Universität Paderborn

Fakultät für Elektrotechnik, Informatik und Mathematik
Arbeitsgruppe Codes und Kryptographie

Studienarbeit

Tabellenbasierte arithmetische Codierung

von

Daniel Kuntze

vorgelegt bei

Herrn Prof. Dr. Johannes Blömer

Paderborn, den 13. August 2003

Erklärung

Ich versichere, dass ich diese Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Paderborn, den 13. August 2003
Daniel Kuntze

Inhaltsverzeichnis

Abbildungsverzeichnis iii

Tabellenverzeichnis iv

1 Einleitung 1

2 Arithmetische Codierung 3
2.1 Prinzip der arithmetischen Codierung . . . 3
2.2 Ideale arithmetische Codierung . . . 7
2.3 Arithmetische Codierung mit Skalierung . . . 10
2.4 Wechselnde Verteilungen . . . 18

3 Tabellenbasierte arithmetische Codierung 19
3.1 Wahl des Quellalphabets . . . 19
3.2 Reduzierung der Zustände . . . 21
3.3 Codiertabellen . . . 25
3.4 Reduzierung der follow-Bits . . . 29
3.5 Alternative Betrachtung des binären Alphabets . . . 30
3.6 Ausgewählte Verteilungen . . . 31
3.7 Beispielcodierer . . . 34

4 Decodierung tabellenbasierter arithmetischer Codierung 36
4.1 Beschränkte Betrachtung der Codierung . . . 36
4.2 Skalierungen rekonstruieren . . . 37
4.3 Zustandsübergänge nachvollziehen . . . 38
4.4 Decodier-Beispiele . . . 38

5 Implementierung in Java 40
5.1 Die Interfaces . . . 40
5.2 Die Klassen . . . 43
5.2.1 BACSymbol . . . 43
5.2.2 BACEntry . . . 43

5.2.3 BACEntryComparator 44
5.2.4 BACState 45
5.2.5 BACTable 47
5.3 Testumgebung 49

6 Fazit **50**

Literaturverzeichnis **51**

Abbildungsverzeichnis

2.1 Partitionierung des Einheitsintervalls 4
2.2 Fortlaufende Partitionierung 5
2.3 Arithmetische Codierung am Beispiel 6
2.4 Präfix der Codierung beim Skalieren 10
2.5 Die Skalierungsprozeduren 11

3.1 Binäre Partition am Beispiel 20
3.2 Beschränkung möglicher Intervallgrenzen 21
3.3 Beschränkung möglicher Wahrscheinlichkeitsverteilungen . . . 23
3.4 Abbildung der Wahrscheinlichkeitsverteilungen 24
3.5 Reduzierte arithmetische Codierung am Beispiel 26
3.6 Anwendung der Erzeugungstabelle 33

Tabellenverzeichnis

2.1 Codierbeispiel mit Skalierung 15
2.2 Decodierbeispiel mit Skalierung 18

3.1 Schema einer Codiertabelle 25
3.2 Reduzierter binärer arithmetischer Codierer 27
3.3 Vermeidung der *follow*-Bits 29
3.4 Verwendung von MPS und LPS 31
3.5 Erzeugungstabelle für Klasse von reduzierten Codierern 32
3.6 Beispielcodierer für N=8 35

5.1 Übersicht Klassen/Interfaces 41

Zusammenfassung

Diese Arbeit beschreibt ein Verfahren von Howard und Vitter [1], dass arithmetische Codierung imitiert ohne wirklich Arithmetik zu betreiben. Dabei ermöglicht die Inkaufnahme eines gewissen Kompressionsverlustes, dass die nötigen Berechnungen im Voraus getätigt und die Ergebnisse in Tabellen abgelegt werden. Während des Codierens wird dann Arithmetik durch Nachschauen in den Tabellen ersetzt. Da Rechenbefehle verhältnismäßig viele Prozessorzyklen benötigen, wird dadurch die Laufzeit positiv beeinflusst.

Für das beschriebene Verfahren wird anschließend eine Implementierung in Java vorgestellt.

1 Einleitung

In der Codierungstheorie geht es unter anderem um die Suche nach effektiven Kompressionsverfahren. Gute Verfahren haben die Eigenschaft, dass sie asymptotisch optimal sind. Grob gesprochen bedeutet das, dass je mehr Symbole auf einmal codiert werden, desto näher kommt die Kompressionsrate an das theoretische Maximum heran. Genaueres kann man bei Sayood [2] oder Cover und Thomas [3] nachlesen.

Eins der bekanntesten asymptotisch optimalen Verfahren ist die Huffman-Codierung. Mit ihr kann man sehr schnell codieren und decodieren. Der große Nachteil ist jedoch, dass eine Datenstruktur, der Huffman-Baum, im Speicher gehalten werden muss. Dieser Baum wächst exponentiell in der Größe der codierten Blöcke. Man kann also aus Speicherplatzgründen nicht beliebig viele Symbole auf einmal codieren, um die Optimalität des Verfahrens auszunutzen.

Ein weiteres asymptotisch optimales Verfahren ist die arithmetische Codierung. Sie hat gegenüber der Huffman-Codierung den Vorteil, dass beliebig große Blöcke codiert werden können, ohne dass dies negative Auswirkungen auf den Speicherplatzverbrauch hat. Die Symbole eines Blocks werden dabei einfach alle nacheinander verarbeitet. Der Nachteil der arthmetischen Codierung ist die Laufzeit. Da zur Codierung jedes Symbols Arithmetik betrieben wird, ist sie langsamer als die Huffman-Codierung.

An dieser Stelle setzt die tabellenbasierte arithmetische Codierung an. Die Idee ist, den Geschwindigkeitsverlust wett zu machen, indem die Berechnungen vorweg durchgeführt und die Ergebnisse in Tabellen abgelegt werden. Dadurch wird die Arithmetik beim Codieren eingespart und durch Lesen der Tabelleneinträge ersetzt. Dabei muss man allerdings einen gewissen Kompressionsverlust in Kauf nehmen. Je kleiner der Kompressionsverlust sein soll, desto größer werden die Tabellen, die im Speicher gehalten werden müssen. Das Verfahren ist also auch nicht perfekt, aber man kann schon mit kleinen Tabellen gute Ergebnisse erzielen.

In Kapitel 2 wird zunächst eine Einführung in die arithmetische Codierung gegeben. Kapitel 3 behandelt dann Schritt für Schritt die Entwicklung der Codiertabellen. In Kapitel 4 wird kurz auf die Decodierung eingegangen und

Kapitel 5 enthält schließlich die Beschreibung einer Java-Implementierung der tabellenbasierten arithmetischen Codierung.

Auf der beiliegenden CD befinden sich die Quelltexte der in Kapitel 5 beschriebenen Java-Implementierung und die zugehörigen javadoc-Dateien. Außerdem befindet sich dort diese Arbeit in digitaler Form im Postscript- und im Portable-Document-Format.

2 Arithmetische Codierung

Dieses Kapitel bietet eine kurze Einführung in die arithmetische Codierung. Es ist geschrieben in Anlehnung an das Buch *Compression and Coding Algorithms* von Moffat und Turpin [4]. Hier werden die Grundlagen vermittelt, die nötig sind, um die Entwicklung des tabellenbasierten Codierers in Kapitel 3 zu verstehen. Eine umfassendere Behandlung der arithmetischen Codierung kann man auch bei Sayood [2] oder Cover und Thomas [3] nachlesen.

Bei der arithmetischen Codierung werden Daten codiert, indem der gesamten Eingabe (z. B. einer Eingabedatei) eine reelle Zahl aus dem Intervall $[0, 1)$ zugeordnet wird. Obwohl die Berechnung dieser Zahl für die Eingabe als Ganzes erfolgt, wird dabei dennoch sequenziell vorgegangen. Ausgehend von einem Alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ wird die Eingabe als Folge von Symbolen $a_i \in \mathcal{A}$ aufgefasst, die in der Eingabereihenfolge verarbeitet werden, um die Ausgabe (die reelle Zahl) zu berechnen. Das Verfahren eignet sich also auch für die Codierung prinzipiell unendlicher Datenströme.

Im Folgenden sei eine Quelle gegeben, die die zu codierende Eingabe erzeugt. Das Quellalphabet sei $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$. Weiter sei eine Wahrscheinlichkeitsverteilung $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$ gegeben, wobei $P(a_i) = p_i$ die Wahrscheinlichkeit dafür ist, dass die Quelle das Symbol a_i erzeugt. Wir nehmen zunächst an, dass die Verteilung statisch ist, sich also während des Codierens nicht verändert.

2.1 Prinzip der arithmetischen Codierung

Die Codierung von Symbolen erfolgt bei der arithmetischen Codierung durch das Bilden von Teilintervallen. Der Ausgangspunkt dafür ist das Intervall [0,1). Anders ausgedrückt steht das Intervall [0,1) für die leere Eingabe.

Um das erste Symbol zu codieren wird nun zunächst das Intervall [0,1) gemäß der Wahrscheinlichkeitsverteilung $\mathcal{P}$ partitioniert, d. h. es wird in disjunkte Teilintervalle der Form $[l, r)$ aufgeteilt. Dazu wird jedem Symbol $a_i \in \mathcal{A}$ ein Teilintervall $[l_i, r_i) \subset [0, 1)$ zugeordnet, dessen Länge genau der Wahrscheinlichkeit $P(a_i) = p_i$ entspricht. Da die p_i eine Wahrscheinlichkeitsverteilung

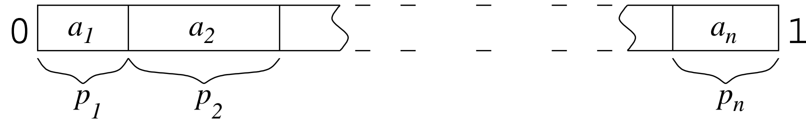

Abbildung 2.1: Partitionierung des Einheitsintervalls

bilden, ist ihre Summe gleich 1. Wenn man nun die Teilintervalle direkt aneinander reiht, also beginnend mit dem Intervall $[l_1, r_1) = [0, p_1)$ so anordnet, dass gilt $l_i = r_{i-1} \ \forall \ i = 2, \ldots, n$, dann erhält man eine Partitionierung des Einheitsintervalls $[0, 1)$, wie sie in Abbildung 2.1 dargestellt ist.

Anschließend wird dasjenige Teilintervall ausgewählt, welches dem Eingabesymbol zugeordnet ist. Nehmen wir an, das zu codierende Eingabesymbol sei a_s. Dann wird das Intervall $[l_s, r_s)$ ausgewählt.

Möchte man nur ein Symbol codieren, kann man an dieser Stelle aufhören und eine beliebige Zahl aus dem Intervall $[l_s, r_s)$ ausgeben. Es kann gezeigt werden, dass es ausreichend ist, den Mittelpunkt $\frac{1}{2}(l_s + r_s)$ mit einer Genauigkeit von $\lceil -\log(r_s - l_s) \rceil + 1 = \lceil -\log p_s \rceil + 1$ Bits auszugeben, um das Intervall $[l_s, r_s)$ eindeutig zu identifizieren. D. h. unabhängig davon, welche Bits noch folgen könnten, liegt diese Zahl dann im Intervall $[l_s, r_s)$. Den Beweis kann man bei Cover und Thomas [3] nachlesen.

Und an dieser Stelle wird es interessant, denn wie man sieht, werden wahrscheinliche Symbole auf große Teilintervalle abgebildet und somit durch weniger Bits codiert als unwahrscheinliche. Das ist also der Punkt, an dem die Kompression erreicht wird.

Auf die oben beschriebene Weise erhält man zu jedem Symbol a_i eine Codierung c_i. Der dadurch erzeugte Code $C = \{c_1, c_2, \ldots, c_n\}$ heißt auch Shannon-Fano-Elias Code. Er ist im Grunde ein Spezialfall der arithmetischen Codierung, deren eigentliche Stärke es ist, große Blöcke aus vielen Symbolen in Folge zu codieren.

Denn anstatt nach dem ersten Symbol aufzuhören, kann die Codierung basierend auf dem Intervall $[l_s, r_s)$ fortgesetzt werden. So wie zuvor das Intervall $[0, 1)$ wird dazu nun das Intervall $[l_s, r_s)$ gemäß der Wahrscheinlichkeitsverteilung $\mathcal{P}$ partitioniert. Die dabei entstehenden Teilintervalle $[l_{s_i}, r_{s_i})$ haben dann nicht mehr Länge p_i, sondern $p_i \cdot p_s$, denn es soll ja nun ein Intervall der Länge p_s partitioniert werden. In Abbildung 2.2 ist diese erneute Partitionierung schematisch dargestellt. Anschließend wird wie oben das Teilintervall ausgewählt, das dem zweiten Eingabesymbol zugeordnet ist.

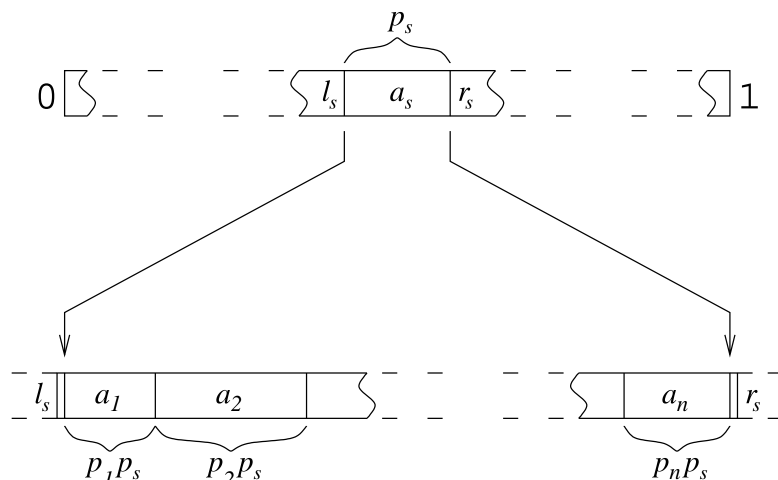

Abbildung 2.2: Fortlaufende Partitionierung

Dieser Vorgang kann nun beliebig oft wiederholt werden. Sind alle zu codierenden Symbole auf diese Weise verarbeitet, kann man wieder eine beliebige Zahl aus dem zuletzt berechneten Teilintervall $[l, r)$ als Ausgabe wählen. Eine Kompression wird wiederum dadurch erzielt, dass es ausreicht, den Intervallmittelpunkt mit einer Genauigkeit von $\lceil -\log(r - l) \rceil + 1 = \lceil -\log P(x) \rceil + 1$ Bits auszugeben. Dabei steht $x = a_{i_1} \ldots a_{i_b}$ für eine Eingabefolge der Länge b mit Wahrscheinlichkeit

$$P(x) = \prod_{j=1}^{b} p_{i_j}.$$

An dieser Stelle muss man noch etwas Überlegung investieren, um die korrekte Decodierung zu gewährleisten. Beim Decodieren wird angefangen mit dem Intervall $[0, 1)$ genau wie beim Codieren das aktuelle Intervall gemäß der Verteilung $\mathcal{P}$ partitioniert. Dann wird geschaut, in welchem Teilintervall die Codierung liegt und zum entsprechenden Symbol decodiert. Wenn man mehrere Symbole in Folge codiert hat, muss der Decodierer wissen, wann er aufhören soll zu decodieren. Denn sonst kann er immer wieder das aktuelle Intervall partitionieren und wird natürlich immer wieder ein Teilintervall finden, in dem die Codierung liegt.

Um das zu verhindern, kann man vorweg eine Blocklänge festlegen. Oder man kann ein spezielles Ende-Symbol vereinbaren, das immer an eine zu codierende Eingabe angehängt wird. Der Decodierer hört dann auf, sobald er das Ende-Symbol decodiert hat.

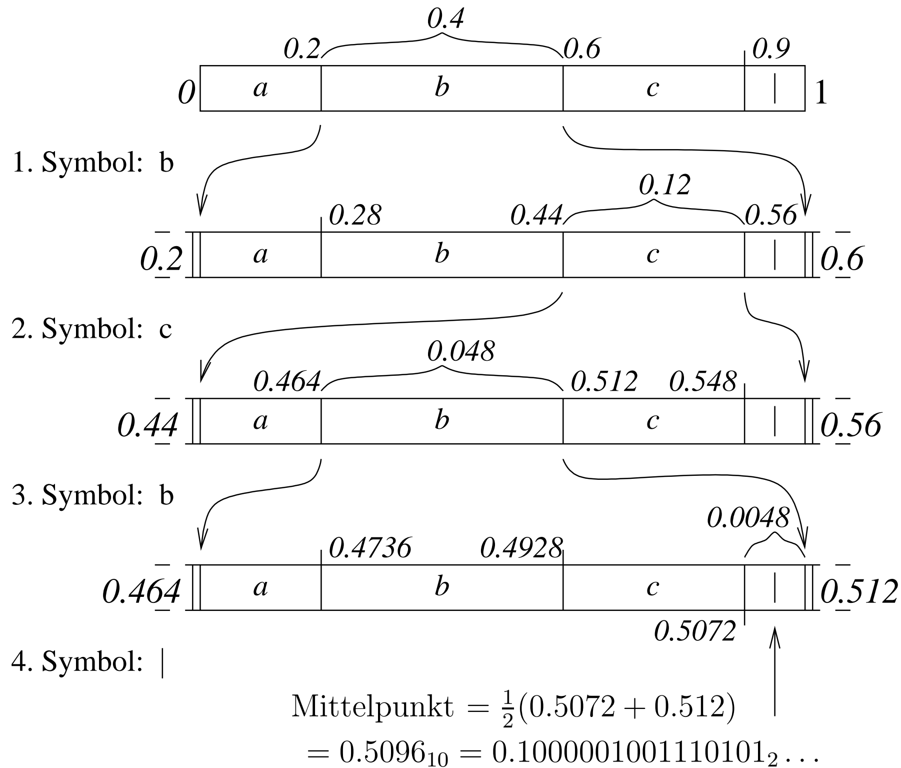

Abbildung 2.3: Arithmetische Codierung am Beispiel

In Abbildung 2.3 ist die Codierung der Beispielsymbolfolge $x = bcb|$ dargestellt. Das zugrunde liegende Quellalphabet ist $\mathcal{A} = \{a, b, c, |\}$ und die Wahrscheinlichkeitsverteilung ist gegeben durch

$$\mathcal{P} = \{P(a) = 0.2 \ , \ P(b) = 0.4 \ , \ P(c) = 0.3 \ , \ P(|) = 0.1\}.$$

| ist dabei das Ende-Symbol.

Die Wahrscheinlichkeit für das Auftreten der Eingabefolge x ist $P(x) = P(b)^2 P(c) P(|) = 0.0048$. Also gilt $\lceil -\log P(x) \rceil + 1 = \lceil -\log 0.0048 \rceil + 1 = 9$ und somit reicht es aus, die ersten 9 binären Nachkommastellen von 0.5096, dem Mittelpunkt des zuletzt berechneten Teilintervalls auszugeben. Die Ausgabe lautet demnach **100000100**.

Die sukzessive berechneten Teilintervalle beschreiben eindeutig die bis zum jeweiligen Zeitpunkt eingelesenen Symbole. Wie man sich an dem Beispiel aus Abbildung 2.3 verdeutlichen kann, ergibt jede andere Eingabefolge auch ein anderes Intervall. Daher werden die Intervallgrenzen auch als Zustand des Codierers interpretiert.

2.2 Ideale arithmetische Codierung

Bei der idealisierten Form der arithmetischen Codierung wird von der Tatsache abstrahiert, dass sich in einem Rechner reelle Zahlen beliebiger Genauigkeit gar nicht darstellen lassen. Sie ist mehr theoretische Grundlage und eignet sich dazu, das Verfahren der arithmetischen Codierung zu formalisieren. Das heißt, man kann Algorithmen zur Codierung und Decodierung angeben, ohne sich mit dem Problem der Rechengenauigkeit auseinander setzen zu müssen. Die nachfolgenden Algorithmen arbeiten dabei nicht mit einem Ende-Symbol, sondern mit einer festen Blocklänge $b \in \mathbb{N}$.

Betrachten wir wie oben eine Quelle mit Alphabet $\mathcal{A} = \{a_1, a_2, \ldots, a_n\}$ und Wahrscheinlichkeitsverteilung $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$. Dann definieren wir zunächst für $i = 1, \ldots, n$:

$$L_i := \sum_{j=1}^{i-1} p_j$$

Zusätzlich setzen wir $L_{n+1} := 1$.

Die L_i bilden dann eine Partition des Einheitsintervalls $[0,1)$ in n Teilintervalle $I_i = [L_i, L_{i+1})$ für $i = 1, \ldots, n$.

Ein Algorithmus zur Codierung lässt sich dann folgendermaßen angeben:

Ideal-Encode-AC (IE-AC)

Eingabe $m = m_1 m_2 \cdots m_b \in \mathcal{A}^b$

1. Schritt Setze $l := 0,\ \ r := 1,\ \ w := 1.$

2. Schritt Für $i = 1, \ldots, b$ wiederhole die Schritte 3 und 4

3. Schritt Bestimme $s \in \{1, \ldots, n\}$, so dass $m_i = a_s$.

4. Schritt Setze

$$\begin{aligned} r &:= l + wL_{s+1} \\ l &:= l + wL_s \\ w &:= wp_s \end{aligned}$$

5. Schritt Gib die ersten $\lceil -\log w \rceil + 1$ binären Nachkommastellen von $\frac{1}{2}(l + r)$ aus.

Im 1. Schritt werden die Variablen l und r für die Intervallgrenzen des aktuellen Intervalls mit dem Einheitsintervall initialisiert. Die Variable w für die Intervalllänge wird entsprechend auf 1 gesetzt.

Der 2. Schritt enthält die Schleife zur Abarbeitung der b Eingabesymbole. Innerhalb der Schleife wird im 3. Schritt der Index s des aktuellen Eingabesymbols ermittelt und im 4. Schritt das zugehörige Teilintervall der aktuellen Partition ermittelt.

Das Intervall $[l, r)$ beschreibt dabei nach dem i. Schleifendurchlauf das Teilintervall, das durch die Verarbeitung der ersten i Eingabesymbole entsteht. Es steht also für die Eingabe $m_1 m_2 \cdots m_i$. Durch die Multiplikation von w mit p_s gilt außerdem nach jedem Durchlauf $w = r - l$. w ist also immer gleich der Länge des aktuellen Intervalls $[l, r)$. Die wird im jeweils nächsten Schleifendurchlauf benötigt, um die Partition des Einheitsintervalls durch die L_i auf eine Partition des Intervalls $[l, r)$ zu skalieren.

Schritt 5 dient nur zur abschließenden Ausgabe des Mittelpunkts des zuletzt berechneten Teilintervalls. Dabei werden nur die ersten zur eindeutigen Identifizierung des letzten Intervalls nötigen Nachkommastellen ausgegeben. Es werden nur Nachkommastellen ausgegeben, da die auszugebende Zahl in dem halboffenen Intervall $[0, 1)$ liegt und daher vor dem Komma immer die 0 steht.

Für die Decodierung lässt sich der folgende Algorithmus formulieren:

Ideal-Decode-AC (ID-AC)

Eingabe $c \in \{0,1\}^*$

1. Schritt Setze $l := 0,\ r := 1,\ w := 1,\ m = \epsilon,\ t = (0.c)_2$.

2. Schritt Für $i = 1, \ldots, b$ wiederhole die Schritte 3 und 4

3. Schritt Bestimme $s \in \{1, \ldots, n\}$, so dass

$$l + wL_s \leq t < l + wL_{s+1}.$$

4. Schritt Setze

$$\begin{aligned} r &:= l + wL_{s+1} \\ l &:= l + wL_s \\ w &:= wp_s \\ m &:= m \cdot a_s \end{aligned}$$

5. Schritt Gib m aus.

Der Algorithmus unterscheidet sich nur wenig vom Codier-Algorithmus.

Im 1. Schritt wird zusätzlich eine Variable m eingeführt und mit ϵ initialisiert, die nach und nach die b Symbole des codierten Blocks aufnimmt. Außerdem wird dort die Variable t mit der Zahl $(0.c)_2$ initialisiert, für die die Codierung c steht.

Der 3. Schritt ist der einzige interessante Schritt, weil dort der Codierungsschritt rückgängig gemacht wird. Dazu werden die Teilintervallgrenzen der aktuellen Partitionierung bestimmt und mit der Zahl t verglichen, um das Teilintervall $[l + wL_s,\ l + wL_{s+1})$ zu ermitteln, in dem t liegt. Daraus ergibt sich das nächste Symbol der Nachricht als a_s.

Im 4. Schritt kommt lediglich die Konkatenation der bisher decodierten Nachricht mit dem gerade decodierten Symbol hinzu.

In Schritt 5 wird schließlich die Rekonstruktion der Nachricht ausgegeben.

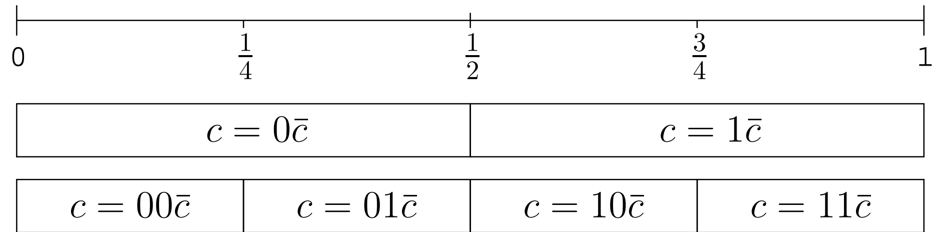

Abbildung 2.4: Präfix der Codierung beim Skalieren

2.3 Arithmetische Codierung mit Skalierung

Beim Codiervorgang, wie er in den vorherigen Abschnitten beschrieben ist, wird erst nach der Verarbeitung des letzten Eingabesymbols die Ausgabe bestimmt, also nach dem Bilden des letzten Teilintervalls. Es sieht also zunächst so aus, als müsste man die Eingabe komplett verarbeiten, bevor man etwas ausgeben kann. Aber dem ist nicht so. Während des Codierens stehen sogar in der Regel immer mehr Bits schon vorher fest.

Dann nämlich, wenn das gewählte Teilintervall $[l, r)$ komplett in der unteren oder oberen Hälfte des Einheitsintervalls $[0, 1)$ liegt, wenn also gilt $[l, r) \subseteq [0, \frac{1}{2})$ bzw. $[l, r) \subseteq [\frac{1}{2}, 1)$, dann liegen folglich auch alle weiteren Teilintervalle und auch die Codierung in der gleichen Hälfte. Dann aber steht das erste Bit der Ausgabe schon fest, denn alle Zahlen aus dem Intervall $[0, \frac{1}{2})$ haben als erste binäre Nachkommastelle eine 0 und alle Zahlen aus dem Intervall $[\frac{1}{2}, 1)$ als erste binäre Nachkommastelle eine 1.

Nehmen wir an, dass $[l, r) \subseteq [0, \frac{1}{2})$ gilt. Dann gilt für die zu berechnende Codierung $c \in \{0, 1\}^* : c = 0\bar{c}$ mit $\bar{c} \in \{0, 1\}^*$. In Abbildung 2.4 ist dieser Sachverhalt veranschaulicht. Ausgedrückt durch die reellen Zahlen, für die c und $\bar{c}$ stehen bedeutet dies $(0.\bar{c})_2 = 2(0.c)_2$.

Man kann also die **0** ausgeben und erhält $\bar{c}$ (den Rest der Ausgabe), indem man mit dem Intervall $[2l, 2r)$ weiterarbeitet. Insgesamt wird die Länge der Ausgabe davon nicht beeinflusst, denn es gilt $\forall x \in \mathbb{R}$:

$$\begin{aligned} -\log(2x) &= -\log(x) - 1 \\ \Rightarrow \qquad \lceil -\log(2r - 2l) \rceil + 1 &= \lceil -\log(r - l) \rceil + 1 - 1 \end{aligned}$$

Die restliche Ausgabe wird also um 1 verkürzt und die Länge bleibt in der Summe gleich.

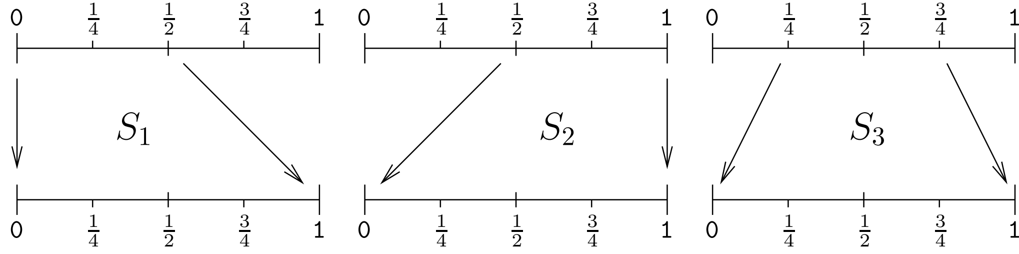

Abbildung 2.5: Die Skalierungsprozeduren

Diese Skalierung des aktuellen Intervalls wird auch als Anwendung der Skalierungsprozedur S_1 bezeichnet. Formal ist S_1 eine Abbildung:

$$\begin{aligned} S_1 : \left[0, \tfrac{1}{2}\right) &\rightarrow [0,1) \\ x &\mapsto 2x \end{aligned}$$

In Abbildung 2.5 ist S_1 schematisch dargestellt.

Im Fall $[l, r) \subseteq [\frac{1}{2}, 1)$ gilt analog $c = 1\bar{c}$ und $(0.\bar{c})_2 = 2\left((0.c)_2 - \frac{1}{2}\right)$. Daraus ergibt sich die Skalierungsprozedur S_2 als Abbildung:

$$\begin{aligned} S_2 : \left[\tfrac{1}{2}, 1\right) &\rightarrow [0,1) \\ x &\mapsto 2\left(x - \tfrac{1}{2}\right) \end{aligned}$$

Der Grund für die Einführung der Skalierungsprozeduren ist aber nicht nur die möglichst frühe Ausgabe von Bits. Das Skalieren der Intervallgrenzen soll verhindern, dass die Intervalle beliebig klein werden können und so ermöglichen, mit endlicher Genauigkeit zu rechnen. Dazu reichen allerdings die bereits eingeführten Abbildungen S_1 und S_2 nicht aus. Denn wenn sich die linke Intervallgrenze von links und die rechte von rechts an $\frac{1}{2}$ annähert, kann das Intervall $[l, r)$ beliebig klein werden, ohne dass jemals $[l, r) \subseteq [0, \frac{1}{2})$ oder $[l, r) \subseteq [\frac{1}{2}, 1)$ gilt. Aus diesem Grund wird folgende Abbildung S_3 eingeführt:

$$\begin{aligned} S_3 : \left[\tfrac{1}{4}, \tfrac{3}{4}\right) &\rightarrow [0,1) \\ x &\mapsto 2\left(x - \tfrac{1}{4}\right) \end{aligned}$$

Die Skalierungsprozedur S_3 findet Anwendung, wenn das aktuelle Teilintervall komplett innerhalb des Intervalls $[\frac{1}{4}, \frac{3}{4})$ liegt und weder S_1 noch S_2 angewendet werden können.

Was aber passiert mit der Ausgabe? Wenn man mit der Abbildung S_3 skaliert, muss sich das natürlich auch in der Ausgabe niederschlagen, damit sie nicht

verfälscht wird. Aber im Gegensatz zu S_1 und S_2 kann weder eine **0** noch eine **1** ausgegeben werden, da sich noch nicht sagen lässt, ob das finale Teilintervall in der oberen oder unteren Hälfte von [0,1) liegen wird. Da aber für das aktuelle Teilintervall $[l, r) \subseteq \left[\frac{1}{4}, \frac{3}{4}\right)$ gilt, werden die nächsten beiden Ausgabebits entweder **01** oder **10** sein. Sie sind gleich **01**, falls $[l, r) \subseteq \left[\frac{1}{4}, \frac{1}{2}\right)$ und gleich **10**, falls $[l, r) \subseteq \left[\frac{1}{2}, \frac{3}{4}\right)$ gilt, wie sich an Abbildung 2.4 nachvollziehen lässt.

Würde man nun also mit der Abbildung S_3 skalieren und den Algorithmus *Ideal-Encode-AC* mit dem skalierten Intervall $[S_3(l), S_3(r))$ weiterrechnen lassen, dann gibt es zwei Fälle zu unterscheiden:

1. Die nächsten beiden Ausgabebits müssten **01** lauten, d. h. $c = 01\bar{c}$. Die tatsächlich berechnete Ausgabe ist dann $c = 0\bar{c}$, da $\left[\frac{1}{4}, \frac{1}{2}\right)$ gleich der unteren Hälfte von $\left[\frac{1}{4}, \frac{3}{4}\right)$ ist.

2. Die nächsten beiden Ausgabebits müssten **10** lauten, d. h. $c = 10\bar{c}$. Die tatsächlich berechnete Ausgabe ist dann $c = 1\bar{c}$, da $\left[\frac{1}{2}, \frac{3}{4}\right)$ gleich der oberen Hälfte von $\left[\frac{1}{4}, \frac{3}{4}\right)$ ist.

Man kann also nach einer Anwendung von S_3 noch nichts ausgeben, weiß aber, dass auf das nächste auszugebende Bit noch seine Negation folgen muss. Wird mehrmals nacheinander mit S_3 skaliert, muss entsprechend auf das nächste auszugebende Bit so oft seine Negation folgen, wie S_3 angewendet wurde. Deshalb merkt man sich jedes Skalieren mit S_3 in einer Zählvariable *ost* (engl. *outstanding bits*), die bei der Ausgabe des nächsten Bits abgearbeitet wird.

Wenn nun nach jeder Teilintervallbildung skaliert wird, solange eine der drei Skalierungsprozeduren S_1, S_2, S_3 anwendbar ist, dann gilt vor jeder Partitionierung des Teilintervalls $[l, r): \; r - l > \frac{1}{4}$. Und das ist der Grund dafür, dass mit endlicher Präzision gerechnet werden kann. Es lässt sich nämlich zeigen, dass es ausreicht, alle Zahlen im Algorithmus mit einer Genauigkeit von k Bits darzustellen ohne die Eindeutigkeit des Codes zu verlieren, wenn für k gilt

$$p_i > 2^{-k+2} \;\; \forall i \in \{1, \ldots, n\}.$$

Der angepasste Algorithmus zum Codieren sieht folgendermaßen aus:

Encode-AC (E-AC)

Eingabe $m = m_1 m_2 \cdots m_b \in \mathcal{A}^b$

1. Schritt Setze $l := 0,\ \ r := 1,\ \ w := 1,\ \ c := \epsilon,\ \ ost := 0$

2. Schritt Für $i = 1, \ldots, b$ wiederhole die Schritte 3 bis 7

3. Schritt Bestimme $s \in \{1, \ldots, n\}$, so dass $m_i = a_s$

4. Schritt Setze

$r := l + wL_{s+1}$
$l := l + wL_s$
$w := wp_s$

5. Schritt Falls $r < \frac{1}{2}$, setze

$c := c \cdot 0 \cdot 1^{ost}$
$ost := 0$
$l := S_1(l) \quad = 2l$
$r := S_1(r) \quad = 2r$
$w := 2w$

Weiter mit Schritt 5

6. Schritt Falls $l \geq \frac{1}{2}$, setze

$c := c \cdot 1 \cdot 0^{ost}$
$ost := 0$
$l := S_2(l) \quad = 2\left(l - \frac{1}{2}\right)$
$r := S_2(r) \quad = 2\left(r - \frac{1}{2}\right)$
$w := 2w$

Weiter mit Schritt 5

7. Schritt Falls $l \geq \frac{1}{4}$ und $r < \frac{3}{4}$, setze

$ost := ost + 1$
$l := S_3(l) \quad = 2\left(l - \frac{1}{4}\right)$
$r := S_3(r) \quad = 2\left(r - \frac{1}{4}\right)$
$w := 2w$

Weiter mit Schritt 5

8. Schritt Setze $\hat{c}$ auf die ersten $\lceil -\log\left(w2^{-ost}\right)\rceil + 1$ binären Nachkommastellen von $\frac{1}{2}\left(S_3^{-ost}(l) + S_3^{-ost}(r)\right)$.
Gib $c \cdot \hat{c}$ aus.

In Schritt 1 kommt die Variable c hinzu, die mit ϵ initialisiert wird. Sie nimmt später nach und nach die Bits der Codierung auf, die vorzeitig feststehen, wenn mit S_1 oder S_2 skaliert wird.

Schritt 2,3 und 4 sind gleich geblieben.

In den Schritten 5 bis 7 werden die Skalierungsprozeduren durchgeführt.

Schritt 8 bedarf noch einiger Erklärung. Dort wird der Rest der Codierung bestimmt, der nicht vorzeitig feststeht. Dazu wird wie beim Algorithmus *Ideal-Encode-AC* der Mittelpunkt des aktuellen Intervalls mit der nötigen Genauigkeit berechnet. Allerdings werden vorher evtl. noch Anwendungen der Skalierungsprozedur S_3 rückgängig gemacht. Denn wenn im letzten Schleifendurchlauf ganz am Schluss ein- oder mehrmals mit S_3 skaliert wurde, dann stehen noch *outstanding bits* aus. Diese können allerdings nicht mehr durch Ausführungen von Schritt 5 oder Schritt 6 abgearbeitet werden. Deshalb werden noch nicht berücksichtigte Skalierungen mit S_3 bei der Berechnung des Mittelpunkts des letzten Intervalls rückgängig gemacht. Dazu wird auf die Intervallgrenzen entsprechend oft die Umkehrabbildung von S_3 angewendet und die in w gespeicherte Intervalllänge entsprechend korrigiert. Die Umkehrabbildung von S_3 ist dabei gegeben durch:

$$\begin{array}{rcl} S_3^{-1} : [0,1) & \to & \left[\frac{1}{4}, \frac{3}{4}\right) \\ x & \mapsto & \frac{1}{2}x + \frac{1}{4} \end{array}$$

i	m_i	$[l, r)$	w	ost	c	Aktion
0		$[0, 1)$	1	0	ϵ	Initialisierung
1	b	$[0.2, 0.6)$	0.4	0	ϵ	Einlesen von $m_1 = b$
2	c	$[0.44, 0.56)$	0.12	0	ϵ	Einlesen von $m_2 = c$
		$[0.38, 0.62)$	0.24	1	ϵ	Skalieren mit S_3
		$[0.26, 0.74)$	0.48	2	ϵ	Skalieren mit S_3
		$[0.02, 0.98)$	0.96	3	ϵ	Skalieren mit S_3
3	b	$[0.212, 0.596)$	0.384	3	ϵ	Einlesen von $m_3 = b$
4	$\vert$	$[0.5576, 0.596)$	0.0384	3	ϵ	Einlesen von $m_4 = \vert$
		$[0.1152, 0.192)$	0.0768	0	1000	Skalieren mit S_2
		$[0.2304, 0.384)$	0.1536	0	10000	Skalieren mit S_1
		$[0.4608, 0.768)$	0.3072	0	100000	Skalieren mit S_1

Tabelle 2.1: Codierbeispiel mit Skalierung

Wenden wir die arithmetische Codierung mit Skalierung auf das Beispiel aus Abschnitt 2.1 (siehe Abbildung 2.3) an, dann erhalten wir die Codierungsschritte, wie sie in Tabelle 2.1 aufgelistet sind. Da der Algorithmus *Encode-AC* mit einer festen Blocklänge anstatt eines Ende-Symbols arbeitet, interpretieren wir das Ende-Symbol aus Abbildung 2.3 als normales Symbol und arbeiten mit einer Blocklänge von 4. Außerdem wird der Übersichtlichkeit halber nicht mit endlicher Genauigkeit gerechnet.

Für die Ausgabe wird vom zuletzt berechneten Intervall $[0.4608, 0.768)$ der Mittelpunkt $0.6144_{10} = 0.1001110_2 \ldots$ berechnet. Da $\lceil -\log w \rceil + 1 = 3$ wird $\hat{c} = 100$ gesetzt. Die Ausgabe lautet also wie bereits in Abschnitt 2.1 ermittelt $c \cdot \hat{c} = 100000100$. Da nicht mit endlicher Genauigkeit gerechnet wurde, musste das auch so sein.

Genau wie bei der Codierung reicht es auch bei der Decodierung aus, mit einer Genauigkeit von k Bits zu rechnen. Insbesondere werden beim Decodieren immer nur k Bits der Codierung auf einmal betrachtet. Dazu werden k Variablen t_i für $i = 1, \ldots, k$ verwendet, die nach und nach alle Bits der Codierung aufnehmen.

Der angepasste Algorithmus zum Decodieren sieht folgendermaßen aus:

Decode-AC (D-AC)

	Eingabe $c \in \{0,1\}^*$
1. Schritt	Setze $l := 0,\ r := 1,\ w := 1,\ m = \epsilon$ $t_i := c_i$ für $i = 1, \dots, k$, j:=k+1
2. Schritt	Für $i = 1, \dots, b$ wiederhole die Schritte 3 bis 7
3. Schritt	Bestimme $s \in \{1, \dots, n\}$, so dass $l + wL_s \leq (0.t_1 \dots t_k)_2 < l + wL_{s+1}$
4. Schritt	Setze $r := l + wL_{s+1}$ $l := l + wL_s$ $w := wp_s$ $m := ma_s$
5. Schritt	Falls $r < \frac{1}{2}$, setze $t_i := t_{i+1}$ für $i = 1, \dots, k-1$ $t_k := c_j$ $j := j + 1$ $l := S_1(l) = 2l$ $r := S_1(r) = 2r$ $w := 2w$ Weiter mit Schritt 5
6. Schritt	Falls $l \geq \frac{1}{2}$, setze $t_i := t_{i+1}$ für $i = 1, \dots, k-1$ $t_k := c_j$ $j := j + 1$ $l := S_2(l) = 2\left(l - \frac{1}{2}\right)$ $r := S_2(r) = 2\left(r - \frac{1}{2}\right)$ $w := 2w$ Weiter mit Schritt 5
7. Schritt	Falls $l \geq \frac{1}{4}$ und $r < \frac{3}{4}$, setze $t_i := t_{i+1}$ für $i = 2, \dots, k-1$ $t_k := c_j$ $j := j + 1$ $l := S_3(l) = 2\left(l - \frac{1}{4}\right)$ $r := S_3(r) = 2\left(r - \frac{1}{4}\right)$ $w := 2w$ Weiter mit Schritt 5
8. Schritt	Gib m aus.

Im 1. Schritt werden die t_i mit den ersten k Bits der Codierung initialisiert. j wird mit $k + 1$ initialisiert und enthält fortan immer das erste Bit der Codierung, das noch nicht gelesen wurde.

Die Skalierungen in den Schritten 5 bis 7 bedürfen noch einiger Erklärung. Im 5. Schritt wird die Abbildung S_1 auf die Intervallgrenzen angewendet. Damit die Codierung auch relativ zu den skalierten Intervallgrenzen betrachtet wird, muss die Codierung ebenfalls skaliert werden. Dazu werden die t_i einfach um eine Position nach links geschoben, was der Multiplikation mit 2 entspricht. Da $(0.t_1 \ldots 0.t_k)_2 \in [l, r)$ und $r < \frac{1}{2}$ gilt, muss vor dem Shift $t_1 = 0$ gelten. An die Position k rückt anschließend das nächste Bit der Codierung c_j nach. Dadurch enthalten die t_i wieder die Codierung mit k Bits Genauigkeit relativ zum skalierten Intervall $[l, r)$. j wird danach inkrementiert, damit c_j wieder das erste noch nicht gelesene Bit der Codierung ist.

Schritt 6 ist entsprechend für die Skalierung mit S_2 zuständig. Da dort entsprechend $l \geq \frac{1}{2}$ gilt, muss vor dem Shift $t_1 = 1$ gelten. Da S_2 gleichbedeutend ist mit "$\frac{1}{2}$ abziehen und dann mit 2 multiplizieren", müsste man korrekterweise $t_1 = 0$ setzen und anschließen um eine Position nach links schieben. Wir betrachten aber nur Nachkommastellen, weil vor dem Komma immer die 0 steht. Darum verschwindet das in t_1 gespeicherte Bit sowieso ganz. Deshalb reicht es aus, genau wie in Schritt 5 die t_i nach links zu schieben und $t_k = c_j$ zu setzen.

Vor der Skalierung mit S_3 gilt $[l, r) \subseteq [\frac{1}{4}, \frac{3}{4})$. Darum ist im 7. Schritt $t_1t_2 = 01$ oder $t_1t_2 = 10$. Zieht man nun $\frac{1}{4}$ von $(0.t_1 \ldots t_k)_2$ ab, ist danach $t_1t_2 = 00$ oder $t_1t_2 = 01$. Nach der Multiplikation mit 2, also nach dem Linksshift, gilt dann $t_1 = 0$ oder $t_1 = 1$. t_1 ändert sich also durch die Skalierung mit S_3 nicht. Darum wird einfach nur ein Shift der $t_2 \ldots t_k$ um eine Position nach links durchgeführt. t_k und j werden danach wie in den Schritten 5 und 6 aktualisiert.

Wenn wir die Codierung **100000100** aus dem Beispiel in Tabelle 2.1 mit diesem Algorithmus decodieren, dann ergibt sich die Abfolge aus Tabelle 2.2. Wir setzen $k := 6$, da $0.1 > 2^{-4}$. Denn mit der kleinsten Wahrscheinlichkeit $P(|) = 0.1$ sind alle Wahrscheinlichkeiten größer als $2^{-4} = 2^{-6+2}$.

Erinnerung: Es muss gelten $p_i > 2^{-k+2} \;\; \forall i \in \{1, \ldots, n\}$.

i	$t_1 \ldots t_6$	$(0.t_1 \ldots t_6)_2$	$[l, r)$	w	m	Aktion
0	100000	0.5	$[0, 1)$	1	ϵ	Initialisierung
1	100000	0.5	$[0.2, 0.6)$	0.4	b	$m_1 = b$
2	100000	0.5	$[0.44, 0.56)$	0.12	bc	$m_2 = c$
	100001	0.515625	$[0.38, 0.62)$	0.24	bc	Anwendung S_3
	100010	0.53125	$[0.26, 0.74)$	0.48	bc	Anwendung S_3
	100100	0.5625	$[0.02, 0.98)$	0.96	bc	Anwendung S_3
3	100100	0.5625	$[0.212, 0.596)$	0.384	bcb	$m_3 = b$
4	100100	0.5625	$[0.5576, 0.596)$	0.0384	$bcb\vert$	$m_4 = \vert$

Tabelle 2.2: Decodierbeispiel mit Skalierung

Eigentlich würde das letzte Intervall noch mehrmals skaliert. Aber da bereits alle Symbole des Blocks decodiert sind, kann abgebrochen werden. Allgemein formuliert kann abgebrochen werden sobald das Symbol m_b decodiert ist. Man könnte diese Abbruchbedingung auch im Algorithmus einbringen, aber wegen der Übersichtlichkeit wurde darauf verzichtet.

2.4 Wechselnde Verteilungen

Bisher haben wir eine statische Wahrscheinlichkeitsverteilung $\mathcal{P}$ über einem Quellalphabet $\mathcal{A}$ angenommen. D. h. während des gesamten Codiervorgangs blieb die Verteilung die gleiche.

Dies muss aber nicht so sein. Es ist sogar eine der Stärken der arithmetischen Codierung, dass jeder Partitionierung eine andere Verteilung zugrunde gelegt werden kann. Darum eignet sie sich z. B. gut für adaptive Verfahren, bei denen die Wahrscheinlichkeitsverteilung ständig angepasst wird.

Die einzige Bedingung dafür ist, dass vor der Decodierung jedes Symbols dem Decodierer die gleiche Wahrscheinlichkeitsverteilung zur Verfügung steht, wie dem Codierer vor der Codierung. Ansonsten kann der Decodierer die Schritte des Codierers nicht korrekt nachvollziehen.

3 Tabellenbasierte arithmetische Codierung

Die Idee der tabellenbasierten arithmetischen Codierung ist es, die Anzahl möglicher Wahrscheinlichkeitsverteilungen über dem Quellalphabet soweit zu reduzieren, dass sich der Codierer vor dem Einlesen eines neuen Quellsymbols nur noch in einer überschaubaren Anzahl von Zuständen befinden kann. Die für die arithmetische Codierung nötigen Berechnungen können dann vorab durchgeführt und ihre Ergebnisse in Tabellen abgelegt werden. Während des Codierens wird dann nicht mehr für jedes neue Eingabesymbol ein Teilintervall berechnet, sondern es wird in einer Tabelle nachgeschaut, welche Ausgabe erzeugt werden muss und in welchen Zustand der Codierer wechselt.

3.1 Wahl des Quellalphabets

Um die Anzahl der Zustandsübergänge möglichst gering zu halten, wird als Quellalphabet das binäre Alphabet $\mathcal{A} = \{0, 1\}$ gewählt. Andere Quellalphabete müssen dann erst in 0,1-Folgen transformiert werden, bevor der tabellenbasierte Codierer Anwendung findet. Howard und Vitter [1] entwickeln zu diesem Zweck eine Datenstruktur namens *compressed trees*, die hier aber nicht näher erläutert wird.

In dieser Arbeit wird davon ausgegangen, dass die Quellsymbole aus dem binären Alphabet stammen. Dies bedeutet insbesondere, dass bei der Partitionierung das aktuelle Intervall $[l, r)$ immer an einer Stelle $m \in (l, r)$ in zwei Teilintervalle $[l, m)$ und $[m, r)$ aufgespalten wird.

Es gibt dann bei gegebener Wahrscheinlichkeitsverteilung noch zwei Möglichkeiten, das Intervall $[l, r)$ zu partitionieren. Einmal kann das linke Intervall $[l, m)$ der 0 zugeordnet sein und das rechte Intervall $[m, r)$ der 1. Die andere Möglichkeit ist der umgekehrte Fall, also $[l, m)$ ist der 1 zugeordnet und $[m, r)$ der 0. In Abbildung 3.1 sind diese beiden Fälle für das Einheitsintervall und die Verteilung $\mathcal{P} = \{P(0) = \frac{2}{3}, P(1) = \frac{1}{3}\}$ beispielhaft dargestellt.

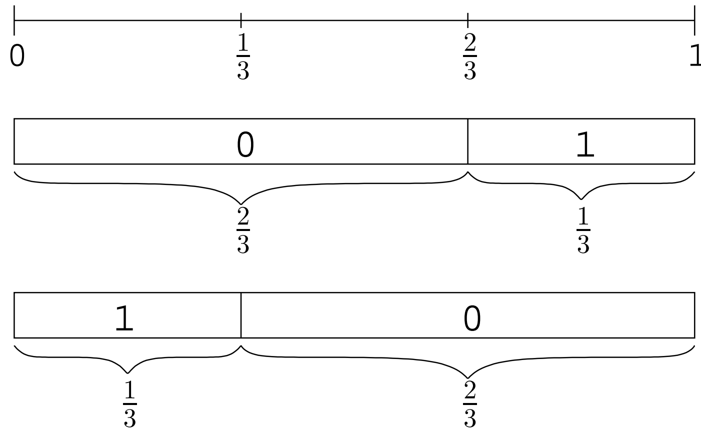

Abbildung 3.1: Binäre Partition am Beispiel

Die Wahl ist beliebig aber fest. Das bedeutet, zum Zeitpunkt des Codierens muss eindeutig bestimmt sein, welche Partitionierungsreihenfolge der Codierer wählt. Außerdem darf diese Wahl nur von Faktoren abhängen, die der Decodierer an gleicher Stelle auch zur Verfügung hat, damit er entsprechend die gleiche Variante wählt.

Wenn man Wahrscheinlichkeitsverteilungen auf dem binären Quellalphabet betrachtet, reicht es aus, eine der beiden Wahrscheinlichkeiten anzugeben, um die Verteilung zu beschreiben. Denn wenn das eine Symbol Wahrscheinlichkeit p hat, dann muss das andere Symbol Wahrscheinlichkeit $1-p$ haben. Im Folgenden wird eine Verteilung deshalb stets nur durch Angabe der Wahrscheinlichkeit für die 0 spezifiziert. Die Schreibweise für eine Verteilung mit $P(0) = p$ ist dann $\mathcal{P}(p)$.

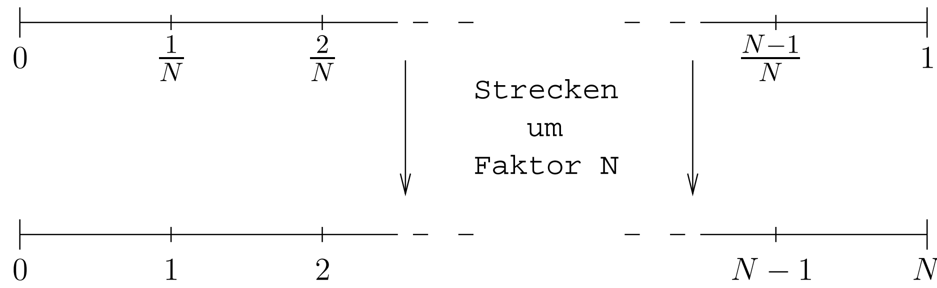

Abbildung 3.2: Beschränkung möglicher Intervallgrenzen

3.2 Reduzierung der Zustände

Die Intervallgrenzen, die bei der idealen arithmetischen Codierung (siehe Abschnitt 2.2) sukzessive mit jedem neuen Eingabesymbol berechnet werden, sind gleichbedeutend mit dem Zustand des Codierers. Sie beschreiben zu einem bestimmten Zeitpunkt eindeutig die bis dahin gelesenen Eingabesymbole. Aus diesem Grund werden die Begriffe Intervall und Zustand im Folgenden oft synonym gebraucht. Wenn die Wahrscheinlichkeiten der Quellsymbole beliebige reelle Werte annehmen können, gibt es also auch überabzählbar unendlich viele mögliche Zustände des Codierers.

Bei der arithmetischen Codierung mit Skalierung (siehe Abschnitt 2.3) werden die Bits der Codierung so früh wie möglich ausgegeben. Durch das Skalieren der Intervallgrenzen wird die Information, die in diesen Bits steckt, aus dem Zustand herausgenommen. Die Anzahl der Zustände des Codierers wird damit reduziert, da alle Intervalle herausfallen, die komplett in einem der Intervalle $[0, \frac{1}{2}), [\frac{1}{2}, 1), [\frac{1}{4}, \frac{3}{4})$ enthalten sind und die damit skaliert werden. Die Intervalle können so also nicht mehr beliebig klein werden, und es ist dadurch möglich, mit endlicher Genauigkeit zu rechnen. Es gibt aber immer noch viel zu viele mögliche Intervallgrenzen, als dass man alle möglicherweise einmal durchzuführenden Berechnungen im Voraus tätigen könnte. Was man haben möchte, ist eine endliche, möglichst kleine Menge von Zuständen.

Zu diesem Zweck geht man von der anderen Seite an die Sache heran und legt einfach die möglichen Intervallgrenzen von vornherein fest. Dazu wählt man ein $N \in \mathbb{N}$ und lässt als Intervallgrenzen nur Vielfache von $\frac{1}{N}$ zu. Um den Umgang mit ihnen zu vereinfachen, kann man das Intervall $[0, 1)$ mit dem Faktor N auf das Intervall $[0, N)$ strecken. Die Intervallgrenzen können dann nur noch ganze Zahlen zwischen 0 und N annehmen. Abbildung 3.2 skizziert diesen Zusammenhang.

Das Prinzip der arithmetischen Codierung wird durch diese Streckung aber nicht verändert. Es handelt sich lediglich um eine Maßnahme, die den Umgang mit den Intervallgrenzen vereinfachen soll. Ganze Zahlen lassen sich in der Regel einfacher darstellen und handhaben als Fließkommazahlen. Ein Intervall $[l, r)$ mit Intervallgrenzen $l, r \in \{0, \dots, N\}$ steht aber weiterhin für das Intervall $[\frac{l}{N}, \frac{r}{N}) \subseteq [0, 1)$.

Um das Skalieren mit den Abbildungen S_1, S_2 und S_3 einfacher zu gestalten, lassen wir für N nur Vielfache von 4 zu. Die oben erwähnten Intervalle $[0, \frac{1}{2}), [\frac{1}{2}, 1), [\frac{1}{4}, \frac{3}{4})$, die zu einer Skalierung führen, haben dann nämlich ebenfalls ganzzahlige Intervallgrenzen.

Die Skalierungsprozeduren lassen sich entsprechend anpassen und sehen dann wie folgt aus:

$$\begin{aligned} S_1 : \left[0, \tfrac{N}{2}\right) &\rightarrow [0, N) \\ x &\mapsto 2x \\[1em] S_2 : \left[\tfrac{N}{2}, N\right) &\rightarrow [0, N) \\ x &\mapsto 2\left(x - \tfrac{N}{2}\right) \\[1em] S_3 : \left[\tfrac{N}{4}, \tfrac{3N}{4}\right) &\rightarrow [0, N) \\ x &\mapsto 2\left(x - \tfrac{N}{4}\right) \end{aligned}$$

Im Fall $N = 4$ bedeutet dies, dass nach evtl. Skalierung nur drei mögliche Intervalle übrig bleiben: $[0, 4)$, $[1, 4)$ und $[0, 3)$. Sie entsprechen den Intervallen $[0, 1)$, $[\frac{1}{4}, 1)$ und $[0, \frac{3}{4})$.

Nun muss aber noch dafür gesorgt werden, dass man mit den wenigen zugelassenen Intervallgrenzen auch auskommt. Zu diesem Zweck werden in jedem Zustand nur bestimmte Wahrscheinlichkeitsverteilungen zugelassen. Die Verteilungen müssen so beschaffen sein, dass die Partitionierung des (dem Zustand entsprechenden) Intervalls nur zu Teilintervallen mit ganzzahligen Intervallgrenzen führt. Anders ausgedrückt muss eine gegebene Wahrscheinlichkeitsverteilung vor dem Codieren durch eine Verteilung ersetzt werden, die, zur Partitionierung herangezogen, nur zu Teilintervallen führt mit ganzzahligen Intervallgrenzen.

Formal bedeutet dies, dass man im Zustand $[l, r)$ als Wahrscheinlichkeiten für die Quellsymbole nur Vielfache von $\frac{1}{r-l}$ zulässt. Das Intervall wird dann beim Codieren des nächsten Quellsymbols genau an einer Grenze $a \in \mathbb{N}$ mit $l < a < r$ aufgeteilt. Die Menge der zugelassenen Wahrscheinlichkeitsverteilungen hängt also direkt von der Länge $r - l$ des dem aktuellen Zustand entsprechenden Intervalls $[l, r)$ ab.

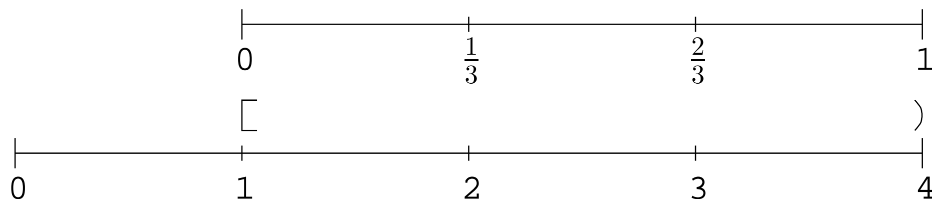

Abbildung 3.3: Beschränkung möglicher Wahrscheinlichkeitsverteilungen

Nehmen wir beispielsweise an, der Codierer befindet sich im Zustand $[1, 4)$. In diesem Zustand sind dann nur Vielfache von $\frac{1}{4-1} = \frac{1}{3}$ als Wahrscheinlichkeiten zugelassen. Das bedeutet, alle evtl. auftretenden Wahrscheinlichkeitsverteilungen müssen auf eine der Verteilungen $\mathcal{P}(\frac{1}{3})$ und $\mathcal{P}(\frac{2}{3})$ abgebildet werden. Je nachdem, welche Partitionierungsreihenfolge (siehe Abschnitt 3.1) und welche der beiden Verteilungen gewählt wird, ergibt sich dann eine Aufteilung des Intervalls $[1, 4)$ an der Stelle 2 oder 3 (siehe Abbildung 3.3).

Wie oben bereits erwähnt, muss eine gegebene Wahrscheinlichkeitsverteilung vor der Codierung eines Symbols zuerst auf eine der zugelassenen Verteilungen abgebildet werden. Denn die Wahrscheinlichkeitsverteilungen, die der Codierer von außen bekommt, unterliegen in der Regel keinen Beschränkungen. Zudem hängen die zugelassenen Verteilungen ja direkt vom Zustand bzw. der Länge des zugehörigen Intervalls ab. Das bedeutet, dass bei den Zustandswechseln während des Codierens die gegebene Verteilung immer wieder auf andere zugelassene Verteilungen abgebildet werden muss, sogar wenn sie selbst immer gleich bleibt.

Die Abbildung wird folgendermaßen vorgenommen. Seien

$$\mathcal{P}(p_1), \mathcal{P}(p_2), \ldots, \mathcal{P}(p_k)$$

die zulässigen Verteilungen in aufsteigender Reihenfolge, d. h. es gilt

$$p_i < p_j \quad \text{für} \quad i < j.$$

Die gegebene Wahrscheinlichkeitsverteilung sei $\mathcal{P}(p)$. Dann werden Zwischenstellen $q_1, q_2, \ldots, q_{k-1}$ bestimmt mit

$$p_i < q_i < p_{i+1} \ \forall i = 1, \ldots, k-1.$$

Zusätzlich werden $q_0 = 0$ und $q_k = 1$ gesetzt. Die gegebene Verteilung $\mathcal{P}(p)$ wird dann abgebildet auf $\mathcal{P}(p_i)$ mit $p \in [q_{i-1}, q_i)$. Der Vorgang ist in Abbildung 3.4 schematisch darstellt.

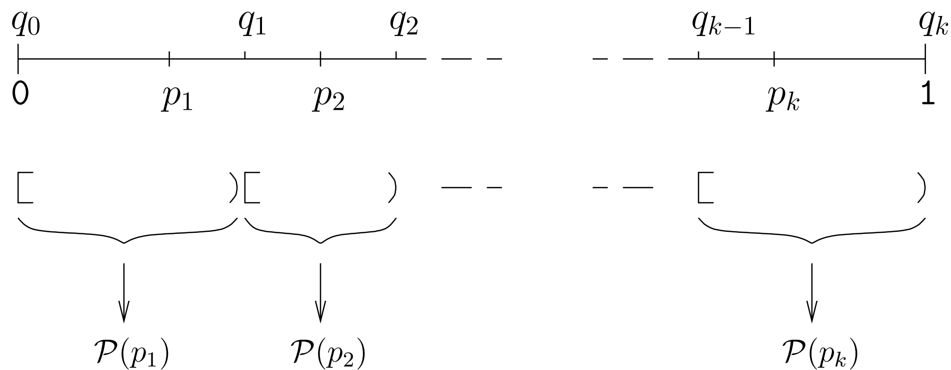

Abbildung 3.4: Abbildung der Wahrscheinlichkeitsverteilungen

An dieser Stelle wird im Vergleich zur idealen arithmetischen Codierung ein Verlust an Kompression in Kauf genommen. Denn die Annahme einer Wahrscheinlichkeitsverteilung, die von der tatsächlichen abweicht, kann sich nur negativ auf die Kompression auswirken. Man möchte natürlich mit der gewählten Wahrscheinlichkeitsverteilung möglichst nah an die gegebene Verteilung herankommen, um den Verlust an Kompression so gering wie möglich zu halten.

Die optimalen Werte für die q_i zu finden ist dabei nicht trivial. Näheres dazu kann man in der Studienarbeit von Christian Soltenborn [5] nachlesen. Nach Howard und Vitter [1] führt aber jede sinnvolle Wahl von Zwischenwerten zu guten Ergebnissen in der Kompression. Man kann z. B. wie in Abbildung 3.4 angedeutet genau die Mittelpunkte zwischen den p_i wählen:

$$q_i = \frac{1}{2}(p_i + p_{i+1}) \;\; \forall i = 1, \ldots, k-1$$

	Eingabe 0		Eingabe 1	
Verteilung	Ausgabe	wechseln zu	Ausgabe	wechseln zu
$\mathcal{P}(p_1)$	$c_{1,0}$	$[l_{1,0}, r_{1,0})$	$c_{1,1}$	$[l_{1,1}, r_{1,1})$
$\mathcal{P}(p_2)$	$c_{2,0}$	$[l_{2,0}, r_{2,0})$	$c_{2,1}$	$[l_{2,1}, r_{2,1})$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
$\mathcal{P}(p_k)$	$c_{k,0}$	$[l_{k,0}, r_{k,0})$	$c_{k,1}$	$[l_{k,1}, r_{k,1})$

Tabelle 3.1: Schema einer Codiertabelle

3.3 Codiertabellen

Kommen wir nun zum Herzstück der tabellenbasierten arithmetischen Codierung: den Tabellen. Wie bereits angesprochen, führt die Reduzierung der möglichen Zustände des Codierers dazu, dass die Menge von Berechnungen, die bei der arithmetischen Codierung durchgeführt werden können, überschaubar wird. Die Berechnungen können dann vorab durchgeführt und ihre Ergebnisse in Tabellen abgelegt werden.

Dazu gibt es für jeden Zustand eine Tabelle, die für jede zugelassene Verteilung einen Eintrag in Form einer Zeile enthält. Die Spalten sind in zwei Hälften aufgeteilt. Zwei Spalten für das Verhalten bei Eingabe **0** und zwei bei Eingabe **1**. Eine Spalte enthält die Ausgabe, die nach der Verarbeitung des entsprechenden Eingabesymbols schon feststeht. Die andere enthält den Zielzustand, also das aktuelle Intervall nach der Verarbeitung. Tabelle 3.1 enthält ein Schema für solch eine Codiertabelle.

Nehmen wir beispielsweise $N = 4$ an. Der Codierer kann sich dann nach evtl. Skalierung der Intervallgrenzen (siehe Abschnitt 2.3) nur in den Zuständen $[0, 4)$, $[1, 4)$ und $[0, 3)$ befinden.

Wie sieht dann die Tabelle für den Anfangszustand $[0, 4)$ aus? Die zugelassenen Wahrscheinlichkeitsverteilungen sind in diesem Zustand $\mathcal{P}(\frac{1}{4})$, $\mathcal{P}(\frac{1}{2})$ und $\mathcal{P}(\frac{3}{4})$. Überlegen wir uns zuerst den Eintrag für die Wahrscheinlichkeitsverteilung $\mathcal{P}(\frac{3}{4})$. Als Partitionierungsreihenfolge (siehe Abschnitt 3.1) wählen wir vorerst immer erst die 0 und dann die 1. Der 0 ist dann also das linke Teilintervall $[0, 3)$ zugeordnet und der 1 das rechte Teilintervall $[3, 4)$.

Ist das Eingabesymbol dann die 0, so würde die arithmetische Codierung mit Skalierung zunächst das Intervall $[0, 3)$ als aktuelles Intervall wählen. Da sich auf dieses Intervall keine Skalierungsprozedur anwenden lässt, kann noch nichts ausgegeben werden und das aktuelle Intervall bleibt $[0, 3)$. In der Co-

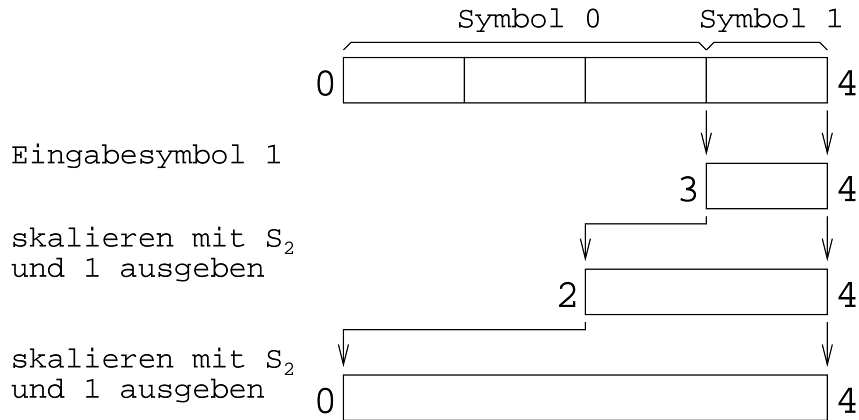

Abbildung 3.5: Reduzierte arithmetische Codierung am Beispiel

diertabelle ist deshalb für die Eingabe 0 keine Ausgabe und als Zielzustand $[0, 3)$ eingetragen.

Für das Eingabesymbol 1 sieht es anders aus. In diesem Fall wird zunächst das Intervall $[3, 4)$ ausgewählt. Dieses Intervall ist komplett in der oberen Hälfte von $[0, 4)$ enthalten. Darum kann eine **1** ausgegeben und die Abbildung S_2 angewendet werden. Nach der Skalierung erhält man das Intervall $[2, 4)$. Auf dieses Intervall lässt sich wiederum die Abbildung S_2 anwenden, was zur erneuten Ausgabe einer **1** und dem Intervall $[0, 4)$ führt. Das Intervall $[0, 4)$ lässt sich offensichtlich nicht mehr skalieren und so ergibt sich in der Codiertabelle für die Eingabe 1 die Ausgabe **11** und der Zielzustand $[0, 4)$. In Abbildung 3.5 ist der Codiervorgang für die Eingabe 1 schematisch dargestellt.

Der Eintrag der Codiertabelle für die Wahrscheinlichkeitsverteilung $\mathcal{P}(\frac{1}{4})$ ergibt sich analog. Bei Eingabe 0 erhält man zuerst das Intervall $[0, 1)$, das zweimal mit der Abbildung S_1 skaliert wird, was zu der Ausgabe **00** und dem Zielzustand $[0, 4)$ führt. Bei Eingabe 1 ergibt sich der Zielzustand $[1, 4)$ und keine Ausgabe, da nicht skaliert werden kann.

Bleibt noch die Verteilung $\mathcal{P}(\frac{1}{2})$ übrig. In diesem Fall wird dem Eingabesymbol 0 das Intervall $[0, 2)$ zugeordnet und dem Eingabesymbol 1 das Intervall $[2, 4)$. Ersteres lässt sich mit der Abbildung S_1 und letzteres mit der Abbildung S_2 zum Intervall $[0, 4)$ skalieren. Also ergibt sich in dem entsprechenden Eintrag der Codiertabelle bei Eingabe 0 die Ausgabe **0** und bei Eingabe 1 die Ausgabe **1**. Der Zielzustand ist in beiden Fällen $[0, 4)$.

Zustand	P(0)	Eingabe 0 Ausgabe	Eingabe 0 wechseln zu	Eingabe 1 Ausgabe	Eingabe 1 wechseln zu
$[0,4)$	$\frac{1}{4}$	00	$[0,4)$	-	$[1,4)$
	$\frac{1}{2}$	0	$[0,4)$	1	$[0,4)$
	$\frac{3}{4}$	-	$[0,3)$	11	$[0,4)$
$[1,4)$	$\frac{1}{3}$	01	$[0,4)$	1	$[0,4)$
	$\frac{2}{3}$	*follow*	$[0,4)$	11	$[0,4)$
$[0,3)$	$\frac{1}{3}$	00	$[0,4)$	*follow*	$[0,4)$
	$\frac{2}{3}$	0	$[0,4)$	10	$[0,4)$

Tabelle 3.2: Reduzierter binärer arithmetischer Codierer

Tabelle 3.2 enthält die oben beschriebene Codiertabelle und der Übersicht halber auch die Codiertabellen für die beiden übrigen Zustände $[1,4)$ und $[0,3)$. Die Verteilungen werden in der Tabelle anhand der Wahrscheinlichkeit $P(0)$ für die 0 identifiziert.

Schauen wir uns nun an, was im Zustand $[1,4)$ passiert. In diesem Zustand sind als Wahrscheinlichkeiten für die Quellsymbole nur Vielfache von $\frac{1}{4-1} = \frac{1}{3}$ zugelassen. Das ergibt die beiden zulässigen Verteilungen $\mathcal{P}(\frac{1}{3})$ und $\mathcal{P}(\frac{2}{3})$. Im vorherigen Abschnitt wurde die Partitionierung des Intervalls $[1,4)$ bereits behandelt und in Abbildung 3.3 dargestellt.

Für die Verteilung $\mathcal{P}(\frac{1}{3})$ wird demnach der 0 das Intervall $[1,2)$ und der 1 das Intervall $[2,4)$ zugeordnet. Das Intervall $[1,2)$ lässt sich mit der Abbildung S_1 zu dem Intervall $[2,4)$ skalieren. Und dieses wiederum lässt sich mit der Abbildung S_2 zum Intervall $[0,4)$ skalieren. Für die Eingabe 0 ergibt sich also die Ausgabe **01** und der Zielzustand $[0,4)$. Für die Eingabe 1 ergibt sich entsprechend die Ausgabe **1** und ebenfalls der Zielzustand $[0,4)$.

Bei der Verteilung $\mathcal{P}(\frac{2}{3})$ kommt es zum ersten Mal zur Anwendung der Skalierungsprozedur S_3. Dem Eingabesymbol 0 wird zunächst das Teilintervall $[1,3)$ zugeordnet und der 1 das Teilintervall $[3,4)$. Das Intervall $[1,3)$ lässt sich weder mit S_1 noch mit S_2 skalieren. Aber die Skalierungsprozedur S_3 ist anwendbar und ergibt das Intervall $[0,4)$. Deshalb wechselt der Codierer bei Eingabe 0 in den Zustand $[0,4)$ und es muss ein noch ausstehendes Ausgabebit (engl. *bit to follow*) vermerkt werden. Dies geschieht durch ein *follow* in der Ausgabe. Bei Eingabe 1 wird das Teilintervall $[3,4)$ durch zweimaliges

Anwenden der Abbildung S_2 zum Intervall $[0, 4)$ skaliert, was zu der Ausgabe **11** und dem Zielzustand $[0, 4)$ führt.

Das Verhalten im Zustand $[0, 3)$ ist analog.

Wir haben mit Tabelle 3.2 also einen ersten Codierer beschrieben, der nach dem Prinzip der arithmetischen Codierung und gleichzeitig nur mit Zustandsübergängen arbeitet.

Die Wahl von $N = 4$ stellt das Minimum dar. Bessere Kompressionsraten werden für größere Werte von N erreicht. Dann ist die Chance größer, dass eine zulässige Wahrscheinlichkeitsverteilung existiert, die nahe an die gegebene herankommt. Natürlich vergrößert sich dann auch die Menge der Zustände, in denen sich der Codierer befinden kann. Deren Anzahl lässt sich in Abhängigkeit von N angeben. Dazu überlegt man sich, dass nach einer eventuellen Skalierung für die untere Intervallgrenze stets $l_i < \frac{N}{2}$ gilt. Man betrachtet nun zwei Fälle:

1. Gilt $l_i \in [0, \frac{N}{4})$, dann folgt daraus, dass für die obere Intervallgrenze $u_i \in [\frac{N}{2}, N)$ gelten muss, da ansonsten skaliert werden würde.
2. Gilt $l_i \in [\frac{N}{4}, \frac{N}{2})$, dann folgt analog, dass für die obere Intervallgrenze $u_i \in [\frac{3N}{4}, N)$ gelten muss.

Das ergibt eine Gesamtanzahl von

$$\frac{N}{4}\frac{N}{2} + \frac{N}{4}\frac{N}{4} = \frac{N^2}{8} + \frac{N^2}{16} = \frac{3N^2}{16}$$

möglichen Intervallen bzw. Zuständen. Man sieht, dass der beste Ansatz für eine Reduzierung der Zustandsmenge die Wahl eines kleinen Wertes für N ist, da er quadratisch als Faktor mit einfließt. Es gibt aber noch weitere Ansätze, die in den folgenden Abschnitten beschrieben werden.

Zustand	P(0)	Eingabe 0 Ausgabe	Eingabe 0 wechseln zu	Eingabe 1 Ausgabe	Eingabe 1 wechseln zu
$[0, 4)$	$\frac{1}{4}$	00	$[0, 4)$	-	$[1, 4)$
	$\frac{1}{2}$	0	$[0, 4)$	1	$[0, 4)$
	$\frac{3}{4}$	-	$[0, 3)$	11	$[0, 4)$
$[1, 4)$	$\frac{1}{3}$	01	$[0, 4)$	1	$[0, 4)$
	$\frac{2}{3}$	1	$[0, 4)$	01	$[0, 4)$
$[0, 3)$	$\frac{1}{3}$	10	$[0, 4)$	0	$[0, 4)$
	$\frac{2}{3}$	0	$[0, 4)$	10	$[0, 4)$

Tabelle 3.3: Vermeidung der *follow*-Bits

3.4 Reduzierung der follow-Bits

In Abschnitt 3.1 wurde bereits darauf eingegangen, dass die Partitionierungsreihenfolge frei gewählt werden kann. Bei den Codiertabellen aus dem vorherigen Abschnitt wurde die Partitionierung stets so gewählt, dass das linke der beiden Teilintervalle der 0 zugeordnet ist und das rechte der 1. Man kann die Reihenfolge aber für jeden Eintrag einzeln wählen, denn sie muss ja nur bei gegebener Wahrscheinlichkeitsverteilung in einem Zustand eindeutig sein, damit der Decodierer später nachvollziehen kann, was der Codierer gemacht hat.

Das lässt sich ausnutzen, um das Vormerken von *follow*-Bits zu vermeiden. Im Beispiel aus Tabelle 3.2 kann man diesen Sachverhalt leicht nachvollziehen. Im Zustand $[1, 4)$ führt eine Vertauschung der Partitionierungsreihenfolge für die Verteilung $\mathcal{P}(\frac{2}{3})$ dazu, dass dann der 0 das Teilintervall $[2, 4)$ und der 1 das Teilintervall $[1, 2)$ zugeordnet ist.

Dies führt zu leicht veränderten Ausgaben. Bei Eingabe 0 wechselt der Codierer dann mit Ausgabe **1** und bei Eingabe 1 mit Ausgabe **01** in den Zustand $[0, 4)$. Die *follow*-Bits sind damit komplett herausgefallen. Wichtig ist, dass sich die Codierungslänge dadurch nicht ändert.

Analog kann man im Zustand $[0, 3)$ vorgehen. Dies führt zum leicht veränderten Codierer aus Tabelle 3.3. Für beliebige N lassen sich die *follow*-Bits damit nicht generell verhindern, aber ihre Anzahl kann verringert werden.

3.5 Alternative Betrachtung des binären Alphabets

Schaut man sich Tabelle 3.3 genauer an, stellt man einige Symmetrien fest. In den Zuständen $[0,3)$ und $[1,4)$ sind für beide möglichen Wahrscheinlichkeitsverteilungen die Ausgaben bei gleicher Eingabe unterschiedlich. Aber für das jeweils wahrscheinlichere Symbol sind sie identisch, genauso wie der Folgezustand. Gleiches gilt für das jeweils unwahrscheinlichere Symbol.

Vertauscht man im Zustand $[0,4)$ entweder für $\mathcal{P}(\frac{1}{4})$ oder für $\mathcal{P}(\frac{3}{4})$ die Partitionierungsreihenfolge, so wie es im letzten Abschnitt beschrieben ist, um *follow*-Bits einzusparen, dann werden auch dort die Symmetrien noch deutlicher, wie man an folgender Tabelle für den Zustand $[0,4)$ beobachten kann.

	Eingabe 0		Eingabe 1	
P(0)	Ausgabe	wechseln zu	Ausgabe	wechseln zu
$\frac{1}{4}$	11	$[0,4)$	-	$[0,3)$
$\frac{1}{2}$	0	$[0,4)$	1	$[0,4)$
$\frac{3}{4}$	-	$[0,3)$	11	$[0,4)$

Es fällt dann sogar noch ein Zustand ganz heraus, denn $[1,4)$ taucht nicht mehr als Zielzustand auf und ist deswegen überflüssig.

Diese Betrachtungen führen uns zu einer Idee, die von Langdon und Rissanen [6] beschrieben wird. Sie führen für das Quellalphabet $\{0,1\}$ eine alternative Identifizierung der Eingabesymbole ein. Dabei steht MPS (engl. *more probable symbol*) für das in der aktuellen Verteilung wahrscheinlichere Symbol und LPS (engl. *less probable symbol*) für das weniger wahrscheinliche Symbol. In den Codiertabellen können die zulässigen Wahrscheinlichkeitsverteilungen dann z. B. durch die Wahrscheinlichkeit $P(MPS)$ für das wahrscheinlichere Symbol identifiziert werden.

Alle Einträge in den Codiertabellen sind dann durch eine Wahrscheinlichkeit $P(MPS) \geq \frac{1}{2}$ gekennzeichnet, da das wahrscheinlichere von zwei Symbolen keine Wahrscheinlichkeit kleiner $\frac{1}{2}$ besitzen kann. Einträge, die vorher durch eine Wahrscheinlichkeit $P(0) < \frac{1}{2}$ gekennzeichnet waren, können so eingespart werden. In Tabelle 3.4 ist dieses Verfahren auf unseren Beispielcodierer für $N = 4$ angewandt. Man kann hier beobachten, dass die Symmetrien, die man in den Tabellen 3.2 und 3.3 beobachten kann, verschwunden sind.

Zustand	P(MPS)	Eingabe MPS Ausgabe	Eingabe MPS wechseln zu	Eingabe LPS Ausgabe	Eingabe LPS wechseln zu
$[0,4)$	$\frac{1}{2}$	0	$[0,4)$	1	$[0,4)$
	$\frac{3}{4}$	-	$[0,3)$	11	$[0,4)$
$[0,3)$	$\frac{2}{3}$	0	$[0,4)$	10	$[0,4)$

Tabelle 3.4: Verwendung von MPS und LPS

3.6 Ausgewählte Verteilungen

Besonders gute Kompressionsraten lassen sich natürlich für schiefe Wahrscheinlichkeitsverteilungen erreichen. Denn je schiefer die Verteilung, desto weniger Bits sind nötig, um wahrscheinliche Symbole zu codieren. Wenn man nun weiß, dass überwiegend schiefe Verteilungen auftreten, kann man sich auf solche beschränken und so weitere Zustandsübergänge einsparen. Dies führt im Fall $N = 4$ zu folgendem einfachen Codierer, in dem die Gleichverteilung eingespart wird:

Zustand	Eingabe MPS Ausgabe	Eingabe MPS wechseln zu	Eingabe LPS Ausgabe	Eingabe LPS wechseln zu
$[0,4)$	-	$[0,3)$	11	$[0,4)$
$[0,3)$	0	$[0,4)$	10	$[0,4)$

Auch im allgemeinen Fall kann man sich natürlich auf wenige Wahrscheinlichkeitsverteilungen beschränken, um Zustandsübergänge einzusparen. So kann man einen großen Wert für N wählen um möglichst schiefe Verteilungen zu erhalten und gleichzeitig die Anzahl der Zustände überschaubar halten, indem man nur ausgewählte Verteilungen zulässt.

Howard und Vitter [1] nutzen diesen Ansatz, um einen Codierer zu entwickeln, der mit $\frac{N}{2}$ Zuständen auskommt, die alle von der Form $[k, N)$ für ein ganzzahliges $k \in [0, \frac{N}{2})$ sind. Alle zugelassenen Wahrscheinlichkeitsverteilungen müssen dann natürlich so gewählt werden, dass die Zielzustände auch wieder von der Form $[k, N)$ sind.

In jedem Zustand wird zuerst eine maximal schiefe Verteilung eingeführt, die das Intervall $[k, N)$ an der Stelle $k + 1$ splittet. Für die Eingabe LPS ergibt sich damit ein Intervall der Länge 1. Damit sich dieses auf jeden Fall zu einem

$k \in$	Partition		Eingabe LPS		Eingabe MPS	
	LPS	MPS	Ausgabe	gehe zu	Ausgabe	gehe zu
$[0, \frac{N}{2})$	$[k, \frac{N}{2})$	$[\frac{N}{2}, N)$	0	$2k$	1	0
$[0, \frac{N}{4})$	$[k, \frac{N}{4})$	$[\frac{N}{4}, N)$	00	$4k$	-	$\frac{N}{4}$
$[\frac{N}{8}, \frac{N}{4})$	$[k, \frac{3N}{8})$	$[\frac{3N}{8}, N)$	$0f$	$4k - \frac{N}{2}$	-	$\frac{3N}{8}$
$[\frac{N}{4}, \frac{3N}{8})$	$[k, \frac{3N}{8})$	$[\frac{3N}{8}, N)$	010	$8k - 2N$	-	$\frac{3N}{8}$
$[\frac{3N}{8}, \frac{N}{2})$	$[k, \frac{5N}{8})$	$[\frac{5N}{8}, N)$	ff	$4k - \frac{3N}{2}$	1	$\frac{N}{4}$
$[\frac{7N}{16}, \frac{N}{2})$	$[k, \frac{9N}{16})$	$[\frac{9N}{16}, N)$	fff	$8k - \frac{7N}{2}$	1	$\frac{N}{8}$
$[\frac{N}{4}, \frac{N}{2})$	$[\frac{3N}{4}, N)$	$[k, \frac{3N}{4})$	11	0	f	$2k - \frac{N}{2}$

Tabelle 3.5: Erzeugungstabelle für Klasse von reduzierten Codierern

Intervall der Form $[k, N)$ skalieren lässt, fordern wir, dass N nicht mehr nur ein Vielfaches von 4 (siehe Abschnitt 3.2), sondern sogar eine Zweierpotenz ist. Damit ergibt sich bei Eingabe LPS eine Ausgabe der Länge $log(N)$ und der Zielzustand $[0, N)$. Für die Eingabe MPS wechselt der Codierer ohne Ausgabe in den Zustand $[k+1, N)$. Es sei denn, es gilt $k+1 = \frac{N}{2}$, denn dann wird mit der Abbildung S_2 skaliert. Der Codierer wechselt dann mit Ausgabe **1** in den Zustand $[0, N)$. Hier lässt sich bereits feststellen, dass alle Intervalle $[k, N)$ mit ganzzahligem $k \in [0, \frac{N}{2})$ auch als Zielzustände auftreten.

Weitere Verteilungen ergeben sich aus der Tabelle 3.5. Dabei wird ein Zustand $[k, N)$ der Einfachheit halber mit k bezeichnet. Ein f in der Ausgabe steht für das Vormerken eines *follow*-Bits. Es sei angemerkt, dass es sich hierbei nicht um eine Codiertabelle wie in den vorherigen Beispielen handelt. Tabelle 3.5 ist vielmehr eine Vorschrift zur Erzeugung der Codiertabellen.

Für ein gegebenes $N = 2^l, 2 < l \in \mathbb{N}$ muss dazu für jedes $\mathbb{N} \ni k \in [0, \frac{N}{2})$ ein Zustand erzeugt werden. Aus $l > 2$ folgt $N \geq 8$. Diese Bedingung ist nötig, da in Tabelle 3.5 bei der Berechnung des Zielzustandes Vielfache von $\frac{N}{8}$ auftauchen, die Intervallgrenzen aber ganze Zahlen sein müssen.

Die Einträge der zugehörigen Codiertabelle ergeben sich dann aus der maximal schiefen Verteilung, wie sie oben beschrieben ist und durch die Zeilen der Tabelle 3.5, die für das k passend sind (siehe erste Spalte der Tabelle). Dabei kann es vorkommen, dass mehrere Einträge dieselbe Wahrscheinlichkeitsverteilung beschreiben. In diesem Fall wird nur der erste von ihnen hinzugenommen. Außerdem kann es sein, dass ein Zielzustand angegeben ist mit $k \geq \frac{N}{2}$.

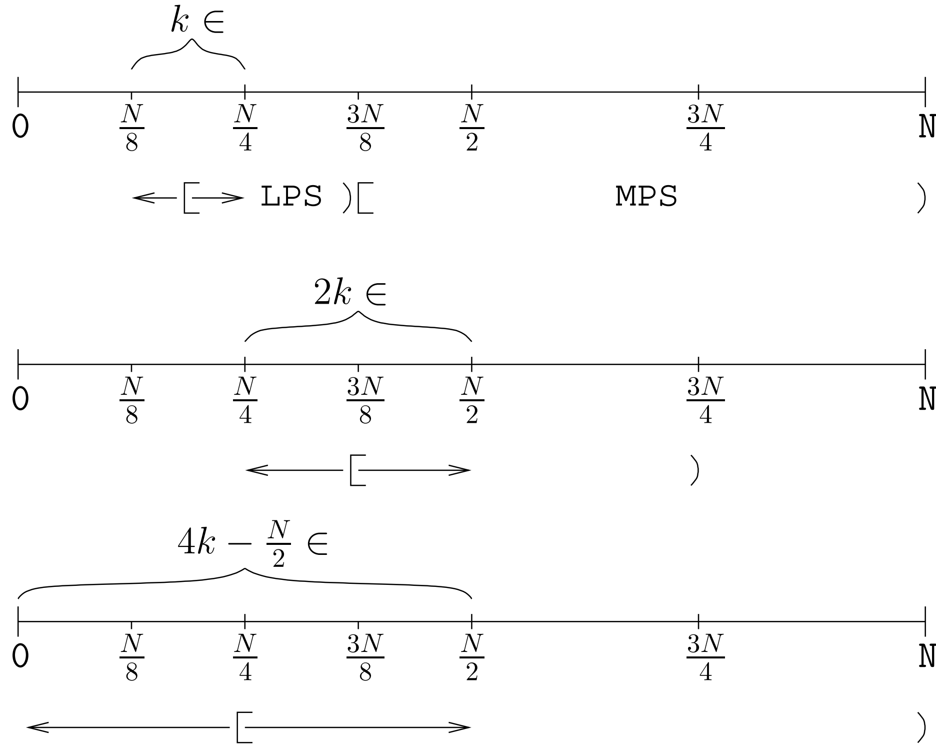

Abbildung 3.6: Anwendung der Erzeugungstabelle

Dann muss noch mit der Skalierungsprozedur S_2 (evtl. mehrmals) skaliert werden und (entsprechend oft) eine **1** an die Ausgabe angehängt werden.

Schauen wir uns zum Beispiel die dritte Zeile der Tabelle 3.5 an, um zu verstehen, wie die Erstellung einer Codiertabelle für einen Zustand $[k, N)$ funktioniert. Seien dazu N und k so, dass $k \in [\frac{N}{8}, \frac{N}{4})$, damit die dritte Zeile passend ist. Abbildung 3.6 soll die Situation veranschaulichen.

Wie aus der zweiten und dritten Spalte zu ersehen ist, soll das Intervall $[k, N)$ in die Teilintervalle $[k, \frac{3N}{8})$ und $[\frac{3N}{8}, N)$ partitioniert werden. Daraus resultiert für den neuen Eintrag in der Codiertabelle eine Wahrscheinlichkeitsverteilung mit

$$P(MPS) = \frac{N - \frac{3N}{8}}{N - k} = \frac{\frac{5N}{8}}{N - k}.$$

Da sich das Intervall $[\frac{3N}{8}, N)$ nicht skalieren lässt, ergibt sich für die Eingabe MPS keine Ausgabe und der Zielzustand $[\frac{3N}{8}, N)$.

Aber es gilt $[k, \frac{3N}{8}) \subseteq [0, \frac{N}{2})$, und deswegen kann bei Eingabe LPS das Intervall $[k, \frac{3N}{8})$ mit S_1 skaliert werden zu $[2k, \frac{6N}{8})$. Aus $k \in [\frac{N}{8}, \frac{N}{4})$ folgt $2k \in [\frac{N}{4}, \frac{N}{2})$

und damit gilt $[2k, \frac{6N}{8}) \subseteq [\frac{N}{4}, \frac{3N}{4})$. Also kann mit S_3 skaliert werden, was zum Intervall $[2(2k - \frac{N}{4}), 2\frac{6N-2}{8}) = [4k - \frac{N}{2}, N)$ führt. Bei Eingabe LPS ergibt sich also der Zielzustand $4k - \frac{N}{2}$ und durch das Skalieren mit S_1 und S_3 die Ausgabe $\mathbf{0}\boldsymbol{f}$.

3.7 Beispielcodierer

Im Folgenden wird nach dem Verfahren aus dem vorherigen Abschnitt ein Codierer für $N = 2^3 = 8$ entwickelt. Die auftretenden Zustände sind $[0, 8)$, $[1, 8)$, $[2, 8)$ und $[3, 8)$. Fangen wir mit dem Ausgangszustand $[0, 8)$ an, also mit $k = 0$.

Zuerst fügen wir die maximal schiefe Verteilung hinzu, die das Intervall $[0, 8)$ in die Teilintervalle $[0, 1)$ und $[1, 8)$ partitioniert. Die Verteilung ist dann bestimmt durch die Wahrscheinlichkeit $P(MPS) = \frac{7}{8}$. Für das Eingabesymbol MPS wird der Codierer ohne Ausgabe in den Zustand $[1, 8)$ wechseln. Ist das Eingabesymbol LPS, dann kann das Teilintervall $[0, 1)$ durch dreimaliges Anwenden der Abbildung S_1 auf das Intervall $[0, 8)$ skaliert werden. Dies führt zur Ausgabe **000** und dem Zielzustand $[0, 8)$.

Für die weiteren Einträge der Codiertabelle wird nun k mit den Einträgen der Tabelle 3.5 verglichen.

Die erste Zeile ist passend, da $k = 0 \in [0, 4) = [0, \frac{N}{2})$. Also wird die Verteilung mit $P(MPS) = \frac{1}{2}$ hinzugenommen, die das Intervall $[0, 8)$ in die Teilintervalle $[0, 4)$ (für LPS) und $[4, 8)$ (für MPS) partitioniert. Bei Eingabe LPS lässt sich aus Tabelle 3.5 die Ausgabe **0** und der Zielzustand $[2k, 8) = [0, 8)$ ablesen. Das macht auch Sinn, denn das Teilintervall $[0, 4)$ lässt sich mit Abbildung S_1 auf das Intervall $[0, 8)$ skalieren. Bei Eingabe MPS ergibt sich die Ausgabe **1** und ebenfalls der Zielzustand $[0, 8)$.

Die zweite Zeile von Tabelle 3.5 passt auch, da $k = 0 \in [0, 2) = [0, \frac{N}{4})$. Also wird als nächstes die Verteilung mit $P(MPS) = \frac{3}{4}$ hinzugenommen. Die zugehörige Partitionierung teilt das Intervall $[0, 8)$ in die Teilintervalle $[0, 2)$ (für LPS) und $[2, 8)$ (für MPS) auf. Aus der Tabelle 3.5 lässt sich für die Eingabe LPS der Zielzustand $[4k, N) = [0, 8)$ und die Ausgabe **00** ablesen, was der zweimaligen Anwendung von S_1 entspricht. Bei Eingabe MPS wechselt der Codierer ohne Ausgabe in den Zustand $[\frac{N}{4}, N) = [2, 8)$.

Da alle weiteren Einträge aus Tabelle 3.5 für $k = 0$ nicht passen, ist damit die Erstellung der Codiertabelle für den Zustand $[0, 8)$ abgeschlossen. Für die anderen Zustände geht man nach demselben Schema vor und erhält dann den Codierer aus Tabelle 3.6.

Zustand	P(MPS)	Eingabe LPS		Eingabe MPS	
		Ausgabe	wechseln zu	Ausgabe	wechseln zu
$[0,8)$	$\frac{7}{8}$	000	$[0,8)$	-	$[1,8)$
	$\frac{3}{4}$	00	$[0,8)$	-	$[2,8)$
	$\frac{1}{2}$	0	$[0,8)$	1	$[0,8)$
$[1,8)$	$\frac{6}{7}$	001	$[0,8)$	-	$[2,8)$
	$\frac{5}{7}$	$0f$	$[0,8)$	-	$[3,8)$
	$\frac{4}{7}$	0	$[2,8)$	1	$[0,8)$
$[2,8)$	$\frac{5}{6}$	010	$[0,8)$	-	$[3,8)$
	$\frac{2}{3}$	01	$[0,8)$	1	$[0,8)$
$[3,8)$	$\frac{4}{5}$	011	$[0,8)$	1	$[0,8)$
	$\frac{3}{5}$	ff	$[0,8)$	1	$[2,8)$

Tabelle 3.6: Beispielcodierer für N=8

Die Generierung der Codiertabellen für die restlichen Zustände gelingt allerdings in einigen Fällen nicht ganz so reibungslos. Im vorigen Abschnitt wurde ja bereits darauf aufmerksam gemacht, dass ein Eintrag in der Erzeugungstabelle zu einer Verteilung führen kann, die bereits in die Codiertabelle aufgenommen wurde. Diese Einträge werden dann einfach ausgelassen.

Im Zustand $[1,8)$ passt die zweite Zeile der Tabelle 3.5 zwar, weil $k = 1 \in [0,2) = [0, \frac{N}{4})$, aber es ergibt sich daraus die Verteilung mit $P(MPS) = \frac{6}{7}$. Die wurde aber bereits als maximal schiefe Verteilung mit aufgenommen.

Gleiches gilt im Zustand $[2,8)$ für die vierte Zeile und $P(MPS) = \frac{5}{6}$. Und die letzte Zeile im Zustand $[2,8)$ findet keine Anwendung, weil sie die gleiche Verteilung produziert wie die erste. Lediglich die Partitionierungsreihenfolge ist dort vertauscht. Im Zustand $[3,8)$ ist es ähnlich.

Der Zustand $[2,8)$ ist auch ein Beispiel für das bereits erwähnte Nachskalieren. Aus der ersten Zeile der Erzeugungstabelle ergibt sich dort der Eintrag für die Verteilung mit $P(MPS) = \frac{2}{3}$. Für die Eingabe LPS ergibt sich so zunächst die Ausgabe **0** und der Zielzustand $[4,8) = [2k, N)$. Dieser muss dann noch mit der Abbildung S_2 zu dem Intervall $[0,8)$ skaliert werden. Erst dadurch ergibt sich der Eintrag aus Tabelle 3.6 mit Ausgabe **01** und Zielzustand $[0,8)$.

4 Decodierung tabellenbasierter arithmetischer Codierung

Im Folgenden soll gezeigt werden, wie die im letzten Kapitel entwickelte tabellenbasierte arithmetische Codierung rückgängig gemacht werden kann. Bei der Decodierung liegen dieselben Tabellen zugrunde, wie bei der Codierung. Außerdem muss der Decodierer natürlich zu jedem Zeitpunkt über die gleiche Wahrscheinlichkeitsverteilung verfügen wie der Codierer. Dann kann der Decodierer anhand der Codierung die Abfolge der Zustände nachvollziehen, die der Codierer durchlaufen hat und so die Eingabe rekonstruieren.

4.1 Beschränkte Betrachtung der Codierung

Da der Codierer nur eine beschränkte Anzahl möglicher Intervallgrenzen zulässt, reicht es aus, wenn der Decodierer nur eine beschränkte Anzahl Bits der Codierung auf einmal betrachtet.

Beim Codieren werden ja nur solche Wahrscheinlichkeitsverteilungen zugelassen, dass die Intervallgrenzen, die den Zustand des Codierers beschreiben, Vielfache von $\frac{1}{N}$ sind. Wählt man nun $N = 2^k$ mit $2 \leq k \in \mathbb{N}$, dann benötigt man nur k Bits der Codierung, um das nächste Symbol zu decodieren. Denn die ersten $k = \log_2 N$ Bits der Codierung beschreiben gerade, in welchem der Intervalle $[\frac{i-1}{N}, \frac{i}{N})$ für $i = 1, \ldots, N$ die Codierung liegt. (Das erste Bit der Codierung beschreibt, ob die Codierung in der oberen oder unteren Hälfte des Intervalls $[0, N)$ liegt, die ersten beiden, in welchem Viertel, usw.)

Diese Intervalle lassen sich für einen Zustand bei gegebener Wahrscheinlichkeitsverteilung eindeutig einem Ausgabesymbol (bzw. einem Eingabesymbol des Codierers) zuordnen. Denn die Partitionierung des aktuellen Intervalls in die beiden Teilintervalle für LPS und MPS ist ja nach Voraussetzung bei gegebener Wahrscheinlichkeitsverteilung in jedem Zustand eindeutig.

Damit Codierer und Decodierer mit den gleichen Tabellen arbeiten können, gehen wir davon aus, dass in den Codiertabellen auch gespeichert ist, welches Teilintervall LPS und welches MPS zugeordnet ist. Diese Information steckt

ohnehin schon in den Tabellen. Man könnte nämlich auch ausgehend vom Zielzustand anhand der eingetragenen Ausgabe die Skalierungsprozeduren S_1, S_2 und S_3 zurückrechnen, um die Teilintervalle für LPS und MPS zu erhalten. Es wäre allerdings nicht sehr effizient, dieses bei der Decodierung jedes Symbols zu tun.

Die Forderung nach einer Zweierpotenz für N galt bereits für die Codierer, die auf der Erzeugungstabelle von Howard und Vitter [1] (siehe Abschnitt 3.6) beruhen. Für diese Codierer stellt sie also keine weitere Einschränkung dar.

4.2 Skalierungen rekonstruieren

Wie beim Algorithmus *Decode-AC* aus Abschnitt 2.3 rücken durch die Skalierung weitere Bits nach, indem die betrachteten k Bits nach links geschoben werden.

Nachdem der Decodierer anhand der Codiertabelle das nächste Symbol entziffert hat, steht die zugehörige Ausgabe, die der Codierer für das Symbol gemacht hat fest. Die nächsten Bits der Codierung müssen sich also mit dem Eintrag in der Codiertabelle decken.

Die **0**en und **1**en in der Ausgabe können also einfach nach links heraus geschoben werden. Der Vorgang ist der gleiche, wie im Algorithmus *Decode-AC*. Die Skalierungen, die sich anhand der Ausgabebits ablesen lassen, werden einfach auch auf die Codierung angewendet, um sie den neuen Intervallgrenzen anzupassen. Jede **0** oder **1** steht für die Anwendung der Abbildung S_1 bzw. S_2. Also kann die Codierung um so viele Stellen nach links geschoben werden, wie **0**en und **1**en in der Codiertabelle als Ausgabe (für das entzifferte Symbol) notiert sind.

Jedes $\boldsymbol{f}$ in der Ausgabe steht für die Anwendung der Abbildung S_3. Um sie korrekt auch auf die Codierung anzuwenden, wird genau wie im Algorithmus *Decode-AC* die Codierung ab der zweiten Stelle entsprechend viele Positionen nach links geschoben. Das entspricht einem Löschen aus der Codierung von den *follow*-Bits, die in der Codiertabelle eingetragen sind. Denn die werden ja immer hinter dem nächsten Symbol ausgegeben, befinden sich also immer eine Stelle hinter der aktuellen Position in der Codierung.

4.3 Zustandsübergänge nachvollziehen

Nachdem dem der Decodierer das nächste Symbol ermittelt hat und evtl. neue Bits nachgerückt sind, wechselt er genau wie der Codierer den Zustand. Der ergibt sich als der Zielzustand, der für das decodierte Symbol und die gegebene Wahrscheinlichkeitsverteilung in der Codiertabelle eingetragen ist.

Dann kann der Decodierer mit der Decodierung des nächsten Symbols fortfahren.

4.4 Decodier-Beispiele

Nehmen wir an, **0010...** seien die ersten Bits einer Codierung, die durch Anwendung des Beispielcodierers aus Tabelle 3.6 erzeugt worden ist. Der Decodierer startet dann im Zustand $[0, 8)$ und liest die ersten $\log 8 = 3$ Bits **001**.

Wir nehmen an, die aktuelle Wahrscheinlichkeitsverteilung wird auf den ersten Eintrag in der Tabelle abgebildet. Damit ist das Intervall $[0, 1)$ LPS und das Intervall $[1, 8)$ MPS zugeordnet.

Es gilt $001_2 = 1_{10}$ und darum liegt die Codierung im Intervall $[1, 8)$. Als erstes Symbol wird demnach das MPS der aktuellen Verteilung decodiert. Da in dem Eintrag für die Eingabe MPS keine Ausgabe eingetragen ist, werden auch keine Shifts auf der Codierung ausgeführt. Der nächste Zustand lässt sich aus der Tabelle als $[1, 8)$ ablesen.

Der Decodierer befindet sich dann also im Zustand $[1, 8)$ und erhält als die ersten 3 Bits der Codierung wiederum **001**. Die Codierung ist demzufolge die gleiche, wenn beim Codieren statt im Zustand $[0, 8)$ im Zustand $[1, 8)$ begonnen und das erste Symbol ausgelassen worden wäre.

Wir nehmen diesmal an, dass die aktuelle Verteilung auf den zweiten Tabelleneintrag im Zustand $[1, 8)$ abgebildet wird. Daraus ergibt sich das LPS-Teilintervall $[1, 3)$ und das MPS-Teilintervall $[3, 8)$.

Aus $1 \in [1, 3)$ folgt nun, dass das nächste Symbol das LPS-Symbol der aktuellen Verteilung ist. Die zugehörige Ausgabe ist $\mathbf{0}\boldsymbol{f}$. Darum ist auch das erste Bit der Codierung die **0**. Deshalb wird die Codierung zunächst um eine Position nach links geschoben. Die Codierung wird dadurch verändert zu **010...** .

Danach wird noch das *follow*-Bit aus der Codierung entfernt. Dazu wird die Codierung ab der zweiten Position um eine Stelle nach links geschoben. Die Codierung lautet danach **00...** .

Der Zielzustand ergibt sich danach als $[0, 8)$.

Danach ist die Codierung so umgeformt, als wären die ersten beiden Symbole nie codiert worden. Alle Ausgabe- und *follow*-Bits, die durch die Codierung der ersten beiden Symbole entstanden sind, wurden aus der Codierung entfernt.

Der Decodierer kann auf diese Weise Symbol für Symbol anhand der Codiertabelle und der aktuellen Wahrscheinlichkeitsverteilung decodieren.

5 Implementierung in Java

Auf der beiliegenden CD befindet sich die Java-Implementierung eines tabellenbasierten binären arithmetischen Codierers, der auf der Erzeugungstabelle von Howard und Vitter [1] (siehe Abschnitt 3.6) basiert.

In Tabelle 5.1 sind alle Klassen und Interfaces aufgelistet und mit einer kurzen Beschreibung versehen. Das `BAC` im Klassennamen steht für *binary arithmetic coding*.

5.1 Die Interfaces

Die Implementierung enthält zwei Interfaces. Das Interface `BACInput` dient zum schrittweisen Lesen der Codierung während des Decodiervorgangs. Die Schnittstelle sieht folgendermaßen aus:

```
interface BACInput {
    int getBits(int amount);
    void shift(int amount);
    void followShift(int amount);
    void finishInput();
}
```

Dabei wird von einer Implementierung folgendes erwartet:

- Ein Aufruf von `getBits(amount)` liefert von der aktuellen Position in der Codierung an den Wert der nächsten `amount` Bits. Lauten die nächsten Bits beispielsweise **100100100**, dann liefert ein Aufruf von `getBits(3)` den Wert $100_2 = 4_{10}$ zurück. Die aktuelle Position innerhalb der Codierung wird dadurch aber nicht verändert.
- Ein Aufruf von `shift(amount)` schiebt die Codierung um `amount` Bitpositionen nach links. Anders ausgedrückt, wird die aktuelle Position in der Codierung um `amount` Bits weitergesetzt. Die `shift`-Methode dient zum Skalieren der Codierung mit den Abbildungen S_1 und S_2.

Name	Typ	Beschreibung
`BACInput`	Interface	Schnittstelle zum Lesen der Codierung (beim Decodieren)
`BACOutput`	Interface	Schnittstelle zur Ausgabe der Codierung (beim Codieren)
`BACSymbol`	Klasse	Symbol (MPS oder LPS)
`BACEntry`	Klasse	Eintrag in einer Codiertabelle (für eine bestimmte Verteilung)
`BACEntryComparator`	Klasse	`Comparator`-Klasse zum Sortieren der `BACEntry`-Objekte nach ihrer MPS-Wahrscheinlichkeit
`BACState`	Klasse	Zustand der Form $[k, N)$ inklusive Codiertabelle und Methoden zum Codieren und Decodieren
`BACTable`	Klasse	Tabellenbasierter Codierer/Decodierer

Tabelle 5.1: Übersicht Klassen/Interfaces

- Ein Aufruf von `followShift(amount)` schiebt die Codierung ab der Stelle hinter der aktuellen Position um `amount` Bitpositionen nach links. Anders ausgedrückt, wird die aktuelle Position in der Codierung um `amount` Bits weitergesetzt, aber das erste Bit an der aktuellen Position bleibt gleich. Sind die nächsten Bits beispielsweise wieder **100100100**, dann ist nach einem Aufruf von `followShift(2)` die Codierung gleich **1100100...**. Die `followShift`-Methode dient zum Skalieren der Codierung mit der Abbildung S_3, bzw. zum Entfernen von *follow*-Bits aus der Codierung.

- `finishInput()` wird am Ende des Decodiervorgangs aufgerufen. Hier können, falls nötig, abschließende Operationen implementiert werden.

Das Interface `BACOutput` dient zur Ausgabe der Codierung während des Codiervorgangs. Die Schnittstelle sieht aus wie folgt:

```
interface BACOutput {
    void output(int bit);
    void increaseBitsToFollow(int amount);
    void finishOutput();
}
```

Eine Implementierung muss folgendes berücksichtigen:

- `output(bit)` wird während des Codiervorgangs aufgerufen, um die Bits der Codierung auszugeben. Der `int`-Wert `bit` nimmt dabei erwartungsgemäß den Wert 0 oder 1 an.

- `increaseBitsToFollow(amount)` wird während des Codierens aufgerufen, wenn *follow*-Bits in der Ausgabe der Codiertabelle stehen. `amount` ist dabei die Anzahl der *follow*-Bits, um die die Anzahl vorgemerkter *follow*-Bits erhöht werden soll. Beim nächsten Aufruf von `output(bit)` sollen dann alle vorgemerkten *follow*-Bits verarbeitet werden.

- `finishOutput()` wird am Ende des Codiervorgangs aufgerufen. Hier können, falls nötig, abschließende Operationen implementiert werden.

5.2 Die Klassen

In der Implementierung finden sich mehrere Klassen. Um den tabellenbasierten Codierer zu nutzen muss man außer den Interfaces aus dem vorigen Abschnitt nur die Klasse `BACTable` (siehe Abschnitt 5.2.5) kennen. Alle anderen Klassen werden nur intern zur Implementierung von `BACTable` benötigt. In den nachfolgenden Abschnitten werden alle Klassen beschrieben.

5.2.1 BACSymbol

Die Klasse `BACSymbol` ist im Prinzip eine Wrapper-Klasse für `int`-Werte. Sie verfügt zusätzlich zum Konstruktor und den get- und set-Methoden für den `int`-Wert noch über drei Konstanten `LPS=0`, `MPS=1` und `UNDEFINED=-1`.

Während des Codiervorgangs wird ein `BACSymbol`-Objekt mit `BACSymbol.LPS` oder `BACSymbol.MPS` initialisiert, bevor es als Parameter an eine Codiermethode übergeben wird.

Beim Decodiervorgang wird ein `BACSymbol`-Objekt erzeugt und dabei mit `BACSymbol.UNDEFINED` initialisiert. Das wird dann als Parameter an eine Decodiermethode übergeben. Die Decodiermethode überschreibt den gespeicherten Wert dann je nach decodiertem Symbol mit `BACSymbol.LPS` oder `BACSymbol.MPS`.

5.2.2 BACEntry

Die Klasse `BACEntry` beschreibt einen Eintrag in der Codiertabelle eines bestimmten Zustands. Sie enthält auch gleich die Methoden um mit dem Eintrag ein Symbol zu codieren und zu decodieren. Werfen wir zunächst einen Blick auf die Schnittstelle:

```
class BACEntry {
    BACEntry(int ik, int isplit, int iN, int soMPS,
             int outBL, int outLL, int btfL, int dSL,
             int outBM, int outLM, int btfM, int dSM);
    int getLPSRange();
    int getMPSRange();
    int encode(BACSymbol symbol, BACOutput out);
    int decode(BACSymbol symbol, BACInput in);
    String toString();
}
```

Die Parameter des Konstruktors sind unter anderem die Initialwerte für die untere und obere Intervallgrenze des zugehörigen Zustands, die Stelle, an der das Intervall in die beiden Teilintervalle für LPS und MPS partitioniert wird und welches der beiden MPS zugeordnet ist. Die restlichen Parameter enthalten Informationen über die Ausgabe und den Zielzustand bei Eingabe LPS bzw. MPS. Eine genauere Beschreibung kann den `javadoc`-Dateien entnommen werden.

Die Methoden `getLPSRange()` und `getMPSRange()` liefern die Länge des jeweiligen Teilintervalls der Partitionierung zurück und dienen in der Klasse `BACState` dazu, die Zwischenwerte zu berechnen, die nötig sind, um einer beliebigen Wahrscheinlichkeitsverteilung einen Eintrag der Codiertabelle zuzuordnen (siehe dazu auch Abschnitt 3.2 und Abbildung 3.4).

`encode(s, out)` codiert das übergebene Symbol `s` mit diesem Codiertabelleneintrag und übergibt die eingetragenen Ausgabe- und *follow*-Bits an das `BACOutput`-Objekt `out`. Danach liefert `encode(s, out)` den eingetragenen Zielzustand zurück. Da die Codiertabellen nach der Erzeugungsvorschrift von Howard und Vitter [1] (siehe Abschnitt 3.6) erstellt werden, haben alle Zustände die Form $[k, N)$ für ein vorher festgelegtes N. Daher wird nur der Wert k zurückgeliefert.

Mit `decode(s, inp)` wird das nächste Symbol decodiert. Dazu werden zuerst mit dem `BACInput`-Objekt `inp` die nächsten $\log N$ Bits der Codierung gelesen. Der Wert der gelesenen Bits wird mit der Stelle verglichen, an der das Intervall des zum Eintrag gehörenden Zustands in die beiden Teilintervalle aufgeteilt wird. So wird bestimmt, ob die Codierung im LPS- oder im MPS-Intervall liegt. Das entsprechende Symbol wird im übergebenen `BACSymbol`-Objekt `s` gespeichert. Danach werden mit dem `BACInput`-Objekt `inp` entsprechend der eingetragenen Ausgabe noch Shifts durchgeführt und *follow*-Bits aus der Codierung entfernt. Schließlich wird wie beim Codieren der eingetragene Zielzustand zurückgeliefert.

Die Methode `toString()` liefert alle relevanten Informationen über den Tabelleneintrag in einem String zurück. Sie ist nur für Debuggingzwecke gedacht.

5.2.3 BACEntryComparator

`BACEntryComparator` ist eine `Comparator`-Klasse. Damit ist gemeint, sie implementiert das Interface `java.util.Comparator`. Zwei `BACEntry`-Objekte werden mit ihr anhand der Größe des Teilintervalls verglichen, das sie dem MPS zuordnen. In der Klasse `BACState` werden damit die Einträge der Codiertabelle aufsteigend nach ihrer MPS-Wahrscheinlichkeit sortiert.

5.2.4 BACState

Die Klasse `BACState` beschreibt einen Zustand des Codierers. Dazu gehören neben der unteren und oberen Intervallgrenze des zugehörigen Intervalls auch die Codiertabelleneinträge.

Es folgt zunächst die Schnittstelle:

```
{
    BACState(int ik, int iN);
    BACEntry mapToEntry(int LPSCount, int MPSCount);
    String toString();
}
```

Die Einträge in Form von `BACEntry`-Objekten werden in einem `TreeSet`-Objekt gespeichert. Der Grund dafür ist, dass die Klasse `TreeSet` eine Speicherung und gleichzeitige Sortierung von Objekten ermöglicht.

Das ist deshalb nötig, weil die Einträge aufsteigend nach ihrer MPS-Wahrscheinlichkeit bzw. absteigend nach ihrer LPS-Wahrscheinlichkeit sortiert werden sollen. Denn dann lässt sich die Abbildung einer beliebigen gegebenen Wahrscheinlichkeitsverteilung auf eine der zugelassenen Verteilungen sehr einfach durchführen. Das heißt, der zuständige Eintrag in der Codiertabelle lässt sich einfach ermitteln.

Zuständig für die Abbildung ist die Methode `mapToEntry(countL, countM)`. `countL` und `countM` geben dabei das Verhältnis zwischen den Wahrscheinlichkeiten für LPS und MPS an. Die Wahrscheinlichkeiten werden daraus errechnet als

$$P(LPS) = \frac{countL}{countL + countM} \quad \text{und} \quad P(MPS) = \frac{countM}{countL + countM}.$$

Um die Abbildung vorzunehmen werden noch die Mittelpunkte zwischen den LPS-Wahrscheinlichkeiten zweier benachbarter Einträge benötigt. Diese werden vom Konstruktor von `BACState` nach der Erstellung der Tabelleneinträge berechnet. Die Mittelpunkte werden als Zwischenstellen der zugelassenen Verteilungen benutzt (siehe dazu auch Abschnitt 3.2 und Abbildung 3.4).

Um den richtigen Eintrag zu ermitteln wird einfach, angefangen beim ersten Eintrag, die LPS-Wahrscheinlichkeit der gegebenen Verteilung mit der nächsten Zwischenstelle verglichen. Da die Einträge absteigend nach der LPS-Wahrscheinlichkeit sortiert sind, ist der erste der mit der größten.

Ist die gegebene LPS-Wahrscheinlichkeit kleiner als die Zwischenstelle, dann wird zum nächsten Eintrag übergegangen, denn dann liegt sie nicht in dem Intervall, das auf den ersten Eintrag abgebildet wird. Ist die LPS-Wahrscheinlichkeit kleiner als alle Zwischenstellen, wird der letzte Eintrag gewählt.

Abschließend liefert die Methode `mapToEntry(countL, countM)` noch das entsprechende `BACEntry`-Objekt zurück.

Der Konstruktor `BACState(k, N)` initialisiert das dem Zustand zugehörige Intervall mit $[k, N)$. Außerdem ist hier die Erzeugungstabelle nach Howard und Vitter [1] aus Abbildung 3.5 implementiert.

Bei der Erstellung eines neuen Zustands sorgt der Konstruktor dafür, dass in die Codiertabelle zuerst eine maximal schiefe Verteilung aufgenommen wird. Dann werden anhand der Erzeugungstabelle alle weiteren Einträge bestimmt. Das entspricht genau der Verfahrensweise, die in Abschnitt 3.6 beschrieben ist.

Außerdem werden noch, wie oben beschrieben, die Zwischenstellen im Konstruktor berechnet.

Die Methode `toString()` liefert für Debuggingzwecke einen String mit Informationen über den Zustand zurück. In dem String sind die Informationen über die Tabelleneinträge integriert. Diese werden über die Methode `BACEntry.toString()` (siehe Abschnitt 5.2.2) ermittelt.

5.2.5 BACTable

Die Klasse `BACTable` beschreibt nun den eigentlichen Codierer. Die vorherigen Klassen werden zur Implementierung dieser Klasse benötigt. Für den Codier- und Decodiervorgang wird aber nur ein `BACTable`-Objekt benötigt und zwei Objekte zu den beiden Interfaces, die die Ein- und Ausgabe übernehmen.

Die Schnittstelle von `BACTable` ist:

```
class BACTable {
    BACTable(int logN);
    void reset();
    void encode(int bit, int c0, int c1, BACOutput out);
    void finishEncode(BACOutput out);
    int decode(int c0, int c1, BACInput in);
    void finishDecode(BACInput in);
    String toString();
}
```

Der Konstruktor `BACTable(logN)` erstellt einen tabellenbasierten Codierer nach der Erzeugungsvorschrift von Howard und Vitter [1]. Die obere Intervallgrenze N der Zustände wird dabei anhand des Parameters `logN` berechnet zu

$$N = 2^{\texttt{logN}}.$$

Für `logN` sind dabei nur Werte $3 \leq \texttt{logN} \leq 15$ zugelassen.

Der Konstruktor erstellt dann für alle ganzzahligen $k \in [0, \frac{N}{2})$ einen Zustand $[k, N)$. Dazu wird jeweils ein `BACState`-Objekt erzeugt. Die `BACState`-Objekte werden in einem Array der Länge $\frac{N}{2}$ gespeichert. Schließlich ruft der Konstruktor noch die Methode `reset()` auf und der Codierer ist damit bereit zum Codieren oder Decodieren.

Die Methode `reset()` versetzt den Codierer in den Startzustand $[0, N)$. Sie muss von außen nur aufgerufen werden, wenn ein Codier- oder Decodiervorgang nicht mit einem Aufruf von `finishEncode(out)` bzw. einem Aufruf von `finishDecode(in)` (siehe unten) ordnungsgemäß beendet wurde.

Um mit dem Codierer zu codieren werden die zu codierenden Bits nacheinander der Methode `encode(bit, c0, c1, out)` übergeben. Der Parameter `bit` nimmt das zu codierende Bit auf. `out` ist ein Objekt dessen Klasse das Interface `BACOutput` (siehe Abschnitt 5.1) implementiert und ist für die Ausgabe der Codierung zuständig. Die Parameter `c0` und `c1` geben das Verhältnis zwischen der Wahrscheinlichkeit für die 0 und die 1 an. Daraus wird die

Wahrscheinlichkeitsverteilung berechnet, mit der codiert werden soll. Sie ist gegeben durch

$$P(0) = \frac{c0}{c0 + c1} \quad \text{und} \quad P(1) = \frac{c1}{c0 + c1}.$$

Die Methode `encode(bit, c0, c1, out)` ermittelt dann zuerst anhand von c0 und c1, welches Symbol (0 oder 1) LPS und welches MPS ist. Dann wird die Methode `mapToEntry(cL, cM)` des zum aktuellen Zustand gehörenden `BACState`-Objekts aufgerufen. Das zurückgelieferte `BACEntry`-Objekt stellt dann den für die errechnete Verteilung zuständigen Codiertabelleneintrag dar. Es wird anschließend benutzt, um das übergebene Bit zu codieren. Das geschieht durch einen Aufruf der Methode `BACEntry.encode(symbol, out)`, die außerdem noch den Zielzustand zurück gibt. In den wird dann zum Schluss noch gewechselt.

Sind alle Eingabebits verarbeitet, muss der Codiervorgang durch einen Aufruf der Methode `finishEncode(out)` beendet werden. Das ist vorallem deshalb nötig, weil es sein kann, dass sich der Codierer am Ende nicht im Zustand $[0, N)$ befindet. Dann steckt aber in dem Zustand noch nicht ausgegebene Information. Da alle Zustände von der Form $[k, N)$ sind für $k < \frac{N}{2}$, kann diese Information durch die Ausgabe einer **1** ausgegeben werden. Denn das Intervall $[\frac{N}{2}, N)$, für das diese Ausgabe steht, ist in allen Intervallen $[k, N)$ enthalten.

Außerdem ruft `finishEncode(out)` noch die Methode `out.finishOutput()` auf und versetzt den Codierer durch einen Aufruf von `reset()` wieder in den Startzustand $[0, N)$.

Das Decodieren läuft ganz ähnlich ab. Die Methode `decode(c0, c1, in)` ermittelt auch zuerst anhand von c0 und c1, welches Symbol (0 oder 1) LPS und welches MPS ist. Dann wird genau wie beim Codieren das zuständige `BACEntry`-Objekt ermittelt. Dieses wird anschließend mit einem Aufruf von `BACEntry.decode(symbol, in)` benutzt, um das nächste Symbol (LPS oder MPS) zu decodieren. Außerdem wird in den zurückgelieferten Zielzustand gewechselt wird. Aus dem decodierten Symbol wird dann das decodierte Bit (0 oder 1) bestimmt und zurückgeliefert.

Auch der Decodiervorgang muss ordnungsgemäß beendet werden. Dazu muss die Methode `finishDecode(in)` aufgerufen werden. Die sorgt durch einen Aufruf von `in.shift(1)` dafür, dass das in `finishEncode(out)` zusätzlich geschriebene Bit auch aus der Codierung herausgeschoben wird. Anschließend wird noch die Methode `in.finishInput()` aufgerufen und durch einen Aufruf von `reset()` in den Startzustand gewechselt.

Die Methode `toString()` erzeugt einen String, der die Codiertabellen für alle Zustände enthält. Diese werden durch Aufrufe von `BACState.toString()` (siehe Abschnitt 5.2.4) ermittelt. Die können dann zu Debuggingzwecken ausgegeben werden.

5.3 Testumgebung

Zusätzlich zu den Interfaces und Klassen befindet sich auf der CD eine Testumgebung in der Datei `BACTableTest.java`. Das Programm `BACTableTest` erwartet als Kommandozeilenparameter nur den Wert `logN` zur Erzeugung des Codierers. Es erzeugt dann von selbst eine 16-Bit lange Eingabe und für jedes Bit eine zufällige Wahrscheinlichkeitsverteilung. Die Eingabebits werden dabei so gewählt, dass überwiegend das wahrscheinlichere Bit in der Eingabe steht, damit auch eine Kompression erreicht werden kann.

Als erstes wird die Codiertabelle auf dem Bildschirm ausgegeben. Dann folgen die erzeugten Eingabebits zusammen mit den Wahrscheinlichkeitsverteilungen. Anschließend wird die Eingabe codiert und wieder decodiert. Die Codierung und die Rekonstruktion werden am Ende zur Kontrolle untereinander ausgegeben.

6 Fazit

In dieser Arbeit wurde eine Einführung in die arithmetische Codierung gegeben und ein tabellenbasiertes Verfahren entwickelt, das deren Laufzeitnachteile umgeht.

Die anschließend vorgestellte Java-Implementierung eines tabellenbasierten arithmetischen Codierers ist allerdings weit davon entfernt, Laufzeitvergleichen standhalten zu können. Denn bei der Implementierung ist das Hauptaugenmerk auf Transparenz gerichtet. Das heißt, es wurde versucht, gut strukturierten, objektorientierten Programmcode zu erstellen, der leicht zu verstehen und wiederverwendbar ist.

Die Implementierung eignet sich aber gut, um das Kompressionsverhalten zu studieren. Sie kann genutzt werden, um die generelle Eignung des Verfahrens in verschiedenen Anwendungsgebieten zu überprüfen.

In der Arbeit wird für den tabellenbasierten Codierer von dem binären Quellalphabet ausgegangen. In der Praxis werden aber in der Regel andere Alphabete auftreten. Es bleibt zu untersuchen, wie man in diesen Fällen am besten vorgeht. Eine Möglichkeit ist es, die Quellsymbole auf Folgen aus $\{0, 1\}^*$ abzubilden und die Wahrscheinlichkeitsverteilungen umzurechnen.

Alternativ kann man auch das tabellenbasierte Verfahren auf allgemeine Alphabete erweitern. Der interessierte Leser sei an dieser Stelle auf die Diplomarbeit von Marcel R. Ackermann [7] verwiesen.

Literaturverzeichnis

[1] Paul G. Howard, Jeffrey Scott Vitter: *Practical Implementations of Arithmetic Coding* in "Image and Text Compression", J. A. Storer, ed., Kluwer Academic Publishers, Norwell, MA (1992), 85-112

[2] Kahlid Sayood: *Introduction to Data Compression*, Morgan Kaufmann Publishers (2000)

[3] Thomas M. Cover, Joy A. Thomas: *Elements of Information Theory*, John Wiley & Sons (1991)

[4] Alistair Moffat, Andrew Turpin: *Compression and Coding Algorithms*, Kluwer Academic Publishers (2002)

[5] Christian Soltenborn, *Kompressionsverlust bei arithmetischer Codierung mit endlicher Genauigkeit*, Studienarbeit an der Universität Paderborn, (zu erscheinen)

[6] G. G. Langdon, J. Rissanen: *Compression of Black-White Images with Arithmetic Coding*, IEEE Trans. Comm. COM-29 (1981), 858-867

[7] Marcel R. Ackermann, *Tabellenbasierte arithmetische Codierung bei allgemeinen Alphabeten*, Diplomarbeit an der Universität Paderborn, (zu erscheinen)