Verlustlose Datenkompression auf Grundlage der Burrows-Wheeler-Transformation•

JÜRGEN ABEL ••
Universität Duisburg-Essen

Die verlustfreie Datenkompression auf Grundlage der Burrows-Wheeler-Transformation ist in den letzten Jahren aufgrund Ihrer Effizienz und bemerkenswerten Eleganz auf ein großes Interesse gestoßen. Selbst einfache Implementierungen erreichen Geschwindigkeiten und Kompressionsraten von bekannten kommerziellen Programmen. Die Burrows-Wheeler-Transformation basiert auf einer Permutation von Eingabedaten, durch welche Zeichen mit ähnlichem Kontext nahe beieinander angeordnet werden. Im vorliegenden Artikel wird das Verfahren der Transformation sowie der Rücktransformation erläutert und zusammen mit einer typischen Implementierung vorgestellt.

1. EINFÜHRUNG

Bei der Datenkompression wird eine Eingangsdatei E, bestehend aus einer Menge von diskreten Zeichen, auf eine Ausgangsdatei A, ebenfalls aus einer Menge von diskreten Zeichen bestehend, abgebildet. Existiert für die Dekompression eine Rücktransformation, welche A wieder auf E abbildet, spricht man von verlustfreier Datenkompression. Als Datenkompressionsrate bezeichnet man den Quotienten der Länge von A zu der Länge von E. Die Datenkompression verfolgt hierbei das Ziel, die Datenkompressionsrate zu minimieren und die Verarbeitungsgeschwindigkeit der Kompression und Dekompression zu maximieren. Seit Ihrer Vorstellung im Jahre 1994 [1] haben verlustfreie Datenkompressionsalgorithmen, welche auf der sogenannten Burrows-Wheeler-Transformation (BWT) aufbauen, einen ständigen Fortschritt erfahren. Viele Autoren haben mit neuen Teilalgorithmen zu immer besseren Kompressionsraten beigetragen [2-10]. Inzwischen gehören BWT-Implementierungen zu den schnellsten und stärksten Datenkompressionsprogrammen weltweit. Im folgenden werden die Grundlagen dargelegt, auf der diese Implementierungen basieren, und anhand eines einfachen Datenbeispieles dem Leser nähergebracht. Neben der BWT werden auch andere Stufen des Kompressionsalgorithmus kurz vorgestellt, welche für eine komplette Implementierung notwendig sind. Danach werden die Ergebnisse anhand des Calgary-Corpus [11] mit anderen Datenkompressionsprogrammen verglichen.

• Vorabdruck vom 12.03.2003 eines Artikels für die Zeitschrift "PIK - Praxis der Informationsverarbeitung und Kommunikation", Herausgeber: Prof. Dr. Hans Werner Meuer, Prof. Dr. Otto Spaniol, Fachgruppe Kommunikation und Verteilte Systeme der Gesellschaft für Informatik eV.
•• Email: juergen.abel@acm.org; Adresse: Universität Duisburg-Essen, Fachgebiet "Nachrichtentechnische Systeme", Fakultät Ingenieurwissenschaften, Bismarckstraße 81, D-47057 Duisburg

2. DER BURROWS-WHEELER-DATENKOMPRESSIONSALGORITHMUS

2.1 Aufbau des Algorithmus

Ein Burrows-Wheeler-Datenkompressionsalgorithmus (BWCA) ist in seiner Gesamtheit in mehrere Stufen gegliedert [1][12]. Diese Stufen sind einzelne Transformationen und werden nacheinander durchlaufen, wobei die Daten jeweils in Blöcken der Größe N verarbeitet werden. Der BWCA wird daher auch als Blocksortierungsalgorithmus bezeichnet. Der Eingang der ersten Stufe besteht aus der Eingangsdatei E, und der Ausgang der letzten Stufe ist die Ausgangsdatei A. Ist die Größe von E kleiner als N, wird als Blockgröße einfach die Größe von E benutzt. Ist die Größe von E größer als N, wird E in mehrere Blöcke der Größe N zerlegt, wobei der letzte Block in der Regel kleiner als N ist. Die Blöcke werden dann einzeln verarbeitet. Bei der Dekompression werden die Stufen des Algorithmus mit den jeweiligen inversen Transformationen in rückwärtiger Richtung durchlaufen. Je nach Implementierung können sich die Stufen unterscheiden. Abbildung 1 zeigt den Aufbau eines typischen Datenkompressionsalgorithmus auf Basis der BWT. Wie man in der Abbildung erkennen kann, ist die BWT die erste Stufe des gesamten Algorithmus. Ihre Aufgabe ist es, Zeichen mit ähnlichem Kontext nahe zusammen zu bringen.

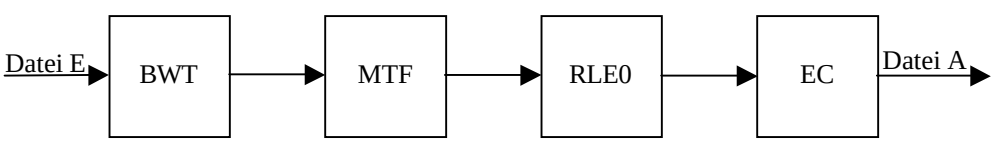

Abbildung 1: Schema des Burrows-Wheeler-Datenkompressionsalgorithmus

Danach erfolgt eine sogenannte Move-To-Front-Transformation (MTF). Diese Transformation wandelt den lokalen Kontext, welchen die Zeichen nach der BWT haben, in einen globalen Kontext um [3][7]. Hierbei entstehen lange Folgen von Nullen, die in der dritten Stufe, einer Lauflängenkodierung für Nullfolgen (RLE0), speziell behandelt werden. Als vierte und letzte Stufe folgt ein Entropiekodierer (EC). Seine Aufgabe besteht darin, die einzelnen Zeichen in eine möglichst kurze Bitfolge umzuwandeln. Nachfolgend werden die einzelnen Stufen näher beschrieben.

Jede Stufe hat ein Eingabefeld und ein Ausgabefeld der Blockgröße N. Die Größenordnung von N liegt normalerweise in einem Bereich von einem Megabyte. Da die Ausgabedaten der RLE-0, die gleichzeitig die Eingabedaten der EC bilden, breiter als 8 Bit sein können, benutzt man als Breite einer Feldkomponente zweckmäßigerweise 32

Bit, wobei die Ein- und Ausgabedaten von BWT und MTF jeweils nur 8 Bit davon nutzen.

2.2 Die Burrows-Wheeler-Transformation

Die BWT ist das Herzstück des BWCA und schafft die Voraussetzung für eine starke Kompression. Die Transformation wurde von Burrows und Wheeler in ihrem Forschungsbericht „A Blocksorting Lossless Data Compression Algorithm“ im Jahre 1994 veröffentlicht [1] und basiert auf einer bis dahin nicht veröffentlichten Arbeit von Wheeler aus dem Jahre 1983 [12]. Die Transformation führt eine Umsortierung der Eingabedaten auf dem gesamten Eingabefeld durch und ist nicht sequentiell. Bei der Umsortierung handelt es sich um eine besondere Sortierung der Daten, welche Zeichen mit ähnlichem Kontext nahe beieinander anordnet und eine spätere Kompression unterstützt. Durch die Transformation bleibt die Anzahl der Zeichen und ihre Häufigkeiten vor und nach der Transformation konstant. Es handelt sich also nicht um eine Kompression, sondern um eine Permutation der Daten, welche die Eingabezeichen lediglich in einer anderen Reihenfolge anordnet. Die Kompression findet erst in späteren Stufen statt. Zu der Transformation existiert eine inverse Rücktransformation, durch welche die ursprüngliche Reichenfolge der Zeichen wiederhergestellt wird. Die Rücktransformation ist wesentlich schneller als die Hintransformation, da die Daten mit linearem Zeitaufwand rekonstruiert werden können [13-15].

Um die generelle Funktionsweise der Hin- und Rücktransformation zu verdeutlichen, wird im folgenden die Zeichenkette *BANANAS* als Eingabe angenommen und entsprechend transformiert. Nummerierungen und Indizierungen beginnen hierbei - wie in der Informatik üblich - bei 0. Zunächst erfolgt die Beschreibung der Hintransformation. Hierzu wird die Zeichenkette so oft untereinander in jeweils einer Zeile angeordnet, wie sie Zeichen hat, und dabei um jeweils ein Zeichen nach rechts rotiert. Die Zeilen werden als Ringpuffer angesehen, so daß Zeichen, welche rechts aus der Zeile hinauslaufen, links wieder hineingeschoben werden. In Zeile 0 steht die ursprüngliche Zeichenkette. Das Ergebnis ist in Abbildung 2 ersichtlich.

Im zweiten Schritt werden die Zeilen lexikographisch von oben nach unten aufsteigend sortiert. Abbildung 3 zeigt das Ergebnis der Sortierung. Wie man sieht, ist die ursprüngliche Zeichenkette mit Index 0 durch die Sortierung in Zeile 3 verschoben worden und die erste Spalte, F-Spalte genannt, komplett alphabetisch geordnet. Das Ergebnis der BWT besteht nun aus der letzten Spalte, welche als L-Spalte bezeichnet wird, sowie dem Index der Originalzeichenkette, nachfolgend als Ursprungsindex bezeichnet. Die Spalte L hat hierbei nach der Sortierung eine bemerkenswerte Eigenschaft. Sie ist zwar nicht alphabetisch geordnet, jedoch ist jedes Zeichen in Spalte L aufgrund des Ringpuffers das

Vorgängerzeichen zu dem Zeichen in Spalte F. Damit stellen die Zeichen in Spalte L den Kontext zu den Zeichen in Spalte F dar.

Index	F-Spalte						L-Spalte
0	*B*	*A*	*N*	*A*	*N*	*A*	*S*
1	*S*	*B*	*A*	*N*	*A*	*N*	*A*
2	*A*	*S*	*B*	*A*	*N*	*A*	*N*
3	*N*	*A*	*S*	*B*	*A*	*N*	*A*
4	*A*	*N*	*A*	*S*	*B*	*A*	*N*
5	*N*	*A*	*N*	*A*	*S*	*B*	*A*
6	*A*	*N*	*A*	*N*	*A*	*S*	*B*

Abbildung 2: Rotation der Zeichenketten

Index	F-Spalte						L-Spalte
6	*A*	*N*	*A*	*N*	*A*	*S*	*B*
4	*A*	*N*	*A*	*S*	*B*	*A*	*N*
2	*A*	*S*	*B*	*A*	*N*	*A*	*N*
0	*B*	*A*	*N*	*A*	*N*	*A*	*S*
5	*N*	*A*	*N*	*A*	*S*	*B*	*A*
3	*N*	*A*	*S*	*B*	*A*	*N*	*A*
1	*S*	*B*	*A*	*N*	*A*	*N*	*A*

Abbildung 3: Sortierung der Zeichenketten

Da die Spalte F komplett sortiert ist, stehen Zeichen mit ähnlichem Folgekontext in Spalte L nahe beieinander [3][7]. Die Spalte L läßt sich im allgemeinen wesentlich besser komprimieren als die ursprüngliche Zeichenkette. In Falle von *BANANAS* ist das Ergebnis somit die Zeichenkette *BNNSAAA* und die Zahl 3. Abbildung 4 zeigt das Ergebnis der Sortierung, welches für den Vergleich mit späteren Stufen hexadezimal dargestellt ist.

Eingabe (hexadezimal)	Ausgabe (hexadezimal)
42 41 4E 41 4E 41 53	42 4E 4E 53 41 41 41

Abbildung 4: *BANANAS*-Beispiel der BWT-Sortierung

2.3 Die Rücktransformation

Eines der erstaunlichsten Merkmale der BWT ist die Eigenschaft der Umkehrbarkeit. Für einen Außenstehenden ist es zunächst geradezu rätselhaft, wie man nur aufgrund der Spalte L und des Ursprungsindex die ursprüngliche Zeichenkette wiederherstellen kann. Zunächst wird hierzu die Spalte F rekonstruiert. Die Spalte F erhält man, indem man die Zeichen der Spalte L alphabetisch sortiert. Weiterhin ist die Spalte L aufgrund der Konstruktion die Vorgängerspalte zu F. Mit Hilfe einer einfachen Regel lassen sich nun die ursprünglichen Zeichen aus den beiden Spalten wieder herstellen, ähnlich dem Durchschreiten einer verketteten Liste. Hierbei werden jeweils die Buchstaben ausgegeben, welche in Spalte F stehen.

Zunächst beginnt man mit dem Ursprungsindex. Das Zeichen in Spalte F, auf welches der Ursprungsindex verweist, ist das erste Zeichen der ursprünglichen Zeichenkette. Es handelt sich um das Zeichen *B*, wie in Abbildung 5 dargestellt.

	Zeile	L-Spalte	F-Spalte
	0	B	A
	1	N	A
	2	N	A
Ursprungsindex →	**3**	S	**B**
	4	A	N
	5	A	N
	6	A	S

Abbildung 5: Wiederherstellen des ersten Buchstabens

Von dem Zeichen *B* der Spalte F geht man zu demjenigen *B* der Spalte L, welches innerhalb des Buchstabens *B* an der gleichen Stelle steht. Da es vom Buchstaben *B* in dem vorliegenden Beispiel nur einen gibt, muß es sich um das *B* in Zeile 0 handeln. Es wird der zugehörige Buchstabe in Spalte F, hier das *A* ausgegeben. Da es sich um das erste *A* in Spalte F handelt, geht man zum ersten *A* in Spalte L. Dieses befindet sich in Zeile 4. Das zugehörige Zeichen in Spalte F ist das *N* und wird ausgegeben. Es handelt sich um das erste *N* in Spalte F, daher geht man zum ersten *N* in Spalte L und so weiter. Der gesamte Prozeß ist in Abbildung 6 wiedergegeben.

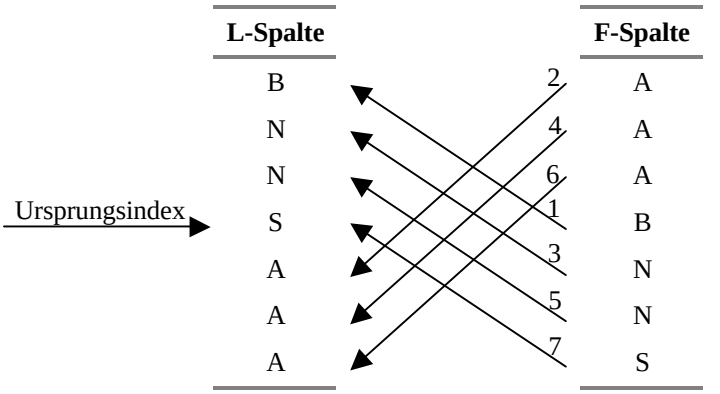

Abbildung 6: Wiederherstellen aller Zeichen

Das Verfahren endet, wenn man auf dieselbe Zeile stößt, auf welche der Ursprungsindex verweist. Abbildung 7 zeigt den Algorithmus noch einmal in Kurzform.

1. Stelle Spalte F durch Sortierung der Spalte L her
2. Gehe zu derjenigen Reihe, auf welche der Ursprungsindex verweist
3. Gib das zugehörige Zeichen in Spalte F aus
4. Gehe zu dem Zeichen in Spalte L, welches innerhalb desselben Buchstaben des letzten ausgegebenen Zeichens an der gleichen Stelle in Spalte L steht
5. Gehe solange zu Punkt 3, bis die aktuelle Zeilennummer gleich dem Ursprungsindex ist

Abbildung 7: Algorithmus zur Rücktransformation

2.4 Die Move-To-Front-Transformation

Bereits die Ausgabedaten der BWT lassen sich mit Hilfe eines Entropiekodierers recht gut komprimieren. Man kann das Ergebnis jedoch mit Hilfe weiterer Transformationen noch deutlich verbessern. Hierzu dient die folgende MTF-Stufe [16][17]. Auch bei der MTF bleibt die Anzahl der Zeichen konstant, obgleich es sich nicht um eine Permutation der Zeichen handelt, sondern um eine Abbildung, bei welcher sich die Häufigkeiten der einzelnen Buchstaben ändern. Aufgabe der MTF ist es, den lokalen Kontext der Zeichen nach der BWT in einen globalen Kontext umzuwandeln. Hierzu wird eine Liste M der Zeichen von 0 bis 255 verwaltet. Am Anfang ist die Liste aufsteigend von 0 an sortiert. Danach wird für jedes Zeichen des Eingabefeldes der aktuelle Index innerhalb von M ausgegeben. Das Zeichen wird dann von seiner alten Stelle an den Anfang von M ver-

setzt, also auf Index 0, und M entsprechend aufgeschoben. Als Ergebnis sind häufige Zeichen weiter vorne in der Liste anzutreffen. Diese Zeichen erhalten einen kleinen Index. Seltenere Zeichen befinden sich weiter hinten in der Liste und erhalten jeweils einen größeren Index. In Abbildung 8 ist die MTF-Transformation der Zeichenkette *BNNSAAA* dargestellt.

Eingabe (hexadezimal)	**Ausgabe (hexadezimal)**
42 4E 4E 53 41 41 41	42 4E 00 53 44 00 00

Abbildung 8: *BANANAS*-Beispiel der MTF-Transformation

Wie man erkennen kann, werden Folgen von gleichen Zeichen durch eine Folge von Nullen wiedergegeben, unabhängig von dem Zeichen selbst. Da die Ausgabe der BWT sehr häufig Folgen gleicher Zeichen beinhaltet, hat die Ausgabe der MTF eine große Anzahl von Nullfolgen. Diese Nullfolgen werden in der nächsten Stufe weiterverarbeitet. In Abbildung 9 wird noch einmal der MTF-Algorithmus in Kurzform beschrieben.

```
∀ 0 <= i <= 255:   Setze M[i] := i
Für alle Zeichen c des Eingabefeldes:
    Gib Index i von M aus, auf dem sich c gerade befindet
    Setzte c auf Index 0 von M und verschiebe M[0 .. i-1] nach M[1 .. i]
```

Abbildung 9: MTF-Algorithmus

2.5 Die Lauflängenkodierung für Nullfolgen

Eine Optimierung des BWCA gegenüber der ursprünglichen Implementierung von Burrows und Wheeler stellt die RLE0-Stufe dar. Diese Lauflängenkodierung stammt von Wheeler und wurde von Fenwick in seinem BWT-Abschlußbericht beschrieben [16]. Auch Balkenhol und Kurtz greifen das Thema Lauflängenkodierung für Nullfolgen auf [3]. Ansatzpunkt des Verfahrens ist die Beobachtung, daß praktisch alle Folgen von gleichen Zeichen hinter der MTF-Stufe aus Nullen bestehen, Folgen anderer Zeichen tauchen normalerweise an dieser Stelle nicht mehr auf. Aus diesem Grunde brauchen nur Folgen von Nullen kodiert zu werden, so daß man die Angabe spart, aus welchem Zeichen die Folge besteht. Der RLE0-Algorithmus, in Abbildung 10 zu erkennen, ist sowohl einfach als auch sehr effizient. Die Länge von Nullfolgen wird um 1 erhöht und als Sequenz von

0 und 1 ohne das höchstwertige Bit abgespeichert, ähnlich dem Binärteil des Elias-Codes [18].

```
Für alle Zeichen c des Eingabefeldes:
  IF C > 0 THEN
    Gib C+1 aus
  ELSE
    Ermittle Lauflänge L der Nullen
    Gib die Zahl (L +1) ohne das höchstwertige Bit als Folge von 0 und 1 aus
```

Abbildung 10: RLE0-Algorithmus

Da alle Zeichen größer als 0 um 1 erhöht worden sind, ist eine eindeutige Darstellung gewährleistet, und die Zeichen 0 und 1 stellen bei der Rücktransformation dieser Stufe immer eine Kodierung von Nullfolgen dar. Durch die RLE0-Stufe wird die Datenmenge beträchtlich verkleinert, welche der Entropiekodierer verarbeiten muß. Dies führt sowohl zu einer Beschleunigung des Gesamtverfahrens, da der Entropiekodierer deutlich langsamer als die RLE0-Stufe arbeitet, als auch zu einer besseren Kompressionsrate. In Abbildung 11 sind RLE0-Beispiele für verschiedene Lauflängen aufgeführt.

Lauflänge	**RLE0-Kodierung**
1	0
2	1
3	00
4	01
5	10
6	11
7	000
8	001
9	010
10	011

Abbildung 11: Beispiele von RLE0-Kodierungen

In Abbildung 12 ist die RLE0-Transformation der MTF-Ausgabe des *BANANAS*-Beispieles zu erkennen.

Eingabe (hexadezimal)	Ausgabe (hexadezimal)
42 4E 00 53 44 00 00	43 4F 00 54 45 01

Abbildung 12: *BANANAS*-Beispiel der RLE0-Transformation

2.6 Der Entropiekodierer

Der Entropiekodierer hat die Aufgabe, die Ausgabe der Daten nach der RLE0-Transformationen durch eine möglichst kurze Bitfolge wiederzugeben. Je nach Implementierung werden hierbei sowohl Huffmankodierungen [1][19] als auch arithmetische Kodierungen eingesetzt [3][7][16]. Huffmankodierungen sind zwar deutlich schneller als arithmetische Kodierungen, in der Regel jedoch nicht so hoch komprimierend. Außerdem ist eine Implementierung mittels arithmetischer Kodierung besser geeignet für Erweiterungen, wie die Nutzung von adaptiven Modellen oder dem Umschalten von verschiedenen Kontexten. Ein arithmetischer Kodierer stellt das Eingabefeld als eine einzige Zahl dar, bestehend aus einer Folge von binären Nachkommastellen. Für weitere Informationen sei der Leser auf das Standardwerk „Arithmetic Coding for Data Compression" von Witten, Neal und Cleary [20] sowie auf den Artikel „Arithmetische Kodierung" von Bodden, Clasen und Kneis [21] verwiesen. Die Daten der vorliegenden Implementierung wurden mit Hilfe eines adaptiven arithmetischen Kodierers erzielt.

3. ERGEBNISSE

3.1 Kompressionsrate

In Abbildung 13 sind die Kompressionsraten von verschiedenen Datenkompressionsprogrammen aufgelistet. Als Testdateien wurde der Standard Calgary-Corpus, bestehend aus 14 Dateien, benutzt [11]. Die Kompressionsrate wurde als Quotient der Länge der Ausgabedatei zu der Länge der Eingabedatei ermittelt, gemessen in Bits der Ausgabedatei zu Bytes der Eingabedatei (bps), so daß kleinere Werte eine höhere Kompression bedeuten. Der ungewichtete Durchschnitt der Kompressionsrate ist in der untersten Zeile aufgeführt. Das in diesem Artikel vorgestellte Programm ist unter der Bezeichnung PIK03 zum finden. Die anderen Programme sind GZIP93 von Gailly mit der Option –9 [22], DMC87 von Cormack und Horspool [23], PPM90 von Moffat [24], PPMZ2-99 von Bloom [25], BW94 von Burrows und Wheeler [1] sowie BZIP2-01 von Seward [19]. GZIP93 baut auf dem LZ77-Verfahren auf, DMC87 basiert auf dynamic Markov compression, PPM90 und PPMZ2-99 basieren auf der Prediction by Partial Matching Methode. Bei BW94 handelt

es sich um die ursprüngliche Implementierung von Burrows und Wheeler. BZIP2-01 ist eine aktuelle BWT-Implementierung, welche eine Huffman-Kodierung als Entropiekodierer benutzt. Mit einem Wert von 2,35 bps liegt PIK03 im Bereich von PPM-Implementierungen, die in der Regel wesentlich mehr Speicherbedarf und Rechenzeit benötigen.

Datei	**Größe**	**GZIP93**	**DMC87**	**PPM90**	**PPMZ2-99**	**BW94**	**BZIP2-01**	**PIK03**
bib	111.261	2,51	2,20	2,12	1,72	2,07	1,98	1,96
book1	768.771	3,25	2,51	2,54	2,20	2,49	2,42	2,40
book2	610.856	2,70	2,19	2,25	1,84	2,13	2,06	2,05
geo	102.400	5,34	4,80	4,91	4,58	4,45	4,45	4,57
news	377.109	3,06	2,77	2,68	2,21	2,59	2,52	2,50
obj1	21.504	3,84	4,12	3,72	3,66	3,98	4,01	3,89
obj2	246.814	2,63	2,76	2,52	2,24	2,64	2,48	2,46
paper1	53.161	2,79	2,73	2,48	2,21	2,55	2,49	2,46
paper2	82.199	2,89	2,59	2,45	2,18	2,51	2,44	2,42
pic	513.216	0,82	0,82	0,99	0,75	0,83	0,78	0,79
progc	39.611	2,68	2,75	2,48	2,26	2,58	2,53	2,50
progl	71.646	1,80	1,99	1,84	1,45	1,80	1,74	1,72
progp	49.379	1,81	2,00	1,80	1,45	1,79	1,73	1,70
Trans	93.695	1,61	1,92	1,72	1,21	1,57	1,53	1,49
		2,70	2,58	2,46	2,14	2,43	2,37	2,35

Abbildung 13: Kompressionsraten in bps von Datenkompressionsprogrammen

3.2 Geschwindigkeit der Kompression und Dekompression

Um die Geschwindigkeit von PIK03 zu verdeutlichen, wird PIK03 mit dem Standardkompressionsprogramm GZIP93 mit Option –9 verglichen, welches zu den schnellsten Kompressionsprogrammen zählt [22], sowie mit der PPM-Implementierung PPMZ2-99 von Bloom [25] und der BWT-Implementierung BZIP2-01 von Seward [19]. Die Meßwerte wurden auf einem WINDOWS 2000 PC mit einem 700 MHz Pentium III ermittelt. Für die Zeitmessungen der Kompression und der Dekompression wurden jeweils die Summe der Zeiten für alle 14 Dateien des Standard Calgary-Corpus zehnmal ermittelt und der Durchschnittswert gebildet. Die Summen umfassen die Zeiten für das Laden der

Eingabedaten und das Schreiben der Ausgabedaten der Programme und haben die Einheit Sekunden. Wie man erkennen kann, ist die Kompressionsgeschwindigkeit von PIK03 etwas größer als die von GZIP93 und BZIP2-01 und mehr als 50 mal höher als die von PPMZ2. Die Dekompressionsgeschwindigkeit von PIK03 ist etwa halb so groß wie die von GZIP93 und BZIP2-01 und mehr als 70 mal größer als die von PPMZ2. Die unterschiedlichen Dekompressionszeiten zwischen PIK03 und BZIP2-01 kommen insbesondere durch die Nutzung des arithmetischen Kodierers von PIK03 zustande. Die arithmetische Kodierung benötigt bei der Dekompression bereits etwa die Hälfte der Gesamtdekompressionszeit.

Vorgang	GZIP93	PPMZ2-99	BZIP2-01	PIK03
Kompression	2,60	148,13	2,56	2,55
Dekompression	0,81	146,04	0,92	1,88

Abbildung 14: Gesamtausführungszeiten in Sekunden für die 14 Dateien des Calgary-Corpus

4. ZUSAMMENFASSUNG

Datenkompressionsalgorithmen auf Grundlage der Burrows-Wheeler-Transformation erzielen gute Kompressionsraten sowie hohe Kompressionsgeschwindigkeiten. Ein solcher Datenkompressionsalgorithmus ist aus mehreren Stufen aufgebaut; in dem vorliegenden Beispiel besteht er aus 4 Stufen. Die erste Stufe stellt die Burrows-Wheeler-Transformation (BWT) dar. Die BWT sorgt dafür, daß Zeichen mit ähnlichem Kontext nahe beieinander angeordnet werden. Danach schließt sich eine Move-To-Front-Stufe (MTF) an. Aufgabe der MTF ist es, aus dem lokalen Kontext der BWT-Ausgabe einen globalen Kontext zu erzeugen. Als nächstes erfolgt eine Lauflängenkodierung der Nullen in der RLE0-Stufe. Zuletzt werden die Daten mit Hilfe eines Entropiekodierers (EC) in eine möglichst kurze Bitfolge umgewandelt.

Die vorgestellte Beispielimplementierung PIK03 erreicht eine Kompressionsgeschwindigkeit ähnlich der von GZIP93 und eine etwa halb so große Dekompressionsgeschwindigkeit wie die von GZIP93. Die Kompressionsrate von PIK03 ist mit 2,35 bps deutlich besser als die von GZIP93 und liegt im Bereich von PPM-Implementierungen, welche im allgemeinen wesentlich mehr Speicherbedarf und Rechenzeit benötigen.

LITERATURVERZEICHNIS

[1] Burrows, M., Wheeler, D.J.; “A block-sorting lossless data compression algorithm”; Technical report, Digital Equipment Corporation, Palo Alto, California; 1994

[2] Arnavut, Z., Magliveras, S.S.; “Block sorting and compression”; Proceedings of the IEEE Data Compression Conference; 1997

[3] Balkenhol, B., Kurtz, S.; “Universal Data Compression Based on the Burrows-Wheeler Transformation: Theory and Practice”; IEEE Transactions on Computers; 49(10); 2000

[4] Balkenhol, B., Kurtz, S., Shtarkov. Y.M. ; “Modification of the Burrows and Wheeler Data Compression Algorithm”; Proceedings of the IEEE Data Compression Conference; 1999

[5] Balkenhol, B., Shtarkov, Y.M.; “One attempt of a compression algorithm using the BWT”; Discrete Structures in Mathematics, Falculty of Mathematics, University of Bielefeld; 1999

[6] Chapin, B.; "Switching Between Two List Update Algorithms for Higher Compression of Burrows-Wheeler Transformed Data"; Proceedings of the IEEE Data Compression Conference, 2000

[7] Deorowicz, S.; “Improvements to Burrows-Wheeler compression algorithm”; Software Practice and Experience, 30(13), 1465-1483; 2000

[8] Deorowicz, S.; “Second step algorithms in the Burrows-Wheeler compression algorithm", Software Practice and Experience, 32(4), 99-111, 2002

[9] Fenwick, P.M.; "Improvements to the block sorting text compression algorithm”; University of Auckland, Auckland, New Zealand, Technical Report 120; 1995.

[10] Manzini, G.; “The Burrows-Wheeler Transform: Theory and Practice”; Proc. 24th Int. Symposium on Mathematical Foundations of Computer Science (MFCS '99). Szklarska Poreba, Poland; Springer Verlag Lecture Notes in Computer Science, 1672, 34-47; 1999

[11] Bell, T.C., Witten, I.H., Cleary, J.G.; Calgary Corpus: "Modeling for text compression"; Computing Surveys 21(4): 557-591; 1989

[12] Nelson, M.; "Data compression with the Burrows-Wheeler transform"; Dr. Dobbs' Journal; September 1996

[13] Seward, J.; "On the Performance of BWT Sorting Algorithms"; Proceedings of the IEEE Data Compression Conference 2000

[14] Seward, J.; "Space-Time Tradeoffs in the Inverse B-W Transform"; Proceedings of the IEEE Data Compression Conference 2001

[15] Larsson, J.N., Sadakane, K.; “Faster Suffix Sorting”, Technical report; 1999

[16] Fenwick, P.M.; "Block-Sorting Text Compression - Final Report"; The University of Auckland, New Zealand, Technical Report 130; 1996.

[17] Fenwick, P.M.; "Experiments with a Block Sorting Text Compression Algorithm", The University of Auckland, New Zealand, Technical Report 111; 1995

[18] Elias, P.; “Universal Codeword Sets and Representations of the Integers”; IEEE Transactions on Information Theory, 21(2):194-203; 1975

[19] Seward, J.; "The BZIP2 program 1.0.2"; http://sources.redhat.com/bzip2; 2001

[20] Witten, I., Neal, R., Cleary, J.; "Arithmetic Coding for Data Compression", Communication of the ACM, 30(6) 520-540; 1987

[21] Bodden, E., Clasen, M., Kneis J.; “Arithmetische Kodierung“; Lehrstuhl für Informatik IV der RWTH Aachen, 2001

[22] Gailly, J.L.; "The GZIP program", Version 1.2.4; 1993

[23] Cormack, G.V., Horspool, R.N.;"Data Compression Using Dynamic Markov Modelling"; Computer Journal, 30(6), 541-550; 1987

[24] Moffat, A.; "Implementing the PPM data compression scheme"; IEEE Transactions on Communications, 38,1917-1921; 1990

[25] Bloom C.; "Solving the problems of context modeling"; http://www.cbloom.com/papers/ppmz.zip; 1998