



---

**HOCHSCHULE COBURG**

**Hochschule für angewandte Wissenschaften  
Coburg**

Institut für Informatik

**Asymmetrische Zahlensysteme**

**Bachelorarbeit**

**von**

**Michael Krasser**

**Betreuer: Prof. Dr. Quirin Meyer**

**Coburg, 3. März 2019**

---

# Inhaltsverzeichnis

<b>1 Einleitung</b>	<b>3</b>
<b>2 Kompression und Dekompression</b>	<b>4</b>
2.1 Grundlagen . . . . .	4
2.2 Über die Möglichkeit der Datenkompression . . . . .	4
2.3 Entropiekodierer . . . . .	5
2.3.1 Die Entropie . . . . .	5
2.3.2 Kompression nach dem Verfahren von Shannon . . . . .	6
2.3.3 Kompression nach dem Verfahren von Fano . . . . .	7
2.3.4 Huffman-Algorithmus . . . . .	7
2.3.5 Erweiterte Huffman-Codierung . . . . .	9
2.3.6 Arithmetisches Kodieren . . . . .	10
<b>3 Kompression von Bilddateien durch Anwendung von Filtern</b>	<b>12</b>
3.1 Verwendete Filter . . . . .	12
3.1.1 Filtertyp 0: <code>none</code> . . . . .	12
3.1.2 Filtertyp 1: <code>sub</code> . . . . .	12
3.1.3 Filtertyp 2: <code>up</code> . . . . .	13
3.1.4 Filtertyp 3: <code>average2</code> . . . . .	14
3.1.5 Filtertyp 4: <code>average3</code> . . . . .	14
3.1.6 Filtertyp 5: <code>average4</code> . . . . .	15
3.1.7 Filtertyp 6: <code>average5</code> . . . . .	16
3.1.8 Filtertyp 7: <code>average6</code> . . . . .	17
3.1.9 Filtertyp 8: <code>average7</code> . . . . .	17
3.1.10 Filtertyp 9: <code>average8</code> . . . . .	18
3.1.11 Filtertyp 10: <code>paeth</code> . . . . .	19
3.2 Fazit zur Wahl der Filter . . . . .	20
<b>4 Asymmetrische Zahlensysteme</b>	<b>21</b>
4.1 Einführung . . . . .	21
4.2 Einführendes Beispiel: Uniform binary variant streaming ( <code>uabs</code> ) . . . . .	21
4.3 <code>uabs</code> für n-elementige Symbolalphabete ( <code>uans</code> ) . . . . .	23
4.4 Range variants and streaming: ( <code>rans</code> ) . . . . .	26
4.5 Vergleich der beiden Verfahren . . . . .	29
<b>5 Vergleich zu Huffmann und Arithmetischem Kodieren</b>	<b>30</b>
<b>6 Vektorisierung</b>	<b>31</b>
6.1 Bestimmung der Häufigkeiten . . . . .	31
6.2 Normalisierung . . . . .	31
6.2.1 Normalisierung der Häufigkeiten des <code>uans</code> -Verfahrens . . . . .	31
6.2.2 Normalisierung der Häufigkeiten des <code>rans</code> -Verfahrens . . . . .	32

<i>INHALTSVERZEICHNIS</i>	2
6.3 Wahl des normalisierten Intervalls . . . . .	32
6.4 Indexmengenbildung des <i>uans</i> -Verfahren . . . . .	32
6.5 Iterationswege der Kompressoren . . . . .	33
6.6 Endresultat der Kompression: Binäre Datei . . . . .	33
<b>7 Ergebnisse</b>	<b>36</b>
7.1 Ergebnisse der Normalisierung . . . . .	36
7.2 Ergebnisse der Filterung . . . . .	39
7.2.1 Verhalten der Entropie unter Verwendung der Filterung und Normalisierung für das <i>uans</i> -Verfahren . . . . .	41
7.3 Ergebnisse der Kompression . . . . .	41
<b>8 Kommandozeilenparameter</b>	<b>42</b>
<b>9 Tabellenverzeichnis</b>	<b>43</b>
<b>10 Abbildungsverzeichnis</b>	<b>44</b>
<b>11 Literaturverzeichnis</b>	<b>45</b>

# 1 Einleitung

Im Zeitalter der Digitalisierung werden immer größere Datenmengen produziert, gespeichert und übertragen, was sowohl den Einsatz als auch die Suche nach effizienten Verfahren zur Datenkompression unerlässlich macht. Hierfür schlägt Jaroslaw Duda[3] ein neues Verfahren mit dem Namen „asymmetrische Zahlensysteme“ vor, das seitdem von Unternehmen wie Facebook[9], Google[11] und Apple[10] verwendet wird. Im Rahmen dieser Bachelorarbeit über asymmetrische Zahlensysteme sollen zunächst grundlegende Begriffe aus dem Bereich der Codierungstheorie geklärt und gängige Entropiecodierer besprochen werden. Anschließend sollen die in [2], [3] und [1] angegebene Verfahren **uabs** und **rans** untersucht werden.

Ziel dieser Arbeit ist Verallgemeinerung des **uabs**-Verfahrens auf ein  $n$ -elementiges Symbolalphabet (**uans**-Verfahren) und die Anwendung des **uans**- sowie des **rans**-Verfahren auf Bilddateien. Zusätzlich soll der Einfluß einer vorgeschalteten Filterung – die in Kapitel 3 besprochen wird – untersucht werden. Hierfür werden die verwendeten Filter und die beiden Verfahren in C++ implementiert.

Nachfolgende Abbildung veranschaulicht den Programmfluss:

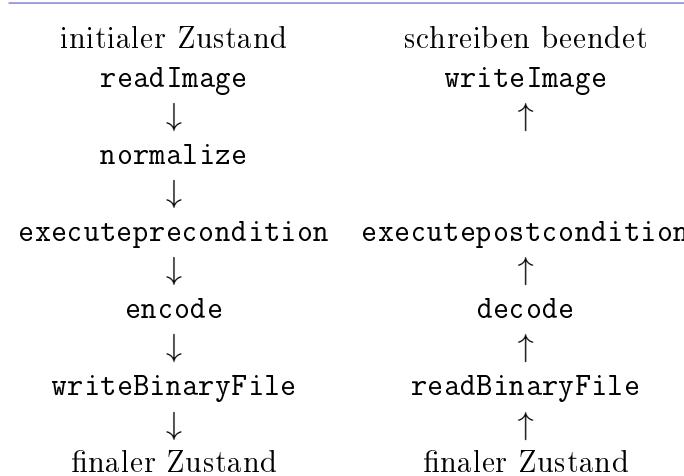


Abbildung 1: Programmfluß der Implementierung

In Leserichtung wird das Bild gelesen, vorkonditioniert, normalisiert und anschließend komprimiert. Die Dekomprimierung erfolgt in umgekehrter Leserichtung.

In Kapitel 8 findet sich eine Anleitung zur Bedienung der C++ Implementierung.

## 2 Kompression und Dekompression

Dieser Abschnitt orientiert sich an[4] sowie [6] und klärt die wichtigsten Begriffe aus dem Bereich Datenkompression und stellt gängige Entropiecodierer vor.

### 2.1 Grundlagen

#### **Definition 2.1.1:**

Unter Datenkompression versteht man ein Paar  $(\mathcal{X}, \mathcal{Y})$  von Algorithmen - Datenkompressionschema genannt, mit folgenden Eigenschaften:

Der Algorithmus  $\mathcal{X}$  (*Kompression*) konstruiert für die Eingabe  $X$  eine Repräsentation  $X_c$ , die (möglichst) weniger Bits als  $X$  benötigt.

Der Algorithmus  $\mathcal{Y}$  (*Dekompression*) generiert zu gegebenem  $X_c$  die Rekonstruktion  $Y$ . Ein Datenkompressionschema heißt verlustfrei, wenn  $X = Y$ , ansonsten verlustbehaftet.

Des Weiteren definiert man zur Beurteilung der Qualität eines Komprimierungsschemas:

#### **Definition 2.1.2:**

Sei  $\mu_c$  die Anzahl der Bits von  $X_c$  und  $\mu$  die Anzahl der Bits von  $X$ . Dann heißt der durch  $\mu_c/\mu$  definierte Ausdruck Kompressionsquotient<sup>1</sup>.

### 2.2 Über die Möglichkeit der Datenkompression

Von einer Datenkompression spricht man, wenn ein Verfahren existiert, das eine bestimmte Repräsentation in eine andere überführt, die stets weniger Bits als die ursprüngliche benötigt, dann zeigt ein einfaches Abzählargument, dass es ein solches Verfahren nicht geben kann. Diese Argumentation wird in Arbeiten zum Thema Kolmogorov-Komplexität als sog. Inkompresibilitätsargument bezeichnet und begründet, warum sich bestimmte Bitfolgen nicht komprimieren lassen: Die Kolmogorov-Komplexität einer Bitfolge  $t$  ist definiert als die Länge des kürzesten „Programms“, welches  $t$  erzeugt<sup>2</sup>. Eine Bitfolge heißt unkomprimierbar, falls sie selbst ihre kürzeste Beschreibung ist. Trotzdem funktioniert die Datenkompression in der Praxis hervorragend:

- Einzelne Zeichen (Grundelemente des Alphabets) tauchen nicht mit der selben Wahrscheinlichkeit auf.  $\Rightarrow$  verschlüsselt wahrscheinlichere Zeichen mit weniger Bit als unwahrscheinlichere (Morse-Alphabet).
- Blöcke von Zeichen lassen sich aufgrund der Abhängigkeiten aufeinander folgender Ereignisse geschickt gemeinsam als lose unabhängige Folge einzelner Zeichen verschlüsseln.

---

<sup>1</sup> Der Kehrwert des Kompressionsquotienten wird auch Kompressionsfaktor genannt.

<sup>2</sup> Bei geeigneter Formalisierung des Begriffs „Programm“ ist die Kolmogorov-Komplexität bis auf eine additive Konstante wohldefiniert.[5]

- Information ist nicht zufällig, sondern enthält zahllose Regelmäßigkeiten.  
Eine Grundstrategie von Kompressionsverfahren beruht darauf, solche Regeln zu entdecken (z.B. Wörterbuchtechniken).
- Für oder von Menschen erzeugte Information ist nicht zusammenhangslos. Vielfach ist es möglich, verschiedene Gattungen von Information zu unterscheiden: Sei z.B. bekannt, dass die vorliegende Bitfolge einen ASCII-Text der englischen Sprache darstellt, so muss ein anderes Kompressionsverfahren (auf Byte-Ebene) als wenn die Bitfolge die zeilenweise Darstellung eines Schwarz-Weiß-Bildes repräsentiert (starke Abhängigkeiten von Bit zu Bit, zeilen- und spaltenweise).
- Oft kann auf eine exakte Rekonstruierbarkeit des Originals verzichtet werden, da gewisse Feinheiten vom menschlichen Auge oder Ohr nicht wahrgenommen werden: Frequenzen von Audiodaten, die für das menschliche Ohr gedacht sind und nicht wahrgenommen werden, können, von vornherein vernachlässigt werden.

## 2.3 Entropiekodierer

### 2.3.1 Die Entropie

#### Definition 2.3.1:

Gegeben sei eine diskrete Quelle  $Q$  mit den Symbolen  $x_1, \dots, x_N$  und dazugehörigen Wahrscheinlichkeiten  $p_1, \dots, p_N$ .

Bezeichne  $I(x_i) = -\log_2(p_i)$  den Informationsgehalt des Symbols  $x_i$ . Die Entropie (mittlere Informationsgehalt) von  $Q$  ist definiert durch

$$H(Q) := \sum_{i=1}^N p(x_i) \cdot I(x_i) = - \sum_{i=1}^N p(x_i) \cdot \log_2(p(x_i)).$$

Die Entropie von  $Q$  heißt maximal, wenn  $H(Q) = \log_2(N)$ . Diese maximale Entropie wird mit  $H_0$  bezeichnet und heißt Entscheidungsgehalt.

#### Definition 2.3.2:

Entropiekodierung bezeichnet ein Verfahren, welches die Entropie einer Nachricht optimiert, d. h., der maximalen Entropie annähert.

Abschließend definieren wir den Begriff der *Redundanz und der erwarteten Länge* eines Codes:

#### Definition 2.3.3:

Sei  $\Sigma = \{a_1, \dots, a_k\}$  ein Alphabet mit zugehörigen Auftrittswahrscheinlichkeiten

$$\{P(a_i) \mid i = 1, \dots, k\}.$$

Die Kodierung  $\mathcal{K} : \Sigma \rightarrow \{0, 1\}^{\mathbb{N}}$  besitzt die erwartete Länge

$$L(\mathcal{K}) := \sum_{i=1}^k P(a_i) \cdot |\mathcal{K}(a_i)|.$$

Der Term

$$r := L(\mathcal{K}) - H(\Sigma)$$

bezeichnet die Redundanz der Kodierung.

### 2.3.2 Kompression nach dem Verfahren von Shannon

Nach Shannon gilt:

**Satz 2.3.1:**

Für Jede Quelle  $Q$  und jede beliebige zugehörige Binärcodierung mit Präfix-Eigenschaft ist die zugehörige mittlere Codewortlänge  $L$  nicht kleiner als die Entropie  $H(Q)$ :  $H(Q) \leq L$

**Beweis:**

Siehe [7].

□

Um für einen gegebenen Text einen dekodierbaren Code mit minimaler Länge zu finden, geht man nach Shannon folgendermaßen vor:

- 1) Sortiere zu kodierende Symbole  $x_1, \dots, x_N$  nach fallender Auftrittswahrscheinlichkeit:  $p_1 \geq p_2 \geq \dots \geq p_N$ .
- 2) Bestimme für  $1 \leq i \leq N$  die Codewortlänge  $\mathcal{K}(x_i) = \lfloor -\log_2(p(x_i)) \rfloor$  jedes Zeichens aus seinem Informationsgehalt.
- 3) Berechne für  $1 \leq i \leq N$  die kumulierte Wahrscheinlichkeiten  $P(x_i) := \sum_{l=0}^{i-1} p(x_l)$ .
- 4) Die Codierung jedes Zeichens  $x_i$  ergibt sich dann aus den ersten  $\mathcal{K}(x_i)$  Zeichen der Mantisse der Binärdarstellung von  $P(x_i)$ .

**Beispiel 2.3.1:**

$x_i$	$p(x_i)$	$\mathcal{K}(x_i)$	$P(x_i)$	binär	Code
a	0.3	2	0	0.000...	00
b	0.3	2	0.3	0.0100...	01
c	0.2	3	0.6	0.1001...	100
d	0.1	4	0.8	0.1100...	1100
e	0.1	4	0.9	0.1110...	1110

Dieses Verfahren ist eines der ersten zur Erstellung von präfixfreien, längenvariablen Codes.

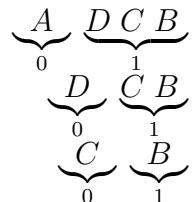
### 2.3.3 Kompression nach dem Verfahren von Fano

Ein weiteres Beispiel für einen Präfixcode ist die Shannon-Fano Codierung:

- 1) Sortiere zu kodierende Symbole  $x_1, \dots, x_N$  nach fallender Auftrittswahrscheinlichkeit:  $p_1 \geq p_2 \geq \dots \geq p_N$
- 2) Teile Menge der Symbole in 2 Teilmengen mit möglichst gleicher Wahrscheinlichkeit.
- 3) Kodiere linke Teilmenge mit 0, die rechte mit 1
- 4) Sind entstandene Teilmengen nicht einelementig, gehe zu 2.

#### Beispiel 2.3.2:

Gegeben sei das vierelementige Alphabet  $\{A, B, C, D\}$  mit zugehörigen Auftrittswahrscheinlichkeiten  $p_A = 0.4, p_B = 0.3, p_C = 0.2, p_D = 0.1$ . Man erhält  $A = 0, B = 111, C = 110, D = 10$



### 2.3.4 Huffman-Algorithmus

Um einen Code auch wieder eindeutig dekodieren zu können, muss er die Kraftsche Ungleichung erfüllen und zusätzlich noch präfixfrei sein, d. h. kein Codewort darf der Beginn eines anderen sein. (An andere Stelle setzen.....) Die Huffman-Kodierung ist eine Form der Entropiekodierung, die 1952 von David A. Huffman entwickelt und in [8] publiziert wurde. Die Kodierung ist ein Präfix-Code (also ein Code mit kleinster erwarteter Länge), der einer festen Anzahl von Quellsymbolen Codewörter mit variabler Länge zuordnet.

#### Algorithmus 2.3.1: (Huffman-Kodierung)

Sei  $\mathcal{A}$  ein Symbolalphabet,  $p_x = p(x)$  die relative Häufigkeit des Symbols  $x \in \mathcal{A}$ ,  $\mathcal{C}$  das Codealphabet (Zeichenvorrat der Codewörter)  $m = |\mathcal{C}|$  die Mächtigkeit von  $\mathcal{C}$ .

#### Aufbau des Baumes

- 1) Erstelle für  $x \in \mathcal{A}$  einen Knoten mit zugehöriger Häufigkeit.
- Wiederhole folgende Schritte, bis nur noch ein Baum übrig ist:
  - i) Wähle die  $m$  Teilbäume mit der geringsten Häufigkeit in der Wurzel, bei mehreren Möglichkeiten die Teilbäume mit der geringsten Tiefe.
  - ii) Fasse diese Bäume zu einem neuen (Teil-)Baum zusammen.
  - iii) Notiere die Summe der Häufigkeiten in der Wurzel.

#### Konstruktion des Codebuchs

- 1) Ordne jedem Kind eines Knotens eindeutig ein Zeichen aus dem Codealphabet zu.

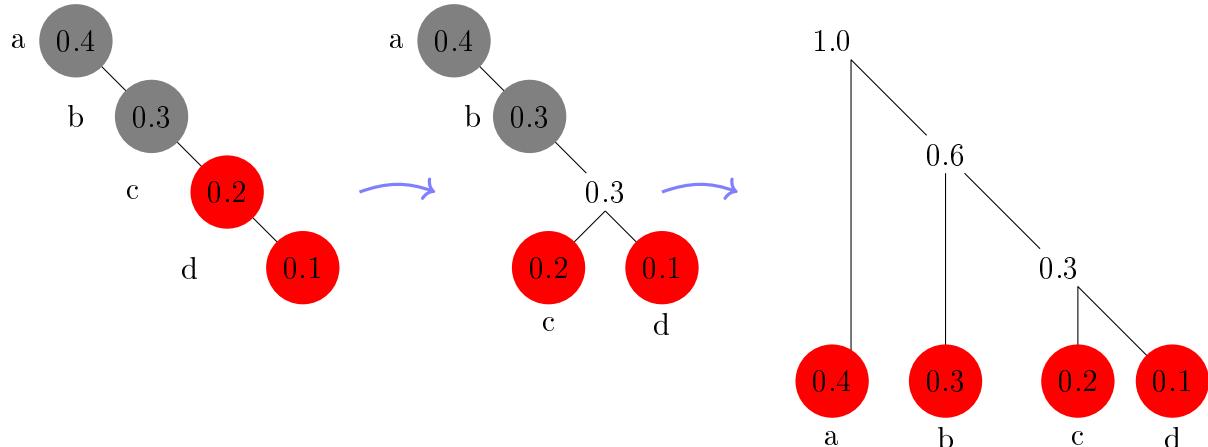
- 2) Lies für jedes Quellsymbol (Blatt im Baum) das Codewort aus: Beginne an der Wurzel des Baums. Die Codezeichen auf den Kanten des Pfades (in dieser Reihenfolge) ergeben das zugehörige Codewort.

**Beispiel 2.3.3:**

Die Nachricht  $aababcabcd$  soll auf Basis des Codealphabets  $\mathcal{C} = \{0, 1\}$  kodiert werden. Zunächst erhält man

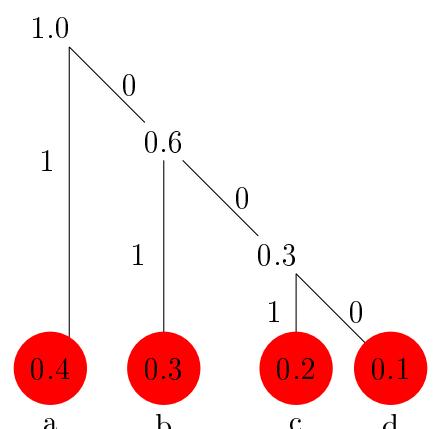
$$\mathcal{A} = \{a, b, c, d\} \quad \text{mit} \quad \begin{array}{c|c|c|c|c} X & a & b & c & d \\ \hline P(X) & 0.4 & 0.3 & 0.2 & 0.1 \end{array}$$

**Konstruktion des Huffman-Baums:**



**Konstruktion des Codebuchs:** Dazu versehen wir jeden linken Zweig mit einer 1, jeden rechten mit einer 0:

Symbol	Verschlüsselung
a	1
b	01
c	001
d	000



Damit kodiert sich die Nachricht  $a a b a b c a b c d$  zu 1 1 01 1 01 001 1 01 001 000.<sup>3</sup> In die-

<sup>3</sup> Bei der Dekodierung verfolgt man ausgehend von der Wurzel den entsprechende Pfad im Baum bis man an einem Blatt ankommt.

sem Fall erhält man für die Entropie und die erwartete Länge:

$$\begin{aligned} H(Q) &= -(0.4 * \log_2(0.4) + 0.3 * \log_2(0.3) + 0.2 * \log_2(0.2) \\ &\quad + 0.1 * \log_2(0.1)) = 1.84643934467 \\ L(\mathcal{K}) &= 0.4 + 2 \cdot 0.3 + 3 \cdot 0.3 = 1.9, \end{aligned}$$

d.h bei einer zu kodierenden Nachricht von 1000000 Symbolen sind etwa 28190 Symbole redundant.

### 2.3.5 Erweiterte Huffman-Codierung

Wir betrachten einführend folgendes Beispiel: Sei  $\Sigma = \{a, b\}$  mit  $p(a) = 0.9$ ,  $p(b) = 0.1$ . Dann liefert die Huffman-Kodierung

	p	Code
a	0.9	1
b	0.1	0

mit erwarteter Länge 1 Bits / Symbol und Entropie  $H(\Sigma) = 0.47$ . Die Redundanz  $r = 0.53$  beträgt also 113% der Entropie, d.h das der Code nicht komprimiert ist. Wir betrachten statt dessen das Alphabet

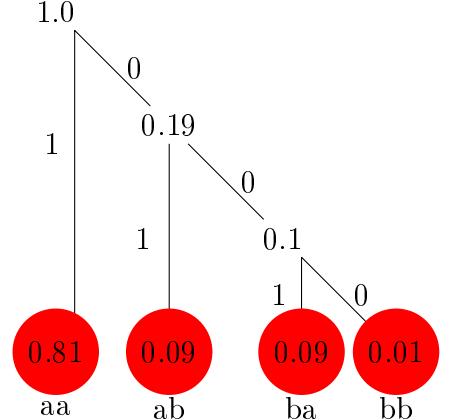
$$\Sigma^2 := \{aa, ab, ba, bb\}$$

mit  $p(aa) = (p(a))^2$ ,  $p(bb) = (p(b))^2$  und  $p(ab) = p(ba) = p(a) \cdot p(b)$ . Die Huffman-Kodierung liefert dann

Symbol	p	Verschlüsselung
aa	0.81	1
ab	0.09	01
ba	0.09	001
bb	0.01	000

$$H(\Sigma^2) = 0.937991187 \text{ Bits / Symbol}$$

$$L((K)) = 1.29 \text{ Bits / Symbol}$$



benutzt also nur noch 37% mehr als eine minimale Kodierung. Hierfür gilt folgender Sachverhalt:

#### Hilfssatz 2.3.1:

Für jedes Quellalphabet  $\Sigma$  gilt

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

Die Anfertigung der Statistiken für jedes Symbol ist praktisch kaum machbar. Abhilfe liefert das nächste Unterkapitel.

### 2.3.6 Arithmetisches Kodieren

Der letztes Unterabschnitt zeigte, dass die erweiterte Huffman-Kodierung eine Ausgabe mit sehr kleiner Redundanz produziert. Die Idee war die Folgende: Statt einzelne Symbole des Quellenalphabets wird die Folge der Symbole mit Hilfe des Huffman-Algorithmus codiert. Der Nachteil des Verfahrens ist aber, dass der Algorithmus Huffman-Codes für alle möglichen Folgen der Quellsymbole konstruieren muss, um einen Eingabetext zu codieren. Das heißt beispielsweise, dass für ein Quellenalphabet mit 256 Symbolen und für Folgen der Länge 3 der Algorithmus 16 777 216 neue Symbole betrachten muss. In diesem Kapitel betrachten wir arithmetische Codes: Statt einzelner Symbole werden Folgen von Symbolen codiert. Der Vorteil des Verfahrens ist dabei, dass jede Folge separat codiert werden kann. Das Schema des Verfahrens ist folgendes: Sei  $u_1 u_2 u_3 u_4 \in \Sigma$  ein Text. Berechne für die Folge eine numerische Repräsentation – einen Bruch zwischen 0 und 1, und für diese Repräsentation den binären Code.

Sei  $\Sigma = \{a_1, a_2, \dots, a_N\}$  das  $N$ -elementige Quellalphabet mit zugehörigen Auftrittswahrscheinlichkeiten  $p(a_1), \dots, p(a_N)$  und  $t := a_{i_1} a_{i_2} \dots a_{i_m}$  ein Text. Für  $i = 0, \dots, N$  definiert man die kumulierte Wahrscheinlichkeit

$$F(i) := \sum_{k=0}^i p(a_k), \quad F(0) := 0.$$

Eine *numerische Representation* von  $t$  ist ein Bruch im Intervall  $[l^m, u^m]$ , wobei

$$l^1 := F(i_1 - 1) \quad u^1 := F(i_1)$$

und

$$\begin{aligned} l^k &:= l^{k-1} + (u^{k-1} - l^{k-1}) \cdot F(i_k - 1) \\ u^k &:= l^{k-1} + (u^{k-1} - l^{k-1}) \cdot F(i_k), \quad k = 2, 3, \dots, m. \end{aligned}$$

#### Hilfssatz 2.3.2:

$$u^m - l^m = \prod_{k=1}^m p(a_{i_k}).$$

#### Beispiel 2.3.4:

Sei  $\Sigma = \{a, b, c\}$  mit  $p(a) = 0.7$ ,  $p(b) = 0.2$  und  $p(c) = 0.1$  und  $t = abb$ . Dann gilt  $i_1 = 1$  und  $i_2 = i_3 = 2$ . Dann erhält man

$k$	$l^k$	$u^k$
1	0.0	0.7
2	0.49	0.63
3	0.588	0.616

Als numerische Repräsentation lässt sich jetzt z.B das arithmetische Mittel

$$\mathfrak{T}(a_{i_1} \dots a_{i_m}) = \frac{l^m + u^m}{2}$$

wählen. Damit ist die einzige Information, die der Algorithmus zur Berechnung der numerischen Repräsentation benötigt, die Funktion  $F$ . Die Rekonstruktion der Folge aus  $\mathfrak{T}$  erfolgt dann über folgenden Algorithmus:

```

Sei  $l^0 = 0, u^0 = 1;$ 
For  $k:=1$  to  $m$  do
    begin
         $\mathfrak{T}^* = (\mathfrak{T} - l^{k-1})/(u^{k-1} - l^{k-1})$ 
        Finde  $i_k$  so dass  $F(i_k - 1) \leq \mathfrak{T} < F(i_k)$ 
        return  $(a_{i_k});$ 
        Berechne  $l^k$  und  $u^k;$ 
    end

```

Sei  $\mathfrak{T}(a_{i_1}a_{i_2} \dots a_{i_m}) = (u^m - l^m)/2$  eine numerische Repräsentation für die Folge  $a_{i_1}a_{i_2} \dots a_{i_m}$ . Die binäre Darstellung der numerischen Repräsentation kann beliebig lang bzw. unendlich sein. In diesem Kapitel zeigen wir, wie man die Repräsentation mit Hilfe einer kleinen Anzahl von Bits kodieren kann.

Es sei

$$p(a_{i_1}a_{i_2} \dots a_{i_m}) = p(a_{i_1})\Delta p(a_{i_2}) \dots p(a_{i_m}),$$

dann definiert man

$$l(a_{i_1}a_{i_2} \dots a_{i_m}) := \left\lceil \log \frac{1}{p(a_{i_1}a_{i_2} \dots a_{i_m})} \right\rceil + 1.$$

Den binären Code der Repräsentation definiert man als  $l(a_{i_1}a_{i_2} \dots a_{i_m})$  höchswertige Bits des Bruchs  $\mathfrak{T}(a_{i_1}a_{i_2} \dots a_{i_m})$ .

### Beispiel 2.3.5:

Sei  $\Sigma = \{a, b\}$ ,  $p(a) = 0.9$ ,  $p(b) = 0.1$  und die Länge  $m = 2$ . Dann erhält man folgende Kodierung für alle Folgen:

$x$	$\mathfrak{T}(x)$	binär	$l(x)$	Code
aa	0.405	0.0110011110...	2	01
ab	0.855	0.1101101011...	5	11011
ba	0.945	0.1111000111...	5	11110
bb	0.995	0.1111111010...	8	11111110

## 3 Kompression von Bilddateien durch Anwendung von Filtern

Die Komprimierung von Bilddateien kann durch eine Filterung erfolgen, welche mit Ausnahme des unten vorgestellten Filtertyps `none` in einer Erniedrigung der Entropie resultieren kann. Im Folgenden sollen die implementierten Filter anhand eines Bildes (i.F. als `img` bezeichnet) mit den Dimensionen  $h$  (Höhe),  $w$  (Breite) und  $c$  (Anzahl der Farbkanäle) kurz besprochen werden. Das Ergebnis der Vorfilterung wird mit `pre`, das der Umkehrung mit `post` bezeichnet. Zusätzlich müssen für die Umkehrung Werte in einem container - der i.F. als `res` bezeichnet wird - gespeichert werden. Die neben den Algorithmen gezeichneten Gitter dienen zur Verdeutlichung des gewählten Iterationsweges.

### Definition 3.0.1:

Sei  $(i, j) \in [0, h - 1] \times [0, w - 1]$  und  $0 \leq k < c$ . Dann bezeichnet  $*(i, j, k) := *_{i,j,k}$  den Wert von  $*$  an der Stelle  $(i, j, k)$ .

Die Wahl der Filter orientiert sich an [12]:

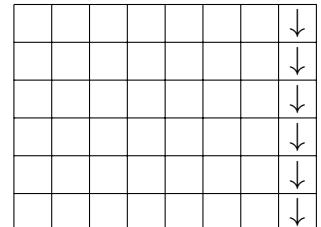
### 3.1 Verwendete Filter

#### 3.1.1 Filtertyp 0: none

Hierbei wird keine Vorfilterung betrieben. Jedes Byte von `img` verbleibt unverändert.

#### 3.1.2 Filtertyp 1: sub

Jedes Pixel von `img` wird durch die Differenz zu seinem linken Nachbarn ersetzt. Für jeden Farbkanal wird die rechte Kante des Bildes gespeichert.



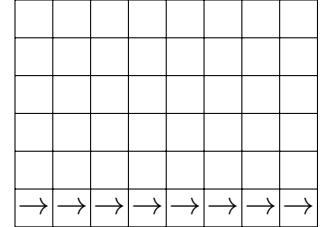
Verschlüsselung	Entschlüsselung
<pre> PROCEDURE ENCODE(img,pre,res)     new buffer; x = 1; y = 0; k = 0;     pre.resize((h - 1) * (w - 1) * c);         WHILE(k &lt; c)             WHILE(y &lt; h)                 WHILE(x &lt; w)                     pre<sub>x,y,k</sub><sup>4</sup> ← img<sub>x,y,k</sub> - img<sub>x-1,y,k</sub>;                     x++; </pre>	<pre> PROCEDURE DECODE(pre,post,res)     post.resize((h - 1) * (w - 1) * c);     x = w - 1; y = h - 1; k = c - 1;         WHILE(k &gt; 0)             buffer = res[k];             WHILE(y &gt; 0)                 post<sub>x,y,k</sub> ← buffer[y];                 WHILE(x &gt; 1) </pre>

<sup>4</sup> Die Berechnung muß ohne Überlauf erfolgen.

Verschlüsselung	Entschlüsselung
<pre>         END WHILE buffer.save(img<sub>w-1,y,k</sub>); x = 1;         y++;         END WHILE res[k] = buffer;         k++; y = 1;         END WHILE END PROCEDURE </pre>	<pre> post<sub>x-1,y,k</sub><sup>4</sup> ← post<sub>x,y,k</sub> - pre<sub>x,y,k</sub>;         x--;         END WHILE; y--;         x = w - 1;         END WHILE         k--;         END WHILE END PROCEDURE </pre>

### 3.1.3 Filtertyp 2: up

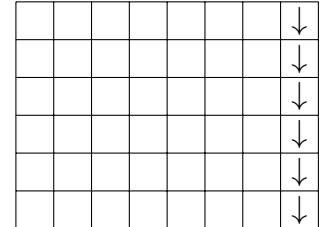
Jedes Pixel von `img` wird durch die Differenz zu seinem oberen Nachbarn ersetzt. Für jeden Farbkanal wird die untere Kante des Bildes gespeichert.



Verschlüsselung	Entschlüsselung
<pre> PROCEDURE ENCODE(img,pre,res)     new buffer; x = 0; y = 1; k = 0;     pre.resize((h - 1) * (w - 1) * c);         WHILE(k &lt; c)             WHILE(x &lt; w)                 WHILE(y &lt; h)                     pre<sub>x,y,k</sub><sup>4</sup> ← img<sub>x,y,k</sub> - img<sub>x,y-1,k</sub>;                     y++;                 END WHILE                 buffer.save(img<sub>x,y,k</sub>); y = 1;                 x++;             END WHILE             res[k] = buffer;             k++; x = 1;         END WHILE     END PROCEDURE </pre>	<pre> PROCEDURE DECODE(pre,post,res)     post.resize((h - 1) * (w - 1) * c);     x = w - 1; y = h - 1; k = c - 1;         WHILE(k &gt; 0)             buffer = res[k];             WHILE(x &gt; 0)                 post<sub>x,y,k</sub> ← buffer[x];                 WHILE(y &gt; 1)                     post<sub>x,y-1,k</sub><sup>4</sup> ← post<sub>x,y,k</sub> - pre<sub>x,y,k</sub>;                     y--;                 END WHILE                 y = h - 1; x--;             END WHILE             k--;         END WHILE     END PROCEDURE </pre>

### 3.1.4 Filtertyp 3: average2

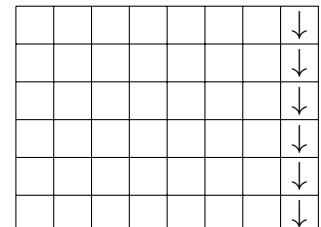
Jedes Pixel von `img` wird durch die Differenz zu dem Mittelwert aus seinem linken und oberen Nachbarn ersetzt. Für jeden Farbkanal wird die rechte Kante des Bildes gespeichert.



Verschlüsselung	Entschlüsselung
<pre> PROCEDURE ENCODE(img,pre,res)     new buffer; x = 0; y = 0; k = 0;     pre.resize((h - 1) * (w - 1) * c);         WHILE(k &lt; c)             WHILE(y &lt; h)                 WHILE(x &lt; w)                     pre<sub>x,y,k</sub><sup>4</sup> ← img<sub>x,y,k</sub> - 1/2 * (img<sub>x-1,y,k</sub>                         + img<sub>x,y-1,k</sub>);                     x++;                 END WHILE                 buffer.save(img<sub>x,y,k</sub>);                 x = 0; y++;             END WHILE             res[k] = buffer;             k++;             y = 0;         END WHILE     END PROCEDURE </pre>	<pre> PROCEDURE DECODE(pre,post,res)     post.resize((h - 1) * (w - 1) * c);     x = w - 1; y = 0; k = c - 1;         WHILE(k &gt; 0)             buffer = res[k];             WHILE(y &lt; h)                 post<sub>x,y,k</sub> ← buffer[y]                 WHILE(x &gt;= 1)                     post<sub>x-1,y,k</sub><sup>4</sup> ← post<sub>x,y,k</sub> - 1/2 * (post<sub>x,y-1,k</sub>                         + pre<sub>x,y,k</sub>);                     x--;                 END WHILE;                 y++;             END WHILE;             x = w - 1;         END WHILE;         k--;     END WHILE END PROCEDURE </pre>

### 3.1.5 Filtertyp 4: average3

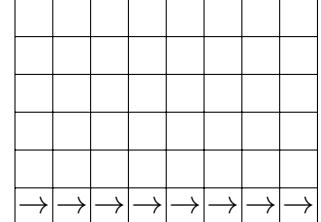
Jedes Pixel von `img` wird durch die Differenz zu dem Mittelwert aus seinem linken, oberen und rechten Nachbarn ersetzt. Für jeden Farbkanal wird die rechte Kante des Bildes gespeichert.



Verschlüsselung	Entschlüsselung
<pre> PROCEDURE ENCODE(img,pre,res)     new buffer; x = 0; y = 0; k = 0     pre.resize((h - 1) * (w - 1) * c);         WHILE(k &lt; c)             WHILE(y &lt; h)                 WHILE(x &lt; w)                     <math>pre_{x,y,k}^4 \leftarrow img_{x,y,k} - 1/3 * (img_{x-1,y,k} + img_{x,y-1,k} + img_{x+1,y,k})</math>;                     x++;                 END WHILE                 buffer.save(img_{x,y,k});                 x = 0; y++;             END WHILE             res[k] = buffer;             k++; y = 0;         END WHILE     END PROCEDURE </pre>	<pre> PROCEDURE DECODE(pre,post,res)     post.resize((h - 1) * (w - 1) * c);     x = w - 1; y = 0; k = c - 1         WHILE(k &gt; 0)             buffer = res[k];             WHILE(y &lt; h)                 post_{x,y,k} ← buffer[y];                 WHILE(x &gt; 1)                     post_{x-1,y,k} ← post_{x,y,k} - 1/3 * (post_{x,y-1,k} + post_{x+1,y,k} + pre_{x,y,k});                     x--;                 END WHILE;                 y++;             END WHILE;             k--;         END WHILE     END PROCEDURE </pre>

### 3.1.6 Filtertyp 5: average4

Jedes Pixel von `img` wird durch die Differenz zu dem Mittelwert aus seinem linken, oberen, rechten und unteren Nachbarn ersetzt. Für jeden Farbkanal wird die untere Kante des Bildes gespeichert.

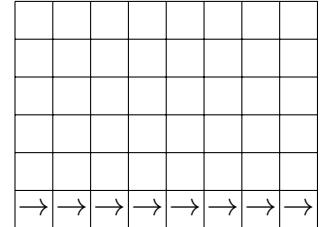


Verschlüsselung	Entschlüsselung
<pre> PROCEDURE ENCODE(img,pre,res)     new buffer; x = 0; y = 0; k = 0;     pre.resize((h - 1) * (w - 1) * c);         WHILE(k &lt; c)             WHILE(x &lt; w)                 WHILE(y &lt; h)                     <math>pre_{x,y,k}^4 \leftarrow img_{x,y,k} - 1/4 * (img_{x-1,y,k} + img_{x,y-1,k} + img_{x+1,y,k} + img_{x,y+1,k})</math>;                     y++;                 END WHILE                 buffer.save(img_{x,y,k});                 y = 0; x++;             END WHILE     END PROCEDURE </pre>	<pre> PROCEDURE DECODE(pre,post,res)     post.resize((h - 1) * (w - 1) * c);     x = 0; y = h - 1; k = c - 1;         WHILE(k &gt; 0)             buffer = res[k];             write(buffer);             WHILE(y &gt; 0)                 WHILE(x &lt; w)                     post_{x,y-1,k} ← post_{x,y,k} - 1/4 * (post_{x-1,y,k} + post_{x+1,y,k} + post_{x,y+1,k} + pre_{x,y,k});                     x++;                 END WHILE;                 y--;             END WHILE;     END PROCEDURE </pre>

Verschlüsselung	Entschlüsselung
<pre>         END WHILE     res[k] = buffer;     k ++; x = 0;         END WHILE     END PROCEDURE </pre>	<pre> y --; x = 0;         END WHILE k --; y = h - 1;         END WHILE     END PROCEDURE </pre>

### 3.1.7 Filtertyp 6: average5

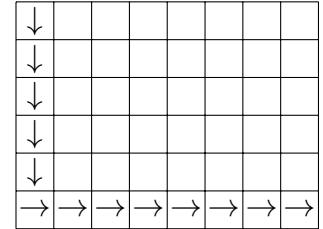
Jedes Pixel von `img` wird durch die Differenz zu dem Mittelwert aus seinem linken, links oberen, oberen, rechten und unteren Nachbarn ersetzt. Für jeden Farbkanal wird die untere Kante des Bildes gespeichert.



Verschlüsselung	Entschlüsselung
<pre> PROCEDURE ENCODE(img,pre,res)     new buffer; x = 0; y = 0; k = 0;     pre.resize((h - 1) * (w - 1) * c);         WHILE(k &lt; c)             WHILE(x &lt; w)                 WHILE(y &lt; h)                     pre<sub>x,y,k</sub><sup>4</sup> ← img<sub>x,y,k</sub> - 1/5 * (img<sub>x-1,y,k</sub>                         + img<sub>x-1,y-1,k</sub> + img<sub>x,y-1,k</sub>                         + img<sub>x+1,y,k</sub> + img<sub>x,y+1,k</sub>);                     y++;                 END WHILE                 buffer.save(img<sub>x,y,k</sub>);                 y = 0; x++;             END WHILE             res[k] = buffer;             k++; x = 0;         END WHILE     END PROCEDURE </pre>	<pre> PROCEDURE DECODE(pre,post,res)     post.resize((h - 1) * (w - 1) * c);     x = 0; y = h - 1; k = c - 1;         WHILE(k &gt; 0)             buffer = res[k];             write(buffer);             WHILE(y &gt; 0)                 WHILE(x &lt; w)                     post<sub>x,y-1,k</sub><sup>4</sup> ← post<sub>x,y,k</sub> - 1/5 * (post<sub>x-1,y,k</sub>                         + post<sub>x-1,y-1,k</sub> + post<sub>x+1,y,k</sub>                         + post<sub>x,y+1,k</sub> + pre<sub>x,y,k</sub>);                     x++;                 END WHILE;                 y--;             END WHILE;             k--;         END WHILE;     END PROCEDURE </pre>

### 3.1.8 Filtertyp 7: average6

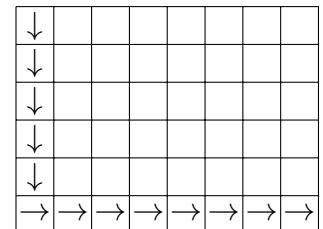
Jedes Pixel von `img` wird durch die Differenz zu dem Mittelwert aus seinem linken, links oberen, oberen, rechts oberen, rechten und unteren Nachbarn ersetzt. Für jeden Farbkanal wird die linke und die untere Kante des Bildes gespeichert.



Verschlüsselung	Entschlüsselung
<pre> PROCEDURE ENCODE(img,pre,res)     new buffer; x = 0; y = 0; k = 0;     pre.resize((h - 1) * (w - 1) * c);         WHILE(k &lt; c)             res[k] = Process(buffer);             buffer.clear();             WHILE(x &lt; w)                 WHILE(y &lt; h)                     pre<sub>x,y,k</sub><sup>4</sup> ← img<sub>x,y,k</sub> - 1/6 * (img<sub>x-1,y,k</sub>                         + img<sub>x-1,y-1,k</sub> + img<sub>x,y-1,k</sub>                         + img<sub>x+1,y-1,k</sub> + img<sub>x+1,y,k</sub>                         + img<sub>x,y+1,k</sub>);                     y++;                 END WHILE                 x++;             END WHILE             k++;         END WHILE     END PROCEDURE </pre>	<pre> PROCEDURE DECODE(pre,post,res)     post.resize((h - 1) * (w - 1) * c);     x = 0; y = h - 1; k = c - 1;         WHILE(k &gt; 0)             buffer = res[k];             write(buffer);             WHILE(y &gt; 0)                 WHILE(x &lt; w)                     post<sub>x+1,y-1,k</sub><sup>4</sup> ← post<sub>x,y,k</sub> - 1/6 * (post<sub>x-1,y,k</sub>                         + post<sub>x-1,y-1,k</sub> + post<sub>x,y-1,k</sub>                         + post<sub>x+1,y,k</sub> + post<sub>x,y+1,k</sub>                         + pre<sub>x,y,k</sub>);                     x++;                 END WHILE                 y--;             END WHILE             k--;         END WHILE     END PROCEDURE </pre>

### 3.1.9 Filtertyp 8: average7

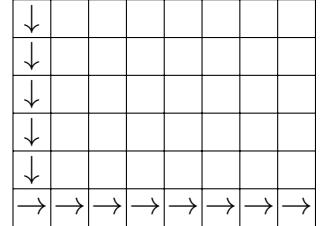
Jedes Pixel von `img` wird durch die Differenz zu dem Mittelwert aus seinem linken, links oberen, oberen, rechts oberen, rechten, rechts unteren und unteren Nachbarn ersetzt. Für jeden Farbkanal wird die linke und die untere Kante des Bildes gespeichert.



Verschlüsselung	Entschlüsselung
<pre> PROCEDURE ENCODE(img,pre,res)     new buffer; x = 0; y = 0; k = 0;     pre.resize((h - 1) * (w - 1) * c);         WHILE(k &lt; c)             res[k] = Process(buffer);             buffer.clear();             WHILE(x &lt; w)                 WHILE(y &lt; h)                     pre<sub>x,y,k</sub><sup>4</sup> ← img<sub>x,y,k</sub> - 1/7 * (img<sub>x-1,y,k</sub>                         + img<sub>x-1,y-1,k</sub> + img<sub>x,y-1,k</sub>                         + img<sub>x+1,y-1,k</sub> + img<sub>x+1,y,k</sub>                         + img<sub>x+1,y+1,k</sub> + img<sub>x,y+1,k</sub>);                     y++;                 END WHILE                 x++;             END WHILE             k++;         END WHILE     END PROCEDURE </pre>	<pre> PROCEDURE DECODE(pre,post,res)     post.resize((h - 1) * (w - 1) * c);     x = 0; y = h - 1; k = c - 1;     WHILE(k &gt; 0)         buffer = res[k];         write(buffer);         WHILE(y &gt; 0)             WHILE(x &lt; w)                 post<sub>x+1,y-1,k</sub><sup>4</sup> ← post<sub>x,y,k</sub> - 1/7 * (post<sub>x-1,y,k</sub>                     + post<sub>x-1,y-1,k</sub> + post<sub>x,y-1,k</sub>                     + post<sub>x+1,y,k</sub> + post<sub>x+1,y+1,k</sub>                     + post<sub>x,y+1,k</sub> + pre<sub>x,y,k</sub>);                 y--;             END WHILE;             x--;         END WHILE;         y--;     END WHILE;     k--; END PROCEDURE </pre>

### 3.1.10 Filtertyp 9: average8

Jedes Pixel von `img` wird durch die Differenz zu dem Mittelwert aus seinem linken, links oberen, oberen, rechts oberen, rechten, rechts unteren, unteren und links unteren Nachbarn ersetzt. Für jeden Farbkanal wird die linke und die untere Kante des Bildes gespeichert.

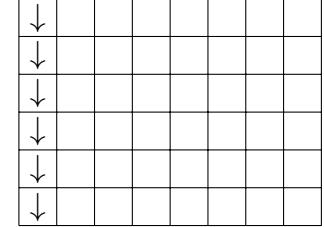


Verschlüsselung	Entschlüsselung
<pre> PROCEDURE ENCODE(img,pre,res)     new buffer; x = 0; y = 0; k = 0;     pre.resize((h - 1) * (w - 1) * c);         WHILE(k &lt; c)             res[k] = Process(buffer);             buffer.clear();             WHILE(x &lt; w)                 WHILE(y &lt; h)                     pre<sub>x,y,k</sub><sup>4</sup> ← img<sub>x,y,k</sub> - 1/8 * (img<sub>x-1,y,k</sub>                         + img<sub>x-1,y-1,k</sub> + img<sub>x,y-1,k</sub>                         + img<sub>x+1,y-1,k</sub> + img<sub>x+1,y,k</sub>                         + img<sub>x+1,y+1,k</sub> + img<sub>x,y+1,k</sub>                         + img<sub>x-2,y,k</sub> + img<sub>x+2,y,k</sub>);                 y++;             END WHILE             x++;         END WHILE     END PROCEDURE </pre>	<pre> PROCEDURE DECODE(pre,post,res)     post.resize((h - 1) * (w - 1) * c);     x = 0; y = h - 1; k = c - 1;     WHILE(k &gt; 0)         buffer = res[k];         write(buffer);         WHILE(y &gt; 0)             WHILE(x &lt; w)                 post<sub>x+1,y-1,k</sub><sup>4</sup> ← post<sub>x,y,k</sub> - 1/8 * (post<sub>x-1,y,k</sub>                     + post<sub>x-1,y-1,k</sub> + post<sub>x,y-1,k</sub>                     + post<sub>x+1,y,k</sub> + post<sub>x+1,y+1,k</sub>                     + post<sub>x,y+1,k</sub> + pre<sub>x,y,k</sub>);                 y--;             END WHILE;             x--;         END WHILE;         y--;     END WHILE;     k--; END PROCEDURE </pre>

Verschlüsselung	Entschlüsselung
$  \begin{aligned}  & + \text{img}_{x+1,y+1,k} + \text{img}_{x,y+1,k} \\  & + \text{img}_{x-1,y+1,k}); \\  & \quad y++; \\  & \quad \text{END WHILE} \\  & \quad x++; y = 0; \\  & \quad \text{END WHILE} \\  & \quad k++; x = 0; \\  & \quad \text{END WHILE} \\  & \text{END PROCEDURE}  \end{aligned}  $	$  \begin{aligned}  & + \text{post}_{x,y+1,k} + \text{post}_{x-1,y+1,k} \\  & + \text{pre}_{x,y,k}); \\  & \quad x++; \\  & \quad \text{END WHILE}; \\  & \quad y--; x = 0; \\  & \quad \text{END WHILE} \\  & \quad k--; y = h - 1; \\  & \quad \text{END WHILE} \\  & \text{END PROCEDURE}  \end{aligned}  $

### 3.1.11 Filtertyp 10: paeth

Jedes Pixel von `img` wird seinen `paeth`-Prädiktor ersetzt. Für jeden Farbkanal wird die linke Kante des Bildes gespeichert.



Verschlüsselung	Entschlüsselung
$  \begin{aligned}  & \text{PROCEDURE ENCODE(img,pre,res)} \\  & \quad \text{new buffer; } x = 0; y = 0; k = 0; \\  & \quad \text{pre.resize}((h - 1) * (w - 1) * c); \\  & \quad \text{WHILE}(k < c) \\  & \quad \quad \text{WHILE}(y < h) \\  & \quad \quad \quad \text{buffer.save}(\text{img}_{0,y,k}); \\  & \quad \quad \quad \text{WHILE}(x < w) \\  & \quad \quad \quad \quad \text{pre}_{x,y,k}^4 \leftarrow \text{paeth}(\text{img}_{x,y,k}) \\  & \quad \quad \quad \quad x++; \\  & \quad \quad \quad \text{END WHILE} \\  & \quad \quad x = 0; y++; \\  & \quad \quad \text{END WHILE} \\  & \quad \quad \text{res}[k] = \text{buffer}; \\  & \quad \quad k++; y = 0; \\  & \quad \quad \text{END WHILE} \\  & \text{END PROCEDURE}  \end{aligned}  $	$  \begin{aligned}  & \text{PROCEDURE DECODE(pre,post,res)} \\  & \quad \text{post.resize}((h - 1) * (w - 1) * c); \\  & \quad x = 1; y = 0; k = c - 1; \\  & \quad \text{WHILE}(k > 0) \\  & \quad \quad \text{buffer} = \text{res}[k]; \\  & \quad \quad \text{WHILE}(y < h) \\  & \quad \quad \quad \text{post}_{x-1,y,k} \leftarrow \text{buffer}[y] \\  & \quad \quad \quad \text{WHILE}(x > 1) \\  & \quad \quad \quad \quad \text{post}_{x,y,k}^4 \leftarrow \text{paeth}(\text{post}_{x,y,k}) + \text{pre}_{x,y,k}; \\  & \quad \quad \quad \quad x++; \\  & \quad \quad \quad \text{END WHILE} \\  & \quad \quad y++; x = 1; \\  & \quad \quad \text{END WHILE} \\  & \quad \quad k--; \\  & \quad \quad \text{END WHILE} \\  & \text{END PROCEDURE}  \end{aligned}  $

Hierbei ermittelt der `paeth`-prediktor folgenden Wert

```

paeth(imgx,y,k) := imgx-1,y,k + imgx,y-1,k - imgx-1,y-1,k;
left := |imgx-1,y,k - PAETH(imgx,y,k)| ;
up := |imgx,y-1,k - PAETH(imgx,y,k)| ;
leftUp := |imgx-1,y-1,k - PAETH(imgx,y,k)| ;
IF(left ≤ up && left ≤ leftUp)
    return left;
ELSE IF(up ≤ leftUp)
    return up;
return leftUp;

```

### 3.2 Fazit zur Wahl der Filter

Sämtliche Filtertypen lassen sich für jeden Farbkanal parallel verarbeiten, zusätzlich lässt sich der Filtertyp `sub` für jede Reihe und der Filtertyp `up` für jede Spalte parallelisieren. Eine Tabelle die den Einfluss der Filterung auf die Entropie wiedergibt, findet sich in den Ergebnissen.

## 4 Asymmetrische Zahlensysteme

### 4.1 Einführung

Sei  $\mathcal{A} := \{a_1, a_2, \dots, a_{n-1}\}$  ein endliches Alphabets,  $\{p_s\}_{s \in \mathcal{A}}$  die Folge der zugehörigen Auftrittswahrscheinlichkeiten mit  $\sum_{s \in \mathcal{A}} p_s = 1$  und

$$\begin{aligned}\mathcal{C} &: \mathcal{A} \times \mathbb{N} \rightarrow \mathbb{N} \\ \mathcal{D} &: \mathbb{N} \rightarrow \mathcal{A} \times \mathbb{N}\end{aligned}$$

die zugehörigen Kodierungs- und Dekodierungsfunktionen. Dabei liefert die Kodierungsfunktion  $\mathcal{C}$  für  $s \in \mathcal{A}$  und  $x \in \mathbb{N}$  den verschlüsselten Wert  $x' = \mathcal{C}(s, x)$  und  $\mathcal{D}$  bezeichnet die Inverse von  $\mathcal{C}$ :

$$\mathcal{D} \circ \mathcal{C} (b, \mathcal{C}(a, x)) = (b, \mathcal{C}(a, x))$$

Nach Konstruktion verhält sich  $x$  wie ein Stack: Der letzte verschlüsselte Wert ist der erste, den die Dekodierungsfunktion liefert. Nachdem  $\mathcal{C}$  für wachsendes  $x$  über alle Grenzen wächst, muß  $x$  auf ein Intervall mit festen Grenzen eingeschränkt werden: Hierfür wird in [3] das sog. „normalisierte Intervall“

$$I(L, b) := \{L, L + 1, \dots, b \cdot L - 1\}$$

für beliebiges  $L \in \mathbb{N}$  und  $b \in \mathbb{N}, b \geq 2$  definiert, wobei  $b$  die Basis des Kodierungsverfahrens bezeichnet. Ferner muß gewährleistet sein, dass für  $x' = \mathcal{C}(s, x)$  und  $(s, y) = \mathcal{D}(y')$  stets  $(x', y) \in I \times I$  gilt. Hierfür wird in [1] folgende Konvention getroffen:

- 1) Sei  $x' > b \cdot L - 1$ . Ersetze  $x'$  solange durch  $\left\lfloor \frac{x'}{b} \right\rfloor$  bis  $x' \in I$
- 2) Sei  $x' < L$ . Ersetze  $x'$  solange durch  $x' \cdot b$  bis  $x' \in I$

Im allgemeinen funktioniert dieser Ansatz nicht, aber es kann folgendes gezeigt werden:  
Sind die „Index-Mengen“

$$I_s := \{x \mid \mathcal{C}(s, x) \in I\}$$

von der Form  $I_s = \{k, k + 1, \dots, b \cdot k - 1\}$ ,  $k \geq 1$  (eine Eigenschaft, die Duda in [3] als „b-Eindeutigkeit“ bezeichnet), dann sind die Algorithmen synchron.

### 4.2 Einführendes Beispiel: Uniform binary variant streaming (uabs)

Vorab lässt sich das Verfahren folgendermaßen beschreiben:

Verschlüsselung	Entschlüsselung
<pre> PROCEDURE ENCODE(x, buffer)   WHILE C(x, s) ∉ I_s do     buffer.save(x mod b)     x ← ⌊x/b⌋   END WHILE   x ← C(x, s) END PROCEDURE </pre>	<pre> FUNCTION DECODE(buffer)   (s, x) ← D(x)   WHILE x ∉ I do     x ← b · x + buffer.read()   END WHILE   return s END FUNCTION </pre>

Einführend wird das in [2] angebene Beispiel betrachtet:

**Beispiel 4.2.1:**

Sei z.B. „babba“ eine auf dem zwei-Symbol Alphabet  $\mathcal{A} := \{a, b\}$  mit  $p_a = 1/4, p_b = 3/4$  bestehende Nachricht und

$$\mathcal{C}(a, x) = 4x, \quad \mathcal{C}(b, x) = 4 \lfloor x/3 \rfloor + (x \bmod 3) + 1.$$

Für  $b = 2$  und  $L = 16$ , d.h.  $I(L, b) = \{16, 17, \dots, 31\}$  berechnet man die den Symbolen korrespondierenden  $b$ -eindeutigen Index-Mengen:

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
a	4				5				6				7			
b		12	13	14		15	16	17		18	19	20		21	22	23

Als Dekodierungsfunktion erhält man:

$$\mathcal{D}(x) : \begin{cases} (a, x/4) & \text{falls } x \equiv 0 \pmod 4, \\ (b, 3 \lfloor x/4 \rfloor + (x \bmod 4) - 1) & \text{sonst.} \end{cases}$$

Wird  $u \in \mathcal{A}$  mit  $\mathcal{C}$  codiert wächst  $x$  um den Faktor  $1/p_u$ . Die Prozedur der Verschlüsselung und Entschlüsselung lässt sich nachfolgender Tabelle entnehmen:

Kodierer	x	Dekodierer
finaler Zustand		finaler Zustand
↑	19	↓
encode 'b'		decode 'b'
↑	14	↓
$x \notin I_b$ ; save(0)		$x \notin I(L, b)$ ; read()=0;
↑	28	↓
encode 'a'		decode 'a'
↑	7	↓
$x \notin I_a$ ; save(1)		$x \notin I(L, b)$ ; read()=1;
↑	15	↓
$x \notin I_a$ ; save(0)		$x \notin I(L, b)$ ; read()=0;
↑	30	↓
encode 'b'		decode 'b'
↑	22	↓
encode 'b'		decode 'b'
↑	16	↓
encode 'a'		decode 'a'
↑	4	↓
$x \notin I_a$ ; save(0)		$x \notin I(L, b)$ ; read()=0;
↑	8	↓
$x \notin I_a$ ; save(0)		$x \notin I(L, b)$ ; read()=0;

Kodierer	x	Dekodierer
initialer Zustand	16	lesen beendet

### 4.3 uabs für $n$ -elementige Symbolalphabete (uans)

Ziel des folgenden Unterkapitels ist es das besprochene uabs-Verfahren auf  $n$ -elementige Symbolalphabete zu verallgemeinern. Im folgenden sei  $\mathcal{A}$  ein  $n$ -elementiges Alphabet ( $n > 1$ ) mit zugehörigen Auftrittswahrscheinlichkeiten  $\mathcal{P} := \{p_{i_k} \mid 1 \leq k \leq n - 1\}$  wobei gilt:

$$\forall p_{i_k} \in \mathcal{P} \exists i_k \in \mathbb{N} : p_{i_k} \cdot 2^{i_k} = 1, \sum_{p \in \mathcal{P}} p = 1.$$

Damit lässt sich das normalisierte Intervall in disjunkte Teilmengen zerlegen:

#### Hilfssatz 4.3.1:

Seien  $a_k, a_l \in \mathcal{A}, k < l$  zwei Symbole mit  $p_k = 1/2^k$  und  $p_l = 1/2^l$ . Sei  $0 \leq r_k < 2^k$  und  $r_l := \min_{r \in \mathbb{N}} \{(r_k - r) \bmod 2^k \neq 0\}$ . Folgende Mengen sind disjunkt<sup>5</sup>:

$$I_{a_k} := \{2^k + n \cdot r_k \mid n \in \mathbb{N}\}, \quad I_{a_l} := \{2^l + n \cdot r_l \mid n \in \mathbb{N}\}.$$

#### Beweis:

Sei  $\mu \in \mathbb{N}$  mit  $\mu \in I_{a_k} \cap I_{a_l}$ . Dann existieren  $s, t \in \mathbb{N}$  mit

$$s \cdot 2^k + r_k = t \cdot 2^l + r_l \quad \Leftrightarrow \quad 2^k \cdot (s - t \cdot 2^{l-k}) = r_l - r_k.$$

Hieraus folgt sofort der Widerspruch  $(r_l - r_k) \bmod 2^k \equiv 0$ , also  $I_{a_k} = I_{a_l}$ .

□

Folgender Algorithmus liefert arithmetische Progressionen zur Zerlegung von  $I(L, b)$  in disjunkte Teilmengen:

#### Algorithmus 4.3.1:

- 1 Fasse Symbole mit gleicher Wahrscheinlichkeit zusammen.
- 2 Sortiere Zusammenfassung nach fallender Wahrscheinlichkeit.
- 3 Für Wahrscheinlichkeit  $p_{i_k}$  existieren die Modulreste  $R_{i_k} := \{0, 1, 2, \dots, 2^{i_k} - 1\}$ .

<sup>5</sup> Dieser Sachverhalt gilt für jede Primzahl.

4 Beginne bei  $k = 0$ .

5 Sei  $k < l \leq N - 1$ . Korrespondiert  $p_{i_k}$  zu  $t \in \mathbb{N}$  Symbolen, selektiere die ersten  $t$  Reste aus  $R_{i_k}$  zu  $R'_{i_k} \subseteq R_k$ . Entferne  $x \in R_{i_l}$  falls ein  $y \in R'_{i_k}$  existiert mit  $0 \equiv (x - y) \pmod{2^{i_k}}$ .

6 Setze  $k = k + 1$  und gehe zu Schritt 5.

### Beispiel 4.3.1:

Für die Demonstration eines konkreten Beispiels wird die auf dem drei-Symbol Alphabet  $\mathcal{A} := \{a, b, c\}$  mit  $p_a = 1/2, p_b = 1/4, p_c = 1/4$  bestehende Nachricht „abbac“ für  $b = 2, L = 16$  kodiert und dekodiert. Algorithmus 4.3.1 liefert die disjunktiven arithmetischen Progressionen:

$x \in \mathcal{A}$	$p_{i_k} \in \mathcal{P}$	$R_{i_k}$
a	1/2	{0, 1}
b	1/4	{0, 1, 2, 3}
c	1/4	{0, 1, 2, 3}

$$\xrightarrow{3} \begin{array}{c|c|c|c}
a & 1/2 & \{0\} \\
b, c & 1/4 & \{0, 1, 2, 3\}
\end{array} \xrightarrow{4} \begin{array}{c|c|c|c}
a & 1/2 & \{0\} \\
b, c & 1/4 & \{\emptyset, 1, 2, 3\}
\end{array}$$

$$\xrightarrow{5} \begin{array}{c|c|c|c}
a & 1/2 & \{0\} \\
b, c & 1/4 & \{1, 3\}
\end{array}$$

Als Kodierungsfunktion erhält man nun

$$\mathcal{C}(a, x) = 2x, \quad \mathcal{C}(b, x) = 4x + 1, \quad \mathcal{C}(c, x) = 4x + 3,$$

also die den Symbolen korrespondierenden  $b$ -einheitigen Index-Mengen:

	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
a	8		9		10		11		12		13		14		15	
b		4				5				6				7		
c				4				5				6				7

Durch Invertierung erhält man sofort die zugehörige Dekodierungsfunktion:

$$\mathcal{D}(x) : \begin{cases} (a, x/2) & \text{für } x \equiv 0 \pmod{2}, \\ (b, (x-1)/4) & \text{für } x \equiv 1 \pmod{4} \\ (c, (x-3)/4) & \text{für } x \equiv 3 \pmod{4} \end{cases}$$

Die Prozedur der Verschlüsselung und Entschlüsselung lässt sich nachfolgender Tabelle entnehmen:

Kodierer	x	Dekodierer
finaler Zustand		finaler Zustand
↑	16	↓
encode 'a'		decode 'a'
↑	8	↓
$x \notin I_a$ ; save(1)		$x \notin I(L, b)$ ; read()=1;
↑	17	↓
encode 'b'		decode 'b'
↑	4	↓
$x \notin I_b$ ; save(0)		$x \notin I(L, b)$ ; read()=0;
↑	8	↓
$x \notin I_b$ ; save(1)		$x \notin I(L, b)$ ; read()=1;
↑	17	↓
encode 'b'		decode 'b'
↑	4	↓
$x \notin I_b$ ; save(1)		$x \notin I(L, b)$ ; read()=1;
↑	9	↓
$x \notin I_b$ ; save(0)		$x \notin I(L, b)$ ; read()=0;
↑	18	↓
encode 'a'		decode 'b'
↑	9	↓
$x \notin I_a$ ; save(1)		$x \notin I(L, b)$ ; read()=1;
↑	19	↓
encode 'c'		decode 'c'
↑	4	↓
$x \notin I_c$ ; save(0)		$x \notin I(L, b)$ ; read()=0;
↑	8	↓
$x \notin I_c$ ; save(0)		$x \notin I(L, b)$ ; read()=0;
↑	16	↓
initialer Zustand		lesen beendet

Für das Wachstumsverhalten der Entropie erhalten wir:

#### Hilfssatz 4.3.2:

Sei  $\mathcal{A}$  ein  $N$  elementiges Symbolalphabet mit zugehörigen Auftrittswahrscheinlichkeiten  $(\frac{1}{2})^{i_k}$  und  $k + k_0 \geq \frac{2^{i_k}}{i_k} \geq k$  für alle  $k$  und  $k_0 \in \mathbb{N}$ , dann ist  $H(\mathcal{A}) - H_0$  beschränkt.

Beweis:

$$H(\mathcal{A}) - H_0 = - \sum_{k=1}^{N-1} \frac{1}{2}^{i_k} \log_2 \frac{1}{2}^{i_k} - \log_2(N-1) = \sum_{k=1}^{N-1} i_k \cdot \frac{1}{2}^{i_k} - \frac{\log(N-1)}{\log(2)}.$$

Damit folgt:

$$\begin{aligned} \sum_{k=1}^{N-1} \frac{1}{k+k_0} - \frac{\log(N-1)}{\log(2)} &\leq H(\mathcal{A}) - H_0 \leq \sum_{k=1}^{N-1} \frac{1}{k+1} - \frac{\log(N-1)}{\log(2)} \\ &< \sum_{k=1}^N \frac{1}{k} - \log(N-1). \end{aligned}$$

Wir betrachten nur die obere Reihe, die Schlußfolgerung für die untere Reihe ist analog:  
Die Folge

$$\gamma_n := \sum_{k=1}^N \frac{1}{k} - \log(N)$$

ist monoton fallend und nach unten beschränkt:

$$\begin{aligned} \gamma_{n+1} - \gamma_n &= \frac{1}{N+1} - (\log(N+1) - \log(N)) \\ &= \frac{1}{N+1} - \log\left(1 + \frac{1}{N}\right) \end{aligned}$$

Da für alle  $x > 0 \log(x) \geq \frac{x-1}{x}$  gilt, folgt:

$$\gamma_{n+1} - \gamma_n \leq \frac{1}{N+1} - \frac{\frac{1}{N}}{1 + \frac{1}{N}} = \frac{1}{N+1} - \frac{1}{N+1} = 0.$$

Also ist  $\gamma_n$  monoton fallend. Da für alle  $x > -1$  gilt  $\log(x+1) \leq x$  folgt

$$\begin{aligned} \gamma_n &= \sum_{k=1}^N \frac{1}{k} - \log\left(\prod_{k=1}^{N-1} \frac{k+1}{k}\right) = \sum_{k=1}^N \frac{1}{k} - \sum_{k=1}^{N-1} \log\left(\frac{k+1}{k}\right) \\ &\geq \sum_{k=1}^N \frac{1}{k} - \sum_{k=1}^{N-1} \frac{1}{k} > 0. \end{aligned}$$

□

#### 4.4 Range variants and streaming: (rans)

In diesem Abschnitt betrachten das in [1] angegebene Verfahren, welches ohne die Benutzung der in der `uabs` verwendeten Index-Mengen auskommt und die Auftrittswahrscheinlichkeiten der einzelnen Symbole nicht approximieren muss:

Sei  $F_s$  die Häufigkeit des Symbols  $s \in \mathcal{A}$  und  $M := \sum_{s \in \mathcal{A}} F_s$ .

$$\begin{aligned}\mathcal{C}(s, x) &= x' := M \cdot \lfloor x/F_s \rfloor + B_s + (x \bmod F_s) \\ \mathcal{D}(x') &= (s, x) := \left( \mathcal{L}(\mathcal{R}), F_s \cdot \left\lfloor x'/M \right\rfloor + \mathcal{R} - B_s \right)\end{aligned}$$

wobei

$$\mathcal{R} := x' \bmod M, \quad B_s := \sum_{i=0}^{s-1} F_s, \quad \mathcal{L}(z) := \max_{B_s \leq z} s.$$

Die Funktion  $\mathcal{L}$  determiniert dabei das Symbol.  $\mathcal{C}$  und  $\mathcal{D}$  sind invers:

$$\begin{aligned}\mathcal{D} \circ \mathcal{C}(s, x) &= (\mathcal{L}(\mathcal{R}), F_s \cdot \lfloor 1/M(M \cdot \lfloor x/F_s \rfloor + B_s + (x \bmod F_s)) \rfloor + \mathcal{R} - B_s) \\ &= \left( \mathcal{L}(\mathcal{R}), F_s \cdot \left\lfloor \lfloor x/F_s \rfloor + \frac{B_s + (x \bmod F_s)}{M} \right\rfloor + \mathcal{R} - B_s \right).\end{aligned}$$

Wegen

$$\frac{B_s + (x \bmod F_s)}{M} \leq \frac{B_s + F_s - 1}{M} = \frac{B_{s+1} - 1}{M} < 1$$

folgt

$$\mathcal{D} \circ \mathcal{C}(s, x) = (\mathcal{L}(\mathcal{R}), F_s \cdot \lfloor x/F_s \rfloor + \mathcal{R} - B_s)$$

Mit  $\mathcal{R} = x' \bmod M = B_s + (x \bmod F_s)$  erhält man

$$\begin{aligned}\mathcal{D} \circ \mathcal{C}(s, x) &= (\mathcal{L}(\mathcal{R}), F_s \cdot \lfloor x/F_s \rfloor + (x \bmod F_s)) \\ &= \left( \mathcal{L}(\mathcal{R}), F_s \cdot \left( \frac{x - (x \bmod F_s)}{F_s} \right) + (x \bmod F_s) \right) \\ &= (\mathcal{L}(\mathcal{R}), x)\end{aligned}$$

Abschließend folgt:

$$\mathcal{L}(\mathcal{R}) = \mathcal{L}(x' \bmod M) = \mathcal{L}(B_s + x \bmod F_s) = \max_{0 \leq x \bmod F_s} s = s.$$

Das Verfahren lässt sich folgendermaßen beschreiben:

Verschlüsselung	Entschlüsselung
<pre> PROCEDURE ENCODE(x, List)     new buffer     WHILE C(x, s) ∉ I do         buffer.save(x mod b)         x ← ⌊ x/b ⌋     END WHILE     List.add(buffer)     x ← C(x, s) END PROCEDURE </pre>	<pre> FUNCTION DECODE(List)     buffer = List.get()     (s, x) ← D(x)     x ← b · x + buffer.read()     RETURN s END FUNCTION </pre>

**Beispiel 4.4.1:**

Zur Demonstration eines konkreten Beispiel wird wieder die auf dem 3-Symbol-Alphabet  $\mathcal{A} := \{a, b, c\}$  mit den Häufigkeiten  $F_a := 11, F_b := 3, F_c := 2$  bestehende Nachricht „abbac“ betrachtet:

Es gilt dann  $M = 16$  und man erhält folgende Lookup-Tabelle:

$x$	$< 11$	$11 \leq x < 14$	$14 \leq x$
Symbol	a	b	c
Kodierer x Dekodierer			
finaler Zustand		finaler Zustand	
↑	26	↓	
encode 'a'		decode 'a'	
↑		↓	
$x \notin I(L, b); \text{save}(1)$	21	read()=1;	
↑		↓	
$x \notin I(L, b); \text{save}(0)$	43	read()=0;	
↑		↓	
$x \notin I(L, b); \text{save}(0)$	86	read()=0;	
↑		↓	
encode 'b'	172	decode 'b'	
↑		↓	
$x \notin I(L, b); \text{save}(0)$	31	read()=0;	
↑		↓	
$x \notin I(L, b); \text{save}(0)$	62	read()=0;	
↑		↓	
$x \notin I(L, b); \text{save}(0)$	124	read()=0;	
↑		↓	
encode 'b'		decode 'b'	
↑	22	↓	
encode 'a'		decode 'a'	
↑	17	↓	
$x \notin I(L, b); \text{save}(1)$		read()=1;	
↑	35	↓	
$x \notin I(L, b); \text{save}(1)$		read()=1;	
↑	71	↓	
$x \notin I(L, b); \text{save}(0)$		read()=0;	
↑	142	↓	
encode 'c'		decode 'c'	
↑	16	↓	
initialer Zustand		lesen beendet	

## 4.5 Vergleich der beiden Verfahren

Das **rans**-Verfahren stellt keine Bedingungen an die Werte der Auftrittswahrscheinlichkeiten oder die Wahl des normalisierten Intervalls, außerdem ist das Verfahren sehr leicht zu implementieren.

Die Generierung der benötigten Indexmengen für das **uans**-Verfahren ist sehr aufwendig und auch nur durch Approximation der Auftrittswahrscheinlichkeiten möglich. Außerdem muss das normalisierte Intervall so gewählt werden, dass jedes Symbol mindestens einen Repräsentanten innerhalb des normalisierten Intervalls besitzt:

Sei z.B.  $L = 2$  und  $b = 3$ ,  $\mathcal{A} := \{a, b, c, d\}$  mit  $p_a = 1/2$ ,  $p_b = 1/4$  und  $p_c = p_d = 1/8$ . Dann gilt:

$$I_a = \{1, 2\}, \quad I_b = \{1\}, \quad I_c = \{1\}, \quad I_d = \emptyset.$$

Wählt man  $L^*$  derart, dass

$$L^* = \max \{r_k \mid r_k \in R_{i_k}, 0 \leq i_k < N - 1\} + 1$$

dann gilt für alle Reste  $0 \leq r_k < L^*$ . Verschiebt man dieses Intervall noch um die größte auftretende Potenz von 2 ( $=:M$ ), so besitzt jede Indexmenge einen Repräsentanten im Intervall  $[M, M + L^* - 1]$ , also auch in  $I(b, L)$  für  $b = 2$  und  $L := M$  (da  $L^* \leq M$ ), also gilt für die korrespondierenden Indexmengen  $I_{i_k} \neq \emptyset$ .

Eine weiterer Nachteil findet sich in dem Umstand, dass **uans**-Verfahren nicht immer terminiert:

Sei z.B.  $b = 4$  und  $L = 16$  und  $\alpha$  ein Symbol mit der relativen Häufigkeit  $p_\alpha = 1/32$ . Angenommen Algorithmus 4.3.1 liefert einen Modulorest  $0 < r < 16$ , dann erhält man als zugehörige Indexmenge  $I_\alpha = \{1\}$ . Falls der Kodierer z.B. bei  $x = 48$  das Symbol  $\alpha$  liest, so teilt er so lange durch  $b$  bis er einen Wert aus  $I_\alpha$  erhält, was nie der Fall ist. Zusammenfassend ist das **rans**-Verfahren damit besser geeignet.

Im Falle der Terminierung liefert das **uans**-Verfahren aber die bessere Kompressionsrate:

### Beispiel 4.5.1:

Man betrachte die Komprimierung des Wortes  $\mathcal{T} := \text{"MISSISSIPPI"}$  für  $b = 2$ ,  $L = 16$ ,  $M = 8$  durch beide Verfahren:

$$\mathcal{T} \xrightarrow{\text{rans}} 001000010010110010110 \quad 19 \quad \mathcal{T} \xrightarrow{\text{uans}} 1|01|100|1|110|00|0|00|1|0010|19$$

Das **rans**-Verfahren benötigt zur Dekomprimierung von  $\mathcal{T}$  zusätzliche Information. Der Grund dieser Umstände liegt in der Konstruktion: Das **rans**-Verfahren normalisiert nur bei der Kodierung aus positiver Richtung in das „normalisierte Intervall“  $I(L, b)$ , während das **uans**-Verfahren das Bitmuster so lange ausliest und den augenblicklichen Wert erhöht, bis ein Wert größer als  $L$  gefunden ist.

## 5 Vergleich zu Huffmann und Arithmetischem Kodieren

Der Huffmann-Algorithmus kodiert jedes Symbol einzeln, während der erweiterte Huffmann-Algorithmus zu einer gegebenen Länge jede erdenkliche Folge einzeln verschlüsselt. Der Vorteil der Erweiterung liegt darin, dass sich ein anfänglich nicht zu komprimierender Code doch komprimieren lässt. Allerdings ist die Codierung ab einer bestimmten Länge kaum realisierbar. Dieses Problem tritt bei der Verwendung von asymmetrischen Zahlensystemen nicht auf: Jede Folge ist komprimierbar.

## 6 Vektorisierung

In diesem Abschnitt werden die beiden in Kapitel 4 besprochenen Verfahren auf Bilddateien angewendet, d.h. im Folgenden gilt  $\mathcal{A} = \{0, 1, \dots, 255\}$ .

### 6.1 Bestimmung der Häufigkeiten

Zum Laden einer Bilddatei wird die Funktion<sup>6</sup>

```
stbi_load(char const *filename, int *x, int *y, int *comp, int req_comp)
```

welche einen Zeiger auf ein Array vom Typ `unsigned char` liefert, verwendet. Die Ermittlung der Häufigkeiten  $F_s$ ,  $s \in \mathcal{A}$  erfolgt dann durch Iteration über dieses Array<sup>7</sup> während derer die jeweiligen Einträge eines `std::vector<uint32>` der Größe 256 inkrementiert werden.

### 6.2 Normalisierung

Wesentliche Voraussetzung für die Anwendung der beiden Verfahren ist eine Normalisierung der Häufigkeiten  $F_s$ ,  $s \in \mathcal{A}$  welche Gegenstand der beiden folgenden Unterabschnitte ist.

#### 6.2.1 Normalisierung der Häufigkeiten des uans-Verfahrens

Zunächst wird die Summe aller Häufigkeiten und jede auftretende Häufigkeit durch eine Potenz von Zwei approximiert:

<pre>FUNCTION NEARESTPO2(uint32 x)   u = 2; b = 1;   WHILE(x &gt; 1) x &gt;&gt;= 1; u &lt;&lt;= 1;   END WHILE   b = u » 1;   return u-x &gt; x - b ? b : u; END FUNCTION</pre>	<pre>FUNCTION NORMALIZE()   y = NEARESTPO2(SUM);   WHILE((S=getSUM()) != y)     IF(S &gt; y)       MAX = getMaximalFrequency();MAX » 1;     ELSE       MIN = getMinimalFrequency();MIN « 1;     END WHILE   END FUNCTION</pre>
---	--

Anschließend müssen die Häufigkeiten derart verändert werden, dass ihre Summe mit der Approximation der ursprünglichen Summe wieder übereinstimmt. Aus Gründen der Performance und der Konsistenz werden hierbei nur echt positive Häufigkeiten berücksichtigt: Solange die Summe der Häufigkeiten nicht mit der Approximation der Summe übereinstimmt wird je nach Wert die maximale Häufigkeit halbiert oder die minimale Häufigkeit verdoppelt.

<sup>6</sup> Hierfür wird die unter [15] zu Verfügung gestellte Bibliothek verwendet.

<sup>7</sup> Die Bestimmung der Häufigkeiten erfolgt parallel, bei der Programmierung der parallelen Algorithmen orientiert sich der Autor an [14].

### 6.2.2 Normalisierung der Häufigkeiten des rans-Verfahrens

Wie in [1] vorgeschlagen, werden die Häufigkeiten zunächst derart skaliert, dass für ihre Summe  $M = 2^{11}$  gilt. Die Normalisierung erfolgt analog zur Normalisierung des uans-Verfahren, hierbei werden nur der Rechts- bzw. Linksshift durch eine Dekrementierung bzw. Inkrementierung durch eins ersetzt.

## 6.3 Wahl des normalisierten Intervalls

Für beide Verfahren werden unterschiedliche Intervalle gewählt. Für das rans-Verfahren erfolgt die feste Wahl von  $b = 2^{16}$  und  $L = 2^{13}$  während bei dem uans-Verfahren  $b = 2$  gewählt wird und  $L$  abhängig von Approximation der Summe der Häufigkeiten ist, also erst zur Laufzeit (vgl. 4.5) bestimmt wird.<sup>8</sup>

## 6.4 Indexmengenbildung des uans-Verfahren

Wie bereits besprochen basiert das Verfahren auf der Generierung von Indexmengen. Hierfür müssen zunächst Symbole mit gleicher Häufigkeit zusammengefasst werden. Hierfür empfiehlt sich folgender Datentyp:

```
struct SymbolsWithSameFrequency
{
    uint32 Fs; //the frequency
    std::vector<Symbol> m_Symbols;
    std::vector<uint32> m_ModuloRests;
    //...
};
```

In der Implementierung wird der die Moduloreste enthaltende `std::vector<uint32>` vorinitialisiert, also z.B mit  $0, 1, 2, \dots, 2^m - 1$  für die Häufigkeit  $F_s = 2^m$ . Anschließend werden die Indexmengen gemäß Algorithmus 4.3.1 generiert:

Zunächst werden alle ermittelten `SymbolsWithSameFrequency` nach wachsender Häufigkeit sortiert:  $S_0, S_1, S_2 \dots$

Angenommen  $S_i$  verfügt über  $n_i$  Symbole und die Häufigkeit  $F_i$ , dann werden  $n_i$  Moduloreste  $r_{n_i}$  gewählt und mit diesen Datentypen der Form

```
class IndexSubset
{
private:
    uint32 m_Frequency;
    uint32 m_ModuloRest;
    Symbol m_Symbol;
    std::vector<uint32> m_Indices;
```

---

<sup>8</sup> Auch hier ist eine Skalierung wie bei der Umsetzung des rans-Verfahrens möglich.

```
\ \ . . .
};
```

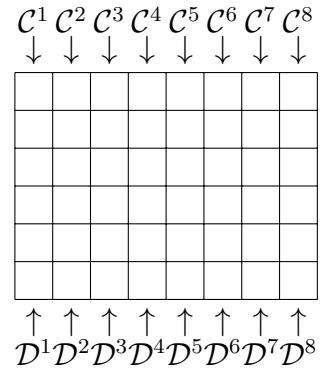
initialisiert. Die Indizes werden anschließend durch Auswertung der generierten arithmetischen Progression bestimmt. Für  $j > i$  werden dann sämtliche Moduloreste  $r_{n_j}$  aus  $S_j$  entfernt für die gilt

$$r_{n_j} - r_{n_i} \equiv 0 \pmod{F_i}.$$

Der Kompressor

## 6.5 Iterationswege der Kompressoren

Bei der Iteration über die Bilddatei stehen den Kompressoren drei mögliche Wege zur Parallelverarbeitung zu Verfügung: Zeilen-, spalten- und kanalweise. Nebenstehende Abbildung zeigt die spaltenweise Iteration: Die Kompressoren  $\mathcal{C}^i$  und die Dekompressoren  $\mathcal{D}^i$  kodieren bzw. dekodieren die  $i$ -ten Pixelreihen parallel. Allerdings ist die Wahl des Weges unerheblich und dient lediglich zur Überprüfung der Stabilität.



## 6.6 Endresultat der Kompression: Binäre Datei

Das Endergebnis der Kompression wird in einem binären Format festgehalten welches parallel zur gelesenen Bilddatei auf Platte geschrieben wird. Hierfür werden zunächst wichtige Parameter wie die Dimension des komprimierten Bildes, Vorkonditionierer, Anzahl der Elemente des Vorkonditioners, Kompressor und Iterationsweg geschrieben. Anschließend werden die normalisierten Häufigkeiten, das Ergebnis der Vorkonditionierung und die ermittelten Moduloreste geschrieben, wobei im Falle des `uans`-Verfahrens die ermittelten bits vor dem Schreibvorgang zu `uint8` Zahlen zusammengefasst werden.

**Beispiel 6.6.1:**

Wir ermitteln für nebenstehendes Bild das Quellalphabet mit zugehörigen Wahrscheinlichkeiten:

Symbol	Häufigkeit	Wahrscheinlichkeit
0	12646	0.0015075206756591797
1	8275	9.864568710327148E-4
2	14176	0.001689910888671875
3	20429	0.00243532657623291
4	27937	0.0033303499221801758
5	36130	0.0043070316314697266
6	42426	0.005057573318481445
20	83726	0.009980916976928711
21	83340	0.00993490219116211
22	83170	0.009914636611938477
23	82148	0.009792804718017578
24	82140	0.009791851043701172
25	81860	0.009758472442626953
26	80743	0.00962531566619873
27	79633	0.009492993354797363
28	79572	0.009485721588134766
29	78508	0.009358882904052734
30	78026	0.009301424026489258
31	76967	0.00917518138885498
32	75330	0.008980035781860352

Abbildung 2: Auszug: erste 50 Symbole



Abbildung 3: Blick von der Antechima in das untere Sarchetal.

Wir dekomprimieren nach obigem Verfahren und setzen sämtliche rgb-Werte auf r00.

Symbol	Häufigkeit	Wahrscheinlichkeit
33	73876	0.008806705474853516
34	72291	0.008617758750915527
35	70384	0.008390426635742188
36	68368	0.008150100708007812
37	66323	0.007906317710876465
38	64607	0.007701754570007324
39	62227	0.007418036460876465
40	59856	0.0071353912353515625
41	57758	0.0068852901458740234
42	55613	0.006629586219787598
43	53136	0.0063343048095703125
44	50396	0.006007671356201172
45	48412	0.005771160125732422
46	45429	0.005415558815002441
47	43096	0.005137443542480469
48	41076	0.004896640777587891
49	38609	0.004602551460266113
50	36859	0.004393935203552246

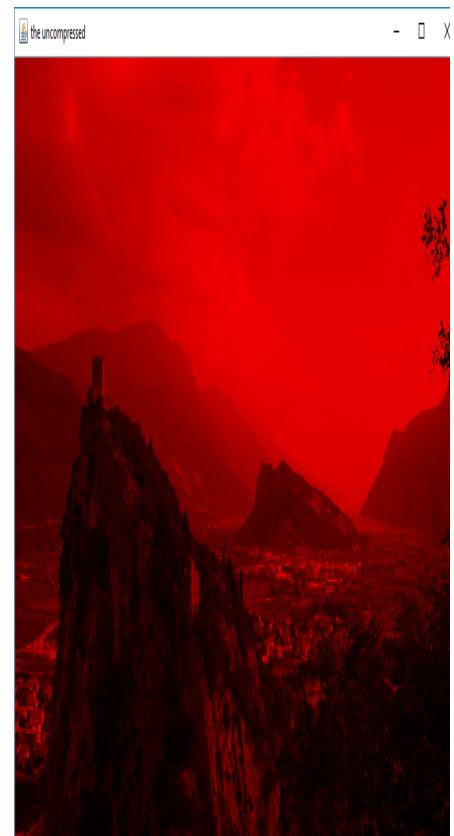


Abbildung 4: Auszug: erste 50 Symbole - Fortsetzung

## 7 Ergebnisse

Im Folgenden wird der Einfluss der Normalisierung und der Filterung auf Bilder demonstriert. Die Implementierung zur Visualisierung wurde in Java umgesetzt und steht unter <https://github.com/Hau-Bold/ImageProcessor> zu Verfügung.

### 7.1 Ergebnisse der Normalisierung

Im Folgenden wird der Einfluss der Normalisierung auf die Häufigkeiten eines Bildes (Siehe nächster Abschnitt Filtertyp `none`) demonstriert:

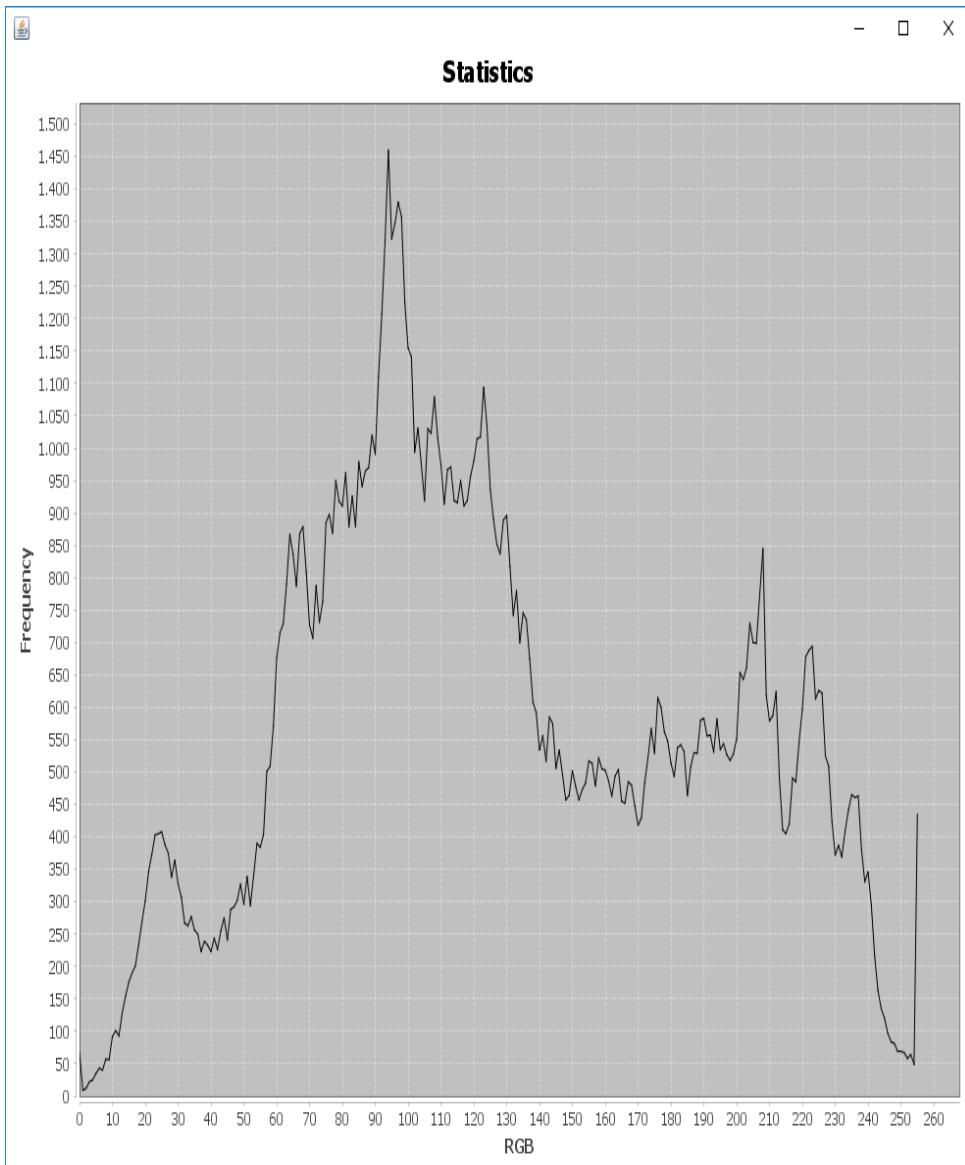


Abbildung 5: Ohne Normalisierung

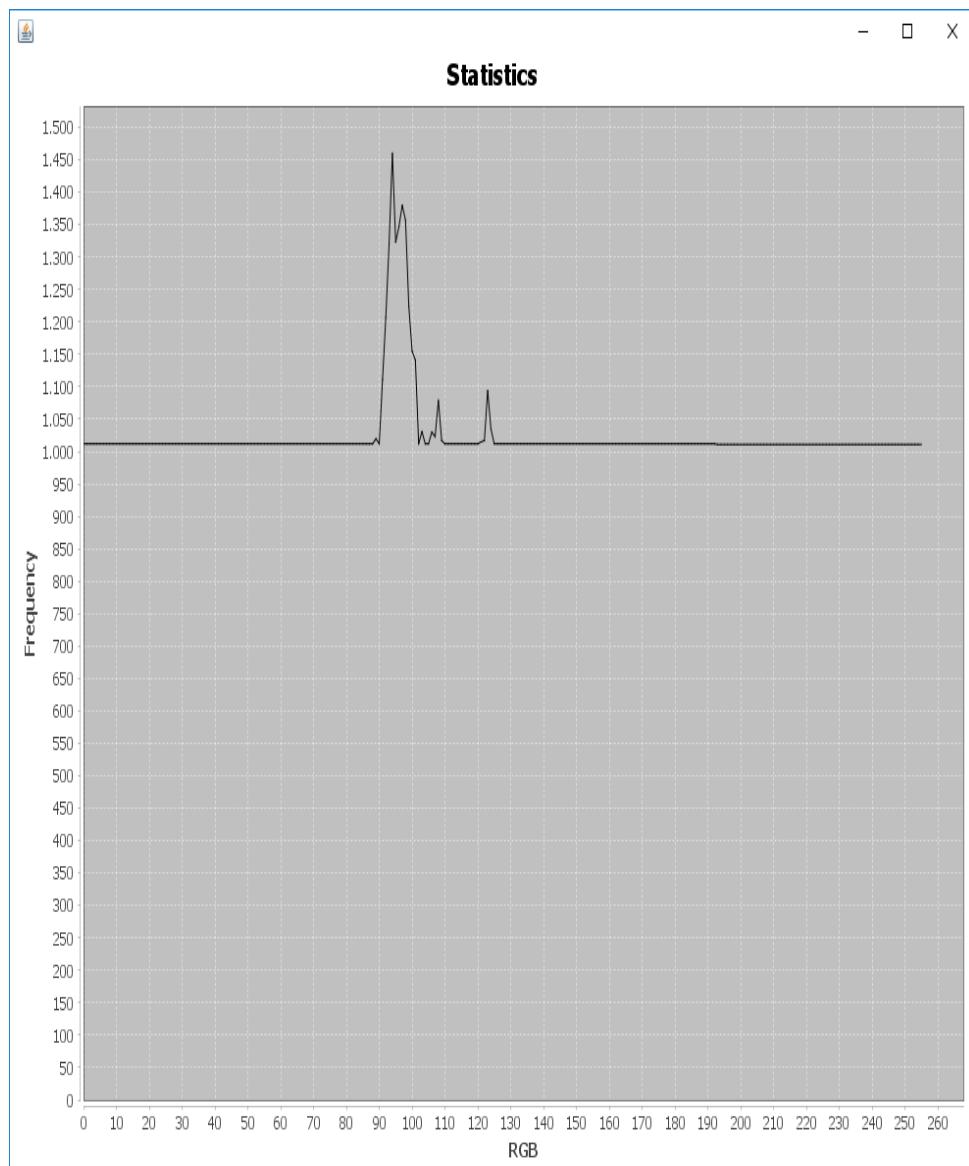


Abbildung 6: Normalisierung `rans`

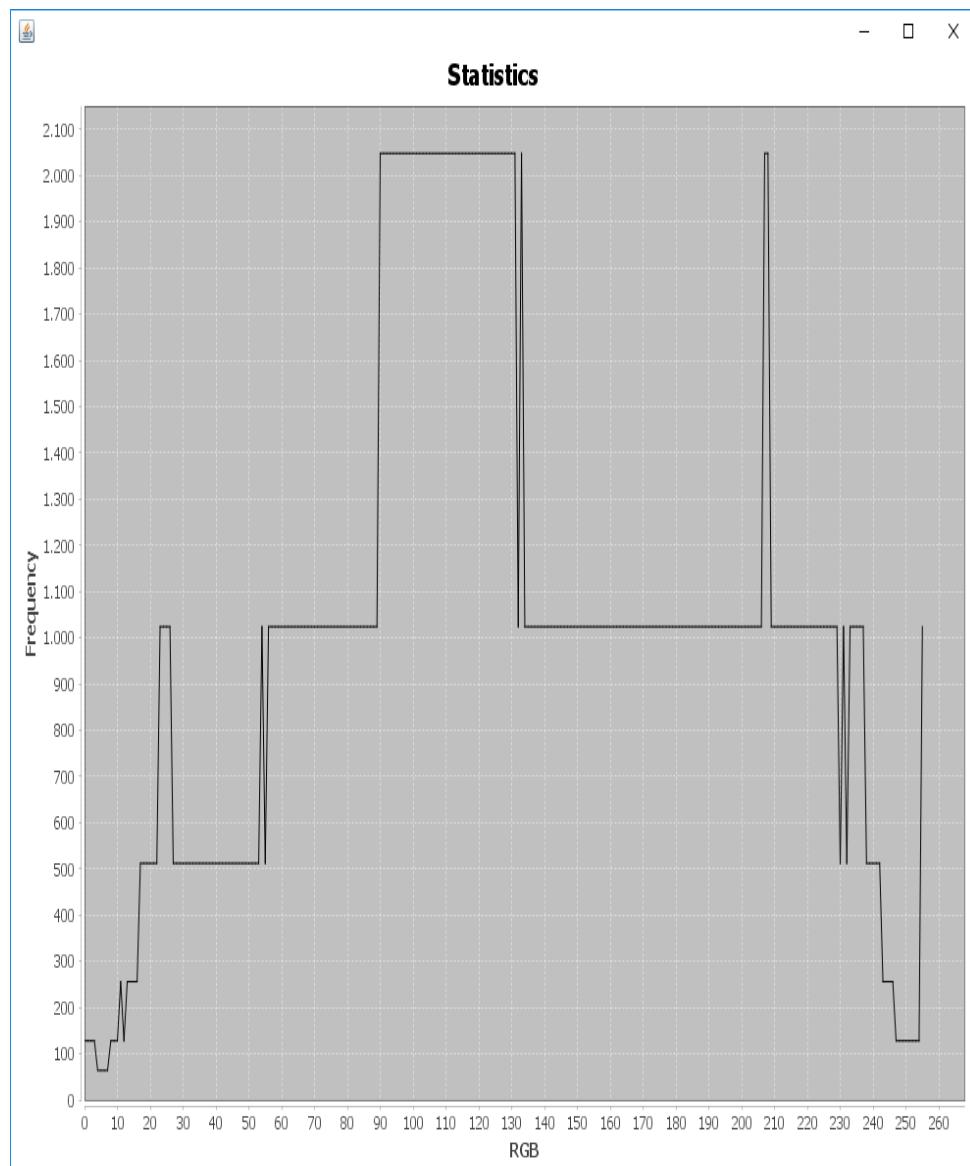


Abbildung 7: Normalisierung uans

## 7.2 Ergebnisse der Filterung

Im Folgenden wird der Einfluss der verwendeten Filter demonstriert. Die Werte der Entropie finden sich in Tabelle



Abbildung 8: **none**



Abbildung 9: **sub**



Abbildung 10: **up**



Abbildung 11: **average2**



Abbildung 12: **average3**



Abbildung 13: **average4**



Abbildung 14: **average5**

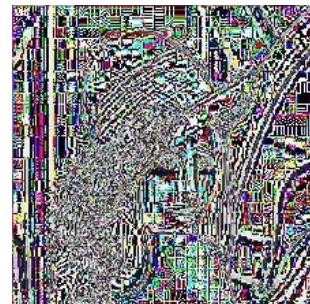


Abbildung 15: **average6**



Abbildung 16: **average7**

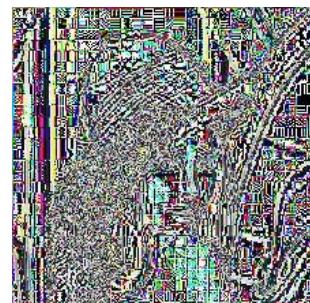


Abbildung 17: **average8**

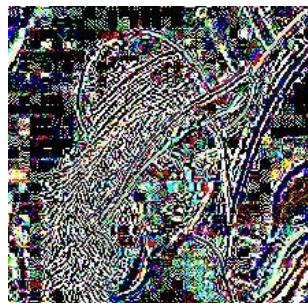


Abbildung 18: **paeth**

### 7.2.1 Verhalten der Entropie unter Verwendung der Filterung und Normalisierung für das uans-Verfahren

Abschließend geben wir das Verhalten der Entropie unter Verwendung einer Filterung und anschließender Normalisierung - also zur Laufzeit<sup>9</sup> - wieder. Zu Beginn beträgt die Entropie des betrachteten Bildes 7.766. Die zweite Spalte gibt den Wert der Entropie nach Verwendung eines Filters wieder, die dritte bzw. vierte Spalte dokumentiert die Entropie nach zusätzlicher Normalisierung

Filtertyp	nach Filterung	uans	rans
none	7.766	7.77295	7.8796
sub	5.49393	5.8999	6.24298
up	4.8265	5.04883	569749
average2	5.97444	6.21094	6.62404
average3	6.12365	6.4113	6.80125
average4	6.3044	6.5791	6.89326
average5	6.79384	6.93262	7.2642
average6	6.96592	7.05664	7.40429
average7	7.14682	7.21094	7.49495
average8	7.21671	7.3125	7.5623
paeth	4.59783	5.24121	5.38627

Tabelle 1: Werte der Entropie zur Laufzeit

## 7.3 Ergebnisse der Kompression

In diesem Unterabschnitt wird das Verhalten der Dateigröße unter Verwendung der beiden Verfahren angegeben. Zu Beginn beträgt die Größe des betrachteten Bildes 145200 Bytes.

---

<sup>9</sup> Die hier aufgelisteten Daten entstammen der C++ -Implementierung.

Filtertyp	uans	rans
none	144.263	
sub	102.400	
up	90.112	
average2	110.592	
average3	114.688	
average4	118.784	
average5	126.976	
average6	131.072	
average7	131.072	
average8	135.168	
paeth	86.016	

Tabelle 2: Verhalten der Dateigröße in Bytes

## 8 Kommandozeilenparameter

## 9 Tabellenverzeichnis

### Tabellenverzeichnis

1	Werte der Entropie zur Laufzeit . . . . .	41
2	Verhalten der Dateigröße in Bytes . . . . .	42

## 10 Abbildungsverzeichnis

### Abbildungsverzeichnis

1	Programmfluß der Implementierung . . . . .	3
2	Auszug: erste 50 Symbole . . . . .	34
3	Blick von der Antechima in das untere Sarchetal. . . . .	34
4	Auszug: erste 50 Symbole - Fortsetzung . . . . .	35
5	Ohne Normalisierung . . . . .	36
6	Normalisierung <code>rans</code> . . . . .	37
7	Normalisierung <code>uans</code> . . . . .	38
8	<code>none</code> . . . . .	39
9	<code>sub</code> . . . . .	39
10	<code>up</code> . . . . .	39
11	<code>average2</code> . . . . .	39
12	<code>average3</code> . . . . .	39
13	<code>average4</code> . . . . .	39
14	<code>average5</code> . . . . .	40
15	<code>average6</code> . . . . .	40
16	<code>average7</code> . . . . .	40
17	<code>average8</code> . . . . .	40
18	<code>paeth</code> . . . . .	40

## 11 Literaturverzeichnis

## Literatur

- [11] *Google Draco 3D compressor* <https://github.com/google/draco>
- [12] *PNG (Portable Network Graphics) Specification* <https://www.w3.org/TR/PNG-Filters.html>
- [13] Breymann U. *DER C++ Programmierer*, Carl Hanser Verlag München, (2015).
- [14] Williams A. *C++ Concurrency IN ACTION*, Manning Publications Co Shelter Island NY 11964 (2012).
- [15] *stb single-file public domain libraries for C/C++* <https://github.com/nothings/stb>

## **Erklärung:**

Die vorliegende Bachelorarbeit wurde am Institut für Informatik der Hochschule Coburg nach einem Thema von Herrn Prof. Dr. Quirin Meyer erstellt. Hiermit versichere ich, dass ich diese Arbeit selbstständig angefertigt und dazu nur die angegebenen Quellen verwendet habe.

Coburg, den 3. März 2019

Michael Krasser