Kompressionsalgorithmen - Die LZ-Familie

Seminar “Effiziente Programmierung”

Hendrik Pfennig 6944373

8. März 2019

Betreuer: Kira Duwe

Inhaltsverzeichnis

1 Einleitung

Das Thema der Datenkompression spielt in der Informationsverarbeitung eine äußerst wichtige Rolle, da Speicherkapazität generell eine begrenzte (und kostenintensive) Ressource in Computersystemen darstellt. Sie findet ihre Anwendungsgebiete unter anderem in der Archivierung, aber auch bei der Beschleunigung von Datenübertragung im Internet. In dieser Arbeit werden ausgewählte, in Abbildung 1 zu sehende, Algorithmen aus der LZ-Familie vorgestellt, welche die Grundlage für die meisten heutzutage verwendeten Kompressionsalgorithmen bildet.

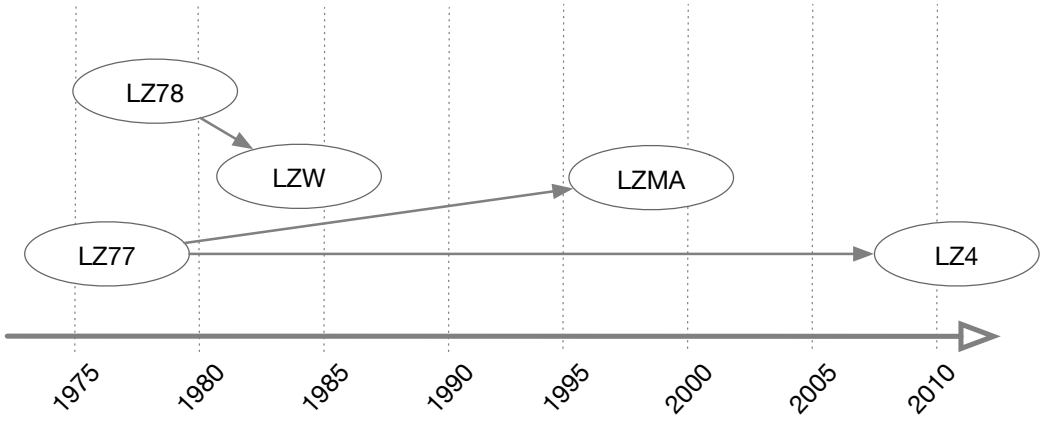

Abbildung 1: Übersicht der vorgestellten Algorithmen

2 Grundlagen der Kompression

In der Kompression gibt es drei wesentliche Qualitäten, die ein Algorithmus haben kann. Das sind zum einen die **Kompressionsrate**, welche das Größenverhältnis einer Datei zu ihrer komprimierten Version beschreibt, zum anderen sind es die **Kompressionsgeschwindigkeit** und die **Dekompressionsgeschwindigkeit**.

Außerdem gibt es zwei Eigenschaften anhand derer man die Eingabe klassifizieren kann, **Entropie** [1] und **Kolmogorov-Komplexität** [2]. Erstere wird über die Formel $-\sum_i p_i \log p_i$ definiert und beschreibt den mittleren

Informationsgehalt der Zeichen einer Eingabequelle, wobei p_i für die Auftrittswahrscheinlichkeit des $i$ten Zeichens im Alphabet steht. Diese Formel bewirkt, dass Zeichen mit einer geringen Auftrittswahrscheinlichkeit, zu höheren Ergebnissen führen. Entsprechend ergeben Zeichen, die häufig auftreten, eine geringere Information.

Kolmogorov-Komplexität misst die Komprimierbarkeit einer Eingabe S und beschreibt das kürzeste, S erzeugende Programm. Somit gilt immer $Kolmogorov(S) \leq S$, da S als triviales, S erzeugendes Programm gilt. Die Kolmogorov-Komplexität ist nicht berechenbar, bildet aber eine theoretische untere Schranke für verlustfreie Komprimierung, ein Prinzip auf das im Folgenden weiter eingegangen wird. Ein Beispiel für Kolmogorov-Komplexität wäre die Erzeugung von *abababababababababab* durch $ab * 10$.

3 Arten von Kompressionsalgorithmen

Auf abstrahierter Ebene lässt sich die Menge der Kompressionsalgorithmen in zwei disjunkte Untermengen aufteilen, die verschiedene Charakteristiken der Kompression aufweisen.

Zum einen gibt es die **verlustbehaftete Kompression**, die einen Informationsverlust mit sich führt um stärkere Verringerung des Speicherbedarfs zu ermöglichen, zum anderen die **verlustfreie Kompression** bei der die vollständige Information der Ursprungsdatei beibehalten wird. Aus diesen beiden Grundprinzipien ergeben sich verschiedene Anwendungsgebiete für Kompressionsalgorithmen. Da alle in dieser Arbeit behandelten Algorithmen Elemente der verlustfreien Kompressionsalgorithmen sind, wird auf verlustbehaftete Kompression nur grundlegend eingegangen.

3.1 Verlustbehaftete Kompression

Diese Art der Kompression findet häufig Verwendung bei Bild-, Audio-, und Videodateien. Entsprechende Algorithmen verringern den Speicherplatz, den eine Datei einnimmt, indem sie deren Kolmogorov-Komplexität verringern, sodass sie sich kürzer kodieren lässt. Dieses Verfahren zieht in aller Regel einen irreversiblen Informationsverlust mit sich. Hierbei wird sich zu Nutze gemacht, dass die menschliche Perzeption, im Bezug auf visuelle und auditive Rezeption, die empfangene Information so filtert, dass nur "relevante" Aspekte im Gehirn verarbeitet werden. Das hat zur Folge, dass gewisse Teile

dieser Information weniger bis gar nicht wahrgenommen werden (können). Um trotz des Informationsverlusts eine größtmögliche Qualität zu erhalten, werden daher nach Möglichkeit jene Informationen vernichtet, die ohnehin vom Menschen übersehen werden. Ein passendes Beispiel dafür bietet die mp3-Kodierung von Audiodaten, bei der unter anderem leise Signalanteile entfernt werden, die unmittelbar nach lauteren auftreten, da solche Geräusche von Menschen nicht wahrgenommen werden. Auch können Signalanteile in Frequenzen, die am Rande der Wahrnehmungsschwelle liegen vereinfacht werden ohne einen hörbaren Unterschied zu verursachen. Auf diese Weise lassen sich hohe Kompressionsraten bei für den Menschen kaum merklichem Qualitätsverlust erzielen.

Ein weiteres Beispiel bietet die JPEG-Kompression, bei der ebenfalls Informationen verworfen werden, die eine geringere Relevanz für die menschliche Informationsverarbeitung haben. Dies ist in Abbildung 2 zu sehen.

(a) Dateigröße: 4,6 MB (b) Dateigröße: 327 KB

Abbildung 2: Beispiel verlustbehaftete JPEG Kompression
Trotz der hohen Kompressionsrate von 13:1 zwischen 2a und 2b, erkennt man den Vordergrund des komprimierten Objekts sehr gut. Im Hintergrund des Bildes sieht man sogenannte Kompressionsartefakte.

3.2 Verlustfreie Kompression

Im Gegensatz dazu, wird bei der verlustfreien Kompression algorithmisch bestimmt, wie sich die zu komprimierenden Daten möglichst effizient kodieren lassen, während die Information vollständig erhalten bleibt. Dazu werden Redundanzen im Datenstrom gefunden und entfernt. Offensichtlich markiert die Kolmogorov-Komplexität eine untere Schranke für die Länge der Kodierung, weshalb verlustfreie Kompressionsalgorithmen im Regelfall deutlich

geringere Kompressionsraten erzielen, als verlustbehaftete. Diese Klasse von Algorithmen findet ihre Anwendung zum Beispiel in der Kompression von ausführbaren Programmen oder Text, da bei ersterem ein Informationsverlust das Programm unbenutzbar machen würde, und bei letzterem für gewöhnlich der vollständige Informationsgehalt erhalten bleiben soll. Die Arbeitsweise dieser Algorithmen sorgt meistens dafür, dass komprimierte Dateien vom Menschen nicht intuitiv lesbar sind; es muss zunächst durch den passenden Dekompressionsalgorithmus die Originaldatei wiederhergestellt werden. Als Beispiele werden im Folgenden die Algorithmen LZ77, LZ78, LZW, LZMA und LZ4 vorgestellt.

4 LZ77

Der im Jahre 1977 von Abraham Lempel und Jacob Ziv in der Veröffentlichung "A Universal Algorithm for Sequential Data Compression" [3] vorgestellte Algorithmus LZ77 bietet das Grundkonzept, auf dem die meisten heute gängigen verlustfreien Kompressionsalgorithmen (LZMA, Deflate, ...) aufbauen. Die wesentliche Mechanik dieses Algorithmus‘ beruht darauf, Redundanzen der Eingabe, die als Strom abgearbeitet wird, durch Zeiger auf vorhergegangene Vorkommnisse zu ersetzen, um somit eine kürzere Ausgabe erzeugen zu können.

4.1 Kompression

Zur Kompression werden zwei Datenstrukturen verwendet, die jeweils ein zusammenhängendes Stück der Eingabe enthalten. Erstere, genannt "Search Buffer" enthält hierbei stets das größte hinein passende Stück der zuletzt gelesenen Zeichen (Beispielsweise die letzten 2000 verarbeiteten Zeichen) des Inputs während letztere, genannt "Look Ahead Buffer" gefüllt ist mit den als nächstes zu lesenden Zeichen. In jedem Verarbeitungsschritt wird nun das längst mögliche Stück vom Anfang des Look Ahead Buffers gewählt, zu dem es eine Übereinstimmung im Search Buffer gibt. Wenn man als aktuelle Position die Stelle zwischen dem letzten Zeichen des Search Buffers und dem ersten Zeichen des Look Ahead Buffers betrachtet, gilt als Abstand zu der Übereinstimmung die Anzahl der Positionen, die man im Search Buffer zurück gehen muss, damit das erste Zeichen der Übereinstimmung rechts

von der Position ist. Die Verfahrensweise der LZ77-Kompression lässt sich in Abbildung 3 erkennen.

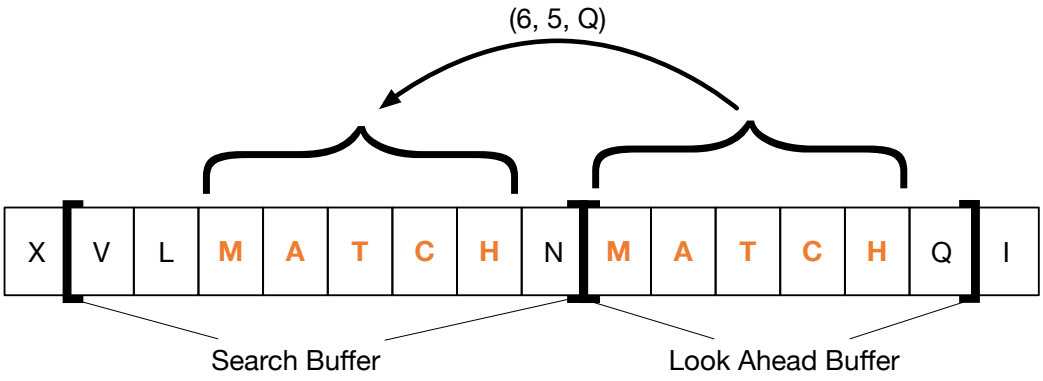

Abbildung 3: Matchfindung LZ77
Orange markiert ist eine Übereinstimmung der beiden Buffer. Im Tripel über dem Pfeil lässt sich die Ausgabe der Form (Abstand, Länge, Literal) erkennen.

Als Ausgabe erfolgt in jedem Schritt nun ein Tripel, bestehend aus ebenjenem Abstand, der Länge des übereinstimmenden Wortes, sowie des Zeichens, das im Look Ahead Buffer an erster Stelle nach dem Treffer steht. Vor dem nächsten Schritt werden dann die Übereinstimmung und das Einzelzeichen ans Ende des Search Buffers gehängt und aus dem Look Ahead Buffer entfernt. Außerdem wird der Look Ahead Buffer, so denn der Eingabestring noch nicht zu Ende ist, am Ende weiter aufgefüllt. Sollte es keine Übereinstimmung geben, wird das Tripel (0, 0, Literal) ausgegeben, wobei Literal für das erste Zeichen im Look Ahead Buffer steht, gefolgt von der eben beschriebenen Aktualisierung der beiden Buffer.

4.2 Dekompression

Zur Dekompression werden die Tripel einzeln aufgelöst. Sollten Distanz und Länge nicht 0 sein, wird das Wort w ausgegeben, wie in Abbildung 4 definiert, gefolgt vom Literal des Tripels.

$$w = \mathrm{SB}[\,\mathbf{len}\,(\mathrm{SB})-\mathrm{Distanz}:\mathbf{len}\,(\mathrm{SB})-\mathrm{Distanz}+\mathrm{Laenge}\,)$$

Abbildung 4: Definition von w
SB steht für den Searchbuffer. Die Notation $SB[a:b)$ beschreibt das Wort, welches sich aus der Konkatenation der Buchstaben, von der Stelle a einschließlich, bis zur Stelle b ausschließlich, von SB ergibt.

Ansonsten wird nur das Literal ausgegeben. In beiden Fällen wird anschließend das nächste Tripel verarbeitet, solange bis kein weiteres vorhanden ist.

5 LZ78

In der Arbeit "Compression of Individual Sequences via Variable-Rate Coding" [4] publizierten die gleichen Autoren ein Jahr später den LZ78-Algorithmus, der ein weiteres, häufig verwendetes Grundkonzept der verlustlosen Datenkompression bietet.

5.1 Kompression

Anders als bei LZ77 werden bei LZ78 keine Zeiger verwendet um Redundanzen zu ersetzen, sondern es wird zur Laufzeit ein sogenanntes Wörterbuch generiert, in dem bestimmte Zeichenketten einem Index zugewiesen werden. Bei erneutem Eintreffen der Zeichenkette wird stattdessen der entsprechende Index ausgegeben. Dazu wird in jedem Iterationsschritt des Kompressors das längste Wort eingelesen, das bereits im Wörterbuch steht. Sollte es keine Übereinstimmung geben, wird eine 0 gefolgt von einem Literal (dem nächsten Byte im Input) ausgegeben. Ansonsten erfolgt als Ausgabe eine 1, gefolgt vom Index des im Wörterbuch gefundenen Wortes, gefolgt von dem unkodierten nächsten Buchstaben. Daraufhin wird ein neuer Eintrag im Wörterbuch

angelegt, der niedrigste noch nicht belegte Index enthält nun die Konkatenation des Wortes mit dem Buchstaben. Da die Indizes des Wörterbuches mit maximal 12 Bit breiten Wörtern gespeichert werden, hat es eine begrenzte Kapazität für Einträge, nämlich $2^{12} = 4096$.

5.2 Dekompression

Auch bei der Dekompression wird das Wörterbuch wieder zur Laufzeit generiert. Bei einer gelesenen 0 wird das nächste Byte ausgegeben und dem Wörterbuch geeignet angefügt. Wenn eine 1 gelesen wird, wird der darauf folgende Index ausgewertet, und das entsprechende Wort zusammen mit dem auf den Index folgenden Byte ausgegeben. Die Konkatenation des ausgegebenen Wortes und des Literals wird auch hier wieder dem Wörterbuch angefügt.

6 LZW

LZW ist vermutlich die am weitesten verbreitete Version des LZ78-Algorithmus, vorgestellt von Terry Welch im Jahre 1984 in dem Artikel "Technique for high-performance data compression" [5]. Der wesentliche Unterschied zu LZ78 ist, dass es keine Ausgabe von Literalen mehr gibt. Stattdessen wird das Wörterbuch mit allen 256 1-Byte Sequenzen initialisiert. Als Ausgabe erfolgen dann lediglich die Indizes des Wörterbuches. Genau wie bei LZ78 Wird das Wörterbuch immer um das Wort erweitert, welches sich aus der Konkatenation des zuletzt kodierten Wortes mit dem nächsten Buchstaben im Eingabestrom ergibt, zusammen mit dem kleinsten noch nicht belegten Index im Wörterbuch. Eine weitere Modifikation ist die Einführung von optionalen "Clear-codes", welche sämtliche Einträge (bis auf die 1-Byte Sequenzen) entfernt, sollte das Wörterbuch voll sein. Wichtig ist hierbei, dass der Dekompressor mit dem Kompressor, im Bezug auf die Verwendung der Clear-codes, übereinstimmt.

7 LZMA

Der LZMA Algorithmus (**L**empel **Z**iv **M**arkov Chain **A**lgorithm) [6] von Igor Pavlov aus dem Jahre 1998, verbindet das LZ77 Konzept mit einer arithmetischen Bereichskodierung. Da das Wörterbuch hier eine Größe von bis zu 4GB hat, sind für die effiziente Verwendung nicht-triviale Suchstrukturen erforderlich. Hierzu werden in der Regel Hashtabellen oder spezielle binäre Suchbäume verwendet. Des Weiteren verwendet LZMA Markov-Ketten um Voraussagen zu treffen, die es ermöglichen Kodierungen zu verkürzen. LZMA findet außerdem bei den verbreiteten Programmen "7zip" und "tar" Verwendung.

7.1 Unterschiede zu LZ77

Während bei LZ77 die Ausgabe in Tripeln der gleichen Struktur erfolgt, unterscheidet LZMA insgesamt sieben verschiedene Ausgaben, die jeweils durch ein Präfix gekennzeichnet sind. Hierbei handelt es sich zunächst um einfache Literalausgabe, sowie die normale LZ77-Sequenz. Zusätzlich gibt es noch fünf Sequenzen zur Wiederholung der vier zuletzt verwendeten Distanzen, welche daher sowohl beim kodieren als auch beim dekodieren gespeichert werden müssen.

Name	**Kodierung**	**Wortbreite** (in Bit)	$\mathbf{\Sigma}$ (in Bit)
Literal	$0 + \text{Literal}$	$1 + 8$	9
LZ77-Sequenz	$10 + \text{len} + \text{dist}$	$2 + [4, 10] + 32$	[38, 44]
Shortrep	1100	4	4
Longrep[0]	$1101 + \text{len}$	$4 + [4, 10]$	[8, 14]
Longrep[1]	$1110 + \text{len}$	$4 + [4, 10]$	[8, 14]
Longrep[2]	$11110 + \text{len}$	$5 + [4, 10]$	[9, 15]
Longrep[3]	$11111 + \text{len}$	$5 + [4, 10]$	[9, 15]

Tabelle 1: Liste der LZMA Sequenzen
Anhand dieser Tabelle lässt es sich gut erkennen, dass die häufige Verwendung von repetitions eine platzsparendere Alternative zur LZ77-Sequenz bietet.

Die Shortrep-Sequenz (Shortrep für short repetition, Longrep analog) ist ein Zeiger auf ein einzelnes Byte, welches genau die zuletzt verwendete Distanz zurückliegt. Dies ermöglicht die Speicherung eines Bytes mit nur vier Bit. Die Longrep-Sequenzen haben eine explizit gespeicherte Länge, und greifen entsprechend ihrer Ordnung auf die vier zuletzt verwendeten Distanzen zu. Da bei den LZ77-Sequenzen die Distanz mit je vier Bytes kodiert wird, profitiert LZMA von häufiger Verwendung der "repetitions", welche die explizite Speicherung von Distanzen umgehen.

7.2 Bereichskodierung

Auf die binäre Ausgabe des LZ77 artigen Kompressors, wird bei LZMA eine arithmetische Bereichskodierung [7] angewandt, um die Kompressionsrate zu steigern. Der Algorithmus des Bereichskodierers erfolgt, bis auf die Ausgabe, wie im Pseudocode 5, es wird eine statische Wahrscheinlichkeitsverteilung von 80% Nullen, sowie das Intervall $[0, 100\,000]$ angenommen.

```
probZero = 0.8
ranges = [0, 100 000]
while inputStream != "":
    c = nextChar(inputStream)
    x = ranges[0] + probZero * (ranges[1] - ranges[0])
    if c == 0:
        ranges = [ranges[0], x]
    else:
        ranges = [x+1, ranges[1]]
```

Abbildung 5: Bereichskodierer
In jedem Schritt wird das Intervall an der Stelle x *geteilt, und* ranges *mit dem* c *entsprechenden Teil des Intervalls überschrieben.*

Hierbei wird zunächst ein zusammenhängender Bereich der natürlichen Zahlen in zwei Bereiche partitioniert, der erste Bereich steht für die 0 und der zweite für die 1. Die Größe dieser Bereiche werden über die jeweilige Auftrittswahrscheinlichkeit bestimmt.

Es wird nun das nächste Bit aus dem Eingabestrom gelesen. Danach werden die Grenzen des Bereiches so überschrieben, dass nur noch das Stück

übrig bleibt, welches für den Wert des gelesenen Bit steht. Der neue Bereich wird nun wieder in zwei Teile geteilt und ein weiteres Bit gelesen, solange bis das Eingabewort fertig gelesen wurde.

Als Ausgabe erfolgt eine Zahl mit möglichst wenigen Ziffern, die in dem zuletzt übrig gebliebenen Bereich liegt. Führende Nullen müssen der Eindeutigkeit halber berücksichtigt werden. Niederwertige Stellen der Zahl, welche keinen Einfluss auf die Zugehörigkeit zum Bereich haben, müssen nicht explizit gespeichert werden, was kürzere Ausgaben ermöglicht. Dies lässt sich anhand von Abbildung 6 nachvollziehen.

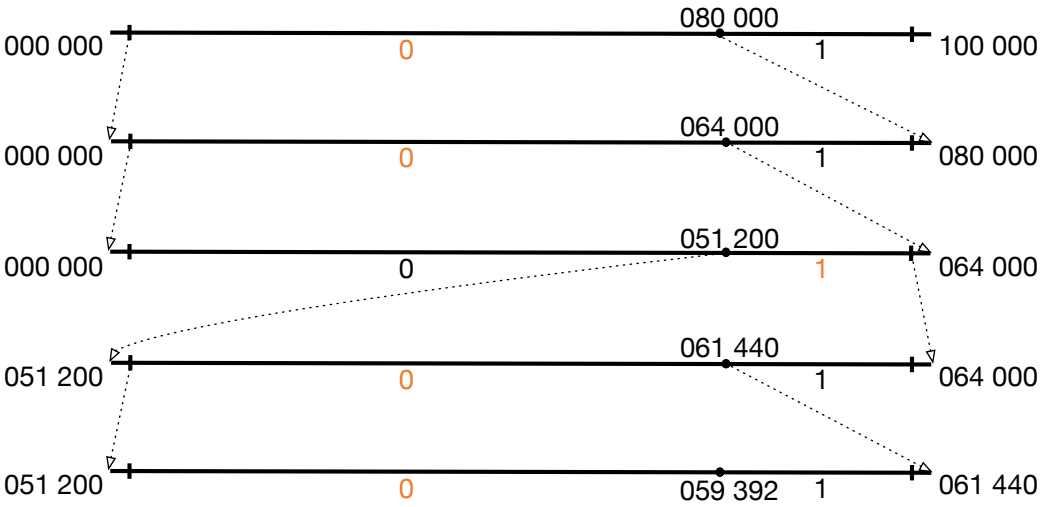

Abbildung 6: Beispiel für die Bereichskodierung
Angenommen wird ein Bereich von $[000\,000, 100\,000]$*, eine statische Wahrscheinlichkeitsverteilung von* 80% *Nullen und das Eingabewort* 00100*. Der resultierende Bereich wird eingegrenzt durch* 051200 *und* 059392*. Als eindeutige Ausgabe genügt das Wort* 052*, da alle sechsstelligen Zahlen, die mit* 052 *beginnen in diesem Bereich liegen.* $05x, x \in [3, 8]$ *sind alle anderen gültigen Ausgaben.*

In der Praxis wird eine dynamische Variante verwendet, bei der in jedem Schritt die Wahrscheinlichkeitsverteilung aktualisiert wird. Dies geschieht durch die Verwendung von Markov-Ketten, da sich, durch die zuletzt verwendete Ausgabesequenz, eine Aussage treffen lässt, welche Sequenzen wie wahrscheinlich als nächstes verwendet werden.

8 LZ4

Im Jahre 2011 veröffentlichte der Datenkompressionsexperte Yann Collet den LZ4-Algorithmus [8], der seinen Schwerpunkt auf hohe Kompressions- und Dekompressionsgeschwindigkeiten legt. Genau wie LZMA, ist auch LZ4 ein Derivat des LZ77 Konzepts, außerdem verwendet der Algorithmus, zur Treffersuche im Search Buffer, ebenfalls Hashtabellen. Allerdings hat LZ4 nur eine Output-Sequenz, die sich von denen von LZMA unterscheidet. Der Aufbau dieser Sequenz ist in Abbildung 7 zu sehen.

Das erste Feld besteht aus einem 1-Byte "Token", der wiederum in zwei 4-Bit Teile unterteilt ist. Der Inhalt des ersten Teils wird als die Anzahl der Bytes für die "Literallänge" interpretiert. Falls der höchste Zustand (15) erreicht ist, wird im nächsten Feld, der "Literallänge", ein Byte gelesen. Sollte dieses ebenfalls maximal (255) sein, wird ein weiteres gelesen und so fort. Die daraus resultierende Zahl bestimmt die Anzahl der "Literale", die byteweise aus dem Ursprungstext kopiert werden. Das vierte Feld der LZ4-Sequenz ist die "Distanz" zum Match, welche mit zwei Bytes im little Endian Format gespeichert wird. Das letzte Feld ist die "Trefferlänge". Um die Länge des Treffers zu bestimmen wird der zweite 4-Bit Teil des Tokens verwendet. Analog zum ersten Teil, werden zusätzliche Bytes bei "Trefferlänge" gespeichert, sollten diese benötigt werden.

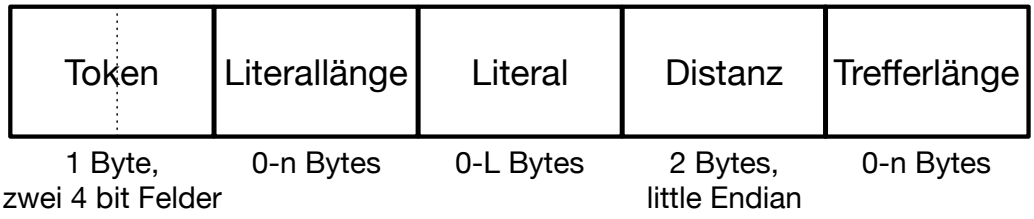

Abbildung 7: Aufbau der LZ4-Sequenz
Die Wortbreiten dieser Sequenz variieren von $3 - \infty$ *Bytes.*

LZ4 kommt außerdem in zwei Varianten: Zum einen dem LZ4 "Fast-Compression" (FC) Algorithmus, bei dem die Geschwindigkeit maximiert wird, zum anderen dem LZ4 "High-Compression" (HC) Algorithmus, der darauf ausgelegt ist, höhere Kompressionsraten zu erzielen. Zusätzlich gibt es die Möglichkeit, die Länge der Blöcke von vier Bytes auf bis zu 10 Bit zu reduzieren, um die Verwendung auf Systemen mit beschränktem Speicher zu ermöglichen.

8.1 LZ4 (FC)

Damit die Treffersuche entsprechend schnell ablaufen kann, werden während der Kompression, an der jeweils aktuellen Position, die nächsten vier Bytes der Eingabe in eine Hashtabelle gespeichert. Bei der schnellen Variante von LZ4 wird immer der erste Treffer in der Tabelle verwendet, sollte einer vorhanden sein.

8.2 LZ4 (HC)

Die andere Variante ist der LZ4 (HC) Algorithmus, bei dem in den Zellen der Hashtabelle immer der Eintrag gewählt wird, der einen möglichst kurzen Output erlaubt. Schließlich ist es stets möglich, dass ein früher gespeicherter Eintrag in der gleichen Zelle ein längeres Match ermöglicht. Dieser wäre dann vorzuziehen, da bei LZ4 die Distanz in konstanter Wortbreite von 2 Bytes gespeichert wird.

9 Zusammenfassung

In Tabelle 2 werden die Qualitäten der vorgestellten Algorithmen im Bezug auf Kompressionsgeschwindigkeit, Dekompressionsgeschwindigkeit sowie Kompressionsrate nebeneinander gestellt.

Algorithmus	**Kompressions-geschwindigkeit**	**Dekompressions-geschwindigkeit**	**Kompressionsrate**
LZ77	0.13 mb/s	0.5 mb/s	$1,21:1$
LZ78	0.006 mb/s	0.5 mb/s	$1,88:1$
LZW	2.4 mb/s	2.2 mb/s	$1,42:1$
LZMA	17 mb/s	45 mb/s	**2,38:1**
LZ4 (HC)	22 mb/s	**2 392 mb/s**	$1,79:1$
LZ4 (FC)	**449 mb/s**	2 176 mb/s	$1,54:1$

Tabelle 2: Vergleich der Algorithmen
Die fett gedruckten Einträge markieren, in der jeweiligen Kategorie, den jeweils am besten abschneidenden Algorithmus. Die verwendete Implementation von LZW stammen von E. Futch [9]. LZMA und LZ4 stammen von P. Skibinski [10]. Für LZ77 und LZ78 wurden eigene Implementation der von Lempel und Ziv vorgestellten Konzepte verwendet.

Ersichtlich wird hier vor allem, dass es keinen Algorithmus gibt, der den anderen in allen Kategorien überlegen ist, weswegen für verschiedene Zwecke verschiedene Algorithmen eingesetzt werden.

Ein Algorithmus wie LZ4 (FC) mit einer besonders hohen Kompressionsgeschwindigkeit eignet sich daher beispielsweise für Echtzeitübertragung oder für Dateien die häufig komprimiert und wieder dekomprimiert werden müssen. Währenddessen sind Algorithmen mit hohen Kompressionsraten wie LZMA gut geeignet, wenn es um langfristige Speicherung respektive Archivierung von Dateien geht. Hierbei wird eine höhere Verarbeitungsdauer bei Kompression und Dekompression in Kauf genommen, um den benötigten Speicherplatz weiter zu drücken, da dieser bei der Archivierung in der Regel die begrenztere Ressource darstellt.

Erwartungsgemäß wird es auch in Zukunft keinen Algorithmus geben der in allen Kategorien übertrifft, da zum Beispiel die Steigerung der Kompressionsrate mit erhöhtem Ver- und Entpackungsaufwand einhergeht.

Literatur

[1] Claude Elwood Shannon. „A mathematical theory of communication“. In: *Bell system technical journal* 27.3 (1948), S. 379–423.

[2] Ming Li und Paul Vitányi. *An introduction to Kolmogorov complexity and its applications*. Springer Science & Business Media, 2013.

[3] J. Ziv und A. Lempel. „A universal algorithm for sequential data compression“. In: *IEEE Transactions on Information Theory* 23.3 (Mai 1977), S. 337–343. ISSN: 0018-9448. DOI: 10.1109/TIT.1977.1055714.

[4] J. Ziv und A. Lempel. „Compression of individual sequences via variable-rate coding“. In: *IEEE Transactions on Information Theory* 24.5 (Sep. 1978), S. 530–536. ISSN: 0018-9448. DOI: 10.1109/TIT.1978.1055934.

[5] Terry A. Welch. „Technique for high-performance data compression“. In: *Computer* 52 (1984).

[6] Igor Pavlov. *LZMA SDK*. 1998. URL: http://www.7-zip.de/sdk.html (besucht am 20. 01. 2019).

[7] GNN MARTIN. „Range encoding: An algorithm for removing redundancy from a digitised message“. In: *Proc. IERE Video & Data Recording Conf., 1979*. 1979.

[8] Yann Collet. *RealTime Data Compression*. 2011. URL: http://fastcompression.blogspot.com/2011/05/lz4-explained.html (besucht am 20. 01. 2019).

[9] Egdares Futch. *github lzw*. 2013. URL: https://github.com/efutch/lzw (besucht am 20. 01. 2019).

[10] Przemyslaw Skibinski. *github lzbench*. 2015. URL: https://github.com/inikep/lzbench (besucht am 20. 01. 2019).