CockeYoungerKasami

Salome Müller

October 1, 2021

Abstract

Summarize Introduction to 2-3 sentences.

Contents

Introduction

write the introduction

1 Background

The Cocke-Younger-Kasami algorithm, which we analyse in this report, solves the membership problem for context free grammars. In this section we show what a context-free grammar is and how the algorithm operates on it. Further, we introduce the three approaches for implementing the algorithm which were used in this evaluation. And whatever we do in step 3

1.1 Context-Free Grammar

Context-free grammars (CFG) can be used to formalize different types of languages. They can, for example, be used in computer science, to define the structure of programming languages, or in linguistics to define the structure of any language.

A CFG contains a set of rules, also called productions. Starting from a certain variable, this productions can be applied to get a sequence of terminal symbols, for example a sentence of the English language. The sequences that can be generated with a CFG build a language, the grammars context-free language (CFL).

Formally, we define a CFG G by the 4-tuple $G = (V, \Sigma, R, S)$, where V are all non-terminal, and Σ all terminal symbols. R is the set of productions and $S \in V$ is the start symbol.

The productions are of the form $A \to \alpha$, where A is a variable in V and α is a string of symbols from $(V \cup T)^*$. If R contains multiple rules for one non-terminal we abbreviate these rules as $A \to \alpha_1|\alpha_2|\dots|\alpha_k$, where α_i, $i \in 1 \dots k$ is the right hand side of one of the rules for non-terminal A.

If for any strings $u, v \in (V \cup \Sigma^*_)$ there is a production which transforms u to v, we say u directly yields v, denoted as $u \Rightarrow v$. If v can be reached by applying multiple productions on u we say u yields v, denoted as $u \Rightarrow^* v$, i.e., there is a set of strings $u_1, u_2, \dots, u_k \in (V \cup \Sigma)^*)$ such that $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$.

The language of G, $L(G)$, contains all strings that can be yielded from S, i.e. $L(G) = w \in \Sigma^* : S \Rightarrow^* w$.

The membership problem, which is solved by the CYK-algorithm, is the problem of determining whether a given string is in the language of a grammar.

The following subsections show one simple example of a context-free grammar and introduce different forms of grammars.

1.1.1 Exemplary Grammar

One simple example is the grammar, whose language consists of all words of the form $(a^n b^n)$, for any $n \in \mathbb{N}$. This grammar can be defined as $G = (S, A, B, a, b, R, S)$, where R contains the following rules:

$$S \rightarrow ASB$$
$$A \rightarrow a$$
$$B \rightarrow b$$

Any string of length $2n$ can be generated by applying $S \rightarrow ASB$ n times, and then replacing all non-terminal A and B with a and b respectively.

1.1.2 Chomsky Normal Form

Every context-free grammar can be transformed into an equivalent representation in Chomsky normal form (CNF). Two grammars are considered equivalent if they generate the same language. A grammar is in CNF, if all its productions are of the form:
$A \rightarrow BC$
$A \rightarrow a$
$S \rightarrow \epsilon$

While S is the start symbol, A, B and C are any non-terminal variables, but neither B nor C may be the start variable. a is a terminal variable. The start symbol is the only variable, which may yield the empty string, provided the empty string is part of the language. Further, a non-terminal must either yield two non terminals or one terminal variable.

In order to transform the grammar from 1.1.1 to CNF, we add the non-terminals C, D to V and get the new set of productions R':

$$S \rightarrow AB|AC$$
$$C \rightarrow DB$$
$$D \rightarrow AC|AB$$
$$A \rightarrow a$$
$$B \rightarrow b$$

The CYK algorithm can only operate on grammars, that are in the (reduced) Chomsky normal form. The reduced CNF is similar to CNF, with the only difference that S may also appear on the right hand side of a production. The following subsection describes how the CYK-algorithm operates.

1.1.3 Linear Grammars

define by using the book.

1.2 Cocke-Younger-Kasami Algorithm

The Cocke-Younger-Kasami algorithm (CYK) solves the membership problem. For a given Grammar G and an input string $s[1..n]$, it returns the truth-value of whether the s is in the G. For the original algorithm, G should be in reduced CNF or CNF.

The algorithm solves the membership problem in a bottom up manner. It maintains a table tab of size $n \times n$, where $tab[i,j]$ contains all non-terminals that can yield the substring of s of length j starting at position i.

First, tab is initialized as an empty $n \times n$-matrix. The algorithm then starts by looking over all substrings of size 1, to find the non-terminals that produce the terminals of the input string. When this is done, $tab[i,1]$ contains $A : A \in V and A \leftarrow s[i] \in R$ for all $i \in 1 \ldots n$. The algorithms continues by increasing j, starting at 2 till n, and iterating over all possible i, $1 \leq i < n - j$. Since the algorithm proceeds bottom-up, when looking at a substring of length j, the solution for all strings of length $j - 1$ is already known. Thus, to deduce whether non-terminal $A \in V$ can yield a substring $s[i...i + j]$ the algorithm iterates over all non-terminal productions of A. For each rule $A \rightarrow BC$ of A in R it uses the solutions of former solved subproblems to find whether B and C yield them. If for any splitting point k of the current substring, $0 < k < j$, B yields the left part of the split substring, $B \Rightarrow s[i..i + k]$, and C yields the left part, $B \Rightarrow s[i + k + 1..i + j]$, then $A \overset{*}{\Rightarrow} s[i..i + j]$.

For each $tab[i,j]$ with $2 \leq j \leq n$ and $1 \leq i \leq n - j$ the algorithm iterates over all non-terminals $A \in V$ and their productions $A \rightarrow BC \in R$. For each rule, it iterates over all possible splitting points k, $1 \leq k \leq j$. If B is in $tab[i,k]$ and C in $tab[i + k, j - k]$, then A is added to $tab[i,j]$.

The technique of dividing the problem into smaller subproblems and use their solutions to solve the problem is called Dynamic programming.

1.2.1 Dynamic Programming

introduce dynammic programming, say how it works.

CYK applies the dynamic programming technique by dividing the problem into two smaller subproblems and solving them each respectively.

In order for dynamic programming to be applicable on a problem, the problem must fulfil two requirements; optimal substructure and overlapping subproblems. Optimal substructure says, that the optimal solution of a problem can be build from the optimal solutions of a set of subproblem. The membership problem looks for a truth-value, the optimal value is therefore true. When trying to find the truth-value for a non-terminal variable A for any string, we thus try to find any combination of a splitting point k and a production in R with A on its left-hand side, for which both subproblems (the left and right substring), are optimal, i.e., true. If we find such a combination, we can use the answers of both subproblem; the answer of our problem is the logical *and* of the solution of both subproblems.

To proof that the problem has overlapping subproblems, we will give a small example. Assume we run the algorithm on an input string of length 6, $s[1..6]$, and our grammar contains a non-terminal variable A with $(A \rightarrow BC), (A \rightarrow CB) \in R$. At some point, the algorithm might check if A can yield certain substrings of length 4, i.e., substrings $s[1..4]$ and $s[2..6]$. When considering the first rule, $(A \rightarrow BC) \in R$, on $s[1..4]$, it will check whether B yields $s[1..2]$ and C yields $s[3..4]$. When considering the second rule, $(A \rightarrow CB) \in R$, on $s[2..6]$, it will check whether C yields $s[3..4]$ and C yields $s[5..6]$. Finding the truth-value for C on $s[3..4]$ is a subproblem, which will be solved twice, and therefore overlapping. draw a small tree to visualize this

The CYK algorithm has a very good worst-case running time of $O(n^3 * |G|)$. In practice there are algorithms with a better average case running times. In the following subsections, we go into more detail on the running time, while introducing three different parsing algorithms, which were used for the evaluation. The first one is a naive approach, the second one the original CYK-algorithm, and the third one a top-down approach, which makes the naive approach more efficient by introducing memoization.

1.2.2 Naive

The naive approach is a recursive depth-first implementation. It does not use dynamic programming, therefore each subproblem may get solved multiple times. The input string will be stored globally, as an array of characters $s[1..n]$. The procedure takes three input arguments, a non-terminal A and the starting and end points of the substring, i and j, which should be considered. If it is called on a substring of length 1, i.e., $i = j - 1$, it returns the truth-value of whether $A \rightarrow s[i]$ holds. This is done in lines 1 to 7 in 1. Otherwise, it iterates over all non-terminal rules of A, $(A \rightarrow BC) \in R$, trying to find a splitting point k, $i <= k < j$, for which B yields $s[i..k]$ and C yields $s[k+1..j]$. If no such rule can be found, the algorithm returns false, since A can not yield $s[i..j]$. The initial call on the method is `Naive(`S`, 0, `n`)`.

Algorithm 1 Naive Parser

```
1: procedure NAIVE(non-terminal A, int i, int j)
2:     if $i = j - 1$ then
3:         if $(A \rightarrow s[i]) \in R$ then
4:             return true
5:         else
6:             return false
7:     for $(A \rightarrow BC) \in R$ do
8:         for $k \in \{i+1, \ldots, j-1\}$ do
9:             if NAIVE($B$, $i$, $k$) and NAIVE($C$, $k+1$, $j$) then
10:                return true
11:    return false
```

The complexity of this algorithm is exponential in n, the length of the input stringadd reference to book. We expect this approach to be the slowest of the three.

1.2.3 Bottom-Up

This is the original CYK-algorithm. It initializes an empty table *tab* of size $|V| \times n \times n$. When the algorithm is finished, $tab[A, i, j]$ will be true, if non-terminal A can yield $s[i..i + j]$. Notice that j is no longer the end point of the substring, but its length.

Since the algorithm performs bottom-up, it first fills the bottom row of the table, i.e. $tab[A, i, 1]$ for $i \in 1, \ldots, n$ and all $A \in V$. It then iteratively increases j, and fills all cells on its way up through the table. At each cell $tab[A, i, j]$, the algorithm checks if there is a rule $A \to BC$ in R and a k, $1 \leq k < j$, for which $tab[B, i, k]$ and $tab[C, i + k + 1, j]$ are both true. This way, it uses the solutions to already solved subproblems to solve the current problem, only accessing cells of *tab*, which were already filled before. If so, A can yield $s[i..i+j]$, and $tab[A, i, j]$ will is set to true. Since the algorithms were implemented in Java, $tab[C, i + k + 1, j]$ will only be accessed if $tab[B, i, k]$ is true.

Algorithm 2 Bottom-Up CYK Parser

```
1: procedure Bottom-Up(input string $s[1..n]$)
2:     allocate table $tab[|V|][n][n]$ initialized with false
3:     for $i \in 1, \ldots, n$ do
4:         for $A : A \to s[i] \in R$ do
5:             $tab[A, i, 1] \leftarrow$ true
6:     for $j \in 2, \ldots, n$ do                      – length of substring
7:         for $i \in 1, \ldots, n - j + 1$ do          – starting point of substring
8:             for $(A \to BC) \in R$ do                – all productions
9:                 for $k \in 1, \ldots, j - 1$ do      – all splitting points
10:                    if $tab[B, i, k]$ and $tab[C, i + k, j - k]$ then
11:                        $tab[A, i, j] \leftarrow$ true
12:                        break loop
13:    return $tab[S, 1, n]$
```

The bottom-up CYK algorithm solves each subproblem exactly once. It has a complexity of $O(n^3)$reference book. The initialization, lines 3 to 5, takes $O(n)$, due to the iteration over all elements of s. The for-loop on line 9 is repeated at most n times since k is in the interval $\{1, \ldots, n\}$ at most (in practice, it will often be executed less, since it breaks, as soon as the condition is met). The two outer loops, lines 6 and 7, are both repeated n times. Thus, the loop at line 9 will be called $O(n^2)$ times, which results in a complexity of $O(n^3)$ for lines 6 to 12. The overall running time is therefore in $O(n^3)$.

We expect this algorithm to behave very similarly on strings of the same length. Further, the order in which the rules of the grammar are provided does have an impact, but should not affect the running time as much as it does for the top down approach, which we show next.

1.2.4 Top-Down

The top-down approach resembles the naive one, as it is recursive. It uses, however, memoization, which makes it a lot more efficient, as each subproblem is solved once at most. When the method `Top-Down-Parse(input string` $s[1..n]$`)`(Algorithm 3) is called, it initializes the global table of size $|v| \times n \times n$, which is similar to the one used for the bottom-up CYK algorithm. It then calls `Top-Down(`S`, 1, `n`)`(Algorithm 4) and returns $tab[S, 1, n]$, which contains the truth value of the membership problem. `Top-Down(`A`, `i`, `j`)` first checks, whether the subproblem of whether A yields $s[i..j]$ was already solved, i.e., if $tab[A, i, j]$ is set. If so, it returns the before computed truth-value. Otherwise, the value is computed recursively, stored in $tab[A, i, j]$ and returned. The next call of `Top-Down(`A`, `i`, `j`)` will not compute anything, but return the truth-value immediately.

Similar to bottom-up, the right-hand side of the if-request on line 11 will only be called, if the left-hand side is true.

Algorithm 3 Top-Down Parser

```
1: procedure TOP-DOWN-PARSE(input string $s[1..n]$)
2:     allocate global table $tab[|V|][n][n]$ initialized with null
3:     TOP-DOWN-PARSER($S$, 1, $n$)
4:     return $tab[S, 1, n]$
```

Algorithm 4 Top-Down

```
1: procedure TOP-DOWN(non-terminal A, int i, int j)
2:     if $tab[A, i.j]$ =null then
3:         return $tab[A, i, j]$
4:     $tab[A, i.j] \leftarrow$ false
5:     if $j = 0$ then
6:         if $(A \to s[i]) \in R$ then
7:             $tab[A, i.j] \leftarrow$ true
8:     else
9:         for $(A \to BC) \in R$ do
10:            for $k \in \{i+1, \dots, j-1\}$ do
11:                if TOP-DOWN($B$, $i$, $k$) and TOP-DOWN($C$, $i+k$, $j-k$) then
12:                    $tab[A, i.j] \leftarrow$ true
13:                    break loop
14:    return $tab[A, i.j]$
```

The complexity of this algorithm is, similar to the bottom-up algorithm, $O(n^3)$. The difference between the two is, that bottom-up fills all cells of tab, while top-down only fills the ones it passes while trying to find a solution. In practice, its running time is therefore more dependant on the input string itself, as well as the grammar. Depending of the order of the rules, it may yield very different running times. If it consults rules, that yield the considered substrings in the beginning, it does not consult other rules, and must therefore solve a lot less subproblems. If this is not the case, and most of the subproblems must be solved, then we expect the algorithm to be slower than bottom-up.

1.2.5 Implementation

Write a little bit about what data structures where used to represent the productions etc. Necessary?

Add a chapter about the enhancements which are made in step 3

2 Evaluation

In this section we show different experiments that were run on the different approaches and analyse their running times, as well as the number of iterations in the inner most loop or their recursive calls respectively. First, we give an overview over different grammars, that were used, and explain how we expect the algorithms to behave when parsing input strings on them. All of them are in (reduced) Chomsky Normal Form.

2.1 Grammars

2.1.1 Dyck Language

This language consists of all words, that have the correct amount of opening and closing parentheses, i.e., strings of the form '()...()' or '((...))'. The grammar that builds these words has the rules:

$$\begin{aligned}
&S \rightarrow SS|LA|LR\\
&A \rightarrow SR\\
&L \rightarrow (\\
&R \rightarrow)
\end{aligned}$$

We will run experiments on words of the language, as well as on strings with additional single parentheses, i.e., ')()...()' and "()...()(', which are not part of the language.

For this grammar, we expect top-down to run faster on strings of the form '()..()'. The algorithm iterates over the rules in the order as they were parse to the program, thus $S \rightarrow SS$ is the first rule. It then iterates over different splitting points, starting with $k = 1$, which will not find a solution, as S can not yield any of the substrings, since they have a different amount f opening and closing parentheses. when trying $k = 2$, `Top-Down(`S`, 1, 2)` is called, which will return true. For the right hand side of the string, this will be repeated. Thus, with the first rule of the grammar and the second splitting point, the optimal answer is returned and the algorithm is expected to terminate relatively fast.

For strings of the form '((..))', the top-down algorithm will look at a lot more subproblems while looking for the solution. Different from strings of the form '()..()' this string has no partitioning into two substrings, where S can yield both substrings. All rules have to be applied on S at least once, thus a lot more rules and parts of the string will be considered before the answer is found.

Further, ')()..()' is parsed faster than '()..()(' by the top-down algorithm. The way the algorithm was implemented makes us not look at the right part of a splitting (second call of `Top-Down` on line 10 in algorithm4). This means, when the first symbol in the string violates the constraints of the language, we look at the left hand side of every partitioning, find that it can not be yielded, and terminate. If, on the other hand, the last symbol violates the constraint, we will find for a lot of splitting points, that S can yield the left substring, and check the right substring. This yields in a lot more subproblems which have to be considered, and thus in a longer running time.

2.1.2 Strings Starting or Ending in a

This grammar contains all worlds with an arbitrary number of a's and b's in any order, but starting resp. ending with a. The rules for strings starting with a are:

$$\begin{aligned} &S \to AB \\ &B \to BB|a|b \\ &A \to a \end{aligned}$$

and those for strings ending in a:

$$\begin{aligned} &S \to BA \\ &B \to BB|a|b \\ &A \to a \end{aligned}$$

For both of these grammars we will run tests on strings of the form 'ab..ab' and 'ba..ba'. We expect a similar behavior as in the Dyck grammar. The bottom-up approach will take a similar amount of time for both sets of test strings with either grammar. Top-down on the other hand will perform better on the strings that are in the language for both grammars, as well as when parsing the strings not starting in a with the grammar starting in a. It performs worse when parsing strings that do not end in a with the grammar that ends in a, with the same reasoning as we applied for the Dyck language. Top down runs faster, when the symbol that violates the constraint of the grammar is in the beginning of the string.

2.1.3 Equal Numbers

Equal numbers is a grammar, that yields all strings with the same amount of a's and b's. This is achieved with the following rules:

$$\begin{aligned}&S \rightarrow SS|AB|BA\\&B \rightarrow SB|BS|b\\&A \rightarrow a\end{aligned}$$

For this grammar we will test strings of the form aa..bb and ab..ab, as well as both of these sets with additional a's. We expect both parser to run slower on this, than on strings starting with a. It may be similar to strings ending in a though, since we always must look at the whole string. An additional a in the end may be worse than one at the beginning. It is also worse than parentheses on '()..()', but similar to '((..))'. top-down will be faster on ab..ab than bottom up. think more about expectations, very little time invested so far

2.1.4 A Finite Language

come up with a finite language ?

2.2 Experiments

We will now show the results of experiments, using the grammars described, and analyze whether the algorithms behave as we expect them to behave. In order to get better results, for each input string the parser was run 10 times. The time showed in the plots is the average of the resulting running times, where the fastest and slowest times were excluded.

2.2.1 Dyck Language

Figure 1 shows the running times of the bottom-up parser on different sets of input strings. The strings were of length 100 to 5000, growing in steps of 100. As we assumed, the times are very similar for all four different sets of input strings. Parsing the strings of the form '((..))' is a little faster. We do not know why.

The curves are asymptotically to $O(n^3)$, in fact, the yellow, blue and green line are very close to $6.4 * 10^{-10} * n^3$.

We split the parsings of top-down into two plots. The first one, Figure 2, is for the slow cases, where we did not run the parser on strings longer than 2500 The second one, Figure 3, was for the fast cases, where we extended the test set to contain strings up to a size of 10000.

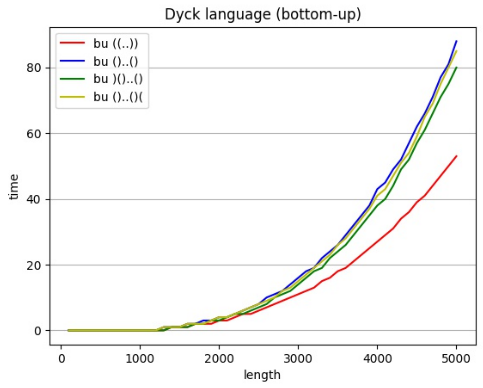

Figure 1: Running time (s) of the bottom-up algorithm when parsing different set of strings of sizes 100-5000, in steps of hundred, for the Dyck Language.

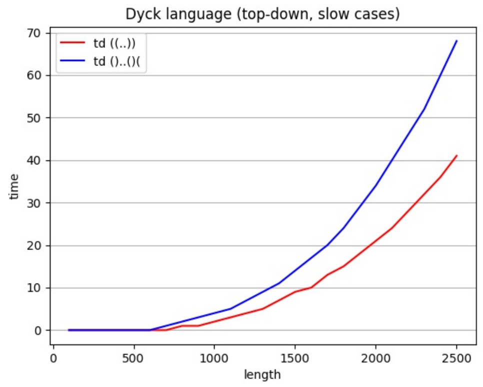

Figure 2: Running time (s) of the top-down algorithm when parsing two set of strings of sizes 100-2500, in steps of hundred, for the Dyck Language.

As we assumed, parsing the test sets of strings of the form '((..))' and '()..()(' with top-down was a lot slower than parsing the strings of the other two sets. While parsing strings of the form '()..()(' of length 2500 took almost 70 seconds, parsing strings of the form '()..()' of length 10'000 took only 0.4 seconds.

The fast cases are a lot faster than the bottom-up parser. This is the case, because bottom up fills all cells of *tab*, regardless of whether or not they are needed to find the solution, while top down only fills the one needed to find the optimal solution. When the rules of the grammar are in a favorable order and the splitting points for finding subproblems that yield the optimal solution is low, as it is the case in the fast cases of Dyck, it can be very fast.

However, if this is not the case, then the parser may take a lot of time. We

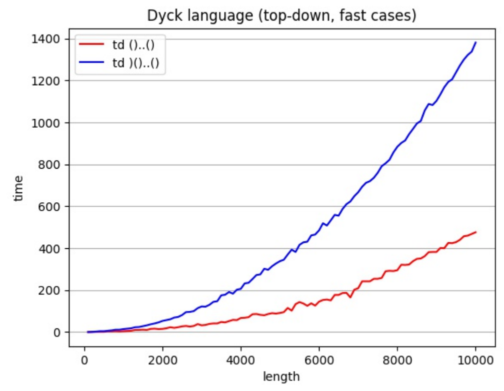

Figure 3: Running time (ms) of the top-down algorithm when parsing two set of strings of sizes 100-10000, in steps of hundred, for the Dyck Language.

can see this at the slow cases of the Dyck language. Here, the parser behaves a lot worse than bottom up. This may be due to the fact, that bottom up fills the cells of *tab* in a structured way, accessing the already filled cells of *tab* not more often, then needed to fill the other cells. Top-down on the other hand may run into the same subproblems very often. This means the algorithm calls itself recursively and to access the cell of the same subproblem more often than bottom up does. Since recursive calls are more time consuming, and more accesses may be performed, this results in a potentially very bad running time.

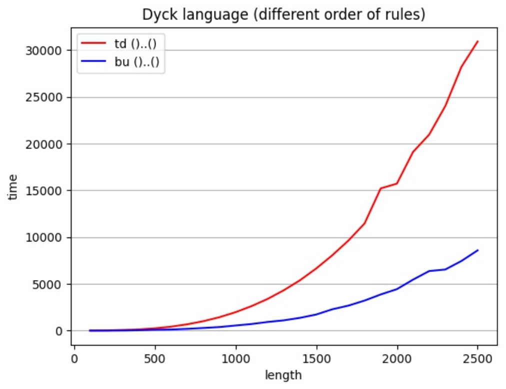

Figure 4: Running time (ms) of both algorithms for parsing a set of strings of sizes 100-2500, in steps of hundred, for the Dyck Language, with a different order of the rules.

I order to verify the hypothesis, that the order of rules matters for the top-down parser, we run experiments on the same grammar, but with the rules of S in opposite order. The results can be seen in figure 4. The parser is in fact a lot slower than it was before, thus the order of the rules may play a major rule, when parsing. We further see that it does not matter that much for the bottom up parser. It has almost the same running time, than it had with the original ordering of the rules.

2.3 Strings starting and ending in a

The running times achieved, did not meat the expectations. Surprisingly, both the top down and bottom-up algorithm performed very differently on the two grammars, being very fast at parsing for the grammar for words starting in a, and slower for words ending in a. In general, we see that the times are lower than they were for the Dyck language. This is presumably because this grammar has only two non-terminal rules, which leads to less subproblems which have to be considered.

We split the results in three graphs, one for the running times of both parsers on the grammar ending in a (Figure 5), one for top-down and one for bottom-up, each for the grammar starting in a (Figure 7 and 6 respectively).

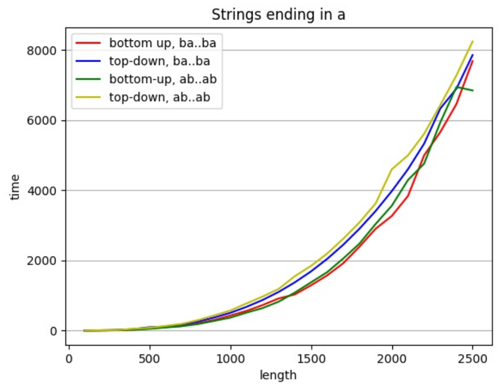

Figure 5: Running time (ms) of the bottom-up and top-down algorithm when parsing two set of strings of sizes 100-2500, in steps of hundred, for the Language of words ending in a.

We see, that the curves for parsing the grammar ending in a are almost identical with the ones of the bottom up parser run on the Dyck Language.

However, the running times for the grammar for strings starting in a are a lot faster. If we consider how *tab* will be filled by the algorithm, it becomes clear, why that is. Remember that *tab* for this grammar is $3 \times n \times n$, since it has three non-terminal variables. we will look at the two dimensional table of each non-terminal.

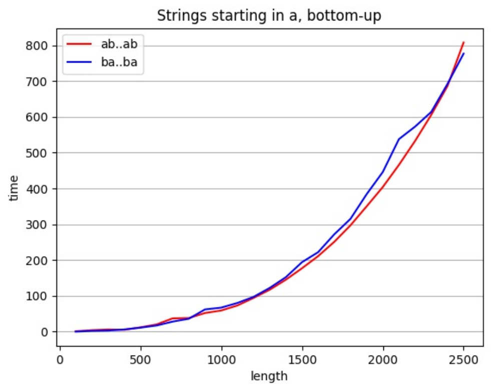

Figure 6: Running time (ms) of the bottom-up algorithm when parsing two set of strings of sizes 100-2500, in steps of hundred, for the Language of words starting in a.

The table for A, say tab_A, has true only in the row for substrings of length 1 and where the symbol is a. As it has no non-terminal rule, when trying to fill the reminder of tab_A the algorithm proceeds very fast, since no splitting has to be considered, as the corresponding loop over k is never even accessed.

The table for B, tab_B, has true in all cells. All substrings of length one can be yielded, since B has the two terminal rules $B \to a|b$. Further, since its only non-terminal rule is $B \to BB$, when trying to fill a new cell of tab_B, only other cells of tab_B must be considered. BEcause they are all true, the loop over the splitting point k gets breaked after the first splitting point, thus filling tab_B can be done in a slow matter, too.

Let's now look at tab_S. Since the only rule for S is $S \to AB$, first the cell of tab_A gets accessed. The first splitting point always generates a substring of length one, on the left of k. If this is an a, then the corresponding cell in tab_A is true, and we must access tab_B- as well. As we argued before, this value will always be true, we are thus not looking at any other splitting points. If the substring is b, then the cell in tab_A is false, and we will not look at tab_B, but continue to the next splitting point.

For the grammar for all words ending in a, tab_A and tab_B are filled in a very similar manner. However, when filling tab_S we have almost twice the amount of table accesses to tab_B. Since the rule for S is $S \to BA$, for every splitting point first tab_B gets accessed, which will always return true, and then tab_A gets accessed, which will return false in most cases. Thus, both tab_B and tab_A are accessed for every k, while for strings starting in a tab_B was only accessed when tab_A was true.

For strings starting in a, we expect very similar running times for strings of the form aa..bb. For strings ending in a, we expect worse running times for those strings. We would expect it to be even worse for strings of the form ab..bb, while for strings starting in a it would result in a similar running time. run experiments, include plots

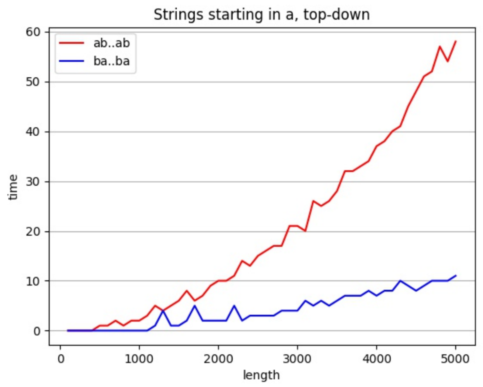

Figure 7: Running time (ms) of the top-down algorithm when parsing two set of strings of sizes 100-5000, in steps of hundred, for the Language of words starting in a.

as we see in Figure 7, the running times for top-down were incredibly low. The three bumps in the blue line are supposedly due to rounding errors of the compiler, seen as the times there are lower than 5 milliseconds. The bumps appear on the red line at the same time after the same period of time after starting the parsers (not at the same length!), though not as distinctive as in the blue line, since the running times are already a little higher at this point.

The incredibly low running times can be explained with similar reasoning, as for the low running times of bottom up on strings starting with a. Top down is even faster, since it does not fill *tab* completely. In fact, when parsing strings not starting in a for the grammar of strings starting in a, it will only ever look at the most left children of the recursive tree, since everything returns falls This results in a ver low amount of recursive calls. In fact, the number of calls on the recursive function is $\Theta(n)$ (figure 8). As we argued in section 1.2.4, the upper bound for this number is $O(n^3)$. The numbers for the counter of bottom up (repetitions of the inner most loop, i.e. over splitting points k) for parsing strings for the grammar of strings starting in a, are somewhere in the middle of n^2 and n^2, still yielding relatively fast running times.

Add subsection for evaluations of step 3

3 Conclusion

summarize the outcome of the experiments and draw some further conclusions.

Botto-up is very steady, it's behavior can easily be predicted, as it is mainly dependent on the length of the input string. Top-down on the other hand, since

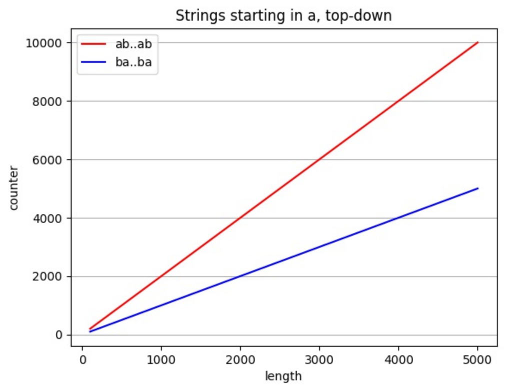

Figure 8: Running time (ms) of the top-down algorithm when parsing two set of strings of sizes 100-5000, in steps of hundred, for the Language of words starting in a.

it only looks at subproblems that are needed to find the optimal solution, can be a lot faster. When the rules of the grammar are in the correct order, i.e., in an order such that the rules used for optimal solutions are considered first, and the splitting points to find said solutions are low, then this parser will yield very good running times. If neither is the case, it can be a lot slower than bottom-up, since bottom-up fills the cells in a structured way and thus accesses the cells less often. Top down may access the same cell a lot of times, when the same subproblem occurs very often, resulting in a bad.

Bibliography. Cited works are mainly both books used in the lecture.