Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Hausarbeit

Kompression

vorgelegt von

Jonathan Balack

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang: Wirtschaftsinformatik
Matrikelnummer: 7028575

Betreuerin: Kira Duwe

Hamburg, 2019-03-09

Inhaltsverzeichnis

1 Einleitung

Das Ziel der Datenkompression ist es, die Datenmenge einer Datei zu verringern. Dabei werden verschiedene Techniken angewandt, um die Informationen in einer kompakteren Art und Weise darzustellen.
Im Feld der Datenkompression gibt es zwei Hauptgruppen, die sich von der Zielerreichung wesentlich unterscheiden. Zum einen existiert die *verlustfreie Kompression*, die ohne Verluste des Informationsgehalt auskommt. So kann eine verlustfrei komprimierte Datei in ihren exakten Ausgangszustand durch eine Dekompression zurück gelangen.
Dabei werden Redundanzen innerhalb der Daten genutzt, um durch die Entfernung dieser die Datenmenge zu reduzieren. Diese Verringerung ist vor allem bei der Speicherung und Übertragung von Daten relevant, da die Menge an Daten Kosten verursachen und zusätzlich auch physische Grenzen, wie in der Speicherdichte eines Medium, bestehen.
Für die Effizienz einer Kompression ist aber auch immer eine Abwägung zwischen dem Aufwand und der zu erreichenden Einsparung zu betrachten. Denn jede Verkleinerung bringt einen Arbeitsaufwand mit sich, der in der Regel mit der Komplexität und dem Grad der Komprimierung steigt.
Hierbei ist es auch wichtig die Grenzen des Einsparens zu kennen, denn unabhängig davon wie viel Rechenleistung und Zeit zur Verfügung steht, gibt es theoretische Grenzen, wie klein eine Information dargestellt werden kann, die nicht unterschritten werden können. Dies zeigt unter anderem das Taubenschlagprinzip, welches aussagt, dass in einen Taubenschlag mit beispielsweise 15 Plätzen und 16 Tauben sich zwei Tauben einen Platz teilen müssen. Für die Datenkompresssion bedeutet dies, dass man bei einer Reduzierung der Datenmenge auch weniger individuelle Informationen darstellen kann.
Hat man zum Beispiel 16 bit an Speicher zur Verfügung, kann man $2^{16} = 65536$ unterschiedliche Informationen speichern. Reduziert man den Speicher nun auf 15 bit, sind es nur noch $2^{15} = 32768$ mögliche Informationen [Wik19c]. Daraus folgt, dass Dateien nicht beliebig stark verlustfrei komprimiert werden können. Denn jede Datei hat einen bestimmten Informationsgehalt, der auch *Entropie* genannt wird, von dem abhängt, wie klein eine Datei komprimiert werden kann. Die Entropie hängt dabei von der Auftrittswahrscheinlichkeit der Information ab. [Duw16]
Als zweite Gruppe existiert die *verlustbehaftete Kompression*, bei der eine stärkere Kompression dadurch erreicht werden soll, dass als unwichtig eingestufte Informationen entfernt werden. Somit kann auch der Ausgangszustand nicht zurück erlangt werden, womit die Anwendungsgebiete spezifischer sind. So ist diese Art der Kompression vor allem bei Bild und Ton zu finden. Im Gegensatz zur verlustfreien Kompression gibt es hier auch keine feste Untergrenze bei der Kompression, da der Informationsgehalt und somit auch der dafür benötigte Datenumfang immer weiter reduziert werden kann, bis dahin, dass die Information komplett entfernt wird.

2 Verlustfreie Kompression

Dieses Kapitel beschäftigt sich mit der verlustfreien Kompression und wird vor allem die häufigsten Vorgehensweisen anhand von verbreiteten Verfahren beispielhaft aufzeigen.
Es gibt kein bestimmtes Verfahren, welches als Ursprung der Kompression angesehen wird. Das aber ein Bedarf für Kompression schon lange vorhanden ist, ist an der Stenografie ersichtlich, die schon im antiken Griechenland und Rom genutzt wurde. Hierbei kann die Schreibgeschwindigkeit erhöht werden, indem die zu schreibenden Informationen in einem schneller zu schreibenden Code codiert werden [Wik19i].

2.1 Morsecode

Ab 1838 wurde unter anderem von Samuel Morse der Morsecode entwickelt, welcher im Aufbau noch heute verwendeten Verfahren ähnelt.
Der Morsecode wurde zur effizienten Übermittlung von Telegrammen über Telegrafen genutzt. Telegrafen verwenden bei der Übermittlung, wie auch aktuelle Computer, ein Binärsystem.
Der internationale Morsecode wurde 1865 standardisiert und bildete eine Weiterentwicklung des von Morse vorgestellten Codes [Wik19h]. Im Morsecode stehen ein kurzes und ein langes Signal, sowie Pausen zur Verfügung. Dabei ist ein langes Signal dreimal so lang wie ein kurzes Signal. Kombinationen der Signale codieren jeweils ein Zeichen des Alphabets und die Pausen bilden einen Trenner, um zwischen den Zeichen und Wörtern trennen zu können.
Hierbei wird jedem Zeichen des Alphabets ein Code zugeordnet, dessen Länge invers zu ihrer jeweiligen Häufigkeit in der englischen Sprache ist. So ist dem Buchstabe *e*, welcher im Englischen der Häufigste ist, die kürzeste mögliche Kombination zugeordnet. Ein daraus resultierender Nachteil ist, dass das Verfahren nicht variabel ist und somit immer die gleiche Codezuordnung macht, auch wenn die Buchstabenverteilung für einen zu komprimierenden Text grundlegend anders ist als in der Standardverteilung. So gibt es schon im Deutschen eine unterschiedliche Verteilung der Zeichen, womit der auf das Englische angepasste Code hier weniger effizient ist.
Ein anderer Nachteil ist, dass der Morsecode nicht *präfixfrei* ist. Das heißt, dass der Code eines Zeichen ein Präfix eines anderen Zeichens ist. Damit trotzdem bei der Dekompression klar ist, wo die Grenzen zwischen den Zeichen sind, werden zwischen den Zeichen Pausen eingebaut. In späteren Algorithmen wird noch zu sehen sein, wie eine Kompression ohne Pausen auskommen kann und somit an Datenmenge gespart wird.

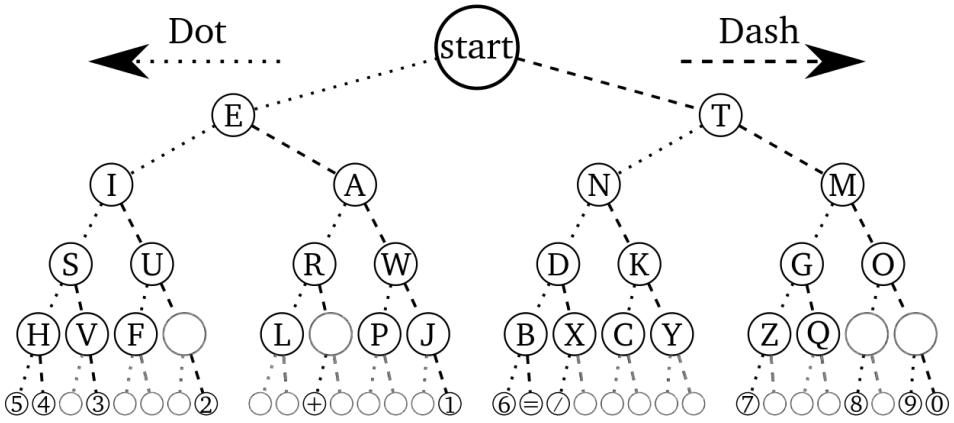

Abbildung 2.1: Binärbaum des Morsecodes [Cin10]

Der Morsecode nutzt die *Verteilungsredundanz* im Inputtext, um eine kleinere Datenmenge zu erreichen. Die Verteilungsredundanz entsteht durch eine ungleiche Verteilung der Zeichen über einen Text. So hat ein Zeichen, welches häufig vorkommt, eine kleinere Entropie, als eins was selten erscheint und kann somit kürzer codiert werden.

2.2 Huffman-Kodierung

1952 wurde von David Huffman im Rahmen einer Hausarbeit ein neues Kompressionsverfahren veröffentlicht. Wie auch der Morsecode basiert die Kompression auf der Verteilungsredundanz. Der neue Aspekt an dem Verfahren von Huffman ist, dass der Binärbaum mit den Codes von unten nach oben aufgebaut wird und für bei jeder Kompression eine individuelle Code Zuordnung geschieht. Zuvor konnte mit einem Baum, der von oben nach unten aufgebaut wurde, keine Optimalität in der Zuordnung der Codes zu den Zeichen erreicht werden. Wegen der Optimalität ist die Huffman Kodierung auch noch heute weit verbreitet.

Um die Kodierung für die Zeichen eines Texts zu ermitteln, werden als erstes die Häufigkeiten aller Zeichen im Input ermittelt. Jedes Zeichen aus dem Inputtext bildet ein Blatt des späteren Baumes und ist mit der jeweiligen Häufigkeit versehen. Nun werden immer jeweils die zwei elternlosen Knoten, die in der Summe die geringste Häufigkeit haben, mit einen gemeinsamen Elternknoten versehen. Diesem Elternknoten wird dann die Summe der Häufigkeiten der mit ihm verbundenen Knoten zugeordnet. Dieser Ablauf wird so lange fortgeführt, bis in der letzten Iteration nur noch ein Knoten ohne Elternknoten existiert, welcher dann die Wurzel des Baumes darstellt. Im nächsten Schritt des Verfahrens wird jeder links von den Elternknoten abgehenden Kante noch eine *0* zugeordnet und jeder rechten Kante eine *1*. Dies ergibt dann den jeweiligen Code für ein Zeichen, wenn der Weg von der Wurzel zum jeweiligen Blatt entlang gegangen wird.
So wird im Beispiel 2.2, dem das Wort *ROKOKOKOKOTTEN* [Sch02, S.6] zugrunde liegt, das *O* als *00* kodiert und das *R* als *110*. Für das gesamte Wort werden in ASCII

kodiert 112 bit benötigt, da in ASCII jedes Zeichen mit 8 bit kodiert ist [Wik19a]. Mit der Huffman-Kodierung kann das Wort mit nur 33 bit dargestellt werden. Allerdings muss zu den 33 bit noch der Huffmanbaum hinzugefügt werden.
Die Huffman-Kodierung ist präfixfrei. Dies wird dadurch erreicht, dass nur die Blätter des Baums Zeichen codieren und es somit nicht vorkommen kann, dass ein Zeichen der Anfang eines anderen Zeichens ist.

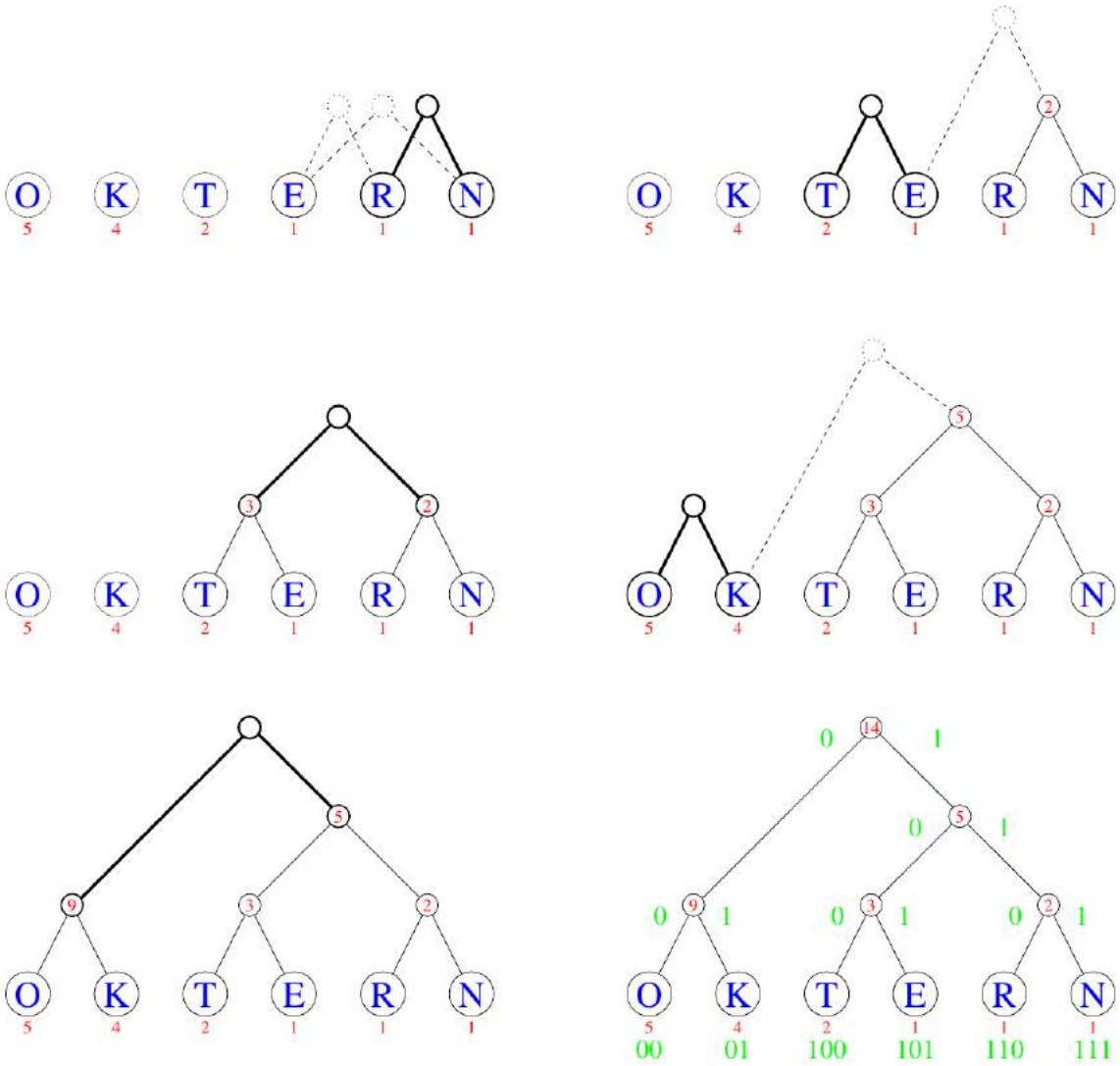

Abbildung 2.2: Erstellung eines Huffmanbaum [Sch02, S.7]

2.3 LZ-Familie

LZ77, als erster Vertreter dieser Gruppe von Algorithmen, wurde 1977 Abraham Zimpel und Jacob Ziv veröffentlicht. Ein Jahr später folgte LZ78 von ihnen [Wik19g]. Beide Verfahren nutzen *Bindungsredundanzen* zwischen Zeichen aus, um eine Kompression zu erreichen. Bindungsredundanzen bestehen, wenn die Auftrittswahrscheinlichkeit für ein Zeichen von einem anderen Zeichen und dessen Auftreten abhängt. So ist es im Deutschen sehr wahrscheinlich, dass in Texten nach dem Buchstaben q ein u folgt. Dabei ist die Bindungsredundanz nicht auf einzelne Zeichen beschränkt, so kann auch eine ganze Sequenz von Zeichen in ihrem Auftreten abhängig von anderen Zeichen oder Zeichensequenzen sein.
Genau das nutzen die Algorithmen der LZ-Familie aus und suchen nach sich wiederholenden Sequenzen. Dabei gibt es einen Suchbereich, der über den Input wandert und einen Bereich, in dem Informationen über den schon analysierten Teil des Inputs vorgehalten werden. Nun wird im Suchbereich eine möglichst lange Sequenz gesucht, die identisch schon bekannt ist.
Die genauen Spezifikationen und Abläufe für das Suchen und Speichern sind dabei zwischen den Verfahren unterschiedlich. LZ77 setzt als erster Vertreter auf ein Pointer-basiertes System, bei dem beim wiederholten Auftreten einer Sequenz auf die Stelle des vorherigen Auftreten verwiesen wird. LZ78 baut im Gegensatz dazu ein *Wörterbuch* auf, in dem bereits aufgetretene Sequenzen gespeichert werden und auf das im Fall einer Wiederholung verwiesen wird. Mittlerweile gibt es viele Weiterentwicklungen ausgehend von den ersten LZ-Kompressionsalgorithmen, sodass nun eine ganze Familie an Verfahren existiert, die jeweils unterschiedliche Eigenschaften haben. Bekannte Vertreter sind zum Beispiel DEFLATE, LZMA, LZA [McA14].

2.4 LZ77

Beim Verfahren LZ77 gibt es ein gleitendes Fenster, welches über den zu komprimierenden Text, den Input, wandert. Das gleitende Fenster besteht dabei aus zwei Teilen, einem Vorschaupuffer und einem Bereich, der folgend Wörterbuch genannt wird. Dieses ist von der Funktionsweise aber deutlich von der des Wörterbuches in LZ78 zu unterscheiden.

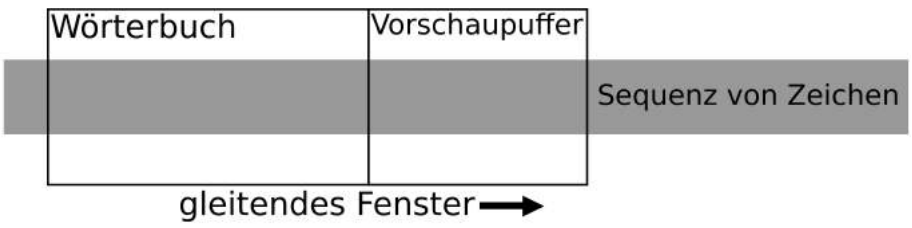

Abbildung 2.3: gleitendes Fenster bei LZ77

Im Wörterbuch bei LZ77 befindet sich der Teil des Input, der gerade gelesen wurde und

dient als Basis für mögliche Verweise. Im Vorschaupuffer sucht man eine möglichst lange Sequenz, die mit dem ersten Zeichen im Puffer beginnt, welche auch im Wörterbuch zu finden ist. Wenn so eine Sequenz gefunden wurde, wird ein Pointer erstellt, der die Position und Länge der Sequenz im Wörterbuch angibt. Ein Pointer besteht aus einem Tripel der Form (position,length,next). Der Eintrag next gibt dabei das Zeichen an, welches nach der referenzierten Stelle im Vorschaupuffer steht. [Wik19f]

Ablauf einer Iteration der LZ77-Kompression:

```
1  while Der Vorschaupuffer nicht leer ist; do
2       Durchsuche rueckwaerts das Woerterbuch nach der
          ↪ laengsten uebereinstimmenden
3                     Zeichenkette mit dem Vorschaupuffer;
4      if Eine Uebereinstimmung wurde gefunden; then
5          Gib das Tripel (Position im Woerterbuch,
6                     Laenge der gefundenen Zeichenkette,
7                     erstes nicht uebereinstimmendes Zeichen
                        ↪ aus dem Vorschaupuffer) aus;
8          Verschiebe das Fenster um die Laenge+1;
9      else
10          Gib das Tripel (0, 0, erstes Zeichen im
              ↪ Vorschaupuffer) aus;
11          Verschiebe das Fenster um 1;
12     end if
13 end while
```

Abbildung 2.4: Pseudocode des LZ77 Kompressionsverfahren [Wik19f]

12	11	10	9	8	7	6	5	4	3	2	1	0	1	2	3	4	5	6	7	Ausgabe
												r	o	k	o	k	o	k	o	(0,0,r)
											r	o	k	o	k	o	k	o	t	(0,0,o)
										r	o	k	o	k	o	k	o	t	t	(0,0,k)
									r	o	k	o	k	o	k	o	t	t	e	(2,5,t)
			r	o	k	o	k	o	k	o	t	t	e	r						(1,1,e)
	r	o	k	o	k	o	k	o	t	t	e	r								(11,1,-)

Tabelle 2.1: Beispiel einer LZ77 Kompression

Beispiel aus Tabelle 2.1: Es wird das Wort *rokokokotten* betrachtet, beginnend in der dritten Iteration. An Position *0* im Vorschaupuffer befindet sind das Zeichen *k*. Man sucht nun im Wörterbuch das Zeichen *k* und wird nicht fündig, womit das Tupel *(0,0,k)* ausgegeben wird. Als nächstes verschiebt man das Suchfenster um ein Zeichen nach

rechts, womit nun ein *o* an Postion *0* steht.
Man schaut, ob es im Wörterbuch das Zeichen *o* gibt und findet es an Position 2. Nun sucht man, ob die Sequenz *ok* zu finden ist und wird ebenfalls an Position 2 fündig. Im Vorschaupuffer folgt nun wieder ein *o*, womit nun die Sequenz *oko* im Wörterbuch gesucht wird. An dieser Stelle hat man einen speziellen Fall, denn mit der ermittelten Referenzsequenz hat man schon das Ende des Wörterbuches erreicht, wenn man aber für das *o* in den Vorschaupuffer hinein verlängert, so ist auch die Sequenz *oko* zu finden. Man darf dies tun, da die Zeichen, die sich überlappen, schon ermittelt wurden, bevor auf sie referenziert wird. Mit der Sequenz *oko* hat man die maximale Sequenz noch nicht erreicht, sondern findet, dass man noch auf *okok* und auch *okoko* verlängern kann. Somit ist die Ausgabe der vierten Iteration *(2,5,t)*.
In 5. Iteration steht ein *t* an der Position 0 und es findet sich ein *t* im Wörterbuch. Da es keine längere passende Kombination gibt, lautet die Ausgabe *(1,1,e)*. In der 6. Iteration hat man nur noch das Zeichen *r* im Vorschaupuffer, welches auch schon an Position 11 des Wörterbuch vorgehalten wird und somit darauf verwiesen wird. Da nun das Ende des Inputs erreicht ist, gibt es kein nächstes Zeichen mehr und die Ausgabe dieser Iteration lautet *(11,1,-)*.

Das Wort *rokokokotter* wurde somit zu *(0,0,r)(0,0,o)(0,0,k)(2,5,t)(1,1,e)(11,1,-)* umcodiert. Auf dem ersten Blick wirkt das nicht wie eine gute Kompression. Also rechnet man zur Probe aus, wie groß die Einsparung ist, nachdem die Ausgabe möglichst kompakt ins Binärsystem übertragen wurde. Ausgehend davon, dass die Ursprungsform in ASCII vorlag, hat jeder Buchstabe eine Länge von 8bit, womit das ganze Wort eine Länge von 96bit hatte.
Bei der Kompression kann man erste Stelle des Tripels mit 4bit codieren und die zweite mit 3bit, da die Größe des Wörterbuches und die des Vorschaupuffer entsprechend gewählt wurden. Der dritte Eintrag benötigt wieder eine ASCII-Kodierung und somit 8bit. Insgesamt brauchen wir somit für unsere sechs Tripel 90bit. Der Gewinn liegt damit bei über 6%.
Man sollte aber auch erkennen, dass bei längeren Texten und unter der Verwendung eines größeren gleitenden Fenster ein größeren Kompressionsgewinn erreichbar ist.
Dabei hängt nicht nur der Kompressionsgewinn von der Größe des Wörterbuches und des Vorschaupuffer ab, sondern auch die Geschwindigkeit, die bei wachsendem gleitenden Fester abnimmt.

Um die komprimierten Daten wieder dekodieren zu können, geht man die Tripel von vorne nach hinten durch und schaut, welche Informationen man jeweils erhält (siehe Tabelle 2.2).

Eingabe	12	11	10	9	8	7	6	5	4	3	2	1	0
(0,0,r)													r
(0,0,o)												r	o
(0,0,k)											r	o	k
(2,5,t)					r	o	k	o	k	o	k	o	t
(1,1,e)			r	o	k	o	k	o	k	o	t	t	e
(11,1,-)	r	o	k	o	k	o	k	o	t	t	e	r	

Tabelle 2.2: Beispiel einer LZ77 Dekompression

2.5 Vergleich

Ein allgemeiner Vergleich der Kompressionsverfahren ist schwierig, da es viele Algorithmen gibt, vielfältige Unterschiede zwischen Eigenschaften der Verfahren bestehen, die zusätzlich noch abhängig von der gewählten Kompressionsstufe sind, sowie auch für jedes System die Ergebnisse der Verfahren wieder anders ausfallen. Das kann unter anderem daran liegen, dass nur eine begrenzte Anzahl an Kernen des Prozessors genutzt werden oder spezielle Features nur bei gewisser Hardware und Software zur Verfügung stehen. Es gibt zu viele Variablen um ein, für alle Einsatzorte gültiges, Ergebnis liefern zu können. In Tabelle 2.3 sind, um einen ersten Eindruck der Unterschiede zu erhalten, einige Programme mit Testwerten angegeben, die Kompressionsalgorithmen der LZ-Familie implementiert haben.

Kompressionsprogramm	Kompression	Dekompression	kompr. Größe	Verhältnis
memcpy	8657 MB/s	8891 MB/s	211947520 B	100.00
yalz77 -1	71 MB/s	358 MB/s	93952728 B	44.33
yalz77 -12	14 MB/s	344 MB/s	84050625 B	39.66
lzsse8 -12	6.08 MB/s	2842 MB/s	75464339 B	35.61
libdeflate -1	117 MB/s	570 MB/s	73318371 B	34.59
libdeflate -12	4.63 MB/s	583 MB/s	64801629 B	30.57
lzma 16.04 -9	1.55 MB/s	67 MB/s	48742901 B	23.00

Tabelle 2.3: Vergleich verschiedener Kompressionsalgorithmen
Testbedingungen: Ein Kern des Intel Core i5-4300U mit Windows 10 64bit
Datensatz: Silesia compression corpus [ini18]

Für den Benchmark wurde der *Silesia compression corpus* als Datensatz genutzt, der aus verschieden Dateitypen, wie Texten und Programmdateien, besteht. Er besitzt eine Größe von 211.947.520 Byte. Als Vergleich, um mögliche Hardwarekapazitätsgrenzen abschätzen zu können, ist auch das Ergebnis von memcpy angegeben. Dieses Programm kopiert die angegebene Datei ohne sie dabei zu verändern [cWc].

Zuerkennen ist, dass DEFLATE (implementiert durch libdeflate) in diesem Benchmark in Kompressionsstufe 1 in allen Parametern ein besseres Ergebnis abliefert als

LZ77 (yalz77). Es gibt zudem auch Verfahren, wie hier LZMA, die eine noch stärkere Kompression erreichen, dafür aber deutlich mehr Zeit benötigen.
Man sollte somit für jeden Anwendungsfall einzeln schauen, welcher Algorithmus von den Parametern auf den eingesetzten Systemen am besten passt. Als Beispiel könnte man, ausgehend von den hier vorliegenden Parametern, sich vorstellen, dass LZSS (lzsse8) sich gut für Videos eignet, da diese einmal komprimiert werden und dann häufig und vor allem mit einer hohen Geschwindigkeit dekomprimiert werden können.

3 Verlustbehaftete Kompression

Bei der verlustbehafteten Kompression werden Dateien unter Verlust von Informationen verkleinert. Dabei werden als irrelevant eingestufte Informationsbestandteile weggelassen. Die Verfahren, die dabei einschätzen, was unwichtig ist, sind meist an die menschliche Wahrnehmung angepasst. In diesem Kapitel wird JPEG als Format mit verlustbehafteter Kompression erklärt.

3.1 JPEG

JPEG ist eines der meistbenutzten Formate zur Speicherung von Bildern. JPEG wurde von der Joint Photographic Experts Group entwickelt und 1992 normiert. Dabei werden nach einem vorgegebenen Schema die Informationen verringert, um eine kleinere Datenmenge zu erlangen. Um ein Bild in das JPEG Format zu formatieren, sind folgende Schritte nötig:

- Umwandlung ins YCbCr-Farbmodell
- Einteilung in 8×8-Blöcke
- Diskrete Kosinustransformation
- Quantisierung
- Huffman-Kodierung

3.1.1 Umwandlung ins YCbCr-Farbmodell

Als erster Schritt steht zumeist eine Umwandlung des Farbmodells an. Das liegt daran, dass viele andere Formate das RGB-Modell nutzen, JPEG aber das YCbCr-Farbmodell. In diesem Modell gibt es drei Farbkanäle, die kombiniert das finale Bild ergeben. Dabei steht das Y für die Luminanz, die in einem eigenen Kanal behandelt wird und für Wahrnehmung wichtiger ist, als die Farben, die über den Blauwert und den Rotwert gespeichert werden (siehe Abbildung 3.1) [Wik19e]. Somit kann an dieser Stelle auch eine Datenverringerung durch eine Verkleinerung der Auflösung der Farbkanäle erreicht werden. Im Folgenden wird nur noch ein beliebiger der drei Kanäle betrachtet, da das Verfahren für sie gleich ist.

Abbildung 3.1: YCbCr-Farbmodell [Wdw06]

3.1.2 Einteilung in 8x8-Blöcke

In diesem Schritt wird das Bild in Blöcke bestehend aus 8x8 Pixeln aufgeteilt. In den nächsten beiden Schritten werden die Blöcke separat verarbeitet. Falls ein Bild eine Auflösung besitzt, bei der die Einteilung nicht aufgeht, werden zusätzliche Pixel ergänzt oder kopiert.

3.1.3 Diskrete Kosinustransformation

Die 64 Pixel werden nun hinter einander aufgeschrieben und mit der *Diskreten Kosinustransformation* (DCT) wird eine Kosinusfunktion berechnet, die den Pixelwerten möglichst genau entspricht. Dafür stehen 64 verschiedene vorgegebene Kosinusfunktionen zur Verfügung, die genutzt werden um diese Kombination zu erreichen (siehe Abbildung 3.2). Es stehen dabei Funktionen für Kombinatinen aus vertikalen und horizontalen Veränderungen und grobe und feine Strukturen zur Wahl. Als Ergebnis dieses Schritts erhält man für jeden 8x8-Pixelblock jeweils einen Wert pro Standardkosinusfunktion, welcher den jeweiligen Anteil der Funktion an der Gesamtfunktion angibt [Pou15].

3.1.4 Quantisierung

Bei der *Quantisierung* wird durch eine Quantisierungstabelle die Gewichtung der 64 einzelnen Kosinusfunktion für die Wahrnehmung angegeben. So haben die Funktionen, die die groben Verläufe darstellen ein höheres Gewicht als die feinen Strukturen. Die Wahl der Quantisierungstabelle und damit die Gewichtungen hängt von der Kompressionstufe

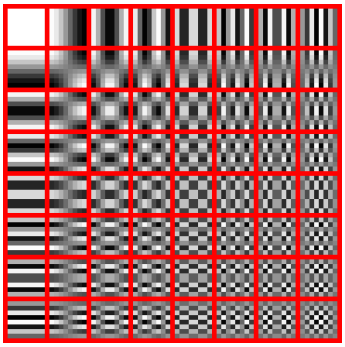

Abbildung 3.2: Kosinusfunktionen in der DCT [Fc07]

und dem genutzten Programm ab. Um die Gewichtungen anzuwenden, werden die Daten aus der DCT F durch die Werte der Quantisierungsmatrix Q geteilt und auf ganze Zahlen gerundet. Dabei kommt es besonders bei höheren Kompressionstufen häufig vor, dass die Werte auf 0 gerundet werden. So werden vor allem feine Strukturen im Bild entfernt. Wie dies geschieht, ist in der Matrix F^Q in Abbildung 3.3 zusehen. Die Quantisierung in der JPEG-Formatierung ist der wesentliche Schritt, an dem Informationen verloren gehen und Redundanzen entstehen [Pou15].

3.1.5 Huffman-Kodierung

Die zwar immer noch 64 Zahlen werden jetzt durch die Huffman-Kodierung komprimiert, womit die bei der Quantisierung entstandenen Redundanzen entfernt werden. Dies ist sehr effizient, da, besonders in höheren Kompressionsstufen, durch das häufige Auftreten der Null eine hohe Verteilungsredundanz vorhanden ist [Wik19e].

$$Q = \begin{bmatrix} 10 & 15 & 25 & 37 & 51 & 66 & 82 & 100 \\ 15 & 19 & 28 & 39 & 52 & 67 & 83 & 101 \\ 25 & 28 & 35 & 45 & 58 & 72 & 88 & 105 \\ 37 & 39 & 45 & 54 & 66 & 79 & 94 & 111 \\ 51 & 52 & 58 & 66 & 76 & 89 & 103 & 119 \\ 66 & 67 & 72 & 79 & 89 & 101 & 114 & 130 \\ 82 & 83 & 88 & 94 & 103 & 114 & 127 & 142 \\ 100 & 101 & 105 & 111 & 119 & 130 & 142 & 156 \end{bmatrix}$$

$$F = \begin{bmatrix} 782{,}91 & 44{,}93 & 172{,}52 & -35{,}28 & -20{,}58 & 35{,}93 & 2{,}88 & -3{,}85 \\ -122{,}35 & -75{,}46 & -7{,}52 & 55{,}00 & 30{,}72 & -17{,}73 & 8{,}29 & 1{,}97 \\ -2{,}99 & -32{,}77 & -57{,}18 & -30{,}07 & 1{,}76 & 17{,}63 & 12{,}23 & -13{,}57 \\ -7{,}98 & 0{,}66 & 2{,}41 & -21{,}28 & -31{,}07 & -17{,}20 & -9{,}68 & 16{,}94 \\ 3{,}87 & 7{,}07 & 0{,}56 & 5{,}13 & -2{,}47 & -15{,}09 & -17{,}70 & -3{,}76 \\ -3{,}77 & 0{,}80 & -1{,}46 & -3{,}50 & 1{,}48 & 4{,}13 & -6{,}32 & -18{,}47 \\ 1{,}78 & 3{,}28 & 4{,}63 & 3{,}27 & 2{,}39 & -2{,}31 & 5{,}21 & 11{,}77 \\ -1{,}75 & 0{,}43 & -2{,}72 & -3{,}05 & 3{,}95 & -1{,}83 & 1{,}98 & 3{,}87 \end{bmatrix}$$

$$F^Q = \begin{bmatrix} 78 & 3 & 7 & -1 & 0 & 1 & 0 & 0 \\ -8 & -4 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & -2 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Abbildung 3.3: Matrizen bei der Quantisierung [Wik19e]

4 Kompressionsdateisysteme

Ein Kompressionsdateisystem ist in der Lage Dateien automatisch beim Schreiben auf das Dateisystem zu komprimieren und beim Lesen zu dekomprimieren. Dies geschieht "*On-the-Fly*", sodass, bis auf einen Geschwindigkeitsunterschied, davon in der Nutzung nichts direkt zu merken sein sollte. Die Geschwindigkeit des Speichers muss, trotz des zusätzlichen Rechenaufwand, nicht zwangsläufig abnehmen. Dies ist unter anderem der Fall, wenn die Rechenleistung des Computers die Geschwindigkeit des Speichers übersteigt. Kompression wird auch innerhalb von Flashspeicher genutzt, um durch eine Kompression weniger Platz auf dem Speicher zu verbrauchen. Dabei wird der Gewinn nicht als zusätzlicher Speicherplatz an das System freigegeben, sondern zur Verlängerung der Lebenszeit des Speichers freigehalten.

4.1 CramFS

CramFS ist ein in eingebetteten Systemen, wie Routern, weit verbreitetes freies Read-only-Dateisystem mit eingebauter Kompression. Die Metadaten liegen dabei unkomprimiert vor, sodass das Dateisystem zur Nutzung nicht erst dekomprimiert werden muss. Deshalb eignet es sich besonders zur Erstellung von Systemabbildern zur Installation und wurde für verschiedene Linux-Distribution genutzt. Dabei wird zlib als Kompressionssoftware genutzt, welche den DEFLATE-Algorithmus verwendet. CramFS kann mit Dateien bis zu einer Größe von 16 Mebibyte umgehen und ein Dateisystem die maximale Größe von knapp über 256 Mebibyte besitzen. Als Nachfolger existiert SquashFS, welches mit größeren Datenmenge agieren kann, aber im Vergleich zu CramFS mehr Hardwareressourcen benötigt [Wik19b].

4.2 Btrfs

Btrfs ist ein modernes Dateisystem, welches von OpenSuse als Standarddateisystem genutzt wird. Ein optionales Feature vom Btrfs ist die Datenkompression, für die die drei Algorithmen ZLIB, LZO, und ZSTD zur Verfügung stehen. Dabei findet eine Kompression auch nur bei den Dateien statt, die dadurch kleiner dargestellt werden können, als die Ursprungsdatei. Btrfs ist in den Linuxkernel integriert und um die Kompression zu nutzen, muss das Dateisystem mit der Option *-o compress* gemountet werden [con18].

5 Zusammenfassung

Zusammenfassend ist zuerkennen, dass Kompression eine wichtige Funktion ist, um Ressourcen besser auszunutzen zu können. Dabei ist die verlustfreie Kompression an vielen Stellen einsetzbar, da keine Informationen verloren gehen und somit darauf auch keine Rücksicht genommen werden muss. Heute gehören viele der aktuell verwendeten Algorithmen zu der Familie, die ihren Ursprung in LZ77 und LZ78 hat.
Jedes Verfahren besitzt eine unterschiedliche Methode die Kompression und Dekompression durchzuführen und damit einhergehend auch andere Eigenschaften.
Verlustfreie Kompression wird unter anderem genutzt, um automatisiert Dateien oder ganze Dateisysteme zu verkleinern.
Zudem gibt es die verlustbehaftete Kompression, die, da gewisse Information verloren gehen, nur in begrenzten Rahmen eingesetzt wird, vor allem bei Bild- und Tondateien.

Literaturverzeichnis

[Cin10] Cinnagingercat. Morse-code-tree.svg, June 2010. [Online; accessed 27-Februar-2019] https://commons.wikimedia.org/wiki/File:Morse-code-tree.svg.

[con18] Kernel Wiki contributors. Compression, 2018. [Online; accessed 24-Februar-2019] https://btrfs.wiki.kernel.org/index.php/Compression.

[cWc] cpluplus Wiki contributors. memcpy. [Online; accessed 09-March-2019] http://www.cplusplus.com/reference/cstring/memcpy/.

[Duw16] Kira Isabel Duwe. Data Reduction Techniques„ September 2016. https://wr.informatik.uni-hamburg.de/_media/teaching/wintersemester_2015_2016/pre-1516-duwe-datenreduktion.pdf.

[Fc07] FelixH-commonswiki. Dctjpeg.png, January 2007. [Online; accessed 22-Januar-2019] https://commons.wikimedia.org/wiki/File:Dctjpeg.png.

[ini18] inikep. lzbench, 2018. [Online; accessed 25-Januar-2019] https://github.com/inikep/lzbench/.

[Lar17] Michael Larabel. Btrfs Zstd Compression Benchmarks On Linux 4.14, November 2017. [Online; accessed 25-Februar-2019] https://www.phoronix.com/scan.php?page=article&item=btrfs-zstd-compress.

[McA14] Colt McAnlis. The LZ77 Compression Family (Ep 2, Compressor Head), May 2014. [Online; accessed 20-Januar-2019] https://www.youtube.com/watch?v=Jqc418tQDkg.

[Nag16] Amir Nagah. 54- The JPEG compression algorithm, May 2016. [Online; accessed 24-Januar-2019] https://www.youtube.com/watch?v=aFbGqXFT0Nw.

[Pou15] Mike Pound. JPEG DCT, Discrete Cosine Transform (JPEG Pt2)- Computerphile, May 2015. [Online; accessed 24-Januar-2019] https://www.youtube.com/watch?v=Q2aEzeMDHMA.

[Sch02] Oliver Schmid. Visualisierung von verlustlosen Kompressionsalgorithmen, May 2002. [Online; accessed 18-Januar-2019] https://pi4.informatik.uni-mannheim.de/pi4.data/content/animations/losslesscompression/studarbeit.pdf.

[Wdw06] Wdwd. Barns grand tetons YCbCr separation.jpg, December 2006. [Online; accessed 21-Januar-2019] https://commons.wikimedia.org/wiki/File:Barns_-grand_tetons_YCbCr_separation.jpg.

[Wik19a] Wikipedia contributors. American Standard Code for Information Interchange — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 08-March-2019] https://de.wikipedia.org/wiki/American_Standard_Code_for_-Information_Interchange.

[Wik19b] Wikipedia contributors. CramFS — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 25-Februar-2019] https://de.wikipedia.org/wiki/CramFS.

[Wik19c] Wikipedia contributors. Datenkompression — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 14-Januar-2019] https://de.wikipedia.org/wiki/Datenkompression.

[Wik19d] Wikipedia contributors. Huffman-Kodierung — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 18-Januar-2019] https://de.wikipedia.org/wiki/Huffman-Kodierung.

[Wik19e] Wikipedia contributors. JPEP — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 24-Januar-2019] https://de.wikipedia.org/wiki/JPEG.

[Wik19f] Wikipedia contributors. LZ77 — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 22-Januar-2019] https://de.wikipedia.org/wiki/LZ77.

[Wik19g] Wikipedia contributors. LZ77 and LZ78 — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 21-Januar-2019] https://en.wikipedia.org/wiki/LZ77_-and_LZ78.

[Wik19h] Wikipedia contributors. Morsezeichen — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 18-Januar-2019] https://de.wikipedia.org/wiki/Morsezeichen.

[Wik19i] Wikipedia contributors. Stenografie — Wikipedia, The Free Encyclopedia, 2019. [Online; accessed 28-Februar-2019] https://de.wikipedia.org/wiki/Stenografie.