Universität Paderborn
Institut für Informatik
Fachgebiet Codes und Kryptograhie
Fürstenallee 11
33102 Paderborn

Arithmetische Kodierung
Implementierung eines arithmetischen Dekodierers in Java

vorgelegt von
Claudia Unterkircher
Matrikelnummer: 6055918
15.10.2003

Betreuer:
Prof. Dr. rer. nat. Johannes Blömer

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die folgende Arbeit selbstständig und ohne unerlaubte fremde Hilfe angefertigt, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und den benutzten Quellen wörtlich oder inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Paderborn, den 15. Oktober 2003

Inhaltsverzeichnis

1 Einleitung

Bei der Speicherung und Übertragung von Daten spielt die Kompression eine große Rolle, da viele der heutigen Anwendungen wie Bild - oder Videoübertragungen ohne eine Minimierung des Speicherbedarfs der Daten durch mathematische Verfahren nicht praktikabel wären.
Eines der Kompressionsverfahren ist die arithmetische Kodierung, die in dieser Ausarbeitung näher erläutert wird. Hierbei wird jedoch hauptsächlich auf die Dekodierung der Daten eingegangen, da die Kodierung in der Studienarbeit „Implementierung eines arithmetischen Kodierers in Java“ [Tod03] behandelt wird.
Als Grundlage für die vorliegende Implementierung wurde ein vorhandener Quelltext in C [Wit87b] sowie der Quelltext des arithmetischen Kodierers [Tod03] genutzt.

1.1 Datenkompression

Als Datenkompression bezeichnet man Verfahren zur zeitweiligen Verkleinerung der benötigten Speicherkapazität von Daten.
Zu einem solchen Verfahren gehört in der Regel nicht nur eine Methode zur Reduktion der benötigten Speicherkapazität (Kompression), sondern auch zur Wiederherstellung der Daten in ihrer ursprünglicher Form (Dekompression). Die Datenkompression wird teils durch günstigere Repräsentation, d.h. Vermeiden von redundanten und oft wiederholten Informationen, teils durch Weglassen von mehr oder weniger unwichtigen Informationen erreicht.
Man unterscheidet verlustfreie und verlustbehaftete Datenkompression, je nachdem, ob die Daten mit einem Dekompressionsverfahren wieder originalgetreu hergestellt werden können (verlustfrei) oder nicht (verlustbehaftet). Verlustfreie Kompression wird insbesondere dort eingesetzt, wo eine Reduktion die Daten vollständig unbrauchbar machen würde z.B. bei Texten. Verlustbehaftete Kompression hingegen wird bei großen Datenmengen verwendet, wo Detailinformationen vernachlässigt werden können z.B. geringe Farbunterschiede bei Bildern.
Bei der verlustfreien Datenkompression werden also nur redundante Informationen verkürzt dargestellt, während bei der verlustbehafteten Datenkompression zusätzlich weniger wichtig erscheinende Informationen weggelassen werden.
Die arithmetische Kodierung ist wie die Huffman Kodierung ein verlustfreies

Komprimierungsverfahren. Indem der gesamte Datenstrom zu einem einzigen Code komprimiert wird, lässt sich eine hohe Effizienz auch in den Fällen erreichen, in denen die Huffman Methode eine vergleichsweise schlechte Rate erzielen würde.

1.2 Historische Entwicklung

1948 veröffentlichte Shannon erste Ansätze zur Verwendung von Verteilungsfunktionen zu Code-Generierung, hieraus ist die bekannte Shannon-Fano-Elias-Methode entstanden, die in Abschnitt 2.4 näher erläutert wird.
Theoretisch ein eher einfaches Verfahren, ergaben sich doch einige Schwierigkeiten in der Implementierung. 1963 unternahmen Elias und Abramson erste Versuche in dieser Richtung, Jelinek entwickelte den Algorithmus 1968 weiter, aber auch er scheiterte am Problem der begrenzten Genauigkeit von Fließkommazahlen. 1976 entwickelten Pasco und Rissanen unabhängig voneinander eine Lösung für dieses Problem, die jedoch in Bezug auf die Speichernutzung noch nicht effizient war.
Erst 1980 wurden Artikel von Rubin, Guazzo, Rissanen und Langdon veröffentlicht, die einen Algorithmus beschreiben, der auch heute noch zur arithmetischen Kodierung benutzt wird. Er basiert auf Arithmetik endlicher Genauigkeit und arbeitet zeichenweise nach der FIFO-Methode, so dass Teile des Codes bereits dekodiert werden können, während die Komprimierung des restlichen Codes noch durchgeführt wird.
Somit ist die arithmetische Kodierung zwar ein relativ junges Verfahren, aber trotzdem ausgereift und effizient.

2 Theoretische Grundlagen

In diesem Abschnitt werden die wichtigsten Definitionen und Sätze vorgestellt, diese basieren auf [Blo03].

2.1 Grundlagen der Datenkompression

Definition 2.1.1 (Alphabet und Symbol)
Ein Alphabet ist eine endliche, nichtleere Menge $A = \{a_1, \ldots, a_n\}$*, wobei die* a_i*'s die Symbole des Alphabets sind, die möglicherweise auch aus mehreren Zeichen bestehen.*
A^n *bezeichnet die Menge aller Worte mit* n *Symbolen aus* A*, also* $A^n = \{b_1 b_2 \ldots b_n |\ b_i \in A \text{ für } i = 1, \ldots, n\}$*,* $n \geq 1$*.*
Man definiert $A^* := \bigcup_{n=0}^{\infty} A^n$ *somit als die Menge aller Worte endlicher Länge über* A*.*

Definition 2.1.2 (Quelle und Sequenz)
Unter einer Quelle versteht man einen Strom von Symbolen aus A*, eine Folge* $S = (a_{i_1}, \ldots, a_{i_m})$ *mit* $m \geqq 1$ *und* $i_j \in \{1, \ldots, n\}$ *der Länge* m *aus* A^* *wird als Sequenz bezeichnet.*

Betrachtet man eine gegebene Sequenz, so lässt sich feststellen, dass einzelne Symbole mit einer gewissen Häufigkeit auftauchen, die teilweise stark variieren kann, z.B. kommt in deutschen Texten das e wesentlich häufiger auf als das y. Da die arithmetische Kodierung unter anderem auf dieser Häufigkeitsanalyse beruht, definiert man die Wahrscheinlichkeit eines Symbols wie folgt:

Definition 2.1.3 (Wahrscheinlichkeit)
Jedem Symbol a_i *ist eine Häufigkeit* F_i *bzw. eine Wahrscheinlichkeit* $p_i = \frac{F_i}{n}$*,* $i = 1, \ldots n$ *zugeordnet, wobei* n *die Anzahl der Symbole in der Quelle bezeichnet.*
Es gilt

$$\sum_{i=1}^{n} p_i = 1,\ p_i \in [0, 1),$$

da die p_i *eine Wahrscheinlichkeitsverteilung bilden.*

Nimmt man an, dass die Wahrscheinlichkeiten der Symbole unabhängig voneinander sind, so ist p_i an jeder Stelle in der Sequenz gleich, d.h., die vorhergehenden Symbole haben keine Auswirkungen auf die Wahrscheinlichkeit des aktuellen Symbols.
Somit ergibt sich die folgende Darstellung, wobei X_j eine Zufallsvariable für die Position j der Sequenz darstellt:

$$p\left(X_j = a_i\right) = p_i \ , \ i = 1, \ldots, n \ , \ j = 1, \ldots, s$$

In der vorliegenden Implementierung bzw. Ausarbeitung wurden jedoch kontextabhängige Wahrscheinlichkeiten benutzt, d.h., abhängig von den zuvor gelesenen Symbolen (Kontext) kann die Wahrscheinlichkeit des folgenden Symbols variieren (s. Order-n-Modell).

Definition 2.1.4 (Modell)
Als Modell M wird eine Abbildung

$$\begin{aligned} \phi : A &\to [0, 1) \\ a_i &\mapsto p_M(a_i) \end{aligned}$$

bezeichnet, die jedem Symbol $a_i \in A$ eine Wahrscheinlichkeit $p_M(a_i)$ zuordnet.

Diese ist errechnet und muss nicht unbedingt mit der tatsächlichen Wahrscheinlichkeit p_i übereinstimmen, im Folgenden wird jedoch davon ausgegangen und deshalb einheitlich p_i als Bezeichnung verwendet.
Da die p_i eine Wahrscheinlichkeitsverteilung bilden, ist der Bildraum durch das halboffene Intervall [0,1) gegeben. Würde eine Wahrscheinlichkeit von 1 auftreten, so müsste die Sequenz nicht kodiert werden, da sie schon vollständig bekannt ist.

Es gibt statische, dynamische und adaptive Modelle.
Ein statisches Modell beinhaltet eine feste Häufigkeitsverteilung, diese wird während des Kodierungsprozesses nicht verändert, d.h. ein häufiges bzw. vermindertes Auftreten eines Symbols hat keine Auswirkungen auf seine Wahrscheinlichkeit.
Dynamische (auch semi-adaptiv genannte) Modelle benutzen ebenfalls feste Häufigkeitsverteilungen, jedoch werden diese für jede Sequenz neu berechnet, indem anfangs die Sequenz eingelesen und daraus die entsprechende Wahrscheinlichkeitsverteilung ermittelt wird. Dann wird die Eingabesequenz gemäß dieser Verteilung kodiert, dies bedeutet zwar einen höheren Zeitbedarf, jedoch ist durch die genaue Anpassung an die tatsächlichen Häufigkeiten ein optimales Resultat zu erwarten.
Bei adaptiven Modellen hingegen wird keine feste Häufigkeitsverteilung benutzt, sondern die Häufigkeiten ergeben sich während des Kodierens durch ihr Auftreten

in der Sequenz. Meist wird als Ausgangsverteilung die Gleichverteilung gewählt und dann die Häufigkeit der jeweiligen Symbole beim Kodieren sukzessiv erhöht, die ideale Datenstruktur für diesen Ansatz ist ein balancierter binärer Baum. Dem Dekodierer muss die vorgegebene Häufigkeitsverteilung ebenfalls bekannt sein, damit das erste Zeichen dekodiert werden kann, im weiteren Prozess wird das Modell analog zur Vorgehensweise des Kodierers angepasst.

Wird der Kontext, also die vorherigen Symbole, zur Bestimmung der Wahrscheinlichkeit für das aktuelle Symbol betrachtet, so bezeichnet man dies als *Order-n-Modell*; hierbei definiert n die Länge des Kontextes. Wird z.B. 'sc' gelesen, so ist die Wahrscheinlichkeit in einem deutschen Text recht groß, dass ein 'h' folgen wird.

Somit ist es einleuchtend, dass die Wahl des Modells entscheidend für die Kompression ist; je mehr sich Modell und Häufigkeitsverteilung einer Sequenz gleichen, desto bessere Ergebnisse lassen sich erzielen. Um die Effizienz der Kompression bestimmen zu können, benötigt man den Begriff der Entropie, der in Abschnitt 2.3.4 näher erläutert wird.

In der Implementierung wurde ein statisches Order-1-Modell realisiert, d.h., man betrachtet das vorherige Symbol und wählt dementsprechend die passende Wahrscheinlichkeitsverteilung aus.

Definition 2.1.5 (Kodierung)
Bei einem gegebenen Alphabet A bezeichnet die Kodierung von A eine injektive Abbildung

$$\begin{aligned} C : A &\to \{0,1\}^* \\ a_i &\mapsto C(a_i) \end{aligned}$$

Die $C(a_i)$ werden auch mit c_i bzw. als Codeworte bezeichnet, C wird Code genannt. Es muss gelten: $C(a_i) \neq C(a_j),\ i \neq j,\ i,j \in \{1, \ldots, n\}$, da sonst für 2 verschiedene Symbole die gleiche Kodierung existieren und somit das Urbild durch den Dekodierer nicht rekonstruiert werden könnte.

Da viele kombinatorisch interessante Eigenschaften eines Codes Eigenschaften der Menge der Codeworte $\{c_1, \ldots, c_n\}$ sind, werden diese ebenfalls mit C bezeichnet.

Durch die Abbildung kann jeweils nur ein Symbol kodiert werden, daher weitet man diese auf Sequenzen durch folgende Definition aus:

$$C(S) = C(a_{i_1}) \ldots C(a_{i_m}).$$

2.2 Dekodierung

Um aus der Kodierung einer Sequenz wieder die Originalsequenz herstellen zu können, muss der Code bestimmte Kriterien erfüllen.

Definition 2.2.1 (Eindeutig entschlüsselbar)
Ein Code ist eindeutig entschlüsselbar, falls jedes Element aus $\{0,1\}^*$ *Bild höchstens einer Sequenz ist, d.h. die Abbildung* C *erweitert auf* A^* *muss injektiv sein.*

Beispiel 2.2.1
Sei $A = \{a_1, a_2\}$ *und* C *definiert durch* $C(a_1) = 0, C(a_2) = 01$. *Dann ist* C *ein eindeutig entschlüsselbarer Code.*

So kann z.B. aus der Kodierung $C(S) = 00100010101$ die Originalsequenz $S = a_1a_2a_1a_1a_2a_2a_2$ hergestellt werden, indem die Kodierung von rechts nach links gelesen werden, da die 0 jeweils den Beginn eines Codewortes markiert.
Somit muss also die vollständige Codierung der Sequenz bekannt sein, um diese zu entschlüsseln.
Um schon Teile einer Sequenz dekodieren zu können, während der Rest noch kodiert wird, muss auf ein stärkeres Kriterium zurückgegriffen werden.

Definition 2.2.2 (Präfixcode)
Ein Code C *heißt Präfixcode, falls es keine Indizes* $1 \leq i, j \leq n, i \neq j$, *gibt, so dass* $C(a_i) = c_i$ *ein Präfix von* $C(a_j) = c_j$ *ist, also* c_i *keine Teilfolge am Anfang von* c_j *darstellt.*

Somit ergibt sich, dass das obige Beispiel keinen Präfixcode darstellt, da $C(a_1)$ ein Präfix von $C(a_2)$ ist. Eine geringe Änderung liefert:

Beispiel 2.2.2
Sei $A = \{a_1, a_2\}$ *und* C *definiert durch* $C(a_1) = 0,\ C(a_2) = 10$. *Dann ist* C *ein Präfixcode, der auch als Komma-Code bezeichnet wird, da 0 das Ende der Kodierung des jeweiligen Symbols charakterisiert.*

Somit ist bei Erreichen einer 0 das Codewort und damit auch die Dekodierung eindeutig bestimmt, daher gehören Präfixcodes zur Klasse der eindeutig dekodierbaren Codes [Say00].

Benutzt man zur Dekodierung eines Präfixcodes einen binären Baum, bei dem die Kanten mit 0 (links) bzw. 1 (rechts) und die Blätter mit den Codeworten gelabelt sind, so darf kein Codewort auf dem Pfad zu einem anderen Codewort liegen. Dann lässt sich leicht zeigen:

Satz 2.2.1
Zu jedem Präfixcode C existiert ein Dekodieralgorithmus, der bei Eingabe $C(S)$, $S \in A^*$, die Originalsequenz S in linearer Zeit in der Länge von $C(S)$ berechnet.

2.3 Effizienz

Um die Effizienz einer Kodierung bestimmen zu können, benötigt man die folgenden Begriffe.

Definition 2.3.1 (Erwartete Länge eines Codes)
Es sei eine Quelle mit Alphabet $A = \{a_1, \ldots, a_n\}$ und den zugehörigen Wahrscheinlichkeiten $p_1, \ldots, p_n$ gegeben. Für $x \in 0, 1^*$ bezeichne $|x|$ die Länge von x, d.h. die Anzahl der Bits in x.
Dann ist die erwartetete Länge eines Codes $C : A \to \{0, 1\}^*$ definiert durch:

$$E(C) := \sum_{j=1}^{n} p_j |C(a_j)|$$

Dies spiegelt den Grundgedanken der Kompression wieder, da die Symbole, deren Wahrscheinlichkeit relativ groß ist, durch Codeworte kurzer Länge kodiert werden sollten.
$E(C)$ ist somit der Erwartungswert der auf A definierten Zufallsvariablen, die jedem Symbol als Wert die Länge seiner Kodierung zuweist.
Um eine gute Kompression zu erreichen, muss dieser Erwartungswert also minimiert werden. Aufgrund der Sätze von Kraft und McMillan (vgl. [Mof02]) werden im Folgenden nur Präfixcodes betrachtet.

Definition 2.3.2 (Kompakte Codes)
Gegeben sei eine Quelle mit Alphabet $A = \{a_1, \ldots, a_n\}$ und den zugehörigen Wahrscheinlichkeiten $\{p_1, \ldots, p_n\}$. Dann bezeichnet man einen Code C als kompakt, falls er ein Präfixcode mit minimaler erwarteter Länge ist.

Durch einen kompakten Code wird also die bestmögliche Kompressionsrate erreicht, diese wird durch die Entropie gemessen.
Die Entropie ist der „mittlere Informationsgehalt pro Symbol“, dieser ist abhängig von der Auftrittswahrscheinlichkeit. Bei $p_i = 1$ ist das Symbol dem Dekodierer schon bekannt und hat daher keinen Informationsgehalt, ein selten auftretendes Symbol hingegen besitzt einen hohen Informationsgehalt.

Definition 2.3.3 (Informationsgehalt)
Der Informationsgehalt eines Symbols a_i wird hier definiert durch:

$$I(a_i) = -\log_2 p_i,$$

da sich dieser auf die Menge der benötigten Bits pro Symbol bezieht.

$H(p_1, \ldots, p_n)$ steht für die Entropie einer Quelle mit einem Alphabet A mit n Zeichen, die mit den Wahrscheinlichkeiten $\{p_1, \ldots, p_n\}$ auftreten.

Definition 2.3.4 (Entropie)
Die Entropie ist definiert als

$$H(p_1, \ldots, p_n) = -\sum_{i=1}^{n} p_i \log_2 p_i,$$

hierbei bezeichnet $\log_2 p_i$ die minimale Länge eines binären Codes für das Symbol a_i und p_i die Häufigkeit, mit der dieses Symbol kodiert werden muss.

Definition 2.3.5 (Alphabet A^b)
Betrachtet man statt $A = \{a_1, \ldots, a_n\}$ nun das Alphabet A^b, so ist die Wahrscheinlichkeit $p(x)$ eines b-Tupels $x = (x_1, \ldots, x_b)$ gegeben als das Produkt der einzelnen Wahrscheinlichkeiten der $x_i \in A$:

$$p(x) = \prod_{i=1}^{b} p(x_i)\,.$$

Notwendige Eigenschaften der Entropie:

- $H(p_1, \ldots, p_n) \geq 0$
- Die Funktion H ist stetig.
- $H(pq) = H(p) + H(q)$, wobei p und q zwei unabhängige Wahrscheinlichkeitsverteilungen sind.
- $H(p^n) = n \cdot H(p),\ \forall n \in \mathbb{N}$

Satz 2.3.1
Es sei eine Quelle mit Alphabet $A = \{a_1, \ldots, a_n\}$ und den zugehörigen Wahrscheinlichkeiten $\{p_1, \ldots, p_n\}$ gegeben. Dann gilt für die erwartete Länge eines kompakten Codes C:

$$H(p_1, \ldots, p_n) \leq E(C) \leq H(p_1, \ldots, p_n) + 1.$$

Somit bildet die Entropie H eine untere Schranke der verlustfreien Kompression, d.h., man benötigt mindestens H Bits, um die kodierte Sequenz zu speichern.

Definition 2.3.6 (Erwartete Kodierungslänge pro Quellsymbol)
Sei das Alphabet A^b mit der Wahrscheinlichkeitsverteilung p^b gegeben, so ist die erwartete Kodierungslänge pro Quellsymbol des Codes C definiert durch $\frac{1}{b}E(C)$.

Nach Satz 2.3.1 erhält man für das Alphabet A^b und die Wahrscheinlichkeitsverteilung p^b eine Kodierung mit erwarteter Länge höchstens $H(p^b) + 1$, aufgrund der Eigenschaften der Entropie also beschränkt durch $bH(p) + 1$. Somit ergibt sich als erwartete Kodierungslänge pro Quellsymbol eine obere Schranke von $H(p) + \frac{1}{b}$, die für $b \to \infty$ gegen $H(p)$ konvergiert.

Definition 2.3.7 (Asymptotisch optimale Codes)
Ein Kodierungsverfahren wird als asymptotisch optimal bezeichnet, falls es für jedes Alphabet A und jede Wahrscheinlichkeitsverteilung p angewandt auf A^b und p^b einen Code C_b liefert, so dass gilt

$$\lim_{b\to\infty} \frac{1}{b} E(C_n) = H(p).$$

Dies bedeutet, dass bei einer beliebigen Quelle mit Alphabet A bei wachsendem b die erwartete Kodierungslänge pro Quellsymbol von A^b gegen die Entropie der Quelle konvergiert.

2.4 Der Shannon-Fano-Elias-Code

Gegeben sind eine Quelle mit Alphabet $A = \{a_1, \ldots, a_n\}$ und eine Wahrscheinlichkeitsverteilung $p = (p_1, \ldots, p_n)$, hierbei müssen die p_i nicht sortiert seien. Nun definiert man für jedes $i = 1, \ldots, n$:

$$\begin{aligned} w_i &:= \lceil \log_2 \frac{1}{p_i} \rceil + 1 \\ L_i &:= \sum_{j=1}^{i-1} p_j \ , \ L_{n+1} = 1 \\ T_i &:= L_i + \frac{p_i}{2} = \sum_{j=1}^{i-1} p_j + \frac{p_i}{2} \end{aligned}$$

Somit wird das Einheitsintervall [0,1) in n Teilintervalle $I_i := [L_i, L_i + 1)$ aufgeteilt, wobei T_i jeweils den Mittelpunkt des Teilintervalls I_i kennzeichnet. Dem

Symbol a_i wird nun das i-te Teilintervall I_i zugeordnet, dessen Breite p_i beträgt. Die SFE-Kodierung C ordnet dem Quellsymbol a_i das Codewort

$$C(a_i) = \lfloor T_i \rfloor_{w_i}$$

zu, wobei $\lfloor x \rfloor_c$ die obersten c Bits der Binärdarstellung von x bezeichnen.

Die SFE-Kodierung C besitzt die folgenden Eigenschaften:

- C liefert einen Präfixcode (s. Abschnitt 3.5).
- $H(p_1, \ldots, p_n) \leq E(C) \leq H(p_1, \ldots, p_n) + 2$
- C ist nach Def. 2.3.7 ein asymptotisch optimales Verfahren.

In der Praxis werden Quellen mit Alphabeten der Form A^b benutzt, das zugehörige Verfahren wird als *arithmetische Kodierung* bezeichnet, die zugrundeliegenden Algorithmen werden in den nächsten zwei Kapiteln vorgestellt.

3 Dekodierung einer reellen Zahl

Bevor man die arithmetische Kodierung entwickelte, galt die Huffman Kodierung als fast optimales Verfahren, da sie der Entropie sehr nahe kommt und sie in Spezialfällen sogar erreicht. Jedoch darf bei der Huffman Kodierung das Quellalphabet A^b nicht zu groß sein, da man die Wahrscheinlichkeiten in der Verteilung sortieren muss bzw. der Baum Größe $\Theta(n)$ besitzt. Die Huffman Kodierung ist eher unflexibel, da man bei einer neuen Wahrscheinlichkeitsverteilung den den Wahrscheinlichkeiten entsprechenden balancierten Binärbaum umstrukturieren müsste, bei der arithmetischen Kodierung wird lediglich das Modell angepasst.
Die arithmetische Kodierung kodiert stets die komplette Sequenz und bildet diese auf das Intervall [0,1) $\subset \mathbb{R}$ ab, es besteht jedoch auch die Möglichkeit, die Sequenz in mehrere Blöcke aufzuteilen.
Da das Intervall [0,1) überabzählbar viele Elemente enthält, kann man jeder Sequenz eine reelle Zahl zuordnen, die nur diese Sequenz beschreibt. Um eine gegebene Sequenz zu komprimieren, muss diese auf eine „kürzere“ Sequenz abgebildet werden, dabei sollte die mittlere Anzahl der benötigten Bits pro Symbol möglichst nah an der Entropie liegen.

3.1 Intervallbildung

Um die SFE-Kodierung auf eine Sequenz anwenden zu können, unterteilt man zunächst das Intervall [0,1) in die den Wahrscheinlichkeiten entsprechenden Teilintervalle $I_1, \ldots, I_n$, wie durch das folgende Beispiel gezeigt wird (entnommen aus [Wit87a, S. 512f].

Beispiel 3.1.1
Sei ein statisches Modell mit dem Alphabet A = {a, e, i, o, u, !} gegeben, dann ergibt die Partionierung des Intervalls [0,1) in Teilintervalle proportional zu den Wahrscheinlichkeiten der Symbole des Alphabets die folgende Tabelle:

Symbol	Wahrscheinlichkeit	Teilintervall
a	0.2	[0.0, 0.2)
e	0.3	[0.2, 0.5)
i	0.1	[0.5, 0.6)
o	0.2	[0.6, 0.8)
u	0.1	[0.8, 0.9)
!	0.1	[0.9, 1.0)

Nun teilt man jedes dieser Teilintervalle I_i wieder proportional zu p_i in n Teilintervalle auf; für das Alphabet A^b muss man also b mal diese Unterteilung vornehmen.

Zur Bestimmung des Codewortes bei gegebener Sequenz $S = s_1 s_2 \dots s_b$ berechnet man zunächst das Teilintervall I_l, in dem das Intervall für S liegt, somit muss man für die weitere Bestimmung nur noch dieses Teilintervall betrachten. Der Index l ergibt sich durch $1 \leq l \leq n$ mit $s_1 = a_l$, die weiteren Intervalle werden analog bestimmt.

Die Grenzen der Teilintervalle sind durch die kumulierten Wahrscheinlichkeiten der Symbole bestimmt, diese werden definiert durch:

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

Im Bezug auf Beispiel 3.1.1 ergibt somit

$K(a_1 = a) = 0.2$	$K(a_2 = e) = 0.5$	$K(a_3 = i) = 0.6$
$K(a_4 = o) = 0.8$	$K(a_5 = u) = 0.9$	$K(a_6 = !) = 1.0$

In der Implementierung werden kumulierte Häufigkeiten benutzt, diese sind in Abschnitt 5.2.1 definiert.

Die untere Grenze des aktuell betrachteten Intervalls wird im Weiteren mit *low*, die obere mit *high* bezeichnet. Initialisiert sind die Grenzen mit 0 und 1, diese Werte werden im Laufe des Verfahrens durch

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

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

angepasst, wobei das gelesene Symbol das k-te des Alphabets ist.

Um das aktuelle Intervall wie oben beschrieben aufzuteilen, ist es außerdem nötig, die Grenzen mit einem Faktor zu multiplizieren, da die den Symbolen zugeordneten Intervalle über das ganze Intervall [0,1) definiert sind, hingegen das aktuelle Intervall ein echtes Teilintervall von [0,1) ist.

Dieser Faktor wird mit *range* bezeichnet, da er sich aus der Größe des aktuellen Intervalls ergibt, dies beeinflusst natürlich auch *high* und *low*. *low* muss außerdem noch zur Berechnung der neuen Grenzen mit einbezogen werden, a_k sei das aktuell gelesene Symbol, dann ergibt sich:

$$\begin{aligned} range &:= & high - low & &:= range \cdot p(a_k) \\ high &:= & low + \textstyle\sum_{i=1}^{k} p(a_i) \cdot range & &:= low + K(a_k) \cdot range \\ low &:= & low + \textstyle\sum_{i=1}^{k-1} p(a_i) \cdot range & &:= low + K(a_{k-1}) \cdot range \end{aligned}$$

3.2 Dekodierung

Eine mit der arithmetischen Kodierung kodierte Sequenz S ist durch ein Intervall bzw. einen beliebigen Wert z aus diesem Intervall gegeben, hieraus soll nun die Originalsequenz S wiederhergestellt werden.

Man überprüft, in welchem Intervall $I := [K(a_{k-1}), K(a_k))$ z liegt und erhält dann eines der Symbole a_k der Originalsequenz.
Danach werden die Grenzen in der oben beschriebenen Methode angepasst, und das nächste Symbol wird bestimmt.

Das Ende der Nachricht muss durch ein spezielles EOF Symbol kodiert sein, da sonst der Dekodierprozess nicht terminieren könnte.

3.3 Beispiel

Um das beschriebene Verfahren zu verdeutlichen, betrachtet man noch einmal Beispiel 3.1.1 und das Intervall $I = [0.23354, 0.2336)$ bzw. $z = 0.23355$; anfangs sei $low = 0$ und $high = 1$.

z liegt offensichtlich im Intervall [0.0, 0.5), somit erhält man als erstes Symbol ein **e** und setzt

$$\begin{aligned} range &= 1 - 0 = 1 \\ high &= 0 + 0.5 \cdot 1 = 0.5 \\ low &= 0 + 0.2 \cdot 1 = 0.2 \end{aligned}$$

Im nächsten Schritt kann man feststellen, dass z zwischen den Grenzen

$$\begin{aligned} range &= 0.5 - 0.2 = 0.3 \\ high &= 0.2 + 0.2 \cdot 0.3 = 0.26 \\ low &= 0.2 + 0.0 \cdot 0.3 = 0.2 \end{aligned}$$

liegt und somit als **a** dekodiert wird. Dann ergibt sich mit

$$\begin{aligned} range &= 0.26 - 0.2 = 0.06 \\ high &= 0.2 + 0.6 \cdot 0.06 = 0.236 \\ low &= 0.2 + 0.5 \cdot 0.06 = 0.23 \end{aligned}$$

als nächstes ein **i** und danach wiederum ein **i** durch

$$\begin{aligned} range &= 0.236 - 0.23 = 0.006 \\ high &= 0.23 + 0.6 \cdot 0.006 = 0.2336 \\ low &= 0.23 + 0.5 \cdot 0.006 = 0.233 \end{aligned}$$

Als letztes Symbol wird ein **!** berechnet, da

$$\begin{aligned} range &= 0.2336 - 0.233 = 0.0006 \\ high &= 0.233 + 1.0 \cdot 0.0006 = 0.2336 \\ low &= 0.233 + 0.9 \cdot 0.0006 = 0.23354 \end{aligned}$$

Als Originalsequenz ergibt sich somit $S = eaii!$, der Vorgang kann anhand von Abbildung 3.1 nachvollzogen werden.

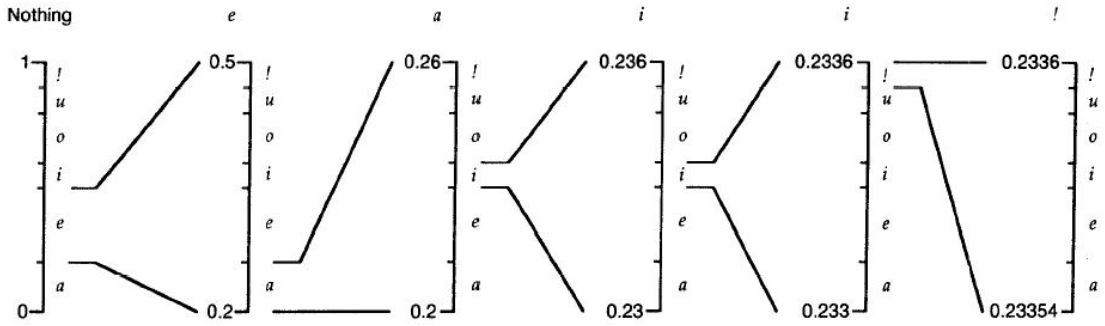

Abbildung 3.1: Beispieldekodierung mit Vergrößerung der Ansicht des aktuellen Bereichs

Dieses Beispiel verdeutlicht u.a. den Bedarf an einer hohen Rechengenauigkeit, jedoch steht diese exakte Arithmetik in der Implementierung nicht zur Verfügung, dieses Problem wird später u.a. durch die Darstellung durch Integer Werte und Skalierung gelöst und in Kapitel 4 genauer beschrieben.

3.4 Algorithmus

Der folgende Algorithmus dekodiert eine Binärzahl z, die ein gegebenes Intervall I eindeutig bestimmt, zu einer Originalsequenz S.

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

1. Setze $low := 0$, $high := 1$, $S = \epsilon$, $t = 0.z$.
2. Führe die fünf folgenden Schritte aus, bis das Ende der Sequenz erreicht ist.
3. Finde k mit $low \leq t < high$.
4. $range := high - low$
5. $high := low + K(a_k) \cdot range$
6. $low := low + K(a_{k-1}) \cdot range$
7. Setze $S = Sa_k$.
8. **Ausgabe** $S \in A^b$

Im obigen Verfahren wurde mit durch Fließkommazahlen begrenzten Intervallen zwischen 0 und 1 gearbeitet, hieraus ergeben sich möglicherweise Zahlen unendlicher Länge. Doch dann reicht die Zahlendarstellung eines Rechners nicht aus, um noch hinreichend genau rechnen zu können, da er mit einer festen Bitbreite arbeitet, d.h. es steht nur eine endliche Genauigkeit zur Verfügung.
Bei der Kodierung einer Sequenz wird dann das berechnete Intervall immer kleiner, *low* und *high* nähern sich immer weiter an und nach einigen Schritten existiert kein Intervall mehr zwischen den Grenzen, dies führt zu Informationsverlust über die kodierte Sequenz, sie kann nicht mehr korrekt dekodiert werden.
Selbst wenn ein Verfahren bekannt wäre, dass mit unendlichen Werten umgehen könnte, so wäre dieses niemals effizient; eine elegante Lösung der Probleme beinhaltet Kapitel 4.

3.5 Eindeutigkeit der Dekodierung

Nach Satz 2.2.1 ist die arithmetische Kodierung eindeutig dekodierbar, falls sie einen Präfixcode bildet, da dieser zur Klasse der eindeutig dekodierbaren Codes gehören.
Der folgende Beweis stammt aus [Say00, S. 90f].

Wie in Abschnitt 2.4 sei

$$C(a_i) = \lfloor T(a_i) \rfloor_{w(a_i)}$$

der auf $w(a_i)$ Stellen gekürzte Binärcode für a_i mit

$$w(a_i) := \lceil \log_2 \frac{1}{p\,(a_i)} \rceil + 1 \tag{3.1}$$

$$T(a_i) := \sum_{j=1}^{i-1} p\,(a_j) + \frac{p\,(a_i)}{2} = K(a_{i-1}) + \frac{p\,(a_i)}{2}. \tag{3.2}$$

Für diesen gilt

$$K(a_{i-1}) \leq C(a_i) < K(a_i).$$

Gegeben sei eine Zahl $y = [y_1 y_2 \dots y_t]$ aus dem Intervall [0,1) mit binärer Repräsentation der Länge t. Jede weitere Zahl z mit $[y_1 y_2 \dots y_t]$ als Präfix muss somit im Intervall $[y, y + \frac{1}{2^t})$ liegen, da durch y die linken Bits in der Binärdarstellung $(2^0 2^{-1} \dots 2^{t-1})$ von z bestimmt sind und das nächste Bit maximal 2^{-t} ist.

Falls $a_i \neq a_j$ zwei unterschiedliche Symbole sind, so liegen die Werte $C(a_i)$ und $C(a_j)$ in zwei disjunkten Intervallen

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

Falls man zeigen kann, dass für jedes Symbol a_i das Intervall

$$[C(a_i),\ C(a_i) + \frac{1}{2^{w(a_i)}})$$

komplett im Intervall $[K(a_{i-1}),\ K(a_i))$ enthalten ist, so kann der Code $C(a_i)$ des Symbols a_i nicht Präfix eines Codes eines anderen Symbols a_j sein.

Da gilt $C(a_i) \geq K(a_{i-1})$, ist die untere Grenze bereits bewiesen und es bleibt zu zeigen, dass

$$K(a_i) - C(a_i) > \frac{1}{2^{w(a_i)}}\,.$$

Es gilt nach (3.2)

$$T(a_i) - K(a_{i-1}) = \frac{p\,(a_i)}{2}\,,$$

somit folgt für die obere Grenze

$$K(a_i) - T(a_i) = \frac{p\,(a_i)}{2}\,,$$

da $T(a_i)$ den Mittelpunkt des Intervalls $[K(a_{i-1}),\ K(a_i))$ bildet.

Außerdem gilt nach Definition von $w(a_i)$

$$\frac{1}{2^{w(a_i)}} \leq \frac{p\,(a_i)}{2}\,.$$

Somit folgt

$$\begin{aligned} K(a_i) - C(a_i) &> K(a_i) - T(a_i) \\ &= \frac{p\,(a_i)}{2} \\ &\geq \frac{p\,(a_i)}{2}\,, \end{aligned}$$

also ist der vorliegende Code präfixfrei.
Insbesondere lässt sich durch Kürzen von $T(a_i)$ auf $w(a_i)$ Bits ein eindeutig dekodierbarer Code erzeugen.

4 Dekodierung einer Bitsequenz

Um die in Beispiel 3.3 geforderte Rechengenauigkeit erreichen zu können, muss die exakte Arithmetik durch endliche Arithmetik ersetzt werden.
In der Darstellung im Rechner wird von Fließkommazahlen zu Integerwerten übergegangen, da Fließkommazahlen unterschiedliche Darstellungen auf verschiedenen Rechnern haben können.
Integer Werte können genauso wie Fließkommazahlen das Intervall [0,1) repräsentieren, z.Bsp. durch [0, 65535), also durch eine Integer Darstellung der Länge 16.
Jedoch behebt dies noch nicht die in Abschnitt 3.4 beschriebene Annäherung der Intervallgrenzen, dieses Problem wird durch die Einführung von Skalierungsfunktionen in Abschnitt 4.2 gelöst.

4.1 Algorithmus

Der Dekodierer bekommt nun statt einer Fließkommazahl aus einem Intervall eine Bitsequenz übergeben, die ebenfalls eindeutig das Intervall bestimmt. Nun muss im ersten Schritt das Symbol bestimmt und dann die Grenzen aktualisiert werden.
Hierzu betrachtet man nicht sofort die gesamte Bitsequenz, sondern immer nur einen Teil bestimmter Länge k, dieser wird im Folgenden mit *buffer* bezeichnet (s. Abschnitt 4.3).
Die Dekodierung wird dann durch den folgenden Algorithmus berechnet, die verwendete Skalierung wird in Abschnitt 4.2 erklärt.

Eingabe $t \in \{0,1\}^k$

1. Setze *low* und *high* entsprechend des Ausgangsintervalls, $S = \epsilon$, $buffer = 0.t$.

2. Führe die drei folgenden Schritte aus, bis das Ende der Sequenz erreicht ist.

3. Finde j mit $low \leq buffer < high$.

4. Skaliere das gefundene Intervall zwischen *low* und *high*, bis keine weiteren Skalierungen mehr möglich sind, hierbei wird jeweils der *buffer* aktualisiert.

5. Setze $S = Sa_j$.

6. **Ausgabe** $S \in A^b$

Der Beweis, dass ein Symbol korrekt durch diesen Algorithmus mit Skalierung des gefundenen Intervalls dekodiert wird, findet sich in [Blo03, S.55f.].

Die Skalierung bewirkt u.a., dass vor jedem Schleifendurchlauf in 2. gilt, dass $high - low > 1/4$ (ausführlicher erläutert in Abschnitt 4.2.2), also dass das aktuelle Intervall mehr als ein Viertel des Ausgangsintervalls einnimmt. Dies ist eine wichtige Voraussetzung, da sonst möglicherweise Intervallgrenzen zusammenfallen und somit Symbole nicht richtig dekodiert werden könnten (s. Satz 4.3.1).

4.2 Skalierungsfunktionen

Um die in Abschnitt 3.4 beschriebene Annäherung der Grenzen *high* und *low* zu verhindern, bedient sich der Kodierer verschiedener Skalierungen des Intervalls, diese müssen natürlich auch vom Dekodierer durchgeführt werden.
De- bzw. Kodierung mittels Skalierung löst die Probleme der endlichen Arithmetik und gewährleistet trotzdem noch die Möglichkeit, dass schon Teile des Codes dekodiert werden können, obwohl die gesamte Kodierung noch nicht abgeschlossen ist, außerdem wird ein Über- bzw. Unterlauf verhindert (s. Kapitel 5).

4.2.1 E1 / E2 - Skalierung

Betrachtet man das Ausgangsintervall [0,1) und teilt es in zwei Teile [0, 0.5) und [0.5, 1), so haben alle Zahlen im ersten Intervall in ihrer binären Repräsentation eine führende 0, die Zahlen im zweiten Intervall beginnen mit einer 1.
Da bei der Intervallbildung in Abschnitt 3.1 jedes neue Intervall ein Teilbereich des vorherigen Intervalls ist, wird dieses Intervall in der restlichen Berechnung nicht mehr verlassen, sondern nur noch weiter eingeschränkt.
Liegt das berechnete Intervall irgendwann in einem der beiden Teile des Ausgangsintervalls, so kann man das jeweilige Bit ausgeben, dass diesen Bereich identifiziert und dann das Intervall skalieren, d.h. auf das Ursprungsintervall vergrößern. Dabei geht die Information über das höchstwertige Bit verloren, aber dies wurde ja ausgegeben bzw. gespeichert, so dass der Dekodierer trotzdem alle nötigen Informationen bekommt.

Die Skalierung des Intervalls [0, 0.5) auf das Ursprungsintervall wird als $E1$-Skalierung bezeichnet und ist formal definiert durch:

$$\begin{aligned} E1 : [0, 0.5) &\to [0, 1) \\ x &\mapsto 2x \end{aligned}$$

Damit diese Skalierung durchgeführt wird, müssen *low* und *high* kleiner als die Hälfte (im Folgenden mit *half* bezeichnet) des Ursprungsintervalls sein, also beginnen beide Binärdarstellungen mit 0. Da $high < half$, kann es nicht vorkommen, dass das skalierte Intervall größer ist als [0,1).

Analog für *low* und *high* größer bzw. gleich *half*, existiert die $E2$-Skalierung. Damit hier die Grenzen richtig angepasst werden können, muss von *low* und *high* jeweils *half* subtrahiert und dann wie bei $E1$ mit 2 multipliziert werden.

$$\begin{aligned} E2 : [0.5, 1) &\to [0, 1) \\ x &\mapsto 2\,(x - 0.5) \end{aligned}$$

4.2.2 E3 - Skalierung

Befindet sich das betrachtete Intervall jedoch im mittleren Bereich des Ursprungsintervalls, also im Bereich [0.25, 0.75)(*firstQuarter, ThirdQuarter*), so kann es vorkommen, dass die Grenzen gegen den Mittelpunkt, also *half*, konvergieren. Um dies zu verhindern, wird eine weitere Skalierung eingeführt, die $E3$-Skalierung.

$$\begin{aligned} E3 : [0.25, 0.75) &\to [0, 1) \\ x &\mapsto 2\,(x - 0.25) \end{aligned}$$

Im Gegensatz zu $E1$ und $E2$ kann noch keine Information ausgegeben werden, da kein führendes Bit feststeht. Falls jedoch auf eine $E3$-Skalierung eine $E1$-Skalierung oder eine $E2$-Skalierung folgt, so steht fest, in welcher Hälfte sich das Intervall befindet.
$E3$ bildet das Teilintervall [0.25, 0.5) auf [0, 0.5) und das Teilintervall [0.5, 0.75) auf [0.5, 1) ab.
Folgt also eine $E1$-Skalierung auf $E3$, so wird [0.25, 0.5) auf [0,1) abgebildet, dies entspricht einer $E1$- gefolgt von einer $E2$-Skalierung. Es wird 01 ausgegeben, da dies die führenden Bits in [0.25, 0.5) sind.
Folgt $E2$ auf $E3$, so wird umgekehrt [0.5, 0.75) auf das Ursprungsintervall skaliert, was einer $E2$- gefolgt von einer $E1$-Skalierung entspricht, hierbei ist die Ausgabe 10.

Allgemeiner formulieren dies die beiden folgenden Formeln, wobei für zwei Funktionen f und g $g \circ f$ die Hintereinanderausführung von f und g bezeichne.

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

Falls mehrere $E3$-Skalierungen hintereinander auftreten, so werden laut den Formeln n inverse Bits abgespeichert, also im Fall einer folgenden $E1$-Skalierung $0(1)^n$, bei $E2$ ergibt sich $1(0)^n$.

Falls keine weitere Skalierung mehr möglich ist, wird das nächste Symbol berechnet, vor der Berechnung gilt immer $high - low > 1/4$, da

$$[low, high) \nsubseteq [0, 0.5)$$
$$[low, high) \nsubseteq [0.5, 1)$$
$$[low, high) \nsubseteq [0.25, 0.75)\,.$$

4.2.3 Code der Skalierung

Im Folgenden wird die Realisierung der oben definierten Funktionen im Dekodierer beschrieben, die Grenzen werden verändert, die Ausgabe der Bits entfällt natürlich.
Zusätzlich zu *low* und *high* muss jeweils noch der *buffer* skaliert werden, damit dieser auch im richtigen Verhältnis zu den Grenzen steht. Diese Aktualisierung verläuft analog, das neue Bit wird aus der kodierten Sequenz ausgelesen.
Die Variablen *firstQuarter*, *half* und *ThirdQuarter* sind wie zuvor definiert.

```
for (;;) { // Mapping
    if (high < half) {
    // do nothing
    }
    else if (low >= half) { //E2
      buffer = buffer - half;
      low = low - half;
      high = high - half;
    }
    else if (low >= firstQtr && high < thirdQtr) { //E3
      buffer = buffer - firstQtr;
      low = low - firstQtr;
      high = high - firstQtr;
    }
    else {
      break;
```

```
    }
    low = 2 * low; //E1
    high = 2 * high + 1;
    try {
      buffer = Decoder.actualiseBuffer(2 * buffer);
    } // if EOF is reached, decode buffer until EOFSymbol is
       decoded
    catch (EOFException e) {
      buffer = 2 * buffer;
    }
}
```

Da alle drei Skalierungen die Multiplikation mit dem Faktor 2 beinhalten, wurde diese aus den if Anweisungen herausgenommen.
Auffällig zu vorher ist, dass *high* außerdem noch um 1 erhöht wird. Da *high* die obere offene Grenze definiert, werden durch die Addition noch mögliche Nachkommastellen berücksichtigt, die durch die ganzzahlige Speicherung sonst verloren gehen würden.

4.3 Bufferlänge

Wie schon in Abschnitt 4.1 erwähnt, wird nicht die komplette Eingabe $t = 0.t_1 \dots t_m$ der Länge m auf einmal betrachtet, sondern immer nur ein bestimmter Teil, nämlich k Bits, im Folgenden dargestellt durch $\bar{t} = 0.t_1 t_2 \dots t_k$.
Somit erhält man nur eine Approximation der Eingabe, da die restlichen Bits $t_{k+1}, \dots, t_m$ abgeschnitten wurden.
Trotzdem muss sichergestellt sein, dass auch diese Approximation das richtige Symbol identifiziert, zu dem die Eingabe dekodiert werden soll.
Dies hängt von der Länge des *buffers* ab, k muss so gewählt werden, dass $\bar{t}$ im selben Intervall liegt, in dem auch t liegen würde.
Es gilt nach Definition von $\bar{t}$

$$|t - \bar{t}| < 2^{-k}\,.$$

Um nun die entsprechende Anzahl k der Bits zu bestimmen, betrachtet man den folgenden Satz aus [Blo03, S.53].

Satz 4.3.1
Es sei $p = \{p_1, \dots, p_n\}$ eine Wahrscheinlichkeitsverteilung mit

$$p_i > 2^{-k+2} \quad \forall i\,.$$

Werden dann alle Zahlen vom Kodierer mit Genauigkeit von mindestens k Bits dargestellt, so berechnet der Kodierer eine injektive Abbildung von A^b nach {0,1}*.

Beweis: Der Kodierer berechnet die Grenzen der Intervalle ebenfalls über die oben beschriebene Skalierung, somit gilt, bevor das nächste Symbol kodiert wird, $high - low > 1/4$. Die neuen Grenzen im nächsten Schritt werden berechnet und es gilt:

$$high - low = (oldHigh - oldLow) \cdot p_j \,,$$

wobei p_j die Wahrscheinlichkeit des aktuellen Symbols ist.
Diese Gleichung kann abgeschätzt werden durch:

$$high - low = (oldHigh - oldLow)p_j > \frac{1}{4}2^{-k+2} = 2^{-k} \,.$$

Damit unterscheiden sich *low* und *high* in den ersten k Bits und die Abbildung ist injektiv, somit kann keinen zwei Symbolen das gleiche Intervall zugeordnet werden. □

Nach Abschnitt 4.2.2 gilt immer $high - low > 1/4$. Betrachtet man ein solches Intervall, so wird dieses in Teilintervalle proportional zu den Wahrscheinlichkeiten aufgeteilt (s. Abschnitt 3.1).
Das Teilintervall geringster Breite gehört zu der kleinsten Wahrscheinlichkeit, für die nach Satz 4.3.1 $p_i > 2^{-k+2}$ gilt. Selbst falls das betrachtete Intervall nur minimal größer als $\frac{1}{4}$ ist, so beträgt die Länge l des kleinsten Teilintervalls immer noch

$$l > \frac{1}{4}2^{-k+2} = 2^{-k} \,.$$

Da der maximale Abstand

$$|t - \bar{t}| < 2^{-k}$$

beträgt, liegt $\bar{t}$ im korrekten Teilintervall.

Der Dekodierer muss mindestens k Bits der Eingabe betrachten, denn falls das Intervall mit weniger als k Bits dargestellt würde, könnte möglicherweise $|t - \bar{t}| > 2^{-k}$ sein, t und $\bar{t}$ würden dann in unterschiedlichen Teilintervallen liegen und somit würde die Eingabe zum falschen Symbol dekodiert.

Wählt man also k so, dass die Bedingung

$$p_i > 2^{-k+2} \quad \forall i$$

erfüllt ist, so werden die korrekten Symbole bestimmt.

Die Realisierung dieser Überlegungen findet sich in Abschnitt 5.4.1.

4.4 Beispiel

Wiederum wird Beispiel 3.1.1 benutzt, um die Wirkung der oben beschriebenen Skalierungsfunktionen zu verdeutlichen, zum direkten Vergleich wurden auch hier

keine Integerwerte zur Darstellung der Grenzen benutzt.
Hierbei wird der *buffer* durch $t_1 t_2 \dots t_k$ bzw. $0.t_1 t_2 \dots t_k$ dargestellt; da $p(i) = 0.1$ die kleinste Wahrscheinlichkeit ist, gilt in diesem Fall $k = 6$, da $0.1 > 2^{-6+2} = 2^{-4} = 0.0625$.
Die Eingabesequenz sei 001110111100101.

	$t_1 t_2 \dots t_k$	$0.t_1 t_2 \dots t_k$	Symbol	*low*	*high*	Skalierung
i=0	001110	0.21875	-	0	1	-
i=1	001110	0.21875	e	0.2	0.5	-
i=2	001110	0.21875	a	0.2	0.26	E1
i=3	011101	0.453125	-	0.4	0.52	E3
i=4	011011	-	-	0.3	0.54	E3
i=5	010111	-	-	0.1	0.58	-
i=6	010111	0.359375	i	0.34	0.388	E1
i=7	101111	-	-	0.68	0.766	E2
i=8	011110	-	-	0.36	0.552	E3
i=9	011100	-	-	0.22	0.604	-
i=10	011100	0.4375	i	0.412	0.4504	E1
i=11	111001	-	-	0.824	0.9008	E2
i=12	110010	-	-	0.648	0.8016	E2
i=13	100101	-	-	0.296	0.6032	E3
i=14	10101	-	-	0.092	0.7064	-
i=15	10101	0.65625	!	0.64496	0.7064	-

Im Schritt i=14 kann kein weiteres Bit in den Buffer eingelesen werden, da das Ende der Sequenz erreicht ist, jedoch muss noch das letzte Symbol berechnet werden.
Die Sequenz wird wie zuvor zu eaii! dekodiert.

5 Implementierung

Die vorliegende Implementierung basiert auf dem Quellcode von [Tod03], lediglich die Klassen Decoder, ArithmeticDecode und Decode1 wurden selber erstellt und hinzugefügt.
In den Klassen IOOperations und ProbabilityDistribution wurden notwendige Erweiterungen vorgenommen, um die Bufferlänge bestimmen und den *buffer* einlesen zu können.

5.1 Anforderungen

Das zentrale Anliegen der Implementierung ist eine objektorientierte Programmstruktur, die zu Forschungszwecken in der AG Codes und Kryptographie weiterentwickelt werden kann. Dies bedeutet, dass Softwaremerkmale wie Verständlichkeit, Erweiterbarkeit und Wartbarkeit eine große Rolle spielen, außerdem sollte der Quellcode gut lesbar und dokumentiert sein.
Wichtiger Punkte sind außerdem mögliche Algorithmenwechsel zur Laufzeit bzw. eine vollständige Ausnahmebehandlung *(Exception Handling)*.
Das Verständnis soll durch eine automatisierte Methodenreferenz *(JavaDoc)* sowie durch grundlegende UML-Diagramme unterstützt werden.

5.2 Voraussetzungen

5.2.1 Kumulierte Häufigkeiten

Da zur Berechnung eines Symbols nach Kapitel 4 Integerwerte verwendet werden sollen, muss man die kumulierten Wahrscheinlichkeiten $K(a_k)$ aus Abschnitt 3.1 durch die entsprechenden kumulierten Häufigkeiten ersetzen.
Diese werden in der Implementierung in einem Array *cumFreq* in der Klasse ProbabilityDistribution gespeichert. Die Größe des Array entspricht der Anzahl der Symbole des Alphabets + 2, in einem dieser zusätzlichen Einträge ist die Häufigkeit aller Symbole gespeichert, der Normalisierungsfaktor, in dem anderen die Häufigkeit des EOF-Symbols.
Der Normalisierungsfaktor ist der 0-te Eintrag des Arrays, an der ersten Stelle

steht der Faktor ohne die Häufigkeit für das EOFSymbol und an der letzten Stelle steht der Wert 0, da die Häufigkeiten in absteigender Reihenfolge abgespeichert sind.
Um auf das zugehörige Symbol zugreifen zu können, existiert in der Implementierung eine Funktion, die abhängig vom Index das gesuchte Symbol liefert.

Die kumulierten Wahrscheinlichkeiten $K(a_k)$ kann man anhand des Normalisierungsfaktors und der kumulierten Häufigkeit des Symbols a_k bestimmen:

$$K(a_k) = \frac{cumFreq[k]}{cumFreq[0]} .$$

Beispiel 5.2.1
Betrachtet man die Sequenz S = aiioueaii!, so ergeben sich die folgenden Werte:

Symbol	Index	Häufigkeit	Kumulierte H.	$K(a_k)$
	0		10	1.0
a	1	2	8	0.8
e	2	1	7	0.7
i	3	4	3	0.3
o	4	1	2	0.2
u	5	1	1	0.1
!	6	1	0	0.0

Diese Veränderung betrifft natürlich auch die Funktionen zur Berechnung der Grenzen *low* und *high*, diese werden wie folgt angepasst:

$$high = low + (range * cumFreq[k])/cumFreq[0] - 1;$$
$$low = low + (range * cumFreq[k+1])/cumFreq[0];$$

Da bei der Berechnung ein halboffenes Intervall verwendet wird, ergibt sich die Subtraktion um 1 bei *high*, ebenso muss *range* verändert werden.

$$range = high - low + 1 .$$

5.2.2 Unterlauf

Ein Unterlauf könnte entstehen, falls bei der Intervallbildung (s. Abschnitt 3.1) zwei Symbolen das gleiche Intervall zugeteilt würde, diese wären dann nicht disjunkt und die Dekodierung fehlerhaft.

Aufgrund der Skalierungsfunktionen in Abschnitt 4.2 gilt für die Grenzen der Intervalle eine der folgenden Aussagen:

$$low < FirstQuarter < half \leq high$$
$$low < half < ThirdQuarter \leq high\,.$$

Somit umfasst das Intervall nach 4.2.2 zu jedem Zeitpunkt mindestens ein Viertel des maximalen Bereichs, d.h. es bleibt zu zeigen, dass die kumulierten Häufigkeiten in diesem Viertel eindeutig bestimmbar sind, dann kann es zu keinem Unterlauf kommen [Wit87a, S.533].

Stellt man die Werte des Intervalls mit einer Präzision von c Bits dar, so ergibt sich das Intervall $[0,\ 2^c - 1)$, der Maximalwert $2^c - 1$ des Intervalls wird als *topValue* bezeichnet.
Die kumulierten Häufigkeiten werden mit f Bits dargestellt, also im Intervall $[0,\ 2^f - 1)$, die maximale Häufigkeit beträgt $maxFreq = 2^f - 1$.

Wenn die folgende Bedingung erfüllt ist, so kann ein Unterlauf verhindert werden:

$$maxFreq \leq \frac{topValue + 1}{4} + 1.$$

Somit ist sichergestellt, dass *maxFreq* kleiner als ein Viertel des Intervalls ist, die Häufigkeiten sind also eindeutig bestimmbar.

5.2.3 Überlauf

Bei der Vergrößerung einer Zahl kann ein Überlauf eintreten, falls die zur Verfügung stehende Darstellungsmöglichkeit des Rechners nicht ausreicht.
Also muss sichergestellt werden, dass kein Schritt in der Dekodierung einen Überlauf erzeugen könnte. Das Produkt

$$range * cumFreq[k]$$

kann als maximalen Wert

$$range * maxFreq$$

annehmen, hierbei gilt

$$range \leq topValue + 1\,.$$

Daher folgt

$$2^c \cdot (2^f - 1) = 2^{c+f} - 2^c \leq 2^{c+f} - 1$$

die benötigte Genauigkeit beträgt also $c + f$ Bits.

Somit kann es zu keinem Über- bzw. Unterlauf kommen, solange die folgenden Gleichungen erfüllt sind:

$$\begin{aligned} f &\leq c + 2 \\ c + f &\leq p\,, \end{aligned}$$

hierbei definiert p die Anzahl der Bits für den maximalen Wert des betrachteten Produkts.
Somit muss die Präzision des Intervalls um 2 größer sein als die Präzision der kumulierten Häufigkeiten.

$p = 31$ entspricht der Integerdarstellung in Java, werden Long Werte benutzt, erhöht sich p auf 63, dieser Wert wurde in der Implementierung verwendet.

5.3 UML-Diagramme

5.3.1 Klassendiagramm

Das folgende Diagramm soll den Zusammenhang der zur Berechnung benötigten Klassen graphisch unterstützen, diese werden im Folgenden kurz erläutert. In Anhang 6.2 stehen genaue Beschreibungen der Methoden der einzelnen Klassen zur Verfügung.

- **Alphabet**
  Diese Klasse repräsentiert ein Alphabet mit den benötigten Informationen wie zugehörige Symbole, deren Anzahl und die Repräsentation durch einen Index, der sowohl zur De- wie auch zu Kodierung benötigt wird.

- **Decoder**
  Im Decoder wird der Dekodierprozess gestartet, er enthält Methoden zur Bestimmung und Aktualisierung des zu betrachtenden Buffers, der dann an das ArithmeticDecode Objekt übergeben wird, sowie alle ProbabilityDistribution Objekte. Abhängig vom aktuellen Kontext wird dann die benötigte Häufigkeitsverteilung in das Model Objekt gesetzt.

- **Decode1**
  Decode1 ist eine Implementierung des Interfaces ArithmeticDecode, sie stellt eine mögliche Berechnungsart für die Symbole dar und kann durch andere ArithmeticDecode Objekte ausgetauscht werden.

- **IOOperations** In dieser Klasse sind alle zur Ein- bzw. Ausgabe benötigten Methoden gespeichert, außerdem steuert sie das Einlesen der Häufigkeitsverteilungen.

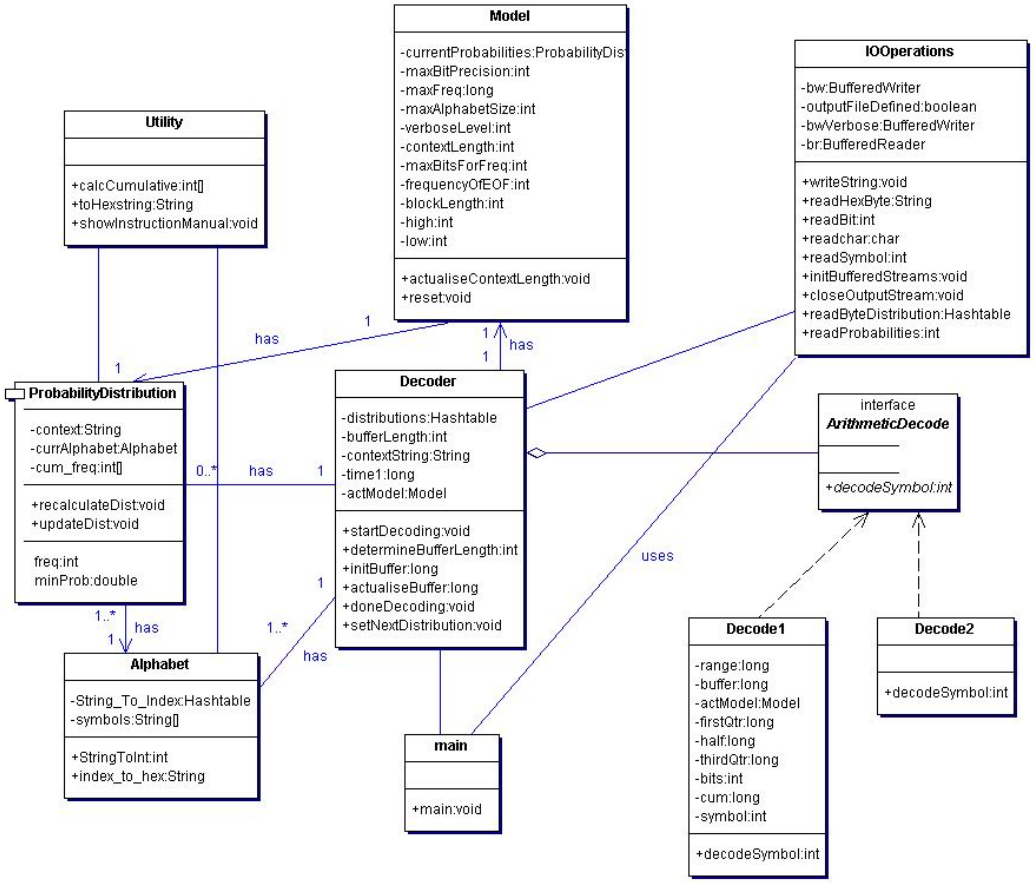

Abbildung 5.1: Klassendiagramm des arithmetischen Dekodierers

- **Model**
  Im Model Objekt sind alle grundlegenden Informationen für den Dekodierprozess zusammengefasst, diese werden teilweise im Ablauf der Dekodierung durch das ArithmeticDecode bzw. das Decoder Objekt verändert.

- **ProbabilityDistribution**
  Jede Häufigkeitsverteilung besteht aus mindestens einem ProbabilityDistribution Objekt, dass das zugeordnete Alphabet, den jeweiligen Kontext sowie die zugehörigen Häufigkeiten und deren Aktualisierungsfunktionen gespeichert hat.

- **Utility**
  Diese Klasse stellt mehrere grundliegende Funktionen zur Verfügung, u.a. die Methode zur Berechnung der kumulierten Häufigkeiten.

5.3.2 Sequenzdiagramm

Der Ablauf der Dekodierung eines Symbols ist in dem Sequenzdiagramm in Abbildung 5.2 näher erläutert.
Am Anfang werden durch die Klasse IOOperations die benötigten Wahrscheinlichkeitsverteilungen eingelesen und in einem statischen Model gespeichert, dieses wird dann an den Decoder über 'startDecoding' übergeben. Der Decoder initialisiert den Ein - und Ausgabestrom sowie den zu betrachtenden Buffer.
Da beim Starten des Dekodierprozesses noch kein Kontext existiert, wird dem Model als erstes eine kontextunabhängige Häufigkeitsverteilung zugewiesen.
Über die im Model gespeicherten Daten werden dann Zeichen aus dem Buffer ausgelesen und der Index eines Symbols berechnet, hierbei wird ggf. der Buffer in einem Berechnungsschritt mehrmals aktualisiert.
Dann werden die alten Grenzen im Model durch die neu berechneten ersetzt, der Index wird in das entsprechende Symbol umgewandelt und die Häufigkeitsverteilung des aktuellen Kontextes gesetzt, um das nächste Symbol zu dekodieren.
Wenn der Dekodierer das kodierte EOF-Symbol erreicht, wird die Berechnung beendet und ggf. eine Programmausgabe erzeugt.

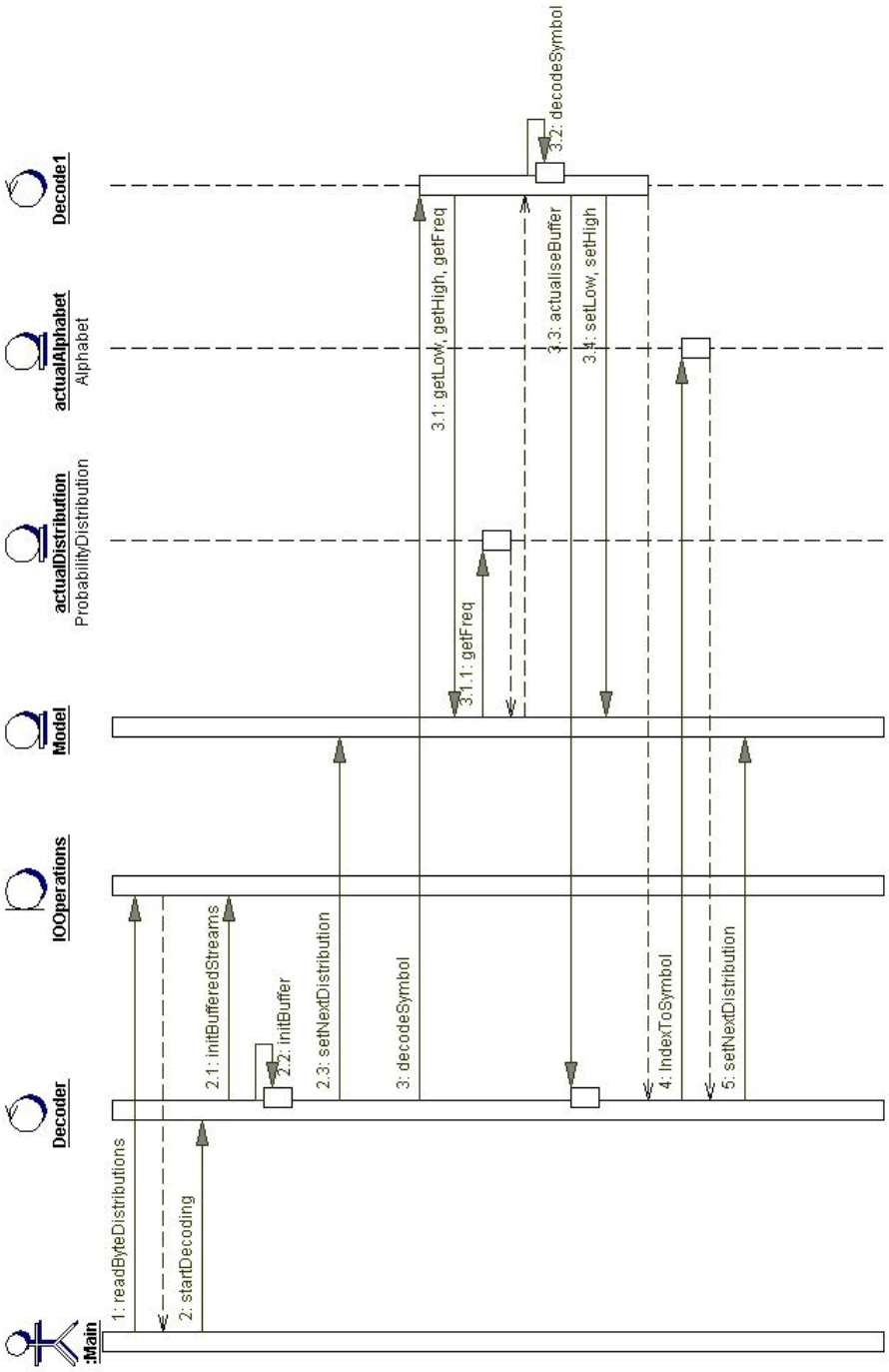

Abbildung 5.2: Sequenzdiagramm der Dekodierung eines Symbols

5.4 Technische Details

In der vorliegenden Realisierung besteht die Möglichkeit, unterschiedliche Alphabete bzw. verschiedene kontextabhängige Wahrscheinlichkeiten einzubinden, diese werden zur Zeit aus einem statischen Order-1-Modell eingelesen. Die Symbole des Alphabets bestehen dabei aus hexadezimalen Werten zwischen 00 (0) und FF (255).
Sämtliche Ausgaben des Programms sind in englischer Sprache gehalten.
Es werden zwei Formate für Häufigkeitsverteilungen unterstützt, eine genaue Beschreibung findet sich in [Tod03, Kapitel 6.2].

5.4.1 Buffer

Bei Programmstart wird eine Häufigkeitsverteilung für jeden Kontext eingelesen und dabei jeweils die kleinste Wahrscheinlichkeit berechnet.
So muss zur Bestimmung der Bufferlänge k lediglich die minimale Wahrscheinlichkeit aller Verteilungen bestimmt werden, dies ist durch die folgende Funktion realisiert.
Hierbei repräsentiert *distributions* eine Hashtable, die als Schlüssel den jeweiligen Kontext und als Wert ein *ProbabilityDistribution* Objekt besitzt, das die Häufigkeiten enthält.
k wird schließlich durch Abschätzung der Gleichung in Satz 4.3.1

$$-log_2 p_i + 2 < bufferLength$$
$$\lceil -log_2 p_i + 2 \rceil = bufferLength$$

durch folgende Funktion berechnet.

```
private int determineBufferLength() {
  double minProb = 1;
  for (Enumeration keys = distributions.keys(); keys.
      hasMoreElements();){
      String key = (String) keys.nextElement();
      // get context
      tmpDist = (ProbabilityDistribution)distributions.get(key);
      //get frequencies
      minProb = Math.min(minProb, tmpDist.getMinProb());
      // get minimal probability of all distributions
  }
  bufferLength = (int) Math.ceil(2 - log2(minProb));
  // minProb > 2^(-bufferLength + 2)
  return bufferLength;
}
```

In der Implementierung existieren zwei Funktionen, die den *buffer* initialisieren bzw. je nach Skalierung aktualisieren, der *buffer* wird durch eine Long Zahl [1] dargestellt.

```
private long initBuffer(int bufferLength) throws EOFException {
  buffer = 0;
  for (int i = bufferLength - 1; i >= 0; i--) {
    int tmp = ioOp.readBit();
    ioOp.verboseOutput( ("Actual Bit: " + tmp), 2);
    buffer = buffer + (long) (Math.pow( (double) 2, (double) (i
        )) * tmp);
  }
  return buffer;
}
```

Um den *buffer* zu aktualisieren, wird das führende der k Bits entfernt und das nächste Bit aus der kodierten Sequenz angehängt, man shiftet also alle Bits um eine Stelle nach links.

```
public static long actualiseBuffer(long inBuffer) throws
EOFException {
  int tmp = ioOp.readBit();
  ioOp.verboseOutput( ("Actual Bit: " + tmp ), 2);
  buffer = inBuffer + tmp;
  statistic[0]++;
  return buffer;
}
```

Erreicht man das Ende der Eingabe, so ist die Berechnung der zu dekodierenden Symbole i.a. noch nicht beendet, dies geschieht erst, wenn das EOFSymbol dekodiert wurde. Man muss auf den alten *buffer* noch zugreifen können, daher wird die EOFException nicht hier abgefangen, sondern in der Skalierung (s. Abschnitt 4.2.3.

5.4.2 Programmparameter

Die Implementierung wird durch einige Parameter gesteuert, die beim Programmstart übergeben werden müssen, bei falscher Eingabe wird eine Übersicht der möglichen Parameter und Optionen angezeigt.

Diese können beliebig mit '-' oder '/' bzw. ohne Vorzeichen beginnen, außerdem spielt die Groß - und Kleinschreibung keine Rolle.

Die Optionen der jeweiligen Unterprogramme können in beliebiger Reihenfolge notiert sein, jedoch müssen die zugehörigen Werte durch ein Leerzeichen getrennt folgen.

[1] Repräsentation eines Integers durch 64 Bits

ac -e [-Optionen]	Hiermit wird, durch [-Optionen] näher definiert, der Kodierprozess gestartet.
ac -d [-Optionen]	Hiermit wird, durch [-Optionen] näher definiert, der Dekodierprozess gestartet.
ac -g	Hiermit wird die graphische Benutzeroberfläche gestartet, falls Optionen angegeben wurden, werden diese ignoriert.

Als mögliche **Optionen** stehen die folgenden zur Auswahl:

Optionen	**Beschreibung**
-a n	Bestimmt die Anzahl n der Symbole im Alphabet, falls die Daten nicht aus einer Eingabedatei ausgelesen werden, der Standardwert beträgt 256.
-b n	Bestimmt die Blocklänge n, standardmäßig wird keine Blocklänge genutzt.
-c n	Bestimmt die verwendete Bit Präzision n für die kumulierten Häufigkeiten, diese muss um 2 kleiner sein als die Präzision der Code-Werte, als Standard ist 14 definiert.
-f Datei	Gibt die Datei an, die die zur Berechnung benötigten Häufigkeiten enthält, hierbei sind die Formate .frq und .xml möglich.
-i Datei	Gibt die Eingabedatei an, die de- bzw. kodiert werden soll.
-o Datei	Gibt die Ausgabedatei an, in die die De- bzw. Kodierung der Eingabe geschrieben werden soll.
-p n	Bestimmt die verwendete Bit Präzision n für die Code-Werte, der Standardwert beträgt 16, der maximale Wert 32.
-v Datei n	Bestimmt das Verbose-Level n und die Datei, in die die je nach Level detaillierten Informationen geschrieben werden. Der Standardwert beträgt 0, somit werden nur die wichtigsten Informationen notiert.
-v n	Analog zu '-v Datei n', hier Ausgabe in Standardausgabe.

6 Anhang

6.1 CD

Die beiliegende CD enthält den Quellcode der arithmetischen Dekodierung aufbauend auf dem Quellcode der arithmetischen Codierung, der von C. Todtenbier erstellt wurde [Tod03]. Aus diesem Quellcode wurde ein JavaDoc System im HTML Format generiert, dass die Klassen mit ihren Funktionen und Konstruktoren näher beschreibt. Zusätzlich zu diesem System sind einige Beispieldateien enthalten, um die Funktionsweise des Programms zu verdeutlichen. Abschließend ist dieses Dokument im PDF Format zu finden.

6.2 JavaDoc

Im folgenden werden Auszüge aus dem JavaDoc System aufgeführt, um die Klassen der eigentlichen Berechnung ausführlicher darzustellen, alle weiteren Beschreibungen der grafischen Benutzeroberfläche, etc. sind auf der beiliegenden CD zu finden.

6.2.1 Interface **ArithmeticDecode**

Title: Studienarbeit - Arithmetischer Kodierer / Dekodierer

Description: The interface for the objects which decode the input to the symbols dependent on the values of the model object.

Copyright: Copyright (c) 2003

Organisation: Universität Paderborn

Declaration

public interface ArithmeticDecode

Methods

- *decodeSymbol*
  `public int` **decodeSymbol**`( )`

6.2.2 Class **Alphabet**

Title: Studienarbeit - Arithmetischer Kodierer / Dekodierer

Description: This class holds a single instance of an Alphabet. In this object all different symbols are saved, and can be transformed to an index. Therefore it is used an Hashtable to fast find the index. It is also possible to transform the index to the symbol.

Copyright: Copyright (c) 2003

Organisation: Universität Paderborn

Declaration

public class Alphabet
extends java.lang.Object

Constructors

- *Alphabet*
  `public` **Alphabet**`( )`
  - **Usage**
    - Creates an empty Alphabet object

- *Alphabet*
  `public` **Alphabet**`( java.lang.String` **type** `)`
  - **Usage**
    - Creates a new Alphabet which is initialized with the type of symbols. Dependent on this type the symbols of the Alphabet are initialized.
  - **Parameters**
    - `type` - The type of the alphabet which should be created. For example hexbyte means that the symbols are initialized with hexadecimal byte symbols. Other possible values are char or nibble.

- *Alphabet*
  `public` **Alphabet**`( java.lang.String [] inAlpha )`
  - **Usage**
    - Creates a new Alphabet Object which is initialized with an array of symbols.
  - **Parameters**
    - `inAlpha` - An Array which holds all the different symbols of the alphabet, with which the Alphabet is initialized.

- *Alphabet*
  `public` **Alphabet**`( java.lang.String [] inAlpha,`
  `java.util.Hashtable` **inHash** `)`
  - **Usage**

    - Creates a new Alphabet which is initialized with an array of symbols and a Hashtable which is used to transform the symbols to an index.
  - **Parameters**
    - `inAlpha` - An Array which holds all the different symbols of the alphabet, with which the Alphabet is initialized.
    - `inHash` - A Hashtable which can convert any symbol of the Alphabet into an index.

Methods

- *compareTo*
  `public boolean` **compareTo**`( java.lang.String [] compare )`

  - **Usage**
    - This method is to compare an array of the symbols with the array of the actual Alphabet object.
  - **Parameters**
    - `compare` - An Array of symbols which should be compared to the symbols of the actual alphabet.
  - **Returns** - Returns whether the symbols in the array are the same like in the alphabet object or not.

- *getIndexOfEOF*
  `public int` **getIndexOfEOF**`( )`

  - **Usage**
    - This method returns the index of the EOF symbol. In the primitive case this is just the number of symbols used, because the index of the first symbol is 0. But if the index method is changed, this must be also changed.
  - **Returns** - The index of the EOF symbol.

- *getSize*
  `public int` **getSize**`( )`

  - **Usage**
    - This method returns the number of symbols which are saved in the Alphabet object.
  - **Returns** - The number of symbols in the Alphabet.

- *getSymbols*
  `public String` **getSymbols**`( )`
  - **Usage**
    - This method returns all the symbols of the Alphabet obejct as an Array.
  - **Returns** - The Array which holds all the symbols of the Alphabet.

- *index_to_String*
  `public String` **index_to_String**`( int  i )`
  - **Usage**
    - This method converts a int value into a hexadecimal byte value
  - **Parameters**
    - `i` - The int value which should be converted into an hexadecimal byte value
  - **Returns** - The hexadecimal byte value of the int value.

- *setHashtable*
  `public void` **setHashtable**`( java.util.Hashtable  inHash )`
  - **Usage**
    - Method to set a new Hashtable for symbol transformation in the Alphabet Object.
  - **Parameters**
    - `inHash` - A Hashtable which can convert any symbol of the Alphabet into an index.

- *StringToInt*
  `public int` **StringToInt**`( java.lang.String  in )`
  - **Usage**
    - This method converts a String to its index.
  - **Parameters**
    - `in` - The symbol which should be converted into its index
  - **Returns** - The index of the symbol.

- *toString*
  `public String` **toString**`( )`
  - **Usage**
    - Returns all the symbols of the alphabet as a String separated by spaces.
  - **Returns** - All symbols of the alphabet as a String.

6.2.3 Class **Decode1**

Title: Studienarbeit - Arithmetischer Kodierer / Dekodierer

Description: This class decodes the input to a symbol dependent on the values in the model object.

Copyright: Copyright (c) 2003

Organisation: Universität Paderborn

Declaration

public class Decode1 **extends** java.lang.Object **implements** ArithmeticDecode

Constructors

- *Decode1*
  `public` **Decode1(** `arithmeticcoding.Model` **inModel,** `long` **inBuffer** **)**
  - **Usage**
    - Constructs a new ArithmeticDecode object which is initialized with an model object which has some parameters for decoding and the buffer.
  - **Parameters**
    - `inModel` - The model object which hold static values like bit precision and values for the upper and lower range of the current code range.
    - `inBuffer` - The input

METHODS

- *decodeSymbol*
  `public int` **decodeSymbol**`( )`
  - **Usage**
    - This method computes the symbol dependent on the input and the parameters given in the model object.
  - **Returns** - The index of the decoded symbol.
  - **Exceptions**
    - `java.util.NoSuchElementException` - if EOF has no frequency ->Model

6.2.4 CLASS Decoder

Title: Studienarbeit - Arithmetischer Kodierer / Dekodierer

Description: This class manages the reading and decoding process. It has a hashtable with all probability distributions and their belonging contexts. Dependent on the actual context one distribution is chosen and set to the model.

Copyright: Copyright (c) 2003

Organisation: Universität Paderborn

DECLARATION

public class Decoder **extends** java.lang.Object

CONSTRUCTORS

- *Decoder*
  `public` **Decoder**`( arithmeticcoding.Model` **inModel**`, java.util.Hashtable` **inDist** `)`
  - **Usage**
    - Creates a new Decoder object which is initialized with a Model which holds the needed variables for decoding and the Hashtable which holds all probability distributions.
  - **Parameters**
    - `inModel` - The Model with all needed variables.
    - `inDist` - All probability distributions in a Hashtable.

METHODS

- *actualiseBuffer*
  `public static long` **actualiseBuffer**`( long` **inBuffer** `)`
  - **Usage**
    - This method actualises the buffer, if the intervall is mapped.
  - **Parameters**
    - `inBuffer` - to be actualised
  - **Returns** - actualised buffer
  - **Exceptions**
    - `java.io.EOFException` -

- *setNextDistribution*
  `public void` **setNextDistribution**`( )`
  - **Usage**
    - This method chooses dependent on the actual context the next distribution. When no context exist the first distribution is chosen. When no distribution with the actual context exist the method looks if it exists a shorter context with belonging distribution. That means it looks for a distribution which has the same context as the actual context without the first or more than the first symbol(s).

- *startDecoding*
  `public void` **startDecoding**`( java.lang.String` **inputFile**`, java.lang.String` **outputFile**`, java.lang.String` **verboseFile** `)`

- **Usage**
  - The main method which manages the decoding process.
- **Parameters**
  - `inputFile` - The path of the input file as String.
  - `outputFile` - The path of the output file as String.
  - `verboseFile` - The path of the verbose file as String.
- **Exceptions**
  - `java.nio.charset.CharacterCodingException` - If the function is called but there are no ProbabilityDistribution objects the Exception is thrown.

6.2.5 Class **IOOperations**

Title: Studienarbeit - Arithmetischer Kodierer / Dekodierer

Description: This class contains all needed functions for input and output. It reads information from files or from stdin and writes to a file or to stdout. It also contains a function for verbose output.

Copyright: Copyright (c) 2003

Organisation: Universität Paderborn

Declaration

public class IOOperations **extends** java.lang.Object

Constructors

- *IOOperations*
  `public` **IOOperations**`( )`
  - **Usage**
    - Creates a new instance of IOOperations to access some functions.

- *closeOutputStreams*
  `public void` **closeOutputStreams**`( )`
  - **Usage**
    - When a file was used for the output data, the stream is closed so that the file can be saved. The same thing applies to the verbose data.

- *initBufferedInputStream*
  `public void` **initBufferedInputStream**`( java.lang.String` **strInFile** `)`
  - **Usage**
    - This method initializes the BufferedReader with the input file or the standard input if no filename was given.
  - **Parameters**
    - `strInFile` - The path of the input file which should be read, or if no input file exists and information should be read from keyboard

---

- *initBufferedOutputStream*
  `public void` **initBufferedOutputStream**`( java.lang.String` **strOutFile** `)`
  - **Usage**
    - This method initializes the BufferedWriter with the output file or the standard output if no filename was given.
  - **Parameters**
    - `strOutFile` - The path of the output file which should be written, or if no output file exists and information should be written to standard output.

- *initBufferedStreams*
  `public void` **initBufferedStreams**`( java.lang.String` **strInFile**`, java.lang.String` **strOutFile**`, java.lang.String` **strVerboseFile** `)`
  - **Usage**
    - This method initializes all streams needed for input and output.
  - **Parameters**

* `strInFile` - The path of the input file which should be read, or if no input file exists and information should be read from keyboard
* `strOutFile` - The path of the output file which should be written, or if no output file exists and information should be written to standard output.
* `strVerboseFile` - The path of the verbose file which should be written, or if no verbose file exists and information should be written to standard output.

- *initBufferedVerboseStream*
  `public void` **initBufferedVerboseStream**`( java.lang.String` **strVerboseFile** `)`
  - **Usage**
    * This method initializes the BufferedVerboseWriter with the verbose file or the standard output if no filename was given.
  - **Parameters**
    * `strVerboseFile` - The path of the verbose file which should be written, or if no verbose file exists and information should be written to standard output.

- *readBit*
  `public int` **readBit**`( )`
  - **Usage**
    * This methods reads a single bit from the input stream and returns it as bit (an int either 0 or 1).
  - **Returns** - The bit which was read, either 0 or 1.
  - **Exceptions**
    * `java.io.EOFException` -

- *readByteDistributions*
  `public Hashtable` **readByteDistributions**`(`
  `arithmeticcoding.Model` **inModel**`, java.lang.String` **filename** `)`
  - **Usage**
    * This method reads Contexts and their dependent Probability Distributions from a file with the following format: Every important line begins with !, followed by the context when the distribution should be used in ", and then followed by the frequencies of the symbols. The frequencies can also be separated by ',' because all ',' are removed.

  - **Parameters**
    * `inModel` - The model which holds some needed variables.
    * `filename` - The name of the file that should be read.
  - **Returns** - The Hashtable which contains all the distributions listed in the file

---

- *readChar*
  `public char` **readChar**`( )`
  - **Usage**
    * Reads a single character from input stream and throws EOFException if no further symbols could be read.
  - **Returns** - The read symbol as a char.
  - **Exceptions**
    * `java.io.EOFException` - If the end of the file was reached this exception is thrown.

- *readHexByte*
  `public String` **readHexByte**`( )`
  - **Usage**
    * This methods reads a single hexadecimal symbol from input Stream and returns it.
  - **Returns** - The hexadecimal symbol which was read.
  - **Exceptions**
    * `java.io.EOFException` - If the end of the file was reached this exception is thrown.

- *readProbabilities*
  `public int` **readProbabilities**`( java.io.File` **filename** `)`
  - **Usage**
    * This method reads probabilities / frequencies given in a File. They can be separated by ',' and the number of values is not needed because the array adapts on the number of values.
  - **Parameters**
    * `filename` - The File which contains the frequencies
  - **Returns** - An Array which contains the frequencies of the symbols

---

- *readSymbol*
  `public int` **readSymbol**`( )`
  - **Usage**

    - This methods reads a single symbol from input Stream and returns it.
  - **Returns** - The symbol which was read.

- *readXMLFile*
  `public Hashtable` **readXMLFile**`( arithmeticcoding.Model` **inModel**`, java.lang.String` **filename** `)`
  - **Usage**
    - This method reads Contexts and their dependent Probability Distributions from a xml file with the format given in the file 'config.dtd'. It also reads some variables which are listed in the dtd file
  - **Parameters**
    - `inModel` - The model which holds some needed variables.
    - `filename` - The name of the file that should be read.
  - **Returns** - The Hashtable which contains all the distributions listed in the file

---

- *saveXMLFile*
  `public boolean` **saveXMLFile**`( arithmeticcoding.Model` **inModel**`, java.util.Hashtable` **distributions**`, java.lang.String` **filepath** `)`
  - **Usage**
    - This method writes contexts and their dependent Probability Distributions in the format like readXMLFile. It also writes some variables which are listed in the dtd file
  - **Parameters**
    - `inModel` - The model which contains the variables which should be written to the output file.
    - `distributions` - The Hashtable which contains all the distributions which should be listed in the file
    - `filepath` - The name of the file which should contain the distributions and variables.
  - **Returns** - Whether the file was correctly saved or not.

- *verboseOutput*
  `public static void` **verboseOutput**`( java.lang.String` **output**`, int` **level** `)`

- *writeByteDistributions*
  `public boolean` **writeByteDistributions**`( java.util.Hashtable` **distributions**`, java.lang.String` **strOutputFile** `)`

- **Usage**
  - This method writes contexts and their dependent Probability Distributions in the format like readByteDistributions to an output File.
- **Parameters**
  - `distributions` - The Hashtable which contains all the distributions which should be listed in the file.
  - `strOutputFile` - The name of the file which should contain the distributions.
- **Returns** - Whether the file was correctly saved or not.

- *writeString*
  `public static void` **writeString**`( java.lang.String` **out** `)`
  - **Usage**
    - This methods puts a given String to the earlier defined output Stream.
  - **Parameters**
    - `out` - The String which should be put to output Stream.

6.2.6 Class Model

Title: Studienarbeit - Arithmetischer Kodierer / Dekodierer

Description: This class contains information which is needed by the encoder to encode the symbols. For example this is the actual interval, the outstanding bits and the values of the used bit precision.*

Copyright: Copyright (c) 2003

Organisation: Universität Paderborn

Declaration

public class Model
extends java.lang.Object

CONSTRUCTORS

- *Model*
  `public` **Model**`( )`
  - **Usage**
    - Creates a new model which initializes the code range with maximum values

METHODS

- *actualizeContextLength*
  `public void` **actualizeContextLength**`( int` **newLength** `)`
  - **Usage**
    - Sets the length of the context to the maximum of the old and the new value.
  - **Parameters**
    - `newLength` - The length of the context for a special distribution.

- *getBitPrecision*
  `public int` **getBitPrecision**`( )`
  - **Usage**
    - Returns the actual value of the maximal bit precision for code values.
  - **Returns** - The actual value of the maximal bit precision for code values.

- *getBitsForFreq*
  `public int` **getBitsForFreq**`( )`
  - **Usage**
    - Returns the value of maximal bit precision used for frequency counts.
  - **Returns** - The value of maximal bit precision used for frequency counts.

- *getBlockLength*
  `public int` **getBlockLength**`( )`

  - **Usage**
    - Returns the actual value of the block length.
  - **Returns** - The value of the block length.

- *getContextLength*
  `public static int` **getContextLength**`( )`
  - **Usage**
    - Returns the value of the maximal context length used.
  - **Returns** - The value of the maximal context length used.

- *getFrequencyOfEOF*
  `public static int` **getFrequencyOfEOF**`( )`
  - **Usage**
    - Returns the frequency which should be used for the EOF symbol
  - **Returns** - The freqeuency of the EOF symbol

- *getHigh*
  `public long` **getHigh**`( )`
  - **Usage**
    - Returns the actual value of the high bound of the code range.
  - **Returns** - The value of the high bound of the code range.

- *getLow*
  `public long` **getLow**`( )`
  - **Usage**
    - Returns the actual value of the low bound of the code range.
  - **Returns** - The value of the low bound of the code range.

- *getMaxAlphabetSize*
  `public static int` **getMaxAlphabetSize**`( )`
  - **Usage**
    - Returns the maximal size of symbols in alphabets.
  - **Returns** - The maximal size of symbols in alphabets.

- *getMaxFreqValue*
  `public static long` **getMaxFreqValue**`( )`
  - **Usage**

    - Returns the maximal value which is allowed to be used for cumulative frequencies.
  - **Returns** - The maximal value which is allowed to be used for cumulative frequencies.

- *getOutstanding*
  `public int` **getOutstanding**`( )`
  - **Usage**
    - Returns the number of outstanding bits.
  - **Returns** - The number of outstanding bits.

- *getProbabilities*
  `public ProbabilityDistribution` **getProbabilities**`( )`
  - **Usage**
    - Returns the actual used probability distribution used by the model.
  - **Returns** - The actual used probability distribution used by the model.

- *getVerbose*
  `public static int` **getVerbose**`( )`
  - **Usage**
    - Returns the actual value of the verbose level.
  - **Returns** - The actual value of the verbose level.

- *increaseOutstanding*
  `public void` **increaseOutstanding**`( )`
  - **Usage**
    - Increases the number of outstanding bits by one.

- *isSetHexCoding*
  `public boolean` **isSetHexCoding**`( )`
  - **Usage**
    - Returns whether the output stream of the encoder should consist of hexadecimal values or not.
  - **Returns** - The boolean value for use of hexadecimal output.

- *reset*
  `public void` **reset**`( )`

- **Usage**
  - Resets the bounds to start values, and resets the outstandig bits. Needed to code more than one time.

- *resetOutstanding*
  `public void` **resetOutstanding**`( )`
  - **Usage**
    - Set the number of outstanding bits to 0.

- *setBitPrecision*
  `public void` **setBitPrecision**`( int` **inPrecision** `)`
  - **Usage**
    - Sets the new value for the maximal bit precision of code values used for encoding. It also sets the high bound of the code range to a new value because it depends on the maximal bit Precision.
  - **Parameters**
    - `inPrecision` - The new value for the maximal bit precision for code values

---

- *setBitsForFreq*
  `public static void` **setBitsForFreq**`( int` **bitsForValue** `)`
  - **Usage**
    - Sets the new value for the maximal bit precision used for frequency counts.
  - **Parameters**
    - `bitsForValue` - The actual value of bits which should be used for freqeuncy counts

---

- *setBlockLength*
  `public void` **setBlockLength**`( int` **newLength** `)`
  - **Usage**
    - Sets a new value for the the block length coding.
  - **Parameters**
    - `newLength` - The new value for the block length.

- *setHexCoding*
  `public void` **setHexCoding**`( boolean` **hex** `)`
  - **Usage**

        * This function set whether the output stream of the encoder should consist of hexadecimal values or not.
    - **Parameters**
        * `hex` - The boolean value for use of hexadecimal output.

- *setHigh*
`public void` **setHigh**`( long` **inHigh** `)`
    - **Usage**
        * Sets a new value for the high bound of the code range.
    - **Parameters**
        * `inHigh` - The new value for the high bound of the code range.

- *setLow*
`public void` **setLow**`( long` **inLow** `)`
    - **Usage**
        * Sets a new value for the low bound of the code range.
    - **Parameters**
        * `inLow` - The new value for the low bound of the code range.

- *setMaxAlphabetSize*
`public void` **setMaxAlphabetSize**`( int` **inSize** `)`
    - **Usage**
        * Sets the new maximal size of symbols in alphabets.
    - **Parameters**
        * `inSize` - The new maximal size of symbols in alphabets.

- *setProbabilities*
`public void` **setProbabilities**`( arithmeticcoding.ProbabilityDistribution` **inProb** `)`
    - **Usage**
        * Sets a new probability distribution to the model which is then used for encoding.
    - **Parameters**
        * `inProb` - The new probability distribution which should be used.

- *setVerbose*
`public void` **setVerbose**`( int` **inVerbose** `)`
    - **Usage**

* Sets the new value for the verbose level

– **Parameters**

* `inVerbose` - The new value for the verbose level

6.2.7 Class **ProbabilityDistribution**

Title: Studienarbeit - Arithmetischer Kodierer / Dekodierer

Description: This class holds a single instance of a Alphabet Object. For this alphabet it contains cumulative frequencies. With two of such values and the total frequency value the probability of each symbol can be calculated. It also contains the context when the distribution should be used.

Copyright: Copyright (c) 2003

Organisation: Universität Paderborn

Declaration

public class ProbabilityDistribution **extends** java.lang.Object

Constructors

- *ProbabilityDistribution*
  `public` **ProbabilityDistribution**`( )`

  – **Usage**

  * Creates an empty Probability Distribution object.

- *ProbabilityDistribution*
  `public` **ProbabilityDistribution**`( arithmeticcoding.Alphabet` **myAlphabet** `)`

  – **Usage**

* Creates a new Probability Distribution which initializes the frequencies of each symbol with one.

– **Parameters**

* `myAlphabet` - The Alphabet which contains the symbols without belonging frequencies.

- *ProbabilityDistribution*
`public` **ProbabilityDistribution**`( int [] myFreq, arithmeticcoding.Alphabet` **myAlphabet** `)`

– **Usage**

* Creates a Probability Distribution object which is initialized with an array which holds the cumulative frequencies of the symbols and the the belonging alphabet to the symbols.

– **Parameters**

* `myFreq` - An Array with cumulative frequencies of the symbols.
* `myAlphabet` - An Alphabet which holds all the symbols.

- *ProbabilityDistribution*
`public` **ProbabilityDistribution**`( int [] myFreq, arithmeticcoding.Alphabet` **myAlphabet**`, int` **myMinFreq** `)`

– **Usage**

* Creates a Probability Distribution object which is initialized with an array which holds the cumulative frequencies of the symbols, the belonging alphabet to the symbols and the minimal frequency.

– **Parameters**

* `myFreq` - An Array with cumulative frequencies of the symbols.
* `myAlphabet` - An Alphabet which holds all the symbols.
* `myMinFreq` - Minimal frequency of myFreq

METHODS

- *getAlphabet*
`public Alphabet` **getAlphabet**`( )`

– **Usage**

* Method which returns the Alphabet which belongs to the Probability. Distribution

  - **Returns** - The Alphabet which belongs to the Probability Distribution.

- *getContext*
  `public String` **getContext**`( )`
  - **Usage**
    - Returns the context when this distribution is to be used.
  - **Returns** - The context when this distribution is to be used.

- *getCumFreq*
  `public int` **getCumFreq**`( )`
  - **Usage**
    - Method which returns the actual Array with the cumulative frequencies of the symbols.
  - **Returns** - Array with the cumulative frequencies of the symbols.

- *getFreq*
  `public int` **getFreq**`( int` **symbol** `)`
  - **Usage**
    - Method which returns the cumulative frequency value at a special position. With two of such values and the total frequencies the probability of the Distribution can be calculates.
  - **Parameters**
    - `symbol` - The position of the symbol
  - **Returns** - The cumulative frequency at a special position.

- *getMinProb*
  `public double` **getMinProb**`( )`
  - **Usage**
    - Method which returns the minimal probability
  - **Returns** - minimal frequency of cumFreq

- *recalculateDist*
  `public void` **recalculateDist**`( )`
  - **Usage**
    - Method which recalculates all frequencies. It is called if the cumulative total frequency of all symbols is greater than a special value. This is needed because if the value is too large an overflow can occur.

- *setAlphabet*
  `public void` **setAlphabet**`( arithmeticcoding.Alphabet` **inAlphabet** `)`

  - **Usage**
    - Method which sets a new Alphabet which belongs to the Probability. Distribution
  - **Parameters**
    - `inAlphabet` - The new Alphabet which should now belong to the Probability Distribution.

- *setContext*
  `public void` **setContext**`( java.lang.String` **inContext** `)`

  - **Usage**
    - Sets the context when this distribution is to be used.
  - **Parameters**
    - `inContext` - The context for use of this distribution.

- *toString*
  `public String` **toString**`( )`

  - **Usage**
    - Returns all the frequencies of the symbols as a String separated by spaces.
  - **Returns** - All frequencies of the symbols as a String.

- *updateDist*
  `public void` **updateDist**`( int` **symbol** `)`

  - **Usage**
    - This method updates the array with cumulative frequencies if a new symbol was read because the probability that this symbol now occurs is greater than before.
  - **Parameters**
    - `symbol` - The index of the symbol which probability should be enlarged.

6.2.8 Class **Utility**

Title: Studienarbeit - Arithmetischer Kodierer / Dekodierer

Description: This class defines some methods which are, or can be needed in different objects or classes.

Copyright: Copyright (c) 2003

Organisation: Universität Paderborn

Declaration

public class Utility **extends** java.lang.Object

Constructors

- *Utility*
  `public` **Utility**`( )`
  - **Usage**
    - Creates a new Utility object to access all methods.

Methods

- *calcCumulative*
  `public int` **calcCumulative**`( int [] freq, int` **increaseArray**
  `)`
  - **Usage**
    - This method calculates the cumulative frequencies in an array with the given frequencies. With an value it is possible to make the cumulative array larger than the original array.

- **Parameters**
  - `freq` - The Array which contains the normal frequencies.
  - `increaseArray` - An integer value that enlarges the cumulative array dependent on the size of the given array and the value. Allowed values are 0 or 2.
- **Returns** - Array that contains the cumulative frequencies

---

- *questionOptionPane*
  `public static int` **questionOptionPane**`( java.lang.Object [] arguments, java.lang.String reason )`
  - **Usage**
    - Creates a new question JOptionPane which is initialized with the reason of the question and the text for the buttons. It returns the result of JOptionPane.
  - **Parameters**
    - `arguments` - The label for the buttons which are choosable in the JOptionPane
    - `reason` - The reason why this question is shown.
  - **Returns** - The result which button chosen as int value.

- *showInstructionManual*
  `public void` **showInstructionManual**`( )`
  - **Usage**
    - This method reads the Instruction manual from a file and shows it to the user.

- *toHexString*
  `public static String` **toHexString**`( int i )`
  - **Usage**
    - This method converts integer values into hexadecimal values. It is needed because the transformation given by Integer.toHexString(int i) returns hexadecimal byte also of the length 1 without the leading zero.
  - **Parameters**
    - `i` - The value which should be transformed into a hexadecimal String.
  - **Returns** - The hexadecimal value of the given integer value

---

- *warningMessage*
  `public static void` **warningMessage**`( java.lang.String` **message**`, java.lang.String` **title** `)`
  - **Usage**
    - Shows a warning message in a popup-window.
  - **Parameters**
    - `message` - String The message of the warning window.
    - `title` - String The title of the warning window.

- *warningOptionPane*
  `public static int` **warningOptionPane**`( java.lang.Object []` **arguments**`, java.lang.String` **reason** `)`
  - **Usage**
    - Creates a new warning JOptionPane which is initialized with the reason of the warning and the text for the buttons. It returns the result of JOptionPane.
  - **Parameters**
    - `arguments` - The label for the buttons which are choosable in the JOptionPane
    - `reason` - The reason why this warning is shown.
  - **Returns** - The result which button chosen as int value.

Literaturverzeichnis

[Bel90] T. Bell, J. Cleary, I. Witten. Text compression. Prentice Hall, 1990.

[Blo03] J. Blömer. Algorithmische Codierungstheorie I. Universität Paderborn, WS 2002/2003.

[Dogma] The data compression library. http://datacompression.info/ ArithmeticCoding.shtml.

[Hel93] G. Held. Data Compression, Techniques and Applications. Third Edition. John Wiley and Sons LTD, 1993.

[Mof95] A. Moffat, R. Neal, I. Witten. Arithmetic Coding Revisted. Proc. IEEE Data Compression Conference, Snowbird, Utah, March 1995.

[Mof02] A. Moffat, A. Turpin. Compression and Coding Algorithms. Kluwer Academic Publishers, 2002.

[Say00] K. Sayood. Introduction to Data Compression. Second Edition. Morgan Kaufmann Publishers, Inc., 2000.

[Sto88] J. Storer. Data Compression, Methods and Theory. Computer Science Press, Inc., 1988.

[Sto92] J. Storer. Image and text compression. Kluwer Academic Publishers, 1992.

[Tod03] C. Todtenbier. Implementierung eines arithmetischen Kodierers in Java. Studienarbeit, 2003.

[Wit87a] I. Witten, R. Neal, J. Cleary. Arithmetic Coding for Data Compression. Communications of the ACM, Volume 30, Number 6, June 1987.

[Wit87b] I. Witten, R. Neal, J. Cleary. Software implementing arithmetic coding in C. http://www.cs.toronto.edu/r̃adford/ac.software.html, 1987.

[Wit99] I. Witten, A. Moffat, T. Bell. Managing Gigabytes, Compressing and Indexing Documents and Images. Second Edition. Morgan Kaufmann Publishers, Inc., 1999.

Index