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 productions, 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 $G = (V, T, P, S)$. V and T are two finite, disjoint sets containing all **variables** and **terminal** symbols respectively. The variables are also called non-terminals. P 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 P contains multiple productions for one non-terminal we abbreviate these productions as $A \to \alpha_1 | \alpha_2 | \ldots | \alpha_k$, where α_i, $i \in 1 \ldots k$ is the right hand side of one of the productions for non-terminal A.

If for any strings $u, v \in (V \cup T)^*$ there is a production which can be applied to u such that the result is v we say u **directly derives** v, denoted as $u \Rightarrow v$. If v can be reached by applying multiple productions on u we say u **derives** v, denoted as $u \overset{*}{\Rightarrow} v$, i.e., there is a set of strings $u_1, u_2, \ldots, u_k \in (V \cup T)^*$) such that $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$.

The language generated by G, $L(G)$, contains all strings that can be yielded from S, i.e. $L(G) = w \in T^* : S \overset{*}{\Rightarrow} w$.

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 $G = (S, a, b, P, S)$, where P contains the following production:

$$S \to aSb|ab$$

Any string of length $2n$ can be generated by applying $S \to ASB$ n times, and then replacing all variables 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:

$A \to BC$
$A \to a$
$S \to \epsilon$

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

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.

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

$$S \to AC|AB$$
$$C \to SB$$
$$A \to a$$
$$B \to b$$

In 1.2.1 we show, why CYK depends on grammars to be in reduced CNF.

1.1.3 Linear Grammars

Similar to CNF, a **Linear Grammar** also has certain restrictions on how the productions may look like. They have, however, only one restriction; each production may at most have one non-terminal variable on its right-hand side.

In Section where? we use **linear context free-grammars in Chomsky normal form** to generalize the CYK algorithm. These are grammars that are linear and where all productions have at either one terminal symbol, or a non-terminal variable and a terminal symbol on their right-hand side. The example we gave before can be easily transformed int o linear CNF, by removing non-terminal variable A

$$S \to aB$$
$$B \to Sb|b$$

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 derive 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 $A : A \in VandA \leftarrow s[i] \in P$ 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 derive a substring $s[i...i + j]$ the algorithm iterates over all non-terminal productions of A. For each production $A \to BC$ of A in P it uses the solutions of former solved subproblems to find whether B and C derive 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 $A \to BC \in P$. For each production, 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

Dynamic programming is a technique to solve optimization problems. reference intro to algorithms The problems are solved, by combining the solutions to subproblems. It is typically applied on problems, where certain subproblems

must be solved multiple times in order to find the solution to a problem. To solve them only once, a table is created that stores the solution to subproblems. This method thus performs a memory-time tradeoff; it uses memory to save computation time.

There are two different approaches on how to apply dynamic programming. **Bottom-up** orders subproblems by size and then solves the smallest first. It then uses the solutions to these subproblems to solve the bigger ones, thus it fills the table from bottom to top, solving all subproblems the problem has. Whenever a new subproblem is solved, all of its subproblems have already been solved. **Top-down with memoization** on the other hand solves the problem recursively. Thus, if the method is called on a subproblem small enough it solves it directly. Otherwise it divides it into more subproblems and calls itself on them. However, at each recursive call it first accesses the table to see, if the current subproblem has been solved before. If so, this solution is returned. If not, it continues as usual, but stores the found solution is stored in the table. When this subproblem appears the next time, it must not be computed again.

In order for dynamic programming to be applicable on a problem, the problem must fulfil two requirements. The first, **Optimal substructure**, says, that the optimal solution of a problem can be build from the optimal solutions of a set of subproblem. **Overlapping subproblems** is the second requirement, assuring that the problem has overlapping subproblems.

CYK applies the bottom-up approach, by dividing the string into two smaller substrings and solving them each respectively. The membership problem looks for a truth-value, for a subproblem are therefore only two possible answers, true or false. The optimal value is true, as the algorithm tries to find a combination of subproblems, that returns true. When trying to find the truth-value for a non-terminal variable A for any string, we try to find any combination of a splitting point k and a non-terminal production in P for A, $A \rightarrow BC$, that derives the string. The resulting subproblems are whether B and C can derive the left and right substring respectively. The answer of our problem is the logical *and* of the solution of both subproblems, i.e., if for any such combination both subproblems return true, then A can derive the string. This already shows, that the optimal substructure holds for the membership problem.

To show that the problem has overlapping subproblems, we will use the tree in figure 1, which shows all subproblems of a string of length four. Any algorithm trying to solve every subproblem would thus compute 26 subproblems. A algorithm using memoization only solves 9 subproblems, and reuses the solution for the remaining 17 subproblems. This tree is a simplification, as for the algorithms presented in the following subsections the actual trees would look different, since they compute the subproblems with regard to given non-terminal variables. Even though the numbers differ, it should become clear that there may be a big number of subproblems, which shows that the membership problem indeed has overlapping subproblems.

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 in-

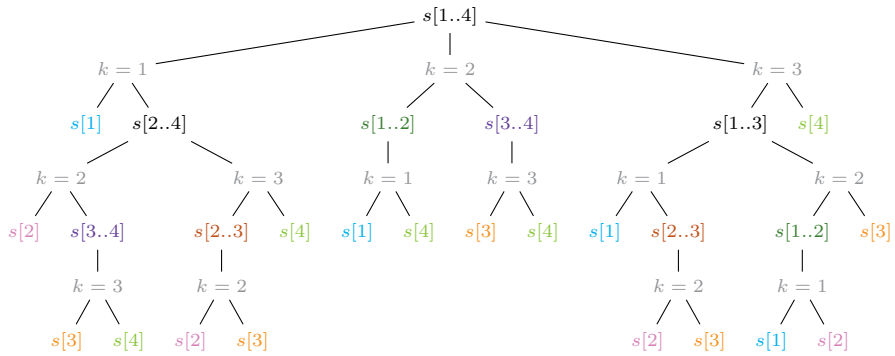

Figure 1: Tree showing the subproblems of the membership problem on a string of length 4, $s[1..4]$. All nodes of same color are the same subproblem. The nodes for splitting points (position k in the string, grey) are added only for readability.

troducing three different parsing algorithms, which were used for the evaluation. The first one is a naïve approach, the second one the original CYK-algorithm, and the third one a top-down approach, which makes the naïve approach more efficient by introducing memoization.

1.2.2 Naïve

The naïve 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 $s[1..n]$. `counter` is a global variable which is initialized with 0 and increased at every recursive call. 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 $A \to s[i]$ holds. This is done in lines 1 to 7 in 1. Otherwise, it iterates over all non-terminal productions of A, $(A \to BC) \in P$, 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 production can be found, the algorithm returns false, since A can not derive $s[i..j]$. The initial call on the method is `Naive(`S`, 1, `n`)`.

Algorithm 1 Naive Parser

```
1:  function NAIVE(non-terminal A, int i, int j)
2:      counter ← counter + 1
3:      if i = j then
4:          if (A → s[i]) ∈ P then
5:              return true
6:          else
7:              return false
8:      for (A → BC) ∈ P do
9:          for k ∈ {i, ..., j − 1} do
10:             if NAIVE(B, i, k) and NAIVE(C, k + 1, j) then
11:                 return true
12:     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 derive $s[i..i + j]$. Notice that j is no longer the end point of the substring, but its length. Further, it initializes a `counter` too. Here, `counter` holds the number of iterations on the innermost loop.

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 production $A \to BC$ in P 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 derive $s[i..i + j]$, 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:  function BOTTOM-UP(input string s[1..n])
2:      allocate table tab[|V|][n][n] initialized with false
3:      counter ← 0
4:      for i ∈ 1, ..., n do
5:          for A : A → s[i] ∈ P do
6:              tab[A, i, 1] ← true
7:      for j ∈ 2, ..., n do                          – length of substring
8:          for i ∈ 1, ..., n − j + 1 do              – starting point of substring
9:              for (A → BC) ∈ P do                   – all productions
10:                 for k ∈ 1, ..., j − 1 do          – all splitting points
11:                     counter ← counter + 1
12:                     if tab[B, i, k] and tab[C, i + k, j − k] then
13:                         tab[A, i, j] ← true
14:                         break loop
15:     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 productions 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 naïve 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, and the `counter`. 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. Like in the naïve approach `counter` holds the number of calls on the recursive function.

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: function TOP-DOWN-PARSE(input string s[1..n])
2:     allocate global table tab[|V|][n][n] initialized with null
3:     counter ← 0
4:     TOP-DOWN-PARSER(S, 1, n)
5:     return tab[S, 1, n]
```

Algorithm 4 Top-Down

```
1: function TOP-DOWN(non-terminal A, int i, int j)
2:     counter ← counter + 1
3:     if tab[A, i.j] =null then
4:         return tab[A, i, j]
5:     tab[A, i.j] ← false
6:     if j = 0 then
7:         if (A → s[i]) ∈ P then
8:             tab[A, i.j] ← true
9:     else
10:        for (A → BC) ∈ P do
11:            for k ∈ {i + 1, ..., j − 1} do
12:                if TOP-DOWN(B, i, k) and TOP-DOWN(C, i+k, j−k) then
13:                    tab[A, i.j] ← true
14:                    break loop
15:    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 productions, it may derive very different running times. If it consults productions, that derive the considered substrings in the beginning, it does not consult other productions, 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 using the different parsers 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 productions:

$$S \to SS|LA|LR$$
$$A \to SR$$
$$L \to ($$
$$R \to)$$

We ran 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.

we expect top-down to run faster on strings of the form '()..()'. The algorithm iterates over the predictions in the order in which they were given to the program, thus $S \to SS$ is the first production. 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 of opening and closing parentheses. when trying $k = 2$, `Top-Down(`S`, 1, 2)` is called, which returns *true*. For the right hand side of the string, this is repeated. Thus, with the first production 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 looks at a lot more subproblems while looking for the solution. Opposed to strings of the form '()..()', '((..))' has no partitioning into two substrings, where $S \leftarrow SS$ can yield both substrings. In fact, all productions must be applied at least once for strings of this form to be produced thus more productions and subproblems are considered before the answer is found.

Further, ')()..()' is parsed faster than '()..()(' by the top-down algorithm. The way the algorithm was implemented, the right part of a splitting (second call of `Top-Down` on line 10 in algorithm4) will not be considered, if the left side returns *false*. This means, when the first symbol in the string violates the constraints of the language, the left hand side of every partitioning can not be

yielded, and the right side will never be considered. If, on the other hand, the last symbol violates the constraint, the algorithm finds for a lot of splitting points that S can yield the left substring, and parses the right substring too. This yields in a lot more subproblems which have to be considered. Thus, strings with an error in front are parsed faster than strings of the same form but with an error at the end

2.1.2 Strings Starting or Ending in a

These grammars contain all worlds with an arbitrary number of a's and b's in any order, but starting resp. ending in a. The productions for strings starting in a are:

$$S \to AB$$
$$B \to BB|a|b$$
$$A \to a$$

and those for strings ending in a:

$$S \to BA$$
$$B \to BB|a|b$$
$$A \to a$$

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 *Dyck language*. The bottom-up approach takes a similar amount of time for both sets of test strings with either grammar. Top-down on the other hand performs 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 which 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 productions:

$$S \to SS|AB|BA$$
$$B \to SB|BS|b$$
$$A \to a$$

For this grammar we test strings of the form 'aa..bb' and 'ab..ab', as well as both of these sets with additional a's. We expect the parsers to yield similar running times on this grammar and for this input strings, than parsing '()...()' and '((...))'respectively for the Dyck language since the arrangement of the terminal symbols is very similar, with a and b instead of opening and closing parentheses. The cases that should return *false* are probably slower, since this grammar has one production more than the grammar for the *Dyck language*.

2.1.4 A Finite Language

come up with a finite language ?

2.2 Experiments

We 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

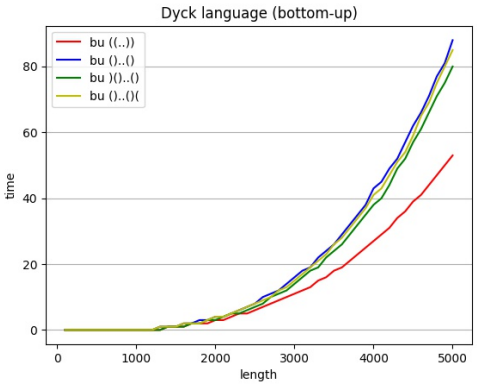

Figure 2: Running time (s) of the bottom-up algorithm when parsing different set of strings of sizes 100-5000, in steps of hundred, for the *Dyck language*.

Figure 2 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. This may be, since... maybe this changes with the new brakpoint. analyse new plots. Otherwise consult one note

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 3, is for the slow cases, where we did not run the parser on strings longer than 2500 The second one, Figure 4, was for the fast cases, where we extended the test set to contain strings up to a size of 10000.

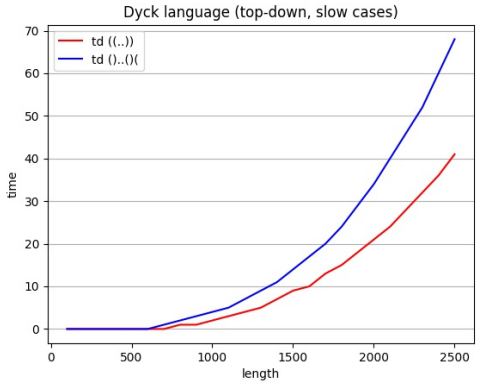

Figure 3: Running time (s) of the top-down algorithm when parsing two set of strings of sizes 100-2500, in steps of hundred, for the *Dyck language*.

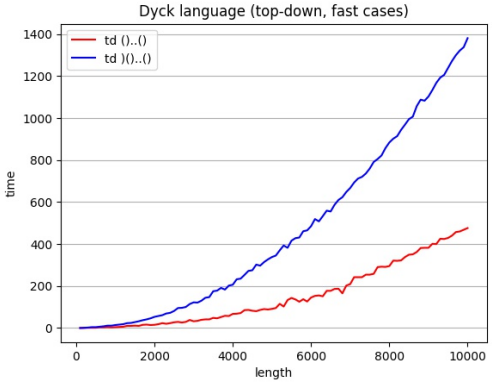

Figure 4: Running time (ms) of the top-down algorithm when parsing two set of strings of sizes 100-10000, in steps of hundred, for the *Dyck language*.

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 productions 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 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 in order 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.

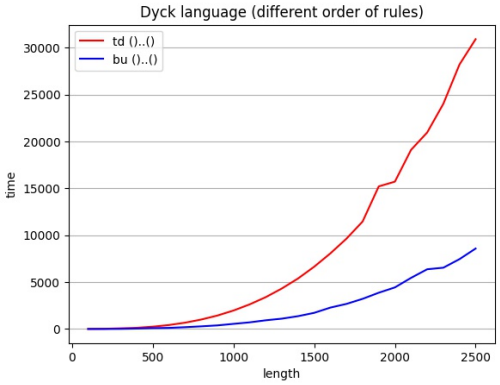

Figure 5: Running time (ms) of both algorithms for parsing a set of strings of sizes 100-2500, in steps of hundred, for the *Dyck language*, with a different order of the productions.

I order to verify the hypothesis, that the order of productions matters for the top-down parser, we run experiments on the same grammar, but with the productions of S in opposite order. The results can be seen in figure 5. The parser is in fact a lot slower than it was before, thus the order of the productions may play a major role, when parsing. The plot further shows that the order 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 productions.

2.3 Strings starting and ending in a

The running times achieved, did not meat all 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 *starting in a*, and slower for *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 productions, 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 6), one for top-down and one for bottom-up, each for the grammar starting in a (Figure 8 and 7 respectively).

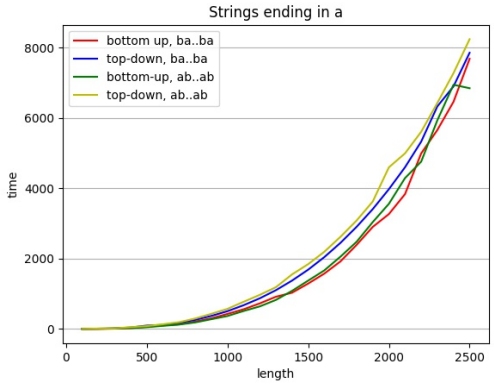

Figure 6: Running time (ms) of the bottom-up and top-down algorithm when parsing two set of strings of sizes 100-2500, in steps of hundred, for the Language of words ending in a.

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 *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 look at the two dimensional table of each non-terminal in turn.

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 production, 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 productions $B \to a|b$. Further, since its only non-terminal production 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 braked after the first splitting point, thus

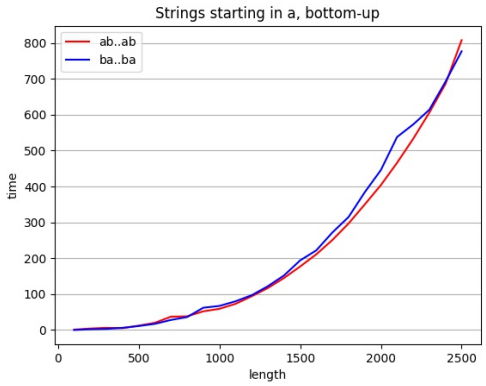

Figure 7: Running time (ms) of the bottom-up algorithm when parsing two set of strings of sizes 100-2500, in steps of hundred, for the Language of words starting in a.

filling tab_B can be done in a slow matter, too.

Let's now look at tab_S. Since the only production 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 *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 production 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 *starting in a* tab_B was only accessed when tab_A was *true*.

For *starting in a*, we expect very similar running times for strings of the form 'aa..bb' and 'bb..aa'. For *ending in a*, we expect worse running times for those strings. We 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

As we see in Figure 8, 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. They could also be reduced by running the parser 40 times per input string

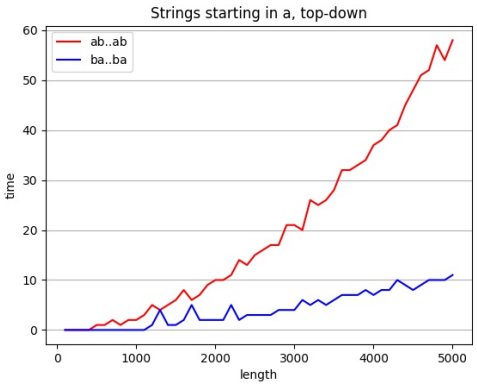

Figure 8: Running time (ms) of the top-down algorithm when parsing two set of strings of sizes 100-5000, in steps of hundred, for the Language of words starting in a.

instead of the 10 times used in all other experiments.

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 *starting in a*, it will only ever look at the most left children of the recursive tree, since each of them returns *false* This results in a very low amount of recursive calls. In fact, the number of calls on the recursive function is $\Theta(n)$ (figure 9). 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 starting in a, are somewhere in the middle of n^2 and n^3, still yielding relatively fast running times. adapt, if new break in bottom up is used.

add evaluation for *equal numbers* Add subsection for evaluations of step 3

3 Conclusion

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

Bottom-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 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

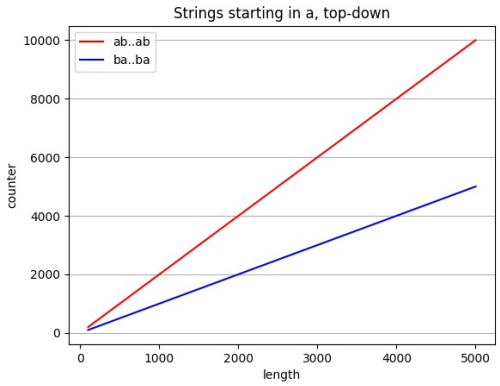

Figure 9: Running time (ms) of the top-down algorithm when parsing two set of strings of sizes 100-5000, in steps of hundred, for the Language of words starting in a.

subproblem occurs very often, resulting in a bad.

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