Proseminar Datenkompression

Zusammenfassung

Gehalten am 19.7.2001 von Dominik Hofmann

Inhaltsangabe

1.Lauflängencodierung
- Prinzip
- Codieren binärer Dateien
- Codieren mit Escape - Sequenzen

2.Codieren mit variabler Länge
- Prinzip
- Präfixcodes

3.Huffman - Code
- Algorithmus
- Laufzeitanalyse
- Anwendung

4. Lempel - Ziv - Algorithmen
    4.1 LZ77
    4.2 LZ78

5. Arithmetische Codierung

6. Literaturverzeichnis

1.Lauflängencodierung

Bei der Lauflängencodierung handelt es sich um ein relativ einfaches Verfahren zur Datenkompression, dass aber richtig eingesetzt hoch effektiv sein kann. Dieses Kompressionsverfahren nutzt lange Folgen sich wiederholender Zeichen, sog. Läufe, den einfachsten Typ einer Redundanz in einer Datei, aus, um eine Datei in kompakter Form zu speichern. Dabei wird jede Serie sich wiederholender Zeichen durch eine einmalige Angabe des Zeichens und der Anzahl der Wiederholungen ersetzt (z.B.: AAAABBBAA => 4A3BAA). Läufe der Länge eins oder zwei zu codieren lohnt sich nicht, da dadurch keine Einsparung erzielt werden kann (|AA| = |2A|, |A| < |1A|).
Für binäre Dateien gibt es eine verfeinerte Variante, die noch deutlich höhere Einsparungen ermöglicht. Dabei wird die Tatsache ausgenutzt, daß sich Läufe von 0 und 1 abwechseln, so daß auf das speichern der Werte 0 und 1 verzichtet werden kann und man nur die Lauflängen angeben muß. Um ein eindeutiges Decodieren zu ermöglichen, muß der Codierer dem Decoder aber mitteilen, mit welchem der Zeichen jede Zeile beginnt und wie Lange die Zeilen sind.
Die beiden bisher vorgestellten Verfahren eignen sich nicht zur Darstellung von Zeichenreihen, die aus einer Mischung von Buchstaben und Ziffern bestehen, da es dem Decoder in so einem Fall nicht möglich wäre bei einer Ziffer im Eingabestrom zu entscheiden, ob es sich bei der Zahl um eine Längenangabe oder um ein Zeichen des original Textes handelt.
Deshalb wurde eine Möglichkeit zur Codierung beliebiger Folgen von Zeichen aus einem festem Alphabet entwickelt, bei dem nur die Zeichen dieses Alphabets verwendet werden. Verwendet man dabei ein reines Buchstaben-Alphabet fehlen Zahlen, die zur Längenangabe verwendet werden können. Deshalb greift man auf die Buchstaben zurück, wobei das i-te Zeichen das Alphabets dazu benutzt wird, um die Zahl i darzustellen. Wie kann man aber nun die Buchstaben, die zur Längenangabe dienen und diejenigen, die Buchstaben repräsentieren unterscheiden? Die Lösung dieses Problems sind sog. Escape - Zeichen, d.h. ein bestimmtes Zeichen, das nur selten in der Datei vorkommt, wird vor jede Sequenz aus Zähler und sich wiederholendem Zeichen gestellt. Für das deutsche Alphabet wäre zum Beispiel Q als Escape - Zeichen gut geeignet. (z.B. AAAABBBBBCC = QDAQEBCC). Da die Escape - Sequenz selbst drei Zeichen lang ist lohnt sich der Aufwand zur Verschlüsselung erst ab einer Folge von mehr als drei Zeichen. Doch was passiert, wenn das Escape - Zeichen selbst verschlüsselt werden soll? Auch wenn Q relativ selten ist, taucht es hin und wieder auf. Für diesen Fall implementiert man eine spezielle Sequenz, die zum Beispiel aus Q<Leerzeichen> bestehen kann.
Wie man leicht erkennen kann lassen sich durch das geschilderte Verfahren erhebliche Einsparungen erzielen, wenn viele lange Läufe in einer Datei vorkommen. Die Verfahren der Lauflängencodierung sind also gut geeignet, wenn es darum geht, Dateien mit langen Läufen gleicher Zeichen, wie zum Beispiel Bilder oder Satz-Systeme zu codieren, während sie für normalen Textdateien eher ungeeignet sind, da hier normalerweise keine Läufe von mehr als drei gleichen Zeichen auftreten.

2. Codierung mit variabler Länge

Dieses Verfahren ist wesentlich besser für Textdateien geeignet als zum Beispiel die Lauflängencodierung. Normalerweise werden Dateien gespeichert, indem jedem Zeichen eine feste Anzahl von Bits zugewiesen wird. Beim ASCII - Code sind es normalerweise sieben oder acht Bits pro Zeichen. Soll nun beispielsweise ein Text mit 100000 Zeichen, der der

Einfachheit halber auf einem Alphabet von nur sechs Buchstaben basiert, codiert werden, so benötigt man bei einem Code fester Länge drei Bits für jedes Zeichen. Um den kompletten Text zu speichern sind also 300000 Bits notwendig. Verwendet man dagegen einen Code variabler Länge geht das erheblich kürzer. Dabei wird die Tatsache ausgenutzt, daß Zeichen in einer normalen Sprache mit unterschiedlicher Häufigkeit auftreten. In der deutschen Sprache sind e oder a sehr häufig, während q oder auch x relativ selten vorkommen. Beim Codieren bekommen nun häufige Zeichen sehr kurze Codes, während seltene Codes relativ lange Zeichen bekommen.
Beispiel:

	a	b	c	d	e	f
Häufigkeit in Tausend	45	13	12	16	9	5
Code fester Länge	000	001	010	011	100	101
Code variabler Länge	0	101	100	111	1101	1100

Benötigte Bits zur Codierung:
(45 * 1 + 13 * 3 + 12 * 3 + 16 * 3 + 9 * 4 + 5 * 4) * 1000 = 224000

In dem Beispiel komm a mit 45000 sehr oft vor und bekommt deshalb einen Code, der nur ein Bit lang ist, während den seltenen Zeichen e und f Codes mit je vier Bits zugewiesen werden. Wenn man nun den Speicherplatz berechnet, den man mit dieser Methode für den 100000-Zeichentext benötigt, kommt man auf 224000 Bits, was eine Einsparung von ungefähr 25% bedeutet. Bei diesem Verfahren gibt es aber noch etwas wichtiges zu beachten, damit es beim Decodieren keine Probleme gibt. Während das Decodieren bei Codes fester Längen sehr einfach ist, da man weiß, daß immer nach der gleichen Anzahl von Bits ein neues Zeichen beginnt, ist das bei Codes variabler Länge nicht der Fall. Trotzdem muß der Code eindeutig zu lesen sein. (Gegenbeispiel: a = 0, b = 1, c = 01 => 01 = ab = c) Dies wird durch sogenannte Präfixcodes, bei denen kein Codewort Vorsilbe des Codes eines anderen Zeichens ist, erreicht. D.h. keine Zeichenfolge stimmt mit dem Anfang einer anderen überein.
Nun stellt sich natürlich die Frage, wie man einen solchen Code erzeugt und darstellt. Die Antwort auf diese Frage liefert der Huffman Code.

3. Der Huffman - Code

Bei diesem bereits 1951 von David A. Huffman veröffentlichtem Algorithmus handelt es sich um eines der bekanntesten und am weitesten verbreiteten Verfahren zur Datenkompression. Huffmans Algorithmus erzeugt einen optimalen Präfixcode variabler Länge, der durch einen Binärbaum dargestellt wird. Als Voraussetzung benötigt der Standard - Algorithmus die Häufigkeitsverteilung der verschiedenen Zeichen in der zu codierenden Datei. Dazu bedient man sich entweder einer vorgefertigten Statistik zur gegebenen Sprache oder man implementiert einen Durchlaufe über die zu codierende Datei, wobei die vorkommenden Zeichen zusammen mit ihrer Häufigkeit ermittelt werden. Erst dann kann der Baum, der den Code repräsentiert und dessen Blätter die Zeichen des Alphabets sind, erstellt werden. Zu Beginn erzeugt man für jedes Zeichen des Alphabets einen Knoten, dem die Häufigkeit des entsprechenden Zeichens zugeordnet wird. Diese Knoten werden in eine Liste eingefügt. Dann werden bei jedem Schritt die zwei Konten mit der geringsten Häufigkeit ausgewählt und an einen gemeinsamen Vater gekoppelt, dessen Häufigkeitswert sich aus der Summe der Häufigkeiten seiner Söhne errechnet. Falls mehr als zwei Knoten als Söhne in Frage kommen ist es egal, welche man auswählt. Danach wird der aus den beiden Knoten mit der geringsten Häufigkeit entstandene Teilbaum wieder in die Liste eingefügt und man wählt erneut die

beiden Knoten mit den geringsten Häufigkeiten, um diese zusammenzufügen. Diese Schleife wird solange durchlaufen, bis die Liste nur noch ein Element, den fertigen Huffman - Baum, enthält. Man erkennt leicht, daß die Blätter die Zeichen des Alphabets zusammen mit ihren Häufigkeiten darstellen, während die inneren Knoten des Baumes nur als Verknüpfungsglieder mit der Summe der Häufigkeiten ihrer Söhne dienen. Den Code für die einzelnen Zeichen kann man nun relativ einfach aus dem Baum lesen. Zuerst beschriftet man die Kanten die links von einem Vaterknoten wegführen mit Null und die rechten Kanten entsprechen mit Eins. Das Codewort für ein bestimmtes Zeichen erhält man indem man von der Wurzel zu dem entsprechendem Blatt läuft und die Knotenbeschriftungen ausgibt. Der Rest ist denkbar einfach. Verwendet man keine vorgefertigte Statistik speichert oder übermittelt man zunächst den Baum oder auch nur die Liste der Zeichen und ihren Häufigkeiten, die man beim Decodieren benötigt um den Baum zu rekonstruieren. Bei einem Durchlauf über einen Text liest man dann für jedes Zeichen den Code aus dem Baum und gibt ihn aus. Das Decodieren ist ebenfalls einfach. Anhand des Codes im Eingabestrom läuft man den Baum ab bis man ein Blatt erreicht hat. Eine Null im Eingabestrom bedeutet dabei, dass man nach links laufen muss, während man bei einer Eins die rechte Kante wählt.Dann gibt man das entsprechende Zeichen unkomprimiert aus und beginnt mit dem nächsten Bit des Eingabestroms wieder bei der Wurzel.

Implementierung des Algorithmus in Pseudo-Code:
Huffman - Algorithmus:

```
Huffman(C)
1       n ← |C|
2       Q ← C
3       for i = 1 to n-1
4       do      z ← ALLOCATE-NODE()
5       x ← left[z] ← EXTRACT-MIN(Q)
6       y ← right[z] ← EXTRACT-MIN(Q)
7       f[z] = f[x] + f[y]
8       INSERT(Q,z)
9       return EXTRACT-MIN(Q)
```

C ist ein Alphabet mit n Zeichen
f[c] ist die Häufigkeit eines Zeichens C
Q ist eine Prioritätswarteschlange, die auf f basiert

In Zeile 2 wird die Warteschlange Q mit dem Alphabet basierend auf den Häufigkeiten der einzelnen Zeichen initialisiert. Die Schleife in Zeile 3-8 erzeugt einen neuen Knoten z, dem dann die beiden Knoten mit den geringsten Häufigkeiten als linker bzw. rechter Sohn zugewiesen werden. Dann wird die Häufigkeit von z berechnet, indem die Werte von x und y addiert werden. Schließlich wird z in die Warteschlange Q eingefügt. Das ganze wird (n-1)-mal wiederholt, bis nur noch der fertige Baum in der Schlange ist. Dieser wird dann mit dem return - Befehl ausgegeben.
Wenn Q als binärer Heap verwirklicht wird, kann die Initialisierung (Zeile 2) in O(n) erfolgen. Die Schleife wird (n-1)-mal durchlaufen und jede Heapoperation benötigt O(log n), insgesamt wird die Schleife also in O(n log n) durchlaufen. Daraus folgt eine Laufzeit von O(n log n).
In der Praxis wird der Huffman - Code normalerweise nicht alleine sondern im Verbund mit anderen Kompressionsverfahren eingesetzt, wo er häufig zur zweiten Codierung einer bereits komprimierten Datei verwendet wird. Beispiele hierfür sind JPEG, MPEG oder MP3.

4.Lempel - Ziv - Algorithmen

Die Ideen hinter dieser Familie von Algorithmen gehen auf Abraham Lempel und Jacob Ziv zurück, die die beiden Basisalgorithmen LZ77 und LZ78 entwickelt und veröffentlicht haben. Das Prinzip dieser Algorithmen ist es die nächsten Zeichen im Eingabestrom zu codieren, indem auf den bereits codierten Teil einer Datei oder Nachricht Bezug genommen wird und dabei entdeckt Übereinstimmungen ausnutzt werden. Der erste dieser Algorithmen war LZ77, der 1977 veröffentlicht wurde.

4.1 LZ77

Beim Codieren wird ein zweiteiliges Fenster benutzt durch das der Eingabestrom von rechts nach links geschoben wird.

SEARCH-BUFFER	LOOK-AHEAD-BUFFER
sir_sid_eastman_easily_t	eases_sea_sick_seals

→ nexte Ausgabe: (16,3,"e")

Der linke Teil des Fensters ist der sogenannte "search buffer", der die zuletzt codierten Symbole enthält, während der rechte Teil die Zeichen enthält, die als nächstes codiert werden sollen. Dieser Teil wird als "look ahead buffer" bezeichnet. Bei einer normalen Implementierung ist der search buffer mehrere Tausend Bytes lang. Während der look ahead buffer um ein Vielfaches kürzer ist.
Beim codieren wird der search buffer von rechts nach links nach einer Übereinstimmung mit dem ersten Symbol im look ahead buffer durchsucht. Der erste Treffer ist das "e" von "easily" bei einer Entfernung von acht Zeichen von der Trennlinie der beiden Fenster. Dieser Wert wird als "offset" bezeichnet. Dann wird überprüft, wie lange die Übereinstimmung ist., nämlich "eas", also drei Symbole. Anschließend wird die Suche nach links fortgesetzt und man versucht eine noch längere Übereinstimmung zu finden. Dabei landet man einen weiteren Treffer bei "eastman" mit einem offset von 16 und einer Länge von ebenfalls 3. Schließlich wird die längste Übereinstimmung, oder wenn alle gleich lang sind diejenige, die zuletzt gefunden wurde ausgewählt und mit dem Tripel (16,3,"e") codiert. Der erste Eintrag ist der offset, der Zweite die Längenangabe und als drittes wird das nächste Symbol des look ahead buffers angegeben, um den Fall abzudecken, daß keine Übereinstimmung gefunden wird.
Nach der Ausgabe wird das Fenster nach rechts bzw. der Inhalt nach links geshifted und zwar um die Länge der Übereinstimmung plus eine Stelle für das unkomprimierte Zeichen.

Codieren des Strings: "AABCAAABCD"

SEARCH-BUFFER	LOOK-AHEAD-BUFFER	AUSGABE
	AABCAAABCD	(0,0,A)
A	ABCAAABCD	(1,1,B)
AAB	CAAABCD	(0,0,C)
AABC	AAABCD	(4,2,A)
AABCAAA	BCD	(5,2,D)
AABCAAABCD		

Das Decodieren ist ebenfalls äußerst einfach. Man unterhält das selbe Fenster wie beim Codieren und verarbeitet die Tripel indem man Kopien der Strings an dem angegebenen offset und der entsprechenden Länge anfertigt und in das Fenster einfügt. Schließlich wird noch das unkomprimiert mitübermittelte Symbol angehängt und entsprechend der Länge des eingefügten Strings shiftet.

LZ77 - Decodieren von (0,0,A);(1,1,B),(0,0,C),(4,2,A),(5,2,D)

EINGABE		AUSGABE
(0,0,A)		A
(1,1,B)	A	AB
(0,0,C)	AAB	C
(4,2,A)	AABC	AAA
(5,2,D)	AABCAAA	BCD
	AABCAAABCD	

Da das Codieren mit LZ77 wegen der komplexen suche im Fenster relativ aufwendig ist, während das Decodieren sehr schnell geht, eignet sich dieses Verfahren vor allem für Archive, da dort der einmalige Aufwand zum Codieren nur eine geringe Rolle spielt, das Verfahren zum Decodieren dagegen sehr schnell sein muss.

4.2 LZ78

LZ78 ist das zweite von Lempel und Ziv entwickelte Kompressionsverfahren. Bei LZ78 gibt es keinen search buffer, sondern ein Wörterbuch, das die bisher entdeckten Strings enthält. Dieses Wörterbuch ist anfangs leer und nur durch den zur Verfügung stehenden Speicherplatz beschränkt. Die Ausgabe besteht aus Tupeln, die sich aus einem Zeiger auf das Wörterbuch und einem Feld für ein weiteres, unverschlüsseltes Symbol zusammensetzen.

LZ78 Codieren:

1. Wörterbuch = leer, y = leer, x = leer;
2. x:= nächstes Zeichen aus dem Eingabestrom
3. Ist String "y+x" im Wörterbuch?
   - Ja: - y = y + x
   - Nein: - (Zeiger auf y, "x") ausgeben
     - Trage "y + x" ins Wörterbuch ein
     - y = leer
4. Ist das Ende des Eingabestroms erreicht?
   -Nein: Gehe zu Schritt 2
   - Ja: ist y leer?
   -Nein: den y entsprechenden Code ausgeben

Ein Symbol wird eingelesen und im Wörterbuch gesucht. Wird es gefunden, wird das nächste Zeichen eingelesen und angehängt. Der dabei entstandene String wird wiederum im Wörterbuch gesucht. Solange er gefunden wird, wird immer das nächste Zeichen des Eingabestroms angehängt. Irgendwann wird der String dann nicht mehr gefunden und man fügt ihn ins Wörterbuch ein. Die Nummer des Wörterbucheintrags der letzten gefundene Übereinstimmung und das Symbol, mit dem die Suche fehlschlug werden als Tupel ausgegeben.

Codieren von "AABCAAABC" mit LZ78

noch nicht verarbeitet	Wörterbucheintrag	Ausgabe
AABCAAABC	1: A	(0,A)
ABCAAABC	2: AB	(1,B)
CAAABC	3: C	(0,C)
AAABC	4: AA	(1,A)
ABC	5. ABC	(2,C)

Das decodieren ist ebenfalls nicht schwer, dabei wird das Wörterbuch auf dieselbe Weide geführt wie beim Codieren.

Decodieren mit LZ78

1. Zu Beginn ist das Wörterbuch leer
2. x:= Wörterbuchverweis des nächsten Tupels
3. z:= das unkomprimierte Zeichen des Tupels
4. Gib die durch x im Wörterbuch repräsentierte
   Zeichenfolge und anschließend z aus
5. Trage "x+z" ins Wörterbuch ein
6. Wenn das ende des Eingabestroms noch nicht erreicht ist, zurück zu Schritt 2

Man beginnt mit einem leeren Wörterbuch. Es wird jeweils das nächste Tupel aus dem Eingabestrom gelesen und anhand des ersten Eintrags der entsprechende String dem Wörterbuch entnommen. An diesen wird das unkomprimiert mit dem Tupel übertragene Zeichen angehängt und der dabei entstandene String ausgegeben und im Wörterbuch eingetragen. Damit fährt man solange fort, bis das Ende des Eingabestroms erreicht ist.

Decodieren mit LZ78
Eingabe (0,A), (1,B), (0,C),(1,A),(2,C)

Eingabe	Wörterbucheintrag	Ausgabe
(0,A)	1: A	A
(1,B)	2: AB	AB
(0,C)	3: C	C
(1,A)	4: AA	AA
(2,C)	5: ABC	ABC

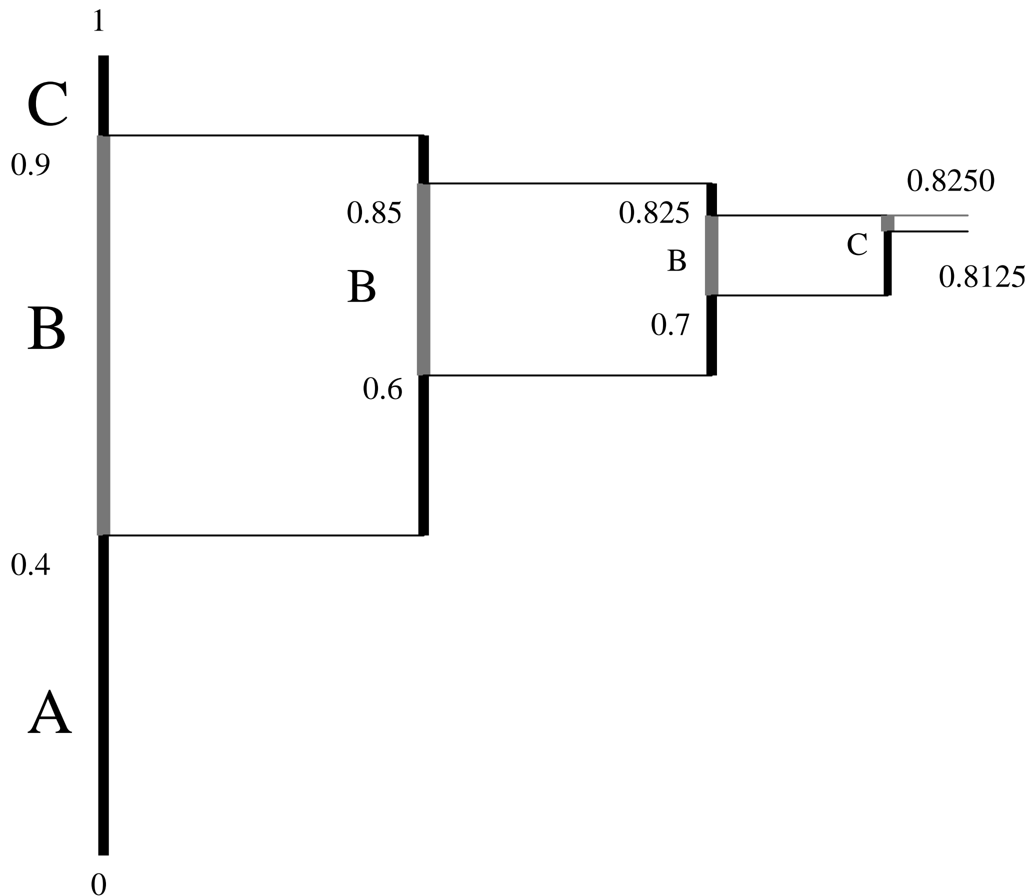
5. Arithmetische Codierung

Bei diesem Verfahren handelt es sich um eine völlig andere Kompressionsmethode, anstatt jedem Zeichen einen eigenen Code zuzuweisen wie zum Beispiel der Huffman Code, wird die komplette Eingabe in einen einzigen, normalerweise sehr langen Code verwandelt. Dieser Code ist eine reelle Zahl aus dem Intervall von Null bis Eins. Die Null vor dem Komma wird beim Speichern weggelassen, um Platz zu sparen, 0.75 wird folglich als 75 gespeichert. Das Verfahren benötigt die selben Voraussetzungen wie der Huffman - Code, nämlich die Häufigkeitsverteilung der Zeichen in der Datei.
Beispiel:

Zeichen	Wahrscheinlichkeit	Intervall
A	0.4	[0 , 0.4)
B	0.5	[0.4 , 0.9)
C	0.1	[0.9 , 1.0)

Dem ersten Beispiel liegt obige Tabelle zu Grunde. Das Intervall von [0,1) wird auf die drei Symbole entsprechend ihrer Wahrscheinlichkeiten aufgeteilt.

Um den String BBBC zu verschlüsseln wird folgendermaßen vorgegangen. Das erste Zeichen B reduziert das Intervall auf das Teilintervall vom 40% Punkt bis zum 90% Punkt, also auf [0.4, 0.9). Mit diesem Intervall fährt man dann fort. Für das zweite B wird wiederum das Teilintervall vom 40% bis zum 90% Punkt, also [0.6, 0.85) ausgewählt. Für das dritte Symbol wird das ganze nochmals wiederholt und man erhält das Intervall [0.7, 0.825). Schließlich muß noch das C codiert werden, was das Intervall auf den Teil [0.8125, 0.8250) verkleinert. Den endgültigen Code erhält man indem man einfach eine beliebige Zahl aus dem Intervall von [0.8125, 0.8250) wählt. Wie man ebenfalls leicht erkennen kann wird das Intervall mit jedem Schritt schmaler und es werden mehr und mehr Bits benötigt, um eine Zahl aus diesem Intervall darzustellen.



2.Beispiel:
Der zu codierende String lautet "SWISS_MISS"

Zeichen	Häufigkeit	Wahrscheinlichkeit	Teilintervall
S	5	0.5	[0.5 , 1.0)
W	1	0.1	[0.4 , 0.5)
I	2	0.2	[0.2 , 0.4)
M	1	0.1	[0.1 , 0.2)
_	1	0.1	[0.0 , 0.1)

Die Tabelle zeigt die Informationen, die man zum codieren benötigt. Die Häufigkeiten werden gezählt, um die Wahrscheinlichkeiten zu berechnen. Dann wird das Intervall von [0,1) wie gehabt aufgeteilt. "S" mit einer Wahrscheinlichkeit von 0.5 erhält zum Beispiel das Intervall [0.5, 1.0) und "I" mit einer Wahrscheinlichkeitswert von 0.2 wird das Intervall [0.2,0.4) zugewiesen. Beim Codieren werden zwei Variablen verwendet:

- Low = untere Grenze des Intervalls
- High = obere Grenze des Intervalls

Dann geht man nach folgendem Algorithmus vor:

Arithmetische Codierung

1. GET-FREQUENCEIES()
2. Low := 0
   High := 1
3. x := nächstes Symbol des Eingabestroms
4. NewHigh = OldLow + Range * HighRange(x)
   NewLow = OldLow + Range * LowRange(x)
5. Ist das Ende des Eingabestroms erreicht?
   - Nein: weiter bei 3
   - Ja: Ausgabe eines Wertes zwischen dem aktuellen High und Low

Range: Breite des Intervalls: High - Low
LowRange(x) bzw. HighRange(x): unteres bzw. oberes Limit der Spanne von x

Bei jedem Schritt werden die Grenzen des Intervalls neu berechnet und zwar nach folgenden Formeln:
- NewHigh = OldLow + Range * HighRange(x);
- NewLow = OldLow + Range * LowRange(x);

Range = Breite des Intervalls
LowRange(x)/HighRange(x) obere und untere Limit der Spanne von x
(Es werden die neuen Grenzen des Intervalls ausgerechnet, indem die den Endpunkten entsprechend Längen auf die untere Grenze des alten Intervalls addiert werden)
"SWISS_MISS":
Für S lautet die Rechnung:
L: 0.0 + (1.0 - 0.0) * 0.5 = 0.5
H: 0.0 + (1.0 -0.0) * 1.0 = 1.0
Für W:
L: 0.5 + (1.0 - 0.5) * 0.4 = 0.70
H: 0.5 + (1.0 - 0.5) * 0.5 = 0.75
Usw.
Decodieren:

1. Rekonstruktion der Tabelle
2. Code := Eingabeausdruck
3. x := das Symbol des Intervalls, das den Code enthält
4. Code := (Code - LowRange(x)) / Range
5. Sind alle Zeichen decodiert?
   - Nein: weiter bei 3

Zuerst muss die Tabelle rekonstruiert werden, dann wird der Code eingelesen. Der Code liegt in dem Intervall des Zeichens, das als erstes codiert worden ist. Dieses wird ausgegeben, dann der Einfluss des decodierten Zeichens auf die Zahl eliminiert, indem man folgende Formel anwendet:
Code := (Code - LowRange(x)) / Range
Danach kann man das als nächstes codierte Zeichen bestimmen, gibt es wiederum aus und wendet die Formel erneut an. Damit fährt man solange fort, bis alle Zeichen decodiert sind.

6. Literaturverzeichnis

T. H. Cormen; C. E. Leiserson; R. L. Rivest: Introduction to Algorithms,
The MIT Press, Cambridge, Mass. u. a., 1990

M. T. Goodrich, R. Tamassia: Data Structures and Algorithms in Java,
John Wiley & Sons, Inc., 1998

R. Sedgewick: Algorithmen, Addison-Wessley, Bonn 1991

T. C. Bell, J. G. Cleary, I. H. Witten: Text Compression,
Prentice Hall, Englewood Cliffs, 1990

D. Salomon: Data Compression, Springer, New York u. a., 1998

G. Held, T. R. Marshall, Data Compression.
Techniques and applications, hardware and software considerations,
Wiley, Chichester u. a., 1987

R. N. Williams: Adaptive data compression, Kluwer Academic Press, Boston u. a., 1991

Prof. Dr. F.J. Brandenburg: Skript zur Vorlesung Grundlagen der Informatik 2