Datenkompression

Joachim Pense

Vorlesung – Sommer 1992

Kapitel 1

Einleitung

Definition 1.1 (a) Unter einem *Alphabet* verstehen wir eine Menge Σ mit mindestens zwei Elementen. Wenn wir keine weiteren Angaben machen, ist Σ immer endlich.

Ist $n \in \mathbb{N}$ (inklusive 0), so ist Σ^n die n-fache kartesische Potenz von Σ, also die Menge der Folgen der Länge n mit Gliedern aus Σ. Mit Σ^* bezeichnen wir die Menge aller endlichen Folgen aus Σ, also

$$\Sigma^* = \bigcup_{n \in \mathbb{N} \cup \{0\}} \Sigma^n.$$

Die Elemente von Σ^* heißen *Texte über* Σ. Der Text der Länge 0 wird mit ε bezeichnet. Sind $A \in \Sigma^n$, $B \in \Sigma^m$, so bezeichnet AB den durch Konkatenation der beiden Texte entstandenen Text aus Σ^{n+m}. Mit $|A|$ bezeichnen wir die Länge von A. Ist $A \in \Sigma^n$, $i \in \{0, \ldots, n\}$, so bezeichnet A_i den aus den ersten i Zeichen bestehenden Teiltext von A.

(b) Sind Σ und Φ Alphabete, so ist eine *Kodierungsfunktion* eine injektive Abbildung $f: \Sigma^* \to \Phi^*$. Ist $A \in \Sigma^*$ ein Text, $f: \Sigma^* \to \Phi^*$ eine Kodierungsfunktion, so heißt A der *Originaltext*, $f(A)$ der *kodierte Text*, $f^{-1}: f(\Sigma^*) \to A$ die *Dekodierungsfunktion*. Die Anwendung von f bzw. f^{-1} heißen Kodierung bzw. Dekodierung.

Solche Funktionen werden in verschiedenen Theorien betrachtet:

- In der *Kryptographie* versucht man (beim Dechiffrieren=unerlaubtes Dekodieren) bei Kenntnis des kodierten Textes $f(A)$ aus statistisch signifikanten „Regelmäßigkeiten" dieses kodierten Textes den Text A und die Funktion f^{-1} zu erraten. Beim Chiffrieren=Kodieren versucht man das zu verhindern, indem man die statistischen Besonderheiten von A (z. B., daß der Buchstabe „e" besonders häufig vorkommt) nicht mit in den kodierten Text übergehen läßt. Man verschleiert also Redundanzen des Originaltextes.

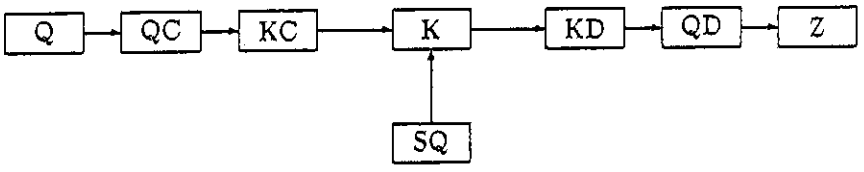
  (Im neueren Teilgebiet der sogenannten Public Key Systems sucht man nach Kodierungsfunktionen f, deren inverse Funktion sogar bei Kenntnis von f nur mit praktisch unvertretbarem Aufwand bestimmbar ist.)

- In der *Kodierungstheorie* (Theorie der Fehlerkorrektur) geht es darum, die Texte störungsunempfindlich zu machen, indem man Redundanz hinzufügt.

- In der Theorie der *Datenkompression* schließlich sucht man nach solchen Funktionen f, bei denen $f(A)$ möglichst kurz wird; es geht also um Entfernung von Redundanz.

Da bei den fehlerkorrigierenden Kodierungen mit einer Verlängerung der Texte zu rechnen ist, arbeiten Datenkompression und Fehlerkorrektur gegeneinander, während die Datenkompression zur Verschlüsselung des Textes im Sinne der Kryptographie einen gewissen Beitrag leistet (Redundanzen werden sicher dadurch verschleiert, daß man sie entfernt.)

Der mathematische Rahmen für diese verschiedenen Theorien wird in der von Claude Shannon 1948 begründeten *Informationstheorie* gelegt. In dieser wird ein digitales Kommunikationssystem wie in der Abbildung zerlegt:



Q – die Textquelle. Sie ist ein Aggregat, das Texte über das Alphabet Σ absondert; etwa ein Affe an der Schreibmaschine, ein Pascal-Compiler oder ein Sprecher des Deutschen.

QC – der Quellencodierer oder Kompressor. Er führt eine Kompression dieser Texte durch, es entstehen (möglichst kurze) Texte über einem zweiten Alphabet Φ.

KC – der Kanalcodierer. Er macht die vom Quellcodierer komprimierten Texte mit Methoden der Kodierungstheorie möglichst störungsunempfindlich.

K – der Kanal. Der Text wird nun abgeschickt.

SQ – die Störquelle. Auf seinem Weg ist der Text evtl. Veränderungen ausgesetzt.

KD – der Kanaldekodierer. Er versucht, den Störeinfluß wieder rückgängig zu machen.

QD – der Quelldekodierer oder Expander. Der Text wird wieder expandiert.

Z – das Ziel. Der Empfänger des Textes.

Es gibt zwei verbreitete Realisationen dieses Modells: einmal die tatsächliche Übermittlung digitaler Nachrichten über Leitungen, dann aber auch die Speicherung von Daten: der Kanal ist dabei das Speichermedium, etwa eine Diskette, die Störquelle vielleicht ein Magnet.

Wir wollen hier die Kanalkodierung, Störung und Kanaldekodierung als schwarzen Kasten auffassen und alle zusammen als (fast) störungsfreien Kanal betrachten. Die bekannten Quellkodierungsverfahren sind leider äußerst fehlerempfindlich. Wird im (z. B. um 50%) komprimierten Text ein Bit verfälscht, so ist es angemessen, von einem stabilen Kompressionsverfahren zu erwarten, daß der expandierte Text zwei falsche Bits enthält. Leider sind solche stabilen Verfahren unbekannt; im Normalfall pflanzt sich so ein Fehler über einen längeren Teiltext hin fort. Hier ist bisher noch nicht sehr viel geforscht worden, was sich zur Zeit allerdings ändert.

In der Praxis treten vor allem zwei typische Anwendungsgebiete der Datenkompression auf: einmal wird sie in der Nachrichtenübertragung verwendet (z. B. in der Raumfahrt), um die Kanalkapazität möglichst wenig zu beanspruchen (und damit mehr Nachrichten übertragen zu können); zum anderen in der Datenspeicherung, etwa in Sammlungen von Public-Domain-Software, in Datenbanken, bei der Speicherung von Graphiken.

Hier sieht man auch verschiedene Anforderungen an den Kompressionsalgorithmus: in der Nachrichtenübertragung wird hohe Kompressions- und auch hohe Expansionsgeschwindigkeit verlangt; in Softwaresammlungen ist das primäre Ziel eine hohe Kompressionseffektivität, bei Datenbanken und Graphiken die Expansionsgeschwindigkeit. Auch der während der Kompression nötige Speicherbedarf (der oft erheblich ist) ist von Bedeutung.

Bei der Bewertung der einzelnen Kompressionsverfahren muß man diese vier Aspekte berücksichtigen. Theoretische Abschätzungen durch Analyse der Algorithmen sind natürlich sinnvoll, aber müssen unbedingt durch praktische Tests flankiert werden: oft genug sind die Verfahren für bestimmte Arten von Texten gut, für andere überhaupt nicht geeignet. Bei der Bewertung der Kompressionseffektivität muß man sich zuerst über die Alphabete klar werden. Bei Computerdateien, also Folgen von Bits, sind Eingabe- und Ausgabealphabet einfach die Menge $\{0,1\}$; dann ist der Kompressionsfaktor (Länge des komprimierten Textes geteilt durch Länge des Originaltextes) ein sinnvolles Maß; oft hat man allgemeinere Eingabealphabete, (manchmal die unendliche Menge der natürlichen Zahlen), so daß hier die durchschnittliche Anzahl der Bits (im komprimierten Text) pro Zeichen (im Originaltext) sinnvoller ist. Oft bilden die Bytes (also die Menge $\{0,1\}^8$) das Eingabealphabet; in diesem Fall gehen die beiden Zahlen durch Multiplikation mit acht auseinander hervor.

Bei einer Dateianwendung kann man so vorgehen, erst den gesamten Originaltext zu lesen, um dann den komprimierten Text zu bestimmen (*off-line*); in einer Kommunikationsanwendung geht das nicht; hier muß gleichzeitig mit den eingehenden Zeichen des Originaltextes schon der komprimierte Text produziert werden (*on-line*), und zwar so, daß die Expansion dieses Teilstückes nur höchstens k Zeichen hinter dem Originaltext hinterherhinkt.

Der Vollständigkeit halber hier die formale Definition, die aber wahrscheinlich erst bei Kenntnis einiger Beispiele (arithmetische Kodierung) gut verständlich wird:

Definition 1.2 Sei $f: \Sigma^* \to \Phi^*$ ein Kodierungsverfahren. Ist $A \in \Sigma^n$, $f(A)$, so ist der *fertige Teil* von $f(A)$ das Anfangsstück $f(A)_j$ der Länge j, wobei j maximal ist mit der Eigenschaft, daß $f(A)_j = f(B)_j$ für alle $B \in \Sigma^*$ mit $A = B_n$. Von diesem fertigen Teil wählen wir das Anfangsstück $f(A)_\ell$ maximaler Länge $\ell \leq j$, das sich dekodieren läßt und dessen Dekodierung $A_m = f^{-1}(f(A)_\ell)$ sich nicht mehr ändert, wenn sich der Text verlängert (also $A_m = B_m$ für alle B mit $A = B_n$). Gibt es nun eine Konstante k, so daß $n - m \leq k$ für alle A, so heißt das Verfahren on-line, sonst off-line.

Das Prinzip von Datenkompressionsverfahren ist immer, irgendwelche Muster, Regelmäßigkeiten, Abweichungen von der Zufälligkeit, also *Redundanzen* im Originaltext aufzuspüren (z. B. unterschiedliche Zeichenhäufigkeiten oder mehrfach auftretende Teiltexte) und sich daraus ein *Modell* von der Textquelle zu machen (reflektiert etwa durch eine Tabelle der Zeichenhäufigkeiten oder ein Wörterbuch der öfter auftretenden Teiltexte). Man kann das Modell als den Teil des Kompressionsalgorithmus begreifen, der vorherzusagen versucht, welches Zeichen als nächstes kommt. Für dieses Modell sucht man dann eine optimale *Kodierung*, also die Vorschrift, welche Bits tatsächlich in den kodierten Text geschrieben werden.

Es gibt drei Arten der Modellverwendung:

Statische Modelle. Hier gibt es ein festes Modell (gegeben z. B. durch eine Häufigkeitstabelle der Buchstaben in der deutschen Sprache), das sowohl dem Kompressor und dem Expander vorliegen muß. Solche Verfahren arbeiten nur mit Texten gut, die dem Modell entsprechen (also z. B. deutschen Texten), können aber für diese sehr gute Kompressionsraten liefern.

Variable Modelle. Sie heißen auch „dynamisch" oder „semiadaptiv". Bei diesen Verfahren werden die Parameter des Modelles erst aus einer Analyse des Textes selbst gewonnen. Daher komprimiert ein solches Verfahren besser als ein statisches mit einem gleichartigen Modell; allerdings muß eine Repräsentation des aus dem Text gewonnenen Modelles (bzw. der daraus entstehenden Kodierungsvorschrift) dem komprimierten Text beigefügt werden, damit das Modell dem Expander zur Verfügung steht. Das reduziert wieder die Kompressionsleistung; daher können statische Verfahren wesentlich umfangreichere Modelle benutzen als variable. Variable Verfahren sind notwendigerweise Off-line-Verfahren.

Einen Extremfall bilden die „selbstexpandierenden Archive". Hier ist das Expansionsprogramm selbst ein Teil des komprimierten Textes.

Adaptive Modelle. Das Modell startet neutral (etwa mit einer gleichverteilten Häufigkeitstabelle oder einer leeren Liste mehrfach verwendeter Teiltexte) und paßt sich während des Kompressionsvorganges Zeichen für Zeichen an den Originaltext an. Die Kompression wird also gegen Ende des Textes besser; der Expander kann ebenfalls mit leeren Tabellen starten und das Modell auf die gleiche Weise wie der Kompressor aus dem jeweils bisher expandierten Text aufbauen. Diese Verfahren komprimieren also nicht so gut wie variable, müssen aber keine Repräsentation des Modells mitliefern; außerdem sind sie on-line-geeignet. In der Praxis sind die adaptiven Verfahren die wichtigsten.

Zwei einfache Beobachtungen können wir jetzt schon machen:

Satz 1.3 *Sei O. B. d. A.* $\Sigma = \Phi = \{0,1\}$, *f ein Kompressionsverfahren. Dann gibt es für jede natürliche Zahl n einen Text der Länge n, der von f nicht komprimiert wird.*

Beweis: Angenommen, jeder Text der Länge n würde von f auf einen Text kürzerer Länge abgebildet. Nun gibt es 2^n Texte der Länge n, aber nur $\sum_{i=0}^{n-1} 2^i = 2^n - 1$ Texte kleinerer Länge, woraus ein Widerspruch zur Injektivität von f folgt. ■

Satz 1.4 *Für jeden Text A gibt es ein Kompressionsverfahren, das diesen Text auf ein Bit komprimiert.*

Beweis: Wir setzen einfach

$$f(X) = \begin{cases} 0 & \text{falls } X = A \\ 1X & \text{sonst.} \end{cases}$$ ■

Läßt man bei der Kompressionsfunktion f die Bedingung der Injektivität fallen, kommt man auf die *Datenreduktion*, auch verlustbehaftete Datenkompression genannt. Man versucht hier, im Text unwesentliche Information einfach fallenzulassen. Was unwesentlich ist, hängt natürlich von der Anwendung ab. In der digitalen Musikverarbeitung (z.B. Digital Compact Cassette DCC) werden solche Verfahren eingesetzt. Rauschen z. B. klingt immer gleich, wenn einige statistische Parameter der Frequenzverteilung übereinstimmen; daher könnte man versuchen, statt des genauen Frequenzverlaufs nur diese Parameter abzuspeichern und anhand dieser Parameter mit einem Zufallsgenerator ein dem Original gleichendes Rauschen bei der Wiedergabe neu zu erzeugen. In dieser Vorlesung soll nur die verlustfreie Datenkompression betrachtet werden.

Klassische Beispiele für Datenkompression jenseits der Informatik liefert die Musiknotation: hier gibt es etwa die Wiederholungszeichen und das „da capo al fine", also Querverweise innerhalb des Textes. Diese Idee wird in den sogenannten „Lempel–Ziv 1977"-Kompressionsverfahren aufgegriffen.

In der Stenographie („Eng-Schreibung") gibt es die sogenannten Kürzel, das sind Zeichen, die für besonders häufige Wortbestandteile, Wörter oder Wortgruppen stehen (z. B. „ver-", „die", „ich stehe auf dem Standpunkt"). Das führt auf die statischen Lexikonverfahren, deren adaptive Versionen als „Lempel–Ziv 1978" bekannt sind. Die Grundidee der Stenographie ist es, besonders einfache Zeichen zu verwenden, und zwar die einfachsten Zeichen für die häufigsten Buchstaben. In der digitalen Welt heißt das, die Zeichen mit möglichst wenigen Bits zu kodieren, und zwar besonders häufige Zeichen mit besonders wenigen Bits. Das ergibt die Huffman-Kodierung (1952). Besonders komprimierte Texte erhält die deutsche Stenographie, daß sie vielen Buchstaben gar kein eigenes Zeichen zuordnet, sondern nur eine Modifikation der umliegenden Zeichen. So ähnlich verläuft es bei der arithmetischen Kodierung, die dadurch bessere Ergebnisse als das bekanntere Huffman-Verfahren liefert (Elias 1962, Rissanen 1979).

Beispiel: Run-Length-Kodierung

Manche Texteditoren (Vi, Emacs) ersetzen regelmäßig zur Platzersparnis im Text Gruppen von je 8 aufeinanderfolgenden Leerzeichen durch Tabulatorzeichen; andere (Turbo, FB17-Eve) unterdrücken Leerzeichen am Zeilenende. Das ist ein verlustbehaftetes Datenreduktionsverfahren (was sich gelegentlich störend bemerkbar macht); ein verlustfreies Datenkompressionsverfahren könnte darin bestehen, hinter jedem Leerzeichen eventuelle weitere Leerzeichen durch ein Byte zu ersetzen, in dem die Anzahl der jetzt noch folgenden Leerzeichen steht.

Dies ist ein typisches statisches Verfahren, das wohl nur für Programmtexte und Tabellen eine Kompression ergibt; gewöhnliche deutsche Texte werden dagegen erheblich verlängert, da ja hier im allgemeinen die Leerzeichen nur einfach vorkommen und jetzt jeweils zwei statt nur ein Byte belegen. Aber selbst in Programmtexten gibt es noch weitere Zeichen, die in Folgen (Runs) auftreten, etwa der * in Kommentarboxen.

Ein adaptives Verfahren bietet sich an:

```
var
    Liste: set of byte
    AZ, NZ: byte  (altes Zeichen, nächstes Zeichen)
    RunL:byte
end var
begin
    Liste mit ∅ initialisieren
    read(AZ)
    while Text noch nicht zuende
        read(NZ)
        if AZ = NZ
            if AZ ∈ Liste
                RunL := RunL + 1
                Behandlung des Falles RunL = 255
            else   (AZ ∉ Liste)
                write(AZ)
                RunL := 0
                Liste := Liste ∪ {AZ}
            end if
        else   (AZ ≠ NZ)
            write(AZ)
            if AZ ∈ Liste
                write(RunL)
                if RunL = 0
                    Liste := Liste \ {AZ}
                else   (RunL ≠ 0)
                    RunL := 0
                end if
                AZ:=NZ
            end if
        end if
```

end while
letztes Zeichen und ggf. dessen Run-Länge schreiben
end

Der Algorithmus in Worten:

Modellierung: Es liegt eine leer initialisierte Zeichenliste vor. Ein Zeichen wird in diese Liste aufgenommen, sowie es zum zweiten Mal hintereinander auftritt, es wird aus der Liste wieder entfernt, sowie es isoliert auftritt. (Ein Spezialfall tritt ein, wenn das Zeichen noch nicht in der Liste ist und dann gerade zweimal hintereinander auftritt. Dann wird es aufgenommen und sofort wieder entfernt.)

Kodierung: Ein Inputzeichen, das nicht in der Liste ist, wird ausgegeben. Bei Aufnahme in die Liste wird ein Zähler (RunL) mit Null initialisiert und das Zeichen ausgegeben. Ein Inputzeichen, das in der Liste ist, wird nicht ausgegeben, stattdessen wird RunL inkrementiert (bis zu einem Maximum von 255). Unterscheidet sich ein Inputzeichen vom vorigen Inputzeichen (oder hat der Zähler das Maximum erreicht), so wird das vorige Inputzeichen ausgegeben; ist das vorige Inputzeichen in der Liste, so wird zusätzlich der Zähler ausgegeben. Folgender Eingabetext:

```
abbcccaabbbcaabbbbccabbccccaaaabcccabccccccccccabcccccacbccccc
```

wird (nicht besonders stark) komprimiert zu:

```
abb0cc1aa0bb1c0a1b3cc0a0b1c3aa2b0c2a0bc9abc4ac0bcc3
```

Der Expander kann dem jeweiligen Byte des komprimierten Textes nicht ansehen, ob es sich um ein Zeichen oder einen Zähler handelt. Das muß er sich aus dem Zusammenhang erschließen. Er führt ebenfalls eine Liste mit, die die Zeichen enthält, deren Run-Length gezählt wird (d. h. das Modell). Für die Expansion liest man an dem Beispieltext die Prinzipien ab, aus denen man sich den Algorithmus selbst konstruieren kann:

- Regeln zum Schreiben von Zeichen:
  - Wurde ein Zeichen gelesen, so schreibe dieses Zeichen.
  - Wurde ein Zähler R gelesen, so schreibe das zuletzt geschriebene Zeichen noch R mal.
- Regeln zur „Erwartungshaltung" bezüglich des folgenden Zeichens:
  - Erwarte am Anfang des Textes ein Zeichen.
  - Erwarte nach einem Zeichen, das nicht in der Liste ist, ein Zeichen.
  - Erwarte nach einem Zeichen, das in der Liste ist, einen Zähler.
  - Erwarte nach einem Zähler ein Zeichen.

- Regeln zur Modellierung:
  - Initialisiere die Liste leer.
  - Nimm ein Zeichen, das zweimal unmittelbar hintereinander gelesen wurde, in die Liste auf.
  - Entferne ein Zeichen aus der Liste, wenn es mit dem Zähler 0 auftritt.

Die Kompressions- und auch die Expansionsgeschwindigkeit ist hoch; auf jeden Fall wird für jedes Zeichen eine konstante Zeit verbraucht.

Das Run-Length-Verfahren ist gut geeignet für Texte mit möglichst langen Runs; dies wären z. B. Programmquelltexte, aber auch (und in viel stärkerem Maße) Graphik-Pixeldateien. In solchen wird jeder Bildpunkt (der zum Beispiel 256 verschiedene Farben annehmen kann) durch (zum Beispiel) ein Byte repräsentiert. Wenn nun größere monochrome Flächen vorkommen, kann eine erhebliche Kompression erreicht werden.

Dies ist allerdings begrenzt durch das Maximum von einem Byte, das wir für die Kodierung Länge reservieren. Erhöhen wir diesen Platz (z. B. auf zwei Byte), so führt das bei Texten mit vielen langen Runs zwar zu einer Verbesserung der Kompressionsleistung, bei Texten mit vielen kurzen Runs aber zu einer Verschlechterung.

Eine platzsparende Methode, beliebig große natürliche Zahlen zu kodieren, ist die *Elias-Kodierung*, die uns auch im weiteren Verlauf der Vorlesung immer wieder begegnen wird. Sie wurde von Peter Elias 1975 vorgeschlagen.

Elias-Code nullter Stufe. Die Zahl n wird einfach durch eine Folge von $\lfloor \mathrm{ld}\, n \rfloor$ (binären) Nullen, gefolgt von ihrer Binärdarstellung repräsentiert. (Wir schreiben $\mathrm{ld}\, n := \log_2 n$, „logarithmus dualis"; $\lfloor \cdot \rfloor$ bezeichnet das Abrunden). Hierbei wird also der eigentlich von einer Zahl benötigte Platz verdoppelt, dafür muß man ihn nicht auf 8 (oder 16 etc.) aufrunden, und es gibt keine Obergrenze. Die Zahlen werden bitweise in den Ausgabetext gepackt, was den Kodierungs/Dekodierungsaufwand stark erhöht (auf jedes Byte wird achtmal zugegriffen!). Platz wird gegenüber der gewöhnlichen byteweisen Zahlenrepräsentation nur bei den Zahlen von 1 bis 8 gespart. Beachte: die Null läßt sich so nicht repräsentieren. (Man kann aber natürlich $\gamma_0(n)$ durch $\gamma_0(n+1)$ ersetzen.)

Elias-Code erster Stufe. Die Zahl n wird durch $\gamma_0(\lfloor \mathrm{ld}\, n \rfloor + 1)$, gefolgt von ihrer Binärdarstellung ohne die führende 1 repräsentiert, sie nimmt also

$$2(\lfloor \mathrm{ld}\, \lfloor \mathrm{ld}\, n \rfloor \rfloor + 1) + \lfloor \mathrm{ld}\, n \rfloor + 1$$

Binärstellen ein. Auch hier ist klar, daß sich die Zahlen aus einer Bitfolge rekonstruieren lassen; der Aufwand (ein Logarithmus und eine Anzahl Bitoperationen pro Zahl) erscheint vertretbar; für kleine Zahlen kodiert γ_1 kürzer, für große γ_2; man sieht, daß $\gamma_2(n)$ für $n \to \infty$ gegen die binäre Stellenzahl von n strebt.

n	$\gamma_0(n)$	$\gamma_1(n)$
1	1	1
2	010	0100
3	011	0101
4	00100	01100
5	00101	01101
6	00110	01110
7	00111	01111
8	0001000	00100000
9	0001001	00100001
10	0001010	00100010
11	0001011	00100011
17	000010001	001010001
20	000010100	001010100
30	000011110	001011110
40	00000101000	0011001000
50	00000110010	0011010010
100	0000001100100	00111100100
1000	0000000001111110010	0001001111110010
10000	00000000000001001110010010	00010110011100100100
100000	00000000000000001100001101010000	00001000110000110101000000

Elias-Codes höherer Stufe sind durch Weiterspinnen dieses Verfahrens möglich, in der Praxis zeigt sich aber, daß der Aufwand den Gewinn nicht rechtfertigt. Im Buch von Storer befindet sich eine Beschreibung von Elias-Codes der Stufe ∞ unter dem Namen „cascading lengths technique".

In der Computergraphik und auch in der digitalen Klangspeicherung hat man eine ganze Anzahl eigener, dafür besonders geeignete Kompressionsverfahren entwickelt, die aber nicht Gegenstand dieser Vorlesung sind. Ein Verfahren soll aber noch erwähnt werden:

Bilder enthalten, vor allem, wenn sie aus Fotos entstanden sind, nur selten große monochrome Flächen; dagegen enthalten sie große fast monochrome Flächen, bei denen sich benachbarte Bytes nur wenig unterscheiden. Das gilt auch für Musik-Samples: Aufeinanderfolgende Bytes (bzw. Wörter etc...) unterscheiden sich im allgemeinen nur wenig. Man kann solche Daten komprimieren, indem man statt der Zeichen die Differenzen aufeinanderfolgender Zeichen in einem Elias-Code abspeichert. Dies kann bei Bilddaten das horizontal und vertikal durchgeführt werden, bei animierten Bildern (Bildplatte) auch noch in die Zeitdimension hinein.

Kapitel 2

Zeichenkodierungen

Dieses Kapitel behandelt die Eigenschaften von Kompressionsverfahren, die den Originaltext Zeichen für Zeichen lesen und für jedes der Eingabezeichen eine Bitkette an den komprimierten Text anfügen. Wir werden im nächsten Kapitel sehen, daß die sogenannten Huffman-Codes unter diesen Verfahren optimale Kompression liefern.

In diesem Kapitel seien Σ und Φ Alphabete, wobei wir aus Bequemlichkeit $\Sigma = \{1, \ldots, n\}$ (wenn nicht anders gesagt) und $\Phi = \{0, 1\}$ annehmen.

Definition 2.1 Ein *Code* ist eine injektive Funktion $c: \Sigma \to \Phi^*$. Die $c(i), i \in \Sigma$ heißen die *Codewörter* von c.

Jedem Code c kann man eine Funktion $c^*: \Sigma^* \to \Phi^*$ zuordnen, indem man für $i_1, \ldots, i_j \in \Sigma$ definiert: $c^*(i_1 \cdots i_j) = c(i_1) \cdots c(i_j)$. Beachte, das diese Funktion noch keine Kodierungsfunktion zu sein braucht, da sie möglicherweise nicht injektiv ist. (Beispiel: $\Sigma = \{1, 2\}$, $c(1) = 0$, $c(2) = 00$; dann $000 = c^*(111) = c^*(12) = c^*(21)$). Ist sie aber injektiv, so heißt der Code c *eindeutig dekodierbar*.

Die wichtigsten eindeutig dekodierbaren Codes sind die *Präfix-Codes:*

Definition 2.2 Ein Präfix-Code ist ein Code mit der Eigenschaft, daß kein Codewort der Anfang (Präfix) eines anderen Codewortes ist.

Offensichtlich sind Präfix-Codes eindeutig dekodierbar; die Umkehrung gilt aber nicht (Beispiel: $c(1) = 01$, $c(2) = 011$); es ist jedoch so, daß nur Präfix-Codes praktische Bedeutung haben; ihr Vorteil ist, daß man ein Zeichen schon beim Lesen des letzten Bit seines Codewortes dekodieren kann und nicht in die dann folgenden Zeichen hineinschauen muß.

Wir interessieren uns (auch im Hinblick auf adaptive Modelle) für allgemeinere Kodierungsfunktionen als solche der Form c^*; wir wollen weiterhin jedes Zeichen einzeln kodieren, aber der Code soll von Zeichen zu Zeichen wechseln dürfen, natürlich nur in Abhängigkeit vom bisherigen Text *ohne* das zu kodierende Zeichen.

Definition 2.3 Eine Kodierungsfunktion $f: \Sigma^* \to \Phi^*$ heißt *Zeichenkodierung*, wenn es für jedes $X \in \Sigma^*$ einen Code $c_X: \Sigma \to \Phi$ gibt, so daß für $i_1, \ldots, i_j \in \Sigma$

immer gilt:

$$f(i_1 i_2 i_3 \cdots i_j) = c_\varepsilon(i_1) c_{i_1}(i_2) c_{i_1 i_2}(i_3) \cdots c_{i_1 \cdots i_{j-1}}(i_j).$$

Ist c_X von X unabhängig, also konstant ein Code c, so heißt die Zeichenkodierung f eine *konstante Zeichenkodierung*. Das sind also gerade die Kodierungsfunktionen der Form c^*. Man kann diese Funktionen auch durch die *Homomorphieeigenschaft* $f(AB) = f(A)f(B)$ für alle Texte A, B charakterisieren.

Offenbar gilt die eindeutige Dekodierbarkeit der Präfixcodes auch allgemeiner:

Satz 2.4 *Sei für jedes $X \in \Sigma^*$ der Code c_X ein Präfixcode. Bilde ich nun wie in Definition 2.3 die Funktion f, so ist f eine Kodierungsfunktion.* ■

Wenn ich eine Datei mit einer Zeichenkodierung komprimiere, ist trivialerweise ein Bit pro Zeichen eine untere Schranke. Eine schärfere Bedingung liefert die *Kraft-McMillan-Ungleichung* (von Kraft (1949) für Präfix-Codes, McMillan (1956) allgemein, der kurze Beweis stammt von Karush (1961)):

Satz 2.5 *Sei $c: \Sigma \to \Phi^*$ eindeutig dekodierbar. Dann gilt*

$$\sum_{i \in \Sigma} \frac{1}{2^{|c(i)|}} \leq 1$$

Beweis: Schreibe $\ell_i := |c(i)|$, $\ell = \max_i \ell_i$. Dann gilt für alle $k > 0$:

$$\begin{aligned} \left(\sum_{i=1}^{n} \frac{1}{2^{\ell_i}} \right)^k &= \sum_{i_1, \ldots, i_k = 1}^{n} \frac{1}{2^{\ell_{i_1}} \cdots 2^{\ell_{i_k}}} \\ &= \sum_{s \in \Sigma^k} \frac{1}{2^{|c^*(s)|}}. \end{aligned}$$

Nun gilt: $\max_{s \in \Sigma^k} |c^*(s)| = k\ell$ und $\min_{s \in \Sigma^k} |c^*(s)| \geq k$, und damit

$$\begin{aligned} \left(\sum_{i=1}^{n} \frac{1}{2^{\ell_i}} \right)^k &= \sum_{j=k}^{k\ell} \frac{|\{s \in \Sigma^k \mid |c^*(s)| = j\}|}{2^j} \\ &\leq \sum_{j=k}^{k\ell} \frac{2^j}{2^j} \\ &\leq k\ell. \end{aligned}$$

Wäre nun $\sum_{i=1}^{n} \frac{1}{2^{\ell_i}}$ größer als 1, so würde die k-te Potenz für hinreichend großes k den Wert $k\ell$ übersteigen, folglich $\sum_{i=1}^{n} \frac{1}{2^{\ell_i}} \leq 1$. ■

Umgekehrt gilt:

Satz 2.6 *Sei $\Sigma = \{1, \ldots, n\}$, und $\ell_1, \ldots, \ell_n$ natürliche Zahlen, die die Kraft-McMillan-Ungleichung $\sum_{i=1}^{n} \frac{1}{2^{\ell_i}} \leq 1$ erfüllen. Dann gibt es einen Präfix-Code $c: \Sigma \to \Phi^*$ mit der Eigenschaft $|c(i)| = \ell_i$.*

Beweis: Wir können ohne Einschränkung $\ell_1 \leq \cdots \leq \ell_n$ annehmen. Wir gehen mit Induktion nach n vor. Für $n = 1$ suchen wir uns einfach ein beliebiges $c(1)$ der Länge ℓ_1; Angenommen, wir haben schon die Codewörter $c(j)$, $j = 1, \ldots, n-1$ gefunden, die die Präfixbedingung erfüllen. Diese Induktionsannahme können wir machen, da die Ungleichung mit einem Summanden weniger erst recht gilt. Nun gibt es $\sum_{i=1}^{n-1} 2^{\ell_n - \ell_i}$ Wörter in Φ^{ℓ_n}, die eines der $c(i)$ als Präfix haben. Nun gilt $|\Phi^{\ell_n}| = 2^{\ell_n} \geq 2^{\ell_n} \sum_{i=1}^{n} \frac{1}{2^{\ell_i}} > \sum_{i=1}^{n-1} 2^{\ell_n - \ell_i}$, so daß noch ein mögliches $c(n)$ übrigbleibt. ■

Korollar 2.7 *Für jeden eindeutig dekodierbaren Code gibt es einen Präfixcode mit den gleichen Zeichenlängen.* ■

Dieses Korollar ist unsere Rechtfertigung, uns auf Präfixcodes zu beschränken.

Ein Text $A \in \Sigma^*$ soll also nun mit einer Zeichenkodierung komprimiert werden; nehmen wir an, daß wir den Anfang A_j schon kodiert haben und jetzt für das $j+1$-te Zeichen eine optimale Kodierungsfunktion suchen. Wir nehmen ferner an, daß wir aufgrund unseres Modelles eine Wahrscheinlichkeitsverteilung $p = (p_1, \ldots, p_n)$ kennen, die uns sagt, daß nach diesem Textanfang das Zeichen i mit der Wahrscheinlichkeit p_i zu erwarten ist. (Es gilt insbesondere: $\sum_{i=1}^{n} p_i = 1$, $0 < p_i < 1$).

Codiere ich nun das nächste Zeichen mit dem Code c, $\ell_i = |c(i)|$, so ist der Erwartungswert für die Anzahl der Bits, die ich für das nächste Zeichen brauche, gleich

$$\sum_{i=1}^{n} p_i \ell_i .$$

Diese Zahl gilt es zu minimieren, wobei die Kraft-McMillan-Ungleichung eine Nebenbedingung liefert.

Lemma 2.8 *Die Funktion* $F: \mathbf{R}^n \to \mathbf{R}$,

$$F(x_1, \ldots, x_n) = \sum_{i=1}^{n} p_i x_i$$

nimmt unter Beachtung der Bedingungen $x_i > 0$,

$$\sum_{i=1}^{n} \frac{1}{2^{x_i}} \leq 1$$

ein absolutes Minimum am Punkt $(X_1, \ldots, X_n)$ *mit* $X_i = -\mathrm{ld}\, p_i$ *an.*

Beweis: Da F eine lineare Funktion ist, die mit steigenden x_i steigt, nimmt sie ihr Minimum auf dem Rand des durch die Bedingungen gegebenen Definitionsbereiches an, also auf der Menge

$$\left\{ (x_1, \ldots, x_n) \mid \sum_{i=1}^{n} \frac{1}{2^{x_i}} = 1 \right\}.$$

Wir suchen ein lokales Extremum der Funktion

$$\begin{aligned} g(x_1,\ldots,x_{n-1}) &= f\left(x_1,\ldots,x_{n-1},-\mathrm{ld}\left(1-\sum_{i=1}^{n-1}2^{-x_i}\right)\right) \\ &= \sum_{i=1}^{n-1}p_i x_i - p_n\left(\mathrm{ld}\left(1-\sum_{i=1}^{n-1}2^{-x_i}\right)\right) \end{aligned}$$

auf dem Definitionsbereich $\{(x_1,\ldots,x_{n-1}) \mid \text{alle } x_i > 0\}$.

Die partiellen Ableitungen

$$\frac{\partial g(x_1,\ldots,x_{n-1})}{\partial x_j} = p_j - \frac{p_n \cdot 2^{-x_j}}{1-\sum_{i=1}^{n-1}2^{-x_i}}$$

müssen für jedes j verschwinden, was auf die Gleichungen

$$\begin{aligned} x_1 + \mathrm{ld}\, p_j &= x_n + \mathrm{ld}\, p_n, \\ &\vdots \\ x_{n-1} + \mathrm{ld}\, p_{n-1} &= x_n + \mathrm{ld}\, p_n, \\ \sum_{i=1}^{n} 2^{-x_i} &= 1 \end{aligned}$$

führt.

Der angegebene Punkt $(X_1,\ldots,X_n)$ ist eine Lösung dieses Gleichungssystems; wir wollen, daß es keine weiteren Lösungen gibt.

Die Lösungsmenge der ersten $n-1$ Gleichungen ist die Gerade

$$(X_1,\ldots,X_n) + \{(t,\ldots,t) \mid t \in \mathbf{R}\}.$$

Schneide ich diese mit der durch die letzte Gleichung gegebene Mannigfaltigkeit so erhalte ich

$$\begin{aligned} 1 &= \sum_{i=1}^{n} 2^{-(X_i+t)} \\ &= 2^{-t}\sum_{i=1}^{n} 2^{-X_i} \\ &= 2^{-t}, \end{aligned}$$

also $t = 0$, was zu zeigen war. ■

Der Wert der Funktion F an der Minimalstelle, also $-\sum_{i=1}^{n} p_i \mathrm{ld}\, p_i$ heißt die *Entropie* der Wahrscheinlichkeitsverteilung p und stellt eine untere Schranke für die zu erwartende Codewortlänge des nächsten Zeichens dar. Dieser Satz gilt viel allgemeiner als nur für Zeichenkodierungen und wird uns in dem Kapitel über Textquellen näher beschäftigen.

Wir brauchen natürlich ganzzahlige Lösungen; wenn man nun die x_i einfach aufrundet, also $\ell_i := \lceil -\mathrm{ld}(p_i) \rceil$ setzt, erfüllen die ℓ_i die Kraft-McMillan-Ungleichung, und nach Satz 2.6 gibt es einen Präfix-Code mit den so vorgegebenen Codewortlängen. So erhält man den *Fano-Shannon-Code*, (Fano 1949, Shannon 1949) der aber heute nur noch historischen Wert hat.

Für die Konstruktion von Präfix-Codes ist die Datenstruktur des *binären Suchbaumes* wichtig.

Definition 2.9 Ein *Baum* $\mathcal{B}$ besteht aus

- einer endlichen Menge K, den *Knoten*,
- einem ausgezeichneten Element $w \in K$, der *Wurzel*,
- einer Funktion $v\colon K \setminus \{w\} \to K$, der *Vorgängerfunktion*,

so daß folgende Bedingunge erfüllt ist:

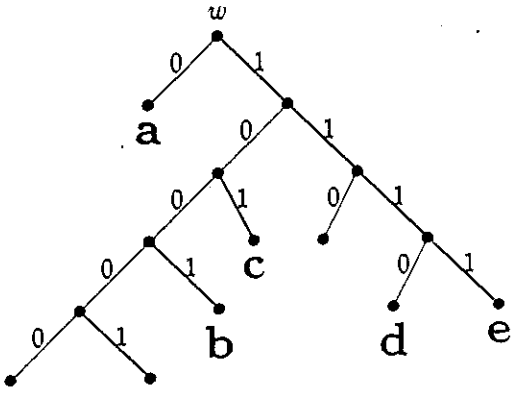
- für jeden Knoten $k \in K$ gibt es eine natürliche Zahl t, so daß $v^t(k) = w$.

Das kleinste derartige t heißt die *Tiefe* des Knotens k, die größte Knotentiefe von $\mathcal{B}$ heißt die Tiefe von $\mathcal{B}$. Ein Knoten, der nicht Vorgänger eines anderen Knotens ist, heißt *Blatt*.

Definition 2.10 Ein Baum, bei dem jeder Knoten, der kein Blatt ist, genau zwei Nachfolger hat, heißt *binär*.

Definition 2.11 Ein binärer Baum heißt *binärer Suchbaum* bezüglich Σ und Φ, wenn eine injektive Abbildung $\mathrm{Bl}\colon \Sigma \to \{\text{Blätter von } \mathcal{B}\}$ gegeben ist, und für jeden Knoten, der kein Blatt ist, die beiden Nachfolger eindeutig den Elementen von Φ zugeordnet sind. Für $i \in \Sigma$ heißt dann das Blatt $\mathrm{Bl}(i)$ *mit i markiert*. Ist die Funktion Bl bijektiv, so heißt der binäre Suchbaum *vollständig*.

Diese formale Definition wird bei Betrachtung der Abbildung klar, die einen binären Suchbaum mit $\Sigma = \{\mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{d}, \mathsf{e}\}$ zeigt.



Definition 2.12 Jedem binären Suchbaum zu Σ und Φ ist ein Code $c: \Sigma \to \Phi^*$ auf folgende Weise zugeordnet: Sicher gibt es für jedes $i \in \Sigma$ einen eindeutig bestimmten Weg von der Wurzel zum mit i markierten Blatt $\mathrm{Bl}(i)$. Auf diesem Weg überstreicht man der Reihe nach Knoten, denen jeweils eines der beiden Elemente von Φ zugeordnet ist; die entstehende Folge aus Φ^* bezeichnen wir mit $c(i)$.

Es ist leicht zu sehen, daß auf diese Weise ein Präfixcode entsteht, und daß man zu jedem gegebenen Präfixcode einen Baum konstruieren kann.

Eine Eigenschaft binärer Suchbäume, die sich nicht auf die Situation $|\Phi| > 2$ verallgemeinern läßt, ist:

Satz 2.13 *Jeder binäre Suchbaum kann zu einem vollständigen binären Suchbaum reduziert werden, indem immer wieder Knoten mit zwei unmarkierten Blättern durch ein unmarkiertes Blatt ersetzt und Knoten mit einem unmarkierten und einem markierten Blatt einfach durch dieses markierte Blatt ersetzt werden. Der entstehende Baumcode hat für jede Wahrscheinlichkeitsverteilung eine kürzere erwartete Codelänge als der ursprüngliche.* ■

Wir können also von vorneherein annehmen, daß unsere binären Suchbäume vollständig sind. Es stellt sich die Frage, welche Codes nun noch darstellbar sind, deren Antwort einfach zu zeigen ist (Übungsaufgabe!):

Satz 2.14 *Ein Code c ist genau dann durch einen vollständigen binären Suchbaum repräsentierbar, wenn gilt*

- *c ist ein Präfixcode;*
- *jede Bitkette $p \in \Phi^*$ ist Präfix eines Codewortes oder hat ein Codewort als Präfix.*

■

Codes, die durch vollständige binäre Suchbäume darstellbar sind, heißen kurz *Baumcodes*.

Der einfache Beweis der folgenden hübschen Tatsache ist ebenfalls dem Leser zur Übung überlassen:

Satz 2.15 *In einem Baumcode wird die Kraft-McMillan-Ungleichung eine Gleichung.*

Vollständige binäre Bäume als Datenstrukturen

Wir gehen von einer Konstanten n und den beiden Typen Sigma = 1..n und Phi = 0..1 aus. Eines vollständigen binären Suchbaum könnte man etwa folgendermaßen implementieren:

```
type
    Knoten = ↑Knoteninhalt
    Knoteninhalt = record
```

```
        Vorgänger: Knoten
        case Blatt: boolean
            true:
                Markierung: Sigma
            false:
                Nachfolger: array[Phi] of Knoten
        end case
    end Knoteninhalt
end type
```

Ohne uns jetzt weiter für diese Implementation zu interessieren, können wir den abstrakten Datentyp „Knoten" durch seine primitiven Operationen beschreiben

function Vorgänger(b: Knoten): Knoten
Liefert, falls er existiert, den Vorgangerknoten des Knotens b.
function Nachfolger(b: Knoten, p:Phi):Knoten
Liefert, falls er existiert, den mit p *markierten Nachfolgerknoten von* b.
function Nachfolgermarkierung(b: Knoten): Phi
Liefert die Markierung von b *als Nachfolger seines Vorgängers, wenn ein solcher existiert*
function Blattmarkierung(b: Knoten): Sigma
Falls b *ein Blatt ist, liefert die Funktion die Markierung dieses Blattes.*
function Blatt(b: Knoten, s: Sigma): Knoten
Liefert, falls es existiert, das mit s *markierte Blatt unter* b.
function IstWurzel(b: Knoten): boolean
true, *wenn der Knoten* b *die Wurzel ist, also keinen Vorgänger hat*
function IstBlatt(b: Knoten): boolean
true, *wenn der Knoten* b *ein Blatt ist.*

function NeuerBaum(a: Sigma): Knoten;
Liefert einen Baum, dessen einziger Knoten ein mit a *markiertes Blatt ist.*
function Verbindung(b,c: Knoten): Knoten;
Hängt die Bäume unter die Wurzel eines neuen Baumes, so daß b *unter den* 0*-Zweig und* c *unter den* 1*-Zweig kommt.*
procedure EntferneBlatt(w,b:Knoten)
Entfernt aus dem Baum mit Wurzel w *das Blatt* b. *Der Vorgängerknoten von* b *wird durch das andere Blatt des Zweiges ersetzt.*

Die Implementation dieser Operationen ist weitgehend klar; um die Funktion „Blatt" effizient zu machen, sollte man für jeden Baum zusätzlich eine Tabelle implementieren, die für jedes $s \in \Sigma$ auf das zugehörige Blatt verweist, also etwa ein **array**[Sigma] **of** Knoten.

Der Kodierungs- und der Dekodierungsalgorithmus sind jetzt klar: Bei der Kodierung arbeitet man sich vom Blatt zur Wurzel hoch und erhält die Bitkette rückwärts, bei der Dekodierung geht es von der Wurzel zum Blatt.

Kodierung:

```
var
    Puffer: string of Phi
    s: Sigma
    Codebaum: Knoten
    Stelle: Knoten
end var

begin
Puffer := Leerstring
    read(s)
    Stelle := Blatt(Codebaum,s)
    while not IstWurzel(Stelle)
        append(Puffer,Nachfolgermarkierung(Stelle))
        Stelle := Vorgänger(Stelle)
    end while
    reverse(Puffer)
    write(Puffer)
end
```

Dekodierung:

```
var
    p: Phi
    Codebaum: Knoten
    Stelle: Knoten
end var

begin
    Stelle := die Wurzel von Codebaum
    while not IstBlatt(Stelle)
        read(p)
        Stelle := Nachfolger(Stelle,p)
    end while
    write(Blattmarkierung(Stelle))
end
```

Kapitel 3

Huffman-Codes

Die bekannteste Kodierungstechnik benutzt die von David Huffman 1952 eingeführten Huffman-Codes. Diese lösen das Problem, für eine gegebene Wahrscheinlichkeitsverteilung einen Code mit optimaler erwarteter Codewortlänge zu finden.

Wir erweitern den Typ des binären Suchbaumes insofern, als wir jedem Knoten eine reelle Zahl, sein *Gewicht*, zuordnen. Der abstrakte Datentyp wird damit um zwei primitive Prozeduren erweitert:

function Gewicht(b: Knoten): real
    Liefert (falls es existiert) das Gewicht des Knotens b.
procedure SetzeGewicht(b: Knoten, g: real)
    Ordnet dem Knoten b *das Gewicht* g *zu.*

Ist jetzt für das Alphabet $\Sigma = \{1, \ldots, n\}$ die Wahrscheinlichkeitsverteilung $p_1, \ldots, p_n$ gegeben, so lautet der Huffman-Algorithmus zur Konstruktion eines Baumcodes:

```
var
    Baumliste: list of Knoten (siehe unten)
    b,x,y: Knoten
    i: integer
end var

begin
    Die Baumliste leer initialisieren
    for i := 1 to n
        b := NeuerBaum(i)
        SetzeGewicht(b,p_i)
        sortiere b in die Baumliste ein, so daß in dieser
            die Wurzelgewichte der Mitglieder steigen
    end for
    for i := n downto 2
        x und y := die beiden Bäume am Anfang von Baumliste
        b := Verbindung(x,y) oder Verbindung(y,x), das ist egal
```

```
        SetzeGewicht(b,Gewicht(x)+Gewicht(y))
        entferne x und y aus der Baumliste
        sortiere b in Baumliste ein, so daß in dieser
            die Wurzelgewichte der Mitglieder steigen
    end for
end
```

Die Implementation des Datentypes „**list**" interessiert uns im Moment noch nicht; wir stellen uns darunter jedenfalls eine Folge von Bäumen vor, die eine Anordnung besitzt.

Beispiel 3.1

Σ	a	b	c	d	e	f
p_i	0.05	0.1	0.15	0.2	0.25	0.25

Beachte: bei den Bäumen ist für diesen Algorithmus immer nur das Gewicht der Wurzel interessant.

Initialisierung

Baumliste	a	b	c	d	e	f
Wurzelgewicht	0.05	0.1	0.15	0.2	0.25	0.25

Erster Schritt

Baum	a b	c	d	e	f
Wurzelgewicht	0.15	0.15	0.2	0.25	0.25

Zweiter Schritt

Baum	d	e	f	c a b
Wurzelgewicht	0.2	0.25	0.25	0.3

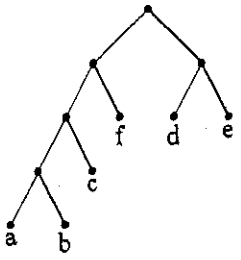
Dritter Schritt

Baum	f	c a b	d e
Wurzelgewicht	0.25	0.3	0.45

Vierter Schritt

Baum	d e	f c a b
Wurzelgewicht	0.45	0.55

Der Baum



Der entstehende Code ist also a: 0000, b: 0001, c: 001, d: 10, e: 11, f: 01. Die erwartete Codewortlänge ist $0.05 \cdot 4 + 0.1 \cdot 4 + 0.15 \cdot 3 + 0.2 \cdot 2 + 0.25 \cdot 2 + 0.25 \cdot 2 = 2.45$ bit.

Die Entropie, also die im letzten Kapitel angegebene untere Schranke ist $-0.05 \cdot \mathrm{ld}\, 0.05 - 0.1 \mathrm{ld}\, 0.1 - 0.15 \mathrm{ld}\, 0.15 - 0.2 \mathrm{ld}\, 0.2 - 2 \cdot 0.25 \mathrm{ld}\, 0.25 = 0.05 \cdot 4.32 + 0.1 \cdot 3.32 + 0.15 \cdot 2.74 + 0.2 \cdot 2.32 + 0.5 \cdot 2 = 0.0216 + 0.332 + 0.411 + 0.464 + 1 = 2.23$; die erwartete Codewortlänge, die ein Fano–Shannon-Verfahren liefert, ist $0.05 \cdot 5 + 0.1 \cdot 4 + 0.15 \cdot 3 + 0.2 \cdot 3 + 0.5 \cdot 2 = 0.25 + 0.4 + 0.45 + 0.6 + 1 = 2.7$.

Satz 3.2 *Sind im Rahmen der Wahlmöglichkeiten des Huffman-Algorithmus zwei verschiedene Codes entstanden, so haben diese dennoch die gleiche erwartete Codewortlänge.*

Beweis: Es gibt zwei Arten der Wahlmöglichkeit im Huffman-Algorithmus: die erste besteht darin, sich bei zwei ausgewählten Bäumen x und y minimalen Gewichtes für eine der beiden Reihenfolgen „Verbinde(x,y)" oder „Verbinde(y,x)" zu entscheiden. Dies ist natürlich ohne Einfluß auf die Codewortlängen.

Die andere Wahlmöglichkeit liegt in der Freiheit, unter mehreren Bäumen gleichen Gewichtes zu entscheiden. Existieren genau $k > 2$ Bäume gleichen minimalen Gewichtes g in der Liste, so kommen diese jedenfalls in irgendeiner Reihenfolge in den nächsten $\lceil \frac{k}{2} \rceil$ Schritten an die Reihe; alle Bäume, die noch entstehen, haben nämlich größeres Gewicht als g, können sich also nicht dazwischenschieben. Zu zeigen ist also, daß eine Permutation von k Bäumen am Anfang der Baumliste nichts an der erwarteten Codewortlänge ändert; da eine Permutation bekanntlich das Produkt von Vertauschungen ist, ist zu zeigen, daß die erwartete Codewortlänge sich durch Vertauschung zweier Bäume gleichen Gewichtes nicht ändert.

Führe ich den Huffman-Algorithmus zweimal nebeneinander aus, wobei ich bei einer der beiden Ausführungen irgendwann während des Ablaufes die gleichschweren Bäume $\mathcal{P}$ und $\mathcal{Q}$ in der Baumliste vertausche, so entstehen am Schluß die beiden Huffman-Bäume $\mathcal{H}$ und $\mathcal{H}'$, die beide sowohl $\mathcal{P}$ als auch $\mathcal{Q}$ als Teilbaum haben, sich aber genau dadurch unterscheiden, daß $\mathcal{P}$ und $\mathcal{Q}$ im Baum $\mathcal{H}'$ gegenüber $\mathcal{H}$ vertauscht sind. Sei g das Wurzelgewicht von $\mathcal{P}$, das ja mit dem von $\mathcal{Q}$ übereinstimmt.

Wir setzen $\Sigma = \Pi \cup \Psi \cup \Gamma$, wobei Π die Zeichen zu den Blättern von $\mathcal{P}$, Ψ dasselbe zu $\mathcal{Q}$, und Γ die restlichen Zeichen bezeichnet.

Wir wollen $\ell(\mathcal{B})$ für die erwartete Codewortlänge eines Huffman-Baumes $\mathcal{B}$ schreiben. $\ell(\mathcal{P})$ ist die erwartete Codewortlänge zu dem Baumcode zum Baum $\mathcal{P}$, der zu dem Alphabet Π gebildet ist, wobei jedes Zeichen i aus Π die Wahrscheinlichkeit $\frac{p_i}{g}$ hat; das ist sinnvoll, denn g ist gerade die Wahrscheinlichkeit,

daß ein Zeichen in Π liegt. Wir beobachten:

$$\sum_{i\in\Pi} p_i = g, \quad \ell(\mathcal{P}) = \sum_{i\in\Pi} \frac{p_i}{g}\left(\ell_i - t(\mathcal{P})\right),$$

wobei $t(\mathcal{P})$ die Tiefe des Wurzelknotens von $\mathcal{P}$ im Baum $\mathcal{H}$ ist.

Dasselbe machen wir für $\mathcal{Q}$ und berechnen

$$\begin{aligned}
\ell(\mathcal{H}) &= \sum_{i\in\Sigma} p_i\ell_i \\
&= \sum_{i\in\Pi} p_i\ell_i + \sum_{i\in\Psi} p_i\ell_i + \sum_{i\in\Gamma} p_i\ell_i \\
&= (\ell(\mathcal{P}) + t(\mathcal{P}))\, g + (\ell(\mathcal{Q}) + t(\mathcal{Q}))\, g + \sum_{i\in\Gamma} p_i\ell_i \\
&= (\ell(\mathcal{Q}) + t(\mathcal{P}))\, g + (\ell(\mathcal{P}) + t(\mathcal{Q}))\, g + \sum_{i\in\Gamma} p_i\ell_i \\
&= = \ell(\mathcal{H}'). \qquad \blacksquare
\end{aligned}$$

Satz 3.3 (Optimalitätssatz für den Huffman-Code) *Bezüglich einer gegebenen Wahrscheinlichkeitsverteilung hat ein Präfixcode genau dann minimale erwartete Codewortlänge, wenn es sich um einen Huffman-Code handelt.*

Beweis: Sei c ein Präfixcode mit minimaler erwarteter Codewortlänge, der aber kein Huffman-Code ist. Sei $n = |\Sigma|$ minimal, daß so ein c existiert, und $p = (p_1, \cdots, p_n)$ die Wahrscheinlichkeitsverteilung.

Nach Satz 2.13 ist c ein Baumcode. Wir können $p_i \geq \cdots \geq p_n$ annehmen; dann gilt sicher auch $\ell_1 \leq \cdots \leq \ell_n$. Die Baumcodeeigenschaft liefert $\ell_{n-1} = \ell_n$, und die beiden Blätter $n-1$ und n haben einen gemeinsamen Vorgänger. Wir fassen nun die Zeichen $n-1$ und n zu einem neuen Zeichen mit der Wahrscheinlichkeit $p_{n-1}+p_n$ zusammen, betrachten also den Baumcode $d\colon \{1, \cdots, n-1\} \to \Phi^*$, der aus dem von c entstanden ist, daß im Baum von c der Zweig mit den beiden zusammengehörigen Blättern $n-1$ und n durch ein einziges Blatt $n-1$ ersetzt wurde. Ist ℓ_c die erwartete Codewortlänge von c, so ist $\ell_d = \ell_c - (p_{n-1} + p_n)$.

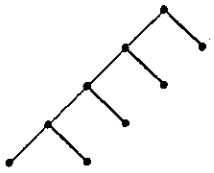
Wäre d ein Huffman-Code, so wäre c auch ein Huffman-Code. Das ist aber nicht der Fall, also ist d nach der Annahme über n nicht optimal. Dan gibt es einen besseren Code $e\colon \{1, \ldots, n-1\} \to \Phi^*$. Es gilt$\ell_e < \ell_d$. Jetzt spalten wir im Code e das Zeichen $n-1$ wieder auf und erhalten einen neuen Code $f\colon \Sigma \to \Phi^*$ mit $\ell_f = \ell_e + (p_{n-1} + p_n) < \ell_d + (p_{n-1} + p_n) = \ell_c$, ein Widerspruch zur Optimalität von c.

Ist umgekehrt c ein Huffman-Code, so betrachte ich einen optimalen Code d. So etwas existiert aus Endlichkeitsgründen, und d ist nach der eben gezeigten anderen Richtung ebenfalls ein Huffman-Code. Nach Satz 3.2 hat dieser die gleiche erwartete Codewortlänge wie c, so daß c ebenfalls optimal ist. $\blacksquare$

Satz 3.4 *Die Erzeugung eines Huffman-Baumes zu n Zeichen kann mit einem Aufwand der Größenordnung $O(n \log n)$ geleistet werden.*

Beweis: Die Anzahl der Huffman-Schritte ist proportional zu der Anzahl der Knoten des Baumes $\mathcal{B}$ (ein Initialisierungsschritt für jedes Blatt und ein weiterer Schritt für jeden anderen Knoten). Diese ist $2n-1$:

Der entartete Baum, bei dem alle Nicht-Blatt-Knoten mindestens ein Blatt als Nachfolger haben,



hat offensichtlich $2n-1$ Knoten. Jeder Baum kann unter Beibehaltung der Knotenanzahl in den entarteten Baum transformiert werden, indem man immer wieder das linkeste Blatt mit dem rechtesten Teilbaum vertauscht, dessen Wurzel mit einer 1 markiert ist.

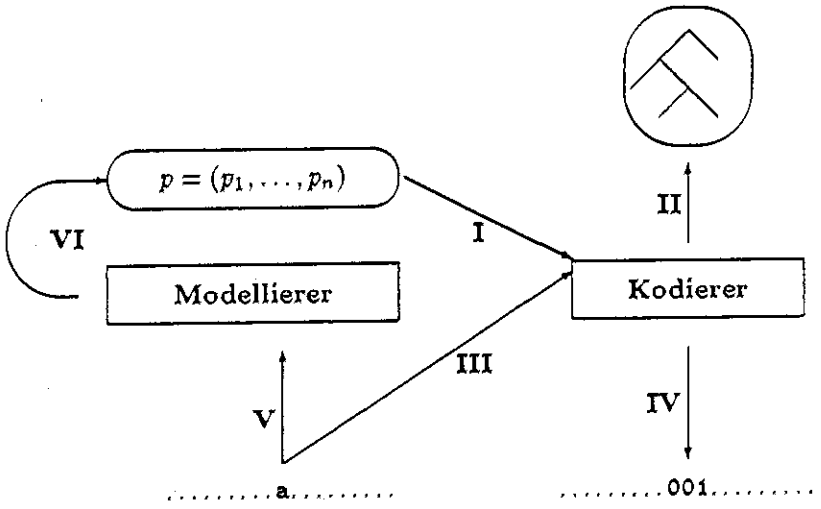
Die Anzahl der Huffman-Schritte ist also von der Größenordnung n. Der Aufwand jedes Huffman-Schrittes ist aber auch von n abhängig, da er das Einsortieren von Bäumen in die Baumliste enthält. Führt man dies naiv durch, so kommt man auf einen Aufwand der Größenordnung n für die Suche und damit n^2 für den gesamten Algorithmus. Um auf $n \log n$ zu kommen, muß man die Baumliste effizient verwalten: dies macht man sinnvollerweise mit der Struktur des sogenannten *balancierten Baumes*, die aus der Vorlesung bzw. der Literatur zum Thema „Datenstrukturen und Algorithmen" bekannt ist. Man weiß, daß in dieser Struktur das Einsortieren und das Streichen mit logarithmischem Aufwand möglich ist, woraus sich die Behauptung ergibt. ■

Der Kodierungsaufwand ebenso wie der Dekodierungsaufwand ist für jedes Zeichen gleich der Tiefe des Zeichens im Baum, im Durchschnitt also ungefähr die Entropie der Wahrscheinlichkeitsverteilung; da diese im schlimmsten Fall (der Gleichverteilung) den Wert $\mathrm{ld}\, n$ hat, haben wir

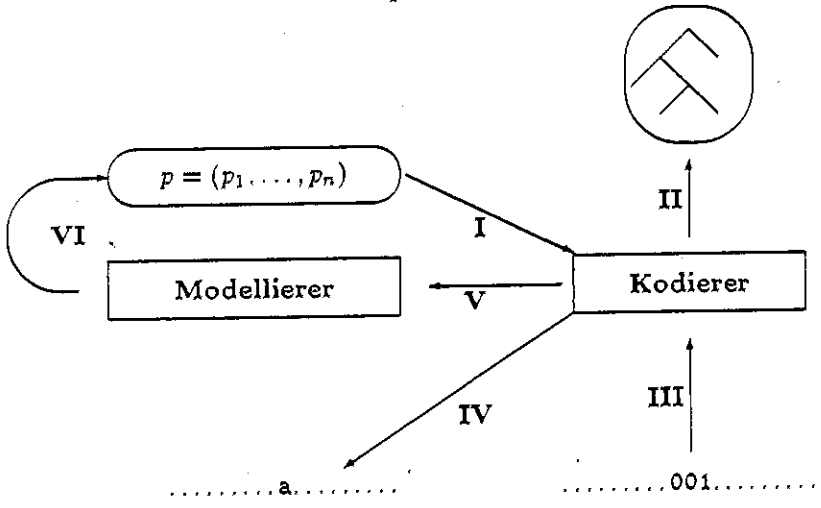
Satz 3.5 *Der Kodierungs- wie der Dekodierungsaufwand ist pro Zeichen des Originaltextes im schlimmsten Fall von der Größenordnung des Logarithmus der Zeichenzahl, pro Bit des Komprimierten Textes von konstanter Größenordnung.*
■

Die Abbildung illustriert das Zusammenwirken des Modellierungsmoduls und des Kodierungsmoduls bei der Huffman-Kompression und bei der Expansion.

Kompression:



Expansion:



In den Schritten I der Kompression und der Expansion überträgt der Modellierer dem (De-)kodierer die Wahrscheinlichkeitsverteilung p. Aus dieser konstruiert der (De-)Kodierer einen Huffman-Baum (Schritt II). Bei der Kompression liest nun der Kodierer in Schritt III das nächste Zeichen des Originaltextes (z. B. ein a) und produziert mit dem Huffman-Baum das Codewort (z. B. 110), das er an den komprimierten Text anfügt (IV). Der Dekodierer liest umgekehrt in Schritt III Bits des komprimierten Textes solange, bis er ein Zeichen erkannt hat, das er in Schritt IV an den expandierten Text anfügt. Bei der Kompression liest der Modellierer ebenfalls das a (Schritt V), bei der Expansion bekommt er es vom Dekodierer übermittelt. Aufgrund dieser neuen Information aktualisiert der Modellierer die Wahrscheinlichkeitsverteilung p (Schritte VI in beiden Fällen),

und der Zyklus kann neu beginnen.

Bei statischer Modellierung werden einige dieser Schritte trivial: es findet keine Aktualisierung der Wahrscheinlichkeitsverteilung statt, also entfallen die Schritte V und VI. Schritte I und II müßte nur ein einziges Mal in der Lebenszeit des Programmes stattfinden, nämlich bei der (einmaligen) Konstruktion des Huffman-Baumes. Sinnvollerweise konstruiert der Programmierer den Huffman-Baum aus einer einmal festgestellten Wahrscheinlichkeitsverteilung und baut ihn fest (natürlich auswechselbar) in den (De-)Kodierungsmodul ein, so daß der Modellierungsmodul im statischen Fall einfach fehlt.

Bei Modellen, die von den Häufigkeiten der Einzelzeichen ausgehen, kommt man leider nur auf erwartete Codewortlängen von 5–6 bit/Byte für Texte der deutschen (oder der englischen) Sprache, für ausführbare Programme sogar auf ungefähr 6–7. Fazit: statische Modelle lohnen sich erst bei komplizierterer Modellstruktur. Man könnte diese kompliziertere Struktur erreichen, indem man nicht Σ, sondern Σ^2 als Alphabet betrachtet, also Zeichenpaare als „Zeichen" auffaßt. Bei Textdateien kommt man hier auch tatsächlich auf eine Verbesserung, allerdings zu keiner sehr erheblichen (so etwas wie 55% statt 60% Kompression). Dafür ist der Speicherplatzaufwand, den der Huffman-Baum einnimmt, groß, etwa ein Megabyte.

Bei variablen Modellen, die von einer einzigen Wahrscheinlichkeitsverteilung für die Zeichen ausgehen, muß ein weiterer Programmodul den Text statistisch analysieren, um daraus die Wahrscheinlichkeitsverteilung zu erhalten. Dann können bei der Kompression die Schritte I und II ein für allemal stattfinden, und der Zyklus reduziert sich wieder auf die Schritte III und IV.

Bei variablen Modellen ist eine kompliziertere Struktur als die einfache Häufigkeitsverteilung normalerweise nicht sinnvoll, da die Abspeicherung des Modelles den Kompressionseffekt zunichte machen würde. Auch ist es besser, nicht die Häufigkeitsverteilung selbst, sondern bereits die daraus gewonnenen Codewörter in den komprimierten Text zu speichern, der Baum läßt sich bei der Dekodierung leicht daraus konstruieren. Die Codewörter haben variable Länge, also muß man, um sie eindeutig dekodieren zu können, die jeweilige Länge in einem Elias-Code mitspeichern. Gehen wir von 6 bit/Byte aus; die Elias-Codes sind in dieser Größenordnung etwa genauso lang wie die codierten Zahlen, so daß man für eine Liste der 256 Codewörter auf $256 \cdot 12\text{bit}$, also weniger als 0.4 Kilobyte kommt. Die Datei muß also mindestens 1.6 Kilobyte lang sein, damit sich ein Kompressionseffekt bemerkbar macht, also eine Schranke von weniger als einer Schreibmaschinenseite lesbarer Text. Noch mehr kann man sparen, wenn man beachtet, daß oft (vor allem bei lesbaren Texten) gar nicht alle möglichen Bytes verwendet werden; man kann sich auf die Kodierung der tatsächlich vorkommenden Bytes beschränken und bei der Speicherung des Modells die Menge der vorkommenden Bits als 256 bit = 32 Byte langen Bitvektor voranstellen.

Ein Anwendungsbeispiel des Huffman-Algorithmus mit variabler Modellierung ist die Run-Length-Kodierung aus dem ersten Kapitel. Wir könnten in zwei Phasen vorgehen: in der ersten Phase werden die Zahlen als Zahlen mit konstantem Platzaufwand abgespeichert, etwa als 32-Bit-Zahlen. In der zweiten Phase wird sowohl für die Zeichen als auch für die Zahlen je ein Huffman-Baum erstellt und die Zwischendatei mit diesem Huffman-Code komprimiert.

Zu adaptiven Modellen gibt es einiges mehr zu sagen. Zunächst sei darauf

hingewiesen, daß beim Huffman-Algorithmus nirgendwo benutzt wird, daß die Wahrscheinlichkeiten p_i wirklich Zahlen zwischen 0 und 1 sind: natürliche Zahlen sind völlig in Ordnung, und da die Gewichte nur addiert werden, ist es gleichgültig, ob ich sie vorher alle mit einer Konstanten multipliziere; also kann ich absolute statt relativer Häufigkeiten verwenden. Das spart Divisionen und erlaubt das Rechnen mit der viel schnelleren Festkommaarithmetik und ist daher auch bei statischen und variablen Modellen zu empfehlen. Bei adaptiven Modellen, die ja auch on-line, also potentiell unendlich lang, ablaufen sollen, werden die Häufigkeitswerte (bzw. deren Summen) irgendwann die maximale darstellbare ganze Zahl erreichen. In diesem Fall dividiert man einfach sämtliche Häufigkeitswerte durch zwei (unter Verlust des Divisionsrestes). Da der Beitrag der Zeichen, die vor der Halbierung gelesen wurden, von jetzt ab nur noch halb so groß ist wie der der neuen Zeichen, kann das auch als ein wünschenswerter Anpassungseffekt des Modells an die Gegenwart (durch „Vergessen") verstanden werden.

Schließlich das Aufwandsproblem: es erscheint unsinnig aufwendig, bei jedem gelesenen Zeichen im Kodierungsprozess und jedem geschriebenen Zeichen im Dekodierungsprozess einen neuen Huffman-Baum zu berechnen. Hier gibt es zum Glück Abhilfe, die auf Faller (1973) und Gallagher (1978) zurückgeht. Der im Folgenden beschriebene Algorithmus für die adaptive Huffman-Kodierung findet sich (in noch etwas verbesserter Form) im UNIX-Kommando „compact".

Beachte: wir berücksichtigen von jetzt ab bei Huffman-Bäume sämtliche Knotengewichte, nicht nur das der Wurzel.

Definition 3.6 Ein binärer gewichteter Baum hat die *Geschwistereigenschaft*, wenn gilt:

- das Gewicht jedes Knotens ist, falls solche existieren, gleich der Summe des Gewichtes seiner Nachfolger,
- haben zwei Knoten den selben Vorgänger (sind sie *Geschwister*), so gibt es keinen dritten Knoten, dessen Gewicht echt zwischen den Gewichten der beiden Geschwister liegt.

Die zweite Bedingung ist äquivalent dazu , daß man die Knoten in einer Folge aufsteigenden Gewichtes derart anordnen kann, daß Geschwisterknoten in dieser Folge benachbart sind.

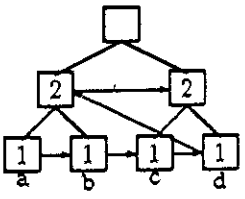
Satz 3.7 *Ein binärer gewichteter Baum ist genau dann ein Huffman-Baum, wenn er die Geschwistereigenschaft hat.*

Beweis: Sei B ein Huffman-Baum, a, b, c Knoten mit den Gewichten $g_a < g_b < g_c$, so daß a und c denselben Vorgänger v haben. Wann kann b entstanden sein? Zum Zeitpunkt der Entstehung von v waren a und c die beiden leichtesten Knoten der Liste, v kann also nicht zwischen ihnen gestanden haben; alle Knoten, die danach entstanden sind, haben mindestens so großes Gewicht wie c. Zum Zeitpunkt der Verarbeitung von b muß a also schon verarbeitet worden sein, c kann es noch nicht sein, ein Widerspruch.

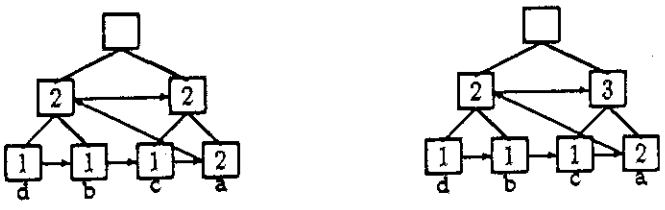
Sei umgekehrt $\mathcal{H}$ ein Baum mit der Geschwistereigenschaft. Dann können wir die Knoten in der Reihenfolge aufsteigenden Gewichtes auflisten, so daß Geschwisterknoten in der Liste benachbart sind. Nun streiche ich die beiden ersten Knoten aus der Liste und hänge sie unter ihren Vorgänger. Die beiden ersten Knoten müssen Blätter gewesen sein, also war das ein Huffman-Schritt. Ist einer der nächsten beiden Knoten kein Blatt, so müssen seine beiden Nachfolger links von ihm in der Liste gestanden haben, es muß sich also um den Vorgänger der beiden zuerst gestrichenen Listenelemente handeln. Es ist klar, daß Weiterführen dieses Vorgehens gerade den Huffman-Algorithmus nachvollzieht. ■

Für die adaptive Huffman-Kodierung erweitern wir die Baumstruktur um eine „Fädelung": wir denken uns die Knoten gemäß der Geschwistereigenschaft nach aufsteigendem Gewicht angeordnet und ordnen jedem Knoten einen Verweis auf seinen Nachfolger in dieser Liste zu. Der Algorithmus geht nun so, daß man sich bei jedem Zeichen wie im Laufe des Kodiervorganges vom Blatt zur Wurzel hocharbeitet, jetzt aber bei jedem Knoten zuerst das Gewicht um 1 erhöht, dann überprüft, ob sein Gewicht jetzt das des Nachfolgers in der Liste übersteigt, und, wenn das zutrifft, den Knoten (und seinen Teilbaum) mit dem am weitesten hinten in der Liste stehenden vertauscht, dessen Gewicht geringer ist. Die Gewichte müssen am Anfang so initialisiert sein, daß kein Blatt das Gewicht 0 hat; die Wurzel, die ohnehin immer am Ende der Liste steht, kann ignoriert werden.

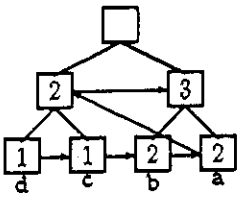
Beispiel 3.8 $\Sigma = \{a, b, c, d\}$, der Text beginnt mit „abbc". Vor Beginn des Textes wird der Baum initialisiert:

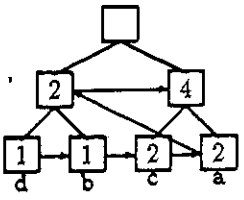


Ein a wird gelesen.

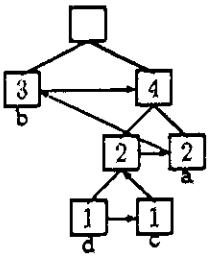


Ein b wird gelesen.

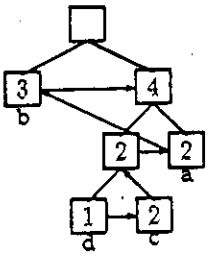


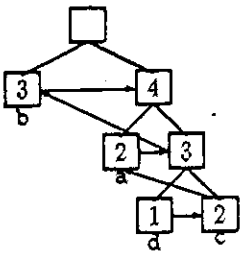


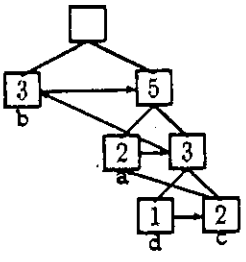
Noch ein b.



Ein c.







Der Satz garantiert mir, daß dieses Verfahren funktioniert, da ich durch das Umhängen immer die Geschwistereigenschaft und damit auch die Huffmaneigenschaft sicherstelle.

Der Aufwand beinhaltet in jedem Schritt das eventuelle Vertauschen zweier Teilbäume (was aber von der Größe dieser Teilbäume unabhängig ist), und das

Auffinden des Knoten mit gegebenem Gewicht, der möglichst weit hinten in der Liste liegt: das kann im unangenehmsten Fall $|\Sigma| - 1$ Vergleiche erfordern. Man kann diese Vergleiche aber auch überflüssig machen, indem man die Gewichte in einer eigenen verketteten Liste organisiert, so daß von jedem Knoten des Baumes ein Verweis auf sein Gewicht ausgeht, und von dem Gewicht ein Verweis auf den letzten Knoten mit diesem Gewicht. Der Aufwand zum Verwalten dieser Gewichtsliste ist wieder unabhängig von $|\Sigma|$.

Im adaptiven Fall ist das theoretische Modell des Zusammenwirkens von Modellierer und (De-)Kodierer aus den Abbildungen ebenfalls in der Praxis verzerrt, da jetzt die Wahrscheinlichkeitsverteilung und der Huffman-Baum in eine gemeinsame Datenstruktur eingefügt sind, auf die Modellierer und Kodierer kooperativ zugreifen. Die Zerlegung des Problems in Modellierung und Kodierung ist immerhin noch erkennbar: der Modellierer verwaltet die Gewichte der Blätter, der Kodierer den Rest.

Kapitel 4

Textquellen

Wir sind bisher in der Theorie davon ausgegangen, daß wir eine Funktion kennen, die jedem Text $X \in \Sigma^*$ eine Wahrscheinlichkeitsverteilung

$$p_X = (p_{X,1}, \ldots, p_{X,n})$$

zuordnet, wobei $p_{X,i}$ die Wahrscheinlichkeit ausdrückt, daß nach dem Textanfang X das Zeichen i kommt. Diesen Sachverhalt wollen wir hier genauer untersuchen und in eine Form bringen, die uns später eine vernünftige Modellierung ermöglicht.

Wir stellen uns die Textquelle als eine Maschine vor, die in regelmäßigen Abständen nach stochastischen Gesetzmäßigkeiten Zeichen produziert. Den semantischen Aspekt dieses Vorganges ignorieren wir. Ein mathematisches Modell, das solche Textquellen sinnvoll approximiert, ist die *endliche Markov-Quelle*.

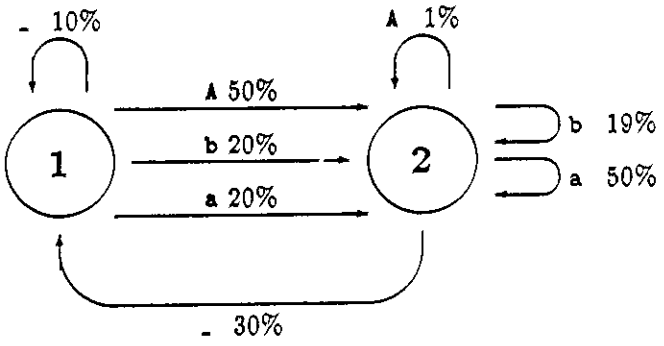
Definition 4.1 Die folgende Situation beschreibt eine endliche Markov-Quelle:

- $\Sigma = \{1, \ldots, n\}$ ist ein Alphabet;
- $Z = \{1, \ldots, m\}$ ist eine weitere endliche Menge, $|Z| \geq 1$, die Menge der *Zustände;*
- $\zeta: Z \times \Sigma \to Z$ ist eine Funktion, die *Übergangsfunktion;*
- Jedem Zustand j ist eine Wahrscheinlichkeitsverteilung $p_j = (p_{j1}, \ldots, p_{jn})$ zugeordnet, die *Zeichenwahrscheinlichkeiten im Zustand j;*

Eine Markov-Quelle läßt sich am besten als gerichteten Graph ansehen; die Zustände sind die Knoten; von jedem Knoten gehen n Pfeile aus, die mit den Zeichen des Alphabetes Σ markiert sind. Der vom Zustand j ausgehende und mit dem Zeichen i markierte Pfeil endet am Zustand $\zeta(j, i)$. Dieser Pfeil ist außerdem noch mit der Wahrscheinlichkeit p_{ji} markiert.

Beispiel 4.2 Wir betrachten im Folgenden die als Graph dargestellte Beispielquelle:

$$Z = \{1, 2\}, \quad \Sigma = \{\text{Leerzeichen}, \mathtt{A}, \mathtt{a}, \mathtt{b}\}$$



Die Textproduktion funktioniert nun so: die Quelle wird am Anfang in einen Zustand j versetzt. Mit der Wahrscheinlichkeit p_{ji} gibt die Quelle nun das Zeichen i aus und nimmt außerdem den Zustand $\zeta(j, i)$ an. Dieser Vorgang wird nun wiederholt und irgendwann abgebrochen.

Beispiel 4.3 Die Quelle aus Beispiel 4.2 könnte folgenden Text produziert haben:

Aabaaaa b a AaaAab aaaabaa a bbbbaaaa Aaab Aabaaaaaa A aaaaa Aa aaa Aab A Aaabaaaba ba ba Aa Aa bababa babaaa b ab Aa a a AabAaaaa Aaabba Aa AAbaaaa a Aaaba A abba Aa baaaa bbaaa Aaaaa baaa A baaa aa

In diesem Fall lassen sich die Zustände, in denen sich die Quelle befindet (außer am Anfang) aus den produzierten Zeichen herleiten. Das liegt daran, daß mit den gleichen Zeichen markierte Pfeile immer zu denselben Zuständen führen; im allgemeinen muß das aber nicht so sein, und dann lassen sich die Zustände nur aus den Wahrscheinlichkeiten an den Pfeilen statistisch erraten.

Die Verbindung zwischen dem zuerst in diesem Kapitel gesagten und den Markovquellen stellen die Kontextquellen her:

Definition 4.4 Eine Markovquelle heißt *Kontextquelle k-ter* Ordnung, wenn gilt:

- $Z = \Sigma^k$;
- $\zeta(i_1 \dots i_k, i) = i_2 \dots i_{k+1} i$.

Eine Kontextquelle nullter Ordnung heißt auch *unabhängig*.

Die diesem Konzept zugrundeliegende Beobachtung ist es, daß die Wahrscheinlichkeit eines Zeichens vor allem von den unmittelbar vorhergehenden Zeichen abhängt.

Beispiel 4.5 Claude Shannon hat 1948 aus Kontextquellen für die englische Sprache folgende Texte produziert:

Ordnung 0. OCRO HLI RGWR NMIELWIS EU LL NBNESEBYA TH EEI ALHENHTTPA OOBBTTVVA NAH BRL

Ordnung 1. ON IE ANTSOUTINYS ARE T INCTORE ST BE S DEAMY ACHIN D ILONASIVE TUCOOWE AT TEASONARE FUSO TIZIN ANDY TOBE SEACE CTISBE

Ordnung 2. IN NO IST LAT WHEY CRATICT FROURE BIRS GROCID PONDENOME OF DEMONSTURES OF THE REPTAGIN IS REGOACTIONA OF CRE

Man sieht, daß mit steigender Ordnung die Struktur der realen Quelle (der englischen Sprache bzw. ihrer Sprecher) zunehmend approximiert wird.

Wenn man sich bei einer Markovquelle nur für die Zustände interessiert und die Pfeile mit jeweils übereinstimmenden Anfangs- und Endpunkten zusammenfaßt, erhält man eine *Markov-Kette.*

Definition 4.6 Die folgende Situation beschreibt eine endliche Markov-Kette:

- $Z = \{1, \ldots, m\}$ ist eine weitere endliche Menge $|Z| \geq 1$, die Menge der *Zustände;*
- Jedem Zustand j ist eine Wahrscheinlichkeitsverteilung $(u_{j1}, \ldots, u_{jm})$ zugeordnet, die *Übergangswahrscheinlichkeiten im Zustand j.*

Man erhält aus einer Markovquelle also die *zugrundeliegende Markovkette,* wenn man

$$u_{jj'} := \sum_{\zeta(j,i)=j'} p_{ji}$$

setzt.

Die $m \times m$-Matrix $U = (u_{jj'})$ heißt die Übergangsmatrix der Markovkette. Sie ist *stochastisch,* daß bedeutet, daß alle Zeilensummen $\sum_{j'=1}^{m} u_{jj'}$ gleich 1 sind, und die Einträge zwischen 0 und 1 liegen.

Beispiel 4.7 Die Übergangsmatrix zu unserer Beispielquelle 4.2 ist

$$\begin{pmatrix} 0.1 & .0.9 \\ 0.3 & 0.7 \end{pmatrix}$$

Wir nehmen für dieses Kapitel eine fest gewählte Markovquelle $\mathcal{Q}$ an.

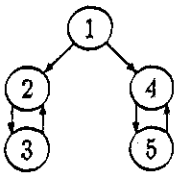
Wir machen eine Anforderung an die Quelle, die plausibel ist, wenn wir uns für ihr langfristiges Verhalten interessieren.

Definition 4.8 Eine Markovquelle (bzw. Kette) heißt *ergodisch,* wenn man von jedem Zustand aus jeden anderen (eventuell in mehreren Schritten) erreichen kann.

Wir wollen annehmen, daß $\mathcal{Q}$ ergodisch ist.

Warum ist diese Annahme plausibel? Eine Markovkette kann auf zwei Weisen gegen die Forderung der Ergodizität verstoßen: einmal kann es flüchtige Zustandsmengen geben, das heißt solche, die nach einmaligem Verlassen nicht wieder auftreten können; außerdem kann es vorkommen, daß manche Zustandsmengen nicht wieder verlassen werden können, wenn sie einmal erreicht wurden („wesentliche" Zustandsmengen). Die flüchtigen Zustandsmengen sind für die langfristige Entwicklung uninteressant, und die wesentlichen Zustandsmengen, in die das System nach Verlassen der flüchtigen Zustände fällt, können einzeln betrachtet, und das Verhalten des Gesamtsystems dann durch aus dem der einzelnen Teile durch Kombination verstanden werden. Eine minimale wesentliche Zustandsmenge ist sicher ergodisch.

Beispiel 4.9 $\{1\}$ ist flüchtig, $\{2,3\}$ und $\{4,5\}$ sind wesentlich.



Man sieht leicht ein: befindet sich das System zu einer Zeit mit Wahrscheinlichkeit ψ_j im Zustand j, und ist ψ der Zeilenvektor $(\psi_1, \ldots, \psi_m)$, so enthält der Zeilenvektor ψU die Wahrscheinlichkeiten der Zustände, nachdem das System einen Schritt gelaufen ist.

Definition 4.10 Eine Wahrscheinlichkeitsverteilung $\psi = (\psi_1, \ldots, \psi_m)$ heißt *stationär*, wenn $\psi U = \psi$.

Satz 4.11 (Ergodensatz) *$\mathcal{Q}$ hat nur eine stationäre Zustandsverteilung π; dabei ist π_j interpretierbar als die Wahrscheinlichkeit, daß sich die Quelle im Zustand j befindet, vorausgesetzt, ihre Laufzeit ist unbekannt.*

Die Matrizenfolge

$$\frac{1}{k}\sum_{t=0}^{k-1} U^t$$

konvergiert gegen die Matrix

$$\begin{pmatrix} \pi \\ \vdots \\ \pi \end{pmatrix}.$$

Beweis: Dies ist kein vollständiger Beweis, sondern eine Plausibilitätsbetrachtung, die mit stochastischem oder matrizentheoretischem Handwerkszeug zu einem richtigen Beweis gemacht werden kann Der Beweis ist nicht schwierig, würde hier aber zu weit führen. (Siehe die Bücher von Kemeny und Snell oder von Fritz, Huppert und Willems).

Wenn wir die Markovkette im Zustand j starten und k mal laufen lassen, dann ist die j-te Zeile der Matrix U^k die Wahrscheinlichkeitsverteilung für die Zustände nach diesem Experiment.

Lassen wir noch einmal die Markovkette im Zustand j starten und höchstens k mal ablaufen. Jetzt sei es uns aber unbekannt, wie oft die Quelle tatsächlich gelaufen ist, und wir nehmen an, daß wir darüber gar keine Informationen besitzen (außer der über die Höchstzahl k). Dann sind alle Laufanzahlen t zwischen 0 und k gleichwahrscheinlich, und somit ist die Wahrscheinlichkeitsverteilung für den Zustand nach dem Experiment gerade das arithmetische Mittel der Wahrscheinlichkeitsverteilungen für die Zustände nach t-maligem Ablaufen ($t = 0, \ldots, k-1$), also die j-te Zeile von $\frac{1}{k}\sum_{t=0}^{k-1} U^t$.

Wissen wir nun überhaupt nicht, wie oft die Quelle gelaufen ist, so muß die Wahrscheinlichkeitsverteilung das arithmetische Mittel über die Wahrscheinlichkeitsverteilungen nach t-maligem Ablauf für alle t sein, also die j-te Zeile von $P := \lim_{k\to\infty} \frac{1}{k}\sum_{t=0}^{k} U^t$. Die Existenz dieses Grenzwertes, die aus der Betrachtung der Jordanschen Normalform einer stochastischen Matrix geschlossen werden kann, entnehmen wir den angegebenen Büchern.

Es ist ziemlich plausibel, daß $PU = P$ gilt, denn mit steigender Anzahl der Versuche ändert noch eine weiteres Ausführung nicht mehr viel. Das aber heißt, daß die Zeilen von P stationäre Zustandsverteilungen sind. (Beweis: Siehe Literatur)

Nun ist die Markovkette ergodisch, und daher wird es nach zunehmender Anzahl von Abläufen immer unerheblicher, in welchem Zustand gestartet wurde; starte ich in j' statt in j dann komme ich jedenfalls irgendwann einmal auch zu j, und ich kann stattdessen dieses j als Startzustand annehmen (das verkleinert nur das k, was auf lange Sicht irrelevant wird.) Fazit: alle Zeilen von P sind gleich, $P = \begin{pmatrix} \pi \\ \vdots \\ \pi \end{pmatrix}$.

Sei nun ψ eine stationäre Zustandsverteilung. Dann $\psi P = \psi$, also ist ψ ein Eigenvektor von P (bezüglich Rechtsmultiplikation) zum Eigenwert 1. Da P den Rang 1 hat, ist 0 ein $m-1$-facher Eigenwert, also ist der Eigenraum zum Eigenwert 1 höchstens eindimensional und kann daher nur eine Wahrscheinlichkeitsverteilung enthalten. Diese muß π sein. ■

Wir nehmen an, daß sich unser System Q in dieser stationären Wahrscheinlichkeitsverteilung befindet.

Beispiel 4.12 In unserer Beispielquelle konvergiert sogar die Folge der Matrizen U^k selbst (und natürlich auch die Folge der Mittelwerte $\frac{1}{k}\sum_{t=0}^{k-1} U^t$), und zwar gegen

$$\begin{pmatrix} 0.25 & 0.75 \\ 0.25 & 0.75 \end{pmatrix}$$

Die Quelle befindet sich also mit Wahrscheinlichkeit $\frac{1}{4}$ im Zustand 1 und mit Wahrscheinlichkeit $\frac{3}{4}$ im Zustand 2.

Wir haben schon die Entropie einer Wahrscheinlichkeitsverteilung p als

$$H(p) = \sum_{i=1}^{n} p_i \mathrm{ld}\, p_i$$

kennengelernt.

Definition 4.13 Die *Entropie von* $\mathcal{Q}$ ist der Erwartungswert der Entropien der Wahrscheinlichkeitsverteilungen der Zeichen in den verschiedenen Zuständen, also

$$\begin{aligned} H(\mathcal{Q}) &= \sum_{j=1}^{m} \pi_j H(p_j) \\ &= \sum_{j=1}^{m} \sum_{i=1}^{n} \pi_j p_{ji} \mathrm{ld}\, p_{ji}. \end{aligned}$$

Ist $\mathcal{Q}$ unabhängig, so ist die Entropie von Q einfach die Entropie der Wahrscheinlichkeitsverteilung der Zeichen im einzigen möglichen Zustand.

Beispiel 4.14 Wir berechnen die Entropie unserer Beispielquelle.

Die Wahrscheinlichkeitsverteilung für Zustand 1 hat die Entropie $0.1\mathrm{ld}\,0.1 + 0.5\mathrm{ld}\,0.5 + 2 \cdot 0.2\mathrm{ld}\,0.2\mathrm{bit} = 1.76\mathrm{bit}$.

Die Wahrscheinlichkeitsverteilung für Zustand 2 hat die Entropie $0.3\mathrm{ld}\,0.3 + 0.01\mathrm{ld}\,0.01 + 0.5\mathrm{ld}\,0.5 + 0.19\mathrm{ld}\,0.19\mathrm{bit} = 1.53\mathrm{bit}$.

Als Entropie unserer Quelle ergibt sich $0.25 \cdot 1.76 + 0.75 \cdot 1.53\mathrm{bit} = 1.59\mathrm{bit}$.

Definition 4.15 Ist $A \in \Sigma^k$, so ist die *Wahrscheinlichkeit von* A die Wahrscheinlichkeit von A unter allen Texten der Länge k, also für $A = a_1 \cdots a_k$

$$P(A) = P_k(A) = \sum_{j=1}^{m} \pi_j p_{ja_1} p_{\zeta(j,a_1)a_2} p_{\zeta(\zeta(j,a_1),a_2)a_3} \cdots p_{\zeta(\zeta(\cdots\zeta(j,a_1),a_2)\cdots,a_{n-1})a_n}.$$

Ist $f: \Sigma^* \to \Phi^*$ eine Kodierungsfunktion ($\Phi = \{0,1\}$), so ist die *mittlere Zeichenlänge von f für Texte der Länge k* definiert als

$$\frac{1}{k} \sum_{A \in \Sigma^k} P(A)|f(A)|\,\mathrm{bit}.$$

(Die Maßeinheit bit deutet darauf hin, daß $|\Phi| = 2$).

Als Umformulierung der Betrachtungen im zweiten Kapitel erhalten wir:

Korollar 4.16 *Ist $\mathcal{Q}$ unabhängig und f eine Zeichenkodierung, so ist die mittlere Zeichenlänge von f für Texte der Länge k (wobei k beliebig vorgegeben sei) mindestens so groß wie die Entropie der Zeichenwahrscheinlichkeitsverteilung von $\mathcal{Q}$.*

Wir wollen uns von den Voraussetzungen der Unabhängigkeit von $\mathcal{Q}$ und der Zeichenkodierungseigenschaft von f frei machen. Das gelingt mit dem *Quellkodierungssatz* von Shannon. Wir brauchen dazu noch eine technische, aber plausible Bedingung an die Kodierungsfunktion, die von der Präfix-Bedingung für Codes abgeleitet ist:

Definition 4.17 Die Kodierungsfunktion $f: \Sigma \to \Phi$ erfüllt die *Präfixbedingung*, wenn für zwei Originaltexte A und B gleicher Länge der kodierte Text $f(A)$ nie Präfix des kodierten Textes $f(B)$ ist.

Satz 4.18 (Quellkodierungssatz) *Sei $f: \Sigma^* \to \Phi^*$ eine Kodierungsfunktion mit Präfixbedingung, und sei die ergodische Markovquelle $\mathcal{Q}$ in der stationären Zustandsverteilung π. Dann ist für jede natürliche Zahl k die mittlere Zeichenlänge von f für Texte der Länge k mindestens so groß wie die Entropie $H_{\mathcal{Q}}$ der Quelle $\mathcal{Q}$.*

Wir werden sehen, daß diese untere Schranke sogar eine untere Grenze ist. Der Beweis des Quellkodierungssatzes verläuft über eine Folge von Hilfssätzen, die aber alle für sich interessant sind.

Lemma 4.19 *Sei ℓ die mittlere Zeichenlänge von f für Texte der Länge k, und G_k die Entropie der Wahrscheinlichkeitsverteilung P_k der Texte der Länge k. Dann $\ell \geq G_k$.*

Beweis: Fasse Σ^k als Alphabet und $f_k := f|_{\Sigma^k}$ als Code auf. Nach der Präfixbedingung der Kodierungsfunktion f ist f_k ein Präfixcode, und nach Kapitel 2 gilt die Behauptung. ■

Das nächste Lemma sagt, daß die Anzahl der Bits pro Zeichen, die ich für einen Text mindestens brauche, mit steigender Textlänge fast sicher gegen die Entropie der Textquelle konvergiert.

Lemma 4.20 *Für jedes $\varepsilon > 0$ und jedes $\delta > 0$ gibt es eine natürliche Zahl k_0 und für jedes $k \geq k_0$ eine Teilmenge $M \subseteq \Sigma^k$ der Wahrscheinlichkeit mindestens $1 - \varepsilon$, so daß für alle $A \in M$ gilt:*

$$\left| \frac{-\mathrm{ld}\, P(A)}{k} - H_{\mathcal{Q}} \right| < \delta$$

Beweis: (Skizze) Wir betrachten einen k-fachen Ablauf der Quelle. Die relative Anzahl der Situationen, in denen die Quelle im Zustand j das Zeichen i ausgegeben hat, geht nach dem Ergodensatz für wachsendes k asymptotisch gegen $\pi_j p_{ji}$. Man errechnet aus dieser Konvergenzeigenschaft, daß ich ein k_0 finden kann, so daß für alle $k \geq k_0$ die Wahrscheinlichkeit, daß diese Häufigkeit um mehr als δ von $\pi_j p_{ji}$ abweicht, kleiner als ε ist. Damit ist (für $k \geq k_0$) die Wahrscheinlichkeit jedes Ablaufes (bis auf eine Menge der Wahrscheinlichkeit $< \varepsilon$) konstant gleich

$$P = \prod_{j,i} p_{ji}^{(\pi_j p_{ji} \pm \delta)k}.$$

Dieses P ist auch die Wahrscheinlichkeit eines Textes der Länge k (nämlich $\sum_{j=1}^{m} \pi_j P$). Wir schließen

$$\frac{\mathrm{ld}\, p}{k} = \sum_{j,i} (\pi_j p_{ji} \pm \delta) \mathrm{ld}\, p_{ji}.$$

■

Lemma 4.21 *Es gilt*

$$-\sum_{X \in \Sigma^{k-1}} P(X) \mathrm{ld}\, P(X) - \sum_{i=1}^{n} p_i \mathrm{ld}\, p_i \geq -\sum_{Xi \in \Sigma^k} P(Xi) \mathrm{ld}\, P(Xi),$$

also die Entropien der Texte der Länge k und die der Zeichen zusammen sind mindestens so groß wie die Entropie der Texte der Länge k.

Beweis: Logarithmenrechnung. ■

Lemma 4.22 *Die Folge der G_k, wobei*

$$G_k = -\frac{1}{k} \sum_{A \in \Sigma^k} P(A) \mathrm{ld}\, P(A),$$

ist monoton fallend und konvergiert gegen die Entropie von Q.

Beweis: Nach Lemma 4.20 ist der Grenzwert, falls er existiert, gleich der Entropie.

Definiere $p_X(i)$ als die bedingte Wahrscheinlichkeit, daß das Zeichen i auf den Text X folgt, also $\frac{P(Xi)}{P(X)}$. Setze

$$F_k = -\sum_{X \in \Sigma^{k-1}, i \in \Sigma} P(Xi) \mathrm{ld}\, p_X(i).$$

Nun $F_{k-1} \geq F_k$ nach 4.21,

$$F_k = kG_k - (k-1)G_{k-1},$$

also $G_k = \frac{1}{k} \sum_{k'=1}^{k} F_{k'}$. Die Folge der Mittelwerte einer monoton fallenden Folge ist selbst monoton fallend; die Konvergenz folgt aus der Beschränktheit. ■

Der Quellkodierungssatz folgt nun durch Zusammensetzen dieser Lemmata.

Wir wollen die Ergebnisse dieses Kapitels im Hinblick auf die Praxis zusammenfassen:

Korollar 4.23 *Sei f eine Kodierungsfunktion, die die Texte der Quelle Q im Sinne dieser Schranken optimal komprimiert, d. h. in der die genannten unteren Schranken erreicht werden.*

- *Ist X ein Text der Länge k, so hat $f(X)$ ungefähr die Länge $k \cdot \mathrm{ld}\, P(X)$.*

- *Texte der Länge k werden im Durchschnitt mit G_k bit pro Zeichen dargestellt, wobei G_k die in 4.19 definierte Größe ist. Der Durchschnitt ist hierbei für Texte der Quelle Q zu bilden.*
- *Für wachsendes k strebt die Anzahl der Bits pro Zeichen gegen die Entropie der Quelle.*
- *Die Entropie ist die durchschnittliche Länge der komprimierten Texte aus Q.*

Wir werden im nächsten Kapitel mit der sogenannten arithmetischen Kodierung ein Kodierungsverfahren angeben, das in Verbindung mit einem Modell für die durch dieses Modell modellierte Quelle die angegebene Optimalitätseigenschaft hat. Ein nur theoretisch interessantes (off-line) Verfahren mit den gleichen Eigenschaften ist es, zuerst die Länge k des Textes festzustellen, dann die Texte der Länge k als Alphabet aufzufassen und für dieses „Alphabet" eine Fano-Shannon-Code c zu bestimmen; der Text X wird dann einfach durch das Codewort $c(X)$ kodiert; die Länge ist dann $\lceil \mathrm{ld}\, P(X) \rceil$.

Beispiel 4.24 Wir stellen fest, daß die Texte unserer Beispielquelle nicht besser als mit 1.6 bit pro Zeichen dargestellt werden können, vorausgesetzt, wir haben ein Verfahren, das für *sämtliche* Texte der Quelle optimale Ergebnisse liefert. Das ist im Vergleich mit den 2 bit pro Zeichen, die man bei einem Alphabet von 4 Zeichen ohne weitere Kompression erreichen könnte, kein besonders gutes Ergebnis.

Normalerweise kennt man die Quelle nicht, die die Texte erzeugt, welche man komprimieren will; man versucht also, diese Quelle zu erraten, das heißt, sich ein Modell dieser Quelle zu bilden. Die durchschnittliche Kompressionsleistung eines Verfahrens kann nicht besser sein als die Entropie der Quelle, die die Texte tatsächlich erzeugt; darüberhinaus sinkt sie um so mehr, je weniger das Modell mit der realen Quelle statistisch übereinstimmt. Für unsere Beispielquelle gibt es also nicht die Möglichkeit, durch ein irgendwie geschickt gewähltes Modell die Kompression unter die genannten vier fünftel zu bringen. Zum Glück sind die Entropien der natürlichen Textquellen wesentlich geringer; Shannon hat zum Beispiel Versuche angestellt, in denen er Personen aus gegebenen Textanfängen die folgenden Zeichen raten ließ und daraus die Entropie der englischen Sprache mit 0.6–1.3 bit geschätzt.

Wir werden im übernächsten Kapitel mehr über die Modellierung von Textquellen sehen.

Kapitel 5

Arithmetische Kodierung

Der Optimalitätssatz für den Huffman-Code sagt aus, daß man Huffman-Codes nicht *durch eine bessere Zeichenkodierung* ersetzen kann. Das arithmetische Kodierungsverfahren komprimiert nun besser als der Huffman-Code, ist folglich keine Zeichenkodierung.

Angenommen, wir wollen den Text $A = a_1 \cdots a_k \in \Sigma^*$ komprimieren. Aufgrund unseres Modelles haben wir Wahrscheinlichkeitsverteilungen p_X, $X \in \Sigma^*$, $p_X = (p_{X1}, \ldots, p_{Xn})$.

Die Wahrscheinlichkeitsverteilungen, die in dieser Vorlesung vorkommen sind in der Terminologie der Stochastik *Dichtefunktionen;* die zur Wahrscheinlichkeitsverteilung $p = (p_1, \ldots, p_n)$ gehörige *Verteilungsfunktion* ist das n-tupel $v = (v_0, \ldots, v_n)$ mit $v_i = \sum_{i'=1}^{i} p_{i'}$. Insbesondere gelten $v_n = 1$ und $v_0 = 0$; dann haben wir $v_i - v_{i-1} = p_i$ für alle $i = 1, \ldots, n$. So haben wir für unseren Fall die Verteilungsfunktionen v_X, $X \in \Sigma^*$. Wir wollen den Verweis auf den Textanfang X bei den Wahrscheinlichkeitsverteilungen und Verteilungsfunktionen weglassen. Mit p bzw. v bezeichnen wir also immer gerade die zur Zeit „gültige" Verteilung bzw. Verteilungsfunktion.

Nun ordnen wir dem Text A ein Intervall $[b, t)$ reeller Zahlen zu, das zwischen 0 und 1 liegt; das machen wir rekursiv durch

$$\begin{aligned} b_1 &= v_{a_1-1}, & t_1 &= v_{a_1}, \\ b_{j+1} &= b_j + v_{a_{j+1}-1}(b_j - a_j), & b_j &+ v_{a_{j+1}}(b_j - a_j), \\ b &= b_k & t &= t_k \end{aligned}$$

In Worten: die erste Verteilungsfunktion definiert eine Einteilung des Einheitsintervalles in n Teilintervalle; wir wählen uns das a_1-te Teilintervall; in dieses hinein projizieren wir die Einteilung in Teilintervalle, die die zweite Verteilungsfunktion liefert und wählen uns darin das a_2-te Teilintervall, und so weiter; wir erhalten eine Folge von Intervallen, von denen jedes das nächste enthält und nehmen das letzte als das gesuchte Intervall.

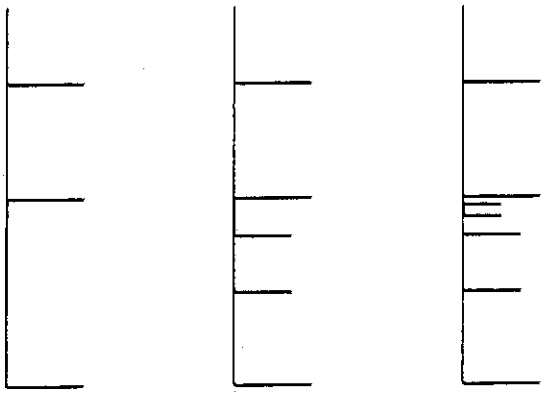
Beispiel 5.1

$$\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}\},$$

$$A = \mathtt{acb},$$
$$p \text{ ist konstant } \{p_{\mathtt{a}} = 0.5, p_{\mathtt{b}} = 0.3, p_{\mathtt{c}} = 0.2\}$$
$$v_0 = 0, \quad v_{\mathtt{a}} = 0.5, \quad v_{\mathtt{b}} = 0.8, \quad v_{\mathtt{c}} = 1$$

Es ergibt sich als Folge der Intervalle:

$$\mathtt{a} \rightsquigarrow [0, 0.5) \quad \mathtt{ac} \rightsquigarrow [0.4, 0.5) \quad \mathtt{acb} \rightsquigarrow [0.45, 0.48).$$



Aus der Kenntnis des Intervalles bzw. seiner beiden Endpunkte läßt sich der Originaltext rekonstruieren. Ist die Länge des Textes bekannt, so reicht zur Identifikation des Originaltextes schon die Angabe einer einzigen Zahl aus diesem Intervall. Die Idee des Kodierungsverfahrens ist es nun, eine solche Zahl zu suchen, die in Binärdarstellung möglichst wenige Stellen hinter dem Komma hat und diese Binärdarstellung als kodierten Text zu übertragen. Die Länge des Textes könnte man z.B. am Anfang als Elias-Code voranschicken; im Hinblick auf On-line-Anwendungen ist es aber praktischer, das Alphabet Σ um ein weiteres Zeichen zu erweitern, das „Textende" bedeutet. Dieses wird am Textende übertragen, und der Expander weiß bei Erkennen dieses Zeichens, daß er fertig ist. Man muß dazu natürlich die Wahrscheinlichkeitsverteilung entsprechend anpassen, indem man dem Textende-Symbol z.B. bei einer konstanten Wahrscheinlichkeitsverteilung den Kehrwert der erwarteten Textlänge zuordnet.

Beispiel 5.2 Wir modifizieren das Beispiel von eben entsprechend, wobei wir von einer durchschnittlichen Textlänge von 5 ausgehen:

$$\Sigma = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \langle\mathtt{EOF}\rangle\}$$
$$A = \mathtt{acb}\langle\mathtt{EOF}\rangle,$$
$$p_1 = p_2 = p_3 = p = \{p_{\mathtt{a}} = 0.4, p_{\mathtt{b}} = 0.24, p_{\mathtt{c}} = 0.16, p_{\langle\mathtt{EOF}\rangle} = 0.2\}$$
$$v_0 = 0, \; v_{\mathtt{a}} = 0.4, \; v_{\mathtt{b}} = 0.64, \; v_{\mathtt{c}} = 0.8 \; v_{\langle\mathtt{EOF}\rangle} = 1$$

Es ergibt sich als Folge der Intervalle:

$$\begin{aligned} \mathtt{a} &\rightsquigarrow [0, 0.4) \\ \mathtt{ac} &\rightsquigarrow [0.256, 0.32) \\ \mathtt{acb} &\rightsquigarrow [0.2816, 0.29696) \\ \mathtt{acb}\langle\mathtt{EOF}\rangle &\rightsquigarrow [0.293888, 0.29696). \end{aligned}$$

Der untere Wert beginnt binär 0.01001011..., der obere 0.01001100...; die Binärzahl 0.010011 liegt also in unserem Intervall, daher kann 010011 als komprimierter Text abgeschickt werden.

Bevor wir uns um die praktische Umsetzung dieser Idee kümmern wollen, erst die Rechtfertigung, daß sich diese Arbeit lohnt:

Satz 5.3 *Ein Text A der Länge k wird arithmetisch mit maximal $\lceil \mathrm{ld}\,(P(A))\rceil$ bit dargestellt, wobei $P(A)$ die im letzten Kapitel beschriebene Wahrscheinlichkeit des Textes unter den Texten der Länge k ist. Der Erwartungswert für die mittlere Zeichenlänge bei Texten der Länge k ist bei arithmetischer Kodierung bis auf Rundungsfehler die im Lemma 4.19 angegebene Schranke G_k, die sich aus der modellierten Textquelle ergibt. Für große k konvergiert dieser Erwartungswert gegen die Entropie der modellierten Textquelle – vorausgesetzt, die Quelle ist eine ergodische Markovquelle mit stationärer Zustandsverteilung, und das Modell repräsentiert die Quelle exakt.*

Beweis: Die Größe des Intervalles nach Verarbeitung des Textes A der Länge k ist gerade die Wahrscheinlichkeit $P = P(A)$ dieses Textes. Nun enthält ein Intervall der Länge $P(A)$ eine Dualzahl mit nur $\lceil \mathrm{ld}\,P(A)\rceil$ Stellen hinter dem Komma, woraus die Ungleichung

$$\ell_k \le \frac{1}{k} \sum_{X \in \Sigma^k} P(A) \lceil \mathrm{ld}\,P(X)\rceil$$

folgt (ℓ_k die gesuchte mittlere Zeichenlänge.)

Nach dem Lemma 4.19 gilt andererseits

$$\ell_k \ge \frac{1}{k} \sum_{X \in \Sigma} P(X) \mathrm{ld}\,P(X),$$

was die erste Behauptung ergibt. Die zweite folgt dann aus dem nachfolgenden Lemma des vorigen Kapitels. ■

Beachte, daß die arithmetische Kompression nicht optimal ist: optimal wäre die Huffman-Kodierung der Wahrscheinlichkeitsverteilung der Texte der Länge k. Diese läßt sich jedoch nicht auf sinnvolle Weise praktisch bestimmen, während wir einen schnellen Algorithmus für die arithmetische Kodierung angeben können.

Kodierung und Dekodierung sind Varianten einer Operation aus einer auf den ersten Blick ganz anderen Problemstellung. Nehmen wir an, es gilt $n = 10$, und die Wahrscheinlichkeiten p_i sind immer konstant $\frac{1}{10}$. Wir nennen die Elemente von $\Sigma = \{0, \ldots, 9\}$ jetzt einmal „Ziffern". Dann ist die arithmetische Kodierung nichts anderes als die Umrechnung des Originaltextes, den wir als Dezimalzahl zwischen 0 und 1 auffassen, in das Binärsystem, und die Dekodierung die Umrechnung vom Binärsystem in das Dezimalsystem.

Dies gilt entsprechend natürlich auch für andere n als 10: liegt Gleichverteilung vor, so ist die Kodierung die Umrechnung vom n-adischen System in das

binäre, die Dekodierung umgekehrt. Jetzt kann man Zahlsysteme zur Darstellung von Zahlen zwischen 0 und 1 noch verallgemeinern: statt das Einheitsintervall in gleiche Teile aufzuteilen, kann ich es in verschieden große Teile aufteilen und diese Einteilung entsprechend immer verfeinern; dabei kann man für jeden Verfeinerungsschritt eine neue Einteilung wählen; es ergibt sich eine eindeutige Zifferndarstellung der Zahlen. Arithmetische Kodierung und Dekodierung sind also Spezialfälle der Umrechnung vom „v-adischen" Zahlsystems in das „v'-adische":

$[b,t) := [0,1)$
$[b',t'] := [0,1)$
loop
  repeat
    read(i) *(nächste v-adische Ziffer)*
    $[b,t) := b + (t-b)[v_{i-1}, v_i)$
  until $\exists i'\colon [b,t) \subseteq [v'_{i'-1}, v'_{i'})$
  write(i') *(nächste v'-adische Ziffer)*
  $[b',t') := b' + (t'-b')[v'_{i'-1}, v'_{i'})$
  nach Belieben die vier Zahlen b', b, t, t' gemeinsam im Intervall [0,1) verschieben
  die vier Zahlen mit einem konstanten Faktor multiplizieren, aber so, daß sie im Einheitsintervall bleiben
end loop

Die letzten vier Zeilen beschreiben den Vorgang des *Hochskalierens*, der dazu dienen soll, die Zahlen für die Festkommaarithmetik in einer vernünftigen Größenordnung zu halten.

In unserem Fall wird also zur Kodierung vom v-adischen System in das dyadische umgerechnet, bei der Dekodierung vom dyadischen in das v-adische. Es ist plausibel, daß in der praktischen Umsetzung in dieser Konstellation die beiden Aktionen sich deutlich unterscheiden werden: der Rechner unterstützt eben die dyadische Arithmetik und nicht die v-adische.

Die Werte p_{ji} sind in der Praxis natürlich keine reellen Zahlen, sondern rationale Zahlen, die nur endlich viele Binärstellen hinter dem Komma haben – wir müssen sie ja immerhin im Computer repräsentieren. Weiterhin wird man versuchen, nicht eine aufwendige und langsame beliebig genaue Arithmetik für sie zu benutzen, sondern die Arithmetik ganzer Zahlen. Man repräsentiert die p_i also als natürliche Zahlen, und an die Stelle der 1 tritt eine entsprechend große Zahl E. In Wirklichkeit ist es so: an die Stelle der 1 treten *zwei* natürliche Zahlen E und V_n, wobei E die Rolle dyadischen 1, V_n die Rolle der v-adischen 1 übernimmt. Da wir jetzt diskret rechnen, arbeiten wir mit abgeschlossenen statt halboffenen Intervallen. In Wirklichkeit repräsentiert also $E+1$ die dyadische 1, während E die höchste darstellbare Zahl bezeichnet; die V-adische 1 entspricht der Zahl $V_n + 1$. Im einzelnen sind also gegeben:

- Eine natürliche Zahl E, die mit halb so vielen Bits dargestellt werden kann, wie unsere Rechengenauigkeit zuläßt. Diese Bedingung ermöglicht exakte Durchführung von Multiplikationen. (Rechnen wir mit 32-Bit-Arithmetik, so wäre E z.B. $2^{16} - 1$).

- Eine sich aus dem Modell mit jedem gelesenen/geschriebenen Zeichen des Originaltextes neu herleitende Liste natürlicher Zahlen V_i ($i \in \Sigma \cup \{0\}$). Wir verlangen $V_0 = 0$, $V_i - V_{i-1} \geq 1$ (i¿0), $V_n \leq \frac{E}{4}$. Die Begründung der zunächst ganz unplausiblen letzten Bedingung folgt noch.

Kodierung Wir bezeichnen das Intervall, das sich aus dem Originaltext ergibt, mit $[B,T]$. Wir haben zunächst also $B = 0$, $T = V_n$ und wählen das Intervall $[V_{i_1-1}, V_{i_1}]$ aus, so daß jetzt $B = V_{i_1-1}$, $T = V_{i_1}$ wird. Im nächsten Schritt ergibt sich dann (mit neuer Verteilungsfunktion V) das Intervall $[B+p_{i_1} \cdot V_{i_2}-1, B+p_{i_1} \cdot V_{i_2}]$, was nicht von ganzen Zahlen begrenzt sein wird. Wir versuchen also, durch Hochskalieren Ganzzahligkeit zu erreichen; die Zahlen V_i' müssen aber mitskaliert werden, und daß ist nicht beliebig lang ohne Erhöhung der Stellenzahl möglich. Wir müssen also runden. Zunächst der (vorläufige) Algorithmus:

```
B := 0
T := E
loop
   read(i)
   R := T - B + 1          (Länge des Intervalles [B,T])
   B := B + ⌊RV_{i-1} / V_n⌋
   T := B + ⌊RV_i / V_n⌋ - 1
   loop
      if T < ⌈E/2⌉
         write(0)
      else if B ≥ ⌈E/2⌉
         write(1)
         B := B - ⌈E/2⌉
         T := T - ⌈E/2⌉ + 1
      else
         exit
      end if
      B:=2B
      T:=2T+1
   end loop
end loop
```

Wir haben also bei der Berechnung der neuen Intervallgrenzen von $[B,T]$ die Divisionsreste weggelassen; das bedeutet, daß wir nicht die eigentlich vorgesehene Verteilungsfunktion V benutzen, sondern die „gerundete" Version. Das führt zu korrekten Ergebnissen, vorausgesetzt, der Expander führt die gleichen Rundungen durch wie der Kompressor; aus dem Rundungsfehler ergibt sich lediglich eine geringere Kompressionsleistung. Zum Glück pflanzt sich der Rundungsfehler nicht fort, da die nächste Verteilungsfunktion nicht von der bisherigen Verteilungsfunktion selbst, sondern nur vom bisher gelesenen Text abhängt.

Allerdings müssen wir darauf achten, daß die Intervallgrößen alle mindestens 1 betragen, da wir ein Zeichen mit einem Intervall der Länge 0 nicht kodieren

können. Ein Teil dieser Aufgabe wird im angegebenen Algorithmus durch Hochskalieren mit dem Faktor zwei erledigt; es kann aber passieren, daß das Intervall $[B,T]$ weder in der unteren Hälfte noch in der oberen Hälfte liegt, und daß dieser Zustand mehrere Male hintereinander auftritt. Wir müssen den Algorithmus noch modifizieren.

Wegen der Bedingung $V_n \leq \frac{E}{4}$, liegt jedes Zeichenintervall $[V_{i-1}, V_i]$ immer in einem der Intervalle $[0, \lceil \frac{E}{2} \rceil]$, $[\lceil \frac{E}{2} \rceil, E]$ oder $[\lceil \frac{E}{4} \rceil, \lceil \frac{3E}{4} \rceil]$ der Länge $\lceil \frac{E}{2} \rceil$. Die ersten beiden Fälle stehen bereits im vorläufigen Algorithmus: sind wir in der unteren Hälfte, so geben wir eine 0 aus und multiplizieren mit 2; sind wir in der oberen Hälfte, so geben wir 1 aus, ziehen $\frac{E}{2}$ ab und multiplizieren mit 2. In der „mittleren Hälfte" machen wir uns eine Beobachtung zunutze: Die Dualdarstellungen von Zahlen in diesem Intervall fangen jedenfalls mit 01 oder 10 an, je nachdem, ob die Zahl kleiner oder größer als $\frac{E}{2}$ ist. Noch mehr:

- Liegt eine Zahl x im Intervall vom Radius 2^{-k} um den Mittelpunkt es Einheitsintervalles $\frac{1}{2}$, so beginnt die Darstellung von Zahlen daraus mit einer 0, gefolgt von k Ziffern 1 oder umgekehrt mit einer 1, gefolgt von k Ziffern 0.

Sind wir also in der mittleren Hälfte, skalieren wir um einen Faktor 2 hoch, geben aber kein Bit aus; stattdessen merken wir uns, wie oft hintereinander wir in der mittleren Hälfte waren. Sind wir dann nach k Schritten nicht mehr in der Mitte, so wissen wir, ob wir größer oder kleiner als $\frac{E}{2}$ sind und können dann die 1, gefolgt von k Nullen oder die 0, gefolgt von k Einsen ausgeben. Der extrem seltene Fall, daß k die höchste darstellbare ganze Zahl übersteigt, braucht uns nicht weiter zu beschäftigen. Der entstehende Algorithmus ist jedenfalls:

$k := 0$
$B := 0$
$T := E$
loop
  read(i)
  $R := T - B + B := B + \left\lfloor \frac{RV_{i-1}}{V_n} \right\rfloor$
  $T := B + \left\lfloor \frac{RV_i}{V_n} \right\rfloor - 1$
  loop
    if $T < \lceil \frac{E}{2} \rceil$
      write($0,\underbrace{1,\ldots,1}_{k}$)
      $k := 0$
    else if $B \geq \lceil \frac{E}{2} \rceil$
      write($1,\underbrace{0,\ldots,0}_{k}$)
      $k := 0$
    else if $B \geq \lceil \frac{E}{4} \rceil$ **and** $T < \lceil \frac{3E}{4} \rceil$
      $k := k + 1$
      $B := B - \lceil \frac{E}{4} \rceil$
      $T := T - \lceil \frac{E}{4} \rceil$
    else

```
            exit
        end if
        B := 2B
        T := 2T + 1
    end loop
end loop
```

Nach jedem gelesenen Zeichen wird also das Intervall so lange hochskaliert, bis es länger als $\frac{E}{4}$, also auch länger als V_n ist; das garantiert, daß die Division durch V_n, die dann nach dem Lesen des nächsten Zeichens durchgeführt wird, nur Intervalle der Mindestlänge 1 hinterläßt.

Nachdem der Originaltext fertig ist, und wir auch noch das EOF-Zeichen übertragen haben, müssen wir zum Ende kommen. Wir wissen, daß wenigstens das zweite oder das dritte Viertel unseres „Einheitsintervalles" $[0, E]$ nach dem Hochskalieren im Intervall $[B, T]$ enthalten sind. Wir schicken also im Fall $T \geq \frac{3E}{4}$ noch 10, im Fall $B \leq \frac{E}{4}$ die Bitfolge 01 ab, gefolgt von unendlich vielen (gedachten) Nullen.

Dekodierung Jetzt lesen wir 2-adische Zahlen und wollen sie in v-adische umrechnen. Die Größen E, V_n etc. definieren wir wie bei der Kodierung. Das sich aus der Bitfolge ergebene „2-adische Intervall", das fortlaufend halbiert wird, halten wir von vorneherein auf einer Größe von $\frac{1}{2}$. Wir können die Intervallgrenzen so gar nicht mit ganzen Zahlen angeben, müssen das aber auch nicht: das Intervall wird immer von der Art $[W, W + \frac{1}{2})$ mit $W \in \mathbb{N}$ sein und ist dann durch Angabe der Zahl W allein eindeutig beschrieben. Auch die Überprüfung, ob dieses Intervall in einem anderen (jetzt ganzzahligen) Intervall enthalten ist, reduziert sich zu der Überprüfung, ob die Zahl W in diesem anderen Intervall enthalten ist.

Der Dekodierungsalgorithmus ist:

```
...
W aus den ersten Bits aufbauen
B := 0
T := 0
loop
    R := T − B + 1
    suche i, so daß B + ⌊RV_{i−1}/V_n⌋ ≤ W < B + ⌊RV_i/V_n⌋
    write(i)
    T := B + ⌊RV_i/V_n⌋ − 1
    B := B + ⌊RV_{i−1}/V_n⌋
    loop
        if T < ⌈E/2⌉
            nichts
        else if B ≥ ⌈E/2⌉
            von W, T, B jeweils ⌈E/2⌉ subtrahieren
        else if B ≥ ⌈E/4⌉ and T < ⌈3E/4⌉
            von W, T, B jeweils ⌈E/4⌉ subtrahieren
```

```
        else
            exit loop
        end if
    end loop
    B := 2B
    T := 2T + 1
    W := 2W + nächstesBit
end loop
...
```

Die Analyse dieses Algorithmus auf Korrektheit überlasse ich zur Übung.

Die arithmetische Kodierung ist nicht on-line, da wir uns beliebig lang in der mittleren Hälfte des Intervalles befinden können, und keine Bits ausgegeben werden. Dennoch ist sie so gut wie on-line: wir können keine Konstante k angeben, hinter der die Dekodierung maximal hinterherhinkt, aber wir haben eine statistische Aussage, die das Erscheinen des nächsten Bits immer wahrscheinlicher macht.

Die Verwaltung der Verteilungsfunktion ist etwas aufwendiger als die der Häufigkeiten selbst. Zunächst bietet es sich an, sie rückwärts zu speichern, also in einem **array**, bei dem V_n den Index 0, und V_0 den Index n hat; dies deshalb, weil auf V_n in jedem Schritt mehrmals zugegriffen wird und der Zugriff auf die 0-te Feldkomponente schneller ist als auf die n-te. Wir belassen die Indizierung jetzt einmal so, wie wir sie gewöhnt sind.

Im Fall der adaptiven Modellierung wird jede Häufigkeit mit 1 initialisiert, also V_i mit i. Wird das Zeichen i gelesen, so müssen alle V_j für $j := 1, \ldots, n$ um eins erhöht werden. (Bei Erreichen des Höchstwertes das Halbieren aller V_i nicht vergessen!) Man kann den Zeitaufwand hierfür noch verringern, indem man dafür sorgt, daß die Zeichen immer in der Reihenfolge aufsteigender Häufigkeit angeordnet sind. Ist das Zeichen i nämlich besonders häufig, dann steht es weit oben in der Liste, und nur wenige Zeichen sind noch häufiger, so daß deren V-Wert erhöht werden muß.

Wir schließen mit Testberichten; die Tests wurden von J. Cleary, R. Neal und I. Witten durchgeführt. Zum Test standen adaptive Huffman-Kodierung (in Form des UNIX-compact-Kommandos), eine „leicht optimierte C-Version" und eine optimierte Assemblerversion der arithmetischen Kodierung in Verbindung mit einem unabhängigen adaptiven Modell mit den 256 Bytes als Alphabet. Die Optimierungstricks in der C-Version bestanden im wesentlichen in der bereits erwähnten Rückwärtsverwaltung der Verteilungsfunktion; ferner wurden die Bit-IO-Routinen als Makros statt als Prozeduren geschrieben, Registervariablen benutzt, Multiplikationen mit 2 durch Additionen und **array**-Indizierung durch Pointerarithmetik ersetzt. Die Maschine, auf der der Test lief, war eine VAX-11/780.

Als Testdateien wurden Dateien von je 100000 byte verwendet, und zwar eine Datei mit englischem Text, ein C-Quellprogramm, ein VAX-Objektprogramm, eine Datei der Form „**abcdefg...xyzabcde...**", und eine Datei der Form

aaaaabaaaacaaaabaaaacaaaabaaaac...

Ergebnisse:

Datei	Huffman			Arith. C			Arith. Ass.	
	Kpr byte	Kpr μS	Exp μS	Kpr byte	Kpr μS	Exp μS	Kpr. μS	Exp μS
Text	57781	550	414	57718	214	262	104	135
C	63731	596	441	62991	230	288	109	111
EXE	76950	822	606	73546	313	406	158	241
abcde	60127	598	411	59292	223	277	105	145
aaab	16257	215	132	12092	143	170	63	81

Die Tatsache, daß der theoretische Aufwand der arithmetischen Kodierung wegen der ungünstigen adaptiven Modellverwaltung mit den Verteilungsfunktionen im schlechtesten Fall $\mathcal{O}(n^2)$ beträgt, ist für die Praxis nicht so wichtig, da die angegebene selbstoptimierende Struktur viel bessere Durchschnittsergebnisse liefert. Zum Speicherplatz ist zu sagen, daß die Verwaltung des Modells bei der arithmetischen Kodierung ähnlichen zusätzlichen Bedarf anzeigt wie bei der adaptiven Huffman-Kodierung. Es gibt aber keinen Huffman-Baum und folglich auch keinen Platzbedarf zu seiner Speicherung.

Noch zwei Punkte, in denen der Huffman-Ansatz völlig versagt: ist ein Zeichen häufiger als 50%, so sinkt der Zweierlogarithmus seiner Wahrscheinlichkeit unter ein Bit; dieses Zeichen kann aber vom Huffman-Code nur mit mindestens einem Bit kodiert werden, was zu um so größeren Verlusten führt, je häufiger dieses Zeichen ist. Hat das Alphabet Σ überhaupt nur zwei Zeichen (z. B. in einer Schwarz-Weiß-Graphik), so erreicht der Huffman-Code einen Kompressionsfaktor von exakt 1, er komprimiert also überhaupt nicht! Man behilft sich mit einer Variante des Run-Length-Verfahrens mit Huffman-Nachkodierung; die arithmetische Kodierung löst das Problem mühelos und kann mit komplizierteren Modellen diesen Run-Length-Huffman-Ansatz leicht schlagen.

Kapitel 6

Statistische Modelle

Die Modelle, die von den in dieser Vorlesung behandelten erwähnten Kodierungen benötigt werden, sind *statistische Modelle*, also solche, die für jedes Zeichen in Abhängigkeit vom bisherigen Text eine Wahrscheinlichkeitsverteilung (bzw. eine Verteilungsfunktion) liefern. Die Standardversion eines statistischen Modelles emuliert eine Markovquelle. Die Zustände, die Pfeile und die Beschriftung der Pfeile mit Zeichen nennen wir die *Modellstruktur*, die Wahrscheinlichkeitsverteilungen der Zeichen, die jedem Zustand zugeordnet sind, die *Modellparameter*. Die Modellparameter organisiert man am besten adaptiv. Die Begründung dafür liegt darin, daß statische Modelle, wenn sie gut sein sollen, nur für sehr spezialisierte Texte geeignet sind; dann allerdings erreichen sie Kompressionen, die anders nicht zu erreichen sind. Der Grad der Spezialisierung könnte etwa bedeuten, daß ein statischer Kompressor mit einem Modell, das auf CD-ROM vorliegt, weil es auf ein kleineres Speichermedium nicht paßt, nur englische, in TEX geschriebene Texte aus dem Themenbereich der Datenkompression signifikant komprimiert, diese aber mit weit unter einem Bit pro Zeichen. Variable Modelle können naturgemäß nur klein sein und sind deshalb adaptiven Modellen meistens unterlegen. Die Häufigkeitswerte bei adaptiver Parameterverwaltung pflegen sich recht schnell gut an die Realität anzupassen, etwa nach halb so vielen Besuchen des Zustandes wie Zeichen im Alphabet sind.

Das zentrale Problem bei der adaptiven Verwaltung der statistischen Modellparameter ist das *Häufigkeit-Null-Problem:* wie verhält sich das Modell, wenn ihm in einem Zustand ein Zeichen zum ersten Mal begegnet? Mit einem Häufigkeitswert von Null können die Kodierer nichts anfangen; die Theorie würde verlangen, dieses Zeichen mit $-\mathrm{ld}\,0 = \infty$ bit zu kodieren. Der einfachste Ansatz, (wir haben ihn auch schon in den Beispielen benutzt), besteht darin, alle Häufigkeiten mit Eins zu initialisieren. Man versucht, nach Möglichkeit im jeweiligen Spezialfall hierzu eine Alternative zu finden, denn die mit Eins initialisierten Verteilungen passen sich viel langsamer an.

Ein allgemeiner Ansatz bei adaptiven Modellen ist es, mehrere verschiedene Modelle gegeneinander antreten zu lassen, typischerweise ein einfaches, schnell anpassendes und kompliziertere, langsamer anpassende, und jeweils das Modell zu benutzen, das für das nächste zu lesende Zeichen die beste Kompression (= kleinste Entropie der Verteilung) voraussagt. Auch hier gibt es in speziellen

Situationen bessere Lösungen.

Wir kommen nun zu der Frage, die eigentlich am nächsten liegt: was für eine Modellstruktur benutze ich denn nun? Der offensichtlichste Antwortvorschlag heißt „Kontextmodelle" und wird später ausführlich behandelt. Zuerst betrachten wir andere Ansätze zu Markovmodellen und Verallgemeinerungen.

Syntaxmodelle

Die Struktur einer Markovquelle ähnelt sehr derjenigen eines endlichen Automaten. Ein endlicher Automat kann beschrieben werden als eine Markovquelle, bei der die Wahrscheinlichkeitswerte p_{ji} auch den Wert Null haben dürfen (die genauen Werte der von Null verschiedenen Wahrscheinlichkeiten werden ignoriert), und wo darüberhinaus noch ein Zustand als Anfangszustand sowie ein anderer als Endzustand ausgezeichnet sind. Ein Text heißt *syntaktisch korrekt*, wenn er bei Start im Anfangszustand eine nichtverschwindende Wahrscheinlichkeit hat und außerdem im Endzustand endet.

Man benutzt endliche Automaten, um einfache Grammatiken zu beschreiben; zum Beispiel beschreibt das Beispiel aus dem Kapitel über Textquellen, wenn man die Wahrscheinlichkeit „1%" in „0%" ändert sowie den Zustand 1 zum Anfangszustand und zum Endzustand macht, eine Grammatik, die besagt, daß Großbuchstaben (des angegebenen Alphabetes) nur am Wortanfang stehen dürfen.

Das liefert auch schon einen Ansatz zum Finden einer Modellstruktur: erwartet man, daß die Texte, die man komprimieren will, einer Syntax genügen, die durch einen endlichen Automaten beschrieben werden kann, dann geht man so vor: man ignoriert den Endzustand; dann übernimmt man die Modellstruktur von dem Automaten. Da dort die mit Null gewichteten Pfeile im Normalfall gar nicht angegeben werden, muß man sich für diese (die „Syntaxfehler") sinnvolle Zielpunkte aussuchen. Alle Häufigkeitsverteilungen werden mit Eins vorbesetzt; am Anfang setzt man das Markovmodell in den Anfangszustand des Automaten und fängt an zu komprimieren. So könnte man etwa das obengenannte Beispiel durch Vergrößerung des Alphabetes zu einem Modell zur Kompression sprachlicher Texte machen; die Modellstruktur dürfte bei deutschen Texten gegenüber dem unabhängigen Modell verbesserte Kompression liefern, bei englischen wird die Kompression schon näher am unabhängigen Modell liegen, und bei Modula-II-Programmen mit Großbuchstaben mitten im Wort keine Verbesserung gegenüber dem unabhängigen Modell gegeben sein.

Ernstzunehmende Grammatiken (z. B. bei Programmiersprachen) beschreibt man nicht durch endliche Automaten sondern durch sogenannte Produktionsregeln; damit werden z. B. auch Klammerstrukturen beliebiger Tiefe ermöglicht. Entsprechend ist die weiterentwickelte Syntaxkompression nicht mit Markovmodellen möglich. Ich will nicht auf die Einzelheiten eingehen; wer sich mit Compilerbau ein wenig auskennt, sieht leicht ein, wie man einen Compiler für eine Programmiersprache in einen Kompressor für Quellprogramme dieser Programmiersprache verwandelt: der Parser und auch der Lexical Analyser ordnen den erkannten syntaktischen bzw. lexikalischen Strukturen adaptiv Häufigkeiten zu, und der Code Generator wird ersetzt durch einen arithmetischen Kodierer. Das

Problem, was man mit einem Text macht, der nicht der Syntax entspricht, ist ebenfalls aus dem Compilerbau bekannt. Dort werden Fehlermeldungen ausgegeben, und dann versucht der Compiler, irgendwo „wieder aufzusetzen." Genau das (bis auf die Fehlermeldungen) wird der syntaktische Kompressor ebenfalls machen.

Der Haupt-Anwendungsbereich syntaktischer Kompression (durchaus auch mit Markovmodellen) dürfte bei Datenbanken liegen, wo die verschiedenen Felder jeweils eine bestimmte Syntax haben, deren Richtigkeit durch Eingabemasken bereits garantiert ist.

DMC-Modelle

Wenn sich von der Struktur der zu komprimierenden Texte kein spezielles Markovmodell anbietet, sind die DMC-Modelle („Dynamic Markov Coding") ein Ansatz: hier werden nicht nur die Modellparameter, sondern auch die Modellstruktur adaptiv verwaltet. Die Idee ist folgende: wird ein Zustand besonders oft über einen bestimmten Pfeil erreicht, so bietet es sich an, das Konzept „Zustand x wird über Pfeil y erreicht" in einen eigenen Zustand zu verwandeln. Man geht also wie folgt vor:

- Angenommen, der Zustand j wird vom Zustand j' aus über das Zeichen i erreicht. Der Häufigkeitswert $a = P_{j'i}$ sei größer als ein vorher festgelegter Schwellenwert S. Außerdem sei die Gesamthäufigkeit b, mit der der Zustand j über einen anderen Zugang erreicht wurde, größer als ein zweiter vorher festgelegter Schwellenwert T.

- Kreiere einen neuen Zustand j''.

- Ändere die Übergangsfunktion: $\zeta(j', i)$ soll jetzt j'' und nicht mehr j sein, d. h. der Pfeil, dessen Häufigkeit den Schwellenwert S überschritten hat, zeigt jetzt auf den neuen und nicht mehr auf den alten Zustand.

- Die Pfeile, die von j'' ausgehen, werden von j kopiert, also $\zeta(j'', i) := \zeta(j, i)$ für jedes i.

- Die Häufigkeiten P_{ji} werdem auf die Zustände j und j'' gemäß der Anzahl der Zugangswege aufgeteilt: für jedes i wird $P_{j''i} =: \frac{a}{a+b} P_{ji}$ und $P_{ji} := \frac{b}{a+b} P_{ji}$ gesetzt, wobei noch auf ganzzahlige Werte gerundet wird, dabei allerdings Häufigkeiten Null durch 1 ersetzt werden.

Beispiel 6.1 $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $S = 2$, $T = 3$, der Text ist `abbabbabb`...Wir beobachten die Entwicklung des dynamischen Markovmodelles:

Anfang

a: 1 bit

b: 1.6 bit

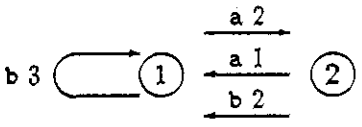
b: 1 bit

a: 1.3 bit

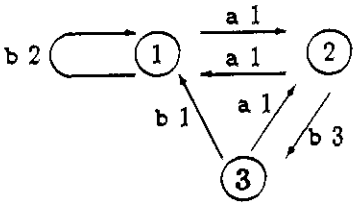
b: 1 bit

b: 0.8 bit

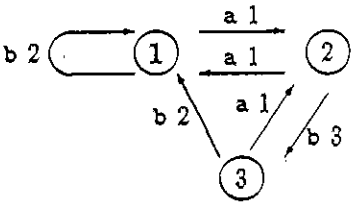
a: 1.4 bit



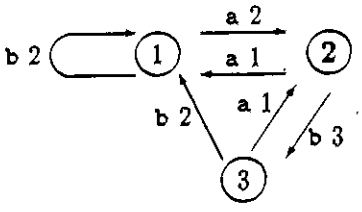
b: 0.6 bit



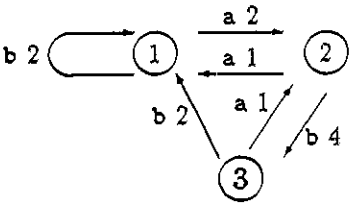
b: 1 bit



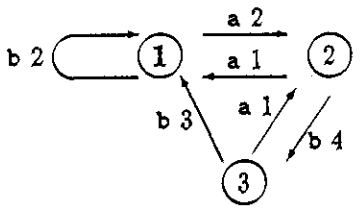
a: 1.4 bit



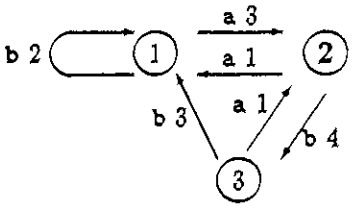
b: 0.4 bit



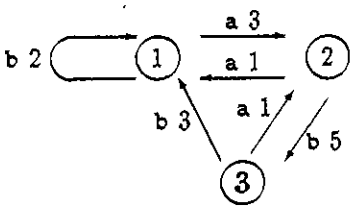
b: 0.6 bit



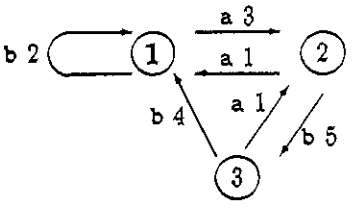
a: 1 bit



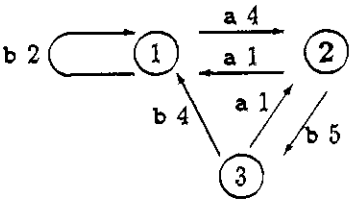
b: 0.3 bit



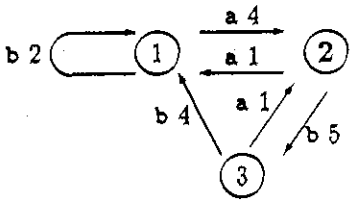
b: 0.4 bit



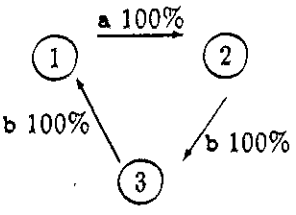
a: 0.7 bit



b: 0.3 bit



In Zukunft finden keine Zellteilungen mehr statt, und die Anzahl der Bit pro Zeichen geht gegen Null. Die Struktur der Textquelle, der (entarteten) Markovquelle



(Pfeile mit Wahrscheinlichkeit Null sind nicht eingezeichnet) ist erreicht; diese hat Entropie 0 (wir können $0 \operatorname{ld} 0 = \lim_{x \to 0} x \operatorname{ld} x = 0$ setzen), woraus sich die immense Kompressibilität des Textes erklärt.

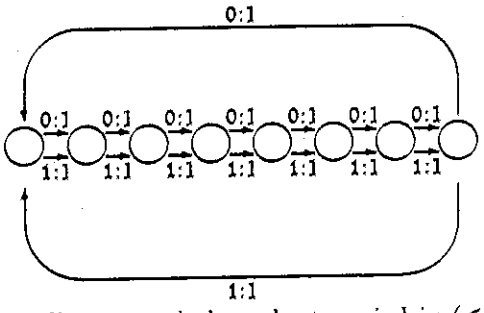
Zur Rechtfertigung der Vorgehensweise folgende Punkte:

- Die Zellteilung an sich fördert eine zunehmende Differenzierung der Struktur.

- Die Einteilung eines Zustandes in einen mit genau einem Zugang und einen, der die anderen Zugänge hat, unterstützt eindeutig erreichbare Zustände. Diese sind erstrebenswert, da bei einem auf mehreren Wegen erreichbaren Zustand zu erwarten ist, daß die Wahrscheinlichkeitsverteilung eine Überlagerung von mehreren profilierteren (und damit niederentropischen) ist.
- Die beiden Schwellenwerte verhindern die Entstehung von selten erreichten Zuständen, deren Häufigkeiten nur langsam adaptieren würden.
- Die Aufteilung der Häufigkeitswerte nach Zugangshäufigkeit der beiden neuen Zustände ergibt für jeden der beiden Zustände die am besten statistisch gerechtfertigte Häufigkeitsverteilung mit der zur Zeit gegebenen Information.

Aus Versuchen mit dem DMC-Verfahren hat man folgende praktischen Empfehlungen geschlossen:

- Man verwendet am besten ein zweielementiges Alphabet Σ; sonst wird der Zeitaufwand bei der Zellteilung zu hoch; außerdem lassen sich bei der arithmetischen Kodierung die Verteilungsfunktionen hier besonders einfach verwalten. Dadurch wird man nicht auf die Kompression von schwarzweiß-Graphiken eingeschränkt: der Text wird eben bitweise statt byteweise gelesen. Bei 8-Bit-Bytes empfiehlt sich eine Initialisierungsstruktur der Form



- Die beiden Schwellenwerte sind am besten niedrig (≤ 5). Das reflektiert die Faustregel

> Besser passende Modellstruktur bei ungenauer Statistik als umgekehrt.

Implementation von Markovmodellen

Für die Implementation eines Markovmodelles bieten sich sofort zwei Ansätze an:

Zeigerversion: **type**
Zustandsverweis = ↑Knoteninhalt
Zustand = **array**[Sigma] **of record**
P: integer

```
            ζ: Zustandsverweis
        end Zustand
    end type
    var
        Markovquelle: Zustandsverweis
    end var

Indexversion: type
        Zustandsindex = 1 . . MaxZustände
        Zustand = array[Sigma] of record
            P: integer
            ζ: Zustandsindex
        end Zustand
    end type
    var
        Markovquelle: array[Zustandsindex] of Zustand
        m: Zustandsindex
    end var
```

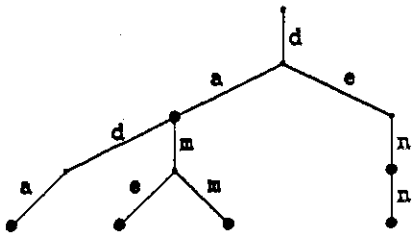
Die Indexversion ist im Vergleich mit der Zeigerversion platzsparender und ähnlich schnell. Die Erzeugung eines neuen Zustandes geht einfach über die Erhöhung der Zahl m, die die Anzahl der Zustände bezeichnet; der neue Zustand ist dann Zustand[m]. Was in der Indexversion nicht gut lösbar ist, ist das Löschen von Zuständen; das aber ist weder in den statisch strukturierten noch in den DMC-Modellen erforderlich.

Digitale Suchbäume

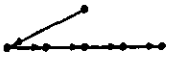
Die naive Implementation eines Kontextmodelles k-ter Ordnung könnte schon bei $k = 3$ den Speicherplatz sprengen; die Erfahrung empfiehlt aber k in der Größenordnung von fünf oder sechs. Wir wollen also nicht alle potentiellen Kontexte der Ordnung 6 abspeichern, sondern nur die tatsächlich vorkommenden. Dazu betrachten wir zunächst die Datenstruktur des *digitalen Suchbaums.* (Die englische Bezeichnung ist „trie", ein aus „retrieval" und „tree" abgeleitetes Kunstwort, das ursprünglich genauso wie „tree" ausgesprochen werden sollte).

Ein digitaler Suchbaum ist eine Struktur, die der des binären Suchbaumes aus dem zweiten Kapitel sehr ähnelt. Die binären Suchbäume dienten zur dynamischen Verwaltung von Texten aus Φ^* (in diesem Fall Codewörtern), die digitalen Suchbäume dienen zur dynamischen Verwaltung von Texten aus Σ^* (in unserem Fall Kontexten). Ein digitaler Suchbaum ist ein Baum, bei dem jeder Knoten maximal n Nachfolger hat ($n = |\Sigma|$), die mit Zeichen aus Σ markiert sind. An jedem Knoten können sich weitere Informationen, Verweise etc. befinden. Ein Text aus Σ^* ist in einem digitalen Suchbaum *repräsentiert,* wenn man von der Wurzel aus wie beim Dekodieren im binären Suchbaum Zeichen für Zeichen sich im Baum nach unten arbeitet, und wenn der Knoten, bei dem man endet, irgendwie als „Endknoten" markiert ist (es muß sich nicht um ein Blatt handeln).

Beispiel 6.2 Ein digitaler Suchbaum, in dem die Wörter **dada**, **dame**, **damm**, **da**, **dann**, **denn**, **den** repräsentiert sind.



Für einen digitalen Suchbaum bieten sich zwei Implementationsmethoden an: einmal kann man wie in unserem Beispiel beim binären Suchbaum vorgehen und die Nachfolger als **array of** Knoten repräsentieren; die nicht vorhandenen Nachfolger werden dann mit **nil** besetzt; die andere Methode organisiert sie als verkettete Liste wie in der Abbildung.



Die erste Methode erhöht die Zugriffsgeschwindigkeit, die zweite spart Speicherplatz. Weitere Methoden werden wir noch im nächsten Kapitel betrachten.

Kontextmodelle

Nun einige Prinzipien zur effektiven adaptiven Verwaltung eines Kontextmodelles k-ter Ordnung:

- Verwalte nicht nur die Kontextquelle der Ordnung k, sondern auch alle kleineren bis hinunter zur Ordnung 0.
- Führe „in Gedanken" einen zusätzlichen „Kontext der Ordnung -1" ein, in dem alle Zeichen gleich wahrscheinlich sind.
- Vergrößere das Alphabet Σ um ein spezielles Zeichen, das *unsichtbare Zeichen*. (Wir wollen es hier einmal mit ? bezeichnen.)
- Wenn nun das Zeichen i nach den k Vorgängerzeichen $i_1, \ldots, i_k$ gelesen wird, überprüfe, ob dieses Zeichen im Kontext k-ter Ordnung $i_1 \cdots i_k$ bereits vorkommt. Ist dies der Fall, übertrage dem arithmetischen Kodierer die Werte V_{i-1} und V_i und erhöhe die Zählung des Zeichens i um 1;
- kommt das Zeichen nicht vor (die *Ausweichsituation*), so führe das Zeichen mit Häufigkeit 1 in den Kontext der k-ten Ordnung ein; übermittle dem Kodierer, daß er erst das unsichtbare Zeichen kodieren soll sowie die Werte $V_?$ und $V_{?-1}$ und suche das Zeichen i im Kontext $i_2 \cdots i_k$ der Ordnung $k-1$;

- dies wird, wenn notwendig, bis hinunter zum Kontext der Ordnung Null durchgeführt. Im Kontext der Ordnung -1 ist das Zeichen i nun sicher vorhanden;

- am Anfang des Textes muß man leicht modifiziert vorgehen: beim Lesen des j-ten Zeichens für $j < k+1$ beginnt die Suche im Kontext $i_1 \cdots i_{j-1}$, in dem das Zeichen aber auch noch nicht aufgetreten ist, so daß der Effekt die Initialisierung der Kontexte $i_1 \cdots i_j$, $i_2 \cdots i_j$ usw. ist.

Beachte, daß der Zähler für das Zeichen i im Kontext $i_j \cdots i_n$ für $j > 1$ nicht die Häufigkeit des Auftretens von i in diesem Kontext bezeichnet, sondern die Häufigkeit, in der dieses Zeichen in diesem Kontext aufgetreten ist, aber in keinem längeren.

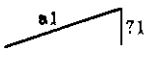
Die Implementation kann über einen digitalen Suchbaum der Tiefe k erfolgen. Die Kontexte der Ordnung j sind dann jeweils die Mengen der Knoten der Tiefe j. Die Repräsentation der Funktion ζ durch Zeiger ist an sich überflüssig, da der Zustand $\zeta(i_1 \cdots i_j, i) = i_2 \cdots i_j i$ auch in j Schritten von der Wurzel aus gefunden werden kann. Will man Zeit sparen, so ist eine Verzeigerung von $i_1 \cdots i_j$ nach $i_2 \cdots i_j$ sinnvoller, da dieser Zustand ohnehin im Falle der Ausweichsituation gebraucht wird. Der Zustand $\zeta(i_1 \cdots i_j, i) = i_2 \cdots i_j i$ kann dann in zwei Schritten erreicht werden.

(Eine Alternative bieten Hash-Tabellen, die wir noch betrachten werden.)

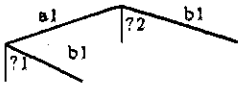
Beispiel 6.3 Σ seien alle Bytes, $k = 2$, der Text sei

abaababaababaab...

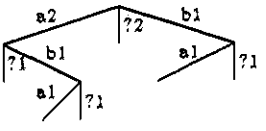
a: 8 bit



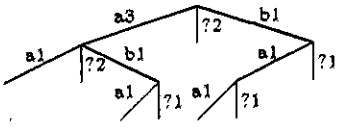
b: 1+8 bit



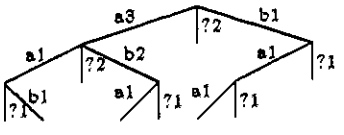
a: 2 bit



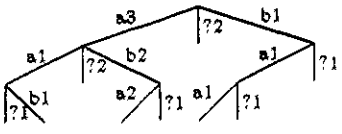
a: 1+1.3 bit



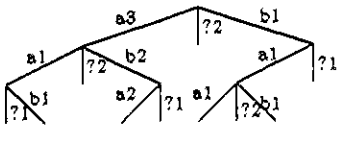
b: 2 bit



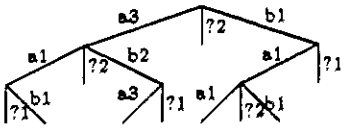
a: 1 bit



b: 1+1.3 bit



a: 0.6 bit



Von nun an wird die Struktur nicht mehr weiterwachsen. Die Kompression geht nicht gegen Null Bit pro Zeichen, da der Kontext „ba" in gleicher Wahrscheinlichkeit mit a oder mit b fortgesetzt wird. In allen anderen Kontexten allerdings werden die Zeichen mit gegen Null strebender Bitzahl dargestellt.

Vergleiche

Es konnte gezeigt werden, daß DMC-Modelle durch Kontextmodelle emuliert werden können, also nicht die volle Kapazität der Markovmodellierung ausnutzen. Dennoch bieten diese einen interessanten alternativen Ansatz zur stochastischen Modellierung.

Kontextmodelle haben nur ein kurzes Gedächtnis. Deshalb können sie z. B. nur geringen Vorteil daraus ziehen, daß nach „(" die Wahrscheinlichkeit für „)" erhöht ist, solange das Zeichen „)" noch nicht erschienen ist. So etwas kann man in allgemeineren Markovmodellen (statisch) kodieren, aber nur bis zu einer fest vorgegebenen Klammerungstiefe. Für beliebig tiefe Klammerung braucht man ein Syntaxmodell.

Einige Testergebnisse von T. Bell, J. Cleary und I. Witten:

Zum Vergleich standen ein adaptives Kontextmodell der Maximalordnung 3, eine verbesserte Variante der hier beschriebenen Art, Kontexte zu verwalten (PPMC, Moffat 1988); ein Kontextmodell, das Wörter als Elemente des Alphabetes verwendet (WORD, Moffat 1987; ein „Wort" ist dabei eine Folge der Höchstlänge 20 aus entweder nur alphabetischen Zeichen oder nur nichtalphabetischen Zeichen), das DMC-Verfahren von Cormack und Horspool 1987 mit den Schwellenwerten $S = 1$ und $T = 8$, und zum Vergleich die adaptive Huffman-Kompression in Verbindung mit einem unabhängigen Modell (das UNIX-Kommando COMPact). Als Texte wurden ein unformatierter belletristischer englischer Text (buch), ein in TROFF geschriebener entlischer technischer Text (tech), 32-Bit-Fließkommazahlen (zahl), ein VAX-EXE-File (exe), ein Schwarzweißbild (bild), ein C-Programm (c), ein LISP-Programm (lisp), ein Pascal-Programm (pas). Die Texte lagen im Bereich von von 40 bis 600 Kilobyte Länge; die Kompression in bit pro Zeichen:

Text	COMP	PPMC	WORD	DMC
buch	4.56	2.48	2.70	2.51
tech	4.83	2.26	2.51	2.25
zahl	5.70	4.78	5.06	4.77
exe	6.06	3.76	4.50	4.56
bild	1.66	1.09	0.89	0.94
c	5.26	2.49	2.71	2.98
lisp	4.81	1.90	1.90	2.17
pas	4.92	1.84	1.92	2.22

Kapitel 7

Wörterbuchmodelle

Die sogenannten Wörterbuchmodelle wurden 1978 von Lempel und Ziv eingeführt. Zusammen mit den im nachfolgenden Kapitel zu behandelnden Fenstermodellen (Lempel und Ziv 1977) werden sie oft unter der etwas irreführenden Bezeichnung „das Lempel-Ziv-Verfahren" geführt.

Diese Modelle sind nicht statistisch in dem Sinne, daß keine expliziten Häufigkeitszählungen stattfinden. Die Grundidee ist es, zwischen die Alphabete Σ und Φ noch ein weiteres, aus Zeichenketten in Σ^* (Strings) bestehendes „Alphabet" dazwischenzuschalten, das sogenannte *Wörterbuch*. Dieser Ansatz taucht auch in dem am Ende des letzten Kapitels am Rand erwähnten Modellierungsverfahren „WORD" auf; dort wurden die Strings nach „semantischen" Gesichtspunkten ausgewählt (Wörter/Nichtwörter) und dann bezüglich dieser Strings ein Kontextmodell gebildet. Jetzt werden die Strings so ausgewählt, daß sie alle möglichst die gleiche Wahrscheinlichkeit haben, also ein zur bisherigen Vorgehensweise völlig gegensätzliches Ziel. Kompression findet statt, da verschieden lange Strings in jeweils gleich vielen Bits dargestellt werden. Es sei bemerkt, daß das Run-Length-Verfahren aus der Einleitung als eine sehr spezielle und partielle Realisation dieser Idee aufgefaßt werden kann.

Der Zweck der Gleichwahrscheinlichkeit ist es, zeitaufwendige Kodierungsverfahren (Huffman, arithmetische Kodierung) überflüssig zu machen; erreicht wird sie, indem man die Länge der Strings kontrolliert: wird ein String zu häufig, so differenziert man ihn, indem man statt seiner längere Strings mit dem ursprünglichen als Anfang betrachtet.

Diese Idee wird auf eine implizite Weise realisiert, in der von Häufigkeiten gar nicht die Rede ist. Wir wollen zuerst den Rahmenalgorithmus betrachten. Auch hier gilt, daß wir uns mit adaptiver Modellierung befassen; die statischen und variablen Versionen können leicht daraus abgeleitet werden.

Das Wörterbuch, das zum Beispiel als Digitaler Suchbaum implementiert sein könnte, sehen wir hier implementationsunabhängig als **set of** Σ^* an.

```
type
    Wörterbuch = set of Σ*
end type
var
```

```
    W, X: Wörterbuch
    S, x: Σ*
end var
function InitW: Wörterbuch
    Anfangsbesetzung des Wörterbuchs
function Match(W: Wörterbuch): Σ*
    Finde einen Wörterbuchstring im Originaltext
function Kodierung(S : Σ*, W: Wörterbuch): Φ*
    Kodiere die Position von S im Wörterbuch
function Copy: Σ*
    liefert ein Anfangsstück des Originaltextes unabhängig von W
function Dazu(W: Wörterbuch): Wörterbuch
    Liefert einige Strings, die in das Wörterbuch aufgenommen werden sollen
function Weg(W: Wörterbuch): Wörterbuch
    Liefert einige Strings, die aus dem Wörterbuch entfernt werden sollen
function IstVoll(W:Wörterbuch): boolean
begin
    W := InitW
    loop
        S := Match(W)
        write(Kodierung(S))
        write(Copy)
        X := Dazu(W)
        while (X ≠ ∅) and (not IstVoll(W) or Weg(W) ≠ ∅)
            wähle ein x ∈ X
            X := X \ {x}
            if Full(W)
                W := W \ Weg(W)
            end if
            W := W ∪ {x}
        end while
    end loop
end
```

Die Werte „Kodierung(S)“, die geschrieben werden, heißen *Zeiger*. Die Dekodierung ist dann ziemlich klar: Man benötigt eine Prozedur „Decode“, die Wörterbuchzeiger und kopierten Originaltext unterscheiden kann, und aktualisiert das Wörterbuch wie bei der Kodierung.

Die verschiedenen Varianten der Wörterbuchkompression unterscheiden sich durch die Spezifikation der in dem Algorithmus genannten Prozeduren und Funktionen.

Zunächst muß garantiert werden, daß in jedem Schritt ein nichtleeres Stück des Originaltextes verarbeitet wird. Das bedeutet: in jedem Schritt muß mindestens die Funktion „Match“ oder die Funktion „Copy“ einen nichtleeren String liefern. Dazu gibt es mehrere Ansätze:

- Der Ansatz von Welch:
  - „Copy“ liefert immer den Leerstring.

- „InitW“ enthält ganz Σ^1.
- „Weg“ liefert nie Strings aus Σ^1.

Damit findet „Match“ sicher jedesmal einen nichtleeren String. Der kodierte Text enthält nur Zeiger.

- Der Ansatz von Lempel und Ziv:
  - „Copy“ liefert immer das erste Zeichen des verbleibenden Originaltextes.
  - „InitW“ ist beliebig, zum Beispiel nur der Leerstring.

  Beachte, daß auch der Leerstring in einem nichtleeren Zeiger kodiert wird. Der kodierte Text enthält abwechselnd einen Zeiger und ein Zeichen im Klartext.

- Der gemischte Ansatz:
  - Immer, wenn „Match“ den Leerstring liefert, liefert „Copy“ das erste Zeichen des verbleibenden Originaltextes, sonst den Leerstring.
  - „InitW“ enthält z. B. nur den Leerstring.

Der gemischte Ansatz liefert sicher die beste Kompression, der Welch-Ansatz ist bequemer zu programmieren und zu diskutieren. Wir gehen in der weiteren Diskussion vom Welch-Ansatz aus.

Für die Wahl der Funktion „Match“ bieten sich ebenfalls mehrere Möglichkeiten an:

- Der „gierige“ (greedy) Ansatz: „Match“ liefert den längstmöglichen Wörterbuchstring, der am Textanfang gefunden werden kann. Das ist nicht notwendig optimal: Wenn zum Beispiel der Text mit „`abrakadabra`“ anfängt, und das Wörterbuch die Strings `ab`, `abr`, `rakadabra` enthält, so würde der gierige Ansatz `abr` liefern, wodurch `rakadabra` nicht mehr gefunden wird.
- Der Look-ahead-Ansatz: die nächsten k Zeichen (vielleicht 100) werden in eine Stringvariable „Lookahead“ gelesen, und in dieser Variablen werden alle möglichen Einteilungen ausprobiert, die beste wird genommen.

In der Praxis zeigt sich, daß der Look-ahead-Ansatz meistens außer Rechenzeitverbrauch nur leicht verbesserte Kompression bringt.

Zur Funktion „Dazu“: wir gehen grundsätzlich so vor, daß wir uns den zuletzt mit „Match“ gefundenen String unter dem Namen T merken und dann die Menge „Dazu“ aus einigen Verlängerungen von T bestehen lassen, die aus dem neu gefundenen Matchstring $S = s_1 \cdots s_k$ abgeleitet sind. Drei sinnvolle Wachstumsstrategien:

- Kopf-Wachstum: „Dazu“ ist die einelementige Menge $\{Ts_1\}$.
- Ganzwort-Wachstum: „Dazu“ ist die einelementige Menge $\{TS\}$.

- Volles Wachstum: „Dazu“ ist die Menge

$$\{Ts_1, Ts_1s_2, \ldots, Ts_1 \cdots s_{k-1}, TS\}$$

Beispiel 7.1 Wir wollen den Text

`das_Schaf_kann_schon_immer_Schlagschach_im_Schlaf`

betrachten. Wir gehen vom Welch-Ansatz für Initialisierung etc. und dem gierigen Matchansatz aus (das läßt sich am bequemsten durchprobieren) und verzeichnen das Wachstum von W in den drei Dazu-Ansätzen.

Kopf-Wachstum		Ganzwort-Wachstum		Volles Wachstum	
Match	Dazu	Match	Dazu	Match	Dazu
d		d		d	
a	da	a	da	a	da
s	as	s	as	s	as
_	s_	_	s_	_	s_
S	_S	S	_S	S	_S
c	Sc	c	Sc	c	Sc
h	ch	h	ch	h	ch
a	ha	a	ha	a	ha
f	af	f	af	f	af
_	f_	_	f_	_	f_
k	_k	k	_k	k	_k
a	ka	a	ka	a	ka
n	an	n	an	n	an
n	nn	n	nn	n	nn
_	n_	_	n_	_	n_
s	_s	s	_s	s	_s
ch	sc	ch	sch	ch	sc sch
o	cho	o	cho	o	cho
n_	on	n_	on_	n_	on on_
i	n_i	i	n_i	i	n_i
m	im	m	im	m	im
m	mm	m	mm	m	mm
e	me	e	me	e	me
r	er	r	er	r	er
_S	r_S	_S	r_S	_S	r_ r_S
ch	_Sc	ch	_Sch	ch	_Sc _Sch
l	chl	l	chl	l	chl
a	la	a	la	a	la
g	ag	g	ag	g	ag
sc	gs	sch	gsch	sch	gs gsc gsch
ha	sch	a	scha	a	scha
ch	hac	ch	ach	ch	ach
_	ch_	_	ch_	_	ch_
im	_i	im	_im	im	_i _im
_Sc	im_	_Sch	im_Sch	_Sch	im_ im_S im_Sc im_Sch
h	_Sch	la	_Schla	la	_Schl _Schla
la	hla	f	laf	f	laf
f	laf				

Volles Wachstum hat den Nachteil, daß das Wörterbuch zu schnell wächst (was die Anzahl der verschiedenen Zeiger und daher die dafür zu verwendenden Bits erhöht). Praktische Tests zeigen, daß das Ganzwort-Wachstum besser komprimiert als das volle, und dieses besser als das Kopfwachstum.

Wir kommen zu den Möglichkeiten für die Funktion „Weg", die das Wörterbuch verkleinert, wenn es voll ist.

- Einfrieren: „Weg" ist immer die leere Menge, wenn das Wörterbuch voll ist, dann ändert es sich für den Rest des Originaltextes nicht weiter.
- Die Least-recently-used-Strategie (LRU): „Weg" enthält immer genau den String, bei dem es am längsten her ist, daß er von „Match" gefunden wurde.
- Die Least-frequently-used-Strategie (LFU): „Weg" enthält immer denjenigen String, der am seltensten von „Match" gefunden wurde. Neu aufgenommene Strings müssen mit einem positiven Zählwert initialisiert werden (z. B. der durchschnittlichen Benutzungshäufigkeit aller Strings), damit sie nicht sofort wieder aus dem Wörterbuch entfernt werden.
- Die Swap-Strategie: Es gibt zwei Wörterbücher; das zweite wird gefüllt, wenn das erste voll ist, „Match" benutzt aber weiterhin das erste. Ist das zweite Wörterbuch ebenfalls voll, so wird das erste Wörterbuch völlig entleert, und die beiden Wörterbücher tauschen ihre Rollen.

Ein Prinzip ist jedenfalls: die mit „Init" am Anfang in das Wörterbuch hineingeschriebenen Zeichen werden grundsätzlich nie gelöscht.

Die Einfrierstrategie ist am schnellsten; sie komprimiert nur gut, wenn die Struktur des Textes sich nicht mehr wesentlich ändert, wenn das Wörterbuch voll ist. Leider ist das oft nicht erfüllt, da bei Texten, seien es auch Quellprogramme, übersetzte Programme, Datenbestände, oft gerade am Anfang Teile stehen, die sich vom Rest sehr unterscheiden (Inhaltsverzeichnisse, Einleitungskommentare, Listen, Header).

Die LFU-Strategie liefert in der Praxis keine so große Kompressionsverbesserung gegenüber der LRU-Strategie, daß der erhöhte Aufwand gerechtfertigt wäre. Die Swap-Strategie arbeitet ähnlich gut wie die LRU-Strategie, verbraucht aber den doppelten Speicherplatz für die Wörterbücher.

Für die Kodierung gibt es auch mehrere Möglichkeiten:

- Feste Zeigergröße: man kodiert jeden Zeiger als Bitkette konstanter Länge; besonders geeignet sind 16 bit, da dann keine Bytegrenzen überschritten werden müssen; auch 12 bit sind die Beachtung wert, wenn mit vielen kurzen Strings gerechnet werden muß. Die Zeigergröße bestimmt die Maximalgröße des Wörterbuchs.
- Steigende Zeigergröße: man kodiert jeden Zeiger in $\lceil \mathrm{ld}\,|W| \rceil$ bit. Die Wörterbuchgröße ist nur noch durch den Speicherplatz beschränkt; die (De-) Kodierung ist aber langsamer.
- LZH: Das sogenannte „Lempel-Ziv-Huffman-Verfahren", in dem die Zeiger Huffman-kodiert werden. Auch die arithmetische Kodierung ist hier

natürlich möglich und besser geeignet; verwendet man aber ein unabhängiges Modell, so ist der Kompressionsgewinn nur gering (einige Prozentpunkte), der Zeitverlust aber hoch: das intuitive Ziel des Wörterbuchmodelles war es ja gerade, die Zeiger möglichst gleichwahrscheinlich zu machen, um den Zeitaufwand bei der Kodierung zu reduzieren. Auch mit Kontextmodellen ist nicht allzu viel Gewinn zu erwarten, da aufeinanderfolgende Strings ja (wenigstens in der Ganzwort-Strategie und in der vollen Wachstumsstrategie) zu einem Wörterbuchstring verschmelzen.

Implementation

Wir implementieren das Wörterbuch als digitalen Suchbaum. Die Wörterbucheinträge sind Wege im Suchbaum, die von der Wurzel ausgehen, und an einem als Wörterbucheintrag markierten Knoten enden. Die Blätter sind grundsätzlich als Wörterbucheintrag markiert. Die markierten Knoten enthalten die Kodierung des Zeigers, der in den komprimierten Text geschrieben wird.

Dabei werden die Zeigerwerte, solange das Wörterbuch noch nicht voll ist, einfach hochgezählt. Das Ereignis „Wörterbuch voll" tritt ein, wenn ein maximaler Zählwert erreicht ist. Danach werden durch das Löschen von Wörterbucheinträgen immer wieder Zeigerwerte frei, die ich in einem Stapel oder einer Liste zwischenspeichern kann, bis sie wieder neu vergeben werden. Beim Dekodieren habe ich dann eine mit den Zeigerwerten indiziertes **array** von Pointern in den Suchbaum, das sich adaptiv mit dem Suchbaum mitentwickelt.

Die Realisierung des Wachstums im Suchbaum selbst ist unproblematisch, es sollte klar sein, wie man einem digitalen Suchbaum neue Strings hinzufügt. Beim Entfernen von Strings muß man sich mehr Gedanken machen. Wir behandeln die LRU-Strategie. Für LFU gilt im wesentlichen dasselbe, Einfrieren und Swap machen ohnehin keine Implementationsprobleme.

Bei LRU wird der String entfernt, dessen letzte Benutzung am weitesten zurückliegt. Man verwaltet die Wörterbuchstrings als Warteschlange, also als eine lineare Liste, bei der jeder neu ins Wörterbuch aufgenommene String hinten angefügt wird, jeder String, der von „Match" gefunden wird, aus der Warteschlange herausgenommen wird, um sich wieder neu hinten anzustellen, und (wenn das Wörterbuch voll ist) der vorne in der Warteschlange befindliche String aus dem Wörterbuch (und der Warteschlange) entfernt wird. Die Warteschlange kann ähnlich wie bei der adaptiven Huffman-Kodierung als zusätzliche Verzeigerung im digitalen Suchbaum realisiert werden.

Wie entfernt man nun aber Strings aus dem Wörterbuch? Ist dieser String durch ein Blatt des digitalen Suchbaumes bezeichnet, so ist das kein Problem, man entfernt dieses Blatt und sukzessive den jeweiligen Vorgängerknoten, bis der Knoten, bei dem man gerade ist,

- selbst als Wörterbuchstring markiert ist, oder
- außer dem gerade gelöschten noch weitere Nachfolger hat.

Ist der Wörterbuchstring aber durch einen inneren Knoten bezeichnet, so kann seine Entfernung aus dem Wörterbuch lediglich in der Entfernung der Markierung als Wörterbucheintrag bestehen, was nicht mit Speicherplatzfreigabe

verbunden ist. Das ist eine unbefriedigende Situation, da trotz der Entfernung eines Wörterbucheintrages keine neuen Strings aufgenommen werden können.

Für eine bessere Lösung beobachten wir zunächst einmal die folgende Eigenschaft der Kopf- und der vollen Wachstumsstrategie:

- Im Kopfwachstum und im vollen Wachstum sind, solange noch kein Wörterbucheintrag entfernt wurde, sämtliche Knoten des digitalen Suchbaumes als Wörterbucheintrag markiert.

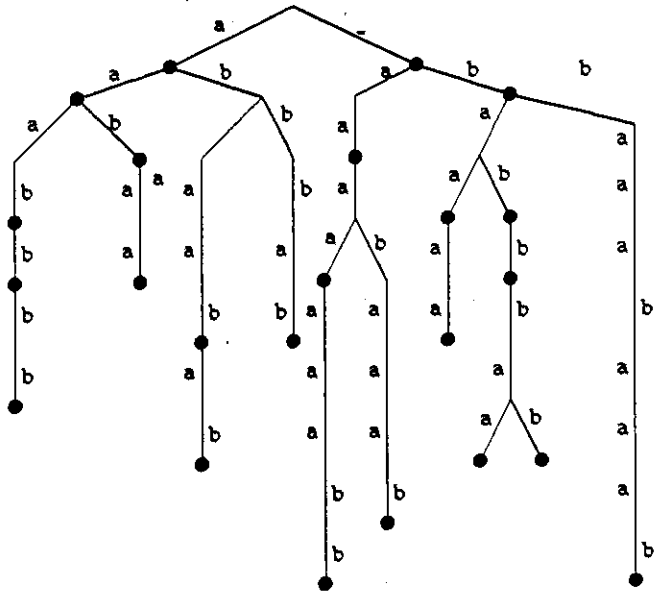
Diese Eigenschaft wollen wir stabilisieren, indem wir durch eine Modifikation der LRU-Löschstrategie vermeiden, innere Knoten zu entfernen. Das hat auch den zusätzlichen Vorteil, daß wir uns um eine gesonderte Markierung von Knoten als Wörterbucheintrag nicht zu kümmern brauchen, was Aufwand spart. Die Entfernung innerer Knoten vermeiden wir, indem wir jedesmal, wenn wir einen Knoten an das Ende der Warteschlange schieben, sämtliche Vorgänger bis zur Wurzel ebenfalls dorthin schieben. Ganz vorn in der Warteschlange steht auf jeden Fall ein Blatt.

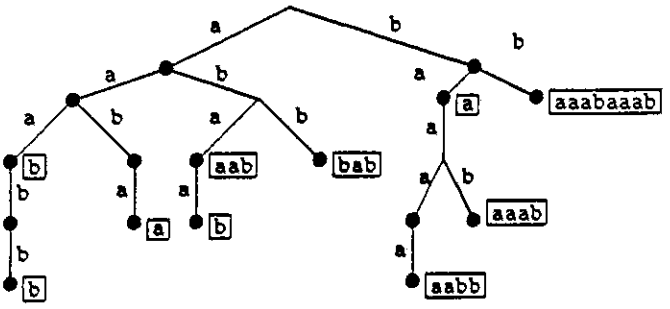
Beim Ganzwortwachstum können wir nicht so vorgehen; hier bietet es sich an, die Darstellung des digitalen Suchbaumes wie im Beispiel zu kompaktifizieren:

Beispiel 7.2 $\Sigma = \{a, b\}$, der Text ist

a a a b aa ab aaab b b baa aa baa abaaab bb baaaabaaab aaab aaabb bb ab bbab bbab bbaa aa b

(die Leerzeichen gehören nicht zum Text, sondern zeigen die von „Match" gefundenen Strings an). Der ohne Löschen im Ganzwortwachstum entstandene Baum in einfacher und in kompakter Form:





Jede Kette von aufeinanderfolgenden Knoten, die weder Verzweigungspunkte noch Wörterbucheinträge sind, wird als ein einziger Knoten dargestellt, und die Zeichen, die den Weg durch diese Kette beschreiben, als diesem Knoten zugeordnete Zeichenkette. (Wir brauchen dazu die programmiertechnische Möglichkeit, für eine Zeichenkette gerade so viel Speicherplatz dynamisch zu reservieren, wie man zu ihrer Darstellung braucht. Das geht in C; in PASCAL z. B. ist das nicht ohne Spracherweiterung möglich – man kann sich hier evtl. behelfen, indem man diese dynamische Speicherverwaltung selbst übernimmt und die Strings als Substrings eines großen Strings realisiert, in den man dann mit Index/Länge-Paaren hineinzeigt)

Das Löschen eines Wörterbucheintrages besteht jetzt meist darin, Platz für zwei Strings freizugeben, dafür Platz für einen String anzufordern, dessen Länge die Summe der beiden freigegebenen ist, und die Konkatenation der beiden freigegebenen Strings in den neuen String zu schreiben, und außerdem einen Knoten freizugeben. Man gewinnt also nicht ganz, aber doch fast den Platz, den ein neuer Wörterbucheintrag belegt.

Hash-Tabellen

Eine andere Möglichkeit, Strings abzuspeichern (sei es bei Kontextmodellen oder bei Wörterbuchmodellen), ist es, sogenannte *Hash-Tabellen* anstelle von Digitalen Suchbäumen zu verwenden. Hier werden sämtliche Strings in linearen Listen („Slots") abgelegt, von denen eine Anzahl (z. B. 256) zur Verfügung stehen. Man hat also ein **array**, von denen jedes Element eine (z. B. einfach) verkettete Liste aus (z. B. durchschnittlich 8) Strings ist. Ein gegebener String S wird nun in die Liste mit dem Index $h(S)$ abgelegt (Sortieren ist überflüssig), wobei $h(S)$ eine heuristisch gefundene Funktion ist, die einmal schnell berechenbar sein muß, und zweitens die Strings möglichst gleichmäßig über die verschiedenen Listen verteilt. Das erreicht man so, daß man die Funktion nur von den ersten (zwei oder drei) Zeichen des Strings abhängen läßt; meistens handelt es sich um bitweise Verknüpfungen der Zeichen (z. B. s_1 **and** s_2 **xor** s_3). Wenn häufig Strings mit gleichen Anfängen auftauchen (das ist bei uns der Fall), so kann man auch die Länge der Strings in $h(S)$ eingehen lassen. Das ist in Pascal schnell möglich, da dort die Länge im String explizit gespeichert ist; in C, wo die Länge immer neu ausgezählt werden müßte, kann man die Verwaltung von Längenzählern ja selbst in die Hand nehmen.

Ein Vorteil dieses Ansatzes ist es, daß das Löschen von beliebigen Strings aus dem Wörterbuch leicht möglich ist. Die Strings nehmen weniger Platz in Anspruch, aber dafür kommen gemeinsame Anfänge mehrerer Strings auch mehrfach vor.

Hash-Tabellen können nicht nur zur direkten Abspeicherung der Strings verwendet werden, sondern bieten auch eine Möglichkeit der platzsparenden Verwaltung Digitaler Suchbäume: die Knoten werden nicht in einer dynamischen Datenstruktur mit Pointern, sondern in einem **array** abgelegt. (Die Indizes können bei der Kodierung in Wörterbuchmodellen direkt als Zeigerwerte verwendet werden.) Ein Knoten enthält nun keine Verweise auf seine Nachfolger. Wollen wir den Nachfolger des Knotens mit dem Index j zum Zeichen a bestimmen, so verwenden wir eine Hashfunktion $h(j, a)$, die uns die Slotnummer in einer Hashtabelle liefert. Die Listenelemente enthalten als Inhalt Tripel der Form (j, a, j'); wir suchen das Listenelement mit den ersten beiden Einträgen j und a und entnehmen daraus den Nachfolger j'; finden wir kein solches Element, so bezeichnet j ein Blatt.

Einige Verfahren

Ein kurzer Abriß der bekannt gewordenen Kombinationen und Varianten der genannten Ansätze zur Wörterbuchmodellierung:

LZ78. In einer Veröffentlichung von 1978 stellten Lempel und Ziv das erste Wörterbuchverfahren vor. Hier wird der Lempel-Ziv-Ansatz zur Fortschrittsgarantie verwendet, d. h.: Zeiger und kopierte Zeichen wechseln ab. Das Wörterbuchwachstum folgt dem Kopf-Ansatz, d. h. jede gefundene Übereinstimmung plus das nächste Zeichen werden in das Wörterbuch aufgenommen. Die Kodierung verwendet Zeiger wachsender Größe. Der Artikel von Lempel und Ziv ist sehr theoretisch, und sein primäres Ziel ist es, zu zeigen, daß dieses Verfahren bei unbeschränkter Wörterbuchkapazität asymptotisch optimal ist, also auf lange Sicht die Entropie der Quelle erreicht, wenn diese eine ergodische Markovquelle mit stationärer Verteilung der Anfangszustände ist. Allerdings ist die Konvergenz nicht besonders schnell, so daß die Vorteile der Wörterbuchmodellierung nicht primär in einer guten Kompression liegen (obwohl sie sicher die Kompression mit einem unabhängigen Modell schlagen), sondern in der Möglichkeit effizienter Implementation.

LZW Welch veröffentlichte 1984 den Welch-Ansatz zur Fortschrittsgarantie (bei dem die einzelnen Buchstaben fest zum Wörterbuch gehören.) Auch Welch benutzte das Kopfwachstum, eine konstante Zeigergröße von 12 bit, und die Einfrierstrategie für die Wörterbuchschrumpfung. All diese Strategien sind besonders schnell, und das ist deshalb besonders wichtig, da Welch seine Kompressionsmethode für eine Hardwareversion in Festplattentreibern vorschlug.

LZC Der Ansatz, der vom UNIX-Kommando „compress“ benutzt wird. Es handelt sich um eine mehrfach verbesserte Version des LZW-Verfahrens; einmal werden wieder Zeiger wachsender Größe verwendet (bis zu 16 bit), und

die Löschstrategie ist eine Verwandte der Swap-Strategie: das Wörterbuch wird eingefroren; das zweite Wörterbuch wird erst dann gestartet, wenn die Kompressionsrate zu sinken beginnt.

LZT Tischer führte 1987 die LRU-Löschstrategie ein, bei sonstiger Übernahme von „compress". Der Leistungsvergleich liefert eine etwas langsamere Laufzeit bei geringerem Speicherplatzbedarf und ungefähr gleicher Kompressionsrate.

LZMW Miller und Wegman führten 1984 den Ganzwortansatz für das Wachstum ein; sie verwendeten den LFU-Löschansatz. Die Implementation wird natürlich komplizierter, aber die Kompressionsrate besser.

In der Kompressionsrate liegen Wörterbuchverfahren etwa ein halbes Bit pro Byte über den Kontextmodellen. Der Geschwindigkeitsgewinn ist jedoch erheblich, denn der Aufwand zur Verwaltung eines Wörterbuches ist ähnlich hoch wie der Aufwand zur Verwaltung der Kontexte; bei Kontextmodellen kommt aber jetzt noch der Aufwand zur Verwaltung der (kumulativen) Häufigkeiten und die Kodierung dazu, während der Kodierungsaufwand bei Wörterbuchmodellen normalerweise gerade im Aufrechterhalten einer ungeordneten Menge von Zahlen besteht.

Kapitel 8

Fenstermodelle

Fenstermodelle können als spezielle Variante der Wörterbuchmodelle angesehen werden. Die Idee ist es, als „Wörterbuch" die letzten N Zeichen, das *Fenster* zu verwenden; die Strings sind einfach sämtliche Teilstrings des Fensters. Die Aufnahme- und die Löschstrategie ergeben sich automatisch aus dieser Vorgabe. Die Kodierung der Zeiger besteht aus zwei Teilen (δ, λ): der Teil δ sagt aus, wie viele Zeichen zurück der mit „Match" gefundene String liegt, und der Teil λ, wie lang er ist.

Auch hier muß sichergestellt werden, daß der Eingabetext immer weiterverarbeitet werden kann. Der Lempel-Ziv-Ansatz (Abwechselnd wird ein Zeichen kopiert und ein Zeiger kodiert, gegebenenfalls der „dummy"-Zeiger auf den Leerstring) ist eine Möglichkeit; der Welch-Ansatz versagt, da das Wörterbuch durch das Textfenster vorgegeben ist und nicht gezwungen werden kann, sämtliche Elemente von Σ zu enthalten. Der entsprechende Effekt aber wird durch den Ansatz von Storer und Szymanski ermöglicht: wir denken uns das Alphabet Σ als „ideell" im Wörterbuch enthalten; ein Zeichen wird als ein Null-Bit, gefolgt von diesem Zeichen (in Binärdarstellung) kodiert, ein echter Zeiger als ein Eins-Bit, gefolgt von Abstands- und vom Längenfeld. Ein gemischter Ansatz wie bei den Wörterbuchmodellen läßt sich natürlich ebenfalls machen.

Beispiel 8.1 Wir wollen wieder den Text

```
das Schaf kann schon immer Schlagschach im Schlaf
```

komprimieren. Wir wählen $N = 25$ und erhalten:

```
                    das Schaf kann
s(9,0)o(4,0)immer(21,2)lag(16,1)a(2,0)(16,0)(14,3)f.
```

Dabei bezeichnet der erste Eintrag im Zahlenpaar die Anzahl der Schritte, die es zurückzugehen gilt, wobei beim unmittelbar dem zuletzt betrachteten Zeichen vorausgehende Zeichen mit Null zu zählen begonnen wird, und die Länge immer der Längenzähler plus zwei ist.

Sinnvolle Größen sind etwa 8191 für die Fenstergröße, 63 für die größte Stringlänge, so daß die Zeiger $13 + 6 + 1 = 20$ bit beanspruchen; in diesem Fall wird man Strings erst ab der Länge 3 berücksichtigen.

Implementation

Das Fenster soll implementiert werden. Da wir keine Texte im Speicher herumschieben wollen, bietet sich die Datenstruktur des *Ringpuffers* an:

```
const N = ···
var
    FensterArray: array[0..N − 1] of byte
    Pos: 0..N − 1
end var
function Fenster(I: integer): byte
    begin
        Fenster := FensterArray[(Pos+I) mod N]
    end Fenster
procedure Weiter
    begin
        Pos := Pos + 1 mod N
    end Weiter
procedure Setze(B: byte)
    begin
        Weiter
        FensterArray[Pos]:=B
    end
begin
    Pos := 0
    ...
```

Das momentan erste Zeichen des Fensters ist also „Fensterinhalt(1)", das letzte Zeichen des Fensters ist „Fensterinhalt(N)". In C läßt sich das besonders praktisch mit Makros implementieren:

```
#define N 8191
int fenster_array[N];
int pos=0;
#define Fenster(i) (fenster_array[((i)+pos)%N])
#define Weiter() {pos++;pos%=N}
#define Setze(b) {Weiter();Fenster(pos)=b}
```

Ein weiteres Problem ist die Funktion „Match". Es ist sehr zeitaufwendig, das Fenster jedesmal neu nach Übereinstimmungen zu durchsuchen.

Ein erster Lösungsansatz besteht darin, für jedes Zeichen aus Σ ein Liste der Fensterpositionen zu verwalten, an denen sich dieses Zeichen befindet. Die Suche nach Übereinstimmung startet dann an allen dieser Stellen, dann wird das nächste Zeichen gelesen und mit den jeweils nachfolgenden Zeichen im Fenster verglichen und so weiter, bis eine Anzahl von Stellen übrigbleibt, die beim nächsten Zeichen alle keine Übereinstimmung ergeben würden. Von denen wählen wir uns eine beliebig als „Match".

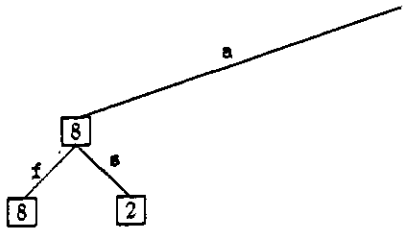
Die genannten Stellen sind sinnvollerweise die „echten" Indizes in FensterArray; denn diese ändern sich nicht Zeichen für Zeichen. Die Verwaltung besteht

darin, vor jedem Lesen eines neuen Zeichens die Position des Fenster am weitesten links stehenden Zeichens, das ja demnächst verschwinden soll, in seiner Einspringpunkt-Liste zu entfernen und nach dem Lesen eines neuen Zeichens diese Position der Einspringliste dieses Zeichens hinzuzufügen.

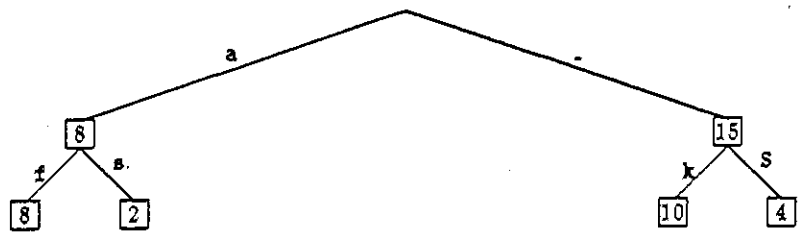
Eine Verfeinerung, die die Anzahl der gleichzeitig vorliegenden Strings reduziert, liegt darin, statt der jeweils ersten Zeichen gleich die String-Anfänge aus zwei oder gar drei Zeichen als Einsprung-Indizes zu verwenden. Da diesem Ansatz wiederum deren große Zahl entgegensteht, organisiert man die Anfänge in einer Hash-Tabelle.

Ist man schon so weit, so kann man es auch in Betracht ziehen, die Einspringstellen mit einem digitalen Suchbaum zu finden. Die Vorgehensweise zeigt sich in unserem Beispiel:

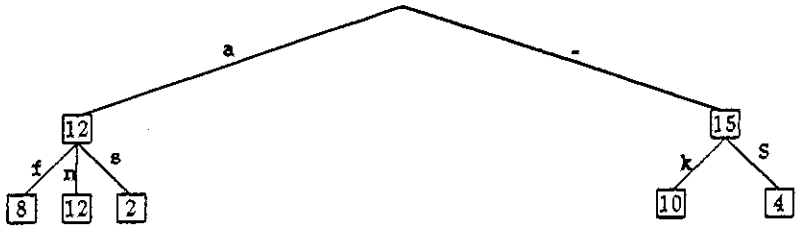
Beispiel 8.2 `das Schaf kann schon immer Schlagschach im Schlaf`, $N = 25$. Der Digitale Suchbaum ist am Anfang leer. Nichts passiert, bis an Position 8 das `a` zum ersten Mal wieder auftaucht. Dieses Zeichen wird unter die Wurzel in den Baum aufgenommen; die beiden verschiedenen Folgezeichen `s` und `f` ebenfalls. Die Knoten erhalten folgende Verweise: der `as`-Knoten die 2, der `af`-Knoten die 8, und der `a`-Knoten ebenfalls die 8. Die letztere Entscheidung begründet sich damit, daß man Stellen möglichst weit rechts im Fenster bevorzugt; diese haben kürzere Sprungweiten, was bei variablen Zeigergrößen von Bedeutung sein kann, und sie verlassen das Fenster später, was dann den Verwaltungsaufwand reduziert.



Beim Leerzeichen an Position 10 passiert dasselbe.

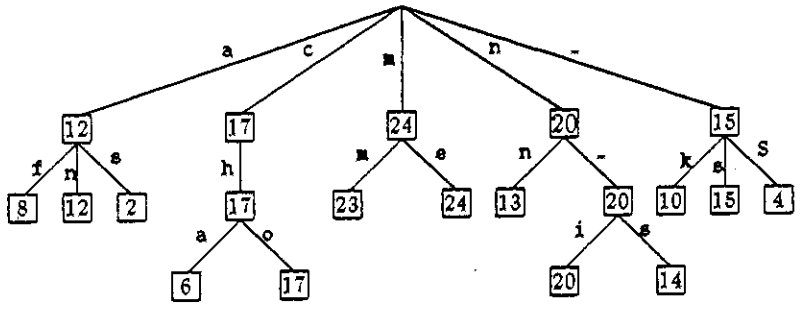


Das a an Position 12 ergibt eine weitere Verzweigung unter dem a-Knoten, sowie eine Aktualisierung des a-Verweises von 8 auf 12.



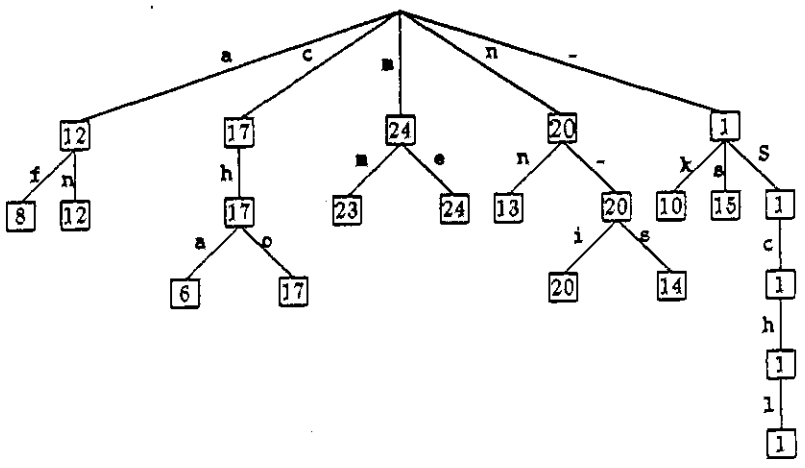
Jetzt kommt die Übereinstimmung n_, bei der wir auf naheliegende Weise verfahren. Die Aktualisierung der Verweise innerer Knoten geht immer bis zur Wurzel.

Der Baum wächst; die Situation an Position 19 ist so:



An Position 27 passieren zwei Dinge. Einmal muß die Übereinstimmung _Sch eingetragen werden, und dann muß der mit 2 markierte as-Knoten entfernt werden, da das a das Fenster verläßt. Der Anfang der Übereinstimmung _Sch

verschwindet, während die Übereinstimmung festgestellt wird; wir lassen _Schl im Baum, _Scha wird gestrichen.



Die Überprüfung, welche Knoten gestrichen werden müssen, geschieht ab Position 25 jedesmal. Der Fensteranfang wird im Suchbaum nach unten verfolgt, so weit es geht. Steht an dem Knoten, den wir auf diese Weise erreichen, ein Verweis auf den augenblicklichen Fensteranfang, so ist dieser Knoten zwangsläufig ein Blatt (das folgt aus der Aktualisierung innerer Knoten). Vom Blatt aus wird der Weg zur Wurzel hin Knoten für Knoten gelöscht, bis wir auf den ersten Knoten mit Verweis auf eine andere Position treffen. Übrigens: welcher Knoten „am weitesten rechts" steht, ergibt sich nicht aus dem **array**-Index sondern aus der berechneten Position im Ringpuffer.

Im Beispiel haben wir nur wiederholte Auftritte von Zeichen bzw. Zeichenketten für Einstiegspunkte verwendet. Natürlich kann man auch bis zu einer vorgegebenen Stringlänge k alle im Fenster vorkommenden Strings in den Suchbaum eintragen. Die Wahl von k ergibt sich aus einer Abwägung des Baumverwaltungsaufwandes mit dem Suchaufwand.

Die kompakte Darstellung Digitaler Suchbäume aus dem letzten Kapitel, bei der verzweigungsfreie Knotenverläufe als Strings variabler Länge realisiert wurden, wird in Verbindung mit Fenstern besonders einfach. Die Strings müssen gar nicht im Baum angelegt werden, sie stehen ja schon im Fenster; es genügt die Angabe von Position und Länge des Strings.

Noch ein Ansatz für die Einstiegsstruktur wird durch eine Variante der binären Suchbäume geboten, in denen man alle Teilstrings des Fensters bis zu einer Länge k wie folgt abspeichert:

- jeder Knoten hat maximal zwei Nachfolger.
- Die Knoten enthalten verweise auf die Strings in Form von Index/Länge-Paaren.

- Die Knoten stehen derart im Baum, daß für jeden Knoten gilt: der dem linken Nachfolger zugeordnete String liegt lexikographisch vor dem gegebenen Knoten, der rechte Nachfolger dahinter.
- Auffinden eines Knotens im Baum ist leicht, indem man sich von der Wurzel aus anhand der lexikographischen Ordnung solange für den linken oder rechten Nachfolger entscheidet, bis man den String gefunden hat.
- Einordnen eines neuen Strings S: man arbeitet sich von der Wurzel nach unten; wir nehmen an, wir befinden uns am Knoten mit dem String K. Folgende Fälle können auftreten:

  $S < K$, K **hat einen linken Nachfolger.** Dann suchen wir bei diesem weiter.

  $S < K$, K **hat keinen linken Nachfolger.** Wir machen S zum neuen linken Nachfolger von K. S wird ein Blatt.

  $S = K$ Wir aktualisieren den Index am Knoten K.

  $S > K$ Wie $S < K$, nur links und rechts vertauschen.
- Um zu vermeiden, daß dieser Baum zu einer linearen Liste ausartet, gibt es den Ausweg der sogenannten balancierten Bäume, die auch schon im Huffman-Kapitel erwähnt wurden. Hier soll nicht weiter darauf eingegangen werden.

Zur Kompressionsleistung ist zu sagen, daß Fenstermodelle noch etwas weniger gut komprimieren als Wörterbuchmodelle; daür ist der Zeitaufwand noch etwas geringer, wenn man eine günstige Suchstruktur hat. Durchschnittswerte von Bell, Cleary und Witten (in Bit pro Byte):

compact	Fenster	Wörterbuch	DMC	Kontext
4.99	3.18	2.95	2.74	2.48

Kapitel 9

Ausblick

Storer und andere haben die Off-line-Komprimierbarkeit von Texten endlicher Länge (im Gegensatz zu der Komprimierbarkeit von Textquellen) untersucht. Sie benutzen die Theorie der formalen Sprachen und das Prinzip der *Programmgrößen-Komplexität:* ein Expansionssalgorithmus wird als ein virtueller Rechner aufgefaßt, und ein komprimierter Text als „Maschinenprogramm" für diesen Rechner. Der daraus expandierte Originaltext ist dann der Output dieses Programmes. Über die Ergebnisse und Methoden dieser hochtheoretischen Untersuchungen kann hier nichts berichtet werden; ich verweise auf das Buch von Storer.

Die Bestrebungen, durch sehr spezialisierte statische Modelle für die geeigneten Texte extrem hohe Kompression zu erreichen, habe ich schon berichtet. Hier gibt es lockere Verbindungen zur künstlichen Intelligenz (Mustererkennung). Ein offenes Problem ist die Verbesserung der adaptiven Kompression durch adaptive Markovstrukturen, die – im Gegensatz zu DMC – über die Leistungsfähigkeit von Kontextmodellen hinausgehen. Auch hier handelt es sich um ein Mustererkennungsproblem. Für die Kontextmodelle selbst wird vermutet, daß die Kapazität für verbesserte Ansätze noch nicht ausgeschöpft ist.

Die Geschwindigkeitssteigerung vor allem durch Parallelisierung (damit ist in den Lempel-Ziv-Algorithmen eine Verbesserung von „Match" möglich) wird im Buch von Storer ausführlich untersucht.

Der in dieser Vorlesung ausgeklammerte Bereich der Bilddatenkompression und der Datenreduktion ist zur Zeit ein noch mehr als die hier behandelte verlustfreie Textkompression explodierendes Forschungsgebiet.

Der Einbau von Kompressions- und Expansionsalgorithmen in Hardware-Einheiten ist ebenfalls im Interesse der Forschung. Auch hierzu steht etwas im Buch von Storer. Anwendungen hiervon liegen natürlich in der Nachrichtentechnik, aber auch in der Entwicklung von externen Speichern, insbesondere Festplattenkontrollern. Probleme, die hier auftreten, sind die Geschwindigkeit, der zeitlich unregelmäßige Datenfluß, der durch die Kompression verhinderte Random Access, und die Fehlerempfindlichkeit.

Zur Fehlerempfindlichkeit gibt es einige Untersuchungen aus den sechziger Jahren über statische Huffman-Codes, die sich teilweise auch auf statische Wörterbuchmodelle übertragen lassen: wird eine statisch Huffman-kodierte Se-

quenz gestört, so ist es unter gewissen, nicht zu unwahrscheinlichen Bedingungen möglich, daß eine Bitsequenz existiert, (von der Störung und dem Text unabhängig), die man in den kodierten Text einfügt, so daß im Anschluß an diese Sequenz der Text wieder korrekt dekodiert wird. Unter diesen Bedingungen ist es auch so, daß der Text sich mit Wahrscheinlichkeit 1 nach einiger Zeit selbst synchronisiert. Die Methoden liegen in der Untersuchung von Idealen freier Monoide. Diese Ergebnisse gelten nur für statische Verfahren und führen nur zu schwacher Hoffnung. Weitere Forschungsergebnisse auf dem Gebiet sind mir nicht bekannt, also ist demnächst von irgendwo ein Durchbruch zu erwarten.

Zum Schluß ein guter kryptographischer Ansatz: Man stellt dem Originaltext ein Schlüsselwort voran und komprimiert ihn adaptiv, ohne allerdings den Teil des komprimierten Textes mit zu übertragen, der dem Schlüsselwort entspricht.

p_{10}	p_2	$-\mathrm{ld}\, p$	$-p\,\mathrm{ld}\, p$	p_{10}	p_2	$-\mathrm{ld}\, p$	$-p\,\mathrm{ld}\, p$
0.01	0.0000001010	6.644	0.066	0.51	0.1000001010	0.971	0.495
0.02	0.0000010100	5.644	0.113	0.52	0.1000010100	0.943	0.491
0.03	0.0000011110	5.059	0.152	0.53	0.1000011110	0.916	0.485
0.04	0.0000101000	4.644	0.186	0.54	0.1000101000	0.889	0.480
0.05	0.0000110011	4.322	0.216	0.55	0.1000110011	0.862	0.474
0.06	0.0000111101	4.059	0.244	0.56	0.1000111101	0.837	0.468
0.07	0.0001000111	3.837	0.269	0.57	0.1001000111	0.811	0.462
0.08	0.0001010001	3.644	0.292	0.58	0.1001010001	0.786	0.456
0.09	0.0001011100	3.474	0.313	0.59	0.1001011100	0.761	0.449
0.10	0.0001100110	3.322	0.332	0.60	0.1001100110	0.737	0.442
0.11	0.0001110000	3.184	0.350	0.61	0.1001110000	0.713	0.435
0.12	0.0001111010	3.059	0.367	0.62	0.1001111010	0.690	0.428
0.13	0.0010000101	2.943	0.383	0.63	0.1010000101	0.667	0.420
0.14	0.0010001111	2.837	0.397	0.64	0.1010001111	0.644	0.412
0.15	0.0010011001	2.737	0.411	0.65	0.1010011001	0.621	0.404
0.16	0.0010100011	2.644	0.423	0.66	0.1010100011	0.599	0.396
0.17	0.0010101110	2.556	0.435	0.67	0.1010101110	0.578	0.387
0.18	0.0010111000	2.474	0.445	0.68	0.1010111000	0.556	0.378
0.19	0.0011000010	2.396	0.455	0.69	0.1011000010	0.535	0.369
0.20	0.0011001100	2.322	0.464	0.70	0.1011001100	0.515	0.360
0.21	0.0011010111	2.252	0.473	0.71	0.1011010111	0.494	0.351
0.22	0.0011100001	2.184	0.481	0.72	0.1011100001	0.474	0.341
0.23	0.0011101011	2.120	0.488	0.73	0.1011101011	0.454	0.331
0.24	0.0011110101	2.059	0.494	0.74	0.1011110101	0.434	0.321
0.25	0.0100000000	2.000	0.500	0.75	0.1100000000	0.415	0.311
0.26	0.0100001010	1.943	0.505	0.76	0.1100001010	0.396	0.301
0.27	0.0100010100	1.889	0.510	0.77	0.1100010100	0.377	0.290
0.28	0.0100011110	1.837	0.514	0.78	0.1100011110	0.358	0.280
0.29	0.0100101000	1.786	0.518	0.79	0.1100101000	0.340	0.269
0.30	0.0100110011	1.737	0.521	0.80	0.1100110011	0.322	0.258
0.31	0.0100111101	1.690	0.524	0.81	0.1100111101	0.304	0.246
0.32	0.0101000111	1.644	0.526	0.82	0.1101000111	0.286	0.235
0.33	0.0101010001	1.599	0.528	0.83	0.1101010001	0.269	0.223
0.34	0.0101011100	1.556	0.529	0.84	0.1101011100	0.252	0.211
0.35	0.0101100110	1.515	0.530	0.85	0.1101100110	0.234	0.199
0.36	0.0101110000	1.474	0.531	0.86	0.1101110000	0.218	0.187
0.37	0.0101111010	1.434	0.531	0.87	0.1101111010	0.201	0.175
0.38	0.0110000101	1.396	0.530	0.88	0.1110000101	0.184	0.162
0.39	0.0110001111	1.358	0.530	0.89	0.1110001111	0.168	0.150
0.40	0.0110011001	1.322	0.529	0.90	0.1110011001	0.152	0.137
0.41	0.0110100011	1.286	0.527	0.91	0.1110100011	0.136	0.124
0.42	0.0110101110	1.252	0.526	0.92	0.1110101110	0.120	0.111
0.43	0.0110111000	1.218	0.524	0.93	0.1110111000	0.105	0.097
0.44	0.0111000010	1.184	0.521	0.94	0.1111000010	0.089	0.084
0.45	0.0111001100	1.152	0.518	0.95	0.1111001100	0.074	0.070
0.46	0.0111010111	1.120	0.515	0.96	0.1111010111	0.059	0.057
0.47	0.0111100001	1.089	0.512	0.97	0.1111100001	0.044	0.043
0.48	0.0111101011	1.059	0.508	0.98	0.1111101011	0.029	0.029
0.49	0.0111110101	1.029	0.504	0.99	0.1111110101	0.014	0.014
0.50	0.1000000000	1.000	0.500	1.00	1.0000000000	0.000	0.000

Kapitel 10

Literatur

Lehrbücher

G. HELD. *Data Compression: Techniques and Applications, Hardware and Software Considerations.* Wiley, New York 1983. Enthält eine Anzahl von Ad-hoc-Verfahren und Beispielprogramme in BASIC. Ich kenne das Buch nicht selbst, aber die Wahl der Programmiersprache dürfte das meiste sagen.

J. A. STORER. *Data Compression: Methods and Theory.* Computer Science Press, Rockville (Maryland) 1988. Der Schwerpunkt liegt bei den Lempel-Ziv-Algorithmen, die systematisch untersucht werden. (Die entsprechenden Kapitel der Vorlesung orientieren sich daran). Die Arithmetische Kodierung und die statistische Modellierung werden praktisch gar nicht behandelt. Dafür enthält das Buch hochtheoretische Untersuchungen über Komprimierbarkeit und die NP-Vollständigkeit gewisser Kompressionsaufgaben, sowie ausführliche Betrachtungen zur Parallelimplementation und zum VLSI-Layout. Vollständige Beispielprogramme in Pascal. Ausführliche kommentierte Bibliographie.

BELL, CLEARY, WITTEN. *Text Compression.* Prentice Hall, New York 1990. Dieses Buch ist mir leider nicht rechtzeitig bekannt geworden. Aus der Kenntnis anderer Werke der Autoren schließe ich aber, daß es sich um das Standardwerk zum Stoff dieser Vorlesung handeln dürfte.

W. HEISE und P. QUATTROCCHI. *Informations- und Codierungstheorie. Mathematische Grundlagen der Daten-Kompression und -Sicherung in diskreten Kommunikationssystemen.* Springer-Verlag, Berlin, Heidelberg, New York, Tokyo 1983. Enthält neben der Betrachtung der Kodierungstheorie vor allem Material zu den Kapiteln über Zeichenkodierungen und über Textquellen.

J. KEMENY und J. L. SNELL *Finite Markov Chains.* Springer-Verlag, New York, Heidelberg, Berlin 1976. Stochastische Einführung in die endlichen Markovketten.

F.-J. FRITZ, B. HUPPERT und W. WILLEMS. *Stochastische Matrizen.* Springer-Verlag, Berlin, Heidelberg, New York 1979. Eine rein matrizentheoretische Behandlung der Markovketten.

A. V. AHO, J. E. HOPCROFT und J. D. ULLMAN. *Data Structures and Algorithms.* Addison-Wesley, Reading 1974. Eines der Standardlehrbücher über Datenstrukturen.

Überblicksartikel

FRANK BAUERNÖPPEL. *Imploding...freezing...done. Verfahren und Techniken zur Datenkompression.* c't Oktober 1991. Eine für die Kürze sehr ausführliche Einführung mit theoretischen Grundlagen, technischen Tips, Beschreibung der Datenstrukturen und Algorithmen, sowie Programmbeispielen in C.

H. ZUR NEDEN. *Squeeze, LZH & Co. Kompressionsverfahren implementiert.* c't Juli 1992. Im Ansatz moderner als der Artikel von Bauernöppel, aber weniger detailliert.

D. A. LELEWER und D. A. HIRSCHBERG. *Data Compression.* ACM Computing Surveys 19(3), September 1987. Ein Überblick über die Entwicklung der Datenkompression mit Schwerpunkt bei den Huffman-Codes. Fast nichts über statistische Modelle, dafür einige Bemerkungen über Fehleranfälligkeit. Ausführliche Bibliographie.

T. BELL, I. H. WITTEN, J. G. CLEARY. *Modeling for Text Compression.* ACM Computing Surveys 21(4), Dezember 1989. Ausführliche Beschreibung der statistischen Modellierungsmethoden, fast nichts über Kodierungsverfahren. Enthält vergleichende Untersuchungen zu den verschiedenen Lempel-Ziv-Verfahren sowie Testergebnisse. Ausführliche Bibliographie.

Artikel (Auswahl)

C. E. SHANNON. *The Mathematical theory of communication.* In: C. E. SHANNON und W. WEAVER. The Mathematical Theory of Communication. The University of Illinois Press, Urbana 1949. Der Klassiker.

D. A. HUFFMAN *A Method for the construction of minimum-redundancy-codes.* Proceedings of the IRE 40, 1952. Ebenfalls ein Klassiker.

J. KARUSH *A simple proof of an inequality of McMillan.* IRE Transactions on Information Theory 7, 1961. Keine halbe Seite lang!

J. J. RISSANEN und G. G. LANGDON. *Universal modeling and coding.* IEEE Transactions on Information Theory 27(1) 1981. Die Zerlegung der Kompressionsaufgabe in Modellierung und Kodierung wird eingeführt und formal studiert.

J. ZIV und A. LEMPEL *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory 23(3), 1977 Der Artikel, in dem die Fenstermodelle eingeführt werden.

J. ZIV und A. LEMPEL *Compression of individual sequences via variable-rate coding* IEEE Transactions on Information Theory 24(5), 1978. Der Artikel, in dem die Wörterbuchmodelle eingeführt werden.

J. J. RISSANEN und G. G. LANGDON. *Arithmetic coding.* IBM Journal on Research and Development 23(2), März 1979. Die Arithmetische Kodierung wird in diesem Artikel der Praxis erschlossen.

I. H. WITTEN, R. M. NEAL und J. G. CLEARY. *Arithmetic coding for data compression.* Communications of the ACM 30(6), 1987. Eine Einführung in Theorie und Praxis mit einem vollständigen C-Programm. Kommentierte Bibliographie.

Inhaltsverzeichnis