Rheinisch-Westfälische Technische Hochschule Aachen

Huffman-, arithmetische Codierung

Junior Lekane Nimpa

Seminar
Kompressionsalgorithmen
im SS 2012

6. Juli 2012

Einführung

Die Datenkompression ist eine Disziplin, die sich damit beschäftigt, Verfahren (Kompressionsalgorithmen) zu entwickeln, die auf digitaler Ebene Anwendung finden. Ziel hierbei ist die Reduzierung der notwendigen Anzahl von Bits zur Speicherung oder Übertragung von Daten.Wenn man von einem Kompressionsalgorithmus A^* spricht, bezieht man sich im Allgemeinen auf zwei Algorithmen: einen Kompressionsalgorithmus A' und einen Dekompressionsalgorithmus A''. Der Kompressionsalgorithmus A' wird durch einen sogenannten Codierer und der Dekompressionsalgorithmus A'' durch einen Decodierer ausgeführt. Der Algorithmus A' bekommt als Eingabe die Daten x und generiert eine Repräsentation y und der Algorithmus A'' generiert aus y die Daten z. Abhängig von den Anforderungen, die an den Dekompressionsalgorithmus A'' gestellt sind, unterteilt man die Kompressionsalgorithmen in zwei Klassen: verlustfreie und verlustbehaftete Kompressionsalgorithmen. Ein Kompressionsalgorithmus A' ist verlustfrei, wenn für x , y und z wie oben definiert immer gilt: $z = x$. Entsprechend ist er verlustbehaftet, wenn im Allgemeinen gilt: $z \neq x$.
Der Kompressionsvorgang in sich kann konzeptuell in zwei Phasen gegliedert werden: Modellierung und Codierung. In der Modellierungsphase wird versucht, möglichst viele Informationen über die Redundanz in den zu codierenden Daten zu gewinnen. Mithilfe dieser Informationen wird dann in der Codierungsphase eine günstige Repräsentation der Daten bestimmt. In dieser Ausarbeitung beschäftigen wir uns mit der Huffman- und arithmetischen Codierung. Wir führen zuerst im ersten Teil eine Reihe von Definitionen und weitere nützliche Grundlagen ein. Diese nützlichen Grundlagen in Abschnitt 1.8 basieren im Wesentlichen auf [Sei06, Kapitel 1]. Im Teil 2 beschäftigen wir uns mit der Huffman-Codierung. Dieser Teil stütz sich auf [Huf52]. Anschließend betrachten wir die arithmetische Codierung. Hier nutzen wir als Grundlage [Say06].

1 Grundlagen

1.1 Alphabet, Symbole, Sequenzen

Ein Alphabet ist eine nichtleere Menge $\Omega = \{a_1, a_2, a_3, \ldots, a_n\}$. Man sagt: Ω ist ein endliches Alphabet, wenn $n \in \mathbb{N}$, also $n \neq \infty$. Die Elemente von Ω werden Symbole genannt. Eine Sequenz $s = (a_{i1}a_{i2}\cdots a_{ik}), k \geq 1$ und $i_j \in \{1, \ldots, n\}$, über dem Alphabet Ω ist eine nichtleere Folge von Symbolen aus Ω. Ω^m bezeichnet die Menge aller Sequenzen über Ω mit genau der Länge m. Mathematisch ausgedrückt: $\Omega^m = \{(b_1b_2 \ldots b_m) | b_i \in \Omega\}, m \geq 1$. Mit $\Omega^* = \bigcup_{m=1}^{\infty} \Omega^m$ bezeichnen wir also die Menge aller Sequenzen über Ω, auch als *Kleenesche Abschluss* von Ω genannt. Betrachte beispielsweise das Alphabet $\Omega = \{a, b, c, \ldots, z, A, B, C, \ldots Z\}$. Die Sequenz s = (Information) wäre eine gültige Sequenz über Ω.

1.2 Diskrete Datenquelle

Unter einer Datenquelle Q versteht man eine Folge von Symbolen aus einem Alphabet Ω, die theoretisch unendlich lang sein kann. Die Datenquelle ist diskret, wenn das Alphabet Ω endlich oder abzählbar unendlich ist. Eine Datenquelle kann beispielsweise ein Text, ein Bild oder ein Video sein, wobei in den zwei letzten Fällen die Symbole von Ω nichts anderes als Pixel sind. Eine Datenquelle ist gedächtnislos, wenn das Auftreten eines Symbols aus der Quelle unabhängig von den vorherigen aufgetretenen Symbolen ist. Mit $p_i = p(a_i)$ bezeichnen wir die Auftretenswahrscheinlichkeit des Symbols a_i. Mit $p = (p_1, p_2, \ldots, p_n)$ bezeichnen wir die Wahrscheinlichkeitsverteilung der Symbole des Quellenalphabetes. Es ist hier zu bemerken, dass für eine gedächtnislose Datenquelle folgendes gilt: $p(s = (a_{i1}a_{i2} \ldots a_{ik})) = p(a_{i1}) \cdot p(a_{i2}) \cdot \ldots \cdot p(a_{ik})$. Wir werden uns in dieser Ausarbeitung grundsächlich mit diskreten und gedächtnislosen Datenquellen befassen.

Da im Allgemeinen die Auftretenswahrscheinlichkeiten der einzelnen Symbole nicht exakt bestimmt werden können, müssen sie geschätzt werden. Dies geschieht mit sogenannten Modellen.

1.3 Modell

Sei Q eine Datenquelle über dem Alphabet $\Omega = \{a_1, a_2, \ldots, a_n\}$. Unter einem Modell M verstehen wir eine Abbildung $M : \Omega \to [0, 1), a_i \mapsto p_M(a_i)$, die jedes Symbol a_i aus Ω eine geschätzte

Auftretenswahrscheinlichkeit $p_M(a_i)$ zuordnet. Es muss gelten: $\sum p_M(a_i) = 1$. Der Einfachheit halber schreiben wir im Folgenden $p_i = p_M(a_i)$.

Eine Möglichkeit, eine Sequenz s über einem Alphabet Ω zu codieren, besteht darin jedes in s vorkommende Symbol a_i durch ein Codewort $C(a_i)$ zu ersetzen. Wir definieren im Folgenden formal, was ein Code ist.

1.4 Code

Sei Q eine diskrete und gedächtnislose Datenquelle über einem endlichen Alphabet Ω. Sei Γ ein weiteres endliches Alphabet mit *Kleeneschem Abschluss* Γ^*. Ein Code C über dem Alphabet Γ ist eine Abbildung

$$C : \Omega \to \Gamma^*; a_i \mapsto C(a_i),$$

die jedes Symbol aus Ω ein Wort aus Γ^* zuordnet. $C(a_i)$ heißt das Codewort vom Symbol a_i und mit l_i bezeichnen wir die Länge von $C(a_i)$, d.h. die Anzahl von Symbolen, aus denen die Sequenz $C(a_i)$ besteht. Im Allgemeinen gilt: $\Gamma = \{0, 1\}$. Wir definieren die Erweiterung eines gegebenen Codes C wie folgt.

1.5 Erweiterung eines Codes

Seien Ω, Γ endliche Alphabete und $C : \Omega \to \Gamma^*$ ein Code. Sei weiterhin $s = (a_{i1}a_{i2}\cdots a_{ik})$ eine Sequenz über Ω. Die Erweiterung von C ist eine Abbildung,

$$C^* : \Omega^* \to \Gamma^*, s = (a_{i1}a_{i2}\ldots a_{ik}) \mapsto C(a_{i1})C(a_{i2})\cdots C(a_{ik})$$

die einer Sequenz s die Konkatenation der Codewörter der Symbole, die in s vorkommen, zuweist.

Um die Decodierung einer codierten Sequenz zu erleichtern, muss ein Code C einige Eigenschaften besitzen.

1.6 Eindeutige Decodierbarkeit

Ein Code $C : \Omega \to \Gamma^*$ ist eindeutig decodierbar, wenn folgendes gilt :

$$\forall s_1, s_2 \in \Omega^* : s_1 \neq s_2 \implies C^*(s_1) \neq C^*(s_2).$$

Mit anderen Worten: jede codierte Sequenz $C^*(s)$ lässt sich auf eindeutige Weise auf die Sequenz s zurückführen. Diese Eigenschaft ist sehr wichtig. Man denkt nur an die Codierung von Bildern im medizinischen Bereich, deren falsche Decodierung beispielsweise den Arzt in die Irre führen könnte. Betrachten wir ein Beispiel.

Beispiel 1. *Seien folgende Alphabete Ω = { A, B, C } und Γ = { 0, 1 } und den folgenden Code:*

Symbol	*Codewort*
A	*0*
B	*01*
C	*10*

Versuchen wir jetzt die Sequenz 001010 *zu decodieren. Es ergeben sich offensichtlich drei Möglichkeiten dies zu tun. Nämlich:*

$$001010 \implies 0 \quad 0 \quad 10 \quad 10 \implies AACC$$

$$001010 \implies 0 \quad 01 \quad 0 \quad 10 \implies ABAC$$

$$001010 \implies 0 \quad 01 \quad 01 \quad 0 \implies ABBA$$

Solche Codes sind praktisch unbrauchbar. Eine weitere nützliche Eigenschaft, die die eindeutige Decodierbarkeit gewährleistet, ist die Präfixfreiheit.

1.7 Präfix-Code

Seien Ω, Γ endliche Alphabete und $C : \Omega \to \Gamma^*$ ein Code. Man sagt: C ist präfixfrei, wenn kein Codewort Präfix eines anderen Codewortes ist. Diese Eigenschaft ist auch in der Literatur unter dem Namen Fano-Bedingung bekannt.

Lemma 1. *Sei $C : \Omega \to \Gamma^*$ ein Präfix-Code, dann ist C auch eindeutig decodierbar.*

Beweis. Dieser Beweis kann in [Bro98, Teil II, Abschnitt 1.1.3] nachgelesen werden.
Angenommen wir haben zwei Sequenzen $s_1 = (a_1 \cdots a_n)$ und $s_2 = (b_1 \cdots b_m)$ und es gelte: $s_1 \neq s_2$ und $C^*(s_1) = C^*(s_2)$. Dann gibt es einen Index i so, dass folgendes gilt:

$$(a_1 \cdots a_i) = (b_1 \cdots b_i)$$

und $i = n$ oder $i = m$ oder $a_{i+1} \neq b_{i+1}$. Die Fälle $i = n$ und $i = m$ können wir ausschließen, da aus $i = n$ und $C^*(s_1) = C^*(s_2)$, die Aussage $C^*(b_{i+1} \cdots b_m) = \epsilon$ folgt. Nun folgt $m = i$ oder $C(b_j) = \epsilon, j \in \{(i+1), \ldots, m\}$. $m = i$ steht im Widerspruch zur Annahme $s_1 \neq s_2$ und $C(b_j) = \epsilon$ steht im Widerspruch zur Fano-Bedingung, da ϵ Präfix von jeder Sequenz ist. Analog führt der Fall $i = m$ zu einem Widerspruch. Es gilt also $i < m$ und $i < n$ und $a_{i+1} \neq b_{i+1}$. Dann gilt:

$$C^*(a_{i+1} \cdots a_n) = C^*(b_{i+1} \cdots b_m)$$

und $C(a_{i+1})$ ist Präfix von $C(b_{i+1})$ oder andersrum. Dies ist aber ein Widerspruch zur Fano-Bedingung. □

Betrachten wir folgendes Beispiel.

Beispiel 2. *Ω, Γ seien wie im vorherigen Abschnitt definiert. Wir definieren jetzt folgenden Code:*

Symbol	*Codewort*
A	*0*
B	*10*
C	*110*

Es lässt sich leicht überprüfen, dass der obige Code präfixfrei ist. Versuchen wir jetzt die gleiche Sequenz wie im vorherigen Abschnitt zu dekodieren.

$$001010 \implies 0 \quad 0 \quad 10 \quad 10 \implies AABB.$$

Es gibt keine weitere Möglichkeit diese Sequenz zu decodieren.

1.8 Nützliche Ergebnisse aus der Informationstheorie

Die Informationstheorie (von *Claude Shannon*) beschäftigt sich grundsächlich mit dem Konzept der Information. Nach dieser Theorie kann die Information quantifiziert werden. Information hat hier aber nichts mit der Semantik zu tun. D.h. nicht, was aus gegebenen Daten verstanden wird, ist Information, sondern die Daten selbst bilden die Information. Die Quantifizierung der Information benötigt die Kenntnis über die Auftretenswahrscheinlichkeiten der Symbole, aus denen die Daten bestehen. Je größer die Auftretenswahrscheinlichkeit eines Datums ist, desto weniger Informationen enthält dieses Datum. Diesem Konzept kann man sich folgendermaßen klar machen:
Wenn man eine Nachricht im Radio hört, beginnt der Journalist im normalen Fall immer mit „Guten Tag“. Dies erwartet man und deshalb ist das „unwichtig “. Wenn aber beispielsweise gesagt wird: „... in einem schweren Unfall sind viele Menschen ums Leben gekommen...“ , achtet man sehr genau darauf, um mehr darüber zu erfahren. Dies liegt daran, dass man solche Nachrichten im Allgemeinen nicht erwartet, deshalb enthalten sie viele Informationen. Dies formalisieren wir jetzt.
Sei Q eine gedächtnislose und diskrete Datenquelle über einem endlichen Alphabet $\Omega = \{a_1, a_2, \ldots, a_n\}$ mit Modell M und Wahrscheinlichkeitsverteilung $p = (p_1, p_2, \ldots, p_n)$.

1.8.1 Informationsgehalt

Der Informationsgehalt $I(a_i)$ eines Symbols a_i aus Ω ist definiert wie folgt:

$$I(a_i) = \log_2(\frac{1}{p_i}) = -\log_2(p_i)$$

Sei $s = (a_{i1} \cdots a_{ik})$ eine Sequenz über Ω mit Auftretenswahrscheinlichkeit $p(s) = \prod_{j=1}^{k} p(a_{ij})$. Der Informationsgehalt von s ist definiert wie folgt:

$$I(s) = I(a_{i1}) + I(a_{i2}) + \cdots + I(a_{ik})$$

Der Informationsgehalt wird in $[Bits]$ gemessen. Diese Größe gibt an, wieviele Bits mindestens notwendig sind, um eine Sequenz verlustfrei codieren zu können.

1.8.2 Entropie einer Datenquelle

Die Entropie von Q ist definiert als der durchschnittliche Informationsgehalt aller Symbole aus Ω. Sie wird in $[Bits/Symbol]$ gemessen. Formal ergibt sich folgende Definition der Entropie (abgekürzt mit $H(p)$).

$$H(p) = p_1 \cdot I(a_1) + \cdots + P_n \cdot I(a_n) = \sum_{i=1}^{n} p_i \cdot \log_2(\frac{1}{p_i})$$

Beispiel 3. *Gegeben sei eine Sequenz $s = Mississippi$. Sie besteht aus $n = 11$ Symbolen über dem Alphabet $\Omega = \{i, M, p, s\}$ mit den Auftretenswahrscheinlichkeiten $p(i) = \frac{4}{11}, p(M) = \frac{1}{11}, p(p) = \frac{2}{11}$, $p(s) = \frac{4}{11}$. Die einzelnen Informationsgehälter berechnen sich wie folgt:*

$$\begin{aligned} I(i) &= \log_2(\frac{11}{4}) = 1.46 \quad Bits; \\ I(M) &= \log_2(11) = 3.46 \quad Bits; \\ I(p) &= \log_2(\frac{11}{2}) = 2.46 \quad Bits; \\ I(s) &= \log_2(\frac{11}{4}) = 1.46 \quad Bits. \end{aligned}$$

Der gesamte Informationsgehalt berechnet sich jetzt wie folgt:

$$\begin{aligned} I_{ges} &= 4 \cdot I(i) + 1 \cdot I(M) + 2 \cdot I(p) + 4 \cdot I(s) \\ &= 4 \cdot 1.46\ Bits + 1 \cdot 3.46\ Bits + 2 \cdot 2.46\ Bits + 4 \cdot 1.46\ Bits \\ &= 5.84\ Bits + 3.46\ Bits + 4.92\ Bits + 5.84\ Bits \\ I_{ges} &= 20.06 \quad Bits. \end{aligned}$$

Für die Entropie gilt:

$$\begin{aligned} H(p(i), p(M), p(p), p(s)) &= \frac{I_{ges}}{11} \\ &= \frac{20.06\ Bits}{11} \\ &= 1.82 \quad Bits/Symbol \end{aligned}$$

Aus der obigen Berechnung des gesamten Informationsgehalts folgt, dass man mindestens 21 Bits benötigt, um die Sequenz „ Mississipi “ verlustfrei codieren zu können.

Sei Ω^m,$m \geq 1$, wie im Abschnitt 1.1 definiert. Ω^m kann selbst als Alphabet betrachtet werden. Die Symbole aus diesem Alphabet sind Sequenzen der Länge m über dem Alphabet Ω. Definiert man p^m als die Wahrscheinlichkeitsverteilung der Symbole aus Ω^m, dann kann man folgendes zeigen: [Say06, Abschnitt 3.2.4]

$$H(p^m) = m \cdot H(p)$$

Die Nützlichkeit von diesen theoretischen Ergebnissen aus der Informationstheorie formulieren wir im folgenden Satz. Aber vorher noch eine Definition.

1.9 Mittlere Länge eines Codes

Sei Q eine diskrete und gedächtnislose Datenquelle über dem endlichen Alphabet $\Omega = \{a_1, \ldots, a_n\}$ mit Modell M und sei $p = (p_1, \ldots, p_n)$ die Wahrscheinlichkeitsverteilung der Symbole aus Ω. Sei C ein Code über dem Alphabet $\Gamma = \{0, 1\}$. Dann ist die mittlere Länge L_C von C wie folgt definiert:

$$L_C = \sum_{j=1}^{n} p_j \cdot l_j,$$

wobei l_j die Länge von $C(a_j)$ bezeichnet.
Man kann L_C als die durchschnittliche Anzahl von Bits zur Codierung eines Symbols von Ω ansehen.

Satz 1. *Noiseless Coding theorem von Claude Shannon*
Sei Q eine diskrete und gedächtnislose Datenquelle über einem endlichen Alphabet $\Omega = \{a_1, \ldots, a_n\}$ mit Modell M und sei $p = (p_1, \ldots, p_n)$ die Wahrscheinlichkeitsverteilung der Symbole aus Ω. Seien weiterhin C ein binär präfixfreier Code und L_C die mittlere Länge von C. Dann gilt:

$$H(p) \leq L_C \qquad \textit{[Sei06, Abschnitt 1.1.2]}$$

Mit anderen Worten müssen zur verlustfreien Codierung von jedem Symbol aus Q im Durchschnitt mindestens $H(p)$ Bits benutzt werden. Die *Entropie* bildet somit die untere Grenze der verlustfreien Codierung. Nach diesem Satz kann man sagen, dass ein präfixfreier Code C optimal ist, wenn $L_C = H(p)$ gilt.

Da optimale Codes eine gute Kompressionsrate garantieren, ist also das Ziel der im Folgenden vorgestellten Codierungsverfahren einen präfixfreien Code so zu generieren, dass die Entropie der Datenquelle möglichst gut angenähert wird.

2 Das Verfahren von Huffman

In 1952 veröffentlichte David Albert Huffman ein Verfahren zur verlustfreien Codierung von Daten, das sehr schnell wegen seiner mathematischen Einfachheit beliebt wurde. Dieses Verfahren generiert für eine gegebene diskrete und gedächtnislose Datenquelle Q über einem endlichen Alphabet Ω mit Modell M und mit bekannter Wahrscheinlichkeitsverteilung p einen präfixfreien Code C. Der generierte Code heißt Huffman-Code. Eine Möglichkeit sich der Arbeitsweise des Algorithmus klar zu machen, ist das schrittweise Erzeugen des zugehörigen Huffman-Baumes. Diese Möglichkeit nutzen wir im Folgenden, um das Verfahren zu erklären. Zuerst werden wir nur die binäre Codierung betrachten, dann präsentieren wir eine Möglichkeit, Daten über einem Alphabet ungleich $\Gamma = \{0, 1\}$ zu codieren.

2.1 Huffman-Verfahren zur binären Codierung

2.1.1 Graphische Darstellung des Huffman-Verfahrens und Code-Generierung

Ausgangspunkt: Sei Q eine diskrete und gedächtnislose Datenquelle über einem endlichen Alphabet $\Omega = \{a_1, \ldots, a_n\}$ mit Modell M und mit bekannter Wahrscheinlichkeitsverteilung $p = (p_1, \ldots, p_n)$.

Der Huffman-Algorithmus basiert auf den folgenden zwei Eigenschaften eines optimalen präfixfreien Codes [Say06, Kapitel 3, Abschnitt 3.2.2]. Wobei man hier unter einem optimalen präfixfreien Code einen Präfix-Code mit minimaler mittlerer Länge verstehen sollte.

Lemma 2. *Symbole mit größeren Auftretenswahrscheinlichkeiten haben kürzere Codewörter als Symbole mit kleineren Auftretenswahrscheinlichkeiten. Formal:*

$$p_j > p_k \implies l_j \leq l_k.$$

Wobei l_i die Länge des Codewortes und p_i die Auftretenswahrscheinlichkeit von a_i bezeichnen.

Beweis. Angenommen: $p_j > p_k$ und $l_j > l_k$ und der zugehörige Code sei optimal, dann gilt:

$$(p_j - p_k) \cdot (l_j - l_k) > 0 \iff p_j \cdot l_j + p_k \cdot l_k > p_j \cdot l_k + p_k \cdot l_j.$$

Die zweite Ungleichung zeigt, dass man einen Code mit kürzerer durchschnittlicher Länge erhalten kann, wenn man die Codewörter von a_j und a_k vertauscht. Was aber ein Widerspruch ist. □

Lemma 3. *Zwei Symbole mit den kleinsten Auftretenswahrscheinlichkeiten haben Codewörter mit gleicher Länge.*

Beweis. Angenommen: C ist ein optimaler präfixfreier Code für das Alphabet $\Omega = \{a_1, \ldots, a_n\}$. O.B.d.A. seien a_1 und a_2 zwei Symbole mit den kleinsten Auftretenswahrscheinlichkeiten und l_1, l_2 die Längen der zugehörigen Codewörter und es gelte $l_1 = l_2 + k$. D.h. $C(a_1)$ ist um k Bits länger als $C(a_2)$. Da C präfixfrei ist, ist $C(a_2)$ nicht Präfix von $C(a_1)$. Wenn wir also die letzten k Bits von $C(a_1)$ entfernen, bleiben diese zwei Wörter unterschiedlich. Insbesondere kommt das um die k letzten Bits verkürzte Codewort von a_1 nicht als Präfix irgendeines anderen Codewortes vor. Wir erhalten somit einen neuen Code mit kürzerer mittlerer Länge. Dies widerspricht aber der Tatsache, dass C optimal war. □

Der im Folgenden beschriebene Algorithmus generiert einen binären Baum, in dem jeder Knoten ein Gewicht hat. Die Blätter des Baumes entsprechen den zu codierenden Symbolen des Alphabets Ω und das Gewicht eines Blattes ist gleich der Auftretenswahrscheinlichkeit des zugehörigen Symbols. Das Gewicht eines inneren Knoten ist gleich der Summe der Gewichte seiner Kinder. Das Gewicht der Wurzel ist gleich 1.

Algorithmus zur Generierung des Huffman-Baumes.

Seien $\Omega = \{a_1, \ldots, a_n\}$ und $p = (p_1, \ldots, p_n)$ wie oben definiert. Sei weiterhin L eine leere Liste.

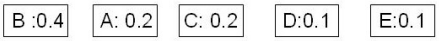
1. Erzeuge für jedes Symbol a_i des Alphabetes einen Knoten (Blatt) mit Gewicht p_i und füge den Knoten in die Liste L ein.
2. Wiederhole, so lange es mehr als einen Knoten in der Liste L gibt.
   - Sortiere die Knoten in L absteigend nach den Gewichten.
   - Wähle 2 Knoten u, v mit den kleinsten Gewichten in L aus.
   - Erzeuge einen Knoten w, dessen Gewicht die Summe der Gewichte von u und v ist.
   - Verbinde w mit u und v, sodass u und v Kinder von w werden.
   - Entferne u und v aus der Liste L und füge w in die Liste ein.

Wir illustrieren die Arbeitsweise des obigen vorgestellten Algorithmus am folgenden Beispiel.

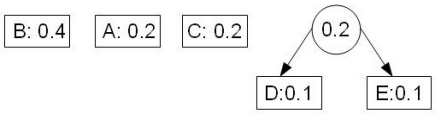
Beispiel 4. *Konstruktion des Huffman-Baumes.*

Symbol	*Auftretenswahrscheinlichkeit*
A	*0.2*
B	*0.4*
C	*0.2*
D	*0.1*
E	*0.1*

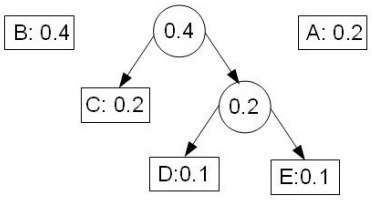
1- Initialisierung und schon sortiert



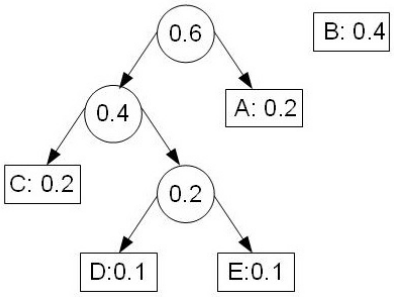
2- Nach dem ersten Schritt



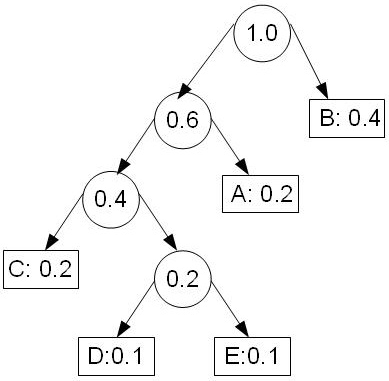
3- Nach dem zweiten Schritt und schon sortiert



4-Nach dem dritten Schritt und schon sortiert



5- Der fertige Huffman-Baum

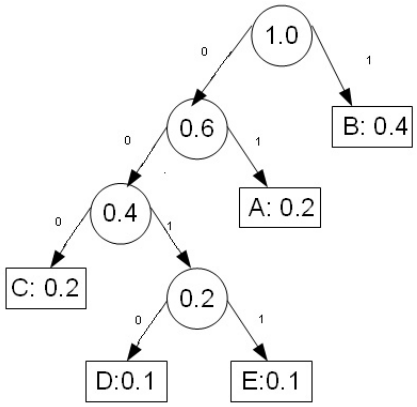


Aus dem Algorithmus und Beispiel lassen sich folgende Bemerkungen machen.

- Wenn zwei Knoten in der Liste durch ihren Vater-Knoten ersetzt werden, dann haben diese am Ende der Berechnung den gleichen Abstand zur Wurzel des Baumes
- Indem im Laufe der Berechnung zwei Knoten mit kleinsten Gewichten durch einen Knoten in der Liste ersetzt werden, sorgt der Algorithmus dafür, dass die Blätter mit kleineren Gewichten einen längeren Abstand zur Wurzel haben als die Blätter mit größeren Gewichten.

Da die Länge des Huffman-Codewortes eines Symbols proportional zum Abstand des entsprechenden Blattes zur Wurzel des Baumes ist, ist offensichtlich klar, dass Huffman-Codes die zwei oben genannten Eigenschaften eines optimalen präfixfreien Codes besitzen.

Nachdem der Huffman-Baum konstruiert wurde, lässt sich jetzt leicht die Codewörter der einzelnen Symbole des Alphabets bestimmen. Dies geschieht, indem man Kanten des Baumes mit 0 oder 1 markiert. Für jeden inneren Knoten u, markiere die linke von u ausgehende Kante mit 0 und die rechte mit 1. Das Codewort eines Symbols ergibt sich jetzt als die Folge der Kantenmarkierungen von der Wurzel bis zum entsprechenden Blatt des Baumes. Für das obige Beispiel ergibt sich folgenden markierten Baum.



Wir erhalten somit folgende Codewort-Tabelle:

Symbol	Wahrscheinlichkeit	Codewort
A	0.2	01
B	0.4	1
C	0.2	000
D	0.1	0010
E	0.1	0011

2.1.2 Analyse des Huffman-Codes

Satz 2. *Ein präfixfreier Code ist optimal, wenn seine mittlere Länge minimal ist. Der Huffman-Algorithmus generiert einen optimalen Präfix-Code. D.h. für alle präfixfreie Code C eines Alphabetes Ω mit durchschnittlicher Länge L_c und einen Huffman-Code C_H mit durchschnittlicher Länge L_H gilt:*

$$L_H \leq L_c.$$

Beweis. [Sei06, Kapitel 1, Abschnitt: Huffman Coding is optimal]
Sei eine diskrete und gedächtnislose Datenquelle Q über einem endlichen Alphabet $\Omega = \{a_1, \ldots, a_n\}$ mit Modell M und mit bekannter Wahrscheinlichkeitsverteilung $p = (p_1, \ldots, p_n)$. Wähle aus Ω zwei Symbole a_{j_1} und a_{j_2} mit den kleinsten Auftretenswahrscheinlichkeiten und ersetze diese durch $a_{(j_1,j_2)}$ mit Auftretenswahrscheinlichkeit $p_{(j_1,j_2)} = p_{j_1} + p_{j_2}$. Sei Ω' das neu gebildete Alphabet. Sei C' ein Huffman-Code für Ω' mit mittlerer Länge L' und sei $C'(a_{(j_1,j_2)})$ das Codewort von $a_{(j_1,j_2)}$. Definiere folgenden Huffman-Code C für Ω:

$$C(a_j) := \begin{cases} C'(a_j) & ,j \neq j_1 \quad \text{und} \quad j \neq j_2 \\ C'(a_{(j_1,j_2)}) \cdot 0 & ,j = j_1 \\ C'(a_{(j_1,j_2)}) \cdot 1 & ,j = j_2 \end{cases}$$

Wobei „ $\cdot$ “ hier die Konkatenation bezeichnet.

Dann gilt: Ist C' optimal, so ist C auch optimal. Denn für die Längen der einzelnen Codewörter gilt:

$$l_j = \begin{cases} l'_{(j_1,j_2)} + 1 & j = j_1 \text{ oder } j = j_2 \\ l'_j \quad \text{sonst} \end{cases}$$

Man erhält für die durchschnittliche Länge L von C folgendes:

$$\begin{aligned} L &= (\sum_{j \neq j_1, j_2} p_j \cdot l_j) + p_{j_1} \cdot l_{j_1} + p_{j_2} \cdot l_{j_2} \\ &= (\sum_{j \neq j_1, j_2} p_j \cdot l_j) + p_{j_1} \cdot (l'_{(j_1,j_2)} + 1) + p_{j_2} \cdot (l'_{(j_1,j_2)} + 1) \\ &= (\sum_{j \neq j_1, j_2} p_j \cdot l_j + \underbrace{(p_{j_1} + p_{j_2})}_{=p_{(j_1,j_2)}} \cdot l'_{(j_1,j_2)}) + p_{j_1} + p_{j_2} \\ &= L' + p_{j_1} + p_{j_2} \end{aligned}$$

Also:

$$L = L' + p_{j_1} + p_{j_2}.$$

Da aber die Auftretenswahrscheinlichkeiten p_{j_1} und p_{j_2}, die kleinsten möglichen Konstanten sind, für die die obige Gleichung gilt, gilt:

$$L' \text{ minimal } \implies L \text{ minimal.}$$

□

Nach dem Satz 1 gilt:

$$H(p) \leq L.$$

Es gilt sogar:

$$L < H(p) + 1 \quad \text{[Say06, Kapitel 3, Abschnitt 3.2.3]}$$

Diese zwei obigen Ungleichungen zeigen, dass ein Huffman-Code die Entropie einer gegebenen Datenquelle annähert. Man kann aber im Allgemeinen die Entropie besser annähert, indem Codewörter für mehr als ein Symbol generiert werden.

2.1.3 Huffman-Code für erweiterte Alphabete

Seien $\Omega = \{a_1, a_2, \ldots, a_n\}$ ein endliches Alphabet und Ω^m wie im Abschnitt 1.1 definiert. Ω^m kann selbst als Alphabet aufgefasst werden und sobald die Auftretenswahrscheinlichkeiten von den einzelnen a_i bekannt sind, kann man auch die Auftretenswahrscheinlichkeiten der Symbole aus Ω^m berechnen. Somit kann der Huffman-Algorithmus zur Codierung der Symbole aus Ω^m eingesetzt werden. Wenn wir mit p^m die Wahrscheinlichkeitsverteilung der Symbole aus Ω^m bezeichnen, dann gilt nach dem vorherigen Abschnitt:

$$H(p^m) \leq L_m < H(p^m) + 1.$$

Wobei L_m die durchschnittliche Länge eines Huffman-Codes für das Alphabet Ω^m bezeichnet.
Es gilt:

$$L = \frac{L_m}{m},$$

mit $L :=$ durchschnittliche Länge eines Huffman-Codes für Ω.
Es folgt:

$$\frac{H(p^m)}{m} \leq L < \frac{H(p^m)}{m} + \frac{1}{m}.$$

Nach Abschnitt 1.8.2 gilt: $H(p^m) = m \cdot H(p)$.
Wir erhalten somit:

$$H(p) \leq L < H(p) + \frac{1}{m}.$$

Die obigen Ungleichungen zeigen also, dass man die Entropie besser annähert, wenn man Codewörter für Blöcke von Symbolen fester Länge erzeugt, was sich positiv auf die Kompressionsrate auswirkt. Im Folgenden beschäftigen wir uns mit der nicht binären Codierung mithilfe des Huffman-Verfahrens.

2.2 Huffman-Verfahren zur nicht binären Codierung

Sei eine diskrete und gedächtnislose Datenquelle Q über einem endlichen Alphabet $\Omega = \{a_1, \ldots, a_n\}$ mit Modell M und mit bekannter Wahrscheinlichkeitsverteilung $p = (p_1, \ldots, p_n)$. Die Sequenzen aus Q sollen mithilfe des Alphabetes $\Gamma = \{d_1, \ldots, d_k\}$,$k > 2$ codiert werden. Der Algorithmus zur Generierung eines Huffman-Baumes ist folgender.
Hier nehmen wir an , dass $n > k$ gilt (für n $\leq k$ ist die Codierung trivial).

Algorithmus:

Sei L eine leere Liste.

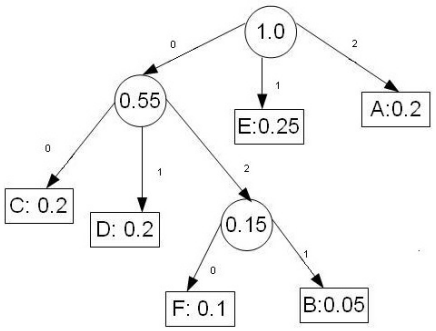
1. Für jedes Symbol a_i aus Ω generiert ein Blatt mit Gewicht p_i und füge dieses in die Liste ein.
   Sortiere die Liste absteigend nach den Gewichten.
2. Finde $m \in \mathbb{N}$; $m \geq 2$ und $m \leq k$ so, dass $\frac{(n-m)}{(k-1)}$ eine positive Ganzzahl ist.
   - Wähle m Blätter aus L mit kleinsten Gewichten
   - Erzeuge einen neuen Knoten u , dessen Gewicht die Summe der Gewichte der m gewählten Blätter ist.
   - Verbinde u so mit diesen m Blättern, dass diese seine Kinder werden
   - Entferne die m Blätter aus der Liste.
   - Füge u in die Liste ein.
3. Wiederhole, so lange L mehr als einen Knoten enthält.
   - Sortiere die Liste absteigend nach den Gewichten.
   - Wähle k Knoten aus L mit kleinsten Gewichten
   - Erzeuge einen neuen Knoten v, dessen Gewicht die Summe der Gewichte der k gewählten Knoten ist.
   - Verbinde v mit diesen k Knoten so, dass diese seine Kinder werden.
   - Entferne diese k Knoten aus der Liste L und füge v in die Liste ein.

Wir illustrieren dies an einem Beispiel.

Beispiel 5. *Die Sequenzen aus einer Datenquelle sollen mithilfe eines ternären Alphabetes* $\Gamma = \{0, 1, 2\}$ *codiert werden. Das Alphabet der Quelle und die Auftretenswahrscheinlichkeiten der Symbole ist durch folgende Tabelle gegeben.*

Symbol	*Wahrscheinlichkeit*
A	*0.2*
B	*0.05*
C	*0.2*
D	*0.2*
E	*0.25*
F	*0.1*

Es gilt hier: $n = 6$ *und* $k = 3$*, folglich* $m = 2$*.*
Wie bei der binären Codierung markieren wir die Kanten des Baumes, um die einzelnen Codewörter zu erhalten. Der markierte Baum am Ende der Berechnung ist folgender:



Wir erhalten somit folgende Codetabelle

Symbol	*Wahrscheinlichkeit*	*Codewort*
A	*0.2*	*2*
B	*0.05*	*021*
C	*0.2*	*00*
D	*0.2*	*01*
E	*0.25*	*1*
F	*0.1*	*020*

3 Arithmetische Codierung

Wir haben im Abschnitt 2.1.3 gesehen, dass man die Entropie einer Datenquelle besser annähert, wenn man Codewörter für Blöcke von Symbolen generiert. Je länger die Blöcke sind, desto besser ist die Annäherung. Ist man aber an dem Huffman-Code einer festen Sequenz interessiert, erweist sich diese Vorgehensweise als unpraktisch. Betrachte beispielsweise eine Datenquelle über einem endlichen Alphabet mit 10 Symbolen. Um mit dem Huffman-Verfahren eine Sequenz der Länge 10 zu codieren, müssen die Codewörter aller Sequenzen dieser Länge generiert werden. Es gibt aber $10^{10} = 10000000000$ Sequenzen der Länge 10 über diesem Alphabet. Der Aufwand zur Verwaltung einer solchen Code-Tabelle kann nicht immer geleistet werden. Diese Vorgehensweise hat also den Nachteil, dass die Codewörter aller möglichen Sequenzen generiert werden müssen. Außerdem wächst die Größe der Code-Tabelle exponentiell mit der Länge der zu codierenden Sequenz. Eine Lösung für dieses Problem bietet die arithmetische Codierung. Die arithmetische Codierung ist wie die Huffman-Codierung verlustfrei mit dem Unterschied, dass hier ein einziges Codewort für eine ganze Sequenz generiert wird. Im Folgenden präsentieren wir zuerst das Prinzip der arithmetischen Codierung und Decodierung. Anschließend werden wir eine Analyse dieses Codierungsverfahrens durchführen und schließlich beschäftigen wir uns mit den Skalierungsfunktionen.

3.1 Codierung als reelle Zahl

Ausgangspunkt: Sei eine diskrete und gedächtnislose Datenquelle über dem Alphabet $\Omega = \{a_1, \ldots, a_n\}$ mit Modell M und mit Wahrscheinlichkeitsverteilung $p = (p_1, \ldots, p_n)$.
Wir definieren zuerst folgende empirische Verteilungsfunktion F.

$$\begin{aligned} F : \{0, 1, \ldots, n\} &\rightarrow [0, 1] \\ F(0) &= 0 \\ F(i) &= \sum_{j=1}^{i} p_j, \text{ für i} \neq 0 \end{aligned}$$

Die oben definierte Funktion F hat folgende Eigenschaften:

$$F(0) = 0$$

$$F(n) = 1$$

$$\nexists i \in \{1,\ldots,n\} : p_i = 0 \implies \forall i,j \in \{0,1,\ldots,n\}, j \neq i : F(i) \neq F(j).$$

Die grundlegende Idee der arithmetischen Codierung ist, jede Sequenz $s \in \Omega^*$ auf ein Intervall $[l_s, u_s)$ Teilintervall von [0, 1) abzubilden, das diese Sequenz unter allen Sequenzen mit gleicher Länge auf eindeutige Weise repräsentiert. Da das Intervall $[l_s, u_s)$ die Sequenz s eindeutig repräsentiert, kann prinzipiell eine beliebige Zahl $T(s) \in [l_s, u_s)$ als Code von s verwendet werden.

Zum Start des Codierungsvorgangs setzt man $l_s = 0$ und $u_s = 1$. Dieses Intervall wird nun nach dem folgenden Schema in Teilintervalle partitioniert. Für jedes $a_i \in \Omega$ gibt es ein Teilintervall $[F(i-1), F(i))$. Nach den oben beschriebenen Eigenschaften von F , erhalten wir somit eine Partition von [0, 1). Jetzt identifiziert man Teilintervalle mit Symbolen aus Ω. Beispielsweise repräsentiert das Teilintervall $[0, F(1))$ das Symbol a_1. Abhängig jetzt vom ersten Symbol der Sequenz werden die Grenzen l_s und u_s aktualisiert. Sei beispielsweise a_j das erste Symbol einer Sequenz s. Dann wird a_j durch das Teilintervall $[F(j-1), F(j))$ repräsentiert. Man setzt $l_s = F(j-1)$ und $u_s = F(j)$. Dieses neue Intervall $[l_s, u_s)$ wird selbst in Teilintervalle gemäß folgender Regel partitioniert:
Für jedes $a_i \in \Omega$ gibt es das Teilintervall $[l_s + (u_s - l_s)F(i-1),\ l_s + (u_s - l_s)F(i))$. Wiederum identifiziert man Symbole mit Teilintervallen und man betrachtet das nächste Symbol der Sequenz und aktualisiert die Grenzen l_s, u_s entsprechend. Dieses Verfahren wird so lange wiederholt, bis die gesamte Sequenz betrachtet wurde. Am Ende erhält man ein Intervall $[l_s, u_s)$, das die Sequenz auf eindeutige Weise identifiziert. Hier wählen wir $T(s)$ als den Mittelpunkt dieses End-Intervalls, also: $T(s) = \frac{l_s + u_s}{2}$. Dieses Codierungsverfahren erklären wir am besten mit einem Beispiel.

Beispiel 6. *Arithmetische Codierung.*
Sei eine diskrete und gedächtnislose Datenquelle über dem Alphabet $\Omega = \{a, b, c\}$ mit Modell M und mit den Auftretenswahrscheinlichkeiten: $p(a) = 0.7; p(b) = 0.1; p(c) = 0.2$. Die Funktionswerte von F sind folgende: $F(0) = 0$; $F(1) = 0.7$; $F(2) = 0.8$; $F(3) = 1$. Die zu codierende Sequenz ist $s = abc$.

1. *Initialisierung: $l_s = 0$ und $u_s = 1$*
   Die Teilintervalle sind : $a = [0, 0.7)$; $b = [0.7, 0.8)$; $c = [0.8, 1)$

2. *Einlesen vom ersten Symbol der Sequenz: a*
   UPDATE: $l_s = 0$, und $u_s = 0.7$ also: $[l_s, u_s) = [0, 0.7)$. $[0, 0.7)$ wird partitioniert wie folgt:

   - *für a erhalten wir das Teilintervall:*

   $$[0 + (0.7 - 0)F(0);\ 0 + (0.7 - 0)F(1))$$

   $$[0 + (0.7)(0);\ 0 + (0.7)(0.7)) = [0; 0.49)$$

   - *für b erhalten wir das Teilintervall:*

   $$[0 + (0.7 - 0)F(1);\ 0 + (0.7 - 0)F(2))$$

   $$[0 + (0.7)(0.7);\ 0 + (0.7)(0.8)) = [0.49; 0,56)$$

   - *für c erhalten wir das Teilintervall:*

   $$[0 + (0.7 - 0)F(2);\ 0 + (0.7 - 0)F(3))$$

   $$[0 + (0.7)(0.8);\ 0 + (0.7)(1)) = [0.56; 0.7)$$

3. *Einlesen vom zweiten Symbol der Sequenz : b*
   UPDATE : $l_s = 0.49$ und $u_s = 0.56$, also $[l_s, u_s) = [0.49, 0.56)$. $[0.49, 0.56)$ wird wiederum partitioniert.

   - *für a erhalten wir das Teilintervall* $[0.49; 0.539)$
   - *für b erhalten wir das Teilintervall* $[0.539; 0.546)$
   - *für c erhalten wir das Teilintervall* $[0.546; 0.56)$

4. *Einlesen vom dritten Symbol der Sequenz: c*
   UPDATE : $l_s = 0.546$ *und* $u_s = 0.56$.
   Wir erhalten am Ende das Intervall $[l_s, u_s) = [0.546, 0.56)$. *Ein möglicher Code von der Sequenz* $s = abc$ *wäre :*

$$T(s) = \frac{0.546 + 0.56}{2} = 0.553.$$

Zu bemerken ist, dass im obigen Beispiel das Intervall $[0.546, 0.56)$ die Sequenz abc unter allen Sequenzen der Länge 3 eindeutig identifiziert. Eine mit dem Symbol b oder c beginnende Sequenz kann nie zum obigen End-Intervall führen. Wir erhalten somit folgenden Codierungsalgorithmus.

Algorithmus: Arithmetische Codierung.
Eingabe: Sequenz $s = a_{j_1} \cdots a_{j_k}$ über dem Alphabet $\Omega = \{a_1, \ldots, a_n\}$ mit Länge k.
Empirische Verteilungsfunktion $F : \{0, 1, \ldots, n\} \to [0, 1]$

$l_s = 0$;
$u_s = 1$;
For i = 1 To k Do
betrachte das i-te Zeichen der Sequenz a_{j_i}
$l_s = l_s + (u_s - l_s)F(j_i - 1)$;
$u_s = l_s + (u_s - l_s)F(j_i)$;
End
$T(s) = (l_s + u_s)/2$;
Ausgabe : T(s)

Um den binären arithmetischen Code einer Sequenz s zu erhalten, definieren wir folgende Größe $l(s)$.

$$l(s) = \lceil \log_2(\frac{1}{p(s)}) \rceil + 1$$

Für eine Zahl x bezeichnet $\lfloor bin(x) \rfloor_y$ die ersten y Bits der binären Darstellung von x. Den binären Code $C(s)$ einer Sequenz s erhalten wir durch folgende Vorschrift:

$$C(s) = \lfloor bin(T(s)) \rfloor_{l(s)}.$$

Beispiel 7. *Binärer arithmetischer Code einer Sequenz.*
Für die Sequenz $s = abc$ *hatten wir* $T(s) = 0.553$ *als Code berechnet.*
Es gilt: $p(s) = p(a) \cdot p(b) \cdot p(c) = 0.014$

$$l(s) = \lceil \log_2(\frac{1}{p(s)}) \rceil + 1 = \lceil \log_2(\frac{1}{0.014}) \rceil + 1 = 8 \tag{1}$$

$$T(s) = (0.553)_{10} = (0.10001101100100010110100001110 01...)_2 \tag{2}$$

$$C(s) = \lfloor bin(0.553) \rfloor_8 = 10001101.$$

$C(s) = 10001101$ *ist der binäre arithmetische Code von* s.

3.2 Decodierung einer reellen Zahl

Ausgangspunkt: Eine diskrete und gedächtnislose Datenquelle über dem Alphabet $\Omega = \{a_1, \ldots, a_n\}$ mit Modell M und mit Wahrscheinlichkeitsverteilung $p = (p_1, \ldots, p_n)$. Weiterhin sei F die empirische Verteilungsfunktion wie im vorherigen Abschnitt definiert und sei s eine Sequenz über Ω mit Codewort $T(s)$. Ziel ist es, die Sequenz s zu rekonstruieren.

Zuerst bemerken wir: Für zwei Teilintervalle $[a, b)$ und $[c, d)$, die im Laufe eines arithmetischen Codierungsvorgangs berechnet werden, gilt: $[a, b) \subseteq [c, d)$ oder $[a, b) \cap [c, d) = \emptyset$. Weiterhin gilt: Sei $[l_s, u_s)$ das Intervall, mithilfe dessen Grenzen $T(s)$ berechnet wurde, dann gilt: falls $T(s) < a$ bzw. $T(s) > b$, dann muss sich das gesamte Intervall $[l_s, u_s)$ links bzw. rechts vom Intervall [a, b) auf der Zahlengerade befinden. Auf diesen Eigenschaften basiert die Decodierung eines arithmetischen

Codes. Die Technik hier besteht einfach darin, den arithmetischen Codierer zu imitieren. Dies erklären wir am besten mit einem Beispiel.

Beispiel 8. *Decodierung eines arithmetischen Codes.*
Wir versuchen hier das im vorherigen Abschnitt erhaltene Codewort $T(s) = 0.553$ zu decodieren.

1. *Initialisierung*
   zum Start gilt: $l_s = 0$ und $u_s = 1$ und $s =$ (leerer String).
   wir hatten folgende Teilintervalle berechnet:

   - *Für a:* $[0, 0.7)$
   - *Für b :* $[0.7, 0.8)$
   - *Für c:* $[0.8, 1)$

   Da $T(s) = 0.553 \in [0, 0.7)$ gilt, setzen wir die Grenzen wie folgt: $l_s = 0$ und $u_s = 0.7$ und wir setzen $s = a$.

2. *Für das Intervall $[l_s, u_s) = [0, 0.7)$ hatten wir folgende Partition:*

   - *Für a :*$[0, 0.49)$
   - *Für b :*$[0.49, 0.56)$
   - *Für c :*$[0.56, 0.7)$

   Da gilt: $T(s) = 0.553 \in [0.49, 0.56)$, aktualisieren wir: $l_s = 0.49$, $u_s = 0.56$ und $s = ab$

3. *Für das Intervall $[l_s, u_s) = [0.49, 0.56)$ hatten wir folgende Partition:*

   - *Für a :*$[0.49, 0.539)$
   - *Für b :*$[0.539, 0.546)$
   - *Für c:*$[0.546, 0.56)$

   Es gilt : $T(s) = 0.553 \in [0.546, 0.56)$, somit erhalten wir die Sequenz $s = abc$.

Wir können also den Decodierungsalgorithmus wie folgt zusammenfassen:

Algorithmus: Decodierung eines arithmetischen Codes
Eingabe: Codewort $T(s)$ einer Sequenz s der Länge k

$l_s = 0$; $u_s = 1$; $s =$;
For i = 1 To k Do
finde das Zeichen a_{j_i} so, dass für das zugehörige Teilintervall [a, b) $\subseteq [l_s, u_s)$gilt:
$T(s) \in [a, b)$.
$s = s \cdot a_{j_i}$;
$l_s = a$;
$u_s = b$;
End

Ausgabe :s

3.3 Terminierung

Im obigen Algorithmus zur Decodierung eines arithmetischen Codes sind wir davon ausgegangen, dass der Decodierer die Länge der codierten Sequenz kennt. Das ist tatsächlich eine Möglichkeit den Decodierungsvorgang zum Ende zu bringen. Die andere Möglichkeit besteht darin, ein Symbol aus dem Quellenalphabet mit kleiner Auftretenswahrscheinlichkeit zu wählen, das nur dazu dient, das Ende einer Sequenz zu kennzeichnen. Die Decodierung von diesem Symbol bringt somit den Decodierungsvorgang zu Ende.

3.4 Eindeutigkeit und Effizienz

Ausgangspunkt: Sei eine diskrete und gedächtnislose Datenquelle über dem Alphabet $\Omega = \{a_1, \ldots, a_n\}$ mit Modell M und mit Wahrscheinlichkeitsverteilung $p = (p_1, \ldots, p_n)$. S sei eine Sequenz der Länge m über Ω. S kann als Symbol des erweiterten Alphabetes Ω^m aufgefasst werden. Für Ω^m definieren wir die empirische Verteilungsfunktion F_m wie folgt:

$$F_m : \{0, 1, \ldots, n^m\} \to [0, 1]$$

$$F_m(i) = \begin{cases} 0 & \text{wenn } i = 0 \\ \sum_{j=1}^{i} p_m(j) & \text{sonst} \end{cases}$$

Mit dieser Funktion erhalten wir wie im Abschnitt 3.1 für jedes Symbol $b_i \in \Omega^m$ ein Teilintervall $[F_m(i-1), F_m(i))$, das b_i eindeutig identifiziert.
Es ist zu bemerken: $F_m(i) - F_m(i-1) = p_m(i) = p_m(b_i)$.
Der Code von b_i war definiert als

$$T(b_i) = \frac{F_m(i-1) + F_m(i)}{2},$$

oder auch

$$T(b_i) = F_m(i-1) + \frac{p_m(b_i)}{2}. \qquad (*)$$

Sei $l(b_i) = \lceil \log_2(\frac{1}{p_m(b_i)}) \rceil + 1$ vgl. Abschnitt 3.1. Der binäre Code von b_i ist die binäre Darstellung von der folgenden Zahl $C(b_i)$:

$$C(b_i) = \lfloor T(b_i) \rfloor_{l(b_i)}.$$

Die im Folgenden aufgeführten Beweise kann man in [Say06, Kapitel 4, Abschnitt 4.4.1] nachlesen.

3.4.1 Eindeutigkeit des arithmetischen Codes

In diesem Teil zeigen wir, dass ein Codewort $C(b_i)$ eine Sequenz b_i eindeutig identifiziert und eindeutig decodierbar ist.

Da das Teilintervall $[F_m(i-1), F_m(i))$ die Sequenz b_i schon eindeutig identifiziert, genügt es also zu zeigen, dass $C(b_i)$ in diesem Teilintervall liegt, um die Eindeutigkeit nachzuweisen. Es gilt per Konstruktion:

$$\begin{array}{rrcll} & C(b_i) & = & \lfloor T(b_i) \rfloor_{l(b_i)} \leq T(b_i) < F_m(i). & \\ \text{Also :} & C(b_i) & < & F_m(i) & (1) \end{array}$$

Außerdem gilt:

$$0 \leq T(b_i) - C(b_i) < \frac{1}{2^{l(b_i)}}$$

Betrachte:

$$\begin{aligned} \frac{1}{2^{l(b_i)}} &= \frac{1}{2^{\lceil \log_2(\frac{1}{p_m(b_i)}) \rceil + 1}} \\ &\leq \frac{1}{2^{\log_2(\frac{1}{p_m(b_i)}) + 1}} \\ &= \frac{1}{2 \cdot \frac{1}{p_m(b_i)}} \\ &= \frac{p_m(b_i)}{2}. \end{aligned}$$

Es gilt nach $(*)$:

$$\frac{p_m(b_i)}{2} = T(b_i) - F_m(i-1).$$

Daraus folgt:

$$0 \leq T(b_i) - C(b_i) < T(b_i) - F_m(i-1)$$

also :

$$C(b_i) > F_m(i-1). \qquad (2)$$

Somit erhalten wir aus (1) und (2) folgende Ungleichungen :

$$F_m(i-1) < C(b_i) < F_m(i). \qquad (3)$$

Die Ungleichungen (3) zeigen also, dass der Code $C(b_i)$ die Sequenz b_i eindeutig identifiziert. Um die eindeutige Decodierbarkeit eines arithmetischen Codes zu zeigen, zeigen wir, dass er präfixfrei ist, dann folgt die eindeutige Decodierbarkeit.

Sei eine Zahl $x \in [0,1)$ mit binärer Darstellung $bin(x) = 0.x_1 \cdots x_n$. Damit eine andere Zahl y eine binäre Darstellung mit dem Präfix $[x_1 \cdots x_n]$ besitzt, muss y im Intervall $[x, x + \frac{1}{2^n})$ liegen. Wenn b_i und b_j zwei unterschiedliche Sequenzen sind, dann liegen $\lfloor T(b_i) \rfloor_{l(b_i)}$ und $\lfloor T(b_j) \rfloor_{l(b_j)}$ in zwei disjunkten Intervallen. Nämlich : $[F_m(i-1), F_m(i))$ und $[F_m(j-1), F_m(j))$. Wenn wir also zeigen, dass für eine beliebige Sequenz b_i das Intervall $[\lfloor T(b_i) \rfloor_{l(b_i)}, \lfloor T(b_i) \rfloor_{l(b_i)} + \frac{1}{2^{l(b_i)}})$ Teilintervall von $[F_m(i-1), F_m(i))$ ist, dann ist die Präfixfreiheit garantiert.

Nach (2) gilt : $\lfloor T(b_i) \rfloor_{l(b_i)} > F_m(i-1)$.
Wir zeigen:

$$\begin{aligned} F_m(i) &> \lfloor T(b_i) \rfloor_{l(b_i)} + \frac{1}{2^{l(b_i)}} \quad \Longleftrightarrow \\ F_m(i) - \lfloor T(b_i) \rfloor_{l(b_i)} &> \frac{1}{2^{l(b_i)}} \end{aligned}$$

Betrachte:

$$\begin{aligned} F_m(i) - \lfloor T(b_i) \rfloor_{l(b_i)} &> F_m(i) - T(b_i) \\ \text{nach} \quad (*) \quad &= F_m(i) - F_m(i-1) - \frac{p_m(i)}{2} \\ &= p_m(i) - \frac{p_m(i)}{2} \\ &= \frac{p_m(i)}{2} \geq \frac{1}{2^{l(b_i)}} \\ &> \frac{1}{2^{l(b_i)}} \end{aligned}$$

Somit folgt, dass ein arithmetischer Code ein Präfix-Code ist. Dies impliziert die eindeutige Decodierbarkeit.

3.4.2 Effizienz der arithmetischen Codierung

In diesem Abschnitt betrachten wir den Zusammenhang zwischen der mittleren Länge eines arithmetischen Codes und der Entropie der zugehörigen Datenquelle.

$l(b_i) = \lceil \log_2(\frac{1}{p(b_i)}) \rceil + 1$ ist die Länge eines binären arithmetischen Codewortes der Sequenz b_i. Die mittlere Länge eines arithmetischen Codes für Sequenzen der Länge m ist durch folgende Vorschrift gegeben:

$$
\begin{aligned}
L_{\Omega^m} &= \sum_{x\in\Omega^m} p(x)\cdot l(x) \\
&= \sum_{x\in\Omega^m} p(x)\cdot(\lceil\log_2(\frac{1}{p(x)})\rceil+1) \\
&< \sum_{x\in\Omega^m} p(x)\cdot(\log_2(\frac{1}{p(x)})+1+1) \\
&= (\sum_{x\in\Omega^m} p(x)\cdot\log_2(\frac{1}{p(x)}))+2\underbrace{\sum_{x\in\Omega^m} p(x)}_{=1} \\
&= H(p^m)+2.
\end{aligned}
$$

Da wir inzwischen wissen, dass gilt :

$$H(p^m) \leq L_{\Omega^m},$$

und da

$$L_\Omega = \frac{L_{\Omega^m}}{m}$$

erhalten wir folgende Schranken für die mittlere Länge eines arithmetischen Codes für das Alphabet Ω (abgekürzt hier mit L_Ω):

$$\frac{H(p^m)}{m} \leq L_\Omega < \frac{H(p^m)}{m} + \frac{2}{m}$$

Da weiterhin gilt:

$$H(p^m) = m\cdot H(p)$$

folgt schließlich:

$$H(p) \leq L_\Omega < H(p) + \frac{2}{m}$$

Die obigen Ungleichungen zeigen, dass die arithmetische Codierung asymptotisch ein optimales Codierungsverfahren ist.

3.5 Skalierungsfunktionen

Dieser Teil basiert im Wesentlichen auf [BCK02]

Wir haben die grundlegende Idee der arithmetischen Codierung kennengelernt. Die vorgestellten Algorithmen werden aber in der Praxis nicht so implementiert. Beim Codierungsvorgang wird das Intervall $[l_s, u_s)$ zum Start auf $[0, 1)$ gesetzt. Dieses wird im Laufe der Codierung immer kleiner. Da aber das Intervall $[0, 1)$ auf einer festen Maschine nur aus endlich vielen Zahlen besteht, konvergieren für genügend lange Sequenzen die Grenzen l_s und u_s gegen einen gleichen Wert. Ab diesem Wert wird die Codierung nicht mehr möglich sein. Es folgt also: je länger die zu codierenden Sequenzen sind, desto mehr Genauigkeit ist gefordert. Rechner haben aber eine feste und endliche Genauigkeit. Ein weiteres Problem ist, dass der Decodierungsvorgang erst gestartet werden kann, wenn der Codierungsvorgang zu Ende ist. Dies ist aber im Allgemeinen nicht praktisch. Um diese zwei Probleme zu lösen, benutzt man in der Praxis Skalierungsfunktionen. Für genügend kleines Intervall $[l_s, u_s)$ Teilintervall von $[0, 1)$ unterscheidet man drei Fälle:

- $[l_s, u_s)$ ist Teilintervall von $[0, 0.5)$
- $[l_s, u_s)$ ist Teilintervall von $[0.5, 1)$
- $[l_s, u_s)$ ist Teilintervall von $[0.25, 0.75)$

3.5.1 Die Skalierungsfunktionen E_1 und E_2

Falls im Laufe der Codierung das Intervall $[l_s, u_s)$ Teilintervall von $[0, 0.5)$ oder von $[0.5, 1)$ ist, wird es für immer in diesem Teilintervall bleiben. Die Intervalle $[0, 0.5)$ und $[0.5, 1)$ haben aber die Eigenschaft, dass das höchstwertige Bit der binären Darstellung von jeder Zahl aus diesen den gleichen Wert hat. Wenn $x \in [0, 0.5)$ ist, dann gilt: $bin(x) = 0.0*$ und für $x \in [0.5, 1)$ gilt: $bin(x) = 0.1*$. Wobei $bin(x)$ die binäre Darstellung der Zahl x und $*$ die anderen Bits der binären Darstellung bezeichnen. Insbesondere ist auch das höchstwertige Bit des binären arithmetischen Codes einer Sequenz schon bestimmt, falls ein dieser Zustände eintritt. Dieses Bit kann also schon zum Decodierer geschickt werden, ohne irgendeine Information zu verlieren. Dies erlaubt also eine inkrementelle Codierung bzw. Decodierung. Um das Problem der Genauigkeit zu lösen, vergrößert man anschließend das aktuelle Intervall $[l_s, u_s)$, indem man abhängig vom konkreten Fall eine der folgenden Funktionen auf die Grenzen des Intervalls anwendet.

$$E_1 : [0, 0.5) \to [0, 1), E_1(x) = 2 \cdot x$$

$$E_2 : [0.5, 1) \to [0, 1), E_2(x) = 2 \cdot (x - 0.5)$$

3.5.2 Die Skalierungsfunktion E_3

Im Fall jetzt, dass $[l_s, u_s)$ Teilintervall von $[0.25, 0.75)$ ist, verwendet man die E_3- Skalierung, die wie folgt definiert ist:

$$E_3 : [0.25, 0.75) \to [0, 1); E_3(x) = 2 \cdot (x - 0.25).$$

Um die Decodierung einer Sequenz zu ermöglichen, muss natürlich dem Decodierer mitgeteilt werden, dass eine E_3-Skalierung auf der Seite des Codierers verwendet wurde. Um zu verstehen, wie das gemacht wird, betrachten wir zuerst dieses Beispiel:

Beispiel 9. *Sei* $[l_s, u_s) = [0.3; 0.45)$. *Es gilt:*$[0.3, 0.45) \subseteq [0.25, 0.75)$. *Nach Anwenden der* E_3*-Skalierung auf die Grenzen des Intervalls erhalten wir folgendes Intervall* $[0.1, 0.4)$. *Dieses Intervall ist Teilintervall von* $[0, 0.5)$. *Nach Anwenden einer* E_1*-Skalierung erhalten wir:* $[0.2, 0.8)$. *Da* $[l_s, u_s) = [0.3, 0.45)$ *auch Teilintervall von* $[0, 0.5)$ *ist, erhalten wir nach Anwenden der* E_1*-Skalierung das Intervall* $[0.6, 0.9)$. *Nach Anwenden der* E_2*-Skalierung auf* $[0.6, 0.9)$ *erhalten wir das Intervall* $[0.2, 0.8)$.

Wir sehen also:

$$E_1 \circ E_3 = E_2 \circ E_1.$$

Analog gilt:

$$E_2 \circ E_3 = E_1 \circ E_2$$

Im Allgemeinen gilt:

$$E_1 \circ (E_3)^n = (E_2)^n \circ E_1;$$

$$E_2 \circ (E_3)^n = (E_1)^n \circ E_2.$$

Mit $f \circ g$ bezeichnen wir die Verkettung der Funktionen f und g.
Um jetzt den Decodierer über die Anwendung der E_3-Skalierung zu informieren, speichert man die Anzahl x der E_3-Skalierungen, die seit der letzten E_1- oder E_2-Skalierung stattgefunden haben. Sobald die erste E_1- oder E_2-Skalierung stattfindet, wird nach dem entsprechenden geschickten Bit x-mal das Inverse davon geschickt.

Beispiel 10. *Codierung mit Skalierung*
Hier versuchen wir die gleiche Sequenz $s = abc$ *wie im Abschnitt 3.1 mithilfe der Skalierungsfunktionen zu codieren. Sei* $C(s)$ *der binäre arithmetische Code von* s *und* x *die Anzahl der aktuellen* E_3*-Skalierungen.*

1. *Initialisierung:* $l_s = 0$ *und* $u_s = 1$ *und* $C(s) =$ *(leerer String) und* $x = 0$

2. *Einlesen vom ersten Symbol der Sequenz: a*
   UPDATE : $l_s = 0$ *und* $u_s = 0.7$ *(vgl. Abschnitt 3.1).*
   Hier kann man die Skalierungsfunktionen nicht anwenden.

3. *Einlesen vom zweiten Symbol der Sequenz: b*
   UPDATE: $l_s = 0.49$ *und* $u_s = 0.56$ *(vgl. Abschnitt 3.1)*
   Es gilt : [0.49, 0.56) $\subseteq$ *[0.25, 0.75)*

   - *Erste* E_3*-Skalierung: Wir setzen* $x = 1$

   $$\begin{aligned} E_3(0.49) &= 2 \cdot (0.49 - 0.25) = 0.48 \\ E_3(0.56) &= 2 \cdot (0.56 - 0.25) = 0.62 \end{aligned}$$

   wir erhalten $[l_s, u_s) = [0.48, 0.62) \subseteq [0.25, 0.75)$
   - *Zweite* E_3*-Skalierung: Wir setzen* $x = 2$

   $$\begin{aligned} E_3(0.48) &= 2 \cdot (0.48 - 0.25) = 0.46 \\ E_3(0.62) &= 2 \cdot (0.62 - 0.25) = 0.74 \end{aligned}$$

   Wir erhalten $[l_s, u_s) = [0.46, 0.74) \subseteq [0.25, 0.75)$
   - *Dritte* E_3*-Skalierung: Wir setzen* $x = 3$

   $$\begin{aligned} E_3(0.46) &= 2 \cdot (0.46 - 0.25) = 0.42 \\ E_3(0.74) &= 2 \cdot (0.74 - 0.25) = 0.98 \end{aligned}$$

   Wir erhalten $[l_s, u_s) = [0.42, 0.98)$.

   Hier kann man keine Skalierungsfunktion anwenden.

4. *Einlesen vom letzten Zeichen der Sequenz: c*
   Wir erhalten folgendes Intervall:

   $$\begin{aligned} [l_s, u_s) &= [0.42 + (0.98 - 0.42) \cdot F(2); 0.42 + (0.98 - 0.42) \cdot F(3)) (vgl. Abschnitt\ 3.1) \\ &= [0.868, 0.98) \end{aligned}$$

   Es gilt: [0.868 ; 0.98) $\subseteq$ *[0.5 ,1)*

5. *Anwenden der* E_2*-Skalierung: Wir setzen* $C(s) = 1000$ *und* $x = 0$

   $$\begin{aligned} E_2(0.868) &= 2 \cdot (0.868 - 0.5) = 0.736 \\ E_2(0.98) &= 2 \cdot (0.98 - 0.5) = 0.96 \end{aligned}$$

   Es gilt: $[l_s, u_s) = [0.736, 0.96) \subseteq [0.5, 1)$

6. *Anwenden der* E_2*-Skalierung: Wir setzen* $C(s) = 10001$

   $$\begin{aligned} E_2(0.736) &= 0.472 \\ E_2(0.96) &= 0.92 \end{aligned}$$

Wir erhalten schließlich : $[l_s, u_s) = [0.472, 0.92)$. *Bis jetzt gilt:* $C(s) = 10001$.
Der Codierer geht jetzt folgendermaßen vor:
Er berechnet zuerst folgende Größe

$$p = u_s - l_s = 0.92 - 0.472 = 0.448.$$

Danach berechnet er die Größe

$$l = \lceil \log_2(\frac{1}{p}) \rceil + 1 = \lceil \log_2(\frac{1}{0.448}) \rceil + 1 = 3 \quad \textit{(vgl. mit } l(s) \textit{ im Abschnitt 3.1)}$$

Jetzt berechnet er den Mittelpunkt des obigen Intervalls

$$T = \frac{0.472 + 0.92}{2} = 0.696 \quad \textit{(vgl. mit } T(s) \textit{ im Abschnitt 3.1)}$$

Zum Schluss nimmt er die ersten l *Bits(hier 3) der binären Darstellung von* $T(= 0.1011001000\cdots)$ *und fügt diese am Ende von* $C(s)$. *Wir erhalten schließlich als Code von* $s = abc$ *folgende Bitsequenz* $C(s) = 10001101$. *Diese Bitsequenz stimmt mit dem im Abschnitt 3.1 erhaltenen Codewort überein.*

4 Statistische Modelle

Bei der Präsentation der obigen zwei Verfahren haben wir immer die Existenz eines Modells mit gegebener Wahrscheinlichkeitsverteilung gefordert. Dies entspricht einem statischen Modell. Es gibt aber heutzutage eine Vielzahl von Modellen, die sich dadurch unterscheiden, wie die einzelnen Wahrscheinlichkeiten geschätzt werden.

4.1 Statische Modelle

Statische Modelle sind die einfachsten Modelle. Hier wird eine feste Wahrscheinlichkeitstabelle zugrunde gelegt, die unabhängig von den zu komprimierenden Daten ist. Die gleiche Tabelle kann zur Kompression von mehreren Daten genutzt werden. Angewandt mit dem Huffman-Verfahren hat ein statisches Modell den Vorteil, dass der Huffman-Baum nicht zum Decodierer geschickt werden muss, da dieser die gleiche Tabelle nutzt. Ein Kompressionsalgorithmus, der aber ein statisches Modell nutzt, erreicht im Allgemeinen keine perfekte Kompressionsrate, da sich die tatsächlichen Auftretenswahrscheinlichkeiten nicht im Modell widerspiegeln.

4.2 Order- n Modelle

Im Allgemeinen ist das Auftreten eines Symbols in einer Sequenz der Datenquelle von den vorher aufgetretenen Symbolen abhängig. Order-n Modelle sind Modelle, die für die Schätzung der Auftretenswahrscheinlichkeit eines Symbols die n vorher aufgetretenen Symbole berücksichtigen.

4.3 Adaptative Modelle

Mit adaptativen Modellen bezeichnet man Modelle, die die Auftretenswahrscheinlichkeiten der Symbole im Laufe des Codierungsvorgangs „ lernen “. Zum Start wird angenommen, dass alle Symbole mit der gleichen Wahrscheinlichkeit auftreten. Im Laufe der Codierung werden die Wahrscheinlichkeiten abhängig von den betrachteten Symbolen angepasst. Adaptative Modelle sind heute sehr beliebt vor allem in Verbindung mit der arithmetischen Codierung.

5 Huffman- vs arithmetische Codierung

Der Vergleich von diesen beiden Codierungsverfahren kann in [Say06, Kapitel 4, Abschnitt 4.5] nachgelesen werden.

Hinsichtlich der Laufzeit ist die Huffman-Codierung schneller als die arithmetische Codierung, da die arithmetische Codierung sehr viele Multiplikationen benötigt. Was der Effizienz angeht, hatten wir folgende Schranken für die mittlere Länge eines arithmetischen Codes:

$$H(p) \leq L_\Omega < H(p) + \frac{2}{m}.$$

Für den Huffman-Code von Sequenzen der Länge m hatten wir auch folgende Schranken:

$$H(p) \leq L_H < H(p) + \frac{1}{m}.$$

Anscheinend ist die Huffman-Codierung besser als die arithmetische Codierung. Man darf aber nicht vergessen, dass die Codewörter aller Sequenzen der Länge m mit dem Huffman-Verfahren generiert werden müssen, wenn man das Huffman-Codewort einer einzigen Sequenz bilden möchte. Dies führt dazu, dass die Größe der Codetabelle exponentiell mit der Länge m der Sequenz wäscht. Beispielsweise kann eine Codetabelle der Größe 10^{10} nicht effizient verwaltet werden. Mit der arithmetischen Codierung braucht man aber nicht die Codewörter anderer Sequenzen zu generieren und je länger die zu codierende Sequenz ist, desto besser wird die Entropie der Datenquelle angenähert. Dies führt dazu, dass die arithmetische Codierung geeigneter mit wachsender Länge der Sequenz ist.

6 Anwendungsfelder

Zur Textkomprimierung werden die Huffman- und arithmetische Codierung häufig eingesetzt. Neben diesem Einsatzgebiet wird die Huffman-Codierung auch zur Fax-Übertragung verwendet. Außerdem ist sie in einigen Datenformaten wie ZIP, GZIP, JPEG und MP3 anzutreffen. Die arithmetische Codierung wird ebenfalls im JPEG-Kompressionsverfahren eingesetzt. Angewandt wird aber die arithmetische Codierung nicht so oft. Dies liegt an der Existenz gültiger Patente.

Literatur

[BCK02] Eric Bodden, Malte Clasen, and Joachim Kneis. *Arithmetische Kodierung.* http://www-users.rwth-aachen.de/eric.bodden/ac/, 2002.

[Bro98] Manfred Broy. *Informatik: eine grundlegende Einführung. Band 1: Programmierung und Rechnerstrukturen.* Springer-Verlag,2.Auflage, 1998.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms.* The MIT-Press, third Edition, 2009.

[Huf52] D.A. Huffman. A method for the construction of minimum-redundancy codes. In *Proceedings of the I.R.E*, pages 1098–1101, 1952.

[KPS06] Herbert Klimant, Rudi Piotraschke, and Dagmar Schönfeld. *Informations- und Kodierungstheorie.* Teubner-Verlag, 3.Auflage, 2006.

[Say06] Khalid Sayood. *Introduction to Data Compression.* Morgan Kaufmann-Publisher, third Edition, 2006.

[Sei06] Peter Seibt. *Algorithmic Information Theory.* Springer-Verlag, 2006.