Kolmogoroff-Komplexität

Vortragsausarbeitung (2. Teil)

Sandro Köppen
Mirko Hochmuth

Berlin, 2001

Einleitung

Im zweiten Teil wollen wir uns mit Anwendungen der Kolmogoroff-Komplexität beschäftigen. Mithilfe der K-Komplexität lassen sich Beweise zu unteren Komplexitätsschranken durchführen.
Nach einer kurzen Wiederholung wollen wir zeigen das es unendlich viele Primzahlen gibt und einen „schwachen" Primzahlsatz beweisen. Danach wird die Schaltkreiskomplexität im Zusammenhang mit der Kolmogoroff-Komplexität betrachtet und zum Ende analysieren wir die Laufzeit von Turingmaschinen auf bestimmte Sprachen.

Wiederholung

1. Kolmogoroff-Komplexität

Im letzten Vortrag haben wir die K-Komplexität vorgestellt und definiert:

$K(x|n)$ ist die Länge des kleinsten Algorithmus, der bei Eingabe n das Wort x ausgibt.

Dies ist die relative K-Komplexität, die absolute Kolmogoroff-Komplexität wird angegeben, wenn wir keine Eingabe haben:

$$K(x) = K(x|e)$$

2. Zufälligkeit

Zufällige Wörter haben eine maximale K-Komplexität ($K(x)=O(1)+|x|$), lassen sich also nicht komprimieren, d.h. man findet keinen Algorithmus der x beschreibt, ohne im Wesentlichen $|x|$ Bits zu benötigen.

3. Universelle Wahrscheinlichkeitsverteilung

Mithilfe der K-Komplexität haben wir eine Wahrscheinlichkeitsverteilung derart definiert, dass Wörter mit geringer Komplexität häufiger auftreten als rein zufällige Wörter:

$$m(x) = c / 2^{2K(x|n)}$$

4. Laufzeit von Algorithmen

Zum Schluss führten wir den Beweis das es eine Wahrscheinlichkeitsverteilung gibt, unter der die durchschnittliche Laufzeit eines (stoppenden) Algorithmus bis auf einen konstanten Faktor mit seiner Schlechtesten-Fall-Laufzeit zusammenfällt:

$$T_{av}(n) = c \cdot T_{wc}(n)$$

Wahl der (Programmier-) Sprache

Wir wollen nun zeigen, dass es im Prinzip egal ist, für welche (universelle) Programmiersprache man sich entscheidet. Die einzelnen Sprachen unterscheiden sich in ihrer Kolmogoroff-Komplexität nur durch eine aufaddierte Konstante. Diese kann im Regelfall vernachlässigt werden.

Gegeben ist ein Interpreterprogramm u, z.B. eine universelle Turingmaschine, geschrieben in einer wiederum universellen Sprache P. Dieses Programm verhält sich bei der Eingabe eines Programms p' (in einer Programmiersprache P' geschrieben) und der dazugehörigen Eingabe y genauso wie das Programm p' bei Eingabe y. Wir bezeichnen mit K_P die Kolmogoroff-

Komplexität in Bezug auf die Sprache P. Wenn p' das kleinste Programm ist, dann gilt also $K_{P'}(x|y)=|p'|$. Unser Interpreter erzeugt auch x mit der Eingabe p'y. So erhalten wir $K_P(x|p'y)\leq|u|$ und wegen $K_P(p')\leq|p'|+c$ erhalten wir:

$$K_P(x|y)\leq|p'|+c+|u| = K_{P'}(x|y)+c+|u| = K_{P'}(x|y)+O(1)$$

Wenn wir also die Konstante vernachlässigen, können wir allgemein von einer Kolmogoroff-Komplexität eines Wortes x sprechen, ohne die jeweilige Programmiersprache betrachten zu müssen.

Es gibt unendlich viele Primzahlen

Als einführendes Beispiel für eine Anwendung der Kolmogoroff-Komplexität wollen wir durch Widerspruch zeigen das es unendlich viele Primzahlen gibt:

Bemerkung:

Mit bin(n) ist die Binärdarstellung des Wortes oder der Zahl n beschrieben.

Annahme:

Es gibt nur endlich viele Primzahlen:

$p_1,p_2, ...,p_k$

Festlegung:

Das Wort n hat maximale K-Komplexität, läst sich also nicht komprimieren:

$$K(bin(n)) \geq |bin(n)| = \log n$$

Wir können n nun in seine Primfaktoren zerlegen:

$$n = p_1^{n1} \cdot p_2^{n2} \cdot ... \cdot p_k^{nk}$$

n kann somit aus den Zahlen n_1 bis n_k beschrieben werden.

Nun ist $p_i\geq2$, daraus folgt $n_i \leq \log n$. Somit ist $|bin(n_i)| \leq \log\log n$.
Für die gesamte Folge $n_1, n_2, ..., n_k$ brauchen wir somit $O(k \cdot \log\log n)$ Platz. Da k konstant und von n unabhängig ist, können wir schreiben:

$|n_1,n_2,...,n_k| = O(\log\log n)$.

Das Wort n lässt sich also mit dem Platzbedarf von $O(\log\log n)$ beschreiben. Nun gilt aber $O(\log\log n) < O(1)+|n|$, was im Widerspruch zu unserer obigen Festlegung steht, das n maximale Kolmogoroff-Komplexität hat und sich nicht komprimieren lässt.

Primzahlsatz

Der Primzahlsatz ist ein äußerst schwer zu beweisender Satz der Zahlentheorie, der Folgendes besagt:

$$\pi(n) \approx n / \ln n$$

Mit deutlich einfacheren Argumenten lässt sich der folgende (schwache) Primzahlsatz zeigen, der aber in vielen Fällen ausreicht:

$$\pi(n) \geq n / \log^2 n$$

Sei p_m die m-te Primzahl. Es genügt nun zu zeigen, dass folgende Ungleichung für unendlich viele n gilt:

$$p_m \leq m \cdot \log^2 m$$

Dann können wir n auf p_m und $\pi(n)$ auf m setzten und erhalten:

$$n \leq \pi(n) \cdot \log^2 \pi(n)$$

was zu unserem schwachen Primzahlsatz führt:

$$\pi(n) \geq n / \log^2 \pi(n) \geq n / \log^2 n$$

Sei nun n eine genügend große natürliche Zahl mit maximaler Kolmogoroff-Komplexität ($K(bin(n)) \geq |bin(n)|$) und sei p_m die größte Primzahl, die n teilt.
Aus m und n/p_m (k) können wir n eindeutig rekonstruieren (der Algorithmus müsste einfach die m-te Primzahl berechnen und diese dann mit k multiplizieren). Das heißt also die Angabe dieser beiden Zahlen (geeignet als Bitstring codiert) beschreibt n. Deshalb gilt:

$$K(bin(n)) \leq O(1) + |\text{Code von m und k}|$$

Wir müssen also eine möglichst gute Codierungsmöglichkeit für m und k finden.
Erster Versuch: Wir codieren m selbstbeendend, das heißt wir schreiben zwischen jede Binärstelle von m eine 0, an das Ende von m schreiben wir eine 1. Aus 0011010 wird dann 00001010001001. Auf diese Weise kann man m und k eindeutig voneinander trennen.
Wir benötigen hiermit also einen Platz von $2 \cdot \log m + \log k$ woraus folgt:

$$\log n \leq K(bin(n)) \leq 2 \cdot \log m + \log n - \log p_m$$

$$p_m \leq m^2$$

Diese Art reicht also noch nicht ganz aus, aber wir verschenken auch noch einige wertvolle Bits. Unser zweiter Versuch wird nun sein, nicht m sondern nur die Länge von m selbstbeendend zu kodieren und dann m hinten anfügen. Aus 0011010 wird dann also 1010110011010. Hierfür benötigen wir offensichtlich nur $2 \cdot \log(\log m) + \log m$ viel Platz. Daraus folgt:

$$\log n \leq \log(\log^2 m) + \log m + \log n - \log p_m.$$

Was letztendlich zu der gewünschten Ungleichung führt:

$$p_m \leq m \log^2 m$$

qed.

Bemerkung:
Man könnte dies immer weiter treiben (also nur die Länge der Länge selbstbeendend zu codieren, dann die Länge der Länge der Länge, etc.) und würde einen immer schärferen Primzahlsatz beweisen können, an den „richtigen“ Primzahlsatz werden wir so jedoch nicht herankommen, besser als $\pi(n) \geq n/\log n + (\log \log n)^2$ werden wir mit dieser Methode nicht.

Schaltkreiskomplexität

Mithilfe der K-Komplexität können wir eine untere Schranke für Schaltkreise angeben. Die Schranke bezieht sich auf die Anzahl der Gatter, die mindestens benötigt werden, um eine boolesche Funktion umzusetzen.

Zuerst soll charakterisiert werden was wir unter einer solchen Funktion verstehen und wie man sie darstellen kann. Eine boolesche Funktion lautet wie folgt:

$$bf: \{0,1\}^n \rightarrow \{0,1\}$$

Sie bildet also aus einer Menge von Nullen und Einsen auf Null oder Eins ab.

Für eine bestimmte Funktion kann man eine Wahrheitstafel angeben, um sie zu beschreiben. Als Beispiel ist hier die UND-Funktion aufgeschrieben:

Wahrheitstabelle für die zweistellige UND-Funktion (∧)

a	b	a∧b
0	0	0
0	1	0
1	0	0
1	1	1

Man erkennt, dass eine solche Wahrheitstafel 2^n Zeilen besitzt (die Spaltenköpfe nicht mitgerechnet) wenn wir eine n-stellige Funktion beschreiben. Wir haben also n Eingangswerte. Wir benötigen für jede mögliche Kombination der Nullen und Einsen eine Zeile. Mit diesem Wissen können wir unsere Funktion eindeutig mithilfe des Ergebnisvektors (im Beispiel: **a∧b)** beschreiben. Wenn man diesen Vektor als String (oder Wort) festhält, hat er eine Länge von genau 2^n. Die Eingabematrix kann automatisch aus dieser Information erzeugt werden und die einzelnen Zeichen aus dem Ergebnisstring werden den entsprechenden Zeilen zugeordnet. Die Reihenfolge spielt hierbei natürlich eine wichtige Rolle.

Sei unser Wort x eine solcher Ergebnisstring mit maximaler Kolmogoroff-Komplexität:

$$\mathbf{K(x) \geq 2^n}$$

Es reicht, wenn wir hier nur die absolute K-Komplexität betrachten. Da sich der String nicht komprimieren lässt, hilft uns auch eine Eingabe (z.B.: seine Länge) nicht weiter.

Die Frage ist nun, wie komplex der dazugehörige Schaltkreis, mit dessen Hilfe man die Funktion umsetzen kann, ist. Wir definieren erst mal die Schaltkreiskomplexität:

Schaltkreiskomplexität
Sei die Schaltkreiskomplexität sk(bf) die kleinstmögliche Anzahl von Gattern, die nötig sind, um die boolesche Funktion bf zu berechnen.

Was verstehen wir nun unter einem Gatter?

Gatter
Wir betrachten binäre boolesche Gatter, d.h. wir haben zwei Eingänge und einen Ausgang. Das Gatter beschreibt also eine kleinste boolesche Funktion von $\{0,1\}^2$ zu $\{0,1\}$. Die Gatterbeschreibung erfolgt mithilfe seines Typs (z.B. ein UND- oder ODER-Gatter) und von welchen Eingängen oder anderen Gatter die Eingaben kommen.

Schaltkreis
Wir beschreiben einen Schaltkreis durch die Auflistung all seiner Gatter. Sei **g** die Anzahl der Gatter und **n** die Anzahl der Eingänge. Sei **c** eine Konstante für die Anzahl der verschiedenen Gattertypen. Wenn wir also 2 Gattertypen haben ist unser **c** gerade ein Bit um diese unterscheiden zu können (0 oder 1). Das erfordert pro Gatter also folgende Anzahl von Bits:

$$\mathbf{|g_i| = c + 2 \bullet \log (g + n)}$$

Im Folgenden wollen wir nun beweisen, dass für die Schaltkreiskomplexität sk(bf) gilt:

$$sk(bf) = \Omega(2^n / n)$$

Beweis:
Unseren gesamten Schaltkreis können wir mit

$$g \bullet (c + 2 \bullet \log (g + n))$$ // g ist Anzahl der Gatter

Bits beschreiben.

Es gilt also:

$$2^n \leq K(x) \leq g \bullet (c + 2 \bullet \log (g + n))$$ // x ist Ergebnisvektor mit $K(x) \geq 2^n$

Es ist allgemein bekannt das man mit $O(2^n)$ Gattern jede n-stellige boolesche Funktion berechnen kann, deshalb gilt:

$$\log(g + n) = O(n)$$ // $g = O(2^n)$
// $\log(2^n + n) \leq \log(2^{n+1}) = O(n)$

Damit wollen wir weiterrechnen:

$$2^n \leq g \bullet (c + 2 \bullet \log (g + n))$$
$$2^n \leq g \bullet (c + 2 \bullet O(n))$$
$$2^n / (c + 2 \bullet O(n)) \leq g$$
$$g \geq 2^n / (c + 2 \bullet O(n))$$

$$g \geq 2^n / (O(n))$$ // da c und 2 konstant

Es folgt also:
$$g = \Omega(2^n / n)$$

Da g die Anzahl der Gatter ist, die den Schaltkreis beschreibt, beträgt die Schaltkreis-komplexität dieser Funktion bf:

$$sk(bf) = \Omega(2^n / n)$$

qed.

Sprachen und Laufzeit von Turingmaschinen

Im Folgenden wollen wir zeigen, dass eine 1-Band-Turingmaschine mindestens eine Laufzeit von $\Omega(n^2)$ für folgende Sprache benötigt:

$L = \{ w0^{|w|}w \mid w \in \{0,1\}^* \}$

In ähnlicher Art funktioniert der Beweis auch für:

$L = \{ ww \mid w \in \{0,1\}^* \}$

$L = \{ ww^R \mid w \in \{0,1\}^* \}$

Dazu benötigen wir den Begriff der CROSSING SEQUENZ ($CS_M(x,i)$). Hier sei M eine Turingmaschine, x die Eingabe auf dem Band und i eine Position zwischen zwei Feldern des Bandes. $CS_M(x,i)$ ist dann die Folge der Zustände die während der Rechnung durchlaufen werden, wenn sich der Kopf über i hinwegbewegt.

Nun müssen wir uns einige wichtige Merkmale dieser Crossingsequenzen (CS) anschauen.

1.) $\Sigma_i|CS_M(x,i)| = time_M(x)$

Dies ist klar, den jeder Zustand der in einer Crossingsequenz auftaucht bedeutet einen Rechenschritt, umgekehrt ist jeder Rechenschritt Teil (genau) einer Crossingsequenz.

Um eine Laufzeit von $\Omega(n^2)$ zu zeigen, genügt es also für $\Omega(n)$ viele Crossingsequenzen zu zeigen, dass sie $\Omega(n)$ lang sind. Bei uns werden dies die n/3 Crossingsequenzen im Mittelteil des Wortes sein.

2.) oBdA stoppe unsere TM am Beginn der Eingabe; sei $i=|x|$:

$CS_M(xy,i) = CS_M(xz,i) \mapsto xy \in T(M)$ gdw. $xz \in T(M)$

Stimmen beide Crossingsequenzen bei i überein, so verhält sich die TM links von i stets absolut identisch, und da unsere TM links von i am Beginn der Eingabe stoppt ist $xy \in T(M)$ gdw. $xz \in T(M)$.

Mit ähnlichen Argumenten lässt sich folgendes zeigen:

3.) Wenn $c = CS_M(xy,i) = CS_M(x'y',i)$, dann auch $c = CS_M(xy',i) = CS_M(x'y,i)$.

Sei nun M eine TM, die L akzeptiert und sei $|w| \leq i \leq 2|w|$.

Aufgrund von 3.) gilt für zwei verschiedene Wörter $x=x_1x_2$ und $y=y_1y_2$ aus der Sprache L: $CS_M(x,i) \neq CS_M(y,i)$, da ansonsten auch x_1y_2 und y_1x_2 in der Sprache sein müssten. Hieraus folgt, das sich also das Wort $w0^{|w|}w$ aus $|w|$, i und c rekonstruieren lässt (Ein Algorithmus müsste via brute force alle Wörter der Länge $|w|$ bestimmen, daraus $w0^{|w|}w$ bilden und schließlich die Crossingsequenz dieses Wortes an der Stelle i mit c vergleichen).

Es folgt also: $K(x) \leq O(\log |x|) + |c|$.

Wir wählen uns ein w mit max. Kolmogoroff-Komplexität, also ein x mit $K(x) \geq |x|$:

$|w| \leq K(w) \leq O(\log |w|) + |c|$

$|w| \leq O(\log |w|) + |c|$

$|c| \geq |w| - O(\log |w|)$

Nun der eigentliche Beweis:

$$time(x) = \sum_{i=-\infty}^{\infty} |CS(x,i)| \geq \sum_{i=\frac{n}{3}}^{\frac{2}{3}n-1} |CS(x,i)| \geq \sum_{i=\frac{n}{3}}^{\frac{2}{3}n-1} \left(\frac{n}{3} - O(\log(n)) \right) \geq \frac{n^2}{9} - O(n \cdot \log(n)) = \Omega\left(n^2\right)$$

qed.