HOCHSCHULE COBURG

Hochschule für angewandte Wissenschaften
Coburg

Institut für Informatik

Asymmetrische Zahlensysteme

Bachelorarbeit

von

Michael Krasser

Betreuer: Prof. Dr. Quirin Meyer

Coburg, 20. März 2019

Vorwort

Diese Arbeit entstand während des Wintersemesters 18/19. Für die Anregung und Ermutigung, mich mit diesem Thema auseinanderzusetzen, für die wertvollen Impulse und seine permanente Unterstützung möchte ich Herrn Prof. Dr. Quirin Meyer meinen besonderen Dank aussprechen.

Insbesondere möchte ich mich bei meiner Freundin Julia Isabel Schmidt und meinen lieben Eltern, Dr. Walter und Cornelia Krasser für die bedingungslose Unterstützung während dieses Studiums bedanken.

Inhaltsverzeichnis

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 gänge Entropiecodierer besprochen werden. Anschließend sollen die in [2], [3] und [1] angegebene Verfahren `uabs` und `rans` untersucht werden.
Ziel dieser Arbeit ist eine eigenständige Verallgemeinerung des `uabs`-Verfahrens auf n-elementige Symbolalphabete (`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[1].

Nachfolgende Abbildung veranschaulicht den Programmfluss:

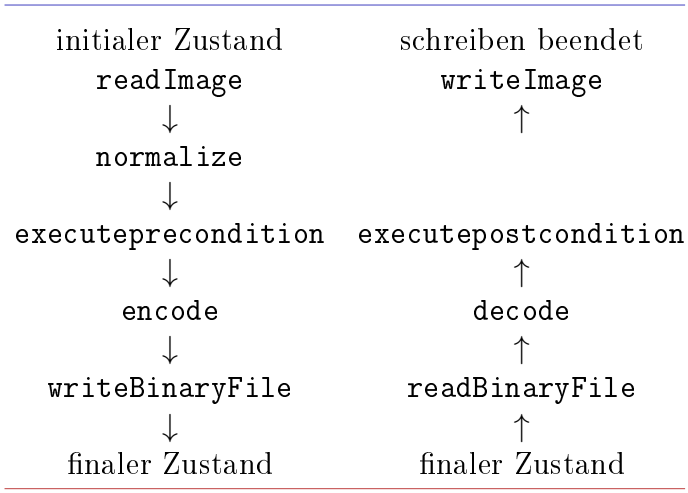


Abbildung 1: Programmfluss der Implementierung

In Leserichtung wird das Bild gelesen, vorkonditionert, normalisiert und anschließend komprimiert. Die Dekomprimierung erfolgt in umgekehrter Leserichtung.
In Kapitel 8 findet sich eine Anleitung zur Bedienung der C++ Implementierung.

[1] Unter `https://github.com/Hau-Bold/ANS_Java` befindet sich eine Umsetzung der Verfahren in Java (Swing)

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

Desweiteren definiert man zur Beurteilung der Qualität eines Komprimierungsschemas:

Definition 2.1.2:
Sei μ_c die Anzahl der Bits von X_c und μ die Anzahl der Bits von X. Dann heißt der durch μ_c/μ definierte Ausdruck Kompressionsquotient[2].

2.2 Über die Möglichkeit der Datenkompression

Vesteht man unter Datenkompression ein Verfahren, das eine bestimmte Repräsentation in eine andere überführt, die stets weniger Bits als die urspüngliche 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. Inkompressibilitätsargument 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[3]. 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. $\Rightarrow$ verschlüssele wahrscheinlichere Zeichen mit weniger Bit als unwahrscheinlichere (Morse-Alphabet).
- Blöcke von Zeichen lassen sich aufgrund der Abhängigkeiten aufeinander folgender Ereignisse geschickter gemeinsam als lose unabhängige Folge einzelner Zeichen verschlüsseln.

[2] Der Kehrwert des Kompressionsquotienten wird auch Kompressionsfaktor genannt.
[3] 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 $x_1, \ldots, x_N$ und dazugehörigen Wahrscheinlichkeiten $p_1, \ldots, p_N$.
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 $\Sigma = \{a_1, \ldots, a_k\}$ ein Alphabet mit zugehörigen Auftrittswahrscheinlichkeiten

$$\{P(a_i) \,|\, i = 1, \ldots, k\}.$$

Die Kodierung $\mathcal{K} : \Sigma \to \{0,1\}^{\mathbb{N}}$ 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 $x_1, \ldots, x_N$ nach fallender Auftrittswahrscheinlichkeit: $p_1 \geq p_2 \geq \ldots \geq p_N$.
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 $x_1, \ldots, x_N$ nach fallender Auftrittswahrscheinlichkeit: $p_1 \geq p_2 \geq \ldots \geq p_N$
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 $A = 0$, $B = 111$, $C = 110$, $D = 10$

$$\underbrace{A}_{0} \quad \underbrace{D\,C\,B}_{1}$$
$$\underbrace{D}_{0} \quad \underbrace{C\,B}_{1}$$
$$\underbrace{C}_{0} \quad \underbrace{B}_{1}$$

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: (Huffmann-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.

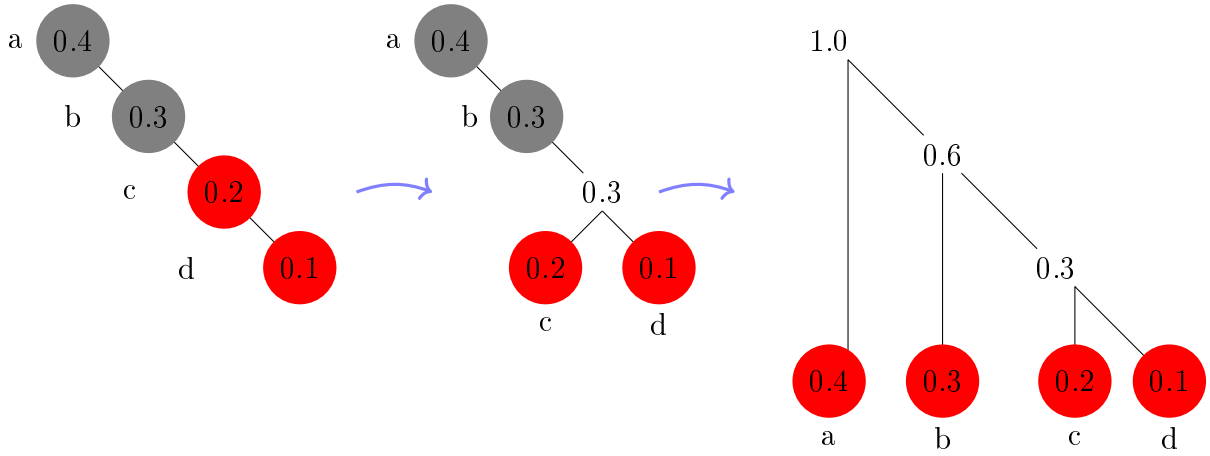
2) Lies für jedes Quellsymbol (Blatt im Baum) das Codewort aus: Beginne an der Wurzel des Baums. Die Codezeichen auf den Kanten des Pfades (in dieser Reihenfolge) ergeben das zugehörige Codewort.

Beispiel 2.3.3:
Die Nachricht *aababcabcd* soll auf Basis des Codealphabets $\mathcal{C} = \{0, 1\}$ kodiert werden. Zunächst erhält man

$$\mathcal{A} = \{a, b, c, d\} \quad \text{mit}$$

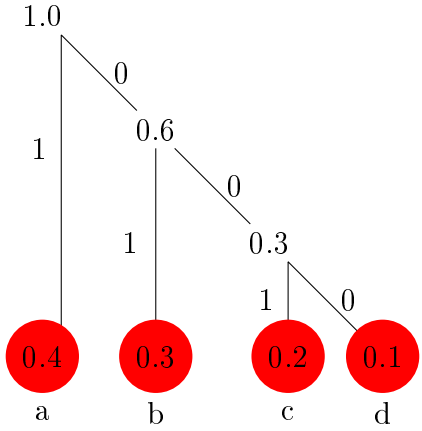
X	a	b	c	d
P(X)	0.4	0.3	0.2	0.1

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.[4] In die-

[4] 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:

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

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

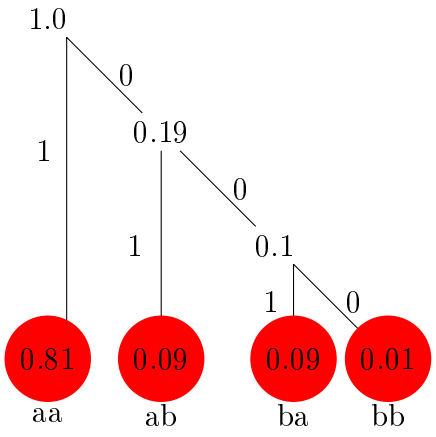
mit erwarteter Länge 1 Bits / Symbol und Entropie $H(\Sigma) = 0.47$. Die Redundanz $r = 0.53$ beträgt also 113% der Entropie, d.h das der Code nicht komprimiert ist. Wir betrachten statt dessen das Alphabet

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

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

$H(\Sigma^2) = 0.937991187$ Bits / Symbol
$L((K)) = 1.29$ Bits / Symbol



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

Hilfssatz 2.3.1:
Für jedes Quellalphabet Σ 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 Quellensymbole 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 u1u2u3u4 $\in \Sigma$ ein Text. Berechne für die Folge eine numerische Repräsentation – einen Bruch zwischen 0 und 1, und für diese Repräsentation den binären Code.
Sei $\Sigma = \{a_1, a_2, \ldots, a_N, \}$ das N-elementige Quellalphabet mit zugehörigen Auftrittswahrscheinlichkeiten $p(a_1), \ldots, p(a_N)$ und $\mathfrak{t} := a_{i_1} a_{i_2} \ldots a_{i_m}$ ein Text. Für $i = 0, \ldots N$ definiert man die kumulierte Wahrscheinlichkeit

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

Eine *numerische Representation* von $\mathfrak{t}$ ist ein Bruch im Interval $[l^m, u^m,)$ wobei

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

und

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

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 $\mathfrak{t} = abb$. 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 benötig, 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

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

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 $l(a_{i_1} a_{i_2} \dots a_{i_m})$ höchstwertige 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 Entropiereduktion 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]:

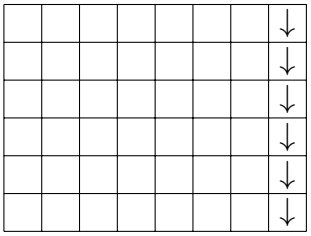
3.1 Verwendete Filter

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.



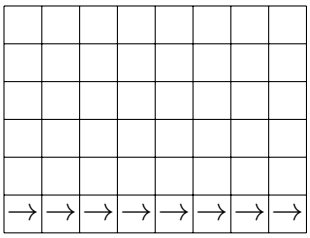
Verschlüsselung	Entschlüsselung
PROCEDURE ENCODE(`img`,`pre`,`res`)	PROCEDURE DECODE(`pre`,`post`,`res`)
new buffer; $x = 1$; $y = 0$; $k = 0$;	`post`.resize($(h-1) * (w-1) * c$);
`pre`.resize($(h-1) * (w-1) * c$);	$x = w - 1$; $y = h - 1$; $k = c - 1$;
`WHILE`($k < c$)	`WHILE`($k > 0$)
`WHILE`($y < h$)	buffer = `res`[k];
`WHILE`($x < w$)	`WHILE`($y > 0$)
$\texttt{pre}_{x,y,k}$[5] $\leftarrow \texttt{img}_{x,y,k} - \texttt{img}_{x-1,y,k}$;	$\texttt{post}_{x,y,k} \leftarrow$ buffer[y]
$x++$;	`WHILE`($x > 1$)

[5] Die Berechnung muss ohne Überlauf erfolgen.

Verschlüsselung	Entschlüsselung
END WHILE	$\texttt{post}_{x-1,y,k}{}^{4} \leftarrow \texttt{post}_{x,y,k} - \texttt{pre}_{x,y,k}$;
buffer.save($\texttt{img}_{w-1,y,k}$); $x = 1$;	$x--$;
$y++$;	END WHILE;
END WHILE	$y--$; $x = w - 1$;
$\texttt{res}[k]$ = buffer;	END WHILE
$k++$; $y = 1$;	$k--$;
END WHILE	END WHILE
END PROCEDURE	END PROCEDURE

3.1.3 Filtertyp 2: up

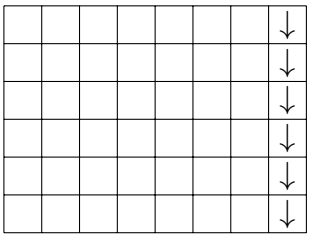
Jedes Pixel von img wird durch die Differenz zu seinem oberen Nachbarn ersetzt. Für jeden Farbkanal wird die untere Kante des Bildes gespeichert.



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

3.1.4 Filtertyp 3: `average2`

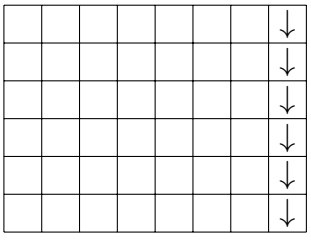
Jedes Pixel von `img` wird durch die Differenz zu dem Mittelwert aus seinem linken und oberen Nachbarn ersetzt. Für jeden Farbkanal wird die rechte Kante des Bildes gespeichert.



Verschlüsselung	Entschlüsselung
PROCEDURE ENCODE(`img`,`pre`,`res`)	PROCEDURE DECODE(`pre`,`post`,`res`)
new buffer; $x = 0$; $y = 0$; $k = 0$;	`post`.resize($(h-1)*(w-1)*c$);
`pre`.resize($(h-1)*(w-1)*c$);	$x = w-1$;$y = 0$; $k = c-1$;
`WHILE`($k < c$)	`WHILE`($k > 0$)
`WHILE`($y < h$)	buffer = `res`[k];
`WHILE`($x < w$)	`WHILE`($y < h$)
$\texttt{pre}_{x,y,k}{}^4 \leftarrow \texttt{img}_{x,y,k} - 1/2 * (\texttt{img}_{x-1,y,k}$	$\texttt{post}_{x,y,k} \leftarrow$ buffer[y]
$+ \texttt{img}_{x,y-1,k})$;	`WHILE`($x >= 1$)
$x++$;	$\texttt{post}_{x-1,y,k}{}^4 \leftarrow \texttt{post}_{x,y,k} - 1/2 * (\texttt{post}_{x,y-1,k}$
`END WHILE`	$+ \texttt{pre}_{x,y,k})$;
buffer.save($\texttt{img}_{x,y,k}$);	$x--$;
$x = 0$; $y++$;	`END WHILE`;
`END WHILE`	$y++$;
`res`[k] = buffer;	$x = w-1$;
$k++$;	`END WHILE`
$y = 0$;	$k--$;
`END WHILE`	`END WHILE`
END PROCEDURE	END PROCEDURE

3.1.5 Filtertyp 4: `average3`

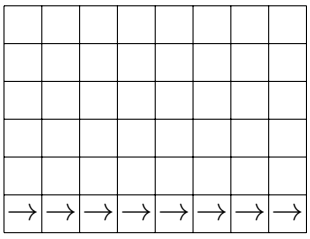
Jedes Pixel von `img` wird durch die Differenz zu dem Mittelwert aus seinem linken,oberen und rechten Nachbarn ersetzt. Für jeden Farbkanal wird die rechte Kante des Bildes gespeichert.



Verschlüsselung	Entschlüsselung
PROCEDURE ENCODE(`img`,`pre`,`res`)	PROCEDURE DECODE(`pre`,`post`,`res`)
new buffer; $x = 0$; $y = 0$; $k = 0$	`post`.resize($(h-1)*(w-1)*c$);
`pre`.resize($(h-1)*(w-1)*c$);	$x = w-1$;$y = 0$; $k = c-1$
`WHILE`($k < c$)	`WHILE`($k > 0$)
`WHILE`($y < h$)	buffer = `res`[k];
`WHILE`($x < w$)	`WHILE`($y < h$)
$\texttt{pre}_{x,y,k}{}^4 \leftarrow \texttt{img}_{x,y,k} - 1/3 * (\texttt{img}_{x-1,y,k}$	$\texttt{post}_{x,y,k} \leftarrow$ buffer[y];
$+ \texttt{img}_{x,y-1,k} + \texttt{img}_{x+1,y,k})$;	`WHILE`($x > 1$)
$x++$;	$\texttt{post}_{x-1,y,k}{}^4 \leftarrow \texttt{post}_{x,y,k} - 1/3 * (\texttt{post}_{x,y-1,k}$
`END WHILE`	$+ \texttt{post}_{x+1,y,k} + \texttt{pre}_{x,y,k})$;
buffer.save($\texttt{img}_{x,y,k}$);	$x--$;
$x = 0$; $y++$;	`END WHILE`;
`END WHILE`	$y++$; $x = w-1$;
`res`[k] = buffer;	`END WHILE`
$k++$; $y = 0$;	$k--$;
`END WHILE`	`END WHILE`
END PROCEDURE	END PROCEDURE

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.

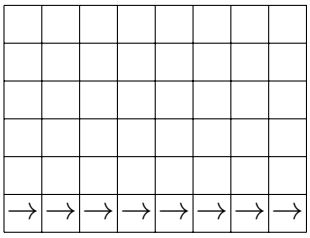


Verschlüsselung	Entschlüsselung
PROCEDURE ENCODE(`img`,`pre`,`res`)	PROCEDURE DECODE(`pre`,`post`,`res`)
new buffer; $x = 0$; $y = 0$; $k = 0$;	`post`.resize($(h-1)*(w-1)*c$);
`pre`.resize($(h-1)*(w-1)*c$);	$x = 0$; $y = h-1$; $k = c-1$;
`WHILE`($k < c$)	`WHILE`($k > 0$)
`WHILE`($x < w$)	buffer = `res`[k];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

Verschlüsselung	Entschlüsselung
`END WHILE`	$y--;\ x = 0;$
`res`$[k]$ = buffer;	`END WHILE`
$k++;\ x = 0;$	$k--;\ y = h - 1;$
`END WHILE`	`END WHILE`
END PROCEDURE	END PROCEDURE

3.1.7 Filtertyp 6: `average5`

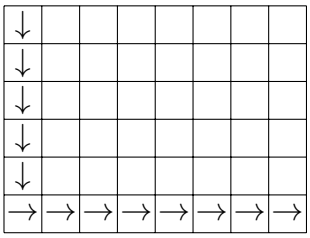
Jedes Pixel von `img` wird durch die Differenz zu dem Mittelwert aus seinem linken,links oberen, oberen, rechten und unteren Nachbarn ersetzt. Für jeden Farbkanal wird die untere Kante des Bildes gespeichert.



Verschlüsselung	Entschlüsselung
PROCEDURE ENCODE(`img`,`pre`,`res`)	PROCEDURE DECODE(`pre`,`post`,`res`)
new buffer; $x = 0$; $y = 0$; $k = 0$;	`post`.resize$((h-1)*(w-1)*c)$;
`pre`.resize$((h-1)*(w-1)*c)$;	$x = 0$; $y = h - 1$; $k = c - 1$;
`WHILE`$(k < c)$	`WHILE`$(k > 0)$
`WHILE`$(x < w)$	buffer = `res`$[k]$;
`WHILE`$(y < h)$	write(buffer);
$\texttt{pre}_{x,y,k}{}^4 \leftarrow \texttt{img}_{x,y,k} - 1/5 * (\texttt{img}_{x-1,y,k}$	`WHILE`$(y > 0)$
$+ \texttt{img}_{x-1,y-1,k} + \texttt{img}_{x,y-1,k}$	`WHILE`$(x < w)$
$+ \texttt{img}_{x+1,y,k} + \texttt{img}_{x,y+1,k})$;	$\texttt{post}_{x,y-1,k}{}^4 \leftarrow \texttt{post}_{x,y,k} - 1/5 * (\texttt{post}_{x-1,y,k}$
$y++;$	$+ \texttt{post}_{x-1,y-1,k} + \texttt{post}_{x+1,y,k}$
`END WHILE`	$+ \texttt{post}_{x,y+1,k} + \texttt{pre}_{x,y,k})$;
buffer.save$(\texttt{img}_{x,y,k})$;	$x++;$
$y = 0;\ x++;$	`END WHILE`;
`END WHILE`	$y--;\ x = 0;$
`res`$[k]$ = buffer;	`END WHILE`
$k++;\ x = 0;$	$k--;\ y = h - 1;$
`END WHILE`	`END WHILE`
END PROCEDURE	END PROCEDURE

3.1.8 Filtertyp 7: `average6`

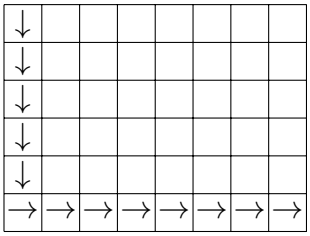
Jedes Pixel von `img` wird durch die Differenz zu dem Mittelwert aus seinem linken, links oberen, oberen, rechts oberen, rechten und unteren Nachbarn ersetzt. Für jeden Farbkanal wird die linke und die untere Kante des Bildes gespeichert.



Verschlüsselung	Entschlüsselung
PROCEDURE ENCODE(`img`,`pre`,`res`)	PROCEDURE DECODE(`pre`,`post`,`res`)
new buffer; $x = 0$; $y = 0$; $k = 0$;	`post`.resize($(h-1) * (w-1) * c$);
`pre`.resize($(h-1) * (w-1) * c$);	$x = 0$; $y = h-1$; $k = c-1$;
`WHILE`($k < c$)	`WHILE`($k > 0$)
`res`$[k]$ = Process(buffer);	buffer = `res`$[k]$;
buffer.clear();	write(buffer);
`WHILE`($x < w$)	`WHILE`($y > 0$)
`WHILE`($y < h$)	`WHILE`($x < w$)
$\texttt{pre}_{x,y,k}{}^4 \leftarrow \texttt{img}_{x,y,k} - 1/6 * (\texttt{img}_{x-1,y,k}$	$\texttt{post}_{x+1,y-1,k}{}^4 \leftarrow \texttt{post}_{x,y,k} - 1/6 * (\texttt{post}_{x-1,y,k}$
$+ \texttt{img}_{x-1,y-1,k} + \texttt{img}_{x,y-1,k}$	$+ \texttt{post}_{x-1,y-1,k} + \texttt{post}_{x,y-1,k}$
$+ \texttt{img}_{x+1,y-1,k} + \texttt{img}_{x+1,y,k}$;	$+ \texttt{post}_{x+1,y,k} + \texttt{post}_{x,y+1,k}$
$+ \texttt{img}_{x,y+1,k}$);	$+ \texttt{pre}_{x,y,k}$);
$y++$;	$x++$;
`END WHILE`	`END WHILE`;
$x++$; $y = 0$;	$y--$; $x = 0$;
`END WHILE`	`END WHILE`
$k++$; $x = 0$;	$k--$; $y = h-1$;
`END WHILE`	`END WHILE`
END PROCEDURE	END PROCEDURE

3.1.9 Filtertyp 8: `average7`

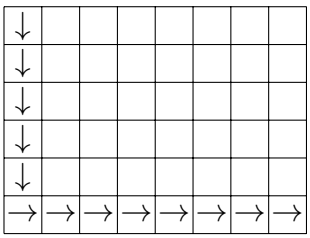
Jedes Pixel von `img` wird durch die Differenz zu dem Mittelwert aus seinem linken, links oberen, oberen, rechts oberen, rechten, rechts unteren und unteren Nachbarn ersetzt. Für jeden Farbkanal wird die linke und die untere Kante des Bildes gespeichert.



Verschlüsselung	Entschlüsselung
PROCEDURE ENCODE(`img`,`pre`,`res`)	PROCEDURE DECODE(`pre`,`post`,`res`)
new buffer; $x = 0$; $y = 0$; $k = 0$;	`post`.resize($(h-1)*(w-1)*c$);
`pre`.resize($(h-1)*(w-1)*c$);	$x = 0$; $y = h-1$; $k = c-1$;
`WHILE`($k < c$)	`WHILE`($k > 0$)
`res`[k] = Process(buffer);	buffer = `res`[k];
buffer.clear();	write(buffer);
`WHILE`($x < w$)	`WHILE`($y > 0$)
`WHILE`($y < h$)	`WHILE`($x < w$)
$\texttt{pre}_{x,y,k}{}^4 \leftarrow \texttt{img}_{x,y,k} - 1/7 * (\texttt{img}_{x-1,y,k}$	$\texttt{post}_{x+1,y-1,k}{}^4 \leftarrow \texttt{post}_{x,y,k} - 1/7 * (\texttt{post}_{x-1,y,k}$
$+ \texttt{img}_{x-1,y-1,k} + \texttt{img}_{x,y-1,k}$	$+ \texttt{post}_{x-1,y-1,k} + \texttt{post}_{x,y-1,k}$
$+ \texttt{img}_{x+1,y-1,k} + \texttt{img}_{x+1,y,k}$	$+ \texttt{post}_{x+1,y,k} + \texttt{post}_{x+1,y+1,k}$
$+ \texttt{img}_{x+1,y+1,k} + \texttt{img}_{x,y+1,k})$;	$+ \texttt{post}_{x,y+1,k} + \texttt{pre}_{x,y,k})$;
$y++$;	$x++$;
`END WHILE`	`END WHILE`;
$x++$; $y = 0$;	$y--$; $x = 0$;
`END WHILE`	`END WHILE`
$k++$; $x = 0$;	$k--$; $y = h-1$;
`END WHILE`	`END WHILE`
END PROCEDURE	END PROCEDURE

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.

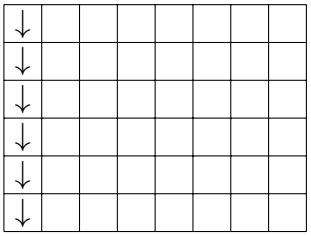


Verschlüsselung	Entschlüsselung
PROCEDURE ENCODE(`img`,`pre`,`res`)	PROCEDURE DECODE(`pre`,`post`,`res`)
new buffer; $x = 0$; $y = 0$; $k = 0$;	`post`.resize($(h-1)*(w-1)*c$);
`pre`.resize($(h-1)*(w-1)*c$);	$x = 0$; $y = h-1$; $k = c-1$;
`WHILE`($k < c$)	`WHILE`($k > 0$)
`res`[k] = Process(buffer);	buffer = `res`[k];
buffer.clear();	write(buffer);
`WHILE`($x < w$)	`WHILE`($y > 0$)
`WHILE`($y < h$)	`WHILE`($x < w$)
$\texttt{pre}_{x,y,k}{}^4 \leftarrow \texttt{img}_{x,y,k} - 1/8 * (\texttt{img}_{x-1,y,k}$	$\texttt{post}_{x+1,y-1,k}{}^4 \leftarrow \texttt{post}_{x,y,k} - 1/8 * (\texttt{post}_{x-1,y,k}$
$+ \texttt{img}_{x-1,y-1,k} + \texttt{img}_{x,y-1,k}$	$+ \texttt{post}_{x-1,y-1,k} + \texttt{post}_{x,y-1,k}$
$+ \texttt{img}_{x+1,y-1,k} + \texttt{img}_{x+1,y,k}$	$+ \texttt{post}_{x+1,y,k} + \texttt{post}_{x+1,y+1,k}$

Verschlüsselung	Entschlüsselung
$+ \texttt{img}_{x+1,y+1,k} + \texttt{img}_{x,y+1,k}$	$+ \texttt{post}_{x,y+1,k} + \texttt{post}_{x-1,y+1,k}$
$+ \texttt{img}_{x-1,y+1,k})$;	$+ \texttt{pre}_{x,y,k})$;
$y++$;	$x++$;
END WHILE	END WHILE;
$x++$; $y = 0$;	$y--$; $x = 0$;
END WHILE	END WHILE
$k++$; $x = 0$;	$k--$; $y = h - 1$;
END WHILE	END WHILE
END PROCEDURE	END PROCEDURE

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

Hierbei ermittelt der `paeth`-prediktor folgenden Wert

$$
\begin{aligned}
\texttt{paeth}(\texttt{img}_{x,y,k}) &:= \texttt{img}_{x-1,y,k} + \texttt{img}_{x,y-1,k} - \texttt{img}_{x-1,y-1,k};\\
\text{left} &:= \left|\texttt{img}_{x-1,y,k} - \text{PAETH}(\texttt{img}_{x,y,k})\right|;\\
\text{up} &:= \left|\texttt{img}_{x,y-1,k} - \text{PAETH}(\texttt{img}_{x,y,k})\right|;\\
\text{leftUp} &:= \left|\texttt{img}_{x-1,y-1,k} - \text{PAETH}(\texttt{img}_{x,y,k})\right|;\\
&\texttt{IF}(\text{left} \leq \text{up} \;\&\&\; \text{left} \leq \text{leftUp})\\
&\quad \texttt{return}\ \text{left};\\
&\quad \texttt{ELSE IF}(\text{up} \leq \text{leftUp})\\
&\quad\ \texttt{return}\ \text{up};\\
&\texttt{return}\ \text{leftUp};
\end{aligned}
$$

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

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

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 “

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

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 $I_s = \{k, k+1, \ldots, b\cdot k-1\}$, $k\geq 1$ (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
PROCEDURE ENCODE(x, buffer)	FUNCTION DECODE(buffer)
`WHILE` $\mathcal{C}(x,s)\notin I_s$ do	$(s,x)\leftarrow D(x)$
buffer.save(x mod b)	`WHILE` $x\notin I$ do
$x\leftarrow\lfloor x/b\rfloor$	$x\leftarrow b\cdot x+$ buffer.read()
`END WHILE`	`END WHILE`
$x\leftarrow\mathcal{C}(x,s)$	`return` s
END PROCEDURE	END FUNCTION

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. $I(L, b) = \{16, 17, \ldots, 31\}$ 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:

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

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
↑	16	↓
initialer Zustand		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 $\mathcal{P} := \{p_{i_k} \,|\, 1 \leq k \leq n-1\}$ 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 $0 <= r_k < 2^k$ und $r_l := \min_{r \in \mathbb{N}} \left\{(r_k - r) \bmod 2^k \not\equiv 0\right\}$. Folgende Mengen sind disjunkt[6]:

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

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

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

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 Moduloreste $R_{i_k} := \{0, 1, 2, \ldots, 2^{i_k} - 1\}$.

[6] Dieser Sachverhalt gilt für jede Primzahl.

4 Beginne bei $k = 0$.

5 Sei $k < l <= N - 1$. 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 $0 \equiv (x - y) \bmod 2^{i_k}$.

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 disjunkten aritmetischen 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	$\{0, 1, 2, 3\}$

$\overset{3}{\Rightarrow}$

a	1/2	$\{0\}$
b, c	1/4	$\{0, 1, 2, 3\}$

$\overset{4}{\Rightarrow}$

a	1/2	$\{0\}$
b, c	1/4	$\{\not{0}, 1, \not{2}, 3\}$

$\overset{5}{\Rightarrow}$

a	1/2	$\{0\}$
b, c	1/4	$\{1, 3\}$

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-eindeutigen 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:

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

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 $\left(\frac{1}{2}\right)^{i_k}$ 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:

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

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:

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

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

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

□

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 $M := \sum_{s \in \mathcal{A}} F_s$.

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

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:

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

Das Verfahren lässt sich folgendermaßen beschreiben:

Verschlüsselung	Entschlüsselung
PROCEDURE ENCODE(x, List)	FUNCTION DECODE(List)
new buffer	buffer = List.get()
`WHILE` $\mathcal{C}(x,s) \notin I$ do	$(s,x) \leftarrow D(x)$
buffer.save(x mod b)	$x \leftarrow b \cdot x +$ buffer.read()
$x \leftarrow \lfloor x/b \rfloor$	`return` s
`END WHILE`	END FUNCTION
List.add(buffer)	
$x \leftarrow \mathcal{C}(x,s)$	
END PROCEDURE	

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'
↑	21	↓
$x \notin I(L, b)$; save(1)		read()=1;
↑	43	↓
$x \notin I(L, b)$; save(0)		read()=0;
↑	86	↓
$x \notin I(L, b)$; save(0)		read()=0;
↑	172	↓
encode 'b'		decode 'b'
↑	31	↓
$x \notin I(L, b)$; save(0)		read()=0;
↑	62	↓
$x \notin I(L, b)$; save(0)		read()=0;
↑	124	↓
encode 'b'		decode 'b'
↑	22	↓
encode 'a'		decode 'a'
↑	17	↓
$x \notin I(L, b)$; save(1)		read()=1;
↑	35	↓
$x \notin I(L, b)$; save(1)		read()=1;
↑	71	↓
$x \notin I(L, b)$; 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:

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

Wählt man L^* derart, dass

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

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 $I_{i_k} \neq \varnothing$.
Eine weiterer Nachteil findet sich in dem Umstand, dass `uans`-Verfahren nicht immer terminiert:
Sei z.B. $b = 4$ und $L = 16$ und α 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 Kodiererer z.B. bei $x = 48$ das Symbol α liest, so teilt er so lange durch b bis er einen Wert aus I_α erhält, was nie der Fall ist.
Zusammenfassend ist das `rans`-Verfahren damit flexibler.
Im Falle der Terminierung liefert das `uans`-Verfahren aber die bessere Kompressionsrate:

Beispiel 4.5.1:
Man betrachte die Komprimierung des Wortes $\mathcal{T}$:=„MISSISSIPPI“ für $b = 2$, $L = 16$, $M = 8$ durch beide Verfahren:

$\mathcal{T} \overset{\texttt{uans}}{\mapsto}$ 00100000100101100101 10 $\qquad$ $\mathcal{T} \overset{\texttt{rans}}{\mapsto}$ 1|01|100|1|110|00|0|00|1|0010|19

Das `rans`-Verfahren benötigt zur Dekomprimierung von $\mathcal{T}$ zusätzliche Information. Der Grund diese Umstandes liegt in der Konstruktion: Das `rans`-Verfahren normalisiert nur bei der Kodierung aus positiver Richtung in das „normalisierte Interval“ $I(L, b)$, 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 Huffman-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 Anwendung auf Bilddateien

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

6.1 Bestimmung der Häufigkeiten

Zum Laden einer Bilddatei wird die Funktion [7]

```
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[8] 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:

```
FUNCTION NEARESTPO2(uint32 x)
u = 2; b = 1;
WHILE(x > 1) x >>= 1; u <<= 1;
END WHILE
b = u » 1;
return u-x > x - b ? b : u;
END FUNCTION
```

```
FUNCTION NORMALIZE()
y = NEARESTPO2(SUM);
WHILE((S=getSUM()) != y)
IF(S > y)
MAX = getMaximalFrequency() » 1;
ELSE
MIN = getMinimalFrequency() « 1;
END WHILE
END FUNCTION
```

Anschließend müssen die Häufigkeiten derart verändert werden, dass ihre Summe mit der Approximation der ursprünglichen Summe wieder übereinstimmt. Aus Gründen der Performanz 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 der Approximation die maximale Häufigkeit (MAX) halbiert oder die minimale Häufigkeit (MIN) verdoppelt.

[7] Hierfür wird die unter [15] zu Verfügung gestellte Bibliothek verwendet.

[8] 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 $L = 2^{16}$ wie unter [16] empfohlen, 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.[9]

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 $0, 1, 2, \ldots, 2^m - 1$ 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: $S_0, S_1, S_2 \ldots$.
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;
```

[9] Auch hier ist eine Skalierung wie bei der Umsetzung des `rans`-Verfahrens möglich.

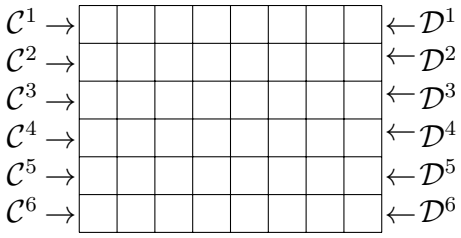
```
\\...
};
```

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

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

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 zeilenweise Iteration: Die Kompressoren $\mathcal{C}^i$ und die Dekompressoren $\mathcal{D}^i$ kodieren bzw. dekodieren die i-ten Pixelzeile parallel. Zunächst wurden alle drei Iterationswege implementiert, allerdings erwies sich die Wahl des Weges als unerheblich und wurde bei Tests nur zur Überprüfung der Stabilität benutzt. Die Iteration des Kompressors in der Implementierung erfolgt stets zeilenweise.



6.6 Beschränkung der Werte auf das normalisierte Intervall $I(L, b)$

6.6.1 `uans`-Verfahrens

Im Falle des `uans`-Verfahrens wird an der Stelle $x \in I(L, b)$ das Symbol α gelesen. Falls $x \in I_\alpha$ ist keine Normalisierung notwendig. Andernfalls erfolgt solange ein Rechtsshift um eine Stelle bis $x \in I_\alpha$. Nach Konstruktion gilt dann $x' = F_\alpha * x + r_\alpha \in I(L, b)$.

6.6.2 `rans`-Verfahrens

Im Fall des `rans`-Verfahrens erfolgt die Normalisierung und Denormalisierung auf das Normalisierte Intervall wie unter [2] empfohlen:

Normalisierung	Denormalisierung
FUNCTION NORMALIZE(x, buffer)	FUNCTION DENORMALIZE(x, buffer)
`WHILE` $\mathcal{C}(x, s) > L * b - 1$ `do`	`WHILE` $x < L$ `do`
buffer.save(x mod b)	$x \leftarrow b \cdot x +$ buffer.read()
$x \leftarrow \lfloor x/b \rfloor$	`END WHILE`
`END WHILE`	END FUNCTION
END FUNCTION	

Nachteilig gegenüber des `uans`-Verfahrens erwies sich hier die Notwendigkeit sich die Anzahl der Normalisierungsschritte (i.F Normalisierungstiefe) pro gelesenem Symbol einzeln abzuspeichern. Dieser Nachteil wurde bereits in Abschnitt 4.5 angedeutet und besitzt wesentlichen Einfluss auf die Größe der Datei nach Komprimierung. Jeglicher Versuch das Programm zu konfigurieren um ein Abspeichern der Normalisierungstiefe zu umgehen resultierte in einem nicht synchronen Datenkompressionschema.

6.7 Endresultat der Kompression: Binäre Datei

Die Komprimierung wird in einem binären Format festgehalten welches parallel der gelesenen Bilddatei auf Platte geschrieben wird. Hierfür werden zunächst Konfigurationsparameter wie die Dimension der Ausgangsdatei, Vorkonditionierer und Kompressor sowie wichtige Parameter wie Anzahl der Elemente des Vorkonditionerers 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. Im Falle des `rans`-Verfahrens müssen zusätzlich die ermittelteten Normalisierungstiefen geschrieben werden welche ebenfalls zu `uint8` Zahlen zusammengefasst werden.

7 Ergebnisse

Im Folgenden wird der Einfluss der Normalisierung und der Filterung auf Bilder demonstriert.

7.1 Ergebnisse der Normalisierung

Im Folgenden wird der Einfluss der Normalisierung auf die Häufigkeiten eines Bildes (Siehe nächster Abschnitt Filtertyp `none`) demonstriert. Die Implementierung zur Visualisierung wurde in Java umgesetzt und steht unter [17] zu Verfügung.

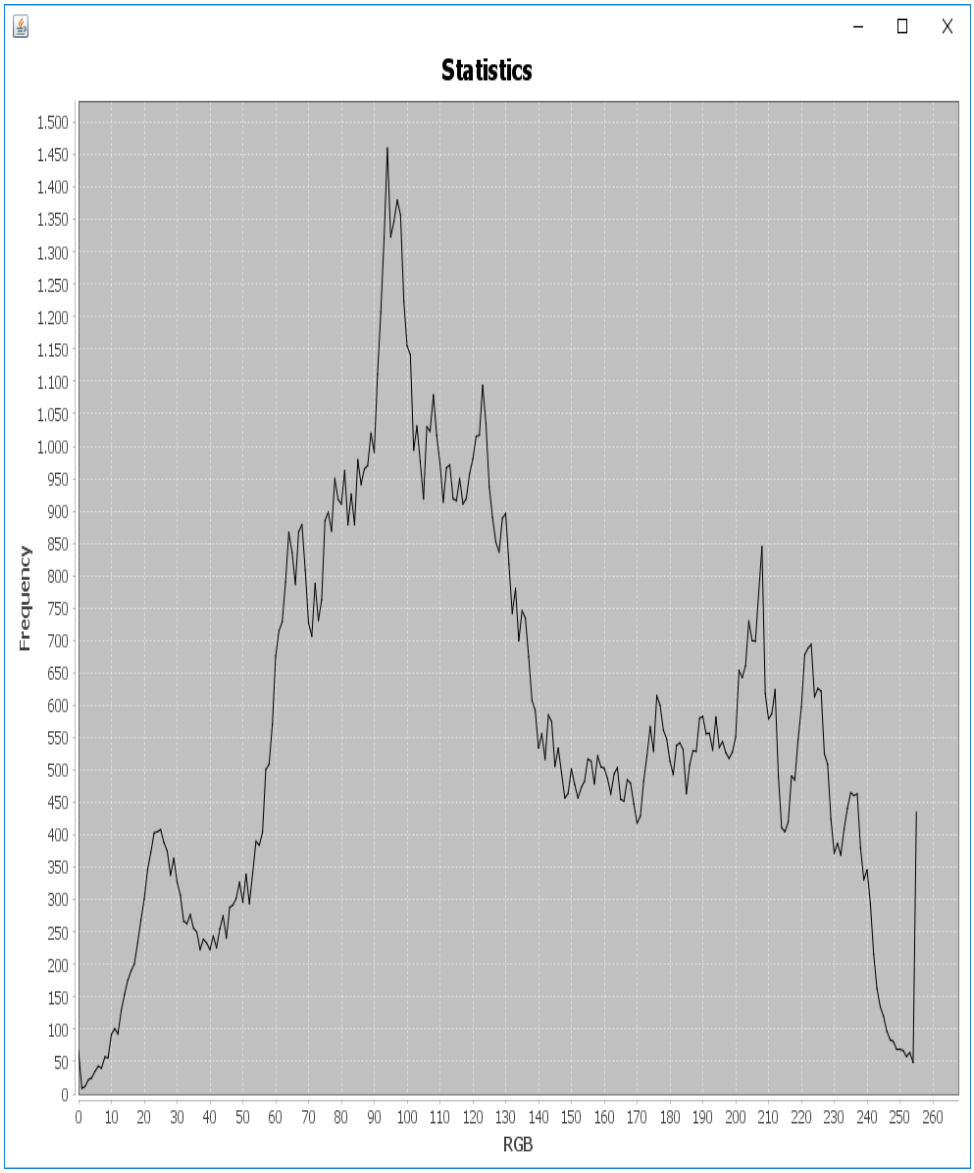


Abbildung 2: Ohne Normalisierung

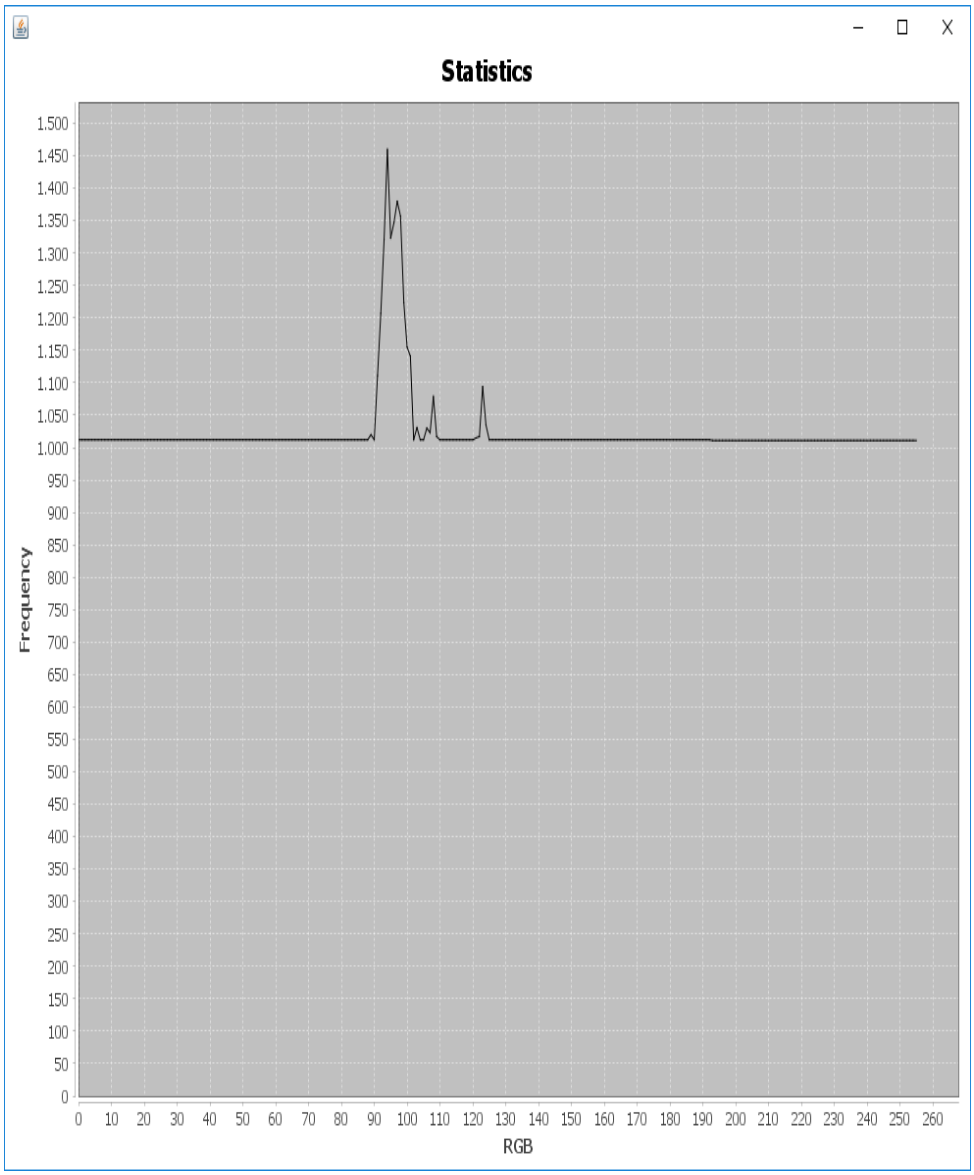


Abbildung 3: Normalisierung `rans`

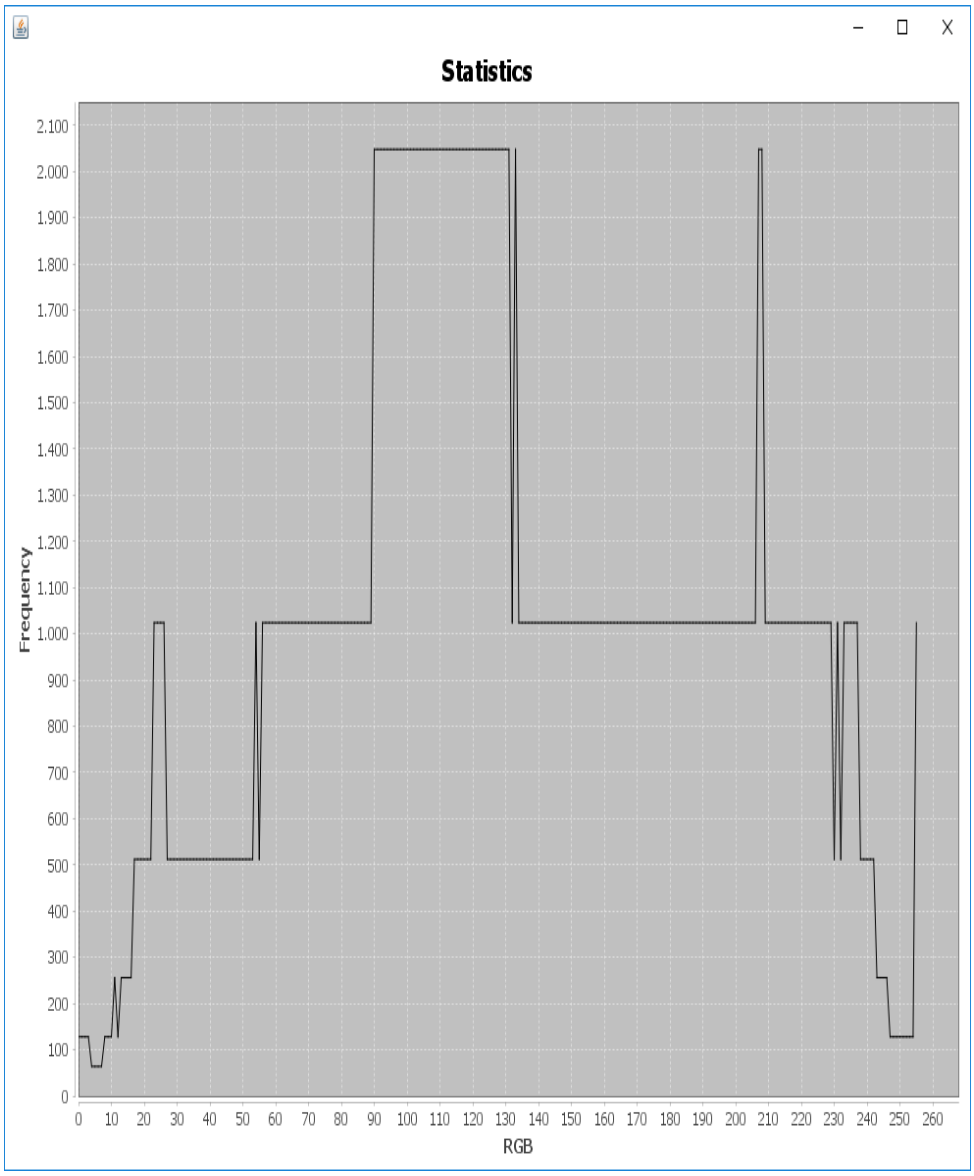


Abbildung 4: Normalisierung `uans`

7.2 Ergebnisse der Filterung

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

Abbildung 5: `none`

Abbildung 6: `sub`

Abbildung 7: `up`

Abbildung 8: `average2`

Abbildung 9: `average3`

Abbildung 10: `average4`

Abbildung 11: `average5`

Abbildung 12: `average6`

Abbildung 13: `average7`

Abbildung 14: `average8`

Abbildung 15: `paeth`

7.2.1 Verhalten der Entropie unter Verwendung der Filterung und Normalisierung

Abschließend geben wir das Verhalten der Entropie unter Verwendung einer Filterung und anschließender Normalisierung - also zur Laufzeit[10] - wieder. Im Grundzustand 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 bzgl. des verwendeten Verfahrens.

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.

Filtertyp	`uans`	`rans`
`none`	144.263	162.851
`sub`	102.400	123.737
`up`	90.112	111.921
`average2`	110.592	132.109
`average3`	114.688	134.987

Tabelle 2: Verhalten der Dateigröße in Bytes

[10] Die hier aufgelisteten Daten entstammen der C++ -Implementierung.

Filtertyp	`uans`	`rans`
`average4`	118.784	137.895
`average5`	126.976	146.629
`average6`	131.072	150.452
`average7`	131.072	153.400
`average8`	135.168	154.936
`paeth`	86.016	107.149

Tabelle 3: Verhalten der Dateigröße in Bytes (Fortsetzung)

Auffällig hierbei sind die schlechten Werte des `rans`-Verfahren: Lediglich bei den Filtertype 1–5 und 10 resultiert eine Anwendung des Verfahrens in einer Komprimierung während das `uans`-Verfahren in jedem Falle die Bilddatei komprimiert.
Interessant ist auch das Ergebnis unter Verwendung des `paeth`- und des `up`-Filters: Diese Filter liefern mit Abstand die beste Komprimierung.

Wie bereits besprochen musste im Falle des `rans`-Verfahrens die Normalisierungstiefe abgespeichert werden um die Bilddatei wiederherstellen zu können. Dieser Umstand liefert die schlechteren Werte des Verfahrens gegenüber `uans`. Ein Verzicht resultiert zwar in ähnlichen Zahlen, bietet aber derzeit keine Möglichkeit zur Dekomprimierung der Ausgangsdatei.

Abschließend hierfür sei noch das Ergebnis der Dekomprimierung unter dieser Voraussetzung gezeigt:

Abbildung 16: Ausgangsdatei

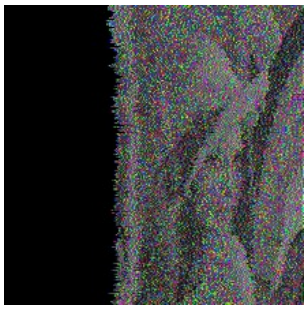


Abbildung 17: Ergebnis ohne Berücksichtigung der Normalisierungstiefe

Im Falle einer Abspeicherung der Normalisierungstiefe lässt sich die Ausgangsdatei wieder erfolgreich durch Dekomprimierung herstellen.

8 Kommandozeilenparameter

Abschließend wird ein Überblick über die Konfiguration der Kommandozeilenparameter für das Programm gegeben:
Der entgegenzunehmende Formatstring setzt sich folgendermaßen zusammen:

`Pfad Vorkonditionerer Kompressor (-v)`

- `Pfad`: Absoluter Pfad
- `Vorkonditionerer`: $n \in \{0, \ldots, 10\}$
  Die Ganzzahlen entsprechen den in Kapitel 3 besprochenen Filtern.
- `Kompressor`: $n \in \{1, 2\}$
  Das `rans`-Verfahren wird durch $n = 0$ selektiert, dass `uans`-Verfahren durch $n = 1$.
- `-v`:
  Der geklammerte Parameter (verbosing) ist optional. Bei Konkatenation werden spezifische Größen wie Entropie, Zeitdauer und Dateigröße während der Laufzeit des Verfahrens auf der Konsole ausgegeben.

9 Tabellenverzeichnis

Tabellenverzeichnis

10 Abbildungsverzeichnis

Abbildungsverzeichnis

11 Literaturverzeichnis

Literatur

[1] Krajcevski P., Pratapa S., Manocha D. *GST: GPU-decodable Supercompressed Textures*, Proc. of ACM SIGGRAPH Asia, (2016).

[2] Giesen F. *Interleaved entropy coders*, arXiv:1402.3392 , (2014).

[3] Duda J. *Asymmetric numeral systems: entropy coding combining speed of Huffman coding with compression rate of arithmetic coding*, CoRR, abs/1311.2540v2, (2014).

[4] Liśkiewicz M., Fernau H. *Datenkompression* https://www.uni-trier.de/fileadmin/fb4/prof/INF/TIN/Folien/DK/script.pd.

[5] Li M., Vitányi P.*An introduction to Kolmogorov complexity and its applications*, 2. Auflage, Springer, (1997).

[6] Lajmi L. *Informationstheorie und Codierung* https://www.ostfalia.de/export/sites/default/de/pws/lajmi/Lehre/Codierungstheorie/pdf/Quellencodierung_SS15.pdf.

[7] Rohling H. *Einführung in die Informations- und Codierungstheorie* B.G. Teubner Stuttgart, (1995).

[8] Huffman D. A. *A method for the construction of minimum-redundancy codes* Proceedings of the I.R.E.., S. 1098-1101, (1952).

[9] *Smaller and faster data compression with Zstandard* https://code.facebook.com/{\protect\protect\protect\edefT1{T1}\let\enc@update\relax\protect\edefcmr{cmr}\protect\edefm{m}\protect\edefn{n}\protect\xdef\T1/cmr/m/it/12{\T1/cmr/m/n/12}\T1/cmr/m/it/12\size@update\enc@update\ignorespaces\relax\protect\relax\protect\edefcmr{cmtt}\protect\xdef\T1/cmr/m/it/12{\T1/cmr/m/n/12}\T1/cmr/m/it/12\size@update\enc@update{post}}s/1658392934479273/smaller-and-faster-data-com{\protect\protect\protect\edefT1{T1}\let\enc@update\relax\protect\edefcmr{cmr}\protect\edefm{m}\protect\edefn{n}\protect\xdef\T1/cmr/m/it/12{\T1/cmr/m/n/12}\T1/cmr/m/it/12\size@update\enc@update\ignorespaces\relax\protect\relax\protect\edefcmr{cmtt}\protect\xdef\T1/cmr/m/it/12{\T1/cmr/m/n/12}\T1/cmr/m/it/12\size@update\enc@update{pre}}ssion-with-zstandard/

[10] *Apple Open-Sources its New Compression Algorithm LZFSE* https://www.infoq.com/news/2016/07/apple-lzfse-lossless-opensource

[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

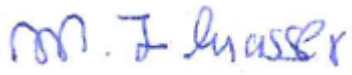
[16] *Range ANS (rANS) - direct alternative for range coding*https://encode.ru/threads/1870-Range-ANS-(rANS)-faster-direct-replacement-for-range-coding

[17] *ImageProcessor* https://github.com/Hau-Bold/ImageProcessor

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 20. März 2019



Michael Krasser