Chomsky-Grammatik zur Wachstumsmodellierung auf Basis einer modifizierten Fibonacci-Folge

Lennart Köhler

12.07.2023

Betreut von PD Dr. Thomas Hinze

Molekulare Algorithmen

Friedrich-Schiller-Universität Jena

1 Einleitung

DNA (**D**esoxyribo**N**ucleic **A**cid) ist ein Molekül, welches in vielen Organismen die Erbinformation enthält. Es handelt sich um ein sehr robustes Molekül, welches trotzdem Operationen wie Synthese, Hybridisierung, Polymerisation, Sequenzierung etc. erlaubt [1]. Im Menschen wird hier der Bauplan für die Zellen gespeichert.
DNA-Computing ist ein Rechenkonzept, welches auf den DNA-Molekülen basiert. Zurückzuführen ist die Idee auf Leonard Adleman von der Univsersity of Southern California. 1994 hat er das erste Experiment zur Lösung des Hamiltonpfadproblems basierend auf DNA-Computing durchgeführt [2]. Es gelang Wissenschaftlern im Jahr 2004 durch selbstassemblierende DNA das Verhalten eines zellulären Automaten zu simulieren [3]. Es stellt sich heraus, dass es sich bei DNA-Computing um Universalrechner handelt [1]. Zudem ist DNA ein sehr robustes und kompaktes Speichermedium [1, 4]. Bei DNA-Computing wird auch die parallele Verarbeitung der Moleküle ausgenutzt. Jedes DNA-Molekül kann selbst eine Reaktion eingehen. Die Operationen eines DNA-Computers kommen der von Chomsky-Grammatiken sehr nahe [1]. Aus diesen Grund wird diese Grammatik im folgenden näher erläutert. In dieser Projektarbeit wurde eine modifizierte Fibonacci-Folge durch eine Chomsky-Grammatik berechnet. Die entstandene Chomsky-Grammatik wird in dem Abschnitt 2 beschrieben. Darauf folgt eine Komplexitätsabschätzung dieser Grammatik.

1.1 Chomsky-Grammatiken

Die Chomsky-Grammatik, benannt nach dem amerikanischen Linguisten Noam Chomsky, ist eine formale Methode, um die Struktur von Sprachen zu beschreiben. Eine Chomsky-Grammatik nutzt Regeln um Symbole abzuleiten. Da die Regeln (auch Pro-

duktionen) den Übergang von Symbolen klar definieren, kann die Berechnung der Folgesymbole parallel durchgeführt werden. Neben der Regelmenge P besteht eine Chomsky-Grammatik noch aus dem Alphabet der Terminalsymbole Σ, dem Alphabet der Variablensymbole V und dem Startsymbol S. Im folgenden werden die formalen Definitionen aufgeführt.

V (Variablensymbole) - eine endliche Menge von Symbolen

Σ (Terminalsymbole) - eine endliche Menge von Symbolen, wobei $V \cap \Sigma = \emptyset$

S (Startsymbol) - ein Element aus V, das Startsymbol

P (Regelmenge) - eine Teilmenge aus $((V \cup \Sigma)^* \otimes V \otimes (V \cup \Sigma)^*) \times (V \cup \Sigma)^*$

Eine Chomsky-Grammatik wird durch diese vier Mengen definiert und wird deswegen als Tupel $G = (V, \Sigma, P, S)$ geschrieben. Bei einem Alphabet handelt es sich um eine nichtleere endliche Menge von Symbolen (Zeichen). Ein Wort über einem Alphabet ist eine endliche Folge von Symbolen aus dem Alphabet. [1]

1.2 Fibonacci-Folge

Die Fibonacci-Folge ist eine Folge von Zahlen, bei der jede Zahl rekursiv durch die vorherigen Zahlen definiert ist. Bei der normalen Fibonacci-Folge ist jede Zahl die Summe der beiden vorherigen Zahlen mit den beiden Startzahlen 0 und 1 (siehe Gleichung (1)).

$$f(n) = \begin{cases} 0 & \text{für} \quad n = 0 \\ 1 & \text{für} \quad n = 1 \\ f(n-1) + f(n-2) & \text{für} \quad n \geq 2 \end{cases} \tag{1}$$

In diesem Projekt wurde eine etwas veränderte Fibonacci-Folge behandelt, welche sich durch eine verlangsamte Wachstumskurve auszeichnet. Jede Zahl $f(n)$ wird durch die Summe von $f(n-1)$ und $f(n-3)$ berechnet (siehe Gleichung (2)). Der Tabelle 2 können die ersten 13 Werte der Folge entnommen werden, welche durch die Funktion (2) entstehen.

$$f(n) = \begin{cases} 0 & \text{für} \quad n = 0, n = 1 \\ 1 & \text{für} \quad n = 2 \\ f(n-1) + f(n-3) & \text{für} \quad n \geq 3 \end{cases} \tag{2}$$

n	0	1	2	3	4	5	6	7	8	9	10	11	12
f(n)	0	0	1	1	1	2	3	4	6	9	13	19	28

Tabelle 1: Tabellarische Darstellung der ersten 13 Werte der modifizierten Fibonacci Folge.

2 Ergebnisse

Um die Chomsky-Grammatik zu entwerfen, die die modifizierte Fibonacci-Folge (2) simuliert, sollten sich die Anforderungen an diese klar gemacht werden. Die Chomsky-Grammatik muss einen internen Speicher für die letzten drei Zeitschritte haben. Es muss die Summe von dem Zeitschritt $f(n-1)$ und $f(n-3)$ berechenbar sein. Zuletzt müssen diese Kriterien in eine klar definierte Ableitungsstruktur eingebettet werden.
In (3) werden die Regeln aufgelistet, welche genutzt werden um die modifizierte Fibonacci-Folge zu reproduzieren. Die Ausgabewerte werden durch Wörter des Terminalsymbols a repräsentiert, wobei die Länge des Wortes dem Ergebniswert entspricht (z.B. $f(5) = \text{"}aa\text{"} \equiv 2$). Die Regeln wurden nach Symbolen gruppiert und deren Funktion beschrieben. Es ist sinnvoll sich die Regeln parallel zu dem Ablauf (4) der Ableitungsregeln anzuschauen. Im Folgenden wird die Funktionsweise der Grammatik interpretiert.

$$
\begin{array}{rrr}
\text{Start}: & S \to & \varepsilon \\
 & S \to & LAYR \\
 & & \\
\text{Start der Termination}: & YR \to & Z \\
\text{Nächste Iteration beginnen}: & YR \to & XR \\
 & & \\
X: & AX \to & XAB \\
 & BX \to & XC \\
 & CX \to & XA \\
 & & \\
Y: & YA \to & AY \\
 & YB \to & BY \\
 & YC \to & CY \\
 & & \\
L: & LX \to & LY \\
 & & \\
\text{Termination}: & AZ \to & Za \\
 & BZ \to & Z \\
 & CZ \to & Z \\
 & LZ \to & \epsilon
\end{array}
\tag{3}
$$

S (Startsymbol) kann zu ε, und damit zum ersten "leeren" Ergebnis, oder dem eigentlichen Beginn des Ableitungsbaums $LAYR$ abgeleitet werden. L und R beschränken das Wort nach links und rechts. YR entspricht dem einzigen Wort, welches zwei Ableitungsregeln besitzt. Der erste Weg, das Z, leitet die Termination des Wortes ein. Z löscht alle Variablen außer A, welches zu a überführt wird. Wenn alle Variablen bis auf $LZaa...$ gelöscht wurden, so wird LZ zu ε und damit resultiert ein Wort, welches nur aus dem Terminalsymbol a besteht und der Länge des Ergebniswertes entspricht. Wird nicht die Termination eingeleitet, sondern YR wird zu XR, so ist das als Beginn der nächsten Iteration zu interpretieren. X und Y fungieren hier als Laufvariablen,

welche sich zwischen L und R bewegen. Nur in der Umgebung von X und Y (und ggf. Z) werden Veränderungen vorgenommen, sie dienen also als Lese-/Schreibkopf. X läuft von rechts nach links und transformiert immer das übersprungene Symbol. A speichert den Wert von $f(n-1)$, B den Wert von$f(n-2)$ und C den Wert von $f(n-3)$. Läuft X über A so wird es zum einen in B überführt und somit "gespeichert", aber auch für die Summenbildung als $f(n-1)$ erhalten. B wird einen Zeitschritt nach hinten verschoben und als C gespeichert. C wird jetzt als $f(n-3)$ benötigt um mit $f(n-1)$ addiert zu werden und wird deswegen als A abgeleitet. Die Anzahl von As entspricht nach Durchlaufen der Iteration also dem neuen Wert.

2.1 Ableitungsbaum der ersten fünf Werte welche durch die Chomsky-Grammatik erzeugt werden

$$
\begin{array}{l}
\boldsymbol{S} \to \varepsilon \\
\downarrow \\
LA\boldsymbol{YR} \to L\boldsymbol{AZ} \to \boldsymbol{LZ}a \to a \\
\downarrow \\
L\boldsymbol{AX}R \\
\downarrow \\
\boldsymbol{LX}ABR \\
\downarrow \\
L\boldsymbol{YA}BR \\
\downarrow \\
LA\boldsymbol{YB}R \\
\downarrow \\
LAB\boldsymbol{YR} \to LA\boldsymbol{BZ} \to L\boldsymbol{AZ} \to \boldsymbol{LZ}a \to a \\
\downarrow \\
LA\boldsymbol{BX}R \\
\downarrow \\
L\boldsymbol{AX}CR \\
\downarrow \\
\boldsymbol{LX}ABCR \\
\downarrow \\
L\boldsymbol{YA}BCR \\
\downarrow \\
LA\boldsymbol{YB}CR \\
\downarrow \\
LAB\boldsymbol{YC}R \\
\downarrow \\
LABC\boldsymbol{YR} \to LAB\boldsymbol{CZ} \to LA\boldsymbol{BZ} \to L\boldsymbol{AZ} \to \boldsymbol{LZ}a \to a \\
\downarrow \\
LAB\boldsymbol{CX}R \\
\downarrow \\
LA\boldsymbol{BX}AR \\
\downarrow \\
L\boldsymbol{AX}CAR \\
\downarrow \\
\boldsymbol{LX}ABCAR \\
\downarrow \\
\ldots \\
\downarrow \\
LABCA\boldsymbol{YR} \to LABC\boldsymbol{AZ} \to LAB\boldsymbol{CZ}a \to LA\boldsymbol{BZ}a \to L\boldsymbol{AZ}a \to \boldsymbol{LZ}aa \to aa \\
\downarrow \\
\ldots
\end{array}
\tag{4}
$$

Der Ableitungsbaum (4) zeigt die einzelnen, chronologisch angeordneten, Ableitungsschritte der Chomsky-Grammatik. Die **fett** markierten Variablen sind solche, welche in dem entsprechenden Zeitschritt abgeleitet werden. Beginnend oben, bei S, wird der Ableitungsbaum nach unten gelesen. Es wird pro Zeitschritt (Pfeil) eine Ableitungsregel angewendet. Mit "..." sind größere Sprünge, also mehrere Ableitungsschritte, gekennzeichnet.

2.2 Komplexitätsschätzung

Für die Komplexitätsschätzung wurde eine numerische Methode verwendet. Durch die Simulation der Chomsky-Grammatik in einem *Python* (v3.7.9) Skript (Abschnitt 3) konnten die Ableitungsschritte für die ersten 16 Werte der modifizierten Fibonacci-Folge berechnet werden. Das Ergebnis, zusammen mit einer Fit-Funktion, wurde in Abbildung 1 abgebildet. Die Fit-Funktion wurde mithilfe von *scipy.optimize.curve_fit* (v1.7.3) erstellt. Hierbei handelt es sich um eine Funktion, welche eine nicht-lineare Methode der kleinsten Quadrate verwendet um die Parameter einer gegebenen Funktion den beobachteten Daten anzupassen.
Die klassische Fibonacci-Folge (1) - als rekursive Funktion, welche zwei Eingangsparameter hat - wird durch $\mathcal{O}(1.62^n)$ abgeschätzt [5]. Es wurde daher für die Schätzung die Funktion $f(n) = a \cdot b^n$ vorgegeben. Da für die Komplexitätsschätzung Vorfaktoren irrelevant sind, ist hier der Parameter b von Interesse.

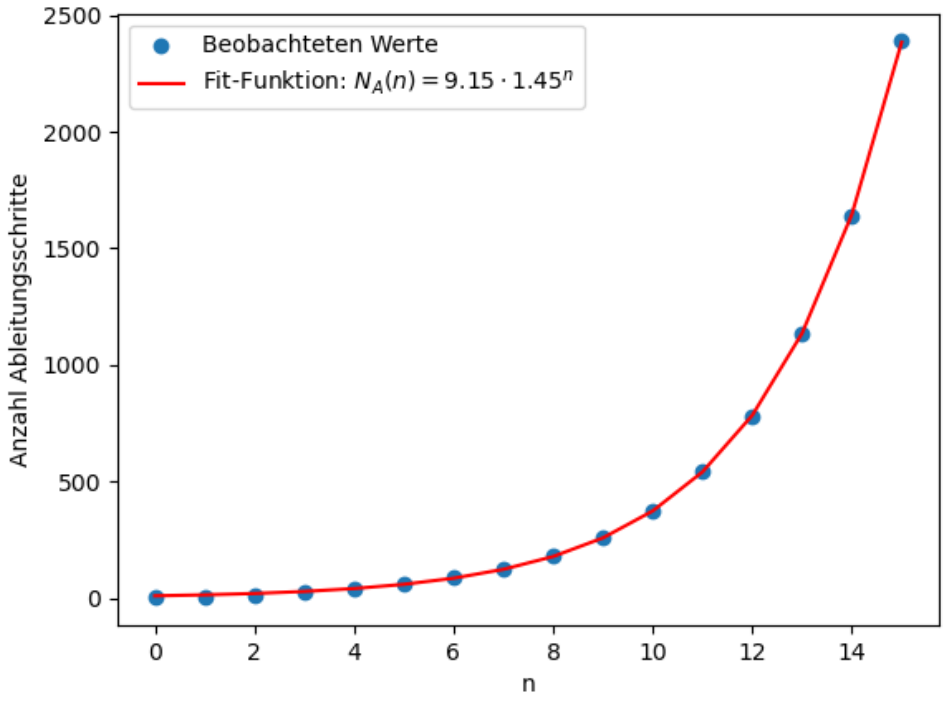

Abbildung 1: Anzahl der Ableitungsschritte, die benötigt werden um das n'te Glied der Folge zu berechnen. Zusätzlich zu den beobachteten Werten (in blau) wurde eine Fit-Funktion für die Werte berechnet und, mit dessen Beschreibung in der Legende, abgebildet (in rot).

Der Parameter b wird auf 1.45 geschätzt (siehe Abbildung 1). Das bedeutet die

Laufzeit der modifizierten Fibonacci-Folge (2) für $n \in \mathbf{N}$ ist $\mathcal{O}(1.45^n)$. Dieser Wert entspricht auch dem Wachstum der modifizierten Fibonacci-Folge: $f(n) = f(n-1) \cdot 1.45$. Das bedeutet die Komplexität der Chomsky-Grammatik ist nur abhängig von dem n. In den folgenden Stichpunkten sind die Berechnungen, die nötig sind um von dem Wort f_n das Wort f_{n+1} zu generieren.

1. $YR \to XR : 1\times$
2. $YR \to Z : 1\times$
3. $V_1X \to XV_2 : (|f_n| - 3)\times$ mit $(V_1, V_2)\epsilon(A, AB), (B, C), (C, A)$
4. $W_1Y \to YW_2 : (|f_n| - 3 + \mathbf{1}(f_{ni} = A))\times$ mit $(W_1, W_2)\epsilon(A, A), (B, B), (C, C)$
5. $K_1Z \to ZK_2 : (|f_n| - 1 + \mathbf{1}(f_{ni} = A))\times$ mit $(K_1, K_2)\epsilon(A, a), (B, \varepsilon), (C, \varepsilon)$
6. $LZ \to \varepsilon : 1\times$

Bei dieser Liste der Ableitungen, die durchgeführt werden müssen um ein Folgeglied zu berechnen, sind besonders die Punkte 4. und 5. von Interesse. In 4. wird beschrieben wie oft Y einen Buchstaben (A, B, C) überspringt. Das kommt so oft vor wie im letzten Folgeglied plus die Anzahl der As, da diese zu AB wurden. Für jedes A, was in der letzten Iteration durchlaufen werden musste, muss jetzt ein A und B durchlaufen werden. Das selbe gilt für die Ableitungen unter 5., hier muss auch jeder Buchstabe von Z abgebaut werden. Die -3 unter 2. und 3. repräsentiert die L, X, R bzw. L, Y, R, welche nicht durchlaufen werden müssen. Die -1 unter 5. entspricht der Ableitung, welche unter 6. notiert ist.
Das bedeutet die Berechnung jedes Folgeglieds f_{n+1} ist proportional zu der Anzahl an Berechnungen für f_n plus die As in f_n. Die As entsprechen aber auch dem Wert des Glieds. Natürlich haben die Cs in f_n einen Einfluss auf den Wert von f_{n+1}, verursachen aber keine höhere Komplexität, da sie zeitlich versetzt im nächsten Zeitschritt als A und damit mit einem zusätzlichen Ableitungsschritt verknüpft sind. Dies hat nur einen Einfluss auf den Faktor der Gleichung, nicht auf die Basis der Exponentialfunktion. Die Werte $f(n)$ beziehungsweise die Länge des Terminalworts wurden in Abbildung 2 dargestellt. Damit ist der Zusammenhang zwischen der Komplexität der Chomsky-Grammatik und den Werten der modifizierten Fibonacci-Folge gezeigt.

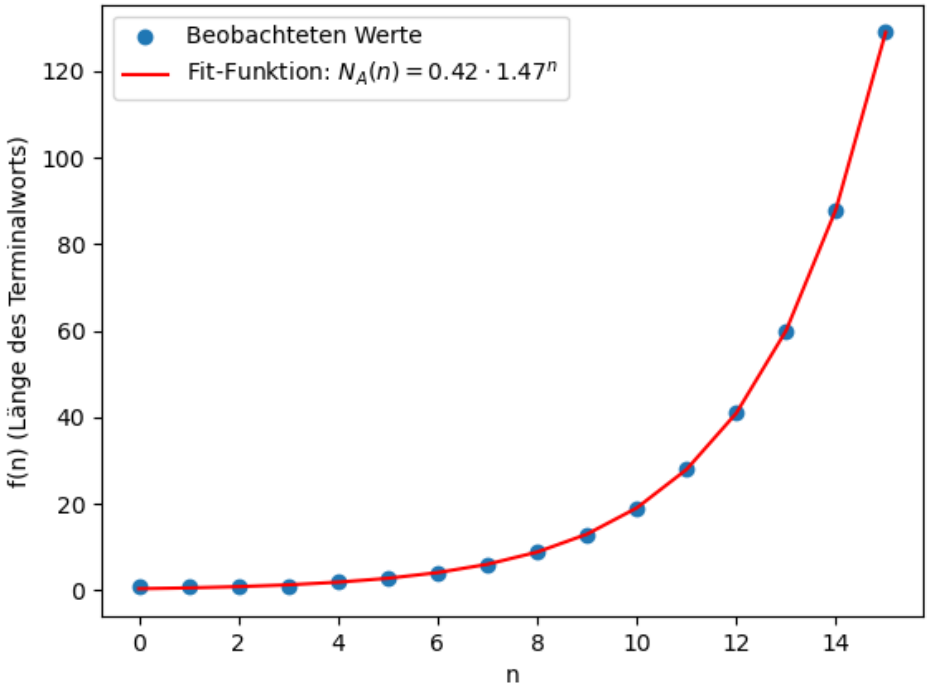

Abbildung 2: Werte f(n) bzw. Länge des Terminalworts des n'ten Gliedes der Folge. Zusätzlich zu den beobachteten Werten (in blau) wurde eine Fit-Funktion für die Werte berechnet und, mit dessen Beschreibung in der Legende, abgebildet (in rot).

3 Python-Skript

Listing 1: Python-Skript der Simulation und Berechnung der Chomsky-Grammatik

```
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit
import sys
print(sys.setrecursionlimit(3000))


grammar={"S":"LAYR","YR":"XR","AX":"XAB","BX":"XC","CX":"XA","YA":"AY",
        "YB":"BY","YC":"CY","LX":"LY"}
#keine Terminalsymbole, abzweigungen sind nicht implementiert

def countA(x): #Funktion welche die Anzahl der ''A'' in string x berechnet
    c=0
    for a in x:
        if a=="A":
            c+=1
    return c

def func(n,a,b): #die Parameter a und b werden geschaetzt, n ist gegeben
        return b*a**n

#Funktion welche Woerter der Chomsky-Grammatik generiert:
def generate_sentence(sentence, grammar, max_i,counter, i=1):
    print(sentence)

    if "YR" in sentence:
    #existiert das Wort "YR" so wird im Anschluss ein Terminalwort
    #generiert und eine neue Iteration gestartet
        i+=len(sentence)-1 #Addition der Ableitungen der Termination
        #counter.append(countA(sentence))
        counter.append(i)

    if i>max_i:
        return

    change=False

    for key,val in grammar.items():
        #print(key,val)

        if not change and key in sentence:
            sentence=sentence.replace(key,val)
            change=True


    generate_sentence(sentence, grammar, max_i,counter,i+1)

    return sentence,counter

#counter speichert die Ableitungsschritte (i) die durchgefuehrt wurden
#bis ein Terminalwort entsteht
result,counter=generate_sentence("S",grammar,2500,[1])
#2500 Rekursionsschritte
```

```
values=list(range(0,len(counter))) #x-Werte
s=curve_fit(func,values,counter)[0] #Fit-Funktion

#plotting
plt.scatter(values,counter,label="Beobachteten Werte")

plt.plot(values,func(values,s[0],s[1]),color="red",
        label=f"Fit-Funktion: $N_A(n)={s[1]:.2f}\cdot{s[0]:.2f}^n$")
plt.legend()
plt.xlabel("n")
plt.ylabel("Anzahl Ableitungsschritte")
#plt.ylabel("Anzahl Ableitungsschritte")
```

Literatur

[1] Thomas Hinze. Molekulare algorithmen - dna-computing. Vorlesungsfolien, 2023.

[2] Leonard M. Adleman. Molecular computation of solutions to combinatorial problems. *Science*, 266(5187):1021–1024, 1994.

[3] Paul W. Rothemund, Nick Papadakis, and Erik Winfree. Algorithmic self-assembly of dna sierpinski triangles. *PLoS Biology*, 2(12), 2004.

[4] Leonard M. Adleman. Computing with dna. *Scientific American*, 279(2):54–61, 1998.

[5] Margarita Esponda. Analyse von algorithmen - die o-notation. Vorlesungsfolien, 2012.