Rheinisch Westfälische Technische Hochschule Aachen
Lehrstuhl für Informatik VI
Prof. Dr.-Ing. Hermann Ney

Proseminar Datenkompression im WS 2001/2002

Arithmetische Kodierung

Eric Bodden

Malte Clasen

Joachim Kneis

18.03.2002

Betreuer: Dr. Ralf Schlüter

Die Autoren sind via Email erreichbar unter:

Eric Bodden
proseminar@bodden.de

Malte Clasen
proseminar@copro.org

Joachim Kneis
proseminar@curan.de

Dieses Dokument, die enthaltenen Quelltexte und Links sind verfügbar unter:
http://www-users.rwth-aachen.de/eric.bodden/ac/

Erstellt mit LaTeX 2ϵ.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Zusammenfassung

Diese Ausarbeitung beschreibt nach einer kurzen Einleitung grundlegende Begriffe und Zusammenhänge der Arithmetischen Kodierung. Diese sind Grundlagen für das 3. Kapitel, in dem wir den Vorgang des Kodierens und Dekodierens in verschiedenen Zahlensystemen vorstellen und gleichzeitig auf dabei auftretende Probleme hinweisen, für die anschließend Lösungsmöglichkeiten aufgezeigt werden. Es folgen eine Effizienzbetrachtung sowie der Beweis der Eindeutigkeit des Verfahrens. Abschließend gehen wir noch kurz auf bekannte Modelle ein, denen sich Arithmetische Kodierer zur Berechnung von Wahrscheinlichkeiten bedienen. Gelegentlich stellen wir Vergleiche zur schon bekannten Huffman Kodierung auf, setzen dabei jedoch nur grundlegende Kenntnisse dieses Verfahrens voraus.

Diese Ausarbeitung stützt sich im Wesentlichen auf [2] und [1]. In Anlehnung an Letzters stellen wir eine selbst entwickelte Implementierung vor, welche im Anhang nachzulesen ist. Anhand dieser Quelltexte werden wir einige Beispiele veranschaulichen. Die mathematischen Definitionen und Beweisführungen sind hingegen stark an [2] und [3] angelehnt. Außerdem werden wir das bekannte Shannon-Theorem [11] benutzen, das die Entropie als Grenze der verlustlosen Komprimierbarkeit festlegt. Die Beispielquelltexte sind in C++ geschrieben und bedienen sich ausschließlich einfacher Syntax, deren Semantik jedem Informatiker bekannt sein sollte.

1 Motivation

Die Arithmetische Kodierung umgeht die Beschränkung, zu kodierende Symbole durch eine ganzzahlige Anzahl von Bits darzustellen, und erreicht dadurch eine hohe Effizienz. Sie erreicht bei jeder Datenquelle nahezu die bestmögliche Kompression, welche durch die Entropie beschränkt wird. Obwohl bei der Kodierung ein Code für den gesamten Datenstrom generiert wird, erfolgt sie zeichenweise sequenziell.

Obwohl vom Prinzip her nicht sehr komplex, ist sie in der hier beschriebenen Form erst seit Ende der 70-er Jahre bekannt. In den 80-er Jahren gelang sie wegen der hohen Effizienz und großen Hardwarenähe zu großer Beliebtheit. Erste Ansätze dieser Kodierung wurden schon 1960 von *Elias* und *Abramson* bekannt, jedoch gab es damals noch keine adäquate Lösung für ein Problem, das uns später noch beschäftigen wird: Die nötige Genauigkeit der Arithmetik musste mit der Länge der Nachricht stetig erhöht werden. 1976 gelang es dann *Pasco* und *Rissanen* zu zeigen, dass bestimmte endlich lange Zahlendarstellungen dem Kriterium der Unterscheidbarkeit noch genügen. Jedoch waren auch ihre Verfahren in Bezug auf die Speichernutzung noch nicht effizient. Erst in den Jahren 1979 und 1980 erschienen dann fast zeitgleich Artikel von *Rubin*, *Guazzo*, *Rissanen* und *Langdon*, die einen Algorithmus beschrieben, wie er auch heute noch zur Arithmetischen Kodierung genutzt wird. Er basiert auf Arithmetik endlicher Genauigkeit und arbeitet zeichenweise nach der FIFO-Methode. *Rissanen* und *Langdon* stellten dabei jedoch bereits sehr hardwarenahe Realisierungen vor.

Wie man sieht, ist die Arithmetische Kodierung noch ein recht junges, jedoch nicht weniger effizientes und ausgereiftes Verfahren, das allen Anforderungen an heutige Kompressionsalgorithmen gerecht wird: Datenquellen können zeichenweise kodiert werden, das Verfahren ist also zur On-The-Fly-Kompression z. B. bei der Datenfernübertragung geeignet. Weiterhin erfolgt diese Kompression in linearer Zeit und mit konstantem Speicherbedarf. Sie begnügt sich dabei mit endlichen Genauigkeiten (in der Praxis oft Integer-Arithmetik). Diese und weitere Eigenschaften erlauben eine einfache Implementation in Hardware. Wie wir später sehen werden, erreicht sie außerdem die bestmögliche Kompressionsrate unter der grundsätzlichen Annahme statistischer Unabhängigkeit der einzelnen Symbole und ist relativ leicht um effizienzoptimierte stochastische Modelle zu erweitern. Der Dekodierer arbeitet dabei auf fast dem gleichen Quelltext wie der Kodierer, so dass außerdem die Implementierung recht leicht vonstatten geht.

Heutzutage gibt es viele Anwendungsbereiche für dieses Kodierungsverfahren. Zu den bekanntesten gehören in Hardware implementierte Systeme wie die Faxprotokolle G3 und G4. Hier z. B. können die später beschriebenen Stärken der Kodierung (kleines Alphabet, ungleichmäßige Wahrscheinlichkeitsverteilung) voll ausgespielt werden.

2 Einführung

Bevor wir die Arithmetische Kodierung im Detail diskutieren, führen wir einige grundlegende Begriffe der Datenkompression ein, die wir später zur näheren Beschreibung des Verfahrens benötigen. Unser Ziel ist, Daten zu komprimieren, die im Computer gespeichert oder durch Leitungen gesendet werden. Für die Speicherung sind verschiedene Darstellungsarten üblich. Beispiele für Daten sind normaler Text, Bilder, Binäre Dateien etc.

Die Darstellung wäre etwa die Repräsentation durch 0 und 1, oder auf einem höheren Level als Buchstaben, Pixel und so weiter. Um nicht weiter auf die verschieden Arten von Daten eingehen zu müssen, führen wir den Begriff des Symbols ein, der eine abstrakte

Beschreibung der Daten ermöglicht.

2.1 Grundlagen

DEFINITION 1 (ALPHABET UND SYMBOL)
Eine endliche, nichtleere Menge A heißt ALPHABET. *Mit $|A|$ bezeichnen wir die* LÄNGE *oder Kardinalität des Alphabets. Die Elemente $\{a_1, \ldots, a_m\}$ eines Alphabets werden* SYMBOLE *genannt.*

Die Anordnung der Symbole in A sei außerdem festgelegt. Heutzutage sind Computer so ausgelegt, dass Daten nur in einer eindeutigen Reihenfolge und mit Hilfe eines endlich großen Alphabets verarbeitet werden. Beispiele hierfür sind etwa die Speicherung auf der Festplatte, die Repräsentation durch das Betriebssystem oder die Übertragung von Daten per Modem. Wir werden also im Weiteren diese Struktur übernehmen und führen dazu den Begriff der SEQUENZ ein.

DEFINITION 2 (SEQUENZ)
Eine Folge $S = (s_1, s_2 \ldots)$ von Symbolen a_i aus einem Alphabet A heißt SEQUENZ. *Abkürzend schreiben wir im Folgenden $S = s_1 s_2 \ldots$*

Analog zum Alphabet beschreibt $|S|$ die Länge der Sequenz, sofern sie endliche Länge besitzt. Für die folgenden Aussagen über Wahrscheinlichkeiten nehmen wir dieses an, da solche Aussagen sonst keinen Sinn ergeben. Dass diese Darstellung nicht vollkommen willkürlich ist, wird auch daran klar, dass fast jede Datenquelle wie Bücher, Sprache und so weiter als Sequenz dargestellt werden kann.

Wenn wir nun eine gegebene Sequenz betrachten, dann können wir oft feststellen, dass einzelne Symbole mit einer gewissen Häufigkeit auftreten, die teilweise sehr stark variieren kann. Beispielsweise ist das e in deutschen Texten wesentlich häufiger als das y . Da die Arithmetische Kodierung unter anderem auf diesen verschiedenen Häufigkeiten beruht, definieren wir die WAHRSCHEINLICHKEIT eines Symbols wie folgt :

DEFINITION 3 (WAHRSCHEINLICHKEIT)
Sei $S = (s_1, \ldots, s_n)$ eine endliche Sequenz der Länge $|S| = n$ mit Symbolen aus $A = \{a_1, \ldots, a_m\}$ und $|S|_{a_i}$ die Häufigkeit des Symbols a_i in S. Dann heißt $P(a_i) := \frac{|S|_{a_i}}{n}$ die WAHRSCHEINLICHKEIT *des Symbols a_i (in S).*

Aus dieser Konstruktion wird klar, dass die Wahrscheinlichkeit $P(a_i)$ eines Elements im Intervall $[0, 1)$ liegt, während die Summe über die Wahrscheinlichkeiten $\sum_{i=1}^{n} P(a_i) = 1$ ist. Man beachte, dass wir nur das halboffene Intervall betrachten, denn offensichtlich müssen konstante Sequenzen mit einem Symbol der Wahrscheinlichkeit 1 nicht kodiert werden, da ihr Inhalt schon vollständig bekannt ist. Wenn wir die Arithmetische Kodierung einführen, wird dies die Arbeit erleichtern. Um auf das Beispiel von oben (e, y) zurückzukommen, wollen wir jedoch betonen, dass die Wahrscheinlichkeit eines Symbols stark vom Kontext abhängt. So ist die binäre Darstellung von e und y in ausführbaren Dateien etwa gleich. Weiterhin unterscheiden sich viele wissenschaftliche Texte, z. B. bezogen auf Sonderzeichen, stark von nichtwissenschaftlichen. Außerdem kann eine Sequenz auch auf unterschiedlich Arten interpretiert werden. Betrachten wir etwa die Zahlenfolge 1111131311. Sie kann als Folge von $1, 3$ oder als Folge von $11, 13$ interpretiert werden. Wir sehen also, daß es offenbar nötig ist, eine Abbildungsvorschrift zu definieren, die besagt, auf welche Art und Weise Wahrscheinlichkeiten einzelnen Symbolen oder Symbolfolgen

a	0,0651	h	0,0476	o	0,0251	v	0,0067
b	0,0189	i	0,0755	p	0,0079	w	0,0189
c	0,0306	j	0,0027	q	0,0002	x	0,0003
d	0,0508	k	0,0121	r	0,0700	y	0,0004
e	0,1740	l	0,0344	s	0,0727	z	0,0113
f	0,0166	m	0,0253	t	0,0615		
g	0,0301	n	0,0978	u	0,0435		

Tabelle 1: Häufigkeitsverteilung der Buchstaben in deutschen Texten (aus [10]).

(die auch wiederum als Symbole aufgefasst werden können) zugeordnet werden sollen. Um dies zu tun, werden wir uns des Konzepts des Modells bedienen.

Definition 4 (Modell)
Sei A ein Alphabet. Ein Modell *M ist eine Abbildung*

$$\phi : A \longrightarrow [0,1) : a_i \longmapsto P_M(a_i) \ ,$$

die jedem Element $a_i \in A$ eine (ggf. geschätzte) Wahrscheinlichkeit $P_M(a_i)$ zuordnet.

Diese ist errechnet und muss nicht zwangsläufig der korrekten Wahrscheinlichkeit $P(a_i)$ entsprechen. Dazu wird später mehr ausgeführt. Beim Modell ist zu beachten, dass ein Alphabet nicht zwingend aus Symbolen der Länge 1 bestehen muß, sondern auch längere Symbole (wie im obigen Beispiel 11 und 13) zulässt. Die einzelnen Symbole können auch aus Wörten oder beliebigen Zeichenketten bestehen. Betrachtet man bei der Bestimmung der Wahrscheinlichkeit des aktuellen Symbols auch die n vorherigen, so spricht man von einem *Order-n-Modell*. Beispielsweise ist die Wahrscheinlichkeit für ein u im Deutschen zwar 0.0435, war das vorherige Symbol hingegen ein q so erhöht sich dieser Wert auf nahezu 1. Wenn eine gegebene Sequenz unter einem Modell M interpretiert wird, dann wird die Häufigkeitsverteilung der Sequenz oft von der des Modells abweichen; z. B. erfüllt kaum ein deutscher Text die Verteilung aus Tabelle 1 genau, sondern nur annäherungsweise oder schlechter. Um die durch das Modell gegebene Häufigkeitsverteilung von der realen zu unterscheiden, bezeichnen wir erstere daher mit $P_M(a_i)$, um die Abhängigkeit vom Modell M zu betonen.

Ein Modell bildet also eine Interpretation zu einer beliebigen Datenmenge. Ein einfaches Modell könnte zum Beispiel auf der in Tabelle 1 angegebenen Häufigkeitsverteilung basieren. Sie zeigt die Häufigkeitsverteilung der einzelnen Buchstaben in deutschsprachigen Texten. Wir werden später Modelle im Detail behandeln; hier soll die Bemerkung reichen, dass die Wahl eines guten Modells entscheidend für die Kompression ist. Je genauer ein Modell der Häufigkeitsverteilung der Sequenz gleicht, desto „besser“ wird die Kompression wirken.

Hier stellt sich das Problem, dass wir ein Maß für die Kompression definieren müssen, um Vergleiche über die Effizienz treffen zu können. Dazu führen wir den Begriff der Entropie ein, der den Informationsgehalt einer Sequenz beschreibt.

Definition 5 (Entropie)
Sei S eine Sequenz über dem Eingabealphabet A. Die Entropie *$H_M(S)$ der Sequenz S*

unter dem Modell M definieren wir als

$$H_M(S) = \sum_{a \in A} P(a_i)\ ld\ \frac{1}{P_M(a_i)}\ . \tag{1}$$

Die Entropie wird in der Einheit [Bits/Symbol] gemessen, da in der Formel nur die relative Häufigkeit benutzt wird und nicht die absolute. Um die Entropie einer Sequenz zu bestimmen, wird die Entropie daher noch mit der Länge der Sequenz multipliziert.

Aus der Formel wird ersichtlich, dass nach unserer Definition die Entropie einer Sequenz abhängig vom Modell M ist, da die Wahrscheinlichkeiten $P_M(a_i)$ von diesem bestimmt werden. Dabei bezeichnet $ld\frac{1}{P_M(a_i)}$ die minimale Länge eines binären Codes für das Symbol a_i und der Vorfaktor $P(a_i)$, welcher ja der tatsächlichen Wahrscheinlichkeit entspricht, zeigt an, wie häufig wir dieses Symbol in der gegebenen Sequenz so binär kodieren müssen.[1] Hätte man ein ideales Modell in dem Sinne, daß es auf die korekkte Verteilung abbildet (viele andere Artikel zu diesem Thema gehen davon aus), so könnte man von folgender abgewandelten Definition ausgehen:

$$H(S) = \sum_{a \in A} P(a_i)\ ld\ \frac{1}{P(a_i)} \tag{2}$$

2.2 Beispiel: Entropie

Betrachten wir einmal die Sequenz $S = abaabcda$ über dem Alphabet $\{a, b, c, d\}$. Wir wollen diese binär kodieren. An sich sagt uns diese Sequenz noch nicht viel, deshalb nehmen wir uns ein im obigen Sinne ideales Modell M_1, das uns die korrekten Wahrscheinlichkeiten $P_{M_1}(a) = 0,5$, $P_{M_1}(b) = 0,25$, $P_{M_1}(c) = 0,125$ und schließlich $P_{M_1}(d) = 0,125$ liefere. Sicherlich sieht man, dass diese Wahrscheinlichkeiten genau der Wahrscheinlichkeit der Symbole in der Sequenz S entsprechen:

$$P_{M_1}(s) = P(s)\ \forall s \in A := \{a, b, c, d\}\ .$$

Wenn wir diese Sequenz speichern, können wir dies naiv durch eine Darstellung mit 2 Zeichen pro Symbol, also $\{00, 01, 10, 11\}$, und benötigen dann $8 * 2bits = 16bits$ Wie sieht nun die Entropie $H_{M_1}(S)$ aus ?

$$\begin{aligned}
H_{M_1} &= \sum_{s \in \{a,b,c,d\}} P(s)\ ld\ \frac{1}{P_{M_1}(s)} \\
&= (0,5 \cdot\ ld\ \frac{1}{0,5}) + (0,25 \cdot\ ld\ \frac{1}{0,25}) \\
&\quad +(0,125 \cdot\ ld\ \frac{1}{0,125}) + (0,125 \cdot\ ld\ \frac{1}{0,125}) \\
&= 0,5 \cdot\ ld\ 2 + 0,25 \cdot\ ld\ 4 + 0,125 \cdot\ ld\ 8 + 0,125 \cdot\ ld\ 8 \\
&= 0,5 + 0,5 + 0,375 + 0,375 \\
&= 1,75\ [Bits/Symbol]
\end{aligned}$$

[1] Kodiert man nicht binär, sondern über einem Ausgabe-Alphabet A mit $|A| = a$, dann ersetze man einfach ld durch log_a.

Dies bedeutet also, wir könnten die obige Sequenz mit $1,75$ Bits pro Symbol, also insgesamt 14 Bits binär kodieren. Mehr lässt diese Sequenz nicht zu.

Was passiert aber, wenn wir ein Modell wählen, das die Wahrscheinlichkeit der Sequenz nicht so genau widerspiegelt ? Betrachten wir dazu ein zweites Modell M_2 mit $P_{M_2}(a) = 0,125$, $P_{M_2}(b) = 0,125$, $P_{M_2}(c) = 0,5$ und $P_{M_2}(d) = 0,25$. Die Entropie berechnet sich dann folgendermaßen :

$$\begin{aligned} H_{M_2} &= \sum_{s\in\{a,b,c,d\}} P(s)\ ld\ \frac{1}{P_{M_2}(s)} \\ &= (0,5\cdot\ ld\ \frac{1}{0,125}) + (0,25\cdot\ ld\ \frac{1}{0,125}) \\ &\quad +(0,125\cdot\ ld\ \frac{1}{0,5}) + (0,125\cdot\ ld\ \frac{1}{0,25}) \\ &= 0,5\cdot\ ld\ 8 + 0,25\cdot\ ld\ 8 + 0,125\cdot\ ld\ 2 + 0,125\cdot\ ld\ 4 \\ &= 1,5 + 0,75 + 0,125 + 0,25 \\ &= 2,625\ [Bits/Symbol] \end{aligned}$$

Hier sieht man also, dass wir das Wort *Kompression* mit Vorsicht verwenden müssen, weil die Kompressionsfähigkeit der Kodierung nur so gut sein kann wie das der Kompression zugrundegelegte Modell. In diesem speziellen Fall würde der Bitstring aufgrund der abweichenden Berechnung der Wahrscheinlichkeiten sogar länger sein, nämlich $2,625 * 8 = 21$ Bits. Gleichzeitig ist von großer Bedeutung, dass die Entropie H einer Sequenz eine untere Schranke der verlustlosen Kompression bildet, das heisst, man benötigt mindestens H Bits um die Sequenz zu speichern. (Man beachte, dass hier die Entropie unter Verwendung eines optimalen Modells gemeint ist, vgl. Formel (2).) Der Beweis hierzu wurde schon 1948 von *Shannon* erbracht. Wir möchten dies hier nur erwähnen und verweisen den interessierten Leser auf [11].

Nachdem wir nun all diese wichtigen Grundbegriffe eingeführt haben, fehlen uns nur noch Metohden, um die eigentliche Datenkompression zu beschreiben. Zuerst benötigen wir dazu naheliegenderweise zwei Algorithmen, einen um die Daten zu kodieren, den anderen, um die Daten aus dem Code wieder herzustellen.

2.3 Kodierer und Dekodierer

Definition 6 (Kodierer & Dekodierer)
Ein Algorithmus, der eine Sequenz kodiert, heißt Kodierer *oder* Encoder. *Den dazu passenden Algorithmus, der die Sequenz wieder herstellt, bezeichen wir als* Dekodierer *bzw.* Decoder.

Im Unterschied zur originalen Eingabesequenz S bezeichnen wir den Output des Kodierers als $Code(S)$. Die Anwendung der Algorithmen nennen wir kodieren bzw. dekodieren.

Wir möchten hier den Begriff des Algorithmus in seiner intuitiven Bedeutung benutzen, also als Verfahren der Datenverarbeitung am Computer. Wichtig ist, dass wir keine konkrete Realisierung betrachten, sondern nur das Verfahren unabhängig vom Rechnertyp oder Betriebssystem. Eine Implementierung eines Arithmetischen Kodierers in C++ werden wir am Ende dieses Artikels behandeln. Die folgenden Beispiel-Algorithmen wurden größtenteils dieser Implementierung entnommen.

In der Theorie der Datenkompression wird zwischen verlustfreier und verlustbehafter Kompression unterschieden; inbesondere analoge Signale werden oft nur mit Verlusten kodiert, weil diese Verluste von der menschlichen Wahrnehmung nicht entdeckt werden können. Auf verlustbehaftete Kodierung werden wir nicht weiter eingehen und somit nur verlustfreie Kodierung betrachten, die für alle Arten von digitalen Daten funktioniert und von fundamentaler Bedeutung ist. Weiterhin benutzen wir nur Kodierer, welche die Daten wieder herstellen können, also einen verlustlosen und optimalen Code erzeugen.

2.4 Begriffe zur Eindeutigkeit und Effizienz

Definition 7 (Eindeutige Dekodierbarkeit)
Wir nennen einen Code eindeutig dekodierbar*, wenn jede Sequenz injektiv auf den entsprechenden Code abgebildet wird. In diesem Fall kann man dem Code eine eindeutige Eingabesequenz zuordnen.*

Eine Besonderheit unter den eindeutig dekodierbaren Codes stellen sogenannte präfixfreie Codes dar. Diese zeichen sich dadurch aus, dass kein einzeln kodiertes Symbol den Anfang eines anderen kodierten Symbols bildet.

Definition 8 (Präfixfreier Code)
Wir bezeichen einen Code als präfixfrei*, falls für kein Paar von Symbolen x, y des Eingabealphabets der Code $C(x)$ Präfix von $C(y)$ ist.*

Präfixfreie Codes haben den Vorteil, dass man, sobald man einen Codeteil $C(x)$ gelesen hat, sofort weiß, dass das Symbol x kodiert wurde, und dass sich nicht etwa nach dem Lesen noch weiterer Zeichen herausstellt, dass $C(x)$ nicht x kodiert, sondern nur ein Präfix eines anderen $C(y)$ war. Dies führt dazu, dass präfixfreie Codes zur Klasse der eindeutige dekodierbaren Codes gehören. Einen konstruktiven Beweis hierzu liefert [2] S.31.

Ausgestattet mit diesem Handwerkszeug, wollen wir uns der Arithmetischen Kodierung zuwenden. im folgenden Kapitel wird dabei das allgemeine Verfahren eingeführt und an einigen Beispielen erläutert. In den weiteren Kapiteln wird das Verfahren verfeinert, bestimmte Probleme werden näher erläutert und die Realisierung am Computer wird besprochen.

3 Kodierung als reelle Zahl

Bevor die Arithmetische Kodierung in den 70ern entwickelt wurde, galt die Huffman Kodierung als fast optimales Verfahren, da ihr Code der Entropie einer Sequenz sehr nahe kommt, sie in Spezialfällen sogar erreicht. Nachteile sind die Konstruktion des Code-Baumes und die Festlegung, nur Symbole oder Symbolgruppen zu kodieren. Denn bei der Huffman-Kodierung wird der Binärcode durch das Aufstellen eines den Wahrscheinlichkeiten entsprechend balancierten Binärbaumes generiert. Ein Codewort ergibt sich hierbei durch einen Durchlauf von der Wurzel bis zu dem mit dem entsprechenden Symbol beschrifteten Knoten. Da diese Pfadlänge stets ganzzahlig ist, folgt nur ganzzahlige Bitanzahlen vergeben werden können. Daher wird stets Symbolweise kodiert. Es läßt sich eindeutig sagen, welche Codebits zu welchem Symbol gehören. Dies ist jedoch eine unnötige Einschränkung, die die Arithmetische Kodierung umgeht, wie später zu sehen sein wird. Im Gegensatz dazu verwendet die Arithmetische Kodierung keinen Baum, sondern nur eine eindimensionale Tabelle von Wahrscheinlichkeiten und kodiert stets die *komplette*

Nachricht. Dies liefert zum einen den Vorteil, dass man bestimmten Symbolen auch Codes zuweisen kann, die Bruchstücke von Bits lang sind. Andererseits bedeutet dies jedoch, dass ein Teil des Codes nur dekodiert werden kann, wenn der ganze vorhergehende Code bekannt ist, während in der Huffman Kodierung mit verschiedenen Startpunkten gearbeitet werden kann.

Wie oben schon erwähnt, wird in der Arithmetischen Kodierung prinzipiell die ganze Nachricht kodiert. Es ist natürlich eine Aufteilung in mehrere Teilnachrichten möglich, jedoch stellt dies nur eine Variante desselben Prinzips dar. Wir suchen daher eine sinnvolle Möglichkeit, eine Nachricht zu kodieren, ohne jedem einzelnen Symbol einen Code zuweisen zu müssen. Betrachten wir nun die o. g. Wahrscheinlichkeiten von Symbolen in einer Sequenz, so stellen wir fest, dass ihre Summe über die gesamte Sequenz immer 1 ergibt, während die einzelnen Wahrscheinlichkeitswerte im Intervall $[0, 1)$ liegen. Weiterhin wissen wir, dass in diesem Intervall unendlich viele Zahlen liegen. Daher bietet sich die Möglichkeit an, auch die komplette Sequenz so zu kodieren, dass der Code aus einer (zunächst reellen) Zahl im Intervall $[0, 1)$ besteht. Dies kann man erreichen, indem man Teilintervalle von $[0, 1)$ betrachtet, denen entsprechende zu kodierende Symbole zugeordnet werden. Durch sukkzessives Aufteilen dieser Teilintervalle erhält man dann nach entsprechend vielen Schritten eine Zahl aus $[0, 1)$, die als Code benutzt werden kann.

Betrachten wir nun ein Modell M, dass uns zu m verschiedenen Symbolen die Wahrscheinlichkeiten $P_M(a_1) \dots P_M(a_m)$ der vorkommenden Symbole in der Sequenz berechnet. Da die Summe der Wahrscheinlichkeiten stets 1 ergibt, können wir das Intervall $[0, 1)$ nun so in Teilintervalle einteilen, dass die Größe des i-ten Teilintervalls genau der Wahrscheinlichkeit des Symbols a_i entspricht.

3.1 Beispiel Intervallbildung

Nehmen wir zum Beispiel ein Modell M über dem Alphabet $A = a, b, c, d$. Die Wahrscheinlichkeiten der Werte in der Eingabesequenz seien

$$P_M(a) = 0,5, P_M(b) = 0,25, P_M(c) = 0,125, P_M(d) = 0,125.$$

Das Intervall $[0, 1)$ wäre dann folgendermaßen aufgeteilt:

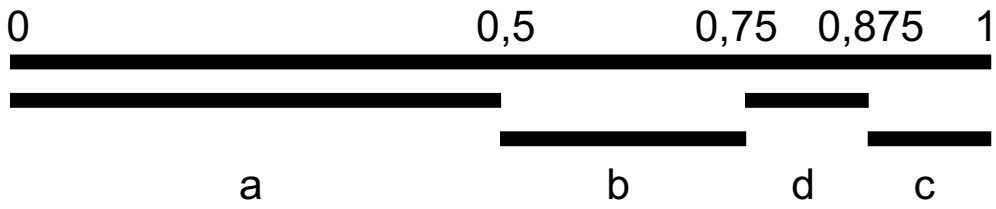

Abbildung 1: Intervallbildung unter einem gegebenen Modell

3.2 Die Intervallgrenzen

Die obere bzw. untere Grenze des *gesamten* betrachteten Intervalls nennen wir ab jetzt *high* (obere Grenze) bzw. *low* (untere Grenze). Weiterhin sind die Grenzen der einzelnen

Teilintervalle von Bedeutung. Da diese aus den kumulierten Wahrscheinlichkeiten gebildet werden, beschreiben wir sie im Folgenden mit

$$K(a_k) = \sum_{i=1}^{k} P_M(a_i) \ .$$

Die Werte für *low* und *high* werden sich im Laufe der Kompression ändern, müssen also ständig neu berechnet werden. Im Gegensatz dazu bleiben die Werte für $K(a_k)$ konstant und werden benutzt, um *high* und *low* zu bestimmen. In Bezug auf das Beispiel erhalten wir also folgende Werte:

high	1,0	$K(0)$	0,0	$K(2)$	0,75
low	0,0	$K(1)$	0,5	$K(3)$	0,875

Wir werden später sehen, dass die Aufteilung des Intervalls eigentlich durch das Modell gegeben und somit durchaus sehr variabel ist. Wir gehen aber zunächst davon aus, dass die Einteilung in einer festen Tabelle $K(a_i)$ $(i = 1, \ldots, m)$ ähnlich dem Beispiel gespeichert ist. Dies entspricht einem statischen Modell.

3.3 Kodierung

Betrachten wir nun die Funktionsweise der Arithmetischen Kodierung. Bevor wir das erste Symbol komprimieren, ist das betrachtete Intervall $I := [low, high)$ mit $low = 0$ und $high = 1$. Nun lesen wir das erste Symbol s_1 und schränken I auf ein Teilintervall I' ein. Die Grenzen von I' bezeichnen wir wieder als *low* bzw. *high*. Dabei wählen wir I' so, dass dieses Intervall genau den Grenzen des Bereichs entspricht, in dem s_1 im Modell liegt. Wie berechnen wir nun die Grenzen dieses Intervalls? Wenn $s_1 = a_k$ das k-te Symbol des Alphabetes ist, dann ist die untere Grenze

$$low := \sum_{i=1}^{k-1} P_M(a_i) = K(a_{k-1})$$

und die obere Grenze analog

$$high := \sum_{i=1}^{k} P_M(a_i) = K(a_k)$$

so dass I' nun das Intervall $[low, high)$ ist. Diese Berechnung ist allerdings noch nichts Neues, da sie nur das mathematische Verfahren zur Konstruktion der Abbildung 1 bildet. Hierbei ist von großer Bedeutung für die Kompression, dass unser Teilintervall I' umso größer wird, je größer die Wahrscheinlichkeit $P_M(s_1)$ des ersten Symbols ist. Denn in diesem Fall reicht eine geringere Anzahl von Nachkommastellen, um eine Zahl eindeutig als aus diesem Intervall stammend zu identifizieren. Anscheinend reicht es aus, nur mit den Werten *low* und *high* weiter zu rechnen, da sie das Intervall I eindeutig festlegen und alle weiteren Zahlen innerhalb dieses Intervalls liegen werden, wie wir im Folgenden sehen. Gehen wir nun einen Schritt weiter und komprimieren das zweite Symbol $s_2 = a_j$. Wieder wollen wir *low* und *high* vom Modell so berechnen, dass die Grenzen zur

Häufigkeitsverteilung passen. Hier stellt sich nun das Problem, dass unser Modell M eine Partition[2] des vollen Intervalls $I = [0, 1)$ liefert, während nun jedoch das Intervall $I' = [low, high)$ betrachtet wird, welches ein echtes Teilintervall von $[0, 1)$ bildet. Wir müssen die Grenzen also noch mit einem Ausgleichsfaktor verrechnen und den Anfang unseres Intervalls berücksichtigen. Ersteres ereichen wir durch die Multiplikation mit $high - low$, also der Länge des Intervalls, und Letzteres durch die Addition der unteren Grenze, also low. Wir erhalten dann die Formel

$$low' := low + \sum_{i=1}^{j-1} P_M(a_i) \cdot (high - low) = low + K(a_{j-1}) \cdot (high - low) \; ; \quad (3)$$

$$high' := low + \sum_{i=1}^{j} P_M(a_i) \cdot (high - low) = low + K(a_j) \cdot (high - low) \; . \quad (4)$$

Diese gilt allgemein für alle Schritte, so speziell auch für den ersten, in dem $low = 0$ sowie $high - low = 1$ ist. Da wir nun die Grenzen unseres alten Intervalls vernachlässigen können, setzen wir einfach:

$$\begin{aligned} low &:= low' \; ; \\ high &:= high' \; . \end{aligned}$$

Die Formeln (3) und (4) wirken auf den ersten Blick recht kompliziert, wir werden ihre Anwendung jedoch gleich an einem Beispiel zeigen. Dieses kann mit Hilfe von Abbildung 3 auf Seite 17 grafisch nachvollzogen werden. Betrachten wir wieder die Sequenz $S = abaabcda$ aus 2.2 unter Annahme unseres idealen Modells M_1.

Wir beginnen mit dem gesamten Intervall [0,1) und betrachten das erste Element von S. Da es sich um ein **a** handelt, berechnen wir die Grenzen wie folgt:

$$\begin{aligned} low &= 0 \\ high &= 0 + 0,5 \cdot 1 = 0,5 \; . \end{aligned}$$

Wir erhalten also das Intervall auf $[0 \ldots 0,5)$. Im zweiten Schritt kodieren wir ein **b** und berechnen daher

$$\begin{aligned} low &= 0 + 0,5 \cdot (0,5 - 0) = 0,25 \\ high &= 0 + 0,5 \cdot (0,5 - 0) + 0,25 \cdot (0,5 - 0) = 0,375 \; . \end{aligned}$$

Nun ist wieder ein **a** an der Reihe, also erhalten wir:

$$\begin{aligned} low &= 0,25 \\ high &= 0,25 + 0,5 \cdot (0,375 - 0,25) = 0,3125 \; . \end{aligned}$$

Als viertes folgt wieder ein **a**

$$\begin{aligned} low &= 0,25 \\ high &= 0,25 + 0,5 \cdot (0,3125 - 0,25) = 0,28125 \; , \end{aligned}$$

[2] Eine *Partition* ist eine disjunkte Vereinigung von Mengen, *Klassen* genannt. Alle Klassen haben leeren Schnitt und die Vereinigung der Klassen ergibt die Ausgangsmenge.

dann ein **b**

$$\begin{aligned} low &= 0,25 + 0,5 \cdot (0,28125 - 0,25) = 0,265625 \\ high &= 0,25 + 0,5 \cdot (0,28125 - 0,25) + 0,25 \cdot (0,28125 - 0,25) = 0,2734375 \ . \end{aligned}$$

Als nächstes lesen wir ein **c**

$$\begin{aligned} low &= 0,265625 + 0,5 \cdot (0,2734375 - 0,265625) + 0,25 \cdot (0,2734375 - 0,265625) \\ &= 0,271484375 \\ high &= 0,265625 + 0,5 \cdot (0,2734375 - 0,265625) + 0,25 \cdot (0,2734375 - 0,265625) \\ &\quad +0,125 \cdot 0,25 \cdot (0,2734375 - 0,265625) \\ &= 0,2724609375 \ , \end{aligned}$$

danach ein **d**

$$\begin{aligned} low &= 0,271484375 + (0,5 + 0,25 + 0,125) \cdot (0,2724609375 - 0,271484375) \\ &= 0,2723388672 \\ high &= 0,2724609375 \ , \end{aligned}$$

und schließlich ein **a**

$$\begin{aligned} low &= 0,2723388672 \\ high &= 0,2723388672 + 0,5 \cdot (0,2724609375 - 0,2723388672) \\ &= 0,2723999024 \ . \end{aligned}$$

Wir erhalten also das Intervall $[0,2723388672; 0,2723999024)$ als Ergebnis unseres Verfahrens.

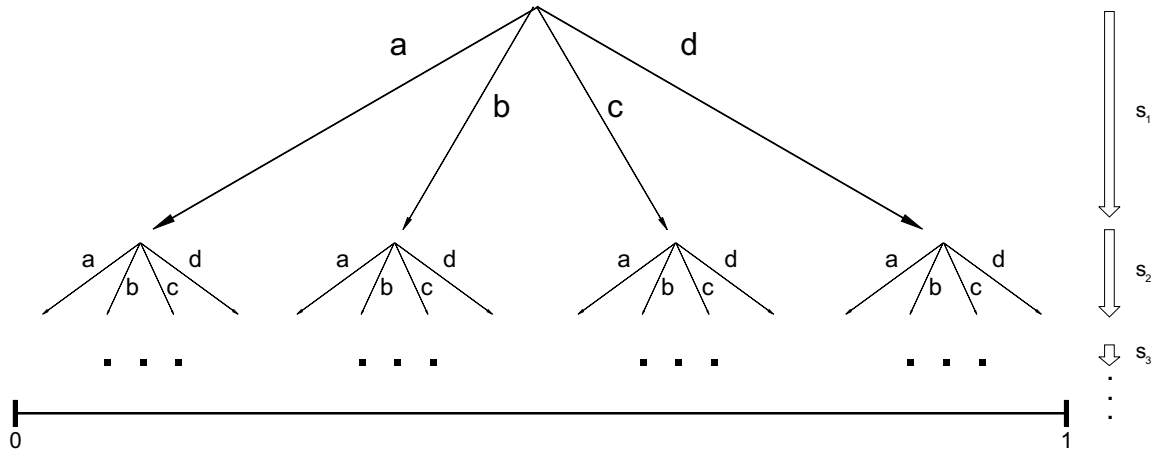

Abbildung 2: Fortschreitende Partitionierung des Intervalls [0,1) (bei Gleichverteilung)

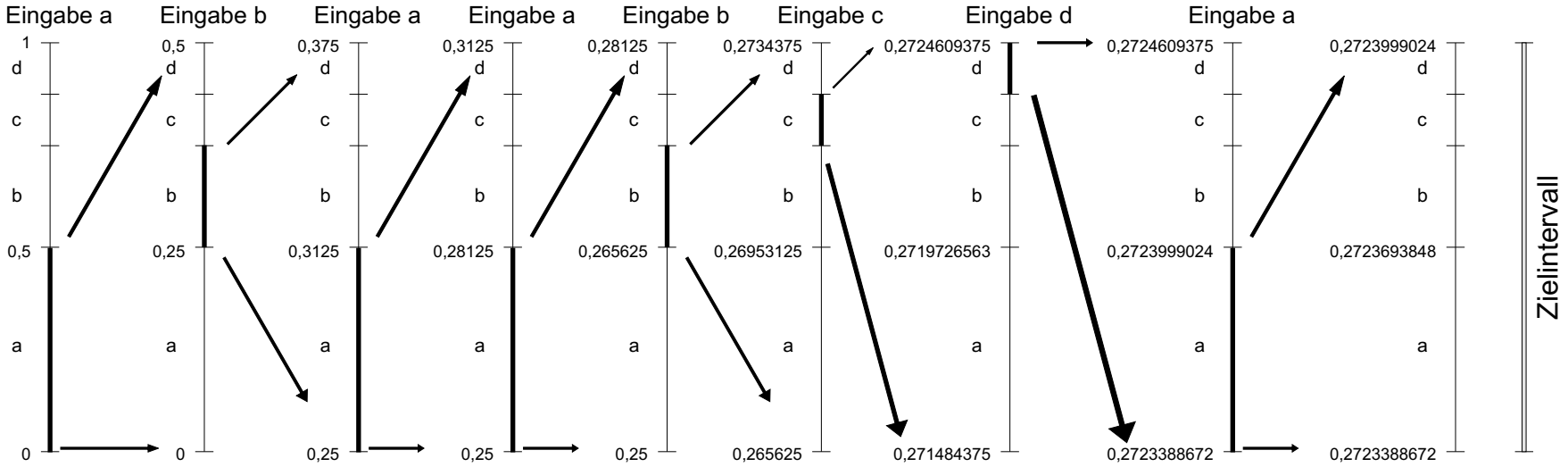

Abbildung 3: Arbeitsweise des Kodierers

Die Frage ist nun, wie unser Code aussieht. Es gilt, das gefundene Intervall eindeutig festzulegen. Ein naiver Ansatz wäre es, die kompletten Grenzen des letzten Intervalls zu speichern, jedoch ist dies relativ aufwändig. Wenn wir unser Verfahren jedoch noch einmal rekapitulieren, dann stellen wir fest, dass es reicht, irgendeinen Wert aus dem Intervall zu speichern, wenn wir wissen, dass unsere Sequenz zu Ende ist. (Dieses Problem werden wir später erläutern.) Dies wird klar, wenn man sich bewust macht, dass jede Sequenz, die wir komprimieren, zu einem anderen Intervall führt. Dies hält folgendes Lemma fest.

LEMMA 1 *Die Codes aller Sequenzen der Länge m bilden eine Partition des Intervalls $I := [0, 1)$.*

Dies folgt anschaulich aus der Abbildung 2 auf Seite 17. Aus dem Lemma folgt sofort, dass für unendlich lange Sequenzen die Klassen der Partition unendlich klein werden. Dies stellt ein Problem für die endliche Arithmetik heutiger Rechner dar, welches wir mit Hilfe einer Umskalierung umgehen wollen. Diesem Thema widmen wir uns in Kapitel 5.

In unserem Beispiel könnten wir also z. B. 0,27234 speichern oder auch jeden anderen Wert im Intervall, um dieses zu kodieren. Hierbei ist zu beachten, dass wir vorausgesetzt haben, dass wir das Ende der Sequenz kennen, was in der Realität selten der Fall sein wird. (Man denke etwa an Kodierungen in der Datenfernübertragung.) Wir werden dies jedoch erst später näher betrachten. Bevor wir nun die Dekompression erläutern, werden wir noch kurz den Kodieralgorithmus zusammenfassen:

```
low =0;
high=1;
do {
    temp= lies_zeichen();
    ival = modell->gib_Intervall(temp); \\liefert das Intervall, in
                                        \\dem temp liegt
    low=  berechne_untere_grenze(ival); \\Anwenden der Formel (3)
    high= berechne_obere_grenze(ival);  \\Anwenden der Formel (4)
} while (!ende_der_sequenz());
return(wert_im_intervall(low,high));
```

3.4 Dekodierung

Um eine mit der Arithmetischen Kodierung kodierte Sequenz wieder zu dekodieren, müssen wir nun das Verfahren rückwärts anwenden. Das heißt, wir beginnen mit einer Zahl $Z := Code(S)$ im Intervall $I := [0, 1)$ und konstruieren daraus die Original-Sequenz S. Wir wollen hier wieder davon ausgehen, dass die Länge von S bekannt und gleich l ist. Betrachten wir nun Z und überprüfen, in welchem Intervall $I' := [K(a_k - 1), K(a_k))$ sie liegt. Das zugehörige Symbol a_k ist unser erstes Zeichen. Nun betrachten wir als nächstes das Intervall I' und überprüfen, in welchem Teilintervall die Zahl Z jetzt liegt, wobei wir Z aber in den Grenzen von I' betrachten, also die Wahrscheinlichkeit anpassen müssen. Dann setzen wir $I := I'$ und beginnen mit dem nächsten Symbol.

Hier tritt analog zur Kodierung das Problem auf, dass wir unser Modell dem neuen Grenzen anpassen müssen. Dies erreicht man durch folgende Berechnung :

$$\begin{aligned} low' &:= low + K(a_{i-1}) \cdot (high - low) \\ high' &:= low + K(a_i) \cdot (high - low) \ , \end{aligned}$$

wobei i so zu wählen ist, dass

$$low \leq Z \leq high$$

gilt. a_i ist dann das nächste Symbol in der gesuchten Sequenz. Wie man leicht sieht, kann man auch diese Formeln wieder allgemein anwenden, da $[0, 1)$ nur einen Spezialfall bildet. Außerdem fällt sofort die Ähnlichkeit zum Kodieren in Formel (3) und (4) auf, da sich beide Algorithmen kaum unterscheiden und so leicht zu implementieren sind. Nach l Schritten erhalten wir dann die Sequenz S.

3.5 Beispiel Dekodierung

Zum besseren Verständnis des Verfahrens werden wir nun den Code aus Beispiel 2 wieder dekodieren. Unser Code war die Zahl $Z = 0,27234$, wir wollen wieder davon ausgehen, dass die Länge der original Sequenz bekannt ist, in unserem Beispiel also $l = 8$. (Die zur Erkennung des Symbols relevanten Stellen sind im Beispiel unterstrichen.) Beginnen wir also mit $low = 0$ und $high = 1$. Offensichtlich liegt $0,27234$ im Intervall $[0 \ldots 0,5)$. Wir erhalten also als erstes Symbol a und setzten

$$\begin{aligned} low &= 0 \\ high &= 0,5 \end{aligned}$$

Im nächsten Schritt stellen wir fest, dass $0,27234$ zwischen den Grenzen

$$\begin{aligned} low &= 0 + \underline{0,5} \cdot (0,5 - 0) = 0,25 \\ high &= 0 + \underline{0,75} \cdot (0,5 - 0) = 0,3125 \end{aligned}$$

liegt und dekodieren daher ein b. Im nächsten Schritt berechnen wir die Grenzen durch

$$\begin{aligned} low &= 0,25 + \underline{0} \cdot (0,3125 - 0,25) = 0,25 \\ high &= 0,25 + \underline{0,5} \cdot (0,3125 - 0,25) = 0,28125 \end{aligned}$$

was einem a entspricht. An dieser Stelle kürzen wir das Beispiel und springen direkt zum letzten Schritt

$$\begin{aligned} low &= 0,2723388672 + \underline{0} \cdot (0,2724609375 - 0,2723388672) = 0,2723388672 \\ high &= 0,2723388672 + \underline{0,5} \cdot (0,2724609375 - 0,2723388672) = 0,2723999024 \end{aligned}$$

also das abschliessende a der Sequenz *abaabcda*. Vergleicht man diese Rechnung mit der zur Kodierung, fällt die Änlichkeit sofort ins Auge. Aufgrund dieser Ähnlichkeit verzichten wir auf eine weitere Abbildung und weisen darauf hin, daß der Dekodiervorgang ebenfalls mit Hilfe von Abbildung 3 grafisch nachvollzogen werden kann. Beschreiben wir nun einen Algorithmus, der dieses Verfahren durchführt :

```
seq ='';
low =0;
high =1;
do {
    low'  = modell->untere_grenze(Zahl,low,high);
    high' = modell->obere_grenze (Zahl,low,high);
    low  = low';
    high = high';
    seq .= modell->symbol_zu_intervall(low,high);
} while ( !ende_der_sequenz() );
return(seq);
```

Im obigen Verfahren haben wir mit durch Gleitkommazahlen begrenzten Intervallen zwischen 0 und 1 gearbeitet. Daraus ergeben sich unter Umständen Zahlen unendlicher Länge. Dieses würde schon nach wenigen Schritten dazu führen, dass die Zahlendarstellung im Rechner nicht mehr ausreicht, um hinreichend genau rechnen zu können. Und selbst wenn es gelänge, ein Verfahren zu entwickeln, um mit diesen unendlichen Werten zu rechnen, könnte es nie effizient sein. Dem widmen wir uns nun im folgenden Kapitel. Hierzu sein erwähnt, daß die genannten Ausführungen gleichermaßen für Symbole wie auch Sequenzen gelten. Zu diesem Schluß kann man leicht kommen, wenn man die Sequenzen, wie sonst die Symbole, in ihrer Wahrscheinlichkeitsverteilung betrachtet. Hierzu sei auch verwiesen auf [2].

3.6 Eindeutigkeit der Darstellung

Sei im Folgenden

$$C(a_i) := K(a_{i-1}) + \frac{1}{2}\ P_M(a_i)\ .$$

$C(a_i)$ bildet also die Mitte des a_i zugeordneten Intervalls. Wir behaupten: Man kann $C(a_i)$ ersetzen durch einen gekürzten Code der Länge

$$l(a_i) = \lceil\ ld\ \frac{1}{P_M(a_i)}\rceil + 1\ .$$

Dann ist $\lfloor C(a_i)\rfloor_{l(a_i)}$ definiert als der auf $l(a_i)$ Stellen gekürzte Binärcode für a_i.

3.6.1 Beispiel

Nehmen wir an, wir hätten eine Sequenz

$$S = s_1 s_2 s_3 s_4$$

über einem Alphabet $A = \{a_1, \ldots, a_4\}$. Die vom Modell M errechneten Wahrscheinlichkeiten seien

$$P_M(a_1) = \frac{1}{2},\ P_M(a_2) = \frac{1}{4},\ P_M(a_3) = \frac{1}{8},\ P_M(a_4) = \frac{1}{8}.$$

Die folgende Tabelle zeigt einen möglichen Binärcode für diese Sequenz. Die Binärdarstellung von $C(a_i)$ wurde auf die Länge $\lceil\ ld\ \frac{1}{P_M(a_i)}\rceil + 1$ abgeschnitten und führt so zum entsprechenden Code.

i	$K(a_i)$	$C(a_i)$	binär	$l(a_i)$	$\lfloor C(a_i) \rfloor_{l(a_i)}$	Code
1	0,5	0,25	0,0100	2	0,01	01
2	0,75	0,625	0,1010	3	0,101	101
3	0,875	0,8125	0,1101	4	0,1101	1101
4	1,0	0,9375	0,1111	4	0,1111	1111

3.6.2 Beweis

Wir werden im Folgenden zeigen, dass ein Code, der auf die obige Art und Weise erzeugt wurde, eindeutig bestimmt ist.

Zuvor haben wir den Code $C(a_i)$ gewählt, um ein Zeichen a_i zu repräsentieren. Jedoch wäre jede andere Zahl aus dem Intervall $[K(a_{i-1}), K(a_i))$ eine ebenso gute Wahl für einen *eindeutigen* Code. Um also zu zeigen, dass der Code $\lfloor C(a_i) \rfloor_{l(a_i)}$ eindeutig ist, genügt es zu zeigen, dass dieser Code im Intervall $[K(a_{i-1}), K(a_i))$ liegt. Da wir die binäre Darstellung von $C(a_i)$ abschneiden, um $\lfloor C(a_i) \rfloor_{l(a_i)}$ zu erhalten, gilt

$$\lfloor C(a_i) \rfloor_{l(a_i)} \leq C(a_i).$$

Etwas genauer betrachtet,

$$0 \leq C(a_i) - \lfloor C(a_i) \rfloor_{l(a_i)} \leq \frac{1}{2^{l(a_i)}}. \tag{5}$$

Da außerdem per Definition $C(a_i)$ echt kleiner als $K(a_i)$ ist, folgt

$$\lfloor C(a_i) \rfloor_{l(a_i)} < K(a_i).$$

Damit ist die obere Schranke erfüllt. Um nun zu zeigen, dass $\lfloor C(a_i) \rfloor_{l(a_i)} \geq K(a_{i-1})$, betrachten wir folgenden Zusammenhang:

$$\begin{aligned}
\frac{1}{2^{l(a_i)}} &\overset{\text{def}}{=} \frac{1}{2^{\lceil ld \frac{1}{P_M(a_i)} \rceil + 1}} \\
&\leq \frac{1}{2^{ld \frac{1}{P_M(a_i)} + 1}} \\
&= \frac{1}{2 \cdot 2^{ld \frac{1}{P_M(a_i)}}} \\
&= \frac{1}{2 \frac{1}{P_M(a_i)}} \\
&= \frac{P_M(a_i)}{2} .
\end{aligned}$$

Weiterhin gilt

$$\frac{P_M(a_i)}{2} = C(a_i) - K(a_{i-1})$$

nach Definition von $C(a_i)$. Deswegen folgt

$$C(a_i) - K(a_{i-1}) \geq \frac{1}{2^{l(a_i)}} . \tag{6}$$

Durch Kombination der Gleichungen (5) und (6) ergibt sich

$$\lfloor C(a_i) \rfloor_{l(a_i)} \geq K(a_{i-1}) . \tag{7}$$

Somit gilt abschließend

$$K(a_{i-1}) \leq \lfloor C(a_i) \rfloor_{l(a_i)} < K(a_i) \ ,$$

also

$$\lfloor C(a_i) \rfloor_{l(a_i)} \in [K(a_{i-1}), K(a_i)) \ .$$

□

Damit ist nun bewiesen, dass $\lfloor C(a_i) \rfloor_{l(a_i)}$ eindeutige Repräsentation von $C(a_i)$ ist. Um nun zu zeigen, dass umgekehrt dieser Code auch eindeutig dekodierbar ist, werden wir zeigen, dass er ein Präfixcode ist. Denn wie wir schon wissen, ist *jeder* Präfixcode eindeutig dekodierbar.

Gegeben sei eine Zahl a aus dem Intervall $[0, 1)$ mit binärer Repräsentation der Länge n, $[a_1, a_2, ..., a_n]$. Dann ist klar, dass jede andere Zahl b mit $[a_1, a_2, ..., a_n]$ als Präfix der Binärdarstellung im Intervall $[a, a + \frac{1}{2^n})$ liegen muss.

Wenn nun a_i und a_j zwei unterschiedliche Symbole sind, dann wissen wir, dass die Werte $\lfloor C(a_i) \rfloor_{l(a_i)}$ und $\lfloor C(a_j) \rfloor_{l(a_j)}$ in zwei *disjunkten* Intervallen

$$[K(a_{i-1}), K(a_i)), [K(a_{j-1}), K(a_j))$$

liegen. Wenn wir also zeigen können, dass für jedes Symbol a_i das Intervall

$$[\lfloor C(a_i) \rfloor_{l(a_i)}, \lfloor C(a_i) \rfloor_{l(a_i)} + \frac{1}{2^{l(a_i)}})$$

komplett im Intervall $[K(a_{i-1}), K(a_i))$ enthalten ist, dann bedeutet dies, dass der Code dieses Symbols a_i nicht Präfix eines Codes eines anderen Symbols a_j sein kann.

Nach Gleichung (7) gilt, dass $\lfloor C(a_i) \rfloor_{l(a_i)} \geq K(a_{i-1})$. Somit ist die untere Grenze bereits bewiesen und es reicht zu zeigen, dass

$$K(a_i) - \lfloor C(a_i) \rfloor_{l(a_i)} > \frac{1}{2^{l(a_i)}}.$$

Dies ist aber klar, denn

$$\begin{aligned} K(a_i) - \lfloor C(a_i) \rfloor_{l(a_i)} &> K(a_i) - C(a_i) \\ &= \frac{P_M(a_i)}{2} \\ &\geq \frac{1}{2^{l(a_i)}}. \end{aligned}$$

Also ist der vorliegende Code präfixfrei. Insbesondere lässt sich somit durch Kürzen von $C(a_i)$ auf $l(a_i)$ Bits ein eindeutig dekodierbarer Code generieren.

Somit wurde auch das Problem der endlichen Zahlendarstellung für Gleitkommazahlen zufriedenstellend gelöst.

3.7 Zusammenfassung

Wir haben nun die theoretische Funktionsweise der Arithmetischen Kodierung kennengelernt und an Beispielen deren Funktionsweise erläutert. Dabei haben wir uns jedoch bisher auf eine bestimmte Arithmetik beschränkt: die Gleitkomma-Arithmetik. Wir haben zwar

gezeigt, dass es durch Wahl bestimmter Codes möglich ist, diese auch in Implementierungen zu verwenden, jedoch ist eine Implementierung mit Integerwerten[3] oft einfacher, da insbesondere die oben geschilderten Probleme der endlich genauen Arithmetik elegant umgangen werden können. Wir wollen eine solche Kodierung mittels Integerwerten im folgenden betrachten. Weiterhin generieren wir nun keine reelle Zahl des Dezimalsystems mehr als Code, sondern kodieren die Quelle als Bitsequenz, also Folge von Nullen und Einsen. Dazu wird es nötig werden, diese Bitsequenz beim Abschluß der Kodierung entsprechend zu terminieren.

4 Kodierung als Bitsequenz

4.1 Motivation

Um das Verfahren effizient zu implementieren gilt es, folgende Einschränkungen gegenüber dem Vorgestellten zu machen: Zum einen stehen keine (unendlichen) reellen Zahlen zur Verfügung, zum anderen ist gerade in einfachen Systemen (z. B. Fax) eine reine Integer-Implementierung deutlich einfacher und vor allem schneller.

In diesem Kapitel wird gezeigt, wie man mit sehr geringem Speicherbedarf (nur je ein Register für die Grenzen, in den Beispielen 32 Bit breit) und wenigen einfachen Integer-Operationen die Arithmetische Kodierung implementieren kann. Die Ausgabe des Kodierers ist dabei eine eindeutige Bitsequenz, welche dann je nach Anwendung gespeichert oder gesendet werden kann.

4.2 Modell-Abstraktion

In ganzen Zahlen kann man die Wahrscheinlichkeiten nicht mehr als Bruchteile von 1 angeben. Da sich die Wahrscheinlichkeiten bei einfachen Modellen aber direkt durch die Häufigkeit der Symbole darstellen lassen, normiert man sie auf die Gesamtzahl der Symbole und gibt als untere Grenze die Summe der Häufigkeiten der kleineren (also in kanonischer Ordnung vorhergehenden) Symbole an, als obere Grenze diese Summe plus die Häufigkeit des zu kodierenden Symbols:

$$\begin{aligned} low\_count &= \sum_{i=0}^{symbol-1} CumCount[i] \ , \\ high\_count &= low\_count + CumCount[symbol] \ . \end{aligned}$$

Das entspricht insofern den aus dem letzten Abschnitt bekannten Wahrscheinlichkeiten, als das nun nicht mehr erst die Häufigkeit eines Symbols durch die Gesamtzahl geteilt wird, um danach zu summieren, sondern nur summiert wird. Daraus ergeben sich also folgende Verhältnisse:

$$\begin{aligned} low\_count &= low \cdot total \ , \\ high\_count &= high \cdot total \ , \end{aligned}$$

wobei `total` jeweils die Gesamtzahl der Häufigkeiten ist.

[3] diese entsprechen der Rechnerdarstellung für natürliche Zahlen

4.3 Kodierung

Der Encoder besteht aus einer Funktion und statischen Variablen, die den Zustand festhalten, in dem sich der Encoder befindet: [4]

- `mLow` [5] speichert die aktuelle untere Grenze und wird mit 0 initialisiert.
- `mHigh` speichert die aktuelle obere Grenze und wird mit `0x7FFFFFFF`, also dem Maximalwert initialisiert.
- `mStep` speichert eine später beschriebene Schrittweite, die im Encoder nicht statisch sein muss, aber aus Gründen der Übersichtlichkeit so eingeführt wird, da der Decoder auf diese Eigenschaft angewiesen ist.

Zu beachten ist, dass man von den 32 Bit nur 31 nutzen kann, um Überläufe zu vermeiden, aber darauf wird an den entsprechenden Stellen noch hingewiesen. Die Funktionsdefinition sieht nun wie folgt aus:

```
void Encoder( unsigned int low_count,
              unsigned int high_count,
              unsigned int total );
```

Beim Aufruf wird dem Encoder also die vom Modell bestimmte kumulierte Wahrscheinlichkeit des aktuellen Symbols sowie des Vorgängers übergeben. Darauf aufbauend werden die obere und die untere Grenze eingeschränkt. Zuerst wird der Bereich zwischem `mLow` und `mHigh` in `total` Schritte unterteilt, was eine Schrittweite von

```
mStep = ( mHigh - mLow + 1 ) / total;
```

ergibt. Zur Differenz von `mHigh` und `mLow` muss 1 addiert werden, da `mHigh` ja die oben offene Grenze darstellt und somit das Intervall um eins größer ist als die gegebene Differenz. Am Beispiel des Dezimalsystems wäre dies äquivalent zu einem Bereich von 0 bis $99.\bar{9}$, bei dem man die obere Grenze auch abgekürzt als 99 speichert. Die Nachkommastellen sorgen aber dafür, dass das Intervall eigentlich um 1 größer wäre als $99 - 0$.

Hier wird auch deutlich, warum der Bereich zu Anfang auf 31 Bit eingeschränkt wurde: Die Differenz von `mLow` und `mHigh` direkt nach der Initialisierung ist die mit der gegebenen Anzahl Bits maximal mögliche darstellbare Zahl. Wenn man zu dieser noch 1 addiert, dann überschreitet man diesen Bereich und erhält 0, was eine weitere Ausführung unmöglich macht.

Die obere Grenze wird nun durch

```
mHigh = mLow + mStep * high_count - 1;
```

und die untere durch

```
mLow = mLow + mStep * low_count;
```

[4] Die Implementierung folgt [7].

[5] Der Präfix m bezeichnet statische Variablen in Anlehnung an Member-Variablen in der objektorientierten Programmierung.

aktualisiert. Da beide Berechnungen auf dem vorherigen Wert von `mLow` aufbauen, muss dieser zuletzt aktualisiert werden, solange man keine Hilfsvariable einführen will. Dass `mHigh` noch um eins dekrementiert wird, liegt daran, dass man es wie gesagt mit einem nach oben offenen Intervall zu tun hat und hier die bei der Berechnung von `mStep` einbezogenen imaginären Nachkommastellen wieder abgezogen werden.

4.4 Beispiel: Kodierung

Um die Funktionsweise des Kodierers zu verdeutlichen, betrachten wir erneut das Beispiel 2.2, allerdings kodieren wir vorerst nur die ersten beiden Zeichen, `ab`. Vom Modell haben wir folgende Daten:

Symbol	Häufigkeit	`low_count`	`high_count`
a	4	0	4
b	2	4	6
c	1	6	7
d	1	7	8

Tabelle 2: Modell zu Beispiel 2.2

Zuerst wird der Encoder initialisiert:

```
mBuffer = 0;
mLow = 0;
mHigh = 0x7FFFFFFF;
```

Anschließend kann man die Symbole der Eingabesequenz kodieren. Man beachte, dass das Modell statisch und daher `total` konstant den Wert 8 hat.

1. 'a'

```
mStep = ( mHigh - mLow + 1 ) / total;
      = ( 0x7FFFFFFF - 0 + 1) / 8
      = 0x80000000 / 8
      = 0x10000000

mHigh = mLow + mStep * high_count - 1;
      = 0 + 0x10000000 * 4 - 1
      = 0x40000000 - 1
      = 0x3FFFFFFF

mLow  = mLow + mStep * low_count;
      = 0 + 0x10000000 * 0
      = 0
```

2. 'b'

```
mStep = ( mHigh - mLow + 1 ) / total;
      = ( 0x3FFFFFFF - 0 + 1 ) / 8
      = 0x40000000 / 8
      = 0x08000000

mHigh = mLow + mStep * high_count - 1;
      = 0 + 0x08000000 * 6 - 1
      = 0x30000000 - 1
      = 0x2FFFFFFF

mLow  = mLow + mStep * low_count;
      = 0 + 0x08000000 * 4
      = 0x20000000
```

Nach diesen zwei Symbolen können wir also einen beliebigen Wert aus dem Intervall `0x20000000` - `0x2FFFFFFF` speichern.

4.5 Dekodierung

Die Aufgabe des Decoders ist es, die Schritte des Encoders 1:1 nachzuvollziehen. Dazu muss zuerst aus der gegebenen Bitsequenz das jeweils folgende Zeichen bestimmt werden, mit welchem dann in einem zweiten Schritt die Grenzen analog zum Encoder aktualisiert werden. Der Decoder gliedert sich also in zwei getrennte Funktionen:

```
unsigned int Decode_Target( unsigned int total );

void Decode( unsigned int low_count,
             unsigned int high_count );
```

Die erste Funktion, `Decode_Target()`, bestimmt, in welchem Intervall das vom Encoder gespeicherte Symbol liegt. Dazu errechnet es den Codewert des Symbols:

```
mStep = ( mHigh - mLow + 1 ) / total;

value = ( mBuffer - mLow ) / mStep;
```

Dabei ist `mBuffer` die Variable, die die zu dekodierende Sequenz enthält. Das Modell kann nun an Hand des Rückgabewerts von `Decode_Target()`, `value`, bestimmen, welches Symbol kodiert wurde, indem es die Häufigkeits-Intervalle aller Symbole durchgeht und prüft, ob `value` hineinfällt. Sobald das passende Symbol gefunden ist, können `mLow` und `mHigh` mittels `Decode()` aktualisiert werden, was genau wie in `Encode()` durchzuführen ist:

```
mHigh = mLow + mStep * high_count - 1;

mLow = mLow + mStep * low_count;
```

Dabei ist zu beachten, dass `mStep` nun funktionsübergreifend zur Verfügung stehen muss, weshalb es zu Anfang statisch definiert wurde.

4.6 Beispiel: Dekodierer

Als Beispiel dekodieren wir die in 4.4 erzeugte Bitsequenz. Wie nehmen an, der aus dem Intervall gewählte Wert ist `0x28000000`. Also initialisieren wir den Dekodierer mit folgenden Werten:

```
mBuffer = 0x28000000;
mLow = 0;
mHigh = 0x7FFFFFFF;
```

1. 'a' Zuerst wird mittels `Decode_Target()` der kodierte Wert für das Modell umgerechnet:

   ```
   mStep = ( mHigh - mLow + 1 ) / total;
         = ( 0x7FFFFFFF - 0 + 1 ) / 8
         = 0x80000000 / 8
         = 0x10000000

   value = ( mBuffer - mLow ) / mStep;
         = ( 0x28000000 - 0 ) / 0x10000000
         = 0x28000000 / 0x10000000
         = 2
   ```

   Diese 2 vergleicht man nun mit dem Modell aus Tabelle 2. Sie liegt im Intervall [0,4), d. h. das kodierte Symbol war ein 'a'. Dementsprechend passen wir nun in `Decode()` die Grenzen an:

   ```
   mHigh = mLow + mStep * high_count - 1;
         = 0 + 0x10000000 * 4 - 1
         = 0x40000000 - 1
         = 0x3FFFFFFF

   mLow  = mLow + mStep * low_count;
         = 0 + 0x10000000 * 0
         = 0
   ```

2. 'b'

   ```
   Decode_Target():

   mStep = ( mHigh - mLow + 1 ) / total;
         = ( 0x3FFFFFFF - 0 + 1 ) / 8
         = 0x40000000 / 8
         = 0x08000000
   ```

```
value = ( mBuffer - mLow ) / mStep;
      = ( 0x28000000 - 0 ) / 0x08000000
      = 0x28000000 / 0x08000000
      = 5

Decode():

mHigh = mLow + mStep * high_count - 1;
      = 0 + 0x08000000 * 6 - 1
      = 0x30000000 - 1
      = 0x2FFFFFFF

mLow  = mLow + mStep * low_count;
      = 0 + 0x08000000 * 4
      = 0x20000000
```

Die 5 liegt im Intervall des 'b', womit die Bitsequenz erfolgreich dekodiert ist.

5 Skalierung in begrenzten Zahlenbereichen

5.1 Motivation

Das Problem bei dem bisher beschriebenen Verfahren ist nun, dass bereits nach wenigen kodierten Symbolen `mLow` und `mHigh` so nahe beieinander liegen, dass ein weiteres Kodieren unmöglich wird. Eine einfache Beobachtung schafft da Abhilfe:

5.2 Die Skalierungsfunktionen E1 und E2

Sobald `mLow` und `mHigh` beide in einer Hälfte des Zahlenbereichs liegen, d. h. in unserem Fall beide kleiner oder beide größer gleich `0x40000000` sind, ist garantiert, dass sie diesen Bereich nicht mehr verlassen, da alle folgenden Symbole den gegebenen Bereich ja nur noch weiter einschränken und nicht erweitern. Das wiederum bedeutet, dass die Eigenschaft, dass beide in einer Hälfte liegen, für die folgenden Symbole unwichtig sind und man diese Information problemlos speichern und aus der Betrachtung ausblenden kann.

In der Praxis sieht das so aus, dass die oberen Bits der Variablen `mLow` und `mHigh` übereinstimmen. Liegen beide in der unteren Hälfte, dann steht dort eine 0, liegen sie in der oberen, dann ist es eine 1. Stellt der Encoder also fest, dass eine solche Situation gegeben ist, so kann er die 0 bzw. 1 schon zur Ausgabesequenz hinzufügen und die Grenzen `mLow` und `mHigh` entsprechend aktualisieren, indem er sie wieder auf den gesamten Bereich erweitert. Wir nennen dies eine $E1$- bzw. $E2$-Skalierung:

```
while( ( mHigh < g_Half ) || ( mLow >= g_Half ) ) {
    if( mHigh < g_Half ) // E1
    {
        SetBit( 0 );
        mLow = mLow * 2;
        mHigh = mHigh * 2 + 1;
```

```
    }
    else if(mLow >= g_Half ) // E2
    {
        SetBit( 1 );
        mLow = 2 * ( mLow - g_Half );
        mHigh = 2 * ( mHigh - g_Half ) + 1;
    }
}
```

`g_Half` ist die globale Konstante `0x40000000`, die die Hälfte des Puffers markiert. Die Multiplikation mit 2 sorgt dafür, dass der Bereich vergrößert wird, die Addition von 1 bei `mHigh` ist wieder darin begründet, dass es sich um ein nach oben offenes Intervall handelt und hier die imaginären Nachkommastellen berücksichtigt werden.

`SetBit()` fügt der Ausgabe-Bitsequenz ein Bit hinzu. Die komplementäre Funktion im Dekodierer heißt `GetBit()` und liest ein Bit. Beide Funktionen arbeiten strikt sequenziell, das heißt, es gibt keine Möglichkeit, in Bitsequenzen zu springen. Das ist aber auch weder notwendig noch wünschenswert, da der Algorithmus an sich sequenziell arbeitet.

5.3 Die Skalierungsfunktion E3

Obgleich die E1- bzw. E2-Skalierungen Schritte in die richtige Richtung sind, sind sie allein doch noch nicht ausreichend. Das Problem ist, dass sie nicht wirken, wenn sich `mLow` von unten und `mHigh` von oben der Hälfte nähern. Das kann so weit gehen, dass $mLow$ den Wert `0x3FFFFFFF` annimmt, `mHigh` aber `0x40000000`. Beide Variablen sind noch in ihren Hälften, die E1-/E2-Skalierung hilft nicht weiter, und doch hat sich der Bereich so weit verkleinert, dass der Encoder nicht mehr sinnvoll weiterarbeiten kann.

An dieser Stelle greift die $E3$-Skalierung: Sobald `mLow` größer als das untere Viertel und `mHigh` kleiner als das obere Viertel ist, dann ist der Bereich auf weniger als die Hälfte geschrumpft und es ist garantiert, dass `mLow` und `mHigh` sich immer zwischen `g_FirstQuarter` und `g_ThirdQuarter`, zwei globalen Konstanten die die Viertel-Grenzen markieren, bewegen. Man kann zu diesem Zeitpunkt noch nicht sagen, in welcher Hälfte das Intervall enden wird, d.h. ob als nächstes E1 oder E2-Skalierung zur Anwendung kommt, aber sobald die nächste dieser Art durchgeführt wird, ist klar, auf welcher Seite sich das Intervall befindet.

Also speichert man in einer Hilfsvariablen, `mScale`, wie oft seit der letzten E1- bzw. E2-Skalierung dieser Zustand eintrat und erweitert den Bereich entsprechend:

```
while( ( g_FirstQuarter <= mLow ) && ( mHigh < g_ThirdQuarter )) {
    mScale++;
    mLow = 2 * ( mLow - g_FirstQuarter );
    mHigh = 2 * ( mHigh - g_FirstQuarter ) + 1;
}
```

Wenn man erst eine E3- und danach eine E1-Skalierung durchführt heißt das, dass das Intervall ohne jede Skalierung zwischen `g_FirstQuarter` und `g_Half` gelegen hätte, was einer E1- gefolgt von einer E2-Skalierung entspricht. Die Folge E3-E2 ist analog zu betrachten, genau wie wiederholte E3-Skalierungen, weshalb man nach der jeweiligen E1- bzw. E2-Skalierung so viele inverse Bits speichert, wie E3-Skalierungen durchgeführt wurden:

```
while( ( mHigh < g_Half ) || ( mLow >= g_Half ) ) {
    if( mHigh < g_Half ) // E1
    {
        SetBit( 0 );
        mLow = mLow * 2;
        mHigh = mHigh * 2 + 1;

        // E3
        for(; mScale > 0; mScale-- )
            SetBit( 1 );
    }
    else if(mLow >= g_Half ) // E2
    {
        SetBit( 1 );
        mLow = 2 * ( mLow - g_Half );
        mHigh = 2 * ( mHigh - g_Half ) + 1;

        // E3
        for(; mScale > 0; mScale-- )
            SetBit( 0 );
    }
}
```

Diesen Zusammenhang sollen auch die beiden Abbildungen 4 und 5 auf Seite 31 verdeutlichen. Man nehme an, das Alphabet sei $A := \{a, b, c, d, e\}$, die Wahrscheinlichkeiten seien gleichverteilt. Zu Abbildung 4: Im ersten Schritt wird ein c gelesen. Dieses c entspricht den Intervall $[0,4; 0,6)$. Dieses liegt im zweiten und dritten Viertel. Daher wenden wir die E3-Skalierung an. Nun liegt es im zweiten Schritt immer noch innerhalb des zweiten und dritten Viertels. Also wenden wir nochmals die E3-Skalierung an. Nun sind wir soweit, dass das Intervall über die Grenzen des zweiten (und sogar auch des dritten) Viertels hinausragt. Also haben wir den Bereich ausreichend angepasst (er umfasst nun wieder mehr als ein Viertel des Intervalls) und können das nächste Zeichen lesen. Dies sei nun ein b. Das zugehörige Teilintervall liegt voll in der unteren Hälfte von $[0,375; 0,5)$. Daher wenden wir die E1-Skalierung an. Daher senden wir für diese eine 0 und dann für die beiden E3-Skalierungen das Inverse $!0 = 1$.

In Abbildung 5 wollen wir klar machen, warum es sinnvoll war, wie oben beschrieben, die Codesequenz 011 zu senden. Wir starten wieder mit dem vollen Intervall $[0, 1)$ und führen nun nur E1- und E2-Skalierungen entsprechend der Bitsequenz 011 aus. Also im ersten Schritt E1, dann zweimal E2. Es fällt auf, dass das Endintervall nach diesen drei Anwendungen dasselbe ist, wie wir in Abbildung 4 durch Anwendung von E3, E3 und E1 erhalten haben.

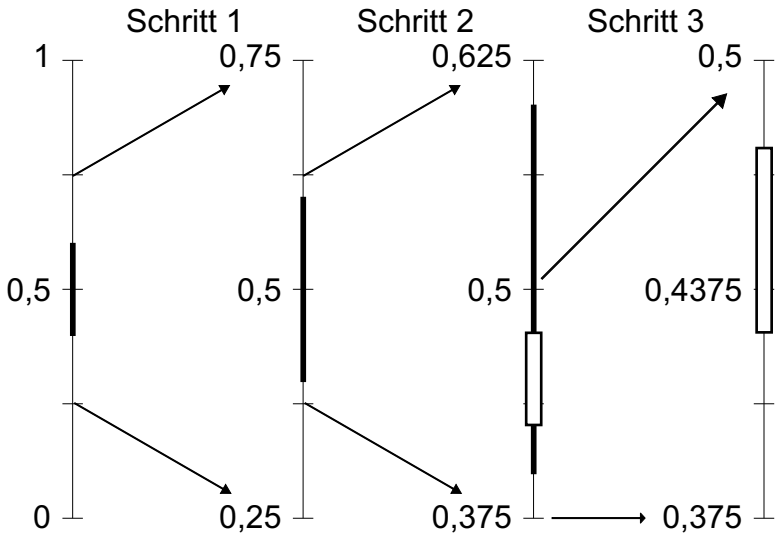

Abbildung 4: Anwendung der E3-Skalierung

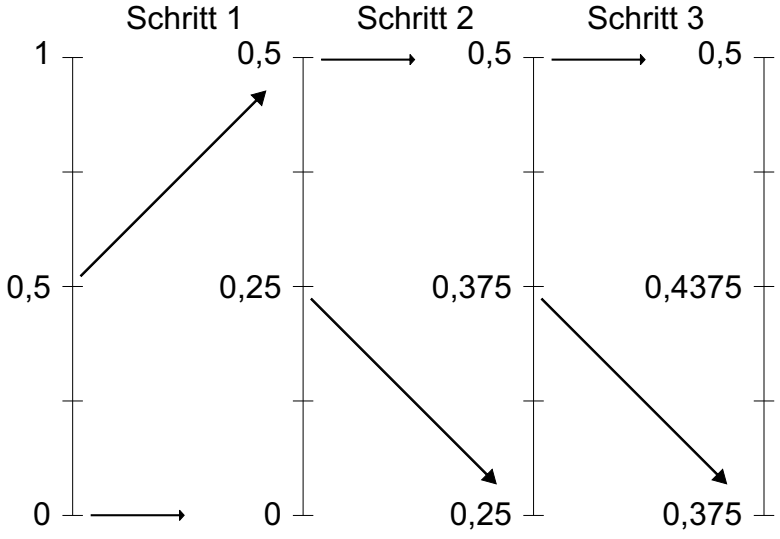

Abbildung 5: Vergleich ohne E3-Skalierung

Dieser bemerkenswerte Zusammenhang besteht nicht nur im gewählten Beispiel sondern allgemein. Für zwei Funktionen f und g bezeichne $g \circ f$ die Hintereinanderausführung von f und g. Dann können wir unsere Erkenntnis wie folgt formulieren.

LEMMA 2 *Angewandt auf jede beliebige Sequenz gelten die Gleichheiten:*
$E1 \circ (E3)^n = (E2)^n \circ E1$,
$E2 \circ (E3)^n = (E1)^n \circ E2$.

Beweis:
Wählt man verkürzend $a := low$ und $b := high$, und beschränkt die Rechnung auf das (reelle) Intervall $[0, 1)$, so kann man die Skalierungsfunktionen beschreiben durch

$$\begin{aligned} E1\begin{pmatrix} a \\ b \end{pmatrix} &= \begin{pmatrix} 2a \\ 2b \end{pmatrix} \\ E2\begin{pmatrix} a \\ b \end{pmatrix} &= \begin{pmatrix} 2a-1 \\ 2b-1 \end{pmatrix} \\ E3\begin{pmatrix} a \\ b \end{pmatrix} &= \begin{pmatrix} 2a-\frac{1}{2} \\ 2b-\frac{1}{2} \end{pmatrix} \end{aligned}$$

Die n-te Iteration ergibt dann

$$\begin{aligned} E1^n\begin{pmatrix} a \\ b \end{pmatrix} &= \begin{pmatrix} 2^n a \\ 2^n b \end{pmatrix} \\ E2^n\begin{pmatrix} a \\ b \end{pmatrix} &= \begin{pmatrix} 2^n a - 2^n + 1 \\ 2^n b - 2^n + 1 \end{pmatrix} \\ E3^n\begin{pmatrix} a \\ b \end{pmatrix} &= \begin{pmatrix} 2^n a - 2^{n-1} + \frac{1}{2} \\ 2^n b - 2^{n-1} + \frac{1}{2} \end{pmatrix} \end{aligned}$$

Der einfache Beweis dieses Zusammenhangs sei durch vollständige Induktion dem Leser überlassen. Daraus ergibt sich für unsere Behauptung:

$$(E1 \circ (E3)^n)\begin{pmatrix} a \\ b \end{pmatrix} = E1\begin{pmatrix} 2^n a - 2^{n-1} + \frac{1}{2} \\ 2^n b - 2^{n-1} + \frac{1}{2} \end{pmatrix} = \begin{pmatrix} 2^{n+1} a - 2^n + 1 \\ 2^{n+1} b - 2^n + 1 \end{pmatrix} \quad (8)$$

$$((E2)^n \circ E1)\begin{pmatrix} a \\ b \end{pmatrix} = (E2)^n\begin{pmatrix} 2a \\ 2b \end{pmatrix} = \begin{pmatrix} 2^{n+1} a - 2^n + 1 \\ 2^{n+1} b - 2^n + 1 \end{pmatrix} \quad (9)$$

Durch Gleichsetzen von (8) und (9) folgt:

$$E1 \circ (E3)^n = (E2)^n \circ E1$$

Der Beweis der zweiten Gleichheit verläuft völlig analog. □

5.4 Beispiel Kodierung

Um das E3-Scaling weiter zu demonstieren, wird nun die Eingabesequenz `abccedac` kodiert. Das Modell muss dementsprechend gemäß Tabelle 3 auf Seite 33 angepasst werden.

Um das Beispiel etwas übersichtlicher zu gestalten, ist es in Tabelle 4, ebenfalls auf Seite 34, zusammengefasst. In der ersten Spalte steht jeweils das zu kodierende Symbol, auf das die an `Encode()` übergebenen Parameter `low_count` , `high_count` und `total` folgen. Daraus werden die neuen Grenzen `mLow` und `mHigh` kodiert. Im nächsten Schritt folgt nun wahlweise eine E1/E2-Skalierung, bei der die ausgegebenen Bits und die danach gesetzten neuen Grenzen angegeben werden. Unterstrichene Bits sind dabei die, die durch den Abbau einer eventuell vorhandenen E3-Skalierung generiert wurden. Im letzten Schritt wird die Anzahl der nötigen E3-Skalierungen bestimmt und wiederum werden die Grenzen angepasst. In der letzten Zeile stehen die Bits, die noch nötig sind, um einen Wert aus dem verbliebenen Intervall eindeutig festzulegen.

Das Beispiel beschränkt sich auf 7 Bit Integer, da diese für eine einfache Sequenz ausreichen und einfacher zu lesen sind als 31 Bit.

Symbol	Häufigkeit	`low_count`	`high_count`
a	2	0	2
b	1	2	3
c	3	3	6
d	1	6	7
e	1	7	8

Tabelle 3: Modell zum Beispiel der Skalierungsfunktionen

Sym	l_c	h_c	t	mStep	mLow	mHigh	Bits	E1/2-mLow	E1/2-mHigh	mScale	E3-mLow	E3-mHigh
a	0	2	8	16	0000000 [0]	0011111 [31]	00	0000000 [0]	1111111 [127]	0		
b	2	3	8	16	0100000 [32]	0101111 [47]	010	0000000 [0]	1111111 [127]	0		
c	3	6	8	16	0110000 [48]	1011111 [95]				1	0100000 [32]	1111111 [127]
c	3	6	8	12	1000100 [68]	1100111 [103]	10			0		
e	7	8	8	9	1001111 [71]	1001111 [79]	100	0111000 [56]	1111111 [127]	0		
d	6	7	8	9	1101110 [110]	1110110 [118]	11	0111000 [56]	1011011 [91]	1	0110000 [48]	1110111 [119]
a	0	2	8	9	0110000 [48]	1000001 [65]				3	0000000 [0]	1000111 [71]
c	3	6	8	9	0011011 [27]	0110101 [53]	0111	0110110 [54]	1101011 [107]	0		
rest							1					

Tabelle 4: Beispiel der Skalierungsfunktionen im Kodierer

Sym	neu eigelesenes Symbol
l_c	low_count, untere Grenze der Häufigkeit des Symbols
h_c	high_count, obere Grenze der Häufigkeit des Symbols
t	total, Summe über die Symbolhäufigkeiten
mStep	Schrittweite
mLow	untere Grenze im neuen Intervall
mHigh	obere Grenze im neuen Intervall
Bits	Bits, die durch E1/E2-Skalierungen gesendet und entfernt werden
E1/2-mLow	untere Grenze nach E1/E2-Skalierung
E1/2-mHigh	obere Grenze nach E1/E2-Skalierung
mScale	Summe der neuen und noch nicht abgebauten E3-Skalierungen
E3-mLow	untere Grenze nach E3-Skalierung
E3-mHigh	obere Grenze nach E3-Skalierung

Tabelle 5: Spaltenerklärungen

5.5 Dekodierung

Da der Decoder die Schritte des Encoders 1:1 verfolgt, sind die Skalierungen analog anzuwenden, nur dass man parallel jeweils noch `mBuffer` skalieren muss, damit dieser wieder im richtigen Verhältnis zu den Grenzen steht. Dazu verwendet man im Prinzip die gleiche Methode wie bei `mLow` und `mHigh`, nur dass jetzt das neue Bit nicht aus imaginären Nachkommastellen stammt, sondern aus der vom Encoder gespeicherten Codesequenz.

```
// bei E1-Skalierung
mLow = mLow * 2;
mHigh = mHigh * 2 + 1;
mBuffer = 2 * mBuffer + GetBit();

// bei E2-Skalierung
mLow = 2 * ( mLow - g_Half );
mHigh = 2 * ( mHigh - g_Half ) + 1;
mBuffer = 2 * ( mBuffer - g_Half ) + GetBit();

// bei E3-Skalierung
mLow = 2 * ( mLow - g_FirstQuarter );
mHigh = 2 * ( mHigh - g_FirstQuarter ) + 1;
mBuffer = 2 * ( mBuffer - g_FirstQuarter ) + GetBit();
```

5.6 Beispiel Dekodierung

Im folgenden Beispiel wird die soeben kodierte Sequenz dekodiert. Als Eingabe dient also die Bitsequenz `00010101001101111`. Die ersten 7 Bit davon werden in `mBuffer` eingelesen, bei Skalierungen werden entsprechend viele Bits vorne entfernt. Da nur immer die ersten 7 berücksichtigt werden sind die restlichen aus Platzgründen weggelassen. Zu beachten ist, daß auch bei E3-Skalierungen `mBuffer` angepaßt wird, auch wenn noch keine Bits gesendet werden können.

Im Gegensatz zur ersten Tabelle mußten hier einige Bezeichner abgekürzt werden: St ist `mStep` , Sy ist Sym, Sc ist `mScale`.

St	mBuffer	l_c	h_c	Sy	mLow	mHigh	Bits	E1/2-mLow	E1/2-mHigh	Sc	E3-mLow	E3-mHigh
16	0001010 [10]	0	2	a	0000000 [0]	0011111 [31]	00	0000000 [0]	1111111 [127]	0		
16	0101010 [42]	2	3	b	0100000 [32]	0101111 [47]	010	0000000 [0]	1111111 [127]	0		
16	1010011 [83]	3	6	c	0110000 [48]	1011111 [95]				1	0100000 [32]	1111111 [127]
12	0100110 [38]	3	6	c	1000100 [68]	1100111 [103]	1$\underline{0}$			0		
9	1001101 [77]	7	8	e	1001111 [71]	1001111 [79]	100	0111000 [56]	1111111 [127]	0		
9	1101111 [111]	6	7	d	1101110 [110]	1110110 [118]	11	0111000 [56]	1011011 [91]	1	0110000 [48]	1110111 [119]
9	1110000 [112]	0	2	a	0110000 [48]	1000001 [65]				3	0000000 [0]	1000111 [71]
9	1000000 [64]	3	6	c	0011011 [27]	0110101 [53]	0$\underline{111}$	0110110 [54]	1101011 [107]	0		

Tabelle 6: Beispiel der Skalierungsfunktionen im Dekodierer

6 Wertebereiche

6.1 Intervallgröße

Durch die gegebenen Skalierungsfunktionen ist inzwischen klar, dass sich die Werte `mLow` und `mHigh` nach einem Durchlauf von `Encode()` bzw. `Decode()` in genau zwei verschiedenen Bereichen befinden können:

- $mLow < FirstQuarter < Half \leq mHigh$,
- $mLow < Half < ThirdQuarter \leq mHigh$.

Das Intervall umfasst also zu jedem Zeitpunkt mindestens ein Viertel des maximalen Bereichs, mehr ist möglich, aber nicht garantiert.

Bei der Bestimmung von `mStep` wird nun das Intervall durch `total` geteilt. Sobald `total` aber größer als das Intervall ist, ergibt diese Integer-Division 0, womit der Algorithmus nicht weiterarbeiten kann. Deshalb ist vom Modell aus sicherzustellen, dass `total` das Minimum der Größe des Intervalls nicht überschreitet, d. h. dass es immer kleiner oder gleich einem Viertel des maximalen Bereichs ist. In den Beispielen bleiben damit von dem 31 Bits großen Bereich (ein Bit dient ja schon zur Vermeidung von Überläufen) noch 29 Bits übrig, was für übliche Modelle (2^{29} = 512MB) immer noch problemlos ausreicht.

6.2 Alternative Berechnung

Der Vollständigkeit halber sei noch auf eine zweite Methode verwiesen, die Grenzen beim (De-)Kodieren zu bestimmen. [6] Im vorgestellten Algorithmus berechnet man zuerst eine Schrittweite `mStep` und multipliziert diese dann mit den kumulativen Wahrscheinlichkeiten aus dem Modell, `low_count` bzw. `high_count` . In ungünstigen Fällen kann das dazu führen, dass man einen großen Teil des Intervalls nicht zur Kodierung eines Symbols nutzen kann:

Wenn man ein Intervall der Größe 7 hat und das Modell für `total` einen Wert von 4 übergibt, so ergibt sich eine Schrittweite von `7 / 4 = 1`, da bei der Integer-Division abgerundet wird. Das hat zur Folge, dass man die obere Grenze `mHigh` bei einem `high_count` von 4 aber nicht auf 7, sondern auf 4 liegen hat, das heißt, fast die Hälfte des Intervalls fällt weg.

Um diese Einschränkung zu umgehen, kann man nun die Reihenfolge der Rechenoperationen vertauschen:

```
range = mHigh - mLow + 1;
mHigh = mLow + ( high_count * range ) / total;
mLow = mLow + ( low_count * range ) / total;
```

Im Beispiel ergäbe das nun für `mHigh` $(4*7)/4 = 28/4 = 7$, d. h. man kann das ganze Intervall nutzen. Dadurch, dass aber jetzt erst eine Multiplikation erfolgt, kann es theoretisch wieder zu Überläufen kommen, weshalb man den möglichen Bereich weiter einschränken muss. Gesetzt den Fall, der Wertebereich ginge von 0 bis 15 (4 Bit), so würde nach der alternativen Methode die erste Multiplikation $4 * 7 \equiv 12 \ (mod\ 16)$ [7] ergeben, was die

[6] Nach [1], Kap. 5.2.5, S. 118.

[7] $4 * 7 = 0100 * 0111 = 0010 * 1110 = 0001 * 1100 = 1100 = 12$

nachfolgende Rechnung unbrauchbar macht[8]. Bei der ersten Methode hingegen ergibt sich $7/4 \equiv 1 \ (mod\ 16)$ bzw. $1 * 4 \equiv 4 \ (mod\ 16)$, also das erwartete Ergebnis. Um in der Multiplikation nicht den Bereich eines 32-Bit Registers zu verlassen, dürfen die Faktoren zusammen nicht mehr als 32 Bits breit sein, da $\lceil ld(a*b+1) \rceil \leq \lceil ld(a+1) \rceil + \lceil ld(b+1) \rceil$.

Weil `total` nach wie vor nicht größer als das minimal zur Verfügung stehende Intervall, d. h. ein Viertel des maximalen Bereichs sein darf, folgt:

$$\begin{aligned} ld(total) &\overset{!}{\leq} ld(range) - 2 \ , \\ ld(total) + ld(range) &\overset{!}{\leq} ld(register) \ . \end{aligned}$$

Das heißt, dass in der Praxis somit nur noch ein Bereich von maximal 17 Bit Breite bzw. 15 Bit für `total` zur Verfügung steht.

Da so also eine geringere Präzision zur Verfügung steht und zum anderen pro Schritt zwei Divisionen statt einer erforderlich sind, ist diese Alternative in den meisten Fällen weniger effizient.

7 Zusammenfassung Kodierung / Dekodierung

En- und Decoder lassen sich zu einer Klasse zusammenfassen. Dabei sind nach außen nur die Methoden sichtbar, die für die Interaktion mit dem Arithmetischen Kodierer benötigt werden. Die statischen Variablen lassen sich einfach als Member-Variablen der Klasse verwalten.

7.1 Kodierung

Für den Encoder ergibt sich folgendes Interface:

```
void Encode( const unsigned int low_count,
             const unsigned int high_count,
             const unsigned int total );

void EncodeFinish();
```

`EncodeFinish()` sorgt dabei dafür, dass die Ausgabesequenz korrekt abgeschlossen wird. Dazu ist zuerst nötig, dass die folgenden Bits eindeutig eine Zahl innerhalb des Endintervalls festlegen. Da wir wissen, dass das Intervall ständig mindestens ein Viertel des gesamten Bereichs umfasst, reicht es, wenn wir als Wert die untere Grenze dieses Viertels übergeben. Wir unterscheiden zwei Fälle:

1. Zweites Viertel

   $$mLow < FirstQuarter < Half \leq mHigh \ .$$

   Um diesen Fall zu kodieren, reicht es, zuerst eine 0, gefolgt von einer 1, zu speichern, das heißt, zuerst die untere Hälfte auszuwählen und von dieser dann die obere. Da

[8] Im Gegensatz zum Leser erkennt der Computer hier nicht, daß er die beiden 4-en kürzen könnte, da es sich reine Ganzzahl-Arithmetik handelt und nicht mit Brüchen gearbeitet wird. Jeder Rechenschritt wird separat betrachtet und als Grundlage der folgenden Schritte verwendet.

der Decoder dem Lesepuffer automatisch Nullen hinzufügt, wenn der Eingabestrom zu Ende ist, erhalten wir so die untere Grenze des zweiten Viertels.

Um eine eventuell noch vorhandene E3-Skalierung abzubauen, müssen analog zum Verfahren bei einer E2-Skalierung jetzt noch `mScale` Bits mit dem Wert 1 folgen. Das lässt sich nun mit der vorherigen 1 zu einer Schleife über `mScale+1` zusammenfassen.

2. Drittes Viertel

$$mLow < Half < ThirdQuarter \leq mHigh \ .$$

Der zweite Fall ist noch einfacher zu kodieren: Theoretisch muss man erst eine 1, gefolgt von `mScale+1` Nullen, speichern. Da der Decoder aber letztere selbst generiert, wenn die Sequenz abbricht, bedarf es nur der 1, um die Sequenz korrekt zu terminieren, weshalb hier auch keine Schleife für die E3-Skalierung verwendet wird.

```
if( mLow < g_FirstQuarter ) // mLow < FirstQuarter < Half <= mHigh
{
    SetBit( 0 );

    for( int i=0; i<mScale+1; i++ ) // 1 + e3-Skalierung abbauen
        SetBit(1);
}
else // mLow < Half < ThirdQuarter <= mHigh
{
    SetBit( 1 ); // der Decoder fügt die Nullen automatisch an
}
```

7.2 Dekodierung

Für den Decoder sind die folgenden drei Methoden notwendig:

```
void DecodeStart();

unsigned int DecodeTarget( const unsigned int total );

void Decode( const unsigned int low_count,
             const unsigned int high_count );
```

`DecodeStart()` dient dazu, den Puffer zu initialisieren, das heißt, die ersten Bits aus der Eingabesequenz zu lesen.

```
for( int i=0; i<31; i++ ) // benutze nur die unteren 31 bit
    mBuffer = ( mBuffer << 1 ) | GetBit();
```

Weitere zusätzliche Funktionen sind für den Decoder nicht notwendig. Damit haben wir unsere Betrachtung des Kodier- bzw. Dekodiervorgangs abgeschlossen.

Wir haben gesehen, dass mittels der Skalierungsfunktionen $E1$ bis $E3$ ein Überlauf der Integer-Arithmetik erfolgreich verhindert werden kann und dass als positiver Nebeneffekt bereits feststehende Ausgabebits schon vor Abschluss der Kodierung dem Ausgabestrom hinzugefügt werden können. Dadurch erhalten wir ein sequenziell arbeitendes Verfahren, was z. B. für die Datenfernübertragung essenziell ist. Da der Decoder auch immer nur so viele Bits berücksichtigt, wie in ein Register passen, reichen in unserem Beispiel schon 31 Bit der kodierten Sequenz, um das erste Symbol auf jeden Fall dekodieren zu können.

Zu beachten ist, daß bedingt dadurch, daß man einen Code für die gesamte Eingabesequenz hat, Fehler in der Übertragung dazu führen, daß ab dort der gesamte Code unbrauchbar wird. Um dem entgegenzuwirken kann man nun abschnittsweise kodieren oder ein Übertragungsverfahren mit Fehlerkorrektur einsetzen bzw. Kombinationen daraus.

7.3 Bitsequenz-Ende

Es gibt verschiedene Ansätze dazu, den Decoder terminieren zu lassen, da die Bitsequenz selbst ja kein Ende impliziert.

Die einfachste Variante ist, getrennt von der Bitsequenz (z. B. als Datei-Header) die Anzahl der kodierten Symbole zu speichern, so dass En- und Decoder durch FOR-Schleifen gesteuert werden können. Der Nachteil dieser Technik ist aber, dass für den Encoder entweder schon zu Anfang feststehen muss, wie lang die Eingabesequenz ist, oder aber dass die Möglichkeit besteht, nachträglich diesen Header zu schreiben. Bei der Fax-Übertragung ist aber beides unmöglich, weshalb man auf End-Symbole zurückgreift.

Man reserviert im Modell ein eigenes Symbol mit minimaler Wahrscheinlichkeit, was nur dazu dient, das Ende der Sequenz zu kennzeichnen. Sobald das Ende der Eingabesequenz erreicht ist, übergibt man dieses Symbol dem Encoder, welcher es wie jedes andere Symbol kodiert. Sobald das Modell beim Dekodieren dieses Symbol erhält, signalisiert es das Ende der Sequenz und der Vorgang kann terminieren.

Somit haben wir nun zwei in der Implementierung recht verschiedene Verfahren kennengelernt, einen arithmetischen Code zu generieren. Wir haben für beide Verfahren gezeigt, dass diese einen eindeutigen Code generieren, der ebenso eindeutig dekodiert werden kann. Im folgenden Kapitel wollen wir uns nun der Effizienz der Arithmetischen Kodierung widmen und einen Vergleich zur Huffman-Kodierung aufstellen.

8 Effizienz des Verfahrens

8.1 Effizienzbetrachtung

Wir haben in Kapitel 3.6 gezeigt, dass die Länge $l(x)$ nötig ist, um eine Sequenz x mit hinreichender Genauigkeit darzustellen. Hieraus leitet sich nun die *durchschnittliche* Länge eines Arithmethischen Codes für eine Sequenz $S^{(m)}$ der Länge m ab als

$$\begin{aligned} l_{A^{(m)}} &= \sum_x P_M(x)\ l(x) && (10)\\ &= \sum_x P_M(x) \left[\lceil\ ld\ \frac{1}{P_M(x)} \rceil + 1 \right] && (11)\\ &\leq \sum_x P_M(x) \left[\ ld\ \frac{1}{P_M(x)} + 1 + 1 \right] && (12)\\ &= -\sum_x P_M(x)\ \ ld\ P_M(x) + 2 \sum_x P_M(x) && (13)\\ &= H_M(S^{(m)}) + 2. && (14) \end{aligned}$$

Da, wie wir schon wissen, die durchschnittliche Länge aber immer größer oder gleich der Entropie ist, folgt

$$H_M(S^{(m)}) \leq l_{A^{(m)}} \leq H_M(S^{(m)}) + 2. \tag{15}$$

Die durchschnittliche Länge pro Symbol l_A, die wir auch als *Kompressionrate* des Arithmetischen Codes bezeichnen, ist $l_A = \frac{l_{A^{(m)}}}{m}$. Daher ergeben sich für l_A die Grenzen

$$\frac{H_M(S^{(m)})}{m} \leq l_A \leq \frac{H_M(S^{(m)})}{m} + \frac{2}{m}. \tag{16}$$

Nun wissen wir außerdem, dass sich die Entropie der Sequenz aus deren Länge mal der Entropie jedes einzelnen Symbols ergibt:[9]

$$H_M(S^{(m)}) = m \cdot H_M(x) \tag{17}$$

Dies liefert uns für l_A die Grenzen

$$H_M(x) \leq l_A \leq H_M(x) + \frac{2}{m}. \tag{18}$$

Man sieht also auf einen Blick, dass man mit steigender Länge der Sequenz garantiert eine Kompressionsrate nahe der vom Modell M bestimmten Entropie bekommt, was ja unsere Absicht war.

[9] Beweis in [2] S.50

8.2 Vergleich zur Huffman Kodierung

Nachdem wir im letzten Abschnitt die Effizienz der Arithmetischen Kodierung gezeigt haben, wollen wir nun einen Vergleich zum bekannten Huffman Code aufstellen. Erinnern wir uns an das Beispiel 3.6.1. Die durchschnittliche Codelänge berechnet sich als

$$\begin{aligned} l &= 0,5 \cdot 2 + 0,25 \cdot 3 + 0,125 \cdot 4 + 0,125 \cdot 4 \\ &= 2,75 \ [Bits/Symbol] \ . \end{aligned}$$

Die Entropie dieser Sequenz hingegen ergibt sich folgendermaßen:

$$\begin{aligned} H_M(x) &= \sum_{i=1}^{4} P(a_i) \ \ ld \ \frac{1}{P_M(a_i)} \\ H_M(x) &= -\left(\sum_{i=1}^{4} P(a_i) \ \ ld \ P_M(a_i) \right) \\ &= -\left(\frac{1}{4} \cdot \ ld \ \frac{1}{2} + \frac{1}{4} \cdot \ ld \ \frac{1}{4} + \frac{1}{4} \cdot \ ld \ \frac{1}{8} + \frac{1}{4} \cdot \ ld \ \frac{1}{8} \right) \\ &= -\left(\frac{1}{4} \cdot (-1) + \frac{1}{4} \cdot (-2) + \frac{1}{4} \cdot (-3) + \frac{1}{4} \cdot (-3) \right) \\ &= 2,25. \end{aligned}$$

Man sieht also, dass die Codelänge der symbolweisen Arithmetischen Kodierung hier der Entropie noch nicht sehr nahe kommt. Würde man diese Sequenz jedoch mit dem Huffman Code kodieren, so würde dieser die Entropie erreichen. Dies liegt daran, dass der Huffman Code genau dann ideal arbeitet, wenn für die Wahrscheinlichkeiten ganze Bits vergeben werden können. Dies ist hier ganz offensichtlich der Fall, da alle Wahrscheinlichkeiten als Potenzen von 2 gewählt wurden. Genau das ist jedoch in der Praxis utopisch, wird aber trotzdem gerne als Argument für Huffman verwendet.[10] Außderdem bleibt anzumerken, daß auch die Arithmetische Kodierungen in diesen Fällen nicht weniger effizient ist. Sie kann nur nicht besser sein als Huffman, da dieser die untere Schranke voll erreicht. Außerdem wird bei der Effizienzbetrachtung aller Verfahren oft davon ausgegangen, daß die Abfolge der Symbole kontextunabhängig ist. Auch dies wird bei realen Datenquellen so gut wie nie wirklich der Fall sein. Jedoch liefert diese Annahme oftmals einfachere Gleichungen, die sich nicht allzu sehr von der Realität unterscheiden. Gleichung (17) setzt dies u.a. voraus. Man sieht hier leicht, daß man anstatt mit Ungleichung (18) auch mit Ungleichung (16) arbeiten könnte, welche aber nur unnötig kompliziert ist und sich bis auf einen zu vernachlässigenden Faktor nicht von der nachfolgenden unterscheidet. Man kann zeigen, dass die Effizienz des Huffman Codes wie folgt beschränkt ist[11]:

$$H_M(S) \leq l_S \leq H_M(S) + 1 \ . \tag{19}$$

Für *Extended Huffman*, eine spezielle Variante der Huffman Kodierung, bei der b Symbole zu längeren zusammengefasst werden, ergibt sich die Effizienz

$$H_M(S) \leq l_S \leq H_M(S) + \frac{1}{b} \ . \tag{20}$$

[10] Siehe auch [2] Kap. 4.5.
[11] Siehe auch [2] Kap. 3.2.3.

Dieses Verfahren ist für nicht-utopische Sequenzen ($\exists x \in S : P(x) \neq 2^n \ \forall n \in \mathbb{N}$) effizienter. Läßt man nun b gegen m laufen, und vergleicht man dies mit Gleichung (18), so scheint die Huffman-Kodierung gegenüber der Arithmetischen Kodierung im Vorteil, wenngleich sich dieser Vorteil auch mit ansteigender Sequenzlänge m verkleinert. Dabei darf man aber nicht aus den Augen verlieren, daß man das o. g. b nicht beliebig groß wählen kann. Hat man ein Alphabet der Länge k und gruppiert b Symbole zusammen, so ergibt sich eine Größe des Codebooks von k^b. Für plausible Werte von $k = 16$ und $b = 20$ ergibt dies schon 16^{20}. Dies würde jeden momentan bekannten Arbeitsspeicher überfordern. Daher ist b also physikalisch nach oben beschränkt, während die Sequenzlänge m für große Sequenzen umso größer wird. Daher liegt auch hier der Vorteil in der Praxis bei der Arithmetischen Kodierung.

Ein weiterer möglicher Vorteil der Arithmetischen Kodierung ist abhängig von der Quelle. Man kann zeigen[12], dass die Huffman Kodierung für eine Sequenz S eine Kompressionsrate von $(0,086 + P_{max}) \cdot H_M(S)$ nie überschreitet, wobei P_{max} die größte aller vorkommenden Wahrscheinlichkeiten ist. Nun ist klar, dass für große Alpabete dieses P_{max} relativ kleine Werte annimmt und die Huffman Kodierung somit an Effizienz gewinnt. Daher hat sie für große Alphabete schon ihre Berechtigung. Im Vergleich dazu ist also die Arithmetische Kodierung für kleine Alphabete, in denen folglich ausschließlich relativ große Wahrscheinlichkeiten vorkommen, im Vorteil. Anwendungen hierfür sind zum Beispiel die Kompressionsstandards *G3* und *G4*, die für den Faxversand verwendet werden. Hier ist das Alphabet binär, hat also Mächtigkeit 2, und die Wahrscheinlichkeiten für schwarze Pixel sind im Normalfall äußerst gering. Daraus resultiert ein P_{max} nahe bei 1, was die Huffman Kodierung hier disqualifiziert und die Arithmetische Kodierung zum Mittel der Wahl macht. Wenn man praktische Ergebnisse [15] zu Rate zieht, zeigt sich, dass die Arithmetische Kodierung gegenüber der Huffman-Kodierung für die meisten realen Quellen einen leichten Vorteil für sich verbuchen kann. Dies liegt daran, dass Huffman nur in dem utopischen Fall wirklich optimal arbeitet, dass alle Symbolwahrscheinlichkeiten Zweierpotenzen sind. Denn dann hat der entsprechende Codebaum minimale Tiefe. Da dies in der Realität eigentlich nie der Fall ist, muss der Huffman-Kodierer notgedrungen bei entsprechenden Symbolen ganze Bitanzahlen mehr verwenden, während die Arithmetische Kodierung, wie wir gesehen haben, auch mit Bruchteilen von Bits kodieren kann.

Ein zusätzlicher Vorteil der Arithmetischen Kodierung, auf den wir hier nicht näher eingehen möchten, ist die leichte Anpassbarkeit an diverse Wahrscheinlichkeitsmodelle. Wie wir in den vorangegangenen Kapiteln gesehen haben, genügt es hier, für verschiedene Quellen jeweils optimierte Modelle aufzusetzen. Der prinzipielle Vorgang des Kodierens bleibt hiervon völlig unberührt. Dies erlaubt eine einfache Implementierung diverser optimierter Codecs. Dieser Vorteil wird vor allem bei sogenannten *adaptiven* Modellen klar, deren Implementierung bei Huffman zu komplexen Umstrukturierungen der Baumstrukturen nötigt.

Auf solche adapativen Modelle möchten wir im folgenden Kapitel näher eingehen, da sie in der Anwendung große Vorteile gegenüber statischen Modellen liefern.

[12] [2] S.37f

9 Alternative Modelle

In den vorangegangenen Kapiteln verwendeten wir die kumulierende Funktion

$$K(a_k) = \sum_{i=1}^{k} P_M(a_i),$$

um das Symbol a_k, das k-te des Alphabets A zu kodieren. Die Wahrscheinlichkeiten $P_M(a_i)$, $(i = 1, \ldots, |A|)$ werden dazu vom Modell M übergeben[13]. Bisher haben wir jedoch vernachlässigt, ob dieses Modell die Wahrscheinlichkeiten eines Zeichens a_k in einer Sequenz S überhaupt genau festzustellen vermag. Und wenn dies möglich ist, wie funktioniert es? Diesen Fragen wollen wir im Folgenden nachgehen.

Zunächst möchten wir anmerken, dass die Entropie $H_M(S)$ per Definition vom Modell M abhängt. Also gilt, dass egal wie gut oder schlecht unser Modell ist, die Arithmetische Kodierung diese Entropie stets nahezu erreicht. Jedoch ist es mit geeigneten Modellen möglich, diese Entropie und somit auch die Schranken aus Gleichung (18) noch zu senken.

9.1 Order-n Modelle

Bislang wurden die Wahrscheinlichkeiten der Symbole stehts als unabhängig betrachtet. Oft ist es jedoch so, daß sich die Wahrscheinlichkeit in Abhängigkeit des Kontexts ändert. So ist die Wahrscheinlichkeit für ein 'u' im Deutschen etwa 4,35%. War jedoch der letzte Buchstabe ein 'q', so geht die Wahrscheinlichkeit gegen 100%. Modelle, die den Kontext eines Symbols berücksichtigen heissen ORDER-N MODELLE. Das 'n' gibt dabei die Größe des Fensters an, d.h. ein Order-3-Modell berücksichtigt bei der Bestimmung der Wahrscheinlichkeit für ein Symbol die vorhergehenden 3 Symbole.

9.2 Adaptive Modelle

Da im Allgemeinen eine Implementierung zu dem Zweck entwickelt wird, *verschiedene* Sequenzen zu kodieren, kann man also nicht davon ausgehen, dass die Wahrscheinlichkeiten schon zu Beginn der Kodierung bekannt sind. Selbst ein einfaches *Abzählen* der Häufigkeiten des Vorkommens eines bestimmten Zeichens ist nicht immer möglich. Nehmen wir zum Beispiel erneut den Faxversand: Mit dem Versand des Faxes soll nach Möglichkeit schon begonnen werden, auch wenn die Seite noch nicht ganz gelesen wurde und somit die genauen Wahrscheinlichkeiten noch nicht feststehen. Was bleibt, sind also Näherungen.

Dabei liegt es auf der Hand, dass diese Näherungen mit laufender Kodierung der Sequenz je nach Häufigkeit der letzten gelesenen Symbole angepasst werden müssen, weshalb man von einem ADAPTIVEN MODELL spricht. Betrachten wir hierzu ein einfaches Beispiel.

9.2.1 Beispiel

Wir wählen zur einfachen Veranschaulichung ein *adaptives Order-0-Modell*.

Order-0 bedeutet hier, dass das Modell stets die Wahrscheinlichkeit des *aktuellen* Zeichens zurückliefert. Dazu genügt es einfach, zu Beginn der Kodierung ein Array K zu definieren, das genauso lang ist, wie das Alphabet Σ an Zeichen hat. Die Werte dieses Arrays sind zunächst mit 0 initialisiert. Vor jedem Kodierungsschritt erhält nun das Modell ein Zeichen s vom Eingabestrom und erhöht den entsprechenden Array-Eintrag sowie einen

[13] Je nach Implementierung übergibt das Modell auch direkt die Grenze $K(a_k)$.

s	K[A]	K[B]	K[C]	K[D]	z	$P_M^{(z)}(A)$	$P_M^{(z)}(B)$	$P_M^{(z)}(C)$	$P_M^{(z)}(D)$
A	1	0	0	0	1	1	0	0	0
B	1	1	0	0	2	1/2	1/2	0	0
A	2	1	0	0	3	2/3	1/3	0	0
D	2	1	0	1	4	1/2	1/4	0	1/4

Tabelle 7: Arbeitsweise eines adaptiven Order-0 Modells

absoluten Zeichenzähler z. Danach werden die Wahrscheinlichkeiten neu verteilt nach der Formel

$$P_M^{(z)}(s) = \frac{K[s]}{z} \ .$$

Wählen wir zum Beispiel das Alphabet

$$\Sigma = A, B, C, D$$

und kodieren die Sequenz $ABAD$. Die sich ergebenden Daten können in Tabelle 7 nachvollzogen werden. Es fällt auf, dass die vergebenen Wahrscheinlichkeiten nach dem Lesen des letzten Zeichens tatsächlich den echten Wahrscheinlichkeiten der Zeichen in der Sequenz entsprechen. Man kann also leicht sehen, dass sich für lange Sequenzen die berechnete der natürlichen Wahrscheinlichkeit annähert.

Da die Initialisierung des Modells bekannt ist und die Aktualisierung des Modells erst nach dem Kodieren eines jeden Symbols statt findet, kannd er Decoder diese problemlos nachvollziehen, da er das Modell anpaßt, sobald er das Zeichen dekodiert hat. Somit ergibt sich auch eine Einsparung dadurch, daß man die Wahrscheinlichkeiten nicht mitübertragen muß sondern direkt aus dem Datenstrom generieren kann.

9.3 Weitere Modelle

Bei bestimmten Anwendungen (z. B. Mischdateien, die aus stark unterschiedlichen Datenklassen bestehen) hingegen kann es sinnvoll sein, Sprünge in der Wahrscheinlichkeitsverteilung aufzuspüren und beim Feststellen solcher Sprünge die gespeicherten (und nun veralteten) Wahrscheinlichkeiten zu verwerfen. Dadurch wird für den folgenden Block eine bessere Kompression erreicht. Viele der modernen adaptiven Modelle berücksichtigen dies durch einfaches Zurücksetzen des Arrays K.

Natürlich kann man sich eine Vielzahl weiterer Modelle denken, die je nach Verwendung der Kodierung bestimmte Vorteile bringen können. Diese möchten wir jedoch hier nicht näher beschreiben, da sie allesamt von der eigentlichen Arithmetischen Kodierung unabhängig sind und es bereits genügend Literatur (z.B. [14]) zu entsprechenden Modellen gibt.

10 Zusammenfassung und Ausblick

Nach all diesen Überlegungen möchten wir an dieser Stelle noch einmal überprüfen, ob wir die anfangs gesetzten Ziele erreicht haben. Wir haben mit der Arithmetischen Kodierung ein Kodierungsverfahren beschrieben, welches sich zur Datenkompression eignet.

Dazu haben wir gezeigt, dass die Voraussetzungen für die Implementierung einer eineindeutigen Kodierung mit heutigen Rechnersystemen gegeben sind. Dabei stehen uns sowohl Gleitkomma- als auch Ganzzahl-Verfahren (Integer) zur Verfügung. Wir haben gesehen, dass die Arithmetische Kodierung sequentiell arbeiten kann, also zeichenweise kodiert und somit bereits kodierte Abschnitte senden bzw. spreichern kann, auch wenn der Kodierungsvorgang noch fortgesetzt wird. Dies nutzten wir dazu aus, mittels dreier Skalierungsfunktionen den betrachteten Bereich wieder derart zu vergrößern, dass eine endlich genaue Rechnerarithmetik zur Kodierung ausreicht. Im weiteren Verlauf zeigten wir die Grenzen der Effizienz der Kodierung und bemerkten, dass sich die durchschnittliche Symbollänge des arithmetischen Codes mit steigender Länge der zu kodierenden Sequenz immer stärker der sequenz- und modellabhängigen Entropie nähert. Hierzu zeigten wir außerdem, unter welchen Umständen die Kodierung besonders effizient ist und sogar die Effizienz der bekannten Huffman-Kodierung übersteigt. Dazu merkten wir letztlich an, dass die Kompression, die durch das Kodieren der Sequenz erreicht wird, in Ihrer Effizienz durch die Qualität des Modells begrenzt ist. Es fiel auf, dass auch hier die Arithmetische Kodierung den Vorteil bildet, dass verschiedene Modelle modular mit dem gleichen Kodierungsalgorithmus arbeiten können.

Somit stellen wir fest, dass alle unsere zu Beginn gesetzten Ziele erreicht wurden. Daher möchten wir zum Abschluß dieses Artikels nur noch kurz auf mögliche Verbesserungsverfahren und aktuelle, praxisbezogene Lösungen eingehen.

10.1 Kompression ist kein Allheilmittel

Obwohl sich die Arithmetische Kodierung gerade in den letzten 10 bis 20 Jahren immer weiter etabliert hat und somit immer weiter perfektioniert wurde, treten hin und wieder noch neue Varianten dieses mittlerweile sehr beliebten Verfahrens auf. Für den interessierten Leser mag z. B. die Newsgroup *comp.compression* ein guter Anlaufpunkt sein, um sich über neue Techniken und weitere Hintergründe zu informieren. Doch hier ist Vorsicht geboten, denn allzu oft berichten übereifrige Leute davon, eine neue, überragende Kompressionsmethode gefunden zu haben, die alles bisher Dagewesene in den Schatten stellt. Sicherlich gibt es mancherorts noch Optimierungsmöglichkeiten, jedoch erweisen sich die meisten solcher Meldungen leider als maßlos übertrieben. Wir wissen sicher durch das Shannon-Theorem [11], dass es keine Kompression geben kann, die die Entropie der Quelle noch unterbietet. Damit können wir die Entropie als untere Grenze der Länge eines Codes ansetzen, die nicht durchbrochen werden kann. Wir wollen damit sagen, dass Datenkompression, und so auch die Arithmetische Kodierung, keine Wunder bewirken kann. Sie vermag nicht mehr als Redundanz aus zu Datenquellen entfernen. Ist keine Redundanz mehr vorhanden, dann hilft auch die beste Umkodierung nicht weiter, sofern sie verlustlos bleiben soll.

10.2 Methoden der Optimierung

Wir können also nur versuchen, die vorhandenen Algorithmen auf verschiedene Arten zu optimieren. In der Datenkompression betrifft das vor allem zwei Bereiche: Speicherbedarf und Geschwindigkeit.

10.2.1 Geringerer Speicherbedarf

Wie wir nun aus den obigen Ausführungen wissen, ist in Bezug auf den Speicherbedarf die arithmetische Kodierung als Verfahren an sich schon fast optimal. Zum einen braucht sie nur konstanten Arbeitsspeicher (im Wesentlichen die Tabelle der Wahrscheinlichkeitsverteilungen) und generiert einen Code, der an sich nicht besser zu komprimieren ist. Dabei dürfen wir jedoch ein kleines Detail nicht vergessen: $H(S) \leq H_M(S) \leq |Code(S)|$. Das soll heißen: Wir müssen unterscheiden zwischen der natürlichen Entropie $H(x)$ der Quell-Sequenz, die die mathematische Grenze darstellt, und der Grenze, die uns vom Modell gesetzt wird; wir hatten Sie mit $H_M(S)$ benannt. Die Arithmetische Kodierung als Implementierung erreicht daher $H_M(S)$ stets voll und ist somit optimal, sie kann jedoch $H(S)$ nur unter der Prämisse eines optimalen Modells erreichen.

Da die zu kodierenden Datenquellen sehr unterschiedlich sind, ist es sehr schwierig, ein Modell auf die breite Masse von Datenquellen hin zu optimieren. Auch hier bietet jedoch die Arithmetische Kodierung den Vorteil, dass ein beliebiges, auf spezielle Quellen hin optimiertes Modell modular aufgesetzt werden kann. Somit ist in den letzten Jahren eine Vielzahl solcher Modelle entwickelt worden, die jeweils für den einen oder anderen Einsatzzweck mehr oder minder optimal sind. Ein vergleichsweise effektives Modell ist unter dem Namen *PPM (Prediction with partial match)* bekannt. PPM-Modelle haben sich bei den meisten Kodierungsverfahren als sehr effizient erwiesen, da sie die Kontextlänge variieren können. Wir wollen an dieser Stelle nicht näher darauf eingehen, da es schon viele umfangreiche Werke zum Thema PPM gibt. Links zu diesem und weiteren Themen rund um die Datenkompression liefert die Seite [14]. Widmen wir uns nun dem zweiten Optimierungsansatz, der Geschwindigkeit.

10.2.2 Schnelle Verarbeitung

Auch in Bezug auf die Kodier- bzw. Dekodiergeschwindigkeit sind in den letzten Jahren einige Optimierungen vorgenommen worden. Hier stehen sich natürlich grundsätzlich verschiedene Verarbeitungsgeschwindigkeiten der Integer- und Float-Arithmetik gegenüber. In den letzten Jahren waren die Integer-Operationen der gängigsten CPUs stets schneller als die entsprechenden für Floats. Daher sind auch die meisten Kodierer über Integers implementiert worden. Mit neuen Chips wie z. B. dem *Itanium* von Intel sind jedoch die Float-Operationen mit den Jahren schneller und schneller geworden, was wiederum zu neuen Implementierungen führen wird. Dass Implementierungen über Floats überhaupt möglich sind, haben wir in Kapitel 3.6 bewiesen. Eine sehr effiziente Integer-Variante ist der so genannte *Range Coder* [12], [13]. Bei diesem Verfahren wird das Scaling mit Bytes statt Bits betrieben. Dies führt zu einer Geschwindigkeitssteigerung von bis zu 50 Prozent, je nach Implementierung und Rechner, bei ca. 0,01 Prozent größerem Code im Vergleich zu Standard-Implementierungen. Es gibt noch viele weitere modifizierte Verfahren. Einen Überblick hierüber liefert [14].

Man sieht also, dass aufgrund der ständigen Änderungen in Hardware und zu komprimierenden Daten oft noch weitere Möglichkeiten existieren, die Arithmetische Kodierung noch weiter zu perfektionieren.

A Eine Beispielimplementierung in C++

auch verfügbar unter: *http://www-users.rwth-aachen.de/eric.bodden/ac/*

Diese Implementierung soll dazu dienen, den kompletten Kodierungsvorgang in eindeutiger, algorithmischer Weise darzustellen und somit eventuell noch offene Fragen zu klären. Wir verwenden ein einfaches, adaptives Order-0 Modell, wie es in Kapitel 9.2 vorgestellt wurde. Daher ist die Kompressionsrate entsprechend gering. Durch Austausch des entsprechenden Moduls kann jedoch jederzeit ein passenderes Modell in diese Implementierung übernommen werden.

A.1 Arithmetischer Kodierer (Header)

```
#ifndef __ARITHMETICCODERC_H__
#define __ARITHMETICCODERC_H__

#include <fstream>
using namespace std;

class ArithmeticCoderC
{
public:
  ArithmeticCoderC();

  void SetFile( fstream *file );

  void Encode( const unsigned int low_count,
               const unsigned int high_count,
               const unsigned int total );
  void EncodeFinish();

  void DecodeStart();
  unsigned int DecodeTarget( const unsigned int total );
  void Decode( const unsigned int low_count,
               const unsigned int high_count );

protected:
  // bit operations
  void SetBit( const unsigned char bit );
  void SetBitFlush();
  unsigned char GetBit();

  unsigned char mBitBuffer;
  unsigned char mBitCount;

  // in-/output stream
  fstream *mFile;

  // encoder & decoder
```

```
  unsigned int mLow;
  unsigned int mHigh;
  unsigned int mStep;
  unsigned int mScale;

  // decoder
  unsigned int mBuffer;
};

#endif
```

A.2 Arithmetischer Kodierer

```
#include "ArithmeticCoderC.h"
#include "tools.h"

// Konstanten zur Bereichsunterteilung bei 32-Bit-Integern
// oberstes Bit wird zur Vermeidung von Überläufen freigehalten
const unsigned int g_FirstQuarter = 0x20000000;
const unsigned int g_ThirdQuarter = 0x60000000;
const unsigned int g_Half         = 0x40000000;

ArithmeticCoderC::ArithmeticCoderC()
{
  mBitCount = 0;
  mBitBuffer = 0;

  mLow = 0;
  mHigh = 0x7FFFFFFF; // arbeite nur mit den unteren 31 bit
  mScale = 0;

  mBuffer = 0;
  mStep = 0;
}

void ArithmeticCoderC::SetFile( fstream *file )
{
  mFile = file;
}

void ArithmeticCoderC::SetBit( const unsigned char bit )
{
  // Bit dem Puffer hinzufügen
  mBitBuffer = (mBitBuffer << 1) | bit;
  mBitCount++;

  if(mBitCount == 8) // Puffer voll
  {
```

```
    // schreiben
    mFile->write(reinterpret_cast<char*>(&mBitBuffer),sizeof(mBitBuffer));
    mBitCount = 0;
  }
}

void ArithmeticCoderC::SetBitFlush()
{
  // Puffer bis zur nächsten Byte-Grenze mit Nullen auffüllen
  while( mBitCount != 0 )
    SetBit( 0 );
}

unsigned char ArithmeticCoderC::GetBit()
{
  if(mBitCount == 0) // Puffer leer
  {
    if( !( mFile->eof() ) ) // Datei komplett eingelesen?
      mFile->read(reinterpret_cast<char*>(&mBitBuffer),sizeof(mBitBuffer));
    else
      mBitBuffer = 0; // Nullen anhängen

    mBitCount = 8;
  }

  // Bit aus Puffer extrahieren
  unsigned char bit = mBitBuffer >> 7;
  mBitBuffer <<= 1;
  mBitCount--;

  return bit;
}

void ArithmeticCoderC::Encode( const unsigned int low_count,
                               const unsigned int high_count,
                               const unsigned int total )
// total < 2^29
{
  // Bereich in Schritte unterteilen
  mStep = ( mHigh - mLow + 1 ) / total; // oben offenes intervall => +1

  // obere Grenze aktualisieren
  mHigh = mLow + mStep * high_count - 1; // oben offenes intervall => -1

  // untere Grenze aktualisieren
  mLow = mLow + mStep * low_count;

  // e1/e2 Mapping durchführen
```

```
  while( ( mHigh < g_Half ) || ( mLow >= g_Half ) )
    {
      if( mHigh < g_Half )
      {
        SetBit( 0 );
        mLow = mLow * 2;
        mHigh = mHigh * 2 + 1;

        // e3
        for(; mScale > 0; mScale-- )
          SetBit( 1 );
      }
      else if( mLow >= g_Half )
      {
        SetBit( 1 );
        mLow = 2 * ( mLow - g_Half );
        mHigh = 2 * ( mHigh - g_Half ) + 1;

        // e3
        for(; mScale > 0; mScale-- )
          SetBit( 0 );
      }
    }

  // e3
  while( ( g_FirstQuarter <= mLow ) && ( mHigh < g_ThirdQuarter ) )
  {
    mScale++;
    mLow = 2 * ( mLow - g_FirstQuarter );
    mHigh = 2 * ( mHigh - g_FirstQuarter ) + 1;
  }
}

void ArithmeticCoderC::EncodeFinish()
{
  // Es gibt zwei Möglichkeiten, wie mLow und mHigh liegen, d.h.
  // zwei Bits reichen zur Entscheidung aus.

  if( mLow < g_FirstQuarter ) // mLow < FirstQuarter < Half <= mHigh
  {
    SetBit( 0 );

    for( int i=0; i<mScale+1; i++ ) // 1 + e3-Skalierung abbauen
      SetBit(1);
  }
  else // mLow < Half < ThirdQuarter <= mHigh
  {
    SetBit( 1 ); // der Decoder fügt die Nullen automatisch an
```

```
  }

  // Ausgabepuffer leeren
  SetBitFlush();
}

void ArithmeticCoderC::DecodeStart()
{
  // Puffer mit Bits aus dem Eingabe-Code füllen
  for( int i=0; i<31; i++ ) // benutze nur die unteren 31 bit
    mBuffer = ( mBuffer << 1 ) | GetBit();
}

unsigned int ArithmeticCoderC::DecodeTarget( const unsigned int total )
// total < 2^29
{
  // Bereich in Schritte unterteilen
  mStep = ( mHigh - mLow + 1 ) / total; // oben offenes intervall => +1

  // aktuellen Wert zurückgeben
  return ( mBuffer - mLow ) / mStep;
}

void ArithmeticCoderC::Decode( const unsigned int low_count,
                               const unsigned int high_count )
{
  // obere Grenze aktualisieren
  mHigh = mLow + mStep * high_count - 1; // oben offenes intervall => -1

  // untere Grenze aktualisieren
  mLow = mLow + mStep * low_count;

  // e1/e2
  while( ( mHigh < g_Half ) || ( mLow >= g_Half ) )
    {
      if( mHigh < g_Half )
      {
        mLow = mLow * 2;
        mHigh = mHigh * 2 + 1;
        mBuffer = 2 * mBuffer + GetBit();
      }
      else if( mLow >= g_Half )
      {
        mLow = 2 * ( mLow - g_Half );
        mHigh = 2 * ( mHigh - g_Half ) + 1;
        mBuffer = 2 * ( mBuffer - g_Half ) + GetBit();
      }
      mScale = 0;
```

```
    }

  // e3
  while( ( g_FirstQuarter <= mLow ) && ( mHigh < g_ThirdQuarter ) )
  {
    mScale++;
    mLow = 2 * ( mLow - g_FirstQuarter );
    mHigh = 2 * ( mHigh - g_FirstQuarter ) + 1;
    mBuffer = 2 * ( mBuffer - g_FirstQuarter ) + GetBit();
  }
}
```

A.3 Modell-Basisklasse (Header)

```
#ifndef __MODELI_H__
#define __MODELI_H__

#include "ArithmeticCoderC.h"

enum ModeE
{
  MODE_ENCODE = 0,
  MODE_DECODE
};

class ModelI
{
public:
  void Process( fstream *source, fstream *target, ModeE mode );

protected:
  virtual void Encode() = 0;
  virtual void Decode() = 0;

  ArithmeticCoderC mAC;
  fstream *mSource;
  fstream *mTarget;
};

#endif
```

A.4 Modell-Basisklasse

```
#include "ModelI.h"

void ModelI::Process( fstream *source, fstream *target, ModeE mode )
{
  mSource = source;
  mTarget = target;
```

```
  if( mode == MODE_ENCODE )
  {
    mAC.SetFile( mTarget );

    // kodieren
    Encode();

    mAC.EncodeFinish();
  }
  else // MODE_DECODE
  {
    mAC.SetFile( mSource );

    mAC.DecodeStart();

    // dekodieren
    Decode();
  }
};
```

A.5 Modell-Order-0 (Header)

```
#ifndef __MODELORDER0C_H__
#define __MODELORDER0C_H__

#include "ModelI.h"

class ModelOrder0C : public ModelI
{
public:
  ModelOrder0C();

protected:
  void Encode();
  void Decode();

  unsigned int mCumCount[ 257 ];
  unsigned int mTotal;
};

#endif
```

A.6 Modell-Order-0

```
#include "ModelOrder0C.h"

ModelOrder0C::ModelOrder0C()
{
```

```
  // Häufigkeiten mit 1 initialisieren
  mTotal = 257; // 256 + Endsymbol
  for( unsigned int i=0; i<257; i++ )
    mCumCount[i] = 1;
}

void ModelOrder0C::Encode()
{

  while( !mSource->eof() )
  {
    unsigned char symbol;

    // Symbol lesen
    mSource->read( reinterpret_cast<char*>(&symbol), sizeof( symbol ) );

    if( !mSource->eof() )
    {
      // Häufigkeiten kumulieren
      unsigned int low_count = 0;
      for( unsigned char j=0; j<symbol; j++ )
        low_count += mCumCount[j];

      // Symbol kodieren
      mAC.Encode( low_count, low_count + mCumCount[j], mTotal );

      // update model
      mCumCount[ symbol ]++;
      mTotal++;
    }
  }

  // End-Symbol schreiben
  mAC.Encode( mTotal-1, mTotal, mTotal );
}

void ModelOrder0C::Decode()
{
  unsigned int symbol;

  do
  {
    unsigned int value;

    // Wert lesen
    value = mAC.DecodeTarget( mTotal );

    unsigned int low_count = 0;
```

```
    // Symbol bestimmen
    for( symbol=0; low_count + mCumCount[symbol] <= value; symbol++ )
      low_count += mCumCount[symbol];

    // Symbol schreiben
    if( symbol < 256 )
      mTarget->write( reinterpret_cast<char*>(&symbol), sizeof( char ) );

    // Dekoder anpassen
    mAC.Decode( low_count, low_count + mCumCount[ symbol ] );

    // update model
    mCumCount[ symbol ]++;
    mTotal++;
  }
  while( symbol != 256 );
}
```

A.7 Tools

```
#ifndef __TOOLS_H__
#define __TOOLS_H__

int inline min( int a, int b )
{
  return a<b?a:b;
};

#endif
```

A.8 Main

```
#include <iostream>
#include <fstream>
using namespace std;

#include "ModelOrder0C.h"

// signature: "ACMC" (0x434D4341, intel byte order)
const int g_Signature = 0x434D4341;

int __cdecl main(int argc, char *argv[])
{
  cout << "Arithmetische Codierung" << endl;

  if( argc != 3 )
  {
    cout << "Syntax: AC source target" << endl;
```

```
    return 1;
  }

  fstream source, target;
  ModelI* model;

  // Modell auswählen, hier nur Order0
  model = new ModelOrder0C;

  source.open( argv[1], ios::in | ios::binary );
  target.open( argv[2], ios::out | ios::binary );

  if( !source.is_open() )
  {
    cout << "Kann Eingabestrom nicht öffnen";
    return 2;
  }
  if( !target.is_open() )
  {
    cout << "Kann Ausgabestrom nicht öffnen";
    return 3;
  }

  unsigned int signature;
  source.read(reinterpret_cast<char*>(&signature),sizeof(signature));
  if( signature == g_Signature )
  {
    cout << "Decodiere " << argv[1] << " zu " << argv[2] << endl;
    model->Process( &source, &target, MODE_DECODE );
  }
  else
  {
    cout << "Codiere " << argv[1] << " zu " << argv[2] << endl;
    source.seekg( 0, ios::beg );
    target.write( reinterpret_cast<const char*>(&g_Signature),
                  sizeof(g_Signature) );
    model->Process( &source, &target, MODE_ENCODE );
  }

  source.close();
  target.close();

  return 0;
}
```

Index

Literatur

[1] Timothy C. Bell, John G. Cleary, Ian H. Witten: 'Text Compression', Prentice Hall, Englewood Cliffs, NJ, 1990, pp. 1-26

[2] Khalid Sayood: 'Introduction to Data Compression', 2nd edition, Academic Press, San Diego, CA, 2000, pp. 1-37, 181-187, 201-210

[3] R. Fano: 'Transmission of Information', M.I.T. Press, 1961

[4] Michael Tamm: 'Packen wie noch nie', c't No. 16, 2000, pp. 194-201

[5] David Salomon: 'Data Compression', 2nd edition, Springer-Verlag, New York, 2000, pp. 1-12

[6] James A. Storer: 'Data Compression', Computer Science Press, Rockville, ML, 1988, pp. 1-17

[7] Ian H. Witten, Alistair Moffat, Timothy C. Bell: 'Managing Gigabytes', 2nd edition, Academic Press, San Diego, CA, 1999

[8] A. Gersho, R. M. Gray: 'Vector Quantization and Signal Compression', Kluwer Academic Publishers, Dordrecht, Niederlande, 1992

[9] G. Held, T. R. Marshall: 'Data Compression', Wiley, Chichester, West Sussex, England, 1992

[10] A. Beutelspecher : 'Kryptologie', Braunschweig, 1993

[11] Claude Shannon : 'A Mathematical Theory of Communication', http://cm.bell-labs.com/cm/ms/what/shannonday/paper.html, 1948, pp. 10ff

[12] G. N. N. Martin : 'Range encoding: an algorithm for removing redundancy from a digitised message', http://www.compressconsult.com/rangecoder/rngcod.pdf.gz, 1979

[13] Arturo Campos: 'Range Coder Implementation', http://www.arturocampos.com/ac_range.html, 1999

[14] 'The Data Compression Library', http://dogma.net/DataCompression/ArithmeticCoding.shtml

[15] 'The Canterbury Corpus', http://corpus.canterbury.ac.nz/summary.html