# Eine Einführung in die Algorithmische Informationstheorie

**Kurt-Ulrich Witt**

**Publisher: Dean Prof. Dr. Wolfgang Heiden**

**Hochschule Bonn-Rhein-Sieg – University of Applied Sciences,
Department of Computer Science**

**Sankt Augustin, Germany**

**May 2018**

**Technical Report 02-2018**

Hochschule
Bonn-Rhein-Sieg
University of Applied Sciences

ISSN 1869-5272

ISBN 978-3-96043-062-9

**Digital Object Identifier** doi:10.18418/978-3-96043-062-9
**DOI-Resolver** http://dx.doi.org/

# Eine Einführung in die Algorithmische Informationstheorie

Kurt-Ulrich Witt

Hochschule Bonn-Rhein-Sieg
Fachbereich Informatik
b-it Applied Science Institute
Arbeitsgruppe Diskrete Mathematik und Optimierung (ADIMO)
kurt-ulrich.witt@h-brs.de

**Zusammenfassung** Bei der Übertragung und Speicherung von Daten ist es eine wesentliche Frage, inwieweit die Daten komprimiert werden können, ohne dass deren Informationsgehalt verloren geht. Ein Maß für den Informationsgehalt von Daten ist also von grundlegender Bedeutung. Vor etwa siebzig Jahren hat C. E. Shannon ein solches Maß eingeführt und damit das Lehr- und Forschungsgebiet der *Informationstheorie* begründet, welches seit dem bis heute hin wesentlich zur Konzeption und Realisierung von Informations- und Kommunikationstechnologien beigetragen hat. Etwa zwanzig Jahre später hat A. N. Kolmogorov ein anderes Maß für den Informationsgehalt von Daten eingeführt. Während die Shannonsche Informationstheorie zum Curriculum von mathematischen, informatischen und elektrotechnischen Studiengängen gehört, ist die *Algorithmische Informationstheorie* von Kolmogorov weit weniger bekannt und eher Gegenstand von speziellen Lehrveranstaltungen. Seit einigen Jahren nimmt allerdings die Beschäftigung mit dieser Theorie zu, zumal in der einschlägigen Literatur von erfolgreichen praktischen Anwendungen der Theorie berichtet wird. Die vorliegende Arbeit gibt eine Einführung in grundlegende Ideen dieser Theorie und beschreibt deren Anwendungsmöglichkeiten bei einigen ausgewählten Problemstellungen der Theoretischen Informatik. Die Ausarbeitung kann als Skript für einführende Lehrveranstaltungen in die Algorithmische Informationstheorie sowie als Lektüre zur Einarbeitung in die Thematik als Ausgangspunkt für Forschungs- und Entwicklungsarbeiten verwendet werden.

*Schlüsselwörter:* Codierung, Kompression und Zufälligkeit von Zeichenketten, Kolmogorov-Komplexität, Chaitin-Konstante

# Prolog

Bei der Übertragung und Speicherung von Daten ist es eine wesentliche Frage, inwieweit die Daten komprimiert werden können, ohne dass deren Informationsgehalt verloren geht. Ein Maß für den Informationsgehalt von Daten ist also von grundlegender Bedeutung. Vor etwa siebzig Jahren hat C. E. Shannon[1] ein solches Maß eingeführt und damit das Lehr- und Forschungsgebiet der *Informationstheorie* begründet, welches seit dieser Zeit bis heute hin wesentlich zur Konzeption und Realisierung von Informations- und Kommunikationstechnologien beigetragen hat. Etwa zwanzig Jahre später hat A. N. Kolmogorov[2] ein anderes Maß für den Informationsgehalt von Daten eingeführt. Während die Shannonsche Informationstheorie zum Inhalt von mathematischen, informatischen und elektrotechnischen Studiengängen gehört, ist die *Algorithmische Informationstheorie* von Kolmogorov weit weniger bekannt und eher Gegenstand von speziellen Lehrveranstaltungen. In letzter Zeit gewinnt die Algorithmische Informationstheorie allerdings an Beachtung und zwar nicht zuletzt durch Fragestellungen in mathematischen und informatischen Anwendungsbereichen, wie z.B. die Mustererkennung und der Abgleich von Datenströmen.

Die vorliegende Ausarbeitung ist wie folgt gegliedert: Im Kapitel 1, der Einleitung, werden zunächst ein paar Beispiele für die Codierung von Bitfolgen aufgelistet, um in die Problematik der Kompression von Datenströmen einzuführen. Des Weiteren werden Begriffe eingeführt, die in den weiteren Kapiteln benötigt werden, unter anderem über Zeichenketten und Mengen von Zeichenketten sowie über die Berechenbarkeit von Funktionen. Im Kapitel 2 wird die Kolmogorov-Komplexität von Zeichenketten eingeführt und in Kapitel 3 werden einige ihrer grundlegenden Eigenschaften vorgestellt. Kapitel 4 führt auf der Basis der Kolmogorov-Komplexität die Begriffe Komprimierbarkeit und Zufälligkeit von Zeichenketten ein. Im Kapitel 5 werden einige theoretische Fragestellungen mithilfe der Kolmogorov-Komplexität behandelt.

[1] C. E. Shannon (1916 - 2001) war ein amerikanischer Mathematiker und Elektrotechniker, der 1948 mit seiner Arbeit *A Mathematical Theory of Communication* die Informationstheorie begründete. Shannon war vielseitig interessiert und begabt. So lieferte er unter anderem beachtete Beiträge zur Booleschen Algebra, er beschäftige sich mit dem Bau von Schachcomputern und der Entwicklung von Schachprogrammen, und er lieferte Beiträge zur Lösung wirtschaftswissenschaftlicher Probleme.

[2] A. N. Kolmogorov (1903 – 1987) war ein russischer Mathematiker, der als einer der bedeutendsten Mathematiker der 20. Jahrhunderts gilt. Er lieferte wesentliche Beiträge zu vielen Gebieten der Mathematik und auch zur Physik. Einer seiner grundlegenden Beiträge ist die Axiomatisierung der Wahrscheinlichkeitstheorie.

# 1 Einleitung

Wir wollen uns einführend mit dem *Informationsgehalt* und der *Komplexität von Zeichenketten* befassen. Zeichenketten werden über Alphabete gebildet. Dass unsere Betrachtungen nicht von den jeweils gewählten Alphabeten abhängen und wir daher von konkreten Alphabeten abstrahiert können, zeigt folgende Überlegung: Jeder Buchstabe eines Alphabets $\Sigma = \{a_0, \ldots, a_{k-1}\}$ mit $k \geq 2$ Symbolen kann durch ein Wort der Länge $\lceil \log k \rceil$ ($\log x$ ist der Logarithmus von $x$ zur Basis 2) über dem Alphabet $\{0, 1\}$ codiert werden, indem z.B. $a_i$ durch die Binärdarstellung von $i$ repräsentiert wird. Für die binäre Codierung $w_{bin}$ eines Wortes $w \in \Sigma^*$ ergibt sich dann die Länge $|w_{bin}| = |w| \cdot \lceil \log k \rceil$. So könnte z.B. $\Sigma = \{a, b, c\}$ durch $\Sigma_{bin} = \{00, 01, 10\}$ codiert werden. Das Wort $bcab \in \Sigma^*$ würde dann durch das Wort $01100001 \in \Sigma_{bin}^*$ dargestellt. Codierung und Decodierung können in linearer Zeit durchgeführt werden.

Aus diesen Überlegungen ergibt sich im Übrigen, dass einelementige (unäre) Alphabete und damit die „Bierdeckelcodierung“ von Zahlen ungeeignet sind, da deren unäre Repräsentation exponentiell länger als deren binäre Darstellung ist.

Wir können uns bei unseren Betrachtungen also auf Zeichenketten über zweielementigen Alphabeten beschränken und wählen dazu *das* Alphabet der Informatik: $\mathbb{B} = \{0, 1\}$. Wir bezeichnen im Folgenden mit $\mathbb{B}^*$ die Menge aller endlichen Bitfolgen, mit $\mathbb{B}^+$ die Menge der nicht leeren Bitfolgen (es ist also $\mathbb{B}^+ = \mathbb{B}^* - \{\varepsilon\}$) sowie mit $\mathbb{B}^\omega$ die Menge aller unendlichen Bitfolgen.

## 1.1 Beispiele zur Kompression von Bitfolgen

Es geht um ein sinnvolles Maß des *Informationsgehaltes von Wörtern* über $\mathbb{B}$. Wegen ihrer wesentlichen Beiträge zu dieser Thematik spricht man auch von *Kolmogorov*-, *Solomonoff*[3]- oder *Chaitin*[4]-Komplexität; eine heute gängige Bezeichnung, die die genannten und weitere damit zusammenhängende Aspekte umfasst, ist *Algorithmische Informationstheorie*.

[3] R. Solomonoff (1926 - 2009) war ein amerikanischer Mathematiker, der schon zu Beginn der sechziger Jahre des vorigen Jahrhunderts Grundideen zur Algorithmischen Informationstheorie veröffentlichte, also vor Kolmogorov, aber diesem waren die Solomonoff-Arbeiten nicht bekannt.

[4] G. Chaitin (1947) ist ein amerikanischer Mathematiker, der sich mit Fragen zur prinzipiellen Berechenbarkeit beschäftigt. Im Kapitel 5 gehen wir näher auf einige seiner Ansätze ein.

Betrachten wir die Zeichenkette

$$v = 101101101101101101101101 \tag{1.1}$$

so erkennen wir eine Regelmäßigkeit, nämlich dass sich $v$ aus acht $101$-Gruppen zusammensetzt. $v$ lässt sich mit bekannten mathematischen Notationen kürzer darstellen – also „komprimieren“ – etwa durch

$$v_1 = [101]^8 \tag{1.2}$$

Betrachten wir die Zeichenkette

$$w = 110101110111001010101111 \tag{1.3}$$

so erscheint diese „komplexer“, da keine Regelmäßigkeit zu erkennen ist und man keine Komprimierung findet. Man könnte $w$ als komplexer ansehen als $v$, da sich der Informationsgehalt von $v$ wesentlich kürzer darstellen lässt als ausgeschrieben wie in (1.1). Wegen seiner regelmäßigen Struktur erscheint das Wort $v$ weniger Information zu tragen als das unregelmäßige Wort $w$, denn es lässt sich ohne Informationsverlust komprimieren.

Betrachten wir das Alphabet $\mathbb{B}$ als eine Quelle

$$Q = \begin{pmatrix} 0 & 1 \\ p & 1-p \end{pmatrix}$$

für Datenströme in Form von Bitfolgen, d.h. das Bit $0$ tritt bei dieser Quelle in den Strömen mit der Wahrscheinlichkeit $\mathrm{p}(0) = p$, $0 \leq p \leq 1$, und folglich das Bit $1$ mit einer Wahrscheinlichkeit $\mathrm{p}(1) = 1 - p$ auf. Dann treten die beiden Datenströme $v$ und $w$ mit derselben Wahrscheinlichkeit auf, nämlich mit

$$Prob(v) = Prob(w) = \mathrm{p}(0)^8 \cdot \mathrm{p}(1)^{16} = p^8 \cdot (1-p)^{16}$$

d.h. $v$ und $w$ tragen (in der Shannonschen Informationstheorie) denselben Informationsgehalt – wie im Übrigen alle Wörter der Länge $24$ über $Q$ mit $8$ Nullen und $16$ Einsen. Die oben beachteten Regelmäßigkeiten spielen hier keine Rolle.

Es stellt sich also die Frage nach einem Informationsbegriff, der die Regelmäßigkeit bzw. die Zufälligkeit von Bitfolgen besser erfasst. Wie schon angedeutet, könnte die Komprimierbarkeit von Bitfolgen ein Kandidat für ein entsprechendes Informationsmaß sein.

**Definition 1.1** *Wir nennen totale Abbildungen* $\kappa : \mathbb{B}^* \to \mathbb{B}^*$ **Komprimierungen** *von* $\mathbb{B}^*$. $\kappa$ *heißt* **echte Komprimierung**, *falls* $\kappa(x) < |x|$ *für fast alle* $x \in \mathbb{B}^*$ *ist.*[5] □

Die Zeichenkette (1.2) ist kein Wort über $\mathbb{B}$, also gemäß Definition 1.1 keine zulässige Komprimierung. Wenn wir den Exponenten dual darstellen, ergibt sich mit

$$v_2 = [101]\,1000$$

ein Wort über dem Alphabet $\{\,0, 1, [\ ,\ ]\,\}$. Wir erreichen eine zulässige Komprimierung etwa mithilfe des Homomorphismus

$$\eta : \{\,0, 1, [\ ,\ ]\,\} \to \mathbb{B}^2$$

definiert durch

$$\eta(0) = 00,\ \ \eta(1) = 11,\ \ \eta([) = 10,\ \ \eta(]) = 01$$

Wir erweitern $\eta$ zu $\eta^* : \{\,0, 1, [\ ,\ ]\,\}^* \to \mathbb{B}^*$ definiert durch

$$\begin{aligned} \eta^*(\varepsilon) &= \varepsilon \\ \eta^*(bw) &= \eta(b) \circ \eta^*(w), \ \text{ für } b \in \{0, 1, [\ ,\ ]\} \ \text{ und } \ w \in \{0, 1, [\ ,\ ]\}^* \end{aligned}$$

Wir werden im Folgenden nicht zwischen $\eta$ und $\eta^*$ unterscheiden und beide Funktionen mit $\eta$ bezeichnen.

Für unser Beispiel ergibt sich dann

$$\eta(v_2) = 10\,110011\,01\,11000000$$

als eine mögliche Komprimierung von $v$; diese ist echt.

Das Wort

$$x = 0000001101010101010101010101010101010101010101010101010$$

lässt sich zunächst wie folgt darstellen:

$$x_1 = [0]^6\,[1]^1\,[10]^{22}$$

Daraus ergibt sich

$$x_2 = [0]\,110\,[1]\,1\,[10]\,10110$$

[5] Die Aussage „für fast alle $x \in M$ gilt das Prädikat $P$“ bedeutet, dass $P(x)$ für höchstens endlich viele Elemente $x$ von $M$ nicht wahr wird.

und schließlich

$$\eta(x_2) = 10\,00\,01\,111100\,10\,11\,01\,11\,10\,1100\,01\,1100111100$$

als eine mögliche echte Komprimierung von $x$.

An den Beispielen sieht man, dass es in aller Regel mehrere Möglichkeiten der Komprimierung gibt. So kann man z.B. Potenzen mehrfach anwenden, um kürzere Darstellungen zu erhalten. So kann man z.B.

$$[10]^{4294967296}$$

kürzer darstellen, etwa durch

$$[10]^{2^{32}}$$

oder noch kürzer durch

$$[10]^{2^{2^5}}$$

Diese Art der Darstellung kann man beliebig weiter treiben, um kurze Darstellungen für

$$[10]^{2^n}, [10]^{2^{2^n}}, [10]^{2^{2^{2^n}}}, \ldots$$

zu erhalten.

Wir können alle Elemente von $\mathbb{B}^+$ als Dualdarstellungen natürlicher Zahlen auffassen. Für $x = x_{n-1} \ldots x_0 \in \mathbb{B}^n$, $n \geq 1$, sei

$$wert(x) = \sum_{i=0}^{n-1} x_i \cdot 2^i$$

der Wert von $x$. Jede solche Zahl kann zudem faktorisiert werden:

$$wert(x) = \prod_{j=1}^{k} p_j^{\alpha_j} \tag{1.4}$$

mit $p_j \in \mathbb{P}$, $1 \leq j \leq k$, und $p_j < p_{j+1}$, $1 \leq j \leq k-1$, sowie $\alpha_j \in \mathbb{N}$, $1 \leq j \leq k$, sei die (eindeutige) kanonische Faktorisierung von $wert(x)$. Die Faktorisierung könnte man über dem Alphabet $\{\, 0, 1, [\ , \ ]\,\}$ wie folgt darstellen:

$$dual(p_1)\,[dual(\alpha_1)]\;dual(p_2)\,[dual(\alpha_3)] \ldots dual(p_k)\,[dual(\alpha_k)] \tag{1.5}$$

wobei $dual(n)$ die Dualdarstellung von $n \in \mathbb{N}_0$ ist mit $wert(dual(n)) = n$. Mithilfe des Homomorphismus $h$ ergibt sich daraus wieder eine Darstellung über $\mathbb{B}$.

Betrachten wir als Beispiel die Zahl

$$n = 1\,428\,273\,264\,576\,872\,238\,279\,737\,182\,200\,000$$

mit der Dualdarstellung $z = dual(n) =$

$$10001100110101101010111110100011110001011000100011011010$$
$$10111110000111111111110000110001000101100100000011000000$$

Die Faktorisierung von $z$ ist

$$2^6 \cdot 3^9 \cdot 5^5 \cdot 7^{11} \cdot 11^3$$

Deren Darstellung über $\{\, 0, 1, [\ ,\ ]\,\}$ ist

$$z_{\mathbb{P}} = 10\,[110]\,11\,[1001]\,101\,[101]\,111\,[1011]\,1011\,[11]$$

Die Anwendung von $\eta$ liefert die Darstellung

$$\eta(z_{\mathbb{P}}) = 1100\,10\,111100\,01\,1111\,10\,11000011\,01\,110011\,10\,110011\,01$$
$$111111\,10\,11001111\,01\,11001111\,10\,1111\,01$$

Man könnte im Übrigen auch anstelle einer Primzahl selbst ihre Nummer innerhalb einer aufsteigenden Auflistung der Primzahlen in der Darstellung einer Zahl verwenden. Damit ergäbe sich für die obige Bitfolge $z$ anstelle von $z_{\mathbb{P}}$ die Darstellung

$$z'_{\mathbb{P}} = 1\,[110]\,10\,[1001]\,11\,[101]\,100\,[1011]\,101\,[11]$$

worauf die Anwendung von $\eta$ die Darstellung

$$\eta(z'_{\mathbb{P}}) = 11\,10\,111100\,01\,1100\,10\,11000011\,01\,1111\,10\,110011\,01$$
$$110000\,10\,11001111\,01\,110011\,10\,1111\,01$$

ergibt.

Die Frage, die sich jetzt stellt, ist, welche Komprimierungsmethode ist besser? Die Methoden sind kaum vergleichbar. Die eine Methode komprimiert Zeichenketten besser als die andere und umgekehrt. Es stellt sich also die Frage nach einer Komprimierungsmethode, die allgemeingültige Aussagen zulässt und die zudem noch in sinnvollerweise den Informationsgehalt einer Zeichenkette beschreibt.

Für die Betrachtungen dazu benötigen wir noch einige Begriffe, die im Folgenden aufgelistet werden.

## 1.2 Grundlegende Begriffe

Wir gehen öfter von der **lexikografischen** oder auch **kanonischen (An-) Ordnung** der Wörter in $\Sigma^*$ über einem Alphabet $\Sigma$ aus. Voraussetzung ist, dass die Buchstaben von $\Sigma = \{ a_1, a_2, \ldots, a_n \}$ einer totalen Ordnung unterliegen, etwa $a_i < a_{i+1}$, $1 \leq i \leq n-1$. Dann ist die Relation $\prec \subseteq \Sigma^* \times \Sigma^*$ definiert durch

$$\begin{aligned} v \prec w \text{ genau dann, wenn } & |v| < |w| \\ \text{oder } & |v| = |w| \text{ und } v = \alpha a \beta \text{ und } w = \alpha b \gamma \\ & \text{mit } a, b \in \Sigma \text{ und } a < b, \alpha, \beta, \gamma \in \Sigma^* \end{aligned}$$

Die Menge $\Sigma^*$ wird durch die Relation $\prec$ total geordnet. Deshalb besitzt jede nicht leere Teilmenge von $\Sigma^*$ bezüglich dieser Ordnung genau ein kleinstes Element.

Wenn wir im Folgenden von einem geordneten Alphabet $\Sigma = \{ a_1, \ldots, a_n \}$ sprechen, dann seien die Buchstaben von $\Sigma$ in ihrer Ordnung aufgezählt, d.h. es sei $a_i < a_{i+1}$, $1 \leq i \leq n-1$. Für das Alphabet $\mathbb{B} = \{0, 1\}$ legen wir also $0 < 1$ fest. Für eine nicht leere Sprache $L \subseteq \Sigma^*$ sei $\min_L$ das bezüglich der Ordnung $\prec$ kleinste Wort von $L$.

Eine Sprache $L \subseteq \Sigma^*$ heißt **entscheidbar** genau dann, wenn ihre **charakteristische Funktion** $\chi_L : \Sigma^* \to \mathbb{B}$ definiert durch

$$\chi_L(w) = \begin{cases} 1, & w \in L \\ 0, & w \notin L \end{cases}$$

berechenbar ist (siehe Abschnitt 1.4).

**Bemerkung 1.1** *Sei $\Sigma = \{ a_1, a_2, \ldots, a_n \}$ ein geordnetes Alphabet.*

**a)** *Es sei $\& \notin \Sigma$. Die Sprache*

$$L_{(\Sigma^*, \prec)} = \{ v \& w \mid v, w \in \Sigma^*, \, v \prec w \}$$

*die alle geordneten Paare von Wörtern über $\Sigma^*$ enthält, ist entscheidbar.*

**b)** *Für alle $v \in \Sigma^*$ ist die Sprache*

$$SUC(v) = \{ w \in \Sigma^* \mid v \prec w \}$$

*die alle Nachfolger von $v$ enthält, entscheidbar.*

**c)** *Die* **Nachfolgerfunktion** $suc : \Sigma^* \to \Sigma^*$ *definiert durch*

$$suc(v) = \min_{SUC(v)}$$

*ist berechenbar.*

**d)** *Sei* $L \subseteq \Sigma^*$ *entscheidbar. Dann existiert eine Turingmaschine, die die Wörter von* $L$ *geordnet auf ein Band schreibt.*

**e)** *Die Funktion* $\tau : \Sigma^* \to \mathbb{N}_0$ *sei definiert durch*

$$\begin{aligned} \tau(\varepsilon) &= 0 \\ \tau(a_i) &= i,\ 1 \leq i \leq k \\ \tau(wa) &= n \cdot \tau(w) + \tau(a),\ a \in \Sigma,\ w \in \Sigma^* \end{aligned}$$

*Das Wort* $w$ *kommt genau dann an der* $i$*-ten Stelle in der lexikografischen Anordnung der Wörter von* $\Sigma^*$ *vor, wenn* $\tau(w) = i$ *ist.*

*Für* $w = w_1 w_2 \dots w_k$ *mit* $w_i \in \Sigma$, $1 \leq i \leq k$, *gilt:*

$$\tau(w) = \tau(w_1 w_2 \dots w_k) = \sum_{i=1}^{k} \tau(w_i) \cdot n^{k-i}$$

$\tau$ *stellt also eine* **Abzählung** *von* $\Sigma^*$ *dar.*

**f)** *Die Funktion* $\tau$ *und ihre Umkehrfunktion* $\nu = \tau^{-1}$ *sind bijektiv und berechenbar (siehe Abschnitt 1.4). Es ist* $\nu(i) = w$ *genau dann, wenn* $w$ *das* $i$*-te Wort in der lexikografischen Anordnung von* $\Sigma^*$ *ist.*

**g)** *Für den Fall* $\Sigma = \mathbb{B}$ *ergibt sich die folgende lexikografische Anordnung der Bitfolgen:*

$$(\varepsilon, 0), (0, 1), (1, 2), (00, 3), (01, 4), (10, 5), (11, 6), (000, 7), \dots \tag{1.6}$$

*Es ist z.B.* $\nu(62) = 11111$ *die 62. Bitfolge in dieser Anordnung. Die Abbildung* $\nu : \mathbb{N}_0 \to \mathbb{B}^*$ *stellt also neben der üblichen Dualcodierung eine weitere Codierung natürlicher Zahlen dar.* □

Wir nennen eine Sprache $L \subseteq \Sigma^*$ **präfixfrei** genau dann, wenn kein Wort in $L$ echtes Präfix eines anderen Wortes in $L$ ist.

**Bemerkung 1.2** **a)** *Ist eine Sprache* $L \subseteq \Sigma^*$ *nicht präfixfrei und* $\& \notin \Sigma$, *dann ist die Sprache* $L_\& = L \circ \{\&\}$ *präfixfrei.*

**b)** *Die Menge der Programme, die in einer Programmiersprache existieren, ist in sehr vielen Fällen präfixfrei, weil etwa deren Syntax eine feste Ende-Anweisung vorsieht. While-Programme enden z.B. immer mit einer* **write***-Anweisung, und diese Anweisung kommt nur als letzte Anweisung vor. Somit ist die Menge der While-Programme präfixfrei.*

**c)** *Eine präfixfreie Codierung der natürlichen Zahlen ist z.B. durch die Abbildung* $\sigma : \mathbb{N}_0 \to \mathbb{B}^*$ *definiert durch* $\sigma(n) = 1^n 0$ *gegeben.* □

## 1.3 Codierung von Bitfolgen-Sequenzen

Im Folgenden benötigen wir (präfixfreie) Codierungen von Sequenzen von Bitfolgen, etwa bei Programmen mit mehreren Eingaben. Auch solche Eingaben sollen insgesamt als Bitfolge codiert werden. Das bedeutet, dass wir zusätzliche Informationen codieren müssen mithilfe derer die einzelnen Eingaben separiert und damit erkannt werden können.

Berachten wir zunächst ein Paar $(x, y) \in \mathbb{B}^* \times \mathbb{B}^*$. Dafür können wir denselben „Trick“ verwenden, den wir im vorigen Abschnitt schon verwendet haben, um Sonderzeichen, hier das Komma zur Trennung von $x$ und $y$, zu codieren: Die Bits der Bitfolge von $x$ werden verdoppelt, das Komma wird durch $01$ oder $10$ codiert und anschließend folgen die Bits von $y$. Eine solche Codierung bezeichnen wir im Folgenden mit $\langle\, x, y \,\rangle$. Es ist also z.B. $\langle\, 01011, 11 \,\rangle = 0011001111\,01\,11$. Für die Länge dieser Codierung gilt:

$$|\langle\, x, y \,\rangle| = 2\,|x| + 2 + |y| = 2(|x| + 1) + |y| \tag{1.7}$$

Wir können diese Art der Codierung auf Folgen $x_i \in \mathbb{B}^*$, $2 \leq i \leq k$, verallgemeinern, indem wir Folgendes festlegen:

$$\langle\, x_1, \ldots, x_k \,\rangle_k = \langle\, x_1, \langle\, x_2, \langle \ldots, \langle\, x_{k-1}, x_k \,\rangle \ldots \rangle \,\rangle \,\rangle \tag{1.8}$$

Dies können wir rerursiv wie folgt definieren:

$$\langle\, x_1, \ldots, x_k \,\rangle_k = \begin{cases} \langle\, x_1, x_2 \,\rangle\,, & k = 2 \\ \big\langle\, x_1, \langle\, x_2, \ldots, x_k \,\rangle_{k-1} \big\rangle\,, & k \geq 3 \end{cases} \tag{1.9}$$

Die Bits der Folgen $x_i$, $1 \leq i \leq k-1$, werden verdoppelt, die Kommata werden jeweils durch $01$ oder $10$ codiert, und die letzte Folge $x_k$ wird unverändert angefügt. Es ist also z.B. $\langle\, 11, 010, 0, 110 \,\rangle_4 = 1111\,01\,001100\,01\,00\,01\,110$.

Ohne Angabe der Stelligkeiten ist diese Codierung nicht eindeutig. So haben z.B. die beiden Sequenzen $\langle\, 1, 11011 \,\rangle_2$ und $\langle\, 1, 1, 1 \,\rangle_3$ dieselbe Codierung $110111011$. Die Angabe der Stelligkeit $k$ als Dualzahl $dual(k)$ benötigt $\lfloor \log k \rfloor$ Bits; diese fügen wir vor dem verdoppelten $x_1$, ebenfalls durch $01$ (oder $10$) getrennt, ein. Für unsere drei Beispiele gilt damit:

$$\begin{aligned} \langle\, 11, 010, 0, 110 \,\rangle_4 &= 100\,01\,1111\,01\,001100\,01\,00\,01\,110 \\ \langle\, 1, 11011 \,\rangle_2 &= 10\,01\,11\,01\,11011 \\ \langle\, 1, 1, 1 \,\rangle_3 &= 11\,01\,11\,01\,11\,01\,1 \end{aligned}$$

Die Gesamtlänge der Codierung (1.8) ist gleich

$$|\langle\, x_1, \ldots, x_k \,\rangle| = 2 \sum_{i=1}^{k-1} (\, |x_i| + 1 \,) + |x_k| \tag{1.10}$$

bzw. bei Angabe der Stelligkeit gleich

$$|\langle\, x_1, \ldots, x_k \,\rangle| = 2 \sum_{i=1}^{k-1} (\, |x_i| + 1 \,) + |x_k| + \lfloor \log k \rfloor$$

Die Codierung $\langle\, \cdot \,\rangle : (\mathbb{B}^*)^k \to \mathbb{B}^*$ ist nicht präfixfrei, denn z.B. ist $\langle\, 01011, 11 \,\rangle$ ein Präfix von $\langle\, 01011, 1 \,\rangle$.

Eine weitere Art der Codierung von Bitfolgen-Sequenzen kann wie folgt durchgeführt werden: Für eine Bitfolge $x \in \mathbb{B}^+$ definieren wir zunächst

$$\overline{x} = 1^{|x|}0x \tag{1.11}$$

Es ist z.B. $\overline{01011} = 11111\,0\,01011$. Diese Codierung stellt also der Bitfolge $x$ ihre Länge in „Bierdeckelnotation“ getrennt durch $0$ voran. Es gilt $|\overline{x}| = 2\,|x| + 1$. Mithilfe dieser Codierung können wir nun eine Sequenz $x_1, \ldots, x_k$ mit $x_i \in \mathbb{B}^*$, $1 \leq i \leq k$, codieren durch

$$\overline{x_1, \ldots, x_k} = \overline{x_1} \ldots \overline{x_k} \tag{1.12}$$

Es ist z.B. $\overline{11, 010, 0, 110} = 11\,0\,11\ 111\,0\,010\ 1\,0\,0\ 111\,0\,110$. Diese Codierung ist präfixfrei für alle Sequenzen mit derselben Anzahl von Komponenten.

Ihre Gesamtlänge ist gleich

$$|\overline{x_1, \ldots, x_k}| = k + 2 \sum_{i=1}^{k} |x_i| \tag{1.13}$$

ist. Wir können bei dieser Codierung etwas an Länge einsparen, wenn wir die letzte Folge $x_k$ der Sequenz – wie bei der Codierung $\langle \cdot \rangle$ unverändert lassen. Das heißt, für $k \geq 2$ legen wir anstelle von (1.12)

$$\overline{\overline{x_1, \ldots, x_k}} = \overline{x_1} \ldots \overline{x_{k-1}} x_k \tag{1.14}$$

fest. Es ist dann $\overline{\overline{11, 010, 0, 110}} = 11\,0\,11\ 111\,0\,010\ 1\,0\,0\ 110$. Diese Codierung ist allerdings nicht mehr präfixfrei, was das folgende Beispiel zeigt: $\overline{\overline{01011, 1}} = 11111\,0\,01011\ 1$ ist ein Präfix von $\overline{\overline{01011, 11}} = 11111\,0\,01011\ 11$.

Die Gesamtlänge dieser Codierung ist

$$\left|\overline{\overline{x_1, \ldots, x_k}}\right| = k - 1 + 2 \sum_{i=1}^{k-1} |x_i| + |x_k| \tag{1.15}$$

Die Codewörter sind bei der Codierung gemäß (1.12) um $|x_k| + 1$ länger als die Codierung durch (1.14).

Für die Differenz der Längen der Codierung (1.8) und (1.14) gilt:

$$|\langle\, x_1, \ldots, x_k \,\rangle| - \left|\overline{\overline{x_1, \ldots, x_k}}\right| = k - 1 = \mathsf{O}(k) \tag{1.16}$$

Die Codewörter sind bei der Codierung gemäß (1.8) in der Größenordnung der Sequenzenlängen länger als die entsprechenden Codewörter bei der Codierung (1.14).

Eine weitere Einsparmöglichkeit bei den Codewortlängen ergibt sich, wenn wir die Längen der einzelnen Komponenten $x_i$ nicht durch $1^{|x_i|}$ codieren sondern mithilfe ihrer Dualdarstellungen $dual(|x_i|)$. Für die Länge dieser Darstellungen gilt nämlich: $|dual(|x_i|)| = \lfloor \log |x_i| \rfloor + 1 \ll |x_i|$.

Zur Vereinfachung der Darstellungen führen wir die Funktion $d : \mathbb{B}^* \to \mathbb{B}^*$ ein, welche einen Bitstring $x$ umwandelt in einen String, in dem alle Bits von $x$ verdoppelt sind:

$$\begin{aligned} d(\varepsilon) &= \varepsilon \\ d(aw) &= aad(w), \ \text{ für alle } \ a \in \mathbb{B},\ w \in \mathbb{B}^* \end{aligned} \tag{1.17}$$

Damit gilt z.B. für $k \geq 2$:

$$\langle\, x_1, \ldots, x_{k-1}, x_k \,\rangle = d(x_1)\,01 \ldots 01\, d(x_{k-1})\,01\, x_k \tag{1.18}$$

Wir verändern nun die Codierung (1.8) wie folgt:

$$\langle\, x_1, \ldots, x_k \,\rangle = d(\mathit{dual}(|x_1|))\, 01\, x_1 \;\ldots\; d(\mathit{dual}(|x_k|))\, 01\, x_k \tag{1.19}$$

Mit dieser Codierung ist z.B.

$$\langle\, 11, 010, 0, 110 \,\rangle = 1100\,01\,11\;\; 1111\,01\,010\;\; 11\,01\,0\;\; 1111\,01\,110$$

Als Codelänge von (1.19) ergibt sich

$$|\langle\, x_1, \ldots, x_k \,\rangle| = \sum_{i=1}^{k} (\, 2\, (\, \lfloor \log |x_i| \rfloor + 1\,) + 2 + |x_i|\,) \tag{1.20}$$

Analog kann die Codierung (1.12) wie folgt geändert werden:

$$\overline{x_1, \ldots, x_k} = \overline{\mathit{dual}(|x_1|)}\, x_1 \;\ldots\; \overline{\mathit{dual}(|x_k|)}\, x_k \tag{1.21}$$

Hiermit ergibt sich z.B.

$$\overline{11, 010, 0, 110} = 11\,0\,10\,11\;\; 11\,0\,11\,010\;\; 1\,0\,1\,0\;\; 11\,0\,11\,110$$

Als Länge für die Codierung (1.21) ergibt sich

$$|\overline{x_1, \ldots, x_k}| = \sum_{i=1}^{k} (\, 2\, (\, \lfloor \log |x_i| \rfloor + 1\,) + 1 + |x_i|\,) \tag{1.22}$$

Die Codierungen (1.19) und (1.21) sind präfixfrei.

## 1.4 Berechenbarkeit

Wir werden im Folgenden von der Berechenbarkeit von Funktionen $f : \mathbb{N}_0^k \to \mathbb{N}_0$, $k \geq 0$, sprechen und Algorithmen bzw. Programme verwenden, um diese zu berechnen. Da für jedes $k \geq 2$ die Menge der $k$-Tupel $\mathbb{N}_0^k$ bijektiv auf die Menge der natürlichen Zahlen $\mathbb{N}_0$ abgebildet werden kann – etwa mithilfe der berechenbaren Cantorschen $k$-Tupelfunktion –, werden wir die Angabe der Stelligkeit, falls diese zum Verständnis nicht nötig ist, weglassen; wir betrachten also in der Regel nur Funktionen $f : \mathbb{N}_0 \to \mathbb{N}_0$ oder, falls in der Darstellung die Art der Eingaben, wie z.B. Programme und Eingaben dafür, unterschieden werden soll, Funktionen $f : \mathbb{N}_0^2 \to \mathbb{N}_0$. Äquivalente Berechenbarkeitskonzepte für solche Funktionen sind z.B. Turingmaschinen, While-Programme und $\mu$-rekursive Funktionen. Diese legen jeweils wegen

ihrer Äquivalenz untereinander dieselbe Klasse $\mathcal{P}$ von Funktionen fest. Wir nennen diese Klasse die **Klasse der partiell berechenbaren Funktionen**.[6] Gemäß der Churchschen These berechnen diese Funktionen genau die im intuitiven Sinn berechenbaren Funktionen $f : \mathbb{N}_0 \to \mathbb{N}_0$.

Wir nennen eine Menge $M \subseteq \mathbb{N}_0$ **entscheidbar** genau dann, wenn ihre **charakteristische Funktion** $\chi_M : \mathbb{N}_0 \to \mathbb{B}$ definiert durch

$$\chi_M(x) = \begin{cases} 1, & x \in M \\ 0, & x \notin M \end{cases}$$

berechenbar ist. Eine Menge $M \subseteq \mathbb{N}_0$ heißt **rekursiv aufzählbar**, falls sie leer ist, oder eine total berechenbare Funktion $f : \mathbb{N}_0 \to M$ existiert mit $W(f) = M$. Es gilt: Durch $f$ wird jeder natürlichen Zahl ein Element aus $M$ zugeordnet, und allen Elementen von $M$ wird eine Zahl zugeordnet (da $f$ nicht injektiv sein muss, können einem Element aus $M$ mehrere Zahlen zugeordnet sein). Es gilt also $M = \{f(0), f(1), f(2), \ldots\}$; die Elemente von $M$ werden durch $f$ nummeriert und durch diese Nummerierung aufgezählt. Man kann zeigen, dass ein Menge $M \subseteq \mathbb{N}_0$ genau dann entscheidbar ist, wenn sie und ihr Komplement $\overline{M} = \mathbb{N}_0 - M$ rekursiv aufzählbar sind.

Falls wir kein spezielles Berechenbarkeitskonzept verwenden, benennen wir ein solches allgemein mit $\mathcal{M}$. Wir betrachten $\mathcal{M}$ als Programmiersprache und nennen die Elemente von $\mathcal{M}$ Programme oder auch Algorithmen. Für ein Programm $A \in \mathcal{M}$ bezeichnen wir mit $\Phi_A^{(\mathcal{M})}$ die von $A$ berechnete Funktion.[7] Wegen Bemerkung 1.2 können wir voraussetzen, dass $\mathcal{M}$ präfixfrei ist. Unabhängig vom konkreten Berechenbarkeitskonzept werden die Programme von $\mathcal{M}$ über einem geeigneten Aphabet $\Sigma^*$ nach eindeutig festgelegten Syntaxregeln gebildet. Mithilfe etwa der Funktion $\tau$ (siehe Bemerkung 1.1 e, f) können wir den Programmen Nummern zuordnen. Da im Allgemeinen nicht alle Elemente von $\Sigma^*$ Programme darstellen, gibt es Nummern $i \in \mathbb{N}$, denen kein Programm zugeordnet ist. Es sollen aber alle Zahlen aus $\mathbb{N}_0$ Nummer eines Programms sein. Das erreichen wir dadurch, indem wir allen Zahlen $i \in \mathbb{N}_0$, denen unter $\tau$ kein Programm zugeordnet wird, irgend ein (Default-) Programm zuordnen; wir nennen dieses $\mathcal{E}$. Damit definieren wir $\xi : \mathbb{N}_0 \to \mathcal{M}$

[6] Wir nennen $\mathcal{P}$ auch die Klasse der **partiell rekursiven Funktionen**. Mit $\mathcal{R} \subset \mathcal{P}$ bezeichnen wir die KLasse der **total berechenbaren** oder **total rekusiven Funktionen**. Wenn wir von rekursiven Fuktionen sprechen, meinen wir partiell rekursive Funktionen.

[7] Falls es aus dem Zusammenhang klar ist, auf welche Programmiersprache $\mathcal{M}$ sich $\Phi$ bezieht, schreiben wir anstelle von $\Phi_A^{(\mathcal{M})}$ nur $\Phi_A$.

durch

$$\xi(i) = \begin{cases} \nu(i), & \text{falls } \nu(i) \in \mathcal{M} \\ \mathcal{E}, & \text{sonst} \end{cases} \tag{1.23}$$

$\nu$ ist total berechenbar und mithilfe der Syntaxregeln von $\mathcal{M}$ kann überprüft werden, ob $\nu(i)$ ein syntaktisch korrektes Programm $A \in \mathcal{M}$ ist. Somit ist auch $\xi$ total berechenbar. Mithilfe von $\xi$ erreichen wir eine rekursive Aufzählung $\varphi : \mathbb{N}_0 \to \mathcal{P}$ der berechenbaren Funktionen durch

$$\varphi(i) = f \text{ genau dann, wenn } \Phi_{\xi(i)} = f \tag{1.24}$$

ist, d.h., wenn die Funktion $f \in \mathcal{P}$ von dem Programm $A = \xi(i)$ mit der Nummer $i$ berechnet wird; $i$ wird auch **Index** oder **Gödelnummer** [8] von $A$ genannt. Das System $(\mathbb{N}_0, \mathcal{P}, \varphi)$ stellt quasi eine (abstrakte) Programmiersprache dar: Jede Nummer $i \in \mathbb{N}_0$ repräsentiert ein Programm $\nu(i) = A \in \mathcal{M}$; $i$ kann als Quellcode (Syntax) von $A$ in der Sprache $(\mathbb{N}_0, \mathcal{P}, \varphi)$ aufgefasst werden, und $\varphi$ ordnet jedem Programm $i$ mit $\nu(i) = A$ seine Bedeutung (Semantik) zu, nämlich die von $A$ berechnete Funktion $f = \Phi_A$. Wir nennen $A$ auch das $i$-te Programm und $f$ die $i$-te berechenbare Funktion. Die Nummerierung $(\mathbb{N}_0, \mathcal{P}, \varphi)$ wird auch **Gödelisierung** von $\mathcal{P}$ bzw. Gödelisierung von $\mathcal{M}$ genannt.

$\varphi(i)$ ist eine Funktion $f$, die auf Argumente $x \in \mathbb{N}_0$ angewendet wird, was wir mit $\varphi(i)(x) = f(x)$ schreiben müssten. Um doppelte Argumente zu vermeiden, schreiben wir $\varphi_i$ anstelle von $\varphi(i)$ und damit $\varphi_i(x) = f(x)$. [9]

Die unter die Churchsche These fallenden Berechenbarkeitskonzepte erfüllen zwei – für das Programmieren sehr wichtige – Anforderungen:

(1) die Existenz eines universellen Programms sowie

(2) effektives Programmieren.

Die Anforderung (1) bedeutet, dass eine Programmiersprache $\mathcal{M}$ ein **universelles Programm** $U$ enthält, welches alle anderen Programme auf deren Eingaben ausführen kann. Dazu müssen die Programme $A \in \mathcal{M}$ noch geeignet

[8] Benannt nach Kurt Gödel (1906 – 1978), einem österreichischen Mathematiker (ab 1947 Staatsbürger der USA), der zu den größten Logikern der Neuzeit gerechnet wird. Er leistete fundamentale Beiträge zur Logik und zur Mengenlehre (im Übrigen mit wesentlicher Bedeutung für die Informatik) sowie interessante Beiträge zur Relativitätstheorie.

[9] Bei dieser Schreibweise ist $i$ in der Tat ein Index.

codiert werden, damit sie von $U$ ausgeführt werden können. Wir bezeichnen eine solche Codierung mit $\langle A \rangle$ (hierauf gehen wir im Folgenden noch näher ein). Für $U$ gilt also:

$$\Phi_U(\langle A \rangle, x) = \Phi_A(x) \text{ für alle } A \in \mathcal{M} \text{ und } x \in \mathbb{N}_0 \tag{1.25}$$

Für die Nummerierung $(\mathbb{N}_0, \mathcal{P}, \varphi)$ von $\mathcal{M}$ bedeutet das, dass es eine rekursive **universelle Funktion** $u_\varphi \in \mathcal{P}$ gibt mit:[10]

$$u_\varphi(i, x) = \varphi_i(x) \text{ für alle } i, x \in \mathbb{N}_0 \tag{1.26}$$

Die Anforderung (2) bedeutet, dass in der Programmiersprache $\mathcal{M}$ bereits existierende Programme genutzt werden können, um daraus neue Programme zu erzeugen. Seien $A$ und $B$ zwei Programme von $\mathcal{M}$, dann kann man diese beiden sicher zu einem Programm $C \in \mathcal{M}$ zusammensetzen, welches zunächst $B$ auf eine Eingabe $x \in \mathbb{B}^*$ und dann $A$ auf das Ergebnis $\Phi_B(x)$ ausführt. Falls $\Phi_B(x)$ nicht definiert ist, dann liefert natürlich die Zusammensetzung von $A$ und $B$ für dieses $x$ kein Ergebnis. Ebenso liefert die Zusammensetzung kein Ergebnis, wenn $\Phi_B(x)$ zwar definiert ist, aber $A$ angewendet auf das Ergebnis kein Resultat liefert. Insgesamt ergibt sich also durch die Hintereinanderausführung $A \circ B$ der beiden Programme $A$ und $B$ eine wohldefinierte **Komposition**:

$$\Phi_{A \circ B} = \Phi_A \circ \Phi_B \tag{1.27}$$

Daraus folgt: Ist $\xi(i) = A$ und $\xi(j) = B$, dann ergibt sich durch $\tau(\xi(i) \circ \xi(j))$ eine Nummer für die Komposition $A \circ B$. Wenn wir nun die Funktion $comp : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ durch

$$comp(i, j) = \tau(\xi(i) \circ \xi(j)) \tag{1.28}$$

definieren, dann gilt für die Nummerierung $(\mathbb{N}_0, \mathcal{P}, \varphi)$: Es existiert eine total berechenbare Funktion $comp : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$, so dass

$$\varphi_{comp(i,j)} = \varphi_i \circ \varphi_j \tag{1.29}$$

für alle $i, j \in \mathbb{N}_0$ gilt. In $\mathcal{M}$ existiert also ein Programm $C$ mit der Nummer $\xi(comp(i, j))$, d.h. mit $C = \xi(comp(i, j))$, welches zwei beliebige Programme aus $\mathcal{M}$ zu deren Komposition zusammensetzt.

[10] In der Literatur heißen die Sätze, welche die Existenz solcher Funktionen garantieren, **utm-Theoreme**; utm ist die Abkürzung für *Universal Turing Machine*.

Dieses Konzept führt dazu, dass Eingaben auch als Programme interpretiert werden können. Damit kann in der Nummerierung $(\mathbb{N}_0, \mathcal{P}, \varphi)$ von $\mathcal{M}$ die Anforderung (2) noch allgemeiner formuliert und bewiesen werden: Es gibt eine total berechenbare Funktion $s \in \mathcal{R}$ mit

$$\varphi_i(x_1, \ldots, x_m, y_1, \ldots, y_n) = \varphi_{s(i,x_1,\ldots,x_m)}(y_1, \ldots, y_n) \tag{1.30}$$

für alle $i, x_1, \ldots, x_m, y_1, \ldots, y_n \in \mathbb{N}_0$.[11] Es gibt also ein Programm, welches $s$ berechnet, d.h., welches die Eingaben $x_1, \ldots, x_m$ als Nummern von Programmen ansieht, aus diesen und dem Programm $i$ ein Programm berechnet, welches die Nummer $s(i, x_1, \ldots, x_m)$ besitzt. Dieses Programm kann dann auf die Eingaben $y_1, \ldots, y_n$ ausgeführt werden.[12]

**Bemerkung 1.3** *Mithilfe des s-m-n-Theorems können wir die obigen Vorüberlegungen zur Komposition von Programmen wie folgt darstellen: Sei* $h : \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ *definiert durch* $h(i, j, x) = \varphi_i \circ \varphi_j(x)$. *Dann ist* $h$ *berechenbar, also ist* $h \in \mathcal{P}$. $h$ *besitzt also einen Index* $k$, *d.h. es ist* $h(i, j, x) = \varphi_k(i, j, x)$. *Gemäß dem s-m-n-Theorem gibt es* $s \in \mathcal{R}$ *mit* $\varphi_k(i, j, x) = \varphi_{s(k,i,j)}(x)$. $s$ *bindet also mit der Information* $k$ *für Komposition beliebige Programme* $i$ *und* $j$ *zu neuen Programmen zusammen, welche die Hintereinanderausführung von diesen realisieren.* □

Wir nennen Nummerierungen bzw. Programmiersprachen, die das utm- und das s-m-m-Theorem erfüllen, **vollständig**.

Wie beim utm-Theorem oben erwähnt, müssen auch beim s-m-n-Theorem noch geeignete Codierungen berücksichtigt werden. Im Hinblick auf die folgenden Kapitel sollen sowohl Programme $A \in \mathcal{M}$ bzw. ihre Indizes $i \in \mathbb{N}_0$ als auch Eingaben $x, y \in \mathbb{N}_0$ über $\mathbb{B}$ codiert werden. Da es sich dabei um Sequenzen von Programmen bzw. Sequenzen von Indizes sowie um Sequenzen von Eingaben handeln kann, können wir dazu die im vorigen Abschnitt betrachteten Codierungen verwenden. Zunächst sollen einzelne Indizes $i \in \mathbb{N}_0$ sowie einzelne Eingaben $x \in \mathbb{N}_0$ als Bifolgen codiert werden, z.B. durch $dual(i)$ bzw. durch $dual(x)$.

Alle im Abschnitt 1.3 vorgestellten Codierungen von Bitfolgen und deren Decodierungen sind berechenbar und damit auch die Transformationen zwischen

[11] In der Literatur heißen die Sätze, welche die Existenz solcher Funktionen garantieren, **s-m-n-Theoreme**.

[12] Programme, die die Funktion $s$ realisieren, werden in praktisch verfügbaren Systemen auch **Binder** oder **Linker** genannt.

den einzelnen Codierungen. Die Transformationen können so programmiert werden, dass jeweils die Länge der Programme unabhängig von der Länge der Eingaben (Sequenzen von Bitfolgen) ist (siehe auch 3.4). Insofern sind die folgenden Darstellungen unabhängig von der gewählten Codierung. Wir werden für Sequenzen von Bitfolgen die Codierung (1.8) verwenden. Dabei werden wir aus schreibtechnischen Gründen (runde) Klammern, falls sie für das Verständnis nicht unbedingt notwendig sind, weglassen, d.h. wir schreiben dann $f \langle x_1, \ldots, x_k \rangle$ anstelle von $f(\langle x_1, \ldots, x_k \rangle)$. Damit können wir das utm-Theorem (1.26) umformulieren: Zur Nummerierung $(\mathbb{N}_0, \mathcal{P}, \varphi)$ existiert eine Funktion $u_\varphi \in \mathcal{P}$ mit

$$u_\varphi \langle i, x \rangle = \varphi_i(x) \text{ für alle } i, x \in \mathbb{N}_0 \tag{1.31}$$

Ist $\xi(i) = A$, dann schreiben wir für (1.31) auch

$$\Phi_U \langle A, x \rangle = \Phi_A(x) \tag{1.32}$$

oder auch

$$U \langle A, x \rangle = A(x) \tag{1.33}$$

Analog zu (1.31) können wir das s-m-n-Theorem (1.30) umformulieren zu: Es gibt eine total berechenbare Funktion $s \in \mathcal{R}$ mit

$$\varphi_i \langle x_1, \ldots, x_m, y_1, \ldots, y_n \rangle = \varphi_{s\langle i, x_1, \ldots, x_m \rangle} \langle y_1, \ldots, y_n \rangle \tag{1.34}$$

für alle $i, x_1, \ldots, x_m, y_1, \ldots, y_n \in \mathbb{N}_0$.

Eine Folgerung des **Äquivalenzsates von Rogers** ist, dass es zu vollständigen Programmiersprachen **Übersetzer** gibt, die Programme aus der einen in äquivalente Programme der anderen Sprache übersetzen. Für zwei vollständie Nummerierungen $(\mathbb{N}_0, \mathcal{P}, \varphi)$ und $(\mathbb{N}_0, \mathcal{P}, \psi)$ existieren also total berechenbare Funktionen $t_{\varphi\to\psi}, t_{\psi\to\varphi} \in \mathcal{R}$ mit $\varphi_i = \psi_{t_{\varphi\to\psi}(i)}$ bzw. mit $\psi_j = \varphi_{t_{\psi\to\varphi}(j)}$ für alle $i, j \in \mathbb{N}_0$.

Analog existieren zu zwei vollständigen Programmiersprachen $\mathcal{M}_1$ und $\mathcal{M}_2$ Programme $T_{1\to2} \in \mathcal{M}_2$ und $T_{2\to1} \in \mathcal{M}_1$ mit

$$\Phi_A^{(\mathcal{M}_1)} = \Phi_{T_{1\to2}(A)}^{(\mathcal{M}_2)} \text{ für alle } A \in \mathcal{M}_1 \tag{1.35}$$

bzw. mit

$$\Phi_B^{(\mathcal{M}_2)} = \Phi_{T_{2\to1}(B)}^{(\mathcal{M}_1)} \text{ für alle } B \in \mathcal{M}_2 \tag{1.36}$$

# 2 Kolmogorov-Komplexität

Ein Ansatz für die Beschreibung der weiter oben beispielhaft betrachteten Komplexität von Bitfolgen ist die *Kolmogorov-Komplexität*. Dazu legen wir eine vollständige Programiersprache zugrunde, d.h. eine, die das utm- und das s-m-n-Theorem erfüllt. Wir können also z.B. Turingmaschinen wählen oder While-Programme oder $\mu$-rekursive Funktionen. Wir nennen das gewählte Berechenbarkeitskonzept (wie im vorigen Abschnitt) allgemein $\mathcal{M}$.

Sei $A \in \mathcal{M}$, dann ist die Funktion $\mathcal{C}_A : \mathbb{B}^* \to \mathbb{N}_0 \cup \{\infty\}$ definiert durch[13]

$$\mathcal{C}_A(x) = \begin{cases} \left|\min_{\Phi_A^{-1}(x)}\right|, & \text{falls } \Phi_A^{-1}(x) \neq \emptyset \\ \\ \infty, & \text{sonst} \end{cases} \tag{2.1}$$

$\mathcal{C}_A(x)$ ist also die Länge des kürzesten Wortes $y$, woraus das Programm $A$ die Ausgabe $x$ berechnet, falls ein solches $y$ existiert.

In dieser Definition hängt $\mathcal{C}_A(x)$ nicht nur von $x$, sondern auch von dem gewählten Programm $A$ ab. Der folgende Satz besagt, dass wir von dieser Abhängigkeit absehen können.

**Satz 2.1** *Sei $U$ ein universelles Programm der vollständigen Programmiersprache $\mathcal{M}$. Dann existiert zu jedem Programm $A \in \mathcal{M}$ eine Konstante $c_A$, so dass für alle $x \in \mathbb{B}^*$*

$$\mathcal{C}_U(x) \leq \mathcal{C}_A(x) + c_A = \mathcal{C}_A(x) + \mathsf{O}(1)$$

*gilt.*

**Beweis** *Es sei $\Phi_A^{-1}(x) \neq \emptyset$. Für alle $y \in \Phi_A^{-1}(x)$ gilt $\Phi_A(y) = x$ und damit $\Phi_U \langle A, y \rangle = x$. Hieraus folgt $\mathcal{C}_U(x) \leq 2\,(\,|\langle A \rangle| + 1\,) + |y|$. Dieses gilt auch für $y = \min_{\Phi_A^{-1}(x)}$. Es folgt $\mathcal{C}_U(x) \leq \left|\min_{\Phi_A^{-1}(x)}\right| + 2\,(\,|\langle A \rangle| + 1\,)$. $A$ ist unabhängig von $x$, d.h. wir können $c_A = 2\,(\,|\langle A \rangle| + 1\,)$ als Konstante annehmen und erhalten damit die Behauptung.*

*Falls $\Phi_A^{-1}(x) = \emptyset$ ist, folgt mit teilweiser analoger Argumentation $\mathcal{C}_U(x) \leq \infty + c_A = \infty$.* □

[13] Wir benutzen im Folgenden das Symbol $\infty$ mit der Bedeutung: $\infty > n$ für alle $n \in \mathbb{N}_0$ sowie mit $n + \infty = \infty$ bzw. $n \cdot \infty = \infty$ für alle $n \in \mathbb{N}_0$.

Wir können also $\mathcal{C}_A(x)$ für jedes $A \in \mathcal{M}$ ohne wesentliche Beeinträchtigung der folgenden Betrachtungen durch $\mathcal{C}_U(x)$ ersetzen. Es bleibt jetzt noch die Abhängigkeit von der gewählten Programmiersprache $\mathcal{M}$. Der folgende Satz besagt, dass wir auch diese Abhängigkeit vernachlässigen können. Als Vorbereitung auf diesen Satz ziehen wir zunächst zwei Folgerungen aus Satz 2.1.

**Folgerung 2.1** *Es seien $\mathcal{M}_1$ und $\mathcal{M}_2$ vollständige Programmiersprachen.*

**a)** *$U_1$ sei ein universelles Programm von $\mathcal{M}_1$. Dann existiert zu jedem Programm $B \in \mathcal{M}_2$ eine Konstante $c$ mit $\mathcal{C}_{U_1}(x) \leq \mathcal{C}_B(x) + c$ für alle $x \in \mathbb{B}^*$.*

**b)** *Seien $U_1$ und $U_2$ universelle Programme von $\mathcal{M}_1$ bzw. von $\mathcal{M}_2$. Dann existiert eine Konstante $c$, so dass für alle $x \in \mathbb{B}^*$ gilt: $\mathcal{C}_{U_1}(x) \leq \mathcal{C}_{U_2}(x) + c$.*

**Beweis** **a)** *Gemäß dem Äquvalenzsatz von Rogers (1.36) existiert ein Programm $T_{2\to1} \in \mathcal{M}_1$ mit*

$$\Phi_B^{(\mathcal{M}_2)}(x) = \Phi_{T_{2\to1}(B)}^{(\mathcal{M}_1)}(x) \tag{2.2}$$

*für alle $x \in \mathbb{B}^*$. Da $T_{2\to1}(B) \in \mathcal{M}_1$ ist, gibt es gemäß Satz 2.1 eine Konstante $c$ mit $\mathcal{C}_{U_1}(x) \leq \mathcal{C}_{T_{2\to1}(B)}(x) + c$. Hieraus folgt mit (2.2) die Behauptung.*

**b)** *folgt unmittelbar aus a), wenn wir $U_2$ für $B$ einsetzen.* □

**Satz 2.2** *Es seien $\mathcal{M}_1$ und $\mathcal{M}_2$ zwei vollständige Programmiersprachen, und $U_1$ und $U_2$ seien universelle Programme von $\mathcal{M}_1$ bzw. von $\mathcal{M}_2$. Dann existiert eine Konstante $c_{1,2}$, so dass für alle $x \in \mathbb{B}^*$ gilt:*

$$|\mathcal{C}_{U_1}(x) - \mathcal{C}_{U_2}(x)| \leq c_{1,2} = \mathsf{O}(1)$$

**Beweis** *Gemäß Folgerung 2.1 b) gibt es Konstanten $c_1$ und $c_2$, so dass $\mathcal{C}_{U_1}(x) \leq \mathcal{C}_{U_2}(x) + c_1$ bzw. $\mathcal{C}_{U_2}(x) \leq \mathcal{C}_{U_1}(x) + c_2$ für alle $x \in \mathbb{B}^*$ gelten. Damit ist $\mathcal{C}_{U_1}(x) - \mathcal{C}_{U_2}(x) \leq c_1$ bzw. $\mathcal{C}_{U_2}(x) - \mathcal{C}_{U_1}(x) \leq c_2$. Für $c_{1,2} = \max\{c_1, c_2\}$ gilt dann die Behauptung.* □

Die Konstante $c_{1,2}$ misst quasi den Aufwand für den (größeren der beiden) Übersetzer zwischen $\mathcal{M}_1$ und $\mathcal{M}_2$.

Wegen der Sätze 2.1 und 2.2 unterscheiden sich die Längenmaße $\mathcal{C}$ für Programme und universelle Programme aus einer und auch aus verschiedenen Programmiersprachen jeweils nur um eine additive Konstante. Diesen

Unterschied wollen wir als unwesentlich betrachten. Insofern sind die folgenden Definitionen unabhängig von der gewählten Programmiersprache und unabhängig von dem in dieser Sprache gewählten universellen Programm. Deshalb wechseln wir im Folgenden je nachdem, wie es für die jeweilige Argumentation oder Darstellung verständlicher erscheint, das Berechnungsmodell; so verwenden wir etwa im Einzelfall Turingmaschinen, While-Programme oder $\mu$-rekursive Funktionen. Mit $\mathcal{M}$ bezeichnen wir im Folgenden unabhängig vom Berechnungsmodell die Menge aller Programme, also z.B. die Menge aller Turingmaschinen, die Menge aller While-Programme bzw. die Menge aller $\mu$-rekursiven Funktionen. Mit $U$ bezeichnen wir universelle Programme in diesen Berechnungsmodellen, also universelle Turingmaschinen, universelle While-Programme bzw. universelle Funktionen, und mit $U_0$ das jeweils kürzeste universelle Programm:

$$U_0 = \min \{|U| : U \in \mathcal{U}\} \tag{2.3}$$

Dabei sei $\mathcal{U}$ die Menge der universellen Programme in $\mathcal{M}$.

**Definition 2.1** *Sei $A$ ein Pogramm von $\mathcal{M}$ und $y \in \mathbb{B}^*$ mit $\Phi_A(y) = x$, dann heißt $b(x) = \langle A, y \rangle$* **Beschreibung** *von $x$. Sei $\mathcal{B}(x)$ die Menge aller Beschreibungen von $x$, dann ist $b^*(x) = \min_{\mathcal{B}(x)}$ die lexikografisch kürzeste Beschreibung von $x$. Wir nennen die Funktion $\mathcal{C} : \mathbb{B}^* \to \mathbb{N}_0$ definiert durch*

$$\mathcal{C}(x) = |b^*(x)|$$

*die* **Beschreibungs-** *oder* **Kolmogorov-Komplexität** *von $x$. Die Kolmogorov-Komplexität des Wortes $x$ ist die Länge der lexikografisch kürzesten Bitfolge $\langle A, y \rangle$, so dass die Turingmaschine $A$ bei Eingabe $y$ das Wort $x$ ausgibt.* □

Wir können $\mathcal{C}(x)$ als den Umfang an Informationen verstehen, welche durch die Bitfolge $x$ dargestellt wird.

**Bemerkung 2.1** *Da die lexikografische Ordnung von Bitfolgen total ist, ist die kürzeste Beschreibung $b^*(x) = \langle A, y \rangle$ einer Folge $x$ eindeutig. Außerdem gilt für eine universelle Turingmaschine $U$: $\Phi_U \langle A, y \rangle = \Phi_A(y) = x$. Deshalb können wir die Maschine $A$ und die Eingabe $y$ zu einer Maschine verschmelzen, d.h. wir betrachten $y$ als Teil der Maschine $A$ ($y$ ist quasi „fest in $A$ verdrahtet"). Diese Verschmelzung bezeichnen wir ebenfalls mit $A$. Wenn wir nun für diese Programme als Default-Eingabe das leere Wort vorsehen (das natürlich nicht codiert werden muss), gilt $\Phi_U \langle A, \varepsilon \rangle = x$. Da*

*$\varepsilon$ die Default-Eingabe ist, können wir diese auch weglassen, d.h. wir schreiben $\Phi_U \langle A \rangle$ anstelle von $\Phi_U \langle A, \varepsilon \rangle$. Mit dieser Sichtweise kann die obige Definition unwesentlich abgeändert werden zu:*

*Gilt $\Phi_U \langle A \rangle = x$, dann heißt $b(x) = \langle A \rangle$ eine* **Beschreibung** *von $x$. Sei $\mathcal{B}(x)$ die Menge aller Beschreibungen von $x$, dann ist $b^*(x) = \min_{\mathcal{B}(x)}$ die lexikografisch kürzeste Beschreibung von $x$. Wir nennen die Funktion $\mathcal{C} : \mathbb{B}^* \to \mathbb{N}_0$ definiert durch*

$$\mathcal{C}(x) = |b^*(x)|$$

*die* **Beschreibungs-** *oder* **Kolmogorov-Komplexität** *von $x$.*

*Alternativ können wir $\mathcal{C}$ auch wie folgt festlegen: Sei $U$ eine universelle Maschine für die Programmiersprache $\mathcal{M}$, dann heißt die Funktion $\mathcal{C} : \mathbb{B}^* \to \mathbb{N}_0$ definiert durch*

$$\mathcal{C}(x) = \min \{|A| : A \in \mathcal{M} \text{ mit } \Phi_U \langle A \rangle = x\}$$

**Beschreibungs-** *oder* **Kolmogorov-Komplexität** *von $x$.* □

Für natürliche Zahlen $n \in \mathbb{N}_0$ legen wir die Kolmogorov-Komplexität wie folgt fest:

$$\mathcal{C}(n) = \mathcal{C}(dual(n)) \tag{2.4}$$

# 3 Eigenschaften der Kolmogorov-Komplexität

Als erstes können wir überlegen, dass $\mathcal{C}(x)$ im Wesentlichen nach oben durch $|x|$ beschränkt ist, denn es gibt sicherlich ein Programm, was nichts anderes leistet, als nur $x$ zu erzeugen. Ein solches Programm enthält Anweisungen und bekommt $x$ als Eingabe oder enthält $x$ „fest verdrahtet“, woraus dann – in beiden Fällen – $x$ erzeugt wird. Somit ergibt sich die Länge dieses Programmes aus der Länge von $x$ und der Länge der Anweisungen, die unabhängig von der Länge von $x$ ist. Ein kürzestes Programm zur Erzeugung von $x$ ist höchstens so lang, wie das beschriebene Programm.

**Satz 3.1** *Es existiert eine Konstante $c$, so dass für alle $x \in \mathbb{B}^*$*

$$\mathcal{C}(x) \leq |x| + c = |x| + \mathsf{O}(1) \tag{3.1}$$

*gilt.*

**Beweis** *Sei $A$ eine Turingmaschine, welche die identische Funktion $id_{\mathbb{B}^*}$ berechnet, wie z.B.*

$$A = (\{0,1\}, \{0,1,\#\}, \{s\}, \delta, s, \#, \{s\})$$

*mit*

$$\delta = \{(s, a, s, a, -) \mid a \in \{0,1,\#\}\}$$

*$A$ stoppt bei jeder Eingabe sofort, d.h. es gilt $\Phi_A(x) = x$. Es ist also $b(x) = \langle A, x \rangle$ eine Beschreibung für $x$. $A$ ist unabhängig von $x$; wir können also $c = |\langle A \rangle|$ setzen. Damit haben wir $|b(x)| = c + |x|$. Mit Definition 2.1 folgt unmittelbar*

$$\mathcal{C}(x) = |b^*(x)| \leq |b(x)| = |x| + c$$

*womit die Behauptung gezeigt ist.* □

Die Information, die eine Bitfolge $x$ enthält, ist also in keinem Fall wesentlich länger als $x$ selbst – was sicherlich auch intuitiv einsichtig ist.

**Bemerkung 3.1** *Im Beweis von Satz 3.1 haben wir eine feste Maschine gewählt, die bei Eingabe $x \in \mathbb{B}^*$ die Ausgabe $x$ berechnet. Gemäß Bemerkung 2.1 können wir aber für jedes $x$ eine eigene Maschine $A_x$ konstruieren, in der $x$ fest verdrahtet ist. Sei $x = x_1 \ldots x_\ell$, $x_i \in \mathbb{B}$, $1 \leq i \leq \ell$, dann gilt z.B. für*

$$A_x = (\{0,1\}, \{0,1,\#\}, \{s_0, \ldots, s_\ell\}, \delta, s_0, \#, \{s_\ell\}) \tag{3.2}$$

*mit $\delta = \{(s_i, \#, s_{i+1}, x_{i+1}, r) \mid 0 \leq i \leq \ell - 1\}$: $\Phi_{A_x}(\varepsilon) = x$. Es ist also $b(x) = \langle A_x \rangle$ eine Beschreibung von $x$. Die Länge von $A_x$ hängt allerdings von der Länge von $x$ ab. Es gilt $|b(x)| = |x| + c$ und damit auch hier*

$$\mathcal{C}(x) = |b^*(x)| \leq |b(x)| = |x| + c$$

*Dabei ist $c$ die Länge des „Overheads“, d.h. $c$ gibt die Länge des Anteils von $A_x$ an, der unabhängig von $x$ ist. Dieser ist für alle $x$ identisch, wie z.B. die Alphabete und das Blanksymbol sowie die Sonderzeichen wie Klammern und Kommata, siehe (3.2).* □

Wie die durch den Satz 3.1 bestätigte Vermutung scheint ebenfalls die Vermutung einsichtig, dass die Folge $xx$ nicht wesentlich mehr Information enthält als $x$ alleine. Das folgende Lemma bestätigt diese Vermutung.

**Lemma 3.1** *Es existiert eine Konstante $c$, so dass für alle $x \in \mathbb{B}^*$*

$$\mathcal{C}(xx) \leq \mathcal{C}(x) + c \leq |x| + \mathsf{O}(1) \tag{3.3}$$

*gilt.*

**Beweis** *Sei $b^*(x) = \langle B \rangle \in \mathcal{M}$ die kürzeste Beschreibung von $x$, d.h. es ist $\Phi_U \langle B \rangle = x$. Sei nun $A \in \mathcal{M}$ eine Maschine, welche die Funktion $f : \mathbb{B}^* \to \mathbb{B}^*$ definiert durch $f(x) = xx$ berechnet: $\Phi_A(x) = f(x) = xx$, $x \in \mathbb{B}^*$. Damit gilt*

$$\Phi_A \langle \Phi_U \langle B \rangle \rangle = f(\Phi_U \langle B \rangle) = \Phi_U \langle B \rangle \Phi_U \langle B \rangle = xx \tag{3.4}$$

*Sei $i$ ein Index von $A$ und $u$ ein Index von $U$, dann gilt wegen (3.4)*

$$\varphi_i \circ \varphi_u \langle B \rangle = \varphi_u \langle B \rangle \varphi_u \langle B \rangle \tag{3.5}$$

*Wegen Bemerkung 1.3 gibt es dann einen Index $k$ und ein total berechenbares Programm $s$ mit*

$$\varphi_{s\langle k,i,u \rangle} \langle B \rangle = \varphi_u \langle B \rangle \varphi_u \langle B \rangle \tag{3.6}$$

*Sei nun $D = \xi(s \langle k, i, u \rangle)$, d.h. wir haben mit (3.4) - (3.6)*

$$\Phi_D \langle B \rangle = xx \tag{3.7}$$

*und damit $\Phi_U \langle D, B \rangle = xx$. Wegen $b^*(x) = \langle B \rangle$ gilt dann*

$$\Phi_U \langle D, b^*(x) \rangle = xx \tag{3.8}$$

*womit $b(xx) = \langle D, b(x^*) \rangle$ eine Beschreibung von $xx$ ist. Es folgt $|b(xx)| = c + |b^*(x)|$ mit $c = 2(|\langle D \rangle| + 1)$ (siehe (1.10)); $c$ ist konstant, da $D$ unabhängig von $x$ ist. Damit sowie mit Satz 3.1 erhalten wir*

$$\mathcal{C}(xx) = |b^*(xx)| \leq |b(xx)| = |b^*(x)| + c = \mathcal{C}(x) + c \leq |x| + \mathsf{O}(1)$$

*womit die Behauptung gezeigt ist.* □

Wie bei der Verdopplung kann man vermuten, dass auch der Informationsgehalt der Spiegelung $sp(x)$ von $x \in \mathbb{B}^*$ sich unwesentlich von dem von $x$ unterscheidet.

**Lemma 3.2** *Es existiert eine Konstante $c$, so dass für alle $x \in \mathbb{B}^*$*

$$\mathcal{C}(sp(x)) \leq \mathcal{C}(x) + c = \mathcal{C}(x) + \mathsf{O}(1) \leq |x| + \mathsf{O}(1) \tag{3.9}$$

*gilt.*

**Beweis** *Sei $b^*(x) = \langle\, B \,\rangle \in \mathcal{M}$ die kürzeste Beschreibung von $x$, d.h. es ist $\Phi_U \langle\, B \,\rangle = x$. Sei nun $A \in \mathcal{M}$ ein Programm, welches die Funktion $g : \mathbb{B}^* \to \mathbb{B}^*$ definiert durch $g(x) = sp(x)$ berechnet: $\Phi_A(x) = g(x) = sp(x)$, $x \in \mathbb{B}^*$. Dann gilt*

$$\Phi_A \langle\, \Phi_U \langle\, B \,\rangle \,\rangle = sp(\Phi_U \langle\, B \,\rangle) = sp(x) \tag{3.10}$$

*Sei $i$ ein Index von $A$ und $u$ ein Index von $U$, dann gilt wegen (3.10)*

$$\varphi_i \circ \varphi_u \langle\, B \,\rangle = sp(\varphi_u \langle\, B \,\rangle) \tag{3.11}$$

*Wegen Bemerkung 4.1 gibt es dann einen Index $k$ und ein total berechenbares Programm $s$ mit*

$$\varphi_{s\langle\, k,i,u \,\rangle} \langle\, B \,\rangle = sp(\varphi_u \langle\, B \,\rangle) \tag{3.12}$$

*Sei nun $R = \xi(s \langle\, k, i, u \,\rangle)$, d.h. wir haben mit (3.10) - (3.12)*

$$\Phi_R \langle\, B \,\rangle = sp(x) \tag{3.13}$$

*und damit $\Phi_U \langle\, R, B \,\rangle = sp(x)$. Wegen $b^*(x) = \langle\, B \,\rangle$ gilt dann*

$$\Phi_U \langle\, R, b^*(x) \,\rangle = sp(x) \tag{3.14}$$

*womit $b(sp(x)) = \langle\, R, b(x^*) \,\rangle$ eine Beschreibung von $sp(x)$ ist. Damit gilt $|b(sp(x))| = c + |b^*(x)|$ mit $c = 2(|\langle\, R \,\rangle| + 1)$ (siehe (1.10)); $c$ ist konstant, da $R$ unabhängig von $x$ ist. Damit sowie mit Satz 3.1 erhalten wir*

$$\mathcal{C}(sp(x)) = |b^*(sp(x))| \leq |b(sp(x))| = |b^*(x)| + c = \mathcal{C}(x) + c \leq |x| + \mathsf{O}(1)$$

*womit die Behauptung gezeigt ist.* □

Aus den beiden Lemmata folgt unmittelbar

**Folgerung 3.1** *Es existieren Konstanten $c$ bzw. $c'$, so dass für alle $x \in \mathbb{B}^*$*

**a)** $|\mathcal{C}(xx) - \mathcal{C}(x)| \leq c$

**b)** $|\mathcal{C}(sp(x)) - \mathcal{C}(x)| \leq c'$

*gilt. Die von $x$ unabhängigen Konstanten $c$ und $c'$ messen den Aufwand für das Verdoppeln bzw. für das Spiegeln.* □

Die beiden Funktionen $f, g : \mathbb{B}^* \to \mathbb{B}^*$ definiert durch $f(x) = xx$ bzw. $g(x) = sp(x)$ sind total und berechenbar. Ihre Werte tragen unwesentlich mehr Informationen in sich als ihre Argumente. Der folgende Satz besagt, dass diese Eigenschaft für alle total berechenbaren Funktionen gilt.

**Satz 3.2** *Sei $h : \mathbb{B}^* \to \mathbb{B}^*$ eine berechenbare Funktion, dann existiert eine Konstante $c$, so dass für alle $x \in Def(h)$*

$$\mathcal{C}(h(x)) \leq \mathcal{C}(x) + c = \mathcal{C}(x) + \mathsf{O}(1) \leq |x| + \mathsf{O}(1)$$

*gilt.*

**Beweis** *Sei $b^*(x) = \langle A \rangle$ die kürzeste Beschreibung von $x$, d.h. es ist $\Phi_U \langle A \rangle = x$. Sei $H$ ein Programm, das $h$ berechnet. Dann gilt*

$$\Phi_H \langle \Phi_U \langle A \rangle \rangle = h(\Phi_U \langle A \rangle) = h(x) \tag{3.15}$$

*Sei $\iota$ ein Index von $H$ und $u$ ein Index von $U$, dann gilt wegen (3.15)*

$$\varphi_\iota \circ \varphi_u \langle A \rangle = h(\varphi_u \langle A \rangle) \tag{3.16}$$

*Wegen Bemerkung 4.1 gibt es dann einen Index $k$ und ein total berechenbares Programm $s$ mit*

$$\varphi_{s\langle k,\iota,u \rangle} \langle A \rangle = h(\varphi_u \langle A \rangle) \tag{3.17}$$

*Sei nun $\mathcal{H} = \xi(s \langle k, \iota, u \rangle)$, d.h. wir haben mit (3.15) - (3.17)*

$$\Phi_{\mathcal{H}} \langle A \rangle = h(x) \tag{3.18}$$

*und damit $\Phi_U \langle \mathcal{H}, B \rangle = h(x)$. Wegen $b^*(x) = \langle A \rangle$ gilt dann*

$$\Phi_U \langle \mathcal{H}, b^*(x) \rangle = h(x) \tag{3.19}$$

*womit $b(h(x)) = \langle \mathcal{H}, b(x^*) \rangle$ eine Beschreibung von $h(x)$ ist. Damit gilt $|b(h(x))| = c + |b^*(x)|$ mit $c = 2(|\langle \mathcal{H} \rangle| + 1)$ (siehe (1.10)); $c$ ist konstant, da $\mathcal{H}$ unabhängig von $x$ ist. Damit sowie mit Satz 3.1 erhalten wir*

$$\mathcal{C}(h(x)) = |b^*(h(x))| \leq |b(h(x)| = |b^*(x)| + c = \mathcal{C}(x) + c \leq |x| + \mathsf{O}(1)$$

*womit die Behauptung gezeigt ist.* □

**Bemerkung 3.2** *Die Lemmata 3.1 und 3.2 sind Beispiele für die allgemeine Aussage in Satz 3.2.* □

Mithilfe der obigen Sätze und Lemmata folgt

**Folgerung 3.2** **a)** *Sind $f, g : \mathbb{B}^* \to \mathbb{B}^*$ total berechenbare Funktionen. Dann existiert eine Konstante $c$, so dass für alle $x \in \mathbb{B}^*$*

$$\mathcal{C}(g(f(x))) \leq \mathcal{C}(x) + c \leq |x| + \mathsf{O}(1)$$

*gilt.*

**b)** *Für jede monoton wachsende, bijektive, berechenbare Funktion $f : \mathbb{N}_0 \to \mathbb{N}_0$ gibt es unendlich viele $x \in \mathbb{B}^*$ mit $\mathcal{C}(x) \leq f^{-1}(|x|)$.*

**c)** *Es gibt unendlich viele $x \in \mathbb{B}^*$ mit $\mathcal{C}(x) \ll |x|$.*

**Beweis** **a)** *folgt unmittelbar aus Satz 3.2.*

**b)** *Zunächst halten wir fest, dass es zu jeder Konstante $c \in \mathbb{N}_0$ ein $n_c \in \mathbb{N}_0$ gibt, so dass $\lfloor \log n \rfloor + c \leq n$ für alle $n \geq n_c$ ist. Dann überlegen wir uns ein Programm $A$, welches bei Eingabe von $dual(n)$ die 1-Folge $x_n = 1^{f(n)}$ berechnet; es ist also $A(dual(n)) = 1^{f(n)}$ und damit $b(x_n) = \langle\, A, dual(n) \,\rangle$ eine Beschreibung für $x_n$. Es folgt*

$$|b(x_n)| = |\langle\, A, dual(n) \,\rangle| = 2(|\langle\, A \,\rangle| + 1) + |dual(n)| = c + \lfloor \log n \rfloor$$

*mit $c = 2\,|\langle\, A \,\rangle| + 3$. Damit gilt für alle $n \geq n_c$*

$$\mathcal{C}(x_n) = |b^*(x_n)| \leq |b(x_n)| = \lfloor \log n \rfloor + c \leq n = f^{-1}(f(n)) = f^{-1}(|x_n|)$$

*womit die Behauptung gezeigt ist.*

**c)** *folgt unmittelbar aus b), wenn wir $f(n) = n$ setzen und damit $x_n = 1^n$ wählen.* □

Nach den Vermutungen über die Komplexität von $xx$ bzw. von $sp(x)$, welche durch die Lemmata 3.1 und 3.2 bestätigt werden, könnte man auch vermuten, dass die Kolmogorov-Komplexität $\mathcal{C}(xx')$ einer Konkatenation von zwei Folgen $x$ und $x'$ sich als Summe von $\mathcal{C}(x)$ und $\mathcal{C}(x')$ ergibt, d.h., dass es ein $c$ gibt, so dass für alle $x, x' \in \mathbb{B}^*$ gilt: $\mathcal{C}(xx') \leq \mathcal{C}(x) + \mathcal{C}(x') + c$. Das ist aber nicht der Fall: Die Kolmogorov-Komplexität ist nicht subadditiv. Der Grund liegt darin, dass eine Information benötigt wird, die $x$ und $x'$ bzw. deren Beschreibungen voneinander abgrenzt. Dazu legen wir zunächst

$\mathcal{C}(xx') = \mathcal{C}(\langle x, x' \rangle)$ fest und schreiben auch hierfür nur $\mathcal{C}\langle x, x' \rangle$. Somit benötigen wir für $x$ die doppelte Anzahl von Bits. Damit können wir mithilfe von Beweisschritten analog zu den in den obigen Beweisen zeigen, dass eine Konstante $c$ existiert, so dass für alle $x, x' \in \mathbb{B}^*$

$$\mathcal{C}(xx') \leq 2\mathcal{C}(x) + \mathcal{C}(x') + c$$

gilt.

Wir können allerdings die Codierung von $x$ kürzer gestalten, als es durch die Verdopplung der Bits geschieht, indem wir vor $xx'$ bzw. vor $b^*(x)b^*(x')$ die Länge von $x$ bzw. die Länge von $b^*(x)$ – dual codiert – schreiben (siehe auch Abschnitt 1.3): $\langle\, dual(|x|), xx' \,\rangle$. Wir legen also $\mathcal{C}(xx') = \mathcal{C}(\langle\, dual(|x|), xx' \,\rangle)$ fest. Da die Bits von $dual(|x|)$ verdoppelt werden, ergibt sich (siehe (1.7)):

$$|\langle\, dual(|x|), xx' \,\rangle| \leq 2(\lfloor \log |x| \rfloor + 1) + |x| + |x'|$$

Falls $x'$ kürzer als $x$ sein sollte, kann man auch die Länge von $x'$ vor $xx'$ schreiben, um eine noch bessere Komprimierung zu erhalten. Diese Überlegungen führen zu folgendem Satz.

**Satz 3.3** *Es existiert eine Konstante $c$, so dass für alle $x, x' \in \mathbb{B}^*$*

$$\mathcal{C}(xx') \leq \mathcal{C}(x) + \mathcal{C}(x') + 2\log(\min\{\mathcal{C}(x), \mathcal{C}(x')\}) + c$$

*gilt.*

**Beweis** *Wir betrachten den Fall, dass $\mathcal{C}(x) \leq \mathcal{C}(x')$ ist, und zeigen*

$$\mathcal{C}(xx') \leq \mathcal{C}(x) + \mathcal{C}(x') + 2\log \mathcal{C}(x) + c' \tag{3.20}$$

*Analog kann für den Fall, dass $\mathcal{C}(x') \leq \mathcal{C}(x)$ ist,*

$$\mathcal{C}(xx') \leq \mathcal{C}(x) + \mathcal{C}(x') + 2\log \mathcal{C}(x') + c'' \tag{3.21}$$

*gezeigt werden. Aus (3.20) und (3.21) folgt dann unmittelbar die Behauptung des Satzes.*

*Sei also $\mathcal{C}(x) \leq \mathcal{C}(x')$ sowie $b^*(x) = \langle A \rangle$ und $b^*(x') = \langle B \rangle$ und damit $\Phi_U \langle A \rangle = x$ bzw. $\Phi_U \langle B \rangle = x'$. Sei $D$ ein Programm, welches zunächst testet, ob eine Bitfolge die Struktur $v\,01\,w$ mit $v \in \{00, 11\}^*$ und $w \in \mathbb{B}^*$ hat. Falls das zutrifft, testet $D$, ob $wert(d^{-1}(v)) \leq |w|$ ist, wobei $d$ die in (1.17) definierte Bit-Verdoppelungsfunktion ist. Falls auch das zutrifft, überprüft $D$,*

*ob der Präfix der Länge $wert(d^{-1}(v))$ von $w$ sowie der verbleibende Suffix Codierungen von Programmen sind. Gegebenenfalls führt $D$ beide Programme aus und konkateniert deren Ausgaben; diese Konkatenation ist die Ausgabe von $D$. In allen anderen Fällen ist $D$ für die eingegebene Bitfolge nicht definiert. Für das so definierte Programm $D$ gilt*

$$\Phi_D \langle\, dual(|\langle\, A \,\rangle|), AB \,\rangle = \Phi_U \langle\, A \,\rangle\, \Phi_U \langle\, B \,\rangle$$

*und damit*

$$\Phi_U \langle\, D, dual(|\langle\, A \,\rangle|\,, AB \,\rangle = xx'$$

*$b(xx') = \langle\, D, dual(|\langle\, A \,\rangle|), AB \,\rangle$ ist also eine Beschreibung von $xx'$. Es folgt*

$$\begin{aligned} |b(xx')| &= 2(|\langle\, D \,\rangle| + |dual(|\langle\, A \,\rangle|)| + 2) + |\langle\, AB \,\rangle| \\ &\leq 2(|\langle\, D \,\rangle| + 2) + 2(\log\lfloor |b^*(x)| \rfloor + 1) + |b^*(x)| + |b^*(x')| \end{aligned} \quad (3.22)$$

*$D$ ist unabhängig von $x$ und $x'$. So setzen wir $c' = 2(|\langle\, D \,\rangle| + 3)$ und erhalten aus (3.22)*

$$\mathcal{C}(xx') = |b^*(xx')| \leq |b(xx')| \leq \mathcal{C}(x) + \mathcal{C}(x') + 2\log\mathcal{C}(x) + c'$$

*womit (3.20) gezeigt ist.* □

**Bemerkung 3.3** *Auch wenn noch Verbesserungen möglich sein sollten, gilt jedoch, dass sich diese logarithmische Differenz zur intuitiven Vermutung generell nicht vermeiden lässt, d.h., dass $\mathcal{C}(xx') \leq \mathcal{C}(x) + \mathcal{C}(x') + c$ im Allgemeinen nicht erreichbar ist. Dieses werden wir im folgenden Kapitel in Satz 4.3 beweisen.* □

Aus dem Satz 3.1 und dessen Beweis lässt sich leicht folgern, dass

$$\mathcal{C}(x^n) \leq \mathcal{C}(x) + \mathsf{O}(1)$$

für jedes fest gewählte $n$ ist.

Ist $n$ variabel, dann kann $n$ in den Programmen zur Erzeugung von $x^n$ nicht „fest verdrahtet“ sein, sondern $n$ muss eine Eingabe für die Programme sein. Es geht also um die Berechnung der Funktion $f : \mathbb{N}_0 \times \mathbb{B}^* \to \mathbb{B}^*$ definiert durch $f(n, x) = x^n$. Diese können wir dual codieren durch $f : \mathbb{B}^* \to \mathbb{B}^*$ mit $f \langle\, dual(n), x \,\rangle = x^n$. Sei $b^*(x) = \langle\, A \,\rangle$ die kürzeste Beschreibung für $x$, womit $\Phi_U \langle\, A \,\rangle = x$ gilt. Sei $F$ ein Programm, welches – wie das oben beschriebene Programm $D$ – zunächst testet, ob eine Bitfolge die Struktur $v\,01\,w$

mit $v \in \{00, 11\}^*$ und $w \in \mathbb{B}^*$ hat. Falls das zutrifft, überprüft $F$, ob $w$ Codierung eines Programms ist. Gegebenenfalls führt $F$ dieses aus und kopiert dessen Ausgabe $(wert(d^{-1}(v)) - 1)$-mal, wobei $d$ die in (1.17) definierte Bit-Verdoppelungsfunktion ist; die Konkatenation der insgesamt $wert(d^{-1}(v))$ Kopien ist die Ausgabe von $F$. In allen anderen Fällen ist $F$ für die eingegebene Bitfolge nicht definiert. Für das so definierte Programm $F$ gilt

$$\Phi_F \langle\, dual(n), A \,\rangle = (\Phi_U \langle\, A \,\rangle)^n$$

Damit gilt

$$\Phi_U \langle\, F, dual(n), A \,\rangle = x^n$$

und $b(x^n) = \langle\, F, dual(n), b^*(x) \,\rangle$ ist eine Beschreibung von $x^n$. Es folgt

$$\begin{aligned} |b(x^n)| &= 2(|\langle\, F \,\rangle| + |dual(n)| + 2) + |\langle\, A \,\rangle| \\ &\leq 2(|\langle\, F \,\rangle| + \lfloor \log n \rfloor + 3) + |b^*(x)| \\ &= 2(|\langle\, F \,\rangle| + 3) + 2\lfloor \log n \rfloor + |b^*(x)| \end{aligned}$$

Es folgt mit $c = 2(|\langle\, F \,\rangle| + 3)$:

$$\begin{aligned} \mathcal{C}(x^n) &= |b^*(x^n)| \\ &\leq |b(x^n)| \\ &\leq |b^*(x)| + 2\lfloor \log n \rfloor + c \\ &= \mathcal{C}(x) + \mathsf{O}(\log n) \end{aligned} \tag{3.23}$$

Diese Aussage kann auch mithilfe von Satz 3.1 hergeleitet werden:

$$\begin{aligned} \mathcal{C} \langle\, dual(n), x \,\rangle &\leq |\langle\, dual(n), x \,\rangle| + \mathsf{O}(1) \\ &\leq 2(|dual(n)| + 1) + \mathcal{C}(x) + \mathsf{O}(1) \\ &\leq 2(\lfloor \log n \rfloor + 2) + \mathcal{C}(x) + \mathsf{O}(1) \\ &= \mathcal{C}(x) + \mathsf{O}(\log n) \end{aligned}$$

Wir betrachten noch den Spezialfall, dass $n$ eine Zweierpotenz ist: $n = 2^k$, $k \in \mathbb{N}$. Dann benötigen wir zur Erzeugung von $x^n$ nicht den Parameter $n$, sondern den Exponenten $k = \log n$. Dann ergibt sich aus den obigen Überlegungen

$$\begin{aligned} \mathcal{C} \langle\, dual(k), x \,\rangle &= \mathcal{C} \langle\, dual(\log n), x \,\rangle \\ &\leq |\langle\, dual(\log n), x \,\rangle| + \mathsf{O}(1) \\ &\leq 2(|dual(\log n)| + 1) + \mathcal{C}(x) + \mathsf{O}(1) \\ &\leq 2(\lfloor \log \log n \rfloor + 2) + \mathcal{C}(x) + \mathsf{O}(1) \\ &= \mathcal{C}(x) + \mathsf{O}(\log \log n) \end{aligned} \tag{3.24}$$

Als weiteren Spezialfall betrachten wir die Folgen $x = 1^\ell$ und $x = 0^\ell$ für ein beliebiges, aber festes $\ell \in \mathbb{N}$. Abbildung 1 zeigt ein Programm, welches $1^\ell$ erzeugt. Es folgt, dass für jedes $\ell$ eine Konstante $c_\ell$ existiert mit

$$\mathcal{C}(1^\ell) \leq c_\ell \tag{3.25}$$

Entsprechendes gilt für $x = 0^\ell$.

```
read();
write(1^ℓ).
```

**Abb. 1: Programm zur Erzeugung des Wortes $1^\ell$.**

Aus (3.24) und (3.25) folgt, dass für $x \in \{0, 1\}$ und $n = 2^k$

$$\mathcal{C}(x^n) \leq \log\log n + \mathsf{O}(1) \tag{3.26}$$

gilt.

Im folgenden Satz greifen wir eine Bemerkung aus Abschnitt 1.4 – jetzt bezogen auf natürliche Zahlen – auf, nämlich dass die Art der Codierung von Zahlen keinen wesentlichen Einfluss auf deren Kolmogorov-Komplexität hat.

**Satz 3.4** *Es existiert eine Konstante $c$, so dass für alle $n \in \mathbb{N}_0$*

$$|\mathcal{C}(n) - \mathcal{C}(1^n)| \leq c \tag{3.27}$$

*gilt.*

**Beweis** *Wir halten zunächst fest, dass wir in (2.4) $\mathcal{C}(n) = \mathcal{C}(dual(n))$ festgelegt haben.*

*Es sei $b^*(1^n) = \langle A \rangle$, also $\Phi_U \langle A \rangle = 1^n$. Sei $B$ ein Programm, welches bei Eingabe $w \in \mathbb{B}^*$ überprüft, ob $w$ die Codierung eines Prgramms ist, und gegebenenfalls das Programm ausführt. Besteht die Ausgabe aus einer Folge von Einsen, dann zählt $B$ diese und gibt die Anzal als Dualzahl aus. In allen anderen Fällen liefert $B$ keine Ausgabe. Es gilt dann offensichtlich $\Phi_B \langle A \rangle =$*

*$dual(n)$ und damit $\Phi_U \langle B, A \rangle = dual(n)$. $b(dual(n)) = \langle B, A \rangle$ ist also eine Beschreibung von $dual(n)$. Es folgt*

$$\begin{aligned}\mathcal{C}(n) = \mathcal{C}(dual(n)) &= |b^*(dual(n))| \\ &\leq |b(dual(n))| \\ &= |\langle B, A \rangle| \\ &= 2(|\langle B \rangle| + 1) + |\langle A \rangle| \\ &= |b^*(1^n)| + c' \\ &= \mathcal{C}(1^n) + c'\end{aligned}$$

*für $c' = 2(|\langle B \rangle| + 1)$. Es existiert also eine Konstante $c'$, so dass*

$$\mathcal{C}(n) - \mathcal{C}(1^n) \leq c' \tag{3.28}$$

*gilt.*

*Sei nun $b^*(dual(n)) = \langle A \rangle$, also $\Phi_U \langle A \rangle = dual(n)$. Sei $B$ ein Programm, welches bei Eingabe $w \in \mathbb{B}^*$ überprüft, ob $w$ die Codierung eines Prgramms ist, und gegebenenfalls die Ausgabe $1^{wert(\Phi_U(w))}$ erzeugt, ansonsten liefert $B$ keine Ausgabe. Es gilt dann*

$$\Phi_B \langle A \rangle = 1^{wert(\Phi_U \langle A \rangle)} = 1^{wert(dual(n))} = 1^n$$

*und damit $\Phi_U \langle B, A \rangle = 1^n$. $b(1^n) = \langle B, A \rangle$ ist also eine Beschreibung von $1^n$. Es folgt*

$$\begin{aligned}\mathcal{C}(1^n) = &= |b^*(1^n)| \\ &\leq |b(1^n)| \\ &= |\langle B, A \rangle| \\ &= 2(|\langle B \rangle| + 1) + |\langle A \rangle| \\ &= |b^*(dual(n))| + c'' \\ &= \mathcal{C}(dual(n)) + c'' \\ &= \mathcal{C}(n) + c''\end{aligned}$$

*für $c'' = 2(|\langle B \rangle| + 1)$. Es existiert also eine Konstante $c''$, so dass*

$$\mathcal{C}(1^n) - \mathcal{C}(n) \leq c'' \tag{3.29}$$

*gilt.*

*Mit $c = \max\{c', c''\}$ folgt aus (3.28) und (3.29) die Behauptung (3.27).* □

Zum Schluss dieses Kapitels betrachten wir noch die Differenz der Kolmogorov-Komplexität $\mathcal{C}(x)$ und $\mathcal{C}(x')$ von Bitfolgen $x, x' \in \mathbb{B}^*$.

**Satz 3.5** *Es existiert eine Konstante $c$, so dass für alle $x, h \in \mathbb{B}^*$ gilt:*

$$|\mathcal{C}(x+h) - \mathcal{C}(x)| \leq 2\,|h| + c \tag{3.30}$$

**Beweis** *Sei $b^*(x) = \langle\, A \,\rangle$ die kürzeste Beschreibung von $x$; es ist also $\Phi_U \langle\, A \,\rangle = x$. Sei $F$ ein Programm, welches zunächst testet, ob eine eingegebene Bitfolge $u$ die Struktur $u = v01w$ mit $v \in \{00, 11\}^*$ und $w \in \mathbb{B}^*$ hat und $w$ die Codierung eines Programms von $\mathcal{M}$ ist. Falls das nicht der Fall ist, liefert $F$ keine Ausgabe, ansonsten berechnet $F$ die Summe von $d^{-1}(v)$ und der Ausgabe der Ausführung von $w$ (falls die Ausführung $w$ nicht terminiert, liefert $F$ natürlich ebenfalls keine Ausgabe.). Dabei ist $d$ die in (1.17) definierte Bit-Verdopplungsfunktion. Für $F$ gilt*

$$\Phi_F \langle\, h, A \,\rangle = h + \Phi_U \langle\, A \,\rangle = h + x$$

*und damit*

$$\Phi_U \langle\, F, h, A \,\rangle = x + h$$

*Es folgt, dass $b(x+h) = \langle\, F, h, A \,\rangle$ eine Beschreibung von $x + h$ ist. Damit gilt*

$$\begin{aligned} \mathcal{C}(x+h) = |b^*(x+h)| \leq |b(x+h)| &= |\langle\, F, h, A \,\rangle| \\ &= 2(|\langle\, F \,\rangle| + |h| + 2) + |\langle\, A \,\rangle| \\ &= 2\,|\langle\, F \,\rangle| + 4 + 2\,|h| + |b^*(x)| \\ &= \mathcal{C}(x) + 2\,|h| + c' \end{aligned}$$

*mit $c' = 2(|\langle\, F \,\rangle| + 2)$. E existiert also eine Konstante $c'$ mit*

$$\mathcal{C}(x+h) - \mathcal{C}(x) \leq 2\,|h| + c' \tag{3.31}$$

*Sei nun $b^*(x+h) = \langle\, A \,\rangle$ die kürzeste Beschreibung von $x + h$; es ist also $\Phi_U \langle\, A \,\rangle = x + h$. Sei $G$ ein Programm, welches zunächst testet, ob eine eingegebene Bitfolge $u$ die Struktur $u = v01w$ mit $v \in \{00, 11\}^*$ und $w \in \mathbb{B}^*$ hat und $w$ die Codierung eines Programms von $\mathcal{M}$ ist. Falls das nicht der Fall ist, liefert $G$ keine Ausgabe, ansonsten führt $G$ das Programm aus und bestimmt, falls die Ausführung von $G$ eine Ausgabe liefert, die Differenz $\Phi_U(w) - d^{-1}(v)$ (falls der Subtrahend größer als der Minuend sein sollte, ist die Ausgabe $0$). Für $G$ gilt*

$$\Phi_G \langle\, h, A \,\rangle = \Phi_U \langle\, A \,\rangle - h = x + h - h$$

*und damit*

$$\Phi_U \langle G, h, A \rangle = x$$

*Es folgt, dass* $b(x) = \langle G, h, A \rangle$ *eine Beschreibung von* $x$ *ist. Damit gilt*

$$\begin{aligned} \mathcal{C}(x) = |b^*(x)| \leq |b(x)| &= |\langle G, h, A \rangle| \\ &= 2(|\langle G \rangle| + |h| + 2) + |\langle A \rangle| \\ &= 2\,|\langle G \rangle| + 4 + 2\,|h| + |b^*(x+h)| \\ &= \mathcal{C}(x+h) + 2\,|h| + c'' \end{aligned}$$

*mit* $c'' = 2(|\langle G \rangle| + 2)$. *Es existiert also eine Konstante* $c''$ *mit*

$$\mathcal{C}(x) - \mathcal{C}(x+h) \leq 2\,|h| + c'' \tag{3.32}$$

*Mit* $c = \max\{c', c''\}$ *folgt dann aus (3.31) und (3.32)*

$$|\mathcal{C}(x+h) - \mathcal{C}(x)| \leq 2\,|h| + c$$

*womit die Behauptung gezeigt ist.* □

# 4 (Nicht-) Komprimierbarkeit und Zufälligkeit

Satz 2.1 zeigt, dass eine minimale Beschreibung einer Zeichenkette niemals viel länger als die Zeichenkette selbst ist. Sicherlich gibt es Zeichenketten, deren minimale Beschreibung sehr viel kürzer als die Zeichenkette selbst ist, etwa wenn sie Redundanzen enthält, wie z.B. die Zeichenkette $xx$ (siehe Satz 3.1). Es stellt sich die Frage, ob es Zeichenketten gibt, deren minimale Beschreibungen länger als sie selber sind. Wir zeigen in diesem Kapitel, dass solche Zeichenketten existieren. Eine kurze Beschreibung einer solchen Zeichenkette besteht dann aus einem Programm, welches im Wesentlichen nichts anderes tut, als diese auszudrucken. Zunächst führen wir den Begriff der *Komprimierbarkeit* von Zeichenketten ein und werden diesen verwenden, um einen *Zufälligkeitsbegriff* einzuführen.

**Definition 4.1** *Sei* $c \in \mathbb{N}$ *und* $x \in \mathbb{B}^*$.

**a)** $x$ *heißt* **c-komprimierbar** *genau dann, wenn* $\mathcal{C}(x) \leq |x| - c$ *gilt. Falls es ein* $c$ *gibt, so dass* $x$ $c$*-komprimierbar ist, dann nennen wir* $x$ *auch* **regelmäßig**.

**b)** *Falls $x$ nicht 1-komprimierbar ist, d.h. wenn $\mathcal{C}(x) \geq |x|$ ist, dann nennen wir $x$* **nicht komprimierbar** *oder* **zufällig**. □

Wörter sind also regelmäßig, falls sie Beschreibungen besitzen, die deutlich kürzer als sie selbst sind. Diese Eigenschaft trifft insbesondere auf periodisch aufgebaute Wörter zu. Wörter sind dementsprechend zufällig, wenn ihr Aufbau keine Regelmäßigkeiten zeigt. Diese können quasi nur durch sich selbst beschrieben werden.

Der folgende Satz besagt, dass es Wörter gibt, die nicht komprimierbar im Sinne der Kolmogorov-Komplexität sind, und zwar gibt es sogar für jede Zahl $n$ mindestens ein Wort mit der Länge $n$, dass nicht komprimierbar ist.

**Satz 4.1** *Zu jeder Zahl $n \in \mathbb{N}$ gibt es mindestens ein Wort $x \in \mathbb{B}^*$, so dass*

$$\mathcal{C}(x) \geq |x| = n$$

*ist.*

**Beweis** *Es gilt $|\mathbb{B}^n| = 2^n$. Es seien $x_i$, $1 \leq i \leq 2^n$, die Elemente von $\mathbb{B}^n$, und $b^*(x_i)$ die kürzeste Beschreibung von $x_i$. Somit gilt*

$$\mathcal{C}(x_i) = |b^*(x_i)| \tag{4.1}$$

*Für $i \neq j$ ist $b^*(x_i) \neq b^*(x_j)$. Es gibt also $2^n$ kürzeste Beschreibungen $b^*(x_i)$, $1 \leq i \leq 2^n$. Es gilt $b^*(x_i) \in \mathbb{B}^*$ sowie $|\mathbb{B}^i| = 2^i$. Die Anzahl der nicht leeren Wörter mit einer Länge von höchstens $n-1$ ist*

$$\sum_{i=1}^{n-1} \left|\mathbb{B}^i\right| = \sum_{i=1}^{n-1} 2^i = 2^n - 2 < 2^n$$

*Unter den $2^n$ Wörtern $b^*(x_i)$, $1 \leq i \leq 2^n$, muss es also mindestens eines mit einer Länge von mindestens $n$ geben. Sei $k$ mit $|b^*(x_k)| \geq n$. Es folgt mit (4.1)*

$$\mathcal{C}(x_k) = |b^*(x_k)| \geq n$$

*und damit die Behauptung. $x = x_k$ ist also nicht komprimierbar.* □

Aus dem Beweis des Satzes folgt: Für $c \in \mathbb{N}$ kann es nur höchstens $2^{n-c+1}-1$ Elemente $x \in \mathbb{B}^*$ geben mit $\mathcal{C}(x) \leq n - c$. Der Anteil komprimierbarer Bitfolgen beträgt also

$$\frac{2^{n-c+1}}{2^n} = 2^{-c+1}$$

Es gibt also z.B. weniger als $2^{n-7}$ Bitfolgen der Länge $n$ mit einer Kolmogorov-Komplexität kleiner oder gleich $n-8$. Der Anteil 8-komprimierbarer Bitfolgen beträgt

$$2^{-7} < 0,8\%$$

Das heißt, dass mehr als 99% Prozent aller Bitfolgen der Länge $n$ eine Kolmogorov-Komplexität größer als $n-8$ haben. Es ist also sehr unwahrscheinlich, dass eine zufällig erzeugte Bitfolge $c$-komprimierbar ist.

Aus dem Satz folgt unmittelbar

**Folgerung 4.1** *Es gibt unendlich viele Wörter $x \in \mathbb{B}^*$ mit $\mathcal{C}(x) \geq |x|$, d.h. es gibt unendlich viele Wörter, die nicht komprimierbar sind.* □

Des Weiteren können wir die Mindestanzahl von Zeichenketten der Länge $n$ angeben, die nicht $c$-komprimierbar sind.

**Folgerung 4.2** *Seien $c, n \in \mathbb{N}$. Es gibt mindesten $2^n(1-2^{-c+1})+1$ Elemente $x \in \mathbb{B}^n$, die nicht $c$-komprimierbar sind.*

**Beweis** *Wie im Beweis von Satz 4.1 können wir überlegen, dass höchstens $2^{n-c+1}$ Zeichenketten der Länge $n$ $c$-komprimierbar sind, weil höchstens so viele minimale Beschreibungen der Länge $n-c$ existieren. Also sind die restlichen $2^n - (2^{n-c+1}-1) = 2^n(1-2^{-c+1})+1$ Zeichenketten nicht $c$-komprimierbar.* □

**Bemerkung 4.1** *Die Folgerung 4.2 kann wie folgt umformuliert werden: Seien $c, n \in \mathbb{N}$. Dann gibt es mindesten $2^n(1-2^{-c+1})+1$ Elemente $x \in \mathbb{B}^n$, für die*

$$\mathcal{C}(x) > n - c = \log 2^n - c = \log |\mathbb{B}^n| - c$$

*gilt.* □

Eine Vermutung, die man haben könnte, ist, dass die Kolmogorov-Komplexität Präfix-momoton ist, d.h., dass $\mathcal{C}(x) \leq \mathcal{C}(xy)$ für alle $x, y \in \mathbb{B}^*$ gilt. Das trifft allerdings im Allgemeinen nicht zu; die Komplexität eines Präfixes kann größer sein als die der gesamten Bitfolge. Dazu betrachten wir folgendes Beispiel: Sei $xy = 1^n$ mit $n = 2^k$, also $k = \log n$, dann gibt es wegen (3.26) ein $c$ mit

$$\mathcal{C}(xy) = \mathcal{C}(1^n) \leq \log\log n + c \tag{4.2}$$

Laut obiger Bemerkung existieren zu $c$ und $k$ mindestens $2^k(1-2^{-c+1})+1$ Elemente $z \in \mathbb{B}^k$ mit

$$\mathcal{C}(z) > k - c = \log n - c \tag{4.3}$$

Für $z \in \mathbb{B}^k$ gilt $wert(z) < 2^k = n$. Damit ist $x = 1^{wert(z)}$ ein Präfix von $xy = 1^n$. Hieraus folgt mit Satz 3.4 (siehe Teil 1 des Beweises) und (4.3), dass eine Konstante $c'$ existiert mit

$$\begin{aligned} \mathcal{C}(x) &= \mathcal{C}(1^{wert(z)}) \\ &\quad \mathcal{C}(dual(wert(z))) - c' \\ &\quad = \mathcal{C}(z) - c' \\ &\quad > \log n - c - c' \end{aligned} \tag{4.4}$$

Aus (4.2) und (4.4) folgt dann für hinreichend große $n$: $\mathcal{C}(x) > \mathcal{C}(xy)$. Komprimierbare Zeichenketten können also nicht komprimierbare Präfixe enthalten.

Andererseits gilt

**Satz 4.2** *Sei $d \in \mathbb{N}$ gegeben. Ein hinreichend langes Wort $x \in \mathbb{B}^*$ besitzt immer ein Präfix $w$, das $d$-komprimierbar ist, d.h. für welches $\mathcal{C}(w) \leq |w| - d$ gilt.*

**Beweis** *Sei $v$ ein Präfix von $x$ mit $\tau(v) = i$, d.h. mit $\nu(i) = v$. Es ist also $x = v\beta$, und $v$ ist das $i$-te Wort in der lexikografischen Aufzählung der Wörter von $\mathbb{B}^*$. Sei nun $w'$ das Infix von $x$ der Länge $i$, also mit $|w'| = i$, welches auf $v$ folgt. Es ist also $x = vw'\beta'$ mit $|w'| = i$. Wir setzen $w = vw'$, womit $x = w\beta'$ ist. Das Wort $w$ kann aus dem Wort $w'$ erzeugt werden, denn es ist $w = \nu(|w'|)w'$. Die Funktionen $\tau$, $|\cdot|$ sowie die Konkatenation von Wörtern sind total berechenbare Funktionen, mit denen aus einem Wort $w'$ das Wort $w = vw'$ berechent werden kann. Wegen Folgerung 3.2 gibt es eine Konstante $c$, so dass $\mathcal{C}(w) \leq |w'| + c$ ist, wobei $c$ weder von $x$ noch von $v$ abhängt. Des Weiteren ist $|w| = |v| + |w'|$. Wenn wir also das Präfix $v$ von $x$ so wählen, dass $|v| \geq c + d$ ist, dann gilt*

$$\mathcal{C}(w) \leq |w'| + c = |w| - |v| + c \leq |w| - (c+d) + c = |w| - d$$

*womit die Behauptung gezeigt ist.* □

Mithilfe der obigen Überlegungen können wir nun auch zeigen, dass die Kolmogorov-Komplexität nicht subadditiv ist (siehe Bemerkung 3.3).

**Satz 4.3** *Es existiert eine Konstante $b$, so dass für alle $x, x' \in \mathbb{B}^*$ mit $|x|, |x'| \leq n$*

$$\mathcal{C}(xx') > \mathcal{C}(x) + \mathcal{C}(x') + \log n - b$$

*gilt.*

**Beweis** *Es sei $B = \{(x, x') \in \mathbb{B}^* \times \mathbb{B}^* \mid |x| + |x'| = n\}$. Für $B$ gilt $|B| = 2^n(n+1)$. Wegen der Folgerung 4.2 gibt es mindestens ein Paar $xx' \in B$, welches nicht $1$-komprimierbar ist. Für dieses Paar gilt wegen Bemerkung 4.1*

$$\mathcal{C}(xx') \geq \log |B| - 1 = \log ( 2^n(n+1) ) - 1 \geq n + \log n - 1 \qquad (4.5)$$

*Wegen Satz 3.1 gibt es eine von $x$ und $x'$ unabhänige Konstante $b$, so dass*

$$\mathcal{C}(x) + \mathcal{C}(x') \leq |x| + |x'| + b \qquad (4.6)$$

*gilt. Mit (4.5) und (4.6) gilt*

$$\begin{aligned} \mathcal{C}(xx') &\geq n + \log n - 1 \\ &= |x| + |x'| + \log n - 1 \\ &\geq \mathcal{C}(x) + \mathcal{C}(x') - b + \log n - 1 \\ &> \mathcal{C}(x) + \mathcal{C}(x') + \log n - b \end{aligned}$$

*womit die Behauptung gezeigt ist.* □

Die obigen Überlegungen können wir ebenfalls verwenden, um die wesentliche Frage zu beantworten, ob es ein Verfahren gibt, mit dem die Kolmogorov-Komplexität $\mathcal{C}(x)$ für jedes $x \in \mathbb{B}^*$ bestimmt werden kann. Der folgende Satz beantwortet diese Frage negativ.

**Satz 4.4** *Die Kolmogorov-Komplexität $\mathcal{C}$ ist nicht berechenbar.*

**Beweis** *Wir nehmen an, dass es ein Programm $C$ gibt, dass die Funktion $\mathcal{C}$ für alle $x \in \mathbb{B}^*$ berechnet. Sei $x_n$ das erste Wort in der lexikografischen Ordnung von $\mathbb{B}^*$ mit $\mathcal{C}(x_n) \geq n$ (ein solches Wort existiert gemäß Satz 4.1 und Folgerung 4.1). Die in Abbildung 2 dargestellte Familie $W_n$, $n \in \mathbb{N}$, von While-Programmen berechnet die Folge dieser $x_n$ mit $\mathcal{C}(x_n) \geq n$. Da alle Programme $W_n$ bis auf die („fest verdrahtete") Zahl $n$ identisch sind, sind die Binärcodierungen aller $W_n$ ohne $n$ identisch und damit unabhängig von $n$ konstant lang. Diese Länge sei $c$. Es folgt*

$$\mathcal{C}(x_n) \leq \lfloor \log n \rfloor + 1 + c$$

```
read();
    x := ε;
    while Φ_C(x) < n do
        x := suc(x)
    endwhile;
write(x).
```

**Abb. 2: While-Programme $W_n$ zur Erzeugung des kürzesten Wortes $x_n$ mit $\mathcal{C}(x_n) \geq n$.**

*Für $x_n$ gilt aber $\mathcal{C}(x_n) \geq n$. Es folgt, dass*

$$n \leq \mathcal{C}(x_n) \leq \lfloor \log n \rfloor + 1 + c$$

*sein muss. Diese Ungleichung kann aber nur für endlich viele $n \in \mathbb{N}$ zutreffen. Unsere Annahme, dass die Funktion $\mathcal{C}$ berechenbar ist, ist also falsch.* □

Leider ist die Kolmogorov-Komplexität einer Zeichenkette nicht berechenbar. Allerdings gibt es eine berechenbare Funktion, mit der wir $\mathcal{C}(x)$ approximieren können.

**Satz 4.5** *Es gibt eine total berechenbare Funktion $\Gamma : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ mit $\lim_{t\to\infty} \Gamma(t, x) = \mathcal{C}(x)$.*

**Beweis** *$\Gamma$ sei durch das in Abbildung 3 dargestellte Programm definiert. Wegen Satz 3.1 existiert eine Konstante $c$ mit $\mathcal{C}(x) \leq |x| + c$ für alle $x \in \mathbb{B}^*$. Der Beweis dieses Satzes zeigt, dass sich ein solches $c$ bestimmen lässt. Deswegen kann im in Abbildung 3 dargestellten Programm $|x| + c$ als Schleifengrenze verwendet werden. Dieses Programm benutzt außerdem das Programm $C$, welches die total berechenbare Funktion $\gamma : \mathbb{N}_0 \times \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$ definiert durch*

$$\gamma(U, A, t) = \begin{cases} 1, & \text{falls } U \text{ angewendet auf } A \text{ innerhalb von } t \text{ Schritten hält} \\ 0, & \text{sonst} \end{cases}$$

*berechnet. Das Programm probiert in lexikografischer Ordnung alle Programme mit einer Länge kleiner gleich $|x|+c$ durch, ob diese in höchstens $t$ Schritten $x$ erzeugen. Falls es solche Programme gibt, wird die Länge des kürzesten von diesen ausgegeben. Falls es ein solches Programm nicht gibt, wird die*

*obere Schranke $|x| + c$ von $\mathcal{C}(x)$ ausgegeben. Es ist offensichtlich, dass das Programm für alle Eingaben $(t, x)$ anhält, $\Gamma$ ist also eine total berechenbare Funktion. Des Weiteren ist offensichtlich, dass $\Gamma$ monoton fallend in $t$ ist, d.h. es ist $\Gamma(t, x) \geq \Gamma(t', x)$ für $t' > t$, und ebenso offensichtlich ist, dass $\mathcal{C}(x) \leq \Gamma(t, x)$ für alle $t$ gilt. Der Grenzwert $\lim_{t\to\infty} \Gamma(t, x)$ existiert für jedes $x$, denn für jedes $x$ gibt es ein $t$, so dass $U$ in $t$ Schritten mit Ausgabe $x$ anhält, nämlich wenn $U$ die Eingabe eines Programms $A$ mit $\langle A \rangle = b^*(x)$, d.h. mit $|\langle A \rangle| = \mathcal{C}(x)$, erhält.*

*Da für jedes $x$ der Grenzwert $\lim_{t\to\infty} \Gamma(t, x)$ existiert, $\Gamma(t, x)$ eine monoton fallende Folge ist, die nach unten durch $\mathcal{C}(x)$ beschränkt ist, folgt die Behauptung.* □

---

```
read(t, x);
    A := ε;
    M := ∅;
    while |⟨A⟩| < |x| + c do
        if Φ_C ⟨arg1U, A, t⟩ = 1 and Φ_U ⟨A⟩ = x then M :=
M ∪ {A}
        else A := suc(A) endif
    endwhile;
    if M ≠ ∅ then return min{|⟨A⟩| : A ∈ M}
    else return(|x| + c)endif.
```

---

**Abb. 3: Programm zur Berechnung der Approximation $\Gamma(t, x)$ von $\mathcal{C}(x)$.**

---

**Bemerkung 4.2** *$\mathcal{C}$ ist keine berechenbare Funktion, deswegen kann algorithmisch nicht entschieden werden, ob $\Gamma(t, x) = \mathcal{C}(x)$ ist. $\mathcal{C}(x)$ kann aber durch die total berechenbare Funktion $\Gamma$ (von oben) approximiert werden; $\mathcal{C}$ ist quasi von „oben berechenbar“.* □

**Satz 4.6** *Es gibt berechenbare Funktionen $g, h : \mathbb{B}^* \to \mathbb{B}^*$ mit $g(b^*(x)) = \langle x, \mathcal{C}(x) \rangle$ bzw. mit $h \langle x, \mathcal{C}(x) \rangle = b^*(x)$.*

**Beweis** *$g$ kann durch folgendes Programm $A$ berechnet werden: $A$ testet, ob eine Eingabe $w$ Codierung eines Programms ist. Falls ja, bestimmt $A$ die*

*Länge von $w$ und führt $w$ aus. Die Ausgabe von $A$ ist dann die Ausgabe des Programms $w$ sowie $|w|$. Es gilt also:*

$$\Phi_A(w) = \begin{cases} \langle\, \Phi_U(w), |w| \,\rangle, & w \in \mathcal{M} \\ \perp, & \textit{sonst} \end{cases}$$

*Daraus folgt*

$$\Phi_A(b^*(x)) = \langle\, \Phi_U(b^*(x)), |b^*(x)| \,\rangle = \langle\, x, \mathcal{C}(x) \,\rangle$$

*und damit $\Phi_A(b^*(x)) = g(b^*(x))$, d.h. $g$ wird von $A$ berechnet.*

*$h$ kann durch das folgende Programm $A$ berechnet werden: $A$ erhält als Eingabe $x$ sowie $\mathcal{C}(x)$, die Länge des kürzesten Programms, welches $x$ berechnet. $A$ erzeugt der Reihe nach alle Programme mit der Länge $\mathcal{C}(x)$ und führt diese jeweils aus. Wenn ein erzeugtes Programm $B$ die Ausgabe $x$ hat, dann gibt $A$ dieses aus, es ist dann nämlich $b^*(x) = \langle\, B \,\rangle$ die kürzeste Beschreibung von $x$. Da $\mathcal{C}(x)$ als Eingabe existiert, muss $A$ ein Programm dieser Länge $x$ erzeugen.* □

**Bemerkung 4.3** *Der Satz 4.6 besagt, dass $b^*(x)$ und $\langle\, x, \mathcal{C}(x) \,\rangle$ im Wesentlichen denselben Informationsgehalt haben. Denn mit Satz 3.2 folgt aus dem Teil 1 des obigen Satzes*

$$\mathcal{C}\,\langle\, x, \mathcal{C}(x) \,\rangle = \mathcal{C}(g(b^*(x))) \leq \mathcal{C}(b^*(x)) + c$$

Wir wollen zum Schluss dieses Kapitels noch die Frage beantworten, ob die kürzeste Beschreibung eines Wortes komprimierbar ist. Die Vermutung, dass das nicht der Fall ist, bestätigt der folgende Satz.

**Satz 4.7** *Es existiert eine Konstante $c \in \mathbb{N}$, so dass $b^*(x)$ für alle $x \in \mathbb{B}^*$ nicht $c$-komprimierbar ist.*

**Beweis** *Sei $b^*(x) = \langle\, B \,\rangle$ die kürzeste Beschreibung von $x$; es gilt also $\Phi_U\,\langle\, B \,\rangle = x$. Wir überlegen uns folgendes Programm $A$ mit Eingabe $y$: $A$ überprüft, ob $y$ Codierung eines Programms ist. Gegebenenfalls führt $A$ dieses Programm aus. Dann überprüft $A$, ob die resultierende Ausgabe ebenfalls die Codierung eines Programms ist. Ist das der Fall, dann führt $A$ auf dieses Programm das universelles Programm aus. Das Ergebnis dieser Anwendung*

*ist dann die Ausgabe von $A$. In allen anderen Fällen ist $A$ nicht definiert. Mit dieser Festlegung von $A$ gilt*

$$\Phi_A(b^*(b^*(x))) = x \tag{4.7}$$

*denn $b^*(b^*(x))$ ist Codierung eines Programms, und $b^*(b^*(x)) = b^* \langle B \rangle$ ist das kürzeste Programm, welches $\langle B \rangle$ erzeugt. Das heißt: $b^* \langle B \rangle$ ist die Codierung eines Programms, dessen Ausführung die Codierung $\langle B \rangle$ erzeugt. Die Anwendung des universellen Programmes hierauf liefert die Ausgabe $x$. Es gilt also*

$$\Phi_U \langle A, b^*(b^*(x)) \rangle = x \tag{4.8}$$

*und damit ist*

$$b(x) = \langle A, b^*(b^*(x)) \rangle \tag{4.9}$$

*eine Beschreibung von $x$. Wir setzen*

$$c = 2\,|\langle A \rangle| + 3 \tag{4.10}$$

*und zeigen im Folgnden, dass dieses $c$ die Behauptung des Satzes erfüllt. Dazu nehmen wir an, dass $b^*(x)$ $c$-komprimierbar für ein $x \in \mathbb{B}^*$ ist, d.h. dass für dieses $x$*

$$\mathcal{C}(b^*(x)) \leq |b^*(x)| - c$$

*und damit*

$$|b^*(b^*(x))| \leq |b^*(x)| - c \tag{4.11}$$

*ist. Mit (4.9), (4.10) und (4.11) folgt*

$$\begin{aligned} |b^*(x)| &\leq |b(x)| \\ &= 2(|\langle A \rangle| + 1) + |b^*(b^*(x))| \\ &= c - 1 + |b^*(b^*(x))| \\ &\leq c - 1 + |b^*(x)| - c \\ &= |b^*(x)| - 1 \end{aligned}$$

*was offensichtlich einen Widerspruch darstellt. Damit ist unsere Annahme widerlegt, und das gewählte $c$ erfüllt die Behauptung.* □

# 5 Anwendungen

In diesem Kapitel werden einige Problemstellungen der Theoretischen Informatik mithilfe der Kolmogorov-Komplexität beleuchtet. Wir werden beispielhaft sehen, dass bekannte Aussagen der Theoretischen Informatik, die üblicherweise mit Methoden und Techniken wie etwa die Diagonalisierung oder das Pumping-Lemma für reguläre Sprachen gezeigt werden, auch mithilfe der Kolmogorov-Komplexität bewiesen werden können. Diese Beweise basieren auf folgender Idee: Zufällige Bitfolgen $x$, d.h. solche, bei denen $\mathcal{C}(x) \geq |x|$ ist, können nicht komprimiert werden. Sei nun das Prädikat $P(y)$ zu beweisen. Für einen Widerspruchsbeweis nimmt man an, dass $\neg P(x)$ gilt für ein nicht komprimierbares $x$. Folgt nun aus der Annahme, dass $x$ komprimiert werden kann, ist ein Widerspruch hergeleitet, und die Annahme muss falsch sein.

Des Weiteren werden wir sehen, wie alle Entscheidbarkeitsfragen mithilfe einer (unendlichen) Bitfolge, der so genannten Haltesequenz codiert werden können.

## 5.1 Unentscheidbarkeit des Halteproblems

Als eine erste Anwendung zeigen wir, wie mithilfe der Unberechenbarkeit der Kolmogorov-Komplexität die Unentscheidbarkeit des Halteproblems gezeigt werden kann. Das **Halteproblem** sei gegeben durch die Sprache

$$\mathcal{H} = \{\langle A, w \rangle \in \mathbb{B}^* \mid w \in \mathit{Def}(\Phi_A)\}$$

**Satz 5.1** *$\chi_{\mathcal{H}}$ ist nicht berechenbar.*

**Beweis** *Wir nehmen an, dass $\chi_{\mathcal{H}}$ berechenbar ist. Es sei $H \in \mathcal{M}$ ein Programm, welches $\chi_{\mathcal{H}}$ berechnet: $\Phi_H = \chi_{\mathcal{H}}$. Mithilfe von $H$ konstruieren wir das in Abbildung 4 dargestellte Programm $C$. Dabei ist $\nu(i) = w$ das $i$-te Wort in der lexikografischen Anordnung der Wörter von $\mathbb{B}^*$ ist (siehe Bemerkung 1.1 f). Das Programm durchläuft die Wörter von $\mathbb{B}^*$ der Größe nach. Es überprüft, ob das aktuelle Wort ein Programm $A \in \mathcal{M}$ darstellt. Falls dieses zutrifft, wird überprüft, ob $A$ angewendet auf das leere Wort anhält. Im gegebenen Fall wird überprüft, ob $A$ dabei die Eingabe $x$ erzeugt. Falls ja, dann ist die Länge dieses Programmes die Kolmogorov-Komplexität von $x$: $|b^*(x)| = |\langle A \rangle|$. Das Programm terminiert und gibt die Kolmogorov-Komplexität*

$$k = |\langle A \rangle| = \mathcal{C}(x)$$

*von $x$ aus. In allen anderen Fällen wird das nächste Wort in der lexikografischen Ordnung entsprechend überprüft.*

```
read(x);
   i := 0;
   gefunden := false;
   while not gefunden do
      A := ν(i);
      if A ∈ M then
         if Φ_H ⟨A, ε⟩ = 1 then
            if Φ_U ⟨A⟩ = x then
               k := |⟨A⟩|;
               gefunden := true
            else i := i + 1
            endif;
         endif;
      endif;
   endwhile;
write(k).
```

**Abb. 4: Programm $C$ zur Berechnung der Kolmogorov-Komplexität unter der Voraussetzung, dass das Halteproblem entscheidbar ist.**

*Da alle Wörter von $\mathbb{B}^*$ durchlaufen werden, werden auch alle Programme von $\mathcal{M}$ durchlaufen. Darunter ist auf jeden Fall eines, welches (ohne Eingabe) das eingegebene Wort $x$ erzeugt (mindestens das in Abbildung 5 dargestellte Programm). Damit ist für jede Eingabe die Terminierung des Programms $C$ gesichert. Da die Programme der Größe nach durchlaufen werden, ist auch gesichert, dass $C$ das kürzestes Programm, welches $x$ erzeugt, findet.*

```
read();
write(x).
```

**Abb. 5: Naives Programm zur Erzeugung des Wortes $x$.**

*Damit berechnet das Programm $C$ die Kolmogorov-Komplexität für jedes $x \in \mathbb{B}$. Daraus folgt, dass $\mathcal{C}$ eine berechenbare Funktion ist, ein Widerspruch zu Satz 4.4. Unsere eingangs des Beweises gemachte und im Programm $C$ verwendete Annahme, dass $\chi_{\mathcal{H}}$ berechenbar ist, muss also falsch sein. Damit ist die Behauptung des Satzes gezeigt.* □

**Bemerkung 5.1** *Der Beweis von Satz 5.1 gibt quasi eine Reduktion der Berechung von $\mathcal{C}$ auf die Berechnung von $\chi_{\mathcal{H}}$ an.* □

## 5.2 Die Menge der Primzahlen ist unendlich

Ein bekannter Beweis für die Unendlichkeit der Menge $\mathbb{P}$ der Primzahlen geht auf Euklid[14] zurück und argumentiert etwa wie folgt: Es wird angenommen, dass es nur endliche viele Primzahlen $p_1, \ldots, p_k$, $k \geq 1$, gibt. Damit bildet man die Zahl $n = p_1 \cdot \ldots \cdot p_k + 1$. Es folgt, dass keine der Primzahlen $p_i$, $1 \leq i \leq k$, ein Teiler von $n$ sein kann. Dies ist ein Widerspruch zu der Tatsache, dass jede Zahl $n \in \mathbb{N}_0$, $n \geq 2$, mindestens einen Primteiler besitzt (so ist z.B. der kleinste Teiler immer eine Primzahl). Die Annahme, dass $\mathbb{P}$ endlich ist, führt also zu einem Widerspruch.

Ein Beweis der Unendlichkeit von $\mathbb{P}$ kann z.B. wie folgt mithilfe der Kolmogorov-Komplexität geführt werden. Für die Kolmogorov-Komplexität natürlicher Zahlen, siehe (2.4), gilt gemäß Folgerung 4.1, dass es unendlich viele natürliche Zahlen $n \in \mathbb{N}_0$ mit

$$\mathcal{C}(n) = \mathcal{C}(dual(n)) \geq |dual(n)| = \lfloor \log n \rfloor + 1 \tag{5.1}$$

gibt. Wir nehmen ein solches $n$ und nehmen an, dass es nur endlich viele Primzahlen $p_1, \ldots, p_k$, $k \geq 1$, gibt. Sei

$$n = p_1^{\alpha_1} \cdot p_2^{\alpha_k} \cdot \ldots \cdot p_k^{\alpha_k} \tag{5.2}$$

die eindeutige Faktorisierung von $n$; $n$ ist also eindeutig durch die Exponenten $\alpha_i$, $1 \leq i \leq k$, erzeugbar, d.h. $b(n) = \langle\, \alpha_1, \ldots, \alpha_k \,\rangle$ ist eine Beschreibung von $n$. Aus (5.2) folgt

$$n \geq 2^{\alpha_1 + \ldots + \alpha_k}$$

[14] Der griechische Mathematiker Euklid lebte wohl im 3. Jahrhundert vor Christus in Alexandria. In seinen *Elementen* stellte er die Erkenntnisse der damaligen griechischen Mathematik in einheitlicher, systematischer Weise zusammen. Sein methodisches Vorgehen und seine strenge, auf logischen Prinzipien beruhende Beweisführung war beispielhaft und grundlegend für die Mathematik bis in die Neuzeit.

und daraus

$$\log n \geq \alpha_1 + \ldots + \alpha_k \geq \alpha_i,\ 1 \leq i \leq k$$

und hieraus

$$|dual(\alpha_i)| \leq \lfloor \log \log n \rfloor + 1,\ 1 \leq i \leq k$$

Insgesamt folgt nun

$$\mathcal{C}(n) = |b^*(n)| \leq |b(n)| = |\langle\, \alpha_1, \ldots, \alpha_k \,\rangle| \tag{5.3}$$

$$= 2 \sum_{i=1}^{k} (|dual(\alpha_i)| + 1) \tag{5.4}$$

$$\leq k \cdot 2(\lfloor \log \log n \rfloor + 2) \tag{5.5}$$

Dies ist aber ein Widerspruch zu (5.1), denn für jedes beliebige, aber feste $k \geq 1$ gilt

$$\lfloor \log n \rfloor + 1 > k \cdot 2(\lfloor \log \log n \rfloor + 2)$$

für fast alle $n$. Damit haben wir auch auf diese Art und Weise die Annahme, dass $\mathbb{P}$ endlich ist, widerlegt.

## 5.3 Reguläre Sprachen

Der (Widerspruchs-) Beweis, dass eine formale Sprache $L$ nicht regulär ist, wird üblicherweise mithilfe des Pumping-Lemmas für reguläre Sprachen geführt. Wir zeigen im Folgenden an zwei Beispielen, wie mithilfe der Kolmogorov-Komplexität gezeigt werden kann, dass eine Sprache nicht regulär ist.

Als Erstes betrachten wir die nicht reguläre Sprache $L = \left\{0^\ell 1^\ell \mid \ell > 0\right\}$ und nehmen an, dass $L$ regulär ist, dass also ein endlicher deterministischer Automat $A$ mit Zustandsmenge $S$ und Endzustandsmenge $F$ existiert, der $L$ akzeptiert. Zu jedem $k$ gibt es (eindeutig) einen (Nicht-End-) Zustand $s_k$ des Automaten $A$, der nach Abarbeitung von $0^k$ erreicht wird, sowie einen Endzustand, in den $A$ nach Abarbeitung von $1^k$ in $s_k$ beginnend gelangt, und auf dem Weg dorthin liegt kein Endzustand. $A$ und der Zustand $s_k$ können als eine Beschreibung von $k$ angesehen werden. Denn wir können uns ein Programm $P$ überlegen, das $A$ als („fest verdrahteten“) Bestandteil enthält, und das, wenn ihm $s_k$ übergeben wird, eine Folge von Einsen produziert und sich damit auf den Weg zum Endzustand macht. Dieser Endzustand wird nach genau $k$ Einsen erreicht, womit $P$ die Zahl $k$ ausgeben kann.

Da $A$ die Wörter $0^k1^k$ für alle $k \geq 1$ akzeptiert, muss für jedes $k$ ein (Nicht-End-) Zustand $s_k$ existieren. Das heißt, wir können dem Programm $P$ alle möglichen (Nicht-End-) Zustände $s$ übergeben. Das sind endlich viele Eingaben, und $P$ geht für jedes $s$ jeweils wie oben beschrieben vor und kann so alle $k \in \mathbb{N}$ erzeugen.

Das bedeutet, dass wir eine geeignete Codierung $\langle P, s_k \rangle$ von $P$ (inklusive des „fest verdrahteten“ Automaten $A$) und $s_k$ als Beschreibung der Zahl $k$ ansehen können: $b(k) = \langle P, s_k \rangle$. Deren Länge $|b(k)|$ ist unabhängig von $k$, also konstant. Damit gilt $\mathcal{C}(k) \leq |b(k)| = |\langle P, s_k \rangle|$. Sei nun

$$c = \max \{|\langle P, s \rangle| \mid s \in S - F\}$$

dann gilt $\mathcal{C}(k) \leq c$ für alle $k$.

Wir verwenden nun wieder Folgerung 4.1, d.h. die Existenz von Zahlen $n \in \mathbb{N}_0$ mit $\mathcal{C}(n) \geq \lfloor \log n \rfloor + 1$, siehe (5.1). Sei $n_c \in \mathbb{N}_0$ eine Zahl mit

$$\mathcal{C}(n_c) \geq \lfloor \log n_c \rfloor + 1 > c \tag{5.6}$$

Wenn wir nun das Wort $0^{n_c}1^{n_c}$, also $k = n_c$, wählen, folgt mit den obigen Überlegungen $\mathcal{C}(n_c) \leq c$, was offensichtlich ein Widerspruch zu (5.6) ist. Damit ist unsere Annahme, dass $L$ regulär ist, widerlegt.

Für das zweite Beispiel verwenden wir das so genannte *Lemma der KC-Regularität.*

**Lemma 5.1** *Sei $L \subseteq \{0,1\}^*$ eine reguläre Sprache, und für $x \in \{0,1\}^*$ sei $L_x = \{y \mid xy \in L\}$. Des Weiteren sei $f : \mathbb{N}_0 \to \{0,1\}^*$ eine rekursive Aufzählung von $L_x$ (siehe Abschnitt 1.4). Dann existiert eine nur von $L$ und $f$ abhängende Konstante $c$, so dass für jedes $x$ und $y \in L_x$ mit $f(n) = y$ gilt: $\mathcal{C}(y) \leq \mathcal{C}(n) + c$.*

**Beweis** *Sei $A$ der endliche deterministische Automat, der $L$ akzeptiert und $s$ der Zustand, den $A$ nach Abarbeitung von $x$ erreicht. Die Zeichenkette $y$ mit $xy \in L$ und $f(n) = y$ kann nun mithilfe von $A$, $s$ und dem Programm $P_f$, welches $f$ berechnet, bestimmt werden. Geeignete Codierungen von $A$, $s$, $P_f$ und $n$ können somit als Beschreibung von $y$ angesehen werden. Dabei sind die Codierungen von $A$, $s$ und $P_f$ unabhängig von $y$. Deren Gesamtlänge fassen wir mit der Konstanten $c$ zusammen. Damit folgt dann: $\mathcal{C}(y) \leq \mathcal{C}(n) + c$.* □

Wir wenden das Lemma an, um zu zeigen, dass die Sprache $L = \{1^p \mid p \in \mathbb{P}\}$ nicht regulär ist. Wir nehmen an, dass $L$ regulär sei, und betrachten das Wort

$xy = 1^p$ mit $x = 1^{p'}$, wobei $p$ die $k+1$-te Primzahl und $p'$ die $k$-te Primzahl ist. Des Weiteren sei $f$ die lexikografische Aufzählung von $L_x$. Dann gilt $f(1) = y = 1^{p-p'}$. Mit dem Lemma folgt dann: $\mathcal{C}(y) \leq \mathcal{C}(1) + \mathsf{O}(1) = \mathsf{O}(1)$. Das bedeutet, dass die Differenz zwischen allen benachbarten Primzahlen jeweils durch eine von diesen unabhängige Konstante beschränkt ist. Das ist ein Widerspruch zur Tatsache, dass zu jeder Differenz $d \in \mathbb{N}$ zwei benachbarte Primzahlen gefunden werden können, die mindestens den Abstand $d$ haben.

## 5.4 Unvollständigkeit formaler Systeme

Kurt Gödel (siehe Fußnote Seite 15) hat gezeigt, dass die Konsistenz eines hinreichend mächtigen formalen Systems (z.B. ein die Arithmetik natürlicher Zahlen enthaltendes System) nicht innerhalb dieses Systems bewiesen werden kann. Auch diese Unvollständigkeit formaler Systeme, die üblicherweise mit einem Diagonalisierungsbeweis gezeigt wird, kann mithilfe der Kolmogorov-Komplexität gezeigt werden.

**Satz 5.2** *Sei $\mathcal{F}$ ein formales System, dass folgende Eigenschaften erfüllt:*

*(1) Falls es einen korrekten Beweis für eine Aussage $\alpha$ gibt, dann ist diese Aussage auch wahr.*

*(2) Für eine Aussage $\alpha$ und einen Beweis $p$ kann algorithmisch entschieden werden, ob $p$ ein Beweis für $\alpha$ ist.*

*(3) Für jedes $x \in \mathbb{B}^*$ und jedes $n \in \mathbb{N}_0$ kann eine Aussage $\alpha(n, x)$ formuliert werden, die äquivalent zur Aussage „$\mathcal{C}(x) \geq n$“ ist.*

*Dann existiert ein $t \in \mathbb{N}_0$, so dass alle Aussagen $\alpha(x, n)$ für $n > t$ in $\mathcal{F}$ nicht beweisbar sind.*

**Beweis** *Wir betrachten zunächst das Programm $A$ in Abbildung 6: Falls es für eine Eingabe $n$ ein $x$ gibt, für das die Aussage $\alpha(x, n)$ beweisbar ist, dann findet $A$ ein solches $x$.*
*Wir nehmen nun an, dass für alle $n$ ein $x$ existiert, so dass $\alpha(x, n)$ beweisbar ist. Dann findet das Programm $A$ ein solches $x$. Die Beweisbarkeit von $\alpha(x, n)$ bedeutet, dass $\mathcal{C}(x) \geq n$ ist. Des Weiteren ist $b(x) = \langle A, n \rangle$ eine Beschreibung von $x$, denn das Programm $A$ erzeugt $x$ bei Eingabe von $n$. Damit gilt*

$$\mathcal{C}(x) = |b^*(x)| \leq |b(x)| = |\langle A, n \rangle| = 2(|\langle A \rangle| + 1) + |dual(n)| = \lfloor \log n \rfloor + c$$

```
read(n);
   k := 1;
   while true do
      for all x ∈ 𝔹* with |x| ≤ k do
         for all p ∈ 𝔹* with |p| ≤ k do
            if p ein korrekter Beweis für α(x,n) ist
               then return x
            endif
         endfor
      endfor;
      k := k + 1
   endwhile;
```

**Abb. 6: While-Programm $A$ zum Finden von Beweisen für die Aussagen $\alpha(x, n)$.**

*mit $c = 2\,|\langle A \rangle| + 3$. Insgesamt erhalten wir also wieder die Ungleichungen $n \leq \mathcal{C}(x) \leq \lfloor \log n \rfloor + c$, die nur für endliche viele und insbesondere nicht für hinreichend große $n$ erfüllt sind. Damit erhalten wir einen Widerspruch, womit unsere Annahme widerlegt ist.* □

**Bemerkung 5.2** *Der obige Satz zeigt uns eine Möglichkeit auf, in einem hinreichend mächtigen formalen System $\mathcal{F}$, welches die Bedingungen (1) - (3) des Satzes erfüllt, Aussagen zu generieren, die wahr aber innerhalb des Systems nicht beweisbar sind. Wir benutzen das Programm $A$, welches die von $n$ und $x$ unabhänge konstante Länge $c$ hat. Für $n \in \mathbb{N}_0$ mit $n > \lfloor \log n \rfloor + c$ ist keine Aussage der Art $\mathcal{C}(x) \geq n$ beweisbar. Wählen zufällig eine Bitfolge $x$ der Länge $n + 20$, dann gilt die Aussage $\mathcal{C}(x) \geq x$ mit einer Wahrscheinlichkeit $\geq 1 - 2^{-20}$, aber diese Aussage ist in $\mathcal{F}$ nicht beweisbar.* □

## 5.5 Die Kolmogorov-Komplexität entscheidbarer Sprachen

Der folgende Satz besagt, dass entscheidbare Sprachen nur eine (sehr) geringe Kolmogorov-Komplexität besitzen.

**Satz 5.3** *Sei $L \subseteq \mathbb{B}^*$ eine entscheidbare Sprache. Des Weiteren seien die Wörter von $L$ lexikografisch angeordnet (siehe Bemerkung 1.1 d) und $x_n$ das*

*$n$-te Wort in dieser Anordnung. Dann gilt*

$$\mathcal{C}(x_n) \leq \lceil \log n \rceil + \mathsf{O}(1)$$

**Beweis** *Da $L$ entscheidbar ist, ist $\chi_L$ berechenbar. Es sei $C_{\chi_L} \in \mathcal{M}$ das Programm, welches $\chi_L$ berechnet, und $suc$ sei die total berechenbare Funktion, welche zu einem Wort in der lexikografischen Anordnung der Wörter von $\mathbb{B}^*$ den Nachfolger bestimmt (siehe Bemerkung 1.1 c). Die in Abbildung 7 dargestellten Programme $X_n$ generieren jeweils das $n$-te Wort $x_n \in L$. Wie in anderen Beispielen in den vorherigen Abschnitten sind alle Programme bis auf die Zahl $n$ identisch, unabhängig von $n$ hat der identische Anteil eine konstante Länge. Hieraus folgt unmittelbar die Behauptung.* □

```
read();
    i := 0;
    z := ε;
    while i ≤ n do
        if Φ_{C_{χL}}(z) = 1 then
            x_n := z;
            i := i + 1;
        endif;
        z := suc(z);
    endwhile;
write(x_n).
```

**Abb. 7: Programm $X_n$ zur Generierung des $n$-ten Wortes der entscheidbaren Sprache $L$.**

Die Wörter entscheidbarer Sprachen haben also eine geringe Kolmogorov-Komplexität.

**Satz 5.4** *Die Sprache der zufälligen, d.h. nicht komprimierbaren, Bitfolgen*

$$L = \{x \in \mathbb{B}^* \mid \mathcal{C}(x) \geq |x|\}$$

*ist nicht entscheidbar.*

**Beweis** *Wir nehmen an, $L$ sei entscheidbar, d.h. $\chi_L$ ist berechenbar. Es sei $C_{\chi_L} \in \mathcal{M}$ das Programm, welches $\chi_L$ berechnet. In Abbildung 8 ist das Programm $A_N$ dargestellt, welches die erste zufällige Bitfolge $x$ in der lexikografischen Anordnung von $\mathbb{B}^*$ mit $|x| > N$ berechnet (wegen Folgerung 4.1 existiert eine solche Folge). $suc$ sei wieder die total berechenbare Funktion, die zu jeder Folge $x$ ihren Nachfolger in der lexikografischen Anordnung von $\mathbb{B}^*$ bestimmt. Für die Ausgabe $x$ des Programms $A_N$ gilt:*

```
read();
    gefunden := false;
    x := ε;
    while not gefunden do
        if |x| > N and Φ_{C_{χ_L}}(x) = 1 then
            gefunden := true;
        else x := suc(x);
        endif;
    endwhile;
write(x).
```

**Abb. 8: Programm $A_N$ zur Berechnung der ersten zufälligen Bitfolge $x$ mit $|x| > N$.**

$$|x| > N \text{ und } \mathcal{C}(x) \leq \lceil \log N \rceil + c \tag{5.7}$$

*Da $x$ zufällig ist, gilt:*

$$\mathcal{C}(x) \geq |x| \tag{5.8}$$

*Aus (5.7) und (5.8) erhalten wir*

$$N < |x| \leq \mathcal{C}(x) \leq \lceil \log N \rceil + c$$

*was für hinreichend große $N$ offensichtlich einen Widerspruch darstellt.* □

## 5.6 Die Chaitin-Konstante

Zunächst übertragen wir den Begriff *zufällig* auf unendliche Bitfolgen, d.h. wir betrachten jetzt die Elemente von $\mathbb{B}^\omega$. Für eine Element $x = x_1x_2x_3 \ldots \in$

$\mathbb{B}^\omega$ und ein $n \in \mathbb{N}$ sei $x[n] = x_1 \dots x_n$ das Anfangsstück der Länge $n$ von $x$, und es sei $x[n](i) = x_i$ für $1 \leq i \leq n$.

**Definition 5.1** *Eine Bitfolge $x \in \mathbb{B}^\omega$ heißt* **zufällig** *genau dann, wenn eine Konstante $c \in \mathbb{N}$ existiert, so dass*

$$\mathcal{C}(x[n]) > n - c$$

*für alle $n \in \mathbb{N}$ gilt.* □

Den folgenden Betrachtungen legen wir wieder eine Gödelisierung $(\mathbb{N}_0, \mathcal{P}, \varphi)$ einer Programmiersprache $\mathcal{M}$ zugrunde.

**Definition 5.2** *Die Bitfolge $H = h_1 h_2 h_3 \dots \in \mathbb{B}^\omega$ definiert durch*

$$h_i = \begin{cases} 1, & \varepsilon \in \mathit{Def}(\varphi_i) \\ 0, & \textit{sonst} \end{cases}$$

*heißt* **Haltesequenz**. □

Die Haltesequenz $H$ ist also eine unendliche Bitfolge, deren $i$-tes Bit gleich $1$ ist, wenn das $i$-te Programm (bei Eingabe des leeren Wortes) anhält, ansonsten ist das Bit gleich $0$. Es ist klar, dass $H$ von der Gödelisierung abhängt. Wir halten fest, dass aus der Nummer $i$ das Programm $\nu(i) = A$ rekonstruiert und dass die Sequenz $H$ wegen der Unentscheidbarkeit des Halteproblems nicht (vollständig) berechnet werden kann.

**Bemerkung 5.3 a)** *Die Sequenz $H[k]$ gibt an,*

*(1) wie viele der ersten $k$ Progamme anhalten und*

*(2) welche Programme das sind.*

*Dabei könnten wir die Information (2) aus der Information (1) gewinnen: Sei $l \leq k$ die Anzahl der terminierenden Programme unter den $k$ ersten. Wir lassen alle $k$ Programme parallel laufen, markieren die haltenden Programme und zählen diese. Falls $l$ Programme gestoppt haben, können wir alle anderen Programme auch anhalten.*

**b)** *Wenn wir wüssten, wie groß die Wahrscheinlichkeit ist, mit der sich unter den $k$ ersten Gödelnummern solche von terminierenden Programmen befinden, dann könnten wir die Anzahl $l$ der terminierenden Programme durch Multiplikation mit $k$ bestimmen, denn es gilt*

$$\mathit{Prob}\,[H[k](i) = 1] = \frac{l}{k}$$

*woraus sich bei gegebenem $k$ und bekannter Wahrscheinlichkeit $Prob\,[h_i = 1]$*

$$l = k \cdot Prob\,[H[k](i) = 1]$$

*berechnen lässt.*

**c)** *Im Folgenden geht es im Wesentlichen darum, wie $Prob\,[H[k](i) = 1]$ bestimmt werden kann. Daraus könnten $l$ und damit gemäß a) die terminierenden Programme $A_i$, $1 \leq i \leq k$, bestimmt werden.* □

Hilfreich dafür ist

**Definition 5.3** *Es sei $x \in \mathbb{B}^n$. Dann heißt $\Omega_n =$*

*Prob*[$x$ beginnt mit der Gödelnummer eines terminierenden Programms]

**Haltewahrscheinlichkeit**. □

**Beispiel 5.1** *Sei etwa*

$$H = 00100000110000001010\ldots$$

*Es ist also $h_3 = h_9 = h_{10} = h_{17} = h_{19} = 1$. Die entsprechenden Gödelnummern seien $a_1 = 11$, $a_2 = 011$, $a_3 = 0101$, $a_4 = 1010$, $a_5 = 0010$. Es ergeben sich folgende Haltewahrscheinlichkeiten:*

1. $\Omega_1 = 0$, *denn es gibt kein Programm mit einstelliger Gödelnummer.*
2. $\Omega_2 = \frac{1}{4}$, *denn $11 \in \mathbb{B}^2$ beginnt mit der Gödelnummer $a_1 = 11$.*
3. $\Omega_3 = \frac{3}{8}$, *denn $110, 111 \in \mathbb{B}^3$ beginnen mit der Gödelnummer $a_1 = 11$ und $011 \in \mathbb{B}^3$ beginnt mit der Gödelnummer $a_2 = 011$.*
4. $\Omega_4 = \frac{9}{16}$, *denn $1100, 1101, 1110, 1111 \in \mathbb{B}^4$ beginnen mit der Gödelnummer $a_1 = 11$; $0110, 0111 \in \mathbb{B}^4$ beginnen mit der Gödelnummer $a_2 = 011$; und $0101, 1010, 0010 \in \mathbb{B}^4$ sind die Gödelnummern $a_3 = 0101$, $a_4 = 1010$ und $a_5 = 0010$.* □

Wir können uns vorstellen, eine Gödelnummer durch Münzwurf zu generieren. Ist die so entstehende Bitfolge eine Gödelnummer, können wir aufhören zu würfeln, denn wegen der Präfixfreiheit kann keine weitere Gödelnummer entstehen. Es kann vorkommen, dass keine Gödelnummer gewürfelt wird (das Würfeln stoppt nicht).

Wir können alle möglichen Würfelsequenzen in einem Entscheidungsbaum darstellen. Dieser Baum enthält unendlich lange Pfade oder solche, die zu Blättern führen und mit der Gödelnummer eines Programmes markiert sind.

Im Beispiel 5.1 wird das Programm mit der Gödelnummer $a_1$ mit der Wahrscheinlichkeit $\frac{1}{4}$ erzeugt, das Programm $a_2$ mit der Wahrscheinlichkeit $\frac{1}{8}$ und die Programme $a_3$, $a_4$ und $a_5$ jeweils mit der Wahrscheinlichkeit $\frac{1}{16}$. Daraus ergibt sich

$$\begin{aligned}\Omega_1 &= 0\\ \Omega_2 &= \frac{1}{4}\\ \Omega_3 &= \frac{1}{4} + \frac{1}{8}\\ \Omega_4 &= \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \frac{1}{16} + \frac{1}{16}\end{aligned}$$

Ist $A$ ein terminierendes Programm und $|\langle A \rangle|$ die Länge seiner Gödelnummer, dann gilt

$$\Omega_n = \sum_{\substack{A \text{ terminiert}\\ |\langle A \rangle| \leq n}} \frac{1}{2^{|\langle A \rangle|}} \tag{5.9}$$

**Bemerkung 5.4** *In der binären Darstellung von $\Omega_n$ sind die Antworten zu allen mathematischen Fragestellungen codiert, die sich über die Terminierungseigenschaft eines Programms der Länge kleiner oder gleich $n$ berechnen lassen.*

*Betrachten wir als Beispiel die Goldbachsche Vermutung: „Jede gerade natürliche Zahl größer zwei lässt sich als Summe zweier Primzahlen darstellen.“ Diese Vermutung ist für eine große Anzahl von Fällen bestätigt, aber nicht endgültig bewiesen. Wir können uns ein Programm $G \in \mathcal{M}$ überlegen, welches die erste Zahl größer zwei sucht, die sich nicht als Summe zweier Primzahlen darstellen lässt. Sei $|\langle G \rangle| = n$. Die Anzahl der Gödelnummern von Programmen mit einer Länge kleiner oder gleich $n$ lässt sich bestimmen. Wenn wir $\Omega_n$ kennen würden, dann könnten wir mit diesen beiden Angaben die Anzahl der Bitfolgen bestimmen, die mit der Gödelnummer eines terminierenden Programms beginnen. Aus der Bemerkung 5.3 folgt, dass wir mithilfe dieser Kenntnisse herausfinden können, ob $G$ terminiert oder nicht. Falls $G$ terminiert, wäre die Goldbachsche Vermutung falsch, falls $G$ nicht terminiert, wäre sie wahr.*

*Dieses Beispiel verdeutlicht die in Bemerkung 5.3 c) geäußerte Bedeutung der Haltesequenz $\Omega_n$ für die Lösung von Entscheidungsfragen, insbesondere auch von derzeit offenen Entscheidungsfragen.* □

**Folgerung 5.1** **a)** *Die Folge $\{\Omega_n\}_{n\geq 1}$ ist monoton wachsend, d.h. es ist $\Omega_n \leq \Omega_{n+1}$.*

**b)** *Die Folge $\{\Omega_n\}_{n\geq 1}$ ist nach oben beschränkt, denn es gilt: $\Omega_n \leq 1$ für alle $n \in \mathbb{N}$.*

**c)** *Die Folge $\{\Omega_n\}_{n\geq 1}$ ist konvergent.*

**Beweis** **a)** *ist offensichtlich.*

**b)** *Es gilt*

$$\Omega_n = \sum_{\substack{A \text{ terminiert} \\ |\langle A\rangle|\leq n}} \frac{1}{2^{|\langle A\rangle|}} \leq \sum_{i=1}^{n} \left(\frac{1}{2}\right)^i = \frac{1-\left(\frac{1}{2}\right)^{n+1}}{1-\frac{1}{2}} - 1 = 1 - \left(\frac{1}{2}\right)^n < 1$$

**c)** *folgt unmittelbar aus a) und b).* □

**Definition 5.4** *Der Grenzwert*

$$\Omega = \lim_{n\to\infty} \Omega_n = \sum_{A \text{ terminiert}} \frac{1}{2^{|\langle A\rangle|}}$$

*heißt* **Chaitin-Konstante**. □

Wir wollen uns mit der Frage befassen, ob sich $\Omega$ bestimmen lässt. Dazu variieren wir die Definition 5.3 wie folgt.

**Definition 5.5** *Es sei $x \in \mathbb{B}^n$. Dann heißt $\Omega_n^k =$*

*Prob*[$x$ beginnt mit der Gödelnummer eines terminierenden Programms, das in $k$ Schritten stoppt]

**Haltewahrscheinlichkeit** *(der in $k$ Schritten terminierenden Programme).* □

Offensichtlich gilt

**Folgerung 5.2** **a)** $\Omega_n^n \leq \Omega_n \leq \Omega$.

**b)** $\lim_{n\to\infty} \Omega_n^n = \lim_{n\to\infty} \Omega_n = \Omega$. □

Trotz dieser gleichen Verhaltensweisen sind die Folgen $\{\Omega_n\}_{n\geq 1}$ und $\{\Omega_n^n\}_{n\geq 1}$ wesentlich verschieden: Im Gegensatz zur ersten Folge sind alle Glieder der zweiten Folge berechenbar. $\Omega_n^n$ kann bestimmt werden, indem jedes Programm $A$ mit $|\langle A \rangle| \leq n$ maximal $n$ Schritte ausgeführt und dabei gezählt wird, wie viele terminieren. Je größer $n$ gewählt wird, je genauer kann so der Grenzwert $\Omega$ bestimmt werden: Die Nachkommabits von $\Omega_n^n$ müssen sich bei wachsendem $n$ von links nach rechts stabilisieren. Allerdings weiß man nie, wann ein Bit sich nicht mehr ändert – die Änderung weit rechts stehender Bits kann durch Überträge Einfluss auf links stehende Bits haben.

**Satz 5.5** *Die Haltewahrscheinlichkeit $\Omega_n$ kann aus $\Omega[n]$, d.h. aus den ersten $n$ Bits der Chaitin-Konstante, bestimmt werden.*

**Beweis** *Abbildung 9 stellt ein Verfahren dar, mit dem $\Omega_n$ aus $\Omega[n]$ konstruiert werden kann. Wir bestimmen $\Omega_1^1, \Omega_2^2, \Omega_3^3, \ldots$ Die Werte der Folgenglieder nähert sich immer mehr an $\Omega$ an. Irgendwann wird ein $t$ erreicht, so dass $\Omega_t^t$ und $\Omega$ in den ersten $n$ Bits übereinstimmen.*

*Für $s > t$ können sich die ersten $n$ Bits von $\Omega_s^s$ nicht mehr ändern, denn dann wäre $\Omega_s^s > \Omega$, was ein Widerspruch zur Folgerung 5.2 a) ist. Da jedes Programm $A$ mit $\frac{1}{2^{|\langle A \rangle|}}$ in die Berechnung von $\Omega_s^s$ eingeht und sich die ersten $n$ Bits nicht mehr ändern können, muss jedes Programm, dass nach mehr als $t$ Schritten anhält, eine Länge größer $n$ haben.*

*Es folgt, dass es kein Programm $A$ mit $|\langle A \rangle| \leq n$ geben kann, dass nach mehr als $t$ Schritten anhält. Deshalb können wir $\Omega_n$ wie folgt bestimmen: Es werden alle Programme $A$ mit $|\langle A \rangle| \leq n$ $t$ Schritte ausgeführt. Terminiert ein Programm $A$ innerhalb dieser Schrittzahl, dann geht $\frac{1}{2^{|\langle A \rangle|}}$ in die Berechnung von $\Omega_n$ ein. Hält ein Programm nicht innerhalb dieser Schrittzahl, dann terminiert es nie und $\frac{1}{2^{|\langle A \rangle|}}$ wird nicht berücksichtigt.* □

**Bemerkung 5.5** **a)** *Das in Abbildung 9 dargestellte Verfahren ist nicht berechenbar. Seine Berechenbarkeit würde voraussetzen, dass das kleinste $t$, für das $\Omega_t^t[n] = \Omega[n]$ gilt, effektiv berechnet werden kann, d.h. dass die Funktion*

$$f(n) = \min\left\{t \mid \Omega_t^t[n] = \Omega[n]\right\}$$

*berechenbar ist. Es kann gezeigt werden, dass $f$ schneller wächst als jede berechenbare Funktion, woraus folgt, dass $f$ nicht berechenbar ist.*

```
read(Ω[n]);
   t := 1;
   while Ω_t[n] ≠ Ω[n] do
      t := t + 1;
   endwhile;
 // Jedes terminierende Programm A mit |⟨A⟩| ≤ n
 // hält in höchstens t Schritten;
   A_n^t := Menge aller Bitfolgen
         mit einer Länge ≤ n,
         die mit der Gödelnummer
         eines Programms beginnen,
         das in höchstens t Schritten stoppt;

   Ω_n := |A_n^t| / 2^n;
write(Ω_n).
```

**Abb. 9: Verfahren zur Konstruktion von $\Omega_n$ aus $\Omega[n]$.**

**b)** *Die Chaitin-Konstante $\Omega$ enthält das Wissen über alle Haltewahrscheinlichkeiten.*[15] *In ihr versteckt sind die Antworten auf alle mathematischen Entscheidungsfragen, wie die Entscheidbarkeit des Halteproblems für alle möglichen Programme. Allerdings ist $\Omega$ nicht berechenbar, d.h. wir sind nicht in der Lage, das in $\Omega$ verborgene Wissen zu erkennen.*

**c)** *Die Haltesequenz $H$ lässt sich komprimieren. Die ersten $2^n$ Bits von $H$ lassen sich aus $\Omega_n$ und $\Omega_n$ lässt sich aus $\Omega[n]$ rekonstruieren. Damit kann jedes Anfangsstück der Länge $m$ von $H$ mit einem Programm generiert werden, dessen Länge logarithmisch in $m$ wächst. Somit ist $H$ keine Zufallszahl.*

**d)** *$\Omega$ ist eine rekursiv aufzählbare Zahl.*

**e)** *In* (Ca02) *werden die ersten $64$ Ziffern von $\Omega$ berechnet.* □

**Satz 5.6** *Die Chaitin-Konstante ist zufällig.*

**Beweis** *Gemäß Satz 5.5 können wir aus den ersten $n$ Bits von $\Omega$ die Menge $T_n$ aller terminierenden Programme mit einer Länge kleiner gleich $n$ bestimmen.*

[15] Diese Konstante wird in einigen Literaturstellen auch „Chaitins Zufallszahl der Weisheit“ genannt.

*Wir führen diese aus und sammeln deren Ausgaben in der Menge $M$.*

*Nun betrachten wir das in Abbildung 10 dargestellte Programm $A_n$. Dabei sei $\nu(i)$ wieder die total berechenbare Funktion, welche die $i$-te Bitfolge in der lexikografischen Anordnung von $\mathbb{B}^*$ bestimmt.*

```
read();
   Berechne M aus Ω[n];
   i := 1;
   x := ν(i);
   while x ∈ M do
      i := i + 1;
      x := ν(i);
   endwhile;
write(x).
```

**Abb. 10: Das Programm $A_n$ gibt die erste Bitfolge in der lexikografischen Anordnung von $\mathbb{B}^*$ aus, die von keinem terminierenden Programm mit einer Länge kleiner oder gleich $n$ ausgegeben wird.**

*Das Programm gibt die erste Bitfolge $x$ in der lexikografischen Anordnung von $\mathbb{B}^*$ aus, die von keinem Programm in $T_n$ erzeugt wird. Es gilt*

$$n < \mathcal{C}(x) \leq |\langle A_n \rangle| \tag{5.10}$$

*Abgesehen von der Berechnung von $\Omega[n]$ ist die Länge des Programmes $A_n$ fest. Wir setzen*

$$c = |\langle A_n \rangle| - \mathcal{C}(\Omega[n])$$

*Dieses eingesetzt in (5.10) liefert*

$$n < \mathcal{C}(x) \leq c + \mathcal{C}(\Omega[n])$$

*woraus*

$$\mathcal{C}(\Omega[n]) > n - c$$

*folgt, womit die Behauptung gezeigt ist.* □

**Bemerkung 5.6** **a)** *Die Chaitin-Konstante ist also nicht komprimierbar. Das Wissen über alle Haltewahrscheinlichkeiten $\Omega_n$ kann also im Wesentlichen nicht kürzer angegeben werden als durch die Chaitin-Konstante selbst.*

**b)** *In* (Ku01) *wird quasi die Umkehrung von Bemerkung 5.5 d) und Satz 5.6 gezeigt: Die rekursiv aufzählbaren Zufallszahlen sind genau die $\Omega$-Nummern, d.h. zu jeder rekursiv aufzählbaren Zufallszahl $\Omega' \in (0,1)$ können eine vollständige Programmiersprache definiert und eine Codierung ihrer Programme über $\mathbb{B}^*$ gefunden werden, so dass $\Omega' = \Omega$ für diese Programmiersprache und die Codierung ist.*[16] □

# Epilog

Diese Ausarbeitung ist eine erste Einführung in die Algorithmische Informationstheorie, die lange Zeit ein Spezialthema gewesen ist, mit dem sich nur wenige Experten beschäftigt haben und welches bis auf ganz wenige Ausnahmen keine Rolle in mathematischen oder informatischen Studiengängen gespielt hat. Seit einigen Jahren ist eine starke Belebung zu beobachten, was sich unter anderem durch die Veröffentlichung von Lehrbüchern widerspiegelt, die dieses Thema mehr oder weniger ausführlich behandeln und in Zusammenhang mit anderen Themen setzen. Des Weiteren ist zu beobachten, dass Konzepte und Methoden der Theorie Eingang in praktische Anwendungen finden. Dabei spielt oft eine Variante der Kolmogorov-Komplexität, die so genannte **bedingte Kolmogorov-Komplexität**, eine Rolle, die wir in dieser Ausarbeitung nicht betrachtet haben:

$$\mathcal{C}(x|y) = \min\{|\langle A \rangle| : A \in \mathcal{M} \text{ mit } \Phi_U \langle A, y \rangle = x\}$$

ist die minimale Länge eines Programms, welches $x$ aus $y$ berechnet; zum Erzeugen der Bitfolge $x$ können die Programme $A \in \mathcal{M}$ die Hilfsinformation $y$ benutzen.[17] Offensichtlich ist $\mathcal{C}(x) = \mathcal{C}(x|\varepsilon)$. Des Weiteren ist einsichtig, dass $\mathcal{C}(x|y) \leq \mathcal{C}(x) + \mathsf{O}(1)$ ist, und, je „ähnlicher“ sich $x$ und $y$ sind, d.h. je einfacher ein Programm ist, welches $x$ aus $y$ berechnet, um so mehr werden sich $\mathcal{C}(x)$ und $\mathcal{C}(x|y)$ von einander unterscheiden. So misst $I(y:x) = \mathcal{C}(x) - \mathcal{C}(x|y)$ die Information, die $y$ von $x$ enthält; $I(y:x)$ ist quasi die Bedeutung, die $y$ für $x$ hat.

[16] Mit $(a,b)$ für $a, b \in \mathbb{R}$ mit $a \leq b$ ist hier das offene Intervall der reellen Zahlen zwischen $a$ und $b$ gemeint: $(a,b) = \{x \in \mathbb{R} \mid a < x < b\}$.

[17] Falls kein $A$ existiert mit $\Phi_U \langle A, y \rangle = x$, dann setzen wir $\mathcal{C}(x|y) = \infty$.

Als **Informationsunterschied** zwischen $x$ und $y$, kann man etwa

$$E(x,y) = \min\{|A| : \Phi_U \langle A, x \rangle = y \text{ und } \Phi_U \langle A, y \rangle = x\}$$

festlegen. Man kann dann zeigen, dass $E(x,y) \approx \max\{\mathcal{C}(x|y), \mathcal{C}(y|x)\}$ gilt, d.h. genauer, dass

$$E(x,y) = \max\{\mathcal{C}(x|y), \mathcal{C}(y|x)\} + \mathsf{O}(\log \max\{\mathcal{C}(x|y), \mathcal{C}(y|x\})$$

gilt. Man kann zeigen, dass die Abstandsfunktion $E$ in einem gewissen Sinne minimal ist. Das heißt, dass für alle anderen möglichen Abstandsfunktionen $D$ für alle $x, y \in \mathbb{B}^*$ gilt: $E(x,y) \leq D(x,y)$ (abgesehen von einer additiven Konstante). Durch die Normalisierung von $E$ durch

$$e(x,y) = \frac{\max\{\mathcal{C}(x|y), \mathcal{C}(y|x)\}}{\max\{\mathcal{C}(x), \mathcal{C}(y)\}}$$

erhält man eine relative Distanz zwischen $x$ und $y$ mit Werten zwischen $0$ und $1$, die eine Metrik darstellt.

Es besteht damit die Möglichkeit, auf dieser Basis ein Ähnlichkeitsmaß zwischen Bitfolgen einzuführen, um Probleme im Bereich der Datenanalyse zu lösen, wie z.B. der Vergleich von Gensequenzen oder das Entdecken von Malware (SPAM). In der praktischen Anwendung kann dabei natürlich nicht die Kolmogorov-Komplexität $\mathcal{C}$ verwendet werden, denn diese ist ja nicht berechenbar (siehe Satz 4.4). Hier muss man in der Praxis verwendete Kompressionsverfahren verwenden, wie z.B. Lempel-Ziv- oder ZIP-Verfahren. Wenn wir die Kompressionsfunktionen dieser Verfahren mit $\kappa$ bezeichnen, gilt natürlich $\mathcal{C}(x) \leq \kappa(x)$ für alle $x \in \mathbb{B}^*$, denn die Kolmogorov-Komplexiät $\mathcal{C}(x)$ ist ja die kürzeste Komprimierung von $x$.

Aus diesen Überlegungen lassen sich eine Reihe von Forschungs- und Entwicklungsthemen ableiten, die im Rahmen von Praxis-, Projekt- und Abschlussarbeiten bearbeitet werden können, wie z.B.

- Entwicklung einer Simulationsumgebung, mit der Algorithmen zur Datenanalyse entwickelt, installiert und getestet werden können;
- Konzeption, Entwicklung und Test von Algorithmen, welche auf der Basis der obigen Überlegungen Maße für die Berechnung der Komplexität von Bitfolgen implementieren;

- Konzeption, Entwicklung und Test von Algorithmen zur Berechnung des Informationsunterschieds und der Klassifizierung von Bitfolgen ;
- Anwendung, Vergleich und Bewertung der Algorithmen in verschiedenen Amnwendungsszenarien;
- Vergleich der Ansätze mit derzeit in den Praxis eingesetzten Verfahren, die nicht auf (der syntaktischen) Ebene der Bitfolgen, sondern auf der Basis von statistischen oder semantischen Informationen Daten analysieren.

In Gesprächen mit Kooperationspartnern haben diese ihr Interesse an diesen Themen bekundet.

# Danksagung

Der Autor dankt Oliver Lanzerath, MSc, für Hinweise, die an vielen Stellen zu einer präziseren und lesbareren Darstellung beigetragen haben.

# Literatur

(Ca94) Calude, C. S.: *Information and Randomness, 2nd Edition.* Springer, Berlin, 2010

(Ca98) Calude, C. S., Hertling, P. H., Khoussainov, B., Wang, Y.: Recursively enumerable reals and Chaitin $\Omega$ numbers, in *Proceedings of the Symposium on Theoretical Aspects of Computer Science (STACS 98).* Springer, Heidelberg, 1998, 596 – 606

(Ca02) Calude, C. S., Dinneen, M. J., Shu, C.-K.: Computing a Glimpse of Randomness, *Exper. Math.* **11**, 2002, 361 – 370

(Ca07) Calude, C. S., Dinneen, M. J.: Exact Approximations of Omega Numbers, *Int. J. Bifur. Chaos* **17**, 2007, 1937 – 1954

(Ch66) Chaitin, G.: On the length of programs for computing binary sequences, *Journal of the ACM* **13**, 1966, 547 – 569

(Ch691) Chaitin, G.: On the length of programs for computing binary sequences: Statistical considerations, *Journal of the ACM* **16**, 1969, 145 – 159

(Ch692) Chaitin, G.: On the simplicity and speed of programs for computing infinite sets of natural numbers, *Journal of the ACM* **16**, 1969, 407 – 412

(Ch693) Chaitin, G.: On the difficulty of computations, *IEEE Transactions on Information Theory* **16**, 1969, 5 – 9

(Ch741) Chaitin, G.: Information-theoretic computational complexity, *IEEE Transactions on Information Theory* **20**, 1974, 10 – 15

(Ch742) Chaitin, G.: Information-theoretic limitations of formal systems, *Journal of the ACM* **21**, 1974, 403 – 424

(Ch75) Chaitin, G.: A theory of program size formally identical to information theory, *Journal of the ACM* **22**, 1975, 329 – 340

(Ch98) Chaitin, G.: *The Limits of Mathematics.* Springer, Heidelberg, 1998

(Ch99) Chaitin, G.: *The Unknowable.* Springer, Heidelberg, 1999

(Ch02) Chaitin, G.: *Conversations with a Mathematician.* Springer, Heidelberg, 2002

(Ch07) Chaitin, G.: *Meta Maths: The Quest of Omega.* Atlantic Books, London, 2007

(Hof11) Hoffmann, D. W.: *Grenzen der Mathematik.* Spektrum Akademischer Verlag, Heidelberg, 2011

(Ho041) Homkrovič, J.: *Theoretische Informatik, 4. Auflage.* Viweg+Teubner, Wiesbaden, 2011

(Kol65) Kolmogorov, A.: Three approaches for defining the concept of information quantity, *Problems of Information Transmission* **1**, 1965, 1 – 7

(Kol68) Kolmogorov, A.: Logical basis for information theory and probabilistic theory, *IEEE Transactions on Information Theory* **14**, 1968, 662 – 664

(Kol69) Kolmogorov, A.: On the logical foundations of information theory and probability theory, *Problems of Information Transmission* **5**, 1969, 1 – 4

(Ku01) Kučera, A., Slaman, T. A.: Randomness and recursive enumerability, *SIAM Journal of Computing* **31**, 199 – 211, 2001

(LiV93) Li, M., Vitànyi, P.: *Introduction to Kolmogorov Complexity and Its Applications.* Springer, Heidelberg, 1993

(LiV08) Li, M., Vitànyi, P.: A New Approach to Formal Language Theory by Kolmogorov Complexity, *arXiv:cs/0110040v1* $[\, cs.CC \,]$ *18 oct 2001*

(Sip06) Sipser, M.: *Introduction to the Theory of Computation, second Edition, International Edition.* Thomson, Boston, MA, 2006

(So164) Solomonoff, R. J.: A formal theory of inductive Inference, Part I, *Information and Control* **7**, 1964, 1 – 22

(So264) Solomonoff, R. J.: A formal theory of inductive Inference, Part II, *Information and Control* **7**, 1964, 224 – 254

(VoWi16) Vossen, G., Witt, K.-U..: *Grundkurs Theoretische Informatik, 6. Auflage.* Springer Vieweg, Wiesbaden, 2016

(We11) Weisstein, E. W.: Chaitin's Constant, *From MathWorld – A Wolfram Web Resource.* http://mathworld.wolfram.com/ChaitinsConstant.html, June 26th, 2011

# Index