

Other Uses for the Linear Emptiness Test

The same data structure and accounting trick that we used in Section 7.4.3 to test whether a variable is generating can be used to make some of the other tests of Section 7.1 linear-time. Two important examples are:

1. Which symbols are reachable?
2. Which symbols are nullable?

1. Work done for that production: discovering the count is 0, finding which variable, say A , is at the head, checking whether it is already known to be generating, and putting it on the queue if not. All these steps are $O(1)$ for each production, and so at most $O(n)$ work of this type is done in total.
2. Work done when visiting the positions of the production bodies that have the head variable A . This work is proportional to the number of positions with A . Therefore, the aggregate amount of work done processing all generating symbols is proportional to the sum of the lengths of the production bodies, and that is $O(n)$.

We conclude that the total work done by this algorithm is $O(n)$.

7.4.4 Testing Membership in a CFL

We can also decide membership of a string w in a CFL L . There are several inefficient ways to make the test; they take time that is exponential in $|w|$, assuming a grammar or PDA for the language L is given and its size is treated as a constant, independent of w . For instance, start by converting whatever representation of L we are given into a CNF grammar for L . As the parse trees of a Chomsky-Normal-Form grammar are binary trees, if w is of length n then there will be exactly $2n - 1$ nodes labeled by variables in the tree (that result has an easy, inductive proof, which we leave to you). The number of possible trees and node-labelings is thus “only” exponential in n , so in principle we can list them all and check to see if any of them yields w .

There is a much more efficient technique based on the idea of “dynamic programming,” which may also be known to you as a “table-filling algorithm” or “tabulation.” This algorithm, known as the *CYK Algorithm*,³ starts with a CNF grammar $G = (V, T, P, S)$ for a language L . The input to the algorithm is a string $w = a_1 a_2 \cdots a_n$ in T^* . In $O(n^3)$ time, the algorithm constructs a table

³It is named after three people, each of whom independently discovered essentially the same idea: J. Cocke, D. Younger, and T. Kasami.

that tells whether w is in L . Note that when computing this running time, the grammar itself is considered fixed, and its size contributes only a constant factor to the running time, which is measured in terms of the length of the string w whose membership in L is being tested.

In the CYK algorithm, we construct a triangular table, as suggested in Fig. 7.12. The horizontal axis corresponds to the positions of the string $w = a_1a_2 \cdots a_n$, which we have supposed has length 5. The table entry X_{ij} is the set of variables A such that $A \xRightarrow{*} a_i a_{i+1} \cdots a_j$. Note in particular, that we are interested in whether S is in the set X_{1n} , because that is the same as saying $S \xRightarrow{*} w$, i.e., w is in L .

X_{15}					
X_{14}	X_{25}				
X_{13}	X_{24}	X_{35}			
X_{12}	X_{23}	X_{34}	X_{45}		
X_{11}	X_{22}	X_{33}	X_{44}	X_{55}	
	a_1	a_2	a_3	a_4	a_5

Figure 7.12: The table constructed by the CYK algorithm

To fill the table, we work row-by-row, upwards. Notice that each row corresponds to one length of substrings; the bottom row is for strings of length 1, the second-from-bottom row for strings of length 2, and so on, until the top row corresponds to the one substring of length n , which is w itself. It takes $O(n)$ time to compute any one entry of the table, by a method we shall discuss next. Since there are $n(n+1)/2$ table entries, the whole table-construction process takes $O(n^3)$ time. Here is the algorithm for computing the X_{ij} 's:

BASIS: We compute the first row as follows. Since the string beginning and ending at position i is just the terminal a_i , and the grammar is in CNF, the only way to derive the string a_i is to use a production of the form $A \rightarrow a_i$. Thus, X_{ii} is the set of variables A such that $A \rightarrow a_i$ is a production of G .

INDUCTION: Suppose we want to compute X_{ij} , which is in row $j-i+1$, and we have computed all the X 's in the rows below. That is, we know about all strings shorter than $a_i a_{i+1} \cdots a_j$, and in particular we know about all proper prefixes and proper suffixes of that string. As $j-i > 0$ may be assumed (since the case $i=j$ is the basis), we know that any derivation $A \xRightarrow{*} a_i a_{i+1} \cdots a_j$ must

start out with some step $A \Rightarrow BC$. Then, B derives some prefix of $a_i a_{i+1} \cdots a_j$, say $B \xRightarrow{*} a_i a_{i+1} \cdots a_k$, for some $k < j$. Also, C must then derive the remainder of $a_i a_{i+1} \cdots a_j$, that is, $C \xRightarrow{*} a_{k+1} a_{k+2} \cdots a_j$.

We conclude that in order for A to be in X_{ij} , we must find variables B and C , and integer k such that:

1. $i \leq k < j$.
2. B is in X_{ik} .
3. C is in $X_{k+1,j}$.
4. $A \rightarrow BC$ is a production of G .

Finding such variables A requires us to compare at most n pairs of previously computed sets: $(X_{ii}, X_{i+1,j})$, $(X_{i,i+1}, X_{i+2,j})$, and so on, until $(X_{i,j-1}, X_{jj})$. The pattern, in which we go up the column below X_{ij} at the same time we go down the diagonal, is suggested by Fig. 7.13.

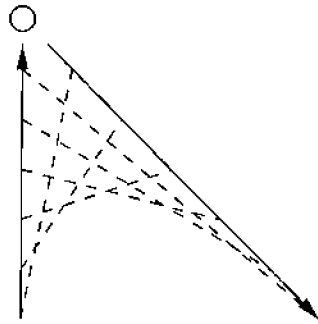


Figure 7.13: Computation of X_{ij} requires matching the column below with the diagonal to the right

Theorem 7.33: The algorithm described above correctly computes X_{ij} for all i and j ; thus w is in $L(G)$ if and only if S is in X_{1n} . Moreover, the running time of the algorithm is $O(n^3)$.

PROOF: The reason the algorithm finds the correct sets of variables was explained as we introduced the basis and inductive parts of the algorithm. For the running time, note that there are $O(n^2)$ entries to compute, and each involves comparing and computing with n pairs of entries. It is important to remember that, although there can be many variables in each set X_{ij} , the grammar G is fixed and the number of its variables does not depend on n , the length of the string w whose membership is being tested. Thus, the time to compare two entries X_{ik} and $X_{k+1,j}$, and find variables to go into X_{ij} is $O(1)$. As there are at most n such pairs for each X_{ij} , the total work is $O(n^3)$. \square

Example 7.34: The following are the productions of a CNF grammar G :

$$\begin{aligned} S &\rightarrow AB \mid BC \\ A &\rightarrow BA \mid a \\ B &\rightarrow CC \mid b \\ C &\rightarrow AB \mid a \end{aligned}$$

We shall test for membership in $L(G)$ the string $baaba$. Figure 7.14 shows the table filled in for this string.

	$\{S, A, C\}$				
-	$\{S, A, C\}$				
-	$\{B\}$	$\{B\}$			
	$\{S, A\}$	$\{B\}$	$\{S, C\}$	$\{S, A\}$	
	$\{B\}$	$\{A, C\}$	$\{A, C\}$	$\{B\}$	$\{A, C\}$
	b	a	a	b	a

Figure 7.14: The table for string $baaba$ constructed by the CYK algorithm

To construct the first (lowest) row, we use the basis rule. We have only to consider which variables have a production body a (those variables are A and C) and which variables have body b (only B does). Thus, above those positions holding a we see the entry $\{A, C\}$, and above the positions holding b we see $\{B\}$. That is, $X_{11} = X_{44} = \{B\}$, and $X_{22} = X_{33} = X_{55} = \{A, C\}$.

In the second row we see the values of X_{12} , X_{23} , X_{34} , and X_{45} . For instance, let us see how X_{12} is computed. There is only one way to break the string from positions 1 to 2, which is ba , into two nonempty substrings. The first must be position 1 and the second must be position 2. In order for a variable to generate ba , it must have a body whose first variable is in $X_{11} = \{B\}$ (i.e., it generates the b) and whose second variable is in $X_{22} = \{A, C\}$ (i.e., it generates the a). This body can only be BA or BC . If we inspect the grammar, we find that the productions $A \rightarrow BA$ and $S \rightarrow BC$ are the only ones with these bodies. Thus, the two heads, A and S , constitute X_{12} .

For a more complex example, consider the computation of X_{24} . We can break the string aab that occupies positions 2 through 4 by ending the first string after position 2 or position 3. That is, we may choose $k = 2$ or $k = 3$ in the definition of X_{24} . Thus, we must consider all bodies in $X_{22}X_{34} \cup X_{23}X_{44}$. This set of strings is $\{A, C\}\{S, C\} \cup \{B\}\{B\} = \{AS, AC, CS, CC, BB\}$. Of the

five strings in this set, only CC is a body, and its head is B . Thus, $X_{24} = \{B\}$.
 \square

7.4.5 Preview of Undecidable CFL Problems

In the next chapters we shall develop a remarkable theory that lets us prove formally that there are problems we cannot solve by any algorithm that can run on a computer. We shall use it to show that a number of simple-to-state questions about grammars and CFL's have no algorithm; they are called "undecidable problems." For now, we shall have to content ourselves with a list of the most significant undecidable questions about context-free grammars and languages. The following are undecidable:

1. Is a given CFG G ambiguous?
2. Is a given CFL inherently ambiguous?
3. Is the intersection of two CFL's empty?
4. Are two CFL's the same?
5. Is a given CFL equal to Σ^* , where Σ is the alphabet of this language?

Notice that the flavor of question (1), about ambiguity, is somewhat different from the others, in that it is a question about a grammar, not a language. All the other questions assume that the language is represented by a grammar or PDA, but the question is about the language(s) defined by the grammar or PDA. For instance, in contrast to question (1), the second question asks, given a grammar G (or a PDA, for that matter), does there exist some equivalent grammar G' that is unambiguous. If G is itself unambiguous, then the answer is surely "yes," but if G is ambiguous, there could still be some other grammar G' for the same language that is unambiguous, as we learned about expression grammars in Example 5.27.

7.4.6 Exercises for Section 7.4

Exercise 7.4.1: Give algorithms to decide the following:

- * a) Is $L(G)$ finite, for a given CFG G ? *Hint:* Use the pumping lemma.
- ! b) Does $L(G)$ contain at least 100 strings, for a given CFG G ?
- !! c) Given a CFG G and one of its variables A , is there any sentential form in which A is the first symbol. *Note:* Remember that it is possible for A to appear first in the middle of some sentential form but then for all the symbols to its left to derive ϵ .

Exercise 7.4.2: Use the technique described in Section 7.4.3 to develop linear-time algorithms for the following questions about CFG's: