Evaluation of the Cocke-Younger-Kasami Algorithm

Salome Müller

October 27, 2021

Abstract

The Cocke Younger Kasami algorithm can solve the membership problem for context free grammars in $O(n^3)$. For this report three different approaches to the algorithm are implemented and evaluated. The evaluation shows that the bottom-up approach is faster on average, while top-down has very fast running times in best-case. Further, bottom-up is specialized to operate on linear grammars, yielding running times in $O(n^2)$. The bottom-up approach can also be used to calculate the number of errors and correct the input string, such that the result belongs to the language of the grammar. While the correction of the string has linear running time and proceeds very fast, the generalized bottom-up algorithm is much slower than the original bottom-up algorithm.

Contents

1 Introduction

The question whether a given string is in the language of a given context free grammar (CFG) is known as the membership problem. A CFG contains a set of variables and productions. Applying the productions on the variables derives a finite set of strings, which form the language of the grammar. Naïve approaches to solve the membership problem have exponential running time. In this report, we evaluate the Cocke-Younger-Kasami algorithm, which solves the problem in $O(n^3)$ for grammars that are in Chomsky normal form. We look at a bottom-up approach and a top-down approach which uses memoization. Further, we try to specialize the algorithm to parse strings for linear grammars.

The evaluation shows that the bottom-up and top-down algorithms behave very differently. The running times of bottom-up are steadier than those of top-down, i.e., the running time varies less for parsing different strings of the same length for the same grammar. We further see that the bottom-up approach behaves better on average. Nevertheless, the top-down approach can yield a lot faster running times, depending on the order of the productions of the grammar and the input string.

However, adapting the bottom-up algorithm to parse linear grammars yields even better results. The running time for the specialized algorithm lies in $O(n^2)$.

The CYK algorithm can further be used to not only solve the membership problem, but to also compute the minimal number of errors in the input string. This is the minimal number of symbols in the input string that must be replaced or deleted for the string to be in the language of the grammar. This algorithm is in $O(n^3)$ too, but in practice it is slow compared to the non-generalized algorithm. In a second step, the error count can be used to correct the string. The corresponding algorithm is in $O(n)$.

In Section 2 we define context-free grammars and introduce the CYK algorithm, as well as the three different approaches used for the evaluation. In Section 3 we show how we specialize the algorithm for linear grammars and how the CYK-algorithm can be used to detect and correct errors. Then, we evaluate all algorithms in Section 4.

2 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 the evaluation.

2.1 Context-Free Grammar

Context-free grammars (CFG) are 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 $G = (V, T, P, S)$. 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 $u \Rightarrow v$. For example, applying rule $B- > bc$ on the string aB directly derives the terminal string abc, i.e. $aB \Rightarrow abc$. 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 $u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \cdots \Rightarrow u_k \Rightarrow v$. If we enhance our example by the rule $S \to aB$, then $S \overset{*}{\Rightarrow} abc$, by applying the rule of S first, and then the rule of B.

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 a special of grammars.

2.1.1 Exemplary Grammar

One simple example for a context free grammar is one, 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 $S \to aSb$ $n - 1$ times and $S \to ab$ once in the end.

2.1.2 Chomsky Normal Form

Every context-free grammar can be transformed into an **equivalent** representation in **Chomsky normal form** (CNF). Two grammars are considered equiv-

alent 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 in 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 Section 2.1.1 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 Section 2.2.1 we show, why CYK depends on grammars to be in reduced CNF.

2.2 Cocke-Younger-Kasami Algorithm

The **Cocke-Younger-Kasami** algorithm (CYK) solves the membership problem [1]. For a given Grammar G and an input string $s[1..n]$, it returns the truth-value of whether s is in $L(G)$. For the original algorithm, G must 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. Since the grammar is in CNF, at each terminal production one non-terminal derives exactly one terminal, therefore this can be done in a straight forward manner. When this is done, $tab[i, 1] = \{A : A \in V \text{ and } A \to s[i] \in P\}$ for all $i \in 1 \dots n$. The algorithm continues by increasing j, from 2 to n, and iterating over all possible i, i.e., $1 \leq i < n - j$. 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 $A \to BC$ derives the substring. If for any splitting point k of the current substring, $0 < k < j$, B derives the left part of the split substring, $B \Rightarrow s[i..i+k]$, and C derives 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.

2.2.1 Dynamic Programming

Dynamic programming is a technique to solve optimization problems [2]. 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 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, $A \to BC$, that derives the string. The resulting subproblems are whether B and C can derive the left and

right substring respectively. We can do this for all non-terminal productions, as they all have exactly two non-terminal variables on their right-hand side, since the grammar is in reduced CNF. 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.

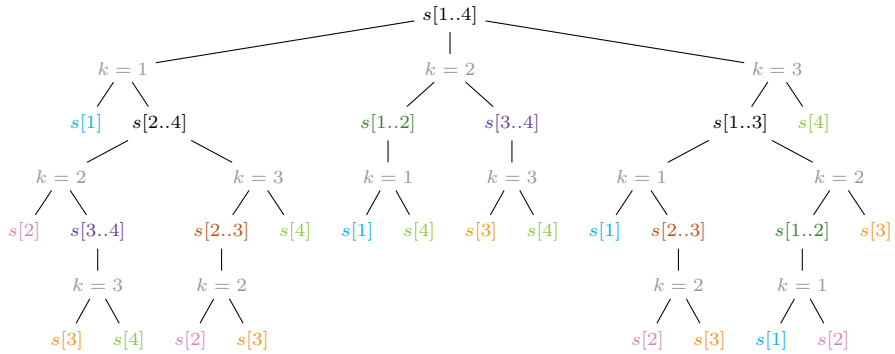


Figure 1: Tree showing the subproblems of the membership problem on a string of length 4, $s[1..4]$. All nodes of same color are the same subproblem. The nodes for splitting points (position k in the string, grey) are added only for readability.

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. The total amount of subproblems is 26. Therefore, any algorithm trying to solve every subproblem computes 26 subproblems. The nodes of the tree are colored, such that the same subproblems have the same color, i.e., the problem has only nine distinct subproblems. An algorithm using memoization thus solves nine subproblems, and reuses the solution for the remaining 17. This tree is a simplification, as for the algorithms presented in the following subsections the actual trees would have more layers, 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 non-distinct subproblems. This 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 introducing 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.

2.2.2 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 $A \in V$ and the starting and end points (i and j) of the substring, 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 Line 3 till Line 7 in Algorithm 1. Otherwise, it iterates over all non-terminal productions of A, $(A \to BC) \in P$. For each production it tries to find a splitting point k, $i <= k < j$, for which B derives $s[i..k]$ and C derives $s[k+1..j]$, using recursive calls. 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
```

Imagine, that instead of all possible splitting points k the algorithm would only try the first and the last one. This would result in four recursive calls. Two of those are on the substrings of length 1, and run in constant time, as they immediately return a result (Line 5 or Line 7) The other two recursive calls would again each call 4 recursive calls, of which two are constant. This is repeated, until a substring of size 2 is reached, for which all calls return in constant time. This would result in a total of 2^n recursive calls, thus the running time of such an algorithm would be in $O(2^{n-1})$. However, Algorithm 1 does not try 2 but $n - 1$ splitting points in worst case, and thus performs $2 * (n - 1)$ recursive calls. The amount of subproblems tht are considered this way is $(n - 1)/2 * O(2^{n-1})$. Thus, the running time is exponential in n [1].

The following algorithms show, how the running time can be improved by using memoization.

2.2.3 Bottom-Up

The bottom-up approach is the original CYK-algorithm. It initializes a table *tab* of size $|V| \times n \times n$. When the algorithm terminates, $tab[A, i, j]$ 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`, which 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 both are true, 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 $O(n^3)$ [1]. The initialization, Line 3 till Line 6, takes $O(n)$, due to the iteration over all elements of s. The for-loop on Line 10 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, Line 7 and Line 8, are both repeated n times. Thus, the loop at Line 10 will be called $O(n^2)$ times, which results in a complexity of $O(n^3)$ for the nested loops. 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.

2.2.4 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(`$s[1..n]$`)` 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`)` and returns $tab[S, 1, n]$, which contains the truth value of the membership problem. `Top-Down(`A`, `i`, `j`)` first checks, if the subproblem of whether A derives $s[i..j]$ 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 the naïve approach, the top-down algorithm has a `counter` which holds the number of calls on the recursive function.

Similar to bottom-up, the right-hand side of the if-request on Line 17 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]
 6: function TOP-DOWN(non-terminal A, int i, int j)
 7:     counter ← counter + 1
 8:     if tab[A, i, j] =null then
 9:         return tab[A, i, j]
10:     tab[A, i, j] ← false
11:     if j = 0 then
12:         if (A → s[i]) ∈ P then
13:             tab[A, i, j] ← true
14:     else
15:         for (A → BC) ∈ P do
16:             for k ∈ {i + 1, ..., j − 1} do
17:                 if TOP-DOWN(B, i, k) and TOP-DOWN(C, i+k, j−k) then
18:                     tab[A, i, j] ← true
19:                     break loop
20:     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.

In the following sections we first show a specializations and a generalization of the CYK algorithm, before we evaluate them and the conventional implementation.

3 Generalization and Specialization

In this section we try a specialization and a generalization of the CYK algorithm. For the specialization we parse grammars in a different form, namely **linear grammars**, instead of grammars in CNF. We first convert these grammars to CNF and compare the running times to the previous experiments. In a second step, we adapt the CYK algorithm to parse strings for those grammars, and compare the efficiency of both approaches.

The generalization extends the bottom-up CYL algorithm by not only returning truth values for the membership problem, but by computing the errors in the input string. This error count is then used to correct the input string, such that it becomes a member of the language.

3.1 Specialization with Linear Grammars

Similar to CNF, linear grammars also have 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.

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 either one terminal symbol, or a non-terminal variable and a terminal symbol on their right-hand side. The example we gave in Section 2.1.1 can be easily transformed into linear CNF, by removing non-terminal variable A, resulting in the following productions:

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

3.1.1 Transform Linear grammars to CNF

A linear grammar can be easily transformed into CNF, by introducing a non-terminal variable a_T for each terminal symbol a which appears in a non-terminal

production. For every variable that is added that way, the terminal production $a_T \to a$ is added to the set of productions. For the exemplary grammar this would give the productions

$$\begin{aligned} S &\to a_T B \\ B &\to S b_T | b \\ a_T &\to a \\ b_T &\to b \end{aligned}$$

By adding non-terminal variables to the grammar, we expand one dimension of the memoization table. Since those have no non-terminal productions, for both the top-down and bottom-up algorithm no additional rules are tested. However, the cells for those non-terminals may be accessed often for strings of length bigger than 1, always returning false. These unnecessary calls may extend the running time. We thus expect test runs on this grammar to yield similar running times than for equivalent grammars already in CNF. If we do not transform the grammar into CNF but instead adapt the CYK-algorithm, this extension may be avoided.

3.1.2 Adapt CYK to Linear Grammars

Grammars in Linear CNF have two types of rules; terminal rules and non terminal rules. Similar to the non-specialized approach, we can use the terminal rules to determine, which non-terminal variables can directly derive which terminal symbols. The non-terminal rules are different, and require the algorithm to act differently.

They all have exactly on terminal and one non-terminal variable on their right hand side. In contrast to the non-specialized approaches, we do not need to look at multiple splitting points, to determine whether a non-terminal rule applied on a given non-terminal variable can derive a string. The terminal symbol must be equal to the last or first symbol of the substring, depending on wether it is the first or second variable on the right-hand side of the production. The non-terminal variable must be able to derive the remaining symbols of the considered string. Therefore, only the last or the first splitting point need to be considered. Algorithm 4 shows how this can be applied to the bottom-up CYK algorithm.

We chose to adapt the bottom-up algorithm rather than top-down, because it is less likely to run into stack-overflow errors. For big input strings, top-down may cause them because the compiler loses track of the recursive calls.

Algorithm 4 Linear Bottom-Up CYK Parser

```
1:  function LIN-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
8:          for i ∈ 1, ..., n − j + 1 do
9:              for (A → v1 v2) ∈ P do
10:                 counter ← counter + 1
11:                 if v1 is terminal symbol then
12:                     if v1 = s[i] and tab[v2, i + 1, j − 1] then
13:                         tab[A, i, j] ← true
14:                         break loop
15:                 else
16:                     if v2 = s[i + j] and tab[v1, i, j − 1] then
17:                         tab[A, i, j] ← true
18:                         break loop
19:     return tab[S, 1, n]
```

The only difference between Algorithm 2 and Algorithm 4 is what happens at the inner most loop, beneath Line 10. Instead of iterating over all possible splitting points, we first check which part of the rule's right-hand side is the terminal symbol. We then ask if this is equal to the respective symbol in the input string. If so check whether the terminal variable can derive the rest of the substring.

The running time of this algorithm is $O(n^2)$, since it is similar to the non-specialized bottom-up algorithm, but the inner most for-loop is not $O(n)$ anymore, but constant.

Compared to parsing strings for a linear grammar, which was transformed to CNF, we expect this algorithm to be more efficient, since it must try less splitting points per production. Further, it will use less memory, since there are less non-terminal variables, compared to the transformed grammar, thus one dimension of *tab* is smaller.

3.2 Generalization for Error Correction

One may not only be interested in whether the input string is in the language, but also in how many errors there are and how the string could be altered to fit into the language. The bottom-up CYK algorithm can be adapted to fit this purpose. Algorithm 5 resembles the bottom-up algorithm from Section 2.2.3 but generalizes it by counting the errors. For this, every cell of the memoization table will contain the configuration, i.e., the rule and splitting point, that derive a string which is closest to the actual substring of the input string. We chose the bottom-up approach, since all cells of the table must be filled to find this minimal configuration, and the bottom-up algorithm performs better on problems where all cells are filled, compared to the top-down algorithm(Section 4).

An error is a symbol in the input string which must be either deleted or replaced with another terminal symbol, in order for the start-variable to be able to derive the string. To count these deletions and replacements, the structure of the memoization table *tab* is altered. *tab* no longer stores boolean values, but 4-tuples, i.e., (e, B, C, k), for every $tab[A, i, j]$. Among all strings which A derives, the closest ones to substring $s[i..i + j]$ of the input string are the ones with e errors. This means that e symbols must either be replaced or deleted from $s[i..i + j]$. This string is derived by applying the rule $A \to BC$, where B and C derive the left and right part of k, respectively.

In a second step, the memoization table can then be used to build a corresponding string that is in the language of the grammar. This is done by applying the productions as they are stored in *tab*, starting at $tab[S, 1, n]$, and deleting or replacing a symbol when necessary.

3.2.1 Counting the errors

Algorithm 5 is the generalized bottom-up algorithm. The generalized algorithm requires an additional step for the initialization. The loop at Line 4 sets the bottom row of *tab* for each non-terminal variable. The value is either set to 1, if the variable has a terminal rule, or to n otherwise. The second initializing loop starts at Line 11 and resembles the one from the initial bottom-up algorithm, with the only difference that it sets the value of cells to 0 instead of *true*.

The algorithm then proceeds in a very similar manner as the original bottom-up algorithm. The only other difference is that instead of looking for a rule and splitting point for which both subproblems yield *true*, we iterate over all subproblems, picking the one with a minimal count of errors e. For this, we compare the value in *tab* with the sum of the value of the two subproblems (Line 21). If the new value is smaller, we memorize the new error value, as well as the configuration that leads to this error count.

If there is a way to transform the string with only replacements into a string of the language, then the algorithm will return the minimal amount of those operations that must be performed. It always chooses the production which leads to the string with the smallest amount of errors. When looking at substrings of length 2 and a non-terminal A, the algorithm iterates over all rules for A,

i.e., $A \to BC$. There is only one splitting point ($k = 1$), and the corresponding entries in *tab* can only have three different values: n, if the non-terminal has no terminal productions, 0, if the variable directly derives the symbol, or 1 if it does not. Out of all productions of A, the algorithm will choose the one where the sum of the two values for B and C are minimal.

If a non-terminal variable has no terminal rules, we call it a dead-end, as it can not derive any substring of length 1. Such variables have error count n for substrings of length 1 (Line 4). Since all cells are initialized with error count n, solutions leading to a dead end are ignored due to the comparison at Line 21. This is important, since these variables can not be used to replace a symbol.

When looking at longer substrings, the algorithm will choose the configurations, that use the former solutions with the smallest amount of errors. Thus, when the algorithm terminates, each cell will hold the configuration to reach the string closest to the input string in terms of errors.

We excluded deletions from this reasoning, since they are not always computed correctly by this algorithm. The value returned by the algorithm is however an upper bound on the sum of corrections and deletions that must be performed.

Algorithm 5 Bottom-Up CYK Parser with Error Count

```
 1: function BOTTOM-UP-ERROR(input string s[1..n])
 2:     allocate table tab[|V|][n][n] initialized with {n, null, null, null}
 3:     counter ← 0
 4:     for A ∈ V do
 5:         e ← n
 6:         if A has terminal rules then
 7:             e ← 1
 8:         for i ∈ {1, ..., n} do
 9:             tab[A, i, 1].e ← e
10:
11:     for i ∈ {1, ..., n} do
12:         for A : A → s[i] ∈ P do
13:             tab[A, i, 1].e ← 0
14:
15:     for j ∈ {2, ..., n} do                    – length of substring
16:         for i ∈ {1, ..., n − j + 1} do        – starting point of substring
17:             for (A → BC) ∈ P do               – All productions
18:                 for k ∈ {1, ..., j − 1} do    – all splitting points
19:                     counter ← counter + 1
20:                     e ← tab[B, i, k].e + tab[C, i + k + 1, j − k − 1].e
21:                     if e < tab[A, i, j].e then
22:                         tab[A, i, j] ← {e, B, C, k}
23:     return tab[S, 1, n]
```

The two initialization loops (Line 4 to 13)are both in $O(n)$, thus, the initialization is in $O(n)$. The nested loops starting at Line 15 are in $O(n^3)$, as the loops are the same as in the original bottom-up algorithm. The running time is thus in $O(n^3)$.

However, we expect this algorithm to be slower than the original bottom-up algorithm. In some cases, each splitting point may yield a slightly smaller error than the previous one, leading to the algorithm updating $tab[A, i, j]$ multiple times. This is more involved than simply setting a boolean value once *true* is found. Further, since we are no longer working with booleans, the cells for both non-terminals on the right hand side of the rule are accessed at each iteration. Thus, the algorithm accesses more cells and updates more objects, leading to a longer running time. Further, since an array of size four is stored in every cell, where the original algorithm only stored a boolean value, more memory is used for counting the errors.

3.2.2 Correcting the Input String

The information stored in *tab* is used to transform the input string into a word of the grammars language. Algorithm 6 resembles the top-down approach in some ways. The function `Correct-String` initializes two global variables: the `counter` holding the number of recursive calls, and `errors`, the number of corrected and deleted symbols. The detected errors are counted again, since Algorithm 5 only gives an upper bound. Algorithm 6 increases the variable whenever a symbol is deleted or replaced, and thus returns the actual number of necessary operations.

`Correct-String` calls `Correct` on the start variable S, the first position of the string (1) and its length n. The input string and *tab* are both global, so `Correct` can operate on them. `Correct` is a recursive function and takes non-terminal variable A, the starting point i and length j of a substring of the input string as parameters. The algorithm distinguishes four different cases:

1. **Correct:** If $tab[A, i, j.e] = 0$ then the substring is correct and can be returned directly (Line 8).

2. **Replacement:** If $j = 0$, then $s[i]$ is wrong and replaced with a symbol that can be derived by A, i.e. any $a \in \Sigma$ such that $A \to a \in P$. If A has no terminal rule (A is a dead end), the input string cannot be transformed into a word of the language (Line 11).

3. **Deletion:** If a lower error count can be reached by deleting either the first or last symbol of the current substring, then the symbol is deleted:

   a) Delete first symbol: if the error count of $tab[A, i+1, j-1]$ is smaller by more than one compared to $tab[A, i, j].e$, then implicitly delete $s[i]$ by returning `Correct(`$A, i+1, j-1$`)` (Line 16).

   b) Delete last symbol: if the error count of $tab[A, i, j-1]$ is smaller by more than one compared to $tab[A, i, j].e$, then implicitly delete $s[i+j]$ by returning `Correct(`$A, i, j-1$`)` (Line 20).

4. **Otherwise:** If none of the above hold, apply the rule stored in $tab[A, i, j]$ and concatenate the corrected version of the corresponding two substrings (Line 27).

Algorithm 6 Error Correction

```
1:  function CORRECT-STRING
2:      counter ← 0
3:      errors ← 0
4:      return CORRECT(S, 1, n)
5:  function CORRECT(non-terminal A, int i, int j)
6:      counter ← counter + 1
7:      if tab[A, i, j].e = 0 then
8:          return s[i, i + j + 1]                    – substring is correct
9:
10:     if j = 0 then
11:         return c : A → c ∈ R                      – replace terminal
12:
13:     first ← tab[A, i + 1, j − 1].e
14:     last ← tab[A, i, j − 1].e
15:     if first < last then
16:         if first < tab[A, i, j].e − 1 then        – delete first terminal
17:             error ← error +1
18:             return CORRECT(A, i + 1, j − 1)
19:     else
20:         if last < tab[A, i, j].e − 1 then         – delete last terminal
21:             error ← error +1
22:             return CORRECT(A, i, j − 1)
23:
24:     B ← tab[A, i, j].B
25:     C ← tab[A, i, j].C
26:     k ← tab[A, i, j].k
27:     return CORRECT(B, i, k) + CORRECT(C, i + k + 1, j − k − 1)
```

The running time of this algorithm depends heavily on the amount and position of the errors in the input string. If the errors are clustered, leaving long error-free substrings, the algorithm will run faster than if the errors are spread evenly.

The worst running time is obtained when no substring of length greater than 1 is correct and if further no symbol must be deleted. In this case, as long as the substring is not just one symbol, Line 27 is executed. This results in $2n$ recursive calls. There are two base cases: either the string is correct (Line 8), which in the worst case only occurs if the string has length one, or it has length one but is incorrect (Line 11). Both cases run in $O(1)$. Thus, the overall worst case is

in $O(n)$, since we perform a $O(1)$ operation $2n$ times. We exclude deletion of a symbol from this reasoning, since deleting a symbol leads to making only one recursive call instead of two.

In the following section, we evaluate all algorithms that were introduced in the first two sections.

4 Evaluation

In this section we show different experiments 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. All configurations were parsed at least 10 times. The time discussed in the following is the average running time of those runs, excluding the fastest and slowest run. While the running time may vary for runs of the same configuration, `counter` remains the same, as it is not influenced by external factors.

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, if not stated otherwise.

4.1 Grammars

4.1.1 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:

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

We 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.

We expect top-down to run faster on strings of the form `()..()`. The algorithm iterates over the productions 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 $S \to SS$ 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 17 in Algorithm 3) 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 derived, 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 the left substring can be derived and thus parses the right substring too. This yields 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.

Linear Form We will use this grammar to compare the non-specialized CYK algorithm to the algorithm adapted to grammars in linear form. Therefore, the productions of the grammar are transformed, such that the variables $L, R \in V$ can be omitted:

$$S \to (A$$
$$A \to)S|S)|)$$

This is an equivalent grammar for the Dyck language. When the parser transforms the grammar back to CNF, the resulting grammar has one non-terminal rule less than the initial grammar. This is, since two more non-terminal variables (one for each parenthesis) are added, such that the transformed set of productions will be very similar to the initial grammar. The parser is thus expected to yield slightly faster running times, if the linear grammar is transformed into CNF.

4.1.2 Strings Starting or Ending in a

These grammars contain all words with an arbitrary number of `a`'s and `b`'s in any order, but starting or ending in a, respectively. 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* are:

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

For both of these grammars, we run tests on strings of the form `ab..ab` and `ba..ba`. To analyse what running times are to be expected, we consider how *tab* is filled by the bottom-up algorithm. *tab* has size $3 \times n \times n$, since the grammar has three non-terminal variables. We look at the two dimensional table of each non-terminal in turn.

The table for A, tab_A, has the value *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 remainder of tab_A, the algorithm proceeds very fast.

The table for B, tab_B, has the value *true* in all cells. All substrings of length one can be derived, since B has the two terminal productions $B \to a|b$. Further, since B's 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 k brakes after trying the first splitting point and the cell is set to *true*. Thus, filling tab_B can be done in a fast manner, too.

We now look at tab_S. For *strings starting in a*, the only production for S is $S \to AB$, thus, the cell of tab_A is accessed first. 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 first symbol of 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. This will be repeated for all further k, because tab_A is *false* for all substrings with length greater than one. Thus, the more `b`'s the input string has, the bigger the running time will be, since for all substrings starting with a `b`, all possible k are considered.

For the grammar *ending in a*, the production for S is $S \to BA$, thus, for every splitting point, first tab_B is accessed, which will always return *true*. Then, tab_A is accessed, which will return *false* in all cases, where the right substring is not `a`. This implies that all k must be considered, since only for the last k, where the right substring has length one, tab_A may be true. Thus, all k are considered for all substrings, whereas for *starting in a*, the loop breaks after

the first k, if the substring can be derived. Further, 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*.

In conclusion, *ending in a* may perform more than twice the amount of table accesses than *starting in a*, and its running time is higher. We further expect the bottom-up algorithm to have very similar running times on any input string of the same length for *ending in a*, while the times for *starting in a* depend on the amount of `b`'s in the substring. Top-down is expected to behave very similarly. It will, however, be faster compared tp *starting in a* than bottom-up, since it looks at less subproblems.

4.1.3 ABC Grammar

The language of this grammar is more complicated to explain than the grammars before. The simplest grammar describing it is $G = (\{S\}, \{a, b, c\}, P, S)$, where P is:

$$S \to aSc|abSc|b$$

The words in the corresponding language $L(G)$ can be divided into three parts:

- The first part starts with an `a`, and consists of `a`'s and `b`'s, such that the number of `a`'s is greater than or equal to the number of `b`'s. There are never two consecutive `b` without an `a` in between.
- In between the two parts is a `b` (this `b` may follow right after a `b` of the first part).
- The third part consists only of `c`'s, the number of `c`'s is equal to the number of `a`'s in the first part.

We will use this grammar only to evaluate the specialization, therefore we transform it to the equivalent linear grammar $G = (\{A, B, S\}, \{a, b, c\}, P, S)$ with the productions:

$$\begin{aligned} S &\to Ac|b \\ A &\to aS|aB \\ B &\to bS \end{aligned}$$

The strings we will test are of the form `abab..b..cc`, `aa..b..cc` and the same with an additional `a` and `c` in the front or end respectively.

4.2 Evaluation of the conventional Implementation

We now show the results of experiments, using the grammars described, and analyze whether the algorithms behave as we expect them to behave. For these tests, the conventional algorithms as described in Section 2.2.4 and 2.2.3 where used.

4.2.1 Dyck Language

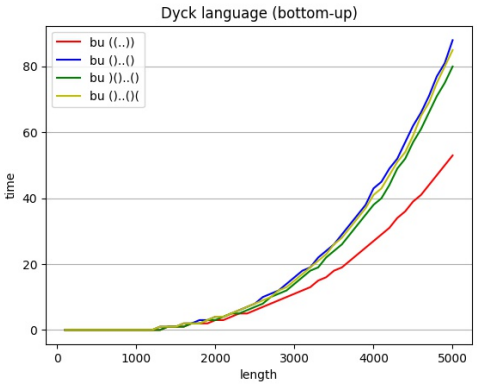


Figure 2: Running time (s) of the bottom-up algorithm when parsing different set of strings of sizes 100-5000, in steps of hundred, for the *Dyck language*.

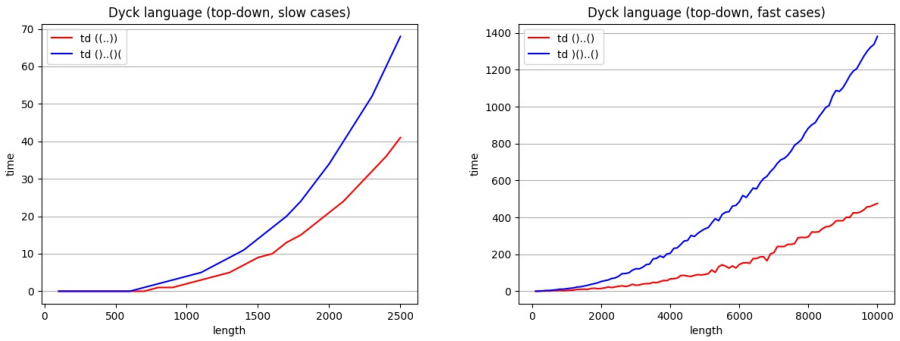
Figure Figure 2 shows the running times of the bottom-up parser on different sets of input strings for the grammar *Dyck language*. The strings were of length 100 to 5000, growing in steps of 100. As expected, the times are very similar for three of the four different sets of input strings. However, parsing the strings of the form `((..))`is a little faster. When looking at how the *tab* is filled, we see that tab_A is very similar in both cases. ta_S on the other hand has *true* in more cells for strings of the form `()..()` than `((..))`. As we argued before, the loop over k breaks, when both subproblems are true, which makes the parser more efficient. Thus, more *true* entries intuitively lead to faster running times, as the loop is executed less often. For strings of the form `()..()`, this is not the case. The additional *true* in ta_S lead to a lot more table accesses, when the rule $S \rightarrow SS$ is considered. Therefore strings of the form `((..))` can be parsed faster by the bottom-up algorithm.

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

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



(a) (b) R

Figure 3: Running times in (a) seconds and (b) milliseconds for the top-down algorithm when parsing four sets of strings of sizes (a) 100-2500 and (b) 100-1000, in steps of hundred, for the *Dyck language*.

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.

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 4. 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.

4.2.2 Strings starting and ending in a

As expected, both the top-down and bottom-up algorithm performed very differently on the two grammars, being rather 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.

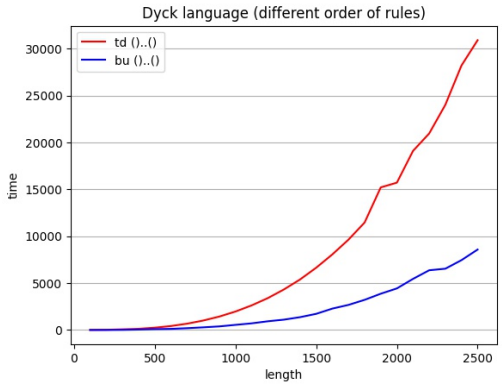


Figure 4: Running time (ms) of both algorithms for parsing a set of strings of sizes 100-2500, in steps of hundred, for the *Dyck language*, with a different order of the productions.

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

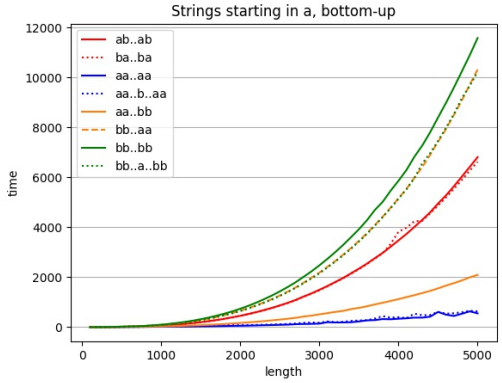
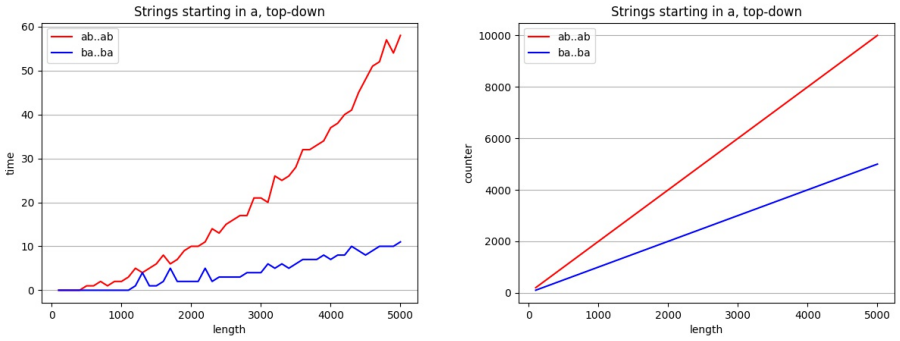


Figure 5: Running time (ms) of the bottom-up algorithm when parsing sets of strings of sizes 100-2500, in steps of hundred, for the language of words starting in a.

Figure 5 shows, that the running times of bottom-up indeed depends on the amount of `b`'s in the input string. We see, that strings of the form `bb..bb` have the slowest running times, while strings of the form `aa..aa` are parsed very fast. While adding an `a` to the first type does influence the running time, adding a single `b` to the second one has almost no influence. Further, for strings with alternating letters (the red lines) the running time does not change if the string

starts with `a` or `b`. However, if the `a`'s and `b`'s are continuously (yellow lines), the strings are parsed a lot faster if the `a`'s are first.



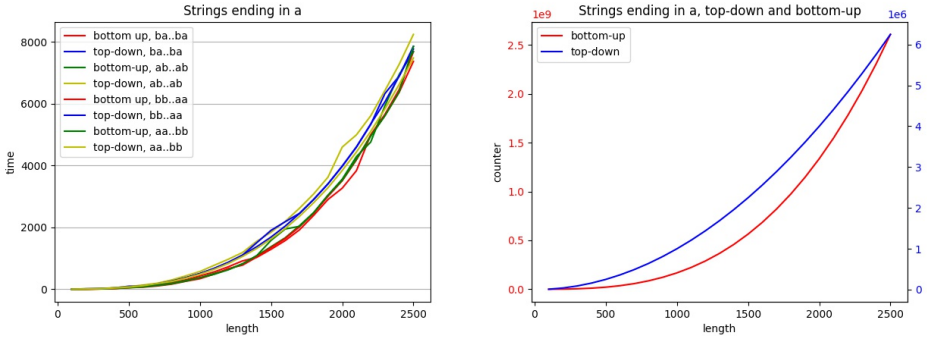
(a) (b)

Figure 6: Sunning time in ms (a) and counter (b) of the top-down algorithm when parsing sets of strings of sizes 100-5000, in steps of hundred, for the Language of words starting in a.

Figure 6a shows the incredibly low running times of top-down on the grammar *strings starting in a*. The times are so low, since the algorithm 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 as can be seen in Figure 6b. The number of calls on the recursive function is $\Theta(n)$ for parsing strings not starting in a for the grammar *starting in a*. As we argued in Section 2.2.4, the upper bound for this number is $O(n^3)$. The numbers for `counter` of bottom-up (repetitions of the inner most loop, i.e. over splitting points k) for parsing strings for *starting in a*, are somewhere in between of n^2 and n^3, still yielding relatively fast running times.

The three bumps in the running time of top-down on strings of the form `ba..ba`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 of strings!), though not as distinctive as in the blue line, since the running times are already a little higher at this point. The bumps could be reduced by running the parser 40 times per input string instead of 10 times which supports this theory.

Figure 7a and Figure 7b show, 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*. We further see, that all of them have a very similar shape. In fact, both the bottom-up and the top-down parser have the same `counter` respectively for any string of the same length, regardless of the order of the letters. As can be seen in Figure 7b, for a string of length 100, bottom-up



(a) (b)

Figure 7: Running time in ms (a) and counter (b) of the bottom-up and top-down algorithm when parsing sets of strings of sizes 100-2500, in steps of hundred, for the language of words ending in a.

iterates 171600 times of its most inner loop while top-down calls itself recursively 9901 times. Despite this big difference of `counter`, the running times are almost identical for both algorithms. This is due to the more efficient manner of bottom-up. One aspect is the more structured way to allocate the cells of *tab*, the second one the fact that Java can not perform recursive functions as efficient as iterative ones.

4.3 Evaluation of the Specialization with Linear Grammars

To test the specialization for linear grammars, we parse strings by using the grammar parser, which transforms the linear grammar provided to CNF. Further, we compare the running times of parsing strings with the transformed grammar to parsing them with the specialized bottom-up algorithm.

4.3.1 Grammar Transformation

In order to test how parsing the grammar from a linear grammar to on in CNF, we use the linear form of the *Dyck language* as introduced in Section 4.1.1. The comparison of those results to the results of the evaluation of the initial *Dyck language* can be seen in Figure 8. As expected, strings could be parsed slightly faster for the transformed grammar than for the initial form. The plot further shows, that the linear bottom-up algorithm is a lot faster than the CNF one, and that `counter` is perfectly asymptotical to the running time.

The CNF-configuration refers to the bottom-up parser run on the initial grammar, time and `counter` are the same as in Section 4.2.1. The second configuration used the linear transformation of the *Dyck language*, but transformed it to CNF before parsing the strings. This results in one non-terminal rule less,

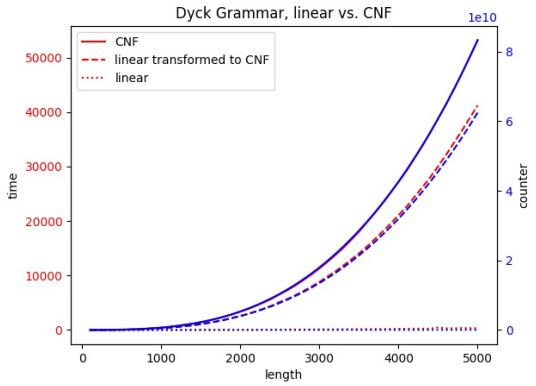
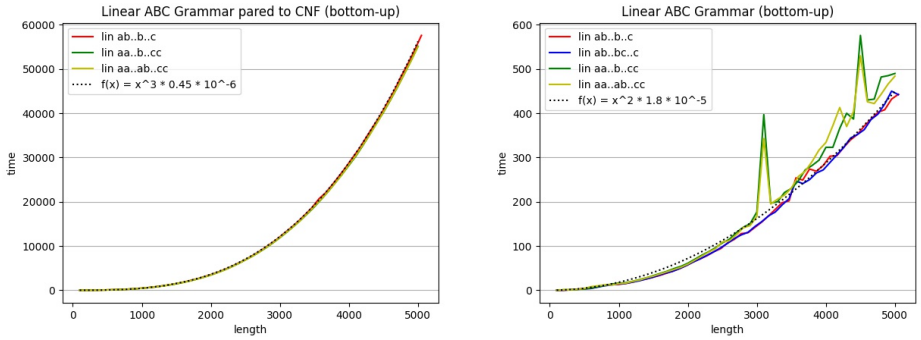


Figure 8: Running time (ms) and counter of the bottom-up algorithm when parsing strings of the form `((..))` for the Dyck language. *CNF* is the running time of parsing with the productions of the initial grammar, *linear transformed to cnf* the one for the linear productions, but transformed to CNF by the parser, and *linear* the running times when the specialized CYK algorithm was used.

which is why the parser is slightly faster. The third configuration parses strings for the linear grammar using the specialized algorithm. It is a lot faster then both of the other configurations. To analyse the difference between the linear and CNF parser better, we consult the experiments run on the *abc grammar*.

4.3.2 Adapted algorithm



(a) (b)

Figure 9: Running time in ms of the bottom-up algorithm when parsing strings for a linear grammar (a) by transforming the grammar to CNF and (b) by using the specialized algorithm. Both plots contain a function of the length as reference.

Figure 9a shows the running time which is yielded when parsing strings for the *abc grammar* by transforming the productions to CNF. Since we used the bottom-up parser we expected the times to be very similar for all sets of input strings, which is the case. The graph also contains a function over the length of the input strings, $f(x) = x^3 * 0.45 * 10^-6$, which shows that the running time indeed is in $O(n^3)$. The specialized algorithm could parse the strings in a tenth of the time, the cnf parser needed. This is partly due to the smaller amount of rules, but also because only one splitting point per rule has to be considered. Here, the running times are very similar for all types of input strings, too. The variations between the different types can be seen better, as the numbers are a lot smaller. Like in Figure 3b, some configurations have bumps. We explain them with the same reasoning, assuming they appear only because the algorithm is very fast. The graph, too, contains a function over the length of the input strings, $f(x) = x^2 * 1.8 * 10^-5$,showing that the running time is in $O(n^2)$, as expected.

4.4 Evaluation of the Generalization with Error Correction

We tested the error counting bottom-up algorithm on the *Dyck language*. Figure 10 shows that the running time of the algorithm barely correlates with the form of the input string.

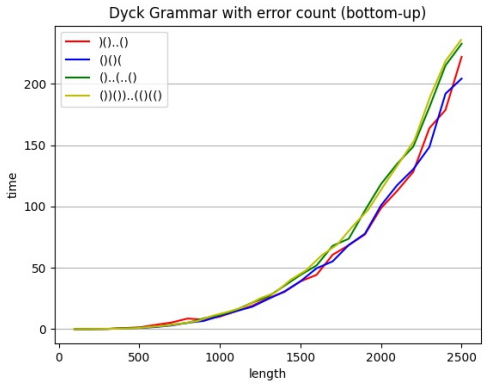


Figure 10: Running time (s) of the bottom-up algorithm with error count when parsing strings of different forms of length 100 to 2500 for the Dyck language.

The used strings are the same as in Section 4.2.1, with two additional sets containing different errors, namely strings of the form `().(..()` and `()).())(().(()`. The first one is correct apart from the single opening parenthesis in the middle, while the number of errors in the second one is linear in the length of the string. Figure 11 shows the number of recursive calls of `Correct-String` that are needed to correct them, respectively. The `counter`

for the other sets of input strings was constant, i.e., two for all strings. Hence, correcting strings with errors at the beginning or the ending of the strings is less expensive than correcting errors in the middle of the string. Further, correcting more errors needs more calls. The plot also shows that the number of recursive calls is indeed linear in the length of the input string.

The running times of the error correction are more interesting to analyse. The two linear configurations take less than two milliseconds to correct regardless of the length. Correcting the first string (length 100) of the two sets of test strings shown in Figure 11 takes about 20 milliseconds each. However, correcting the following strings takes less than two milliseconds as well. This shows that the Java Runtime Environment can process an algorithm faster when it operates on similar input multiple times in a row. Thus, the longer substrings are corrected very fast too, despite their higher counter.

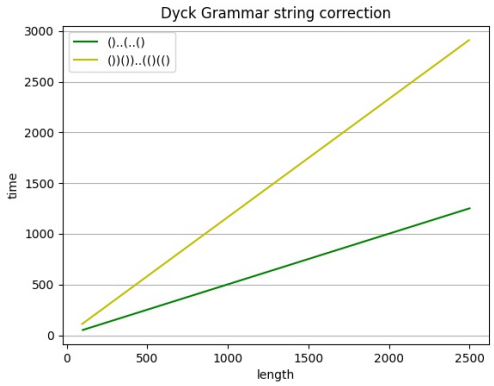


Figure 11: Counter of the bottom-up algorithm with error count when parsing strings of two different forms containing errors of length 100 to 2500 for the Dyck language.

Comparing the running times of the error counting algorithm to the conventional one shows that the generalized one is a lot slower. In order to compare the two algorithms, we parse error-free strings with the generalized algorithm. Figure 12 shows the difference between the algorithms. Even though they both are in $O(n^3)$, the running times are very different. Comparing integers is thus more involved than looking for boolean values. Further, accessing two cells of *tab* and assigning multiple values if a smaller error count is found both enhance the running time as well. Each of those factors alone would not increase the running time significantly, but all three of them together have a noticeable impact.

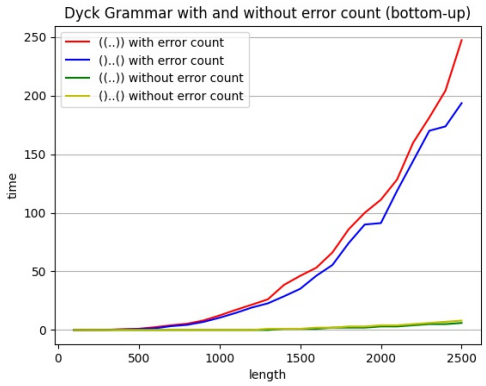


Figure 12: Running time (s) of the bottom-up algorithm with and without error count when parsing strings of two different forms of length 100 to 2500 for the Dyck language.

5 Conclusion

Evaluating the non-specialized bottom-up and top-down algorithms showed multiple things. Firstly, bottom-up has steadier running times than top-down. This means, that even though the times may vary for different strings of the same length for the same grammar, they do not vary as much as for the top-down parser. Further, it's worst case times are better, than those of top-down. However, if the top-down parser is run on a favorable combination of a good ordering of the rules and form of input string, its running time may be very low. Thus, its best-case running time is a lot lower than that of bottom-up.

The analysis further showed, that the running times could often be explained, by analyzing how the memoization table is filled and how many accesses to the table are performed in order to solve the problem. Thus, if the input string is known, the faster algorithm can be detected with reasoning.

In general, it is safe to say, that both algorithms perform better if the grammar produces words with a distinctive property in the beginning, rather than in the end. Parsing strings for the grammar of strings starting in yielded different running times for different input strings of the same length. Contrarily, for the grammar of strings ending in a, both the top-down and bottom-up parser yielded the same running times for any input string of same length, being slower than for any input string on the grammar for strings starting in a.

Transforming a linear grammar to an equivalent grammar in CNF can easily be done. However, it does not improve the running time compared to parsing an equivalent grammar which is already in CNF. Specializing the bottom-up algorithm on the other hand improves the running time a lot, as the algorithm is not in $O(n^3)$ anymore, but in $O(n^2)$. It would be interesting to analyze, how the specialized algorithm performs on different grammars. We could transform the

grammars for strings starting and ending in a to equivalent linear grammars, and verify whether it still holds, that the running times are better if the distinctive property is in the beginning.

The bottom-up CYK algorithm can also be generalized to return the number of errors in the input string, rather than merely the truth value. Despite the fact that the it is also in $O(n^3)$, the generalized algorithm is a lot slower than the original bottom-up algorithm. Correcting the input string using the memoization table on the other hand is very fast.

References

[1] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, ser. Addison-Wesley Series in Computer Science. Addison-Wesley Publishing Company, 1979, ISBN: 0-201-02988-X.

[2] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*. MIT press, 2009.