3 Datenkomprimierung

In diesem Kapitel werden wir uns mit effizienten Verfahren zum verlustlosen Komprimieren von Daten und insbesondere Textdateien beschäftigen.

Definition 3.1 *Sei Σ ein Alphabet und $S \subseteq \Sigma^*$ die Menge der zulässigen Eingaben. Ein* verlustloser Code *ist ein Code (c, d) bestehend aus einem Kodierverfahren $c : S \to \{0,1\}^*$ und einem Dekodierverfahren $d : \{0,1\}^* \to S$, so dass für alle $x \in S$ gilt $d(c(x)) = x$. Die Menge $c(S)$ ist die Menge der* Codeworte *des Codes.*

Um die Bedingung $d(c(x)) = x$ für alle x zu erfüllen, muss c offensichtlich injectiv sein. Haben wir injectives Kodierverfahren c, ist es im Prinzip immer möglich, ein geeignetes Dekodierverfahren d anzugeben. Unser Ziel wird es sein, Kodierverfahren zu finden, die eine möglichst hohe Komprimierung erreichen und Kodier- und Dekodierverfahren zu finden, die möglichst schnell laufen.

3.1 Paradoxen der Datenkomprimierung

Zunächst zeigen wir, dass es unmöglich ist, einen Code zu finden, der jede mögliche Eingabe komprimieren kann.

Satz 3.2 *Sei $S = \{0,1\}^*$. Dann gibt es für jeden Code (c, d) und jedes $n \geq 1$ eine Eingabe $x \in S$ der Länge n mit $|c(x)| \geq n$.*

Beweis. Betrachte ein beliebiges $n \geq 1$. Sei $S_n = \{0,1\}^n$. Da c injektiv sein muss, muss jedes $x \in S_n$ einem anderen $c(x) \in \{0,1\}^*$ zugeordnet werden. Also ist $|c(S_n)| = 2^n$, und da mit bis zu $n-1$ Bits nur $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ verschiedene Codeworte erzeugt werden können, muss es ein Codewort $x \in S_n$ geben mit $|c(x)| \geq n$. □

Auf der anderen Seite gilt der folgende Satz, der besagt, dass die Encyclopedia Britannica im Prinzip auf ein Bit komprimiert werden kann.

Satz 3.3 *For jede Eingabe x gibt es einen Code (c, d), so dass $|c(x)| = 1$.*

Beweis. Weise einfach x das Codewort 0 und jeder anderen Eingabe $y \neq x$ das Codewort $1y$ zu. Das erzeugt offensichtlich ein injektives Kodierverfahren. □

Die beiden Sätze zeigen, dass man sich darauf konzentrieren sollte, Komprimierungsverfahren zu finden, die möglichst alle Eingaben möglichst gut komprimieren können. Wir werden im folgenden untersuchen, wie man das erreichen kann. Dabei werden wir uns auf statistische Codes beschränken. Das sind Codes, die entweder eine vorgegebene Häufigkeitsverteilung von Mustern verwenden oder die Häufigkeitsverteilung von Mustern in Eingaben analysieren, um eine mglichst gute Komprimierung zu erreichen.

3.2 Präfix-Codes

Der grundlegende Ansatz statistischer Kodierungstheorie ist wie folgt: finde eine geeignete Wahrscheinlichkeitsverteilung über S (das Datenmodell) und ein geeignetes Kodierungsverfahren c, das die erwartete Codelänge $\mathrm{E}[c(x)]$ unter dieser Wahrscheinlichkeitsverteilung minimiert.

Satz 3.2 besagt, dass die Komprimierung aller Eingaben in $S \in \{0,1\}^*$ unmöglich ist. Was gilt aber für die erwartete Codelänge bei einer uniformen Verteilung über S?

Das folgende Verfahren legt nahe, dass dann eine nichttriviale Komprimierung möglich ist. Sei $S = \{0,1\}^2$, $c(00) = \epsilon$, $c(01) = 0$, $c(10) = 1$ und $c(11) = 01$. Für ein uniform zufällig gewähltes $x \in S$ gilt dann $\mathrm{E}[|c(x)|] = 1$, und wir scheinen damit zwei Bits in ein Bit im Durchschnitt komprimiert zu haben!

Obwohl das gewissermaßen korrekt ist, ist diese Codierungsmethode nicht relevant für die Codierung längerer Worte. Wir können nicht einfach sagen, verwende den Code oben, um 4 Bits zu komprimieren (indem $c(xy) = c(x) \cdot x(y)$ ist), denn der resultierende Code wäre nicht injektiv (wir hätten $c(0011) = c(0110)$). Wir bräuchten in diesem Fall ein explizites Endsymbol, um die verschiedenen kodierten Zeichen voneinander zu trennen, wodurch sich der Erwartungswert von einem Bit auf mindestens zwei Bits je zwei Originalbits erhöhen würde. Die folgende Definition spezifiziert genau, was wir brauchen, damit wir eine Codierungstechnik für eine endliche Anzahl an Symbolen oder Worten auch für beliebige Worte anwenden können.

Definition 3.4 *Ein* Präfix-Code *c hat die Eigenschaft, dass für alle $x, y \in S$ mit $x \neq y$ gilt: $c(x)$ ist kein Präfix von $c(y)$.*

Damit folgt natürlich automatisch, dass $c(x) \neq c(y)$ für alle $x \neq y$ ist und damit jeder Präfix-Code für Eingaben in S eindeutig dekodierbar ist. Präfix-Codes haben weiterhin die folgende wichtige Eigenschaft, die sicherstellt, dass diese im Gegensatz zum Code oben skalierbar, also auf beliebige Worte in S^* anwendbar sind:

Satz 3.5 *Wenn c ein Präfix-Code über Σ ist, dann ist c^n mit der Regel*

$$c^n(x_1x_2\ldots x_n) = c(x_1) \cdot c(x_2) \cdot \ldots \cdot c(x_n)$$

ein Präfix-Code über Σ^n.

Beweis. Übungsaufgabe. □

Es gibt im wesentlichen zwei Möglichkeiten, Präfix-Codes zu repräsentieren: als Bäume und als Intervalle.

- Betrachte einen beliebigen Präfix-Code c und den Aho-Corasick Automaten im letzten Kapitel. Wenn wir diesen auf $c(S)$ anwenden, dann erhalten wir einen Baum von Zuständen, in dem die Blätter den Codeworten in $c(S)$ entsprechen. D.h. für einen Präfix-Code kann es nie einen inneren Knoten im AC-Automaten geben, der einem Codewort in $c(S)$ entspricht. Das ist wichtig für die eindeutige Unterteilung einer kodierten Eingabe in Codeworte.

- Eine alternative Interpretation ist die folgende: Für ein Codewort $c \in \{0,1\}^*$ definieren wir das *dyadische Intervall* $I_c \subseteq [0,1)$ als

$$I_c = [0.c, 0.c1111\ldots)$$

wobei $0.c$ und $0.c1111\ldots$ Binärdarstellungen von Zahlen in $[0,1)$ repräsentieren. Zum Beispiel ist $I_{101} = [0.101, 0.1011111\ldots) = [0.101, 0.110)$, oder in dezimaler Notation $[5/8, 3/4)$. In diesem Fall ist ein Präfix-Code eine Zuweisung von Eingaben zu einer Menge paarweise disjunkter Intervalle.

Für die zweite Interpretation gilt die folgende Aussage:

Lemma 3.6 *Ein Code ist ein Präfixcode genau dann, wenn die dyadischen Intervalle der Codewörter disjunkt sind.*

Beweis. Übungsaufgabe. □

Diese Aussage ist nützlich für den Beweis des folgenden Satzes.

Satz 3.7 (Kraft-McMillan Ungleichung)

1. *Seien $m_1, m_2, \ldots$ die Längen der Codewörter eines Präfix-Codes. Dann gilt $\sum_i 2^{-m_i} \leq 1$.*
2. *Falls auf der anderen Seite $m_1, m_2, \ldots$ natürliche Zahlen sind mit $\sum_i 2^{-m_i} \leq 1$, dann gibt es einen Präfix-Code c, so dass $m_1, m_2, \ldots$ die Längen der Codewörter sind.*

Beweis. Zunächst beweisen wir (1). Falls $m_1, m_2, \ldots$ die Längen der Codewörter eines Präfix-Codes sind, dann sind die Längen der entsprechenden dyadischen Intervalle gleich $2^{-m_1}, 2^{-m_2}, \ldots$ Da diese paarweise disjunkt und ein Teilinterval in $[0,1)$ sind, muss gelten, dass $\sum_i 2^{-m_i} \leq 1$.

Als nächstes beweisen wir (2). Um disjunkte Intervalle der Größe $2^{-m_1}, 2^{-m_2}, \ldots$ in $[0,1)$ zu finden, sortieren wir zunächst die Werte $m_1, m_2, \ldots$, so dass wir ohne Beschränkung der Allgemeinheit annehmen können, dass $m_1 \leq m_2 \leq \ldots$ gilt. Angefangen mit 0 packen wir nun die Intervalle so dicht wie möglich in das $[0,1)$-Intervall. Dazu weisen wir das Intervall der Länge 2^{-m_i} dem Platz $[\sum_{j<i} 2^{-m_j}, \sum_{j<i} 2^{-m_j} + 2^{-m_i})$ zu. Das ergibt offensichtlich disjunkte Intervalle, und da $\sum_i 2^{-m_i} \leq 1$ ist, passen alle Intervalle in das $[0,1)$-Intervall hinein. Für das Interval zu m_i gilt weiterhin, dass es sich in binärer Form als $[0.x, 0.x1111\ldots)$ darstellen lässt mit $|x| = m_i$, da jedes 2^{-m_j} nur $m_j \leq m_i$ Bits benötigt. Interpretiert man nun die Intervalle als dyadische Intervalle von Codewörtern, so erhält man nach Lemma 3.6 einen Präfix-Code c mit Codewörtern der Länge $m_1, m_2, \ldots$ □

3.3 Entropie

Die Entropie einer Wahrscheinlichkeitsverteilung ist wie folgt definiert:

Definition 3.8 *Sei X eine beliebige Zufallsvariable auf S mit Wahrscheinlichkeitsverteilung $p : S \to [0,1)$, d.h. $\sum_{x \in S} p(x) = 1$. Dann ist die* Entropie *von X definiert als*

$$\mathrm{H}[X] = \sum_{x \in S} p(x) \cdot \log_2(1/p(x))$$

Die folgenden Tatsachen sind leicht zu zeigen:

- Die Entropie der uniformen Wahrscheinlichkeitsverteilung auf $\{0,1\}^n$ ist n.

- Falls X_1 und X_2 unabhängige Zufallsvariablen sind, dann ist $\mathrm{H}[X_1, X_2] = \mathrm{H}[X_1] + \mathrm{H}[X_2]$, d.h. die Entropie des Tupels (X_1, X_2) ist gleich der Summe der Entropie von X_1 und X_2.
- Für eine beliebige Funktion f und Zufallsvariable X gilt $H(f(X)) \leq H(X)$, d.h. Entropie kann nicht deterministisch erzeugt werden.

Der folgende Satz verknüpft Präfix-Codes mit der Entropie und ist eine vereinfachte Version des berühmten Source Coding Theorems von Shannon.

Satz 3.9 *Sei S eine beliebige Menge und X eine Zufallsvariable auf S. Dann gilt:*

- *Für alle Präfix-Codes c,* $\mathrm{E}[|c(X)|] \geq \mathrm{H}[X]$.
- *Es gibt einen Präfix-Code c mit* $\mathrm{E}[|c(X)|] < \mathrm{H}[X] + 1$.

Beweis. Für (1) gilt:

$$\begin{aligned} \mathrm{H}[X] &\leq \mathrm{E}[|c(X)|] \\ \Leftrightarrow \quad \sum_{x \in S} p(x) \log(1/p(x)) &\leq \sum_{x \in S} p(x)|c(x)| \\ \Leftrightarrow \quad \sum_{x \in S} p(x)[\log(1/p(x)) - |c(x)|] &\leq 0 \\ \Leftrightarrow \quad \sum_{x \in S} p(x) \log(2^{-|c(x)|}/p(x)) &\leq 0 \end{aligned}$$

Weiterhin ist leicht zu sehen, dass die Funktion $f(x) = \log x$ konkav ist, d.h. für alle $x, y \in \mathrm{I\!R}^+$ gilt

$$f\left(\frac{x+y}{2}\right) \geq \frac{1}{2} f(x) + \frac{1}{2} f(y)$$

Mit dieser Eigenschaft ist es nicht schwer zu zeigen, dass für jede Folge von Werten $x_1, \ldots, x_k$ und jede Verteilung $p_1, \ldots, p_k$ mit $\sum_i p_i = 1$ gilt

$$f\left(\sum_i p_i x_i\right) \geq \sum_i p_i f(x_i)$$

Diese Ungleichung ist auch bekannt als Jensens Ungleichung. Angewandt auf unseren Fall erhalten wir

$$\sum_{x \in S} p(x) f(2^{-|c(x)|}/p(x)) \leq f\left(\sum_{x \in S} p(x) \cdot 2^{-|c(x)|}/p(x)\right) = \log\left(\sum_{x \in S} 2^{-|c(x)|}\right)$$

Da nach dem Kraft-McMillan Satz $\sum_{x \in S} 2^{-|c(x)|} \leq 1$ gilt, folgt die Behauptung (1) von Satz 3.9.

Als nächstes beweisen wir (2). Wenn wir jeder Wort $x \in S$ ein Codewort der Länge $\ell(x) = \lceil \log(1/p(x)) \rceil$ zuweisen, so gilt

$$\begin{aligned} \sum_{x \in S} 2^{-\ell(x)} &= \sum_{x \in S} 2^{-\lceil \log(1/p(x)) \rceil} \\ &\leq \sum_{x \in S} 2^{-\log(1/p(x))} \\ &= \sum_{x \in S} p(x) = 1 \end{aligned}$$

Nach dem Kraft-McMillan Satz gibt es also einen Präfix-Code c mit Codeworten der Länge ℓ. Für diesen Code gilt

$$\begin{aligned}
\mathrm{E}[|c(X)|] &= \sum_{x \in S} p(x) \cdot \ell(x) \\
&= \sum_{x \in S} p(x) \cdot \lceil \log(1/p(x)) \rceil \\
&< \sum_{x \in S} p(x) \cdot (1 + \log(1/p(x))) \\
&= 1 + \sum_{x \in S} p(x) \log(1/p(x)) = 1 + \mathrm{H}[X]
\end{aligned}$$

Damit ist auch Behauptung (2) gezeigt. □

3.4 Huffman Codes

Gegeben eine Wahrscheinlichkeitsverteilung p, wie können wir daraus einen optimalen Präfix-Code erzeugen? Eine Möglichkeit wäre, S in zwei Teilmengen S_0 und S_1 aufzuspalten, so dass $\sum_{x \in S_0} p(x)$ und $\sum_{x \in S_1} p(x)$ möglichst nah an $1/2$ sind. Die Codeworte von S_0 würden dann mit einer 0 starten und die Codeworte von S_1 mit einer 1, und wir würden rekursiv mit den Partitionierungen fortfahren, bis jedem $x \in S$ ein eindeutiger Code zugewiesen worden ist. Für einen solchen Präfix-Code c gilt allerdings nur $\mathrm{E}[|c(X)|] \leq \mathrm{H}[X] + 2$ im worst case. Wir brauchen also eine andere Strategie.

Wir präsentieren nun einen sehr einfachen Algorithmus zur Berechnung eines optimalen Präfix-Codes, der auch als Huffman Code bekannt ist. Der Trick des Algorithmus ist, den Code durch den Aufbau eines Baumes von unten nach oben zu erzeugen.

- Starte mit einer Menge isolierter Knoten, einem Knoten pro Element $x \in S$. Jeder Knoten $x \in S$ hat ein Gewicht von $p(x)$.
- Wiederhole, bis nur noch ein einziger Baum übrig ist:
  - Wähle zwei Bäume aus, deren Wurzeln r_1 und r_2 das niedrigste Gewicht haben.
  - Kombiniere die beiden Bäume in einen Baum, indem eine neue Wurzel r mit Gewicht $p(r) = p(r_1) + p(r_2)$ erzeugt wird und diese mit r_1 und r_2 verbunden wird. Dabei ist es unerheblich, welche der beiden Wurzeln r_1 und r_2 das linke oder das rechte Kind von r werden.

Eine einfache Priority-Queue reicht aus, um den Huffman-Baum nach obigem Verfahren zu erzeugen. Verwendet man als Priority-Queue einen Heap, dann ist die Laufzeit des Algorithmus beschränkt durch $O(n \log n)$, wobei $n = |S|$. Sobald wir die Elemente in S in einem Baum organisiert haben, können wir daraus sofort den Präfix-Code für die Elemente in S ablesen, indem wir jeder linken Kante eines inneren Knotens eine 0 und jeder rechten Kante eines inneren Knotens eine 1 zuweisen. Der nächste Satz zeigt, dass der Huffman Code optimal ist.

Satz 3.10 *Der Huffman Algorithmus erzeugt einen optimalen Präfix-Code.*

Beweis. Wir beweisen den Satz durch vollständige Induktion über die Anzahl der Elemente in S. Falls $|S| = 1$ ist, so besteht der Huffman-Baum nur aus einem einzelnen Knoten und der entsprechende Präfix-Code ist das leere Wort, was sicherlich optimal ist.

Angenommen, wir hätten schon gezeigt, dass für ein $n \geq 1$ der Huffman-Algorithmus einen optimalen Präfix-Code für alle Wahrscheinlichkeitsverteilungen über einer Menge S' mit $|S'| = n$ erzeugt. Betrachten wir nun eine beliebige Menge S der Größe $n + 1$, und sei p eine beliebige Wahrscheinlichkeitsverteilung über dessen Elemente. Seien x und y die Elemente mit der niedrigsten Wahrscheinlichkeit in S. Betrachte nun die Menge S', die aus S hervorgeht, indem wir x und y durch ein neues Element z mit $p(z) = p(x) + p(y)$ ersetzen. Sei T' der Baum, den der Huffman-Algorithmus für S' erzeugt, und sei d die Tiefe von z in diesem Baum. Hängen wir nun Knoten für x und y unter z an, so erhalten wir einen Baum für S. Wenn $\ell(t)$ die erwartete Codelänge für einen Baum t ist, so erhalten wir

$$\ell(T) = \ell(T') + p(x)(d + 1) + p(y)(d + 1) - p(z) \cdot d = \ell(T') + p(x) + p(y)$$

Nach der Konstruktion des Huffman-Algorithmus wäre T der Baum, den dieser erzeugt. Betrachte nun einen optimalen Baum T_{opt} für S. Falls x und y in diesem Baum Geschwister sind, dann können wir einfach x und y wegnehmen und deren Vater z zuweisen, um einen Baum T'_{opt} für S' zu erhalten. Das vermindert die erwartete Codelänge für T_{opt} um $p(x) + p(y)$, und da $\ell(T') \leq \ell(T'_{opt})$ sein muss, folgt, dass $\ell(T) \leq \ell(T_{opt})$ und somit $\ell(T) = \ell(T_{opt})$ ist.

Es bleibt, den Fall zu betrachten, dass x und y keine Geschwister in T_{opt} sind. Sei $d(v)$ die Tiefe eines Knotens v in T_{opt}. Ohne Beschränkung der Allgemeinheit nehmen wir an, dass $d(x) \geq d(y)$ ist. Sei x' das Geschisterteil von x. Falls $d(x) > d(y)$, so muss $p(x') \leq p(y)$ sein, da wir sonst durch Austausch von x' und y einen Baum mit kleinerem ℓ hinkriegen könnten. Weiterhin kann nicht $p(x') < p(y)$ gelten, da wir angenommen haben, x und y haben das minimale Gewicht. Daher kann für $d(x) > d(y)$ nur $p(x') = p(y)$ sein, so dass wir x' mit y vertauschen können, ohne ℓ zu verändern. Falls $d(x) = d(y)$ gilt, führt dieser Austausch auch nicht zu einer Veränderung von ℓ. Wir können also immer einen optimalen Baum finden, in dem x und y Geschwister sind, was den Beweis abschließt. □

Huffman Codes werden üblicherweise auf Eingabealphabete angewandt, d.h. S repräsentiert die Menge aller Zeichen, die eine Eingabe enthalten kann. Gegeben ein Huffman Code für ein Eingabealphabet S, der sowohl der Kodierungs- als auch der Dekodierungsseite bekannt ist (wie das z.B. für den Faxstandard der Fall ist), so ist alles, was die Kodierungsseite für einen Eingabetext t tun muss, die Zeichen in t systematisch in Huffman-Codes umzuwandeln. Ist der Huffman Code nicht vorgegeben, so kann das folgende Verfahren angewandt werden:

- **Komprimierung:** Zunächst wird der Text t einmal durchlaufen, um die Häufigkeiten der einzelnen Elemente zu ermitteln. Mittels dieser Häufigkeiten wird dann der Huffman Code c erzeugt und t mit diesem Code kodiert. Die komprimierte Form von t ist dann $c \circ c(t)$, d.h. der Huffman Code selbst (der für die Dekomprimierung gebraucht wird) gefolgt vom komprimierten Text $c(t)$.

- **Dekomprimierung:** Zunächst wird c gelesen, dann $d = c^{-1}$ berechnet, und schliesslich $c(t)$ mittels d in t zurückverwandelt.

Bei sehr großen Texten oder kontinuierlichen Datenströmen ist es zu aufwänding oder unmöglich, den Text zweimal zu durchlaufen (einmal zur Ermittlung der Häufigkeiten und einmal für die Komprimierung). Dann kann man z.B. Häufigkeiten über gewisse Intervalle zählen und den Huffman-Baum nach jedem Intervall anpassen, wobei die Intervallgröße so groß sein sollte, dass die Konstruktion des Huffman-Baums vernachlässigende Kosten verursacht.

3.5 Arithmetische Codes

Huffman-Codes haben den Nachteil, dass sie ein Bit pro Zeichen gegenüber einer optimalen Kodierung verlieren können. Im Fall, dass wir Texte $t \in \Sigma^*$ haben, in dem ein bestimmtes Zeichen x einen Anteil von $0,999$ ausmacht, muss man mit dem Huffman-Code z.B. mindestens $|t|$ viele Bits aufwenden, um t zu kodieren. Hier sollten viel bessere Methoden möglich sein. Eine solche Methode sind die arithmetischen Codes.

Arithmetische Codes basieren auf dem Shannon-Fano Code, der wie folgt funktioniert. Sei S eine beliebige Eingabemenge mit einer totalen Ordnung und X eine Zufallsvariable auf S, und sei das dyadische Intervall I_x von $x \in S$ das längste dyadische Intervall (d.h. der Form $[0.z, 0.z1111\ldots)$ für ein $z \in \{0,1\}^*$) innerhalb von

$$[\Pr[X < x], \Pr[X \leq x])$$

Da $([\Pr[X < x], \Pr[X \leq x]))_{x \in S}$ eine Partition von $[0,1)$ ist, sind damit die Intervalle I_x paarweise disjunkt, d.h. der Code c definiert durch die dyadischen Intervalle I_x ist ein Präfix-Code. Für diesen Code gilt die folgende Eigenschaft.

Satz 3.11 *Der Shannon-Fano Code c zu S hat die Eigenschaft, dass $\mathrm{E}[|c(X)|] < \mathrm{H}[X] + 2$ ist.*

Beweis. Sei $I = [\Pr[X < x], \Pr[X \leq x])$ für ein beliebiges $x \in S$. Dann ist $|I| = \Pr[X = x]$. Für $k = \lceil \log(1/|I|) \rceil + 1$ gilt damit, dass $2^{-k} \leq |I|/2$. Also muss es in der ersten Hälfte von I ein ganzzahliges Vielfaches von 2^{-k} geben, sagen wir $m \cdot 2^{-k}$. Das Intervall $[m2^{-k}, (m+1)2^{-k})$ ist damit ein möglicher Kandidat für I_x, und da I_x das längste dyadische Intervall in I ist, gilt $|I_x| \geq 2^{-k}$. x kann damit einem Präfix-Code der Länge höchstens

$$\lceil \log(1/\Pr[X = x]) \rceil + 1 \leq \log(1/\Pr[X = x]) + 2$$

zugewiesen werden. D.h. der Shannon-Fano Code c erfüllt

$$\begin{aligned} \mathrm{E}[|c(X)|] &= \sum_{x \in S} \Pr[X = x] \cdot |c(x)| \\ &\leq \sum_{x \in S} \Pr[X = x](\log(1/\Pr[X = x]) + 2) \\ &= \mathrm{H}[X] + 2 \end{aligned}$$

□

Als einfache Regel für das Codewort einer Eingabe $x \in S$ definieren wir $c(x)$ als die Binärdarstellung von $\lceil \Pr[X < x]/2^{-k} \rceil$, wobei $k = \lceil \log(1/\Pr[X = x]) \rceil + 1$ ist. Das ergibt einen Code der Länge k für $c(x)$, was ausreicht, um Satz 3.11 zu erfüllen.

```
Algorithmus ArithEncode I:
        // t: Text der Länge n auf Σ
        // p: Wahrscheinlichkeitsverteilung auf Σ
        low := 0    // untere Grenze vom aktuellen Intervall
        range := 1    // Länge des aktuellen Intervalls
        FOR i := 1 TO n DO
            low := low + (∑_{k<t[i]} p(k)) · range
            range := p(t[i]) · range
        k := ⌈log(1/range)⌉ + 1
        c := ⌈low/2^{-k}⌉
        return c
```

Figure 1: Die Berechnung des arithmetischen Codes mit unbegrenzter Präzision.

```
Algorithmus ArithDecode I:
        // c: Arithmetischer Code eines Textes der Länge n
        // p: Wahrscheinlichkeitsverteilung auf Σ
        low := 0    // untere Grenze vom aktuellen Intervall
        range := 1    // Länge des aktuellen Intervalls
        FOR i := 1 TO n DO
            x := eindeutiges Symbol in Σ, so dass
                    c ∈ [low + (∑_{k<x} p(k)) · range, low + (∑_{k≤x} p(k)) · range)
            t[i] := x
            low := low + (∑_{k<x} p(k)) · range
            range := p(x) · range
        return t
```

Figure 2: Die Dekodierung eines arithmetischen Codes mit unbegrenzter Präzision.

Statische arithmetische Kodierung

Sei Σ ein beliebiges Eingabealphabet und p eine Wahrscheinlichkeitsverteilung auf Σ. Können wir mit unbegrenzter Präzision rechnen, so können wir die Algorithmen in Abb. 1 und 2 verwenden, um einen Text in einen arithmetischen Code umzuwandeln oder aus einem arithmetischen Code einen Text zu gewinnen.

Das Problem mit diesen Algorithmen ist allerdings, dass eine unbegrenzte Präzision sehr teuer, wenn nicht unmöglich ist. Arbeitet man statt dessen mit gerundeten Zahlen, so kann es zu numerischen Problemen kommen. Es kann z.B. passieren, dass wir für einen Text t ein Intervall der Länge 0 bekommen, womit alle Texte mit Präfix t denselben Code hätten. Um diese Probleme zu verhindern, brauchen wir eine geeignete Arithmetik, die nur eine endliche Präzision erfordert. Die Tricks dafür sind überwiegend durch Patente geschützt, was erklärt, warum arithmetische Codes trotz ihrer hervorragenden Eigenschaften keine große Verbreitung gefunden haben. Im folgenden zählen wir einige dieser Tricks auf. Sei $high = low + range$ die obere Grenze des Intervalls.

- Jedesmal, wenn die Zahlen in $[low, high)$ dasselbe höchste Bit haben, hängen wir dieses Bit an das Codewort c an und entfernen das höchste Bit aus low und $range$, d.h. wir multiplizieren low und $range$ mit 2 und nehmen beide Werte modulo 1. Das nennen wir im folgenden auch eine *Standardskalierung*.

  Haben wir z.B. die Zahlen $low = 0.101011$ und $range = 0.001$, d.h. $high = 0.110011$, dann hängen wir eine 1 an c an und setzen $low := 0.01011$ und $range := 0.01$.

- Angenommen, wir verwenden eine m-Bit Arithmetik (d.h. es können beliebige Vielfache von 2^{-m} im Bereich $[0, 1)$ dargestellt werden). Falls wir die Werte in p so runden, dass alle Werte $p(x)$ Vielfache von 2^{-m+k} sind (d.h. die niedrigsten k Bits sind 0), so können wir jedes $p(x)$ mit den höchsten k Bits in $range$ multiplizieren, ohne runden zu müssen. Damit wir so ein neues Intervall mit $range > 0$ bekommen, muss sichergestellt werden, dass zu jedem Zeitpunkt $range \geq 2^{-k}$ ist. Das kann natürlich problematisch sein, falls low sehr nah bei $1/2$ liegt, da wir dann nicht die Standardskalierung oben verwenden können. Als Ausweg gibt es eine spezielle Skalierung, die wir im folgenden vorstellen.

- Unser Ziel wird es sein, die Invariante $range \geq 1/4$ zu erhalten. Falls das verletzt ist und wir keine Standardskalierung anwenden können, dann ist $range < 1/4$ und $1/4 < low < 1/2$. In diesem Fall verwenden wir eine spezielle Skalierung, indem wir $low := 2(low - 1/4)$ und $range := 2range$ setzen.

  Wir fahren dann mit dem Kodierungsalgorithmus fort, bis wir die Standardskalierung anwenden können oder die Invariante wieder verletzt ist. Können wir die Standardskalierung anwenden und die spezielle Skalierung ist seitdem s-mal verwendet worden, dann unterscheiden wir zwischen den folgenden Fällen:

  - Ist das höchste Bit in low gleich 0, dann hängen wir 01^s an das Codewort c an.
  - Ist das höchste Bit in low gleich 1, dann hängen wir 10^s an das Codewort c an.

- Schließlich müssen wir noch angeben, was zu tun ist, wenn der Algorithmus alle Zeichen der Eingabe gelesen hat. Da $range > 1/4$ ist, ist eine der Zahlen mit Nachkommastellen in $N = \{01, 10, 11\}$ in $[low, high)$. Wir unterscheiden zwischen den folgenden Fällen (s ist die Anzahl spezieller Skalierungen nach der letzten Standardskalierung):

  - $s = 0$: hänge ein beliebiges $a \in N$ mit $0.a \in [low, high)$ an c an.
  - $s > 0$ und $0.01 \in [low, high)$: hänge 01^s1 an c an.
  - $s > 0$ und $0.1b \in [low, high)$ für ein $b \in \{0, 1\}$: hänge 10^sb an c an.

Um die Auswirkungen der speziellen Skalierung zu verstehen, brauchen wir das folgende Lemma.

Lemma 3.12 *Die spezielle Skalierung erhöht* $|low - 1/2|$ *und* $|high - 1/2|$ *um den Faktor 2.*

Beweis. Sei $low = 1/2 - \delta_1$ und $high = 1/2 + \delta_2$. Dann gilt für die Ergebnisse low' und $high'$ der speziellen Skalierung:

$$low' = 2((1/2 - \delta_1) - 1/4) = 1/2 - 2\delta_1$$

und

$$high' = 2((1/2 + \delta_2) - 1/4) = 1/2 + 2\delta_2$$

was die Behauptung beweist. □

Dieses Lemma ermöglicht den Beweis des folgenden zentralen Lemmas.

Lemma 3.13 *Nach jeder Standardskalierung entspricht das relative* $[0,1)$*-Intervall dem dyadischen Intervall* $[0.c, 0.c1111\ldots)$ *für das bisher ausgegebene Codewort c, und nach* $k \geq 1$ *speziellen Skalierungen entspricht das relative* $[0,1)$*-Intervall dem absoluten Intervall* $[0.c01^k, 0.c10^k1111\ldots)$.

Beweis. Das Lemma kann durch vollständige Induktion über die Anzahl s der Skalierungen bewiesen werden. Am Anfang, d.h. $c = \epsilon$, ist das Lemma trivialerweise wahr. Nehmen wir also an, dass es für s Skalierungen schon bewiesen worden ist. Dann betrachten wir die $s+1$-te Skalierung. Wir unterscheiden zwischen 4 Fällen:

1. Die s-te Skalierung ist eine Standardskalierung und die $s+1$-te auch. Sei b das Bit, das an das Codewort c angehängt wird. Dann entspricht das neue relative Intervall dem Intervall $[0.b, 0.b1111\ldots)$ des vorherigen relativen Intervalls. Das absolute Intervall verändert sich also von $[0.c, 0.c1111\ldots)$ zu $[0.cb, 0.cb1111\ldots)$. Da b an c angehängt wird, beweist dieser Fall die Induktionsbehauptung.

2. Die s-te Skalierung ist eine Standardskalierung und die $s+1$-te eine spezielle Skalierung. Dann sind die Grenzen des neuen relativen Intervalls nach Lemma 3.12 nur noch halb so weit von der Mitte $0.c1$ entfernt wie die Grenzen des alten relativen Intervalls. Das absolute Intervall verändert sich also von $[0.c, 0.c1111\ldots)$ zu $[0.c01, 0.c101111\ldots)$.

3. Die s-te Skalierung ist eine spezielle Skalierung und die $s+1$-te auch. Dann sind die Grenzen des neuen relativen Intervalls nach Lemma 3.12 nur noch halb so weit von der Mitte $0.c1$ entfernt wie die Grenzen des alten relativen Intervalls. Das absolute Intervall verändert sich also von $[0.c01^k, 0.c10^k1111\ldots)$ für ein $k \geq 1$ zu $[0.c01^{k+1}, 0.c10^{k+1}1111\ldots)$.

4. Die s-te Skalierung ist eine spezielle Skalierung und die $s+1$-te eine Standardskalierung. Sei b das Bit, das an das Codewort c angehängt wird. Dann entspricht das neue relative Intervall dem Intervall $[0.b, 0.b1111\ldots)$ des vorherigen relativen Intervalls. Das absolute Intervall verändert sich also von $[0.c01^k, 0.c10^k1111\ldots)$ für ein $k \geq 1$ zu $[0.c01^k, 0.c01^k1111\ldots)$ falls $b = 0$ oder $[0.c10^k, 0.c10^k1111\ldots)$ falls $b = 1$. Da 01^k für $b = 0$ und 10^k für $b = 1$ an c angehängt wird, beweist dieser Fall die Induktionsbehauptung.

In allen Fällen ist die Induktionsbehauptung also erfüllt, was den Beweis abschließt. □

Die hinzugefügten Bits am Ende der Kodierung entsprechen dem Fall, dass low auf eine Zahl mit Nachkommastellen in $[low, high)$ gesetzt wird und $range$ auf 0 gesetzt wird, so dass am Ende ein c ausgegeben wird, das im letztendlichen Intertall liegt. Die Kodierungsmethode mit begrenzter Präzision ist also korrekt. Der Algorithmus dazu findet sich in Abb. 3. Die Dekodierung mit begrenzter Präzision verläuft ähnlich zum Algorithmus in Abb. 2 und ist eine Übungsaufgabe.

```
Algorithmus ArithEncode II:
    // t: Text der Länge n auf Σ
    // p: Wverteilung auf Σ, kodiert als Vielfache von 2^(−m+k)
    low := 0    // untere Grenze vom aktuellen Intervall
    range := 0    // Länge des aktuellen Intervalls ist 2^m = 0 (mod 2^m)
    special := 0    // Zähler für spezielle Skalierungen
    j := 1    // Position in Codewort c
    FOR i := 1 TO n DO
        IF range = 0 THEN norm = 2^k ELSE norm = range ≫ m − k
        low := low + ((∑_{j<t[i]} p(j)) ≫ k) · norm
        range := (p(t[i]) ≫ k) · norm
        WHILE (range > 0 AND range < 2^(m−2)) DO
            IF low ≥ 2^(m−1) OR low + range ≤ 2^(m−1) THEN
                b := low ≫ m − 1    // gib Bits aus
                c[j] := b
                j := j + 1
                WHILE special > 0 DO
                    c[j] := 1 − b
                    j := j + 1
                    special := special − 1
                low := low ≪ 1                // Standardskalierung
                range := range ≪ 1
            ELSE
                low := (low − 2^(m−2)) ≪ 1    // spezielle Skalierung
                range := range ≪ 1
                special := special + 1
    low := low + range − 1    // setze low auf high-1
    b := low ≫ m − 1    // extrahiere 1. Bit
    c[j] := b
    j := j + 1
    WHILE special > 0 DO
        c[j] := 1 − b
        j := j + 1
        special := special − 1
    b := (low ≫ m − 2) − 2 · b    // extrahiere 2. Bit
    c[j] := b
    return c
```

Abbildung 3: Die Berechnung des arithmetischen Codes mit m-Bit Präzision. “$\gg$” ist der Rechtsshift-Operator und “$\ll$” ist der Linksshift-Operator.

Dynamische arithmetische Kodierung

Bisher haben wir angenommen, dass die Wahrscheinlichkeitsverteilung p vorgegeben ist. Wenn das nicht so ist, gibt es eine relativ einfache Möglichkeit, die Wahrscheinlichkeitsverteilung während des

Textdurchlaufs anzupassen.

Wir starten zunächst mit der Gleichverteilung, d.h. für 2^z Zeichen und eine Bitauflösung von 2^m ordnen wir jedem Zeichen x den Wert $h(x) = 2^{m-z}$ zu, was der Wahrscheinlichkeit 2^{-z} entspricht. Weiterhin unterteilen wir den Text in Blöcke der Größe $B = 2^b - 2^z$ für ein geeignet großes b. Zu Anfang eines jeden Blockes setzen wir $num(x) = 1$ für jedes $x \in \Sigma$. Jedesmal, wenn wir ein Zeichen x lesen, erhöhen wir $num(x)$ um 1. Am Ende des Blocks setzen wir dann $h(x) = (1-\alpha)h(x) + \alpha 2^{m-b} num(x)$ für ein festes $0 < \alpha < 1$, wobei α die Aggressivität der Anpassung bestimmt. Je größer α, desto schneller ist die Anpassung.

Arithmetische Kodierung mit Kontext

Bisher haben wir nur Verfahren kennengelernt, die ohne Kontext komprimieren. Viel bessere Komprimierungsraten sind aber erreichbar, wenn man mit Kontext komprimiert. Angenommen, das letzte gelesene Zeichen war ein "q". Dann ist es sehr wahrscheinlich, als nächstes Zeichen ein "u" zu lesen. Man könnte nun natürlich für jeden möglichen Kontext aus k Zeichen die Wahrscheinlichkeitsverteilung für das nachfolgende Zeichen aufstellen. Das wäre allerdings sehr speicherintensiv und daher nur für kleine k machbar. Ein viel besseres Verfahren dafür ist das PPM (Prediction by Partial Matching) Verfahren und seine zahlreichen Varianten. Der PPM Algorithmus baut während des Textdurchlaufs ein Wörterbuch in Form eines Tries auf. Anfangs sind dort nur alle Zeichen des Alphabets Σ enthalten, und die Häufigkeit dieser Zeichen wird zu 1 initialisiert, was einer Gleichverteilung entspricht. Danach wird der Trie durchlaufen wie im Aho-Corasick Automaten. Falls für ein zuletzt gelesenes Wort w im Trie und die aktuelle Position i im Text t gilt, dass wt_i auch im Trie ist, wird der t_i-Übergang benutzt. Ansonsten wird ein neuer Übergang für t_i angelegt und der Fehlerübergang benutzt zum Wort w', das einem längsten Suffix in w entspricht. Das wird fortgesetzt, bis ein t_i-Übergang gefunden wurde, was spätestens dann gilt, wenn $w' = \epsilon$ ist.

Bei der arithmetischen Kodierung wird der Fehlerübergang wie ein spezielles Zeichen gewertet. Jedesmal, wenn ein Zeichen benutzt wird, wird dessen Häufigkeit um 1 erhöht und die arithmetische Kodierung mit dem Aufruf Arith(h, z) aufgerufen, wobei h eine Häufigkeitsverteilung ist und z das aktuelle Zeichen bzw. die Position in h darstellt. Genauer gesagt ruft Arith() den inneren Teil der FOR-Schleife in Abb. 3 mit $t[i] = z$ und $p(j) = h(j)/(\sum_x h(x))$ auf.

Der Kodierungsalgorithmus ist in Abb. 4 dargestellt. Der er im Prinzip eine online-Version des AC-Automaten ist, ist seine Laufzeit (abgesehen von den Arith-Aufrufen) $O(n)$. Es bleibt, die Laufzeit der Arith-Aufrufe zu beschränken. Hier ist das Problem, dass $\sum_{j<i} h(j)$ berechnet werden muss, was $O(|\Sigma|)$ Zeit kosten kann. Um eine bessere Laufzeit zu erreichen, müssen wir die $h(j)$-Werte in einen Baum organisieren.

Betrachte den vollständigen binären Baum T mit 2^s Blättern, wobei $s = \lceil \log |\Sigma| \rceil$. Wir fordern die Aufrechterhaltung der Invariante, dass zu jeder Zeit Blatt i die Häufigkeit von $h(i)$ speichert und jeder innere Knoten die Summe der Häufigkeiten aller Blätter unter ihm speichert. Dann gilt für die Wurzel v, dass $h(v) = \sum_{i=1}^{\Sigma} h(i)$ ist. Sei nun $Inc(i)$ die Operation, die die Häufigkeit von $h(i)$ unter Beachtung der Invariante um 1 erhöht und $Sum(i)$ die Operation, die den Wert $\sum_{j<i} h(i)$ berechnet. Dann gilt:

Lemma 3.14 *Im Häufigkeitsbaum mit 2^s Blättern benötigt jede $Inc(i)$-Operation und $Sum(i)$-Operation nur $O(s)$ Zeit.*

Beweis. Für jede $Inc(i)$ Operationen müssen nur die Häufigkeiten entlang des eindeutigen Pfades von Blatt i zur Wurzel angepasst werden, um die Invariante oben zu erhalten. Für jede $Sum(i)$-Operation reicht es, den eindeutigen Pfad von der Wurzel bis nach Blatt i zu durchlaufen und dabei die Häufigkeiten der linken Kinder zu addieren. □

```
Algorithmus PPM-Encode:
    // t: Text der Länge n auf Σ
    // m: maximale Größe des AC-Automaten
    // d[q][c]: für Übergäng von q bei Zeichen c ≥ 1
    // d[q][0]: Fehlerfunktion für Zustand q ∈ {0, ..., m}
    // h[q][c]: Häufigkeit von Zeichen c bei Zustand q
    q[0][0] := −1
    FOR c := 1 TO |Σ| DO h[q][c] := 1    // Gleichverteilung
    FOR q := 0 TO m DO
        h[q][0] := 0
        FOR c := 0 TO |Σ| DO d[q][c] := 0
    qnew := 1; q := 0
    FOR i := 1 TO n DO
        WHILE d[q][t[i]] = 0 DO
            d[q][t[i]] = qnew    // erzeuge neuen Zustand
            qnew := qnew + 1
            q′ := d[q][t[i]]    // setze Fehlerübergang für q′
            r := d[q][0]
            IF r ≠ −1 THEN
                IF d[r][t[i]] = 0 THEN d[q′][0] := qnew ELSE d[q′][0] := d[r][t[i]]
                h[q][0] := h[q][0] + 1    // benutze Fehlerübergang
                Arith(h[q], 0)
                q := r
        h[q][t[i]] := h[q][t[i]] + 1    // benutze t[i]-Übergang
        Arith(h[q], t[i])
        q := d[q][t[i]]
```

Abbildung 4: Der PPM Kodierungsalgorithmus.

Alternativ kann man auch einen Häufigkeitsbaum als dynamischen Huffman Baum verwalten. Dann ist der Aufwand für $Inc(i)$ und $Sum(i)$ gleich der momentanen Codelänge für i, so dass man insgesamt den folgenden Satz erhalten kann.

Satz 3.15 *Der PMM Kodierungsalgorithmus kann so implementiert werden, dass er in Zeit $O(n)$ läuft.*

Die Dekodierung ahmt einfach die Kodierung nach, um aus dem Code den ursprünglichen Text zurückzugewinnen. Das ist möglich, da von Anfang an jedes Zeichen in Σ eine positive Häufigkeit hat, so dass es immer legale Übergänge im AC-Automaten für jedes gelesene Zeichen gibt.

3.6 Lempel-Ziv Codes

Zum Schluss stellen wir noch zwei Varianten der bekannten Lempel-Ziv Komprimierungsverfahren vor. Um deren Güte zu analysieren, verallgemeinern wir die Entropie ohne Kontext auf die Entropie mit Kontext k.

Definition 3.16 *Sei t ein beliebiger Text über dem Alphabet Σ und sei n_w für ein beliebiges Wort w die Anzahl der Vorkommen von w in t. Dann ist die* Entropie k-ter Ordnung *von t definiert als*

$$H_k(t) = \sum_{w \in \Sigma^k} \frac{n_w}{|t|} \left(\sum_{z \in \Sigma} \frac{n_{wz}}{n_w} \log \left(\frac{n_w}{n_{wz}} \right) \right)$$

Dieser Entropiebegriff hat die folgende Eigenschaft.

Satz 3.17 *Für jede Eingabe t und jedes $k \geq 0$ gilt für jeden Präfix-Code c für einen Kontext der Länge k, dass $|c(t)| \geq |t| \cdot H_k(t)$ ist.*

Beweis. Analog zu Satz 3.9. □

Die Frage ist also, wie nah man an $|t|H_k(t)$ herankommen kann. Dazu verwenden wir die folgende Definition.

Definition 3.18 *Ein Präfix-Code c heißt* grob optimal*, falls für alle $k \geq 0$ und alle Eingaben t gilt, dass*

$$|c(t)| \leq |t| \cdot H_k(t) + o(|t|)$$

Ein grob optimaler Code ist also dann besonders gut, wenn die Entropie k-ter Ordnung hoch ist. Wenn sie allerdings $o(1)$ ist, kann dieser Code weit weg von $|t| \cdot H_k(t)$ sein. Um die grobe Optimalität eines Codes nachzuweisen, werden wir als zentrales Hilfsmittel die folgende Aussage benutzen, die wir hier nicht beweisen werden, da der Beweis recht komplex ist.

Lemma 3.19 *Seien $w_1, \ldots, w_m$ eine beliebige Zerlegung eines Textes t, so dass jedes Wort w_i höchstens q-mal vorkommt. Dann gilt für jedes $k \geq 0$:*

$$m \log m \leq |t| \cdot H_k(t) + m \log \left(\frac{|t|}{m} \right) + m \log q + \Theta((k+1)m)$$

Im folgenden bedeutet w^- für ein Wort w das Wort w ohne den letzten Buchstaben, und $(w^-)^-$ kürzen wir ab zu w^{--}.

LZ77 Code

Zunächst stellen wir den LZ77-Code vor. Dieser Code arbeitet wie folgt. Angenommen, der Text t sei bereits bis zu einer gewissen Position j in die Worte $w_1, \ldots, w_{i-1}$ zerlegt worden. Dann wählt LZ77 als das nächste Wort w_i ab Position j das längste Wort, das dadurch erhalten werden kann, dass man ein einzelnes Zeichen an ein Teilwort von $(w_1 w_2 \ldots w_i)^{--}$ dranhängt. D.h. w_i hat die Eigenschaft, dass w_i^- ein Teilwort von $(w_1 w_2 \ldots w_i)^{--}$ ist, aber nicht w_i.

Diese Definition ist eindeutig, auch wenn sie auf den ersten Blick verwirrend klingt. Betrachten wir einmal das Wort $t = abaabbaabbaaa$. Dann ist die Zerlegung von t gegeben durch $w_1 = a$, $w_2 = b$, $w_3 = aa$, $w_4 = bb$ und $w_5 = aabbaaa$.

Sobald w_i gefunden worden ist, hängt LZ77 das Tripel (p_i, ℓ_i, z_i) an das Codewort c an, wobei p_i die Anfangsposition von w_i^- in $(w_1 \dots w_i)^{--}$ ist, $\ell_i = |w_i|$ ist, und z_i das letzte Zeichen von w_i ist.

Der LZ77 Code kann effizient mit Hilfe eines Suffix-Baum Algorithmus implementiert werden. Solch ein Algorithmus kann für ein Eingabewort t einen Suffix-Baum von t in Zeit $O(|t|)$ aufbauen, so dass für jedes Suchwort s in Zeit $O(|s|)$ festgestellt werden kann, ob s ein Teilwort in t ist. Aus Zeitgründen stellen wir diesen Algorithmus nicht vor, aber eine gute Referenz ist z.B. das Buch "Algorithms on Strings, Trees, and Sequences" von Dan Gusfield. Der LZ77 Code hat die folgende Eigenschaft.

Satz 3.20 *Der LZ77 Code ist grob optimal.*

Beweis. Zunächst berechnen wir, wieviele Bits wir für ein Tripel (p_i, ℓ_i, z_i) benötigen. Da $1 \leq p_i \leq \sum_{j<i} \ell_j$ ist, reichen $\lceil \log(\sum_{j<i} \ell_j) \rceil$ Bits zur Kodierung von p_i. Weiterhin benögtigen wir $\lceil \log s \rceil$ Bits für z_i unter der Annahme, dass $|\Sigma| = s$ ist. A priori ist die Größe von ℓ_i zwar nicht bekannt, aber eine einfache Kodierung hilft, diese anzugeben:

Zunächst wird $1^k 0$ ausgegeben für das kleinste k, so dass $|\ell_i| \leq 2^k$ ist. Darauf folgt die genaue Kodierung der Länge von ℓ_i, was weitere $\lceil \log |\ell_i| \rceil$ Bits benötigt, und danach die Kodierung von ℓ_i selbst. Insgesamt werden dafür

$$(\lceil \log \log \ell_i \rceil + 1) + \lceil \log \log \ell_i \rceil + \lceil \log \ell_i \rceil$$

Bits benötigt. Für eine Zerlegung von t in m Worte folgt damit, dass

$$LZ77(t) = \sum_{i=1}^{m} \left(\log \left(\sum_{j<i} \ell_j \right) + \log \ell_i + 2 \log \log \ell_i \right) + O(m)$$

Offensichtlich ist $\sum_{j<i} \ell_j \leq |t|$. Weiterhin folgt aus der Tatsache, dass $\log x + \log \log x$ konkav ist, dass

$$\sum_i (\log \ell_i + 2 \log \log \ell_i) \leq m \left(\log(|t|/m) + 2 \log \log(|t|/m) \right)$$

Daher gilt

$$\begin{aligned} LZ77(t) &\leq m \log |t| + m \log \left(\frac{|t|}{m} \right) + O(m \log \log(|t|/m)) \\ &= m \log(m \cdot (|t|/m)) + m \log \left(\frac{|t|}{m} \right) + O(m \log \log(|t|/m)) \\ &= m \log m + 2m \log \left(\frac{|t|}{m} \right) + O(m \log \log(|t|/m)) \end{aligned}$$

Da LZ77 die Eingabe in paarweise verschiedene Worte zerlegt, folgt aus Lemma 3.19, dass

$$LZ77(t) \leq |t| \cdot H_k(t) + 3m \log \left(\frac{|t|}{m} \right) + O\left(km + m \log \log(|t|/m) \right)$$

Weiterhin gilt, dass $m = O(|t|/\log|t|)$ unter der Annahme, dass $s = |\Sigma|$ konstant ist, da für jeden Text t und jede Zerlegung von t in paarweise verschiedene Worte $w_1, \ldots, w_m$ gelten muss, dass es mindestens ein w_i gibt mit $|w_i| \geq \log_s(|t|/\log_s|t|)$. Daraus folgt, dass

$$LZZ77(t) \leq |t| \cdot H_k(t) + O\left(\frac{|t|\log\log|t|}{\log|t|} + \frac{|t|k}{\log|t|}\right)$$

ist, was die grobe Optimalität von LZ77 beweist. □

Trotz dieses Optimalitätsresultats zeigt der folgende Satz, dass für $H_1(t) \to 0$ die Kompressionsrate von LZ77 asymptotisch größer als $H_1(t)$ sein kann.

Satz 3.21 *Es gibt eine Konstante $c > 0$, so dass für jedes $n > 1$ es einen Text t der Länge n gibt mit*

$$\frac{LZ77(t)}{|t|} \geq c\frac{\log|t|}{\log\log|t|} \cdot H_1(t)$$

Beweis. Betrachte den folgenden Text:

$$t = 10^k2^{2^k}1\,101\,10^21\,10^31\,10^41\ldots10^k1$$

Es gilt $|t| = 2^k + O(k^2)$. Weiterhin sind $n_0 = \binom{k+1}{k}$, $n_1 = 2(k+1)$ und $n_2 = 2^k$ sowie $n_{00} = \binom{k}{2} + (k-1)$, $n_{01} = k$, $n_{02} = 1$, $n_{10} = k+1$, $n_{11} = k$, $n_{12} = 0$, $n_{20} = 0$, $n_{21} = 1$ und $n_{22} = 2^k - 1$. Der dominierende Term $n_{wi}\log(n_w/n_{wi})$ ist $n_{01}\log(n_0/n_{01}) = k\log((k+3)/2)$ und alle anderen Terme sind $O(k)$. Daraus folgt, dass $|t| \cdot H_1(t) = k\log k + O(k)$ ist. Die LZ77-Zerlegung von t ist

$$1 \quad 0 \quad 0^{k-1}2 \quad 2^{2^k-1}1 \quad 101 \quad 10^21 \quad 10^31 \quad \ldots10^k1$$

Für $i \geq 5$ gilt $\sum_{j<i}\ell_j > 2^k$. Also benötigt die Kodierung von p_i mindestens k bits. Da es $k+4$ Worte gibt, ist daher $LZ77(t) = \Omega(k^2)$, was den Satz beweist. □

LZ78 Code

Der LZ78 Algorithmus arbeitet wie folgt. Angenommen, der Text t sei bereits bis zu einer gewissen Position j in die Worte $w_1, \ldots, w_{i-1}$ zerlegt worden. Dann wählt LZ78 als das nächste Wort w_i ab Position j das längste Wort, für das $w_i^- = w_j$ für ein $j < i$ gilt. Sobald w_i gefunden worden ist, gibt LZ78 die Kodierung des Paares (j_i, z_i) aus, wobei j_i so gewählt ist, dass $w_{j_i} = w_i^-$ ist und z_i das letzte Zeichen in w_i ist. Wenn wir mit einem Alphabet mit s Zeichen arbeiten, dann braucht diese Kodierung höchstens $\lceil\log i\rceil + \lceil\log s\rceil$ Bits. Wenn der Text t also in m Worte zerlegt wird, dann gilt

$$LZ78(t) = m\log m + m\log s + O(m)$$

Der LZ78 Algorithmus kann effizient durch einen Trie implementiert werden, der für jedes w_i an der Stelle w_{j_i} um einen z_i-Übergang erweitert wird. Der nächste Satz zeigt, dass LZ78 leider noch nicht mal bezüglich H_0 gut komprimieren kann.

Satz 3.22 *Es gibt eine Konstante $c > 0$, so dass es für jedes $n \geq 1$ einen Text t der Länge n gibt mit*

$$\frac{LZ78(t)}{|t|} \geq c\sqrt{n}H_0(t)$$

Beweis. Betrachte den Text $t = 01^{n-1}$. LZ78 zerlegt t in die Worte

$$0 \quad 1 \quad 11 \quad 111 \quad 1111 \quad \ldots$$

Die Anzahl dieser Worte ist mindestens $\sqrt{n}$ und damit

$$LZ78(t) \geq \sqrt{n}\log(\sqrt{n}) = (\sqrt{n}/2)\log n$$

Auf der anderen Seite ist

$$H_0(t) = (1/n)\log n + (1 - 1/n)\log(1/(1 - 1/n)) = (\log n)/n + O(1/n)$$

woraus sich der Satz ergibt. □

Der LZ78 Algorithmus für sich alleine kann also im Allgemeinen nicht sonderlich gut komprimieren. Die Situation sieht anders aus, wenn man ihn mit dem RLE (Runlength Encoding) Algorithmus kombiniert.

RLE Code

Sei t ein Wort über dem Alphabet Σ und t' ein Teilwort von t. t' heißt α-*Segment* von t, falls t' nur aus α-Zeichen besteht und die Zeichen direkt vor und nach t' in t ungleich α sind. Sei 0 ein spezielles Symbol, das nicht in Σ ist. Für jedes $k \in \mathbb{N}$ und $\alpha \in \Sigma$ sei $B(\alpha^k)$ die Binärkodierung von k, in der jede 1 durch α ersetzt wird. D.h.

$$B(\alpha^5) = \alpha 0 \alpha$$

Für ein Wort t, so ist RLE(t) definiert als das Wort $\tilde{t}$, das wir erhalten, indem wir jedes α-Segment α^k in t durch $B(\alpha^k)$ ersetzen. D.h. aus $t = \alpha\alpha\alpha\beta\beta\alpha\gamma\gamma\gamma\gamma$ ergibt sich

$$\tilde{t} = \alpha\alpha\beta 0 \alpha\gamma 00$$

Offensichtlich ist $\tilde{t}$ eindeutig dekodierbar. In der Tat ist α immer das erste Symbol in jedem $B(\alpha^k)$ und wir wissen, dass wir das Ende von $B(\alpha^k)$ erreicht haben, wenn wir auf ein von α und 0 verschiedenes Symbol treffen. Außerdem gilt, dass $|B(\alpha^k)| = \lfloor \log k \rfloor + 1 \leq k$ und damit $|\tilde{t}| \leq |t|$ ist.

Analyse von LZ78$_{RL}$

Der LZ78$_{RL}$ Algorithmus transformiert die Eingabe t zunächst in die RLE-Kodierung $\tilde{t}$ und wendet dann den LZ78 Algorithmus auf $\tilde{t}$ an. Für diesen Algorithmus gilt der folgende Satz.

Satz 3.23 *Der LZ78$_{RL}$ Code ist grob optimal.*

Beweis. Zunächst brauchen wir das folgende technische Lemma, dessen Beweis wir hier nicht geben werden, da er recht komplex ist.

Lemma 3.24 *Eine LZ78 Zerlegung von $\tilde{t}$ in m Worte impliziert eine Zerlegung von t in m Worte, so dass jedes Wort höchstens $2\log|t|$-mal vorkommt.*

Damit folgt aus Lemma 3.19, dass

$$m \log m \leq |t| \cdot H_k(t) + m \log \left(\frac{|t|}{m} \right) + m \log \log |t| + O(k \cdot m)$$

Da die m Worte in $\tilde{t}$ alle verschieden sind, gilt $m = O(|\tilde{t}| / \log |\tilde{t}|) = O(|t| / \log |t|)$ und damit $\log |t| = O(|t|/m)$. Daraus ergibt sich, dass

$$m \log m \leq |t| \cdot H_k(t) + 2m \log \left(\frac{|t|}{m} \right) + O(k \cdot m)$$

ist. Da $m = O(|t| / \log |t|)$ ist, gilt für konstantes s, dass

$$\begin{aligned} LZ78_{RL}(t) &= m \log m + m \log s + O(m) \\ &\leq |t| \cdot H_k(t) + 2m \log \left(\frac{|t|}{m} \right) + O(k \cdot m) \\ &= |t| \cdot H_k(t) + O\left(\left(\frac{\log \log |t|}{\log |t|} + \frac{k}{\log |t|} \right) \cdot |t| \right) \end{aligned}$$

und damit LZ78$_{RL}$ grob optimal ist. □