FRIEDRICH-SCHILLER-
UNIVERSITÄT
JENA

Chomsky-Grammatik zur Erzeugung einer überexponentiell wachsenden Zahlenfolge

Friedrich-Schiller-Universität Jena
Fakultät für Mathematik und Informatik
Molekulare Algorithmen SS 2025

Arne Konrad, Anshul Khubchandani

16. August 2025

Inhaltsverzeichnis

1 Einleitung

Die Erzeugung von stark wachsenden Zahlenfolgen stellt ein interessantes Problem in der formalen Sprachtheorie dar – insbesondere dann, wenn deren Länge nicht nur exponentiell, sondern überexponentiell anwächst. Die vorliegende Arbeit beschäftigt sich mit der Entwicklung einer Chomsky-Grammatik, die genau diese Herausforderung bewältigt: die Generierung von Wörtern der Form n^n für $n \geq 1$.

Im Folgenden wird die Konstruktion einer Grammatik zur Erzeugung der Sprache $L(G) = \{a^{n^n} \mid n \in \mathbb{N},\ n \geq 1\}$ behandelt. Diese Sprache enthält genau ein Wort für jedes n, dessen Länge n^n beträgt, beispielsweise $a^1, a^4, a^{27}, a^{256}, \ldots$. Der Fokus liegt dabei auf einem modularen Aufbau der Grammatik, bei dem jeder Iterationsschritt eine neue Potenz n^n erzeugt, ohne dass eine externe Zähl- oder Rechenlogik erforderlich ist. Die verwendeten Transformationen erfolgen ausschließlich durch lokale Regelanwendungen, wodurch sich ein symbolischer Prozess mit deterministischer Struktur ergibt.

Formale Grammatiken sind ein fundamentales Werkzeug der Theoretischen Informatik, um komplexe Strukturen und Wachstumsdynamiken zu modellieren [1]. Während kontextfreie Grammatiken typischerweise Sprachen mit polynomialem oder exponentiellem Wachstum erzeugen, erfordert die hier betrachtete Sprache $\{a^{n^n}\}$ eine Grammatik, die in der Lage ist, überexponentielles Wachstum rein durch lokale Regelanwendungen zu realisieren.Das Problem ist daher nicht nur von theoretischem Interesse, sondern besitzt auch eine enge Verbindung zu algorithmischen Fragestellungen der DNA- und Molekularen Algorithmen, in denen komplexe Berechnungen durch lokale Operationen modelliert werden können [2] [3].

Ziel dieser Arbeit ist die Entwicklung einer solchen erweiterten Chomsky-Grammatik. Dabei wird die Funktionsweise der Regelanwendungen formal beschrieben. Neben der präzisen Definition der verwendeten Symbolmengen und Produktionsregeln wird auch exemplarisch gezeigt, wie ein Wort der Form $\{a^{n^n}\}$ abgeleitet werden kann. Schließlich erfolgt eine Komplexitätsanalyse, in der die Anzahl der Regelanwendungen und die resultierende Laufzeitabschätzung hergeleitet werden.

1.1 Formale Grammatiken

Formale Grammatiken sind zentrale Werkzeuge der theoretischen Informatik. Eine Grammatik $G = (V, \Sigma, P, S)$ besteht aus:

- einem endlichen Nichtterminalalphabet V,
- einem Terminalalphabet Σ,
- einer endlichen Regel oder Produktionsmenge P,
- und einem Startsymbol S.

Eine Regel $(u, v) \in P$ wird als $u \to v$ geschrieben, und somit wird u durch v ersetzt.

Durch wiederholte Anwendung der Regel kann aus dem Startsymbol ein Wort aus den Terminalsymbolen erzeugt werden.

2 Zielsetzung

Eine Chomsky Grammatik stellt grundlegende Werkzeuge bereit, um algorithmische Prozesse abstrakt zu modellieren. Mit unterschiedlicher Typen von Chomsky Grammatiken lassen sich Sprachen mit verschieden hohem Ausdrucksvermögen beschreiben. Ziel dieser Arbeit ist die Konstruktion und Analyse einer Grammatik, die eine überexponentiell wachsende Zahlenfolge abbildet: Für jedes $n \geq 1$ soll das Wort n^n erzeugt werden.

Die Sprache:

$$L(G) = \{n^n \mid n \in \mathbb{N},\ n \geq 1\} = \{1, 4, 27, 256, \dots\}$$

besteht aus Wörtern der Form $\{a\}^{n^n}$, wobei jedes Wort eine einheitliche Wiederholung des Symbols a darstellt. Für jedes $n \in \mathbb{N},\ n \geq 1$ enthält die Sprache genau ein Wort der Länge n^n.

Im Vergleich zu kontextfreien Sprachen, die typischerweise Wörter mit polynomialem oder exponentiellem Wachstum erzeugen (z.B. $a^n b^n$, a^{2^n}), stellt $\{a\}^{n^n}$ eine deutlich komplexere Wachstumsdynamik dar. Die Konstruktion einer Grammatik, die diese Sprache erzeugt, erfordert eine geschickte Verkettung von Regeln, die sowohl das Zählen als auch die exponentielle Wiederholung einer Struktur erlauben.

Ziel ist also eine Chomsky-Grammatik zu entwickeln, mit der die Zahlen n^n mit $n \in \mathbb{N},\ n \geq 1$ sukzessive gebildet und ausgegeben werden. Nach unärer Kodierung der Zahl n durch das Wort $\{a\}^n$ ist unser Ziel also die Wortmenge $\{\{a\}^{n^n} \mid n \geq 1\}$ zu generieren, wobei die Zahlen durch ein Trennsymbol (,) in eigene abschnitte unterteilt werden.

3 Implementierung der Grammatik

Zur Generierung der Sprache $L(G) = \{a^{n^n} \mid n \in \mathbb{N}, n \geq 1\}$ wurde eine erweiterte Chomsky-Grammatik entworfen. Diese basiert auf einer schrittweisen Blockkopie und Potenzierung, welche mithilfe zahlreicher Hilfssymbole koordiniert wird. Die nachfolgenden Abschnitte beschreiben die verwendeten Symbole, Produktionsregeln sowie die zugrunde liegende algorithmische Struktur im Detail.

3.1 Symbolmengen und Produktionen

Die Grammatik basiert auf einer endlichen Menge von Symbolen, bestehend aus:

- **Startsymbol:** S
- **Terminalsymbole:** a und ,
- **Nichtterminalsymbole (Auszug):** $S\ b\ c\ M\ N\ G\ I\ K\ F\ J\ V\ U\ Q\ \$\ D\ W\ Y\ A\ Z\ H\ L$ und X

Die Produktionsregeln transformieren symbolische Ausdrücke über mehrere Schritte in die gewünschte Zielstruktur. Im Folgenden ist ein Auszug der Produktionen dargestellt:

Startkonfiguration (Initialisierung)

1. $S \to , a, aa, c,$
2. $a, c \to M, ca$
3. $aM \to Ma$
4. $, M \to , N$
5. $NM \to MN$
6. $N, c \to N, a$
7. $N, a \to G, a$
8. $MG \to Ga$
9. $NG \to Ga$

Konvertierung durch Blockkopie

10. $,G \rightarrow ,aJL$
11. $La \rightarrow aLb$
12. $Lb \rightarrow bL$
13. $L, \rightarrow F,$
14. $ba \rightarrow ab$
15. $bF \rightarrow Fb$
16. $Ja \rightarrow aJ$
17. $JFb \rightarrow \$QbVc$
18. $cb \rightarrow bc$
19. $Vb \rightarrow bVc$
20. $Vc \rightarrow cV$
21. $V, \rightarrow U,$
22. $cU \rightarrow Uc$
23. $Qb \rightarrow bQ$
24. $QU \rightarrow \$$

Potenzierung

25. $\$c \to WD\$$
26. $bW \to WYb$
27. $\$W \to \$$
28. $bD \to Db$
29. $bY \to Yb$
30. $\$D \to D\$$
31. $AD \to Da$
32. $aD \to Da$
33. $, D \to ,$
34. $\$Y \to Y\$$
35. $aY \to YAa$
36. $AY \to YA$
37. $, Y \to ,$

Aufräumen

38. $\$, \to Z,$
39. $bZ \to Z$
40. $\$Z, \to , X$
41. $Xa \to aX$
42. $X, \to , c,$

Modulare Struktur

Die Regeln lassen sich in vier Phasen gliedern:

a) **Initialisierung** (Regeln 1–9): Aufbau der iterativ hoch zählenden Startkonfiguration: $\{a\}^1, \{a\}^2, ..., \{a\}^n, \{a\}^{n+1}, ...$

b) **Konvertierung durch Blockkopie** (Regeln 10–24): Iteratives Duplizieren des a-Blocks unter Nutzung von Zwischensymbolen.

c) **Potenzierung** (Regeln 25–37): Vereinheitlichung der Kopien und Vorbereitung der Aufräumphase.

d) **Aufräumen** (Regeln 38–42): Entfernung sämtlicher Hilfssymbole, bis nur noch a in dem bearbeiteten abschnitt verbleibt.

Die Grammatik umfasst insgesamt 42 Regeln, die strikt modular aufgebaut sind. Dadurch lässt sich jeder Funktionsblock (Kopie, Potenzierung etc.) unabhängig entwerfen, testen und erweitern. Diese Struktur erleichtert auch eine formale Analyse der Komplexität und Laufzeit.

3.2 Algorithmische Idee

Zur Generierung der Zahlenfolge $\{a\}^{n^n}$ bietet es sich an, das Problem in Teilaufgaben zu zerlegen. Beginnend mit der sukzessiven Erzeugung der Zahlenfolge $1, 2, 3, \ldots$, dargestellt als a, aa, aaa, $\ldots$, erfolgt zunächst eine Initialisierung. Diese erfordert im Wesentlichen, dass nach jedem Trennsymbol der gesamte Inhalt des vorherigen Abschnitts kopiert und ein weiteres Symbol a angehängt wird. In der vorliegenden Implementierung werden die ersten beiden Abschnitte bereits durch die erste Regel nach dem Startsymbol erzeugt. Anschließend wird mithilfe des Symbols c der a-Block von der linken auf die rechte Seite des Trennsymbols übertragen. Die Hilfssymbole M und N gewährleisten, dass alle a-Symbole kopiert werden, indem jedes übertragene a links in ein M umgewandelt welches nach links durch den verbleibenden Block wandert. Nach Erreichen des linken Endes wird M in N umgewandelt und wandert wieder nach rechts, wobei N nicht an a vorbei gelangen kann. Sobald das Paar N, c erreicht wird, ist sichergestellt, dass alle a kopiert wurden. Abschließend werden alle verbleibenden M und N wieder in a rücktransformiert, womit ein Initialisierungsschritt abgeschlossen ist.

Da das Ziel eine unendliche Folge ist, muss die Initialisierung vor der Vorbereitung und Durchführung der Potenzberechnung erfolgen. Andernfalls wäre es notwendig, zu einem späteren Zeitpunkt die n-te Wurzel zu berechnen, um den nächsten Wert n zu bestimmen. Gleichzeitig darf die Generierung der hoch zählenden Zahlenfolge nicht unbegrenzt fortgeführt werden, ohne die Potenzen auch tatsächlich auszurechnen. Um beiden Anforderungen gerecht zu werden, erfolgt die Initialisierung inkrementell und bleibt der Potenzierung stets einen Schritt voraus.

Im Anschluss erfolgt die Konvertierung. Dieser Schritt bereitet einen Abschnitt bestehend aus n-mal a, wie zuvor in der Initialisierung generiert, auf die Potenzierung vor. Ziel ist es, an den Block mit n-vielen a ein Trennsymbol \$ sowie $n-1$ Symbole b anzuhängen. Dies entspricht der Struktur $n \cdot n$ beziehungsweise $n \cdot (n-1) + n$, da das ursprüngliche n erhalten bleibt und lediglich um $n \cdot (n-1)$ weitere a ergänzt wird (eine genauere Erläuterung folgt im Potenzierungsschritt). Da für n^n genau $n-1$ Multiplikationen durchgeführt werden müssen, wird ein zweites Trennsymbol \$ eingefügt und $n-1$ Symbole c ergänzt. Für das Beispiel $n = 4$ ergibt sich somit: $aaaa \rightarrow aaaa\$bbb\$ccc$, entsprechend $4 \cdot 4^3$.

Für die Erstellung des b-Blocks werden mehrere Hilfssymbole verwendet: J, L und F. Am linken Rand des Abschnitts wird ein einzelnes J erzeugt, das sich ausschließlich durch a nach rechts bewegen kann. Parallel dazu wird ein L am Beginn des a-Blocks eingeführt. Dieses wandert durch die a-Symbole und erzeugt pro Symbol jeweils ein b. Da L rechts vom ersten a positioniert wird, werden nur für die verbleibenden $n-1$ a jeweils ein b erstellt. L kann auch durch b nach rechts wandern, und b wiederum kann sich durch a bewegen. Wenn L das rechte Ende erreicht, wird es in ein F überführt, das anschließend durch b nach links wandert. Trifft es auf J, werden beide in ein Trennsymbol \$ umgewandelt und initiieren gleichzeitig die Erzeugung der c-Symbole. Diese erfolgt analog zur b-Erstellung, mit dem Unterschied, dass das erste c-Symbol nicht übersprungen wird und andere Hilfssymbole zum Einsatz kommen.

Die Potenzierung erfolgt auf der Grundlage, dass jeder c-Block einzeln abgearbeitet wird und für jedes c der a-Block b-mal dupliziert wird. Dazu wird für jedes c ein W und ein D generiert. Das W wandert nach links durch die b und erzeugt dabei jeweils ein Y. Dieses Y überquert das Trennsymbol zum a-Block und läuft dort von rechts nach links, wobei pro a ein A erstellt wird. Die Funktion des Symbols D besteht darin, die A-Symbole in a umzuwandeln und so sicherzustellen, dass jede Multiplikation unabhängig abgeschlossen wird. Da die ursprünglichen a-Symbole beibehalten werden, muss die Replikation nur $n - 1$-mal durchgeführt werden. Alternativ hätte man auch n-mal b erzeugen und die a-Symbole nach jeder Multiplikation löschen können – dies hätte jedoch die Anzahl der notwendigen Schritte erheblich erhöht.

Abschließend erfolgt ein sogenanntes Clean-up: Mithilfe des Hilfssymbols Z werden überflüssige Trennsymbole \$ sowie die b-Symbole entfernt. Der bearbeitete Abschnitt enthält danach nur noch a. Zuletzt wird mit dem Symbol X ein neues c im übernächsten Abschnitt erzeugt, womit die Iteration abgeschlossen ist und die nächste Runde beginnen kann. Dadurch ergibt sich ein fester Rhythmus: Jeder Initialisierungsschritt wird gefolgt von einer Konvertierungs- und Potenzierungsphase im vorangegangenen Abschnitt. Dieser Rhythmus ist essenziell, um die unendliche Folge korrekt und vollständig zu generieren.

3.3 Beispielableitung ($n = 2$)

Regelanwendung n

1. S → ,a,aa,c,
2. ,a,aa,c, → ,a,aM,ca,
3. ,a,aM,ca, → ,a,Ma,ca,
4. ,a,Ma,ca, → ,a,Na,ca,
5. ,a,Na,ca, → ,a,NM,caa,
6. ,a,NM,caa, → ,a,MN,caa,
7. ,a,MN,caa, → ,a,NN,caa,
8. ,a,NN,caa, → ,a,NN,aaa,
9. ,a,NN,aaa, → ,a,NG,aaa,
10. ,a,NG,aaa, → ,a,Ga,aaa,
11. ,a,Ga,aaa, → ,a,aJLa,aaa,
12. ,a,aJLa,aaa, → ,a,aJaLb,aaa,
13. ,a,aJaLb,aaa, → ,a,aJabL,aaa,
14. ,a,aJabL,aaa, → ,a,aJabF,aaa,
15. ,a,aJabF,aaa, → ,a,aJaFb,aaa,
16. ,a,aJaFb,aaa, → ,a,aaJFb,aaa,
17. ,a,aaJFb,aaa, → ,a,aa$QbVc,aaa,
18. ,a,aa$QbVc,aaa, → ,a,aa$bQVc,aaa,
19. ,a,aa$bQVc,aaa, → ,a,aa$bQcV,aaa,
20. ,a,aa$bQcV,aaa, → ,a,aa$bQcU,aaa,
21. ,a,aa$bQcU,aaa, → ,a,aa$bQUc,aaa,
22. ,a,aa$bQUc,aaa, → ,a,aa$b$c,aaa,
23. ,a,aa$b$c,aaa, → ,a,aabWD,aaa,
24. ,a,aabWD,aaa, → ,a,aa$WYbD$,aaa,
25. ,a,aa$WYbD$,aaa, → ,a,aaYbD,aaa,
26. ,a,aaYbD,aaa, → ,a,aaYDb,aaa,
27. ,a,aaYDb,aaa, → ,a,aa$YDbZ,aaa,
28. ,a,aa$YDbZ,aaa, → ,a,aaY$DbZ,aaa,
29. ,a,aaY$DbZ,aaa, → ,a,aaYD$bZ,aaa,
30. ,a,aaYD$bZ,aaa, → ,a,aYAaD$bZ,aaa,
31. ,a,aYAaD$bZ,aaa, → ,a,aYAaD$Z,aaa,
32. ,a,aYAaD$Z,aaa, → ,a,aYADa$Z,aaa,
33. ,a,aYADa$Z,aaa, → ,a,aYDaa$Z,aaa,
34. ,a,aYDaa$Z,aaa, → ,a,YAaDaa$Z,aaa,
35. ,a,YAaDaa$Z,aaa, → ,a,YAaDaa,Xaaa,
36. ,a,YAaDaa,Xaaa, → ,a,YADaaa,Xaaa,
37. ,a,YADaaa,Xaaa, → ,a,YDaaaa,Xaaa,
38. ,a,YDaaaa,Xaaa, → ,a,Daaaa,Xaaa,
39. ,a,Daaaa,Xaaa, → ,a,aaaa,Xaaa,
40. ,a,aaaa,Xaaa, → ,a,aaaa,aXaa,
41. ,a,aaaa,aXaa, → ,a,aaaa,aaXa,
42. ,a,aaaa,aaXa, → ,a,aaaa,aaaX,
43. ,a,aaaa,aaaX, → ,a,aaaa,aaa,c,

Nach Anwendung der 43 Regeln ist bereits das charakteristische Muster `a,aaaa,aaa,c,` erkennbar, die auf die Struktur $\{a\}^{n^n}$ für $n = 2$ hindeutet. Die Teilwörter a, $aaaa$ und aaa, c entsprechen 1^1 und 2^2; das letzte Element c signalisiert, dass die nächste Phase zur Erzeugung von 3^3 bereits vorbereitet wird. (Initialisierungsphase beginnt.)

Hinweis zur Initialisierung
Die erste Phase der Ableitung enthält bereits implizit das Ergebnis für $n = 1$. Das Teilwort `a` entspricht $1^1 = 1$ und wird durch die Regel $S \rightarrow, a, aa, c,$ automatisch erzeugt. Die Iteration startet ab $n = 2$ mit vollständiger Initialisierungs- und Konvertierungsphase.

4 Komplexitätsanalyse

Die Komplexität der Generierung der gesamten Folge sollte im Idealfall so gering wie möglich sein. Da jedoch eine Folge mit superexponentiellem Wachstum erzeugt wird, liegt bereits der Best Case im superexponentiellen Bereich. Im Folgenden wird ein Überblick über die Zeitkomplexität der finalen Implementierung gegeben.

Die Initialisierungsphase lässt sich weiter untergliedern. Beginnend mit den Regeln zur Kopie der a-Symbole: Diese werden genau $n-1$ Mal ausgeführt. Die Verschieberegel für das Symbol M benötigt im schlimmsten Fall höchstens $\sum_{i=1}^{n-1} i$ Schritte. Es können maximal $n-1$ N-Symbole generiert werden, die sich jeweils bis zu $1+\sum_{i=2}^{n-1} i$ Schritte nach rechts bewegen, bevor das erste N das Trennsymbol erreicht. Danach wandert das G maximal $n-1$ Schritte nach links. Insgesamt ergibt sich somit ein Anstieg von n Schritten für jede Iteration von n, sodass die Initialisierung in $\mathcal{O}(n^2)$ erfolgt.

Die Konvertierungsphase besteht aus zwei Duplizierungsschritten, die jeweils eine lineare Anzahl an Schritten erfordern. Daraus folgt, das in dieser Phase insgesamt $\mathcal{O}(n)$ Schritte benötigt.

Die Potenzierungsphase enthält zwar nicht die meisten Regeln, ist jedoch mit Abstand die rechenintensivste. Für jedes Vorkommen von c werden $n-1$ Symbole Y erzeugt, die jeweils durch den gesamten a- bzw. A-Block wandern. Die Anzahl der hierfür benötigten Schritte steigt mit der Abarbeitung jedes weiteren c-Symbols um den Faktor n. Die erste Multiplikation (für das erste c) erfordert n Schritte, die letzte (für das $n-1$-te c) insgesamt n^n Schritte. Dies entspricht einer geometrischen Reihe mit einer Gesamtschrittanzahl von

$$\sum_{i=1}^{n} n^i = \frac{n(n^n-1)}{n-1}.$$

Da der Zähler eine Wachstumsrate von $\mathcal{O}(n^{n+1})$ hat und der Nenner $\mathcal{O}(n)$ ergibt sich eine Gesamtschrankenordnung von $\mathcal{O}(n^n)$.

Die abschließende Aufräumphase benötigt erneut nur eine lineare Anzahl an Schritten. Zunächst werden $n-1$ b-Symbole entfernt, gefolgt von etwa $n+1$ Schritten durch die verbleibenden a-Symbole des nächsten Abschnitts.

Zusammenfassend ergibt sich für einen vollständigen Durchlauf von einem Iterationsschritt im schlechtesten Fall eine Laufzeitkomplexität von $\mathcal{O}(n^n)$.

5 Zusammenfassung

In dieser Arbeit wurde eine erweiterte Chomsky-Grammatik vorgestellt, die die der Sprache $L(G) = \{a^{n^n} \mid n \in \mathbb{N},\ n \geq 1\}$ erzeugt. Durch eine strukturierte Aufteilung in Initialisierung, Blockkopie, Potenzierung und Aufräumphase wird ein deterministischer Prozess definiert, der sowohl die Hochzählung der Eingabewerte als auch deren wiederholte Vervielfältigung effizient simuliert.

Jeder Iterationsschritt besteht aus der Generierung eines neuen Blocks $\{a\}^n$, der Umwandlung in ein Zwischenformat zur Potenzierung, der rekursiven Multiplikation sowie der Rückführung in einen normierten Zustand. Die Komplexitätsanalyse zeigt, dass der ressourcenintensivste Teil die Potenzierung ist. Somit ergibt sich für diese Grammatik eine Laufzeit von $\mathcal{O}(n^n)$.

Die Arbeit demonstriert, dass auch extrem schnell wachsende Folgen mit rein grammatikalischen Mitteln modelliert und generiert werden können, sofern ein entsprechendes Regelsystem mit modularer Steuerung eingesetzt wird. Dies unterstreicht das Potenzial formaler Grammatiken sowohl in der theoretischen Informatik als auch in biochemisch motivierten Berechnungsmodellen.

6 Literatur

Literatur

[1] Michael Sipser. “Introduction to the Theory of Computation”. In: *ACM Sigact News* 27.1 (1996), S. 27–29.

[2] Junzo Watada und Rohani binti abu Bakar. “DNA computing and its applications”. In: *2008 Eighth international conference on intelligent systems design and applications*. Bd. 2. IEEE. 2008, S. 288–294.

[3] Thomas Hinze und Monika Sturm. *Rechnen mit DNA: eine Einführung in Theorie und Praxis*. Oldenbourg Wissenschaftsverlag GmbH, 2004.