Implementation of a Membership Algorithm for Context Free Languages

Ji Kim, Alex Cook, and Aljulanda Almahfudhi

I. Problem Description

Our groups problem will be on the implementation of a membership algorithm for context free languages. This will include converting from Context Free Grammar to Chomsky Normal Form and then implementing the CYK algorithm using the CNF.

II. Approach to Solving

We will be using $C++$ to implement the algorithms. The breakdown of tasks for each step is as follows:

Steps to Convert CFG to CNF:

1. Eliminate lambda productions.

2. Eliminate Unit productions.

3. Eliminate non-terminating and unreachable productions.

Each step will be done is succession and with assumption the previous step has been completed. Implementing CYK algorithm:

1. Obtain a given user input string to test if it can be generated from the CFG.

2. Create n by n triangular array of sets of variables, where n is equal to the length of the given string.

3. Determine the X_{ij} in the array where variables $A \in V | A \Rightarrow *\mathrm{A}_i \ldots A_j$, and the induction on j is $i+1$.

4. Determine if the S is in X_{in}, the beginning to the end of the string. Like CFG to CNF, each step will be done after the previous step has been completed.

III. CFG to CNF

A. Eliminate lambda productions

To eliminate lambda productions, we first have to find all the nullable variables. After finding all the nullable variables, we can now start to construct the null production free grammar. For all the productions in the original grammar , we add the original production as well as all the combinations of the production that can be formed by replacing the nullable variables in the production by . If all the variables on the RHS of the production are nullable , then we do not add '$A \rightarrow \lambda$' to the new grammar.

Pseudo-code:

Given:

$CFGG = (V, T, S, P)$ construct a $CFGG' = (V, T, S, P')$ with no λ productions as follows:

1. Initialize $P' = P$.

2. Find all nullable variables in V, using FindNull.

3. For every production

$A \Rightarrow x$ in P ($x \in \{\ V \cup T\ \}$*),

where x contains nullable variables, add to P' every production that can be obtained from this one by deleting from x one or more of the occurrences in x of nullable variables.

4. Delete all λ productions from P'.

5. In addition, delete any duplicates and delete productions of the form $A \rightarrow A$.

B. Data/Test-Cases:

Part 1 - CFG to CNF test 1: Input: $S \Rightarrow ABAC$

$A \Rightarrow aA \mid \lambda$

$B \Rightarrow bB \mid \lambda$

$C \Rightarrow c$

Output: $S \Rightarrow A$

$\Rightarrow B$

$\Rightarrow AC$

$S \Rightarrow B$

$\Rightarrow AC$

$S \Rightarrow A$

$\Rightarrow BC$

$S \Rightarrow BC$

$S \Rightarrow A$

$\Rightarrow AC$

$S \Rightarrow AC$

$S \Rightarrow C$

A -¿ A A -¿ a B -¿ B B -¿ b -¿ a -¿ b

IV. Methods to Evaluate Results

First, we will be testing various input cases for the CFG, and see if it gets converted to CNF file by comparing map of expected output with another map that was obtained in the program. Second, we will also be creating multiple input cases to test if the array we created is matching the expected array for CYK algorithms. Lastly, we will see if the program is able to successfully determine if the given string can be generated from the CFG.

Description of task:

- Definitions of the problem and terms including formal definitions
- Intended approach to solving the problem
- Pseudo-code, language used, data structures used, techniques used
- Formatting of input/output, data type
- Handling of special cases
- Iterative solution vs recursive solution

Summary of results:

- Showing of the code, explanation of any way it differs from the pseudocode/specific implementation
- Time complexity analysis of each step of the process
- Testing methodology and verification of results

A. Problem Description

Context-free grammars are a class of formal grammars that have applications in language parsing, most notably in programming language compilers. Given a context free grammar CFG and a string, determining if the string is a member of the language described by the CFG is an important decidable problem. There are various algorithms to solve this problem. One such algorithm is The CYK (Cocke Younger Kasami) Algorithm. This paper is focused on the implementation of this algorithm. The algorithm is designed to require the input CFG to be in Chomsky Normal Form CNF, that is a CFG containing no null productions, unit productions, and no non-terminating/unreachable variables in the form:

$$A- > a \oplus A- > CB$$

The right hand side having only one terminal or two non-terminal variables. It is desirable to be able to provide an arbitrary CFG in any form and algorithmically convert it to CNF.

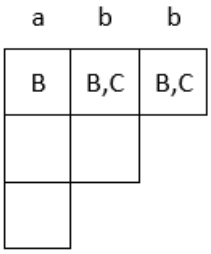
Described informally the process of the CYK algorithm is as follows: The algorithm operates on a triangular array where the longest point is the length of the input string. Each cell is progressively filled with the non-terminal variables that are capable of generating that substring. If the bottom of the array (length 1) contains the start symbol, the entire string can be generated from it. So the string is a member of the language in this case. The first row is handled separately, checking if

the substring of length 1 is generated by any production. The non-terminals that produce that character are added to the cell.

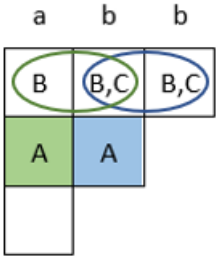
Example CFG:

$$S->AB, A->BC, B->a|b, C->b$$

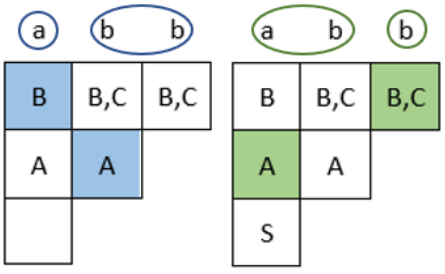
and example string: abb.



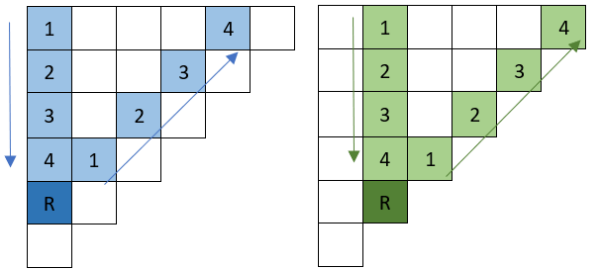
Each subsequent row increases the length of the substring, so a length of 2 for row 2. Each cell is filled with all non-terminal variables that can generate that substring. To determine these variables, the cartesian product is taken for all combinations of substrings.



For the green cell, the cartesian product produces BB, and BC. BC is a production of A, so A belongs to that cell. Likewise for the blue cell. For the last cell we are considering all ways to break up the string into lengths of 1 and 2. Conveniently we already found all ways to generate substrings of lengths 1 and 2 for previous layers.



This pattern is predictable and continues for larger arrays. In this next example the fifth row is being filled. For each cell along the row this pattern must be iterated through.



This concludes the informal description sufficient for tracing the algorithm by hand. The psuedocode for this process is as follows:

```
Inputs: An input string 'I' of length 'n'
        A CFG in CNF 'G' with set of productions 'R' and number of non-terminal variables 'r'
        A 3d array of type bool P[n,n,r]

Step 1: Fill first row of the array

    for S = 1 to n{
        for v = 1 to r{
            if production v produces S, set P[1,S,v] to true
        }
    }
Step 2: Fill rest of the array

    for L = 2 to n{
        for S = 1 to n-L+1{
            for p = 1 to L-1{
                for each combination of P[p,S,v to r] and P[L-p-1, S+p+1, z to r]{
                if P[p,S,v] AND P[L-p,S+p,z] are true,
                set P[L,S,A] to true for each production A -> vz
                }
            }
        }
    }
```

V. Implementation Details

We have chosen to implement the CYK algorithm in $C++$.

For inputs, we are using std::string for our input string. Our CFG will be stored as a vector of pairs of type $(char, string)$. Lastly the main data structure the algorithm operates on will be stored as a 3d array of type bool. To understand this, picture the triangular table from the prior examples as a 2d array, where each cell contains a list of “True” or “False” for each non-terminal variable possible.

VI. Conclusion

we enjoyed working in this project. We have the chance to implement what we have learned in class and that gave us a deeper understanding on implementation of a membership

algorithm for Context Free Languages. Each one of the group member put some efforts on the project, we had some issues and difficulties and we solved them by getting together and work as a team.

VII. Citations

[1] https://www.geeksforgeeks.org/simplifying-context-free-grammars/

[2]

[3]