Datenkompression mit der Burrows-Wheeler-Transformation

Andreas Junghans, IM2

14. Juni 2001

Inhaltsverzeichnis

1 Einleitung

Die Burrows-Wheeler-Transformation, kurz BWT, wurde erstmals im Mai 1994 von Michael Burrows und David Wheeler in [BW94] dokumentiert. Ihren heutigen Namen erhielt sie allerdings erst später zu Ehren ihrer Erfinder. Die BWT ist eine reversible Transformation und arbeitet blockorientiert, d. h. ein Block von Eingangsdaten wird in einen Block von Ausgangsdaten transformiert, ohne daß dabei Informationen verloren gehen. Der eigentliche Nutzen der BWT besteht darin, daß sich die Ausgangsdaten in den meisten Fällen wesentlich besser komprimieren lassen als die Eingangsdaten.

Im folgenden werde ich zunächst einen (kurzen) Überblick über die heute eingesetzten Familien von verlustfreien Kompressionsverfahren geben. Daran schließt sich eine Beschreibung der BWT in beiden Richtungen, d. h. Transformation und Rücktransformation, an. Schließlich werde ich auf die Anwendung der BWT in der Datenkompression eingehen und eine einfache Implementierung des Verfahrens vorstellen.

Neben der vorliegenden Arbeit bieten auch [Nel96] und [Tam00] gute Einführungen in die BWT. Eine weitergehende Analyse des eigentlichen Kompressionsschritts liefert [Deo01].

2 Verfahren zur Datenkompression

2.1 Wörterbücher und Statistiken

Es gibt im wesentlichen zwei Familien von verlustfreien Kompressionsverfahren, die heute eingesetzt werden: die *Wörterbuch-basierten* und die *statistischen*.

Wörterbuch-basierte Algorithmen arbeiten, wie schon aus dem Namen hervorgeht, mit einem sogenannten Wörterbuch. Darin werden häufig vorkommende Zeichenfolgen der Eingangsdaten abgelegt, die in den Ausgangsdaten dann nur noch als Index ins Wörterbuch auftauchen. Diese Art von Algorithmen zeichnet sich durch gute Kompressionsraten und schnelle Kompression und Dekompression aus. Bekanntester Vertreter ist der LZW-Algorithmus (beschrieben in [LZ77]), der beispielsweise bei GIF-Bildern verwendet wird.

Statistische Verfahren dagegen basieren auf der Wahrscheinlichkeit, mit der ein bestimmtes Zeichen in den Eingangsdaten auftaucht. Diese Wahrscheinlichkeit wird im einfachsten Fall durch Zählung der verschiedenen vorkommenden Zeichen gewonnen. In den Ausgangsdaten erhalten dann häufig vorkommende Zeichen kurze, weniger häufig vorkommende dagegen lange Codes. Komplexere Algorithmen berücksichtigen außerdem noch den *Kontext* der einzelnen Zeichen. So ist z. B. in deutschen Texten die Wahrscheinlichkeit für den Buchstaben „u” sehr hoch, wenn das vorhergehende Zeichen ein „q” war. Andernfalls liegt sie eher niedrig. Je nach Länge des betrachteten Kontextes (0 für kein Zeichen, 1 für ein Zeichen, etc.) spricht von Verfahren der Ordnung n (engl. *Order-n*). Statistische Kompressionsalgorithmen arbeiten (v. a. bei höherer Ordnung) wesentlich effizienter als Wörterbuch-basierte, allerdings auch bedeutend langsamer. Bekannte Vertreter sind die Huffman- und die arithmetische Codierung.

Als Ergänzung zu den oben beschriebenen Verfahren können sogenannte *Lauflängen-Codierer* eingesetzt werden, wenn die Eingangsdaten lange Folgen von gleichen Zeichen aufweisen. Das ist beispielweise oft bei Graphiken mit Farbpaletten der Fall. Die Lauflängen-Codierung arbeitet sehr schnell, funktioniert aber nur bei dafür geeigneten Daten und erzielt für sich selbst eher geringe Kompressionsgewinne.

2.2 Adaptive und statische Verfahren

Sowohl bei Wörterbuch-basierten als auch bei statistischen Verfahren werden zur Dekompression nicht nur die komprimierten Daten, sondern zusätzlich noch das Wörterbuch bzw. die Codetabelle benötigt. Wenn diese Zusatzinformationen explizit angegeben werden, spricht man von *statischer* Kompression. Im Gegensatz dazu bauen *adaptive* Algorithmen die Wörterbücher bzw. Codetabellen

während der Kompression und Dekompression anhand der vorkommenden Zeichen auf. Dadurch ist die Kompression zu Beginn noch sehr ineffizient und erreicht erst mit zunehmender Anzahl Zeichen die Rate, die beim statischen Vorgehen sofort zur Verfügung steht. Außerdem sinkt die Effizienz, wenn sich die Art der Daten während der Übertragung ändert (z. B. in ausführbaren Dateien, die zum einen Maschinencode und zum anderen Texte für die Benutzeroberfläche enthalten). Der Vorteil ist aber, daß keine Zusatzinformationen gespeichert bzw. übertragen werden müssen, die die komprimierten Daten sonst wieder aufblähen würden.

Wörterbuch-basierte Kompression arbeitet üblicherweise adaptiv, statistische Packer gibt es in beiden Varianten.

2.3 Ströme und Blöcke

Eine weitere Klassifizierung ergibt sich aus der Art, in der die Eingangsdaten gelesen werden. Von einem *Strom* (engl. *Stream*) spricht man dann, wenn die Daten Zeichen für Zeichen verarbeitet werden.[1] Wichtigster Vorteil ist, daß gleich zu Beginn einer Datenübertragung mit der Kompression bzw. Dekompression begonnen werden kann.

Blockorientierte Verfahren dagegen benötigen immer erst eine gewisse Datenmenge (einen „Block"), um sie verarbeiten zu können. Dadurch lassen sich zwar bessere Ergebnisse erzielen, aber eine kontinuierliche Kompression bzw. Dekompression, wie sei z. B. im Videobereich benötigt wird, ist nicht mehr gegeben.

2.4 Einordnung der BWT

Die Burrows-Wheeler-Transformation ist, wie bereits erwähnt, kein Kompressionsverfahren. Sie sortiert Daten lediglich so um, daß sie anschließend sehr gut gepackt werden können. Als besonders effizient hat sich hier eine Kombination von Lauflängen-Codierung mit Huffman- oder arithmetischer Codierung erweisen (siehe Abschnitt 4.3).

Die BWT funktioniert nur blockweise, und zwar um so besser, je größer die Blöcke ausfallen. Damit ist sie nicht für die Kompression von Datenströmen geeignet, wie sie im Video- und Audiobereich vorkommen, sondern hauptsächlich für die Archivierung und/oder den nicht-isochronen Versand großer Dateien. Ihr großer Vorteil liegt darin, daß BWT-Packer mit einer Geschwindigkeit ähnlich der von Wörterbuch-basierten Verfahren arbeiten, aber Kompressionsraten erzielen, die sehr nahe an die besten statistischen Algorithmen herankommen.

[1] Das bedeutet nicht, daß immer nur ein Zeichen gelesen wird. Zur schnelleren Abarbeitung wird üblicherweise ein Puffer gefüllt, der anschließend komprimiert wird.

3 Die BWT

3.1 Vorwärtstransformation

Als Beispiels für die Durchführung der BWT soll hier der Text „HelloCello” dienen.[1] Zunächst werden alle möglichen Rotationen der Eingangsdaten gebildet, wodurch man die in Abbildung 3.1 gezeigte Matrix erhält. Zeile 0 enthält die ursprünglichen Daten, Zeile 1 die um eins nach links rotierten Daten usw. Wichtig dabei: Sowohl jede Zeile als auch jede Spalte enthält alle Zeichen des ursprünglichen Datenblocks.

	0	1	2	3	4	5	6	7	8	9
0	H	e	l	l	o	C	e	l	l	o
1	e	l	l	o	C	e	l	l	o	H
2	l	l	o	C	e	l	l	o	H	e
3	l	o	C	e	l	l	o	H	e	l
4	o	C	e	l	l	o	H	e	l	l
5	C	e	l	l	o	H	e	l	l	o
6	e	l	l	o	H	e	l	l	o	C
7	l	l	o	H	e	l	l	o	C	e
8	l	o	H	e	l	l	o	C	e	l
9	o	H	e	l	l	o	C	e	l	l

Abbildung 3.1: Die Ausgangsmatrix der BWT

Im nächsten Schritt werden die Zeilen der Matrix sortiert. Das Sortierkriterium spielt dabei, wie später noch zu sehen sein wird, nur eine untergeordnete Rolle, so daß nicht lexikographisch exakt sortiert werden muß. Eine auf- oder absteigende Anordnung der rotierten Eingangsdaten anhand ihrer Binärdarstellung reicht aus. Abbildung 3.2 zeigt die Matrix nach Abschluß einer aufsteigenden Sortierung anhand der ASCII-Codes der einzelnen Zeichen.

Damit ist die Vorwärts-BWT bereits abgeschlossen. Die Ausgabe der Transformation ist nämlich schlicht die letzte Spalte der Matrix, die mit L bezeichnet wird, sowie der Index I derjenigen Zeile, die die unveränderten Ausgangsdaten enthält (siehe Abbildung 3.2). Man erhält also einen Block von

[1] Hierbei handelt es sich um eine leichte Abwandlung des in [Tam00] verwendeten und wenig sinnreichen „Hallo-Ballo”. Burrows und Wheeler selbst benutzten „abacab” zur Veranschaulichung ihres Algorithmus (siehe [BW94]).

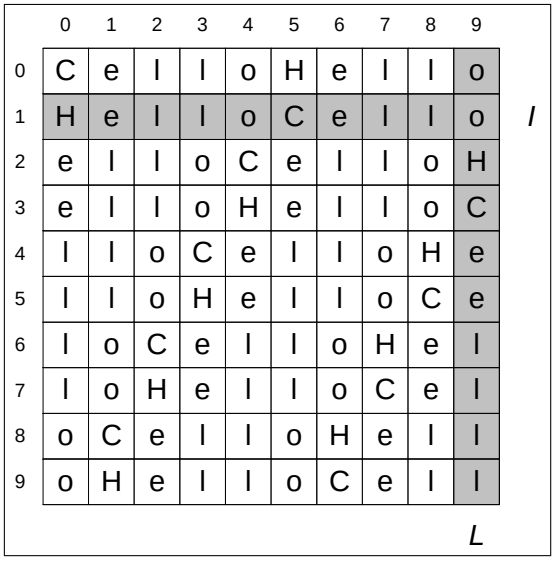

Abbildung 3.2: Die sortierte BWT-Matrix

der Länge der Ausgangsdaten[2], in dem sich nur die Reihenfolge der Zeichen geändert hat. Bevor ich nun darauf eingehe, warum die L-Spalte im allgemeinen gut komprimierbar ist, folgt zunächst eine Beschreibung der Rücktransformation, die sich überraschenderweise allein durch Kenntnis von L und I durchführen läßt.

3.2 Rücktransformation

Die Rücktransformation der BWT ist etwas komplexer als die Vorwärtstransformation. Die erste Spalte (F) der Matrix läßt sich allerdings noch sehr einfach rekonstruieren. Dazu muß man sich vor Augen führen, daß jede Spalte der Matrix alle Zeichen des originalen Datenblocks enthält – nur in anderer Reihenfolge. Berücksichtigt man außerdem, daß die Zeilen der Matrix bei der Vorwärtstransformation sortiert wurden, so wird klar, daß die erste Spalte F ganz einfach durch Sortierung von L gewonnen werden kann. Das erste Zeichen des Datenblocks ist damit auch bekannt, da der Index I die Zeile angibt, die die unveränderten Daten enthält (siehe Abbildung 3.3). Im Beispiel handelt es sich um den Buchstaben 'H'.

Um nun das nächste Zeichen zu ermitteln, sucht man die Zeile N_1, die eine Linksrotation der Zeile I um ein Zeichen enthält. In dieser Zeile steht in der Spalte F das gesuchte Zeichen, da die Linksrotation es an die erste Stelle befördert hat. Sie können sich das leicht klarmachen, wenn Sie noch einmal Abbildung 3.1 betrachten. Dort finden Sie in Spalte 0 exakt die Ausgangsdaten wieder, da in Zeile 1 das 2. Zeichen nach vorne gewandert ist, in Zeile 2 das 3. usw.

Jetzt stellt sich natürlich die Frage, wie man die Zeile N_1 findet. Hierzu macht man sich zunutze, daß bei der Rotation nach links das erste Zeichen an die letzte Stelle wandert. Man braucht also in unserem Beispiel lediglich in der Spalte L nach dem 'H' zu suchen. Das nächste Zeichen im Datenblock ist dann in dieser Zeile in der Spalte F zu finden – in diesem Fall ein 'e'.

[2] Der Speicherbedarf des Index I ist dabei vernachlässigbar.

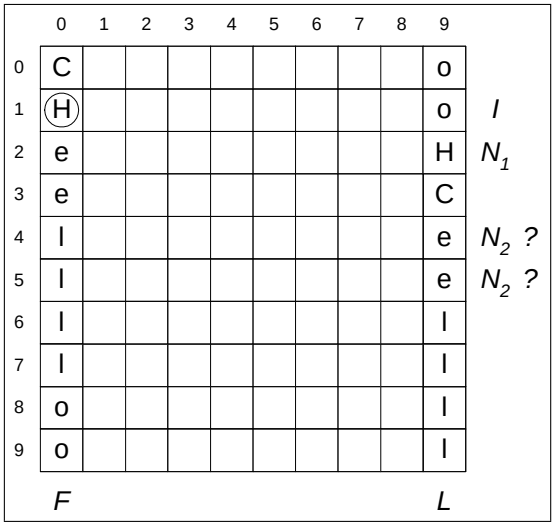

Abbildung 3.3: Rücktransformation mittels L und I

Für die restlichen Zeichen könnte man im Prinzip genauso vorgehen, wenn es nicht doppelt oder noch häufiger vorkommende Zeichen gäbe. In 3.3 ist zu sehen, daß prinzipiell zwei Zeilen für N_2 in Frage kommen, da beide mit einem 'e' enden[3]. Glücklicherweise ist aber durch die Eigenschaften der BWT eindeutig festgelegt, welche Zeile die richtige ist:

- Alle Zeilen der Matrix sind sortiert.
- Das gilt auch, wenn man die letzte Spalte entfernt, da für die Sortierung die Zeichen von links nach rechts betrachtet werden.
- Daraus folgt, daß alle Zeilen mit dem gleichen Zeichen am Ende in sortierter Reihenfolge auftreten.
- Das gilt auch für Zeilen, die mit dem gleichen Zeichen beginnen, denn diese sind anhand der nachfolgenden Zeichen sortiert.

Aus diesen Eigenschaften ergibt sich, daß alle Zeilen mit 'e' am Anfang in der gleichen Reihenfolge auftreten wie die mit 'e' am Ende. In unserem Beispiel ist N_1 die erste Zeile mit 'e' am Anfang, so daß N_2 die erste Zeile mit 'e' am Ende sein muß.

Auf diese Weise erhält man Schritt für die Schritt die originale Reihenfolge der Zeilen und kann aus der Spalte F die Ausgangsdaten rekonstruieren. Abbildung 3.4 zeigt das Ergebnis der kompletten Rücktransformation für unser Beispiel. In der Literatur ist an dieser Stelle übrigens häufig von einem *Transformationsvektor* die Rede. Gemeint ist damit die Folge der Zeilenindizes, die zu den ursprünglichen Daten führt. Im gewählten Beispiel ist der Transformationsvektor gleich $(1, 2, 4, 6, 8, 0, 3, 5, 7, 9)$ (siehe Abbildung 3.4).

[3]Das ist hier noch nicht kritisch, weil in beiden Zeilen die erste Spalte ein 'l' enthält. Ein Problem bekommt man erst danach, da es vier Zeilen mit 'l' am Ende gibt und in der ersten Spalte entweder ein 'l' oder ein 'o' steht.

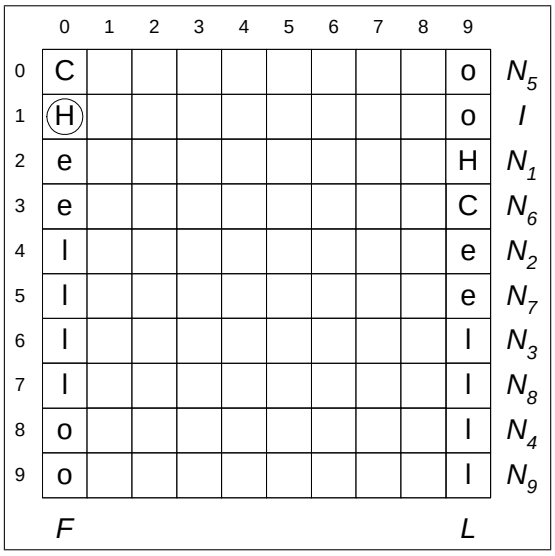

Abbildung 3.4: Ergebnis der Rücktransformation

4 Datenkompression mit der BWT

4.1 Eigenschaften der transformierten Daten

Nachdem die Funktionsweise der BWT klar geworden ist, stellt sich die Frage nach ihrem Nutzen. Dieser wird in Abbildung 4.1 verdeutlicht, die einen Ausschnitt aus der sortierten Transformationsmatrix zeigt. Quelle der Daten ist ein Teil des zu dieser Ausarbeitung erstellten Programmcodes. Vor dem Doppelpunkt ist die L-Spalte zu sehen, danach die ersten Spalten der Matrix, die aufgrund der Arbeitsweise der BWT (Rotation der Ausgangsdaten) die auf L folgenden Zeichen darstellen.

```
a: rray swap(pStart
a: rray localIndex =
a: rray  * (n
a: rray (i.e. the posit
a: rray (needed for rec
a: rray (which must be
A: rray = new byte[1];
a: rray and outputs the
a: rray for  * sortin
a: rray must not be ide
a: rray of <code>int</c
a: rray of MTF codes
a: rray of MTF codes
a: rray of that length
a: rray so that all equ
a: rray to write the or
a: rray using the speci
a: rray where the swap
a: rray with input data
a: rray!  * <p> *
A: rray) + ": " + new S
a: rray).  * The inde
a: rray.  *  * @exc
a: rray.  *  * @par
a: rray.  *  * @par
```

Abbildung 4.1: Beispiel für das Ergebnis der BWT

Im abgebildeten Beispiel ist deutlich zu sehen, daß vor den Zeichen „rray” entweder ein „a” oder ein „A” steht. Durch die Sortierung der Matrix erscheinen in der L-Spalte sehr viele dieser beiden Zeichen hintereinander. Die BWT führt also bei Daten mit ausgeprägter Kontextabhängigkeit (beispielsweise Texten, aber auch ausführbaren Dateien) dazu, daß ähnliche Zeichen gruppiert auftreten

und oftmals lange Ketten von gleichen Zeichen zu beobachten sind. Das gilt auch, wenn die Sortierung nicht exakt lexikographisch ist. Bei reiner ASCII-Sortierung sind z. B. „r" und „R" durch viele andere Zeichen getrennt. An der grundsätzlichen Eigenschaft, daß Zeichen mit gleichem oder ähnlichem Kontext gruppiert auftreten, ändert das aber nichts. Es werden lediglich Gruppen, die sonst zusammenhängend wären, in zwei Teile aufgespalten, so daß die Komprimierbarkeit geringfügig ungünstiger ausfällt (siehe nächster Abschnitt).

4.2 Move-To-Front-Codierung

Die beschriebenen Eigenschaften lassen sich sehr gut ausnutzen, indem ein sogenannter *Move-To-Front-Codierer (MTF)* verwendet wird. Das Prinzip dabei ist einfach: Zunächst werden alle vorkommenden Zeichen in einer Tabelle festgehalten. Bei der Codierung wird nun statt des ursprünglichen Zeichens seine Position in der Tabelle geschrieben. Dann wird es ganz vorne einsortiert, so daß ein erneutes Vorkomment des gleichen Zeichens als 0 codiert wird. Auf diese Weise werden lange Ketten gleicher Zeichen zu 0-Folgen. Gruppen von ähnlichen Zeichen werden mit kleinen Zahlen codiert, und nur „Ausreißer" führen zu höheren Werten.

Die Tabellen 4.1 und 4.2 zeigen die Häufigkeitsverteilung der einzelnen Bytes vor und nach BWT und MTF-Codierung. Als Ausgangs-Datei wurde der Quellcode des Hauptprogramms der zu dieser Arbeit erstellten Implementierung verwendet.

Wie man sehen kann, zeigt der erhaltene Datenstrom jetzt ausgezeichnete Kompressionseigenschaften. Die Häufigkeitsverteilung ist stark zugunsten der niedrigen Werte verschoben, und oft treten lange 0-Folgen auf.

Byte	*absolute Häufigkeit*	*relative Häufigkeit*
32 (Leerzeichen)	2643	10.6 %
9 (Tabulator)	2553	10.2 %
116 (t)	1662	6.6 %
101 (e)	1564	6.2 %
110 (n)	1075	4.3 %
105 (i)	1066	4.3 %
114 (r)	1010	4.0 %
10 (Zeilenvorschub)	900	3.6 %
13 (Wagenrücklauf)	900	3.6 %
111 (o)	788	3.1 %
Rest	10874	43.4 %

Tabelle 4.1: Byte-Häufigkeiten der Ausgangsdaten

Byte	*absolute Häufigkeit*	*relative Häufigkeit*
0	18311	73.1 %
1	2553	9.0 %
2	821	3.3 %
3	486	1.9 %
4	337	1.3 %
5	284	1.1 %
6	232	0.9 %
7	199	0.8 %
8	163	0.7 %
9	143	0.6 %
Rest	1800	7.2 %

Tabelle 4.2: Byte-Häufigkeiten nach BWT und MTF-Codierung

4.3 Eigentliche Kompression

Die MTF-codierten Daten werden jetzt mit einer Mixtur aus Lauflängen- und statistischer Codierung komprimiert. Für letzteres wird zumeist die Huffman-Codierung verwendet oder – wenn Lizenzfragen keine Rolle spielen und höchste Effizienz gefragt ist – die Arithmetische Codierung[1]. Dabei können sowohl adaptive als auch statische Verfahren verwendet werden.

Für die Beispielimplementierung zu dieser Arbeit habe ich ein relativ simples Verfahren gewählt. Genau wie Burrows und Wheeler verwende ich zwei Huffman-Bäume: Einen für die MTF-codierten Daten und für 0-Folgen, den anderen für das jeweils nach einer 0-Folge auftretende Byte. Diese Trennung führt zu einer höheren Kompressionseffizienz, da direkt nach einer 0-Folge keine weitere auftreten kann und somit Codes im zweiten Huffman-Baum gespart werden.

Der erste Baum enthält Codes für die einzelnen Bytes (maximal 255, da 0 nicht als einzelnes Zeichen vorkommt) und 255 weitere für 0-Folgen von 1 bis 255 Byte Länge. Ein weiterer spezieller Code steht für 255 0-Bytes, denen noch weitere folgen. Dieser Spezialcode ist nötig, um ein Umschalten auf den zweiten Huffman-Baum zu verhindern, wenn sehr lange 0-Ketten auftreten. Der zweite Baum ist einfacher strukturiert. Er enthält lediglich Codes für die Bytes von 1 bis 255.

Das von mir gewählte Verfahren ist vergleichsweise primitiv. Speziell die Codierung von 0-Folgen kann noch wesentlich verbessert werden (siehe beispielsweise [Deo01]). Dennoch erreicht diese Implementierung schon sehr gute Kompressionsraten, wie in Tabelle 5.1 zu sehen ist.

[1] Die arithmetische Codierung führt zu besseren Resultaten als das Huffman-Verfahren, ist aber patentrechtlich geschützt.

5 Implementierung

5.1 Einführung

Als konkretes Beispiel habe ich einen BWT-basierten Packer/Entpacker in Java realisiert. Durch die Einfachheit des Verfahrens reicht das vollkommen aus, um in vernünftiger Zeit und mit annehmbarem Speicherbedarf die Kompression durchzuführen. Der Aufruf erfolgt über die Kommandozeile, wobei drei Optionen zur Verfügung stehen:

- *-compress* Durchführung der BWT, anschließend Kompression wie in Abschnitt 4.3 beschrieben.
- *-decompress* Dekompression der Daten, anschließend Durchführung der Rücktransformation.
- *-matrix* Durchführung der BWT und Ausgabe der Transformationsmatrix (z. B. verwendet, um den Text in Abbildung 4.1 zu erhalten).

Die für die Vorwärts- und Rücktransformation verantwortlichen Programmabschnitte sind in Anhang A enthalten. Der gesamte Code steht auf meiner Homepage unter `http://www.fh-karlsruhe.de/~juan0014/bwt/index_de.html` zur Verfügung. Dort findet sich auch ein komprimiertes Archiv der ausführbaren Version, das bei installiertem JDK folgendermaßen ausgeführt werden kann:

```
java -jar BWT.jar <-compress|-decompress|-matrix> <input file>
<output file>
```

Um z. B. eine Datei zu komprimieren sieht der Aufruf wie folgt aus:

```
java -jar BWT.jar -compress eine_datei.txt eine_datei.txt.pack
```

Auf die Eingangsdaten wird direkt die BWT angewendet. Für einen echten Produktiveinsatz der Implementierung müßte noch eine Zerlegung in einzelne Blöcke vorgenommen werden. Bzip2 (siehe Abschnitt 5.5 und [Sew]) beispielsweise verwendet Blockgrößen zwischen 100 und 900 KB.

Achtung: Die Zieldatei (output file) wird ohne Rückfrage überschrieben!

5.2 Reduktion des Speicherbedarfs

Das Hauptproblem bei der Umsetzung der BWT ist die Sortierung der Matrix (siehe Abschnitt 3.1). Wenn man eine Blockgröße von 900 KB zugrunde legt, erhält man theoretisch eine Matrix mit einer Größe ca. 800 Gigabyte (900 KB * 900 KB)! Es müssen also diverse Tricks zum Einsatz kommen, um die Sortierung auch mit handelsüblicher Hardware bewerkstelligen zu können.

Zunächst einmal handelt es sich bei den einzelnen Zeilen der Matrix um Rotationen der Ausgangsdaten. Daher genügt es, die Daten nur einmal zu speichern. In den Matrixzeilen verwendet man nun – statt **Kopien** der Daten – **Zeiger** in die Daten. Dabei ergibt sich allerdings ein weiteres Problem: Ein Zeiger auf das letzte Datenbyte verweist wirklich nur auf dieses eine Byte. Man benötigt allerdings eine komplette Rotation der Daten, die mit diesem Byte beginnt. Hier gibt es zwei Lösungsansätze:

- Man kopiert die Ausgangsdaten einmal hinter sich selbst. Dadurch kann man auf eine beliebige Stelle zeigen und findet dahinter immer die kompletten Daten.
- Man prüft vor jedem Datenzugriff, ob man außerhalb des Speicherbereichs liegt. Wenn ja, zieht man einmal die Länge der Ausgangsdaten vom verwendeten Zeiger ab.

In der vorliegenden Implementierung habe ich mich für die zweite Möglichkeit entschieden. Grund war hauptsächlich das Fehlen von Zeigern in Java, so daß ich mit Array-Indizes arbeiten mußte. Die einfachste Lösung für das „Ring-Verhalten" des Arrays war dann, den Index bei jedem Zugriff modulo der Gesamtlänge zu nehmen:

```
einByte = daten[index % laenge];
```

Damit ist das Speicherproblem gelöst, denn für einen Block von 900 KB werden jetzt lediglich 900*1024 zusätzliche Integer-Werte benötigt. Bei einer Größe von 4 Byte pro Integer ergibt sich insgesamt ein Speicherbedarf von ca. 4,4 MB, der auf heutigen Systemen problemlos befriedigt werden kann.

5.3 Effiziente Sortierung

Ein weiterer Punkt betrifft das Sortierverfahren selbst. Ein Standard-Quicksort ist hier nicht angebracht, weil er nicht auf das Sortieren von langen Zeichenketten ausgelegt ist: Quicksort zerlegt die Ausgangsdaten in zwei Hälften, indem alle Elemente, die kleiner oder gleich einem bestimmten Wert (dem sogenannten *partitionierenden Element*) sind, an den Anfang wandern, während die größeren Elemente ans Ende geschoben werden. Auf beide Hälften wird der Algorithmus dann rekursiv angewendet, bis eine Gruppe nur noch aus einem (oder gar keinem) Element besteht. Das Problem in Bezug auf die BWT ist, daß Quicksort bei Zeichenketten extrem viele Einzelvergleiche benötigt. Wenn zwei Zeichenketten beispielsweise in den ersten 100 Zeichen übereinstimmen, müssen 100 Vergleiche durchgeführt werden, bis entschieden werden kann, welche die größere ist.

Diesen Nachteil umgeht der *Multikey-Quicksort-Algorithmus* (beschrieben in [BS97]): Als partitionierendes Element wird jeweils ein einzelnes Zeichen genommen. Im Gegensatz zu Standard-Quicksort werden dann **drei** Gruppen gebildet, die mit Zeichen kleiner, gleich und größer dem partitionierenden Element beginnen. Beachten Sie, daß für diese Unterteilung von jeder Zeichenkette

nur ein einzelnes Zeichen betrachtet werden muß! Genau wie bei Standard-Quicksort werden die Gruppen anschließend rekursiv sortiert. Allerdings wird jetzt bei der „gleich”-Gruppe nicht mehr das erste Zeichen betrachtet (das ja in der ganzen Gruppe identisch ist), sondern das nachfolgende. Auf diese Weise werden Zeichenketten, die mit n gleichen Zeichen beginnen, nicht jedesmal komplett untersucht, sondern immer nur ab den bisher noch nicht betrachteten Zeichen.

5.1 zeigt noch einmal den Algorithmus am Beispiel. Die Zeichen, die auf der jeweiligen Rekursionsebene verglichen werden, sind unterstrichen dargestellt. Das partitionierende Element wird zufällig daraus gewählt. Anschließend werden die „kleiner”-, „gleich”- und „größer”-Gruppen gebildet, die wiederum genauso behandelt werden. Man kann an diesem Beispiel einfach erkennen, wie in jedem Vergleichsschritt nur ein Zeichen statt der ganzen Zeichenkette verglichen wird.

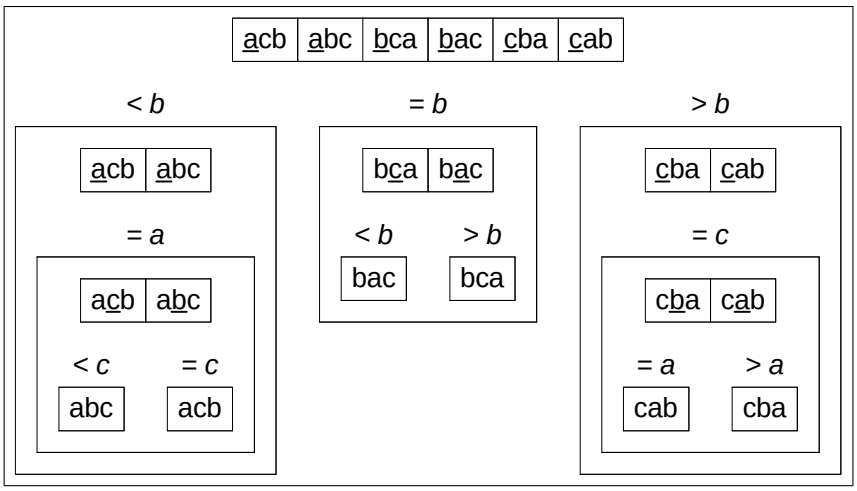

Abbildung 5.1: Der Multikey-Quicksort-Algorithmus

Eine genauere Beschreibung von Multikey-Quicksort findet sich in [BS97]. Burrows und Wheeler selbst verwenden in [BW94] einen komplexeren Algorithmus, der schneller arbeitet und weniger Probleme mit gleichen Zeichen hat (siehe nächster Abschnitt). Für die Demonstration der BWT ist Multikey-Quicksort jedoch vollkommen ausreichend.

5.4 Initiale Lauflängen-Codierung

Ein Nachteil des beschriebenen Multikey-Quicksort-Algorithmus ist, daß der Zeit- und Speicherbedarf bei sehr vielen gleichen Zeichen stark ansteigt. Wenn beispielsweise ein Block von 900 KB vorliegt, der nur aus 0-Bytes besteht, dann erhält man in jedem Rekursionsschritt immer dieselbe „gleich”-Gruppe. Die Rekursionstiefe steigt allerdings bis auf 900*1024 = 921600!

Um dem entgegenzuwirken, wird vor der Sortierung eine Lauflängen-Codierung vorgenommen, die Folgen von gleichen Zeichen verschwinden läßt. Dadurch ließ sich eine ursprünglich nicht verarbeitbare Datei von 2 MB Größe problemlos komprimieren.

5.5 Kompressionsvergleich

In der folgenden Tabelle finden Sie Vergleichswerte für die Kompressionseffizienz (alle Angaben in Bytes). Die gleichen Dateien wurden einmal mit WinZip (LZW-basiert), meiner eigenen Implementierung und mit bzip2 gepackt. Die zur Kompression und Dekompression benötigte Zeit unterscheidet sich kaum zwischen WinZip und bzip2. Meine Implementierung ist durch die Verwendung von Java, die fehlende Optimierung und die Verarbeitung der Dateien in einem einzigen Block deutlich langsamer, bewältigt aber die Kompression von powerpnt.exe auf einem System mit 350 MHz Pentium II und 256 MB Hauptspeicher in ca. einer Minute. WinZip und bzip2 liegen hier im Bereich von 10 Sekunden.

Datei	*unkomprimiert*	*WinZip*	*bzip2*	*BWT.jar*	*Beschreibung*
SRC-124.pdf	107.864	87.628	89.355	90.898	Beschreibung der BWT im PDF-Format (siehe [BW94])
opngl32p.cpp	873.331	76.600	52.221	57.510	C++-Quellcode
powerpnt.exe	4.247.604	2.151.382	2.030.911	2.132.403	Microsoft PowerPoint

Tabelle 5.1: Vergleich der Kompressionseffizienz

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

In dieser Arbeit wurde gezeigt, wie sich die Burrows-Wheeler-Transformation (BWT) zur effizienten Kompression von kontextabhängigen Daten einsetzen läßt. Es wurden die beiden Richtungen dieser reversiblen Transformation ebenso vorgestellt wie daran anschließende Möglichkeiten zur eigentlichen Kompression. Ferner wurde eine einfache Implementierung des Verfahrens vorgestellt, deren Kompressionsraten sich nicht hinter verbreiteten Programmen – wie z. B. WinZip – verstecken müssen.

6.2 Ausblick

Viele Probleme der Informatik erscheinen heute – mehr oder weniger – endgültig gelöst. Die BWT zeigt, daß auch in vergleichsweise gut erforschten Bereichen trotzdem noch überraschende neue Ansätze gefunden werden können. Das Verfahren ist ebenso elegant wie effizient und bei weitem noch nicht ausgereizt. Es erscheinen regelmäßig verbesserte Varianten, die zwar auf der gleichen Transformation aufsetzen, die resultierenden Daten aber mit immer ausgefeilteren Methoden komprimieren. Die mit der BWT erzielbaren Kompressionsraten werden also vermutlich noch weiter steigen.

Die BWT arbeitet mit der Vorsortierung der Daten vor der eigentlichen Kompression. Es ist sehr gut möglich, daß in Zukunft weitere reversible Transformationen entdeckt werden, die die Kompressionseigenschaften der Ausgangsdaten verbessern. Es stellt sich außerdem die Frage, ob die BWT oder ähnliche Verfahren auch bei verlustbehafteter Kompression Vorteile bieten können.

Es bleibt festzuhalten, daß die Datenkompression noch lange kein abgeschlossenes Kapitel ist. Lassen wir uns überraschen, was die Zukunft für uns bereithält ...

A Programmcode

A.1 Vorwärtstransformation

```
/**
 * Recursively Sorts a BWT matrix using a multi-key quicksort algorithm.
 * This is an adaptation of the string sorting algorithm from
 * <a href="http://www.cs.princeton.edu/~rs/strings">
 *   http://www.cs.princeton.edu/~rs/strings
 * </a>.
 * <p>
 * The private class variables _input and _pointers contain the input data
 * and the current pointer array for sorting.
 * </p>
 *
 * @param       pStart              startIndex (inclusive) in the string
 *                                  pointers array (needed for recursive sort).
 * @param       pLength             length of pointers (starting from pStart)
 *                                  to consider in the pointer array (needed
 *                                  for recursive sort).
 * @param       compIndex           index at which comparison of the string
 *                                  should start (needed for recursive sort).
 */

static public void sortByteStrings(int pStart, int pLength, int compIndex) {
    int a, b, c, d;

    // end recursion if we're left with one or no pointer
    if (pLength <= 1) {
        return;
    }

    // choose the partitioning element randomly
    _partIndex = pStart + (int)(Math.random()*(double)pLength) % pLength;
        // the modulus operator is there to make _absolutely_ sure that we're
        // not out of bounds

    // place the partitioning element at the beginning of the array
    swap(pStart, _partIndex);
    _partByte = _input[(_pointers[pStart] + compIndex) % _input.length];
        // the modulus operator provides the necessary wrap-around
        // functionality
```

```
// partition the pointer array so that all equal elements are placed on the
// left and right and the lesser and greater elements around the middle
a = pStart + 1;
b = a;
c = pStart + pLength - 1;
d = c;
while (b <= c) {
    do {
        _currentByte = _input[(_pointers[b] + compIndex) % _input.length];
        if (b <= c && _currentByte <= _partByte) {
            if (_currentByte == _partByte) {
                swap(a, b);
                ++a;
            }
            ++b;
        }
    } while (b <= c && _currentByte <= _partByte);
    do {
        _currentByte = _input[(_pointers[c] + compIndex) % _input.length];
        if (b <= c && _currentByte >= _partByte) {
            if (_currentByte == _partByte) {
                swap(c, d);
                --d;
            }
            --c;
        }
    } while (b <= c && _currentByte >= _partByte);
    if (b <= c) {
        // "equal" cannot happen, it’s just here to provide a 1:1
        // adaptation of the algorithm mentioned above
        swap(b, c);
        ++b;
        --c;
    }
}

// swap the left part of elements equal to the partitioning element
// to the middle
_swapLength = a-pStart;
_swapLength = (_swapLength <= (b-a) ? _swapLength : b-a);
for (_swapIndex = 0; _swapIndex < _swapLength; ++_swapIndex) {
    swap(pStart+_swapIndex, b-_swapLength+_swapIndex);
}

// swap the right part of elements equal to the partitioning element
// to the middle
_swapLength = pLength - (d+1-pStart);
_swapLength = (_swapLength <= (d-c) ? _swapLength : d-c);
for (_swapIndex = 0; _swapIndex < _swapLength; ++_swapIndex) {
    swap(b+_swapIndex, pStart+pLength-_swapLength+_swapIndex);
}
```

```
    // sort the lesser elements
    sortByteStrings(pStart, b-a, compIndex);

    // sort the equal elements
    if (compIndex < (_input.length - 1)) {
        sortByteStrings(pStart + b-a, (a-pStart) + (pLength-(d+1-pStart)),
                        compIndex+1);
    }

    // sort the greater elements
    sortByteStrings(pStart + pLength - (d-c), d-c, compIndex);

}


/**
 * Perfoms a Burrows-Wheeler transform on an input array. The result is written
 * to the specified output array (which must be of the same size or larger than
 * the input array). The index of the first input byte in the output array
 * (i.e. the position where the first byte of the original input is found after
 * resorting it to the output) is returned.
 * <p>
 * Attention: The input and output array must not be identical!!
 * Also, this method does stupid things if the input length is 0!
 * </p>
 *
 * @param       input               the input block.
 * @param       output              the output block.
 *
 * @return      the index of the first input byte in the output array.
 */

static public int compressBWT(byte[] input, byte[] output) {
    int[] pointers = new int[input.length];
    int i;
    int ret = 0;
    for (i = 0; i < input.length; ++i) {
        pointers[i] = i;
    }
    _input = input;
    _pointers = pointers;
    sortByteStrings(0, input.length, 0);
    for (i = 0; i < input.length; ++i) {
        output[i] = input[(pointers[i]+input.length-1) % input.length];
        //output[i] = input[pointers[i]];
        //System.out.println(pointers[i]);
        if (pointers[i] == 0) {
            ret = i;
        }
    }

    return ret;
}
```

A.2 Rücktransformation

```
/**
 * Performs a reverse Burrows-Wheeler transform on an array using the specified
 * index (see forward transform).
 * <p>
 * Attention: The input and output arrays must not be identical!!
 * </p>
 *
 * @param       input[]             the result of a BWT.
 * @param       index               the index resulting from a BWT.
 * @param       output[]            array to write the original bytes to.
 */

static public void decompressBWT(byte[] input, int index, byte[] output) {
    int i, j;
    byte nextByte;
    int localIndex, shortcutIndex;

    // reconstruct the first column of the matrix
    byte[] firstCol = new byte[input.length];
    System.arraycopy(input, 0, firstCol, 0, input.length);
    System.out.println("[decompress] Sorting first column of BWT matrix ...");
    Arrays.sort(firstCol);
    System.out.println("[decompress] Reconstructing the original " +
                       "input data ...");

    // build shurtcut arrays
    int[] count = new int[256];
    int[] byteStart = new int[256];
    int[] shortcut = new int[input.length];
    for (i = 0; i < 256; ++i) {
        count[i] = 0;
        byteStart[i] = -1;
    }
    for (i = 0; i < input.length; ++i) {
        shortcutIndex = (input[i] >= 0 ? (int)input[i] : (int)input[i] + 256);
        shortcut[i] = count[shortcutIndex];
        count[shortcutIndex] += 1;
        shortcutIndex =
            (firstCol[i] >= 0 ? (int)firstCol[i] : (int)firstCol[i] + 256);
        if (byteStart[shortcutIndex] == -1) {
            byteStart[shortcutIndex] = i;
        }
    }

    // reconstruct the original byte array
    localIndex = index;
    for (i = 0; i < input.length; ++i) {
        nextByte = input[localIndex];
        output[input.length-i-1] = nextByte;
        shortcutIndex = (nextByte >= 0 ? (int)nextByte : (int)nextByte + 256);
        localIndex = byteStart[shortcutIndex] + shortcut[localIndex];
    }
}
```

B Literaturverzeichnis

[BS97] Jon L. Bentley, Robert Sedgewick. *Fast Algorithms for Sorting and Searching Strings*, Januar 1997.
http://www.cs.princeton.edu/~rs/strings/paper.pdf.

[BW94] M. Burrows, D. J. Wheeler. *A Block-sorting Lossless Data Compression Algorithm*, Mai 1994.
http://gatekeeper.dec.com/pub/DEC/SRC/research-reports/abstracts/src-rr-124.html.

[Deo01] S. Deorowicz. *An analysis of second step algorithms in the Burrows-Wheeler compression algorithm*, Mai 2001.
http://www-zo.iinf.polsl.gliwice.pl/~sdeor/pub/deo01.ps.

[LZ77] J. Ziv, A. Lempel. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory*, Mai 1977.

[Nel96] Mark Nelson. Data Compression with the Burrows-Wheeler Transform. *Dr. Dobb's Journal*, September 1996.
http://www.dogma.net/markn/articles/bwt/bwt.htm.

[Sew] Julian Seward. *The bzip2 and libbzip2 home page*.
http://sourceware.cygnus.com/bzip2/.

[Tam00] Michael Tamm. Packen wie noch nie. *c't*, 16/2000.