Institut für Formale Methoden der Informatik
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3117

Eine Variante der Burrows-Wheeler Transformation mit Permutationen

Philipp M. Riegger

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. habil. Volker Diekert
Betreuer:	Dr. rer. nat. Manfred Kufleitner
begonnen am:	22. November 2010
beendet am:	24. Mai 2011
CR-Klassifikation:	E.4

Inhaltsverzeichnis

1 Einleitung **4**

2 Grundlagen **5**
2.1 Notation 5
2.2 Permutationen 5

3 Die Burrows-Wheeler Transformation **7**
3.1 Die Burrows-Wheeler Transformation 8
3.2 Die Umkehrung der Burrows-Wheeler Transformation 9
3.3 Ein schneller Algorithmus zur Umkehrung der BWT 11
3.4 Varianten der BWT 12

4 Die Burrows-Wheeler Transformation mit Permutationen **13**
4.1 BWT mit Permutationen 13
4.2 Umkehrung der BWT mit Permutationen 15

5 Implementierung **19**
5.1 `Bzip2` als Vorbild 19
5.2 Architekturentscheidungen 20
5.3 Verwendete Hilfsmittel und Bibliotheken 20
5.4 Implementierte Datentypen und Algorithmen 21
5.5 Eine Übersicht über `bwt_enc` 25
5.6 Die Funktionsweise von `bwt_dec` 29

6 Evaluation **30**
6.1 Einleitung 30
6.2 Kompressionsrate der Programme 31
6.3 Auswirkung der Parameter auf die Kompressionsrate 34

7 Ausblick **39**
7.1 Grundlagen 39
7.2 Implementierung 39
7.3 Evaluation 40

8 Zusammenfassung **41**

Abbildungsverzeichnis **42**

Tabellenverzeichnis **43**

Algorithmenverzeichnis **44**

Literaturverzeichnis **45**

Kapitel 1

Einleitung

In der Datenkompression werden häufig verschiedene Verfahren miteinander kombiniert, um höhere Kompressionsraten zu erzielen. Die Burrows-Wheeler Transformation (BWT) komprimiert einen gegebenen Datenblock zwar nicht, sortiert ihn jedoch so um, dass er mit einfachen Verfahren wie der Huffman-Kodierung besser komprimiert werden kann und eignet sich daher als Vorverarbeitungsschritt. Sowohl die Transformation selbst als auch ihre Umkehrung sind in Linearzeit berechenbar.

Bei der BWT werden Zeichen gruppiert, auf die gleiche oder ähnliche Zeichenketten, sogenannte Kontexte, folgen. Es werden also Ähnlichkeiten innerhalb der Eingabedaten genutzt, um einen besser komprimierbaren Datenblock zu erzeugen. Eine Variante der BWT nach Kufleitner erweitert diesen Begriff der Ähnlichkeit. Diese echte Verallgemeinerung der BWT nutzt Permutationen, um Teile der Eingabedaten so zu manipulieren, dass die Kontexte gleicher Zeichen ähnlicher und diese Zeichen damit besser gruppiert werden.

Wir stellen hier diese Variante der BWT sowie Algorithmen für die Transformation und ihre Umkehrung vor. Die Burrows-Wheeler Transformation mit Permutationen (BWTP) wird darin erstmals veröffentlicht und ein Beweis für die Umkehrbarkeit dargestellt. Ein an `bzip2` angelehntes, im Rahmen dieser Diplomarbeit entwickeltes Datenkompressionsprogramm namens `bwt_enc` wird vorgestellt. Es kombiniert einen effizienten Algorithmus zur Berechnung der BWTP mit der Huffman-Kodierung und einigen anderen Verfahren. Die Auswirkung verschiedener Parameterkombinationen und Permutationen werden untersucht und `bwt_enc` wird mit mehreren verbreiteten Datenkompressionsprogrammen verglichen.

Kapitel 2

Grundlagen

2.1 Notation

Sei Σ ein nicht-leeres endliches *Alphabet* mit einer *totalen Ordnung* $\leq$. Ein *Wort* w ist eine Aneinanderreihung $w_1 \cdots w_n$ von *Buchstaben* $w_i \in \Sigma$. Mit Σ^n wird die Menge aller Wörter der Länge n über dem Alphabet Σ bezeichnet. Außerdem sei $\Sigma^* = \bigcup_{n \in \mathbb{N}} \Sigma^n$ die Menge aller Wörter über Σ und $\Sigma^+ = \Sigma^* \setminus \{\epsilon\}$ die Menge aller nicht-leeren Wörter, wobei ϵ das leere Wort ist. Die *Rechtsrotation* $r(w)$ eines Wortes $w = w_1 \cdots w_n$ wird definiert als $r(w) := w_n w_1 \cdots w_{n-1}$, die i-fache Rechtsrotation ist rekursiv definiert durch $r^0(w) := w$ und $r^{i+1}(w) := r(r^i(w))$. Sei M eine Matrix, bezeichnen wir ihre Transponierte als M^T.

In Algorithmen und Beweisen wird ein an *C* [KR88] und Python[1] angelehnter Pseudokode verwendet: Die Buchstaben des Wortes w der Länge n werden als $w[i]$ mit $i \in [0, n-1]$ bezeichnet, die n Zeilen der Matrix M mit $M[i]$ für $i \in [0, n-1]$. Analog dazu ist $M[i][j]$ das Element an Stelle j in Zeile i der Matrix M. Zuweisungen werden mit $\leftarrow$ dargestellt und der Rumpf einer Schleife durch Einrückung identifiziert. Zur Konkatenation zweier Wörter und zum Anhängen von Buchstaben an Wörter wird das Pluszeichen verwendet: $abc \cdot def = abcdef$.

2.2 Permutationen

Eine *Permutation* π der Menge (des Alphabets) Σ ist eine bijektive Abbildung der Menge auf sich, $\pi : \Sigma \to \Sigma$. Sie wird auch als Σ-Permutation bezeichnet. Die Menge aller Σ-Permutationen mit der Hintereinanderausführung als Verknüpfung bildet eine Gruppe, die *Permutationsgruppe* auf Σ. Sie ist isomorph zur *symmetrischen Gruppe* über Σ. Diese wird als $S(\Sigma)$ bezeichnet. Die Anzahl der Elemente der Permutationsgruppe beträgt $|S(\Sigma)| = |\Sigma|!$, die Fakultät der Größe der zugrunde liegenden Menge. Permutationen werden üblicherweise als zweizeilige Tabellen dargestellt. Eine weitere Möglichkeit ist die *Zyklenschreibweise*.

[1] Python Programming Language, URL http://www.python.org/

Eine Permutation besteht aus Null oder mehr Zyklen. Zyklen sind dabei Sequenzen von Gruppenelementen, die zyklisch vertauscht werden.

Beispiel 1. *Sei* $\Sigma = \{1,2,3,4,5,6\}$*, betrachte die Permutation* $\pi : \Sigma \to \Sigma$ *mit* $\pi(1) = 2$*,* $\pi(2) = 1$*,* $\pi(3) = 5$*,* $\pi(4) = 4$*,* $\pi(5) = 6$ *und* $\pi(6) = 3$*. In zweizeiliger Tabellenschreibweise entspricht dies*

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 1 & 5 & 4 & 6 & 3 \end{pmatrix}$$

und in Zyklenschreibweise $(1,2)(3,5,6)$*, da hier die Elemente* 1 *und* 2 *vertauscht und die Elemente* 3*,* 5 *und* 6 *zyklisch rotiert werden. Die* 4 *wird auf sich selbst abgebildet und daher in der Zyklenschreibweise nicht notiert.*

Die Zyklenschreibweise ist nicht eindeutig, da z.B. $(1,2,3) = (2,3,1)$. Jede Permutation lässt sich jedoch in Zyklenschreibweise schreiben. Ein Beweis hierzu sowie weitere Details befinden sich in [Fol94].

Gegeben eine Menge Σ und die Permutationsgruppe $S(\Sigma)$. Eine Untergruppe $G \subseteq S(\Sigma)$ operiert auf Σ vermöge der Abbildung $G \times \Sigma \to \Sigma$, $(\pi, x) \mapsto \pi(x)$ (siehe auch [Boso6]). Wir erweitern die Abbildung auf Wörter $w = w_1 \cdots w_n \in \Sigma^+$ indem wir die Σ-Permutation buchstabenweise anwenden: $\pi(w) = \pi(w_1) \cdots \pi(w_n)$. Geben ein Wort $w \in \Sigma^+$ und eine Menge von Permutationen $P \subseteq S(\Sigma)$ definieren wir $\arg\min_{\pi \in P} \pi(w)$ als diejenige Permutation $\pi \in P$ für die $\pi(w)$ minimal bezüglich der lexikographischen Ordnung ist:

$$\arg\min_{\pi \in P} \pi(w) = \{\pi \in P \mid \forall p \in P : \pi(w) \leq p(w)\}.$$

Sei Σ ein Alphabet und Q eine Menge von Σ-Permutationen $Q \subseteq S(\Sigma)$. Die von Q erzeugte Untergruppe der Permutationsgruppe auf Σ wird mit $\langle Q \rangle_{S(\Sigma)}$ bzw. $\langle Q \rangle$ bezeichnet. Sie enthält die Identität id, alle Elemente der Menge Q sowie alle Permutationen die sich aus Verknüpfung der Elemente von Q erzeugen lassen.

Beispiel 2. *Sei* $\Sigma = \{i,m,p,s\}$*,* $\tau = (p,s)$ *die Permutation die die Elemente p und s vertauscht und* $Q = \{\tau\}$*. Die von Q erzeugte Untergruppe von* $S(\Sigma)$ *ist* $\langle Q \rangle = \{id, \tau\}$*, wobei id die Identität ist. Gegeben das Wort* $w = mississippi$ *ist* $id(w) = w$ *und* $\tau(w) = mippippissi$*, da* $\tau(w) < w$ *ist* $\arg\min_{\pi \in \langle Q \rangle} \pi(w) = \tau$*.*

Sei $w \in \Sigma^+$ ein Wort und $P \subseteq S(\Sigma)$ eine Untergruppe der Permutationsgruppe auf Σ. Die Äquivalenzrelation $\sim_P$ sei gegeben durch $v \sim_P w \Leftrightarrow \exists p \in P$ so, dass $p(v) = w$. Diese Relation teilt die Menge Σ^* in Äquivalenzklassen ein. Die P-Äquivalenzklasse von w ist $[w]_P = \{v \in \Sigma^* \mid v \sim_P w\}$. Gegeben zwei P-Äquivalenzklassen $[v]_P$ und $[w]_P$, sei $[v]_P \leq [w]_P \Leftrightarrow \min([v]_P) \leq \min([w]_P)$.

Kapitel 3

Die Burrows-Wheeler Transformation

> We describe a block-sorting, lossless data compression algorithm, and our implementation of that algorithm. We compare the performance of our implementation with widely available data compressors running on the same hardware.
>
> The algorithm works by applying a reversible transformation to a block of input text. The transformation does not itself compress the data, but reorders it to make it easy to compress with simple algorithms such as move-to-front coding.
>
> Our algorithm achieves speed comparable to algorithms based on the techniques of Lempel and Ziv, but obtains compression close to the best statistical modeling techniques. The size of the input block must be large (a few kilobytes) to achieve good compression.
>
> [BW94, authors abstract]

Mit diesen Worten beginnen Burrows und Wheeler ihren Forschungsbericht aus dem Jahre 1994. Sie beschreiben ein in der Datenkompression verwendetes Verfahren, das heute als "Burrows-Wheeler Transformation" bezeichnet wird. Wheeler entdeckte den Algorithmus im Jahre 1983, veröffentlichte ihn jedoch erst im o.g. Bericht. Der Algorithmus verarbeitet die Eingabe blockweise und nicht sequentiell. Die Transformation an sich komprimiert nicht, sondern sortiert einen gegebenen Datenblock so um, dass gleiche Buchstaben mit hoher Wahrscheinlichkeit hintereinander auftreten. Das wiederum erhöht die Kompressionsrate von einfachen herkömmlichen Datenkompressionsverfahren wie der Lauflängenkodierung (run-length encoding, RLE) oder der rotierenden Kodierung (move-to-front transform, MTF, [BSTW86]) kombiniert mit einer Huffman-Kodierung [Huf52].

1	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*	*p*	*p*	*i*
2	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*	*p*	*p*
3	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*	*p*
4	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*
5	*i*	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*
6	*s*	*i*	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*
7	*s*	*s*	*i*	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*
8	*i*	*s*	*s*	*i*	*p*	*p*	*i*	*m*	*i*	*s*	*s*
9	*s*	*i*	*s*	*s*	*i*	*p*	*p*	*i*	*m*	*i*	*s*
10	*s*	*s*	*i*	*s*	*s*	*i*	*p*	*p*	*i*	*m*	*i*
11	*i*	*s*	*s*	*i*	*s*	*s*	*i*	*p*	*p*	*i*	*m*

(a) Rotationen (M)

2	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*	*p*	*p*
5	*i*	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*
8	*i*	*s*	*s*	*i*	*p*	*p*	*i*	*m*	*i*	*s*	*s*
11	*i*	*s*	*s*	*i*	*s*	*s*	*i*	*p*	*p*	*i*	*m*
1	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*	*p*	*p*	*i*
3	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*	*p*
4	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*
6	*s*	*i*	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*
9	*s*	*i*	*s*	*s*	*i*	*p*	*p*	*i*	*m*	*i*	*s*
7	*s*	*s*	*i*	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*
10	*s*	*s*	*i*	*s*	*s*	*i*	*p*	*p*	*i*	*m*	*i*

(b) Sortierte Matrix (M')

Abbildung 3.1: Berechnung der BWT des Wortes $w = mississippi$

3.1 Die Burrows-Wheeler Transformation

Die Idee hinter der Burrows-Wheeler Transformation (BWT) ist wie folgt: Gegeben ein Wort w der Länge n, generiere die Matrix M aller n Rechtsrotationen[1] von w und sortiere die Zeilen dieser Matrix lexikographisch. Das Ergebnis der Transformation ist die letzte Spalte v der sortierten Matrix M' und der Index I von w in M'.

Da in der Matrix Rotationen des Eingabewortes w stehen, sind die Buchstaben in der letzten Spalte die Vorgänger der der Anfänge der entsprechenden Zeile. Damit werden im transformierten Wort v solche Buchstaben gruppiert, nach denen häufig dieselbe Buchstabenkombination folgt. Wenn in einem Eingabewort z.B. sehr häufig die Buchstabenfolge *en* auftritt, existiert für jedes dieser dieser Paare eine Rotation des Eingabewortes das mit n beginnt und mit e endet. Treten andere Buchstaben deutlich seltener oder nie als als Nachfolger von e auf, werden die Buchstaben e in v an den Positionen gruppiert, an denen die entsprechenden Zeilen der sortierten Matrix mit n beginnen.

Beispiel 3. *Die Zwischenschritte bei der Berechnung der Transformation des Wortes $w = mississippi$ sind in Abbildung 3.1 angegeben. Abbildung 3.1(a) zeigt die Rotationen des Wortes w, Abbildung 3.1(b) die sortierten Rotationen. Die Transformation des Wortes $w = mississippi$ ist damit $v = pssmipissii$ (die letzte Spalte der sortierten Matrix M') und der Index von w in M' ist $I = 5$.*

Die Transformation des Wortes $w = mississippi$ erhält beide Buchstabenpaare *ss* und erzeugt ein im Eingabewort noch nicht vorhandenes Buchstabenpaar *ii*. Das Paar *pp* wird in diesem Beispiel durch die Transformation zwar getrennt, jedoch überwiegen in der Praxis bei größeren Eingaben die Vorteile, also die Bildung von Buchstabengruppierungen den Nachteilen, der Aufspaltung vorhandener Gruppierungen.

[1] Alle Rechtsrotationen entsprechen allen Linksrotationen, daher wird im folgenden auch der Begriff Rotation verwendet.

Algorithmus 3.1 BWT (Burrows-Wheeler Transformation)

Input: Ein Wort $w \in \Sigma^+$ der Länge n
Output: Ein Wort $v \in \Sigma^+$ der Länge n und ein Index $I \in [0, n-1]$

1: **for** $i \leftarrow 0$ **to** $n-1$ **do**
2: $\quad M[i] \leftarrow r^i(w)$ ▷ Füge i-fache Rechtsrotation der Menge M hinzu
3: $M' \leftarrow \text{sort}(M)$ ▷ Sortiere die Rotationen
4: **for** $i \leftarrow 0$ **to** $n-1$ **do**
5: $\quad v \leftarrow v \cdot M'[i][n-1]$ ▷ Generiere das Wort v aus der letzten Spalte von M'
6: $I \leftarrow \text{find}(M', w)$ ▷ Berechne Index von w in M'
7: **return** (v, I)

Der Pseudokode der BWT ist in Algorithmus 3.1 gegeben. Zeilen 1 und 2 benötigen lineare Laufzeit, da hier nicht alle Rotationen explizit generiert werden müssen. Es werden lediglich Tupel bestehend aus einem Zeiger auf das Eingabewort w und der Verschiebung i gespeichert. Die Zeilen 4 und 5 benötigen auch lineare Zeit, die Zeile 6 nur logarithmische Zeit. Hier reicht eine simple binäre Suche, da die Matrix M' sortiert ist. Bleibt die Zeile 3, die Sortierung der Matrix, die in der Regel der Hauptaufwand der BWT ist.

Seward, der Autor des Datenkompressionsprogramms `bzip2`[2], hat das von Burrows und Wheeler [BW94] verwendete Sortierverfahren genauer untersucht [Sew00]. Diese schlagen eine optimierte Kombination aus Quicksort [Hoa61] und Radixsort vor, analysieren die Geschwindigkeit jedoch nur sehr knapp. Seward stellt weitere Optimierungen vor, analysiert das Sortierverfahren genauer und vergleicht es mit dem von Sadakane [Sad98]. Er kommt zu dem Schluss, dass trotz der besseren asymptotischen Laufzeit des Verfahrens von Sadakane in manchen Fällen der modifizierte Quicksort Algorithmus schneller ist. Dies ist der Fall wenn die durchschnittliche Länge zweier Präfixe die verglichen werden müssen um die kleinere Rotation zu identifizieren (average match length, AML) kleiner als 100 ist. Seward hat daher in `bzip2` ein hybrides Verfahren aus mehreren Sortieralgorithmen verwendet.

3.2 Die Umkehrung der Burrows-Wheeler Transformation

Erstaunlicherweise existiert ein effizienter Algorithmus zur Umkehrung der Burrows-Wheeler Transformation. Wir stellen hier zuerst einen langsamen einfachen Algorithmus vor, geben jedoch eine schnelle Variante in Abschnitt 3.3 auf Seite 11 an.

Satz 1. *Sei $n \in \mathbb{N}$ mit $n > 0$ und sei $w \in \Sigma^n$ ein Wort der Länge n. Sei M eine $n \times n$ Matrix, Zeile i der Matrix sei die i-fache Rechtsrotation des Wortes w: $M = (r^0(w), r^1(w), \ldots, r^{n-1}(w))^T$. Sei M' die Matrix bestehend aus den Zeilen von M in lexikographisch geordneter Reihenfolge. Sei $v \in \Sigma^n$ das aus dem letzten Buchstaben jeder Zeile von M' gebildete Wort: $v = M'[0][n-1] \cdots M'[n-1][n-1]$.*

[2] `Bzip2`, URL http://bzip.org/

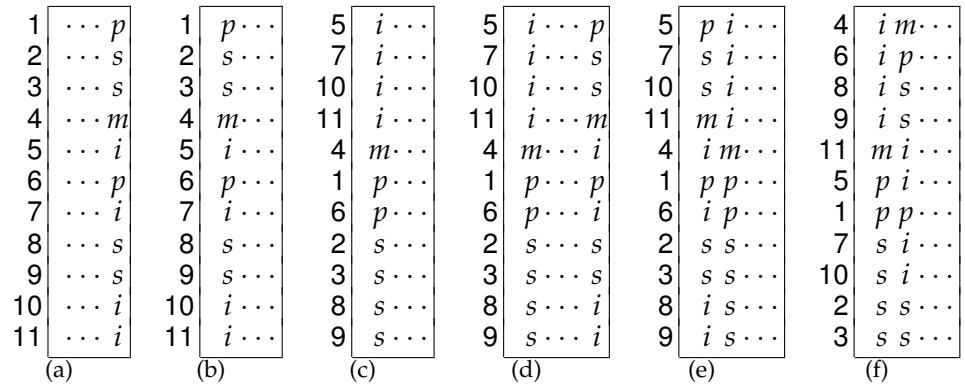


Abbildung 3.2: Berechnung der Matrix M' aus dem transformierten Wort $v = pssmipissii$

Dann gilt: Die Matrix M' lässt sich aus v rekonstruieren.

Beweis. Die Aussage kann mit vollständiger Induktion über die Länge der bekannten Präfixe jeder Zeile von M' bewiesen werden.

Das leere Wort ϵ ist das Präfix der Länge 0 jeder Zeile von M'. Gegeben ein Präfix der Länge k jeder Zeile der Matrix M', kann die bekannte letzte Spalte v eingefügt werden. Man erhält $M' = (m_1 \sqcup v_1, m_2 \sqcup v_2, \ldots, m_n \sqcup v_n)^T$, wobei "$\sqcup$" für momentan nicht bekannte Buchstaben steht. Da die n Zeilen der Matrix alle n Rotationen von w enthalten, kann jede Zeile um eins nach rechts rotiert werden. Die neue Matrix M'_{rot} enthält danach nach wie vor alle Rotationen von w: $M'_{\text{rot}} = (v_1 m_1 \sqcup, v_2 m_2 \sqcup, \ldots, v_n m_n \sqcup)^T$. Sei σ die Permutation, die die Zeilen von M'_{rot} sortiert, so ist $M'_{\text{rot+sort}} = (v_{\sigma(1)} m_{\sigma(1)} \sqcup, v_{\sigma(2)} m_{\sigma(2)} \sqcup, \ldots, v_{\sigma(n)} m_{\sigma(n)} \sqcup)^T$. Da M' die sortierte Matrix aller Rotationen des Eingabewortes w ist und M'_{rot} alle Rotationen von w enthält, gilt $M'_{\text{rot+sort}} = M'$. Da die Länge der vorher bekannten Präfixe $|m_i| = k$ ist, haben wir somit die Präfixe der Länge $|v_i m_i| = |m_i| + 1 = k + 1$ rekonstruiert. □

Korollar 1. *Seien n, w, M' und v wie in Satz 1 auf der vorherigen Seite. Sei zusätzlich zu v noch die Position I von w in M' gegeben, so kann das Wort w rekonstruiert werden.*

Beweis. Nach Satz 1 kann die Matrix M' aus v in n Schritten vollständig rekonstruiert werden. Da I so gewählt wurde, dass $M'[I] = w$, kann w rekonstruiert werden. □

Beispiel 4. *Abbildung 3.2 zeigt die ersten 6 Schritte der Umkehrung der BWT. Die letzte Spalte der Matrix M' ist bekannt (siehe 3.2(a)). Nach dem Schema in Beweis 3.2 kann nach rechts rotiert (siehe 3.2(b)) und sortiert werden (siehe 3.2(c)). Damit erhält man die erste Spalte von M'. Nun ist wieder die letzte letzte Spalte bekannt (siehe 3.2(d)). Da es sich um Rotationen handelt, kann rotiert (siehe 3.2(e)) und sortiert (siehe 3.2(f)) werden. So entsteht nach jeder Sortierung ein korrektes Präfix jeder Zeile der Matrix M' und die Matrix ist nach n Durchläufen wiederhergestellt. Aus dieser Matrix kann nun die korrekte Zeile I ausgeben und damit das Wort w wiederhergestellt werden.*

Der Pseudokode ist in Algorithmus 3.2 gegeben. Dabei wird das Einfügen der letzten Spalte und die Rotation direkt durch das Einfügen einer neuen ersten Spalte ersetzt. In den Zeilen 1 bis 3 wird die erste Spalte berechnet und die Matrix initialisiert. Bei jedem Durchlauf der Schleife in Zeile 4 bis 7 kommt eine neue Spalte hinzu, was am Ende des Algorithmus eine vollständige Rekonstruktion von M' bedeutet. Die Matrix ist nach dem Einfügen einer neuen ersten Spalte (vor Zeile 7) ab dem zweiten Zeichen sortiert. Somit ist es ausreichend, die Matrix anhand der ersten Spalte mittels eines stabilen[3] Sortierverfahrens zu sortieren oder eine entsprechende Permutation einmalig zu berechnen und in jedem Schritt anzuwenden.

Algorithmus 3.2 BWT^{-1} (Langsame Umkehrung der Burrows-Wheeler Transformation)

Input: Ein Wort $v \in \Sigma^+$ der Länge n und ein Index $I \in [0, n-1]$
Output: Ein Wort $w \in \Sigma^+$ der Länge n

1: **for** $i \leftarrow 0$ **to** $n-1$ **do**
2: $\quad M[i] \leftarrow v[i]$ ▷ Füge letzte Spalte in M ein
3: $M \leftarrow \text{sort}(M)$ ▷ Sortiere um erste Spalte zu erhalten
4: **for** $i \leftarrow 1$ **to** $(n-1)$ **do**
5: $\quad$ **for** $j \leftarrow 0$ **to** $(n-1)$ **do** ▷ Füge letzte Spalte hinzu
6: $\quad\quad M[j] \leftarrow v[j] \cdot M[j]$
7: $\quad M \leftarrow \text{sort}(M)$ ▷ Sortiere um Präfix zu erhalten
8: **return** $M[I]$ ▷ Die Matrix M ist nun die in BWT generierte Matrix M'

3.3 Ein schneller Algorithmus zur Umkehrung der BWT

Der in 3.2 beschriebene Algorithmus benötigt quadratischen Speicherplatz für die Matrix M' und damit mindestens quadratische Laufzeit. Es existiert jedoch ein einfacher Linearzeitalgorithmus zur Rekonstruktion des Wortes w aus v und I.

Durch das Wort v haben wir die letzte Spalte der Matrix M' gegeben. Aus dieser kann, wie oben beschrieben, die erste Spalte berechnet werden. Diese ist lediglich eine Umsortierung, die mit Hilfe einer Permutation aus v erzeugt werden kann. Diese Permutation wird auch Standardpermutation π_v genannt und kann in linearer Zeit berechnet werden (siehe [BW94, Sew01]).

Für jede Spalte von M' die in Algorithmus 3.2 erzeugt wird, wird eine neue erste Spalte v in unserer Matrix eingefügt und die Standardpermutation π_v angewandt. Am Ende des Algorithmus ist die erste Spalte von M' $\pi_v^1(v)$, die zweite Spalte $\pi_v^2(v)$, ... und die letzte Spalte $\pi_v^n(v) = v$. Dies ist unabhängig von v und so kann die I-te Zeile der Matrix direkt berechnet werden. Dazu wird die Inverse von π_v mit 1, 2, ..., n potenziert und

[3] Ein Sortierverfahren gilt als stabil, wenn es die Reihenfolge gleicher Zeichen oder Datensätze beibehält (und damit in unserem Fall die Sortierung ab dem zweiten Zeichen erhält).

der entsprechende Buchstabe des Wortes v am Index $\pi_v^{-i}(I)$ auslesen. Eine ausführliche Diskussion dieses Algorithmus mit diversen Implementierungsvarianten kann in [Sew01] nachgelesen werden.

Algorithmus 3.3 BWT^{-1} (Schnelle Umkehrung der Burrows-Wheeler Transformation)

Input: Ein Wort $v \in \Sigma^+$ der Länge n und ein Index $I \in [0, n-1]$
Output: Ein Wort $w \in \Sigma^+$ der Länge n

1: $S \leftarrow \text{sort\_perm}(v)$ ▷ Berechne Standardpermutation von v
2: $S^{-1} \leftarrow \text{invert}(S)$ ▷ Berechne Inverse der Standardpermutation
3: **for** $i \leftarrow 0$ **to** $n-1$ **do**
4: $\quad I \leftarrow S^{-1}(I)$ ▷ Benutze Zyklus anstatt Potenz explizit zu berechnen
5: $\quad w[i] \leftarrow v[I]$
6: **return** w

3.4 Varianten der BWT

Es existieren diverse Varianten der BWT. Teilweise wird dabei die Geschwindigkeit der Transformation erhöht, teilweise die Kompressionsrate verbessert. Fügt man am Ende des Eingabewortes ein neues Symbol $\perp$ ein, das nicht im Alphabet Σ enthalten ist, so spart man sich den Index I. Das Eingabewort wird zwar künstlich vergrößert, die richtige Zeile in der Matrix M' kann jedoch an ihrem letzten Buchstaben $\perp$ erkannt werden [BK00, Man01]. Definiert man die Ordnung auf $\Sigma \cup \{\perp\}$ so, dass $\perp$ das kleinste Symbol ist, verringert sich die zum Sortieren der Matrix benötigte Zeit. Hier entscheidet sich der Vergleich zweier Rotationen spätestens dann, wenn das Symbol $\perp$ auftritt. Bei der klassischen BWT müssen im schlimmsten Fall bei jedem Vergleich n Buchstaben verglichen werden.

Bei der klassischen BWT muss zusätzlich zum transformierten Eingabewort der Index I gespeichert werden. Bei der Variante mit $\perp$ Symbol wird das Eingabewort künstlich vergrößert. Damit ist keine dieser Varianten bijektiv. Gil und Scott haben dieses Problem analysiert und eine bijektive Variante der BWT vorgestellt [GS09]. Dabei wird das Eingabewort vor der Transformation mit der Lyndon Faktorisierung [CFL58] zerlegt. Dies ermöglicht die Rekonstruktion aus dem transformierten Wort ohne einen Index I oder ein zusätzliches Symbol $\perp$ und übertrifft die klassische BWT in der Kompressionsrate.

Eine weitere Variante der BWT ist die Sort Transform oder Schindler Transformation (ST) [Sch97]. Hierbei werden bei der Sortierung der Matrix M nur Präfixe der Länge k betrachtet. Abhängig vom Parameter k und der genauen Implementierung kann damit die Kompression beschleunigt werden, wobei jedoch die Dekompression leicht verlangsamt wird. Kufleitner hat dieses Verfahren mit der Lyndon Faktorisierung kombiniert und die bijektive BWT und ST in [Kuf09] genauer untersucht.

Kapitel 4

Die Burrows-Wheeler Transformation mit Permutationen

Bei der Burrows-Wheeler Transformation werden Rotationen mit gleichem oder ähnlichem Präfix gruppiert. Für das Beispielwort $w = mississippi$, werden einmal zwei s gruppiert, weil danach ip bzw. is folgen. Es werden nochmals zwei s gruppiert, weil danach jeweils si folgt und es werden zwei i gruppiert, weil danach die Zeichenkette ssi steht. Es wäre jedoch interessant die Transformation auf eine Weise zu modifizieren, die diese Gruppierungen gleicher Buchstaben vergrößert. Könnte der Sortieralgorithmus so angepasst werden, dass er pp und ss als identisch betrachtet und nebeneinander sortiert, würden mehr Buchstaben i gruppiert.

4.1 BWT mit Permutationen

Eine Variante der BWT nach Kufleitner sieht vor, nicht den Sortieralgorithmus zu ändern sondern einzelne Rotationen durch Permutationen gezielt zu modifizieren. Sein Verfahren sowie die Algorithmen zur Transformation und Umkehrung werden im folgenden vorgestellt. In Abschnitt 4.2 auf Seite 15 wird die Umkehrbarkeit formal bewiesen.

Wie bei der herkömmlichen BWT werden alle Rotationen eines Eingabewortes w erzeugt. Zu jeder Rotation wird die Permutation berechnet, die das Wort minimiert. Dazu wird eine Untergruppe P der Permutationsgruppe auf Σ verwendet. Im nächsten Schritt werden dann die permutierten Rotationen sortiert. Die letzte Spalte der Matrix, der Index der minimalen Permutation des Eingabewortes w in der sortierten Matrix sowie die zur Minimierung von w verwendete Permutation werden zurückgegeben.

Für die triviale Untergruppe der Permutationsgruppe ist jede minimierende Permutation die Identität. Somit handelt es sich bei der Variante um eine echte Verallgemeinerung der BWT. Eine Untergruppe der Permutationsgruppe wird verwendet, da im Beweis zur Umkehrbarkeit die Abgeschlossenheit verwendet wird. Dies wird in Beispiel 7 auf Seite 18 genauer erläutert.

1	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*	*p*	*p*	*i*
2	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*	*p*	*p*
3	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*	*p*
4	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*
5	*i*	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*
6	*s*	*i*	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*
7	*s*	*s*	*i*	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*
8	*i*	*s*	*s*	*i*	*p*	*p*	*i*	*m*	*i*	*s*	*s*
9	*s*	*i*	*s*	*s*	*i*	*p*	*p*	*i*	*m*	*i*	*s*
10	*s*	*s*	*i*	*s*	*s*	*i*	*p*	*p*	*i*	*m*	*i*
11	*i*	*s*	*s*	*i*	*s*	*s*	*i*	*p*	*p*	*i*	*m*

(a) Rotationen

1	*m*	*i*	*p*	*p*	*i*	*p*	*p*	*i*	*s*	*s*	*i*
2	*i*	*m*	*i*	*p*	*p*	*i*	*p*	*p*	*i*	*s*	*s*
3	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*	*p*
4	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*
5	*i*	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*
6	*p*	*i*	*s*	*s*	*i*	*m*	*i*	*p*	*p*	*i*	*p*
7	*p*	*p*	*i*	*s*	*s*	*i*	*m*	*i*	*p*	*p*	*i*
8	*i*	*p*	*p*	*i*	*s*	*s*	*i*	*m*	*i*	*p*	*p*
9	*p*	*i*	*p*	*p*	*i*	*s*	*s*	*i*	*m*	*i*	*p*
10	*p*	*p*	*i*	*p*	*p*	*i*	*s*	*s*	*i*	*m*	*i*
11	*i*	*p*	*p*	*i*	*p*	*p*	*i*	*s*	*s*	*i*	*m*

(b) Permutierte Rotationen (M)

2	*i*	*m*	*i*	*p*	*p*	*i*	*p*	*p*	*i*	*s*	*s*
5	*i*	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*
11	*i*	*p*	*p*	*i*	*p*	*p*	*i*	*s*	*s*	*i*	*m*
8	*i*	*p*	*p*	*i*	*s*	*s*	*i*	*m*	*i*	*p*	*p*
1	*m*	*i*	*p*	*p*	*i*	*p*	*p*	*i*	*s*	*s*	*i*
3	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*	*p*
9	*p*	*i*	*p*	*p*	*i*	*s*	*s*	*i*	*m*	*i*	*p*
6	*p*	*i*	*s*	*s*	*i*	*m*	*i*	*p*	*p*	*i*	*p*
4	*p*	*p*	*i*	*m*	*i*	*s*	*s*	*i*	*s*	*s*	*i*
10	*p*	*p*	*i*	*p*	*p*	*i*	*s*	*s*	*i*	*m*	*i*
7	*p*	*p*	*i*	*s*	*s*	*i*	*m*	*i*	*p*	*p*	*i*

(c) Sortierte Matrix (M')

Abbildung 4.1: Berechnung der BWTP des Wortes $w = mississippi$

Beispiel 5. *Abbildung 4.1 zeigt die Zwischenschritte bei der Berechnung der Transformation des Wortes $w = mississippi$. Hierbei wird die BWT mit Permutationen und als Untergruppe der Permutationsgruppe die von der Vertauschung der Buchstaben p und s erzeugte Gruppe verwendet. Diese besteht aus der Identität und der Vertauschung. Die Matrix aller Rotationen (siehe 4.1(a)) ist identisch zu der bei der herkömmlichen BWT generierten Matrix. Abbildung 4.1(b) zeigt die permutierten Rotationen. Dabei wurde in den Zeilen $1-2$ und $6-11$ die Vertauschung von p und s angewendet und in den Zeilen $3-5$ die Identität, also die Rotation nicht verändert. Die sortierte Matrix in 4.1(c) weicht in der Reihenfolge der Zeilen von der sortierten Matrix M' bei der herkömmlichen BWT ab. Die Transformation des Wortes w ist nun $v = ssmpipppiii$ und der Index $I = 5$ (in diesem Fall zufällig derselbe wie in Beispiel 3 auf Seite 8). Die zur Minimierung von w verwendete Permutation ist die Vertauschung von p und s.*

Algorithmus 4.1 BWTP (BWT mit Permutationen)

Input: Ein Wort $w \in \Sigma^+$ der Länge n und eine Untergruppe P der Permutationsgruppe $S(\Sigma)$

Output: Ein Wort $v \in \Sigma^+$ der Länge n, ein Index $I \in [0, n-1]$ und eine Permutation $\pi_0 \in P$

1: **for** $i \leftarrow 0$ **to** $n-1$ **do**
2: $\quad \pi \leftarrow \arg\min_{p \in P} p(r^i(w))$ ▷ Minimale Permutation berechnen
3: $\quad M[i] \leftarrow \pi(r^i(w))$ ▷ Permutation der i-fachen Rechtsrotation hinzu
4: $M' \leftarrow \text{sort}(M)$ ▷ Sortiere die permutierten Rotationen
5: **for** $i \leftarrow 0$ **to** $n-1$ **do**
6: $\quad v \leftarrow v \cdot M'[i][n-1]$ ▷ Generiere das Wort v aus der letzten Spalte von M'
7: $\pi_0 \leftarrow \arg\min_{p \in P} p(w)$ ▷ Berechne verwendete Permutation
8: $I \leftarrow \text{find}(M', \pi_0(w))$ ▷ Berechne Index der minimalen Permutation von w
9: **return** (v, I, π_0)

Algorithmus 4.1 auf der vorherigen Seite illustriert dieses Verfahren. Der Algorithmus ist identisch zu Algorithmus 3.1 auf Seite 9, sofern die triviale Untergruppe als Parameter P übergeben wird. In diesem Fall kann auf die Berechnung der minimalen Permutation in den Zeilen 2 und 7 verzichtet werden, da dies jeweils das einzige Element der Permutationsgruppe, die Identität, ist. Für nicht-triviale Untergruppen wird der Algorithmus deutlich komplexer. Die minimale Permutation die in Zeile 7 benötigt wird, wird schon in Zeile 2 berechnet und muss nur in einer weiteren Variablen gespeichert werden. Das gesuchte Element der Permutationsgruppe in Zeile 8 kann auch in Zeile 3 mitberechnet werden. Auf Strategien zur schnellen Berechnung der minimalen Permutation wird in Kapitel 5 auf Seite 19 und in Kapitel 7 auf Seite 39 näher eingegangen.

Um den Speicherplatz gering zu halten wird nicht direkt mit den Permutationen des Eingabewortes w gearbeitet. Stattdessen werden die im BWT Algorithmus verwendeten Tupel um die anzuwendende Permutation erweitert. Mehr Details sind Abschnitt 5.4.6 auf Seite 23 zu finden.

4.2 Umkehrung der BWT mit Permutationen

Algorithmus 4.2 BWTP^{-1} (Umkehrung der BWT mit Permutationen)

Input: Ein Wort $v \in \Sigma^+$ der Länge n, ein Index $I \in [0, n-1]$, eine Untergruppe P der Permutationsgruppe $S(\Sigma)$ und die verwendete Permutation $\pi_0 \in P$
Output: Ein Wort $w \in \Sigma^+$ der Länge n

1: **for** $i \leftarrow 0$ **to** $n-1$ **do**
2: $\quad \pi \leftarrow \arg\min_{p \in P} p(v[i])$
3: $\quad M[i] \leftarrow \pi(v[i])$ ▷ Füge letzte Spalte in M ein
4: $M \leftarrow \text{sort}(M)$ ▷ Sortiere um erste Spalte zu erhalten
5: **for** $i \leftarrow 1$ **to** $(n-1)$ **do**
6: $\quad$ **for** $j \leftarrow 0$ **to** $(n-1)$ **do** ▷ Füge neue erste Spalte hinzu
7: $\quad\quad \pi \leftarrow \arg\min_{p \in P} p(v[j] \cdot M[j])$
8: $\quad\quad M[j] \leftarrow \pi(v[j] \cdot M[j])$
9: $\quad M \leftarrow \text{sort}(M)$ ▷ Sortiere um Präfix zu erhalten
10: **return** $\pi_0^{-1}(M[I])$ ▷ Die Matrix M ist nun die in BWT generierte Matrix M'

Der langsame Algorithmus 3.2 auf Seite 11 lässt sich mit nur wenigen Modifikationen auch für die BWT mit Permutationen verwenden. Der Pseudokode ist als Algorithmus 4.2 angegeben. Zusätzlich zum kodierten Wort v und dem Index I wird die bei der Transformation verwendete Untergruppe P der Permutationsgruppe $S(\Sigma)$ sowie die zur Minimierung von w verwendete Permutation π_0 übergeben. Die Sortierung in den Zeilen 4 und 9 ist üblicherweise nicht die Standardpermutation von v. In Algorithmus 3.1 auf Seite 9 können wir davon ausgehen, dass die Zeilen der Matrix vor der Sortierung ab dem zweiten Zeichen sortiert sind. Das ist hier nicht mehr der Fall, da die Zeilen in jedem Schritt buchstabenweise

permutiert werden und damit die Sortierung verloren gehen kann. Wir können daher die zur Sortierung notwendige Permutation bei üblichen Blockgrößen nicht aus dem Wort v berechnen, sondern benötigen für ein festes[1] k ein Präfix der Länge k jeder Zeile von M, so dass kein Präfix mehrmals auftritt.

Dasselbe gilt für die Permutationen in den Zeilen 2 und 7: Sobald ausreichend lange Präfixe berechnet wurden, dass für jede Zeile der Matrix eine eindeutige Permutation aus P, die die Zeile minimiert, bestimmt werden kann, kann die Permutationen in einem Datenfeld gespeichert werden und in konstanter Zeit abgefragt werden. Es gilt jedoch folgendes zu beachten: Gegeben das Präfix m_i einer Zeile der Matrix M'. Das Präfix m_i sei mit der Permutation ρ_i buchstabenweise permutiert. Sei unsere Datenstruktur das Tupel (m_i, ρ_i), so ist $\rho_i^{-1}(v_i)$ das letzte Zeichen der Rotation in Zeile i. Sei die auf $\rho_i(\rho_i^{-1}(v_i)m_i)$ anzuwendende Permutation π_i, so ergibt sich im nächsten Schritt das Tupel $(\rho_i^{-1}(v_i)m_i, \pi_i \circ \rho_i)$. Daher muss nicht nur die korrekte Permutation schnell berechnet werden können, sondern auch das Produkt (die Hintereinanderausführung) zweier Permutationen. Verwenden wir nicht die vorgeschlagene Tupel-Datenstruktur, muss die buchstabenweise Permutation jedes Präfixes in jedem Schritt explizit berechnet werden.

Für eine Blockgröße von 100 kB und ein Alphabet der Größe 256 werden im besten Fall Präfixe der Länge 3 benötigt, im schlechtesten Fall[2] muss die komplette Matrix berechnet werden. Zur Bestimmung der Permutation für jede Zeile der Matrix werden im besten Fall Präfixe der Länge 0 benötigt ($|P| = 1$) und im schlechtesten Fall wieder die komplette Matrix[3].

Satz 2. *Sei $n \in \mathbb{N}$ und sei $w \in \Sigma^n$ ein Wort der Länge n. Sei $P \subseteq S(\Sigma)$ eine Untergruppe der Permutationsgruppe auf Σ. Sei M eine $n \times n$ Matrix und sei Zeile i der Matrix die i-fache Rechtsrotation des Wortes w: $M = (r^0(w), r^1(w), \ldots, r^{n-1}(w))^T$. Sei M' die Matrix mit den Zeilen von M in bezüglich ihrer P-Äquivalenzklasse lexikographisch geordneter Reihenfolge. Sei $v \in \Sigma^n$ das aus dem letzten Buchstaben jedes minimalen Vertreters der Äquivalenzklasse jeder Zeile von M' gebildete Wort: $v = \min([M'[0]]_P)[n-1] \cdots \min([M'[n-1]]_P)[n-1]$.*

Dann gilt: Die Matrix M' lässt sich aus v und P rekonstruieren.

Beweis. Dieser Beweis ist zu Beweis 3.2 auf Seite 10 identisch aufgebaut und daher deutlich knapper formuliert.

Das leere Wort ϵ ist das Präfix der Länge 0 jeder Zeile von M'. Wir betrachten hierbei die P-Äquivalenzklassen der Zeilen von M'. Gegeben sei ein Präfix m_i der Länge $|m_i| = k$ des minimalen Vertreters der P-Äquivalenzklasse jeder Zeile der Matrix M'. Dies sind die Präfixe der Zeilen von M', da die Matrix per Definition die minimalen Vertreter der Äquivalenzklassen enthält. Nun kann die bekannte letzte Spalte v eingefügt werden. Man erhält $M' = (m_1 \llcorner v_1, m_2 \llcorner v_2, \ldots, m_n \llcorner v_n)^T$, wobei "$\llcorner$" wieder für momentan nicht bekannte Buchstaben steht. Da die n Zeilen der Matrix Vertreter der P-Äquivalenzklassen

[1] Es genügen auch Präfixe unterschiedlicher Länge, sofern kein Präfix Präfix eines anderen Präfixes ist.
[2] Z.B. $w = 0 \cdots 0$ für ein festes $0 \in \Sigma$.
[3] Für $\tau = (1,2)$, $P = \langle\{\tau\}\rangle$ und $w = 0 \cdots 01$ wird erst mit dem letzten Zeichen der 0-fachen Rotation von w entschieden, ob τ angewendet wird oder nicht.

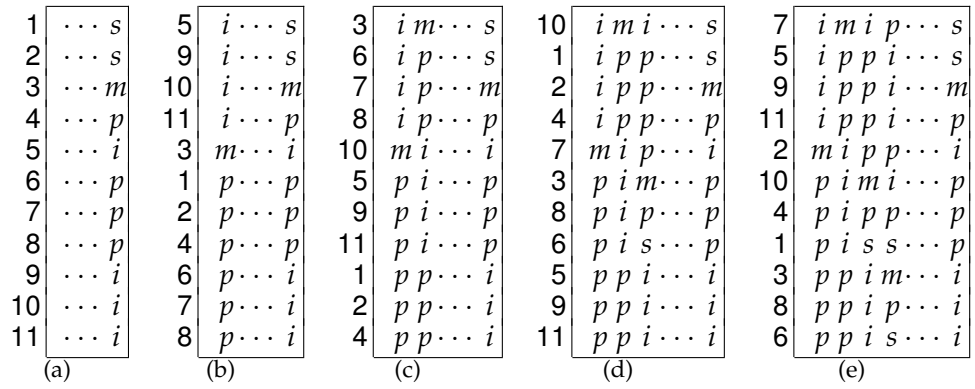


Abbildung 4.2: Berechnung der Matrix M' aus dem transformierten Wort $v = ssmpipppiii$

aller n Rotationen von w enthalten, kann jede Zeile um eins nach rechts rotiert werden. Die neue Matrix $M'_{\text{rot}} = (v_1 m_1 \sqcup, v_2 m_2 \sqcup, \ldots, v_n m_n \sqcup)^T$ enthält danach immer noch Vertreter der Äquivalenzklassen alle Rotationen. Da mit Äquivalenzklassen gearbeitet wird, muss an dieser Stelle nicht auf die zu verwendende Permutation geachtet werden. Die Zeilen von M' sind sortiert, daher kann M' aus M'_{rot} rekonstruiert werden, indem die Äquivalenzklassen der Zeilen sortieren und die minimalen Vertreter jeder Äquivalenzklasse ausgewählt werden. Sei σ die Permutation der Zeilen, die die Matrix sortiert, erhält man $M' = (\min([v_{\sigma(1)} m_{\sigma(1)} \sqcup]_P), \min([v_{\sigma(2)} m_{\sigma(2)} \sqcup]_P), \ldots, \min([v_{\sigma(n)} m_{\sigma(n)} \sqcup]_P))^T$. Da die Länge der bekannten Präfixe der Äquivalenzklassen $|m_i| = k$ ist, haben wir somit die Präfixe der Länge $|v_i m_i| = |m_i| + 1 = k + 1$ rekonstruiert. □

Korollar 2. *Seien n, w, P, M' und v wie in Satz 2 auf der vorherigen Seite. Sei π_0 so gewählt, dass $\pi_0(w) = \min([w]_P)$. Seien zusätzlich zu v und P noch π_0 und die Position I von $\pi_0(w)$ in M' gegeben, dann kann das Wort w rekonstruiert werden.*

Beweis. Nach Satz 2 kann M' aus v und P rekonstruiert werden. Da I so gewählt wurde dass $M'[I] = [w]_P$, kann $\min([w]) = M'[I]$ berechnet werden. Da die zur Minimierung verwendete Permutation π_0 ist, ist $w = \pi_0^{-1}(\min([w]_P))$. □

Im Beweis wird mit Äquivalenzklassen der Matrixzeilen gearbeitet, während in Algorithmus 4.2 der minimale Repräsentant immer explizit berechnet wird.

Beispiel 6. *Abbildung 4.2 illustriert den Algorithmus. Im ersten Schritt wird die bekannte letzte Spalte eingefügt (siehe 4.2(a)). Danach wird die Matrix rotiert, die Zeilen werden permutiert und sortiert. Nach dem erneuten Einfügen der letzten Spalte erhält man die in Abbildung 4.2(b) gezeigte Matrix. Die Abbildungen 4.2(c) bis 4.2(e) zeigen weitere Schritte in dem Verfahren, jeweils nach dem neuen Einfügen der letzten Spalte.*

Im Beweis zu Satz 2 wird die Abgeschlossenheit von P verwendet. Nach dem Hinzufügen einer neuen Spalte und der Rotation wird die Permutation berechnet, die die Zeile der

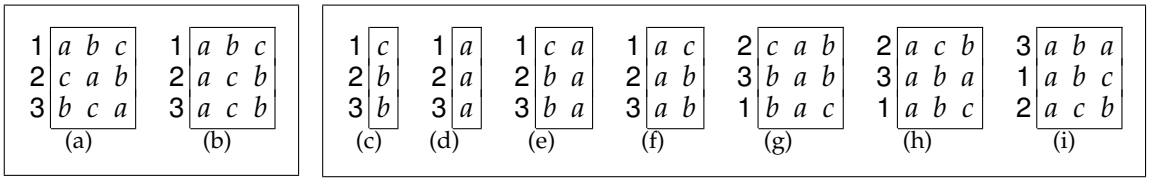


Abbildung 4.3: Probleme bei der Umkehrung bei Verzicht auf Abgeschlossenheit

Matrix minimiert. Bei der Zeile handelt es sich um eine Rotation des unbekannten Wortes w auf die eine Permutation aus P angewendet wurde. Nach der erneuten Minimierung wird also das Produkt zweier Permutationen auf die Rotation angewendet. Ist P eine Menge von Permutationen, die bezüglich der Verkettung nicht abgeschlossen ist, so muss dieses Produkt nicht in P enthalten sein und wir arbeiten möglicherweise mit einer permutierten Rotation, die in M' nicht enthalten ist. Beispiel 7 verdeutlicht, warum ohne die Abgeschlossenheit von P eine Umkehrung der BWTP nicht möglich ist.

Beispiel 7. *Sei $\Sigma = \{a, b, c\}$ ein Alphabet und $w = abc \in \Sigma^+$ ein Wort. Seien $\pi_0 = id$, $\pi_1 = (a, b)$ und $\pi_2 = (a, c)$ Permutationen. Sei $P = \{\pi_0, \pi_1, \pi_2\}$. Damit ist $P \neq \langle P \rangle_{S(\Sigma)}$, da $\pi_1 \circ \pi_2 = (a, c, b)$ nicht in P enthalten ist. Transformiert man nun w mit der BWTP und der nicht abgeschlossenen Permutationsmenge P, so erhält man $v = cbb$ und den Index $I = 1$. Abbildung 4.3(a) zeigt dabei die Matrix aller Rotationen und Abbildung 4.3(b) die permutierten Rotationen. In Zeile 2 wurde dabei π_2 angewendet, in Zeile 3 π_1. Die Matrix ist in diesem Fall schon sortiert, also können v und I direkt ausgelesen werden.*

Bei der Umkehrung wird die bekannte letzte Spalte v in die Matrix eingefügt (siehe 4.3(c)) und es werden die minimalen Permutationen berechnet und angewendet (siehe 4.3(d)). Hierbei wird auf Zeile 1 die Permutation π_2 angewendet und auf die Zeilen 2 und 3 π_1. Danach handelt es sich bei Zeile 1 um $\pi_2 \circ id(r^0(w)) = \pi_2(r^0(w))$, bei Zeile 2 um $\pi_1 \circ \pi_2(r^1(w))$ und bei Zeile 3 um $\pi_1 \circ \pi_1(r^2(w)) = id(r^2(w))$. Somit wird in Zeile 2 auf die zugrunde liegende 1-fache Rechtsrotation eine Permutation angewendet, die nicht in P enthalten ist. Dass wir dabei ein Präfix jeder Zeile der Matrix erhalten ist Zufall, in Abbildung 4.3(f) stimmen die Präfixe nicht mehr. Die rekonstruierte Matrix in 4.3(i) stimmt mit der in 4.3(b) nicht überein, insbesondere unterscheidet sich die erste Zeile der beiden Matrizen, in denen das Wort w stehen sollte. Daher ist die BWTP mit diesem Algorithmus bei Verzicht auf die Abgeschlossenheit der Permutationsmenge nicht umkehrbar.

Kapitel 5

Implementierung

Im Rahmen der Diplomarbeit wurden die in Kapitel 4 auf Seite 13 beschriebenen Algorithmen implementiert und evaluiert. Die Kompressionsrate des entstandenen Programms `bwt_enc` wird in Kapitel 6 auf Seite 30 genauer untersucht. Das zugehörige Programm zur Dekompression ist `bwt_dec`.

5.1 `Bzip2` als Vorbild

Das vorgestellte Programm `bwt_enc` ist großen Teilen Sewards `bzip2`[1] nachempfunden. `Bzip2` ist ein Datenkompressionsprogramm dessen zentraler Bestandteil die Burrows-Wheeler Transformation ist. Es ist auf hohe Geschwindigkeit und geringen Ressourcenverbrauch optimiert. Der hier zu treffende Kompromiss kann durch Kommandozeilenoptionen beeinflusst werden. `Bzip2` stellt nicht nur Programme zur Kompression und Dekompression von Daten bereit, sondern auch eine Bibliothek. Diese stellt Funktionen zur Verwendung von komprimierten Dateien und Speicherbereichen zur Verfügung.

`Bzip2` ist weit verbreitet. Es gehört zur Standardinstallation der meisten Linux Distributionen und der Quelltext von Programmen wird häufig als mit `bzip2` komprimiertes Archiv zum Download angeboten. Des weiteren bietet der Linux Kernel die Option, den erzeugten Betriebssystemkern mit `bzip2` zu komprimieren. Die dadurch deutlich kleinere Datei eignet sich damit z.B. zum Booten über Netzwerk.

`Bzip2` wendet eine Sequenz von Transformationen auf die Eingabedaten an. Diese wurde größtenteils in `bwt_enc` nachgebaut. `Bzip2` startet die Kompression mit einer Lauflängenkodierung. Dies dient der Vermeidung eines Falls, bei dem durch bestimmte Eingabedaten der Sortieralgorithmus sehr langsam wird. Seward bezeichnet diesen Schritt inzwischen jedoch als Fehler, den er auf Grund der notwendigen Rückwärtskompatibilität des Dateiformats leider nicht mehr ändern kann[2].

[1] `Bzip2`, Version 1.0.6, URL http://bzip.org/
[2] `Bzip2` Dokumentation, Abschnitt 4.1, URL http://www.bzip.org/1.0.5/bzip2-manual-1.0.5.html

Als nächstes werden die Eingabedaten mit der Burrows-Wheeler Transformation umgeformt. Hier ist eine Blockgröße in 100 kB Schritten zwischen 100 kB und 900 kB wählbar. Dem folgt eine rotierende Kodierung, eine weitere Lauflängenkodierung und schließlich eine Huffman Kodierung, wahlweise mit mehreren Huffman Tabellen.

Zu Beginn einer `bzip2` komprimierten Datei steht ein Dateikopf in dem unter anderem die verwendete Blockgröße gespeichert wird. Dem folgen ein oder mehrere Datenblöcke, jeweils bestehend aus einem Kopfteil und den komprimierten Daten. Im Kopfteil sind die Position des Eingabeblocks in der BWT Matrix, die Huffman Tabellen sowie weitere Steuerdaten gespeichert. Am Ende der Datei steht ein weiterer Block mit Steuerdaten, unter anderem mit der CRC-32 Prüfsumme der unkomprimierten Datei.

5.2 Architekturentscheidungen

Die von `bzip2` zur Kompression verwendeten Algorithmen sind aus Effizienzgründen so stark ineinander integriert, dass das Vorhaben zur Evaluation der Burrows-Wheeler Transformation mit Permutationen den BWT Teil von `bzip2` durch die BWTP zu ersetzen bei den gegebenen Rahmenbedingungen nicht umsetzbar war.

Daher wurde das modulare, stark an `bzip2` orientierte Programm `bwt_enc` geschrieben. Die Ziele der Implementierung waren die Möglichkeit, neue Transformationen und Algorithmen problemlos in die zur Kompression verwendete Sequenz von Algorithmen zu integrieren sowie die verwendeten Algorithmen möglichst stark durch Parameter anpassen zu können.

Auf eine hohe Geschwindigkeit wurde nur begrenzt Wert gelegt. Der Schwerpunkt war hier die Ermöglichung der Evaluation des Verfahrens, nicht die Minimierung der Laufzeit. Dasselbe gilt für den Speicherbedarf des Algorithmus. Hier wurde darauf geachtet, dass das Testsystem mit 2 GB für die Tests verfügbaren Arbeitsspeicher ausreichend war, nicht jedoch auf Einsparung jedes möglichen Bytes.

5.3 Verwendete Hilfsmittel und Bibliotheken

Bei der Implementierung wurde die Programmiersprache `C` [KR88] verwendet. Als Erweiterung der Standardbibliothek und für komplexere Datenstrukturen wie lineare Listen, Datenfelder variabler Größe, in logarithmischer Zeit durchsuchbare Listen, Bäume und assoziative Datenfelder wurde die Bibliothek `GLib`[3] verwendet. Aus ihr stammen auch das Sortierverfahren für die BWT sowie Funktionen für die Ausgabe von Statusmeldungen und zur Zeichenkettenmanipulation.

[3] `GLib`, Version 2.28.6, URL http://developer.gnome.org/glib/

Es wurden Modultests (Unit Tests) für die einzelnen Komponenten des Programms mit dem Testframework `CUnit`[4] geschrieben. Dies wurde mit einer regelmäßigen Überprüfung der Testabdeckung (Code Coverage) mit `lcov`[5] sowie Tests der Programmschnittstelle mit `shUnit2`[6] kombiniert. Daher waren auch größere Veränderungen am Quelltext problemlos durchführbar, da die korrekte Funktionalität der Algorithmen schnell und einfach getestet werden konnte.

Zur Verifikation der Speicherverwaltung wurde `Valgrind`[7] verwendet. Dieses Programm gibt Warnungen sowohl beim Zugriff auf nicht belegten bzw. nicht initialisierten Speicher als auch bei Versäumnissen bei der Speicherbereinigung (Speicherlecks) aus. Zur Optimierung des Laufzeitverhaltens des Programms wurden mit Hilfe des Profilers `gprof`[8] langsame Unterprogramme identifiziert. Diese konnten daher gezielt optimiert werden.

5.4 Implementierte Datentypen und Algorithmen

Die bei `bzip2` verwendeten Algorithmen wurden so genau wie möglich nachgebaut. Als Typ für Eingabe- und Ausgabedaten wird `GByteArray` aus der Bibliothek `GLib` verwendet, ein Datenfeld variabler Größe das die Zahlen 0 bis 256, also genau 1 Byte, speichern kann. Wenn von Zeichen die Rede ist, sind daher Bytes gemeint. Die implementierten Algorithmen verändern die Eingabedaten nicht, sondern erzeugen ein neues `GByteArray` für die Ausgabedaten.

5.4.1 Lauflängenkodierung

Bei der Lauflängenkodierung (run-length encoding, RLE) wird ein Datenblock so transformiert, dass lange Folgen sich wiederholender Zeichen zusammengefasst werden. Der hier verwendete Algorithmus wandelt Folgen gleicher Buchstaben der Länge 4 bis 259 in eine Zeichenfolge bestehend aus 4 mal dem gegebenen Buchstaben sowie der Anzahl der danach folgenden gleichen Buchstaben um. Im schlechtesten Fall wird der Datenblock dabei um den Faktor $1,25$ größer, im besten Fall wird er auf ca. 2% des Eingabeblocks reduziert. Das Verfahren wurde als einfacher Zustandsautomat implementiert.

5.4.2 Rotierende Kodierung

Die rotierende Kodierung (move-to-front transform, MTF) verändert die Größe des Datenblocks nicht, wandelt jedoch die einzelnen Zeichen so um, dass bei gehäuft auftretenden

[4] `CUnit`, Version 2.1-2, URL http://cunit.sourceforge.net/
[5] `lcov`, Version 1.7, URL http://ltp.sourceforge.net/coverage/lcov.php
[6] `shUnit2`, Version 2.1.5, URL https://code.google.com/p/shunit2/
[7] `Valgrind`, Version 3.6.1, URL http://valgrind.org/
[8] `gprof`, Teil von `binutils`, Version 2.21, URL http://sourceware.org/binutils/

Eingabezeichen diese bevorzugt in Zeichen mit kleinem Wert kodiert werden. Der Algorithmus verwaltet eine Tabelle, mit der jedes gelesene Zeichen übersetzt wird und die in jedem Schritt aktualisiert wird. Sie wird so initialisiert, dass jedes mögliche Eingabezeichen auf sich selbst abgebildet wird. Wird nun ein Zeichen z gelesen, werden die Position von z in der Tabelle ausgegeben, alle Einträge links dieser Position um eins nach rechts verschoben und z an die erste Stelle der Tabelle gesetzt. Kommt ein Zeichen häufig, jedoch nicht immer direkt hintereinander vor, wird es in der Regel eine relativ kleine Position in der Tabelle haben. Daher treten nach dieser Transformation kleine Zeichen gehäuft auf. Bei der Implementierung wurde zusätzlich zur o.g. Tabelle die inverse Übersetzungstabelle verwaltet. Damit kann mit einer Abfrage bestimmt werden, an welcher Stelle der Tabelle sich ein gegebenes Zeichen befindet.

5.4.3 Huffman-Kodierung

Um einen Datenblock mittels Huffman-Kodierung zu komprimieren, müssen zuerst die Anzahlen der Vorkommen jedes Zeichens gezählt werden. Daraus wird ein optimaler[9] Kode in Form eines binären Baumes generiert. Für eine detaillierte Beschreibung des Algorithmus siehe [Huf52] oder [Mac03].

Der Huffman-Baum wird bei dieser Implementierung in Form einer `GNode`, eines Baum-Datentyps aus der Bibliothek `GLib` gespeichert. Im binären Baum wird jedes im zu komprimierenden Datenblock vorkommende Zeichen in einem Blatt gespeichert. Der Pfad von der Wurzel zu einem Blatt ist eine mögliche Kodierung für dieses Zeichen. Hierbei kann beispielsweise der Schritt von einem Knoten zum linken Kind als 0-Bit und der Schritt zum rechten Kind als 1-Bit interpretiert werden. Um die Kodierungsvorschrift in der komprimierten Datei speichern zu können, werden hier weitere Schritte angewendet: Die Entfernungen von der Wurzel zu jedem Blatt im Huffman-Baum werden notiert. Damit sind die Längen der Kodewörter aller vorhandener Zeichen bekannt. Im Datenblock nicht enthaltene Zeichen müssen nicht kodiert werden und bekommen intern die Kodewortlänge 0 zugewiesen. Die Liste der Kodewortlängen wird in der komprimierten Datei gespeichert. Aus dieser kann ein eindeutiger Huffman-Baum erzeugt werden, aus dem wiederum die Kodewörter ausgelesen werden, mit denen der Datenblock komprimiert wird.

Zur einfacheren Handhabung der einzelnen Bits wird intern mit Zeichenketten bestehend aus den Zeichen 0 und 1 gearbeitet. Dies erhöht zwar den Speicherbedarf um mindestens den Faktor 8, stellt jedoch eine deutliche Vereinfachung der Schnittstellen und Datenstrukturen dar. Des weiteren hat dieses Detail der Implementierung keinerlei Auswirkungen auf die Funktionalität oder das Dateiformat, kann also jederzeit problemlos ausgetauscht werden.

Zur Dekodierung wird die gespeicherte Liste der Kodewortlängen gelesen. Aus diesen wird derselbe Huffman-Baum wie bei der Komprimierung generiert. Nun wird der komprimierte

[9] Wie in [WNC87] beschrieben, ist die Huffman-Kodierung nur dann optimal, wenn die Häufigkeit jedes Buchstabens eine Potenz von 1/2 ist. Dies ist in der Regel nicht der Fall. Jedoch ist der Huffman-Kode der optimale Kode, der eine ganzzahlige Anzahl von Bits zur Darstellung jedes Zeichens verwendet.

Datenblock bitweise eingelesen und beginnend bei der Wurzel, in jedem Schritt zu einem Knoten im Baum dessen linkes bzw. rechtes Kind ausgewählt. Ist ein Zeichen vollständig dekodiert, also ein Blatt im Baum erreicht, wird dieses ausgegeben und es wird wieder mit der Wurzel fortgefahren.

5.4.4 Deltakodierung

Zur Kompression der Metadaten anderer Verfahren wird die Deltakodierung verwendet. Hierbei wurde ein existierender Algorithmus von der Plattform *Rosetta Code*[10] übernommen. Bei der Deltakodierung wird das erste Zeichen eines Datenblocks mit sich selbst und jedes folgende Zeichen mit der Differenz zu seinem Vorgänger kodiert.

5.4.5 Burrows-Wheeler Transformation

Für die Burrows-Wheeler Transformation wird der in Kapitel 4 auf Seite 13 beschriebene Algorithmus mit der trivialen Untergruppe der Permutationsgruppe verwendet. Die Implementierung wird im nächsten Abschnitt genauer beschrieben. Für die Rückrichtung wird der in Kapitel 3 auf Seite 7 beschriebene Algorithmus zur schnellen Umkehrung der BWT verwendet. Hier wird die Standardpermutation als `GArray` erzeugt, invertiert und auf den Eingabedatenblock angewendet. Zur Generierung der Permutation werden zuerst die Vorkommen jedes Zeichens in den Eingabedaten gezählt, daraus die jeweils erste Position jedes Zeichens im sortierten Datenfeld berechnet und die Zeichen dann entsprechend einsortiert. Da hier aus den Indizes der Eingabedaten gearbeitet wird, ist das Ergebnis eine Permutation. Sowohl die Permutation als auch die Inverse werden in linearer Zeit erzeugt, daher benötigt der Algorithmus insgesamt lineare Zeit.

5.4.6 Burrows-Wheeler Transformation mit Permutationen

Die nötigen Modifikationen zur Berechnung der BWT mit Permutationen wurden wie in Kapitel 4 auf Seite 13 beschrieben in den BWT Algorithmus integriert. Gegeben ein Datenblock w und die Untergruppe der Permutationsgruppe P wird zuerst für jede Rotation von w ein Datentyp bestehend aus einem Zeiger auf w, der Verschiebung sowie der minimalen Permutation erzeugt. Zur Berechnung der minimalen Permutation wird ein zuvor generierter gerichteter azyklischer[11] Graph verwendet. Dieser stellt einen Automaten dar, bei dem jeder Zustand eine Teilmenge von P enthält. Für jeden Knoten k, dessen Permutationsmenge M mehr als ein Element enthält und jedes mögliche Zeichen z existiert eine von k ausgehende Kante. Diese zeigt auf den Knoten, dessen Permutationsmenge die Teilmenge von M ist, deren Elemente z auf das kleinstmögliche Zeichen abbilden. Dabei wird ein Knoten durch seine Permutationsmenge eindeutig identifiziert. Wird nun eine Rotation zeichenweise eingelesen, kann im Graph beginnend bei dem Zustand der

[10] Rosetta Code, URL http://rosettacode.org/wiki/Welcome_to_Rosetta_Code
[11] Der Graph enthält Schleifen, jedoch keine Zyklen der Länge 2 oder größer.

P enthält in jedem Schritt der Kante mit dem aktuellen Eingabezeichen gefolgt werden. Ist die Rotation komplett gelesen oder eine eindeutige Permutation identifiziert worden, terminiert der Algorithmus.

Diese minimalen Vertreter der Äquivalenzklasse bezüglich P jeder Rotation von w werden in einem `GPtrArray` gespeichert und mittels `g_ptr_array_sort` und einer selbst geschriebenen Vergleichsfunktion sortiert. Am Schluss wird das transformierte Wort aus dem sortierten Datenfeld ausgelesen, die Position des minimalen Vertreters der P-Äquivalenzklasse von w bestimmt und gemeinsam mit der vorher berechneten zur Minimierung von w verwendeten Permutation ausgegeben.

Ist der Parameter P die triviale Untergruppe der Permutationsgruppe, können die minimalen Permutationen in konstanter Zeit berechnet werden. Damit ist das von der Bibliothek `GLib` bereitgestellte Sortierverfahren bzw. die selbst geschriebene Vergleichsfunktion laut `gprof` der aufwendigste Teil der Transformation.

Für nicht-triviale Untergruppen wird ein Großteil der zur Transformation benötigten Zeit zur Berechnung der minimalen Permutationen verwendet. Dies variiert mit den gegebenen Permutationen. Ein besonders schlechter Fall ist, wenn durch eine Permutation Zeichen verändert werden, die nicht in den Eingabedaten vorkommen. Hier muss beim momentan implementierten Algorithmus jede Rotation komplett gelesen werden um am Ende zu entscheiden, dass es keine eindeutige Permutation gibt die die Rotation minimiert.

Zur Umkehrung der BWTP in `bwt_dec` kommt Algorithmus 4.2 auf Seite 15 zum Einsatz. Er rekonstruiert die komplette Matrix M', liest die durch I spezifizierte Zeile aus und macht die zum Minimieren verwendete Permutation rückgängig. Sei w ein Datenblock der Länge n und sei v die BWTP von w. Dann hat v auch die Länge n und M' ist eine $n \times n$ Matrix. Bei der Umkehrung der BWTP werden die n Spalten der Matrix nacheinander rekonstruiert. Dabei wird zuerst für jede der n Zeilen die minimierende Permutation gesucht (maximal n Schritte). Danach werden alle Zeilen sortiert ($n \log n$ Schritte). Insgesamt handelt es sich damit um einen kubischen Algorithmus.

5.4.7 Permutationen

Da hier mit Bytes als Eingabezeichen gearbeitet wird, werden zur zeichenweisen Permutation der Rotationen bijektive Abbildungen verwendet, die jedes der 256 möglichen Bytes auf ein anderes Byte abbilden. Als Datentyp kommt dafür ein `GByteArray` der Länge 256 zum Einsatz. Des weiteren wurde mit Hilfe von `flex`[12] ein Parser realisiert, der aus in Zyklenschreibweise angegebenen Permutationen das o.g. `GByteArray` erzeugt. Dabei werden Zyklen in runden Klammern angegeben und wahlweise mit Leerzeichen oder Kommas getrennt. Innerhalb eines Zyklus werden die Werte der einzelnen Zeichen als Dezimalzahl angegeben und mit mindestens einem Komma oder Leerzeichen voneinander getrennt. Zum Beispiel vertauscht die Permutation `(65, 97)(120, 121)` die Buchstaben a und A sowie die Buchstaben x und y.

[12] `Flex`, Version 2.5.35, URI http://flex.sourceforge.net/

Zur expliziten Berechnung der Untergruppe der Permutationsgruppe aus einer Menge von Permutationen wird ein optimierter Brute-Force-Algorithmus verwendet. Sei M die Permutationsmenge und S die zu erzeugende Untergruppe. Zu Beginn wird S mit der Identität (der Permutation, die jedes Zeichen auf sich selbst abbildet) initialisiert. Nun wird in jedem Schritt eine Permutation p aus M entfernt, in S eingefügt und jedes Produkt aus p und einer Permutation aus S der Menge M hinzugefügt, sofern es nicht in M oder S enthalten ist. Dies geschieht bis M leer ist. Als Datentypen für M und S wird `GSequence` aus der Bibliothek `GLib` verwendet. Dies ist eine als Baum gespeicherte Liste, bei der in logarithmischer Zeit Elemente entnommen, gesucht und sortiert eingefügt werden können. Am Ende wird S in ein `GPtrArray` umgewandelt, um mit Hilfe eines Indexes schnell auf die einzelnen Permutationen zugreifen zu können. Da jede im weiteren benötigte Permutation der Eingabezeichen in S enthalten und jede Menge von Permutationen eine Teilmenge der von S ist, wird dafür ein Zeiger auf S sowie ein Index oder eine Menge von Indices verwendet (siehe z.B. Abschnitt 5.4.6 auf Seite 23).

Eine für unseren Anwendungsfall unschöne Eigenschaft von Permutationen ist, dass die Permutationsgruppe auf der Menge aller Bytes sehr groß ist ($256! > 10^{500}$) und es damit nicht möglich ist, jede Untergruppe der Permutationsgruppe zu berechnen. Des weiteren lassen sich sehr einfach zwei Permutationen finden, die die gesamte Permutationsgruppe erzeugen (z.B. sind die Permutationen `(0, 1, ..., 255)` und `(0, 1)` Erzeugende). In diesem Fall würde das Programm voraussichtlich auf Grund eines Fehlers bei der Speicherreservierung terminieren, das Verhalten ist jedenfalls undefiniert.

5.5 Eine Übersicht über `bwt_enc`

Das Kompressionsprogramm `bwt_enc` wurde im Zuge dieser Diplomarbeit geschrieben. Es liest Dateien beliebiger Größe ein und erzeugt daraus eine neue komprimierte Datei mit einem speziellen Dateiformat. Die Eingabedatei wird dabei in Blöcke gleicher Größe zerlegt, wobei der letzte Block in der Regel kleiner ist und den Rest der Datei enthält. Die genaue Funktionsweise wird über Kommandozeilenoptionen gesteuert. Das zur Dekompression benötigte Programm ist `bwt_dec` und wird in Abschnitt 5.6 auf Seite 29 genauer beschrieben.

5.5.1 Kommandozeilenoptionen

Zusätzlich zur zu komprimierenden Datei verarbeitet `bwt_enc` verschiedene Kommandozeilenoptionen (siehe Abbildung 5.1 auf der nächsten Seite). Diese dienen der genauen Einstellung des Verhaltens und sind an `bzip2` orientiert. Es sind Blockgrößen zwischen 100 kB und 900 kB in 100 kB Schritten möglich, steuerbar durch die Optionen `-1` bis `-9`.

Die implementierten und über `-A` bzw. `--algorithms` spezifizierbaren Algorithmen sind `bwt` (die Burrows-Wheeler Transformation, mit und ohne Permutationen), `huffman` (die

```
Usage: bwt_enc <options> <filename>

<options> may contain:
    -1 .. -9                set block size to 100k .. 900k
    -A                      list of algorithms
    -a <permutation>        cycles: "(12, 24) (13, 23, 42)"
    -d                      print debug information
    -h, -?                  print this usage information
    -o <outfile>            write data to file instead stdout
    -v                      be verbose (multiple -v give more information)
```

Abbildung 5.1: Die Ausgabe von `bwt_enc --help`

Huffman-Kodierung), `mtf` (die rotierende Kodierung) sowie `rle` (die Lauflängenkodierung). Eine durch Komma getrennte Liste dieser Algorithmen kann übergeben werden. Diese werden dann bei der Kompression in der gegebenen Reihenfolge ausgeführt und in der komprimierten Datei gespeichert. Dabei können Algorithmen mehrfach angegeben werden. Um beispielsweise ein `bzip2`-ähnliches Kompressionsverfahren zu erhalten wird `bwt_enc` mit der Option `--algorithms rle,bwt,mtf,rle,huffman` ausgeführt. Die einzige Einschränkung[13] bei der Sequenz der Algorithmen ist, dass sowohl `bwt` als auch `huffman` genau einmal vorkommen müssen und dass letzteres an letzter Stelle stehen muss.

Die BWT mit Permutationen wird dann verwendet, wenn mittels `-a` bzw. `--add-permutation` eine oder mehrere Permutationen übergeben werden. Hier wird die Zyklenschreibweise und die Bezeichnung der Zeichen nach ihrer ASCII-Kodierung[14] in Dezimalschreibweise verwendet. Außerdem müssen je nach verwendeter Kommandozeilenumgebung die Klammern sowie die Leerzeichen maskiert werden.

Die Hilfe des Programms wird mittels `-h`, `-?` oder `--help` abgefragt. Die komprimierte Eingabedatei wird in die Standardausgabe geschrieben. Ist dies nicht erwünscht, kann mit `-o` oder `--outfile` eine Datei angegeben werden. In der Standardeinstellung gibt `bwt_enc` nur Fehlermeldungen und die komprimierte Datei aus. Hier können durch ein- oder mehrmalige Angabe der Option `-v` bzw. `--verbose` an `bzip2` angelehnte Statusinformationen ausgegeben werden (siehe Abbildung 5.2 auf der nächsten Seite). Dabei wird in der ersten Zeile der Name der zu komprimierenden Datei ausgegeben. Dem folgt für jeden Block die Prüfsumme des unkomprimierten Blocks, die kombinierte Prüfsumme der Eingabedatei bisher sowie die Blockgröße. Am Ende werden die kombinierte Prüfsumme der Eingabedatei sowie diverse Statistiken wie die Kompressionsrate, die Größe der Eingabedatei und die Größe der komprimierten Datei ausgegeben.

[13] Diese künstliche Einschränkung ist lediglich daher gegeben, da sie die Entwicklung der Metadaten (Dateikopf, Kopfteil der Datenblöcke) vereinfacht.

[14] American Standard Code for Information Interchange, ASA X3.4-1963, American Standards Association, June 17, 1963

```
files/large/bible.txt:
  block 1: adler = 0xb2c70ebc, combined adler = 0xb2c70ebc, size = 900000
  block 2: adler = 0x9c836045, combined adler = 0xf8436f00, size = 900000
  block 3: adler = 0x46f7c4cf, combined adler = 0x396333dd, size = 900000
  block 4: adler = 0xf4e34d66, combined adler = 0x08fd8142, size = 900000
  block 5: adler = 0xbfe29982, combined adler = 0xdb1c1ad2, size = 447392
  final combined adler = 0xdb1c1ad2
  4.302:1, 1.859 bits/byte, 76.76% saved, 4047392 in, 940714 out.
```

Abbildung 5.2: Die Ausgabe von `bwt_enc -9vv -o /dev/null files/large/bible.txt`

```
bwtp file header:

magic:16                    = signature/magic number
version:8                   = '0' for current unstable file format
number_of_algorithms:8      = length of sequence of algorithms
number_of_permutations:8    = number of permutations used in bwtp

algorithms:>=16             = sequence of algorithms
permutations:>=0            = compressed permutations used in bwtp
```

Abbildung 5.3: Der Dateikopf einer mit `bwt_enc` komprimierten Datei

5.5.2 Dateiformat

Beim Entwurf der Kopfteile der komprimierten Datei und der Datenblöcke wurde sich wieder an `bzip2` orientiert. Abbildung 5.3 listet die Felder des Dateikopfes auf. Dabei sind der Name des Feldes, die Größe in Bit sowie eine kurze Erklärung gegeben.

Am Anfang der Datei steht eine magische Zahl (2 Byte), anhand derer sich verifizieren lässt, dass es sich wirklich um eine mit `bwt_enc` komprimierte Datei handelt. Dem folgt die Version, gespeichert in einem Byte, sowie die Länge der Kette der Algorithmen und die Anzahl der bei der BWTP verwendeten Permutationen. Wird bei letzterem eine Null angegeben, kommt die herkömmliche BWT zum Einsatz.

Die Algorithmen werden intern als Aufzählungsdatentyp verwaltet und daher durch eine eindeutige Nummer repräsentiert. Diese Folge von Nummern wird als Folge von Bytes in die komprimierte Datei geschrieben. Die Folge `rle,bwt,mtf,rle,huffman` entspricht beispielsweise der Folge 3, 0, 2, 3, 1.

Die verwendeten Permutationen werden der Reihe nach komprimiert gespeichert. In einem Datenfeld der Länge 256 wird für jedes Zeichen die Differenz des Zeichens und des durch die Permutation gegebenen Bildes berechnet. Da in der Regel durch eine Permutation gezielt Zeichen ausgetauscht oder rotiert werden, sind die meisten Einträge hier 0. Um Rotationen aufeinanderfolgender Zeichen auch auf die Null abzubilden wird das Datenfeld

```
bwtp block header:

position:32                 = index of original string in bwt matrix
permutation_id:32           = index of permutation used for minimizing
enc_size:32                 = size of compressed block in bytes
bwt_size:32                 = size of block after bwt in bytes
adler:32                    = checksum of block

huffman_depths:>=16         = compressed huffman depths
```

Abbildung 5.4: Der Kopfteil eines Datenblocks einer mit bwt_enc komprimierten Datei

danach mittels Deltakodierung kodiert. Danach wird das Datenfeld in 16 Blöcke der Länge 16 unterteilt, für jeden Block wird mit einem Bit gespeichert ob dieser Block Zeichen ungleich Null enthält und jeder so identifizierte nicht-triviale Block wird mit einem Byte pro Eintrag des Datenfeldes in die Datei geschrieben.

Dem Dateikopf folgen beliebig viele Blöcke mit komprimierten Daten. Jeder Block besteht aus einem Kopfteil (Abbildung 5.4) und den eigentlichen Daten. Im Kopfteil werden die zur Umkehrung der Kompression benötigten Daten gespeichert. Sowohl Lauflängencodierung als auch rotierende Kodierung benötigen keine Metadaten. Zum Umkehrung der BWT wird der Index der nicht-rotierten Daten in der Matrix sowie im Falle der BWTP zusätzlich die zur Minimierung verwendete Permutation benötigt. Diese wird als Index der Permutation in der sortierten Untergruppe der Permutationsgruppe gespeichert.

Des weiteren werden die Größe des komprimierten Datenblocks sowie die Größe des Datenblocks nach der BWT gespeichert. Ersteres ist nötig da bekannt sein muss, welcher Datenblock mittels der Huffman-Kodierung entpackt werden muss. Letztere wird benötigt, da beim Packen und Entpacken eines Datenblocks mit der Huffman-Kodierung zusätzliche Daten am Ende des Blocks entstehen können. Dieser Effekt entsteht, da der bei der Kompression entstehende Datenblock eine Länge in Bit haben kann, die nicht durch 8 teilbar ist. In dem Fall wird das letzte Byte aufgefüllt. Hierbei lässt es sich nicht vermeiden, dass diese "Füllbits" ein neues Datenwort bilden. Dies wird bei der Dekomprimierung ans Ende des entstehenden Blocks angehängt. Sowohl Lauflängencodierung als auch rotierende Kodierung können das Präfix eines Datenblocks korrekt wiederherstellen auch wenn am Ende zufällige Daten angehängt sind. Bei der Umkehrung der BWT muss jedoch bekannt sein, welche Größe die Matrix und welche Länge die Standardpermutation haben muss. Daher muss diese Größe im Kopfteil des Blocks gespeichert werden.

Seward hat darauf hingewiesen, dass auf Grund der höheren Geschwindigkeit die Adler32 Prüfsumme der CRC-32 Prüfsumme vorzuziehen ist[15]. Daher wird dieses Verfahren verwendet und wie bei bzip2 die Prüfsumme jedes unkomprimierten Datenblocks im

[15] Bzip2 Dokumentation, Abschnitt 4.1, URL http://www.bzip.org/1.0.5/bzip2-manual-1.0.5.html

zugehörigen Kopfteil gespeichert. Dieser Algorithmus wurde nicht selbst implementiert, sondern es wurde auf die Bibliothek `zlib`[16] zurückgegriffen.

Zur Umkehrung der Huffman-Kodierung wird der verwendete Huffman-Baum benötigt. Dieser kann aus den Kodewortlängen rekonstruiert werden. Daher werden diese in einem komprimierten Format im Kopfteil der Datei gespeichert. Das Format ist dasselbe, das für die Permutationen im Dateikopf verwendet wird: Da nicht immer alle Zeichen verwendet werden, und nicht verwendete Zeichen mit einer Kodewortlänge von 0 gespeichert werden, wollen wir große Blöcke die nur aus Nullen bestehen nicht speichern. Daher unterteilen wir die Liste der Kodewortlängen aller Zeichen in 16 Blöcke mit jeweils 16 Zeichen, speichern in 16 Bit welche dieser Blöcke nicht nur aus Nullen bestehen und speichern dann diese Blöcke mit einer Kodewortlänge pro Byte ab.

Beim `bzip2`-Dateiformat wird am Ende jeder komprimierten Datei ein Steuerdatenblock gespeichert, der unter anderem die Prüfsumme der kompletten unkomprimierten Datei enthält. Dies wurde bisher nicht in `bwt_dec` integriert, da es verglichen mit anderen Angleichungen an `bzip2` viel Aufwand ist (es muss entschieden werden, ob gerade ein Block komprimierter Daten oder der Steuerdatenteil am Ende der Datei eingelesen wird) und keinen Einfluss auf die zu analysierende Kompression hat (jede Datei wird lediglich um eine kleine konstante Anzahl an Bytes größer).

5.6 Die Funktionsweise von `bwt_dec`

Das zum Entpacken von mit `bwt_enc` erzeugten Dateien verwendete Programm `bwt_dec` bietet deutlich weniger Kommandozeilenoptionen. Hier kann entweder die Ausgabedatei mittels `-o` bzw. `--outfile` angegeben werden oder die Hilfe mittels `-h`, `-?` oder `--help` abgefragt werden. Alle anderen Parameter sind in der komprimierten Datei hinterlegt worden.

Das Programm öffnet die übergebene Datei und liest zuerst den Dateikopf aus. Es erzeugt die Untergruppe der Permutationsgruppe aus den gegebenen Permutationen und dekomprimiert danach Block für Block. Dabei wird der Kopfteil des Blocks gelesen und dann die Liste der verwendeten Algorithmen in umgekehrter Reihenfolge abgearbeitet. Es kommen die Dekodierungsalgorithmen der Verfahren zum Einsatz. Am Ende wird die Prüfsumme des wiederhergestellten Datenblocks überprüft und dieser dann je nach Kommandozeilenparameter in eine Datei oder in die Standardausgabe geschrieben.

[16] `zlib`, Version 1.2.5, URL http://zlib.net/

Kapitel 6

Evaluation

Das in Kapitel 5 vorgestellte im Rahmen dieser Diplomarbeit entwickelte Programm `bwt_enc`[1] wurde genauer untersucht und mit anderen Datenkompressionsprogrammen verglichen. In diesem Kapitel werden die Ergebnisse dieser Untersuchung sowie einige weitere Beobachtungen vorgestellt.

6.1 Einleitung

Bei der Untersuchung von Datenkompressionsprogrammen kann die Beachtung Richtlinien die Aussagekraft und Vergleichbarkeit der Ergebnisse deutlich erhöhen. Arnold und Bell [AB97] haben dabei die wichtigsten Punkte zusammengefasst.

So gibt es neben einer theoretischen Untersuchung des Algorithmus, bei der die Kompressionsrate abhängig von der Entropie der Eingabedaten berechnet werden kann noch die empirische Untersuchung durch praktische Tests und Experimente. Bei dieser können mehrere Eigenschaften des Algorithmus untersucht werden. Neben der Kompressionsrate sind oft auch die Geschwindigkeit und der Speicherverbrauch von Datenkompressionsprogrammen relevant. Geschwindigkeit und Speicherverbrauch können für die Kompression und Dekompression unterschiedlich ausfallen.

Die Wahl der Testdateien hat auch großen Einfluss auf die Ergebnisse einer solchen Untersuchung. Oftmals lassen sich Testfälle finden, für die ein gegebener Algorithmus besonders gut geeignet ist. Dem gegenüber stehen Dateien, für die der Algorithmus vergleichsweise schlecht abschneidet. Aus diesem Grund haben sich einige Korpora von Testdateien etabliert, welche zur Untersuchung neuer Kompressionsverfahren genutzt werden. Witten, Bell und Cleary haben im Jahre 1987 das Calgary Korpus zusammengestellt und in späteren Veröffentlichungen verwendet (siehe [BWC89, BCW90]). Als sich Arnold und Bell 10 Jahre später mit dem Thema beschäftigten, empfanden sie das Korpus als veraltet und entschieden, ein neues Korpus zusammenzustellen. Auf der Webseite des von

[1] Der Quelltext ist auf Anfrage bei Philipp Riegger (Email `philipp@riegger.name`) erhältlich.

ihnen erstellten Canterbury Korpus[2] werden zusätzlich noch das Calgary Korpus und das Large Corpus (eine Sammlung größerer Dateien) bereitgestellt.

Diese Korpora vereinfachen Experimente, da nicht für jedes neuen Kompressionsverfahren von Hand interessante Testfälle zusammengestellt werden müssen. Außerdem sind die Ergebnisse so leichter reproduzierbar, da die Testdateien allgemein zugänglich sind. Für das Canterbury Korpus wurden eine repräsentative Menge unterschiedlicher Dateitypen aus unterschiedlichen Anwendungsgebieten ausgewählt. Die Sammlung ist frei verfügbar und nicht durch das Urheberrecht oder sonstige Regelungen eingeschränkt. Um lange Verzögerungen bei der Verbreitung der Dateien und bei den Tests zu vermeiden, sind die Dateien nicht größer als unbedingt nötig. Zu guter Letzt wurden darauf geachtet, die Dateien nach einem klar definierten Verfahren so auszuwählen, dass sich die Zuverlässigkeit und Nützlichkeit objektiv begründen lässt.

Zur Evaluation der BWTP wurden die Dateien des Calgary, Canterbury und Large Corpus verwendet. Für die Tests stand ein Rechner mit AMD Phenom II Prozessor mit ca. 2 GB freiem Arbeitsspeicher zur Verfügung. Als Betriebssystem kam Gentoo Linux[3] in einer 64-Bit Version zum Einsatz. Da bei der Implementierung nicht speziell auf Geschwindigkeit und Speicherverbrauch geachtet wurde, wird im Folgenden nur die Kompressionsrate untersucht. Die Kompressionsraten sind hier wie allgemein üblich in Bits/Byte angegeben. Der implementierte Algorithmus zur Umkehrung der BWTP wurde anhand von kleinen Testfällen verifiziert, da Zeit und Arbeitsspeicher nicht zur Dekompression aller komprimierten Testdateien ausgereicht haben. Der implementierte Dekompressionsalgorithmus für die Burrows-Wheeler Transformation mit Permutationen ist nicht effizient genug, um in der Praxis verwendbar zu sein, die Burrows-Wheeler Transformation ohne Permutationen kann jedoch problemlos umgekehrt werden.

6.2 Kompressionsrate der Programme

Die Kompressionsrate von `bwt_enc` wurde mit verschiedenen allgemein verbreiteten Kompressionsprogrammen verglichen. Hier kamen neben `bzip2` (siehe Abschnitt 5.1) auch `xz`[4], `gzip`[5] und `lzop`[6] zum Einsatz. Während `bzip2` wie `bwt_enc` die Burrows-Wheeler Transformation verwendet, basiert `xz` auf dem Lempel-Ziv-Markov Algorithmus[7] und ist ein vergleichsweise junges Format. `Gzip` basiert auf dem Deflate Algorithmus und bietet im Schnitt eine etwas schlechtere Kompressionsrate als `bzip2` und `xz`. `Lzop` verwendet das Lempel-Ziv-Oberhumer Verfahren und ist speziell auf eine hohe Geschwindigkeit bei der Dekompression optimiert. Es erreicht dadurch üblicherweise deutlich schlechtere Kompressionsraten.

[2] Canterbury Korpus, URL http://corpus.canterbury.ac.nz/
[3] Gentoo Linux, URL http://www.gentoo.org/
[4] `XZ Utils`, Version 5.0.2, URL http://tukaani.org/xz/
[5] `gzip`, Version 1.4, URL http://www.gzip.org/
[6] `lzop`, Version 1.03, URL http://www.lzop.org/
[7] `XZ` Dateiformat, Version 1.0.4, URL http://tukaani.org/xz/xz-file-format-1.0.4.txt

name	programm	incovation and parameters
xz	XZ Utils	xz -9c infile > outfile
bz2	bzip2	bzip2 -9c infile > outfile
gz	gzip	gzip -9c infile > outfile
lzo	lzop	lzop -9c infile > outfile
bwt	bwt_enc	bwt_enc -9vv -A rle,bwt,mtf,rle,huffman infile > outfile
bwo	bwt_enc	–[8]
bwp	bwt_enc	–[8]

Tabelle 6.1: Verwendete Kompressionsverfahren und Parameter

Die verwendeten Programme mit den genauen Aufrufparametern sind in Tabelle 6.1 aufgelistet. Es wurde jeweils die Blockgröße und damit auch die Kompressionsrate auf das Maximum gesetzt. Dateien mit der Endung `bwt`, `bwo` und `bwp` wurden mit `bwt_enc` komprimiert. Während bei `bwt` feste Parameter angelehnt an den `bzip2` Algorithmus sowie die maximale Blockgröße verwendet wurden, wurden für `bwo` bzw. `bwp` aus einer großen Menge zufälliger Tests für jede Testdatei der beste Parametersatz ohne bzw. mit Permutationen ausgewählt.

6.2.1 Zufällige Tests

Nach dem Scheitern anfänglicher Versuche, durch Intuition und die Analyse der Häufigkeit von Buchstabenpaaren und einzelnen Buchstaben der Testdateien gute Permutationen zu identifizieren, wurde eine große Anzahl von Parametern zufällig ausgewählt und automatisiert getestet.

Insgesamt wurden über 32.000 Parametersätze an zufällig ausgewählten Dateien aus dem Calgary, Canterbury oder Large Korpus getestet und die Größe der komprimierten Datei gespeichert. Neben der Blockgröße wird die Sequenz der Algorithmen variiert und in 80% der Fälle eine Permutation übergeben. Bei der Sequenz der Algorithmen wurden maximal zwei Verarbeitungsschritte vor der BWT und ein bis drei Verarbeitungsschritte zwischen der BWT und der Huffman-Kodierung ausgeführt. Diese wurden zufällig aus `mtf` und `rle` ausgewählt, jedoch wurde vermieden, `rle` mehrmals direkt hintereinander anzuwenden. Die Blockgröße wurde zufällig zwischen 1 und 9 gewählt und für die Permutationen wurden zufällige Zeichen aus der Eingabedatei in ihrem Zustand direkt vor der BWTP ausgewählt und vertauscht.

6.2.2 Calgary Korpus

Der Calgary Korpus besteht aus 14 Dateien mit unterschiedlichen Dateiformaten, unter anderem Texte, Binärdaten und ein Bildformat. Die Kompressionsraten der getesteten

[8] Siehe Abschnitt 6.2 auf der vorherigen Seite.

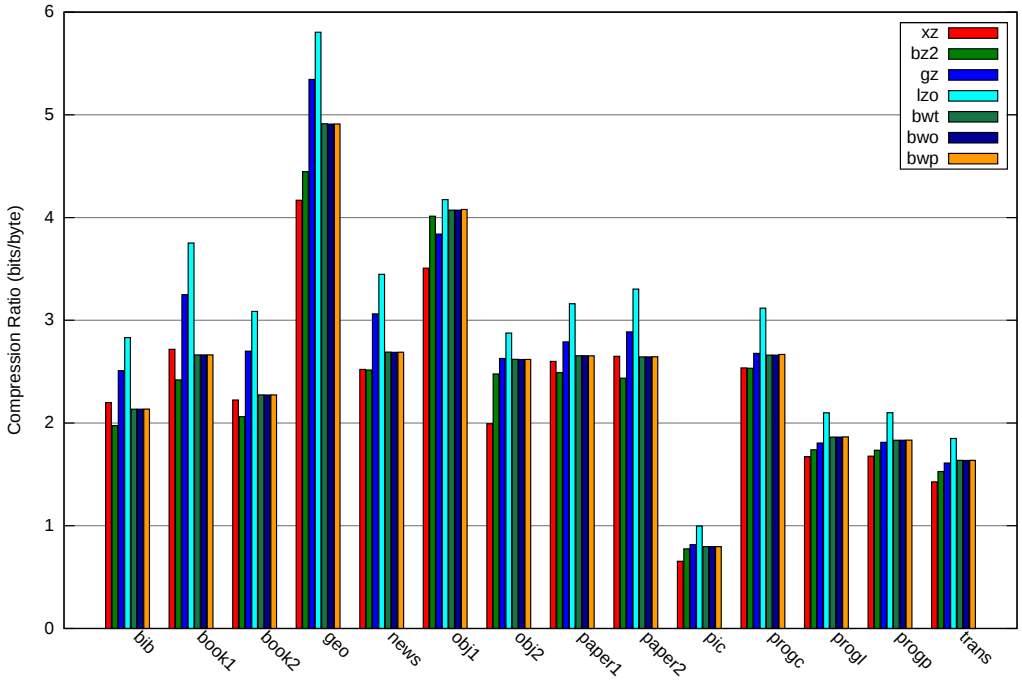


Abbildung 6.1: Kompressionsraten des Calgary Korpus

filename	filesize	xz	bz2	gz	lzo	bwt	bwo	bwp
bib	111261	2.19	1.97	2.50	2.83	2.13	2.13	2.13
book1	768771	2.71	2.42	3.24	3.75	2.66	2.66	2.66
book2	610856	2.22	2.06	2.69	3.08	2.27	2.27	2.27
geo	102400	4.16	4.44	5.34	5.80	4.91	**4.90**	4.91
news	377109	2.52	2.51	3.06	3.44	2.68	2.68	2.68
obj1	21504	3.50	4.01	3.83	4.17	4.07	4.07	4.07
obj2	246814	1.99	2.47	2.62	2.87	2.62	**2.61**	**2.61**
paper1	53161	2.60	2.49	2.79	3.16	2.65	2.65	2.65
paper2	82199	2.64	2.43	2.88	3.30	2.64	2.64	2.64
pic	513216	0.65	0.77	0.81	0.99	0.79	0.79	0.79
progc	39611	2.53	2.53	2.67	3.11	2.66	2.66	2.66
progl	71646	1.67	1.73	1.80	2.09	1.86	1.86	1.86
progp	49379	1.67	1.73	1.81	2.10	1.83	1.83	1.83
trans	93695	1.42	1.52	1.61	1.85	1.63	1.63	1.63

Tabelle 6.2: Kompressionsraten der Dateien des Calgary Korpus

Verfahren sind in Tabelle 6.2 und Abbildung 6.1 auf der vorherigen Seite zu sehen. Die linken vier Balken sind jeweils die Kompressionsraten der angegebenen Vergleichsverfahren, die rechten drei Balken `bwt_enc` mit unterschiedlichen Parametersätzen. Dabei ist zu erkennen, dass `xz` und `bzip2` den ersten Platz für die beste Kompression jeweils unter sich ausmachen. `Gzip` ist mit der Ausnahme von `obj1` bei allen Dateien zu `bwt_enc` gleichauf oder unterlegen. `Lzop` liefert wie zu erwarten die schlechtesten Kompressionsraten. Die unterschiedlichen Parameter für `bwt_enc` machen praktisch keinen Unterschied. Die in der Tabelle erkennbaren minimal besseren Ergebnisse der optimierten Verfahren bei den Dateien `geo` und `obj2` begründen sich auf einen Verzicht der Lauflängenkodierung vor der Burrows-Wheeler Transformation.

6.2.3 Canterbury Korpus

Die Testergebnisse für die 11 Dateien des Canterbury Korpus sind in Tabelle 6.3 und Abbildung 6.2 auf der nächsten Seite zu sehen. Hier hat bei den Dateien `grammar.lsp` und `xargs.1` erstmals `gzip` die beste Kompressionsrate. Wie beim vorigen Test wechseln sich bei den anderen Dateien `xz` und `bzip2` mit der besten Kompressionsrate ab. `Lzop` ist wieder das Schlusslicht. Bei der Datei `sum` führt wie bei einigen vorigen Testfällen der Verzicht auf `rle` als Vorverarbeitungsschritt zu einer minimal besseren Kompressionsrate. Die überragend guten Ergebnisse von `bwt_enc` mit optimierten Parametern bei der Datei `kennedy.xls` sind auf `mtf` als Vorverarbeitungsschritt vor der BWT zurückzuführen.

6.2.4 Large Korpus

Der Large Korpus, dessen Testergebnisse in Tabelle 6.4 und Abbildung 6.3 auf Seite 37 einsehbar sind, liefert keine neuen Erkenntnisse. Hier ist wieder der Verzicht auf `rle` als Vorverarbeitungsschritt für das minimal bessere Ergebnis bei der Testdatei `world192.txt` verantwortlich.

6.3 Auswirkung der Parameter auf die Kompressionsrate

Bei den getesteten Dateien und Parametern schneidet die BWTP schlecht ab. Hier lässt sich durch die Verwendung von Permutationen keine Verbesserung in der Kompressionsrate erzielen. Der Überlegenheit von `bzip2` trotz ähnlichem Verfahren ist durch Optimierungen wie die Verwendung mehrere Huffman-Tabellen zu erklären. Wie die einzelnen Parameter den `bwt_enc` genau beeinflussen, wird im Folgenden genauer untersucht. Dass die Blockgröße eine zentrale Rolle bei der Kompressionsrate spielt wurde schon von Burrows und Wheeler beschrieben [BW94]. Daher wird im folgenden nur auf die Sequenz der Algorithmen und die Permutationen genauer eingegangen.

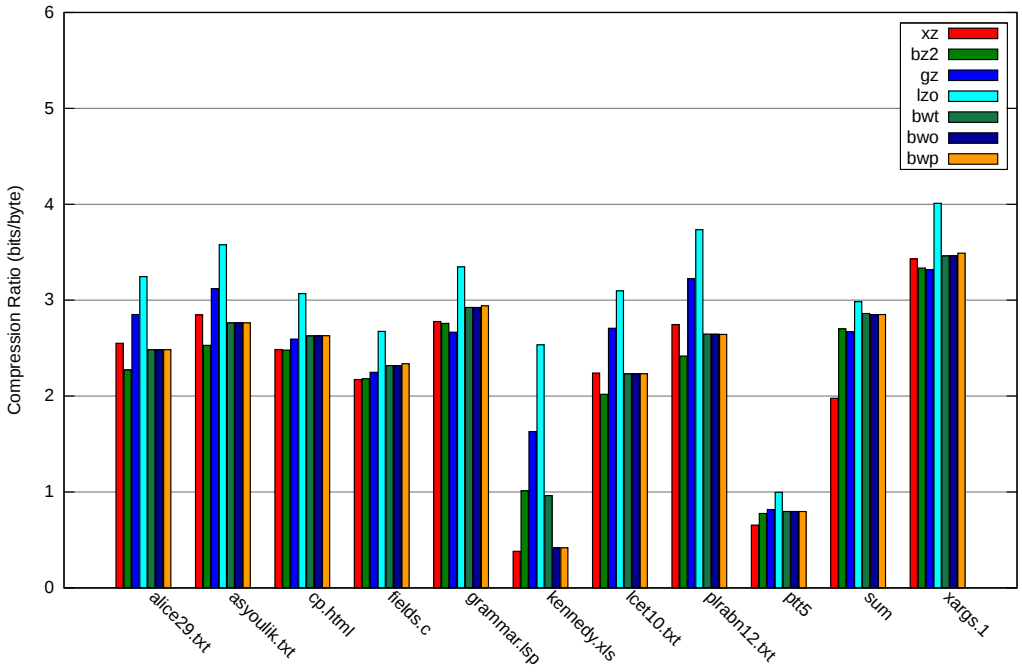


Abbildung 6.2: Kompressionsraten des Canterbury Korpus

filename	filesize	xz	bz2	gz	lzo	bwt	bwo	bwp
alice29.txt	152089	2.55	2.27	2.85	3.24	2.48	2.48	2.48
asyoulik.txt	125179	2.84	2.52	3.12	3.57	2.76	2.76	2.76
cp.html	24603	2.48	2.47	2.59	3.06	2.62	2.62	2.62
fields.c	11150	2.17	2.18	2.24	2.67	2.31	2.31	2.33
grammar.lsp	3721	2.77	2.75	2.66	3.34	2.92	2.92	2.94
kennedy.xls	1029744	0.38	1.01	1.62	2.53	0.96	**0.41**	**0.41**
lcet10.txt	426754	2.24	2.01	2.70	3.09	2.23	2.23	2.23
plrabn12.txt	481861	2.74	2.41	3.22	3.73	2.64	2.64	2.64
ptt5	513216	0.65	0.77	0.81	0.99	0.79	0.79	0.79
sum	38240	1.97	2.70	2.67	2.98	2.86	**2.84**	**2.85**
xargs.1	4227	3.42	3.33	3.31	4.01	3.46	3.46	3.48

Tabelle 6.3: Kompressionsraten der Dateien des Canterbury Korpus

6.3.1 Sequenz der Algorithmen

Abbildung 6.4 auf der nächsten Seite zeigt die Verteilung der Kompressionsraten einer mit `bwt_enc` komprimierten Datei mit verschiedenen Parametern. Die Datei `lcet10.txt` aus dem Canterbury Korpus wurde dabei mit zufällig generierten Kommandozeilenoptionen insgesamt 956 Mal komprimiert. Es kam das in Abschnitt 6.2.1 auf Seite 32 beschriebene Verfahren zur Generierung zufälliger Parameter zum Einsatz. Im Schaubild 6.4(a) sind die relative Häufigkeit verschiedener Kompressionsraten und die Verteilungsfunktion abgebildet. Hierbei ist deutlich erkennbar, dass es zwei Häufungen von Kompressionsraten bei 2,5 Bits/Byte bzw. 5 Bits/Byte gibt. Eine Überprüfung der jeweils verwendeten Parameter ergibt, dass bei der schlechteren Ansammlung von Kompressionsraten jedes Mal eine rotierende Kodierung (`mtf`) vor der Burrows-Wheeler Transformation ausgeführt wurde. Die besten Kompressionsraten ließen sich mit der von `bzip2` genutzten Sequenz `rle,bwt,mtf,rle,huffman` erzielen. Filtert man alle anderen Algorithmensequenzen heraus, bleiben 24 Parametersätze übrig von denen 21 die BWTP mit Vertauschungen nutzen. Das zugehörige Schaubild ist in Abbildung 6.4(b) abgebildet. Hier ist die Streuung deutlich geringer. Bei der schlechteren Hälfte der Parameter wurde die Blockgröße so eingestellt, dass die Datei in mehrere Blöcke zerlegt wurde.

Insgesamt hängt also die Kompressionsrate des Algorithmus primär von der Sequenz der Algorithmen und der gewählten Blockgröße ab. Die Permutationen scheinen hier kaum eine Rolle zu spielen. Aus diesem Grund wird ihr Einfluss im folgenden Kapitel genauer untersucht.

6.3.2 Permutationen

Zur Untersuchung des Einfluss der Permutationen wurde die Datei `E.coli` aus dem Large Korpus ausgewählt. Diese 4638690 Byte Datei enthält das komplette Genom des E. Coli Bakteriums und besteht daher nur aus den Zeichen a, c, g und t. Für Alphabet $\Sigma = \{a, c, g, t\}$ kann die Permutationsgruppe $S(\Sigma)$ berechnet werden, sie besteht aus insgesamt 24 Elementen und hat 23 Untergruppen. Für jede dieser Untergruppen wurde eine Menge erzeugender Permutationen berechnet. Zusätzlich zu den Permutationen wurden die Parameter `-9 -A rle,bwt,mtf,rle,huffman` übergeben. Abbildung 6.5 auf Seite 38 zeigt die Häufigkeitsverteilung der Kompressionsraten sowie die Verteilungsfunktion. Hier gilt zu beachten, dass im Gegensatz zu Abbildung 6.4 nicht das Intervall $[0, 1]$ abgebildet wurde, sondern das Schaubild auf den relevanten Ausschnitt eingeschränkt wurde. Die beste Kompressionsrate erzielte die BWTP mit der trivialen Untergruppe bei einer Kompressionsrate von 2,2136 und einer Größe der komprimierten Datei von 1283684 Byte. Die schlechteste Kompressionsrate trat bei der Verwendung der kompletten Permutationsgruppe auf, hier betrug die Kompressionsrate 2,2408 und die Größe der komprimierten Datei 1299398 Byte. Die Permutationen haben also in diesem Fall die Kompressionsrate um bis zu 1,2% verschlechtert.

Untersuchungen der Kompressionsrate bei anderen Dateien mit unterschiedlichen Kombinationen aus Permutationen und Sequenzen von Algorithmen haben keine Fälle ergeben,

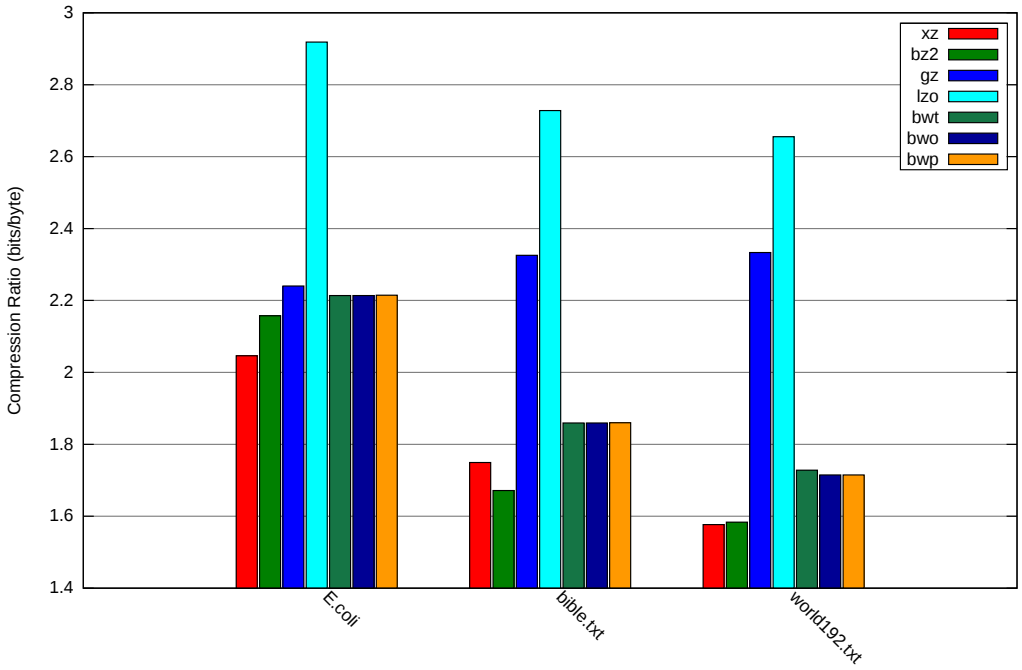
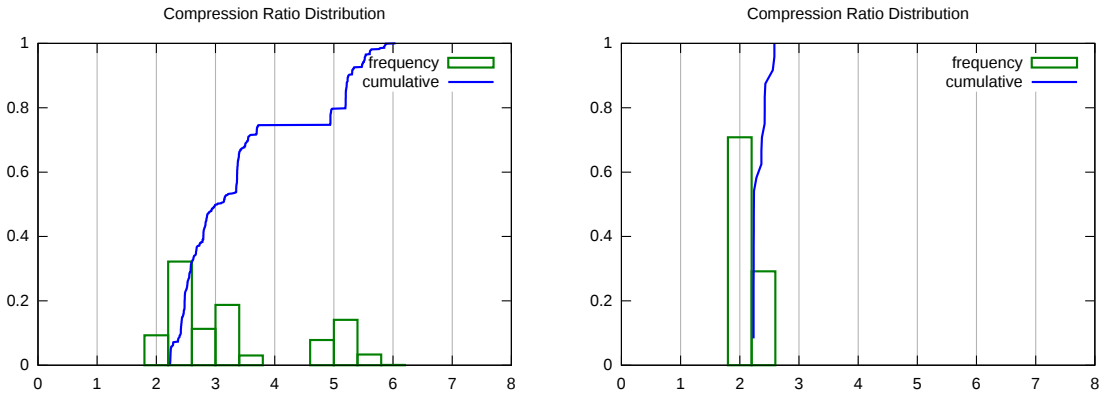


Abbildung 6.3: Kompressionsraten des Large Korpus

filename	filesize	xz	bz2	gz	lzo	bwt	bwo	bwp
E.coli	4638690	2.04	2.15	2.24	2.91	2.21	2.21	2.21
bible.txt	4047392	1.74	1.67	2.32	2.72	1.85	1.85	1.86
world192.txt	2473400	1.57	1.58	2.33	2.65	1.72	**1.71**	**1.71**

Tabelle 6.4: Kompressionsraten der Dateien des Large Korpus



(a) Zufällige Sequenz von Algorithmen

(b) Sequenz von Algorithmen aus `bzip2`

Abbildung 6.4: Verteilung der Kompressionsraten der Datei `cantrbry/lcet10.txt`

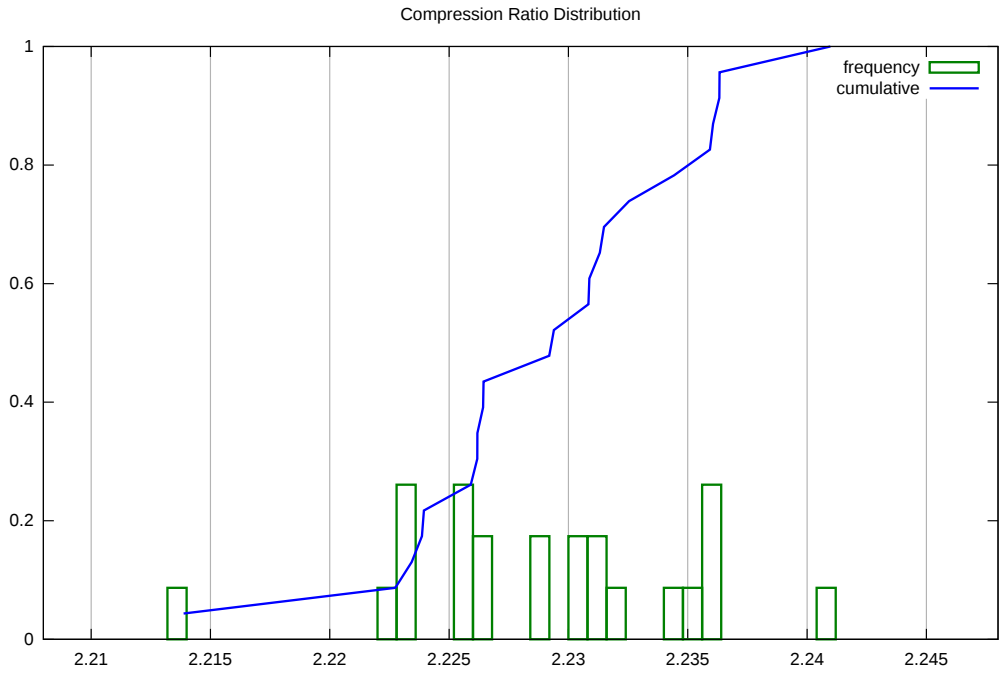


Abbildung 6.5: Verteilung der Kompressionsraten der Datei `large/E.coli`

bei denen die Verwendung von Permutationen zu einer spürbaren Verbesserung der Kompressionsrate führte. Bei Tests mit englischsprachigen Dokumenten und verschiedenen Permutationen wie der Vertauschung häufiger Großbuchstaben mit den entsprechenden Kleinbuchstaben, der Rotation der Vokale oder der Rotation verschiedener Teilmengen der häufigsten Zeichen im Dokument ließen sich größere Unterschiede in den Kompressionsraten beobachten. Jedoch hat der Verzicht auf Permutationen immer zu den besten Ergebnissen geführt.

Kapitel 7

Ausblick

Die untersuchte Variante der Burrows-Wheeler Transformation ist neu und konnte in dieser Arbeit nicht erschöpfend behandelt werden. Sowohl bei den theoretischen Grundlagen als auch bei der Implementierung und Evaluierung mussten auf Grund der zeitlichen Beschränkung Abstriche gemacht werden.

7.1 Grundlagen

Die Idee hinter den Algorithmen zur Transformation und Umkehrung sowie der Korrektheitsbeweis der Umkehrung wurden in Kapitel 4 vorgestellt. Die bekannten Varianten der BWT (siehe [GS09, Kuf09]) auf die BWTP zu übertragen könnte zu schnelleren Algorithmen und einer besseren Kompression führen. Insbesondere die Umsetzung eines bijektiven Verfahrens sollte untersucht werden, da nicht offensichtlich ist, was für Auswirkungen die Verwendung von Permutationen auf die Lyndon Faktorisierung hat.

Für die Umkehrung der BWTP wurde kein schneller Algorithmus vorgestellt. Es wurde begründet, warum sie sich nicht analog zur Umkehrung der BWT auf eine feste Permutation der Positionen zurückführen lässt. Die genauen Unterschiede sowie deren Bedeutung und die Frage, ob ein anderer schneller Algorithmus existiert wurde nicht untersucht.

Die verwendeten Permutationen wurden entweder intuitiv ausgewählt oder nach einer festen Vorschrift zufällig generiert. Die genauen Auswirkungen der Wahl der Permutationen auf die BWTP konnte so nicht ermittelt werden. Eine Analyse dieser Fragestellung könnte zu einem besseren Verständnis führen, welche Permutationen in Kombination mit welchen Eigenschaften einer Testdatei hohe Kompressionsraten erzielen.

7.2 Implementierung

Die Implementierung des auf der BWTP basierenden Datenkompressionsprogramms befindet sich in einem sehr frühen Stadium. An vielen Stellen wurde auf Grund einer einfacheren

Implementierung auf Standardalgorithmen zurückgegriffen. So wurde beispielsweise bisher auf optimierte Suchalgorithmen verzichtet (siehe [BW94, Sew00]).

Die minimalen Permutationen aller Rotationen werden bei der BWTP explizit zu Beginn des Algorithmus berechnet. Dies ist sehr zeitaufwändig und evtl. nicht notwendig. Ein reaktiveres Verfahren wie die Berechnung der minimalen Permutation bei Bedarf während der Sortierung könnte zu einer besseren Laufzeit führen.

Das Format der komprimierten Datei schränkt die Modularität von *bwt_enc* ein. Ein modulares Format für den Kopfteil eines Blockes ermöglicht hier mehr Flexibilität und kann auch die Speicherung unnötiger Daten vermeiden. Bei der Entwicklung des Dateiformat wurde nicht auf Portabilität geachtet, hier kann es beispielsweise zu zu Kompatibilitätsproblemen bei Architekturen mit unterschiedlicher Byte-Reihenfolge (Byte-Order, Endianness) kommen.

Die Implementierung weiterer Algorithmen wie der arithmetischen Kodierung [WNC87] oder anderer Blockkodierungsverfahren [Rez07] würde eine bessere Untersuchung der BWTP ermöglichen.

7.3 Evaluation

Bei der Evaluation wurden verbreitete Korpora von Testdateien verwendet. Die Kompressionsrate des auf dem BWTP Verfahren basierenden Kompressionsprogramm konnte dabei nicht überzeugen. Es besteht jedoch die Möglichkeit, dass bei den inzwischen recht alten Sammlungen Dateien nicht enthalten waren, für die der Algorithmus unerwartet gute Ergebnisse liefert.

Das Verhalten von Permutationen wurde nur an einer Datei mit sehr einfach aufgebautem Inhalt ausgiebig getestet. Hier wären weitere Untersuchungen der Auswirkung von Permutationen hilfreich. Insbesondere über die Auswirkung der Anzahl der verwendeten Permutationen, der Anzahl der Zyklen der Permutationen sowie die Länge der Zyklen.

Kapitel 8

Zusammenfassung

In dieser Arbeit wurde eine Variante der Burrows-Wheeler Transformation (BWT) nach Kufleitner, die Burrows-Wheeler Transformation mit Permutationen (BWTP), vorgestellt und genauer untersucht. Der Algorithmus zur Berechnung der BWTP wurde vorgestellt und auf die durch die Variation entstandenen neuen Herausforderungen eingegangen. Ein konstruktiver Beweis für die Umkehrbarkeit der BWTP wurde gegeben und zum bisher einzigen bekannten Algorithmus zur Umkehrung der BWTP entwickelt. Dieser ist allerdings sehr langsam und ohne weitere Optimierungen in der Praxis nicht verwendbar. Die Probleme bei der Portierung des Verfahrens zur schnellen Umkehrung der BWT auf die BWTP wurden erläutert.

Die im Rahmen dieser Diplomarbeit entwickelte Implementierung der BWTP wurde vorgestellt und die Gründe für die gewählte Architektur wurden dargelegt. Neben den verwendeten Algorithmen, Datenstrukturen und dem Format der komprimierten Datei wurden die über Kommandozeilenoptionen manipulierbaren Parameter des Datenkompressionsprogramms beschrieben, die später zur genauen Analyse verwendet wurden.

Die Leistungsfähigkeit der BWT wurde anhand des Calgary Korpus, Canterbury Korpus und Large Canterbury Korpus untersucht und mir verschiedenen anderen Datenkompressionsprogrammen verglichen. Die Auswirkungen der Permutationen und anderen Parameter auf die Kompressionsrate wurde experimentell ermittelt und an Beispielen dargestellt. Dabei waren bei den getesteten Parameterkombinationen und den verwendeten Testdateien keine Vorteile in der Erweiterung der BWT um Permutationen erkennbar.

Die Ergebnisse dieser Untersuchung sind zwar nicht positiv ausgefallen, jedoch wurde diese vielversprechende Modifikation der BWT nicht erschöpfend bearbeitet und viele der Probleme auf Grund der beschränkten Bearbeitungszeit nicht näher untersucht. In Kapitel 7 sind daher einige Themen aufgezählt, deren genauere Analyse zu einem besseren Verständnis des Verfahrens führen könnte.

Abbildungsverzeichnis

3.1 Berechnung der BWT des Wortes $w = mississippi$ 8
3.2 Berechnung der Matrix M' aus dem transformierten Wort $v = pssmipissii$ . . 10

4.1 Berechnung der BWTP des Wortes $w = mississippi$ 14
4.2 Berechnung der Matrix M' aus dem transformierten Wort $v = ssmipppiii$. 17
4.3 Probleme bei der Umkehrung bei Verzicht auf Abgeschlossenheit 18

5.1 Die Ausgabe von `bwt_enc --help` 26
5.2 Die Ausgabe von `bwt_enc -9vv -o /dev/null files/large/bible.txt` . . 27
5.3 Der Dateikopf einer mit `bwt_enc` komprimierten Datei 27
5.4 Der Kopfteil eines Datenblocks einer mit `bwt_enc` komprimierten Datei . . . 28

6.1 Kompressionsraten des Calgary Korpus 33
6.2 Kompressionsraten des Canterbury Korpus 35
6.3 Kompressionsraten des Large Korpus 37
6.4 Verteilung der Kompressionsraten der Datei `cantrbry/lcet10.txt` 37
6.5 Verteilung der Kompressionsraten der Datei `large/E.coli` 38

Tabellenverzeichnis

6.1 Verwendete Kompressionsverfahren und Parameter 32
6.2 Kompressionsraten der Dateien des Calgary Korpus 33
6.3 Kompressionsraten der Dateien des Canterbury Korpus 35
6.4 Kompressionsraten der Dateien des Large Korpus 37

Algorithmenverzeichnis

3.1 BWT (Burrows-Wheeler Transformation) 9
3.2 BWT^{-1} (Langsame Umkehrung der Burrows-Wheeler Transformation) . . . 11
3.3 BWT^{-1} (Schnelle Umkehrung der Burrows-Wheeler Transformation) 12

4.1 BWTP (BWT mit Permutationen) 14
4.2 $BWTP^{-1}$ (Umkehrung der BWT mit Permutationen) 15

Literaturverzeichnis

[AB97] Arnold, R.; Bell, T.C.: A corpus for the evaluation of lossless compression algorithms. Data Compression Conference, S. 201–210 (1997)

[BK00] Balkenhol, B.; Kurtz, S.: Universal data compression based on the Burrows-Wheeler transformation: Theory and practice. IEEE Trans. Computers 49(10), 1043–1053 (2000)

[BCW90] Bell, T.; Cleary, J.; Witten, I.: Text Compression. Prentice-Hall (1990)

[BWC89] Bell, T.C.; Witten, I.H.; Cleary, J.G.: Modeling for text compression. ACM Comput. Surv. 21(4), 557–591 (1989)

[BSTW86] Bentley, J.L.; Sleator, D.D.; Tarjan, R.E.; Wei, V.K.: A locally adaptive data compression scheme. Commun. ACM 29(4), 320–330 (1986)

[Bos06] Bosch, S.: Algebra, 6. Ausgabe. Springer (2006)

[BW94] Burrows, M.; Wheeler, D.: A block-sorting lossless data compression algorithm. SRC Research Report (1994)

[CFL58] Chen, K.; Fox, R.; Lyndon, R.: Free differential calculus, IV. The quotient groups of the lower central series. The Annals of Mathematics 68(1), 81–95 (1958)

[Fol94] Foldes, S.: Fundamental structures of algebra and discrete mathematics. Wiley-Interscience (1994)

[GS09] Gil, J.; Scott, D.: A bijective string sorting transform (2009). Submitted

[Hoa61] Hoare, C.A.R.: Algorithm 64: Quicksort. Commun. ACM 4(7), 321 (1961)

[Huf52] Huffman, D.: A method for the construction of minimum-redundancy codes. Proceedings of the IRE 40(9), 1098–1101 (1952)

[KR88] Kernighan, B.W.; Ritchie, D.: The C Programming Language, Second Edition. Prentice-Hall (1988)

[Kufo9] Kufleitner, M.: On bijective variants of the Burrows-Wheeler transform. J. Holub, J. Zdárek (Hrsg.) Stringology, S. 65–79. Prague Stringology Club, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague (2009)

[Maco3] MacKay, D.J.C.: Information theory, inference, and learning algorithms. Cambridge University Press (2003)

[Mano1] Manzini, G.: An analysis of the Burrows-Wheeler transform. J. ACM 48(3), 407–430 (2001)

[Rezo7] Reznik, Y.A.: Practical binary adaptive block coder. DCC, S. 399. IEEE Computer Society (2007)

[Sad98] Sadakane, K.: A fast algorithms for making suffix arrays and for Burrows-Wheeler transformation. Data Compression Conference, S. 129–138 (1998)

[Sch97] Schindler, M.: A fast block-sorting algorithm for lossless data compression. Proceedings of the Conference on Data Compression, Bd. 469. Citeseer (1997)

[Sewoo] Seward, J.: On the performance of BWT sorting algorithms. Data Compression Conference, S. 173–182 (2000)

[Sewo1] Seward, J.: Space-time tradeoffs in the inverse B-W transform. Data Compression Conference, S. 439–448 (2001)

[WNC87] Witten, I.H.; Neal, R.M.; Cleary, J.G.: Arithmetic coding for data compression. Commun. ACM 30(6), 520–540 (1987)

Alle Links wurden zuletzt am 20. Mai 2011 besucht.

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

(Philipp M. Riegger)