Studienarbeit
„Arithmetische Codierung“

Fachhochschule Karlsruhe, Master Informatik und Multimedia, WS 99/00
erstellt von Gertrud Grünwied und Jürgen Baier,
betreut von Herrn Prof. Dr. Hoffmann

Inhaltsverzeichnis

1. Aufgabenstellung

Die vorliegende Arbeit ist eine Studienarbeit mit dem Ziel, das Kompressionsverfahren der *Arithmetischen Codierung* (kurz: AC) in seinen Grundzügen zu behandeln. Zu den theoretischen Themen zählen dabei die Einordnung des AC innerhalb der Kompressionsverfahren, die Funktionsweise, die Anwendungsgebiete und erweiterte Theorie, wie die adaptive Arithmetische Codierung. Der praktische Teil basiert auf der Implementierung des Algorithmus mit Hilfe eines Java-Programms. Dabei werden die Merkmale und Problemstellungen bei der praktischen Umsetzung herausgearbeitet. Der Schwerpunkt liegt auf der algorithmischen Behandlung, und nicht auf der Anfertigung eines durch eine komfortable Oberfläche ergänzte Software-Anwendung.

Die Ergebnisse der Studienarbeit werden in der vorliegenden Dokumentation dargestellt und in einem 30 min-Vortrag präsentiert.

2. Einführung

2.1 Datenkompression und Multimedia-Daten

Datenkompression ist heute vor allem bei Multimedia-Daten notwendig, da die darin enthaltenen Datenraten sehr groß sind. Die Datenrate von Festplatten liegt beispielsweise um den **Faktor 15 unter** der Echtzeitdigitalsierung von Videobildern. Dieses Problem wird durch Datenkompression der Multimedia-Daten gelöst. Zu den Multimedia-Daten zählen Schrift, Audio, Bild- /Videodaten s. Tabelle [Henning].

Multimedia-Daten	Auflösung	Sampling Rate	Datenrate
Schrift	8 Bit/Zeichen	40 Zeichen/s	320 Bit/s
Audio (Sprache)	8 Bit/Zeichen	10.000 Samples/s	80 kBit/s
Audio (CD)	16 Bit/Sample Stereo	44.100 Samples/s	1,4 MBit/s
Bilddaten	640 x 480 Pixel, 32 Bit Farbtiefe	1 Bild/s	10 MBit/s
Videodaten (PAL)	720 x 525 x 16 Bit	25 Vollbilder/s	100 MBit/s bei MPEG

Bei der Datenkompression werden redundante oder unnötige Teile aus dem Datenstrom entfernt. Das Prinzip der Datenkompression wurde bereits vom Morsecode verwendet:

- kurze Codes für häufig auftretende Zeichen. Beispiel: der am häufigsten auftretende Buchstabe „e“ hat einen einzelnen „Punkt“.
- längere Codes für selten auftretende Zeichen.
  Beispiel: der selten auftretende Buchstabe „y“ hat zwei „Striche“, einen „Punkt“ und ein „Komma“.

2.2 Klassifizierung

Verlustfreie und verlustbehaftete Kompression

Grundsätzlich sind verlustfreie und verlustbehaftete Kompression zu unterscheiden.

- verlustfreie Kompression

Die verlustfreie Datenkompression sorgt dafür, dass die Codierung des Datenbestandes möglichst der theoretischen Grenze der Entropie nahekommt, entfernt aber keine Informationen aus dem Datenbestand. Die Originalnachricht kann aus der komprimierten Nachricht vollständig ermittelt werden. Dieses Verfahren eignet sich für Textdaten oder schwarz-weiss Bilddaten. Sowohl Huffman wie auch die Arithmetische Codierung sind Beispiele einer verlustfreien Komprimierung.

- verlustbehaftete Komprimierung

Die verlustbehaftete Komprimierung ist nur bei Daten möglich, die für menschliche Sinnesorgane bestimmt sind. Aufgrund der Tätigkeit unseres Gehirns kann man bei Bildern, Audio- und Videodaten Informationen entfernen, ohne dass dies den subjektiven Eindruck verschlechtert.

Statistische Verfahren und Dictionary Verfahren

Die meisten üblichen Kompressionsverfahren teilen sich in zwei Gruppen: die statistischen Verfahren und die Dictionary Verfahren.

- Statistische Verfahren

Bei statistischen Verfahren werden die Symbole in Codes mit variabler Länge verschlüsselt (*VLC=Variable Length Coding*). Die Länge der Codes variiert abhängig von der relativen Häufigkeit der Symbole. Symbole mit geringer Wahrscheinlichkeit werden mit vielen Bits kodiert, Symbole mit hohen Wahrscheinlichkeiten benötigen weniger Bits beim Codieren. Die Arithmetische Codierung gehört zu dieser Gruppe. Weitere Verfahren sind Huffman, Shannon-Fano Coding.

- Dictionary Verfahren

Dieses Verfahren arbeitet so, dass Gruppen von Symbolen im eingehenden Datenstrom mit festen Codelängen ersetzt werden. Ein bekanntes Beispiel dafür ist die LZW-Kompression. LZW ersetzt von Strings mit unendlicher Länge durch Codes, deren Länge 9 bis 16 Bits beträgt.

2.3 Patente für die Arithmetische Codierung

Leider sind die Standard-Algorithmen für die Arithmetische Codierung von IBM patentiert. Die Compression FAQ-Website [Compression FAQ 99] listet 19 IBM-Patente auf und verweist auf zahlreiche weitere Patente. Das Problem ist, dass man bei Codierung und Decodierung die gleichen Algorithmen benötigt, da zwei verschiedene Implementierungen meistens nicht kompatibel sind. Für die JPEG-Kompression z.B. kann man nicht einfach einen selbstgeschriebenen Algorithmus verwenden. Man muss hier für die Kompression die patentierten Standard-Algorithmen verwenden, da sonst die zahlreichen Programme, die JPEG verwenden, das Bild nicht dekomprimieren können. Nähere Informationen zu den Standard-Algorithmen und deren Patentierung findet man unter [Compression FAQ 99].

3. Grundzüge der Arithmetische Codierung

3.1 Das Prinzip

Die Arithmetische Codierung ist ein patentiertes Verfahren, das nicht ohne Lizensierung verwendet werden darf. Auf dem Markt sind jedoch viele Implementierungen patentfrei verfügbar, die Ergänzungen und Optimierungen zum Original-Algorithmus enthalten, wie beispielsweise der „Range Encoder“. Das Prinzip ist wie folgt:

- Jedes **Zeichen** wird durch ein **Wahrscheinlichkeitsintervall** codiert.
- **Zeichenfolgen** werden durch ineinander **geschachtelte Wahrscheinlichkeitsintervalle** codiert.
- Der Eingabestrom wird Symbol für Symbol gelesen. Für jedes Symbol werden **weitere Bits an den Code** angehängt.
  Die Codelänge steigt mit wachsenden Zeichenfolgen.
- Die gesamte Nachricht wird schlussendlich durch **einen einzigen Code** repräsentiert (final code).
- Der Komprimier- und Dekomprimier-Vorgang entspricht dem *first-in first-out* (**FIFO**) Prinzip. Die Symbole werden in der gleichen Reihenfolge dekodiert, in der sich kodiert wurden. FIFO Kompression ermöglicht die *adaptive* Arithmetische Codierung.
- Der Algorithmus ist unabhängig von dem verwendeten **Modell**. Die idealisierte Arithmetische Codierung geht von einer stochastischen Unabhängigkeit der Symbole aus. Das verwendete Modell 0ter Ordnung ist unter dieser Annahme optimal im informationstheoretischen Sinne der Entropie. Im Gegensatz dazu bezieht das Modell 1ter Ordnung die Kontextabhängigkeit der Symbole ein. Enkoder und Dekoder müssen grundsätzlich das gleiche Modell verwenden!

3.2 Einführungsbeispiel

Der Ausgabewert des arithmetischen Kodierungsprozesses ist ein einziger numerischer Wert im Bereich [0,1). Dieser Codewert kann eindeutig dekomprimiert werden, um die Originalnachricht zu rekonstruieren. Damit der Codewert erzeugt werden kann, muss die statistische Verteilung der Symbole innerhalb der Nachricht ermittelt werden.

Beispiel:
Die drei Zeichen a, b, c treten mit den Wahrscheinlichkeiten 0.2, 0.2 und 0.6 auf. Die Intervalle lauten:

Symbol	**Wahrscheinlichkeit**	**Intervall [Low, High)**
a	0.2	[0.8, 1)
b	0.2	[0.6, 0.8)
c	0.6	[0.0, 0.6)

Als Beispiel soll die Zeichenfolge **cbb** kodiert werden.

Kodierung

$$NewLow = OldLow + (OldHigh - OldLow) * Low$$
$$NewHigh = OldLow + (OldHigh - OldLow) * High$$

Erstes Zeichen **c**:
NewLow = 0
NewHigh = 0.6

Zweites Zeichen **b**:
NewLow = 0 + (0.6 – 0) * 0.6 = 0.36
NewHigh = 0 + (0.6 – 0) * 0.8 = 0.48

Drittes Zeichen **b**:
NewLow = 0.36 + (0.48 – 0.36) * 0.6 = 0.432
NewHigh = 0.36 + (0.48 – 0.36) * 0.8 = 0.456

Neues Symbol	**Low**	**High**
c	0.0	0.6
b	0.36	0.48
b	**0.432**	0.456

Der Code für die Zeichenfolge **cbb** ist der letzte Low-Wert **0.432**.

Dekodierung

Der übertragene Code lautet **0.432**. Das erste dekodierte Symbol wird ermittelt, indem das Symbol gesucht wird, in dessen Intervall der Code liegt. Da der Code 0.432 zwischen 0.0 und 0.6 liegt, muss das erste Symbol ein **c** sein. Das Symbol c wird ausgegeben und anhand dieses Symbols der neue noch verbleibende Code ermittelt. Die Formel lautet:

$$Code = (Code - Low) / Range$$
$$\text{mit } Range = (High - Low)$$

Die einzelnen Zwischencodes lauten:

Code	**Ausgabe-Symbol**	**Neuer Code**	
0,432	c	(0.432 – 0) / 0.6	= 0.72
0.72	b	(0.72 – 0.6) / 0.2	= 0.6
0.6	b	(0.6 – 0.6) / 0.2	= 0

Die dekodierten Symbole ergeben nach dem FIFO-Prinzip **cbb**. Der Prozess wird solange wiederholt, bis der neue Code den Wert Null ergibt. An dieser Stelle wird noch nicht darauf eingegangen, dass der Wert im Dekodier-Prozess auch auftreten kann, obwohl noch nicht alle Symbole gelesen wurden. In diesem Fall wird das Zeichen EOF aufgenommen, das das Ende der Nachricht mit Sicherheit kennzeichnet.

3.3 Unterschiede zu Huffman

Bei der Huffman-Komprimierung werden den einzelnen Zeichen eines Datenstroms Codes verschiedener Länge zugewiesen. Diejenigen Zeichen, die am häufigsten auftreten, erhalten die kürzesten Codes,. ist durch folgende Merkmale charakterisiert:

- Bei Huffman ist für jedes in der Nachricht verwendete Symbol ein eigener Code notwendig.

- Die Huffman-Komprimierung ist fast optimal, wenn die relativen Wahrscheinlichkeiten p_i, mit denen die einzelnen Symbole innerhalb einer Nachricht auftreten, möglichst Potenzen von 2 sind. 2^{-x}, wobei x = ganzzahliger Wert.

  Beispiele:
  $p_i = 0,5$ $2^{-1} = 0,5$ (geeignet, da genau 1 Bit/Symbol benötigt)
  $p_i = 0,25$ $2^{-2} = 0,25$ (geeignet, da genau 2 Bits/Symbol benötigt)
  $p_i = 0,125$ $2^{-3} = 0,125$ (geeignet, da genau 3 Bits/Symbol benötigt)
  $p_i = 0,4$ $2^{-1,32} = 0,4$ (schlecht geeignet, da 1,32 Bits(!) benötigt)

  Bei $p_i = 0,4$ beträgt das Informationsmaß $-\log_2 0,4 = 1,32$. Idealerweise müssten dem i-Symbol damit ein 1- *oder* 2 Bit langer Code zugeordnet werden, was jedoch nicht möglich ist, da ein Code jedem einzelnen individuellen Zeichen zugeordnet wird.
  Dieses Huffman-Problem existiert bei der Arithmetischen Codierung nicht, da die gesamten Nachricht durch einen einzigen Code repräsentiert wird.

3.4 Vorteile und Nachteile im Überblick

Vorteile

- Optimale Codierung bei sehr langen Nachrichten.
- Effizienter als Huffman, da relative Häufigkeiten der Symbole nicht Potenzen von 2 sein müssen.

Nachteile

- Verwendung benötigt (kostenpflichtige) Lizenzierung, da patentiertes Verfahren.
- Vor der eigentlichen Codierung muss die statistische Verteilung der Zeichen innerhalb der gesamten Nachricht bekannt sein. Dieser Nachteil kann durch adaptives Verfahren behoben werden.
- Die Implementierung des Algorithmus ist aufwendiger als Huffman.

4. Merkmale der arithmetischen Codierung

4.1 Datenformate

Die Arithmetische Codierung ist in allen Fällen anwendbar, in denen auch Huffman Kodierung möglich ist. Im folgenden werden einige Beispiele gegeben:

- Textformate
  Im Unterschied zu Huffman muss bei einem z.B. 24-Zeichen Alphabet nicht ein entsprechend komplexer Binärbaum aufgebaut werden, sondern nur die statistische Verteilung der Buchstaben bekannt sein bzw. adaptiv im Lauf des Kodiervorgangs ermittelt werden.

- Grafikformat JPEG und Videoformat MPEG
  Die Arithmetische Codierung kann beim Encoder nach der Quantisierung der Pixelwerte verwendet werden. Anschließend werden die komprimierten binären Werten abgespeichert und an den Decoder übertragen.

- Binärdateien (z.B. exe-Dateien)

4.2 Kompressionsraten abhängig von statistischer Verteilung

Die Länge des Final Code in der Arithmetische Codierung ist abhängig von den Wahrscheinlichkeiten der Symbole. Die Wahrscheinlichkeiten wirken sich folgendermaßen auf die Kompressionsrate aus:

- Je **ähnlicher** die relativen Wahrscheinlichkeiten sind, umso kürzer ist der *Final Code* für die Arithmetische Codierung.

  Beispiel für „kurzen“ Final Code bei folgenden ähnlichen relativen Wahrscheinlichkeiten:
  $p_i(S)$ = 0,5
  $p_i(W)$ = 0,1
  $p_i(I)$ = 0,1
  $p_i(M)$ = 0,2
  $p_i(BLANK)$ = 0,2

  Der Final Code für SWISS MISS ist 0.71753375, davon werden jedoch nur die **8 Stellen** „71753375“ in den Ausgabestrom geschrieben. Die Länge der Original-Nachricht betrug 10 Stellen.

- Wenn die Wahrscheinlichkeiten der Symbole **sehr unterschiedlich** sind, kann der *Final Code* sogar länger werden als die ursprüngliche Nachricht. Man spricht von sog. „Skewed Probabilities“ [Salomon].

  $p_i(a)$ = 0,001837
  $p_i(b)$ = 0,975
  $p_i(c)$ = 0,023162
  $p_i(EOF)$ = 0,000001 $p_i = 1$

  Der Final Code für *cccc*EOF lautet:
  Dezimalwert: 0.0000002878086184764172
  Die Länge der Original-Nachricht einschließlich EOF ist in diesem Beispiel kürzer als der komprimierte Code.

4.3 Final Code

Der Final Code ist der komprimierte Code, der an den Dekodierer übertragen wird und der die Original-Nachricht eindeutig repräsentiert.

Zwei wichtige Merkmale

- Die Null vor dem Nachkomma wird eingespart, da der Code immer zwischen 0 (inklusiv) und 1 (exclusiv) liegt, d.h. das Intervall ist [0,1). Beispiel: Nicht 0.97, sondern nur 97.

- In der Literatur wird als Final Code häufig der letzte untere Wert (Low) angegeben. In der Praxis jedoch werden die binären Gleitkommazahlen für die letzte obere und untere Intervallgrenze berechnet und der obere Wert wird nach der ersten Stelle abgebrochen, welche vom unteren Wert verschieden ist [Henning].

Intervallwert	**dezimal**	**binär**
Low	`0,71753`**`3`**`75`	`0.1011 0111 1011 0000 0100 1010 0111`
High	`0,71753`**`5`**	`0.1011 0111 1011 0000 0101 1111 1011`
Final Code	`0.71753`**`5`**	`0.1011 0111 1011 0000 0101` abgebrochen

4.4 Der Komprimierungsvorgang

Der Komprimierungsvorgang gliedert sich in folgende Einzelschritte:

Encoder

1. Modell festlegen, z.B. Modell 0. Ordnung.

2. Gesamte Nachricht lesen und statistische Verteilung der Zeichen ermitteln und Codetabelle aufbauen

3. Eigentliche Codierung gemäß Algorithmus und Codetabelle aufbauen und Ergebnis im Final Code festhalten.

4. Codetabelle (nur bei *statischer* Arithmetischer Codierung) und Final Code an den Decoder übertragen. Ggf. auch Länge der Nachricht (unkodiert!) übertragen, falls kein EOF verwendet wird.

Decoder

1. Gleiches Modell wie Encoder verwenden.

2. Dekodierung gemäß Decodier-Algorithmus bis Ende der Nachricht. Die Symbole werden nach dem FIFO-Prinzip ausgegeben.

4.5 Die EOF-Problematik

Für die richtige und vollständige Dekomprimierung ist es notwendig, dass der Decoder zuverlässig das Ende des Datenstromes erkennt. Wenn jedoch beim schrittweisen Abarbeiten des Codes die Null auftritt, ohne dass der Datenstrom tatsächlich zu Ende ist, wird die Nachricht nicht vollständig dekodiert. Diese Problematik bezeichnet man auch als EOF-Problematik. Um das Ende des Datenstroms eindeutig zu kennzeichnen, kann ein zusätzliches Symbol, das EOF-Zeichen, aufgenommen werden und an das Ende der Nachricht gesetzt werden. Dem EOF-Symbol wird in der Codetabelle nur eine geringe Wahrscheinlichkeit p_{EOF} zugeordnet.

Anstelle des EOF-Zeichens könnte auch die Gesamtlänge der Nachricht an den Dekodierer in unkodierter Form übertragen wird. Diese Alternative ist nur möglich, wenn die Gesamtlänge bekannt ist!

Das Problem der vorzeitigen Null (Abbruchkriterium) tritt immer dann auf, wenn das letzte Symbol im Eingabe-Datenstrom dasjenige ist, dessen untere Intervallgrenze bei 0 startet.

Beispiel für die Notwendigkeit des EOF-Symbols

Die Zeichen a, b, c sind bereits aus dem Einführungsbeispiel bekannt. Das Zeichen **c** ist kritisch, da dessen Intervall bei 0 beginnt.

Symbol	**Wahrscheinlichkeit**	**Intervall [Low, High)**
a	0.2	[0.8, 1)
b	0.2	[0.6, 0.8)
c	**0.6**	**[0.0, 0.6)**

Als Beispiel soll die Zeichenfolge **bbc** kodiert werden, wobei **c** das letzte Zeichen der Nachricht ist.

Neues Symbol	**Low**	**High**
b	0.6	0.8
b	0.72	0.76
c	**0.72 final code**	0.748

Die Dekomprimierung erreicht bereits nach dem 2. Zeichen die Null.

Code	**Ausgabe-Symbol**	**Neuer Code**	
0.72	b	(0.72 – 0.6) / 0.2	= 0.6
0.6	b	(0.6 – 0.6) / 0.2	= **0 !!**

Die Lösung:
In die Codetabelle wird das Symbol EOF mit geringer Wahrscheinlichkeit aufgenommen. Der Eingabe-Datenstrom ist damit **bbc*EOF***.

Symbol	**Wahrscheinlichkeit**	**Intervall [Low, High)**
EOF	0.001	[0.999, 1)
a	0.199	[0.8, 0.999)
b	0.2	[0.6, 0.8)
c	0.6	[0.0, 0.6)

Neues Symbol	**Low**	**High**
b	0.6	0.8
b	0.72	0.76
c	0.72	0.748
EOF	**0.747972 final code**	0.748

5. Ausgewähltes Beispiel

Dieses Beispiel beinhaltet die drei Zeichen a, b und c, wobei **a** eine sehr kleine und **b** eine sehr große Wahrscheinlichkeit hat.

Symbol	**Wahrscheinlichkeit**	**Intervall [Low, High)**
a	0.001838	[0.998162, 1)
b	0.975	[0.023162, 0.998162)
c	0.023162	[0.0, 0.023162)

Vergleichsweise werden die Zeichenfolgen **bbbb** und **cccc** untersucht.

Zeichenfolge bbbb:

Neues Symbol	**Low**	**High**
b	0.023162	0.998162
b	0.04574495	0.99636995
b	0.06776322625	0.99462270125
b	**0.08923124309375**	0.99291913371875

Als Ergebnis kann festgehalten werden:
- Das Zeichen b hat eine sehr hohe Wahrscheinlichkeit.
- Die Low- und High-Werte sind deshalb zu Beginn sehr verschieden.
- Die Low- und High-Werte nähern sich nur langsam an.

Zeichenfolge ccccc:

Neues Symbol	**Low**	**High**
c	0.0	0.023162
c	0.0	0.000536478244
c	0.0	0.000012425909087528
c	**0.0**	0.0000028780890628532 35

Als Ergebnis kann festgehalten werden:
- Das Zeichen c hat eine sehr geringe Wahrscheinlichkeit.
- Die Low- und High-Werte liegen deshalb zu Beginn sehr nah zusammen.
- Die Low- und High-Werte nähern sich sehr schnell aneinander an.
- Der High-Wert nähert sich sehr schnell der Null.

6. Adaptive Arithmetische Codierung

Die *adaptive* Arithmetische Codierung stellt eine Optimierung der Arithmetische Codierung dar.

Zur Unterscheidung werden hier nochmals Merkmale der statischen Arithmetische Codierung aufgeführt:

- Beim Kodieren muss die gesamte Nachricht zuerst abgetastet werden, damit die statistische Verteilung der Symbole ermittelt werden kann. Die einzelnen Wahrscheinlichkeitswerte werden in der Codetabelle protokolliert.

- Der Dekodierer muss die Symbole, die zu den berechneten dekodierten Codes passen, dann nur aus der Codetabelle ablesen.

Im Gegensatz dazu ist das adaptive Verfahren im Laufe des Kodierprozesses veränderlich.

- Die statistische Verteilung steht nicht von vornherein fest, sondern wird erst im Laufe der Kodierung sukzessive ermittelt. Die Symbole werden nach und nach gelesen und entsprechend dazu ändern sich die Wahrscheinlichkeiten, mit denen die Symbole auftreten. Die Codetabelle wird nur soweit aufgebaut wie nötig, was ein wesentlicher Vorteil gegenüber der statischen Codierung ist. Die gesamte Nachricht wird damit nicht zu Beginn vollständig gelesen.

- Am Anfang ist das adaptive Verfahren sehr ungenau, mit wachsender Länge der Nachricht jedoch immer genauer.

- Voraussetzung für die Adaption ist ein gleicher Algorithmus auf Sender- und Empfängerseite.

7. Von der Theorie zur Praxis: Die Implementierung

Die Arithmetische Codierung ist ein Kompressionsverfahren, das die auftretenden Symbole in Form von Wahrscheinlichkeitsintervallen codiert. Da es bei einer Codierung darauf ankommt, für häufig vorkommende Zeichen kurze Codes und für selten vorkommende Zeichen längere Codes zu verwenden, ist eine Nachricht, die mit der arithmetischen Codierung komprimiert wurde, sehr nahe an der (idealen) Entropie.

7.1 Die Entropie

Die theoretisch erreichbare Entropie einer Nachricht lässt sich relativ einfach ermitteln. Dazu benötigt man die Anzahl der in der Nachricht verwendeten Symbole und die relativen Wahrscheinlichkeiten aller Symbole. Jetzt kann man mit

$$H = -\sum_{i=1}^{N} p_i \cdot \log_2(p_i)$$

die Entropie H ausrechnen. Die mittlere Codelänge einer Nachricht ist dabei gleich der Entropie. Wenn alle Symbole einer Nachricht die gleiche Wahrscheinlichkeit haben, dann ist die Entropie stets 1, es wird also für die Codierung 1 Bit pro Zeichen benötigt. Wenn nun die Entropie einer Nachricht kleiner ist als 1, muss bei Codes mit fester Codelänge (z.B. Huffman) "aufgerundet" werden: die Kompression ist geringer, als theoretisch möglich wäre. Die Arithmetische Codierung ist ein Kompressionsverfahren, das mit variablen Codelängen arbeitet und Codes erzeugt, die unter Umständen näher an der Entropie liegen (ist die Entropie 1, so ist die komprimierte Nachricht bei Huffman-Codierung und arithmetischer Codierung gleich lang).

7.2 Ordnung eines Modells

Die Wahrscheinlichkeitsverteilung der Symbole beeinflusst die Qualität der Codierung, also die Kompressionsrate. Diese Wahrscheinlichkeitsverteilung wird normalerweise als Modell bezeichnet. Für die Implementierung der arithmetischen Codierung ist der Begriff der Ordnung des verwendeten Modells von großer Bedeutung.

Im einfachsten Fall wird ein Modell der Ordnung 0 verwendet. Bei einem solchen Modell werden die auftretenden Zeichen als unabhängig angesehen. Anders ausgedrückt, es wird lediglich die Häufigkeit der Symbole benutzt, um die Nachricht zu codieren. Man erreicht auch schon mit einem Modell der Ordnung 0 gute Ergebnisse. Bei der theoretischen (idealisierten) arithmetischen Codierung wird eine stochastische Unabhängigkeit der Symbole angenommen. Das heißt, dass

der Standard-Algorithmus optimal codiert, wenn eine Modell der Ordnung 0 verwendet wird.

Es kommt jedoch häufig vor, dass die Zeichen der Nachricht nicht unabhängig voneinander sind. So kommt z.B. bei einem Text in deutscher Sprache nach dem Symbol 'c' meistens ein 'h'. Wird dies berücksichtigt, ist die komprimierte Nachricht oft deutlich kleiner als das Original. Dazu wird ein Modell höherer Ordnung verwendet. Bei einem solchen Modell wird ein Zeichen nicht isoliert sondern im Zusammenhang (Kontext) mit dem folgenden Zeichen gesehen. Bei Nachrichten in "natürlicher" Sprache erhält man bei der Komprimierung mit einem Modell höherer Ordnung zumeist eine bessere Kompression. Auch bei Binärdateien (*.exe, ...) ist ein gewisser Kontext vorhanden, so dass es sich zumeist lohnt, ein rechenintensiveres Modell zu verwenden.
In einer Implementierung wird man versuchen, den Kompressionsteil des Programmes so generisch zu machen, dass das Modell beliebig ausgetauscht werden kann.

7.3 Algorithmus für Arithmetische Codierung

Bei der arithmetischen Codierung werden Symbole durch Wahrscheinlichkeitsintervalle beschrieben. Genauer gesagt, es werden den einzelnen Symbolen disjunkte Teile des Intervalls [0..1) zugeordnet. Für diese Zuordnung (Codierung) wird in der Literatur folgender Algorithmus beschrieben:

```
while (!endOfFile) {
        x = getChar();
        unten = unten + bereich * untereGrenze(x);
        oben = unten + bereich * obereGrenze(x);
        bereich = oben – unten;
}
```

Die Werte für *unten* und *oben* sind dabei 0.0 und 1.0. Der Startbereich ist damit 1.0. Die Dekomprimierung wird mit folgendem Algorithmus vorgenommen:

```
getString(y);
while (!endOfFile) {
        x = suchePassendesIntervall(y);
        putChar(x);
        bereich = obereGrenze(x) – untereGrenze(x);
        y = (y – untereGrenze(x)) / bereich;
}
```

Die Probleme, die diese vereinfachte Sicht verursachen kann, werden im nächsten Abschnitt näher beschrieben.

8. Probleme bei der Implementierung

Der oben beschriebene Algorithmus funktioniert theoretisch recht gut. Die Frage, die sich jedoch stellt, ist: Wie lässt sich dieser Algorithmus implementieren? Dabei stößt man auf drei Probleme:

- Die Wahrscheinlichkeitsintervalle werden schnell zu klein, um mit Floating Point-Datentypen dargestellt werden zu können. Eigentlich müssten Datentypen mit beliebiger Genauigkeit verwendet werden.
- Man benötigt ein inkrementelles Lese- und Codierschema, um den Eingabestrom nicht erst komplett lesen, dann komplett codieren und anschließend komplett ausgeben zu müssen.
- Das verwendete Modell sollte so gestaltet sein, dass man beim Decodieren effizient darauf zugreifen kann.

Die folgenden Abschnitte beschreiben diese Probleme detaillierter.

8.1 Die Wahrscheinlichkeitsintervalle werden zu klein

Das Hauptproblem ist offensichtlich: Die Wahrscheinlichkeitsintervalle, die bei der Codierung auftreten, werden sehr schnell sehr klein. Zu klein, als dass man sie mit einem Integer-Datentyp (*int*, *float*, *double*) erfassen könnte. In der Programmiersprache Java ist der genaueste Datentyp *double*. Mit einem *double*-Typen (nach IEEE 754 floating-point) lassen sich Zahlen mit 64 Bit darstellen. Der Bereich geht also von

$$\pm 1.79769313486231570 \cdot 10^{308} \text{ bis}$$
$$\pm 4.94065645841246544 \cdot 10^{-324}.$$

Prinzipiell wäre es also bei kleineren Nachrichten möglich, den "theoretischen" Algorithmus anzuwenden. Allerdings nur bei sehr kleinen Nachrichten. Zur Verdeutlichung folgendes Beispiel:

Mit einem Byte können 256 Werte erfasst werden. Wenn jedes Symbol mit gleicher Wahrscheinlichkeit auftritt (oder erfasst wird), dann hat man eine relative Wahrscheinlichkeit von $\frac{1}{256}$ für jedes Symbol. Bei zwei Bytes beträgt das entsprechende Intervall $\frac{1}{65536}$, bei drei Bytes hat man ein Intervall von $\frac{1}{2^{24}}$, bei vier Bytes ist das Intervall $\frac{1}{2^{32}}$ groß und bei acht Bytes ist man schon bei der maximal darstell-

baren Genauigkeit: $\frac{1}{2^{64}}$. Acht Bytes sind sicherlich nicht genug, um "normale" Dateien komprimieren zu können.

8.2 Inkrementelles Lese- und Codierschema ist notwendig

Es ist wünschenswert, eine Nachricht inkrementell (Byte für Byte) lesen und codieren zu können, damit nicht die ganze Nachricht auf einmal im Speicher gehalten werden muss. Deshalb muss der Algorithmus eine schrittweise Codierung und damit eine schrittweise Aktualisierung des Modells zulassen. Es ist also wichtig, Mechanismen zur automatischen Bereichserweiterung zur Verfügung zu stellen, wenn die high- und low-Werte eines Bereichs zu nahe beieinander liegen.

8.3 Effizienter Zugriff auf die Repräsentation des Modells

Um ein schnelles Decodieren zu ermöglichen, muss das Modell so repräsentiert werden, dass man effizient damit arbeiten kann. Die Überlegungen zur Repräsentation hängen nicht zuletzt mit der verwendeten Programmiersprache zusammen: Welche Datentypen sollen verwendet werden? Sollen objektorientierte Datentypen wie die Collection Classes von Java zum Einsatz kommen? Man sollte jedenfalls beachten, dass spezielle Klassen in anderen Programmiersprachen oft nur eingeschränkt verwendet werden können und sich auch in der Effizienz stark unterscheiden können. Witten [Witten 87] beschränkt sich in seiner C-Implementation auf Arrays elementarer Datentypen:

- Ein Array für die Symbole selbst.
- Ein Array, das auf die Symboltabelle verweist und nach der Häufigkeit der Symbole sortiert ist
- Ein Array, das die kumulierten Häufigkeiten (als Integerwerte) enthält.

9. Die Implementation

In diesem Abschnitt werden die Aspekte aufgeführt, die bei der Implementierung der arithmetischen Codierung wichtig sind. Eine spezielle Implementation wird weiter unten näher beschrieben.

9.1 Inkrementelle Ausgabe durch Binärbrüche/Integerarithmetik

Prinzipiell durchläuft die Codierung folgende Schritte:

- Dem ersten Symbol wird anhand seiner Wahrscheinlichkeit ein Intervall zugewiesen.
- Jedes weitere Symbol schränkt dieses Intervall weiter ein.
- Am Ende erhält man eine Fließkommazahl (normalerweise wird sie durch $\frac{\text{obere Grenze - untere Grenze}}{2}$ berechnet), welche die gesamte Nachricht repräsentiert.

Wenn man nun statt mit Fließkommazahlen mit Binärbrüchen arbeitet, ist es möglich, die codierten Bits während des Kodiervorganges auszugeben, da sich die ersten Bits nach dem Komma nicht mehr ändern. Dazu startet man mit einem Bereich von 0.0000 bis 0.9999 statt mit 0 bis 1. Das entspricht binär dem Bereich 0.0000 bis 0.1111.

Diese Binärzahlen sollen nun als Integerzahlen gespeichert werden. Dazu verschiebt man den Dezimalpunkt um eine Stelle nach links und verwendet so viele Dezimalziffern, wie beim verwendeten Datentyp dargestellt werden können. Die meisten Implementierungen verwenden 16 Bit-Arithmetik, was dem Java-Datentyp *long* entspricht und auch völlig ausreicht. Bei 16 Bit hat man einen Bereich von 0 bis 65535, es stehen also 5 Dezimalziffern zur Verfügung. Es wird also mit dem Bereich 0 bis 65535 gearbeitet. Welche Bits ausgegeben werden, hängt vom aktuellen Bereich ab.

9.2 Renormalisierung des Intervalls

Um zu vermeiden, dass die Wahrscheinlichkeitsintervalle zu klein werden, ist es notwendig, das aktuelle Intervall (das die komplette Nachricht repräsentiert) ständig anzupassen. Dazu ist eine Renormalisierung notwendig. Durch die Renormalisierung des Intervalls verhindert man, dass die Wahrscheinlichkeitsintervalle zu klein werden.

9.3 Underflow und Overflow

Wenn die Bereichsgrenzen *high* und *low* sehr nahe beieinander liegen, besteht die Möglichkeit, dass verschiedenen Symbolen der gleiche Integer-Wert zugewiesen wird. In diesem Fall kann die Nachricht nicht weiter codiert werden; das Ergebnis ist nicht definiert und fehlerhaft. Der Codieralgorithmus muss also sicherstellen, dass high und low stets weit genug auseinanderliegen.

Zur Verhinderung eines Overflows muss kein spezieller Code geschrieben werden. Ein Overflow wird durch geeignet Wahl der Datentypen für bestimmte Variablen verhindert.

9.4 Modelle n-ter Ordnung für den arithmetischen Codierer

Bei der arithmetischen Codierung wird im Prinzip nicht berücksichtigt, dass die Symbole meist in einem bestimmten Kontext stehen. Will man nun diese Kontextinformation einsetzen, um noch mehr Redundanz zu beseitigen, benötigt man Modelle höherer Ordnung. Bei diesen Modellen werden die Symbole vor der Codierung dekorreliert, um ein besseres Ergebnis zu erzielen [Kuhn 95].

Der Artikel "Arithmetic Coding and Statistical Modeling" von Mark Nelson im Dr. Dobb's Journal vom Februar 1991 [Nelson 91] befasst sich mit Modellen höherer Ordnung für die Arithmetische Codierung (der folgende Text basiert auf diesem Artikel). Hier findet man auch eine gut kommentierte Implementierung in C.
Um bei der Codierung einer Nachricht den Kontext zur berücksichtigen, muss zunächst geprüft werden, ob das aktuelle Symbol schon einmal in Kombination mit einem anderen Symbol vorgekommen ist. Ist dies der Fall, wird die Häufigkeit für dieses Symbol in diesem bestimmten Kontext um 1 erhöht. Es ist also auch möglich, dass ein Symbol in verschiedenen Kontexten vorkommt.

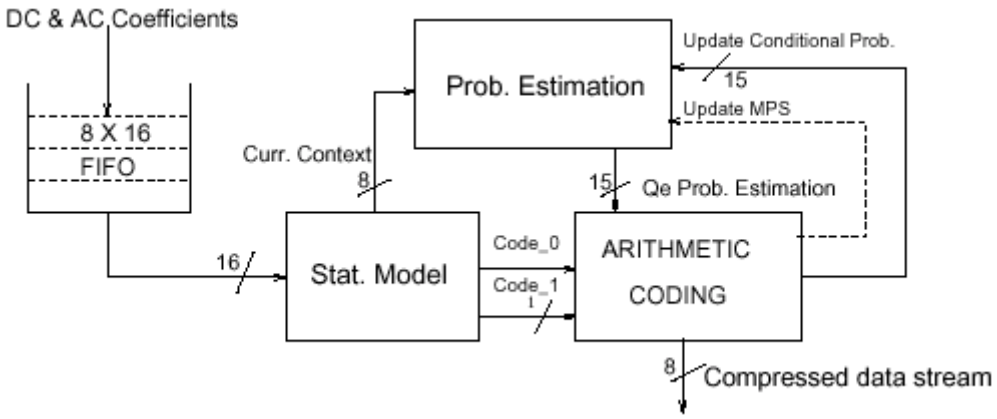
Wird z.B. ein Modell der Ordnung 3 verwendet, wird zunächst geprüft, ob das aktuelle Symbol schon einmal nach den drei vorhergehenden Symbolen vorgekommen ist. Ist dies nicht der Fall, so wird geprüft, ob es schon einmal in Kombination mit den zwei vorhergehenden Symbolen vorgekommen ist, usw.

Der Implementierungsaufwand ist also sehr viel höher, wenn Modelle höherer Ordnung verwendet werden. Oft ist es einfacher, eine Nachricht mit der "normalen" arithmetischen Codierung zu codieren und anschließend nochmals einen Algorithmus zu verwenden, der die vorhergehenden (bzw. nachfolgenden) Symbole berücksichtigt. Ein einfacher Algorithmus ist die Lauflängencodierung (Run Length Enco-

ding, RLE). Bei diesem Algorithmus werden häufige Symbolfolgen kompaktiert, so dass z.B. die Bitfolge 000001111000011111 zu (5*0), (4*1), (4*0), (5*1) komprimiert wird. Da festgelegt ist, dass die erste Zahl die Anzahl der Nullen angibt, kann man dafür einfach "5445" schreiben.

10. Hardware-Realisierung

In der Veröffentlichung von Georgios A. Dimitriadis [Dimitriadis 94] wird ein VLSI-Chip zur JPEG-Kompression beschrieben, der mit der arithmetischen Codierung arbeitet. Folgende Abbildung zeigt den Kodiervorgang:



Die AC- und DC-Koeffizienten, die von einem Quantisierungsmodul berechnet werden, werden in einem FIFO-Register zwischengespeichert. Jeder Koeffizient wird anhand einer Wahrscheinlichkeitsabschätzung codiert. Die Wahrscheinlichkeiten werden dabei dynamisch aktualisiert (adaptive Arithmetische Codierung).

Dimitriadis behandelt sowohl die Theorie als auch die Implementation sehr detailliert. Die 100-seitige Veröffentlichung geht dabei weit über den Rahmen dieser Studienarbeit hinaus.

11. Prinzipieller Aufbau des Java-Programms

Das Programm besteht aus zwei Klassen:

- *test* und
- *cacm87*

Die Klasse *test* dient lediglich dazu, eine minimale Benutzerschnittstelle zur Verfügung zu stellen. Sie kann jederzeit durch eine komfortablere Version ausgetauscht werden.

Die Klasse *cacm87* ist eine Java-Adaption von Michael Lecuyer [Lecuyer 98] des originalen C-Programms von Witten, Nealy und Cleary [Witten 87]. Diese Klasse stellt alle Methoden bereit, die zum Codieren und zum Decodieren notwendig sind. Lecuyer stellt außerdem noch die Klasse test zur Verfügung, um die Verwendung von cacm87 zu demonstrieren. Im Rahmen dieser Studienarbeit wurde *test* um einige Funktionen erweitert. Bei *cacm87* sind lediglich einige Kommentare und Ausgaberoutinen, die das Verständnis für die Funktionsweise verbessern sollen, hinzugekommen.

11.1 Überblick über die Methoden (Funktionen) von cacm87

Die Klasse *cacm87* enthält alle Funktionen[1], die man zum Codieren und Decodieren benötigt. Die folgenden Abschnitte ordnen einzelnen Funktionen kurz ein.

11.1.1 Public-Funktionen

Diese Funktionen können von einer Benutzerschnittstelle (z.B. von der Klasse *test*) direkt aufgerufen werden. Die eigentlichen Codier- und Decodierfunktionen sind private Funktionen und sind nach außen hin nicht sichtbar.

Name der Funktion	Beschreibung
public byte[] byteArrayCompress(byte inData[], int offset, int length)	Komprimieren eines Byte-Arrays in ein anderes Byte-Array.
public byte[] byteFileCompress(String fName)	Komprimieren einer Datei in ein Byte-Array.
public boolean fileCompress(String fIn, String fOut)	Komprimieren einer Datei in eine andere Datei.

[1] *Methode* ist die objektorientierte Bezeichnung für Prozeduren und Funktionen, die einer bestimmten Klasse zugeordnet sind. Da das besprochene Programm keine objektorientierte Sicht auf den Algorithmus bietet, wird zumeist die Bezeichnung *Funktion* verwendet.

public byte[] byteArrayDecompress(byte inData[], int offset, int length)	Dekomprimieren eines Byte-Arrays in ein anderes Byte-Array.
public byte[] byteFileDecompress(String fName)	Dekomprimieren einer Datei in ein Byte-Array.
public boolean fileDecompress(String fIn, String fOut)	Dekomprimieren einer Datei in eine andere Datei.

11.1.2 Initialisieren des Modells

Die folgenden Funktionen werden sowohl vor dem Codieren als auch vor dem Decodieren aufgerufen.

Name der Funktion	Beschreibung
cacm87()	Konstruktor der Klasse, muss beim jedem Codier- und Dekodiervorgang aufgerufen werden.
private void start_model()	Initialisierung des Modells und der verwendeten Arrays und Variablen.

11.1.3 Codieren

Der Codieralgorithmus ist in der Funktion *encode()* enthalten. Diese Funktion durchläuft nach dem Aufruf der Initialisierungsfunktionen eine Schleife, bis die Nachricht codiert ist.

Name der Funktion	Beschreibung
private void start_encoding()	Initialisierung der Variablen für die Bereichsangabe (high, low).
private void start_outputing_bits()	Initialisierung der Variablen für die bitweise Ausgabe.
private void encode()	Aufruf der Initialisierungsmethoden, danach Abarbeitung des kompletten Algorithmus (unter Verwendung diverser Hilfsfunktionen).
private void encode_symbol(int symbol)	Codierung eines Symbols.
private void done_encoding()	Diese Funktion wird am Ende des Kodiervorganges aufgerufen.
private void done_outputing_bits()	Übergabe der letzten Bits im Puffer an die Ausgabe-Funktion.
private void bit_plus_follow(int bit)	Ausgabe der Bits und der folgenden "opposite" Bits.
private int output_bit(int bit)	Fügt das entsprechende Bit zum Puffer hinzu und schreibt den Puffer bei Bedarf über writeByte() in einen Output-Stream.

11.1.4 Decodieren

Der Decodieralgorithmus ist in der Funktion decode() enthalten. Von hier werden auch die Initialisierungsfunktionenen aufgerufen.

Name der Funktion	Beschreibung
private void start_decoding()	Initialisierung der Variablen für die Bereichsangabe (high, low).
private void start_inputing_bits()	Initialisierung des Bit-Inputs.
private void decode()	Aufruf der Initialisierungsmethoden, danach Abarbeitung des kompletten Algorithmus (unter Verwendung diverser Hilfsfunktionen).
private int decode_symbol()	Decodierung eines Symbols.
private int input_bit()	Gibt ein Bit, das sich im Puffer befindet, zurück und füllt den Puffer bei Bedarf mit der Hilfsfunktion readByte() wieder auf.

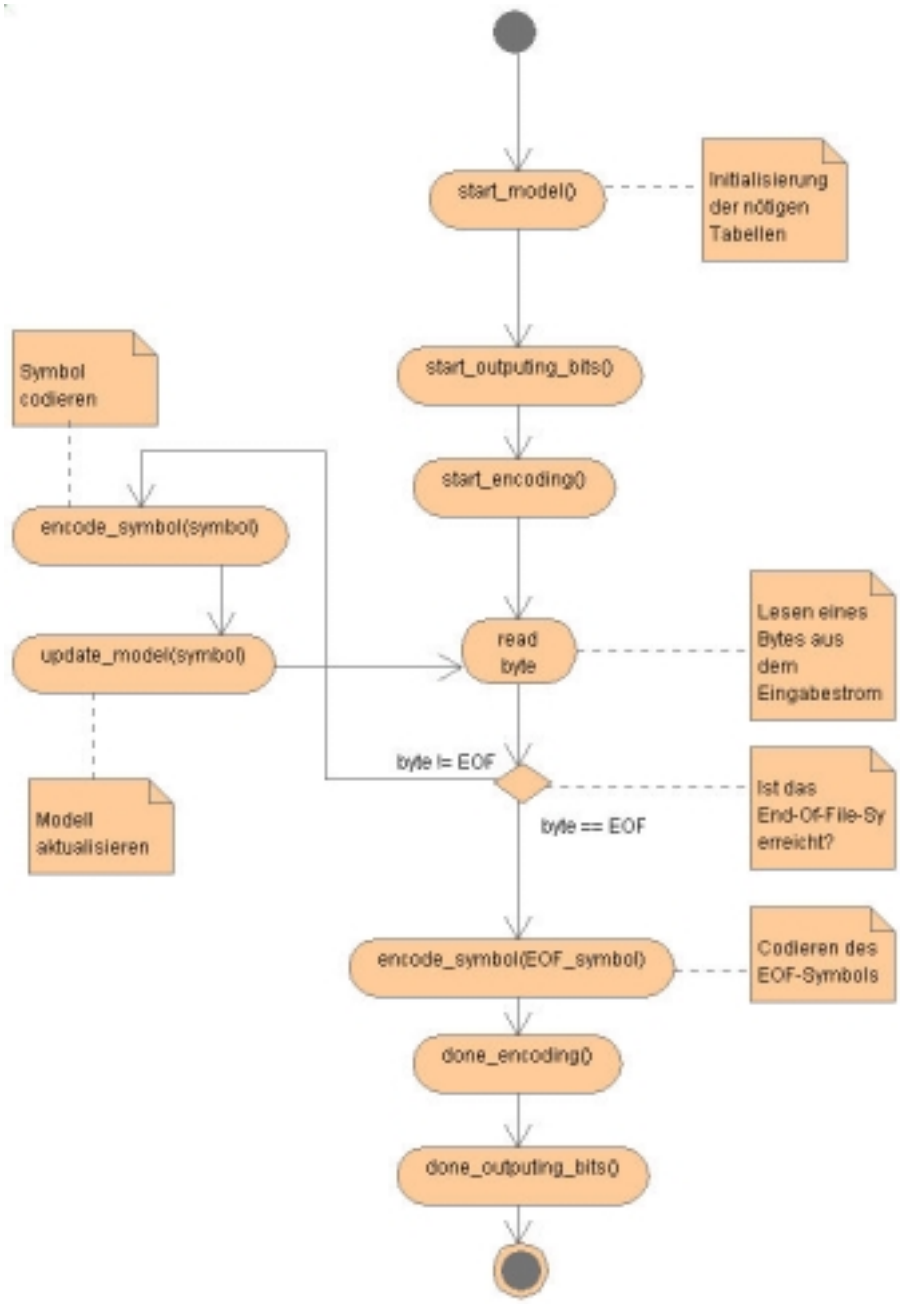
11.1.5 Hilfsfunktionen

Die folgenden Funktionen werden sowohl beim Codieren als auch beim Decodieren benötigt.

Name der Funktion	Beschreibung
private void update_model(int symbol)	Aktualisierung des Modells für ein bestimmtes Symbol.
private int readByte()	Lesen eines Bytes aus einem Input-Stream.
private void writeByte(int bt)	Schreiben eines Bytes in einen Output-Stream.

11.2 Der Kodiervorgang

Nach der Initialisierung des Modells die entsprechende Nachricht Byte für Byte gelesen. Das Byte wird dann in einen Index übersetzt und codiert. Danach muss noch das Modell aktualisiert werden. Ist das Ende der Datei erreicht, wird noch das EOF-Symbol codiert und der Kodiervorgang zum Abschluss gebracht. Der prinzipielle Ablauf beim Kodieren geht aus folgendem Ablaufdiagramm hervor.



Nach dem Start des Codierers wird die Methode *fileCompress* aufgerufen. Diese Methode öffnet die Streams, um die notwendigen Dateien lesen und schreiben zu können.

Anschließend wird *encode* aufgerufen. Mit *encode* wird der gesamte Kodiervorgang abgewickelt. Zunächst werden die Methoden

- *start_model*
- *start_outputing_bits* und
- *start_encoding*

aufgerufen. Diese Methoden initialisieren die notwendigen Arrays und setzen die entsprechenden Startwerte.

Nach der Initialisierung wird der eigentliche Kodiervorgang gestartet. Dabei wird die folgende *while*-Schleife durchlaufen:

```
while (true) {
        int ch = readByte();  // Read the next character.
        if (ch == EOF) break;  // Exit loop on end-of-file.
        int symbol = char_to_index[ch];   // Translate to an index.
        encode_symbol(symbol); // Encode that symbol.
        update_model(symbol);  // Update the model.
}
```

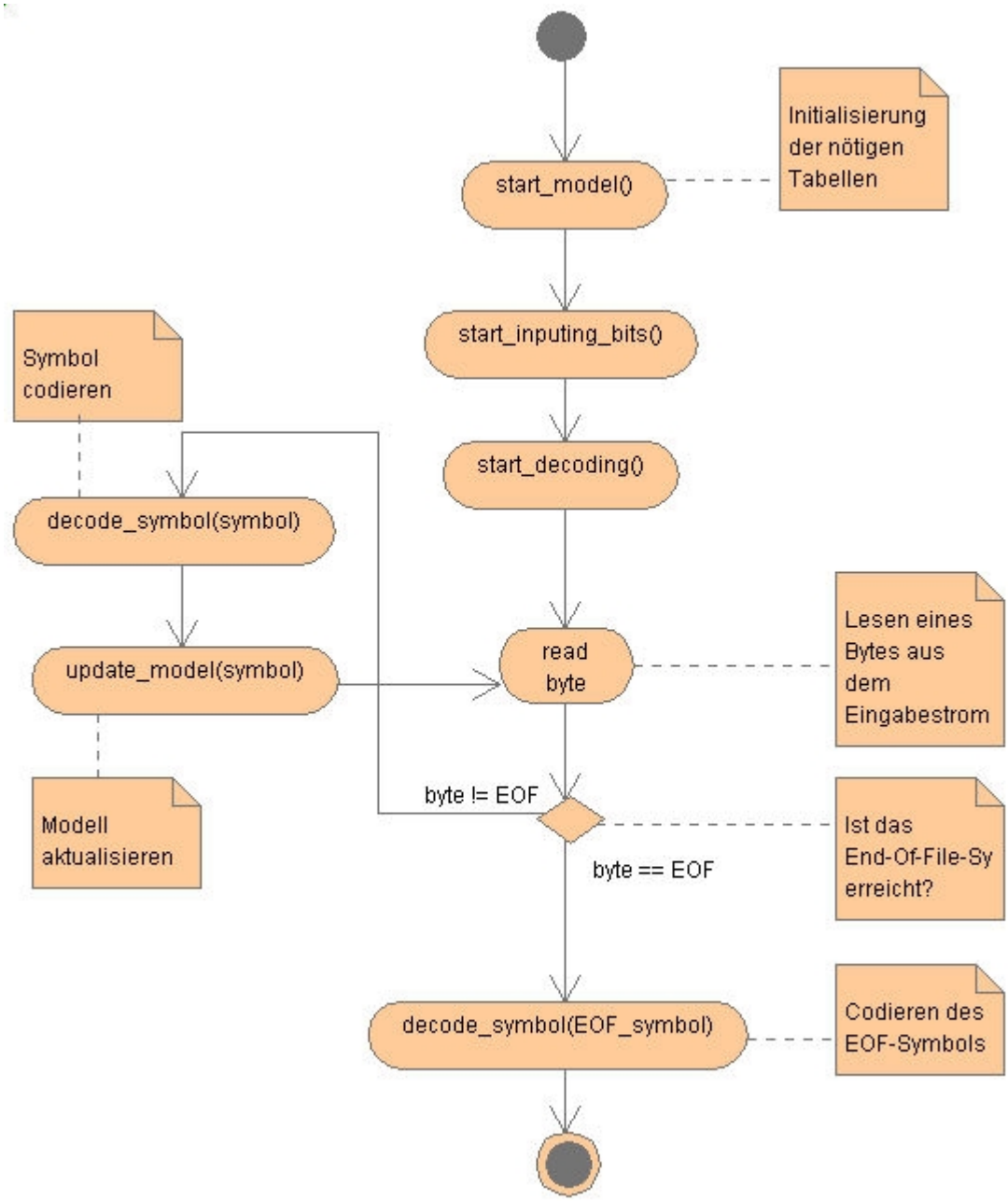
Es wird also aus dem Eingabestrom jeweils das nächste Zeichen gelesen, in einen Index übersetzt und codiert. Danach wird stets das Modell aktualisiert. Zum Abschluss wird noch das End-Of-File-Symbol codiert und der Kodiervorgang mit dem Aufruf der Methoden

- *done_encoding* und
- *done_outputing_bits*

beendet.

11.3 Der Dekodiervorgang

Der Dekodiervorgang läuft ähnlich wie der Kodiervorgang ab. Folgendes Ablaufdiagramm beschreibt den Vorgang:



Nach dem Start des Decodierers wird die Methode *fileDecompress* aufgerufen. Diese Methode öffnet die Streams, um die notwendigen Dateien lesen und schreiben zu können.

Anschließend wird *decode* aufgerufen. Mit *decode* wird der gesamte Dekodiervorgang abgewickelt. Zunächst werden die Methoden

- *start_model*
- *start_inputing_bits* und
- *start_decoding*

aufgerufen. Diese Methoden initialisieren die notwendigen Arrays und setzen die entsprechenden Startwerte.

Nach der Initialisierung wird der eigentliche Kodiervorgang gestartet. Dabei wird die folgende *while*-Schleife durchlaufen:

```
while (true) {
        int symbol = decode_symbol();  // Decode next symbol.
        if (symbol==EOF_symbol)        // Exit loop if EOF symbol.
                break;
        int ch = index_to_char[symbol]; // Translate to a character.
        writeByte(ch);                 // Write that character.
        update_model(symbol);          // Update the model.
}
```

Es wird also aus dem Eingabestrom jeweils das nächste Zeichen gelesen, in einen Index übersetzt und codiert. Danach wird stets das Modell aktualisiert. Zum Abschluss wird noch das End-Of-File-Symbol decodiert und der Dekodiervorgang beendet.

12. Details zur Implementation

Die folgenden Abschnitte geben eine detaillierte Beschreibung aller Funktionen, die beim Codieren und Decodieren verwendet werden. Einen tabellarischen Überblick über alle Funktionen von *cacm87* findet man weiter oben.

12.1 Das Modell

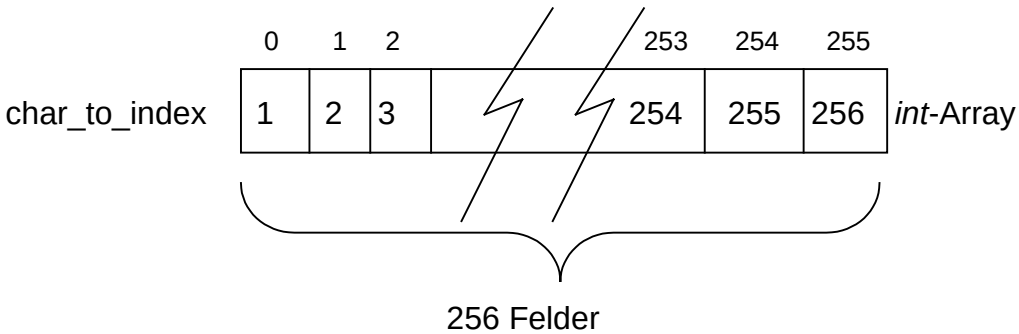
Die Methode *start_model* dient zur Initialisierung des Modells (also der Symbolwahrscheinlichkeiten). Dazu werden folgende Arrays definiert und mit den entsprechenden Startwerten vorbelegt:

- *char_to_index* (*int*-Array)
- *index_to_char* (*byte*-Array)
- *cum_freq* (*int*-Array)
- *freq* (*int*-Array)

Da die Bedeutung dieser Arrays wichtig für das Verständnis der Algorithmus ist, folgt in den nächsten Abschnitten eine detaillierte Beschreibung.

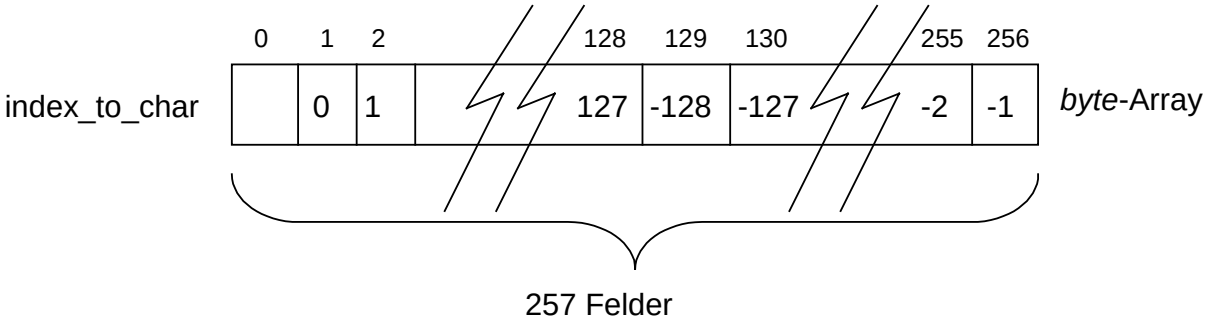
12.1.1 char_to_index[] und index_to_char[]

Diese Arrays übersetzen zwischen den Symbol-Indizes und den Symbolen. Dabei wird ein Byte als Integerzahl (Datentyp: *int*) zwischen 0 und 255 dargestellt. Außerdem soll das End-Of-File-Symbol noch einen separaten Wert haben. Zur Darstellung benötigt man also ein Array mit 255+1, also mit 256 Feldern. Dieses Array wird *char_to_index* genannt.



Das Array *char_to_index* ist ein *int*-Array mit 256 Feldern (256 entspricht der Anzahl der Symbole). Es wird in der Methode *start_model* folgendermaßen vorbelegt:

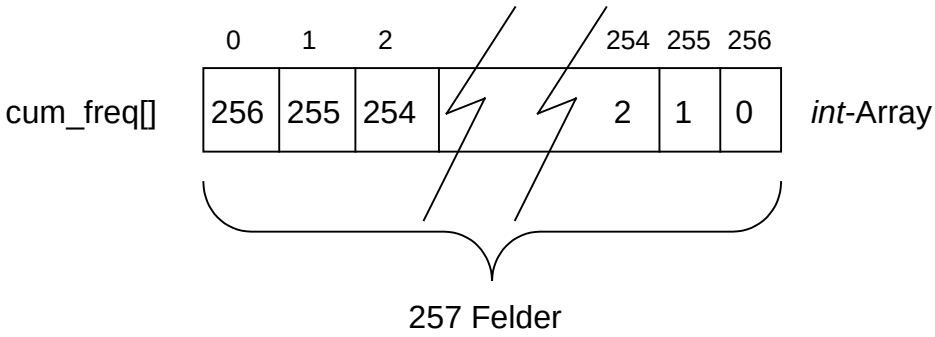
Um effizient auf dieses Array zugreifen zu können, sollte dieses Array nach der Häufigkeit der Symbole geordnet werden. Dazu wird also ein weiteres Array benötigt, das einen Index in ein Byte (also ein *char*) übersetzt. Das neue Array heißt sinnigerweise *index_to_char*:



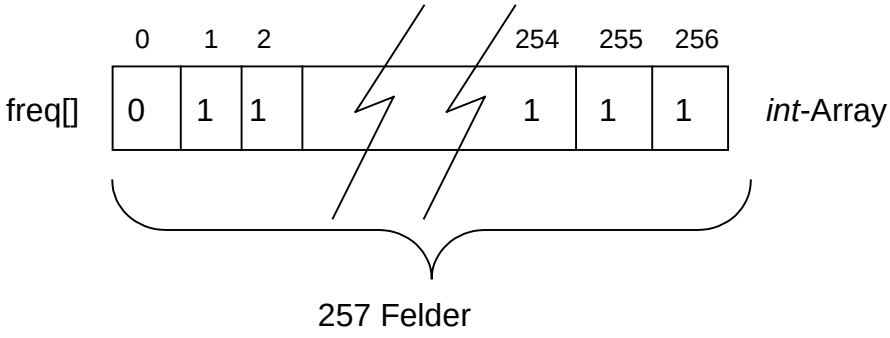
Anmerkung: Ein *byte* ist (in Java) ein vorzeichenbehafteter Integer-Datentyp mit einer Größe von 8 Bits (also 1 Byte). Damit sind 2^8, also 256 Werte möglich. Der Wertebereich geht dabei von –128 bis +128.

12.1.2 cum_freq[] und freq[]

Für die Darstellung der Häufigkeit eines Symbols werden die Arrays *cum_freq* und *freq* verwendet. Dabei steht *cum_freq[]* für die kumulierte Häufigkeit eines Symbols. Die absolute kumulierte Häufigkeit ist 256 und wird zur Normalisierung des Modells verwendet. Diese absolute kumulierte Häufigkeit hat den Index 0. Wenn i kleiner wird, wird cum_freq[i] größer. Grund dafür ist, dass die Häufigkeiten normalisiert werden müssen. Wenn man sich den Algorithmus (siehe unten) genauer ansieht, stellt man fest, dass die Berechnungen, für die *cum_freq[]* benötigt wird, einfacher werden, wenn man nicht einfach die Häufigkeiten bei 0 beginnen läßt und hochzählt.



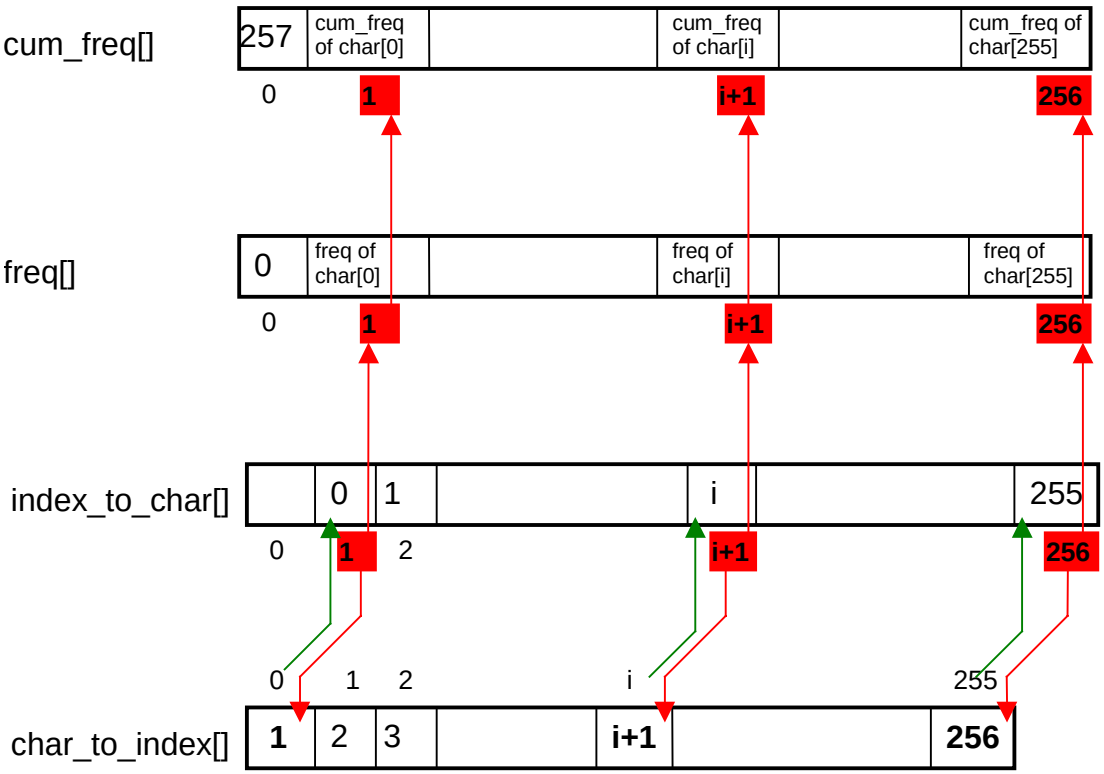
Bei der Initialisierung des Modells werden alle Häufigkeiten außer freq[0] gleich 1 gesetzt. freq[0] wird gleich 0 gesetzt:



Wie nun tatsächlich mit diesen Arrays gearbeitet wird, ist im Unterkapitel *update_model* näher beschrieben.

12.1.3 Zusammenhänge zwischen den Arrays

Das Zusammenspiel der einzelnen Arrays soll folgende Abbildung verdeutlichen:



Wie man sieht, dient das Array *index_to_char* dazu, um zu einem gegebenen Symbolindex das zugehörige Symbol, die zugehörige Häufigkeit und die zugehörige kumulierte Häufigkeit zu finden. Will man zu einem bestimmten Symbol den Index herausfinden, benutzt man das *char_to_index*-Array.

12.2 Aktualisierung des Modells

Nach der Codierung oder Decodierung eines Symbols muss das Modell mit *update_model(int symbol)* aktualisiert werden:

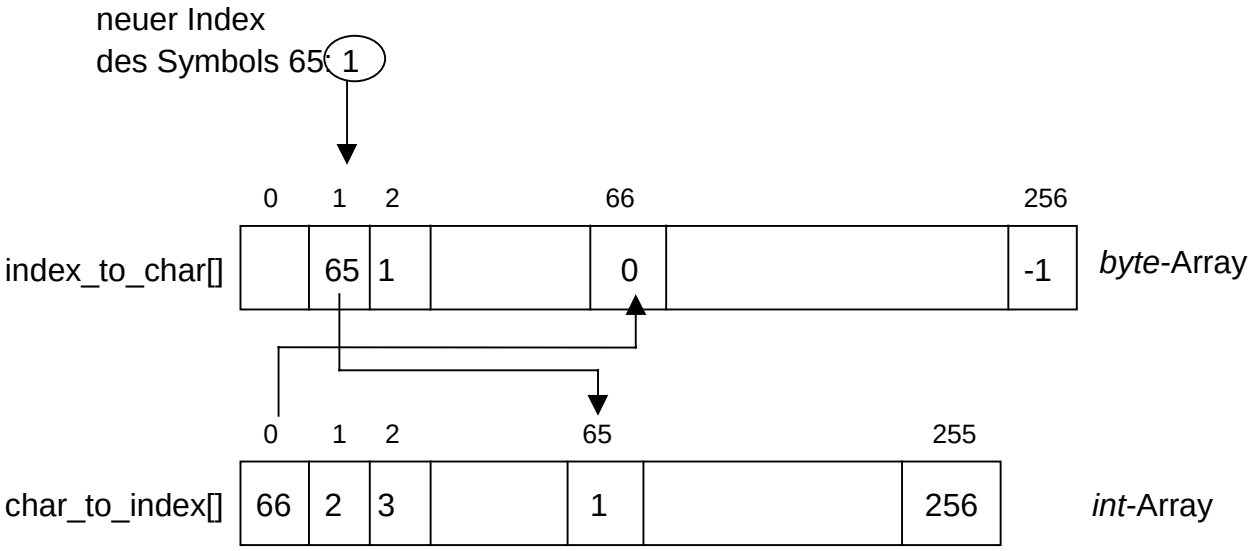
```
int i;      // New index for symbol
if (cum_freq[0] == Max_frequency) {      // See if frequency counts
        int cum;                         // are at their maximum.
        cum = 0;
        for (i = No_of_symbols; i >= 0; i--) {      // If so, halve all the
                freq[i] = (freq[i] + 1) / 2;        // counts (keeping
                cum_freq[i] = cum;                  // them non-zero).
                cum += freq[i];
        }
}

for (i = symbol; freq[i] == freq[i-1]; i--) ;   // Find symbol's new index.
if (i < symbol) {
        int ch_i;
        int ch_symbol;
        ch_i = index_to_char[i];                // Update the trans-
        ch_symbol = index_to_char[symbol];      // lation tables if the
        index_to_char[i] = (byte)ch_symbol;     // symbol has moved
        index_to_char[symbol] = (byte)ch_i;
        char_to_index[ch_i] = symbol;
        char_to_index[ch_symbol] = i;
}
freq[i]++;                          // Increment the frequency
while (i>0) {                       // count for the symbol and
        cum_freq[--i]++;            // update the cumulative frequencies.
}
```

Zuerst wird geprüft, ob die Häufigkeiten, die in *cum_freq[0]* gespeichert sind, das Maximum erreicht haben (*Max_frequency* hat den Wert 16383). Falls dies der Fall ist, werden die Arrays *cum_freq* und *freq* aktualisiert. Dabei werden in einer for-Schleife

- die Häufigkeiten (Array *freq*) halbiert und
- die kumulierten Häufigkeiten entsprechend den Werten in *freq[]* geändert.

Jetzt wird der neue Index des Symbols gesucht. Dabei wird einfach eine for-Schleife solange durchlaufen, bis *freq[i]* gleich *freq[i-1]* ist. i ist dann der neue Index.
Wenn der neue Index kleiner als der bisherige Symbol-Index ist, dann werden die Arrays *index_to_char* und *char_to_index* aktualisiert, damit das entsprechende Symbol auch mit dem neuen Index gefunden werden kann. Folgende Abbildung veranschaulicht den Aktualisierungsvorgang:



Jetzt wird die Häufigkeit (*freq[]*)mit dem aktuellen Index um 1 erhöht. Danach werden die kumulierten Häufigkeiten inkrementiert.
Anmerkung: Die Methode *update_model* muss nur bei der adaptiven arithmetischen Codierung verwendet werden. Bei der statischen Codierung (also mit festen Symbolwahrscheinlichkeiten) ist diese Methode leer. In der C-Implementation von Witten [Witten 87] kann entweder ein statisches Modell oder ein adaptives Modell verwendet werden.

12.3 Codierung eines Zeichens

Ein Zeichen wird mit der Routine *encode_symbol(int symbol)* codiert. Hier der relevante Code:

```
long range;                 // Size of the current code region
range = (long)(high-low)+1;

// Narrow the code region to that allotted to this symbol.
high = low + (range * cum_freq[symbol-1]) / cum_freq[0] - 1;
low = low + (range * cum_freq[symbol]) / cum_freq[0];

for (;;) {
        // Loop to output bits.
        if (high < Half) {
                bit_plus_follow(0);         // Output 0 if in low half.
        }
        else if (low >= Half) {
                bit_plus_follow(1); // Output 1 if in high half.
                low -= Half;
                high -= Half;               // Subtract offset to top.
        }
        else if (low>=First_qtr && high<Third_qtr) {
                // Output an opposite bit later if in middle half.
                bits_to_follow++;
```

```
                low -= First_qtr; // Subtract offset to middle
                high -= First_qtr;
        }
        else break;       // Otherwise exit loop.

        low = 2*low;
        high = 2*high+1;          // Scale up code range.
}
```

Zuerst wird der Bereich anhand der oberen (*high*) und der unteren Grenze (*low*) berechnet. Danach wird der Bereich, der diesem Symbol zugewiesen wurde, verkleinert. Dazu wird das Array *cum_freq* benutzt, das die kumulierten Häufigkeiten der Symbole enthält. Der neue Wert von *high* wird durch

$$low + \frac{\text{range} \cdot \text{cum\_freq[symbol - 1]}}{\text{cum\_freq[0] - 1}}$$

berechnet, der neue Wert von *low* wird ermittelt durch

$$low + \frac{\text{range} \cdot \text{cum\_freq[symbol]}}{\text{cum\_freq[0]}}$$

Mit diesen Formeln wird der aktuelle Bereich entsprechend den kumulierten Häufigkeiten angepasst und durch Division normalisiert.

Jetzt müssen dem Symbol die entsprechenden Bits zugeordnet werden. Diese Zuordnung hängt von dem aktuellen Bereich ab. Hier gibt es drei Möglichkeiten:

- Die obere Grenze liegt im unteren Bereich (d.h. sie ist kleiner als die Hälfte des gesamten Bereichs). Dann wird eine 0 ausgeben.
- Die untere Grenze liegt im oberen Bereich (d.h. sie ist größer oder gleich der Hälfte des gesamten Bereichs). Dann wird eine 1 ausgegeben.
- Der Bereich liegt in der Mitte des gesamten Bereichs (d.h. *low* ist größer oder gleich dem ersten Viertel und *high* ist kleiner oder gleich dem dritten Viertel). In diesem Fall wird kein Wert ausgegeben, vielmehr wird von *high* und *low* das erste Viertel abgezogen.

Die beiden ersten Punkte sind leicht nachzuvollziehen. Der dritte Punkt dagegen ist zunächst schwer zu verstehen. Hier wird eigentlich nur einem möglichen Underflow vorgebeugt. Wie man einen Underflow verhindert, wird im nächsten Abschnitt näher beschrieben. Anschließend wird der Bereich "hochskaliert", d.h. *low* und *high* werden mit 2 multipliziert. Zu *high* wird zusätzlich noch 1 addiert. Diese Skalierung wird vorgenommen, um zu erreichen, dass $low < Half \leq high$ ist.

12.4 Verhinderung eines Underflows

Ein Underflow kann eintreten, wenn *high* und *low* zu dicht aneinander liegen. Um dies zu verhindern, gibt es verschiedene Möglichkeiten. Die Implementation von Witten [Witten 87] und die Java-Adaption von Lecuyer [Lecuyer] verwenden folgende Methode:

Wenn sich der aktuelle Bereich zwischen

$$First\_qtr \leq low < Half \leq high < Third\_qtr$$

befindet, dann unterscheiden sich *low* und *high* in der ersten Binärziffer nach dem Komma. Das heißt, man kann zunächst nicht entscheiden, ob eine 0 oder eine 1 ausgegeben werden soll. Also wird von *low* und *high* ein Viertel des gesamten Bereichs (also 16384 beim Standard-Bereich von 0 bis 65535) abgezogen, damit der aktuelle Bereich nach der Skalierung wieder im Bereich [0, 65535) liegt.

Außerdem wird der Zähler *bits_to_follow* um 1 erhöht, um zu zeigen, dass bei diesem Bit der Wert noch nicht feststand. Das nächste Bit entscheidet dann, ob hier eine 0 oder eine 1 hätte ausgegeben werden müssen. Wenn beim nächsten Schleifendurchlauf der erste Fall auftritt, muss 10 ausgegeben werden und wenn der zweite Fall auftritt, muss 01 ausgegeben werden. Auch beim Decodieren muss ein Underflow verhindert werden. Dazu wird prinzipiell der gleiche Algorithmus verwendet.

Nähere Details zu Underflow-Verhinderung findet man im Witten-Artikel.

12.5 Verhinderung eines Overflows

Ein Overflow kann bei der Integer-Multiplikation eintreten, wenn nicht verhindert wird, dass $range \cdot Max\_frequency$ außerhalb des Bereichs des verwendeten Datentyps liegt. Wenn der für *range* und für *code_value* verwendete Datentyp (wie in der C/Java-Implementation) *long* ist, dann kann dieser Fall nicht auftreten, da *range* maximal so groß wird wie Top_value + 1 (also 65536).

12.6 Decodierung eines Zeichens

Die Decodierung eines Zeichens (*decode_symbol*) funktioniert ähnlich wie die Codierung. Allerdings wird eine zusätzliche Variable benötigt: *value*. Diese Variable enthält die jeweils aktuellen Bits, aus denen das entsprechende Symbol ermittelt werden soll.

```
long range;                 // Size of current code region
int cum;                    // Cumulative frequency calculated
int symbol;                 // Symbol decoded

range = (long)(high-low)+1;

// Find cum freq for value.
cum = (int)(((value - low + 1) * (long)cum_freq[0] - 1) / range);

 // Then find symbol.
for (symbol = 1; cum_freq[symbol] > cum; symbol++) ;

// Narrow the code region to that allotted to this symbol
high = low + (range*cum_freq[symbol-1])/cum_freq[0]-1;
low = low + (range*cum_freq[symbol])/cum_freq[0];

for (;;) {              // Loop to get rid of bits.
        if (high<Half) { // Expand low half
                // nothing
        }
        else if (low>=Half) {       // Expand high half.
                value -= Half;
                low -= Half;        // Subtract offset to top.
                high -= Half;
        }
        else if (low>=First_qtr    && high<Third_qtr) {
                // Expand middle half.
                value -= First_qtr;
                low -= First_qtr;           // Subtract offset to middle
                high -= First_qtr;
        }
        else break;         // Otherwise exit loop.
        low = 2 * low;
        high = 2 * high + 1;        // Scale up code range.
        value = 2 * value + input_bit();    // Move in next input bit.
}
return symbol;
```

12.7 Ausgaberoutinen

Während des Programmablaufs werden einzelne Bits in einen Puffer geschrieben. Ist der Puffer voll (d.h. ist die Anzahl der Bits gleich 8), wird mit *writeByte()* der Pufferinhalt als Integerzahl in die Ausgabedatei geschrieben.

13. Bedienung des Programmes

13.1 Allgemeines

Das Java-Programm wurde mit dem JDK 1.2 kompiliert. Es ist aber auch möglich, das Programm mit einer Laufzeitumgebung der Version 1.0 oder 1.1 zu starten. Eine Java-Laufzeitumgebung findet man unter http://java.sun.com.

13.2 Codierung

Zur Codierung ruft man das Java-Programm auf mit:

```
java test –compress sourceFileName destinationFileName
```

Damit wird die angegebene Quelldatei komprimiert und unter dem Namen *destinationFileName* gespeichert. Übrigens: Wenn man nur

```
java test
```

eingibt, wird die Datei *source.txt* gelesen und nach *compress.txt* komprimiert. Anschließend wird *compress.txt* dekomprimiert und unter dem Namen *uncompress.txt* abgespeichert. Die Dateien *source.txt* und *uncompress.txt* sollten somit identisch sein.

13.3 Decodierung

Um eine Datei zu dekomprimieren, gibt man ein:

```
java test –decompress sourceFileName destinationFileName
```

Damit wird die Datei *sourceFileName* dekomprimiert und die dekomprimierte Datei unter *destinationFileName* gespeichert.

13.4 Zusätzliche Informationen

Ist man an der genauen Funktionsweise des Algorithmus interessiert, stehen zwei Flags zur Verfügung:

- Anzeige der aufgerufenen Funktionen.
- Detaillierte Anzeige des Kodier- oder Dekodiervorganges.

Will man nur eine Ausgabe der Funktionsnamen gibt man ein:

```
java test –compress –methods sourceFileName destinationFileName
```

Eine detaillierte Auswertung wird angezeigt mit:

```
java test –compress –details sourceFileName destinationFileName
```

14. Literatur

[Bloom]
Charles Bloom, "Statistical Coders";
http://www.cbloom.com/algs/statisti.html

[Compression FAQ 99]
"Compression FAQ, Part 1"
http://www.faqs.org/faqs/compression-faq/part1/

[Dimitriadis 94]
Georg A. Dimitriadis, "An Arithmetic Entropy Codec VLSI Chip for JPEG Image Compression", Technical Report FORTH-ICS/TR-114, January 1994, Institute of Computer Science, Foundation for Research and Technology Hellas; http://archvlsi.ics.forth.gr/other.html

[Flanagan 97]
David Flanagan, "Java in a Nutshell", 2nd Edition; Verlag O'Reilly

[Held 96]
Gilbert Held, Thomas R. Marshall; "Data and Image Compression" Verlag John Wiley & Sons Ltd.

[Henning]
Prof. Dr. Peter Henning, Multimedia Skript WS 99/00, FH Karlsruhe

[Hoffmann]
Prof. Josef Hoffmann; "Die Entropie als Maß der Information"

[Kuhn 95]
Markus Kuhn; "Effiziente Kompression von bi-level Bilddaten durch kontextsensitive Arithmetische Codierung"; Studienarbeit an der Friedrich-Alexander-Universität Erlangen-Nürnberg, 1995

[Lecuyer 98]
Michael Lecuyer; "Adaption der CACM-Implementation von Witten" (Software); Theorem, 1998; http://www.theorem.com/java/Free.htm

[Nelson 91]
Mark Nelson; "Arithmetic Coding + Statistical Modeling = Data Compression"; Dr. Dobb's Journal, 1991

[Salomon 98]
David Salomon; **Data Compression**, The complete reference, 1998, Springer-Verlag New York

[Witten 87]
I. Witten, R. Neal, J. Clearly; "Arithmetic Coding for Data Compression"; Dr. Dobb's Journal, 1987