Technische Universität Berlin

CV–Bericht–30
Diplomarbeit

25. November 1994

Verlustfreie Datenkompression
– Überblick und Kategorisierung –

Oskar Schirmer
Matrikel Nr 133795

Kurzfassung:

Um bei der Übertragung und Speicherung von Daten Zeit und Platz zu sparen, können diese Daten komprimiert dargestellt werden.

Die Datenkompression wird als Abbildung von Dateien auf Dateien definiert. Es wird ein Schema zur Kategorisierung von Kompressionsverfahren erstellt. Dieses Schema ist nicht linear, das heißt es wird die Unterscheidung von Datenkompressionsverfahren anhand verschiedener Kriterien diskutiert. Es folgt ein Überblick über Verfahren zur verlustfreien Datenkompression, jeweils mit Kurzbeschreibung, Diskussion spezifischer Probleme und Einteilung in Kategorien.

Einige Randbereiche und verwandte Probleme werden behandelt.

Computer Vision
Fachbereich 13
Prof. R. Klette

Technische Universität Berlin
FR 3-11, Franklinstraße 28/29
10587 Berlin

Inhaltsverzeichnis

1 Einleitung

Wenn man den Duden [DUDEN] zur Hand nimmt und das Stichwort *Rakete* nachschlägt, trifft man etwa auf folgenden Eintrag:

> Ra|ke̲|te, die, _, _n (Feuerwerkskörper, Flugkörper, Geschoß) ⟨*germ* → *ital*⟩. *Zus K 365 ff.*: Raketen‿treibstoff usw. · Ra|ke̲|ten‿an|trieb, ...trieb|werk

Das sieht ein wenig wirr aus, so wie es da steht, aber man liest etwas anderes, unter Weglassung der Trennzeichen etwa folgendes:

> Rakete, die, Rakete, Raketen (Feuerwerkskörper, Flugkörper, Geschoß), Herkunft von germanisch nach italienisch. Zusammensetzung(en) Kennzahl 365 und folgende: Raketentreibstoff und so weiter. Raketenantrieb, Raketentriebwerk.

Das sieht so schon leserlicher aus, bis auf die „Kennzahl 365 und folgende", hinter denen sich diverse Regeln „zur Zusammensetzung von Substantiven, Adjektiven usw." verbergen, die knapp zwei Seiten des Anhanges des Dudens ausmachen. Um diese leserlichere Form zu erhalten, wurden diverse Regeln zur Anwendung gebracht, die in der Einleitung des Dudens mehr oder weniger vage beschrieben sind. Das sind zunächst Abkürzungen, zu finden in der Übersicht über die verwendeten Sprachbezeichnungen (Einführung 8.4.: germ, ital), weiterhin in der Übersicht der im Wörterverzeichnis und Leitfaden verwendeten Abkürzungen (Einführung 9.4.: Zus, K, usw.) und im Verzeichnis von Abkürzungen, Einheitenzeichen und Zeichen (Anhang: ff.). Außerdem werden eine Reihe von Zeichen verwendet, die ihre Bedeutung in einer Liste in der vorderen Einbandinnenseite zugewiesen bekommen. Hier finden sich unter anderem folgende Erläuterungen:

- _ Der waagrechte Strich auf der Zeile vertritt das unveränderte Stichwort.
- .. Zwei Punkte ersetzen in den grammatischen Angaben, bei der Silbentrennung und in den Ausspracheangaben den fehlenden Teil des Stichwortes, usw.
- ... Drei Punkte stehen für den fehlenden Teil eines zusammengesetzten Stichwortes oder auch eines Satzes innerhalb eines Fügungsbeispiels, usw.

In diesen Sätzen sind vor allem die Verben wichtig: *vertreten, ersetzen, stehen für*: Die Zeichen werden dazu verwendet, ganze Worte zu ersetzen, die beim Lesen wieder hinzugedacht werden müssen, ebenso wie bei den verwendeten Abkürzungen, die ursprünglich längere Worte ersetzen. Die Antwort auf die Frage, wieso dieser Aufwand des Ersetzens getrieben wird, wenn doch das so Gedruckte effektiv schwerer zu lesen ist, ist klar: Es soll Platz gespart werden, und damit Papier.

Was vorliegt, ist eine rudimentäre Form der Datenkompression[1]. Was beim Lesen geschieht ist eine Dekompression, was die Autoren des Duden beim Schreiben getan haben ist die Kompression.

Ein anderes Beispiel, die Zutatenliste auf einer Packung Schokoladensoßenpulver:

> Zucker, stark entöltes Kakaopulver, modifizierte Stärke, Verdickungsmittel, naturidentische Aromastoffe.

Worauf es hier ankommt, sind die Inhaltsstoffe selbst, nicht der Text an sich. Das Alphabet ist in diesem Falle die Menge der Zutaten in einer Küche. Betrachtet werden soll die letzte Angabe: naturidentische Aromastoffe. Warum werden für die Herstellung von Schokoladensoßenpulver naturidentische Aromastoffe verwendet und nicht natürliche? Ganz offensichtlich sind die lebensmittelchemisch erzeugten billiger in der Herstellung als die natürlichen in der Gewinnung, was im Effekt einen Wettbewerbsvorteil im System der Marktwirtschaft bedeutet. Wieso aber wurden die natürlichen Aromastoffe durch naturidentische ersetzt, und nicht einfach durch noch weitaus billigeren Essig oder gar weggelassen? Der springende Punkt, auf den es hier ankommt, ist, daß Essig in Schokoladensoßenpulver den erwarteten Geschmack verfälscht, der Verbraucher schmeckt den Unterschied sofort. Den Unterschied zwischen naturidentischen und natürlichen Aromastoffen bemerkt hingegen niemand, immer voraus gesetzt, daß der künstliche ein Imitat des natürlichen ist. Es können also bestimmte Zutaten durch andere ersetzt werden, solange dadurch der Geschmack nicht oder nicht wesentlich verfälscht wird: Es kommt nicht auf die Zutaten an, auch nicht auf den Namen des Produzenten oder das Herstellungsland, sondern auf den Geschmack: Der Geschmack ist relevant, andere Faktoren sind irrelevant. Die irrelevanten Anteile können weggelassen oder zwecks Optimierung verändert werden.

[1] Rudimentär deshalb, weil die Regeln zur Wiederherstellung des Gedruckten in den zu lesenden vollständigen Text nicht immer eindeutig sind. Deutlich wird dies bei anderen Stichworten, hauptsächlich für das Kürzel „..“ in den grammatischen Angaben, beispielsweise zum Stichwort „Ergebnis, das, ..isses, ..isse“ usw.: Es ist natürlich *klar*, daß der Genitiv „Ergebnisses“ heißt, aber gemäß Regelwerk könnte zum Beispiel auch „Ergebisses“ gelesen werden.

Wenn es sich bei diesem Beispiel nun nicht um Materie sondern um Information handeln würde, wäre es ein weiteres Beispiel für Datenkompression. Was dieses Beispiel von dem ersten grundlegend unterscheidet, ist die Entscheidungsgrundlage, aufgrund derer eine Optimierung vorgenommen werden kann: Beim ersten Beispiel läßt sich die gesamte darzustellende Information um gewisse Anteile reduzieren, die ohne Verlust wiederhergestellt werden können; diese Anteile werden als *redundant* bezeichnet. Beim zweiten Beispiel wird auf gewisse Qualitäten verzichtet; die Anteile, die diese Qualitäten beschreiben, können weggelassen oder verändert werden, sie werden als *irrelevant* bezeichnet.

2 Ein Modell für verlustfreie Datenkompression

Da im Wesentlichen verlustfreie Datenkompression betrachtet wird, werden hier einige Definitionen gegeben. Dabei ist zu beachten, daß diese Definitionen nur ein mögliches Modell darstellen, es sind auch andere Herangehensweisen vorstellbar. Insbesondere gilt, daß der folgende Ansatz nichts über die bei der Datenkompression zur Anwendung gelangenden Algorithmen aussagt: Es werden nur Quelle und Ergebnis betrachtet. Wie man vom einen zum anderen gelangt wird hier nicht untersucht, sondern erst bei der Beschreibung der einzelnen Verfahren.

Zunächst sollte der Begriff $Daten$ präzisiert werden. Daten bestehen aus einer Folge diskreter Zeichen, die ihrerseits einer endlichen $Grundmenge$ angehören. Die Länge dieser Folge kann bestimmt, also zum Zeitpunkt der Verarbeitung oder Übertragung fest und bekannt, oder aber unbestimmt sein. Im Falle einer Folge von Zeichen von bestimmter Länge heißen die Daten $Datei$, im Falle einer Folge unbestimmter Länge $Datenstrom$.

Als Grundmenge sind zum Beispiel denkbar die Menge der Bytes G=$\{$0,1, ...,255$\}$, der ASCII-Satz G=$\{$`NUL`,...,`@`,A,B,...$\}$ oder die Menge der Gennukleotide G=$\{$Adenin,Cytosin,Guanin,Thymin$\}$. Der Einfachheit halber soll im folgenden als Grundmenge immer die Menge der Bits G=$\{$0,1$\}$ angenommen werden.

G sei eine endliche Menge mit $\#G = 2$.
Def: Die Folge $d = (g_1, ..., g_n)$ heiße $Datei$ für $g_i \in G$, $n \geq 0$, falls n bestimmt ist, ansonsten heiße die Folge $Datenstrom$.

Eine $Datei$ ist also einfach eine endliche Folge von Zeichen, genau so, wie sie üblicherweise auf Disketten und Festplatten zu finden ist. Als Beispiel für einen $Datenstrom$ denke man an Datenübertragungsstrecken, etwa Modem- oder Funkstrecken: Wie bei einer Datei sind alle Zeichen der Daten linear aufgereiht, ein Zeichen folgt dem anderen, jedoch ist die Anzahl der Zeichen, die auf der Datenübertragungsstrecke aufeinander folgen, zum Zeitpunkt der Übertragung unbestimmt.

$D_n(G)$ sei die Menge aller Dateien d über G der Länge n,
$D(G) = \bigcup_{n \in N_0} D_n(G)$ somit die Menge aller Dateien d über G.

Natürlich enthält $D(G)$ auch alle Datenströme, nur ist bei diesen nicht feststellbar, in welchem der vereinigten $D_n(G)$ sie liegen. Für Dateien gebe die Funktion $length$ die Länge der entsprechenden Datei an:

Def: $length(d)$ sei die Länge einer Datei $d \in D_n(G)$:
$length((g_1, ..., g_n)) := n.$

Für Datenströme ist das Ergebnis der Funktion *length* unbestimmt. Da es sich mit einem Datenstrom unbestimmter Länge schlecht rechnen läßt, von einem Datenstrom aber, wenn überhaupt schon ein Teil bekannt ist, dann dieser der Anfang des Datenstroms ist, soll eine Funktion eingeführt werden, die von einem Datenstrom die ersten k Zeichen liefert. Diese Funktion heiße *prefix*:

$k \in N_0,\ d = (g_1, g_2, ...) \in D(G)$:
Def: $\mathit{prefix} : N_0 \times D(G) \to D(G)$:
Falls $0 \leq k \leq \mathit{length}(d) : \mathit{prefix}(k,d) := (g_1, ..., g_k)$,
sonst $k > \mathit{length}(d) : \mathit{prefix}(k,d) = d$.

Das Ergebnis ist also immer, auch für Datenströme d, eine Datei, es sei denn, k ist größer als der bereits bekannte Teil von d.

Unter diesen Voraussetzungen läßt sich *Datentransformation* als Funktion beschreiben, die die Eingabedaten der Transformation auf die Ausgabedaten abbildet:

Def: $f : D(G) \to D(G)$ heißt *Datentransformation*, falls f *injektiv*[2] ist.
$F(G,G)$ sei die Menge aller Datentransformationen von $D(G)$ nach $D(G)$.

Man beachte, daß die Funktion f nicht einzelne Zeichen der Dateien aufeinander abbildet, sondern jeweils eine komplette Eingabedatei auf eine komplette Ausgabedatei, am Stück. f ist also eine echte Funktion, nicht etwa ein Automat. Fitingof [FI] stellt genau dieses Modell vor, nur bezeichnet er die Datentransformation als *code mapping*, die von einer Menge von Nachrichten X auf eine Menge von Codes Y abbildet.

$f \in F(G,G)$ heiße *Datenkompression*, wenn es für jedes übliche, meist lange $d \in D(G)$ eine Schranke $M_d > 0$ gibt, so daß $\forall m \geq M_d$ gilt:
$\mathit{length}(f(\mathit{prefix}(m,d))) < \mathit{length}(\mathit{prefix}(m,d))$.

Da Daten auch unbestimmte Länge haben können, muß hier mit einer Schranke operiert werden. Für Dateien kann trivialerweise $M_d = \mathit{length}(d)$ gewählt werden, so daß sich die Ungleichung reduziert auf
$\mathit{length}(f(d)) < \mathit{length}(d)$.

Durch die Forderung nach der Injektivität von f wird sichergestellt, daß zu jedem $f(d)$ das d wiederherstellbar ist, daß heißt es gibt zu $f \in F(G,G)$ ein $f' \in F(G,G)$ mit $f'(f(d)) = d$. Diese Gleichheit entfällt bei der verlustbehafteten Datenkompression und es wäre an dieser Stelle eine *Fehlerfunktion* für den Vergleich von $f'(f(d))$ und d zu definieren: $\delta_f : D(G) \to R_0^+$.

Um die Wiederherstellbarkeit von Datenströmen auch im praktischen Fall zu gewährleisten, wird gefordert, daß für $f \in F(G,G)$: $\forall S$: $\exists S'$: $\forall d, d' \in D(G)$:
$\mathit{prefix}(S, f(d)) = \mathit{prefix}(S, f(d')) \Rightarrow \mathit{prefix}(S', d) = \mathit{prefix}(S', d')$.

[2] eineindeutig: $\forall d, d' \in D(G)$: $f(d) = f(d') \Rightarrow d = d'$ [BS,p.550]

Wenn also die Daten nur bis zu einer bestimmten Schranke S komprimiert vorliegen, so kann trotzdem die Wiederherstellung der Daten bis zu einer Schranke S' erfolgen, und dies gilt für alle Schranken S. Ohne diese Forderung müßte gegebenenfalls zur Bestimmung von g_1 das gesamte $f(d)$ vorliegen, dessen Länge ist aber unbestimmt.

In der Begriffsdefinition der Datenkompression fällt der vage Begriff *üblich* auf. Dieser ist notwendig, da ist nicht möglich ist, eine allumfassende Datenkompression zu finden, mit der für alle $d \in D(G)$ gilt:
$length(f(prefix(m, d))) < length(prefix(m, d))$.

Satz: $\forall f \in F(G, G)$ entweder:
$\exists d \in D(G) : length(f(d)) > length(d)$
oder[3]: $\forall d \in D(G) : length(f(d)) = length(d)$

Beweis: $n \geq 0$:
$c_G(n) := \#D_n(G) = (\#G)^n > 0$ die Anzahl der Dateien mit $length(d) = n$.
$s_G(n) := \sum_{i=0}^{n} c_G(i) = \frac{\#G^{n+1}-1}{\#G-1}$ die Anzahl der Dateien mit $length(d) \leq n$.
Annahme: $\exists f \in F(G, G)$:$\forall d \in D(G)$: $length(f(d)) \leq length(d)$
und $\exists d_0 \in D(G)$: $length(f(d_0)) < length(d_0)$.
$n_0 := length(f(d_0))$
$\#\{d | length(f(d)) \leq n_0\} > s_G(n_0) \Rightarrow$
$\exists d, d' \in D(G), d \neq d' : f(d) = f(d') \Rightarrow$
f ist nicht injektiv $\Rightarrow$ f ist keine Datentransformation $\Box$

Es gibt also keine allumfassende Datenkompression, vielmehr gibt es zu jedem verlustfreien Kompressionsverfahren Daten, die nicht komprimiert werden können. Diese Daten können bei Kenntnis der Funktionsweise des Kompressionsverfahren auch konstruiert werden durch Auswahl des jeweils maximal unerwarteten nächsten Zeichen.

Ähnliche Beweise wie der obige wurden auch mehrfach im Zusammenhang mit dem sogenannten *WEB 16:1* Kompressionsverfahren im internet vorgeführt. Dabei hatte ein US-amerikanisches Unternehmen April 1992 behauptet „alle Typen binärer Dateien“ von mindestens 64 kB „auf etwa ein Sechzehntel der ursprünglichen Größe“ zu komprimieren, verlustfrei. Dadurch könne „nahezu jede Menge von Daten auf unter 1024 Bytes komprimiert werden, indem der Kompressor wiederholt auf seine eigene Ausgabe angewendet“ werde. Das ist jedoch nach obigem Beweis unmöglich, und so mußte der zweite Verkaufsleiter

[3] Im zweiten Falle heißt f „Permutation über G“

der Firma Anfang August 1992 mitteilen, daß „die Programmierer nicht in der Lage seien, das Problem im Zusammenhang mit einer Matrix, in der vier gleiche Werte stünden, zu lösen“ und daher das Produkt absetzen [FAQ].

Falls nun der Kompressor eine *unübliche* Datei vorgesetzt bekommt, die er nicht zu komprimieren imstande ist, sollte verhindert werden, daß das Ergebnis allzu lang wird. Der folgende Satz konstruiert aus einem gegebenen Datenkompressionsverfahren f ein solches Verfahren f', für das keine Zieldatei um mehr als ein bit länger ist als die Quelldatei:

Satz: $\forall f \in F(G,G) : \exists f' \in F(G,G) :$
$\forall d \in D$: $length(f'(d)) \leq length(d) + 1$
und $length(f'(d)) \leq length(f(d)) + 1$

Beweis: $c_0, c_1 \in G, c_0 \neq c_1$.
Konstruiere f' wie folgt:
falls $length(f(d)) < length(d)$: $f'(d) := (c_0, f(d))$
sonst: $f'(d) := (c_1, d)$ $\square$

Außerdem ist es natürlich möglich, beliebig hohe Kompressionsraten zu erreichen, indem ganz bestimmte lange, vielleicht feste Zeichenfolgen durch kurze Codes dargestellt werden. Der Extremfall ist erreicht, wenn eine bestimmte Datei der Länge n als *die besondere* ausgewählt wird, die als einzelnes 0-Bit codiert wird, während alle anderen Dateien unter Vorausschickung eines 1-Bit im übrigen unverändert abgelegt werden. Für die eine Datei ergibt sich somit eine extreme Kompressionsrate von $1 : n$, aber eben nur für diese eine, jede andere Datei wird länger. Auch wenn dieses Beispiel realitätsfern erscheint, so zeigt es doch, daß die Behauptung, mit verlustfreien Verfahren ließen „sich heute Kompressionsfaktoren von maximal 1:4 erreichen“ [CM], jeder Grundlage entbehrt. Die erreichbaren Kompressionsraten hängen vielmehr stark von der Beschaffenheit der zu komprimierenden Daten ab, sowie davon, wie gut das Kompressionsverfahren auf diese Beschaffenheit abgestimmt ist.

3 Redundanz und Irrelevanz

In der Einleitung wurden bereits zwei Beispiele gegeben, die sich genau in diesem Kriterium unterscheiden. Im ersten Fall beruht die Möglichkeit zu Komprimieren darauf, daß die darzustellenden Daten Regelmäßigkeiten aufweisen, die auf verschiedene Weisen codiert werden können, so daß die *redundanten* der Daten entfallen. Beim zweiten Beispiel hängt die Möglichkeit zu Komprimieren nur von der *Erwartungshaltung* des *Rezipienten* ab, es werden also Annahmen darüber erstellt, wie die Daten verändert und vereinfacht werden können, ohne den *relevanten* Anteil der darzustellenden Information zu verändern.

Es soll hier unterschieden werden zwischen den Begriffen *Daten* und *Information*, wobei Information der Inhalt der Daten ist, Information also nur durch die *Interpretation* von Daten entsteht, während Daten frei und nur frei von Interpretation bestehen. Daten, die interpretiert wurden, sind keine Daten mehr, sondern sie sind vollständig in Information übergegangen. Die *Interpretation* ist der Vorgang der Verknüpfung der beim Rezipienten eingehenden Daten mit der beim Rezipienten bereits vorliegenden Erwartungshaltung, sie ist also eine Funktion (Bild 1). Die erhaltene Information verändert natürlich ihrerseits wieder die

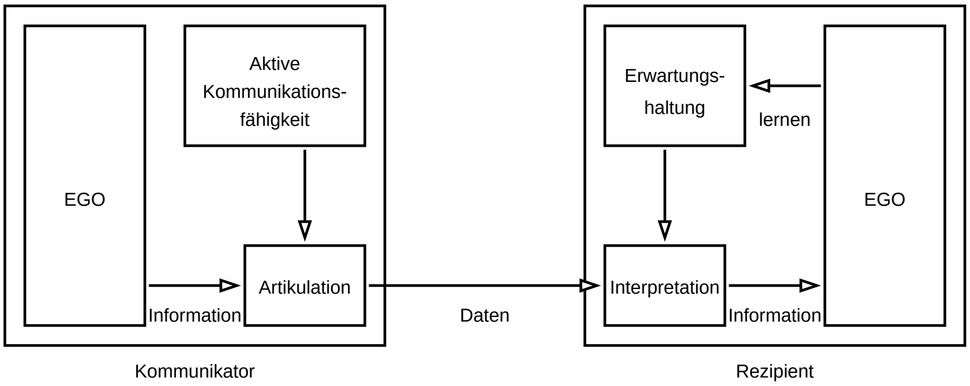

Bild 1

Erwartungshaltung gegenüber weiteren eingehenden Daten, dies stellt den Lernvorgang dar. Die Erwartungshaltung wird auch Vorwissen genannt, wobei dieser Begriff ungünstig erscheint, da er einen direkten Bezug zum Bewußtsein suggeriert, während der beschriebene Vorgang nicht auf dieses beschränkt ist, sondern natürlich auch unbewußt ablaufen kann [ELL]. Ebenso kann auch auf der Seite des *Kommunikators* der Übergang von Information zu Daten dargestellt werden als Verknüpfung der zu vermittelnden Information mit der aktiven Kommunikationsfähigkeit des Kommunikators zu Daten. Diese Verknüpfung heiße *Artikulation*. Dieses Modell des Monologs ist sowohl nach außen unvollständig, da zum Beispiel eine Rückwirkung der vom Kommunikator ausgehenden Daten auf den Kommunikator selbst und damit über dessen Interpretation auf seine aktive Kommunikationsfähigkeit fehlt, und es ist nach innen unvollständig, da die beiden im Bild als *Ego* bezeichneten Bereiche nicht näher beschrieben sind.

Wieder zur Datenkompression: Die auf Redundanzuntersuchungen basierende Kompression beschränkt sich auf die Strecke zwischen Artikulation und Interpretation, es ist also eine echte Datenkompression. Die auf Relevanzuntersuchungen basierende Kompression hingegen liefert dem Rezipienten im allgemeinen andere Daten als die vom Kommunikator gesendeten, jedoch derart, daß bei einer Interpretation dieser Daten dieselbe Information entsteht wie bei einer Interpretation der ursprünglichen Daten bei gleicher Erwartungshaltung. Wird diese Kompression als *Informationskompression* bezeichnet, so ist dieser Begriff nicht ganz zutreffend, da nur in die Strecke der Datenübertragung eingegriffen wird. Auch der Begriff *relevanzerhaltende Datentransformation* ist nicht einwandfrei, da auch die Datenkompression eine relevanzerhaltende Datentransformation ist, nur ist diese zwingend injektiv, was sie auszeichnet gegenüber der Informationskompression.

Ein Mißverständnis, das im Zusammenhang mit dem eingangs angeführten Beispiel aus dem Duden aufkommen kann, besteht in der Möglichkeit, in diesem Beispiel auch eine Informationskompression zu sehen. Dies ist natürlich richtig unter der Voraussetzung, daß alle Stufen der Verarbeitung der komprimierten Daten durch den Rezipienten als Interpretation betrachtet werden. Es genügt jedoch, den Teil des Rezipienten, der mit der Expandierung der Abkürzungen zu vollen Worten beschäftigt ist, von dem eigentlich interpretierenden Teil des Rezipienten des so Gelesenen abzutrennen und als nicht zum Rezipienten gehörig zu betrachten, um das Modell der exakten Datenkompression zu erhalten. Auf der Seite des Kommunikators wird dies deutlicher, da die Vorstellung nicht schwer fällt, daß in der Redaktion des Duden erst alle Texte im expandierten Zustand geschrieben werden um dann vor Layout und Druck — womöglich automatisch — in die komprimierte Form transformiert zu werden.

3.1 Verlustfreiheit

Das soeben beschriebene Kriterium, das danach unterscheidet, ob der Rezipient die ursprünglichen Daten oder sein Ego die ursprüngliche Information erhalten soll, wird in der Praxis meist durch das Kriterium der *Verlustfreiheit* ersetzt — obwohl es sich mit diesem nicht unbedingt decken muß — und ist damit in seiner grundlegenden Bedeutung schwer zu erkennen. Datenkompression im hier vorgestellten Sinne ist zwar immer verlustfrei und verlustbehaftete Kompression kann immer nur Informationskompression sein, der jeweilige Umkehrschluß trifft jedoch nicht zu. Dennoch kann jede verlustfreie Informationskompression als Datenkompression eingesetzt werden. Aus diesem Grunde wird die Datenkompression im allgemeinen als *verlustfreie* (lossless) oder *exakte* Datenkompression und die Informationskompression als *verlustbehaftete* (lossy) Datenkompression bezeichnet. Auch in dieser Arbeit wird im folgenden zwischen verlustfreier und verlustbehafteter Datenkompression unterschieden.

Eine mögliche Betrachtungsweise des Kriteriums *Verlustfreiheit* besteht darin, eine Funktion $\delta_f : D(G) \to R_0^+$ zu finden, die den Unterschied zwischen den Originaldaten und den wiederhergestellten Daten einer verlustbehafteten Datenkompression derart beschreibt, daß umso größere Werte eine größere Abweichung bezeichnen. Diese Funktion wird *Fehlerfunktion* genannt, und die Kategorie der verlustfreien Datenkompressionen wird als Spezialfall der verlustbehafteten mit $\forall d \in D : \delta_f(d) = 0$ betrachtet [ST,p.2].

Eine andere Sicht der Dinge wäre die Annahme, daß jedes verlustbehaftete Datenkompressionsverfahren im Grunde aus zwei Phasen besteht, von denen die erste eine Datenreduktion darstellt (eine Auswahl einzelner Daten oder das Bündeln der möglichen Eingabedaten zu Clustern) und die zweite eine verlustfreie Datenkompression. Die Datenreduktion ist dabei eine nicht injektive Funktion, die Datenkompression ist injektiv.

Da die verlustbehafteten Kompressionsverfahren im allgemeinen auf Irrelevanzuntersuchungen der Daten beruhen, die verlustfreien hingegen auf Redundanzuntersuchungen, werden diese beiden Bereiche nach Art der jeweils betrachteten Irrelevanz beziehungsweise Redundanz aufgegliedert.

3.2 Irrelevanzkriterien bei verlustbehafteter Datenkompression

Verlustbehaftete Datenkompressionsverfahren können immer dann eingesetzt werden, wenn über das tatsächliche oder gewünschte Interpretationsverhalten der möglichen Rezipienten Aussagen getroffen werden können, da es dann möglich ist, zu beurteilen, welche Information der Rezipient aus den Daten gewinnen kann oder soll. Der Anschaulichkeit halber wird angenommen, daß es sich beim Rezipienten um einen Menschen handelt, und daß die zur Datenaufnahme verwendeten Organe das Auge und das Ohr sind. Daß hier nur Auge und Ohr betrachtet werden, liegt daran, daß die anderen Sinnesorgane des Menschen datenkommunikationstechnisch so gut wie nicht erschlossen sind. Ausnahmen könnte man hier in den Versuchen zum Riechfilm[4] sehen, wobei sich das Datenaufkommen hier auf wenige indizierte Gerüche beschränkt, und im Flugsimulator, der Daten in Eindrücke auf den Gleichgewichtssinn des Menschen umwandelt, also ein vestibuläres Datenendgerät ist.

Außerdem entwickeln Sehbehinderte aufgrund ihrer eingeschränkten Fähigkeit zu sehen den Tastsinn zu einem wesentlich ausgereifteren Stadium. Während die meisten normal Sehenden auf der Fingerkuppe kaum mehr spüren als eine einzelne relative Druckempfindung, können viele Blinde die Blindenschrift, deren jedes Zeichen aus sechs Positionen besteht, an denen sich je eine Erhebung befinden kann, in hoher Geschwindigkeit ertasten. Dementsprechend werden taktile Datenendgeräte entwickelt, die über mechanisch in der Höhe variable Pixel zur Anzeige von Schrift und Bildern verfügen. Nöth [NÖ,p.364] schließlich erwähnt den Optohapten, durch den die Information „in Vibrationsimpulse an neun Körperstellen übertragen und somit taktil gelesen werden kann".

3.3 Irrelevanzkriterien beim menschlichen Ohr

Zunächst einmal ist die primäre Aufnahmekapazität des menschlichen Ohrs begrenzt. Frequenzen über etwa 20 kHz sind nicht hörbar, Frequenzen unter etwa 20 Hz ebensowenig, obwohl diese noch als Luftdruckschwankungen wahrgenommen werden [ME,p.254]; bei 1300 Hz kann das Ohr etwa 370 Amplitudenstufen zwischen Hör- und Überlastungsschwelle unterscheiden [ME,p.268]. Es genügt also, mit einer Abtastrate von etwa 40 kHz je zwei Werte (Stereo) zu knapp 9 bit zu übertragen [VÖ83,p.487]. Für bestimmte Anwendungen müssen aber die Grenzwerte für einfache Stereo- oder Monodarstellung nicht eingehalten werden. Bei einem Telefongespräch zum Beispiel genügt eine Abtastrate von 8 kHz, da Frequenzen über 4 kHz für das Verständnis gesprochenen Textes nicht erheblich sind [CM]. Auch 9 bit je Abtastwert sind nicht erforderlich, gesprochener Text ist sogar noch bei 1 bit je Abtastwert zu verstehen. Werden von Gesprochenem nicht die Eigenschaften der Stimme des Sprechers benötigt, sondern nur der transportierte Text, dann kann dieser Text natürlich als solcher codiert werden (z.B. ASCII oder Lautschrift) und für den Rezipienten durch einen Sprachsynthesizer reproduziert werden.

Selbst wenn es auf eine Übertragung von Ton in High-Fidelity Stereo Qualität ankommt, können viele Anteile der Daten von vornherein weggelassen werden, da die vom Ohr aufgenommenen Daten vor Eintreffen im Hörzentrum des Gehirns bereits erheblich reduziert werden (erste Stufen der Interpretation). So sind

[4] Nummerierte Riechfelder auf der Begleitkarte, die der Kinobesucher durch im Film eingeblendete Nummern zu reiben aufgefordert wird [JW]

zum Beispiel Frequenzanteile, die in der Nähe anderer, stärkerer Frequenzanteile liegen, nicht hörbar (*Verdeckungseffekt*).

3.4 Irrelevanzkriterien beim menschlichen Auge

Ebenso wie beim Ohr ist auch beim Auge die Aufnahmekapazität begrenzt. Die für die primäre Umwandlung von optischen in neuronale Reize zuständigen Stäbchen und Zapfen in der Netzhaut des Auges sind nicht gleichverteilt. Die Gesamtanzahl der helligkeitsempfindlichen Stäbchen liegt bei etwa einer Million, die Zahl der farbempfindlichen Zapfen bei etwa einer halben Milliarde [VÖ83,p.145ff]. Die Anzahl der vom Auge abgehenden Nervenfasern liegt bei etwa einer Million. Übliche Werte für die Bildwiederholfrequenz liegen zwischen 20 und 60 Hz. Es kann also die vorsichtige Rechnung angestellt werden, daß eine Obergrenze für die Kapazität des Auges bei etwa 60 Millionen Abtastwerten je Sekunde liegt, wobei die Abtastwerte nicht binär sind. Ob die Übertragung im Sehnerv tatsächlich so hoch ist, ist damit noch nicht gesagt, da ja nicht unbedingt alle Fasern die volle Kapazität von 60 Hz mittragen müssen. Für die Genauigkeit der Abtastwerte gibt Meyer-Eppler als Zahl der unterscheidbaren Helligkeitsstufen 570 an [ME,p.269], als Zahl der unterscheidbaren Farbwerte etwa 10000 [ME,p.288], die Gesamtzahl der unterscheidbaren Farb- und Helligkeitsstufen liege bei $1{,}5 \cdot 10^5$ .. $7{,}5 \cdot 10^6$. Die Abtastwerte müßten also mit etwa 17..23 bit dargestellt werden. Auch diese Genauigkeit ist wesentlich von Rahmenbedingungen abhängig, zum Beispiel sind keine 10000 Farbstufen unterscheidbar, wenn die optischen Reize von besonders kurzer Dauer oder besonders geringer scheinbarer Fläche sind.

4 Kategorisierung

Nachdem die verlustbehaftete Datenkompression von der verlustfreien getrennt worden ist, soll nun versucht werden, Kriterien aufzuzeigen, anhand derer verlustfreie Datenkompressionsverfahren untersucht und in *Kategorien* eingeteilt werden können. Dabei wird nicht eine lineare Skala von Attributen vergeben, sondern vielmehr eine Einteilung der Verfahren nach mehreren Kriterien vorgenommen.

4.1 Redundanzkriterien bei verlustfreier Datenkompression

Redundanz bedeutet, daß Teile der ursprünglichen Daten überflüssig, also in einem gewissen Maße vorhersagbar oder wahrscheinlich sind. Dies wiederum kann als Regelmäßigkeit bezeichnet werden, womit nicht unbedingt Gleichmäßigkeit oder Periodizität gemeint sein müssen. Wie diese Redundanzen aussehen und in welchem Maße sie auftreten, hängt ganz vom jeweiligen Typ der betrachteten Daten ab. Ein Versuch, diese Redundanzen zu ordnen, ist in Bild 2 zu sehen.

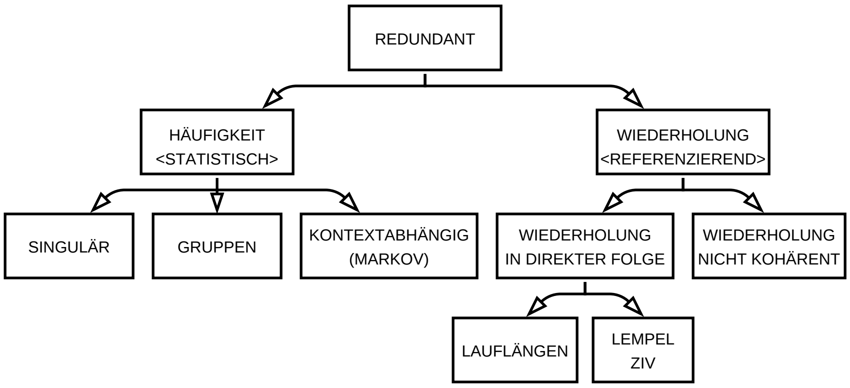

Bild 2

Dabei werden zunächst zwei Hauptgruppen von Redundanzen getrennt: Die erste Hauptgruppe ist die der *statistischen* Redundanz. Hierbei treten einige Zeichen der Eingabe häufiger auf als andere und sind so leichter vorhersagbar, da ihr Auftreten wahrscheinlicher ist. Die statistischen Redundanzen lassen sich wiederum aufteilen in solche Redundanzen, die die Häufigkeiten der Zeichen in *Abhängigkeit* der benachbarten Zeichen oder anderer Kriterien betrachten, und solche, die die Zeichen *unabhängig* betrachten. Für letzteren Typ von statistischer Redundanz gilt insbesondere, daß ein Vertauschen der einzelnen Zeichen in der Eingabe keinen Einfluß auf das Ergebnis hat:

$$c_0, c_1 \in G, c_0 \neq c_1 : \#\{i|g_i = c_0\} < \#\{i|g_i = c_1\}$$

Übliche Kompressionsverfahren, die sich auf das Vorhandensein statistischer Redundanz stützen, sind die Huffman-Codierung und die arithmetische Codierung. Beide Verfahren zählen die auftretenden Zeichen und versuchen, die häufigeren durch weniger oder kürzere Zeichen in der Ausgabe darzustellen.

Die andere Hauptgruppe ist die der *referenzierenden* Redundanz. Diese Redundanzen beschreiben das wiederholte Auftreten einer Konstellationen von Zeichen der Eingabe. Die simple Variante dieser Gruppe ist die Wiederholung einer Teilfolge der Eingabe (in diesem Text tritt beispielsweise die Teilfolge „ndanz" wiederholt auf). Diese Art von Redundanzen heißt *referenzierend*, da sich jeweils eine auftretende Konstellation von Zeichen auf eine vorhergehende bezieht:

$$0 < a < b, a < c : \forall i, a \leq i \leq b : g_i = g_{i-a+c}$$

Die betrachtete Teilfolge muß jedoch nicht unbedingt zusammenhängend sein:

$$d > 0, a_i > 0 : \forall i, 0 \leq i < n : g_{a_i} = g_{a_i+d}$$

Wichtig für den Unterschied der referenzierenden Redundanz zur statistischen ist, daß bei der referenzierenden Redundanz die aktuelle Teilfolge gegenüber *einem* vorherigen Auftreten dieser Teilfolge direkt redundant ist, während bei der statistischen Redundanz die Häufigkeit des Auftretens ausschlaggebend ist.

Eine primitive, eingeschränkte Variante der referenzierenden Redundanz ist das wiederholt in Folge auftretenden Zeichen, die *Lauflänge*:

$$0 < a < b : \forall a < i \leq b : g_i = g_a$$

Einen systematischen Überblick über Lauflängenverfahren geben Thuner und Otter [TO], sie variieren die Parameter Musterlänge, Muster, Zählerlänge und Längenfeld jeweils nach variabel und fest (wobei Zählerlänge und Längenfeld mit Redundanzbetrachtungen eher nichts zu tun haben). Dabei werden vor allem Einschränkungen der Wahlfreiheit von $0 < a < b$ und des g_i beschrieben. Außerdem beschränken Thuner und Otter die Lauflängenverfahren nicht auf einzelne sich wiederholende Zeichen, sondern lassen auch sich direkt wiederholende Gruppen zu. Das Lauflängenverfahren wird hier nur deshalb gesondert erwähnt, weil es historisch gesehen eines der ersten angewandten Verfahren ist, was wiederum daran liegt, daß es ein sehr offensichtlicher Spezialfall von Redundanz ist. Held [HE] gesteht den Lauflängenverfahren noch ein Kapitel von sechs Seiten zu.

4.2 Strategien der Redundanzanpassung

Steht ein Datenkompressor vor der Aufgabe, Daten zu komprimieren, hat er mehrere Möglichkeiten, seine Annahmen über mögliche Redundanzen und die tatsächlich auftretenden in Einklang zu bringen (Bild 3). Die eine extreme

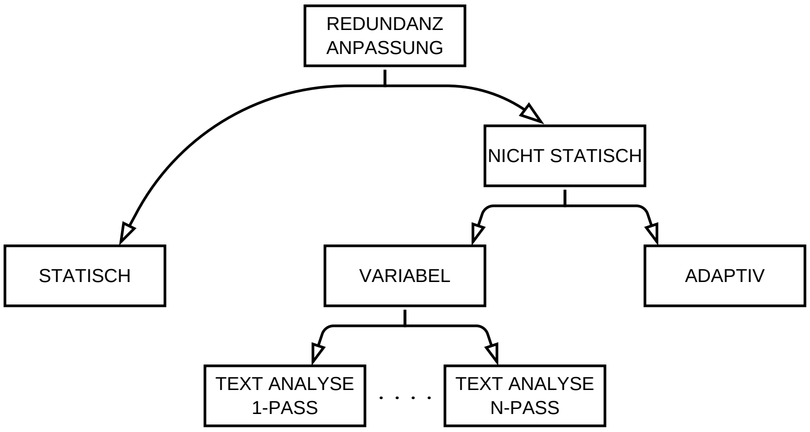

Bild 3

Möglichkeit besteht darin, den Einklang nicht zu suchen, sondern nur einen festen Typ von Redundanzen anzunehmen. Diese *statische Strategie* ist am einfachsten zu realisieren, jedoch können Diskrepanzen zwischen Annahmen und Wirklichkeit zur Wirkungslosigkeit des Kompressors führen. Soll der Kompressor seine Strategie den zu komprimierenden Daten anpassen, kann er entweder die Daten erst komplett analysieren und anhand der so gewonnenen Werte ein optimales Verfahren auswählen (*variable Strategie*), oder er kann während des Lesens und Analysierens die bisherige Auswertung als Strategiegrundlage verwenden und für die folgenden Daten anpassen (*adaptive Strategie*, auch *dynamisch* genannt).

Während sowohl die statische als auch die adaptive Strategie zum Einsatz in einem Kanal taugen, also zur Kompression von Datenströmen, ist für die variable Strategie ein *wahlfreier Zugriff* auf die Eingabedatei erforderlich, da sie mindestens zweimal vollständig gelesen werden muß[5]. Wird die Datei bei der variablen Strategie mehrfach jeweils komplett sequenziell gelesen, dann wird die Strategie je nach Anzahl der erforderlichen Analysepässe in die Kategorien

[5] Die Datei kann auch in einen Buffer gelesen werden, dann ist aber wahlfreier Zugriff auf den Buffer erforderlich.

1-Paß-Verfahren, allgemein *n-Paß-Verfahren* und *Mehrpaßverfahren* eingeteilt. Ansonsten kann nur von *wahlfreiem Zugriff* gesprochen werden.

Auch die adaptive Strategie läßt sich weiter kategorisieren, und zwar nach der *Dynamik* der Anpassung des Kompressors an die sequentiell gelesenen Daten. Bei *schwach adaptiven Strategien* wird die Anpassung erst nach jeweils einer ganzen Reihe gelesener Daten vorgenommen, bei *stark adaptiven Strategien* werden alle Parameter des Kompressors nach jedem gelesenen Zeichen angepaßt.

Die Klassifizierung der Strategie erfolgt auf *verschiedenen Ebenen*. Ein Kompressionsverfahren, das zwar nur statistische Redundanzen behandelt, diese jedoch in Abhängigkeit des bisher sequentiell gelesenen Datenstroms, wendet zwar bei der Anpassung an die Redundanz der Eingabe bezüglich der Wahl der Redundanzgruppe eine statische Strategie an, bezüglich der Wahl der Parameter innerhalb des Verfahrens aber eine adaptive Strategie. Ein ideales Anschauungsbeispiel einer in allen Hinsichten statischen Strategie ist die Lauflängenkompression durch das Tabulatorzeichen HT des ASCII. Es ist sowohl in der Wahl des Verfahrens statisch (nur Lauflängen) als auch in den Parametern (nur Leerzeichen SP werden komprimiert und nur jeweils eine feste Länge, die abhängig von der relativen Position zum letzten Zeilenanfang oder Tabulator ist). Komplexe Archivierprogramme wenden dagegen meist auf mehreren Ebenen variable Strategien an, sie wählen anhand der Struktur der Daten verschiedene Verfahren aus, die wiederum mit verschiedenen Parametern zum Einsatz kommen. Ein Kompressor, der Daten blockweise entgegennimmt und die Blöcke dann erst analysiert und danach komprimiert, ist auf höherer Ebene schwach adaptiv (Anpassung bezüglich der Wahl des Kompressionsverfahren je Block), auf mittlerer Ebene variabel (Voranalyse der Blöcke) und auf unterster Ebene je nach aktuell gewähltem Verfahren statisch, variabel oder adaptiv.

4.3 Analyse und Synthese

In den einleitenden Definitionen ist die Datenkompression als Teilmenge der Datentransformationen, also der Abbildungen von Eingabedaten auf Ausgabedaten, dargestellt worden. Die Redundanzanalyse (*Textanalyse*) befaßt sich lediglich mit den Eigenschaften der Eingabedaten. Was zur vollständigen Datenkompression also noch fehlt, ist der Part, der die Ausgabedaten generiert (Bild 4). Diese *Codesynthese* nutzt die Ergebnisse der Textanalyse, um nach Möglichkeit eine optimale Darstellung des Codes zu erreichen. Auch die Funktion der Dekomprimierung des Codes teilt sich in zwei Teilschritte, die *Codeanalyse* und die *Textsynthese*. Während die Eingabe des Analyseteils und die Ausgabe des Syntheseteils jeweils Daten sind, bestehen zwischen Analyse und Synthese eventu-

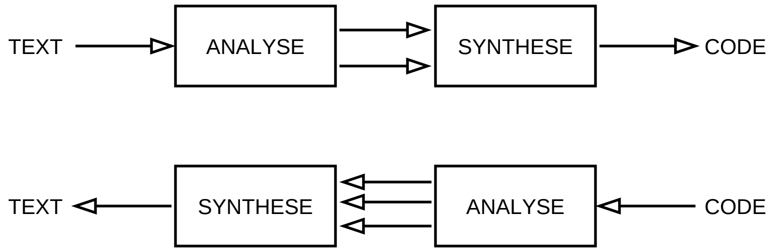

Bild 4

ell mehrere Verbindungen. Für eine variable statistische Datenkompression mit Huffman-Codierung zum Beispiel werden von der Analyse an die Synthese nicht nur die Daten weitergegeben, sondern auch ein Codebaum, der die Grundlage für die Codierung ist.

4.4 Symmetrie und Asymmetrie von Analysen und Synthesen

Auf Unterschiede in den Algorithmen zur Analyse und Synthese von Text und Code gründet sich das folgende Klassifizierungskriterium, die Einteilung in *symmetrische* und *asymmetrische* Verfahren. Kompression und Dekompression lassen sich in je zwei Teile teilen, bei der Kompression sind dies die Textanalyse und die Codesynthese, bei der Dekompression die Codeanalyse und die Textsynthese. Da der Code zweckmäßigerweise meist so gestaltet wird, daß sowohl Codesynthese als auch Codeanalyse relativ einfach bleiben, sind diese meist symmetrisch. Die Textsynthese ist meist entweder der umgekehrte Vorgang der Textanalyse (Bild 5) oder die Ausführung der durch die Codeanalyse gegebenen Anweisun-

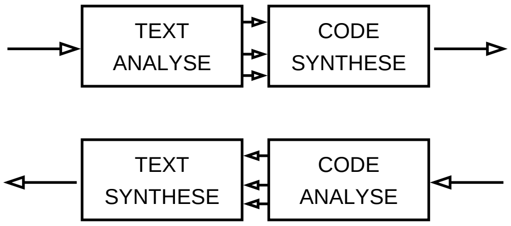

Bild 5

gen, während diese Anweisungen in der Textanalyse gefunden werden mußten (Bild 6). Im ersten Fall (zum Beispiel Lauflängen) sind Kompression und Dekompression im Aufwand symmetrisch, im zweiten Fall (zum Beispiel Huffman

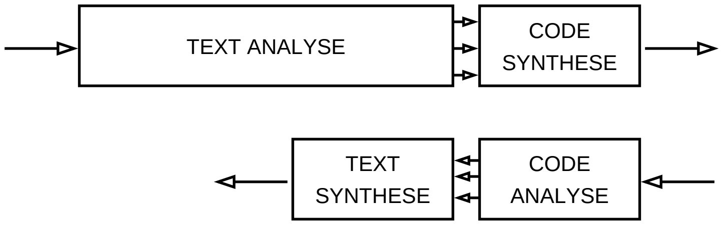

Bild 6

mit variabler Redundanzanpassung) asymmetrisch. Siehe auch den Abschnitt 14 „Verkehrt asymmetrische Datenkompression“.

4.5 Codierung

Datenkompression wurde im Abschnitt 2 „Ein Modell für verlustfreie Datenkompression“ als Abbildung von Quelldaten auf Zieldaten definiert. Da die Daten in der Praxis jedoch viel zu lang sind, und die Anzahl der möglichen Dateien in vielen Fällen unendlich ist, kann die Datenkompression so nicht programmiert werden. Statt dessen werden die Quelldaten in kürzere Sequenzen unterteilt. Diese Sequenzen können die einzelnen Zeichen des Quellalphabets sein, sie können aber auch aus mehreren dieser Zeichen bestehen. Es ist Aufgabe der Textanalyse, die Quelldaten in solche Sequenzen zu zerlegen. Die nun folgende *Codierung* ist der erste Teil der Codesynthese. Die Codierung bildet die Sequenzen aus den Quelldaten nun auf Codeworte ab, also auf Sequenzen von Zeichen des Zielalphabets. Der zweite und vergleichsweise einfache Teil der Codesynthese ist es, die sich ergebenden Codeworte wieder zur Zieldatei zusammenzufügen.

Die Quelldaten werden also in eine Anzahl Sequenzen zerlegt, die auf Codeworte abgebildet werden. Die Anzahl der Sequenzen in Quelldaten und Zieldaten sind demnach gleich. Je nachdem, ob die betreffenden Sequenzen der Quelle beziehungsweise des Ziels alle gleichlang sind oder aber unterschiedlich lang, spricht man von Codes fester Länge (*fixed length*) beziehungsweise variabler Länge (*variable length*). Diese Unterscheidung tritt für die Kategorisierung einer Datenkompression zweimal auf, einmal bezüglich der Quelldaten und einmal bezüglich der Zieldaten. Die Huffmancodierung beispielsweise ist eine Codierung von fester nach variabler Länge, also *fixed-to-variable*. Die Codebook-Verfahren nach Lempel und Ziv bilden Sequenzen variabler Länge auf Zeichen fester Länge ab, also *variable-to-fixed* [ZI]. Mitunter wird statt von fixed-length von *block* gesprochen [ZL77]. Eine Codierung *variable-to-variable* entsteht zum Beispiel bei der Verkettung eines Codebook-Verfahrens mit einer Huffman-Codierung.

Abgesehen davon gibt es auch Verfahren, die sich nicht in dieses Schema pressen lassen, wie zum Beispiel die arithmetische Kompression. Diese ähnelt zwar der Huffman-Codierung, kann aber nicht einfach mit fixed-to-variable charakterisiert werden, da in der Zieldatei keine klar abgegrenzten Zeichen (kein Blockcode, siehe unten) erzeugt werden, oder, anders betrachtet, da nicht jeweils eine Quellsequenz auf ein Codewort abgebildet wird. Die Klassifizierung von Datenkompressionsverfahren nach fixed- und variable-length kann daher nicht als vollständig angesehen werden.

Je einfacher die Struktur eines Codes ist, desto schneller kann er verarbeitet werden. Da man sich die Struktur der Quelle schon nicht aussuchen kann, wird bei der Wahl der Zielstruktur, also des Codes, auf Einfachheit und Eindeutigkeit geachtet, um die Codeanalyse und damit die Dekompression möglichst einfach zu machen. Dazu wird an den Code eine Reihe von Anforderungen gestellt `[AB,p.46ff]`. Die Anforderungen an den Code sind der Reihe nach die folgenden:

Blockcode: Jedes Wort des Codes besteht aus einer ganzen Zahl von Bits.

Nichtsingularität: Alle Worte des Codes sind unterschiedlich.

Decodierbarkeit: Alle Worte des Codes sind eindeutig decodierbar. Daß heißt, daß nicht nur alle Worte des Codes unterschiedlich sein müssen, sondern auch alle Folgen von Worten des Codes.

Unverzüglichkeit: Alle Worte lassen sich ohne Vorgriff auf nachfolgende Bits decodieren, also ohne look-ahead.

Eine notwendige und hinreichende Bedingung dafür, daß ein Code ein Blockcode, nichtsingulär, eindeutig decodierbar und unverzüglich ist, ist die, daß kein ganzes Codewort *Präfix* eines anderen Wortes des Codes ist, es gibt also keine zwei Codeworte $a \neq b$ mit $a = prefix(k, b)$. Die entsprechenden Codes heißen *Präfixcodes*. Die Präfixcodes sind eben aufgrund ihrer Eigenschaft, daß kein Codewort Präfix eines anderen Wortes des Codes sein kann, äquivalent zu einem *binären Baum*, an dessen Blättern sich die Codeworte befinden, deren Darstellung an den zu ihnen führenden Kanten verzeichnet ist. Denn wenn der Code kein Präfixcode ist, so gibt es zwei Codeworte $a \neq b$ mit $a = prefix(k, b)$, und um diese beiden Codeworte in einem binären Baum in Blättern darzustellen, müßte es eine Astgabel geben, an denen sich die beiden dargestellten Codeworte verzweigen, wobei die Astgabel noch vor dem k-ten Knoten liegen müßte. Dann würden dort aber die beiden Äste die gleiche binäre Bezeichnung tragen, so daß der Baum kein binärer mehr wäre. Wenn umgekehrt der binäre Codebaum nicht nur Codeworte an allen Blättern, sondern auch an einem inneren Knoten a enthält, so gibt es ein Blatt b an dem an a hängenden Teilbaum, so daß für die zugehörigen Codeworte $a = \mathit{prefix}(k, b)$ gilt, und somit der Code kein Präfixcode ist.

4.6 Quellstruktur

Bei der Konzipierung eines Datenkompressors spielt die Quellstruktur eine entscheidende Rolle. Grob lassen sich drei Strukturen abgrenzen: Zum einen Datenströme *unbestimmter* Länge, zum anderen Dateien von *variabler*, aber *bestimmter* Länge, und schließlich Dateien einer *festen* Länge, also Datenblöcke. Datenströme liegen bei Übertragungskanälen vor, werden aber im allgemeinen nicht als solche komprimiert, sondern meist als Folge von Datenblöcken, um für die Fehlerbehandlung einigermaßen brauchbare Wiedereintrittspunkte zu bieten. Das Verfahren MPEG [MI] komprimiert digitalisierte Videosignale, wobei die einzelnen Bilder nach einem festen Schema gegeneinander differenzcodiert werden. Wichtig ist, daß in regelmäßigen Abständen Immediate-Bilder codiert werden, die von den anderen Bildern unabhängig codiert sind, während die dazwischenliegenden Predicted- und Bidirectional-Bilder in Differenz zu einem oder mehreren der benachbarten Bilder codiert werden. Der Nachteil der Blockung sind geringere Kompressionsraten, so daß ein Kompromiß zwischen Datensicherheit und Kompression getroffen werden muß. Übliche Archivierprogramme (arc, compress, zoo...) komprimieren dagegen Dateien variabler Länge, und diese auch oft am Stück, das heißt ohne sie in Blöcke zu zerlegen.

Wird dagegen die Kompression tiefer in der Struktur des BIOS angesetzt, nämlich in den blockorientierten Schreib-Lese-Zugriffen auf das Medium, ist die Dateistruktur nicht mehr zu erkennen, statt dessen nur noch die Blockstruktur des Massenmediums. Daher komprimieren diese Programme blockweise, wobei die Größe der Blöcke nicht mit der Größe der Cluster übereinstimmen muß, auf die das Filesystem zugreift [RE]. Da für viele Kompressionsverfahren die zu erwartende Datenlänge ein entscheidender Faktor ist, ist die Quellstruktur nicht unwesentlich. Siehe auch den Abschnitt 11 „Einbindung ins Betriebssystem“.

5 Kompressionsverfahren

Es werden diverse Kompressionsverfahren vorgestellt, in ihren Varianten diskutiert und auf Einteilbarkeit nach den oben geschaffenen Kategorien untersucht.

5.1 Huffman

Während bei der Binärcodierung eines Zeichenvorrates normalerweise jedes Zeichen mit der gleichen Anzahl Bits codiert wird, liegt es nahe, zu untersuchen, ob verschiedene Zeichen häufiger auftreten als andere. Die häufigeren Zeichen werden dann durch kürzere und die selteneren durch längere Bitfolgen dargestellt. Ein bekanntes Kompressionsverfahren, das sich die unterschiedliche Häufigkeit einzelner Zeichen in Daten zunutze macht, wurde 1952 von David A. Huffman [HU] vorgestellt. Es ist zugleich ein Verfahren, das bei bekannter Wahrscheinlichkeitsverteilung einen optimalen Präfixcode liefert, also minimale Zieldatenlänge. Der Code ist jedoch nur im Sinne einer Zuordnung von je einem Zeichen zu einem Zeichen optimal, wird dagegen die Forderung nach getrennten Zeichen im Ziel (Blockcode) aufgegeben, läßt sich eine bessere Kompression erreichen. Der Beweis für die Optimalität des Code wird in [BO] gegeben.

Das Verfahren nach Huffman besteht im wesentlichen aus drei Teilen:

– Bestimmen der Wahrscheinlichkeiten p_i der einzelnen Zeichen i.

– Konstruktion der Codes entsprechend den Wahrscheinlichkeiten p_i.

– Codierung der Datei, zeichenweise.

Das Bestimmmen der Wahrscheinlichkeiten erfolgt durch Abzählen der verschiedenen Zeichen in der Quelle, die Codierung besteht darin, daß nacheinander die den Zeichen der Quelldatei zugeordneten Codes ausgegeben werden. Außerdem muß bei variabler Anpassungsstrategie dem Empfänger die Zuordnung der Codes mitgeteilt werden. Der wesentliche Teil des Verfahrens nach Huffman ist die Konstruktion der Codes. Hierzu werden alle Zeichen i mit ihren Wahrscheinlichkeiten p_i in eine Liste eingetragen. Es werden die zwei Einträge i_0, i_1 mit minimalen Wahrscheinlichkeiten p_{i_0}, p_{i_1} ausgewählt und zu einem gemeinsamen Eintrag i' zusammengefaßt („RE“ und „MI“ in Bild 7). Diesem Eintrag werde die Summe der betreffenden Wahrscheinlichkeiten zugewiesen: $p_{i'} := p_{i_0} + p_{i_1}$. Die beiden verwendeten Einträge i_0 und i_1 werden aus der Liste gestrichen, so daß die Gesamtzahl der Einträge nun um eins geringer ist als zuvor. Dieses Zusammenfassen der beiden Einträge mit jeweils minimaler Wahrscheinlichkeit

i	$48p_i$							p_i	L_i	$p_i \cdot L_i$	Code	Code_*
DO	5	5	8	15	15	29	48	$0.1041\overline{6}$	4	$0.41\overline{6}$	0000	1110
RE	1	3						$0.0208\overline{3}$	5	$0.1041\overline{6}$	00010	11110
MI	2							$0.041\overline{6}$	5	$0.208\overline{3}$	00011	11111
FA	14	14	14	14	14			$0.291\overline{6}$	2	$0.58\overline{3}$	01	00
SOL	10	10	10	10	19	19		$0.208\overline{3}$	2	$0.41\overline{6}$	10	01
LA	9	9	9	9				0.1875	2	0.375	11	10
TI	7	7	7					$0.1458\overline{3}$	3	0.4375	001	110
										$2.541\overline{6}$		

Bild 7

wird fortgesetzt, bis nur noch ein Eintrag übrig ist, der alle ursprünglichen Zeichen umfaßt. Die Schritt für Schritt entstandene Struktur ist ein binärer Baum, an dessen Blättern sich die Zeichen befinden. Um den Zeichen nun Codes zuweisen zu können, werden im Baum jeweils eine Kante mit „0“, die andere mit „1“ markiert. Welche der beiden Kanten jeweils wie markiert wird, ist dabei frei, aber fest zu wählen. Der Code für ein Zeichen läßt sich ablesen, indem man den Kanten von der Wurzel bis zum entsprechenden Blatt folgt (top-down, wohingegen die Konstruktion des Baumes von den Blättern zur Wurzel hin erfolgte, also bottom-up).

Einen Algorithmus zur Bestimmung einer *eindeutigen* Zuordnung von Codes in Abhängigkeit von den Wahrscheinlichkeiten der Zeichen liefern Schwartz und Kallick [SK]. Durch geschickte Wahl der Markierungen der Baumkanten kann bei der Übermittlung der Codetabelle gespart werden. Es ist nicht notwendig, die Zuordnung der Codes oder die genauen Wahrscheinlichkeiten p_i zu übermitteln. Es genügt völlig, die Längen L_i der zugewiesenen Codes mitzuteilen, da aus diesen eine eindeutige Markierung des Baumes gewonnen werden kann. Dazu wird zunächst der Baum generiert wie beschrieben. Statt nun die Kanten zu markieren, werden die ursprünglichen Zeichen mit neuen, veränderten Wahrscheinlichkeiten $p_i' := 2^{-L_i}$ versehen. Jetzt wird der Baum erneut generiert, wobei darauf geachtet werden muß, daß für den Fall, daß es mehr als zwei minimale Wahrscheinlichkeiten gibt, zwei auf eindeutige Weise ausgewählt werden müssen. Zu beachten ist, daß diesmal ein anderer Baum entstehen kann als beim ersten Mal, daß aber die Tiefen der jeweiligen Blätter und damit die Codelängen L_i die gleichen sind wie zuvor.

Noch einfacher ist es, gar nicht erst einen neuen Baum zu generieren, sondern die Codes der Reihe nach zuzuteilen: Zunächst den Zeichen mit der kürzesten Codelänge, dann denen mit der jeweils nächstlängeren, und innerhalb dieser Gruppen nach der ursprünglichen Ordnung der Zeichen selbst (Der Code_* in

Bild 7 wurde so erzeugt). Der Unterschied ist der, daß der neue Baum allein aus den geforderten Codelängen L_i konstruiert wurde. Werden nun noch die Kanten eindeutig markiert, genügt für eine Rekonstruktion der komprimierten Daten das Vorliegen der den ursprünglichen Zeichen zugeordneten Codelängen L_i. Auch diese Tabelle von Codelängen muß natürlich nicht direkt abgespeichert werden, sondern kann komprimiert werden. Es handelt sich hierbei um eine relativ kurze Datei, so daß sich der Einsatz eines allgemeingültigen Verfahrens nicht lohnt[6].

So wie das Huffmanverfahren beschrieben wurde, läßt es sich kategorisieren als verlustfreies Verfahren, ASYMMETRISCH, basierend auf STATISTISCHER REDUNDANZ, mit VARIABLER ANPASSUNG in einem Analysepaß, Codierung FIXED-TO-VARIABLE.

Wegen der variablen Anpassungsstrategie weist es zwei Nachteile auf: Erstens läßt es sich nicht auf Datenströme anwenden und zweitens ist immer die Beigabe der Codelängentabelle erforderlich. Beide Nachteile lassen sich beheben, indem eine ADAPTIVE ANPASSUNG gewählt wird. Dabei wird zunächst ein leerer oder gleichverteilter Codebaum angenommen und auf die Analyse verzichtet. Das erste Zeichen wird hiernach codiert. Nun wird die durch das Auftreten des ersten Zeichens veränderte Wahrscheinlichkeit vermerkt und der Codebaum entsprechend angepaßt. Das wird für jedes Zeichen aus der Eingabe wiederholt. Man könnte natürlich nach jedem gelesenen Zeichen den Codebaum komplett neu konstruieren, es gibt jedoch Strategien geringeren Aufwandes, den alten Codebaum an die neuen Werte anzupassen [CH,KN,VI]. Bei adaptiver Redundanzanpassung ist das Huffmanverfahren SYMMETRISCH.

5.2 Shannon-Fano

Während beim variablen Huffmanverfahren der Codebaum *bottum-up* konstruiert wird, geht das ältere Verfahren nach Shannon und Fano *top-down* vor. Es soll allerdings nur nebenbei erwähnt werden, da es schlechtere Ergebnisse liefert als Huffman. Der Beweis von Balfanz und Bergholz [BB] hierzu zeigt nur, daß es für jedes Alphabet mit mindestens fünf Zeichen einen Verteilungsbereich gibt, für den Huffman besser ist als Shannon-Fano. Der Beweis, daß Shannon-Fano

[6] Da die Codelängen benachbarter Zeichen eher nicht in Beziehung zueinander stehen, jedoch meist eine Häufung um eine mittlere Codelänge auftritt, könnte man in der Tabelle die Zahlen $i = 0..n$ darstellen durch $n - i + 2$ Bit (.. 0001, 001, 01), die Zahlen $i = n + 1...$ durch $i - n + 1$ Bit (10, 110, 1110 ..). Für das Beispiel in Bild 7 ergibt sich mit $n = 3$: „10 110 110 001 001 001 01". Der optimale Wert für n kann experimentell bestimmt werden und er muß noch nicht einmal mit der Tabelle abgelegt werden, da er bei der Dekompression überhaupt nicht benötigt wird. Für die Konstruktion der oben beschriebenen Codetabelle werden nur die relativen Codelängen benötigt, nicht jedoch absolute.

nie besser sein kann als Huffman, und, allgemeiner, daß Huffman immer den optimalen Code liefert, findet sich in [BO].

Wie bei Huffman werden die Wahrscheinlichkeiten der Eingabezeichen ermittelt. Das Eingabealphabet wird nun zusammen mit den Wahrscheinlichkeiten in eine Liste eingetragen und nach letzteren fallend sortiert. Nun wird die Liste in zwei zusammenhängende Teile geteilt. Die Kriterien, nach denen dies geschieht, werden in verschiedenen Quellen unterschiedlich angegeben. Nach Thuner und Otter [TO] werden zwei Teillisten mit „möglichst gleich großer Gesamtwahrscheinlichkeit" gebildet, nach Völz [VÖ91,p.113] wird die Codeliste „so geteilt, daß die Summe der Wahrscheinlichkeiten oben größer oder gleich unten ist", wobei oben die größeren Einzelwahrscheinlichkeiten stehen. Letzteres Vorgehen führt aber zu etwas schlechteren Ergebnissen. Wie auch immer, nun wird durch wiederholtes Teilen der Teillisten — bis allein einelementige Teillisten übrig sind — wie beim Huffmanverfahren ein binärer Baum konstruiert (allerdings top-down, nicht bottom-up). Alles weitere erfolgt ebenfalls wie bei Huffman.

5.3 Arithmetische Codierung

Bereits Huffman selbst [HU] bezeichnet sein Verfahren als optimal bezüglich der Codierung von Nachrichten, mit einem Code von minimaler Redundanz. Unter der Voraussetzung, daß bei der Codierung ein Blockcode zur Anwendung gelangen muß, ist diese Behauptung richtig, sind jedoch auch Nichtblockcodes zugelassen, so lassen sich bessere Ergebnisse erzielen. Als Beispiel dient ein simpler Fall, ein Quellalphabet mit nur zwei Zeichen, $G = \{0, 1\}$, mit den Wahrscheinlichkeiten p und $1 - p$. Ganz offensichtlich führen die Verfahren nach Huffman sowie Shannon und Fano zu keiner Kompression. Zur Motivation der folgenden arithmetischen Kompression werden zwei verschiedene Ansätze gegeben.

Der erste Ansatz findet sich bei Shannon und Weaver [SW,p.58ff] und besteht darin, „die Menge aller Folgen von N Zeichen, die von der Quelle erzeugt werden, zu berücksichtigen". Er findet sich auch bei Bauer und Goos [BG,p.299]: Man codiert „nicht jedes der n Zeichen einzeln, sondern betrachtet stattdessen Binärcodierungen für die n^k Gruppen von je k Zeichen". Für $k = 2$ und das obige Beispiel würden sich also vier Gruppen ergeben, mit $p = 0.7$ als Beispiel ergibt sich bereits ein Mittel von $m_2 = 0.905$ bit/Zeichen (Bild 8), während $m_1 = 1$ bit/Zeichen. Wird nun k schrittweise vergrößert, so nähert sich das Mittel von m_k bit/Zeichen dem theoretischen Grenzwert asymptotisch[7]. Hat k

k=2	p_i	L_i	Code	$p_i \cdot L_i$
00	0.49	1	0	0.49
01	0.21	2	10	0.42
10	0.21	3	110	0.63
11	0.09	3	111	0.27
				1.81 /2= 0.905

Bild 8

k=3	p_i	L_i	Code	$p_i \cdot L_i$
000	0.343	2	00	0.686
001	0.147	2	01	0.294
010	0.147	3	100	0.441
011	0.063	4	1100	0.252
100	0.147	3	101	0.441
101	0.063	4	1101	0.252
110	0.063	4	1110	0.252
111	0.027	4	1111	0.108
				2.726 /3= $0.908\overline{6}$

Bild 9

die Länge der Datei erreicht, dann wird die ganze Datei als eine Gruppe von Zeichen aufgefaßt. Dies ist der Grenzfall, den die arithmetische Kompression darstellt. Da der benötigte Codebaum in der Praxis meist sehr groß sein würde, läßt sich die arithmetische Kompression so nicht realisieren. Daher wird noch ein zweiter Ansatz vorgestellt.

Dieser findet sich bei Abramson `[AB,p.61,Fußnote 1]`: Das rechts offene Intervall $[0, 1)$ wird gemäß den Wahrscheinlichkeiten der Zeichen des Quellalphabets in verschieden große Teilintervalle zerlegt. Das dem ersten zu codierenden Zeichen entsprechende Teilintervall wird ausgewählt und wiederum in Teilintervalle zerlegt. Auf diese Weise wird schließlich die zu codierende Datei durch ein sehr kleines Intervall repräsentiert[8]. Dabei genügt es für die Decodierung, eine beliebige rationale nicht periodische Zahl aus diesem Intervall und die Länge der Datei zu kennen. Zur Decodierung wird genauso vorgegangen wie bei der Codierung, nur wird das jeweilige Teilintervall nicht anhand der Quelle bestimmt,

[7] Jedoch nicht monoton: $k = 3 \rightarrow$ Mittel $m_3 = 0.908\overline{6}$ bit/Zeichen (Bild 9), $k = 4 \rightarrow$ Mittel $m_4 = 0.8918$. Monotonie herrscht nur bei Vervielfachung der Gruppenlängen, also $m_k \geq m_{k \cdot i}$ für $i \in N$, da für $k \cdot i$, wenn nicht strenge Monotonie herrscht, der Code mindestens als Verkettung der den verketteten Zeichen des Quellalphabets entsprechenden Codes gebildet werden kann, so daß $m_k = m_{k \cdot i}$.

[8] Allgemeiner formuliert ist die Idee der Zuordnung von sich nicht überlappenden Intervallen zu ganzen Nachrichten durch Fitingof `[FI]`: Ein Wert der Funktion s, *estimate of usefulness*, wird jeder Nachricht (Datei) zugeordnet und bestimmt die Größe des zugehörigen Intervalls.

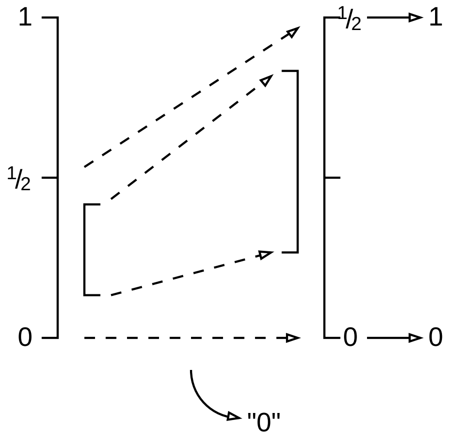

Bild 10

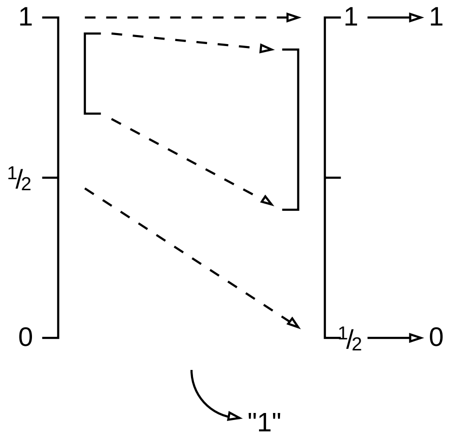

Bild 11

sondern danach, in welchem Teilintervall sich die repräsentierende Zahl befindet. Durch diesen zweiten Ansatz ist im Prinzip auch schon der Algorithmus beschrieben, allerdings ist er so praktisch nicht anwendbar: Zur Darstellung des Teilintervalls wäre eine extrem hohe Genauigkeit notwendig, nämlich die durch die Länge der gesamten Zieldatei selbst gegebene.

Der praktisch anwendbare Algorithmus wurde von Witten, Neal und Cleary `[WNC]` vorgestellt: Um nicht die gesamten Intervallgrenzen bis zum Ende der Codierung mitführen zu müssen, wird während der Codierung laufend kontrolliert, ob nicht die ersten Nachkommastellen der gesuchten Zahl im Intervall bereits feststehen. Dazu werden zunächst zwei einfachere Fälle betrachtet. Ist das Intervall ein Teilintervall von $[0, \frac{1}{2})$, dann steht fest, daß die erste binäre Nachkommaziffer der repräsentierenden Zahl eine 0 sein wird. In diesem Fall können die beiden Grenzen des Intervalls verdoppelt werden und die 0 kann schon ausgegeben werden, das zu suchende Intervall wird sozusagen mit der Lupe weiterverfolgt (Bild 10). Ähnlich kann, wenn das Intervall in $[\frac{1}{2}, 1)$ liegt, als erste Nachkommaziffer eine 1 ausgegeben werden. Wieder werden die Grenzen des Intervalls verdoppelt, danach aber noch um 1 vermindert, damit es sich wieder in $[0, 1)$ befindet (Bild 11).

Das war leider noch nicht die ganze Lösung, denn es kann passieren, daß das Intervall zwar sehr klein wird, aber die Zahl $\frac{1}{2}$ hartnäckig beinhaltet. Um auch in diesem Fall vergrößern zu können, muß das Intervall $[\frac{1}{4}, \frac{3}{4})$ näher betrachtet werden: Entweder wird die Obergrenze der Intervallschachtelung irgendwann doch kleinergleich $\frac{1}{2}$, oder die Untergrenze wird größer, oder das gesuchte schlußendliche Intervall enthält tatsächlich auch $\frac{1}{2}$. Im ersten Fall sind die nächsten zwei Ziffern 01, wobei die 0 daher rührt, daß die Obergrenze schließlich kleinergleich als $\frac{1}{2}$ wurde, die 1 daher, daß die Untergrenze größergleich als $\frac{1}{4}$ war und nun, nach Ausgabe der 0 und Vergrößerung größer als $\frac{1}{2}$ ist. Im zweiten Fall ist es genau umgekehrt, es wird 10 ausgegeben. Statt nun abzuwarten, bis die erste 0 ausgegeben werden kann, kann auch das Intervall, das in $[\frac{1}{4}, \frac{3}{4})$ liegt, verdoppelt und um $\frac{1}{2}$ vermindert werden, dann liegt es wieder in $[0, 1)$ (Bild 12). Allerdings

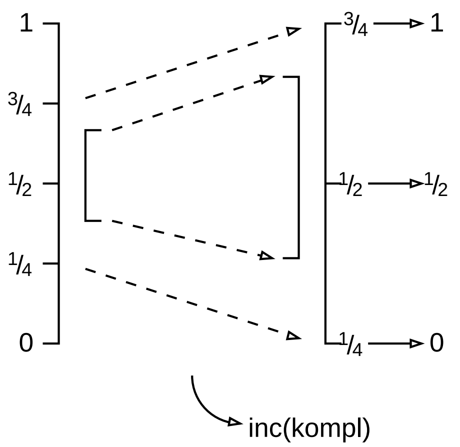

Bild 12

muß vermerkt werden, daß nach Eintreten eines der ersten beiden Fälle nach dem entsprechenden Bit ein komplementäres ausgegeben werden muß. Führt man diese Überlegungen weiter, findet man, daß die Erweiterung des Intervalls $[\frac{1}{4}, \frac{3}{4})$ beliebig wiederholt werden kann, bevor die Grenze $\frac{1}{2}$ überschritten wird. Es muß dann eben nur mitgezählt werden, wieviele komplementäre Bits ausgegeben werden müssen, sobald ein anderer als der mittlere Fall eingetreten sein wird. Für die anderen beiden Intervalle gelten ähnliche Überlegungen. Die Verdopplungen des Intervalls werden nach jedem Teilungsschritt jeweils sooft vorgenommen, bis keiner der drei Fälle mehr anwendbar ist (Bild 13).

Soll die Codierung der Daten abgeschlossen werden, wird je nachdem, ob die Untergrenze des verbleibenden Intervalls kleinergleich $\frac{1}{4}$ ist, eine 0 oder eine 1 ausgegeben, der Zähler für die ausstehenden komplementären Bits um eins erhöht und entsprechend viele komplementäre Bits ausgegeben.

Bei der Intervallberechnung wird es trotz der zwischenzeitlichen Vergrößerungen zu Rechenungenauigkeiten kommen, die das mathematisch exakte Ergebnis, nämlich eine Zahl im gesuchte Intervall, verfälschen können. Voraussetzung

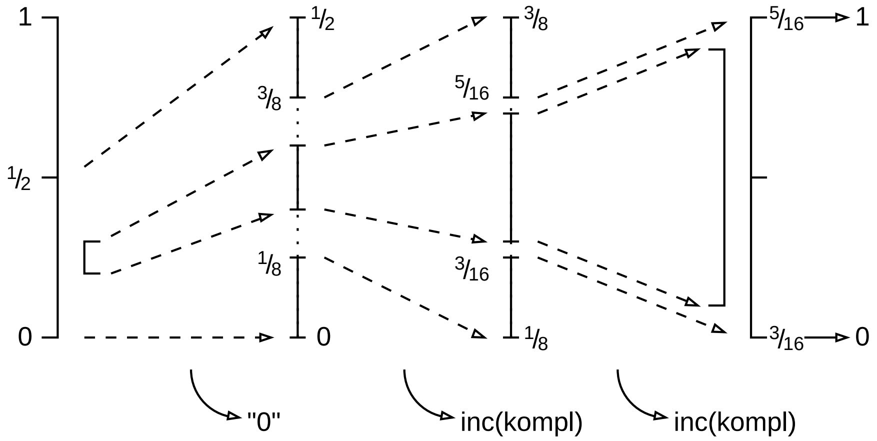

Bild 13

dafür, daß der Quelltext wiederherstellbar ist, ist zum einen, daß nie ein Teilintervall der Länge Null entsteht, und zum anderen, daß Coder und Decoder die gleichen Rechenfehler machen, also immer in exakt der gleichen Weise runden. Daher empfiehlt sich die Verwendung von Festpunktzahlen, da hier die auftretenden *Rundungsfehler* wesentlich leichter in den Griff zu bekommen sind als bei Fließpunkteinheiten. Als Beispiel dafür, daß tatsächlich genau die gleichen Rechenfehler gemacht werden müssen, wird in Bild 14 die Codierung der

in	-x	Oben	Mitte	Unten	kompl	out
		1.0000	0.1010	0.0000		
0		0.1010	0.0110*	0.0000		
1		0.1010		0.0110		
	½	0.1100		0.0100	1	
	½	1.0000	0.1010	0.0000	2	
0		0.1010	0.0110	0.0000	2	
0		0.0110		0.0000	2	
	0	0.1100	0.1000	0.0000		011
0		0.1000		0.0000		
	0	1.0000	0.1010	0.0000		0
0		0.1010	0.0110	0.0000		
0		0.0110		0.0000		
	0	0.1100	0.1000	0.0000		0

0.101·0.101 = 0.011001
*4stellig: 0.0110 abgerundet
5stellig: 0.01101 aufgerundet

Bild 14

binären Folge 010000... mit $p_0 = \frac{5}{8}$, $p_1 = \frac{3}{8}$ durch eine Festkommaarithmetik mit vier Nachkommastellen vorgeführt. In Bild 15 werden die erzeugten Code-

in	-x	Oben	Mitte	Unten	out
0.01100		1.00000	0.10100	0.00000	
0.01100		0.10100	0.01101*	0.00000	0
0.01100		0.01101		0.00000	0
0.11000	0	0.11010	0.10000	0.00000	
0.11000		0.11010		0.10000	1
0.10000	1	0.10100	0.01101	0.00000	
0.10000		0.10100		0.01101	1
0.10000	½	0.11000	0.10011	0.01010	
0.10000		0.10011		0.01010	0
0.10000	½	0.10110	0.01111	0.00100	
0.10000		0.10110		0.01111	1
0.10000	½	0.11100	0.10111	0.01110	
0.10000		0.10111		0.01110	0
0.10000	½	0.11110	0.10111	0.01100	
0.10000		0.10111		0.01100	0
0.10000	½	0.11110	0.10110	0.01000	
0.10000		0.10110		0.01000	0
0.10000	½	0.11100	0.10010	0.00000	
0.10000		0.10010	0.01011	0.00000	0
0.10000		0.10010		0.01011	1
0.10000	½	0.10100	0.01111	0.00110	
0.10000		0.10100		0.01111	1
0.10000	½	0.11000	0.10100	0.01110	
0.10000		0.10100		0.01110	0
0.10000	½	0.11000	0.10100	0.01100	
0.10000		0.10100		0.01100	0
0.10000	½	0.11000	0.10010	0.01000	
0.10000		0.10010		0.01000	0
0.10000	½	0.10100	0.01101	0.00000	

Bild 15

daten, 011000..., mit den gleichen Wahrscheinlichkeiten $p_0 = \frac{5}{8}$, $p_1 = \frac{3}{8}$ decodiert, mit dem einzigen Unterschied, daß die verwendete Festkommaarithmetik mit fünf Nachkommastellen, also genauer, rechnet. Da durch die unterschiedlichen Rechengenauigkeiten die für die Codierung wichtigen Teilintervalle minimal voneinander abweichen (Bild 16), ist der decodierte Text ein anderer als der ursprüngliche. Die entscheidenden Stellen im Codier- und im Decodiervorgang sind mit Sternchen (*) markiert.

Die arithmetische Codierung fällt ebenfalls in die Kategorie der verlustfreien Verfahren, ist ASYMMETRISCH, basiert auf STATISTISCHER REDUNDANZ, mit VARIABLER ANPASSUNG in einem Analysepaß, die Codierung ist allerdings ein NICHTBLOCKCODE. Natürlich kann auch die arithmetische Codierung mit ADAPTIVER REDUNDANZANPASSUNG betrieben werden und ist dann SYMMETRISCH. Der besondere Vorteil gegenüber einem adaptiven Huffmanverfahren ist der, daß das komplizierte Aktualisieren des binären Baumes entfällt, es muß nur eine Liste von Wahrscheinlichkeiten jeweils aktualisiert werden, was bei kon-

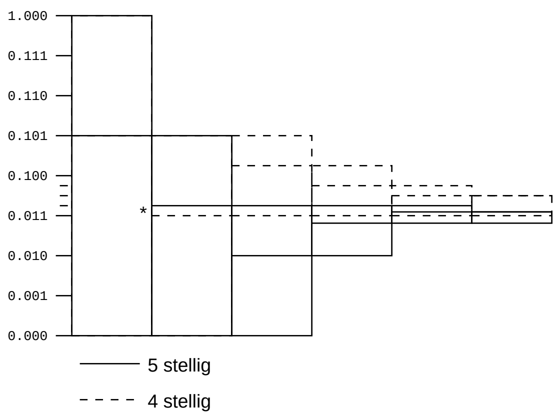

Bild 16

stantem Aufwand möglich ist.

5.4 Markovketten

Bei der Bestimmung der Wahrscheinlichkeiten im Rahmen statistischer Verfahren wurde bisher davon ausgegangen, daß die Zeichen unabhängig voneinander auftreten, also unabhängig vom Kontext. Tatsächlich besteht aber häufig eine starke *Korrelation* zwischen aufeinanderfolgenden Zeichen, beliebtes Beispiel ist die Buchstabenfolge `Qu`: auf `Q` folgt meist `u`, obwohl die unkorrelierte Wahrscheinlichkeit des `u` gering ist [LL]. Je nachdem, in Abhängigkeit wievieler der vorangegangenen Zeichen die Wahrscheinlichkeit des nächsten bestimmt wird, spricht man von einer *Markovkette* m-ter Ordnung.

Bei Bildern ist die Korrelation zwischen benachbarten Bildpunkten relativ hoch. Geht man davon aus, daß Pixelbilder zeilenweise abgelegt sind, und ist auch eine Abhängigkeit in vertikaler Richtung vorgesehen, so führt dies zu einer formell sehr hohen Ordnung der betrachteten Markovkette. Anschaulich wird die Ordnung jedoch wesentlich geringer sein, nur eben zweidimensional. Für duale Pixelbilder liefert eine zeilenweise arithmetische Codierung mit adaptiver Anpassungsstrategie über Markovketten bereits durchaus akzeptable Ergebnisse, siehe den Abschnitt 5.11 „Schwarzweißbilder“.

Bei der praktischen Realisierung einer statistischen Kompression mit Markovketten m-ter Ordnung ergeben sich Probleme der Darstellbarkeit der benötigten Tabellen. Immerhin wird bei 2. Ordnung und einem Alphabet von n Zeichen (Bytes: $n = 256$) jeweils die Wahrscheinlichkeit eines Zeichen in Abhängigkeit der zwei vorangegangenen ausgewertet. Das führt zu einer Tabellengröße von zunächst n^3 Einträgen (Bytes: $256^3 = 4$M). Daher sind vor allem Strategien wichtig, die zur Begrenzung des temporären Speicherplatzbedarfs geeignet sind. Die Tabelle kann in zwei Gruppen geteilt werden: Zum einen die häufigeren Markovketten, die weiter in der Tabelle bleiben, zum anderen die selteneren Markovketten, denen einfach eine konstante niedrige Standardhäufigkeit zugewiesen wird, die nicht in der Tabelle vermerkt wird. Dies ist praktikabel, da bei hohen Ordnungen weite Teile der Tabelle ohnehin leer bleiben werden. Der verbleibende Teil der Tabelle, die häufigeren Markovketten also, werden dann in anderer Form, in einer Hashtabelle oder einem Baum abgelegt. Als Entscheidung, welche Kombinationen als häufig anzusehen sind, können Mindestwahrscheinlichkeiten oder eine feste maximale Anzahl von gewünschten Markovketten dienen. Bei variabler Redundanzanpassung folgt natürlich sofort das Problem, daß die entsprechenden Tabellen auch im Code mit abgelegt werden müssen. Noch dringender als beim temporären Speicherplatzbedarf muß hier auf optimierte Darstellung der benötigten Tabellen geachtet werden.

Neben Markovketten können, um der Abhängigkeit der benachbarten Zeichen Rechnung zu tragen, bei natürlichem Text auch Silben und Worte des Textes als Einheiten betrachtet und deren Häufigkeiten untersucht werden [VI]. Auch hier ergeben sich wieder die Probleme der großen Tabellen.

5.5 Codebook-Verfahren

Neben der Gruppe der auf statistischer Redundanz basierenden Verfahren gibt es die zweite Gruppe der auf referenzierender Redundanz basierender Verfahren. Einige dieser Verfahren sind Abwandlungen der Arbeiten von Abraham Lempel und Jacob Ziv [ZL77,ZL78]. Das neue in [ZL77] war ein *adaptives* Codebook im Gegensatz zu einem *statischen*, wie es zuvor schon in Kompressionsverfahren vorgeschlagen worden war, zum Beispiel von Held [HE,p.48f] als „pattern substitution".

Alle Verfahren dieser Gruppe verfügen über ein *Codebook*[9] — das ist eine Tabelle von Zeichenfolgen — und ersetzen im Text auftretende Zeichenfolgen, die im Codebook eingetragen sind, in der Ausgabe durch Verweise in das Codebook. Je nachdem, auf welche Art das Codebook erstellt und aktualisiert wird, liegt eine unterschiedliche Redundanzanpassungsstrategie vor: Prinzipiell kann ein statisches, ein variables und ein adaptives Codebook unterschieden werden. Statisch hieße, daß sowohl Coder als auch Decoder eine feste Tabelle

[9] Das Codebook wird oft auch als *dictionary* bezeichnet.

haben, zum Beispiel Listen mit den n häufigsten Wörtern und ihren gebeugten Formen, die nicht verändert werden (*static dictionary method* [ST,p.60]). Im Falle einer variablen Strategie wird der Quelltext zuerst auf die häufigsten Zeichenfolgen mit zum Beispiel mindestens k Zeichen untersucht, im zweiten Durchgang können die entsprechenden Ersetzungen vorgenommen werden. Ein Codebook-Verfahren mit variabler Anpassungsstrategie stellt Lanzerath [LA] als *Clusterkompression* vor. Dabei werden zwei Codebooks streng abwechselnd verwendet, eins für alphanumerische Worte, eins für die Sonderzeichenfolgen, die diese Worte trennen. Die Folge numerischer Werte, die als Code erzeugt wird, stellt also Zeiger in die beiden Codebooks dar, unmittelbarer Text tritt im Code nicht mehr auf.

Da die statische Anpassung unflexibel ist und die variable die Beigabe großer Tabellen von Zeichenfolgen erfordert, wird meist die adaptive Strategie verwendet: Das Codebook ist zu Beginn üblicherweise leer, in manchen Fällen enthält es auch das Quellalphabet oder Nullen. Während der Quelltext abgearbeitet wird, werden nun neue Zeichenfolgen in das Codebook eingetragen und, je nach Strategie, andere wieder aus dem Codebook gelöscht.

5.6 Lempel-Ziv Storer-Szymanski

Das 1977 von Ziv und Lempel [ZL77] vorgestellte Verfahren wird im allgemeinen als *LZSS* (*Lempel Ziv Storer Szymanski*) [SS] bezeichnet. Die dem Verfahren zugrunde liegende Idee ist, Zeichenfolgen, die wiederholt im Quelltext auftreten, jeweils durch einen Verweis auf das vorangehende Exemplar der gleichen Zeichenfolge zu ersetzen und den Rest — alle sich nicht wiederholenden Teile des Textes — beizubehalten. Die Textanalyse besteht darin, den vorangegangenen Text — oder Teile desselben — nach Übereinstimmungen mit den anstehenden Zeichen abzusuchen. Die Codesynthese besteht darin, in den Resttext die entsprechenden Rückwärtsverweise einzufügen. Diese Verweise sind einfach Paare $\langle p, l\rangle$, bestehend aus einem Index p (positiv oder als relativer Rückwärtszeiger) und einer Länge l. Der Satz „Fischers_Fritz_fischt_frische_Fische." kann zum Beispiel codiert werden als „Fischers_Fritz_f$\langle 1, 4\rangle$t$\langle 14, 2\rangle$r$\langle 1, 5\rangle$_$\langle 0, 6\rangle$." (positive Zeiger bezüglich Dateianfang) oder „Fischers_Fritz_f$\langle -15, 4\rangle$t$\langle -7, 2\rangle$r$\langle -23, 5\rangle$_$\langle -30, 6\rangle$." (relative Rückwärtszeiger).

Wenn jeweils der komplette bisher gelesene Text nach Zeichenfolgen abgesucht wird, führt dies dazu, daß die Analyse quadratischen Aufwand $O(n^2)$ bezüglich der Dateilänge n hat. Um dies zu vermeiden, beschränkt das Verfahren *LZSS* die Suche nach Übereinstimmungen auf die jeweils letzten k Zeichen des bisherigen Textes. Diese letzten k Zeichen werden während der Analyse als das

Codebook aufgefaßt. Das Codebook hat also eine feste Länge k. Nach jedem bearbeiteten Zeichen verschiebt sich dieses Codebook ebenfalls um ein Zeichen über die Quelldaten, weswegen von der *sliding dictionary method* [ST,p.64] gesprochen wird. Alle Zeichenfolgen, deren Beginn mehr als k Zeichen zurückliegt, können als referenzierbare Muster nicht mehr dienen. Dafür lassen sich die Verweise ins Codebook aber in konstanter Länge codieren.

Beim Komprimieren wird einfach die längste mit der an der Eingabe anstehenden identische Zeichenfolge gesucht und codiert, beim Dekomprimieren geht es noch einfacher, es muß nur vom Index aus in die Ausgabe kopiert werden. Dabei ist zu beachten, daß sich das Muster und die Kopie überlappen können. Insbesondere im Falle einer *Lauflänge*, also eines sich mehrfach wiederholdenden Zeichens, zeigt der Index auf das unmittelbar vorangehende Zeichen, vom Muster ist also zunächst nur ein Zeichen verfügbar. Sobald dieses aber in die Ausgabe kopiert wurde ist es dort als zweites Zeichen vorhanden. Bei der Dekompression muß also der Text jeweils zeichenweise wiederhergestellt werden.

Dieses Verfahren gehört in die Kategorie der verlustfreien Verfahren, basierend auf REFERENZIERENDER REDUNDANZ, mit ADAPTIVER ANPASSUNG. Die Codierung erfolgt in zwei Ausgabeströmen, die ineinandergemischt werden müssen, siehe den Abschnitt 7 „Codierung der Ausgabeströme". Die einzelnen Zeichen der Ausgabeströme sind von fester Länge, die Codierung ist daher VARIABLE-TO-FIXED.

Ob das Verfahren als SYMMETRISCH oder ASYMMETRISCH einzuordnen ist, hängt davon ab, wie die Suche des optimalen Musters implementiert wird. Ohne die Hilfe weiterer Datenstrukturen ist die Suche des optimalen Musters im Codebook recht aufwendig: Bei einem optimalen Muster der Länge m ergibt sich für das aktuelle Zeichen ein Aufwand von bis zu $O(k \cdot m)$. Dennoch ist bei einer Dateilänge von n Zeichen der Gesamtaufwand höchstens $O(k \cdot n)$, da ja im Falle eines optimalen Musters der Länge m bereits m Zeichen abgearbeitet werden, so daß der mittlere Aufwand je Zeichen bei $O(k)$ liegt.

Bauernöppel [BN] schlägt vor, parallel zum Codebook, das ja in diesem Fall einen Datenblock der Länge k darstellt, ein Zeigerfeld gleicher Länge zu führen, in dem einfach verkettete Listen jeweils die Positionen miteinander verbinden, an denen im Codebook gleiche Zeichen stehen. Als Wurzeln für diese Listen wird über das Quellalphabet ein Feld von Zeigern deklariert, die auf die jeweils hintersten Elemente der rückwärts verlaufenden Listen zeigen (Bild 17). Dadurch wird die Analyse zwar beschleunigt, aber in dem banalen, jedoch ungünstigen Fall von Lauflängen ist der Aufwand je Zeichen wieder $O(k)$, denn es muß für jede Position im Codebook wieder die ganze Länge des Musters abgesucht werden.

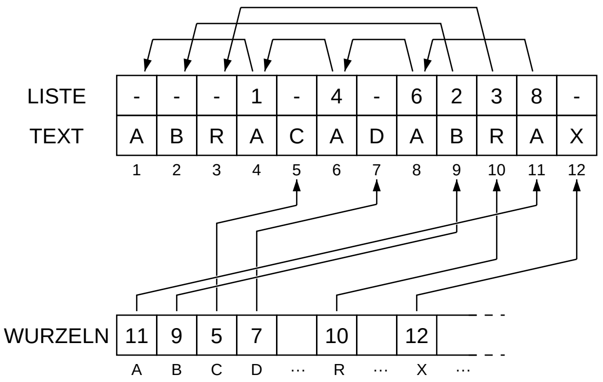

Bild 17

Ein ergebnisorientierter Ansatz ist eine Baumstruktur, die zu einem Eingabealphabet Worte beschreibt. Dabei werden die Kanten von einem Elternknoten zu seinen Kinderknoten mit unterschiedlichen Zeichen des Eingabealphabets bezeichnet [RPE]. Diese Struktur wird *Suffixbaum* genannt. Bild 18 zeigt einen

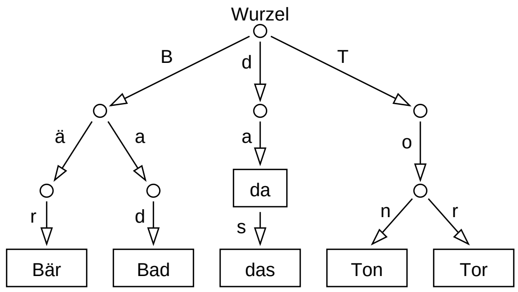

Bild 18

Suffixbaum, der die Worte „Bär“, „Bad“, „da“, „das“, „Ton“ und „Tor“ enthält. Um ein Wort im Baum zu finden, geht man, beginnend bei der Wurzel, jeweils ein Zeichen des Wortes nach dem anderen nehmend, die entsprechenden Kanten hinab. Erreicht man so ein Blatt oder einen markierten Knoten, so ist das Wort Element der dargestellten Menge. Am gefundenen Knoten können weitere Merkmale zum aktuellen Wort vorhanden sein, für den Algorithmus *LZSS* werden Verweise in das Codebook mitvermerkt. Vom nächsten zu verarbeitenden Zeichen an schreitet der Suchalgorithmus also, beginnend bei der Wurzel, die Kanten hinab, bis ein Blatt erreicht oder keine dem folgenden Zeichen entsprechende Kante vorhanden ist.

Das Problem, das sich bei Verwendung dieser Baumstruktur in Verbindung mit Lempel-Ziv-Algorithmen ergibt, besteht darin, daß wegen des sich verändernden Codebooks auch die Baumstruktur ständig aktualisiert werden muß. Beim Algorithmus $LZSS$ müssen für jedes gelesene Zeichen soviele neue Einträge in den Suffixbaum geschrieben werden, wie das Codebook lang ist. Ebensoviele Einträge müssen gelöscht werden. Der Aufwand zum Aktualisieren des Baumes ist mit Sicherheit höher als $O(k)$ je Zeichen. Es wird sich jedoch zeigen, daß für den Algorithmus LZW der Aufwand wesentlich geringer ist.

Da auch der ergebnisorientierte Ansatz nicht zum Ziel führt, bleibt noch die Möglichkeit, den Ansatz der über gleiche Zeichen rückwärts verketteten Zeigerlisten durch Behandeln der häufigeren Problemfälle zu verbessern. Das weitaus größte Problem stellen Lauflängen dar. Es wird also eine Fallunterscheidung vorgenommen, die bei Auftreten von Lauflängen verhindert, daß diese je Zeichen von vorne bis hinten durchsucht werden. Die Abfrage prüft in der Eingabe zunächst, wie lang die Lauflänge sein wird. Dann sucht sie im Codebook entweder eine Lauflänge gleicher Länge und untersucht deren Nachfolgerzeichen, oder codiert — wenn keine solche vorhanden ist — die neue Lauflänge in sich durch den oben beschriebenen Index auf das erste Zeichen der Lauflänge.

Ein weiteres Problem — das jedoch in den meisten Implementationen ignoriert wird, da es geringfügig ist — besteht darin, daß die bei linearer Suche gefundenen Muster nicht zu einer optimalen Codierung führen. Als Beweis genügt ein Gegenbeispiel: Es soll die Zeichenfolge „bananisbanis“ codiert werden. Für die Ausgabe gelte, daß ein Verweis ins Codebook doppelt so lang ist wie ein Zeichen. Die Eingabe besteht aus zwölf Zeichen. Die lineare Suche geht wie folgt: Die ersten sieben Zeichen werden übernommen, da nur eine Codierung des zweiten „an“ in Frage käme, was aber nicht lohnt. Dann wird im Codebook „ban“ gefunden und codiert. Übrig bleibt „is“, eine Codierung lohnt wieder nicht. Das Ergebnis ist also „bananis$\langle 0,3\rangle$is“, was 11 Zeichen entspricht. Besser wäre jedoch die Codierung „bananisb$\langle 3,4\rangle$“ von nur 10 Zeichen Länge. Das Problem läßt sich leicht in der Eigenschaft des Suchalgorithmus ausmachen, mögliche Verweise *gierig* zu suchen. Um dieses Problem weitestgehend zu vermeiden, wird der Algorithmus dahingehend verfeinert, daß immer, wenn ein möglicher Verweis ansteht ($\langle 0,3\rangle$), auf den kein weiterer folgen kann („is“), untersucht wird, ob das nächste Zeichen im Quelltext einen weitergehenden Verweis liefern würde ($\langle 3,4\rangle$). Gelöst ist das Problem damit jedoch nicht, denn das leicht abgewandelte Beispiel „bananissbaniss“ wird bei gieriger Suche codiert zu „bananiss$\langle 0,3\rangle\langle 5,3\rangle$“, besser wäre jedoch „bananissb$\langle 3,5\rangle$“. Das geschilderte Problem hängt wesentlich von der gewählten Codierung der Verweise ab: Ist ein Verweis nicht länger als ein Zeichen, so tritt das Problem nicht auf. Ist ein

Verweis so lang wie n Zeichen, so kann auch ein um $n-1$ Zeichen verspätetes Muster noch Vorteile bringen.

5.7 Lempel-Ziv Welch

Terry A. Welch [WE] stellt das Codebook-Verfahren *LZW* vor (Storer [ST,p.69] bezeichnet diese Methode als *dynamic dictionary method*). Das Codebook ist zunächst leer. Während der Quelltext verarbeitet wird, werden in das Codebook Teilfolgen des Quelltextes eingetragen. Die ersten Einträge ins Codebook sind alle auftretenden Zeichenpaare. Sobald ein solches Zeichenpaar abermals auftritt, wird ein Verweis ins Codebook ausgegeben und — da dieses Zeichenpaar ja schon im Codebook steht — das Zeichenpaar um das folgende Zeichen erweitert und ins Codebook eingetragen. Als Beispiel wird wieder der Satz „Fischers_Fritz_fischt_frische_Fische." codiert. Das Ergebnis ist „Fischers_Fritz_f$\langle is\rangle$ $\langle ch\rangle$t$\langle _f\rangle\langle ri\rangle\langle sc\rangle\langle he\rangle\langle _F\rangle\langle isc\rangle\langle he\rangle$.". Das Codebook hat zuletzt einen Eintrag weniger, als Zeichen und Codes ausgegeben wurden: $\{Fi, is, sc, ch, he, er, rs, s\_, \_F, Fr, ri, it, tz, z\_, \_f, fi, isc, cht, t\_, \_fr, ris, sch, he\_, \_Fi, isch, he.\}$ Bis zum kleinen „f" tritt kein Zeichenpaar wiederholt auf, daher sind die ersten 16 Einträge im Codebook Zeichenpaare. Dann wird das Paar $\langle is\rangle$ verwendet und die Zeichenfolge $\langle isc\rangle$ wird ins Codebook eingetragen. Später wird $\langle isc\rangle$ verwendet und daher als neuer, vorletzter Eintrag $\langle isch\rangle$ erzeugt. Es ist sichergestellt, daß keine Zeichenfolge, die ins Codebook eingetragen wird, schon dort enthalten war, da sie sonst zuvor gefunden und verwendet worden wäre.

Codiert wird einfach, indem das Quellalphabet und der Inhalt des Codebooks zu einem gemeinsamen Zielalphabet vereinigt werden. Das Codebook wird mit jedem Zeichen, das ausgegeben wird, um einen Eintrag erweitert. Daraus ergibt sich, daß für die binäre Codierung des Ausgabealphabets im Laufe der Kompression eine steigende Anzahl von Bits je Zeichen benötigt wird. Welch bemerkt, daß zwölf Bit je Zeichen üblich seien, wobei Quellalphabete zu acht oder neun Bit angenommen werden. Bei einem Quellalphabet zu acht Bit kann jedoch das erste Zeichen der Ausgabe wieder in acht Bit codiert werden, die folgenden 256 Zeichen in höchstens neun Bit, weitere 512 Zeichen in höchstens zehn Bit, und so weiter.

Das Codebook wird prinzipiell mit jedem neuen Eintrag größer. Es ist jedoch auch möglich, das Codebook auf eine feste maximale Anzahl von Einträgen zu begrenzen, und, sobald es gefüllt ist, entweder keine neuen Einträge aufzunehmen oder — nach aus dem Bereich der cache-Speicher bekannten Strategien — alte Einträge zu löschen. Gelöscht werden können etwa diejenigen, die am längsten nicht benutzt wurden, oder solche, die am seltensten benutzt wurden.

Mitunter wird auch einfach das gesamte Codebook gelöscht, sobald es voll ist und die Kompressionseffizienz nachläßt, um dann von neuem gefüllt zu werden [FAQ].

Das Verfahren *LZW* ist verlustfrei, basierend auf REFERENZIERENDER REDUNDANZ, mit ADAPTIVER REDUNDANZANPASSUNG. Codiert wird entweder direkt — wie beschrieben — oder durch einen nachgeschalteten Kompressor, zum Beispiel Huffman. Dabei ist zu beachten, daß der Code ständig erweitert wird. Die direkte Codierung ist VARIABLE-TO-FIXED, bei nachgeschalteter Huffmankompression ist die Codierung VARIABLE-TO-VARIABLE. Wie bei *LZSS* hängt die Frage der SYMMETRIE vom Suchalgorithmus im Codebook ab.

Die Implementierung des Verfahrens *LZW* wird wesentlich vereinfacht, wenn alle Zeichen des Quellalphabets als einelementige Zeichenfolgen in das Codebook mit aufgenommen werden. Alle Zeichen der Ausgabe können dann als Verweise ins Codebook aufgefaßt werden. Das Codebook läßt sich leicht als zweispaltige Tabelle implementieren (Bild 19, die Spalte „s“ gibt den repräsentierten Inhalt wider und gehört nicht zur Tabelle). Die erste Spalte enthält dabei das letzte

	1	2	s
1	a	-	a
2	b	-	b
3	c	-	c
4	d	-	d
5	r	-	r
6	x	-	x
7	b	1	ab
8	r	2	br
9	a	5	ra
10	c	1	ac
11	a	3	ca
12	d	1	ad
13	a	4	da
14	r	7	abr
15	x	9	rax

Bild 19 ←

Bild 20 →

	1	2	3	4	s
1	a	-	7	-	a
2	b	-	8	-	b
3	c	-	11	-	c
4	d	-	13	-	d
5	r	-	9	-	r
6	x	-	-	-	x
7	b	1	14	10	ab
8	r	2	-	-	br
9	a	5	15	-	ra
10	c	1	-	12	ac
11	a	3	-	-	ca
12	d	1	-	-	ad
13	a	4	-	-	da
14	r	7	-	-	abr
15	x	9	-	-	rax

Zeichen des repräsentierten Wortes, die zweite enthält den Code für den Anfang des Wortes. Die zweite Spalte enthält für die initialen Einträge, die Zeichen des Quellalphabets also, nil-Zeiger „–“. Der Code wird nicht verzeichnet, da er implizit durch den Index in die Tabelle gegeben ist.

Um den Algorithmus zu beschleunigen, kann der weiter oben beschriebene *Suf-*

fixbaum erfolgreich angewendet werden. Dieser kann, etwas entartet, implementiert werden, indem die Codebooktabelle um weitere zwei Spalten erweitert wird (Bild 20). Die in diesen beiden Spalten enthaltenen Werte sind — wie auch die Werte in der zweiten Spalte — Zeiger, die als Indizes dargestellt sind. Während es die ersten beiden Spalten ermöglichen, die in einer Zeile enthaltene Zeichenfolge in ihre Bestandteile aufzulösen („rax": 15 $\rightarrow$ 9+x $\rightarrow$ 5+a+x $\rightarrow$ r+a+x), dienen die dritte und vierte Spalte dazu, ein gesuchtes Wort aus seinen Bestandteilen zusammenzusetzen und so in der Tabelle zu finden. Die dritte Spalte zeigt jeweils auf ein Wort, dessen Anfang das aktuelle Wort ist — das ist der umgekehrte Zeiger aus Spalte zwei. Die vierte Spalte zeigt auf ein weiteres Wort mit demselben Anfang wie das aktuelle Wort, also eines, dessen Eintrag in Spalte zwei der gleiche ist wie der des aktuellen.

Zwei Beispiele: Um das Wort „rax" zu finden, fängt die Suche in Zeile 5 an: „r". Spalte drei verweist auf ein Wort, das mit „r" anfängt: Zeile 9 enthält das Wort „ra". Da „a" (Spalte eins) das gesuchte zweite Zeichen ist, wird wieder über Spalte drei ein Wort gesucht, das mit „ra" anfängt: Zeile 15 enthält das Wort „rax". Damit ist die Suche beendet.
Um das Wort „ad" zu finden, fängt die Suche in Zeile 1 an: „a". Spalte drei verweist auf ein Wort, das mit „a" anfängt: Zeile 7 enthält das Wort „ab". Aber „b" (Spalte eins) ist nicht das gesuchte zweite Zeichen. Deshalb werden von hier aus über Spalte vier weitere Worte gesucht, die mit „a" anfangen: Zeile 10 ist auch noch nicht das gewünschte Wort („ac"), aber auch hier zeigt Spalte vier auf ein weiteres Wort, das mit „a" anfängt: Zeile 12 enthält das gesuchte Wort „ad".

Für die Implementierung mit vier Spalten ist die jeweilige Aktualisierung der Tabelle relativ einfach. Wurde ein Wortanfang w gefunden, nicht jedoch das neue Wort wx, so wird ein neuer Eintrag erzeugt, dessen erste Spalte „x" enthält. Die zweite Spalte zeigt auf die Zeile, die w enthält, die dritte und vierte Spalte sind leer. Ist bei w die dritte Spalte noch leer, so wird dort ein Zeiger auf die neue Zeile wx eingetragen, ist sie nicht leer, so wird — ausgehend von der Zeile, auf die der Eintrag der dritten Spalte bei w zeigt — an die Liste der Zeiger in der vierten Spalte die Zeile wx angehängt. Der Aufwand je Zeile, die neu eingetragen wird, hängt nur von der Größe des Quellalphabets ab, gleiches gilt für den Aufwand bei der Suche. Für eine Datei der Länge n ist die Aufwandsklasse also $O(n)$.

Noch schneller ist es natürlich, statt nur einer „Spalte 3" für jedes Zeichen z des Quellalphabets je eine „Spalte 3_z" anzulegen: Zu jedem bereits gefundenen Teilwort und jedem Zeichen kann sofort die Zeile gefunden werden, die das sich ergebende Wort enthält. Der Speicherplatzaufwand für die Tabelle ist allerdings sehr hoch, nämlich $k \cdot \#G$ Einträge, für Codebooklänge k und das Quellalphabet G [10].

Bei der Dekompression nach *LZW* wird das Codebook jeweils genauso aufgebaut, wie es auch bei der Kompression entstanden ist. Welch [WE] weist darauf hin, daß jedoch der letzte Eintrag des Codebooks vom Kompressor bereits verwendet werden kann, bevor der Dekompressor diesen Eintrag konstruieren kann. Dies ist der Fall, wenn eine Zeichenfolge der Form kωkωk codiert werden soll, während kω im Codebook eingetragen ist. Der Kompressor wählt diesen Eintrag aus und trägt kωk in sein Codebook ein. Als nächstes wählt der Kompressor den eben erzeugten Eintrag kωk aus. Wenn der Dekompressor das erste kω wiederhergestellt hat und den Code für kωk erhält, findet er diesen Code nicht in seinem Codebook, da dieser dort erst eingetragen werden kann, wenn das zweite k erzeugt worden ist. Also muß der Dekompressor für den Fall, daß er diesen ihm unbekannten Code empfängt, den letzten verwendeten Code kω und noch einmal dessen erstes Zeichen k ausgeben.

Wie Welch einräumt, wählt der Algorithmus die Einträge für das Codebook nicht optimal aus. Statt dessen sei darauf geachtet worden, den Algorithmus möglichst einfach zu halten. Um die Dynamik des Codebook zu ändern, werden in Varianten von *LZW* die Einträge in das Codebook nach anderen Kriterien bestimmt. Storer [ST,p.70] schlägt unter dem Namen *all-prefixes-heuristic* vor, öfters gleich mehrere neue Einträge zu machen, die aus den zwei zuletzt erkannten Worten gebildet werden (aus $x_1..x_{i-1}$ und $x_i..x_j$ bilde $x_1..x_i$, $x_1..x_{i+1}$, .. $x_1..x_j$). Gegenüber dem klassischen Algorithmus *LZW* erfolgen die Neueinträge verzögert, was dazu führt, daß das von Welch beschriebene Problem des unbekannten Codes entfällt. Außerdem wird das Codebook wesentlich schneller gefüllt, was größere Dynamik zur Folge hat.

Eine Kombination der *sliding dictionary method* und der *dynamic dictionary method* findet sich in [FAQ] [11]. Das Verfahren geht zunächst vor wie *LZSS*. Sobald ein Verweis in das Codebook erkannt und ausgegeben wird, wird dieses referenzierte Wort in ein zweites Codebook eingetragen. Das zweite Codebook stellt ein Langzeitgedächtnis dar und wird parallel zum ersten verwendet, welches entsprechend ein Kurzzeitgedächtnis ist. Ob auch Referenzen in das zweite Codebook zu dessen Veränderung führen, wird nicht beschrieben. Zwar werden beide Codebooks als von fester Länge beschrieben, beim zweiten Codebook kann aber je nach zu erwartender Dateilänge auch ein beliebig langes Codebook angenommen werden, da die Dynamik in diesem relativ gering ist. Man könnte

[10] Das sind bei einer Codebooklänge von $k = 4096$ und Bytes als Eingabe immerhin 1M Einträge.

[11] Abschnitt 8, Patent No 5001478.

auch die Länge des zweiten Codebook fest lassen und ein drittes, unbegrenztes Codebook einführen, in das Einträge aus dem zweiten Codebook gelangen, die mindestens n-mal referenziert worden sind. Damit wäre dann ein dreistufiges Gedächtnismodell geschaffen, ob jedoch eine Implementierung von Dementia Senilis sinnvoll ist, sei dahin gestellt.

5.8 Blaschkowski

Während die Codebook-Verfahren nur das wiederholte Auftreten zusammenhängender Worte berücksichtigen, erkennt Blaschkowski [BL] außerdem sowohl Zeichenketten, die nicht direkt zusammenhängen, sondern Fremdzeichen (*wildcards*) enthalten, als auch das regelmäßige Auftreten von Zeichen oder Zeichenketten in gleichen Abständen. Das verspricht höhere Kompressionsraten zu liefern, weniger bei natürlichsprachlichen Texten als bei Programmdateien und insbesondere Datenbänken.

Das Verfahren wird für $\#G = 2$ beschrieben, ist aber auf beliebige Grundmengen erweiterbar. Das insgesamt am häufigsten auftretende Zeichen wird ermittelt und besonders behandelt, im folgenden sei dies die Null. Nun wird über die gesamte Datei ermittelt, in welchen Abständen zueinander gleiche Zeichen auftreten (Die Null wird nicht betrachtet) (Bild 21). Dabei wird die Datei als

ORIGINAL	0	0	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0	0	0	0	1	0	0	1
ABSTÄNDE	^{1} 0	^{2} 0	^{3} 4	^{4} 0	^{5} 1	^{6} 2	^{7} 1	^{8} 2	^{9} 1	^{10} 2	^{11} 1	^{12} 0	^{13} 1	^{14} 2	^{15} 1	^{16} 2	^{17} 1	^{18} 2	^{19} 1	^{20} 0	^{21} 4	^{22} 0	^{23} 0	
	0	0	0	0	$\frac{1}{3}$	0	0	0	0	0	$\frac{1}{3}$	0	0	0	0	0	0	0	0	0	$\frac{1}{3}$	0	0	0
RESULTAT	0	0	0	0	$\frac{1}{3}$ 6	0	0		0	0		0	0		0	0	0	0	0	0	$\frac{1}{3}$	0	0	

Bild 21

Ring aufgefaßt, Ende und Anfang also verbunden. Der häufigste Abstand a wird ermittelt und es wird überall dort, wo zwei gleiche Zeichen in diesem Abstand auftreten, das erste zu einem Tupel erweitert, und zwar durch Anhängen des Abstandes, das zweite wird durch eine Null ersetzt. Aus jedem a$x_1..x_{n-1}$a wird also ${}^{\text{a}}_{n}x_1..x_{n-1}0$. Dieser Vorgang des Ermittelns der am häufigsten in einem bestimmten Abstand auftretenden Tupel — die Datei besteht jetzt aus Tupeln, nicht mehr aus einfachen Zeichen — und des anschließenden Anhängen des Abstandes und Ersetzen des hinteren Tupels wird solange wiederholt, bis in der Datei keine zwei gleichen Tupel mehr zu finden sind. Schließlich wird geprüft, welche der übrig gebliebenen Tupel groß genug sind, daß sich eine Codierung lohnt. Diejenigen, für die sich eine Codierung nicht lohnt, werden wieder expandiert und aufgelöst. Für die lohnenden Tupel werden alle bei der Erzeugung des jeweiligen Tupel erzeugten Nullen aus der Datei herausgenommen. Die Datei wird also um die vielen referenzierten Nullen gekürzt.

Das Verfahren läßt sich natürlich noch in allen Parametern optimieren, die erfolgversprechende Grundidee jedoch ist wichtig: es werden nicht Häufigkeiten von Zeichen oder sich wiederholende kohärente Zeichenfolgen betrachtet, sondern Häufigkeiten von Abständen, beziehungsweise sich regelmäßig wiederholende Zeichen oder nicht kohärente Zeichenfolgen.

Für die praktische Anwendung sind die folgenden Fragen interessant:

Führt eine Limitierung der untersuchten Abstände zur Verringerung der Aufwandsklasse ohne nennenswerte Beeinträchtigung des Kompressionsergebnisses?

Da ja die Tupel zum einen aus dem ursprünglichen Zeichen und zum anderen aus einer Abfolge von Abständen bestehen: Kann das Resttupel der Abstände getrennt von den Zeichen bearbeitet werden?

Führt die Wahl des jeweils häufigsten Abstandes zum besten Kompressionsergebnis?

Die Blaschkowskicodierung ist ein verlustfreies ASYMMETRISCHES Verfahren, basierend auf REFERENZIERENDER REDUNDANZ, mit VARIABLER ANPASSUNG in mehreren Analysepässen.

5.9 Hilberg

Hilberg [HI] schlägt die *texturale Sprachmaschine* vor, bei der in einer Reihe von Assoziativspeichern Zusammenhänge im Text abgelegt werden, woraufhin sich der eigentliche Text als Tupel von Initialwerten in diese Speicher repräsentieren läßt. Hierzu zunächst einige Vorbemerkungen:

Während in einem normalen Speicher die Auswahl von Daten über Adressen geschieht, denen jeweils eine physikalische Speicherzelle zugeordnet ist, wird der Wert beim *Assoziativspeicher* durch einen Schlüssel ausgewählt. Dabei ist unerheblich, in welcher Speicherzelle das Paar aus Schlüssel und Wert gefunden werden. Während der teilassoziative Speicher explizit zwischen Schlüssel und Wert unterscheidet (Bild 22), kann beim vollassoziativen Speicher jedes Datum sowohl Schlüssel als auch Wert sein (Bild 23), indem einige Schlüssel an den Speicher angelegt und andere maskiert werden. Das Problem bei assoziativen

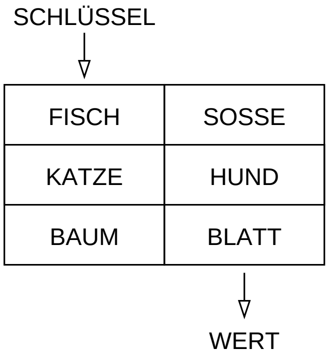

Bild 22

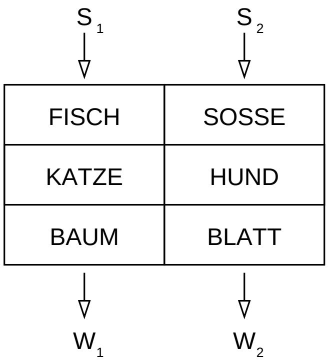

Bild 23

	HUNDE	KATZEN	BELLEN	FRESSEN	LAUT	FISCH
HUNDE			×	×		
KATZEN				×		
BELLEN					×	
FRESSEN					×	×
LAUT		×				×
FISCH	×					

Bild 24

Speichern ist, daß eventuell zu einem Schlüssel keine Zelle gefunden wird, oder mehr als eine.

In konkreten Texten folgen Wörter aufeinander, und zwar in bestimmten Abfolgen. Es gibt bestimmte Wortkombinationen, die häufig vorkommen, andere kommen in sinnvollen Texten mit Sicherheit nicht vor. Als Übergansmatrix wird eine boolsche Funktion bezeichnet, die zu zwei Wörtern, dem Vorgängerwort und dem Nachfolgerwort, angibt, ob diese aufeinander folgen können (oder sollen, je nach Interpretation. Für den Text „Hunde bellen laut, Katzen fressen Fisch, Hunde fressen laut Fisch“ ergibt sich unter Weglassung der Interpunktion die Matrix in Bild 24. Das erweiterte assoziative Feld ist eine Speicheranordnung, durch die anhand des Vorgängerwortes (W) und eines Auswahlschlüssels (X) das Nachfolgerwort (W') bestimmt werden kann (Bild 25). Es kann allerdings auch zu Vorgänger- und Nachfolgerwort durch Maskierung des Eingangs X der Auswahlschlüssel bestimmt werden. Dies wird für den gesamten zu codieren-

Bild 25

den Text gemacht. Die vom erweiterten assoziativen Feld erzeugten Auswahlschlüssel werden unter Auslassung des ersten zu Paaren zusammengefaßt und in einer zweiten Ebene in ein weiteres assoziatives Feld gesteckt. Hierbei ergeben sich neue Übergangscodes. Es können beliebig viele Ebenen folgen, wobei sich in jeder Ebene die Anzahl der Wörter und Übergangscodes halbiert (Bild 26). Die so gebildete Struktur wird semantischer Speicher genannt. Als Codierung des Eingabetextes erhält man dadurch für einen Text der Länge n größenordnungsmäßig $\log(n)$ Schlüssel von jeweils wenigen Dutzend bit Länge, ein Text von mehreren Megabyte läßt sich auf wenige Hundert bit verlustfrei komprimieren. Das klingt phantastisch, und natürlich ist ein Haken bei der Sache: Der semantische Speicher muß natürlich alle Wortkombinationen und alle Kombinationen dieser Kombinationen und so fort kennen, um jeweils im erweiterten assoziativen Feld den richtigen Folgeschlüssel zu finden. In der Praxis wird dies für vorab unbekannte Texte wohl kaum gelten, und daher muß das Konzept um eine Ausnahmebehandlung erweitert werden. Bisher beinhaltete der semantische Speicher eine statische Redundanzanpassung. Als Änderung beschreibt Hilberg die Einführung von Escape-Codes, um für nicht im erweiterten assoziativen Feld befindliche Schlüssel eine Ausnahmebehandlung zu liefern, denkbar ist auch die Erweiterung des semantischen Speichers im Betrieb, also eine adaptive Redundanzanpassung. Weniger sinnvoll wird eine variable Redundanzanpassung sein,

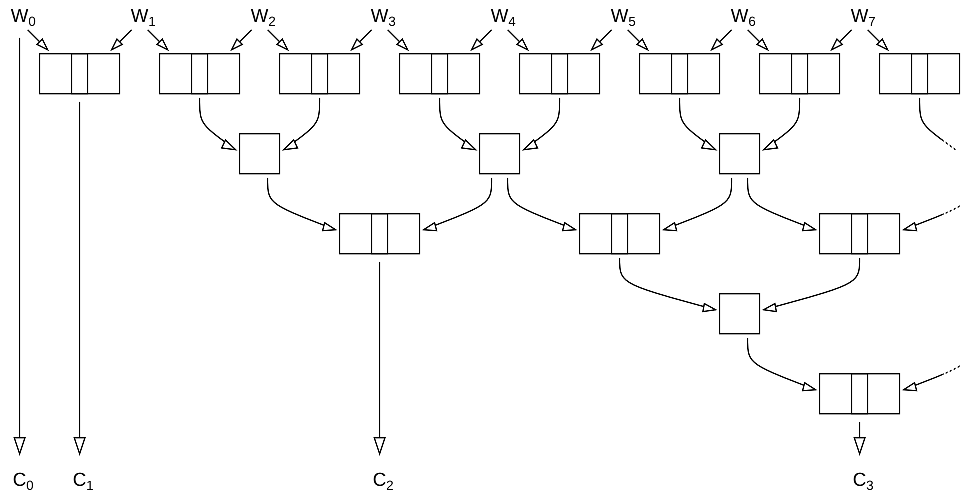

Bild 26

da hierbei der Inhalt des semantischen Speichers mit dem Komprimat mitgeliefert werden müßte. Ob im Falle einer adaptiven Redundanzanpassung der initiale Zustand des Speichers leer sein kann, ist ebenfalls fraglich, da eine nur sehr träge Anpassung zu erwarten ist. Denkbar wäre der semantische Speicher als dauerhafter zentraler, gemeinsamer Komprimierer eines größeren Archives.

Die bisherigen Versuche mit einem statischen Modell ergaben schon bei unbekannten Texten der gleichen natürlichen Sprache so viele Escape-Codes, daß durch die Codierung in Übergangscodes eine Kompression gerade auf die Hälfte des Originaltextes erreicht wurde. Die beschriebene Struktur *semantisch* zu nennen, ist wohl übertrieben, da das paarweise Bündeln von Wörtern nicht viel mit Verständnis zu tun hat.

Der semantische Speicher ist ein verlustfreies Verfahren, basierend auf REFERENZIERENDER REDUNDANZ, je nach Implementierung mit STATISCHER oder ADAPTIVER ANPASSUNG. Die einzelnen Zeichen der Ausgabeströme sind jeweils fest, die Codierung ist also VARIABLE-TO-FIXED.

5.10 Primitive Verfahren

Der Vollständigkeit halber sollen ein paar Verfahren erwähnt werden, die zwar nicht besonders effektiv, meist jedoch sehr simpel sind und daher heute noch breite Verwendung finden.

Die Kompression von *Lauflängen* wurde schon erwähnt: Dieses Verfahren, das sich wiederholende Zeichen zählt und dann nur den Zähler speichert, bietet sich oft geradezu an. Die Lauflängenverfahren gehören in die Kategorie der REFERENZIERENDEN REDUNDANZ, die Anpassung ist STATISCH. Die allgemeineren Varianten dieses Verfahrens lassen jedes Zeichen als sich wiederholendes zu und vermerken bei einer Lauflänge das Zeichen und die Länge. Die Codierung erfolgt dabei zum Beispiel mit Escape-Code, oder durch regelmäßiges Mischen zweier Codeströme, also abwechselndes Zählen der normalen Zeichen und der Lauflänge. Beim Verfahren MNP-5, das in Modems eingesetzt wird, werden die ersten drei Zeichen der Lauflänge unverändert ausgegeben, dann folgt obligatorisch ein Zählerbyte [DR]. Technisch ist das sehr einfach zu realisieren, der Nachteil ist aber, daß Sequenzen von genau drei Zeichen nicht verkürzt sondern verlängert werden. Durchaus üblich sind aber auch spezielle Lauflängenverfahren, zum Beispiel der horizontale Tabulator. Dieser ist in allen Hinsichten statisch: Es werden nur Folgen von Leerzeichen komprimiert und die Länge der komprimierten Folge kann nur einen festen Wert annehmen, der von der relativen Position des Tabulatorzeichens zum Zeilenanfang abhängt. Man kann den Tabulator als Escape-Code auffassen, dem weder Zeichen noch Länge der dargestellten Lauflänge folgen, da diese Werte implizit sind. Ebenfalls ein spezielles Lauflängenverfahren ist das DLE (*Data Link Escape*), wie es im UCSD-System verwendet wird. Ähnlich wie beim Tabulator können auch hier nur Folgen von Leerzeichen behandelt werden, allerdings ist die Länge variabel, sie wird durch das dem DLE-Code folgende Byte angegeben. Vom UCSD-System selbst wird der DLE-Code nur am Zeilenanfang verwendet.

Held [HE,p.26ff] widmet sechs Seiten einzig dem Thema Lauflängen, nachdem die vorangegangenen acht Seiten der Nullunterdrückung (Lauflängen aus Nullen) und dem Bitmapping gelten. Bitmapping ist an sich kein Kompressionsverfahren, sondern eine Codierungstechnik zum Mischen zweier Codeströme. Erwähnenswert ist noch ein Verfahren, das Held als „Half-digit suppression“ [HE,p.25] oder als „Half-byte packing“ [HE,p.32] bezeichnet. Hierbei werden in einem laufenden Text, in dem Folgen von Ziffern auftreten, diese Ziffern in BCD (*Binary Coded Decimal*) dargestellt, so daß je Ziffer nur vier statt acht Bit anfallen. Allerdings müssen auch hier wieder zwei Codeströme gemischt werden, der restliche Text und die BCD-Gruppen.

Weiter schlägt Held [HE,p.40ff] die *diatomische Codierung* vor, wobei häufig auftretende Zeichenpaare durch ein einzelnes Sonderzeichen ersetzt werden sollen. Dieses Verfahren gehört in die Kategorie der STATISTISCHEN REDUNDANZ, da die Häufigkeiten der Paare ausschlaggebend für die Codierung ist.

Für die Kompression von digitalisierten akustischen Daten findet das Verfahren

ShortPack verbreitet Anwendung. Während die Quelldaten in Werten zu 16 Bit vorliegen, sind die höherwertigen Bits oft über lange Strecken redundant, besonders wenn die Daten akustische Signale geringer Amplitude repräsentieren. Das Programm prüft, ob die Amplitude der Daten betragsmäßig mittelfristig unter einer Schranke von 2^{n-1} bleibt und stellt die entsprechenden Daten mit nur n Bit dar, geführt von einer Formatinformation. ShortPack ist ein einfaches STATISTISCHES Verfahren mit ADAPTIVER ANPASSUNG.

5.11 Schwarzweißbilder

Die für Bilder mit einem Bit je Bildpunkt zur Anwendung kommenden Kompressionsverfahren leiten sich meist aus eindimensionalen Verfahren ab und sind somit keine grundsätzlich neuen Verfahren.

Als Beispiel für ein zweidimensionales statistisches Verfahren wird hier eine arithmetische Datenkompression mit adaptiver Anpassungsstrategie und Markovverhalten beschrieben. Das Bild werde von links oben nach rechts unten zeilenweise abgetastet. Alle Wahrscheinlichkeiten werden in Abhängigkeit von oberhalb und links liegenden Punkten betrachtet. Als Referenzpunkte für den Punkt (x,y) dienen die vier Punkte (x-1,y), (x-2,y), (x,y-1) und (x-1,y-1). Daraus ergeben sich sechzehn verschiedene Fälle nach Markov, die getrennt zu behandeln sind. Es gibt also für jeden Fall $i = 0..15$ je eine Wahrscheinlichkeit $p_{[i]}$ für das Auftreten einer Eins. Diese Wahrscheinlichkeiten werden zunächst allesamt auf $p_{[i]} = \frac{1}{2}$ gesetzt. Für jeden verarbeiteten Bildpunkt muß die entsprechende Wahrscheinlichkeit $p_{[i]}$ angepaßt werden. Das kann umgekehrt exponentiell geschehen (z.B. $p_{[i]} := \frac{3}{4}p_{[i]} + \frac{30b+1}{128}$, so daß immer $\frac{1}{32} \leq p_{[i]} \leq \frac{31}{32}$ (Bild 27); einfacher mit Rundung), oder linear durch Merken der vorangegangen aufgetretenen Bits in gleichen Fällen (z.B. Merkregister je 30 bit, altes Bit a rausschieben, neues Bit b reinschieben, $p_{[i]} := p_{[i]} + \frac{b-a}{32}$, die Merkregister werden mit einem Graumuster initialisiert (Bild 28)). Bei einer maximalen Grenzwahrscheinlichkeit p_{max} (im Beispiel $p_{max} = \frac{31}{32}$) ergibt sich also für den Fall, daß immer das wahrscheinlichere Bit auftritt, eine obere Grenze für die Kompressionsrate von $1 : \log_{p_{max}} \frac{1}{2}$ (im Beispiel etwa $1 : 21.86$). Ähnliche Verfahren werden von Langdon und Rissanen [LR] ausführlich untersucht, das Markovverhalten bezeichnen sie als *neighborhood template conditioning*.

Oft sind Schwarzweißbilder flächig monochrom, so daß sich ein zweidimensionales Lauflängenverfahren anbietet. Dabei wird versucht, möglichst große Flächen gleicher Färbung auszumachen und durch Vektoren zu beschreiben. Im einfachsten Fall sind diese Flächen Rechtecke. Es sind natürlich auch Kreise oder andere

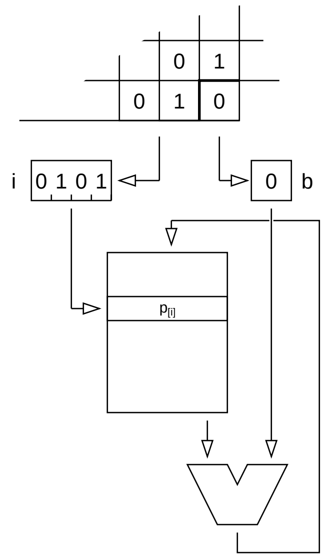

Bild 27

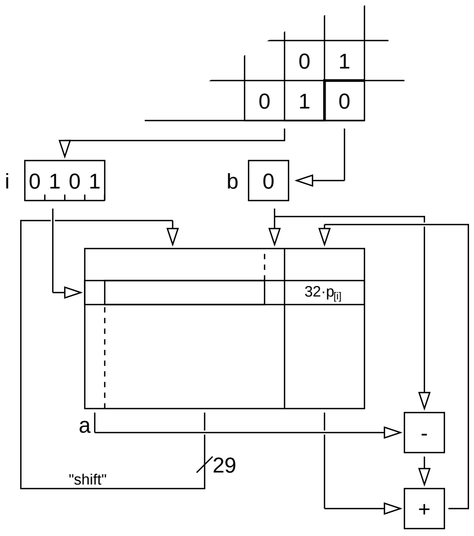

Bild 28

Flächen als Überdeckungsflächen denkbar, da jedoch die Grundstruktur des gerasterten Bildes das Rechteck ist, führen Rechtecke als Überdeckungsflächen zu einem besonders einfachen Algorithmus. Der aufwendigste Teil ist die Textanalyse, bei der möglichst große Rechtecke gesucht werden. Krüger [KR] stellt ein Verfahren vor, das eben solche Rechtecküberdeckungen verwendet. Krüger versucht, das gesamte Bild mit Rechtecken zu überdecken, was natürlich dazu führt, daß besonders kleine Flächen gute Kompressionsraten verhindern, denn Flächen der Größe $1 \cdot 1$ lassen sich schwerlich durch ein einzelnes Bit beschreiben. Zu diesem Problem gibt es zwei Lösungsansätze. Der eine, den Krüger verfolgt, besteht darin, das Bild als zeilenweise Datei von Bytes zu betrachten und Flächenbreiten nur in Vielfachen von acht Pixeln zu verwenden. Das löst das Problem der besonders kleinen Flächen zwar nur teilweise, bringt aber für die Realisierung des Algorithmus auf einem gewöhnlichen Prozessor den Vorteil, daß die Daten nicht mehr bitweise verarbeitet werden müssen. Eine echte Lösung für das Problem ist hingegen, nur Flächen ab einer bestimmte Größe zu beschreiben. Diese Größe hängt dabei direkt davon ab, wieviele Bits für die Beschreibung eines Rechtecks benötigt werden. Alle Pixel, die durch kein Rechteck überdeckt werden, bilden einen separaten Codestrom und es ist Aufgabe der Codierung, die beiden Ströme zu mischen, zum Beispiel durch Abzählen von Rohpixeln.

Als Versuch habe ich das Verfahren *SQ91* entwickelt, mit dem echt bitweise

Überdeckung realisiert wird. Es codiert das Bild zeilenweise durch einer Folge von Befehlen `R` und `Z`. Die Befehle `R` beschreiben überlappungsfreie Rechtecke, die Befehle `Z` zählen Restpixel ab, die nicht von Rechtecken überdeckt sind (Bild 29). Der Dekompressionsalgorithmus stellt das Bild von links oben `(0,0)`

```
00000aaa aaaaaaab bbbbbbbb: R(a,b)
0001aaaa aabbbbbb: R(a,b)
0010aaaa aabbbbbb: R(a,b+64)
0011bbbb bbaaaaaa: R(a+64,b)
0100aaaa abbbbbbb: R(a,b+128)
0101bbbb baaaaaaa: R(a+128,b)
0110aaaa bbbbbbbb: R(a,b+256)
0111bbbb aaaaaaaa: R(a+256,b)
10cccccc: Z(c+1)
11cccccc cccccccc: Z(c+65)
```

Bild 29

zeilenweise Punkt für Punkt wieder her. `R(a,b)` zeichnet ein Rechteck an der aktuellen Position (x,y)..(x+a,y+b), `Z(c)` erzeugt im freien Bereich, also dort, wo noch keine Rechtecke vorgesehen sind, c Pixel. Punkte, die durch einen Befehl `R` bereits Teil eines Rechteckes geworden sind, werden bei der punktweisen Wiederherstellung des Bildes übersprungen. In den aus Befehlen bestehenden Datenstrom werden Bytes eingemischt, die jeweils acht Farbbits für Pixel oder Rechtecke enthalten[12].

Die Methode nach Krüger und auch die Methode der echt bitweisen Überdeckung *SQ91* von Rechtecken gleicher Färbung führen zu ähnlich guten Ergebnissen. Wird das Bild einfach vertikal byteweise ausgelesen, also in Spalten von 8 Pixeln Breite, und diese dann durch einen speziellen Lauflängencodierer geschickt, ergeben sich allerdings vergleichbare Kompressionsraten. Natürlich ist dieses Verfahren auch wesentlich schneller, da die einigermaßen aufwendige Suche nach Rechtecken entfällt.

Es sei noch angemerkt, daß die Zahl der Dimensionen natürlich variabel ist, daß das, was im zweidimensionalen funktioniert, natürlich auch räumlich anwendbar ist. Zum einen kann die dritte Dimension hier einfach die dritte räumliche Dimension bedeuten, Krüger führt als Beispiel medizinische Schichtbilder an, zum anderen kann die dritte Dimension aber auch die Zeitachse bedeuten, ein Film

[12] Als kleines Beispiel hier die Codierung, mit der ein Bild der Größe $8 \cdot 8$ beschrieben wird, das komplett schwarz (0) ist, bis auf vier weiße (1) Punkte in den Ecken: `Z(1) <10100000> R(5,7) Z(15) <00000001> <1>` → `$80.A0.1147.8E.01.80 .` `Z(1) <1>` erzeugt den ersten Punkt, `R(5,7) <0>` erzeugt rechts davon ein schwarzes Rechteck der Größe $6 \cdot 8$ und `Z(15) <10..011>` füllt die übrigen 15 Punkte des Bildes auf.

also als räumliches Gebilde aufgefaßt werden. Als Beispiel dafür, daß in der zeitlichen Dimension ungeahnte Lauflängen ruhen, diene der Nachrichtensprecher, der nichts weiter tut als den Mund auf und zu zu machen. Das mindeste, was in solch einem Fall zu tun ist, ist eine Differenzcodierung von einem Bild zum nächsten.

Praktisch angewendet werden jedoch, da aus technischen Gründen gerasterte Bilder meist getrennt zeilenweise bearbeitet werden, eindimensionale Verfahren. So wird bei der Übertragung von gescannten Schwarzweißbildern als Fax nach dem Standard für Geräte der Gruppe 3 eine Verkettung eines zeilenweisen Lauflängenverfahrens mit einer statischen Huffmancodierung eingesetzt [HR].

6 Vorbehandlungsverfahren

Die folgenden Verfahren stellen für sich keine Kompressionsverfahren dar, sind jedoch geeignet als Vorstufen zur Vorbereitung der Daten auf die Kompression.

6.1 Abstandscodierung

Lanzerath [LA] stellt Ansätze vor, die versuchen, Zeichen dadurch zu beschreiben, daß der Abstand zur Stelle ihres letzten Auftretens angegeben wird. Dabei wird im großen und ganzen bei den verschiedenen Ansätzen der Begriff *Abstand* variiert. Die simplen Formen sind der Abstand als Anzahl der dazwischen befindlichen Zeichen, oder als Anzahl der dazwischen befindlichen unterschiedlichen Zeichen. Das Wort *abracadabra* wird also codiert zu *abr2c1d1662* beziehungsweise *abr2c1d1442*. Bemerkenswert ist, daß jedes Zeichen des Quellalphabets in der Ausgabe nur einmal vorkommen kann. Zunächst führt der Algorithmus noch nicht zu einer Kompression, da genausoviele Zeichen ausgegeben werden, wie der Kompressor liest, es sollte also eine geschickte Codierung folgen, beispielsweise könnte man ein adaptives Huffman-Verfahren auf die Ausgabe anwenden. Die zweite Variante ist deswegen interessant, da für ein Quellalphabet der Größe $\#G$ nur die Abstände 0 bis $\#G-1$ auftreten können. Weiterhin kann das Auftreten der ursprünglichen Zeichen in der Ausgabe dadurch verhindert werden, daß als Vorlauf vor der Codierung das Quellalphabet angenommen wird, das erste Zeichen wird also bereits als Abstand codiert.

Eine weitere Verfeinerung der Abstandsdefinition besteht darin, die unterschiedlichen Zeichen in Abhängigkeit des jeweils vorangehenden Zeichens zu zählen. Lanzerath erwähnt es nicht, aber dies ist nichts anderes als die Ergänzung der zweiten Abstandsdefinition um Markovketten erster Ordnung. Es wird gezählt, wie oft das Vorgängerzeichen gefolgt von unterschiedlichen, anderen als dem aktuellen Zeichen seit dem letzten Auftreten der aktuellen Konstellation auftrat. Das Wort *abracadabra* wird codiert als *abracada200*: Die Codierung in Abständen setzt langsamer ein, dafür sind die Abstände jedoch tendenziell wesentlich kleiner, übersteigen auf jeden Fall für ein Quellalphabet der Größe $\#G$ wiederum nicht die Grenze $\#G-1$. Auch tritt jedes Zeichen nach einem bestimmten Vorgänger nur höchstens einmal auf. Dieses System kann man natürlich noch weiter ausbauen, indem Markovketten höherer Ordnung eingebracht werden, Lanzerath zieht hier jedoch zu Recht „die Grenze des noch Beherrschbaren“.

Dieses Verfahren basiert auf REFERENZIERENDER REDUNDANZ, mit ADAPTIVER ANPASSUNG.

6.2 Differenzcodierung

In vielen Fällen, besonders bei beabsichtigter verlustbehafteter Datenkompression, wird vor der eigentlichen Kompression die erste Ableitung über die Daten gebildet indem das einzelne Datum durch die Differenz zu seinem Vorgänger ersetzt wird. Diskrete Daten, die wenig variieren, werden dadurch im Wertebereich auf ein enges Intervall um den Nullpunkt eingeschränkt, was einem nachfolgenden statistischen Verfahren zuträglich ist. Besonders für Bilder ist dies interessant, da bei vielen Bildern scharfe Kante eher die Ausnahme sind. Bei Bildern wird dann die Differenz in horizontaler Richtung [RP] oder in vertikaler Richtung [HE,p.51ff], oder allgemeiner sowohl in horizontaler als auch in vertikaler Ausrichtung gebildet [HA,p.105ff]. Auch bei der Codierung von Sprache und anderen akustischen Daten wird Differenzcodierung eingesetzt [ANB].

Dieses Verfahren basiert auf REFERENZIERENDER REDUNDANZ, mit STATISCHER ANPASSUNG.

Verallgemeinernd kann die Differenzcodierung als spezieller Fall der prediktiven Codierung [EL] angesehen werden. Dabei werden vorangegangene Zeichen der Quelle ausgewertet um aus ihnen das zu erwartende folgende Zeichen vorauszusagen. Eine prediktive Codierung ist demnach umso besser, je öfter oder genauer das nächste Zeichen vorausgesagt wird. Anderson und Bodie [ANB] geben ein prädiktives Verfahren an, das sich jeweils mehrere Möglichkeiten, die gewünschten Daten zu approximieren, merkt und die sich als beste erweisende Möglichkeit erst einige Schritte später auswählt.

7 Codierung der Ausgabeströme

Bei mehreren Kompressionsverfahren ergibt sich bei der Codierung die Situation, daß zwei oder mehr getrennte Ausgabeströme erzeugt werden. Dies ist zum Beispiel bei Lauflängen der Fall, hier ist der eine Datenstrom der restliche Text, der andere sind die die Lauflängen beschreibenden Paare aus Länge und Zeichen. Ebenso tritt bei *LZSS* ein zweiter Datenstrom auf, der aus den Paaren aus Länge und relativem Abstand besteht.

Unter der Annahme, daß beide Datenströme als Präfixcode vorliegen, lassen sie sich jeweils als Baum darstellen, dessen Blätter die dargestellten Zeichen repräsentieren. Die einfachste Methode, die beiden Datenströme zu mischen, besteht demnach darin, beide Bäume an der Wurzel zu vereinigen, also eine neue Wurzel zu schaffen, von der über zwei Kanten jeweils die Wurzeln der beiden Teilbäume zu erreichen sind (Bild 30). Praktisch heißt das, daß die Codeworte

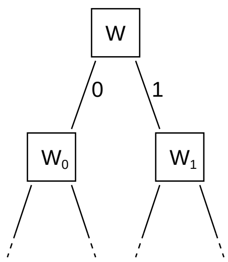

Bild 30

des einen Code um eine führende „0", die des anderen um eine führende „1" erweitert werden. Es entsteht wieder ein Präfixcode. Held `[HE,p.20]` nennt dieses Verfahren *bit mapping*.

Der gemeinsame Codebaum kann natürlich auch anders zusammengestellt werden oder umsortiert werden, die Benutzung eines *Escape-Codes* entspricht demnach einfach dem Anfügen des zweiten Baumes an das Escape-Blatt des Hauptbaumes. Das ausgeklinkte Escape-Blatt wird an den Escape-Teilbaum wieder als Blatt angefügt (Bild 31).

Während übliche Rechner Bits in Gruppen zu je acht oder Vielfachen hiervon verarbeiten, führt das bit-mapping dazu, daß pro Datum im Code ein zusätzliches Bit anfällt. In üblichen Rechnern führt dies weiterhin dazu, daß die abgelegten Daten vor und nach Verarbeitung aufgrund des relativ unüblichen Formats vermehrt zurechtgeschoben werden müssen. Ein Trick, diesem Problem zu begegnen, besteht darin, das jeweils zusätzliche Bit nicht dort zu platzieren, wo es

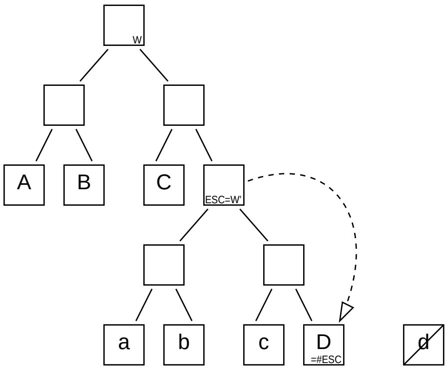

Bild 31

hingehört, sondern jeweils acht dieser Bits zusammenzufassen und auf Vorrat an der Stelle einzuwerfen, an der das erste der acht zusammengefaßten Bits erwartet wird. Verlangt also der Decodierer ein Bit zur Entscheidung darüber, welcher Codestrom gerade vorliegt, so greift er zum vorliegenden Byte, und nimmt aus diesem das erste Bit als Entscheidungsgrundlage. Die restlichen sieben Bit werden aufgehoben für die folgenden sieben derartigen Fälle. Der einzige Vorteil dieser Methode ist der, daß die Verarbeitung durch einen gewöhnlichen byteorientierten Prozessor beschleunigt wird[13]. Andere Prozessoren kennen Befehle, die eine Bearbeitung von Daten, die nicht im Byteformat vorliegen, unterstützen[14].

Natürlich kann auch der zusätzlich erzeugte Datenstrom einzelner Bits, die zur Unterscheidung der beiden Codeströme dienen, selbst einer Kompression unterzogen werden (Bild 32). Allerdings wird ein allgemeingültiges Kompressionsverfahren übertrieben sein. Sind die Daten des zweiten der ursprünglichen Codeströme eher selten, bietet es sich neben der Verwendung eines Escape-Code

[13] Beim mc68000 läßt sich ein Makro zum Lesen des jeweils nächsten Bit leicht in vier Befehlen unterbringen:

```
getbit: lsl.b   #1,dx
        bne     getbi2
        move.b  (ax)+,dx
        roxl.b  #1,dx
getbi2:
```

Das gewünschte Bit wird immer im Carryflag geliefert. Das Register dx wird mit $80 initialisiert, wobei das gesetzte Highbit eine Ringmarke ist, die zum Abzählen von je acht Bit dient. Rutscht die Marke ins Carryflag, so ist dx leer und es werden acht neue Bits geholt, die Ringmarke wird wieder nach dx gerollt. Das Register ax ist der Zeiger in den Eingabestrom.

[14] Zum Beispiel die Bit Field Befehle des mc68020.

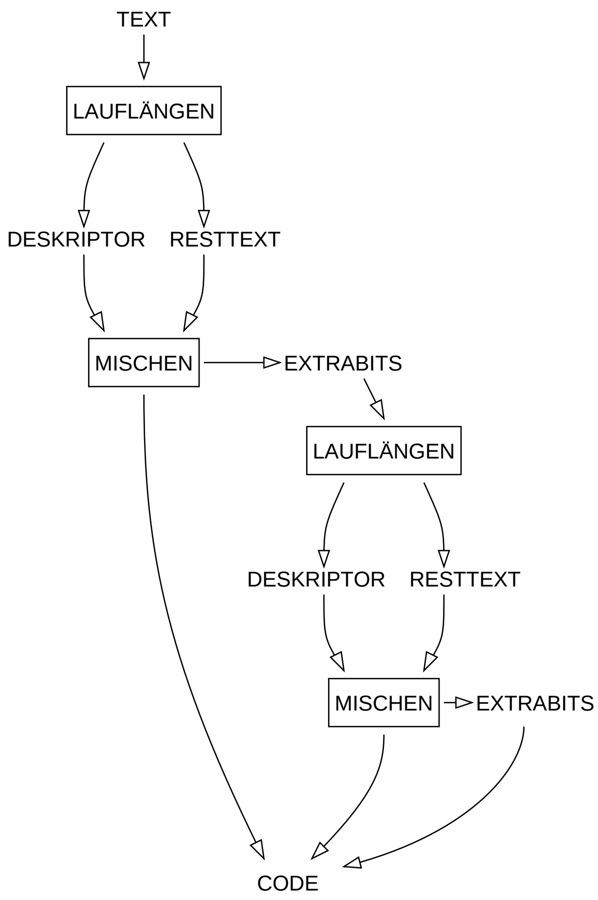

Bild 32

auch an, die Entscheidungsbits einer Lauflängencodierung zu unterziehen, was dann zu einer simplen Mischung aus Abzählen und Maskieren führt. Für den Fall etwa, daß Nullbits häufiger und in Folgen auftreten, könnten sieben einzelne Bits im Byte mit gesetztem Highbit verpackt oder ein Zähler zu sieben Bit, der 8 bis 135 Nullbits darstellt, mit gelöschtem Highbit untergebracht werden (Bild 33). Auf diese Art wird nichts anderes gemacht, als wieder zwei Datenströme zu mischen und einen Entscheidungsbitstrom hinzuzufügen.

Zwei Codeströme können auch gemischt werden, indem bei jedem Wechsel des Codestroms ein Zähler die Zahl der folgenden zum selben Codestrom gehörigen Zeichen angibt. In dem Fall, daß dies zusätzlich zu einer Lauflängencodierung der ursprünglichen Daten geschieht, sieht eine typische Codefolge etwa so aus:

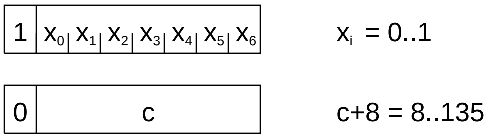

Bild 33

c a n x_1 .. x_n. Dies sei eine Codierung für c-mal a, gefolgt von n Zeichen x_1 .. x_n. Der Code besteht aus einer Abfolge solcher $(n+3)$-Tupel. Durch Setzen von $c = 1$ oder $n = 0$ kann jeweils ein Teil der abwechselnden Codierung bei Bedarf quasi ausgelassen werden. Der Unterschied zur Codierung mit Escape-Code ist der, daß kein besonderes Zeichen mehr als Escape-Zeichen aus dem Code herausgenommen werden muß.

Die theoretisch einfachste, praktisch jedoch etwas diffizile Methode, zwei Codeströme zu mischen, ist, einfach alle Zeichen beider Codes zusammenzuwerfen und dann beispielsweise nach dem Verfahren Huffman neu zu codieren. Das muß natürlich in vielen Fällen adaptiv geschehen, wenn die Zeichenmengen der einzelnen Codes zu Beginn der Codierung noch gar nicht vollständig bekannt sind.

8 Grauwertbilder und Farbbilder

Auf Farbbilder lassen sich einerseits gängige verlustfreie Kompressionsverfahren wie oben beschrieben anwenden (siehe den Abschnitt 5.11 „Schwarzweißbilder“), es ist jedoch bei Farbbildern wesentlich naheliegender als bei Schwarzweißbildern, verlustbehaftete Datenkompression einzusetzen, da mit zunehmender Darstellungsgenauigkeit die subjektiven Fehler bei minimaler Veränderung des Bildes abnehmen.

Es lassen sich im Groben zwei verschiedene Ansätze zur verlustbehafteten Datenkompression für Bilder unterscheiden: Der *transformierende* Ansatz geht vom kompletten Bild aus und transformiert dieses in einen solchen Bereich, in dem sich wichtige und unwichtige Anteile leicht trennen lassen. Dieser Bereich ist üblicherweise der Frequenzbereich. In diese Gruppe gehört das Verfahren JPEG (*Joint Photographic Experts Group*) [BA,BSR,BU], das auf einer Verkettung einer DCT (*Diskrete Cosinus Transformation*) [BT] mit weiteren Transformationen basiert (Bild 34). Anstatt der DCT lassen sich aber auch an-

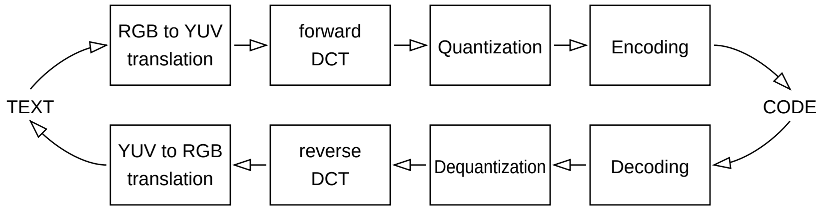

Bild 34

dere Transformationen zugrunde legen, wie die *Haar-Transformation* und die *Hadamard-Walsh-Transformation* [HA,p.84ff]. Der *vektorisierende* Ansatz geht davon aus, daß sich das Bild (textuell) konstruktiv beschreiben läßt (siehe den Abschnitt 10 „Vektorisierung“). Hierzu gehören die fraktalen Kompressionsverfahren wie IFS (Iterated Function System) [BSN], und blockfraktale Codierung [BVN]. Hierzu gehören auch Verfahren, die ein Bild durch geometrische Vektoren, die durch klassische Mustererkennung gefunden werden, zu beschreiben versuchen. Romaniuk [RO] schließlich stellt einen Ansatz vor, durch den zu einem gegebenen Bild ein neuronales Netzwerk konstruiert wird, welches wiederum das Ausgangsbild verlustfrei erzeugen kann.

9 Akustische Daten

Auch akustische Daten bieten sich im allgemeinen zur verlustbehafteten Datenkompression an. Da das menschliche Gehör den tatsächlichen Verlauf der Schallkurve nicht erkennt, sondern nur die enthaltenen Frequenzanteile, genügt es, bei der Wiederherstellung der akustischen Daten solche zu generieren, die ein ähnliches Spektrum liefern wie das Original. Dabei können Frequenzanteile weggelassen werden, sofern das menschliche Gehör diese nicht wahrnehmen kann. Das sind zum einen solche über der oberen Hörbarkeitsgrenze von 20 kHz, zum anderen sind aber auch schwache Frequenzanteile in der Nähe von starken nicht hörbar, sie werden überdeckt. Einfach dargestellt genügt es also, die Daten blockweise in den Frequenzbereich zu transformieren, etwa per DCT, dann jeweils den stärksten Frequenzanteil zu suchen und anhand einer Überdeckungskurve festzustellen, welche benachbarten Frequenzen vernachlässigt werden dürfen (Bild 35). Dies erfolgt solange, bis keine Frequenzanteile mehr

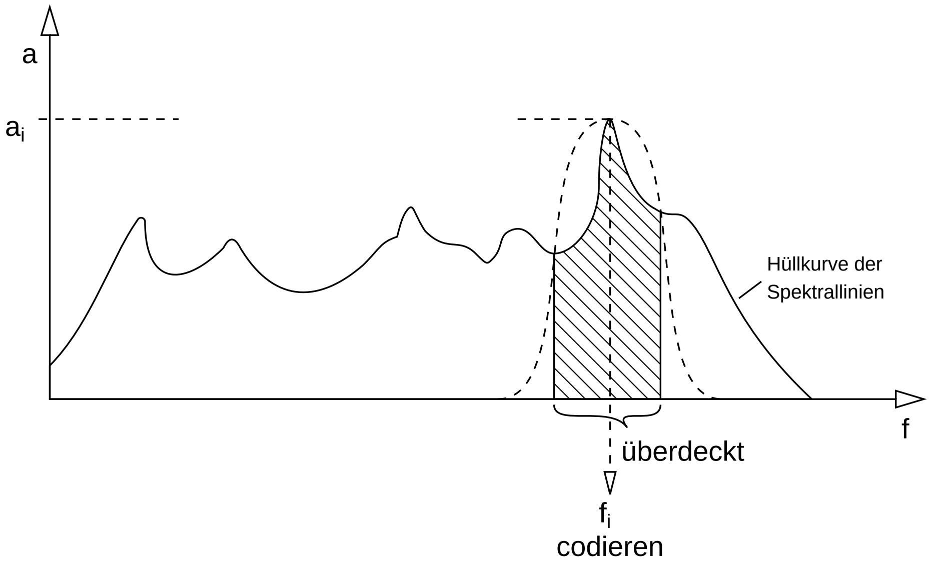

Bild 35

betrachtet werden müssen oder eine ausreichende Anzahl codiert ist.

Praktisch wird das Frequenzspektrum in mehrere Abschnitte aufgeteilt, die getrennt untersucht und codiert werden. Zwei derartige Verfahren werden in [FS] verglichen, *ATRAC* (*Adaptive Transform Acoustic Coding Technik*) und *PASC* (*Precision Adaptive Subband Coding*). Sollen die akustischen Daten weitere Verarbeitungsschritte durchlaufen können, so dürfen natürlich die unhörbaren Frequenzanteile nicht weggelassen werden, da diese Verluste durch nachfolgende Bearbeitungsschritte hörbar werden können. Als Beispiel stelle man sich einen starken Frequenzanteil f vor, in dessen Nähe alle Frequenzen überdeckt werden. Wird der Anteil f jedoch durch eine Bandsperre ausgefiltert, so werden die benachbarten Frequenzen wieder hörbar, was jedoch nicht der Fall wäre, wären sie zuvor aufgrund ihrer Unhörbarkeit entfernt worden.

10 Vektorisierung

Der Begriff *Vektorisierung* wird hier benutzt im Sinne von *(textuelle) konstruktive Beschreibung*. Alle Ansätze zur Vektorisierung von Daten haben gemeinsam, daß es besonders schwierig ist, geeignete, die Daten beschreibende Vektoren automatisch zu finden. Aus gerasterten Bilddaten die konstruktiven Elemente wieder heraus zu filtern ist wesentlich aufwendiger als die Bilddaten direkt konstruktiv zu erstellen. Daher kommen auch viele Vektorisierungsprogramme nicht ohne Hilfestellung durch den User aus, das heißt, sie sind halbautomatisch. Man beachte dabei, daß eine Vektorisierung nicht nur über gerasterte Bilddaten erfolgen kann, sondern zum Beispiel auch über akustische Daten. Außerdem hängt die Art der Vektorisierung stark von den gewünschten Information ab, die durch die Vektoren dargestellt werden sollen. Das Bild einer ingenieurtechnische Konstruktionszeichnung wird durch geometrische Objekte beschrieben, während die auf einem Überweisungsformular befindliche Beschriftung sich sehr bekömmlich als Text darstellen läßt, etwa in ASCII. Eine verbreitete Beschreibungssprache, die sich besonders für geometrisch konstruierte Bilddaten eignet, ist Postscript [AS]. Auch bei akustischen Daten ist die gewünschte Information ausschlaggebend, gesprochene Sprache kann vorzugsweise in die internationale Lautschrift [IPA] oder eine ähnliche Beschreibung übergeführt werden, für musikalische Kompositionen hingegen eignet sich eher eine notenorientierte Vektorisierung, etwa MIDI [PH]. In diesem Zusammenhang ist die klassische abendländische Notenschrift als bildliche Darstellung einer notenorientierten Vektorisierung akustischer Daten aufzufassen.

Besonders beachtenswert bei der Vektorisierung ist die erreichbare hohe Informationsverdichtung: Obwohl die als Text dargestellte Information die gleiche ist, produziert ein Telex ein wesentlich geringeres Datenaufkommen als ein Telefax.

11 Einbindung ins Betriebssystem

Bei Einbindung eines Datenkompressors in ein klassisches dateiorientiertes Betriebssystem stellt sich die Frage, auf welcher Ebene der Hierarchie der Datenkompressor angesiedelt werden soll. Diese Frage muß jeweils anhand von Kriterien wie technischer Machbarkeit und Zielsetzung entschieden werden. So ist es in einigen Betriebssystemen nur möglich, einen Kompressor als Anwendung bereitzustellen, weil andere Möglichkeiten nicht vorgesehen sind. Von der Ebene der Anwendungsprogramme abwärts werden noch drei Ebenen unterschieden (Bild 36): Das *Filesystem*, das *BIOS* und das *Medium* selbst.

<table>
<tr><th>Ebene</th><th>Kennung</th><th>Beispiele</th><th colspan="4">Merkmale</th></tr>
<tr><td>Anwendung (Archiver)</td><td>Filename (z.B.: *.z)</td><td>zoo, arc...</td><td rowspan="2">unabhängig von Hardware</td><td rowspan="3">Softwarelösung</td><td>manuell</td><td>Kontext frei</td></tr>
<tr><td>Filesystem</td><td>Kennung im Directory</td><td></td><td rowspan="3">automatisch</td><td>Kontext fileweise</td></tr>
<tr><td>BIOS</td><td>Markiertes Medium oder Medium als File</td><td>Stacker, DoubleSpace...</td><td>Eingriff ins Medium</td><td rowspan="2">Kontext unbekannt</td></tr>
<tr><td>Medium</td><td>implizit</td><td>MNP-5, FAX (3)...</td><td>implizit</td><td>Hardware-lösung</td></tr>
</table>

Bild 36

Eine Einbindung in die Dateiverwaltung (*Filesystem*), so daß der Kompressor automatisch über Funktionen wie Öffnen, Lesen und Schreiben einer Datei aktiviert wird. Für einen solchen Kompressor müßte die Dateistruktur auf dem Medium dahingehend geändert werden, daß zu allen Dateien Kennungen hinzugefügt werden, die Aussagen über die Art der Kompression treffen, die angewendet wurde.

Die nächsttiefere Stufe im Betriebssystem wäre das *BIOS* (*Basic Input Output System*), genauer die blockorientierten Zugriffsfunktionen zu den Massenmedien. Es wird naheliegenderweise immer ein ganzes Medium komplett der Kompression unterzogen, die Blöcke des Mediums werden einzeln und unabhängig voneinander komprimiert. Das zentrale Problem, das sich hierbei stellt, ist die

Tatsache, daß die vormals gleichlangen Blöcke nach der Kompression unterschiedlich lang sind, insbesondere können zum Beispiel bei einer Diskette nicht einfach die komprimierten Blöcke auf denselben Spuren abgelegt werden, auf denen sie ursprünglich lagen. Es muß also ein Verzeichnis angelegt werden, das Angaben über Lage, Länge und angewendete Kompressionsverfahren zu jedem einzelnen Block enthält. Natürlich ist dies ein erhöhter Verwaltungsaufwand im System und es muß abgewägt werden, ob sich dieser durch vermehrte Speicherkapazität bezahlt macht. Einen Überblick über Kompressionsraten und Geschwindigkeit einiger solcher BIOS-orientierter Kompressoren (Stacker 3.0, SuperStor, Doublespace und Xtradrive) liefert Reznik [RE]. Ein gravierendes Problem, das bei BIOS-orientierten Kompressoren zwangsläufig auftritt, ist das Problem des realen Speicherüberlaufs ohne virtuelles Hinzuspeichern. Um dieses Problem zu verstehen, stelle man sich ein reales Medium vor, das n gut komprimierte virtuelle Blöcke enthält. Außerdem soll angenommen werden, daß das reale Medium voll ist (Bild 37). Versucht man nun, einen zusätzlichen Block auf

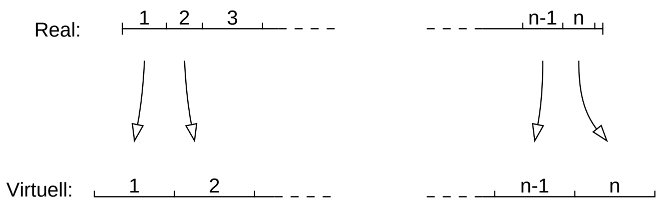

Bild 37

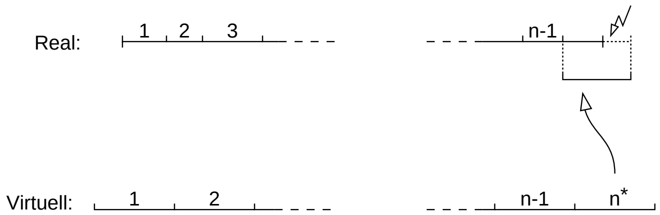

Bild 38

dem Medium abzulegen, so meldet das Medium, zu Recht, einen Fehler, da das Medium voll ist. Wird aber ein virtueller Block gelesen, verändert und wieder geschrieben, so ändert sich zwar die Größe des virtuellen Mediums nicht, im realen Medium jedoch kann mehr Platz benötigt werden, als zuvor belegt war, im ungünstigen Fall paßt der Block, der eben noch auf dem Medium geschrieben stand, auf das selbe Medium nun nicht mehr drauf (Bild 38). Für ein Betriebssystem gibt es in diesem Fall keinen korrekten Ausweg aus der Situation. Alle Lösungsansätze sind unvollkommen, sei es die Beschränkung des Mediums auf einmaliges Schreiben je Block, das Begrenzen der Erstbeschreibbarkeit auf einen niedrigeren Wert, so daß noch eine gewisse Toleranz für nachträglich geänderte Blöcke bliebe, oder ein schlichter Schreibfehler und die Weigerung, den veränderten Block zu speichern. Auch virtuelle Datenblöcke, die vom Filesystem bereits freigegeben wurden, belegen, wie bei normalen Medien auch, weiterhin realen Speicherplatz, obwohl die enthaltenen Daten irrelevant (!) sind. Weiter ist zu beachten, daß eine zusätzliche Stufe in der Zugriffshierarchie auf Massenmedien auch zu zusätzlicher Fehleranfälligkeit führt. Nicht umsonst widmen Computerzeitschriften ganze Artikel dem Thema *Probleme mit dem BIOS-Kompressor* [SC].

Die niedrigste Ebene ist die Kompression direkt im oder am *Medium*. Hierzu gehören unter anderem Kompressoren im Modem [DR], Kompression bei der Übertragung eines Fax [HR] und Kompressoren bei der Bildübertragung vom Satelliten zur Bodenstation [RP], aber auch speicherorientierte Anwendungen wie Sicherung auf Magnetbändern [WE]. Blockzugriffsorientierte Medien eignen sich für implizite Kompression weniger, da, wie schon erwähnt, der zusätzliche Verwaltungsaufwand derart hoch ist, daß ein im Gerät untergebrachtes Verwaltungsprogramm einen erhöhten Hardwareaufwand bedeuten würde, der bei Unterbringung im Hauptrechner nicht anfällt. Da Magnetbänder aber nicht blockzugriffsorientiert sind, eignen sie sich, wie Welch [WE] betont, besonders für implizite Kompression.

12 Effizienz und Geschwindigkeit

Als Faustregel wird angenommen, daß ein Kompressionsalgorithmus umso langsamer ist, je effizienter er komprimiert. Wenn der Durchsatz einer Datenübertragungsstrecke durch Anwendung eines Kompressionsverfahrens verbessert werden soll, benötigen sowohl der Kompressor als auch der Dekompressor eine durchschnittliche Zeit t_k und t_d je übertragenem Bit im komprimierten Kanal. Der Kanal seinerseits kann höchstens $\frac{1}{t_0}$ bit/sec bewältigen. Wenn nun $t_k > t_0$, so bedeutet das, daß der Kompressor nicht in der Lage ist, die Codedaten so schnell zu liefern, daß der Kanal ausgelastet ist. t_k hängt aber auch von der Effizienz des Kompressors ab, denn wenn der Kompressor die Textdaten bei gleicher Eingangsgeschwindigkeit doppelt so gut komprimiert, so verdoppelt sich t_k. Es ist also sinnlos, den Kompressor derart zu optimieren, daß $t_k > t_0$ gilt [WE]. Tritt auf der Empfängerseite der Fall $t_d > t_0$ ein, so bedeutet das, daß die ankommenden Daten entweder zwischengespeichert werden müssen oder einen Rückstau auf den Kanal bilden, der damit wieder nicht voll ausgelastet wäre. Auch eine Effizienzsteigerung mit der Folge $t_d > t_0$ ist also überflüssig.

13 Parallelisierung

Stehen mehrere Prozessoren für die Erledigung einer Kompression zur Verfügung, so stellt sich die Frage, wie die Aufgabe auf die Prozessoren verteilt werden kann. Es gibt mehrere Varianten, die stark von der Struktur des Kompressors abhängen.

Werden die Daten blockweise komprimiert, so kann sich jeder Prozessor je einen Block vornehmen, diesen komprimieren und an die Codeausgabe abgeben (Bild 39). Wenn die Kompression je Block gleichlang dauert, ist dies sehr

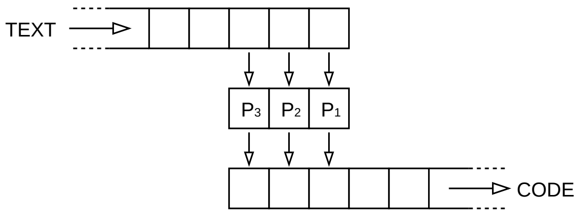

Bild 39

einfach zu realisieren. Gibt es hingegen große Unterschiede in der Bearbeitungsdauer der einzelnen Blöcke, so müssen die Prozessoren möglicherweise aufeinander warten (zum Beispiel P_2 auf P_1). Dieses Problem läßt sich einschränken, wenn genügend Pufferspeicher vorhanden ist, um bereits komprimierte Blöcke vorläufig aufzunehmen.

Für den Fall, daß die Wahl des Kompressionsverfahrens adaptiv geschieht, muß meist für alle anwendbaren Kompressionsverfahren die zu erwartende Länge des Codes berechnet werden. Dies kann natürlich parallel geschehen (Bild 40), wobei

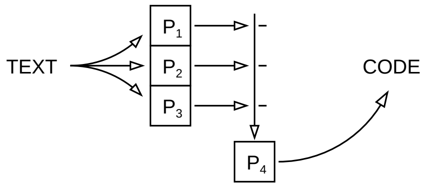

Bild 40

jeder Prozessor die Analyse für je ein Kompressionsverfahren übernimmt. Ein weiterer Prozessor ermittelt dann das Minimum und wendet den entsprechenden Algorithmus an (P_4). Es ist zu beachten, daß diese Art der Parallelisierung nur bei der Kompression anwendbar ist, während bei der Dekompression keine Analyse für die Wahl des Verfahrens notwendig ist.

Läßt sich der Kompressionsalgorithmus selbst in aufeinander folgende Schritte aufteilen, dann kann die Kompression leicht als Pipeline angeordnet werden (Bild 41). Dies ist besonders dann möglich, wenn der Kompressor eine Ver-

$$\text{TEXT} \rightarrow \boxed{P_1} \rightarrow \boxed{P_2} \rightarrow \boxed{P_3} \rightarrow \text{CODE}$$

Bild 41

kettung mehrerer Algorithmen darstellt. Ein Beispiel für eine derartige Pipeline findet sich bei Rice und Plaunt [RP], es werden eine Differenzcodierung und zwei einfache Codewandler verkettet, von denen der zweite in Abhängigkeit des Ergebnisses des ersten arbeitet.

Für den Algorithmus JPEG [BA] bietet sich sowohl eine blockweise Parallelisierung, da ein Bild hier in Blöcken zu acht mal acht Pixeln komprimiert wird, als auch eine Pipelinestruktur je Block an, denn es werden fünf Transformationen verkettet: Transformation von RGB in den Farbraum YUV, Diskrete Cosinus Transformation, Quantisierung, Lauflängen- und Huffmancodierung.

14 Verkehrt asymmetrische Datenkompression

Bei Vorliegen einer Asymmetrie von Kompression und Dekompression stellt sich die Frage, ob zwangsläufig die Kompression aufwendiger sein muß. Künstliche Satelliten werden für gewöhnlich ins All geschossen, um dort irgendwelche Daten, im allgemeinen umfangreiche Bilddatenmengen, aufzunehmen und diese sodann in digitalisierter, diskretisierter Form zur Erde zu senden. Hierfür steht jedoch nur eine geringe Übertragungsgeschwindigkeit zur Verfügung, während die Daten bei der Aufnahme wesentlich schneller anfallen und daher zwischengespeichert werden. Beim Tiefraumvideospeicher R3m beispielsweise beträgt die Datenrate bei der Aufzeichnung 2.08 Mbit/s, bei der Wiedergabe jedoch nur 4096 beziehungsweise 8192 bit/s [WEI,p.17]. Wünschenswert wäre es also, die anfallenden Bilddaten im Satellit stark zu komprimieren, um die Bilddaten in kürzerer Zeit senden zu können.

Während es für die Empfängerseite technisch kein Problem ist, Rechenleistung und Zwischenspeicher bereit zu halten [GR], ist im Satelliten die Möglichkeit zur Unterbringung von Rechenkapazität nur begrenzt vorhanden [HEM]. Für den Fall, daß also kein zufriedenstellendes Bilddatenkompressionsverfahren der symmetrischen Art gefunden werden kann, das sowohl den Codier- als auch den Decodiervorgang zügig bewerkstelligt, wäre es durchaus vorstellbar, die Daten schnell zu komprimieren, um dann in der Bodenstation mit erhöhter Rechenleistung die empfangenen Daten zu dekomprimieren. Völlig absurd wäre es hingegen, im Satelliten langwierig zu komprimieren, womöglich länger als die eigentliche Übertragung Zeit beanspruchen würde: Der Kompressionsvorgang darf im Mittel nicht länger dauern als die zur Übertragung der komprimierten Daten benötigte Zeit, wie im Abschnitt 12 „Effizienz und Geschwindigkeit“ dargelegt wurde.

Gibt es also Datenkompressionsverfahren, die schnell komprimieren, aber nur langsam dekomprimieren? Diese Verfahren sollen *verkehrt asymmetrisch* genannt werden.

Im oben angeführten Fall, daß die Textsynthese der umgekehrte Vorgang der Textanalyse ist, wurde stillschweigend angenommen, daß damit beide Vorgänge gleichen Aufwandes sind. Dem muß aber nicht so sein. Es ist immerhin denkbar, daß die Umkehrfunktion zu einer Textanalysefunktion wesentlich aufwendiger ist als die Textanalysefunktion selbst. Ein Beispiel für eine solches Funktionenpaar sind die Multiplikation und die Division: Während es relativ einfach ist, zwei große Festkommazahlen zu multiplizieren, ist es ungleich schwerer, sie zu dividieren.

Genau aus dieser Asymmetrie soll nun eine Bilddatenkompression konstruiert werden, die ein verkehrt asymmetrisches Verfahren darstellt. Zunächst werde der Einfachheit halber angenommen, daß das zu komprimierende Bild aus fünf mal fünf Punkten besteht und jeder Punkt nur schwarz (0) oder weiß (1) sein kann. Es wird eine Matrix P konstruiert, deren Elemente die ersten 25 Primzahlen sind. Die Matrix A enthalte das Bild:

$$P = \begin{pmatrix} 2 & 3 & 5 & 7 & 11 \\ 13 & 17 & 19 & 23 & 29 \\ 31 & 37 & 41 & 43 & 47 \\ 53 & 59 & 61 & 67 & 71 \\ 73 & 79 & 83 & 89 & 97 \end{pmatrix}, \qquad A = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Aus den Matrizen P und A wird nun gemäß der Formel

$$p = \prod_{i,j:A_{i,j}=1} P_{i,j}$$

eine natürliche Zahl ermittelt. Diese wird in einer für die Theorie nicht weiter interessanten Art und Weise in eine Folge von Bits umgewandelt. Das Verfahren ist im Prinzip eine Anwendung der Gödelcodierung [GÖ]. Für das Beispiel ergibt sich folgender Wert:

$$p = 5 \cdot 43 \cdot 53 = 11395$$

Das Verfahren kann nur als theoretisches Beispiel dienen, da es überhaupt nur zu Kompression führt, wenn das Bild fast schwarz ist, sich also bis zu drei oder vier Punkte darauf befinden.

Worauf es jedoch ankommt, ist, daß der Decodieralgorithmus bei weitem komplizierter ist als die Kompression, denn es muß für eine eventuell große Zahl eine Primfaktorzerlegung durchgeführt werden. Da jeder Primfaktor in P nur einmal vorkommt, ist eine Zuordnung der Faktoren zu Bildpunkten eindeutig möglich. Es handelt sich also tatsächlich um eine verlustfreie Datenkompression.

Fazit ist also, daß es durchaus nicht ausgeschlossen ist, daß es eine Bilddatenkompression gibt, die den eingangs genannten Forderungen entspricht, also als *verkehrt asymmetrisch* zu bezeichnen ist. Eine Perspektive in dieser Richtung könnte die fraktale Codierung sein, die zwar dem Stand der Forschung entsprechend bedingt durch eine aufwendige Analyse als asymmetrisch, nicht jedoch als verkehrt asymmetrisch zu bezeichnen ist. Sie kann jedoch, vorausgesetzt es gelingt, für bestimmte Spezialanwendungen den Analyseteil einzuschränken, auch als verkehrt asymmetrische Variante auftreten. Dies kann der Fall sein, wenn bei speziellen Bildern bereits sehr viel über die vorhandenen Selbstähnlichkeiten bekannt ist. Was jedoch bleibt ist die Decodierung mit Hilfe eines iterierten Funktionensystems [BSN].

Es sei noch auf weitere Anwendungsmöglichkeit für verkehrt asymmetrische Bilddatenkompression verwiesen. Ähnlich wie im Falle eines Satelliten könnte von einer beweglichen Fernsehkamera eine Bildfolge in komprimierter Form zu einer Fernsehanstalt gesendet werden. Auch hier ist auf Empfängerseite wesentlich mehr Rechenkapazität ansiedelbar als auf Senderseite. Alle anderen Fälle der mobilen Datenaufnahme sind ebenfalls als Anwendungsfälle geeignet, besonders solche mit Bilddatenaufnahme.

Im übrigen stellt sich die Frage, ob Kryptographie als verkehrt asymmetrische Datenkompression bezeichnet werden kann. Aufgrund der Tatsache, daß kryptographische Funktionen im allgemeinen keine Kompression beinhalten, dürfte die Bezeichnung nicht ganz angebracht sein. Was bleibt, ist jedoch die verkehrt asymmetrische Eigenschaft bezüglich der Aufwandsbestimmung. Jedoch läßt sich auch hier ein bedeutender Unterschied erkennen: Während ein verkehrt asymmetrisches Kompressionsverfahren nicht verkehrt asymmetrisch sein sollte, um die Decodierung möglichst kompliziert zu machen, sondern um eine möglichst einfache Codierung zu unterstützen, ist ersteres bei der Kryptographie gerade der Fall: ein überhöhter Aufwand ist gewollt.

15 Zusammenfassung

Es wurde ein Modell für verlustfreie Datenkompression vorgestellt. Auf dessen Grundlage ließ sich besonders einfach der Beweis zur Nichtexistenz eines allumfassenden Kompressionsverfahrens führen. Allerdings ist das Modell nicht algorithmisch, sondern funktional. Daher eignet es sich nicht zur Beschreibung der einzelnen Kompressionsverfahren. Hierzu müßte die verlustfreie Datenkompression beispielsweise als Automat definiert werden.

Zur Vorbereitung der Kategorisierung der verlustfreien Datenkompressionsverfahren wurden zunächst die Begriffe *Redundanz* und *Irrelevanz* unterschieden sowie die verlustbehaftete Datenkompression von der verlustfreien getrennt. Um den Unterschied möglichst deutlich zu machen, wurden die für verlustbehaftete Datenkompression von Bildern und akustischen Daten wichtigen Irrelevanzkriterien beim menschlichen Ohr und Auge beschrieben.

Die folgenden Kategorien zur Einteilung verlustfreier Kompressionsverfahren wurden beschrieben:

Redundanz: statistisch oder referenzierend.
Redundanzanpassung: statisch, variabel oder adaptiv.
Symmetrie: Analyse und Synthese symmetrisch oder asymmetrisch.
Codierung: variable oder feste Länge von Textsequenzen und Codeworten.
Quellstruktur: Länge der Daten unbestimmt, bestimmt oder fest.

Es wurden sowohl klassische und weitverbreitete Kompressionsverfahren beschrieben (Huffman, arithmetische Codierung, Codebook-Verfahren) als auch überkommene einfache Verfahren (Shannon-Fano, Lauflängen...) und neue Ideen (Blaschkowski-Codierung, Hilbergs texturale Sprachmaschine). Soweit möglich, wurden Angaben über die Zugehörigkeit zu Kategorien gemacht. Dabei zeigte sich, daß die Kategorisierung nicht vollständig ist — nicht alle Verfahren lassen sich einer Kategorie zuteilen. Außerdem lassen sich viele Verfahren in einigen Kategorien variieren, zum Beispiel können viele Verfahren mit statischer, variabler oder adaptiver Redundanzanpassung verwendet werden.

Zusätzlich zu den verlustfreien wurden verlustbehaftete Verfahren erwähnt, die bei Bild- und Audiokompression verwendet werden.

Drei Kapitel beschäftigen sich mit der Umsetzung von Kompressionsverfahren in Komprimierer (Einbindung ins Betriebssystem, Effizienz und Geschwindigkeit, Parallelisierung).

Ob die verkehrt asymmetrische Datenkompression praktische Bedeutung hat, bleibt offen. Immerhin zeigte sich, daß ein solches Kompressionsverfahren nicht von vornherein sinnlos ist. Vielmehr geben die Einsatzbedingungen an, in welchen Hinsichten ein Kompressor optimiert werden muß.

16 Literatur

[AB] Norman Abramson: Information Theory and Coding, McGraw-Hill, 1963

[ANB] John B. Anderson, John B. Bodie: Tree Encoding of Speech, IEEE Transactions on Information Theory, IT-21(4), 1975, p. 379..387, auch [DG,p.390ff]

[AS] Adobe Systems: Postscript Language Reference Manual, Addison-Wesley 1986, ISBN 0-201-10174-2

[BA] Nick Baran: Putting the Squeeze on Graphics, Byte, Dec 1990, p. 289..294

[BB] D. Balfanz, A. Bergholz: Digitale Codierung von Alphabeten. Nachrichtentechnik Elektronik Berlin, 39 (1989) 6, p. 220..222

[BG] Friedrich L. Bauer, Gerhard Goos: Informatik, Teil 1 und 2, Springer-Verlag Heidelberg 1982/84, ISBN 3-540-11722-9, ISBN 3-540-13121-3

[BL] Daniel Blaschkowski, Persönliches Gespräch, Aug 1994

[BN] Frank Bauernöppel: Imploding...freezing...done, c't 1991, Heft 10, p. 278..286

[BO] Erwin Bolthausen: Skript MAFI IV (Mathematik für Informatiker, Stochastik), Fachbereich Mathematik, Technische Universität Berlin, 1991, p. 131..135

[BS] Ilja N. Bronstein, Konstantin A. Semendjajew: Taschenbuch der Mathematik, Thun, Frankfurt(Main) 1987, ISBN 3-87144-492-8

[BSN] Michael F. Barnsley, Alan D. Sloan: A Better Way to Compress Images, Byte, Jan 1988, p. 215..223

[BSR] Alex Balkanski, Michael Slater: Ein Single macht's möglich. Elektronik 3/1991, p. 109..111

[BT] Nick Birch, Philippe Thomas: Datenkompression löst Übertragungs- und Speicherprobleme. Elektronik 19/1989, p. 44..49

[BU] Jürgen Buck: Der JPEG-Algorithmus, hilfreicher Verlust. mc, Jun 1993, p. 94..101

[BVN] Kai Uwe Barthel, Thomas Voyé, Peter Noll: Improved Fractal Image Coding, Institut für Fernmeldetechnik, Technische Universität Berlin, PCS '93, Section 1.5

[CH] Gordon V. Cormack, R. Nigel Horspool: Algorithms for Adaptive Huffman Codes, Information Processing Letters, Vol. 18, No. 3, 1984, p. 159..165

[CM] Christoph Cavigioli, Gerhard Moosburger: Weniger ist oft mehr. Elektronik 23/1992, p. 138..141, 24/1992, p. 32..43

[DG] Lee D. Davisson, Robert M. Gray: Data Compression, Halsted Press, 1976, ISBN 0-87933-089-9, ISBN 0-470-15053-X

[DR] Frank J. Derfler, Thomas Röder: Wie funktioniert MNP 5 ? PC Professionell, Aug 1991, p. 176..177

[DUDEN] Der Große Duden, Bibliographisches Institut Leipzig, 1990, ISBN 3-323-00030-7

[EL] Peter Elias: Predictive Coding - Part I, IEEE Transactions on Information Theory, IT-1(1), 1955, p. 16..23, auch [DG,p.35ff]

[ELL] Wolfram Ellwanger: Kommunikationspsychologie (Skriptum), Pädagogische Hochschule Karlsruhe, 1985

[FAQ] comp.compression FAQ, March 18th, internet 1994, <compr1_18mar94@chorus.fr>

[FE] Solomon Feferman et al.: Kurt Gödel Collected Works Volume I, Oxford University Press, Clarendon Press, 1986, ISBN 0-19-503964-5

[FI] B. M. Fitingof: The Compression of Discrete Information, Problemy Peredachi Informatsii, Vol. 3, No. 3, p. 28..36, 1967

[FS] Neues Meßverfahren, Minidisc gegen DCC. Funkschau 8/1993, p. 90..93

[GÖ] Kurt Gödel: Über formal unentscheidbare Sätze der Principia mathematica und verwandter Systeme I, Monatshefte für Mathematik und Physik 38, p. 173..198, auch [FE]

[GR] James L. Green: Space Data Management at the NSSDC: Applications for Data Compression, National Space Science Data Center, 1988

[HA] I. Hartmann: Skript Bilddatenverarbeitung, Institut für Regelungstechnik und Systemdynamik, Technische Universität Berlin, 1991

[HE] Gilbert Held: Data Compression, John Wiley & Sons, 1983, ISBN 0-471-26248-X

[HEM] Martin E. Hellmann: Joint Source and Channel Encoding, Proc. Seventh Hawaii Intern. Conf. System Sci., p. 46..48, Western Priodicals Co., 1974, auch [DG,p.69ff]

[HI] Wolfgang Hilberg: Die texturale Sprachmaschine als Gegenpol zum Computer, Sprache und Technik, Groß-Bieberau/Odenwald 1990, ISBN 3-928161-00-8

[HR] Roy Hunter, A. Harry Robinson: Internal Digital Facsimile Coding Standards, Proceedings of the IEEE, 68(7), Jul 1980, p. 854..867

[HU] David A. Huffman: A Method for the Construction of Minimum-Redundancy Codes, Proceedings of the IRE, 40(9), 1952, p. 1098..1101, auch [DG,p.31ff]

[IPA] International Phonetic Association: IPA-Alphabet, in: Suplémant au journal l'Institituteur sténografe, august/september 1888, p. 57..60

[JW] Odorama, 1981, Titel: Polyester, Regie: John Waters, Darsteller: Lady Divine u.a., auch [NR]

[KN] Donald E. Knuth: Dynamic Huffman Coding, J. Algorithms 6 (1985), p. 163..180

[KR] Christian Krüger: Verlustfreie Bildkomprimierung von Schwarzweißbildern unter Berücksichtigung von orthogonal ausgerichteten Rechteckflächen. Seminar Komprimierung von Bilddateien, Prof. Völz 1993, unpublished

[LA] Erhard Lanzerath: Verfahren der Kompression digitaler Daten. PIK 13 (1990) 2, p. 83..90

[LL] J. A. Llewellyn: Data Compression for a Source with Markov Characteristics, The Computer Journal, Vol. 30, No. 2, 1987, p. 149..156

[LR] Glen G. Langdon, Jorma Rissanen: Compression of Black-White Images with Arithmetic Coding, IEEE Transactions on Communications, COM-29(6), Jun 1981, p. 858..867

[ME] W. Meyer-Eppler: Grundlagen und Anwendungen der Informationstheorie, Springer-Verlag Heidelberg, 1969

[MI] Tim Miner: Film ab für MPEG, Computer Persönlich 1/1994, p. 28..34

[NÖ] Winfried Nöth: Handbuch der Semiotik, Metzler, Stuttgart 1985, ISBN 3-476-00580-1

[NR] Jay Robert Nash, Stanley Ralph Ross: The Motion Picture Guide 1927-1983, ISBN 0-933997-00-0

[PH] S. Philipp: MIDI-Kompendium, Verlag Kapehl & Philipp, Wiesbaden 1984, ISBN 3-925020-00-4

[RE] G. A. Reznik: Real-Time Data Compression Algorithms, Index PRO V.3: 18..29, EuroIndex Ltd., Kiev 1994

[RO] Steve G. Romaniuk: Theoretical Results for Applying Neural Networks to Lossless Image Compression, Department of Information Systems & Computer Science, National University of Singapore 1994

[RP] Robert F. Rice, James R. Plaunt: Adaptive Variable-Length Coding for Efficient Compression of Spacecraft Television Data, IEEE Transactions on Communication Technology, COM-19(6), Dec 1971, p. 889..897

[RPE] Michael Rodeh, Vaughan R. Pratt, Shimon Even: Linear Algorithm for Data Compression via String Matching. Journal of the ACM, 28(1), Jan 1981, p. 16..24

[SC] Alfred Schweiger: So werden Sie den DOS-6-Komprimierer wieder los. Computer Persönlich, Nr. 15, 7. Jul 1993, p. 64..67

[SK] Eugene S. Schwartz, Bruce Kallick: Generating a canonical prefix encoding, Communications of the ACM, 7(3), Mar 1964, p. 166..169

[SS] James A. Storer, Thomas G. Szymanski: Data Compression via Textual Substitution, Journal of the ACM, 29(4), Oct 1982, p. 928..951

[ST] James A. Storer: Data Compression, Methods and Theory, Computer Science Press, Rockville 1988, ISBN 0-88175-161-8

[SW] Claude E. Shannon, Warren Weaver: The Mathematical Theory of Communication, University of Illinois Press, 1949, ISBN 0-252-72548-4

[TO] Erwin M. Thuner, Monika Otter: Komprimieren ohne Datenverlust. Elektronik 23/1992, p. 134..137, 25/1992, p. 38..44

[VI] Jeffrey Scott Vitter: Design and Analysis of Dynamic Huffman Codes, Journal of the ACM, 34(4), Oct 1987, p. 825..845

[VÖ83] Horst Völz: Information II, Akademie-Verlag Berlin 1983, ISSN 0232-1351

[VÖ91] Horst Völz: Grundlagen der Information, Akademie-Verlag Berlin 1991

[WE] Terry A. Welch: A Technique for High-Performance Data Compression, Computer, Jun 1984, p. 8..19

[WEI] Hans-Günter Weide: Der Tiefraumvideospeicher R3m für das Phobosprojekt — ein Ergebnis der digitalen Dichtspeicherentwicklung aus der DDR, Dissertation an der Akademie der Wissenschaften der DDR, 1989

[WNC] Ian H. Witten, Radford M. Neal, John G. Cleary: Arithmetic Coding for Data Compression, Communications of the ACM, 30(6), Jun 1987, p. 520..540

[ZI] Jacob Ziv: Variable-to-Fixed Length Codes are Better than Fixed-to-Variable Length Codes for Markov Sources. IEEE Transactions on Information Theory, IT-36(4), Jul 1990, p. 861..863

[ZL77] Jacob Ziv, Abraham Lempel: A Universal Algorithm for Sequential Data Compression, IEEE Transactions on Information Theory, IT-23(3), May 1977, p. 337..343

[ZL78] Jacob Ziv, Abraham Lempel: Compression of Individual Sequences via Variable-Rate Coding, IEEE Transactions on Information Theory, IT-24(5), Sep 1978, p. 530..536