Kolmogorov – Komplexität

Im Rahmen des Hauptseminars Datenkompression

Zusammenfassung

In dieser Ausarbeitung wird eine Einführung in die Kolmogorov - Komplexität gegeben, die ein Maß für die Komprimierbarkeit von Informationen liefert. Es werden einige ihrer wichtigen Eigenschaften sowie Anwendungen vorgestellt.

Sommersemester 2002

Jörg Rüdenauer

1. Einleitung

Mit der Entropie, wie sie von Shannon entwickelt wurde, ist ein Maß für die Komprimierbarkeit von Informationen vorhanden. Die Entropie hat aber zwei große Nachteile: Zum Einen achtet sie nicht auf die Semantik von Nachrichten, sondern nur auf deren Komprimierbarkeit zum Zwecke der Datenübertragung. Sie bezieht sich daher immer auf die Menge, in der sich ein Objekt befindet; falls die Menge nur zwei Elemente hat, lässt sich ein Objekt durch ein Bit codieren. Man hätte aber gerne ein Maß, das im Objekt inhärent ist. Zum Anderen gibt die Entropie nur ein Maß für die durchschnittliche Komprimierbarkeit der Objekte in der Menge, kein Maß für die individuelle Komprimierbarkeit eines spezifischen Objekts.
Die sog. Kolmogorov - Komplexität löst diese beiden Probleme, indem sie ein Komprimierbarkeits-Maß angibt, das nur vom zu komprimierenden Objekt abhängt. Sie wurde (relativ unabhängig voneinander) von den Mathematikern Kolmogorov und Solomonoff entwickelt; auch mit dem Ziel, Zufälligkeit besser definieren zu können.
Im zweiten Abschnitt dieser Ausarbeitung wird die Kolmogorov – Komplexität vorgestellt. Der dritte Abschnitt beschäftigt sich dann mit einigen ihrer wichtigen Eigenschaften. Im vierten Abschnitt werden einige Anwendungen der Kolmogorov – Komplexität vorgestellt; diese liegen z.B. in der Datenkompression oder der Beweisführung.

2. Informations-Komplexität

Sei Σ ein Alphabet, und $\Sigma_0 = \Sigma \setminus \{*\}$. Man kann Σ_0 mit der Menge {0, 1, ..., m-1} identifizieren. Man betrachte nun eine universelle Turing-Maschine T mit zwei Bändern über Σ. Man sagt, dass das Wort q (aus Σ^*) das Wort x *schreibt*, falls T bei der Eingabe von q auf dem zweiten Band und einem leeren ersten Band in endlich vielen Schritten stoppt und dann x auf dem ersten Band steht. Man kann dann also auch x als die Ausgabe des Programms q bezeichnen.
Es ist klar, dass jedes Wort von T geschrieben werden kann. Es gibt z.B. eine einfache Turing-Maschine S_x, die x auf das leere Band schreibt und hält. S_x kann von einem Programm q_x auf T simuliert werden, wodurch x geschrieben wird.
Die *Informations-Komplexität* oder auch *Kolmogorov - Komplexität* eines Wortes x aus Σ_0^* ist die Länge des kürzesten Wortes, das T veranlasst, x zu schreiben. Sie wird mit $K_T(x)$ bezeichnet. Lose formuliert ist die Kolmogorov - Komplexität die Länge des kürzesten Programms, das x ausgibt. Falls bei der Berechnung von x eine weitere Information y als bekannt vorausgesetzt werden kann, so wird die *bedingte Kolmogorov - Komplexität* mit $K_T(x \mid y)$ bezeichnet. Es gilt also $K_T(x) = K_T(x \mid \varepsilon)$.
Man kann das Programm, das x schreibt, auch als Code von x ansehen, wobei die Turing-Maschine T die Decodierung vornimmt. Diese Art von Code wird als *Kolmogorov - Code* bezeichnet. Vorerst machen wir keine Annahmen über die Dauer des Decodierens (oder Codierens).

Nun ist es offensichtlich unschön, dass die Komplexität noch von der Turing-Maschine T abhängt; man hätte gerne, dass sie eine reine Eigenschaft des Wortes x sei. Man könnte sich z.B. eine Turing-Maschine denken, die nur jeden zweiten

Buchstaben der Eingabe verarbeitet und die anderen Buchstaben überspringt. Obwohl eine solche Turing-Maschine universell wäre, würde doch die Komplexität eines Wortes bei ihrer Verwendung doppelt so hoch sein wie ohne dieses Verhalten.
Im Folgenden wird gezeigt, dass es nicht mehr wichtig ist, welche Turing-Maschine für die Definition der Kolmogorov - Komplexität verwendet wird, falls T einige einfache Bedingungen erfüllt. Grob gesagt, muss man nur annehmen, dass jede Eingabe einer Berechnung auf T auch als Teil des Programms angegeben werden kann. Genauer nimmt man an, dass es ein Wort – nennen wir es $DATA$ – gibt, für welches folgende Bedingungen zutreffen:

(a) Jede einbändige Turing-Maschine kann durch ein Programm simuliert werden, das nicht $DATA$ als Teilwort enthält
(b) Wenn die Maschine so gestartet wird, dass das zweite Band ein Wort der Form $xDATAy$ enthält (wobei x nicht das Teilwort $DATA$ enthält), dann hält die Maschine genau dann an, wenn sie auch anhalten würde, falls man sie mit y auf dem ersten Band und x auf dem zweiten Band gestartet hätte, und liefert in beiden Fällen die gleiche Ausgabe auf dem ersten Band.

Es ist klar, dass jede universelle Turing-Maschine so modifiziert werden kann, dass sie die Annahmen (a) und (b) erfüllt. Im Folgenden wird angenommen, dass die Maschine T diese Eigenschaft hat.

Lemma 2.1: Es gibt eine Konstante c_T (die nur von T abhängt), so dass gilt: $K_T(x) \leq |x| + c_T$.

Beweis: Da T universell ist, kann die triviale einbändige Turing-Maschine, die sofort stoppt, auf ihr durch ein Programm p_0 simuliert werden (das nicht das Wort $DATA$ enthält). Mit Annahme (b) bedeutet das, dass das Programm $p_0 DATAx$ für jedes Wort x aus Σ^* x schreibt und anhält. Also wird die Annahme von der Konstante $c_T = |p_0| + 4$ erfüllt. ■

Es folgt der Beweis, dass die Komplexität nicht zu sehr von der benutzten Maschine abhängt (unter den obigen Voraussetzungen).

Theorem 2.2 (Invarianz - Theorem): Seien T und S universelle Turing-Maschinen, die die Bedingungen (a) und (b) erfüllen. Dann gibt es eine Konstante c_{TS}, so dass für jedes Wort x gilt: $|K_T(x) - K_S(x)| \leq c_{TS}$.

Beweis: Wir können die zweibändige Turing-Maschine S durch eine einbändige Turing-Maschine S_0 simulieren, so dass, falls ein Programm q auf S das Wort x schreibt, S_0 mit q auf dem Band ebenfalls in endlichen vielen Schritten anhält und dann x auf dem Band steht. Weiterhin können wir S_0 auf T durch ein Programm p_{S_0} simulieren, das nicht das Teilwort $DATA$ enthält.

Sei nun x ein beliebiges Wort aus Σ_0^* und q_x das kürzeste Programm, das x auf S schreibt. Man betrachte das Programm $p_{S_0} DATAx$ auf T: es schreibt offensichtlich x und hat als Länge $|q_x| + |p_{S_0}| + 4$. Die Ungleichheit in der anderen Richtung wird entsprechend bewiesen. ■

Wegen diesem Theorem wird die Allgemeinheit nicht verletzt, wenn man T als fest betrachtet und den Index weglässt, also nur noch $K(x)$ schreibt.

3. Eigenschaften der Kolmogorov - Komplexität

Zunächst einige einfache Eigenschaften:

(a) $K(x) \le |x| + c$
(b) $|\{x : K(x) = k\}| \le m^k$ (wobei m die Größe des Alphabets ist)
(c) Für jede berechenbare Funktion f gibt es eine Konstante c, so dass $K(f(x)) \le K(x) + c$ (falls x im Definitionsbereich von f ist)
(d) Sei V_n eine endliche aufzählbare Menge mit höchstens 2^n Elementen. Dann gibt es eine Konstante c, so dass alle Elemente von V_n eine Komplexität von höchstens $n + c$ haben (für alle $n \in \mathrm{N}$).
(e) $K(xy) \le K(x) + K(y) + O(\log(\min(K(x), K(y))))$

Beweis: zu (a): Dies wurde schon bei Lemma 2.1 bewiesen.
Zu (b): Es gibt genau m^k verschiedene Programme der Länge k. Damit es mehr als m^k Worte mit $K(x) = k$ gäbe, müsste mindestens eins dieser Programme mehr als ein Wort erzeugen, was unmöglich ist.
Zu (c): Sei T die Turing-Maschine, die für die Definition von K benutzt wurde. Man vergleiche T mit einer (universellen) Turing-Maschine S, die die Funktion f auf das Ergebnis von T anwendet – diese kann ebenfalls zur Definition einer Kolmogorov - Komplexität verwendet werden. Dann gilt:
$K_T(f(x)) \le K_S(f(x)) + c \le K_T(x) + c$, da jeder Kolmogorov - Code von x bezüglich T als Kolmogorov - Code von $f(x)$ bezüglich S betrachtet werden kann.
Zu (d): Man kann ein Wort $x \in V_n$ durch seine Position in der Aufzählung der Menge codieren; diese Zahl braucht nicht mehr als n Zeichen.
Zu (e): Es ist klar, dass das Problem nur im Trennen der Eingabe besteht, also im Erkennen des Endes von x. Dies kann man lösen, indem man z.B. die Länge des kürzeren Wortes codiert voranstellt, wie in $0^{|x|}1xy$. ■

Weiterhin betrachten wir einige algorithmische Eigenschaften von K:

Theorem 3.1: Die Funktion K ist nicht berechenbar.

Beweis: Der Beweis des Theorems erfolgt indirekt. Sein Prinzip ist das des klassischen logischen Paradoxons „Sei n die kleinste Zahl, die nicht mit weniger als 20 Worten definiert werden kann." (sie wurde gerade mit 15 Worten definiert).
Nehmen wir also an, dass K berechenbar sei. Sei dann c eine natürliche Zahl. Man ordne die Worte in Σ_0^* lexikographisch und $x(k)$ sei das k-te Wort in dieser Ordnung. Sei x_0 das erste Wort mit $K(x_0) \geq c$. Nun betrachten wir das folgende Programm in Pascal-ähnlicher Notation:

```
var k: integer;
function x(k: integer): integer;
.
.
.
function Kolm(k: integer): integer;
.
.
.
begin
  k := 0;
  while Kolm(k) < c do k := k + 1;
  print(x(k));
end.
```

Die beiden Funktionen x und Kolm wurden hier nicht explizit aufgeführt. Bei x wäre dies möglich gewesen, sie berechnet $x(k)$. Die Funktion Kolm berechnet $K(x(k))$und ist hypothetisch; unter der Annahme, dass K berechenbar ist, kann sie aber mit einer konstanten Anzahl Zeichen notiert werden.
Offensichtlich gibt das Programm x_0 aus, ist also ein Kolmogorov - Code für x_0. Da beide Subroutinen nur eine konstante Anzahl Zeichen haben (die auch nicht von c abhängt), hat das gesamte Programm eine Länge von $\log(c)+O(1)$. Somit ist $K(x_0) \leq \log(c)+O(1)$, was für genügend große c ein Widerspruch ist. ■

Die Unberechenbarkeit von K ist natürlich schade; jedoch kann man zeigen, dass die Komplexität K *im Durchschnitt* sehr gut abgeschätzt werden kann. Dafür muss man erst definieren, was „im Durchschnitt" bedeutet. Nehmen wir an, dass die Eingabeworte zufällig erzeugt werden; das bedeutet, dass jedes Wort x eine gewisse Wahrscheinlichkeit $p(x)$ hat, wobei $p(x) \geq 0$ und $\sum_{x \in \Sigma_0^*} p(x) = 1$. Wir nehmen an, dass $p(x)$ berechenbar ist; genauer gesagt, dass $p(x)$ eine rationale Zahl ist, deren Zähler und Nenner aus x berechnet werden können. Ein einfaches Beispiel einer Wahrscheinlichkeitsverteilung, die die Bedingungen erfüllt, wäre $p(x_k) = 2^{-k}$ mit x_k als dem k-ten Wort in lexikographischer Ordnung.

Theorem 3.2: Für jede berechenbare Wahrscheinlichkeitsverteilung gibt es einen Algorithmus, der einen Kolmogorov - Code $f(x)$ für jedes Wort x berechnet, so dass der Erwartungswert von $|f(x)| - K(x)$ endlich ist.

Beweis: Für die Einfachheit der Darstellung nehmen wir an, dass $p(x) > 0$ für alle x ist. Sei (x_i) eine Ordnung der Worte in Σ_0^*, für die $\forall i : p(x_i) \geq p(x_{i+1})$, und die Worte mit gleicher Wahrscheinlichkeit seien in aufsteigender Ordnung. Für den weiteren Beweis wird zunächst noch ein kleines Lemma benötigt:

Lemma 3.3: (a) Für jedes Wort x ist der Index i mit $x_i = x$ berechenbar.
(b) Für jede natürliche Zahl i ist x_i berechenbar.

Beweis: (a) Sei (y_i) die lexikographisch aufsteigende Ordnung aller Worte. Für ein bestimmtes Wort x ist es einfach, den Index j zu finden, für den $y_j = x$ ist. Danach suche man das erste $k \geq j$ mit $\sum_{l=1}^{k} y_l > 1 - p(y_j)$. Da die linke Seite der Ungleichung auf 1 zuläuft, die rechte Seite aber kleiner als 1 ist, wird k schließlich auf jeden Fall gefunden.
Es ist klar, dass jedes der Worte y_{k+1}, y_{k+2}, ... eine Wahrscheinlichkeit von weniger als $p(y_j)$ hat, und daher muss man nur noch die endliche Menge $\{y_1, \ldots, y_k\}$ gemäß absteigender Wahrscheinlichkeit sortieren und den Index von y_j in dieser Menge finden. Dies ist dann auch der Index von x.
(b) Mit einem gegebenen Index i kann man die Indizes von y_1, y_2 ... gemäß (a) berechnen, bis i auftaucht. ■

Zurück zum Beweis von Theorem 3.2: Der Algorithmus im Beweis des obigen Lemmas, zusammen mit der Zahl i, liefert einen Kolmogorov - Code $f(x_i)$ für jedes Wort x_i. Es wird nun gezeigt, dass dieser Code die Anforderungen des Theorems erfüllt. Natürlich ist $|f(x)| \geq K(x)$. Weiterhin ist der Erwartungswert von $|f(x)| - K(x)$

$\sum_{i=1}^{\infty} p(x_i)(|f(x_i)| - K(x_i))$.
Gezeigt werden soll, dass diese Summe endlich ist. Da die einzelnen Terme nicht negativ sind, reicht es zu zeigen, dass die Teilsummen beschränkt sind, also dass gilt
$\sum_{i=1}^{N} p(x_i)(|f(x_i)| - K(x_i)) < C$ für ein C unabhängig von N. Dafür schreibt man

$$\sum_{i=1}^{N} p(x_i)(|f(x_i)| - K(x_i)) = \sum_{i=1}^{N} p(x_i)(|f(x_i)| - \log_m i) + \sum_{i=1}^{N} p(x_i)(\log_m i - K(x_i)).$$

Beide Teilsummen bleiben beschränkt. Die Differenz $|f(x_i)| - \log_m i$ ist einfach die Länge des Programms, das x_i berechnet, ohne die Länge des Parameters i, somit ist sie eine Konstante und die erste Teilsumme ist höchstens C.
Um die zweite Teilsumme abzuschätzen, benutzt man das folgende einfache, aber nützliche Prinzip. Sei $a_1 \geq a_2 \geq \ldots \geq a_m$ eine abfallende Folge und b_1, b_2, ..., b_m eine

beliebige Folge von reellen Zahlen. Sei $b_1^* \geq b_2^* \geq \ldots \geq b_m^*$ die Folge b absteigend und sei $b_1^{**} \leq b_2^{**} \leq \ldots \leq b_m^{**}$ sie aufsteigend geordnet. Dann ist $\sum_i a_i b_i^{**} \leq \sum_i a_i b_i \leq \sum_i a_i b_i^*$.

Sei (z_i) eine Ordnung aller Worte, so dass $K(z_1) \leq K(z_2) \leq \ldots$ (diese Ordnung kann nicht berechnet werden, aber das ist auch nicht nötig). Dann ist wegen dem obigen Prinzip $\sum_{i=1}^{N} p(x_i)K(x_i) \geq \sum_{i=1}^{N} p(x_i)K(z_i)$. Wie oben gezeigt, ist die Anzahl der Worte mit $K(x)=k$ höchstens m^k, und daher ist die Anzahl der Worte x mit $K(x) \leq k$ höchstens $1+m+\ldots+m^k < m^{k+1}$. Substituiert man $K(z_i)$ für k, so erhält man wegen der Ordnung von (z_i) die Aussage $i \leq m^{K(z_i)+1}$ und somit $K(z_i) \geq \log_m i - 1$. Eingesetzt:

$$\sum_{i=1}^{N} p(x_i)(\log_m i - K(x_i)) \leq \sum_{i=1}^{N} p(x_i)(\log_m i - K(z_i)) \leq \sum_{i=1}^{N} p(x_i) = 1.$$

Damit ist Theorem 3.2 bewiesen. ■

4. Anwendungen der Kolmogorov - Komplexität

4.1 Der Begriff einer Zufallssequenz

In diesem Abschnitt wird angenommen, dass $\Sigma_0 = \{0,1\}$ ist; man betrachtet also nur die Komplexität von Binärstrings. Grob gesagt, soll eine Sequenz zufällig genannt werden, wenn sie keine „Regelmäßigkeit“ enthält. Um so allgemein wie möglich zu bleiben, betrachtet man jede Art Regelmäßigkeit, die eine bessere Codierung der Sequenz ermöglichen würde (so dass die Komplexität der Sequenz klein wäre).
Zuerst wird die Komplexität von „durchschnittlichen“ Binärstrings abgeschätzt:

Lemma 4.1: Die Zahl der Binärstrings x der Länge n mit $K(x) \leq n-k$ ist kleiner als 2^{n-k+1}.

Beweis: Die Anzahl der „Codeworte“ der Länge $n-k$ ist höchstens 1 + 2 + ... + 2^{n-k} < 2^{n-k+1}, also können nur weniger als 2^{n-k+1} Strings x solch einen Code haben. ■

Korollar 4.2: Die Komplexität von 99% der n-stelligen Binärstrings ist größer als $n-7$. Wenn man einen Binärstring der Länge n zufällig erzeugt, dann gilt $|K(x)-n| \leq 100$ mit einer Wahrscheinlichkeit von $1 - 2^{100}$.

Diese Zahlen folgen aus Lemma 4.1 durch einfaches Einsetzen; z.B. ist die Zahl der Binärstrings mit Komplexität ≤ n – 7 kleiner als 2^{n-6}, die Anzahl von Binärstrings der Länge n aber 2^n; $\frac{2^{n-6}}{2^n} = 2^{-6} = \frac{1}{64} = 0{,}016 = 1{,}6\%$.
Korollar 4.2 besagt umgangssprachlich, dass es sehr unwahrscheinlich ist, dass man einen zufällig erzeugten Binärstring gut komprimieren kann.

Eine weitere Folgerung aus Lemma 4.1 ist, dass es in gewisser Weise ein „Gegenbeispiel“ zur These von Church liefert. Man betrachtete das folgende Problem: Konstruiere für gegebenes n einen Binärstring der Länge n, dessen Kolmogorov - Komplexität größer ist als $n/2$. Dieses Problem ist wegen Theorem 3.1 (K ist nicht berechenbar) algorithmisch nicht lösbar. Das obige Lemma zeigt aber, dass mit großer Wahrscheinlichkeit ein zufällig gewählter String die Anforderung erfüllt.

Theorem 3.1 besagt, dass es unmöglich ist, den besten Code für ein Wort algorithmisch zu finden. Es gibt aber einige einfach erkennbare Eigenschaften, die von einem Wort zeigen, dass es kürzer als mit seiner Länge codiert werden kann. Das nächste Lemma zeigt solch eine Eigenschaft:

Lemma 4.3: Wenn die Anzahl von 1en in einem Binärstring x der Länge n k ist, dann gilt

$$K(x) \le \log_2 \binom{n}{k} + \log_2 n + \log_2 k + O(1).$$

Sei $k = pn$ $(0 < p < 1)$, dann kann dies als

$$K(x) \le (-p \log p - (1-p)\log(1-p))n + O(\log n)$$

abgeschätzt werden. Insbesondere gilt für $k > (1/2 + \varepsilon)n$ oder $k < (1/2 - \varepsilon)n$

$$K(x) \le cn + O(\log n),$$

wobei $c = -(1/2+\varepsilon)\log(1/2+\varepsilon) - (1/2-\varepsilon)\log(1/2-\varepsilon)$ eine positive Konstante ist, die kleiner als 1 ist und nur von ε abhängt.

Beweis: x kann als „lexikographisch t-te Sequenz unter allen Sequenzen, die die Länge n haben und genau k 1en besitzen“ beschrieben werden. Da die Anzahl der Sequenzen der Länge n mit genau k 1en $\binom{n}{k}$ ist, braucht die Beschreibung der Zahlen t, n und k nur $\log_2 \binom{n}{k} + 2\log_2 n + 2\log_2 k$ Bits. Der Faktor 2 wird benötigt, um die drei Ziffern voneinander zu trennen; man kann z.B. kn als $0^k 1\, k\, n$ codieren. Das Programm, das dann die richtige Sequenz auswählt, braucht nur eine konstante Zahl von Bits. Die Abschätzung des Binomialkoeffizienten geschieht mit einer Methode aus der Wahrscheinlichkeitstheorie. ■

Man kann nun entweder $|x| - K(x)$ oder $|x| / K(x)$ als Maß für die Zufälligkeit bzw. Nicht-Zufälligkeit eines Wortes x betrachten. Je größer diese Ausdrücke werden, desto kleiner ist $K(x)$ relativ zu $|x|$, was bedeutet, dass x „regelmäßiger“ und somit weniger zufällig ist.

Für unendliche Folgen kann man eine genauere Unterscheidung treffen: man kann definieren, *ob* eine bestimmte Sequenz zufällig ist. Dabei sind mehrere verschiedene Definitionen möglich; hier wird die einfachste vorgestellt.
Sei x ein unendlicher Binärstring, und x_n seine ersten n Zeichen. Wir nennen den String *informatisch zufällig*, wenn für $n \to \infty$ gilt $K(x_n)/n \to 1$.

Wenn man die Zahl der 1en im String x_n mit a_n bezeichnet, so geht aus Lemma 4.3 direkt das folgende Theorem hervor:

Theorem 4.4: Falls x informatisch zufällig ist, dann gilt (für $n \to \infty$) $a_n / n \to 1/2$.

Es stellt sich die Frage, ob die Definition einer algorithmisch zufälligen Sequenz nicht zu strikt ist, also ob es überhaupt unendliche Sequenzen gibt, die algorithmisch zufällig sind. Es wird nun gezeigt, dass es nicht nur solche Sequenzen gibt, sondern dass sogar fast alle Sequenzen diese Eigenschaft haben:

Theorem 4.5: Die Elemente eines unendlichen Binärstring x seien (unabhängig voneinander) mit Wahrscheinlichkeit 1/2 0en oder 1en. Dann ist x mit einer Wahrscheinlichkeit von 1 informatisch zufällig.

Beweis: Sei S für ein festes $\varepsilon > 0$ die Menge aller endlichen Sequenzen y, für die $K(y) < (1-\varepsilon)|y|$, und sei ferner A_n das Ereignis $x_n \in S$. Dann gilt wegen Lemma 4.1:
$Prob(A_n) \le \sum_{y \in S} 2^{-n} < 2^{(1-\varepsilon)n+1} 2^{-n} = 2^{1-\varepsilon n}$,

die Summe $\sum_{k=1}^{\infty} Prob(A_k)$ ist also konvergent. In diesem Fall besagt das Borel-Cantelli-Lemma aus der Wahrscheinlichkeitstheorie, dass mit einer Wahrscheinlichkeit von 1 nur endlich viele der Ereignisse A_k auftreten. Und dies wiederum heißt, dass für $n \to \infty$ $K(x_n)/n \to 1$ ist.

4.2 Datenkompression

Sei $L \subset \Sigma_0^*$ eine berechenbare Sprache und nehmen wir an, dass wir ein kurzes Programm, einen „Code“, nur für die Worte in L finden möchten. Für jedes Wort x in L suchen wir also ein Programm $f(x) \in \{0,1\}^*$, das x ausgibt. Man nennt die Funktion $f : L \to \Sigma^*$ *Kolmogorov - Code* von L. Die *Kürze* des Codes ist die Funktion $\eta(n) = \max\{|f(x)| : x \in L, |x| \le n\}$.

Man kann einfach eine untere Grenze der Kürze jedes Kolmogorov - Codes jeder Sprache bekommen. Sei L_n die Menge der Worte aus L, die höchstens die Länge n haben. Dann ist offensichtlich $\eta(n) \ge \log_2 |L_n|$. Diese Abschätzung nennt man die *informationstheoretische untere Grenze*.
Diese untere Grenze ist (bis auf eine additive Konstante) scharf. Man kann jedes Wort x in L codieren, indem man einfach seine Nummer in der aufsteigenden Ordnung angibt. Falls das Wort x der Länge n das t-te Element ist, dann braucht man dafür $\log_2 t \le \log_2 |L_n|$ Bits, plus eine konstante Anzahl Bits für das Programm, das die Elemente in Σ^* in lexikographische Ordnung bringt, überprüft, ob sie in L sind und das t-te Wort ausgibt.

Interessantere Fragen ergeben sich, wenn man fordert, dass der Code aus dem Wort und umgekehrt das Wort aus dem Code in polynomieller Zeit berechenbar sein soll. Mit anderen Worten: wir suchen eine Sprache L´ und zwei in polynomieller Zeit berechenbare Funktionen $f : \mathrm{L} \to \mathrm{L}'$, $g : \mathrm{L}' \to \mathrm{L}$ mit $g \circ f = \mathrm{id}_{\mathrm{L}}$, so dass für jedes x in L der Code $|f(x)|$ „kurz“ im Vergleich zu $|x|$ ist. Solch ein Paar von Funktionen wird *Polynomialzeit-Code* genannt.

Es folgt ein Beispiel für einen Polynomialzeit-Code, der nahe an der informatischen unteren Grenze ist:
Im Beweis von Lemma 4.3 wurde für die Codierung der Binärstrings der Länge n mit genau m Einsen als Code einer Sequenz seine Nummer in der lexikographischen Ordnung benutzt. Es wird gezeigt, dass diese Codierung polynomiell ist.
Man betrachte jeden Binärstring als (offensichtlichen) Code einer Teilmenge der n-elementigen Menge $\{n-1, n-2, ..., 0\}$. Jede solche Teilmenge kann als $\{a_1, ..., a_m\}$ mit $a_1 > a_2 > ... > a_n$ geschrieben werden. Dann kommt die Menge $\{b_1, ..., b_m\}$ genau dann lexikographisch vor der Menge $\{a_1, ..., a_m\}$, wenn es ein i gibt, so dass $b_i < a_i$ und für alle $j < i$ $a_j = b_j$ gilt. Sei $\{a_1, ..., a_m\}$ die lexikographisch t-te Teilmenge. Dann ist die Anzahl der Teilmengen $\{b_1, ..., b_m\}$ mit dieser Eigenschaft genau $\binom{a_i}{m-i+1}$. Wenn man dies über alle i aufsummiert, so erhält man $t = 1 + \sum_{i=1}^{m} \binom{a_i}{m-i+1}$.

Falls $a_1, ..., a_m$ gegeben sind, so kann t einfach in polynomieller Zeit berechnet werden. Umgekehrt ist t einfach in der obigen Form zu schreiben: zuerst sucht man (z.B. mit binärer Suche) die größte natürliche Zahl a_1 mit $\binom{a_1}{m} > t - 1$, dann die größte Zahl a_2 mit $\binom{a_2}{m-1} > t - 1 - \binom{a_1}{m}$, usw. für m Schritte. Es ist einfach zu sehen, dass dann $a_1 > a_2 > ... > a_m \geq 0$ gilt und nach m Schritten nichts „übrig bleibt“. Es kann also in polynomieller Zeit bestimmt werden, welche Teilmenge lexikographisch die t-te ist.

4.3 Die Unkomprimierbarkeits – Methode

Eine große Bedeutung hat die Kolmogorov - Komplexität im Bereich der Beweisführung mit Hilfe der Unkomprimierbarkeits - Methode erlangt. Ein Problem bei Beweisen ist häufig, dass man eine bestimmte Eigenschaft nur für ein spezielles Element einer Menge zeigen müsste, ein solches aber nicht einfach findet und daher den Beweis allgemein führen muss. Lemma 4.1 besagt nun, dass es auf jeden Fall Worte gibt, die nicht komprimierbar sind. Bei Anwendung der Unkomprimierbarkeits – Methode geht man normalerweise davon aus, dass die gewünschte Eigenschaft nicht gilt, und zeigt dann, dass in diesem Fall doch alle solchen Worte komprimierbar wären, was den gewünschten Widerspruch liefert. Es werden nun zwei Beispiele für die Anwendung der Methode gegeben.

Theorem 4.6: Für unendlich viele n ist die Anzahl der Primzahlen kleiner als n mindestens $\log n / \log \log n$.

Beweis: Sei n eine natürliche Zahl, die nicht mit weniger als $\log n$ Zeichen darstellbar ist. Laut Lemma 4.1 gibt es unendlich viele solcher Zahlen. Seien $p_1, p_2, ..., p_m$ die Primzahlen, die kleiner als n sind. Dann kann man $n = p_1^{e_1} p_2^{e_2} \cdots p_m^{e_m}$ aus dem Vektor der Exponenten rekonstruieren. Jeder Exponent ist höchstens $\log n$ groß und kann daher mit $\log\log n$ Bits dargestellt werden. Also kann n mit $m \log\log n$ Bits dargestellt werden, woraus Theorem 4.6 folgt. ■

Im folgenden Beispiel wird die Unkomprimierbarkeits – Methode angewendet, um die durchschnittliche Komplexität von Heapsort abzuschätzen.

Heapsort ist ein häufig genutztes Sortierverfahren, weil es ohne zusätzlichen Platz auskommt und einen Aufwand von garantiert höchstens $n \log n$ besitzt. Jedoch war lange Zeit seine durchschnittliche Komplexität nicht gut analysiert. Mit Hilfe der Unkomprimierbarkeit wird dies erstaunlich einfach.
Einen Heap kann man sich als (bis auf die unterste Ebene) vollständigen Binärbaum vorstellen. Jeder Knoten ist mit einem Schlüssel markiert. Der größte Schlüssel ist bei der Wurzel, und jeder andere Knoten hat einen Schlüssel, der nicht größer als der seines Vaterknotens ist. Formal: Ein Feld aus Schlüsseln $k_1, ..., k_n$ ist ein *Heap,* falls für $1 \le \lfloor j/2 \rfloor < j \le n$ gilt: $k_{\lfloor j/2 \rfloor} \ge k_j$. Wir betrachten das Sortieren innerhalb eines solchen Feldes A ohne zusätzlichen Speicher. Es läuft in zwei Schritten ab:
Aufbau des Heaps: Betrachte A als einen Baum: die Wurzel ist in $A[1]$; die zwei Söhne von $A[i]$ sind in $A[2i]$ und $A[2i+1]$. Wiederhole für $i = n/2, n/2-1, ..., 1$: {der Teilbaum mit Wurzel in $A[i]$ ist jetzt bis auf die Wurzel selbst schon ein Heap} Lasse den Schlüssel k von $A[i]$ in den Baum einsinken - vertausche ihn mit einem Sohn, der einen Schlüssel größer als k hat -, bis er keinen solchen Sohn mehr besitzt.
Sortieren des Heaps: Wiederhole für $i = n, n-1, ..., 2$: { $A[1..i]$ ist der verbleibende Heap und $A[i+1..n]$ enthält die schon sortierte Liste aus den größten Schlüsseln $k_{i+1}, ..., k_n$. Laut Definition muss $A[i]$ (die Wurzel des Heaps) den größten der verbleibenden Schlüssel, also k_i, enthalten.} Vertausche den Schlüssel von $A[1]$ mit dem Schlüssel von $A[i]$ und erweitere so die sortierte Liste zu $A[i..n]$. Forme $A[1..i-1]$ wieder zu einem Heap mit dem größten Schlüssel bei $A[1]$ um.
Es ist bekannt, dass der Aufbau des Heaps in $O(n)$ ablaufen kann. Es ist ebenfalls bekannt, dass der Sortierschritt nicht mehr als $O(n \log n)$ Zeit braucht. Es wird nun die genaue durchschnittliche Komplexität des Sortierschritts abgeschätzt wird. Dabei wird die Umformung im Sortierschritt nach **Williams Methode** vorgenommen:
{Am Anfang ist $A[1] = k$.}. Wiederhole: vergleiche die Schlüssel der beiden Söhne von k; falls m der größere ist, vergleiche k und m; wenn $k < m$ ist, dann vertausche k und m; bis $k \ge m$ ist. Williams Methode macht pro Anwendung $2d$ Vergleiche und d Datenverschiebungen.
Man nehme eine gleichförmige Wahrscheinlichkeitsverteilung über der Liste der n Schlüssel an, so dass alle Eingabelisten gleich wahrscheinlich sind.

Theorem 4.7: Im Durchschnitt macht Heapsort $n \log n - O(n)$ Datenverschiebungen. Williams Methode macht im Durchschnitt $2n \log n - O(n)$ Vergleiche.

Beweis: Mit n Schlüsseln gibt es $n! \approx n^n e^{-n} \sqrt{2\pi n}$ Permutationen. Wir wählen eine Permutation p der n Schlüssel, so dass $C(p \mid n) \geq n \log n - 2n$. Laut Theorem 4.1 gibt es eine solche Permutation; in der Tat haben sogar die meisten Permutationen diese Eigenschaft.

Behauptung 4.8: Sei h der nach dem ersten Schritt aufgebaute Heap mit der Eingabe p. Dann ist $C(h \mid n) \geq n \log n - 6n$.

Beweis: Man nehme im Gegenteil an, dass $C(h \mid n) < n \log n - 6n$. Dann wird gezeigt, wie man p unter Verwendung von h und n in weniger als $n \log n - 2n$ Bits beschrieben werden kann. Man codiert die Aufbauprozedur des Heaps, die h aus p erzeugt. In jeder Schleife wird, wenn $k = A[i]$ in den Heap einsinkt, der Pfad auf dem k läuft, folgendermaßen codiert: 0 für einen linken Zweig, 1 für einen rechten Zweig, 2 für Stopp. Insgesamt braucht man dafür $\log 3 * \sum_i n \frac{i}{2^{i+1}} \leq 2n \log 3$ Bits. Bei gegebenem Heap h und dieser Beschreibung der Pfade kann man die Prozedur umkehren und p rekonstruieren. Somit ist
$C(p \mid n) < C(h \mid n) + 2n \log 3 + O(1) < n \log n - 2n$;
dies ist ein Widerspruch. □

Jetzt wird eine Beschreibung von p gegeben, indem man die Historie der $n-1$ Heap - Umordnungen während des Sortierschrittes benutzt. Man muss nur für jeden Durchlauf im Sortierschritt die letzte Position notieren, an der $A[i]$ in den Heap eingefügt wird. Eine solche Position wird durch den Pfad von der Wurzel zu ihr codiert. Ein Pfad kann durch eine Sequenz von 0en und 1en repräsentiert werden, wobei 0 einen linken Zweig und 1 einen rechten Zweig angibt. Da die Länge mitcodiert werden muss, codiert man den Pfad als (a, l, s), wobei s der eigentliche Pfad ist, für $a = 0$ ist $l = |s|$, und für $a = 1$ ist $l = \log n - |s|$; l ist in selbstbegrenzender Form. Falls der i-te Pfad die Länge d_i hat, braucht diese Beschreibung höchstens $1 + d_i + \min\{2\log(\log n - d_i), 2 \log d_i\}$ Bits. All diese Pfadbeschreibungen werden zu einer Sequenz H zusammengefasst.

Behauptung 4.9: Aus H und n kann man den Heap h rekonstruieren.

Beweis: Seien H und die Tatsache, dass h ein Heap von n unterschiedlichen Schlüsseln ist, bekannt. Man simuliert den Sortierschritt rückwärts. Zuerst enthält $A[1..n]$ eine sortierte Liste mit dem kleinsten Element in $A[1]$.
Für $i = 2,...,n-1$ wiederhole: {jetzt enthält $A[1..i-1]$ den teilweise rekonstruierten Heap und $A[i..n]$ enthält die verbleibende sortierte Liste mit dem kleinsten Element in $A[i]$.} Setze den Schlüssel von $A[i]$ nach $A[1]$ und bewege jeden Schlüssel auf dem $(n-i)$-ten Pfad in H eine Position nach unten, wobei man mit der Wurzel in $A[1]$ anfängt. Der letzte Schlüssel auf dem Pfad wird nach $A[i]$ gesetzt.
Nach dem letzten Schleifendurchlauf enthält $A[1..n]$ den Heap h. □

Aus Behauptung 4.9 folgt, dass $C(h \mid n) \leq |H| + O(1)$ ist. Laut Behauptung 4.8 gilt also $|H| \geq n \log n - 6n$. Vergleicht man dies mit der oben angegebenen Länge der Beschreibung eines Pfades, so erkennt man, dass dies nur möglich ist, wenn die durchschnittliche Pfadlänge zumindest $\log n - c'$ für eine feste Konstante c' ist. Diese Zahl gibt die durchschnittliche Anzahl von Datenverschiebungen in jedem Durchlauf des Sortierschrittes an. Also macht Heapsort ausgehend vom Heap h zumindest $n \log n - O(n)$ Datenverschiebungen und mit Williams Methode $2n \log n - O(n)$ Vergleiche.
Da die meisten Permutationen zufällig sind, gelten diese Grenzen nicht nur für eine zufällige Permutation p, sondern *im Durchschnitt* für alle Permutationen. ■

5. Literatur

Der größte Teil dieser Ausarbeitung basiert auf einem Kapitel eines Skripts von L. Lovász [1]. M. Li und P. Vitányi haben ein großes Buch über die Kolmogorov - Komplexität geschrieben, das weit über den Inhalt dieser Ausarbeitung herausgeht [2]. Aus ihm habe ich hauptsächlich die Beispiele in Abschnitt 4.3 entnommen. Im WWW gibt es zudem noch eine Reihe weiterer Artikel, Seiten und Skripte, die einen Überblick über das Themengebiet geben [3] – [6].

Referenzen

[1] ‘Information complexity: the complexity-theoretic notion of randomness’, in: László Lovász, Complexity of algorithms, Skript, 1997-1999

[2] M.Li und P. Vitányi, ‘An Introduction to Kolmogorov Complexity and its Applications’, Springer, New York, 1997

[3] Alexander Shen, ‘Kolmogorov Complexity and its Applications’, http://www.csd.uu.se/~vorobyov/Courses/KC/2000/

[4] Nick Szabo, ‘Introduction to Algorithmic Information Theory’, http://szabo.best.vwh.net/kolmogorov.html

[5] Lance Fortnow, ‘Kolmogorov Complexity’, http://www.neci.nec.com/homepages/fortnow/papers/kaikoura.pdf

[6] A. Gammerman und Vladimir Vovk, ‘Kolmogorov Complexity: Sources, Theory and Applications’, The Computer Journal, Vol.42, No.4, 1999