Verlustfreie Datenkompression: Deflate Und Bzip2

Seminar Effiziente Programmierung

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

Vorgelegt von:	Lennart Uhrmacher
Studiengang:	Informatik
Matrikelnummer:	6948328
Betreuer:	Kira Duwe

Hamburg, 09. März 2019

1. Einleitung

Während die Rechenleistung von Prozessoren weiterhin gemäß Moore's Gesetz wächst, ist die Entwicklung von Speichermedien seit Jahren abgeflacht. Zudem wächst in vielen Bereichen auch die Menge der zu verarbeitenden Daten. Aus diesen Gründen werden Speichergeräte immer mehr zum Bottleneck für moderne Programme. Eine Lösung, um diesem Bottleneck entgegenzuwirken, ist Datenkompression.

Datenkompression kann verlustbehaftet oder verlustfrei erfolgen. Während verlustbehaftete Kompression sich für Daten wie Bilder oder Videos eignet, wird verlustfreie Kompression für Daten genutzt, bei denen sich kein einziges Bit durch die Kompression verändern darf, wie z.B. Programmcode. In dieser Arbeit geht es um verschiedene Algorithmen zur verlustfreien Datenkompression.

Bei der Datenkompression spielt immer die Entropie der Daten eine Rolle. Die Entropie gibt den Informationsgehalt einer Nachricht an. Eine Nachricht, die in den gesamten Daten nur einmal auftritt und einzigartig ist, hat beispielsweise eine sehr hohe Entropie, während eine Nachricht, die mehrfach vorkommt, eine geringere Entropie hat. Bei Abschnitten mit hohen Entropien lässt sich wenig komprimieren, da der Informationsgehalt sehr hoch ist und wenig weggelassen werden kann. Ein hohes Ersparnis kann an den Stellen erzielt werden, die eine niedrige Entropie haben. In dieser Arbeit werden zwei verschiedene Techniken zur Datenkompression vorgestellt, die niedrige Entropie ausnutzen.

Die erste grundsätzliche Technik der Datenkompression ist das Ausnutzen der Redundanzen von Daten. Treten einzelne Teile der Daten mehrfach auf, so müssen diese nicht mehrfach gespeichert werden. Hierzu gibt es verschiedene Ansätze, von denen einige vorgestellt werden. Die zweite Methode ist das Überführen der Daten in eine kürzere Repräsentation. Häufig sind Daten z.B. mit mehr Bits gespeichert, als notwendig. Dies wird in Form der Huffman-Kodierung vorgestellt.

2. Deflate

Deflate ist ein Algorithmus zur verlustfreien Datenkompression. Er wurde von Phil Katz für das zip-Dateiformat entwickelt und ist hardwareunabhängig und patentfrei [Def19]. Mittlerweile gibt es mehrere Implementationen des Deflate-Algorithmus, wie z.B. zip, gzip und Zopfli.

2.1. Der Deflate-Algorithmus

Der Deflate-Algorithmus besteht aus drei verschiedenen Schritten [Deu96]:

1. Die Datei wird in Blocks aufgeteilt (häufig 32 KB)
2. Anwendung des LZ77-Algorithmus auf die einzelnen Blocks
3. Huffman-Kodierung der einzelnen Blocks

Der LZ77-Algorithmus sowie die Huffman-Kodierung werden im Folgenden ausführlich erklärt.

2.2. LZ-77

LZ77 (Lempel-Ziv 77) ist ein Verfahren zur verlustfreien Datenkompression. Es wurde 1977 von Abraham Lempel und Jacob Ziv veröffentlicht. Es macht sich das mehrfache Vorkommen von Wortsequenzen zu Nutze. Bei Wiederholungen wird lediglich eine Referenz auf die Position gespeichert, an der die Wortsequenz zum ersten Mal aufgetreten ist [ZL77].

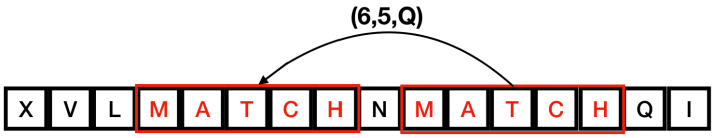

Abb. 1: LZ77-Verfahren

2.3. Huffman-Kodierung

Die Ausführungen in Kapitel 2.3 und 2.4 basieren auf [Ble01].

Die Huffman-Kodierung ist eine Entropiekodierung, die üblicherweise für verlustfreie Datenkompression genutzt wird. Entwickelt wurde sie 1952 von David Huffman. Sie ist ein optimaler Präfix-Code, das heißt, durch die Huffman-Kodierung eine Nachricht optimal verkleinert werden kann.

2.4. Algorithmus zur optimalen Huffman-Kodierung

Im Folgenden wird ein Algorithmus zur optimalen Huffman-Kodierung vorgestellt. Er lässt sich mit folgenden Schritten beschreiben:

1. Erstelle einen Wald mit Bäumen für jedes Zeichen. Jeder Baum enthält nur einen Knoten.

2. Kombiniere die beiden Bäume mit der geringsten Häufigkeit zu einem neuen Baum.

3. Wiederholen, bis nur noch ein Baum übrig ist.

In den folgenden Abbildungen wird beispielhaft die Ausführung des Algorithmus an der Wortsequenz „AAAAAABBCD“ dargestellt.

1. Für jedes Zeichen im Alphabet wird ein Baum mit einem einzigen Knoten erstellt. Dieser Knoten enthält das Zeichen und dessen Häufigkeit.

Abb. 2: Huffman-Kodierung (1)

2. Die beiden Bäume mit den geringsten Häufigkeiten werden zu einem neuen Baum verknüpft.

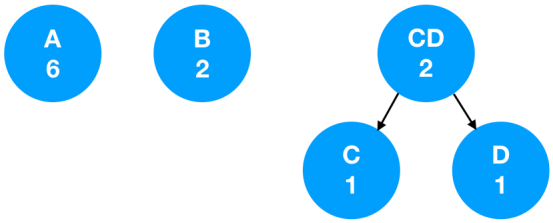

Abb. 3: Huffman-Kodierung (2)

3. Dies wird wiederholt, bis nur noch ein Baum existiert. Dieser Baum gibt nun die optimale Huffman-Kodierung für die Wortsequenz an.

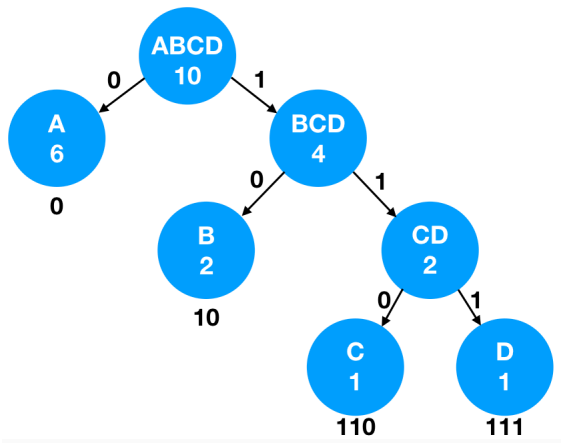

Abb. 4: Huffman-Kodierung (3)

Die optimale Huffman-Kodierung für das Wort „AAAAAABBCD“ lautet wie folgt:

A = 0, B = 10, C = 110, D = 111.

3. Verschiedene Kompressionsalgorithmen

Im Folgenden werden verschiedene Kompressionsalgorithmen/ Kompressionsprogramme vorgestellt und bewertet. Hierbei wird auf die einzelnen Kompressionsraten und die Kompressionsgeschwindigkeiten eingegangen.

Die Kompressionsrate gibt an, wie stark die Dateigröße durch die Kompression verkleinert wird. Wird eine Datei von 200 MB auf 100 MB komprimiert, so ist die Kompressionsrate 2. Eine hohe Kompressionsrate ist erstrebenswert.

Die Kompressionsgeschwindigkeit gibt an, wie schnell die Kompression abläuft. Das gleiche ist die Dekompressionsgeschwindigkeit für die Dekompression. Beide Geschwindigkeiten können stark voneinander abweichen.

Zip: Zip ist ein Dateiformat, welches 1989 von Phil Katz entwickelt wurde. Es gibt mehrere Algorithmen, die ins zip-Format komprimieren, wobei Deflate der Gebräuchlichste ist [zip19].

Gzip: Gzip ist ein Kompressionsprogramm, das Dateien mit dem Deflate-Algorithmus komprimiert. Es wurde 1992 von Jean-Loup Gailly und Mark Adler entwickelt. Es ist frei für alle Betriebssysteme verfügbar [gzi18].

Zopfli: Zopfli ist eine Implementation des Deflate-Algorithmus vom schweizerischem Google-Team aus 2012. Sie hat eine sehr hohe Kompressionsrate, wobei die Kompression jedoch sehr langsam ist (ca. 80-mal langsamer als Gzip). Die Dekompressionsgeschwindigkeit bleibt jedoch normal. Zopfli ist somit geeignet für Dateien, die einmalig komprimiert werden, und dann vielfach verteilt und dekomprimiert werden (z.B. Downloads von Websites) [zop18].

Gipfeli: Gipfeli ist ebenfalls ein Kompressionsalgorithmus von Google zur High-Speed-Kompression von Daten. Er nutzt das LZ77-Verfahren, aber im Gegensatz zu Brotli keine Huffman-Kodierung.

Brotli: Brotli ist eine weitere Entwicklung des schweizerischen Google-Teams aus 2014. Es kodiert nicht ins Deflate-Format, sondern in ein eigenes (.brotli). Der Brotli-Algorithmus nutzt jedoch auch das LZ77-Verfahren und die Huffman-Kodierung, ähnlich wie Deflate. Zusätzlich benutzt Brotli ein vordefiniertes Wörterbuch, bestehend aus ca. 13000 Wortsequenzen. Wie stark Brotli komprimiert, kann durch verschiedene Stufen von 1 bis 11 festgelegt werden. Mit steigender Stufe steigt die Kompressionsrate, während die Kompressionsgeschwindigkeit sinkt [Ala15]. In der folgenden Abbildung werden die Kompressionsraten und -geschwindigkeiten von Brotli mit dem normalen Deflate-Algorithmus verglichen.

Algorithmus	Kompressionsrate	Kompressionsgeschwindigkeit (MB/s)	Dekompressionsgeschwindigkeit (MB/s)
Brotli 1	3.381	**98.3**	334.0
Brotli 9	3.965	17.0	**354.5**
Brotli 11	**4.347**	0.5	289.5
Deflate 1	2.913	93.5	323.0
Deflate 9	3.371	15.5	347.3

Abb. 5: Performance von Brotli und Deflate,
Ausschnitt aus [AKSV15]

4. Bzip2

Bzip2 ist ein eigener Kompressionsalgorithmus. Er benutzt eine Kombination von verschiedenen Methoden, um die Datei zu komprimieren. Zuerst werden die Daten mit der Burrows-Wheeler-Transformation und der Move-To-Front-Transformation transformiert, wobei die Dateigröße sich noch nicht verändert. Danach wird die Datei mit der Run-Length-Kodierung und der Huffman-Kodierung komprimiert. Die beiden Transformationen und die Run-Length-Kodierung werden im Folgenden im Detail besprochen. Die Huffman-Kodierung wurde bereits in Kapitel 2 erklärt [EVKV02].

4.1. Burrows-Wheeler-Transformation

Die Ausführungen in diesem Kapitel basiert auf [BW94] und [bwt18].

Bei der Burrows-Wheeler-Transformation werden die einzelnen Buchstaben umsortiert und in eine neue Reihenfolge gebracht. In der neuen Sortierung besteht eine erhöhte Wahrscheinlichkeit, dass gleiche Buchstaben hintereinander auftreten.

4.1.1. Hintransformation

Der Ablauf ist folgendermaßen: Zuerst werden alle Rotationen der Eingabe gebildet (Schritt 1). Diese werden dann alphabetisch sortiert (Schritt 2). Das Ergebnis ergibt sich aus den jeweils letzten Buchstaben der sortierten Spalte (Schritt 3). Aus der Eingabe „ANANAS#“ wird somit ein „S#NNAAA“. Außerdem muss der Index der Spalte, in welcher das Eingabewort steht, ebenfalls gespeichert werden. Dies ist für die Rücktransformation notwendig. In diesem Beispiel ist der Index 1.

1. Rotieren	2. Sortieren	3. Ergebnis
ANANAS#	#ANANAS	S
NANAS#A	ANANAS#	#
ANAS#AN	ANAS#AN	N
NAS#ANA	AS#ANAN	N
AS#ANAN	NANAS#A	A
S#ANANA	NAS#ANA	A
#ANANAS	S#ANANA	A

Abb. 6: Burrows-Wheeler-Transformation (1)

4.1.2. Rücktransformation

Die Rücktransformation ist etwas komplizierter als die Hintransformation, da als Informationen nur das Ausgabewort (in diesem Beispiel „S#NNAAA“) und der Index (in diesem Beispiel 1) vorliegen. Zuerst kann die sortierte Spalte teilweise ausgefüllt werden, indem für jeden Eintrag der erste und letzte Buchstaben eingesetzt wird. Die Endbuchstaben ergeben sich aus dem Ausgabewort, die Anfangsbuchstaben ergeben sich, indem das Ausgabewort alphabetisch sortiert wird. Aus dem Index geht hervor, an welcher Stelle das vorherige Eingabewort stand (in diesem Fall an der Stelle 1, wo nun ein „A????#“ steht). Nun können nacheinander die einzelnen Fragezeichen aufgedeckt werden: Aus der alphabetischen Sortierung der Spalte ergibt sich, dass das „A“ am Anfang im Eintrag 1 dasselbe ist, wie das „A“ am Ende von Eintrag 5. In Eintrag 5 lässt sich nun ablesen, dass auf dieses „A“ ein „N“ folgt. So kann Schritt für Schritt das Eingabewort rekonstruiert werden, bis das „#“ erreicht ist.

1. Rotieren	2. Sortieren	3. Ergebnis
?	#?????S	S
?	A?????#	#
?	A?????N	N
?	A?????N	N
?	N?????A	A
?	N?????A	A
?	S?????A	A

Abb. 7: Burrows-Wheeler-Transformation (2)

4.2. Move-To-Front-Transformation

Die Ausführungen in diesem Kapitel basieren auf [BW94] und [mov15].

Bei der Move-To-Front-Transformation wird ebenfalls eine Permutation der Eingabe erzeugt. Hierbei resultieren mehrfach nacheinander auftretende Zeichen in der Eingabe in vielen Nullen in der Ausgabe. Dies ist der Grund, weshalb die Move-To-Front-Transformation nach der Burrows-Wheeler-Transformation angewendet wird - zuerst wird die Wortsequenz umsortiert, um gleiche Zeichen hintereinander zu erhalten, welche dann durch Nullen ersetzt werden.

4.2.1. Hintransformation

Zuerst wird das Alphabet der Wortsequenz erzeugt (in diesem Beispiel „#ANS“). Die Transformation läuft nun folgendermaßen ab: Es werden alle Zeichen der Eingabe durchgegangen. Für jedes Zeichen wird überprüft, an welcher Stelle das Zeichen im Alphabet steht. Diese Position ist der Ausgabewert für das Zeichen. Danach wird das Zeichen im Alphabet an den Anfang bewegt. Für das Beispiel „S#NNAAA“ ergibt sich die Ausgabe „3130300“. Zusätzlich muss das Alphabet nach dem letzten Schritt gespeichert werden, damit die Datei wieder dekodiert werden kann (in diesem Beispiel „AN#S“).

Input	Alphabet	Output	Alphabet’
S	#ANS	3	S#AN
#	S#AN	1	#SAN
N	#SAN	3	N#SA
N	N#SA	0	N#SA
A	N#SA	3	AN#S
A	AN#S	0	AN#S
A	AN#S	0	AN#S

Abb. 8: Move-To-Front-Transformation

4.2.2. Rücktransformation

Bei der Rücktransformation muss lediglich die Tabelle von unten aufgebaut werden. Das letzte Zeichen der Eingabe steht am Anfang des Ausgabealphabets „AN#S“ und ist somit ein „A“. Dann wird das letzte Zeichen der Ausgabe (die 0) gelesen, welche angibt, dass das „A“ im Alphabet an die Position 0 bewegt werden muss. Nun wird wieder der Anfangsbuchstabe des Alphabets gelesen, und das Ganze wiederholt, bis das komplette Eingabewort generiert wurde.

4.3. Run-Length-Kodierung

Die Ausführungen in diesem Kapitel basieren auf [Ble01].

Bei der Run-Length-Kodierung werden gleiche aufeinanderfolgende Zeichen abgekürzt. Anstatt ein Zeichen mehrfach aufzuschreiben, wird es nur einmal aufgeschrieben, und

dahinter eine Zahl, die die Häufigkeit des Zeichens angibt. Dadurch werden die Sequenzen aus Nullen, die in den beiden vorherigen Schritten generiert wurden, stark komprimiert.

4.3.1. Kodierung

Die Run-Length-Kodierung ist relativ simpel. Die Eingabe wird in Blocks aufgeteilt, in denen nur ein Zeichen vorkommt, und diese werden dann mit der Kodierung zusammengefasst. Eine Eingabe „300010003000000“ wird somit zu einer „31 03 11 03 31 06“.

4.3.2. Dekodierung

Die Dekodierung ist ähnlich wie die Kodierung sehr simpel. Es wird einfach jedes zweite Zeichen der Ausgabe als eine Häufigkeitsangabe des vorherigen Zeichens interpretiert. Somit wird „31 03 11 03 31 06“ zu „3 000 1 000 3 000000“.

5. Zusammenfassung

In Zeiten, in denen immer mehr Daten gespeichert werden, ist Datenkompression nicht wegzudenken. Zwei Methoden zur verlustlosen Datenkompression wurden in dieser Arbeit vorgestellt: der weit verbreitete Kompressionsalgorithmus Deflate und der Kompressionsalgorithmus bzip2. Anhand von ihnen wurden verschiedene Techniken erläutert, die zu einer Verkleinerung der Dateigröße führen, wie zum Beispiel die Huffman-Kodierung.

Außerdem wurde eine Handvoll weiterer Kompressionsalgorithmen vorgestellt, wobei deutlich wurde, dass es verschiedene Anwendungsbereiche für verschiedene Kompressionsalgorithmen gibt. So eignen sich einige wie z.B. Gipfeli durch hohe Kompressionsgeschwindigkeiten zur schnellen Kompression „On-the-fly“, während andere wie z.B. bzip2 oder Zopfli sich durch hohe Kompressionsraten bei langsamerer Kompressionsrate zur Archivierung eignen.

Dem Leser wurde somit ein grundlegender Einblick in Kompressionsalgorithmen und deren Funktionsweise gegeben.

6. Literaturverzeichnis

[AKVS15] Jyrki Alakuijala, Evgenii Kliuchnikov, Zoltan Szabadka, and Lode Vandevenne. Comparison of Brotli, Deflate, Zopfli, LZMA, LZHAM and Bzip2 Compression Algorithms. Technical report, Google, Inc., September 2015.

[Ala12] Jyrki Alakuijala. Brotli Compressed Data Format. https://datatracker. ietf.org/doc/rfc7932/, 2015. ((letzter Zugriff: 05-01-2018).

[Ble01] Guy E Blelloch. Introduction to data compression. Computer Science Department, Carnegie Mellon University, 2001.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical report, 1994.

[bwt18] Burrows-Wheeler-Transformation. https://de.wikipedia.org/wiki/Burrows-Wheeler-Transformation, 2018. (letzter Zugriff: 06-03-2019).

[def19] Deflate. https://de.wikipedia.org/wiki/Deflate, 2019. (letzter Zugriff: 06-03-2019).

[gzi18] gzip. https://de.wikipedia.org/wiki/Gzip, 2018. (letzter Zugriff: 06-03-2019).

[mov15] Move to front. https://de.wikipedia.org/wiki/Move_to_front, 2015. (letzter Zugriff: 06-03-2019).

[zip19] ZIP-Dateiformat. https://de.wikipedia.org/wiki/ZIP-Dateiformat, 2019. (letzter Zugriff: 06-03-2019).

[zop18] Zopfli. https://de.wikipedia.org/wiki/Zopfli, 2019. (letzter Zugriff: 06-03-2019).

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. IEEE Transactions on information theory, 23(3):337-343, 1977.