Masterarbeit
Informatik-Ingenieurwesen

Analyse und Optimierung von Kompressionsverfahren für Smart Metering mit Sensornetzen

von

Martin Ringwelski

November 2011

Betreut von

Christian Renner
Institut für Telematik, Technische Universität Hamburg-Harburg

Erstprüfer	Prof. Dr. Volker Turau Institut für Telematik Technische Universität Hamburg-Harburg
Zweitprüfer	Prof. Dr.-Ing. Andreas Timm-Giel Institute of Communication Networks Hamburg University of Technology

Institut für Telematik
Technische Universität Hamburg-Harburg
TUHH

Danksagung

Ich möchte mich bei allen Personen bedanken, die mir beim Erstellen dieser Arbeit geholfen haben und beratend zur Seite standen. Insbesondere bei meiner Frau, Jennifer Van, meinen Geschwistern, Laura und Dominik Ringwelski, und meinem Betreuer Christian Renner. Mein Dank geht auch an Herrn Andreas Reinhardt der mir zu Beginn meiner Arbeit den Squeeze.KOM-Layer hat zukommen lassen.

Eidesstattliche Erklärung

Ich, MARTIN RINGWELSKI (Student im Studiengang Informatik-Ingenieurwesen an der Technischen Universität Hamburg-Harburg, Matr.-Nr. 20624206), versichere an Eides statt, dass ich die vorliegende Masterarbeit selbständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskomission vorgelegt.

Hamburg, 1. November 2011

Martin Ringwelski

Inhaltsverzeichnis

Verwendete Symbole

$\mathcal{Z}$	Alphabet
z	Ein Zeichen aus dem Alphabet $\mathcal{Z}$
$\mathcal{C}$	Die Menge der Codes
c	Code aus der Menge der Codes $\mathcal{C}$
$\mathcal{K}$	Kodierung
$\mathcal{X}$	Nachrichtenquelle
l_z	Codewortlänge eines Symbols z
$L(\mathcal{K})$	Mittlere Codewortlänge einer Kodierung $\mathcal{K}$
$I(z)$	Informationsgehalt eines Zeichens z
$H(\mathcal{X})$	Entropie der Quelle
$R(\mathcal{K})$	Redundanz einer Kodierung $\mathcal{K}$
$H_M(\mathcal{X})$	Markov-Entropie einer Quelle
K_L	Kompressionsleistung

Kapitel 1

Einführung

Intelligente Stromzähler, sogenannte Smart Meter, sind für die Einführung von intelligenten Stromnetzen, auch Smart Grids genannt, von großer Bedeutung. Diese Smart Grids spielen eine wichtige Rolle für den weiteren Ausbau von regenerativen Energien. Da regenerative Energiequellen wie Wind- und Solarenergie nicht ständig zur Verfügung stehen, sollen Smart Grids dazu beitragen, die Energie dann zu nutzen, wenn gerade viel in das Netz eingespeist wird. Bei Versorgungsengpässen müssen schnell andere Kraftwerke eingeschaltet werden können. Auch aus diesem Grund ist es für die Energielieferanten wichtig, die Energieverbrauchsdaten schnell zur Verfügung zu haben. Bei Smart Metern geht es darum, den Verbrauch, z.B. von elektrischer Energie, nicht nur ein Mal im Jahr ablesen zu lassen, sondern diesen auch tages-, stunden- oder minutengenau messen zu können. Zum einen sollen dadurch Geräte im Haushalt für den Kunden besser sichtbar werden, die im Standby Zustand weiter viel Energie verbrauchen, zum anderen sollen durch angepasste Tarife die Kunden dazu animiert werden, Geräte wie Waschmaschinen zu Tageszeiten einzuschalten, in denen gerade viel Energie produziert wird. Damit für die Datenübertragung keine Verträge mit den Kunden geschlossen werden müssen und da eine direkte Funkübertragung durch die häufige Lage von Stromzählern in Kellern gestört ist, bietet sich die Übertragung mittels eines drahtlosen Sensornetzes an. Drahtlose Sensornetze werden häufig für die Datensammlung, auch Data-Gathering genannt, genutzt.

Da die direkte Kommunikation mit einer Zentrale meist nicht möglich ist, wird eine Multi-Hop-Kommunikation verwendet, bei der die Sensor-Informationen eines

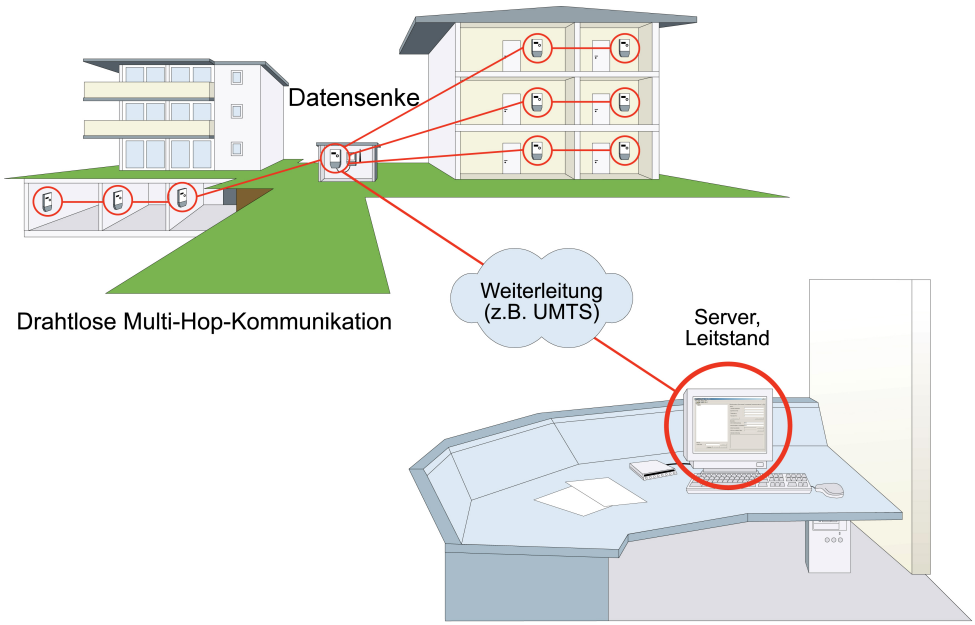


Abbildung 1.1.: Multi-Hop Data-Gathering Kommunikation der Smart Meter

Knotens an den nächsten per Funk übertragen werden. Dieser leitet die erhaltenen Informationen und seine eigenen Daten weiter an den nächsten Knoten. Am Ende dieser Kommunikationskette steht eine Datensenke, welche die Daten sammelt und über das Internet an die Zentrale der Energielieferanten weiterleitet. Abbildung 1.1 zeigt, wie so ein Netzwerk aussehen kann.

Die Datensenke stellt in der Kommunikationskette einen Flaschenhals dar, weil sie nur Daten von einem Knoten zu einer Zeit empfangen kann. Da eventuell hunderte Smart Meter-Knoten im Einzugsbereich einer Datensenke liegen, die übertragenen Daten bis zu 10 KByte groß sind und die Funkübertragung relativ langsam ist, bietet es sich an, die Zählerdaten zu komprimieren.

Die Knoten haben, verglichen mit einem modernen PC, nur einen geringen Arbeitsspeicher (RAM) und eine geringe Rechenleistung. Daher sind moderne Verfahren der Datenkompression, die oft intensive RAM- und Rechenleistungsansprüche haben, für Smart Meter nicht implementierbar. Bisherige Arbeiten zur Datenkomprimierung von drahtlosen Sensornetzen behandelten vornehmlich die Dekomprimierung von Firmwareupdates und hatten vor allem eine Reduktion des Stromverbrauchs der Knoten zum Ziel. Da Smart Meter mit dem Stromnetz verbunden sind, ist der Energiehaushalt der Knoten in dieser Arbeit von keiner großen Bedeutung.

In dieser Arbeit werden verschiedene Komprimierungsmethoden auf ihre Anwendbarkeit in Smart Metern analysiert und optimiert. Nach einer Vorstellung von bekannten Methoden werden die Stromzählerdaten analysiert. Es stehen dieser Arbeit 3500 Auslesungen aus einem Testnetzwerk eines Smart Meter-Herstellers als Referenz zur Verfügung. Durch eine Analyse der Daten werden mögliche Kompressionsleistungen und Besonderheiten der Daten ermittelt. Anschließend werden durch Anpassen an die Eigenschaften der Zählerdaten und die begrenzten Ressourcen verschiedene Verfahren für den Einsatz auf intelligenten Stromzählern entwickelt und implementiert, welche die Daten gut komprimieren und die vorhandenen Ressourcen optimal nutzen. Die Ergebnisse von aktuellen Kompressionsverfahren für Desktop-Computer dienten als Vergleich der Kompressionsleistung. Das in dieser Arbeit entwickelte Lempel-Ziv-Markov-Chain-Huffman-Verfahren (LZMH) erreicht ähnliche Kompressionsleistungen wie das LZMA-Verfahren, welches auf Desktop-Computern eines der besten Verfahren ist. Das Verfahren verwendet nur wenig RAM und hat eine Programmgröße von ca. 2 KByte. Dabei ist es weiterhin flexibel und müsste bei Erweiterungen des Stromzählerdatenformats nicht angepasst werden. Daher bietet sich das Verfahren besonders für die Verwendung auf Smart Metern an.

Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen der Datenkomprimierung besprochen. Bei der Komprimierung geht es darum, Daten in möglichst kurzer Form zu speichern oder zu übertragen, um Speicherplatz oder Bandbreite bei der Übertragung zu sparen. Dabei gibt es zwei grundlegende Formen der Komprimierung, die verlustfreie und die verlustbehaftete Komprimierung. Aus verlustfrei komprimierten Daten können die Originale wieder hergestellt werden, während Daten, die verlustbehaftet komprimiert wurden, nicht vollständig wieder hergestellt werden können. Verlustbehaftete Verfahren werden z.B. bei der Audio- und Videospeicherung verwendet. In dieser Arbeit werden nur verlustfreie Verfahren betrachtet, da eine Verfälschung der Stromzählerdaten bei der verlustbehafteten Komprimierung zu falschen Rechnungen und falschen Stromanforderungen führen könnte.

2.1. Formale Definitionen

Nachrichten bestehen aus Symbolen $z \in \mathcal{Z}$. $\mathcal{Z}$ ist das Alphabet der möglichen Symbole, die aus einer Nachrichtenquelle $\mathcal{X}$ kommen können. Für die Verarbeitung in digitalen Systemen müssen die Symbole z in binäre Codeworte $c \in \mathcal{C} = \{b_1 b_2 \cdots b_l | b_i \in \{0,1\}, l \in \mathbb{N}\}$ übertragen werden, wobei die Codeworte nicht gleichlang sein müssen. Eine Kodierung $\mathcal{K}$ wird durch folgende Abbildungsvorschrift beschrieben:

$$\mathcal{K}\colon z \to c\,, \qquad z \in \mathcal{Z}, c \in \mathcal{C} \tag{2.1}$$

Handelt es sich um eine verlustfreie Kodierung, so ist die Abbildung injektiv.

Für die einfache Verarbeitung werden in der Regel gleich lange Codewörter bestimmt. Die Mindestlänge dieser binären Codewörter für $|\mathcal{Z}|$ Symbole ergibt sich aus:

$$l_{\mathcal{Z}} = \lceil \log_2 |\mathcal{Z}| \rceil \tag{2.2}$$

Beispiel: Für ein Alphabet $\mathcal{Z} = \{a, b, c, d, e\}$ könnte beispielsweise folgende Kodierung gewählt werden:

Symbol	a	b	c	d	e
Codewort	000	001	010	011	100

Die Codewortlänge l_z eines Symbols z beträgt in dieser Beispielkodierung für jedes Symbol 3 Bit.

2.1.1. Entropie und Redundanz

Sind für eine Nachrichtenquelle die verschiedenen Auftrittswahrscheinlichkeiten der Symbole p_z gegeben, kann die mittlere Codewortlänge $L(\mathcal{K})$ für die Kodierung $\mathcal{K}$ berechnet werden:

$$L(\mathcal{K}) = \sum_{z \in \mathcal{Z}} l_z \cdot p_z \tag{2.3}$$

Der Informationsgehalt $I(z)$ eines Symbols z errechnet sich aus:

$$I(z) = -\log_2 p_z \tag{2.4}$$

Der Informationsgehalt eines Symbols gibt an, wie viele Bit mindestens nötig sind, um ein Symbol zu kodieren, wenn für die Kodierung die Wahrscheinlichkeit der Symbole berücksichtigt wird. Symbole, die häufig vorkommen, haben einen niedrigen Informationsgehalt und könnten mit weniger Bit kodiert werden. Selten vorkommende

Symbole haben einen hohen Informationsgehalt und sollten daher mit mehr Bit kodiert werden.

Wird der Informationsgehalt eines Symbols als dessen Codewortlänge in die Formel der mittleren Codewortlänge eingesetzt, ergibt sich die Entropie $H(\mathcal{X})$ der Nachrichtenquelle $\mathcal{X}$:

$$H(\mathcal{X}) = \sum_{z \in \mathcal{Z}} I(z) \cdot p_z \tag{2.5}$$

Die Entropie $H(\mathcal{X})$ einer Nachrichtenquelle ist deren mittlerer Informationsgehalt und beschreibt die Anzahl der Bit, die im Mittel für jedes Symbol nötig ist, um die Nachricht ohne Informationsverlust zu kodieren.

Die Redundanz $R(\mathcal{K})$ eines Codes zeigt die Differenz der verwendeten Kodierung zur Entropie, also wie viele Bits in der Kodierung verschwendet werden:

$$R(\mathcal{K}) = L(\mathcal{K}) - H(\mathcal{X}) \tag{2.6}$$

Beispiel: Gegeben sind für das Alphabet $\mathcal{Z}$ die Auftrittswahrscheinlichkeiten $p_a = 0,4$, $p_b = p_c = p_d = p_e = 0,15$. Für die betrachtete Kodierung ergibt sich eine mittlere Codewortlänge von $L(\mathcal{K}) = 3$ Bit.

Für die gegebenen Auftrittswahrscheinlichkeiten ergeben sich die Informationsgehalte $I(a) = 1,3219$ Bit und $I(b) = I(c) = I(d) = I(e) = 2,7370$ Bit. Die Nachrichtenquelle hat somit eine Entropie von $H(\mathcal{X}) = 2,1710$ Bit. Daraus ergibt sich eine Redundanz der Kodierung von $R(\mathcal{K}) = 0,8290$ Bit.

2.2. Entropie-Kodierung

Bei der Entropie-Kodierung werden Symbole durch unterschiedlich lange Folgen von Bits repräsentiert. Das Ziel ist die Redundanz der Nachricht zu verringern. Die Länge der Bitfolge ist dabei abhängig von der Wahrscheinlichkeit des Symbols. Hierbei ist wichtig, dass die verwendeten Codes präfixfrei sind, d.h. ein Symbol darf nicht durch eine Bitfolge kodiert werden, die gleichzeitig der Beginn einer anderen Bitfolge ist. Dadurch kann die Eindeutigkeit der Kodierung gewahrt werden. Sei $\mathcal{C}$ die Menge der Codewörter und ϵ das leere Codewort, dann gilt:

$$\forall c_1, c_2, c_3 \in \mathcal{C}, (c_1 = c_2 c_3 \Rightarrow (c_3 = \epsilon \wedge c_1 = c_2)) \tag{2.7}$$

Beispiel: Eine Kodierung könnte das Symbol `a` auf die Bitfolge c_a = «`0`», das Symbol `b` auf c_b = «`10`» und das Symbol `c` auf c_c = «`01`» abbilden. In diesem Fall wird die Nachricht «`acab`» zu der Bitfolge «`001010`». Da diese Kodierung jedoch nicht präfixfrei ist, könnte diese Bitfolge auch für die Nachricht «`aabb`» oder «`acca`» stehen, welche durch diese Kodierung auf die gleiche Bitfolge abgebildet werden.

2.2.1. Shannon-Fano-Kodierung

Ein Verfahren zur Erzeugung eines präfixfreien Codes ist das Shannon-Fano-Verfahren (benannt nach Claude Shannon und Robert Fano)[Fan49].

Die n Symbole werden hierfür nach ihrer Wahrscheinlichkeit geordnet:

$$p(z_i) > p(z_{i+1}), \qquad i = 0, 1, \cdots, n-2$$

Daraufhin wird die Menge der Symbole aufgeteilt, sodass beide Hälften der Menge möglichst gleich wahrscheinlich sind:

$$\left| \sum_{i=0}^{k-1} p(z_i) - \sum_{i=k}^{n-1} p(z_i) \right| \stackrel{!}{=} min$$

Die beiden Mengen werden in einen binären Baum eingehängt. Die Untermengen, welche die Blätter des Baumes bilden, werden wieder in zwei möglichst gleich wahrscheinliche Mengen unterteilt und der binäre Baum so vertieft. Die Mengen werden so oft unterteilt, bis die Blätter des Baumes nur noch aus den Symbolen selbst bestehen. Codes, die in einem Baum dargestellt werden können, sind immer präfixfrei, da nur die Blätter Symbole darstellen.

Beispiel: Abbildung 2.1 zeigt am Beispiel einer Symbolmenge mit den Wahrscheinlichkeiten $p_a = 0,4$ und $p_b = p_c = p_d = p_e = 0,15$, wie der Shannon-Fano-Algorithmus den Baum erstellt.

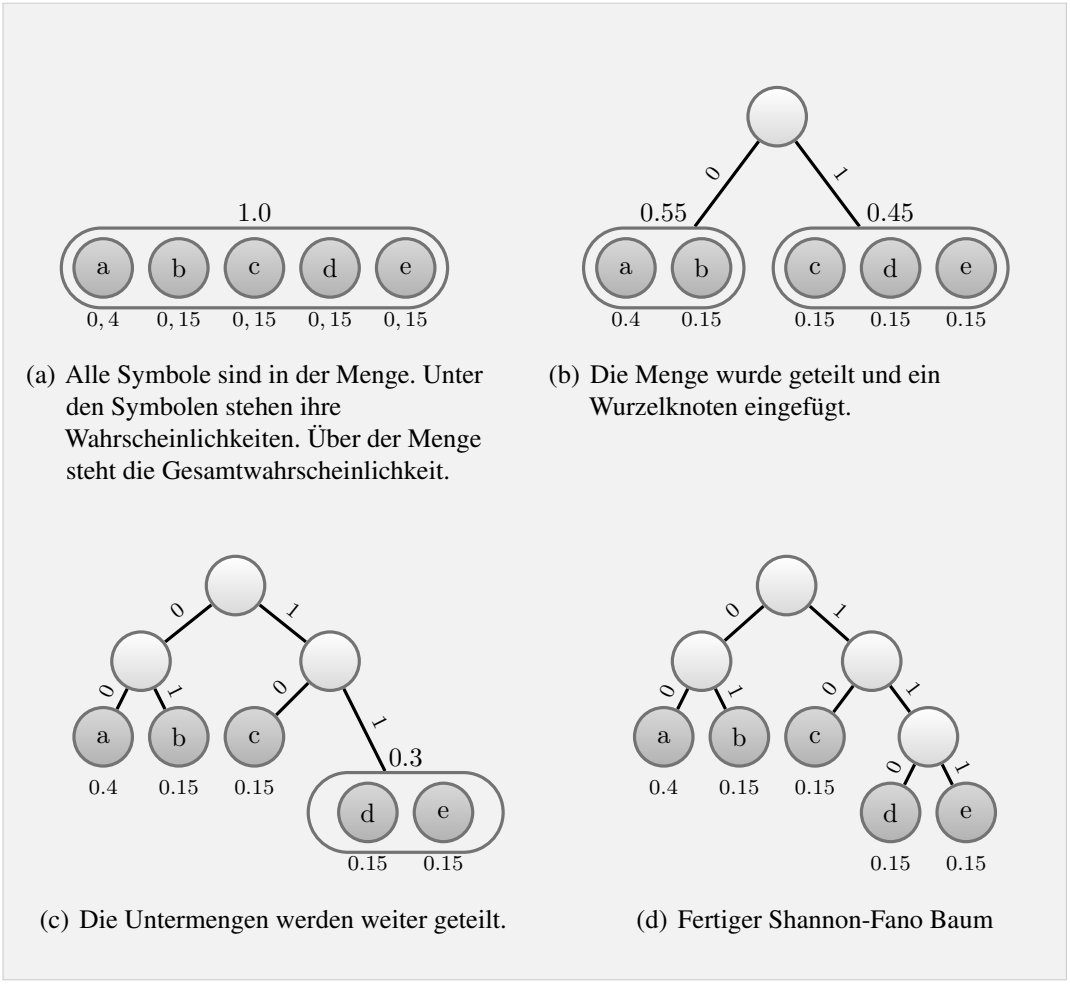
Der Pfad von der Wurzel zu den Blättern des Baumes repräsentiert den präfixfreien Shannon-Fano Code. Dieser Code hat eine mittlere Codewortlänge von

$$L(\mathcal{K}_{SF}) = \sum_{z \in \mathcal{Z}} l_z \cdot p_z = 2,3\,\text{Bit}$$

Bei einer Entropie von

$$H(\mathcal{X}) = \sum_{z \in \mathcal{Z}} I(z) \cdot p_z = 2,17\,\text{Bit}$$

ergibt sich eine Redundanz von $R(\mathcal{K}_{SF}) = 0,13\,\text{Bit}$.



(a) Alle Symbole sind in der Menge. Unter den Symbolen stehen ihre Wahrscheinlichkeiten. Über der Menge steht die Gesamtwahrscheinlichkeit.

(b) Die Menge wurde geteilt und ein Wurzelknoten eingefügt.

(c) Die Untermengen werden weiter geteilt.

(d) Fertiger Shannon-Fano Baum

Abbildung 2.1.: Schritte zum Aufbau des Shannon-Fano Baumes. Die Kanten auf dem Weg von der Wurzel zu den Symbolen beschreiben deren Kodierung.

2.2.2. Huffman-Kodierung

Codes, die durch den Shannon-Fano-Algorithmus berechnet werden, sind nicht immer optimal. Es kommen häufig Codes mit einer geringeren Redundanz vor. Ein Algorithmus für die Berechnung einer optimalen Entropie-Kodierung einzelner Symbole wurde von David Huffman im Jahr 1952 vorgestellt [Huf52].

Dieses Verfahren läuft den umgekehrten Weg des Shannon-Fano-Verfahrens. Die zwei Symbole mit der geringsten Wahrscheinlichkeit werden zu einem Knoten zusammengefasst. Dieser Knoten bekommt die summierte Wahrscheinlichkeit der beiden Symbole und wird fortan in der Berechnung des Baumes als Symbol behandelt. Dieser Schritt wird so oft wiederholt, bis nur noch ein Knoten mit der Wahrscheinlichkeit 1 übrig ist. Von diesem Knoten wird ein Baum aufgespannt, dessen Blätter die Symbole bilden.

Beispiel: Abbildung 2.2 zeigt den Aufbau des Baumes mit den gleichen Symbolwahrscheinlichkeiten wie bei der Shannon-Fano Kodierung.

Dieser Huffman-Code hat eine mittlere Codewortlänge von $L(\mathcal{K}_H) = 2,2$ Bit und ist damit um $0,1$ Bit kürzer als der erzeugte Shannon-Code für das gleiche Alphabet. Die Redundanz verringert sich auf $R_{Code} = 0,03$ Bit.

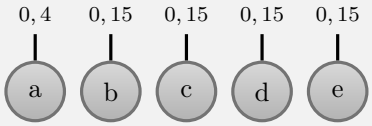
Eine Redundanz von 0 ist nur in Sonderfällen erreichbar, wenn die Wahrscheinlichkeit der Symbole durch 2^{-x}, $x \in \mathbb{N}$ dargestellt werden kann. Es gibt keine Kodierung für Einzelsymbole, welche die Redundanz weiter verringern kann. Allerdings kann durch Kodierung von jeweils zwei oder mehr Symbolen die Entropie und ggf. auch der Huffman-Code weiter verringert werden.

2.2.3. Arithmetische Kodierung

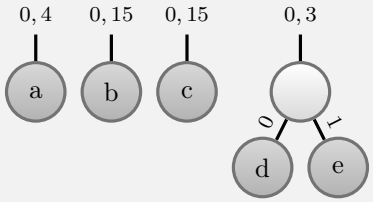
Bei der arithmetischen Kodierung wird nicht jedes Symbol einzeln, sondern die Nachricht als Ganzes kodiert. Dadurch kann die Kodierung ggf. zu besseren Ergebnissen führen als die Huffman-Kodierung.

Die Ausgabe des Kodierers ist eine reelle Zahl im Intervall $[0;1)$.

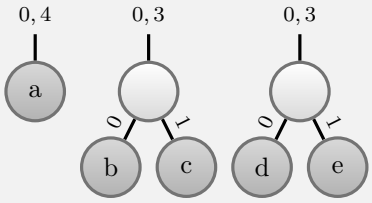
Es wird mit einem Intervall $[0;1)$ begonnen. Jedes Symbol steht für ein Subintervall dieses Intervalls. Die Länge des Subintervalls ist abhängig von der Wahrscheinlichkeit des Symbols. Das Subintervall, welches für das nächste Symbol der Nachricht steht, wird zum aktuellen Intervall. Das aktuelle Intervall wird wieder in Subintervalle



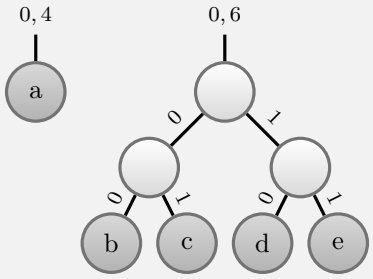
(a) Alle Symbole einzeln



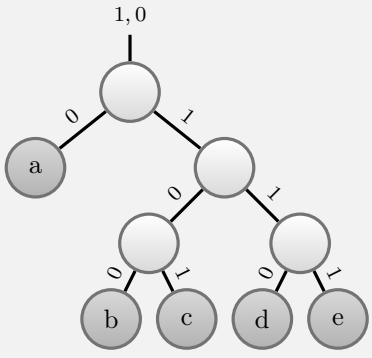
(b) Symbole mit der geringsten Wahrscheinlichkeit werden zusammengefasst und als ein Knoten behandelt.



(c) Symbole mit der geringsten Wahrscheinlichkeit werden zusammengefasst.



(d) Die beiden Knoten der zusammengefassten Symbole haben die geringste Wahrscheinlichkeit und werden zusammengefasst.



(e) Fertiger Huffmanbaum

Abbildung 2.2.: Erstellen eines Huffman Baumes. Der Weg vom Wurzelknoten zu den Symbolen beschreibt die Kodierung.

unterteilt und der letzte Schritt wird so lange wiederholt, bis keine Symbole mehr in der Nachricht folgen. Zum Schluss wird eine beliebige Zahl mit möglichst wenigen Nachkommastellen aus dem aktuellen Intervall gewählt. Weiterhin wird die Anzahl der kodierten Symbole zum Dekodieren benötigt.

Für die Dekodierung wird wie bei der Kodierung mit dem Anfangsintervall $[0;1)$ begonnen und dieses in Subintervalle unterteilt, welche für die Symbole stehen. Es muss dabei die gleiche Unterteilung vorliegen, wie bei der Kodierung. Das Subintervall, in dem die aus dem Kodierer gegebene Zahl liegt, wird zum aktuellen Intervall und erneut in Subintervalle unterteilt. Der letzte Schritt wird so lange wiederholt, wie Symbole zu dekodieren sind. Daher wird die Anzahl der kodierten Symbole benötigt.

Als Alternative zu der Übertragung der kodierten Symbole, kann ein Symbol im Intervall berücksichtigt werden, welches das Ende der Nachricht markiert.

Da die Nachricht bei der arithmetischen Kodierung als Ganzes kodiert wird, erreicht der Kodierer in einigen Fällen bessere Ergebnisse als die Huffman-Kodierung.

Ein Nachteil der arithmetischen Kodierung liegt in dem höheren Rechenaufwand. Es wird mit Fließkommazahlen gerechnet und die Intervalle müssen immer wieder neu berechnet werden.

Beispiel: Abbildung 2.3 zeigt die Schritte der Kodierung für die Nachricht «`abacd`» und das Alphabet $Z = \{a, b, c, d, e\}$ mit den gleichen Wahrscheinlichkeiten wie bei der Shannon-Fano- und der Huffman-Kodierung. Die Nachricht kann mit einer Zahl zwischen $0,17572$ und $0,17626$ kodiert werden. Binär kodiert liegt die $0,00101101$ in diesem Intervall. Es sind also 8 Bit für die Kodierung nötig. Wird die Nachricht mit der erstellten Huffman-Kodierung aus dem vorherigen Abschnitt kodiert, werden 11 Bit nötig.

2.3. Stringersatzverfahren

Stringersatzverfahren betrachten nicht die einzelnen Symbole, sondern ganze Folgen von Symbolen innerhalb einer Nachricht. Auf diese Weise werden bekannte Strukturen innerhalb einer Nachricht berücksichtigt.

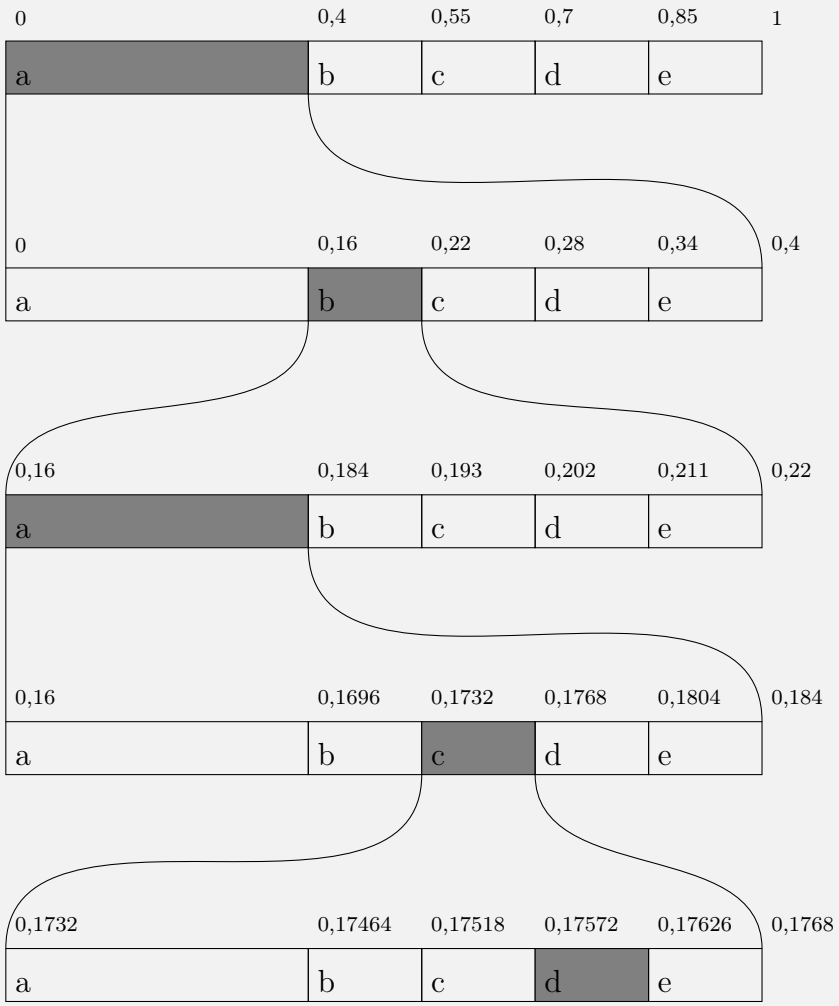


Abbildung 2.3.: Arithmetische Kodierung für die Nachricht «`abacd`». Das `d` liegt am Ende zwischen $0,17572$ und $0,17626$.

Stringersatzverfahren werden auch Wörterbuchverfahren genannt. In dieser Arbeit werden jedoch Wörterbuchverfahren als Untergruppe der Stringersatzverfahren behandelt und nicht mit diesen gleichgesetzt.

Der Wörterbuchkodierer durchsucht eine Nachricht nach mehrfach auftretenden Symbolfolgen. Diese Folgen werden nur noch einmal kodiert und anschließend referenziert. Zwei verschiedene Ansätze wurden von Jakob Ziv und Abraham Lempel in LZ77 [ZL77] und LZ78 beschrieben.

2.3.1. Lauflängen-Kodierung

Das einfachste Stringersatzverfahren ist die Lauflängen-Kodierung (RLE - Runlength Encoding). Es handelt sich hierbei nicht um ein Wörterbuchverfahren, allerdings werden Folgen von gleichen Symbolen durch die Anzahl der Wiederholungen und ihrem Symbol kodiert.

So wird aus der Nachricht »`aaaabbbbbaaaaaccccccccaaaa`« die kodierte Nachricht »`4a5b5a8c4a`«.

Dieses Verfahren ist nur bei Nachrichten mit langen Folgen gleicher Symbole effizient. Zum Beispiel in Icon-Dateien, bei denen in einer Zeile oft lange Folgen einer Farbe vorkommen.

Es gibt auch Varianten, die mehrere Symbole zusammenfassen und deren Wiederholungen kodieren, sodass aus der Nachricht »`abababababacacacaccbcbcb`« die kodierte Nachricht »`5ab4ac3cb`« wird.

2.3.2. LZ77

Der LZ77 Algorithmus ist ein sogenanntes Sliding-Window Verfahren, d.h. es wird ein Ausschnitt fester Länge aus der Nachricht betrachtet, der aus einem Such-Puffer und einem Vorschau-Puffer besteht. In dem Such-Puffer stehen die zuletzt kodierten Symbolfolgen. Der Ausschnitt wird während der Kodierung vom Beginn der Nachricht bis zu ihrem Ende schrittweise verschoben. Dabei wird in dem Ausschnitt der bereits kodierten Nachricht nach Symbolfolgen gesucht, die der kommenden Symbolfolge gleichen. Die Ausgabe besteht aus Tripeln mit einem Offset, einer Länge und dem darauf folgenden Symbol. Der Offset und die Länge geben eine Position im Suchpuffer an, in der die gleiche Symbolfolge steht, wie die folgende Symbolfolge mit der

gegebenen Länge. Anschließend wird der betrachtete Ausschnitt um die Anzahl der kodierten Stellen weiter verschoben.

Ähnlich wie die Komprimierung kann die Nachricht wieder dekomprimiert werden. Die bereits dekodierten Symbole werden in einen Puffer geschrieben. Folgt eine Referenz, kann in diesem Puffer die Symbolfolge nachgeschlagen werden.

Das Verfahren ist effizienter, je größer der Puffer ist, in dem nach bereits vorgekommenen Symbolfolgen gesucht wird. Allerdings führt ein größerer Puffer auch zu mehr Aufwand bei der Suche.

Beispiel: Abbildung 2.4 zeigt, wie die Nachricht »`abcdabcdabcb`« kodiert wird. Die kodierte Nachricht lautet »`(0,0,a) (0,0,b) (0,0,c) (0,0,d) (4,4,a) (4,2,b)`«.

Die folgende Tabelle zeigt die Dekodierung der Nachricht:

Tripel	Puffer					Ausgabe
	5	4	3	2	1	
(0,0,a)						`a`
(0,0,b)					a	`b`
(0,0,c)				a	b	`c`
(0,0,d)			a	b	c	`d`
(4,4,a)		a	b	c	d	`abcda`
(4,2,b)	a	b	c	d	a	`bcb`

LZSS

Da in schlechten Fällen, in denen keine Wiederholungen von Symbolfolgen vorkommen, die kodierte Nachricht größer sein kann als die unkodierte, wurde der Algorithmus 1982 von James A. Storer und Thomas G. Szymanski modifiziert [SS82].

Der LZSS Algorithmus gibt nicht nur Tripel aus, sondern entweder einzelne Symbole oder Tupel, welche ein vorheriges Vorkommen der Symbolfolge referenzieren.

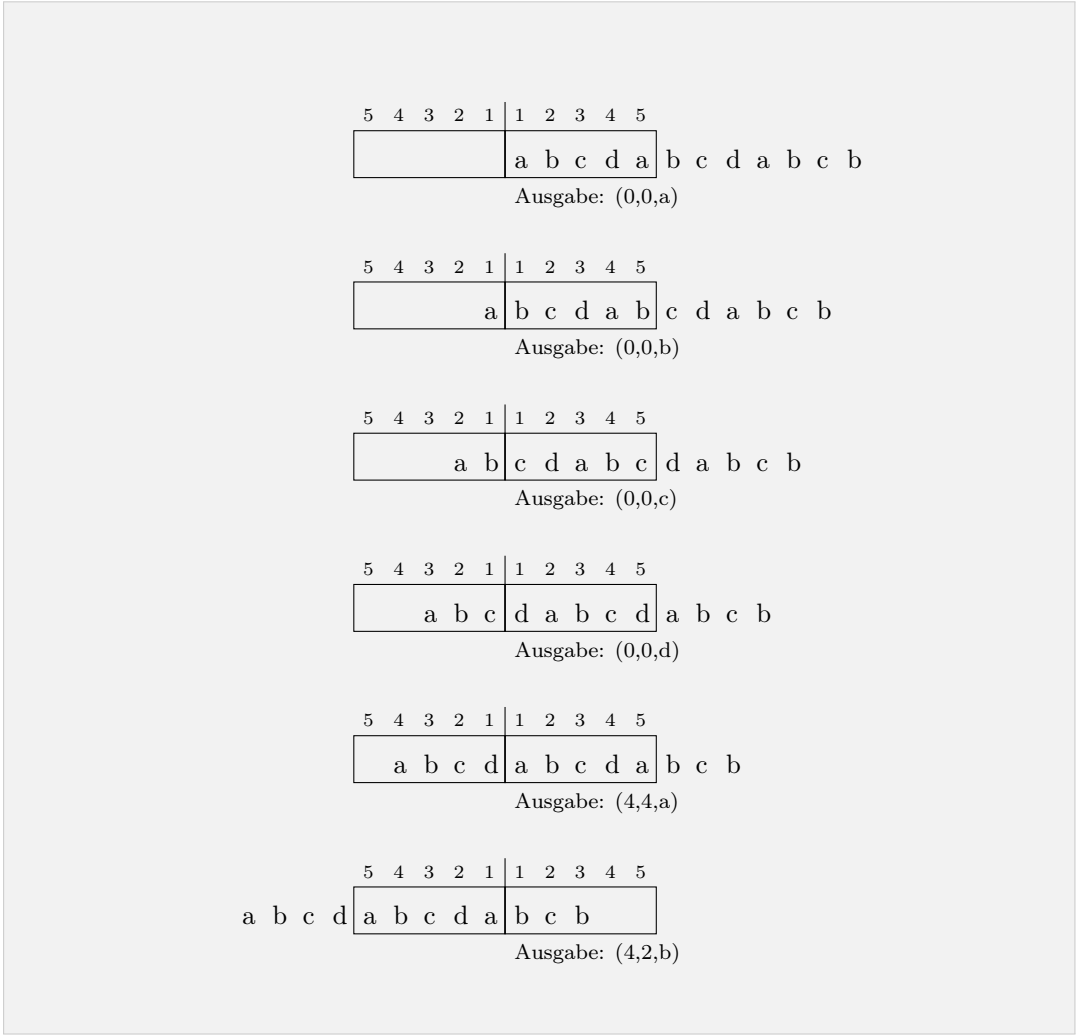


Abbildung 2.4.: LZ77 Kodierung der Nachricht »`abcdabcdabcb`«

Ein Marker-Symbol oder ein Marker-Bit signalisiert, ob es sich um ein Symbol oder um eine Referenz handelt.

Um den Speicherverbrauch zu reduzieren, wird das Vorschaufenster in einem Ringpuffer gehalten. Für eine Performance-Steigerung wird das Suchfenster in einem binären Baum gespeichert, wodurch sich die Suche besonders bei großen Fenstern beschleunigt.

Beispiel: Abbildung 2.5 zeigt, wie die gleiche Nachricht aus der LZ77-Kodierung mit LZSS kodiert wird. Die kodierte Nachricht lautet »`abcd(4,4)(4,3)b`«.

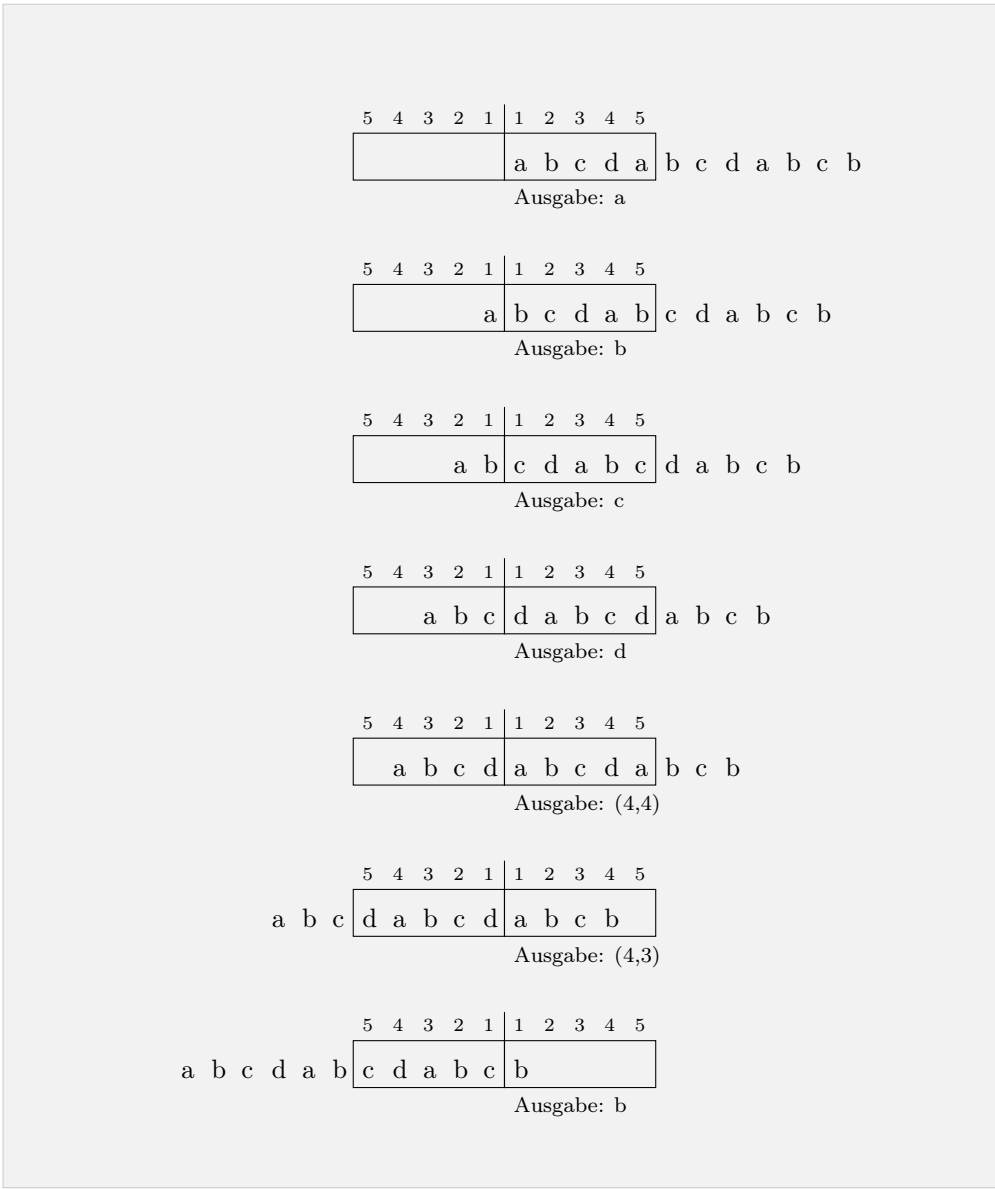


Abbildung 2.5.: LZSS Kodierung der Nachricht »`abcdabcdabcb`«

2.3.3. LZ78

Der LZ78 Algorithmus ist eine Weiterentwicklung des LZ77 Algorithmus [ZL78]. Da die Suche in dem Puffer aufwändig ist und es oft gleiche Treffer gibt, wird bei diesem Algorithmus auf eine Suche im Puffer verzichtet und stattdessen ein Wörterbuch aufgebaut. Die Ausgabe besteht nun aus Tupel, die aus einem Wörterbuch-Index und dem Folgesymbol bestehen.

Zu Beginn ist das Wörterbuch leer und die Ausgabe besteht aus einem Tupel, welches nur das Symbol enthält. Dieses Symbol wird dann in das Wörterbuch eingetragen. Wenn dieses Symbol erneut auftritt, bildet die Nummer des Eintrages im Wörterbuch und das Folgesymbol die Ausgabe. Diese Folge von Symbolen wird wieder in das Wörterbuch eingetragen, sodass ein erneutes Auftreten dieser Folge nur noch durch den Wörterbucheintrag kodiert wird.

Nachteilig bei diesem Verfahren ist, dass eventuell Symbolfolgen in das Wörterbuch aufgenommen werden, die nicht erneut vorkommen. Ein größeres Wörterbuch erreicht bessere Ergebnisse.

Beispiel: Es wird die Nachricht »`abcdabcdabcb`« kodiert:

Zeichenkette	gefundener Eintrag	Ausgabe	neuer Eintrag
`abcdabcdabcb`	0 (-)	(0,a)	1:a
`bcdabcdabcb`	0 (-)	(0,b)	2:b
`cdabcdabcb`	0 (-)	(0,c)	3:c
`dabcdabcb`	0 (-)	(0,d)	4:d
`abcdabcb`	1 (a)	(1,b)	5:ab
`cdabcb`	3 (c)	(3,d)	6:cd
`abcb`	5 (ab)	(5,c)	7:abc
`b`	2 (b)	(2,0)	

Die kodierte Nachricht lautet »`(0,a) (0,b) (0,c) (0,d) (1,b) (3,d) (5,c) (8,0)`«.

LZW

Terry Welch verbesserte die LZ78 Kodierung 1983 zur LZW Kodierung [SS82].

Hierbei wird das Wörterbuch zu Beginn mit dem Alphabet initialisiert. Daher ist es möglich, dass nicht mehr Tupel, sondern nur noch die Wörterbuchreferenzen ausgegeben werden. Die letzte Wörterbuchreferenz mit dem ersten Symbol der folgenden Wörterbuchreferenz wird in das Wörterbuch eingetragen.

Es ist möglich, dass Wörterbuchreferenzen ausgegeben werden, die erst mit der Ausgabe dieser Referenz bekannt sind. Für die Dekodierung ist dies jedoch unproblematisch, da der Anfang der Referenz gleich der vorherigen Referenz ist und das letzte Symbol gleich dem Anfangssymbol ist.

Für die Dekodierung müssen die anfänglich initialisierten Symbole bekannt sein.

Beispiel: Es wird die Nachricht »`abcdabcdabcb`« kodiert:

Zeichenkette	gefundener Eintrag	Ausgabe	neuer Eintrag
Initialisierung			0 :a 1 :b 2 :c 3 :d
`abcdabcdabcb`	0:a	0	4 :ab
`bcdabcdabcb`	1:b	1	5 :bc
`cdabcdabcb`	2:c	2	6 :cd
`dabcdabcb`	3:d	3	7 :da
`abcdabcb`	4:ab	4	8 :abc
`cdabcb`	6:cd	6	9 :cda
`abcb`	8:abc	8	10:abcb
`b`	1:b	1	

Die kodierte Nachricht lautet »`01234681`«.

2.4. Andere Verfahren

Es gibt diverse andere Verfahren der Kompression. In diesem Abschnitt werden zwei Verfahren vorgestellt, welche versuchen die Nachricht vor der Kodierung in eine andere Form zu transformieren, um bessere Komprimierungen zu erreichen.

2.4.1. Move-To-Front-Kodierung

Die Grundidee der Move-To-Front-Kodierung (MTF) ist ein Alphabet mit sich ändernder Reihenfolge der Symbole [BSTW86]. Ausgegeben wird die Position des Symbols im Alphabet zur Zeit des Auftretens. Dabei wird das Symbol, das zuletzt aufgetreten ist, an die erste Stelle geschoben. Häufige Symbole befinden sich so am Anfang des Alphabets und der Ausgabestrom besteht zu einem großen Teil aus kleinen Zahlen. Dabei ist das Verfahren lokal adaptiv, d.h. es passt sich an den Bereich der Nachricht an, in dem es sich befindet. Während am Anfang einer Nachricht ein bestimmtes Symbol häufig ist, kann am Ende der Nachricht ein anderes Symbol häufig sein.

Die Ausgabe ermöglicht bei geeigneten Daten eine sehr effiziente Entropie-Kodierung.

Es wird zunächst ein Alphabet der möglichen Symbole $Z = \{z_1, z_2, \cdots, z_n\}$ initialisiert. Tritt nun ein Symbol z_i auf, wird die Position in dem Alphabet ausgegeben und das Alphabet ändert sich zu $Z = \{z_i, z_1, \cdots, z_{i-1}, z_{i+1}, \cdots, z_n\}$. Für das nächste auftretende Symbol z_j ändert sich das Alphabet zu $Z = \{z_j, z_i, z_1, \cdots, z_{j-1}, z_{j+1}, \cdots, z_n\}$ und so weiter.

Anschließend kann diese Nachricht mit einer Huffman-Kodierung komprimiert werden. Da kleine Zahlen sehr wahrscheinlich sind, kann auch eine fest vorgegebene Huffman-Kodierung verwendet werden, sodass nicht für jede Nachricht ein eigener Code berechnet werden muss.

Für die Dekodierung wird das gleiche Alphabet initialisiert. Für jede Position, die auftritt, wird nun das Symbol der Position ausgegeben und anschließend das Symbol der Position an die erste Stelle im Alphabet gerückt.

Beispiel: Im Folgenden wird die MTF-Kodierung der Nachricht »`ANANASSAFT`« gezeigt:

Symbol	Position	Alphabet
A	0	AFNST
N	2	AFNST
A	1	NAFST
N	1	ANFST
A	1	NAFST
S	3	ANFST
S	0	SANFT
A	1	SANFT
F	3	ASNFT
T	4	FASNT

Die Kodierte Nachricht lautet »`0211130134`«.

2.4.2. Burrows-Wheeler Transformation

Die Burrows-Wheeler Transformation (BWT) wird genutzt, um die Symbole einer Nachricht umzusortieren [BW94]. Dabei ist sichergestellt, dass die Symbole wieder eindeutig in ihre richtige Reihenfolge gebracht werden können. Durch die Umsortierung stehen gleiche Symbole häufiger nebeneinander, sodass eine anschließende Move-To-Front-Kodierung bessere Ergebnisse erzielen kann.

Zunächst wird die Nachricht in die erste Zeile einer Matrix geschrieben. Jede weitere Zeile der Matrix wird durch die Rotationen der Nachricht gefüllt - die Matrix ist also quadratisch. Anschließend werden die Zeilen der Matrix geordnet. Hierfür muss eine vollständige Ordnung der Symbole vorliegen. Welche Ordnung benutzt wird, ist irrelevant. Die letzte Spalte der Matrix bildet nun die Ausgabe der Transformation. Zusätzlich wird auch noch der Index der Matrixzeile benötigt, in welcher die Nachricht in ihrer ursprünglichen Form steht.

Beispiel: Abbildung 2.6 zeigt die BWT anhand der Nachricht »`ANANASSAFT`«. Die transformierte Nachricht lautet »`STNNAAASAF`«. Für die Rücktransformation muss der Index der Matrixzeile, in welcher die ursprüngliche Nachricht steht, bekannt

sein. In diesem Beispiel muss also die ’2’ ebenfalls mit der transformierten Nachricht zurückgegeben werden.

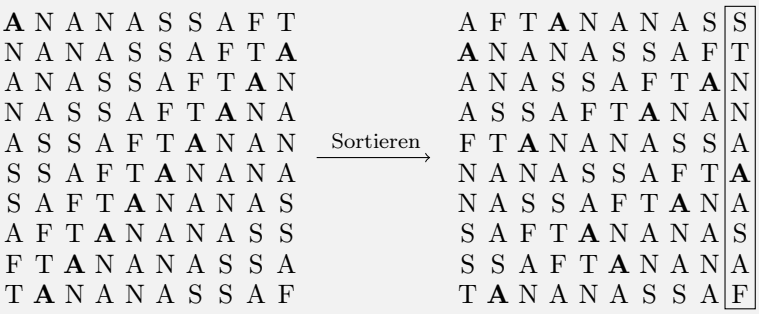
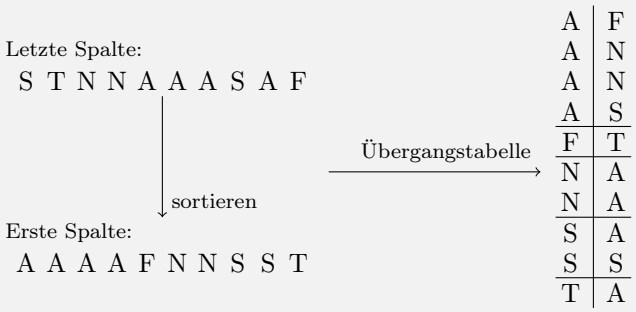


Abbildung 2.6.: Burrows-Wheeler-Transformation der Nachricht »ANANASSAFT«. Der erste Buchstabe der Nachricht wurde fett markiert, um den Beginn in der Matrix leichter zu finden.

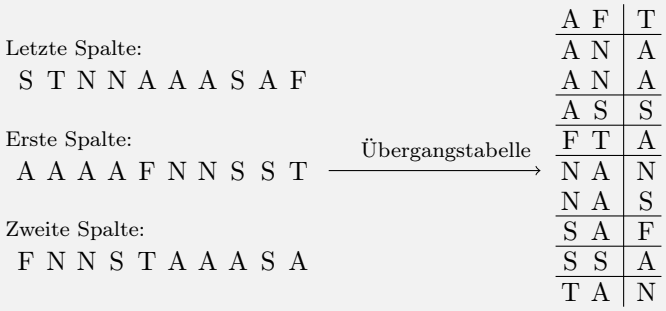
Für die Rücktransformation muss die Matrix wiederhergestellt werden. Die erste und die letzte Spalte der Matrix können sofort eingefügt werden, denn die letzte Spalte ist bekannt und die erste Spalte beinhaltet die gleichen Symbole wie die letzte Spalte mit der genutzten Ordnung sortiert.

Durch die Übergänge der letzten zur ersten Spalte sind so auch alle möglichen Symbolübergänge bekannt und die folgenden Spalten können nacheinander aufgebaut werden. Wenn die Matrix komplett ist, kann in der Zeile, dessen Index bekannt ist, die originale Nachricht ausgelesen werden.

Beispiel: Abbildung 2.7 zeigt die Rücktransformation der Nachricht aus dem Beispiel.



(a) Aus der letzten Spalte kann durch Sortieren die erste Spalte generiert werden. Durch die Übergänge von der letzten zur ersten Spalte erhält man die Übergangstabelle.



(b) Durch Vorstellen der letzten Spalte vor die erste, erhält man die Übergangstabelle für die dritte Spalte.

-	1	2	3	4	5	6	7	8	9	10
1	A	F	T							S
2	A	N	A							T
3	A	N	A							N
4	A	S	S							N
5	F	T	A							A
6	N	A	N							A
7	N	A	S							A
8	S	A	F							S
9	S	S	A							A
10	T	A	N							F

(c) Zweite und dritte Spalte aus den Folgesymbolen konstruiert

Abbildung 2.7.: Rücktransformation der Transformierten Nachricht »STNNAAASAF«

Kapitel 3

Bekannte Kompressionsverfahren

Aktuelle häufig verwendete Verfahren kombinieren verschiedene der vorgestellten Methoden der Datenkompression, um gute Kompressionsergebnisse zu erzielen. Da diese Verfahren sehr groß sind, viel Zeit für die Komprimierung benötigen und einen großen Speicherbedarf haben, eignen sie sich nicht für die Implementierung auf einem Stromzähler. Sie zeigen aber, welche Kompressionsraten für die Daten möglich sind. Es ist in einigen Fällen möglich, die Verfahren für die Nutzung auf den Stromzählern umzuschreiben.

Die letzten beiden Verfahren in diesem Kapitel beschreiben Verfahren, die auf einem Stromzähler realisierbar sind, bzw. für drahtlose Sensorknoten entwickelt wurden.

3.1. DEFLATE (GZIP)

Der DEFLATE Algorithmus wurde von Philip W. Katz entwickelt und wird seit 1993 in der PKzip Komprimierung verwendet. Da der Algorithmus unter Public Domain steht, wird er auch in der ZLIB Bibliothek genutzt, welche von GZIP, dem HTTP Protokoll und PNG (Portable Network Graphics) verwendet wird. Auch Adobes PDF (Portable Document File) Dateien nutzen den DEFLATE Algorithmus.

Das Verfahren kombiniert das LZSS-Verfahren und die Huffman-Kodierung (s. Kap 2.3.2 und 2.2.2). Dabei werden die drei Kompressionsstufen »schnell«, »normal« und »hohe Kompression« (»fast«, »normal« und »high-compression«) angeboten. Diese Stufen legen fest, wie genau der LZSS Algorithmus nach bereits

vorgekommenen Symbolfolgen suchen soll. Im »fast«-Modus wird das reine LZSS Verfahren angewendet (s. Kap 2.3.2). Der »normal«- und »high-compression«-Modus prüft ebenfalls, ob das aktuelle Symbol direkt ausgegeben und mit dem nächsten Symbol folgend eine längere Referenz gefunden werden kann. Die Stufen »normal« und »high-compression« unterscheiden sich lediglich durch die Intensität der Suche. Zusätzlich kann die Fenstergröße eingestellt werden, die auf 32 KByte voreingestellt ist.

Das Resultat des LZSS-Verfahrens wird anschließend mit Huffman kodiert, wobei die Einzelsymbole zusammen mit der Länge einer Referenz in einem Baum kodiert werden. Das heißt zu den 256 möglichen Symbolen eines Byte kommen weitere 31 Symbole, die das Ende des Blocks oder den Beginn einer Referenz markieren. Tabelle 3.1 zeigt die zusätzlichen Symbole. Es wird also kein Präfix-Bit oder Marker-Symbol benötigt, da die Länge und Symbole zusammen kodiert werden. Das Auftreten einer Länge signalisiert gleichzeitig, dass ein Offset folgt. Die Offsets werden wiederum durch einen eigenen Huffman-Baum kodiert.

Weiterhin kodiert das Verfahren einzelne Blöcke verschiedener Größe einer Datei und kann für jeden Block einzeln entscheiden, ob dieser überhaupt komprimiert wird oder ob eine generische oder eine angepasste Huffman-Kodierung verwendet wird.

Das DEFLATE-Verfahren bietet auf handelsüblichen Desktop-PCs gute Kompressionsraten bei einer relativ hohen Geschwindigkeit sowohl beim Komprimieren, als auch beim Dekomprimieren. Die ZLIB-Bibliothek ist ca. 100 KByte groß.

3.2. BZIP2

Der BZIP2-Komprimierer wurde von Julian Seward 1996 veröffentlicht und steht unter der BSD Lizenz. Er verwendet Lauflängenkodierung, Burrows-Wheeler-Transformation, Move-To-Front und Huffman-Kodierung.

Zunächst werden die Daten mit einer Vier-Symbol-Lauflängenkodierung kodiert (s. Kap 2.3.1). Das bedeutet, nach vier Symbolen folgt eine Anzahl der Wiederholungen dieser Folge. Allerdings bedeutet eine `1` nicht, dass alle vier Symbole einmal wiederholt werden, sondern dass das erste Symbol der Folge wiederholt wird.

Anschließend wird das Resultat in Blöcke unterteilt, deren Größe von 100 bis 900 KByte eingestellt werden kann. Diese Blöcke werden mit der Burrows-Wheeler-Trans-

Symbole	Extra Bits	Beschreibung
0 - 255	0	Byte Symbol
256	0	Ende des Blocks
257 - 264	0	Referenz der Länge 3 bis 10
265 - 268	1	Referenz der Länge 11 bis 18
269 - 272	2	Referenz der Länge 19 bis 34
273 - 276	3	Referenz der Länge 35 bis 66
277 - 280	4	Referenz der Länge 67 bis 130
281 - 284	5	Referenz der Länge 131 bis 257
285	0	Referenz der Länge 258
286 - 287	0	Reserviert, nicht benutzt

Tabelle 3.1.: 288 Symbole bilden den Huffman-Baum beim DEFLATE Algorithmus. Die ersten 256 Symbole bilden die möglichen Symbole der Eingabedaten, die weiteren Symbole bilden einen Präfix-Code für die Kodierung der Länge.

formation umsortiert und das Move-To-Front-Verfahren überführt die Symbolfolge in eine Darstellung mit höherer Entropie (s. Kap 2.4.2 und 2.4.1).

Darauf wird erneut eine Lauflängenkodierung angewandt. Für diese Lauflängenkodierung werden zwei zusätzliche Symbole `RUNA` und `RUNB` eingeführt, mit denen die Anzahl der Wiederholungen kodiert wird. Die Kodierung durch `RUNA` und `RUNB` findet folgendermaßen statt: Die Position hinter dem Symbol, auf das die Längenkodierung folgt, bestimmt die Wertigkeit, also 1, 2, 4, 8 usw. Dabei steht `RUNA` direkt für den Wert dieser Position und `RUNB` für den doppelten Wert.

Zum Schluss wird eine Huffman-Kodierung angewendet (s. Kap 2.2.2), in der 259 Symbole verwendet werden, also 256 Symbole für die Ausgabe des MTF Verfahrens, 2 Symbole für `RUNA` und `RUNB` und ein weiteres Symbol, um das Ende des Blockes zu markieren.

Ein Vorteil der BZIP2 Kompression ist, dass durch das blockweise Kodieren bei Fehlern nur der Block mit dem Fehler verloren ist. Alle anderen Blöcke können weiter dekomprimiert werden. Bei den meisten anderen Verfahren ist durch einen Fehler die gesamte Nachricht verloren.

Das Verfahren verspricht bessere Kompressionsergebnisse als der GZIP, bzw. der DEFLATE Algorithmus. Die Geschwindigkeit beim Komprimieren ist geringer als bei GZIP, beim Dekomprimieren ist die Geschwindigkeit sogar viel geringer als bei

GZIP. Die LIBBZ2-Bibliothek ist ca. 70 KByte groß. Da hier blockweise vorgegangen wird, ist der Speicherbedarf besonders groß.

Beispiel: Es wird die Nachricht »`abcdabcabcdacd`« betrachtet. Durch die erste Lauflängenkodierung wird aus der Nachricht »`abcd3abcd1cd`«. Die BWT sortiert die Folge um, zu »`dd3daabb1ccc3`« (die letzte Drei ist der Index der Zeile, die nach der Rücktransformation die ursprüngliche Reihenfolge enthält). Nach dem MTF-Verfahren, mit dem Alphabet »`0123abcd`« erhalten wir die Folge »`7, 0, 4, 1, 5, 0, 6, 0, 5, 7, 0, 0, 5`«. Durch die zweite Lauflängenkodierung verändert sich die Folge zu »`7, 0, 4, 1, 5, 0, 6, 0, 5, 7, 0, RUNA, 5`«. Diese Folge wird mit einer Huffman-Kodierung komprimiert und bildet die Ausgabe des BZIP2-Verfahrens.

3.3. Lempel-Ziv-Markov Chain-Algorithmus (LZMA)

Der Lempel-Ziv-Markov-Ketten-Algorithmus ist eine Weiterentwicklung des DEFLATE-Algorithmus. Das Verfahren wurde seit 1996 von Igor Pavvlov entwickelt und wird seit dem Jahr 2001 von dem Dateiarchivierungstool 7-Zip verwendet. Es steht unter Public Domain.

Der Algorithmus komprimiert zunächst die Nachricht mit einem LZSS Kodierer. Dabei werden die letzten vier Offsets in einer kurzen History gespeichert. Tritt einer dieser vier Offsets auf, wird statt des Offsets der Index in der History ausgegeben. Die unkodierten Symbole werden im Gegensatz zum DEFLATE Algorithmus statt mit einer Huffman-Kodierung durch eine Variante der arithmetischen Kodierung (s. Kap 2.2.3), der Bereichskodierung, komprimiert. Dadurch ist eine bessere Entropie-Kodierung möglich. Die Bereichskodierung arbeitet im Gegensatz zur arithmetischen Kodierung nur mit Integern und benutzt anstatt Division Shift-Operationen, die eine schnellere Programmausführung ermöglichen. Außerdem ist die verwendete Variante der Kodierung adaptiv und passt sich daher an die gegebenen Daten an. Es müssen dem Dekomprimierer daher keinen zusätzlichen Daten mitgeteilt werden.

Tabelle 3.2 zeigt wie die Ausgabe aussieht. Anders als beim DEFLATE Verfahren, werden die Referenzen nicht mit den Symbolen durch den Bereichskodierer kodiert.

Code	Beschreibung
0 + Bytecode	Ausgabe der Bereichskodierung
10 + Offset + Länge	LZ77 Referenz, Distanz ist 7 Bit Wert (1 bis 128)
1100	LZ77 Referenz der Länge 1 (Distanz gleich wie beim letzten)
1101 + Länge	LZ77 Referenz (Distanz gleich wie beim letzten)
1110 + Länge	LZ77 Referenz (Distanz gleich wie beim vorletzten)
11110 + Länge	LZ77 Referenz (Distanz gleich wie beim drittletzten)
11111 + Länge	LZ77 Referenz (Distanz gleich wie beim viertletzten)

Tabelle 3.2.: Präfix Codes der LZMA Kodierung

Präfix	Extra Bits	Längen
0	3	2 - 9
10	3	10 - 17
11	8	18 - 273

Tabelle 3.3.: Referenzlängen Kodierung beim LZMA Verfahren

Ein Präfix zeigt, ob eine Referenz oder die Ausgabe des Bereichskodierers folgt. In Tabelle 3.3 kann die Kodierung der Referenzlänge gesehen werden.

LZMA verspricht ca. 30% bessere Ergebnisse als GZIP und ca. 15% bessere Ergebnisse als BZIP2. Die Kompressionsgeschwindigkeit ist relativ gering, wobei die Dekomprimierung schneller als die BZIP2-Dekomprimierung ist. Die LIBLZMA-Bibliothek ist ca. 135 KByte groß.

3.4. LZSS-Variante

Das aktuell auf den Stromzählern verwendete Verfahren ist eine Variante des LZSS Verfahrens (s. Kap 2.3.2), deren Such- und Vorschaufenster jeweils auf 128 Byte beschränkt wurden. Außerdem wird wegen der begrenzten Ressourcen und der kleinen Fenster auf eine Indizierung durch einen binären Baum verzichtet. Stattdessen wird eine erschöpfende Suche verwendet. Da die Daten komplett im Speicher liegen wird auf einen Ringpuffer für das Vorschaufenster verzichtet.

Für die Referenz wird zu Beginn ein Symbol gesucht, das nicht oder nur selten in der Nachricht vorkommt. Dieses Referenzsymbol wird am Anfang der Nachricht einmal

ausgegeben, um es dem Dekomprimierer mitzuteilen. Kommt das Referenzsymbol selbst in der Nachricht vor, folgt anstatt der Länge eine Null (die Länge Null ist nicht zulässig). Offset und Länge werden jeweils mit mindestens einem Byte kodiert, wodurch sich eine Referenzlänge von drei Byte ergibt, d.h. Referenzen sind erst ab einer Länge von mehr als 3 Symbolen kürzer als die unkomprimierten Symbole. Im Programm befindet sich ein Fehler, der dazu führt, dass Symbolfolgen von sechs und sieben Byte nicht referenziert werden.

Das erste Bit des Längen- und Offset-Bytes bestimmt, ob ein weiteres Byte folgt. Somit sind theoretisch unendlich lange Referenzen möglich. Der Suchalgorithmus startet beim Offset 3 Symbole hinter der aktuellen Position und geht Byte-weise zum Ende des Sliding-Window. Die Implementierung verhindert, dass sich die Referenz und die Folgesymbole überschneiden. Diese Beschränkung ist überflüssig und kann das Kompressionsergebnis verschlechtern, da so eine mögliche Lauflängenkodierung verhindert wird.

Die Implementierung ist ca. 3 KByte groß.

3.5. Adaptive-Trimmed-Huffman (ATH)

Die Adaptive-Trimmed-Huffman-Kodierung [RCH+10] (ATH) wurde für drahtlose Sensorknoten entwickelt. Sie arbeitet mit einem vorgegebenen leeren Baum. Ein Präfix-Bit bestimmt, ob das folgende Symbol mit dem vorgegebenen Baum kodiert oder unkodiert ausgegeben wird. Dem Dekomprimierer muss der verwendete Baum bekannt sein. Die Huffman-Kodierung ist getrimmt, da die Codes eine bestimmte maximal Länge nicht überschreiten.

Tritt ein Symbol auf, wird überprüft, ob dieses in der Liste der bereits vorgekommenen Symbole steht. Ist dies nicht der Fall, wird das Symbol unkodiert ausgegeben und in die Liste der vorgekommenen Symbole an das Ende eingefügt. Wenn das Symbol bereits in der Liste steht, wird je nach Position in der Liste eine Kodierung durch den gegebenen Baum gewählt. Das Symbol, das an erster Stelle in der Liste steht, bekommt die kürzeste Kodierung. Existiert für die Position keine Kodierung, wird das Symbol unkodiert ausgegeben. Für das Symbol wird ein Zähler erhöht, der die Position in der Liste bestimmt. Das Symbol mit der größten Häufigkeit steht in der Liste vorne.

Der feststehende Baum und die Realisierung über eine geordnete Liste macht die Implementierung der Kodierung sehr einfach und schnell. Das Verfahren ist für drahtlose Sensornetze mit beschränkten Energie- und Speicherressourcen besonders gut geeignet.

Die Wahl des Baumes ist entscheidet für die Ergebnisse des Verfahrens. In der ursprünglichen Arbeit [RCH+10] wird ein Links-, bzw. ein Rechtslastiger Baum verwendet. Die Wahl des Baumes ist jedoch abhängig von den erwarteten Daten. Trotz einer bestmöglichen Anpassung an die Daten, werden bei dieser Komprimierung jedoch nicht so gute Ergebnisse erwartet wie bei den vorherigen Verfahren, da höchstens die Entropie der Nachricht erreicht werden kann. Dieses Verfahren wurde mit dem Anspruch entwickelt Energie zu sparen und hat nicht den Anspruch eine hohe Kompressionsleistung zu erreichen.

Das Verfahren ist sehr schnell, erreicht aber nur eine geringe Kompressionsleistung. Eine Implementierung ist weniger als 2 KByte groß.

3.6. Vergleich

Tabelle 3.4 zeigt die in diesem Kapitel vorgestellten Verfahren mit ihren Eigenschaften. Wenn die Größen der Bibliotheken ZLIB, LIBBZ2 und LIBLZMA mit den Größen der Implementierungen des momentan verwendeten Verfahrens und der ATH-Komprimierung verglichen werden, muss bedacht werden, dass die Bibliotheken auch Routinen zum Dekomprimieren enthalten und durch Parameter die Kompressionseinstellungen vorgenommen werden können. Bei der LZSS-Variante und der ATH-Komprimierung sind die Routinen bereits mit den Parametern kompiliert. Außerdem sind diese Algorithmen mit einer Optimierung auf geringe Größe kompiliert, während die Bibliotheken in der Regel auf Geschwindigkeit optimiert kompiliert werden. Auch wenn die Bibliotheken mit einer Optimierung auf geringe Größe kompiliert werden, stellen die zusätzlichen Einstellungsmöglichkeiten der Verfahren einen größeren Aufwand dar, der die Größe der Bibliotheken beeinflusst. Ein direkter Vergleich ist daher nicht möglich.

	DEFLATE	BZIP2	LZMA	LZSS	ATH
Programmgröße [KB]	⊖ / 100	⊖ / 70	⊖⊖ / 135	⊕ / 3	⊕⊕ / 2
RAM Verbrauch	⊖	⊖⊖	⊖	○	⊕
Kompressl.	⊕	⊕	⊕⊕	○	⊖
Geschwindigkeit	○	⊖	⊖⊖	⊕	⊕⊕
Verwendete Verf.	LZSS, Huffman	RLE, BWT, MTF, Huffman	LZSS, Arithmetische Kodierung	LZSS	Huffman

Tabelle 3.4.: Vergleich der bekannten Verfahren

Kapitel 4

Analyse realer Stromzählerdaten

In diesem Kapitel werden die Nachrichten, die vom Stromzähler geliefert werden, analysiert. Durch ein Verständnis des Datenformats kann eine Optimierung der Verfahren vorgenommen werden. Für diese Arbeit standen 3500 Datensätze aus Stromzähler-Auslesungen zur Verfügung.

4.1. Datenformat

Die Stromzählerdaten werden nach der Format-Spezifikation EN-62056-21 ausgegeben. Diese Daten sind ASCII-kodiert (American Standard Code for Information Interchange, 128 Symbole) und bestehen aus mehreren Zeilen der Form »A.B.C*D(Wert)«. Die Buchstaben A, B, C und D beschreiben, welche Art von Wert in der Klammer folgt. Dabei sind die Angaben C und D optional. In seltenen Fällen wird statt des * ein & verwendet.

Pro Auslesung werden mehrere Zeilen solcher Werte ausgegeben. Die Zeilen sind durch die Bytes <CR><LF> getrennt (ASCII-Steuersymbole: <CR> - Rücksetzten des Cursors, <LF> - Zeilenvorschub). Durch bestimmte Angaben von A und B können mehrere Zeilen von Werten folgen, ohne dass die Art der Werte erneut angegeben werden muss. Das wird z.B. für Lastprofile genutzt, die durch »P.01(Z)(S)(RPL)(RA)(R)(E)« eingeleitet werden. Die Buchstaben in den Klammern stehen für: Zeit (Z), Status (S), Register Periodenlänge (RPL), Register Anzahl (RA), Register (R) und Einheit (E).

Da der Standard des Ausgabeformats erweitert werden kann und einige Codes herstellerspezifisch sind, ist es nicht praktikabel die Struktur so zu nutzen, dass die Daten extrahiert und binär kodiert werden können. Weiterhin gibt es viele unregelmäßige Datensätze, die ein Verständnis der Daten und damit auch eine Binärkodierung derselben erschwert. So können z.B. Prüfsummenausgaben (eingeleitet durch »`C.77`«) verschiedene Längen haben.

Im folgenden sind einige Zeilen verschiedener Nachrichten zusammengefasst, um einen Eindruck des Formats der Stromzählerdaten zu gewinnen.

```
F.F(00000000)
0.0.0(00003407)
0.0.1(00920743)
0.1.0(16)
0.1.2*16(0091201000000)
72.25(229.8*V)
C.7.1(0472)
P.01(0080313051500)(00000000)(15)(1)(1.5)(kW)
(00.26)
(00.20)
(00.25)
(00.24)
(01.00)
C.77.2(0A78422F69654D2D4952481FBF16)
C.77.1*1(685F5F6808087217027393AE4C48077D600000143E700A0000146D070B2A19243E0[...]
```

Listing 4.1: Beipieldaten einer Stromzähler-Auslesung

4.2. Datensatz-Größen

Laut Herstellerangaben können die Stromzählerdaten bis zu 10 KByte groß sein. Der größte Datensatz der gegebenen Auslesungen ist 3100 Byte groß. Die kleinsten Datensätze sind 76 Byte groß. Etwa die Hälfte der Auslesungen ist kleiner als 1000 Byte. Nur 12 Dateien sind größer als 1614 Byte.

Abbildung 4.1 zeigt die Verteilung der Datensatzgrößen, wobei mehrere Größen gebündelt wurden. Am häufigsten sind Auslesungen von 1367 Byte (744 Dateien), in denen besonders viele Lastprofile (eingeleitet durch `P.01`) übermittelt werden.

Interessant ist, dass sich die Datensatzgrößen oft nur durch wenige Bytes unterscheiden und sie sich deshalb gruppieren lassen. Um die Datensätzen von 950 Byte finden

sich aber keine anderen Datensätze, die sich nur gering in der Größe unterscheiden. In den betreffenden Auslesungen kann keine weitere Besonderheit festgestellt werden.

Alle Datensätze zusammen sind insgesamt 3 514 KByte groß.

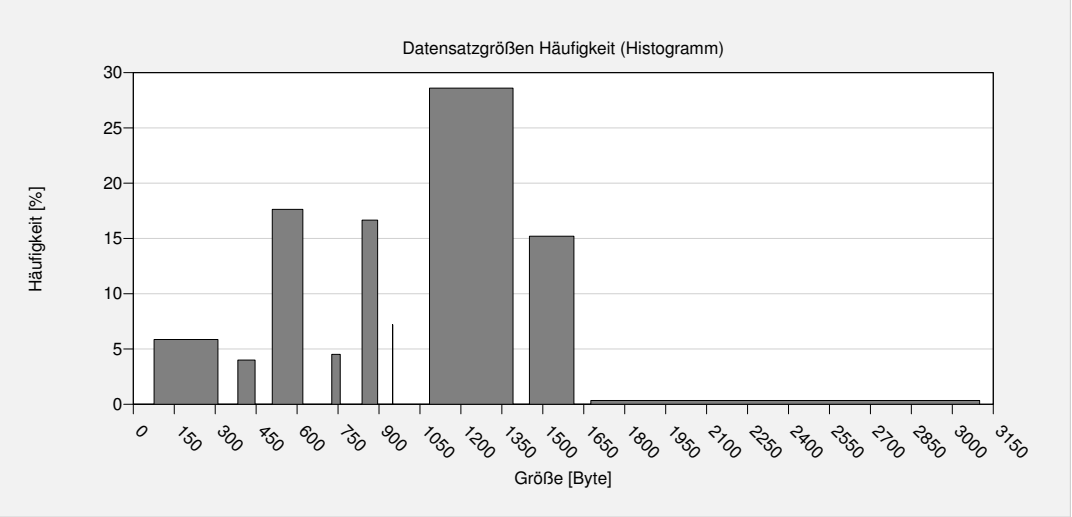


Abbildung 4.1.: Verteilung der Datensatzgrößen. Die Datensätze wurden zusammengefasst, sodass die Gruppen mindestens 50 Byte auseinander liegen und mindestens 50 Dateien enthalten. In der letzten Gruppe sind die restlichen 12 Datensätze.

4.3. Entropie

Bei der Analyse der Daten zeigt sich, dass von den 128 möglichen ASCII-Symbolen nur 31 genutzt werden. Dabei wird das Symbol »`0`« besonders häufig benutzt und bildet mit 29% fast ein Drittel der Ausgaben. Die Symbole »`R`« und »`O`« sind sehr selten und werden nur in dem Wort »`ERROR`« verwendet. Abbildung 4.2 zeigt das Symbol-Histogramm.

Wird die Wahrscheinlichkeit der Symbole über alle Dateien betrachtet ergibt sich nach der Gleichung 2.5 eine Entropie der Quelle von

$$H_1(\mathcal{X}) \approx 3,61\,\text{Bit}\,.$$

Eine Entropie-Kodierung kann daher theoretisch eine Kompressionsleistung von fast 55% erreichen. (Für die Formel der Kompressionsleistung siehe Gleichung 7.1.)

Die Auslesungen können jedoch sehr unterschiedliche Daten enthalten. Einige Auslesungen bestehen z.B. zu einem großen Teil aus Prüfsummen. In Prüfsummen sind die Wahrscheinlichkeiten der Symbole »A«, »B«, »C«, »D«, »E« und »F« höher als in den anderen Daten, da die Prüfsummen hexadezimal dargestellt werden, während die meisten anderen Zählerwerte dezimal dargestellt werden. Einige Datensätze bestehen nur aus Prüfsummen.

Wird die Entropie für jede Datei einzeln berechnet, ergibt sich im Mittel eine Entropie von

$$H_1(\mathcal{X}) \approx 3,53\,\text{Bit}\,.$$

Gegenüber der Entropie-Kodierung für alle Auslesungen könnte sich eine spezifische Entropie-Kodierung auf eine Mittlere Kompressionsleistung von 56% verbessern. Abbildung 4.3 zeigt die Entropie der einzelnen Dateien. Die minimale Entropie beträgt 2,57 Bit, die maximale Entropie 4,11 Bit.

Wenn jeweils zwei, drei oder vier Zeichen zu einem Symbol zusammengefasst werden, um zu berücksichtigen, dass bestimmte Symbolfolgen häufiger sind als andere, ergeben sich Entropien von:

$$H_2(\mathcal{X}) \approx 5,74\,\text{Bit}$$
$$H_3(\mathcal{X}) \approx 7,48\,\text{Bit}$$
$$H_4(\mathcal{X}) \approx 8,76\,\text{Bit}$$

Dadurch wären durch entsprechende Entropie-Kodierung Kompressionsgrade von 64%, 69% und fast 73% respektive möglich.

4.4. Übergangswahrscheinlichkeiten

Die Übergangswahrscheinlichkeiten zeigen für ein gegebenes Symbol die Wahrscheinlichkeiten der Folgesymbole. Durch die Struktur und die Art der Daten sind hier besonders viele Korrelationen zu erwarten. Nach dem Symbol `)` kommt z.B entweder das Symbol `(` oder das Steuersymbol `<CR>`, auf das Symbol `k` folgt immer das Symbol `W` etc. Abbildung 4.4 zeigt in einer Heatmap die Wahrscheinlichkeit der Folgesymbole.

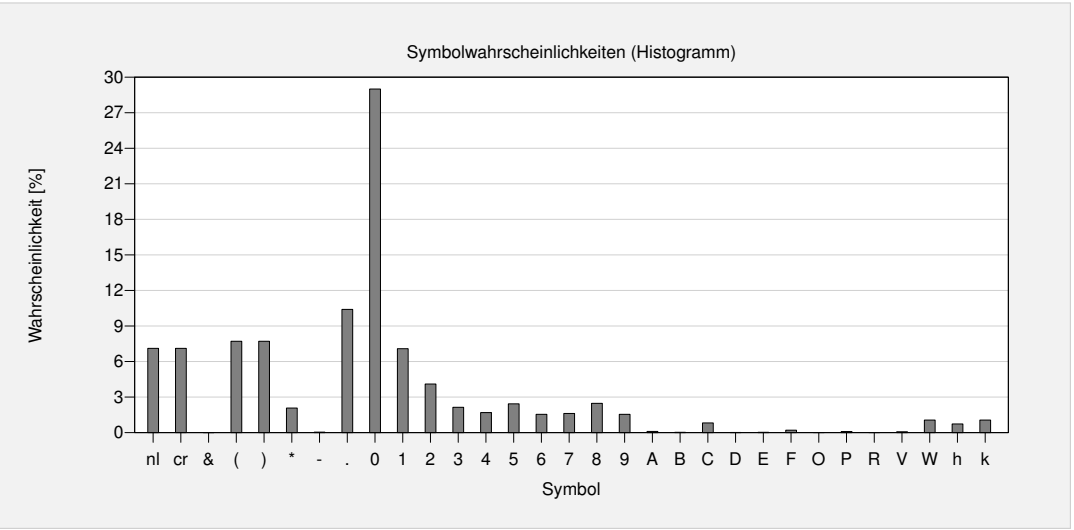


Abbildung 4.2.: Symbolverteilung

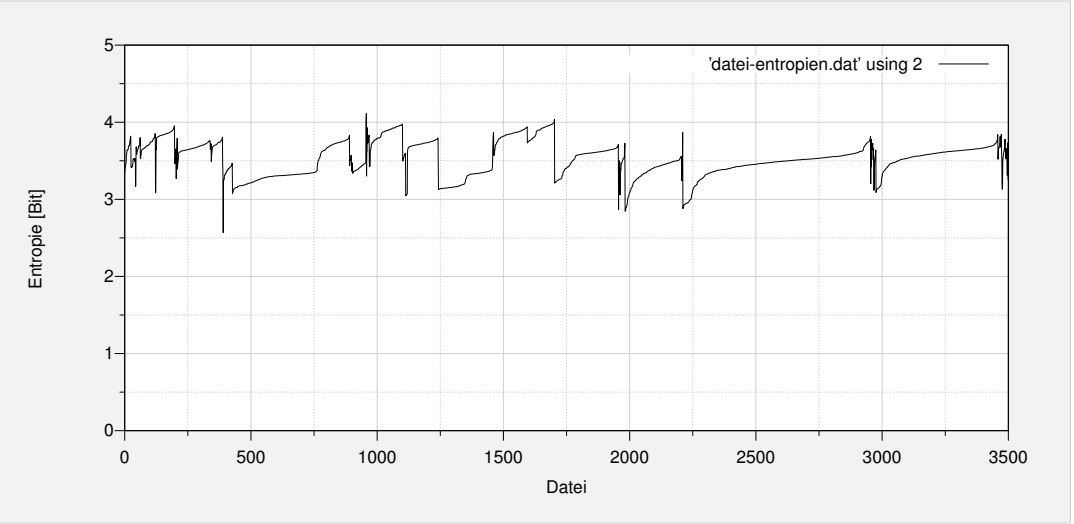


Abbildung 4.3.: Entropie der einzelnen Dateien. Die Dateien sind nach Größe geordnet.

Es ist eine Entropie-Kodierung möglich, welche die Übergangswahrscheinlichkeiten berücksichtigt. Mit der Formel 4.1 kann die bedingte Entropie für solch eine Markov-Kette berechnet werden.

$$H_M(\mathcal{X}) = -\sum_{z\in\mathcal{Z}} p(z) \cdot \sum_{y\in\mathcal{Z}} p(y|z) \cdot \log_2 p(y|z) \tag{4.1}$$

Für die Wahrscheinlichkeiten aus den Datensätzen errechnet sich eine bedingte Entropie von

$$H_M(\mathcal{X}) \approx 2,12\,\text{Bit}\,.$$

Eine entsprechende Entropie-Kodierung könnte einen Kompressionsgrade von mehr als 73% erreichen und wäre damit aussichtsreicher, als eine Vier-Symbol-Entropie-Kodierung. Allerdings bleibt zu bedenken, dass in verschiedenen Auslesungen die Übergangswahrscheinlichkeiten unterschiedlich sein können. In Auslesungen, in denen viele Prüfsummen übertragen werden und die damit nicht den Erwartungen entsprechen, wird eine solche Kodierung nicht die gleichen Ergebnisse erreichen können.

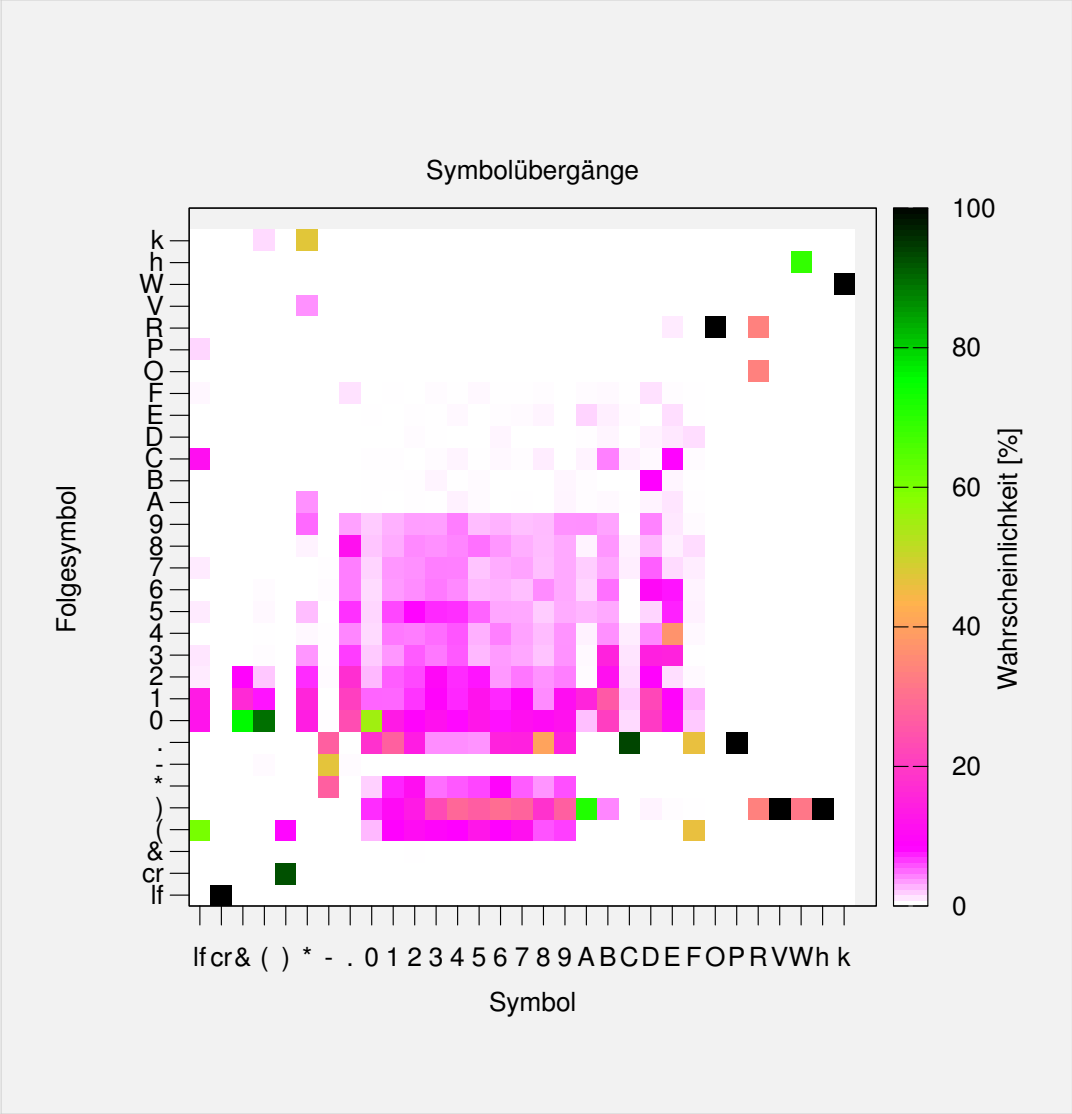


Abbildung 4.4.: Heatmap der Folgesymbol-Wahrscheinlichkeiten

Kapitel 5

Effiziente Kompressionsverfahren für Smart Metering

Für diese Arbeit wurden verschiedene Verfahren für die Verwendung auf den Sensorknoten angepasst und für die analysierten Daten optimiert. Bisherige Arbeiten haben ihren Schwerpunkt auf geringen Stromverbrauch der Verfahren gelegt. Da der Stromverbrauch bei Stromzählern keine so große Rolle spielt wie bei Sensoren, die mit sehr geringen Energieressourcen arbeiten müssen, liegt der Schwerpunkt der Verfahren dieser Arbeit auf geringem Speicherverbrauch und geringer Codegröße.

5.1. Adaptive-Trimmed-Huffman (ATH)

Das Adaptive-Trimmed-Huffman-Verfahren muss für die gegebenen Daten angepasst werden. Dabei muss festgestellt werden, welcher Baum die besten Komprimierungsergebnisse liefert. Die aufgetretenen Symbole werden in eine Liste eingetragen und diese nach der Häufigkeit der Symbole geordnet. Jedes Mal, wenn beim Einlesen der Datensätze ein Symbol auftritt, das sich in der Liste befindet, wurde ein Zähler für diese Position erhöht. Angewendet auf alle Datensätze erhält man eine Liste mit Häufigkeiten der Positionen in der Liste.

Mit diesen Häufigkeiten wurde ein Huffman-Baum erstellt und dessen Codelängen ermittelt. Tabelle 5.1 zeigt, wie häufig Codes einer bestimmten Länge vorkommen.

Codelänge [Bit]	1	2	3	4	5	6	7	≥ 8
Häufigkeit	0	1	2	4	3	6	3	13

Tabelle 5.1.: Optimale Huffman-Code-Längen

Ein Baum mit diesen Codelängen wäre optimal. Da jedoch ein Präfix-Bit eingefügt wird, um Symbole, die noch nicht in der Liste stehen, einzuleiten und dadurch auch Codelängen von mehr als 8 Bit obsolet werden, muss der Baum angepasst werden.

Da der Baum getrimmt ist und ausgenutzt wird, dass von einem Byte nur 7 Bit genutzt werden, ist sichergestellt, dass die Ausgabe nie größer als die Eingabe ist.

5.2. Lempel-Ziv-Welch (LZW)

Es wurde ein LZW-Komprimierer implementiert. Als initialisierte Symbole wurden nur die 31 vorkommenden Symbole verwendet. Es wurden verschiedene Tabellengrößen getestet.

Die genutzten Bytes der Ausgabe verändern sich je nach Einträgen im Wörterbuch. Existieren weniger als 32 Einträge im Wörterbuch, werden für die Kodierung 5 Bit genutzt, bei weniger als 64 Einträgen 6 Bit und so weiter. So ist sichergestellt, dass bereits bei kleinen Eingabedaten eine Komprimierung stattfindet.

5.3. Lempel-Ziv-Welch mit statischem Wörterbuch (SLZW)

Die SLZW-Kodierung arbeitet wie die LZW-Kodierung mit einem Wörterbuch. Da in den Auslesungen jedoch häufig gleiche Symbolfolgen zu erwarten sind, wird ein festes Wörterbuch benutzt, anstatt das Wörterbuch während der Übertragung aufzubauen. Somit können von Anfang an längere Symbolfolgen referenziert werden.

Zur Feststellung, welche Symbolfolgen benutzt werden sollten, wurde der LZW-Algorithmus mit einem dynamisch wachsenden Wörterbuch auf alle Daten angewandt. Anschließend wurde gezählt, wie häufig die so entstandenen Wörterbucheinträge referenziert wurden. Die Folgen wurden der Häufigkeit nach geordnet in ein Array eingetragen. Zur Überprüfung, wie groß das statische Wörterbuch sein sollte, wurden

von dem Array nacheinander nur die ersten 32, 64, 128, 256, 512, 1024, 2048 und 4096 Einträge benutzt und die Ergebnisse miteinander verglichen (siehe Kap 7.2.4).

Die Tabelle wird wie in der LZW Implementierung gespeichert. Allerdings muss diese Tabelle direkt im Programmcode implementiert werden, wodurch dieser größer wird, während der RAM-Verbrauch dagegen sinkt.

Da das Wörterbuch von Beginn an gefüllt ist, muss vom ersten Symbol an jede Wörterbuchreferenz mit den benötigten Bits kodiert werden. Bei 512 Einträgen müssen also immer 9 Bit genutzt werden. Dadurch ist es theoretisch möglich, dass die Ergebnisse größer als die Eingabedaten sind.

5.4. Markov-Chain-Huffman (MCH)

Dieser Algorithmus macht sich die Übergangswahrscheinlichkeiten der Symbole zunutze (siehe Kap 4.4). Für jedes Symbol wurde ein eigener Huffman-Baum für die Folgesymbole erstellt.

Zu Beginn einer Nachricht wird angenommen, dass das letzte Symbol ein Zeilenumbruch <LF> gewesen ist. Für dieses letzte Symbol wird dann in dem Huffman-Baum für dieses Symbol nachgeschlagen, wie das folgende Symbol zu kodieren ist. Da einige Symbole nur ein mögliches Folgesymbol besitzen, könnte das Folgesymbol also sogar mit 0 Bit kodiert werden. Tatsächlich muss jedoch berücksichtigt werden, dass sich das Format der Nachricht von den Daten unterscheiden kann, die dieser Arbeit zur Verfügung standen. Daher wird in jedem Baum ein Symbol für ein nicht berücksichtigtes Symbol eingefügt, wodurch sich die kürzeste Kodierung auf 1 Bit verschlechtert.

Tritt ein unvorhergesehenes Symbol auf, wird der entsprechende Code ausgegeben, gefolgt vom ASCII-Code des Symbols. Wenn ein Symbol gar nicht vorgesehen ist und daher auch kein Huffman-Baum der Folgesymbole existiert, werden die Folgesymbole so lange ASCII-kodiert ausgegeben, bis wieder ein Huffman-Baum für das letzte Symbol gefunden wird.

Es wurden daher Tests durchgeführt, bei denen die Bäume aus einer Lernmenge erstellt wurden und das Verfahren anschließend auf die restlichen Datensätze angewendet wurde. Dabei wurden verschiedene Varianten an Präfix-Codes für nicht vorgesehene Symbole eingefügt. Außerdem wurden verschiedene Beschränkungen,

wie der maximalen Anzahl an Symbolen im Baum und einer maximalen Codewortlänge, vorgegeben.

Die Präfix-Codes wurden bei der Erstellung der Bäume berücksichtigt. Die Varianten bestanden aus einem Präfix-Bit, einem Symbol mit der Wahrscheinlichkeit 0 und einem Symbol mit der mittleren Wahrscheinlichkeit der anderen Symbole (siehe Kap 7.2.5).

Durch das Verfahren wird eine sehr gut Komprimierung erwartet. Auch der RAM-Verbrauch wird voraussichtlich sehr gering sein, da keine dynamischen Listen oder Wörterbücher benötigt werden. Allerdings müssen die verschiedenen Huffman-Bäume im Programmcode fest implementiert sein, sodass dieser, verglichen mit den anderen Methoden, größer ausfällt.

5.5. Adaptive Markov-Chain-Huffman (AMCH)

Die Adaptive Markov-Chain-Huffman-Kodierung (AMCH) verbindet die Ideen der Adaptive-Trimmed-Huffman- (ATH) mit denen der Markov-Chain-Huffman-Kodierung (MCH). Statt einer Liste mit Symbolen, geordnet nach deren Häufigkeit, wird für jedes Symbol eine eigene Liste der Folgesymbole geführt.

Wie bei der MCH-Kodierung wird zu Beginn als letztes Symbol der Zeilenumbruch `LF` angenommen. Da der Zeilenumbruch zuvor noch nicht vorgenommen ist, existiert noch keine Liste der Folgesymbole. Diese wird angelegt und das erste Symbol eingetragen. Wenn in der Liste der Folgesymbole noch keine Einträge vorhanden sind, wird das Folgesymbol ASCII-kodiert ausgegeben.

Existiert bereits eine Liste der Folgesymbole, wird in dieser nach dem aktuellen Symbol gesucht. Wird es gefunden und ist es an einer Position, die der Baum darstellen kann, wird der entsprechende Huffman-Code ausgegeben. Wird es nicht gefunden oder ist es in der Liste weit hinten, wird ein im Baum vorgesehener Code für ASCII-kodierte Symbole ausgegeben, gefolgt von dem ASCII-Code des Symbols. Ist das Symbol nicht in der Liste, wird es eingetragen.

Je nach Länge der Liste werden verschiedene Bäume benutzt. Der letzte Eintrag im Baum ist immer für ein nicht im Baum vertretenes Symbol vorgesehen. Für die Berechnung der Bäume wurde ähnlich vorgegangen wie bei der Berechnung des Baumes für das ATH-Verfahren. Das AMCH-Verfahren wurde in einer abgewandelten

Form auf die Datensätze angewandt. Statt die Daten zu kodieren, wurden die Treffer in den verschiedenen Listen ermittelt.

Für jedes Symbol wurde ein Array angelegt, in dem die Folgesymbole während des Einlesens der Daten, eingetragen wurden. Die Arrays wurden dabei stets nach der Häufigkeit der Symbole geordnet. Je nachdem, ob ein Symbol beim Auftreten bereits in dem Array steht (Treffer) oder nicht (kein Treffer), wurde es in das Array aufgenommen oder dessen Häufigkeit erhöht.

Für jede mögliche Länge eines Arrays wurde ein weiteres Array angelegt. In dem zweiten Array wurden die Treffer pro Position gezählt. Am Ende konnte man so für jede Array-Länge der Folgesymbole die Häufigkeiten der Treffer pro Position ermitteln. Aus diesen Häufigkeiten konnten verschiedene Huffman-Bäume erzeugt werden, die bei der Kodierung verwendet werden. Je nachdem, wie lang ein Array der Folgesymbole für ein Symbol ist, wird ein bestimmter Huffman-Baum gewählt.

Bei der Erzeugung der Huffman-Bäume wurden auch die Nicht-Treffer berücksichtigt. Der Code dieser Nicht-Treffer wird als Präfix-Code für Symbole verwendet, die sich nicht in dem Array der Folgesymbole befinden. Da diese Präfix-Codes nicht alleine stehen, sondern von einem ASCII-Code begleitet werden, wurden verschiedene Maßnahmen getestet, die Länge der Präfix-Codes anzupassen. In Variante 1 wurde der Baum ohne Modifikation erzeugt, in Variante 2 bis 4 wurden die Anzahl der Nicht-Treffer jeweils mit 8, 16 und 32 abzüglich der Array-Länge multipliziert (siehe Kap 7.2.6).

Das Verfahren wird einen größeres Speicherverbrauch haben als die ATH-Kodierung, da mehrere Listen der Symbole geführt werden müssten.

5.6. tiny LZMA (TLZMA)

Bei tiny LZMA (TLZMA) handelt es sich um eine eigene Implementierung der LZMA-Kodierung (siehe Kap 3.3), die für die Stromzählerdaten optimiert ist. Es wird auf eine Bereichskodierung verzichtet, da diese zu aufwändig wäre. Außerdem wurden die Präfix-Codes modifiziert.

Es wird ausgenutzt, dass nur ASCII-Symbole, also 7 Bit, benutzt werden. Dadurch ist garantiert, dass die resultierende Datei im schlechtesten Fall genauso groß ist, wie die ursprüngliche Datei. Das Suchfenster der Symbolfolgen wurde auf 128 Byte

Code	Beschreibung
0 + ASCII-Code	7 Bit ASCII-Code
10 + Offset + Länge	LZ77 Referenz, Offset ist ein 7 Bit Wert (1 bis 128)
110 + Länge	LZ77 Referenz (Distanz gleich wie beim letzten)
1110 + Länge	LZ77 Referenz (Distanz gleich wie beim vorletzten)
11110 + Länge	LZ77 Referenz (Distanz gleich wie beim drittletzten)
11111 + Länge	LZ77 Referenz (Distanz gleich wie beim viertletzten)

Tabelle 5.2.: Präfix-Codes der TLZMA-Kodierung

Präfix	Extra Bits	Längen
0	3	2 - 9
10	3	10 - 17
11	8	18 - 273

Tabelle 5.3.: Referenzlängen-Kodierung beim TLZMA-Verfahren.

beschränkt. Dadurch und durch das Fehlen der Bereichskodierung sind die Codegröße und der Speicherverbrauch geringer als beim LZMA-Verfahren.

Ein Präfix Code bestimmt, ob ein 7 Bit ASCII-Symbol oder eine Referenz folgt. Tabelle 5.2 zeigt die verschiedenen Präfix-Codes.

Damit auch Referenzen für kurze Symbolfolgen eine Ersparnis bringen, ist die Kodierung der Referenzlänge unterschiedlich lang. Kurze Referenzlängen werden kürzer kodiert als lange. Tabelle 5.3 zeigt die Kodierung der Länge. Somit ist eine kurze Referenz, je nachdem ob der Offset in der History steht oder übertragen werden muss, zwischen 7 und 12 Bit lang. Die längste Referenz ist 19 Bit lang.

Obwohl eine Referenz höchstens 128 Byte entfernt beginnen kann, kann sie bis zu 273 Byte lang sein. Das liegt daran, dass der referenzierte Text den Beginn der Folgesymbole enthalten kann. Ohne diese Möglichkeit würde der Beispieltext `abcabcabc` nur zu `abc(3,3)(3,3)` verkürzt werden können. Durch das mögliche Überschneiden von referenziertem Text und der Folgesymbole kann der Text zu `abc(3,6)` verkürzt werden. Somit wird eine Art der Lauflängen-Kodierung unterstützt.

Code	Beschreibung
1 +	ATH Code
00 + ASCII-Code	7 Bit ASCII-Code
010 + Offset + Länge	LZ77 Referenz, Offset ist ein 7 Bit Wert (1 bis 128)
0110 + Länge	LZ77 Referenz (Distanz gleich wie beim letzten)
01110 + Länge	LZ77 Referenz (Distanz gleich wie beim vorletzten)
011110 + Länge	LZ77 Referenz (Distanz gleich wie beim drittletzten)
011111 + Länge	LZ77 Referenz (Distanz gleich wie beim viertletzten)

Tabelle 5.4.: Präfix-Codes der LZMH-Kodierung

5.7. Lempel-Ziv-Markov-Chain-Huffman (LZMH)

Lempel-Ziv-Markov-Chain-Huffman ist eine Weiterentwicklung des TLZMA-Verfahrens, die der tatsächlichen LZMA-Kodierung näher kommt, da auf die ASCII-Symbole eine ATH-Kodierung angewandt wird, wodurch der Kompressionsgrad weiter gesteigert wird. Im schlechtesten Fall kann die Nachricht dadurch $12,5\%$ (ein Bit mehr pro Byte) größer werden. Dies kann jedoch für die Datenstruktur ausgeschlossen werden, da nur wenig Symbole genutzt werden.

Die Liste der Symbole für die ATH-Kodierung (s. Kap 3.5) wird nur von ASCII- und ATH-Codes aktualisiert, d.h. Symbolfolgen, die durch Referenzen kodiert werden, aktualisieren nicht die Liste der vorgekommenen Symbole. Für die Erstellung des ATH-Baumes wurde wie beim ATH-Verfahren vorgegangen. Allerdings wurden auch »Nicht-Treffer« und Referenzen gezählt. So konnte ein Baum berechnet werden, der nicht durch das Präfix-Bit angepasst werden musste.

Die Präfix-Codes sind in Tabelle 5.4 dargestellt. Für die Längen wird die Kodierung aus Tabelle 5.5 genutzt. Da die ATH-Kodierung die Symbole komprimiert, die nicht in einer Referenz liegen und die kürzeste Referenz zwischen 8 und 13 Bit lang ist, wird hier erst eine Referenz ab einer Länge von drei Symbolen, im Vergleich zu zwei Symbolen beim TLZMA-Verfahren, verwendet.

Präfix	Extra Bits	Längen
0	3	3 - 10
10	3	11 - 18
11	8	19 - 274

Tabelle 5.5.: Referenzlängen-Kodierung beim LZMH-Verfahren.

Kapitel 6

Implementierung

In diesem Kapitel wird auf die Implementierung der angepassten Kompressionsverfahren aus Kap. 5 eingegangen. Am Ende des Kapitels werden die Programmgröße der kompilierten Implementierungen, der Speicherverbrauch und die Kompressionszeit der verschiedenen Verfahren miteinander verglichen.

Die Algorithmen wurden mit dem GCC (GNU Compiler Collection) in der Version 4.6.1 und dem Optimierungsparameter -OS, also einer Optimierung auf kleine Programmgrößen, für den Cortex-M3 Mikroprozessor von ARM kompiliert. Aus den Kompressionsalgorithmen wurden statische Bibliotheken erstellt und deren Programmgrößen gemessen.

Alle Verfahren verwenden keine dynamische Speicherallozierung, d.h. alle Variablen liegen auf dem Stack, um eine Fragmentierung und Speicherlecks zu verhindern.

6.1. Schnittstelle

Die Verfahren wurden in C++ implementiert. Jedes Verfahren wird durch eine Klasse mit der statischen Methode `Compress` implementiert (s. Listing 6.1). Die Methode erhält als Parameter einen Zeiger auf den Beginn der zu komprimierenden Nachricht im Speicher und dessen Größe in Byte. Die komprimierte Nachricht wird an die Position der ursprünglichen Nachricht geschrieben und überschreibt diese, um keinen zusätzlichen Speicher zu beanspruchen. Die Größe in Byte der komprimierten Nachricht wird von der Funktion zurückgegeben.

Wird die Klasse mit der Konstanten `DECOMPRESSOR` kompiliert, befindet sich in ihr auch die Dekompressionsmethode `DeCompress`. Diese Methode erhält die gleichen Parameter wie die Kompressionsfunktion und einen zusätzlichen Zeiger für das Ergebnis der Dekompression. Da das Ergebnis weit größer als die Eingabedaten sein kann, wäre ein Überschreiben dieser Daten nicht praktikabel. Weiterhin wird die Nachricht erst beim Empfänger dekomprimiert, bei dem keine engen Ressourcengrenzen vorliegen.

```
class Compress {
public:
   static size_t Compress(uint8_t* buffer, size_t insize);
#ifdef DECOMPRESSOR
   static size_t DeCompress(uint8_t* inbuffer, size_t insize, uint8_t* outbuffer);
#endif
};
```

Listing 6.1: Öffentliche statische Funktionen der Kompressionsklassen

6.2. Implementierung der Kompressionsverfahren

In diesem Abschnitt werden die Implementierungen der Verfahren und ihre jeweiligen Besonderheiten vorgestellt.

6.2.1. Adaptive-Trimmed-Huffman (ATH)

Der zentrale Teil der ATH-Implementierung (s. Kap 5.1) ist ein Array. Jedes Element des Arrays enthält das Symbol und dessen Häufigkeit (s. Listing 6.2). Das Array wird bei jedem Aufruf der Methode auf Null initialisiert und die Symbole werden bei ihrem ersten Auftreten eingetragen. Bei der Änderung einer Häufigkeit in dem Array wird dieses gegebenenfalls mit der »Blasensortierung«-Methode umsortiert, sodass das Array immer nach der Häufigkeit der Symbole geordnet ist. Das häufigste Symbol steht an erster Stelle des Arrays. Die Größe des Arrays kann durch eine Konstante zur Kompilierung eingestellt werden und ist auf 48 voreingestellt.

Beim Auftreten eines Symbols wird anhand der Position in dem Array ein Huffman-Code gewählt und ausgegeben. Die Huffman-Codes sind in einem statischen, kon-

stanten Array vorgegeben. Ein Symbol, welches im Array an erster Position steht, wird durch den Huffman-Code an Position Eins kodiert. Findet sich für die Position des Symbols kein Huffman-Code oder ist das Symbol noch nicht in dem Array eingetragen, so wird es durch ein Präfix-Bit eingeleitet und ASCII-kodiert ausgegeben.

Die Codelänge eines ASCII-Symbols kann eingestellt werden. Sie ist auf 7 Bit voreingestellt, sodass mit dem Präfix-Bit ein nicht Huffman-kodiertes Symbol mit insgesamt 8 Bit kodiert wird.

Es findet keine Überlaufüberprüfung des Symbol-Häufigkeit-Zählers statt. Da dieser mit 16 Bit erst beim 65 536-maligen Auftreten überlaufen würde. Bei einer maximalen Datensatzgröße von 10 000 Byte kann diese Zahl nicht erreicht werden.

```
typedef struct {
   uint8_t  symbol;
   uint16_t count;
} hufflist_t;
```

Listing 6.2: Definition der Array Elemente.

6.2.2. Lempel-Ziv-Welch (LZW)

In der LZW-Implementierung (s. Kap 5.2) wird ein Array angelegt, in dem das Wörterbuch gespeichert wird. Um den RAM-Verbrauch gering zu halten, werden in dem Array nicht ganze Strings gespeichert, sondern jedes Element des Arrays besteht aus einem Index auf das vorhergegangene Wort im Wörterbuch und dem Folgesymbol. Zu Beginn wird das Wörterbuch mit den möglichen Symbolen initialisiert, wobei der Index einen vordefinierten Wert bekommt, der signalisiert, dass dieser Eintrag der Beginn eines Wortes ist. Tabelle 6.1 zeigt, wie solche Einträge aussehen können.

Durch diese Speicherung vereinfacht sich auch die Suche nach dem längsten und passenden String im Wörterbuch. Beim Komprimieren wird in einer Schleife über das Array iteriert. In einer Variablen wird der Index des letzten Treffers im Wörterbuch gespeichert. Dieser Index wird mit dem vordefinierten Wert initialisiert, der anzeigt, dass es sich um den Beginn eines Wortes handelt. Während über das Array iteriert wird, wird überprüft, ob der gespeicherte Index mit dem Wert des vorhergehenden Eintrages übereinstimmt und ob das Symbol an der aktuellen Pufferposition dem

Symbol in der aktuellen Array Position entspricht. Wird so ein Treffer gefunden, wird der Index des Array-Eintrags gespeichert und weiter iteriert.

Ist die Schleife beendet, wird der zuletzt gespeicherte Index ausgegeben und ein neuer Eintrag im Array vorgenommen, mit dem gespeicherten Index als vorhergehendes Wort und dem aktuellen Symbol als Folgesymbol. Dadurch ist sichergestellt, dass ein Wort, das aus einem anderen Wort besteht, im Array weiter hinten folgt. Neue Einträge in dem Wörterbuch werden nur bis zu der vordefinierten Wörterbuchgröße vorgenommen.

Im Folgenden wird ein Ausschnitt aus der Implementierung gezeigt, der die Definition der Listenelemente und die Suche nach Treffern in dem Wörterbuch zeigt.

```
typedef  struct {
   code_t   prev;
   uint8_t  symbol;
} LZWLIST_t;

size_t CompressLZW::Compress(uint8_t* buffer, size_t insize) {
   [..]
   static LZWLIST_t  Table[LZWLENGTH];
   [..]
      for (uint16_t i = 0; i < count && inpos < insize; i++) {
         if (last == Table[i].prev && buffer[inpos] == Table[i].symbol) {
            last = i;
            inpos++;
         }
      }
   [..]
```

Listing 6.3: Ausschnitte aus der LZW Implementierung

Die Ausgabe des Index erfolgt je nach Einträgen im Wörterbuch. Bei Einträgen von bis zu 32 Wörtern kann der Index mit 5 Bit kodiert werden, bei bis zu 64 Einträgen mit 6 Bit u.s.w. Für die Dekomprimierung wird am Ende der Ausgabe die ursprüngliche Datensatzgröße in zwei Byte kodiert.

Da es bei einer Wörterbuchgröße von mehr als 256 Einträgen vorkommen kann, dass die Ausgabe größer ist als die Eingabe, wird in diesem Fall ein Schreibpuffer angelegt, in den die Ausgabe geschrieben wird, bevor sie die Eingabedaten überschreibt.

Einstellungen: Über Konstanten kann zur Kompilierzeit eingestellt werden, wie groß das Wörterbuch und wie groß der Schreibpuffer ist. Außerdem kann eingestellt

Nummer	Referenz	Symbol	ganzer String
1	-	a	a
2	-	b	b
3	-	c	c
4	1	b	ab
5	4	c	abc

Tabelle 6.1.: Beispiele für Einträge in einem LZW-Wörterbuch

werden, wie viele Symbole bei der Initialisierung des Wörterbuches eingetragen werden. In den Standard-Einstellungen werden nur die 31 bekannten Symbole eingetragen, es können jedoch auch 98 Symbole eingetragen werden, die sich aus den bekannten 31 Symbolen und allen weiteren ASCII-Symbolen, ausschließlich der Steuerzeichen, zusammensetzen. Die Steuerzeichen wurden herausgenommen, damit bei einer Wörterbuchgröße von 128 noch neue Einträge in das Wörterbuch eingetragen werden können. Es wurde angenommen, dass eine Erweiterung des Ausgabestandards ohne weitere Steuerzeichen auskommt.

6.2.3. Lempel-Ziv-Welch mit statischem Wörterbuch (SLZW)

Das SLZW-Verfahren (s. Kap 5.3) ist ähnlich wie das LZW-Verfahren umgesetzt, allerdings werden keine neuen Symbolfolgen in das Wörterbuch eingetragen, sodass dieses keinen zusätzlichen Arbeitsspeicher belegt und fest im Programm-Code implementiert ist.

Einstellungen: Über eine Konstante kann zu Kompilierzeit eingestellt werden, wie groß das Wörterbuch ist. Die Ausgabe des Index erfolgt je nach Größe des Wörterbuchs. Bei einem Wörterbuch von 128 Einträgen werden 7 Bit verwendet, bei einem Wörterbuch von 512 Einträgen werden 9 Bit verwendet. Dadurch ist es möglich, dass die Ausgabe größer ist als die Eingabe und es wird wie beim LZW-Verfahren ein Ausgabepuffer verwendet. Die Größe des Ausgabepuffers kann über eine Konstante zur Kompilierzeit eingestellt werden.

6.2.4. Markov-Chain-Huffman (MCH)

Bei der MCH-Implementierung (s. Kap 5.4) wurden 31 verschiedene Huffman-Bäume fest im Programm-Code gespeichert. Jeder dieser Bäume steht für ein Symbol, die Blätter der Bäume stehen für die möglichen Folgesymbole.

Abbildung 6.1 zeigt den Ablauf der Kodierung. Zu Beginn wird als vorheriges Symbol der Zeilenumbruch initialisiert. In dem Baum für das letzte Symbol wird nach dem aktuellen Symbol gesucht. Wird das Symbol gefunden, wird der entsprechende Huffman-Code des Symbols ausgegeben. Das aktuelle Symbol wird als letztes Symbol gespeichert und die Position im Puffer erhöht.

Wird das aktuelle Symbol im Baum nicht gefunden, befindet sich in jedem Baum ein Symbol, welches nicht gefundene Symbole repräsentiert. Der Huffman-Code wird ausgegeben, gefolgt von dem ASCII-Code des aktuellen Symbols. Anschließend wird das aktuelle als letztes Symbol gespeichert und die Position im Puffer erhöht.

Wird für das letzte Symbol kein Baum gefunden, wird das aktuelle Symbol ASCII-kodiert ausgegeben und als letztes Symbol gespeichert.

Einstellungen: Da die Huffman-Codes länger als 8 Bit sein können, kann es passieren, dass die Ausgabe größer ist als die Eingabe. Daher wird ein Ausgabepuffer implementiert, dessen Größe zur Kompilierzeit durch eine Konstante eingestellt werden kann. Der Puffer ist standardmäßig auf 8 Byte gesetzt.

6.2.5. Adaptive Markov-Chain-Huffman (AMCH)

Die AMCH-Implementierung (s. Kap 5.5) arbeitet ähnlich wie die ATH-Implementierung, allerdings wird für jedes Symbol ein eigenes Array der Folgesymbole geführt. Die Anzahl der Arrays und ihre Größe kann durch eine Konstante zur Kompilierzeit eingestellt werden.

Die Größe der Array-Elemente wurde von drei auf zwei Byte verringert, da angenommen wird, dass der Zähler für die Häufigkeit der Folgesymbole selten über 255 steigt. Erreicht ein Folgesymbolzähler 255, wird dieser nicht mehr erhöht und behält von nun an seine Position im Array.

Ein weiterer Unterschied zur ATH-Implementierung ist, dass mehrere Huffman-Bäume statisch im Programm-Code implementiert wurden. Je nachdem, wie viele Symbole in dem Array der Folgesymbole eingetragen sind, wird ein Baum ausgewählt.

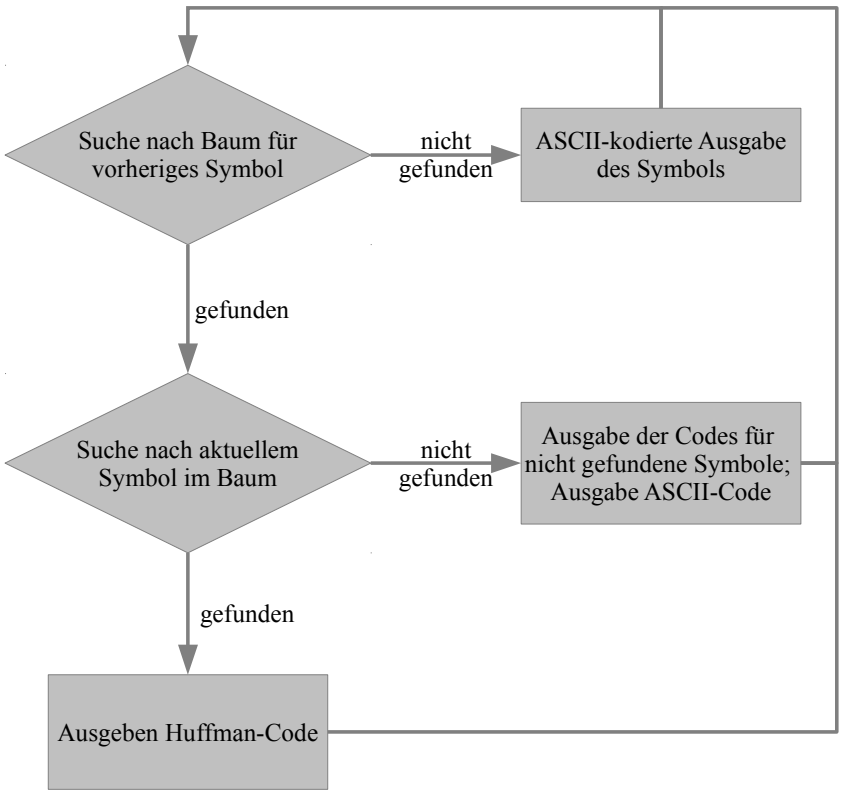


Abbildung 6.1.: Ablauf des MCH-Verfahrens

In diesem Baum befindet sich auch ein Code für nicht durch den Baum kodierbare Symbole, sodass kein Präfix-Bit benötigt wird.

Da die Huffman-Codes nicht in ihrer Länge beschränkt sind, ist ein Ausgabepuffer nötig. Die Größe des Ausgabepuffers kann über eine Konstante zur Kompilierzeit eingestellt werden.

6.2.6. tiny LZMA (TLZMA)

Das TLZMA-Verfahren (s. Kap 5.6) wurde beginnend mit der LZSS-Variante (s. Kap 3.4) implementiert. Statt der Byte-weisen Ausgabe wird ein Bitstrom ausgegeben. Außerdem werden die letzten vier Offsets gespeichert und können kürzer referenziert werden.

Es wird ausgehend von der aktuellen Position im Puffer überprüft, ob die folgenden Symbole bereits in dieser Reihenfolge übertragen wurden. Dafür wird schrittweise im Puffer zurück iteriert und überprüft, ob die folgenden Symbole im vergangenen Puffer mit denen im folgenden übereinstimmen. Werden solche Ketten gefunden, werden der Offset und die Länge der Symbolfolge gespeichert. Ist bereits eine solche Kette gefunden worden, wird nach längeren Ketten weiter gesucht. Die Suche erstreckt sich bis zu 128 Symbole vor das aktuelle Symbol.

Wurde eine Symbolfolge von mindestens zwei Symbolen gefunden, wird zunächst überprüft, ob der Offset in dem Array der letzten vier Offsets steht. Ist dies der Fall wird er durch einen Bitstrom referenziert, welcher der Position in dem Array entspricht. Ansonsten wird der Offset, eingeleitet durch ein Präfix-Bit, direkt ausgegeben. Die Länge wird präfixkodiert ausgegeben. Kleine Zahlen werden mit wenigen Bit kodiert, größere mit mehr.

Wurde keine Symbolfolge gefunden, wird das aktuelle Symbol, durch ein Präfix-Bit eingeleitet, ASCII-kodiert ausgegeben. Die Anzahl an Bit pro Symbol kann durch eine Konstante zur Kompilierzeit eingestellt werden und ist auf 7 Bit voreingestellt. Dadurch ist sichergestellt, dass das Ergebnis nicht größer sein kann als die Eingabe.

Einstellungen: Da die letzten 128 Byte als Suchfenster benötigt werden, können diese nicht direkt von der Ausgabe überschrieben werden. Daher wird bei der Implementierung ein Ausgabepuffer von mindestens 129 Byte benötigt. Die Größe des Puffers kann zur Kompilierzeit durch eine Konstante eingestellt werden.

6.2.7. Lempel-Ziv-Markov-Chain-Huffman (LZMH)

Ausgehend vom TLZMA-Verfahren wurde das LZMH-Verfahren (s. Kap 5.7) implementiert. Es wurde statt der ASCII-Symbol-Kodierung auf eine ATH-Kodierung der Symbole gesetzt, welche nicht referenziert werden können.

Das Vorgehen zum Suchen einer Symbolfolge in dem bereits bearbeiteten Puffer ist gleich der TLZMA-Implementierung. Symbolfolgen müssen allerdings mindestens drei Symbole lang sein, um referenziert zu werden.

Wird ein Symbol nicht referenziert, wird in einem Array überprüft, ob dieses bereits ausgegeben wurde. Entsprechend der Position in dem Array wird ein Huffman-Code ausgegeben. Ist das Symbol noch nicht in der Liste oder befindet es sich an einer

Position, die nicht durch die Huffman-Codes dargestellt werden kann, wird es ASCII-kodiert ausgegeben.

Einstellungen: Da es bei diesem Verfahren vorkommen kann, dass die Ausgabe größer ist als die bisher verarbeiteten Eingabedaten und hier, wie beim TLZMA-Verfahren, die letzten 128 Byte benötigt werden, wird ein Ausgabepuffer von mindestens 129 Byte benötigt.

6.3. Eigenschaften der Verfahren

In diesem Abschnitt werden die Eigenschaften Programmgröße, Speicherverbrauch und Ausführungsgeschwindigkeit der Implementierungen miteinander verglichen. Die verschiedenen Eigenschaften werden in Tabelle 6.3 zusammengefasst.

6.3.1. Programmgröße

Der kompilierte Code der ATH-Implementierung ist nur von dem gewählten Huffman-Baum abhängig und variiert nur gering. Mit dem in Kapitel 7.2.2 entwickelten Baum ist die Bibliothek 1 494 Byte groß.

Die Programmgröße der LZW-Implementierung wird indirekt durch die Wahl der Wörterbuchgröße verändert. Je nachdem, ob das Wörterbuch größer oder kleiner als 256 Einträge gewählt wird, muss ein Ausgabepuffer implementiert werden. Damit ist der Code zwischen 1 530 und 1 742 Byte groß. Die Wahl des größeren initialisierten Wörterbuchs vergrößert den Code um jeweils 64 Byte.

Bei der SLZW-Implementierung ist die Programmgröße stark abhängig von der gewählten Wörterbuchgröße. Mit einem Wörterbuch von 32 Einträgen ist die Implementierung 1 392 Byte groß. Bei einem Wörterbuch von 4096 Einträgen beträgt die Größe 34 212 Byte. Da auf dem Stromzähler voraussichtlich 128 KByte für den Programmspeicher zur Verfügung stehen, würde diese Implementierung mehr als ein Viertel des Speichers verbrauchen und wäre damit nicht für den Einsatz geeignet. Abbildung 6.2 zeigt die Größe der Implementierung in Abhängigkeit von der Größe des Wörterbuchs.

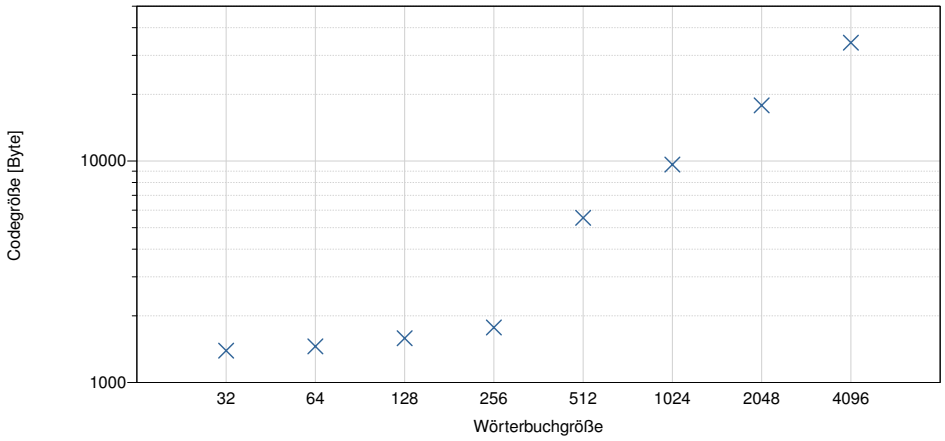


Abbildung 6.2.: Programmgröße der SLZW-Implementierung nach Wörterbuchgröße. Die Achsen sind logarithmisch.

Die Größe der Implementierung des MCH-Verfahrens hängt allein von der Wahl der Bäume ab. Mit den Bäumen, die im Kapitel 7.2.5 entwickelt wurden, ergibt sich eine Größe der Bibliothek von 7 622 Byte.

Auch bei der AMCH-Implementierung hängt die Programmgröße allein von den gewählten Bäumen ab. Mit den aus Kapitel 7.2.6 entwickelten Bäumen ergibt sich eine Programmgröße von 3 788 Byte.

Die Implementierungen des TLZMA- und des LZMH-Verfahrens wird nicht durch Parameter beeinflusst und beträgt 1 610 und 2 036 Byte respektive.

Bei der Programmgröße sind die ATH- und die LZW-Implementierung, sowie die SLZW-Implementierung mit einer Wörterbuchgröße von weniger als 128, am besten. Bis auf die SLZW-Implementierung mit mehr als 256 Wörterbucheinträgen, die MCH-Implementierung und die AMCH-Implementierung ist keine Implementierung größer als 3 KByte.

6.3.2. Speicherverbrauch

Zum Abschätzen des Speicherverbrauchs wurden die Variablen gezählt. Für Arrays wurden die Array-Größe und ein Zeiger angenommen. Da es sich bei dem Cortex-M3 um eine 32-Bit-CPU handelt, werden Zeiger mit 32 Bit gezählt. Tatsächlich ist es möglich, dass die Zeiger beim Kompilieren relational umgesetzt wurden oder sogar

weggefallen sind. Jedes Verfahren erhält einen 32-Bit-Zeiger und einen 16-Bit-Integer auf dem Stack für den Nachrichtenpuffer und die Nachrichtengröße.

Die ATH-Implementierung benötigt in jedem Fall 27 Byte für Zähler und Zeiger. Hinzu kommt die Liste der verwendeten Symbole, die 3 Byte pro Eintrag benötigt. Da nur 31 Symbole genutzt werden, sollte eine Listenlänge von 31 ausreichen. Jedoch können weitere Symbole in späteren Versionen dazu kommen, daher wurde eine Listenlänge von 48 verwendet. Damit werden insgesamt 171 Byte für das ATH-Verfahren auf dem Stack verwendet.

Die LZW-Implementierung benötigt durch Zähler und Zeiger 23 Byte. Hinzu kommen das LZW-Wörterbuch und ein eventueller interner Schreibpuffer. Der Schreibpuffer wurde auf 32 Byte gesetzt und benötigt weitere 6 Byte zur Verwaltung. Das LZW-Wörterbuch benötigt bis zu einer Größe von 256 Einträgen pro Eintrag 2 Byte, bei größeren Wörterbüchern 3 Byte. Bei einer Wörterbuchgröße von 512 werden somit 1597 Byte benötigt, bei einer Größe von 256 jedoch nur 535 Byte.

Die SLZW-Implementierung benötigt 18 Byte für Zähler und Zeiger. Bei einer Wörterbuchgröße von mehr als 256 Einträgen, kommt ein interner Schreibpuffer hinzu, der in dieser Arbeit auf 32 Byte gesetzt ist. Wird der Schreibpuffer genutzt, werden 6 zusätzliche Byte zur Verarbeitung benötigt. Außerdem wird bei Wörterbüchern mit mehr als 256 Einträgen ein weiteres Byte für die Verarbeitung der Ausgabe gebraucht. Dies ergibt eine gesamte Speicheranforderung von 18 bis 57 Byte.

Das MCH-Verfahren benötigt in dieser Implementierung 24 Byte für Zähler und Zeiger. Hinzu kommt ein interner Schreibpuffer, der in dieser Arbeit auf 32 Byte gesetzt ist. Dies ergibt eine gesamte Speicheranforderung von 56 Byte.

Bei der AMCH-Implementierung werden 37 Byte für interne Zähler und Zeiger benötigt. Außerdem wurde ein Schreibpuffergröße von 8 Byte verwendet. Den größten Einfluss haben die 32 Symbollisten mit einer Länge von jeweils 18 Einträgen und 2 Byte pro Eintrag. Das ergibt eine Anforderung von 1152 Byte für die Listen und eine Gesamtanforderung von 1197 Byte.

Das TLZMA-Verfahren benötigt 50 Byte für Zeiger und Zähler. Der Schreibpuffer muss mindestens 129 Byte betragen, da sonst das Suchfenster überschrieben wird. Insgesamt werden somit 179 Byte benötigt.

Beim LZMH-Verfahren werden 54 Byte für Zeiger und Zähler verwendet. Der Schreibpuffer muss wie beim TLZMA-Verfahren mindestens 129 Byte betragen, jedoch ist es beim LZMH-Verfahren möglich, dass die Ausgabe größer ist, als die

Wörterbuch	128	256	512	1024	2048	4096
Größe [Byte]	1430	1434	1570	1570	1570	1570
RAM [Byte]	279	535	1597	3133	6205	12349
Speicher (gesamt) [Byte]	1709	1969	3167	4703	7775	13919

Tabelle 6.2.: Speicherverbrauch des LZW-Verfahrens nach Wörterbuchgröße.

Eingabe, daher wurde ein Puffer von 192 Byte gewählt. Eine Liste vorgekommener Symbole mit 3 Byte pro Eintrag und einer Listenlänge von 48 ergibt 144 Byte. Insgesamt werden somit 390 Byte benötigt.

Es steht noch nicht fest, wie viel RAM auf dem Cortex-M3 zur Verfügung stehen werden, es wird sich voraussichtlich um 24 KByte handeln. Das implementierte Kompressionsverfahren sollte nicht mehr als 1 KByte verwenden, da durch den Übertragungsroutinen und Routing-Tabellen viel Speicher beansprucht werden. 10 KByte werden durch den unkomprimierten Datensatz beansprucht. Die Implementierung des LZW-Verfahrens mit einem Wörterbuch von mehr als 256 Einträgen benötigt damit zu viel RAM. Auch die AMCH-Implementierung kommt mit dem Bedarf von 1 197 Byte an diese Grenze.

6.3.3. Ausführungsdauer

Für den Test der Ausführungsdauer wurden die Kompressionsalgorithmen mit dem GCC-Compiler in der Version 4.6.1 für die x86-Architektur mit dem Optimierungsparameter -Os kompiliert und auf einem AMD E-350 mit 1,6 GHz und 3,5 GByte RAM ausgeführt. Jeder Datensatz wurde 512-mal komprimiert und mit Hilfe der `gettimeofday()`-Funktion für jede Ausführung die beanspruchte Zeit in Millisekunden errechnet. Aus den erhaltenen 512 Zeiten wurde der Mittelwert gebildet, wobei die minimale und die maximale Zeit verworfen wurden. Durch die Mittelwerte und die jeweilige Datensatzgröße wurde ein Quotient gebildet, der die Ausführungsdauer pro Byte angibt.

Abbildung 6.3(a) zeigt die Dauer der Komprimierung pro Byte für die verschiedenen Implementierungen. Das ATH-Verfahren arbeitet am schnellsten. Es benötigt zwischen 0,046 und 0,079 $\frac{ms}{Byte}$.

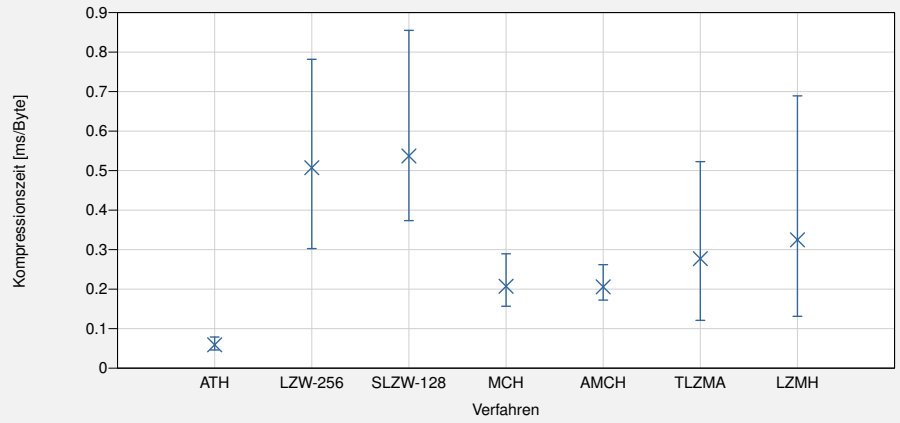
Abbildung 6.3(b) vergleicht die Geschwindigkeit der LZW- und SLZW-Implementierungen mit verschiedenen Wörterbuchgrößen. Die Größe des zu durchsu-

chenden Wörterbuches beeinflusst die Dauer des SLZW-Verfahrens enorm. Beim LZW-Verfahren hat die Größe des Wörterbuches keinen so starken Einfluss, da das Wörterbuch erst aufgebaut wird und sich erst bei größeren Dateien, bei denen das Wörterbuch voll ausgeschöpft wird, ein Unterschied zeigt.

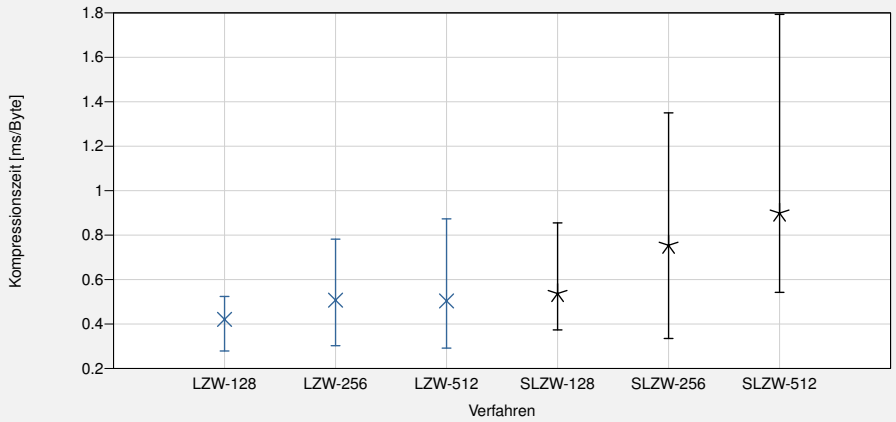
Die beiden Implementierungen des TLZMA- und LZMH-Verfahrens unterscheiden sich nur gering, da die Suche nach Referenzen in beiden Implementierungen gleich umgesetzt ist. Das LZMH-Verfahren benötigt im Mittel ca. $0,301\ \frac{ms}{Byte}$, während das TLZMA-Verfahren im Mittel ca. $0,277\ \frac{ms}{Byte}$ benötigt. Der Unterschied von $0,036\ \frac{ms}{Byte}$ ist geringer, als die ATH-Implementierung benötigt.

Verfahren:	ATH	LZW	LZW	SLZW	SLZW	MCH	AMCH	TLZMA	LZMH
Wörterbuchgröße:		**≤ 256**	**> 256**	**≤ 256**	**> 256**				
Quellcode									
Programmzeilen	126	128	128	113	113	74	129	159	254
Speicherverbrauch									
Programmgröße [Byte]	1 494	1 530	1 742	1 392 - 1 772	5 540 - 34 212	7 622	3 788	1 610	2 036
Ausgabepuffer / Größe [Byte]	–	–	✓/ 32	–	✓/ 32	✓/ 32	✓/ 8	✓/ 129	✓/ 192
RAM Anforderung [Byte]	171	279 - 535	1 597 - 12 349	18	57	56	1 197	179	390
Ausführungsdauer									
Mittelwert [ms/KByte]	59	447 - 540	452	271 - 754	899 - 2 990	202	206	277	301
Minimum [ms/KByte]	46	98 - 291	258	231 - 335	182 - 542	155	172	121	131
Maximum [ms/KByte]	79	381 - 549	778	383 - 1 350	1 793 - 10 363	285	262	523	640

Tabelle 6.3.: Implementierungseingenschaften der Verfahren



(a) Kompressionszeiten der implementierten Verfahren



(b) Geschwindigkeiten der LZW- und SLZW-Implementierungen

Abbildung 6.3.: Kompressionszeiten der einzelnen Verfahren. Die Abbildungen zeigen die minimale, die maximale und die mittlere Kompressionszeit pro Eingabebyte. Geringere Werte bedeuten eine höhere Geschwindigkeit.

Kapitel 7

Auswertung

In diesem Kapitel wird der Einfluss der Parameter auf die Kompressionsleistung der angepassten Verfahren betrachtet. Die besten Ergebnisse der Verfahren, welche für die Smart Meter-Daten, die dieser Arbeit zur Verfügung standen, realisierbar sind, werden miteinander verglichen. Als Referenz dient das bisher benutzte LZSS-Verfahren und die in Kapitel 3 beschriebenen Verfahren GZIP, BZIP2 und LZMA.

7.1. Metriken und Methodik

In dieser Arbeit wird die Kompressionsleistung zur Darstellung der Ergebnisse verwendet. Wenn die Größe der Eingabedaten S_I und die Größe der Ausgabedaten S_O ist, dann wird die Kompressionsleistung definiert als

$$K_L = 1 - \frac{S_O}{S_I}. \tag{7.1}$$

Eine Kompressionsleistung von 60% bedeutet, dass die Ausgabedaten 40% der Eingabedatengröße besitzen. Die zu übertragende Datenmenge wird also um 60% reduziert. Die Verfahren wurden, soweit nicht anders beschrieben, auf alle Datensätze angewandt.

Die Kompressionsleistungen werden in Boxplots dargestellt, bei denen die Box die Ergebnisse zwischen dem oberen und dem unteren Quartil darstellt. Der Strich in

der Mitte zeigt den Median. Die Linien oberhalb und unterhalb der Box zeigen die minimalen und maximalen Ergebnisse.

7.2. Ergebnisse

In diesem Abschnitt werden die Einflüsse der Parameter auf die einzelnen Verfahren betrachtet.

7.2.1. GZIP, BZIP2, LZMA und LZSS

Die Verfahren wurden ohne Anpassung der Parameter über alle Daten ausgeführt. Bei den Verfahren GZIP, BZIP2 und LZMA kann es vorkommen, dass das Ergebnis größer wird, als die Eingabedaten. Das passiert bei kleinen Datensätzen, da diese Verfahren ihre Ausgabe in einem Containerformat ausgeben, welches Informationen zum Dekomprimieren enthält. Das LZSS-Verfahren hat kein solches Containerformat, sondern gibt seine Ergebnisse direkt aus.

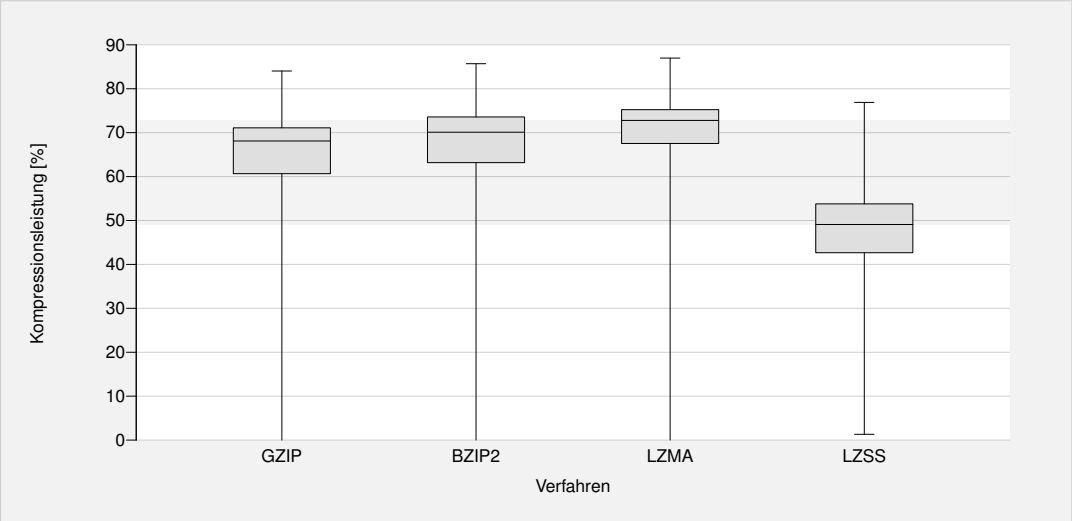


Abbildung 7.1.: Ergebnisse der Verfahren GZIP, BZIP2, LZMA und LZSS. Der Bereich des Median der LZSS-Variante und des LZMAVerfahrens wurde durch eine graue Fläche gekennzeichnet.

Abbildung 7.1 zeigt die Ergebnisse dieser Verfahren. Das LZMA-Verfahren erreicht mit einem Median von 72, 8% die beste Kompressionsleistung. Die Kompressionsleistung vom BZIP- und GZIP-Verfahren beträgt im Mittel 70, 1%, bzw. 68, 1%. Im

schlechtesten Fall liegen die Kompressionsraten vom GZIP, BZIP und LZMA bei $-63,2\%$, $-23,8\%$ und $-2,6\%$. Trotz einer maximalen Kompressionsleistung von $76,9\%$, erreicht die LZSS-Variante im Mittel nur $49,1\%$.

7.2.2. Adaptive-Trimmed-Huffman (ATH)

Der einzige Parameter für das ATH-Verfahren ist die Wahl des Baums. Es wurden Bäume auf Grundlage der errechneten idealen Codelängen (s. Kap 5.1) erstellt. Da diese idealen Codelängen durch das Präfix-Bit nicht erreicht werden können – ein ideales Codewort der Länge 2 Bit wird 3 Bit lang –, wurden verschiedene Varianten der Bäume erstellt. Abbildung 7.2 zeigt die verschiedenen Baum-Varianten, die erstellt und getestet wurden.

CL [Bit]	1	2	3	4	5	6	7
N	0	0	1	2	4	3	10

(a) Baum A (Optimale Huffman-Code-Längen plus 1 Bit)

CL [Bit]	1	2	3	4	5	6	7
N	0	0	1	2	4	8	0

(b) Baum B (Ab der Codelänge 6 wird die optimale Codelänge erreicht)

CL [Bit]	1	2	3	4	5	6	7
N	0	0	1	2	7	2	0

(c) Baum C (Ab der Codelänge 5 wird die optimale Codelänge erreicht)

CL [Bit]	1	2	3	4	5	6	7
N	0	0	1	6	0	0	0

(d) Baum D (Ab der Codelänge 4 wird die optimale Codelänge erreicht)

CL [Bit]	1	2	3	4	5	6	7
N	0	0	3	2	0	0	0

(e) Baum E (Ab der Codelänge 3 wird die optimale Codelänge erreicht)

CL [Bit]	1	2	3	4	5	6	7
N	0	1	2	0	0	0	0

(f) Baum F (Ab der Codelänge 2 wird die optimale Codelänge erreicht)

CL [Bit]	1	2	3	4	5	6	7
N	0	1	0	2	4	0	0

(g) Baum G (Der kürzeste Code wird erhalten, alle weiteren Codes werden um ein Bit erhöht)

Abbildung 7.2.: Codelängen-Verteilung der Baumvarianten (CL - Codelänge, N - Anzahl der Codes)

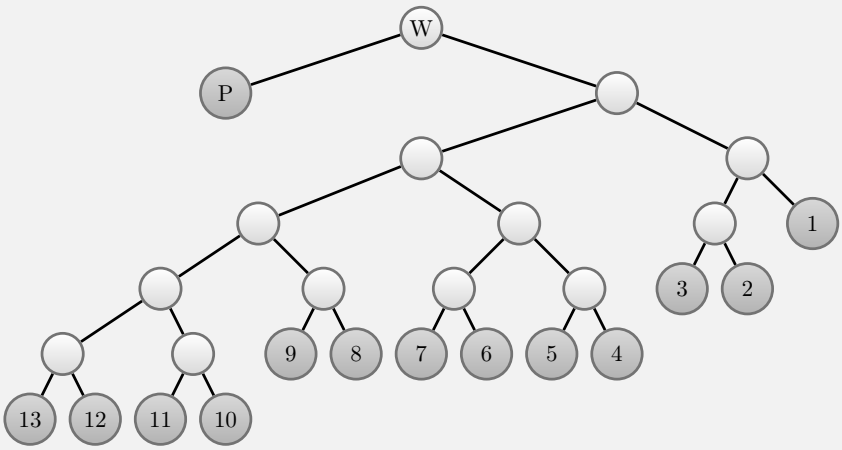
Bei den Tests der Kompressionsleistungen zeigte sich, dass Baum B und Baum C ähnlich gute Ergebnisse erzielten (s. Abbildung 7.4), daher wurden zwei Varianten der Bäume erstellt (s. Abbildung 7.3) und ebenfalls getestet. Während bei Baum B ab einer Codewortlänge von 6 Bit die Codewörter wieder ihre optimale Codewortlänge erreichen, wird bei Baum C bereits ab einer Codewortlänge von 5 Bit die optimale Codewortlänge erreicht. Bei den Bäume BC1 und BC2 wird die optimale Codewortlänge für die ersten zwei Symbole mit der optimalen Codewortlänge 5, bzw. für das ersten Symbol dieser Länge, erreicht.

CL	1	2	3	4	5	6	7
N	0	0	1	2	6	4	0

(a) Baum BC1

CL [Bit]	1	2	3	4	5	6	7
N [Bit]	0	0	1	2	5	6	0

(b) Baum BC2



(c) Grafische Darstellung des Baumes BC1. Das Symbol P ist der Präfix für nicht kodierbare Symbole, W ist der Wurzelknoten. Die Pfadlänge von der Wurzel zu den Blättern entspricht den benötigten Bits.

Abbildung 7.3.: Codelängen-Verteilung Baumvarianten zwischen den Bäumen B und C. (CL - Codelänge, N - Anzahl der Codes)

Die Bäume A, B und C haben nur eine geringe Streuung der Kompressionsleistungen und ihr Median liegt bei jeweils 41,8%, 42,3% und 42,5%. Der Baum F erreicht mit einem Median von 32,9% die schlechtesten Ergebnisse. Die maximale Kompressionsleistung des Baumes G ist die beste und erreicht 53,5%. Im Mittel ist die

Kompressionsleistung mit 41,3% jedoch etwas geringer, als bei Baum A. Die Bäume BC1 und BC2 erreichen die gleiche maximale und mittlere Kompressionsleistung. Jedoch ist die minimale Kompressionsleistung vom Baum BC1 mit 30,3% die beste. Der Median der Kompressionsleistung liegt bei 42,6%.

Mit der Baum Variante BC1 lassen sich im Mittel die besten Ergebnisse erzielen. Trotzdem kommen die Ergebnisse mit einer Kompressionsleistung von maximal 49,1% nicht an die Leistung einer optimalen Entropie-Kodierung von 55% heran. Weiterhin sind die Ergebnisse im Mittel schlechter als die bisher verwendete LZSS-Variante.

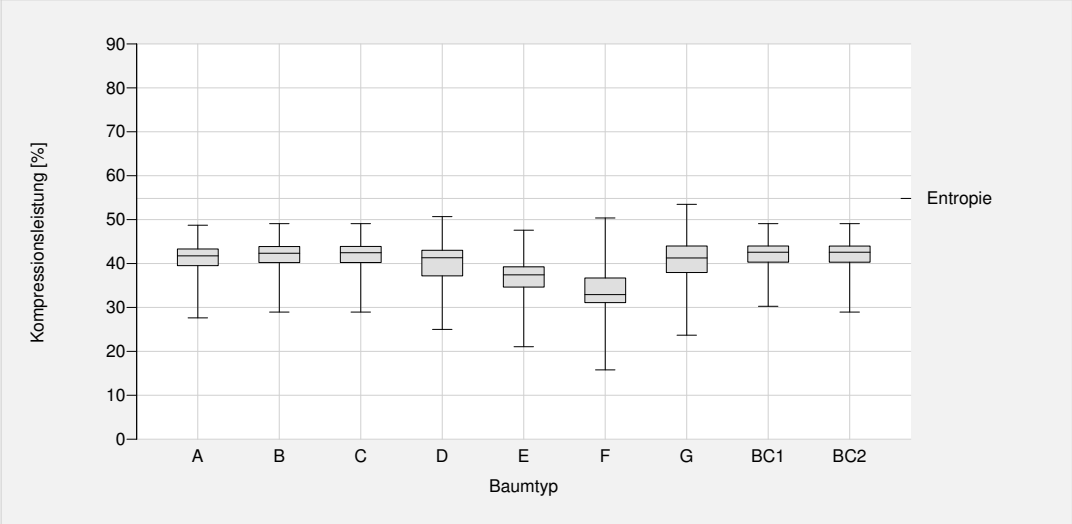


Abbildung 7.4.: Analyse der Kompressionsleistung der ATH-Komprimierung mit verschiedenen gegebenen Bäumen (Abbildung 7.2 und 7.3). Die Entropie zeigt die maximal im Mittel erreichbare Kompressionsleistung.

7.2.3. Lempel-Ziv-Welch

Das LZW-Verfahren wurde mit den Wörterbuchlängen von 128, 258, 512, 1024, 2048 und 4096 Einträgen getestet. Es wurde ein initialisiertes Wörterbuch von 31 und 98 Symbolen verglichen (s. Kap 6.2.2).

Abbildung 7.5 zeigt die Ergebnisse der verschiedenen Varianten. Ab einer Wörterbuchgröße von 512 Einträgen gibt es keine Veränderungen der Kompressionsleistungen, da die Datensätze nicht groß genug sind, um das Wörterbuch voll auszuschöpfen.

Die minimale Kompressionsrate liegt für 31 initialisierte Symbole konstant bei 30,3%. Kleine Datensätze können bereits das Wörterbuch mit maximal 128 Einträgen nicht komplett ausnutzen. Bei 98 initialisierten Symbolen, ist das Wörterbuch von 128 Einträgen offenbar zu klein, um alle möglichen Symbolfolgen eintragen zu können und das Verfahren erreicht minimal nur 16,9%, während größere Wörterbücher mindestens 18,4% erreichen.

Für Wörterbücher mit 128, 256 und 512 maximalen Einträgen bei 31 initialisierten Symbolen beträgt die Kompressionsrate im Mittel 55,9%, 60,0% und 64,1%. Es ist zu bedenken, dass ein Wörterbuch der Länge 512 einen RAM-Verbrauch von 1536 Byte hat und daher wahrscheinlich nicht realisierbar ist. Ein Wörterbuch der Länge 256 benötigt 512 Byte an RAM. Für eine Verbesserung der Kompressionsleistung von weniger als 5% ist eine Erhöhung der RAM-Anforderung um 300% nicht angemessen.

Da die möglichen Symbole bereits in dem Wörterbuch initialisiert werden müssen, wird das Verfahren mit 31 initialisierten Symbolen bei einer Erweiterung des Ausgabeformats um weitere Symbole scheitern. Es müsste auf die geänderten Symbole angepasst werden, da die neuen Symbole nicht kodiert werden können. Das bedeutet, dass eine neue Firmware auf alle Smart Meter installiert werden müsste. Ein LZ78-Verfahren würde dieses Problem umgehen, konnte in dieser Arbeit jedoch aus zeitlichen Gründen nicht weiter verfolgt werden.

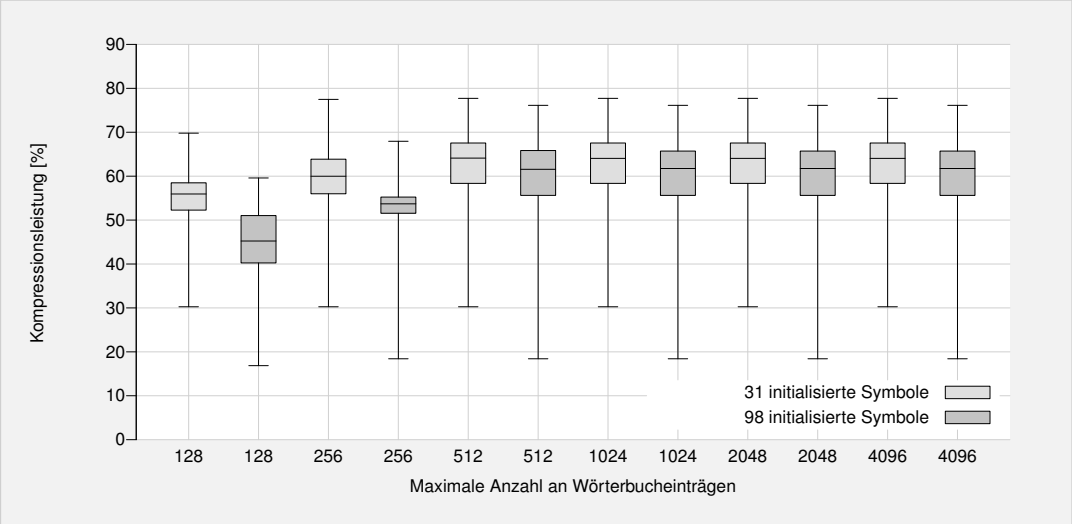


Abbildung 7.5.: Analyse der Kompressionsleistung der LZW-Komprimierung mit verschiedener Anzahl an maximalen Wörterbucheinträgen und 31, bzw. 98 initialisierten Symbolen

7.2.4. Lempel-Ziv-Welch mit statischem Wörterbuch (SLZW)

Die häufigsten LZW-Wörterbucheinträge wurden nach der Häufigkeit geordnet und als SLZW-Wörterbuch verwendet (s. Kap 5.3). Abbildung 7.6 zeigt die Ergebnisse des Verfahrens für Wörterbücher von 32 bis 4096 Einträgen. Eine Vergrößerung des Wörterbuches führt nicht immer zu signifikanten Verbesserungen der Ergebnisse. Dies ist mit den längeren Codewörtern begründbar, die für ein größeres Wörterbuch benötigt werden.

Bereits die Variante mit einem Wörterbuch von 32 Einträgen erreicht mit einem Mittleren Kompressionsgrad von 44, 1% bessere Ergebnisse als das ATH-Verfahren. Bei einem Wörterbuch von 64 Einträgen nimmt die Streuung zu und der minimale Kompressionsgrad verschlechtert sich. Im Mittel verbessert sich die Kompressionsleistung auf 57, 2%. Mit größeren Wörterbüchern rücken das obere und das untere Quartil näher zusammen. Der Median der Kompressionsleistung für Wörterbücher von 128, 256 und 512 Einträgen beträgt 61, 2%, 66, 5% und 70, 2% respektive.

Ein Wörterbuch der Länge 128 erreicht somit ähnliche Ergebnisse, wie das LZW-Verfahren mit einer Wörterbuchlänge von 256, benötigt dabei jedoch nur 18 Byte RAM bei einem ca. 150 Byte größeren Programmcode (s. Tabelle 6.3).

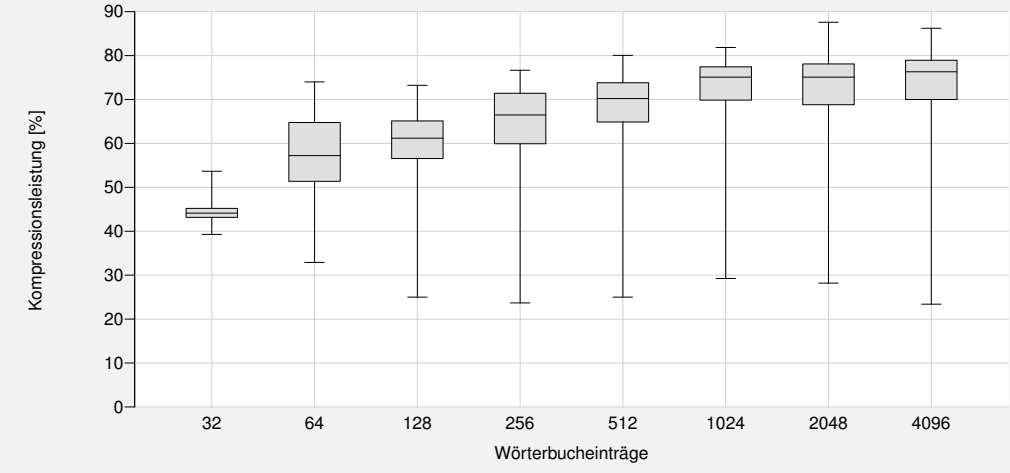


Abbildung 7.6.: Analyse der Kompressionsleistung der SLZW-Komprimierung mit verschiedenen Wörterbuchlängen

Eingeschränkt ist das Verfahren durch das gegebene Wörterbuch. Im Falle einer Erweiterung des Ausgabeformats durch andere Symbole, müsste das Verfahren angepasst

werden. Es wäre somit ein Firmware Update nötig.

7.2.5. Markov-Chain-Huffman (MCH)

Beim MCH-Verfahren ist es wichtig zu wissen, wie sich die Wahl der Bäume auf die Kompressionsleistung auswirkt. Außerdem gilt es zu erfahren, wie sich die Leistung verändert, wenn sich die Daten bei späteren Übertragungen stark von den Daten, die dieser Arbeit zur Verfügung standen, unterscheiden.

Abbildung 7.7 zeigt, dass die Kodierung von nicht berücksichtigten Symbolen durch ein Symbol mit Wahrscheinlichkeit 0 die besten Ergebnisse liefert, sofern der Huffman-Baum nicht begrenzt wurde. Der Median der Kompressionsleistung liegt bei $69,9\%$ und unterschiedet sich nur gering von dem Median bei einer Kodierung mit einer Mittelwertkodierung, welche $69,2$ erreicht. Die Kodierung mit einem Präfix-Bit erreicht im Mittel eine Kompressionsrate von $58,9\%$.

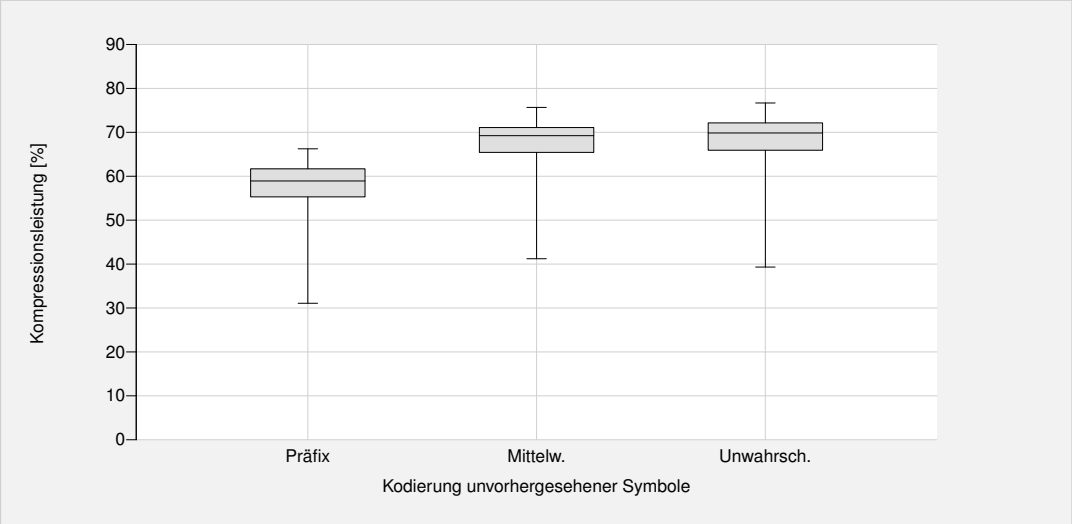
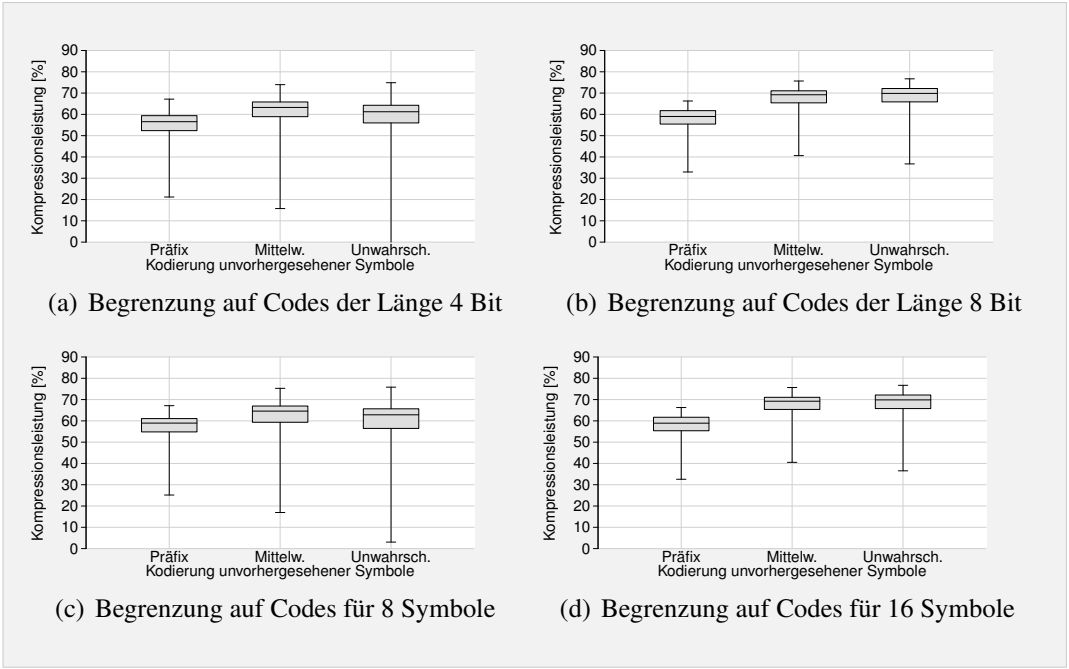


Abbildung 7.7.: Ohne Beschränkung der Huffman-Codes

Um zu erfahren, wie sich die Verfahren verhalten, wenn unberücksichtigte Symbole vorkommen, wurden die berücksichtigten Symbole in den Bäumen begrenzt – ohne Begrenzung kann ein Symbol bis zu 23 Folgesymbole haben und die Codes bis zu 14 Bit lang sein. Abbildung 7.8 zeigt die Kompressionsraten für verschiedene Begrenzungen der Symbole. Bei einer Begrenzung des Huffman-Baumes auf höchstens 8 Symbole oder auf eine Codewortlänge von 4 erreicht die Kodierung von unberück-

sichtigten Symbolen durch ein Symbol mittlerer Wahrscheinlichkeit leicht bessere Ergebnisse als die Kodierung durch ein Symbol der Wahrscheinlichkeit 0. Bei einer Beschränkung auf Codewörter von 4 Bit erreicht die Mittelwertkodierung im Median eine Kompressionsleistung von 63,2% und die unwahrscheinliche Kodierung 61,2%. Da der minimale Kompressionsgrad der Mittelwertkodierung immer am besten ist, wird diese Kodierung bevorzugt.



(a) Begrenzung auf Codes der Länge 4 Bit

(b) Begrenzung auf Codes der Länge 8 Bit

(c) Begrenzung auf Codes für 8 Symbole

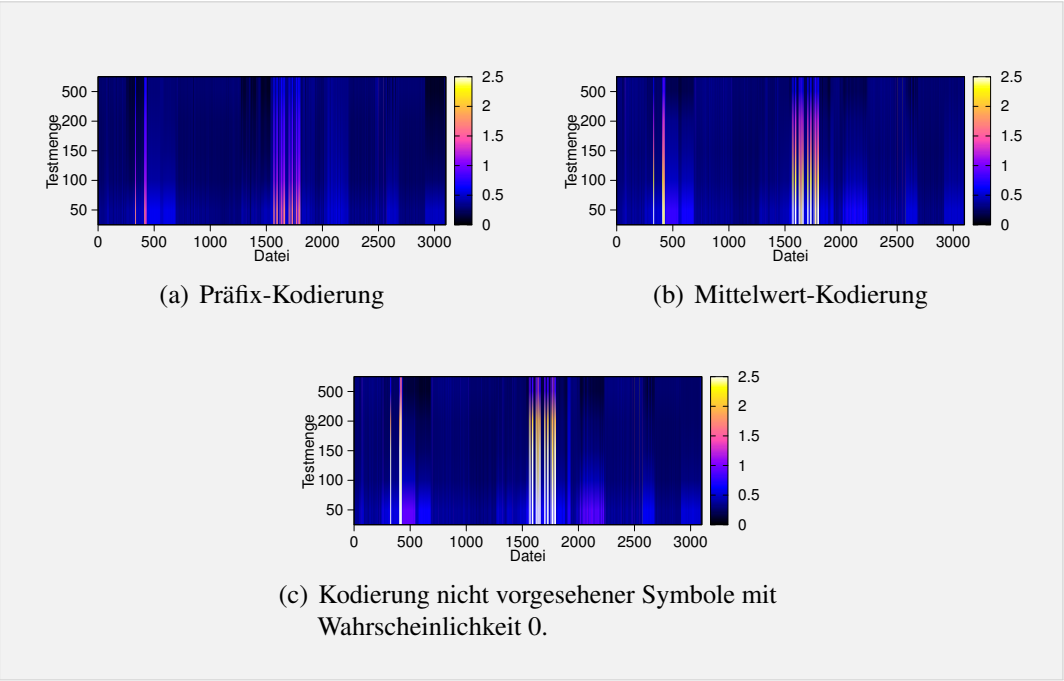
(d) Begrenzung auf Codes für 16 Symbole

Abbildung 7.8.: Kompressionsleistungen des MCH-Verfahrens mit einer zufälligen Auswahl an 500 Datensätzen für die Erstellung der Huffman-Codes.

Wird die Standardabweichung der verschiedenen Kodierungen betrachtet (Abbildung 7.9), zeigt sich, dass die beiden Varianten mit den besten Ergebnissen besonders stark von der Lernmenge abhängen.

7.2.6. Adaptive Markov-Chain-Huffman (AMCH)

Jede Variante aus Kapitel 7.2.6 wurde auf den kompletten Datensatz angewandt, wobei die Anzahl der Bäume beschränkt wurde. Für das AMCH-Verfahren wird nicht wie beim ATH-Verfahren ein einziger Baum verwendet, sondern je nach Anzahl der jeweiligen Folgesymbole ein Baum ausgewählt. Weiterhin ist kein Präfix-Bit für die Kodierung von noch nicht vorgekommenen Symbolen verantwortlich, sondern ein in



(a) Präfix-Kodierung

(b) Mittelwert-Kodierung

(c) Kodierung nicht vorgesehener Symbole mit Wahrscheinlichkeit 0.

Abbildung 7.9.: Standardabweichung der Kompressionsleistung der Kodierungen bei unterschiedliche großen Lernmengen

jedem Baum eingebrachtes Symbol. Erste vorab durchgeführte Tests haben hierfür bessere Ergebnisse gezeigt, was sich auch durch die Ergebnisse des MCH-Verfahrens bestätigen lässt. Varianten mit einer multiplizierten Wahrscheinlichkeit des Symbols für noch nicht vorgekommene Symbole kommen der Kodierung mit einem Präfix-Bit nahe. Das Verfahren wurde mit allen 18, sowie mit 14 und 10 Huffman-Bäumen ausgeführt.

Abbildung 7.10 zeigt die Kompressionsleistungen der verschiedenen Varianten. Bei den Varianten mit unmodifizierten Wahrscheinlichkeiten zeigen sich nur geringe Unterschiede beim Median der Kompressionsleistung. Das Verfahren mit 10, 14 und 18 Bäumen erreicht jeweils eine mittlere Kompressionsleistung von 62, 8%, 63, 4% und 63, 4%. Bei den modifizierten Varianten mit 14 Bäumen erreicht die Versionen mit dem Multiplikator 8 62, 9%, mit dem Multiplikator 16 60, 1% und mit dem Multiplikator 32 58, 6% im Median.

Die Variante mit den unmodifizierten Symbolen für nicht Treffer mit 14 Bäumen im Mittel liefert die besten Ergebnisse. Die schlechten minimalen Kompressionsleistungswerte werden nur bei kleinen Datensätzen erreicht, bei Datensätzen von über

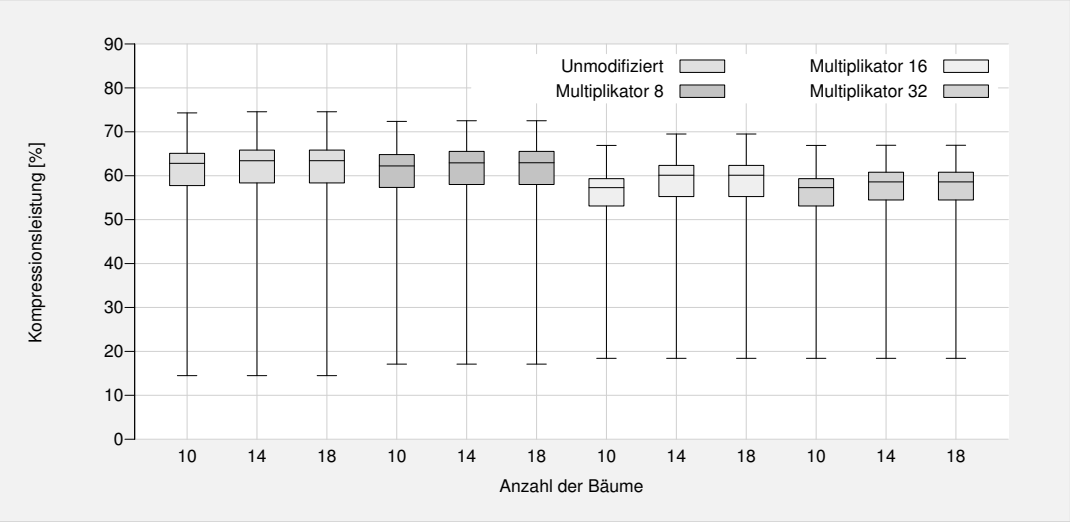


Abbildung 7.10.: Kompressionsleistung des AMCH-Verfahrens mit verschiedenen Huffman-Bäumen

1 000 Byte liegt die Kompressionsleistung meist bei über 50%.

7.2.7. Tiny Lempel-Ziv-Markov-Chain-Algorithm (TLZMA)

Das TLZMA-Verfahren hat keine Parameter, welche die Kompressionsleistung beeinflussen. Die minimale Kompressionsleistung für die gegebenen Daten beträgt 18.4%, die maximale Kompressionsleistung liegt bei 85, 9%. Der Median der Kompressionsleistung des Verfahrens liegt bei 68, 3%. Die minimale Kompressionsleistung wird nur bei kleinen Datensätzen erreicht. Das obere und untere Quartil liegt bei 70, 4% und 64, 2%.

7.2.8. Lempel-Ziv-Markov-Chain-Huffman (LZMH)

Die Kompressionsleistung des LZMH-Verfahrens wird durch keine Parameter beeinflusst. Mit einer Kompressionsleistung im Median von 72, 0% erreicht das Verfahren ähnlich gute Werte, wie das LZMA-Verfahren. Die maximale und minimale Kompressionsrate, sowie das untere Quartil liegen mit 31, 5%, 87, 1% und 69, 1% sogar höher, als beim LZMA-Verfahren.

7.3. Vergleich und Diskussion

Die besten realisierbaren Ergebnisse der angepassten Verfahren werden in Abbildung 7.11 gezeigt. Als realisierbar wurden Arbeitsspeicheransprüche von maximal 1 KByte und eine maximale Programmgröße von 3 KByte angesehen. Das ATH-Verfahren liefert von den angepassten Verfahren die schlechtesten Ergebnisse und der Median der Kompressionsleistung befindet sich als einziger nicht in dem grauen Bereich, der den Median der LZSS-Variante und des LZMA-Verfahrens anzeigt. Für das Verfahren sprechen jedoch die geringe Codegröße, der geringe Speicherverbrauch, die hohe Geschwindigkeit und die geringe Streuung der Kompressionsleistung.

Beim LZW- und SLZW-Verfahren wurden die Varianten mit einer Wörterbuchgröße von 256 Einträgen betrachtet. Dabei liegt der Median des SLZW-Verfahrens um 6% höher, als beim LZW-Verfahren.

Die Verfahren MCH, TLZMA und LZMH liefern die besten Ergebnisse, wobei das LZMH-Verfahren sogar die Kompressionsleistung des LZMA-Verfahrens erreicht. Da das TLZMA- und das LZMH-Verfahren unabhängig von der Datenstruktur sind und auf alle ASCII-Kodierten Daten angewendet werden können, sind diese Verfahren dem MCH-Verfahren vorzuziehen, welches bei einer Änderung des Standards keine guten Ergebnisse liefern kann. Bereits bei den Datensätzen in denen sich nur Prüfsummen befinden, bei denen eine Entropie-Kodierung keine guten Ergebnisse liefern kann, zeigt sich, dass das LZMH- und das TLZMA-Verfahren diese Daten besser komprimieren kann. Bei einem Datensatz von 2 690 Byte in dem sich nur Prüfsummen befinden, erreicht das MCH-Verfahren eine Kompressionsleistung von 45%, während das TLZMA-Verfahren 71% und das LZMH-Verfahren 76% erreichen.

In Abbildung 7.12 werden die Kompressionsleistungen und die resultierende Datensatzgröße für das ATH-, das MCH- das LZMH-Verfahren auf den einzelnen Datensätzen dargestellt. Für Datensätze von mehr als 1 000 Byte werden beim LZMH-Verfahren nur selten Kompressionsleistungen von weniger als 70% erreicht. In einigen Fällen reicht erreicht die Kompressionsleistung sogar mehr als 85%. Bei größeren Daten ist zu erwarten, dass sich diese guten Ergebnisse mehren. Es kann daher möglich sein, dass Daten von 10 KByte auf 1,5 KByte Komprimiert werden. Insgesamt ist die Kompressionsleistung des MCH-Verfahrens stabiler über alle Dateigrößen, erreicht aber geringere maximale Kompressionsleistungen.

Da die in dieser Arbeit betrachteten Datensätze aus einem realen Testbetrieb der

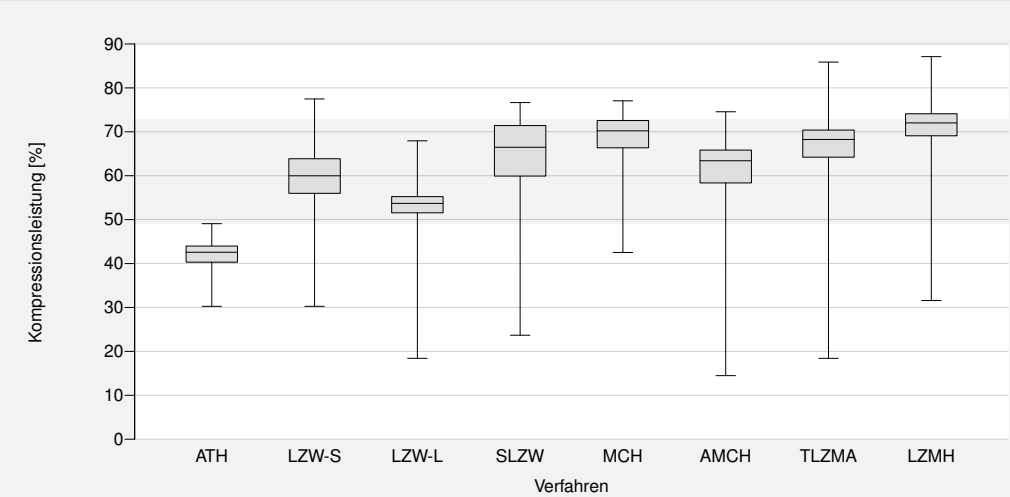


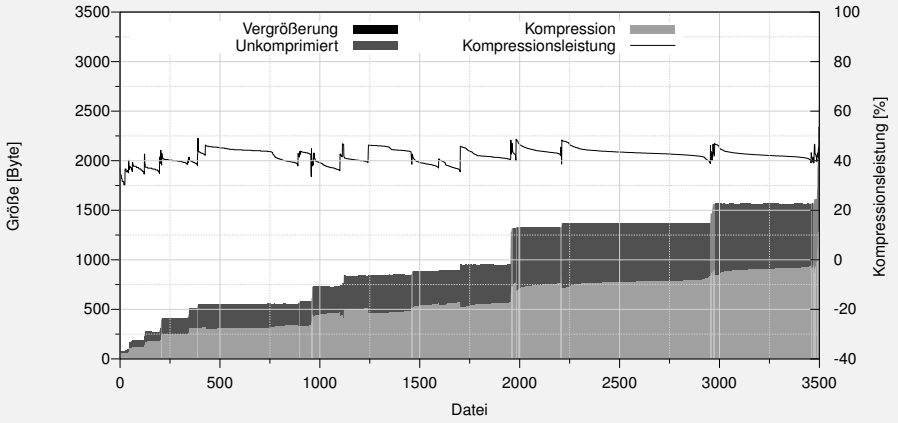
Abbildung 7.11.: Boxplots der angepassten Verfahren. Der Bereich des Medians vom LZSS-Verfahren bis zum LZMA-Verfahren ist durch eine graue Fläche gekennzeichnet.

Stromzähler stammen, wird die Verteilung der Datensatzgrößen sich nicht stark ändern. Wird die Größe aller Datensätze nach Kompression betrachtet (Abbildung 7.13), erreicht auch hier das LZMH-Verfahren mit einer Gesamtgröße nach dem Komprimieren von 970 KByte die besten Ergebnisse. Das MCH-Verfahren ist mit 1 042 KByte zwar nur geringfügig schlechter, allerdings bleibt zu beachten, dass dieses Verfahren statisch auf die Daten angepasst wurde. Zwar behebt das AMCH-Verfahren dieses Problem des MCH-Verfahrens, verliert dadurch aber deutlich an Kompressionsleistung und hat hohe RAM-Anforderungen.

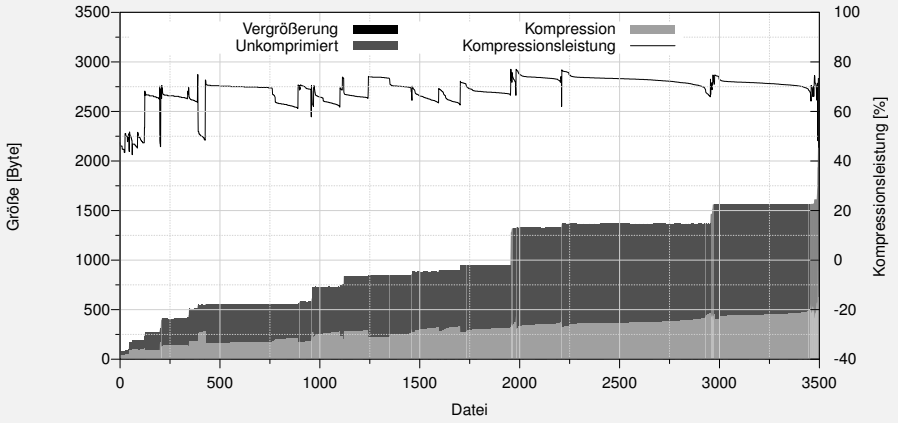
Für einen Einsatz auf Stromzählern ist das LZMH-Verfahren die beste Kombination aus sehr guter Kompressionsleistung, geringer Programmgröße und geringen Speicherverbrauch.

7.4. Einschränkungen

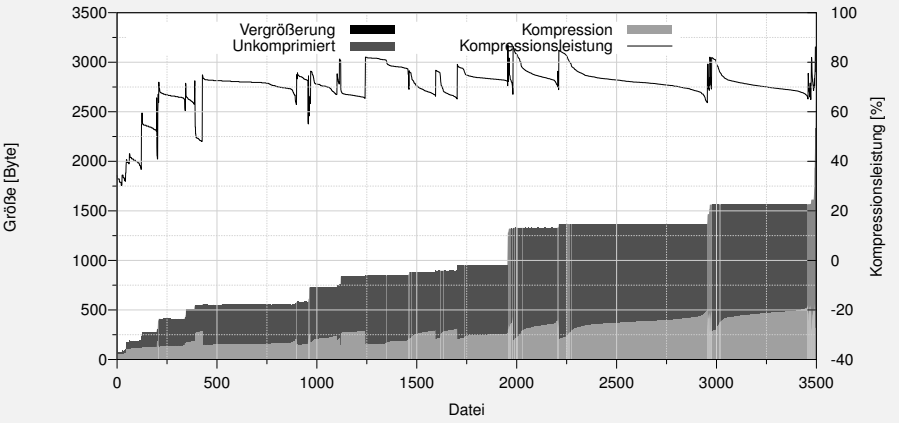
Die verwendeten Verfahren sind auf das Datenformat der Smart Meter-Daten hin optimiert. Bei einer grundlegenden Änderung des Ausgabeformats der Daten, sodass diese nicht mehr ASCII-Kodiert ausgegeben werden, müssten die ATH-, MCH-, AMCH-, TLZMA- und LZMH-Verfahren angepasst werden. Dabei wird das MCH-



(a) Datensätze vor und nach ihrer Kompression mit dem ATH-Verfahren



(b) Datensätze vor und nach ihrer Kompression mit dem MCH-Verfahren



(c) Datensätze vor und nach ihrer Kompression mit dem LZMH-Verfahren

Abbildung 7.12.: Kompressionsleistung und die resultierenden Datensatzgrößen

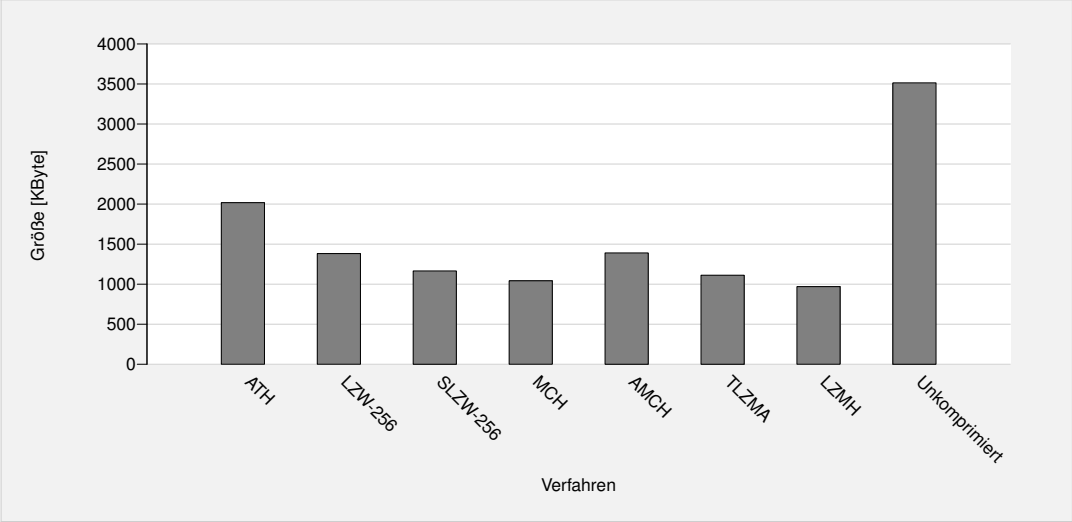


Abbildung 7.13.: Größe aller Datensätze nach der Kompression

Verfahren bereits bei der Einführung eines neuen Symbols nicht mehr die gleichen Ergebnisse liefern können. Die LZW-, SLZW-Verfahren würden, bereits bei einer Erweiterung des Formats um ein neues Symbol, eine Anpassung benötigen.

Kapitel 8

Zusammenfassung und Ausblick

Smart Meter werden wegen der vermehrten Nutzung regenerativer Energiequellen und für die Bildung eines intelligenten Stromnetzes eine wichtige Rolle spielen. Ein Aspekt der Smart Meter ist es, mindestens ein Mal am Tag ein Lastprofil von bis zu 10 KByte an den Stromanbieter zu senden. Damit für die Datenübermittlung keine Verträge mit den Kunden geschlossen werden müssen, bietet sich die Übertragung per Funk an. Da Stromzähler jedoch oft im Hauskeller verbaut sind, ist eine direkte Verbindung der Stromzählers mit einer Zentrale nicht möglich. Jeder Stromzähler muss seine Messdaten an den Nächsten bis zu einer Datensenke weiterleiten. Diese sendet die Informationen an den Stromanbieter. Bei großen Netzen häufen sich Kollisionen, sodass sich ein Flaschenhals zu der Datensenke bildet. Eine Kompression der Daten kann diese Probleme vermindern.

In dieser Arbeit wurden Methoden der Datenkompression für Smart Meter analysiert. Heutige Kompressionsverfahren auf Desktop-Computern kombinieren mehrere dieser Methoden, um gute Kompressionsergebnisse zu erreichen. Diese Verfahren – wie der DEFLATE-Algorithmus, BZIP2 und der Lempel-Ziv-Markov-Chain-Algorithmus (LZMA) – benötigen jedoch sehr viel Arbeitsspeicher (RAM). Sensorknoten bieten nur wenig RAM, um den Eigenverbrauch und die Produktkosten gering zu halten. Auch der Programmspeicher ist mit 128 KByte sehr gering, sodass diese Verfahren mit 70 bis 135 KByte einen großen Teil des Speichers verbrauchen würden. Mit dem Adaptive-Trimmed-Huffman-Verfahren (ATH) und einer LZSS-Variante, die bisher auf Smart Meter verwendet wird, gibt es zwei Verfahren, die mit weniger als

3 KByte Programmgröße auf diese geringen Ressourcen hin optimiert sind. Allerdings bieten diese Verfahren nur eine geringe Kompressionsleistung.

In dieser Arbeit wurden daher Verfahren entwickelt, welche bessere Kompressionsleistungen bieten und auf den begrenzten Hardwareressourcen der Smart Meter lauffähig sind. Durch eine Analyse realer Stromzählerdaten aus einem Testnetzwerk eines Smart Meter-Herstellers wurden optimale Kompressionsraten, sowie Datenspezifika zur Gewinnung neuer Kompressionsverfahren identifiziert. Es zeigte sich eine große Redundanz der Daten durch eine ungünstige Kodierung und viele Symbole zur Datenstrukturierung. Mit einer einfachen Entropiekodierung könnten die Daten im Mittel mit einer Kompressionsleistung von 56% komprimiert werden. Eine Entropiekodierung, welche die Übergänge der Symbole berücksichtigt, könnte eine Kompressionsrate von bis zu 73% erreichen.

Auf Grundlage dieser Erkenntnisse wurde das ATH-Verfahren durch die Suche nach einem für die Daten geeigneten Baum verbessert. Es wurde ein Lempel-Ziv-Welch-Verfahren (LZW) und eine LZW-Variante mit statischem Wörterbuch (SLZW) implementiert. Die in der Analyse beobachteten Übergangswahrschlichkeiten der Symbole wurden in dem Markov-Chain-Huffman-Verfahren ausgenutzt. Auf dieser Grundlage wurde auch eine adaptive Variante des MCH-Verfahrens (AMCH) entwickelt, welche unabhängig von den verwendeten Symbolen arbeitet. Schließlich wurde versucht, die Vorgehensweise des LZMA-Verfahrens auf die begrenzten Ressourcen der Smart Meter anzupassen.

Das ATH-Verfahren kann durch die Anpassung einen mittleren Kompressionsgrad von fast 43% erreichen. Bei einer Wörterbuchlänge von 256 Einträgen erreicht das LZW-Verfahren im Mittel einen Kompressionsgrad von ca. 60%. Größere Wörterbücher würden unverhältnismäßig viel Arbeitsspeicher benötigen. Das SLZW-Verfahren benötigt sehr wenig Arbeitsspeicher, allerdings wird das Programm ab einem Wörterbuch von 512 Einträgen sehr groß. Mit einem Wörterbuch der Länge 256 erreicht dieses Verfahren einen mittleren Kompressionsgrad von 66%. Das speziell auf die Daten angepasste MCH-Verfahren erreicht gute Kompressionsleistungen von 70% im Median. Dieses müsste bei einer Erweiterung oder Änderung des Formats jedoch angepasst werden. Das AMCH-Verfahren behebt dieses Problem, erfordert dafür jedoch mehr Arbeitsspeicher und erreicht mit einer Kompressionsleistung von 60% lediglich die Leistungen des LZW-Verfahrens. Die Vereinfachung des LZMA-Verfahrens TLZMA erreicht im Mittel eine Kompressionsleistung von 68% und ist dabei

unabhängig vom Format der Eingabedaten. Mit dem LZMH-Verfahren konnte die Kompressionsleistung noch auf 72% gesteigert werden. Damit erreicht das LZMH-Verfahren fast die Leistung des LZMA-Verfahrens, das im Mittel eine Leistung von 73% erreicht.

Das LZMH-Verfahren bietet damit von den angepassten Verfahren die beste Kompressionsleistung bei geringen Anforderungen an den Arbeits- und Programmspeicher. Zusätzlich ist das Verfahren unabhängig vom Format der Stromzählerdaten. Es eignet sich daher ideal für die Verwendung auf Smart Metern.

Sollten die Stromzählerdaten in Zukunft nicht mehr ASCII-kodiert ausgegeben werden, stößt das Verfahren in der jetzigen Form an seine Grenzen und müsste angepasst werden. Weiterhin kann das Verfahren nicht garantieren, dass die Ausgabedaten kleiner sind als die Eingabedaten. In den Tests kam ein solcher Fall, bei dem die Daten vergrößert wurden, jedoch nicht vor.

Literaturverzeichnis

[ACG07] ALIPPI, C. ; CAMPLANI, R. ; GALPERTI, C.: Lossless Compression Techniques in Wireless Sensor Networks: Monitoring Microacoustic Emissions. In: *Proceedings of the International Workshop on Robotic and Sensors Environments (ROSE '07)*. Ottawa, Canada, Oktober 2007

[BSTW86] BENTLEY, J. L. ; SLEATOR, D. D. ; TARJAN, R. E. ; WEI, V. K.: A Locally Adaptive Data Compression Scheme. In: *Communications of the ACM* 29 No.4 (1986), S. 320–330

[BW94] BURROWS, M. ; WHEELER, D. J.: A Block-sorting Lossless Data Compression Algorithm / Systems Research Center. Palo Alto, CA, USA, 1994. – Forschungsbericht

[Fan49] FANO, R. M.: The Transmission of Information / Research Laboratory of Electronics at MIT. Cambridge, MA, USA, 1949. – Forschungsbericht

[Huf52] HUFFMAN, D. A.: A Method for the Construction of Minimum-Redundancy Codes. In: *Proceedings of the Institute of Radio Engineers (IRE '52)* 40 (1952), September, Nr. 9, S. 1098 –1101

[MV09] MARCELLONI, F. ; VECCHIO, M.: An Efficient Lossless Compression Algorithm for Tiny Nodes of Monitoring Wireless Sensor Networks. In: *The Computer Journal* 52 (2009), Nr. 8, S. 969–987

[RCH+10] REINHARDT, A. ; CHRISTIN, D. ; HOLLICK, M. ; SCHMITT, J. ; MOGRE, P. ; STEINMETZ, R.: Trimming the Tree: Tailoring Adaptive Huffman Coding to Wireless Sensor Networks. In: *Proceedings of the 7th European Conference on Wireless Sensor Networks (EWSN '10)*. Coimbra, Portugal, Februar 2010

[RHS09] REINHARDT, A. ; HOLLICK, M. ; STEINMETZ, R.: Stream-oriented Lossless Packet Compression in Wireless Sensor Networks. In: *Proceedings of the Sixth Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON '09)*. Rome, Italy, Juni 2009

[Sal07] SALOMON, D.: *Data compression. The complete reference.* 4. London, Great Britain : Springer, 2007

[SR09] STECK, J. B. ; ROSING, T. S.: Adapting Performance in Energy Harvesting Wireless Sensor Networks for Structural Health Monitoring Applications. In: *Proceedings of the International Workshop on Structural Health Monitoring (IWSHM '09)*. Stanford, CA USA, September 2009

[SS82] STORER, J. A. ; SZYMANSKI, T. G.: Data Compression via Textual Substitution. In: *Journal of the ACM* 29 (1982), Oktober, S. 928–951

[TDV08] TSIFTES, N. ; DUNKELS, A. ; VOIGT, T.: Efficient Sensor Network Reprogramming through Compression of Executable Modules. In: *Proceedings of the 5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks (SECON '08)*. San Francisco Bay Area, CA, USA, Juni 2008

[Tsi07] TSIFTES, N.: Using Data Compression for Energy-Efficient Reprogramming of Wireless Sensor Networks / Swedish Institute of Computer Science. Uppsala, Sweden, Dezember 2007. – Forschungsbericht

[Wel84] WELCH, T. A.: A Technique for High-Performance Data Compression. In: *Computer* 17 (1984), Juni, Nr. 6, S. 8 –19

[ZL77] ZIV, J. ; LEMPEL, A.: A Universal Algorithm for Sequential Data Compression. In: *IEEE Transactions on Information Theory* 23 (1977), Mai, Nr. 3, S. 337 – 343

[ZL78] ZIV, J. ; LEMPEL, A.: Compression of Individual Sequences via Variable-rate Coding. In: *IEEE Transactions on Information Theory* 24 (1978), September, Nr. 5, S. 530 – 536

Anhang A

Inhalt der CD

Auf der CD befinden sich die 3500 Stromzählerdaten und die Ergebnisse der Kompressionsverfahren. Außerdem befinden sich auf der CD der Quelltext der in dieser Arbeit implementierten Kompressionsverfahren. Die Verzeichnisstruktur gliedert sich wie folgt:

- data
  - sensordata
    Auslesungen, die dieser Arbeit zur Verfügung standen
  - results
    Kompressionsleistungen und Laufzeiten der Verfahren für die Daten
- src
  - libamch
    Implementierung des Adaptive-Markov-Chain-Huffman-Verfahrens
  - libath
    Implementierung des Adaptive-Trimmed-Huffman-Verfahrens
  - liblzmh
    Implementierung des Lempel-Ziv-Markov-Chain-Huffman-Verfahrens
  - liblzw
    Implementierung des Lempel-Ziv-Welch-Verfahrens

- libmch
  Implementierung des Markov-Chain-Huffman-Verfahrens
- libslzw
  Implementierung des Static-Lempel-Ziv-Welch-Verfahrens
- libtlzma
  Implementierung des Tiny-Lempel-Ziv-Markov-Chain-Verfahren

- thesis
  - images
    Grafiken der Arbeit
  - tex
    LaTeXQuelltext der Arbeit
- README.TXT
- thesis.pdf
- thesis_sw.pdf