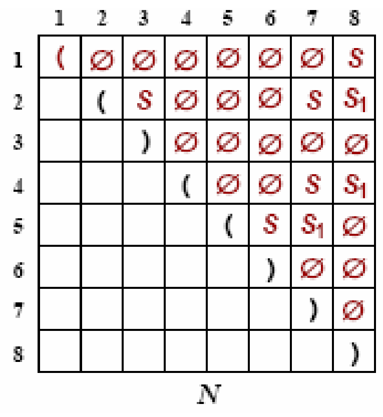
Picture 3. Third step of CYK

R: $S \rightarrow SS$

$S \rightarrow (S_1$

$S_1 \rightarrow S)$

$S \rightarrow ( )$



	1	2	3	4	5	6	7	8
1	(	∅	∅	∅	∅	∅	∅	S
2		(	S	∅	∅	∅	S	S_1
3			)	∅	∅	∅	∅	∅
4				(	∅	∅	S	S_1
5					(	S	S_1	∅
6						)	∅	∅
7							)	∅
8								)

N

Picture 4. Last step of CYK

R: $S \rightarrow SS$

$S \rightarrow (S_1$

$S_1 \rightarrow S)$

$S \rightarrow ( )$

4. CONCLUSION

Nowadays, CFG play many important roles in computer science field. CFG is used in starting from compiler design, natural language processing, to analyzing parantheses in modern computer languages. For that reason, we need powerful algorithm to compare the input string and the grammar.

CYK is a non-directional bottom-up parser for CFG. This algorithm is known as one of the best parser algorithm for CFG. If used with CNF, it is very efficient with time complexity is $O(n^3)$. The transformation into CNF can be undone after parsing, i.e., we still have a parser for arbitrary CFGs (as long as ε is not in language).

REFERENCE

[1] J. K. Baker. "Trainable Grammars for Speech Recognition". In J. J. Wolf and D. H. Klatt, editors, *Speech Communication Papers for the 97th Meeting of the Acoustical Society of America*, pages 547-550, 1979.

[2] E. Charniak. "Statistical Language Learning". MIT Press, 1993.

[3] Chomsky, Noam. 1956. "Three Models for The Description of Language". Information Theory, IEEE Transaction 2.