TECHNISCHE UNIVERSITÄT
CHEMNITZ

Technische Universität Chemnitz
Fakultät für Informatik
Professur Theoretische Informatik

Bachelorarbeit

Burrows-Wheeler-Transformation - Betrachtung und Implementation

Von:

Andy Oertel

Prüfer:

Prof. Dr. Andreas Goerdt

Betreuer:

Dr. Lutz Falke

Chemnitz, 27. Juni 2018

Inhaltsverzeichnis

1. Einleitung

Diese Arbeit beschäftigt sich mit der Burrows-Wheeler-Transformation (BWT) im Bezug auf die Datenkompression. Diese Transformation wurde erstmals 1994 von M. Burrows und D.J. Wheeler in [BW94] veröffentlicht. Kompression kann die BWT aber nur in Kombination mit andern Algorithmen erreichen.
Bevor wir zur Burrows-Wheeler-Transformation kommen, werden wir uns die Motivation für dieses Verfahren anschauen. Besonders betrachten wir die BWT als Teil eines Kompressionsalgorithmus.
Allgemeines Ziel einer Kodierung oder Kompression von Daten ist, dass mit weniger Speicherverbrauch die originalen Daten ausgedrückt werden sollen. Einfache Kodierungsverfahren ordnen den einzelnen Zeichen oder Zeichenketten alternative Zeichen oder Zeichenketten zu, welche möglicherweise kürzer sind. Die BWT versucht diese einfache Methode zu verbessern, indem sie nicht einfach die Zeichen ersetzt. Ziel dieser Transformation ist eine bessere Kompression zu erreichen, aber trotzdem eine Laufzeit zu haben, welche minimal schlechter als linear ist.
Algorithmen, welche beide Kriterien erfüllen, sind stark gefragt, um die immer größeren Datenmengen schnell zu archivieren. Um Speicherplatz zu sparen werden diese Daten oft komprimiert. Doch es müssen bei einer der beiden Eigenschaften oft Abstriche gemacht werden. Denn bei den bekannten Algorithmen ist entweder die Laufzeit oder die Kompression besser. Mit der BWT existiert für beide Kriterien ein gutes und oft genutztes Mittelmaß.
Ziel dieser Arbeit ist die Burrows-Wheeler-Transformation vorzustellen und ausführlich zu erklären. Besonderes Augenmerk liegt auf der Erklärung der Funktionsweise und der Effizienz der Kompression. Mit diesen Grundlagen wird danach besprochen, wie die BWT implementiert wird. Zum Ende der Arbeit wird anhand von Beispieldaten die Güte der Kompression für verschiedene Implementationen der BWT analysiert. Besonders wir der Unterschied zwischen Bytes und Bits als Größe eines Zeichen betrachtet.
Dies ist zum Beispiel der Fall bei Text, wo die Buchstaben als Zeichen angesehen werden. Diese haben in der Regel die Größe von einem Byte. Alternativ kann auch die binäre Kodierung dieser Buchstaben betrachtet werden. So kann der Algorithmus auf die Buchstaben oder auch der binären Darstellung dieser angewendet werden.

2. Vorbetrachtungen

2.1. Grundlegende Begriffe und Notation

Wir betrachten *Daten* als eine Folge von Zeichen aus einem Alphabet. Dabei ist die Bezeichnung als Folge wichtig, da die Reihenfolge der Zeichen entscheidend ist.
Ein *Zeichen* ist ein atomares Element, welches gespeichert wird. Es ist sozusagen in einer Folge das kleinste nicht weiter unterteilbare Element. Diese Zeichen kommen aus einer Menge, welche *Alphabet* genannt wird.
Das Wort *speichern* bezeichnet in unserem Zusammenhang alles, um eine Zeichen oder Zeichenfolge zu repräsentieren, darzustellen oder zu kodieren. Es ist nicht nur der Prozess des aktiven Schreibens in den Speicher gemeint.
Text in Schreibmaschinenschrift gibt an, dass der Text ein Zeichen oder eine Zeichenfolgen ist. Manchmal sind die Zeichen in einer Zeichenfolge durch Kommata getrennt. Das

Komma ist kein Zeichen, sondern dient nur der besseren Darstellung der Zeichenfolge. Zum Beispiel ist `A,B,\0,AB,z` eine Zeichenfolge mit fünf Zeichen.

2.2. Grundlegende Informationstheorie

Im Wort Informationstheorie ist der Begriff der *Information* enthalten, welcher für diesen Zusammenhang erstmals geklärt werden muss. Die Bedeutung dieses Begriffs hilft beim Verständnis der theoretischen Aussagen. Das Konzept der Information bezieht sich auf Daten. Dieser Begriff ist, entgegen der umgangssprachlichen Bedeutungen, nicht der Inhalt oder die Aussage von Daten. Es ist damit mehr eine Aussage über das Datum im Vergleich mit den möglichen Daten. Also werden die Daten auf einer sehr niedrigen Ebene und ohne allgemeinen Kontext betrachtet.
Wir betrachten Datum x als Folgen von Zeichen $x_i \in \Sigma$, welche eine feste Reihenfolge haben. Wir werden nur binär kodierte Daten in Form von Bits betrachten, welche das Alphabet $\{0, 1\}$ als Grundlage haben. Dieses Alphabet wird auch betrachtet, wenn es mehr als zwei mögliche Zeichen gibt. Solche Zeichen werden als eine Zeichenkette fester Länge mit einer entsprechender Bedeutung binär abgespeichert. Ein Beispiel dafür sind Buchstaben, welche mit 8 Bits langen Zeichenketten pro Zeichen kodiert werden. Im Folgenden sind mit dem Begriff Zeichen auch immer solche Zeichenketten gemeint.
Jetzt können wir auch die Information genauer definieren. Des Weiteren werden wir auch noch die eng damit zusammenhängende Begriffe des *Informationsgehaltes* und der *Entropie* klären.

Definition 2.1 (k-te Information)**.** *Die k-te Information von Daten bezeichnet die Statistik eines Zeichens des Datums. Dabei ist das jeweilige Zeichen abhängig von den k Zeichen vor dem aktuellen Zeichen. Es ist $k \geq 0$. Je ungenauer die Vorhersage des Zeichens ist, umso größer ist die Information.*

Um diese Definition zu verstehen, betrachten wir als Beispiel die Folge `ABABABABAB...` als Datum. Bei der Betrachtung der 0-ten Information kann ohne Betrachtung der vorherigen Zeichen nicht festgestellt werden, welches das kommende Zeichen ist. Das Zeichen `A` und `B` sind gleich wahrscheinlich. Bei Betrachtung der 1-ten Information kann sofort mit dem vorherigen Zeichen gesagt werden, welches Zeichen folgen wird. Für den Fall der 0-ten Information ist die Information des aktuellen Zeichens von größeren Wert, da wir das Zeichen nicht vorhersehen können.
Die folgende Definition wurde so ähnlich von Claude E. Shannon in [Sha01] erstmals erwähnt.

Definition 2.2 (Informationsgehalt (Entropie) nach C. E. Shannon)**.** *Der Informationsgehalt eines Zeichens c aus einem Alphabet Σ wird wie folgt beschrieben.*

$$I(c) := \log_{|\Sigma|}\left(\frac{1}{p_c}\right)$$

mit Σ die Menge (Alphabet), woraus die Zeichen gewählt werden und p_c die relative Häufigkeit des Zeichens c im Datum.
Die Entropie einer Zeichenfolge s mit n verschiedenen Zeichen ist wie folgt definiert.

$$H_0(s) := \sum_{c \in \Sigma} H(c) \cdot I(c)$$

mit $H(c)$ die absolute Häufigkeit des Zeichens c.

Dieser Begriff von Entropie nach Shannon betrachtet keine Zeichen in der Umgebung des aktuell betrachteten Zeichens. Es wird die Information kontextlos betrachtet. Es gibt auch noch einen Begriff von der Entropie, welcher auch die k Zeichen vor dem aktuellen Zeichen betrachtet. Diese Definition lässt sich auch in [Man01] finden.

Definition 2.3 (k-te empirische Entropie)**.** *Die k-te empirische Entropie einer Zeichenfolge s und $k \geq 0$ ist*

$$H_k(s) := \frac{1}{|s|} \sum_{w \in \Sigma^k} |w_s| H_0(w_s).$$

Σ^k ist jeder Folge der Länge k aus Zeichen aus dem Alphabet Σ. $H_0(w_s)$ ist der Informationsgehalt der Folge w_s. Die Notation w_s ist die Folge der Zeichen, welche bei jedem Auftreten der Zeichenfolge w direkt davor stehen.

An einem Beispiel lässt sich Notation w_s kurz verdeutlicht. Für die Zeichenfolge $s =$ `abcbab` und $w =$ `b` ist die Folge $w_s =$ `a,c,a`.
Bei binären Daten wird die Einheit *Bits* für die Entropie verwendet. Es gibt für ein Zeichen oder eine Zeichenfolge an, wie viele Bits mindestens benötigt werden, um diese zu kodieren. Dazu müssen die k Zeichen vor dem aktuellen Zeichen betrachtet werden. Der Informationsgehalt $H_0(s)$ ist damit die untere Schranke an Bits, welche für die Kodierung eines Zeichens ohne Kontext benötigt wird.
Es lässt sich auch die Verwendung des Logarithmus der relativen Häufigkeit hier erkennen. Pro Bit wird die Anzahl der möglichen Bitfolgen verdoppelt. Damit wachsen die Möglichkeiten der Werte exponentiell zur Anzahl der Bits.
Etwas umgangssprachlicher formuliert ist die Entropie ein Maß für die Freiheit der Wahl eines Zeichens. Ist die allgemeine Wahrscheinlichkeit für ein Zeichen hoch, so tritt es auch im aktuellen Fall mit hoher Wahrscheinlichkeit auf. Damit ist dieses Zeichen sehr erwartet an dieser Stelle und bringt nicht viel Information. Ein wenig häufig auftretendes Zeichen ist hingegen etwas besonders und zeigt einen Gegensatz zu den sonst erwarteten Zeichen. Damit ist auch die Information hoch. Das Ganze gilt sowohl für die Betrachtungen ohne Kontext als auch bei der Betrachtung der k Zeichen vor dem aktuellen Zeichen.
Es gibt auch Definitionen, welche beachten, dass die Entropie manchmal Null ist. Das ist der Fall bei einer Folge genau gleicher Zeichen. Diese liefern eine bessere Abschätzung, welche wir aber nicht benötigen. Wir werden deshalb diese Definition nicht betrachten. Die Definitionen werden in [Man01] erwähnt.

2.3. Kompression

Ziel einer Kompression ist den Speicherverbrauch von Daten zu minimieren. Das wird durch Transformation in andere Daten erzielt.
Es wird allgemein zwischen der verlustbehafteten und verlustfreien Kompression unterschieden. Die unterscheidende Eigenschaft ist, ob nach der Rücktransformation wieder die gleichen originalen Daten vorhanden sind. Bei verlustbehafteter Kompression müssen die Daten nach der Dekompression nicht mehr genau den originalen Daten entsprechen. Sie können also leicht bis beliebig stark abweichend sein. Es sollten aber immer noch die originalen Daten in etwa erkennbar sein. Verlustfreie Kompressionen müssen nach der Rücktransformation die originalen Daten wieder zurückliefern.

In dieser Arbeit werden nur Kompressionen betrachtet, welche verlustfrei sind. Es gibt auch keinen einfachen Bezug der BWT zur verlustbehafteten Kompression, weswegen diese irrelevant für diese Arbeit ist.
Einige wichtige Kompressionsverfahren, welche auch später in dieser Arbeit Erwähnung finden, werden im Folgenden kurz vorgestellt. Diese werden im Zusammenhang mit der BWT benutzt. Dabei wird nicht genauer auf den Beweis der Korrektheit dieser eingegangen. Es wird rein die Funktionsweise erklärt und durch Beispiele verdeutlicht.

2.3.1. Entropie-Kodierer

Entropie-Kodierer orientieren sich am Informationsgehalt des Zeichens. Diese Klasse an Verfahren erhält ihre Bezeichnung, durch die auch verwendetet Bezeichnung als Entropie. Ziel von diesen Kodierern ist es, dass Zeichen mit niedrigen Informationsgehalt auch mit wenigen Bits kodiert werden.
Beispiele dafür sind die Huffman-Kodierung und die Klasse der arithmetischen Kodierer. Diese werden wir nun näher betrachten.

Huffman-Kodierung Die Kodierung wurde in grober Form von David A. Hufmann in [Huf52] erstmals erwähnt. Die genaue Funktionsweise ist dort bereits erkennbar.
Bei der Huffman-Kodierung werden lediglich die Kodierungen von Zeichen ersetzt. Dabei werden die vorherigen Kodierungen gleicher Länge mit Kodierungen unterschiedlicher Länge ersetzt. Diese Länge wird auf Basis der Häufigkeit des speziellen Zeichens ermittelt. Ein häufiges Beispiel ist die Kodierung von Buchstaben in einem Text. Nach der Häufigkeit des Buchstabens in Relation zu den anderen wird jedes auftreten dieses Buchstaben durch eine entsprechende Bitfolge kodiert. Es wird dabei aber pro Kodierung immer mindestens ein Bit verwendet, da die festen Kodierung nur ersetzt wird.
Zur Umsetzung dieser Kodierung werden die Häufigkeiten der Zeichen und binäre Bäume genutzt. Das Verfahren, welches zur Erstellung dieses Baumes genutzt wird, ist in Algorithmus 1 dargestellt. Der Schritt des Zusammenfügens der Bäume wird in Abbildung 1 bildlich dargestellt. In dieser Abbildung sind x und y die kleinsten Häufigkeiten in den Wurzeln.

Algorithmus 1: Erstellung des Baums für die Huffman-Kodierung.

```
1 Erstelle für jedes Zeichen einen Baum mit der Häufigkeit in der Wurzel und einem
    Blattknoten mit dem Zeichen als Wert;
2 solange mehr als ein Baum noch übrig tue
3 |   Wähle die zwei Bäume mit der niedrigsten Häufigkeit in der Wurzel;
4 |   Fasse durch einen neuen Wurzelknoten die beiden Bäume zusammen;
5 |   Speichere in der Wurzel die addierten Häufigkeiten;
6 Ende
```

Aus diesem Baum wird die Kodierung erstellt. In den Blättern des Baums stehen die Zeichen. Von der Wurzel aus wird analysiert, ob in einem Knoten der rechte oder der linke Teilbaum gewählt wird, um das Zeichen zu erreichen. Wenn der linke Teilbaum gewählt wird, dann wird eine `0` zu der Kodierung hinzugefügt und sonst eine `1`. Nachdem alle Kodierungen bestimmt sind, werden die Auftreten des jeweiligen Zeichens mit dieser

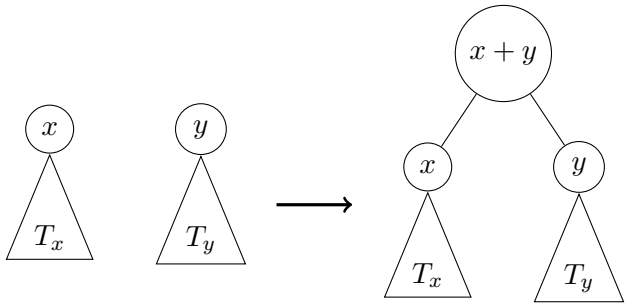

Abbildung 1: Darstellung des Zusammenfügens der Bäume aus Algorithmus 1.

Kodierung ersetzt. Es wird weiterhin noch die entsprechende Zuordnung zwischen Zeichen und Kodierung gespeichert, um dies zur Rücktransformation zu nutzen.
In Kapitel 3.7 wird ein genaues Beispiel mit Beispieldaten dargestellt.

Arithmetische Kodierung Arithmetische Kodierer nutzen ein Intervall und die Häufigkeiten von Zeichen, um die Intervallgrenzen nach dem auftretenden Zeichen anzupassen. Das Startintervall wird vorher festgelegt. Dem Kodierer stehen die absoluten oder relativen Häufigkeiten jedes Zeichens zur Verfügung. Für die Zeichen existiert außerdem eine Ordnungsrelation.
Der Kodierer liest die Daten in Reihenfolge ein. Dabei wird immer ein Zeichen betrachtet und entsprechend diesem das Intervall verändert. Nachdem die Folge komplett betrachtet wurde, wird eine Zahl aus dem Intervall kodiert. Durch diese Zahl wird die Folge repräsentiert.
Am besten lässt sich das Verfahren an einem Beispiel-Kodierer mit Beispiel-Eingabe verdeutlichen. Das Startintervall sei $[0, 1]$ und die Häufigkeiten seien ausgedrückt als eine relative Häufigkeit. Die Eingabe ist `AABBAAAA`. Die Anpassung des Intervalls findet immer so statt, dass das aktuelle Intervall so eingeteilt wird, wie die Häufigkeiten der Zeichen sind. Dies wird am Beispiel klar und in Abbildung 2 nochmal verbildlicht.
Das resultierende sortierte Alphabet mit relativen Häufigkeiten ist in der Tabelle 1 dargestellt.

Buchstabe	relative Häufigkeit
A	0,75
B	0,25

Tabelle 1: Häufigkeiten für das Beispiel

Nun wird die Eingabe Zeichen für Zeichen betrachtet und das Intervall entsprechend der relativen Häufigkeit und Sortierung aktualisiert. Dies wird in der folgenden Tabelle 2 dargestellt.

gelesenes Zeichen	untere Schranke	ober Schranke
A	0	0,75
A	0	0,5625
B	0,421875	0,5625
B	0,52734375	0,5625
A	0,52734375	0,5537109375
A	0,52734375	0,547119140625
A	0,52734375	0,54217529296875
A	0,52734375	0,5384674072265625

Tabelle 2: Beispiel Arithmetische Kodierung.

In der Abbildung 2 werden die ersten vier Schritte nochmal bildlich dargestellt. In einer Zeile ist jeweils ein Schritt dargestellt und das aktuell betrachtete Intervall wird immer bildlich vergrößert. Dadurch sind die wirklichen Grenzen des Intervalls nicht erkennbar und an den Seiten der Rechtecke angeführt.

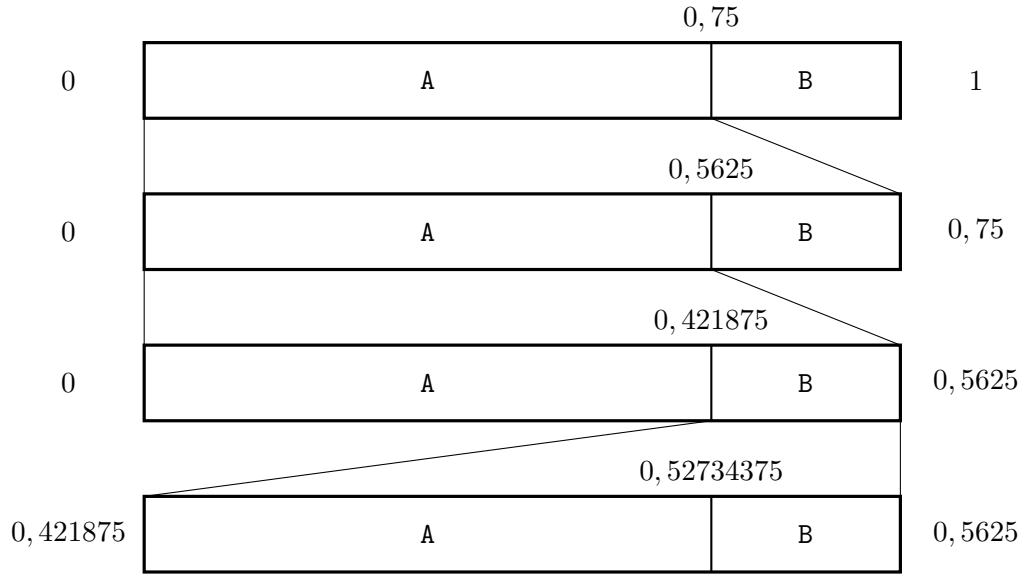

Abbildung 2: Darstellung des Intervalls als Rechteck mit einem Schritt pro Zeile für das Beispiel `AABBAAAA`. Nur die ersten vier Schritte werden hier dargestellt.

Nun wird zur Kodierung irgendeine Zahl aus dem Intervall gewählt. Diese Zahl kann und sollte mit möglichst wenig Bits zu kodieren sein. Die so optimal Zahl aus dem Intervall $[0,52734375; 0,5384674072265625]$ ist $0,52734375$. Die Kodierung dieser Zahl lautet somit `10000111` und braucht damit 8 Bits. Wichtig bei der Kodierung und der Wahl der Zahl ist, dass es egal ist, ob Nullen oder Einsen an die Kodierung angehängt werden. Der Wert muss trotzdem noch im Intervall sein. Dieser Punkt wird benötigt, wenn mehrere Kodierungen hintereinander geschrieben werden. Es muss trotzdem erkannt werden, wann einen neue Bitfolge beginnt.
Des Weiteren muss das Alphabet mit den Häufigkeiten gespeichert werden. Damit sind alle Daten für die Rücktransformation vorhanden. Bei dieser wird die gespeicherte Zahl

rekonstruiert und dann wird nachvollzogen in welchem Intervall diese Zahl liegt. Das Intervall entspricht dann der originalen Zeichenfolge.

2.3.2. Move-to-front-Kodierungsverfahren

Die Move-to-front-Kodierung (MTF) verändert die Länge der Kodierung im Gegensatz zu den vorherigen Verfahren nicht. Die MTF ersetzt jedes Zeichen durch eine gleichgroß kodiertes Zeichen. Das Ziel ist, dass aus einer Zeichenfolge mit vielen gleichen direkt hintereinander auftretenden Zeichen eine Folge mit vielen Nullen wird. Die Zeichen sind in der Eingabe nicht global gleich, sondern nur lokal gleich, damit die Transformation sinnvoll ist. Das heißt, dass gleiche Zeichen nah beieinander stehen, aber in der ganzen Folge verschiedene Zeichen stehen. Die Folge in der Ausgabe soll dann global gleich sein. Eine Erläuterung der Kodierung ist auch in [BW94] zu finden. In Algorithmus 2 ist die MTF-Kodierung beschrieben.

Algorithmus 2: Move-to-front-Kodierung.

```
1 Erzeuge aus dem Alphabet die sortierte Folge alpha;
2 für jedes Zeichen z in der Eingabefolge tue
3 |   Füge zur Ausgabefolge die Position von z in alpha hinzu;
4 |   Lösche z aus alpha;
5 |   Füge z an der ersten Position von alpha hinzu;
6 Ende
7 Gebe die Ausgabefolge und das Alphabet aus;
```

Um diese Funktionsweise zu verdeutlichen werden wir kurz ein Beispiel behandeln. Es wird das Wort `BANANE` kodiert.

Alphabet	Zeichen	gespeicherte Position	aktualisiertes Alphabet
ABEN	B	1	BAEN
BAEN	A	1	ABEN
ABEN	N	3	NABE
NABE	A	1	ANBE
ANBE	N	1	NABE
NABE	E	3	ENAB

Tabelle 3: Beispiel Move-to-front-Kodierung.

Die Ausgabe des Kodierungsverfahren ist das Alphabet `A,B,E,N` und die Folge `1,1,3,-1,1,3`.
Es gibt auch noch eine Abwandlung der MTF-Kodierung, welche als MTF-1-Kodierung bezeichnet wird. Die Besonderheit dieser Transformation ist, dass ein Zeichen im Alphabet nicht sofort an die erste Stelle verschoben wird. Es wird nur an die erste Stelle des Alphabetes verschoben, wenn es an der ersten Stelle steht oder das zweite Mal in Folge auftritt. Sonst wird das Zeichen nur an die zweite Stelle verschoben. Diese Veränderung soll vermeiden, dass ein unterschiedliches Zeichen in einer Folge gleicher Zeichen zu weniger Nullen führt.
Am Beispiel der Folge `A,A,B,A,A,B,A,A` lässt sich das verdeutlichen. Das sortierte Al-

phabet lautet dabei `A,B`. Die transformierte Folge mit der MTF-Kodierung lautet `0,0,1,1,0,1,1,0`. Mit der MTF-1-Kodierung ergibt sich `0,0,1,0,0,1,0,0`. Damit ist die MTF-1-Kodierung hier besser, da das Ziel ist möglichst viele Nullen zu haben. Es gibt aber auch Gegenbeispiele, welche ein besseres Ergebnis bei der MTF-Kodierung liefern.

2.3.3. Lauflängenkodierung

Das allgemeine Konzept hinter der Lauflängenkodierung besteht darin, die Lauflänge eines Zeichens als Zahl zu kodieren. Die Lauflänge ist die Anzahl der direkt hintereinander auftretenden gleichen Zeichen. Weiter muss noch die Information über das aktuell betrachtete Zeichen kodiert werden. Es gibt verschiedene Varianten, wie genau diese Kodierung realisiert wird. Im Folgenden werden einige verschiedene Arten der Lauflängenkodierung vorgestellt. Die Varianten können teilweise auch kombiniert werden.

Allgemeine Lauflängenkodierung Ein zusammenhängendes Auftreten eines Zeichens wird als das Zeichen gefolgt von der Lauflänge kodiert. Dabei wird vorher festgelegt, wie viele Bits zur Kodierung der Lauflänge genutzt werden. Ohne diese Festlegung ist es schwierig zu entscheiden, wann das nächste Zeichen beginnt.
Zum Beispiel wird `AABBAAAA` als `A,2,B,2,A,4` kodiert. Alle ungeraden Positionen in der Folge sind die Zeichen und die geraden Positionen sind die Lauflängen.

Lauflängenkodierung ab bestimmter Lauflänge Es kann auch effizient sein, erst ab einer bestimmter Lauflänge die Länge zu kodieren. Dazu muss vorher eine Länge k festgelegt werden, ab der begonnen wird die Lauflänge zu kodieren. Für die Kodierung der Anzahl werden oft genau so viele Bits verwendet wie für die anderen Zeichen.
Zum Beispiel wird mit der Länge $k = 4$ aus der Folge `AAAAA` die Folge `AAAA\1`, wobei `\1` das Zeichen mit der Wertigkeit 1 ist. Alle kürzeren Folgen von Zeichen werden einfach kopiert und bleiben unverändert. Das Grenzbeispiel für $k = 4$ ist die Folge `AAAA`, welche als `AAAA\0` kodiert wird. Der Vorteil ist, dass das aktuell betrachtete Zeichen bereits kodiert wurde.

Lauflängenkodierung mit fester Reihenfolge Bei kleineren Alphabeten wird eine feste Reihenfolge der aktuellen Zeichen gewählt. So muss das aktuelle Zeichen nicht extra kodiert werden. Als Beispiel betrachten wir ein Alphabet mit den Zeichen `A,B,C`. Mit `A` beginnend werden die Zeichen aufsteigend betrachtet und dann wird wieder mit `A` begonnen. Es wird immer die Lauflänge des aktuell betrachteten Zeichens kodiert. Dabei kann es auch die Lauflänge Null auftreten.
Als Beispiel kodieren wir die Folge `AACCAABBBB`. Das Ergebnis der Kodierung ist `2,0,2,2,4`. Das bedeutet, dass zuerst das Zeichen `A` mit Lauflänge 2 auftritt. Dann tritt das Zeichen `B` mit Lauflänge 0 auf, und so weiter.

Binäre Lauflängenkodierung Ein oft benutzter Spezialfall des vorherigen Verfahrens ist, wenn die Größe des Alphabetes zwei ist. Dann wird abwechseln die Lauflänge der beiden Zeichen kodiert.
Das Beispiel von `AABBAAAA` ergibt somit die Kodierung `2,2,4`. Es wird dabei mit dem Zeichen `A` begonnen.

MTF Lauflängenkodierung Die Kodierung hat ihren Namen von der Verwendung nach einer MTF-Kodierung. Es wird nur eine Lauflängenkodierung auf das Zeichen `0` angewendet. Dieses Zeichen gibt es nach der MTF-Kodierung oft, wie schon in Kapitel 2.3.2 erwähnt wurde. Damit ist es sinnvoll nur die Lauflänge der Nullen zu kodieren und die restlichen Zeichen unverändert zu lassen. Es ist wichtig das bei der Kodierung der Lauflänge neue Zeichen verwendet werden, welche sonst nicht in der Folge auftauchen. Allgemein kann dieses Verfahren auch auf mehrere Zeichen angewendet werden.

Lauflängenkodierung mit binärer Kodierung der Zahlen Dieses Verfahren ist eine Alternative zur Einführung vieler verschiedener neuer Zeichen, welche die Lauflängen repräsentieren. Mit zwei neuen Zeichen wird die Lauflänge binär kodiert. Dabei werden pro Zeichen zwei neue Zeichen eingeführt. Der große Vorteil besteht darin, dass vorher keine festen und damit maximalen Größen für die Lauflängen festgelegt werden müssen. Es kann auf die Kodierung der Null verzichtet werden, wenn keine Lauflänge der Größe Null auftreten kann. Das ist im Allgemeinen der Fall.
Bei einer oft verwendete Kodierung hat das Zeichen an erster Stelle den niedrigsten Wert. Es wird mit Basis 2 kodiert wie bei Binärzahlen. Durch den Verzicht auf die Kodierung der Null, haben die beiden Zeichen die Wertigkeit 1 und 2 statt 0 und 1.
Die Beispielkodierung der Zahl 42 ist `BBABA`, wobei das Zeichen `A` eine Wertigkeit von 1 und `B` eine Wertigkeit von 2 hat. Multipliziert mit den Wertigkeiten der Stellen ergibt sich die Summe $2 \cdot 1 + 2 \cdot 2 + 1 \cdot 4 + 2 \cdot 8 + 1 \cdot 16 = 42$ und damit der richtige Wert.

Probleme bei konstanter Größe der Lauflänge Es ist sehr schwierig stark unterschiedlich große Zahlen effizient zu kodieren. Der Grund ist, dass bekannt sein muss, welche Bitfolge zu welcher Zahl gehört. Besonders schwierig ist es, wenn die Zahlen ohne eine klare Trennung direkt hintereinander stehen. Deswegen wird oft eine konstante Länge der Kodierung gewählt. Bei der Wahl von zu wenig Bits pro Zahl werden große Zahlen als viele maximal große Zahlen kodiert. Bei der Wahl von zu vielen Bits werden für die Kodierung von kleinen Zahlen unnötig viele Bits verwendet.
Die optimale Anzahl der Bits kann unter Betrachtung der Daten vorher ermittelt werden. Dieser Prozess ist aber auch wieder mit viel Aufwand verbunden.
Eine gute Strategie ist es, die Zahlen zuerst mit möglichst vielen Bits zu kodieren. Auf das Ergebnis wird dann eine Entropie-Kodierung angewendet. Durch dieses Vorgehen kann erreicht werden, dass im Anschluss durch die Entropie-Kodierung nur so viele Bits wie nötig gebraucht werden.

3. Die Burrows-Wheeler-Transformation

Bevor wir zur Transformation kommen, ist noch der Zweck dieser zu klären. Die Burrows-Wheeler-Transformation (BWT) verändert die benötigte Speichergröße der Daten nicht. Die Transformation permutiert nur die Zeichen in einer Zeichenfolge. Somit ist das Ziel keine Kompression, sondern eine Vorbereitung der Folge für andere Verfahren. Dabei ist das Ziel, dass andere Kompressionsverfahren diese umsortierten Daten besonders gut komprimieren können. Auch muss die BWT nicht unbedingt für Kompression genutzt werden, obwohl es der größte Anwendungsbereich ist.
In Zusammenhang mit anderen Verfahren soll sie Daten besser komprimieren als anderer Algorithmen mit noch guter Laufzeit. Natürlich gibt es Verfahren, welche mit höherer

Laufzeit bessere Kodierungen finden.
Im Folgenden wird die BWT kurz vorgestellt und an einem Beispiel verdeutlicht. Auf die genauen Hintergründe der Funktionsweise und den Sinn der Permutation wird danach eingegangen. Die Gründe lassen sich nicht ohne ein grundlegendes Verständnis über die Transformation erkennen.

3.1. Die Transformation

Zuerst folgt eine Beschreibung der einzelnen Schritte des Algorithmus im Pseudocode. Diese Schritte werden im Folgenden genauer erklärt. Die Eingabe ist eine Folge von Zeichen. In Algorithmus 3 wird das Verfahren beschrieben.

Algorithmus 3: Grobe Burrows-Wheeler-Transformation.

Eingabe : originale Folge von Zeichen
Ausgabe: transformierte Folge von Zeichen und Index der Zeile mit originaler Folge

1 Erzeuge alle möglichen Rotationen der Eingabefolge;
2 Sortiere diese Rotationen;
3 Erzeuge die Ausgabe als Folge der letzten Zeichen in den Rotationen entsprechend der Sortierung;
4 Gebe mit der Ausgabe auch den Index aus in welchem die gleiche Rotation steht wie in der Eingabe;

Zu Zeile 1 sei kurz erklärt, was eine Rotation im Zusammenhang mit der BWT ist. Einen Rotation ist eine Funktion, welche als Eingabe eine Folge von Zeichen und einem Index aus der Folge hat. Die Ausgabe ist eine gleichlange Folge von Zeichen. Der erste Teil der Ausgabefolge ist die Teilfolge der Eingabefolge vom Index bis zum Ende der Folge. Der zweite Teil ist die Teilfolge vom Anfang bis zum Zeichen vor dem Index in der Eingabefolge. Die beiden Teile werden einfach aneinandergehängt. In Abbildung 3 ist das Verfahren skizziert. Ein Rechteck symbolisiert dabei eine Folge und der senkrechte Strich die Trennung am Index.

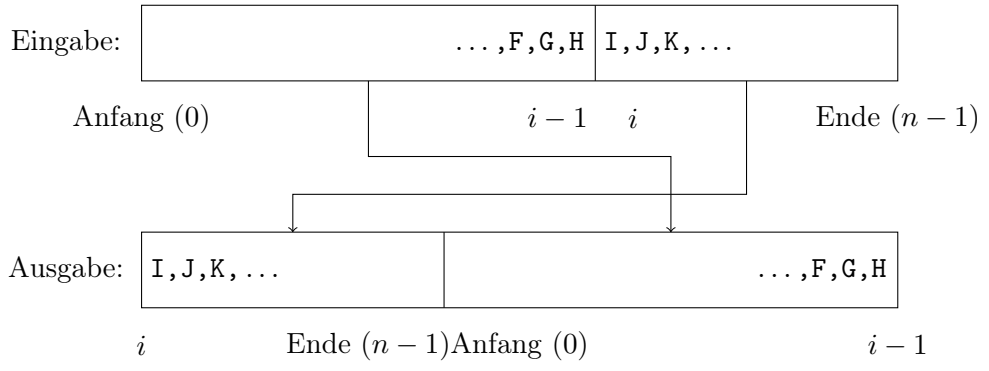

Abbildung 3: Rotation einer Folge der Länge n mit Index i und $0 \leq i \leq n-1$.

Für die nächste Zeile brauchen wir noch eine weitere Definition.

Definition 3.1 (Lexikografische Ordnung)**.** *Es seien* $a = (a_1, a_2, \dots)$ *und* $b = (b_1, b_2, \dots)$ *Zeichenfolgen mit* $a_i, b_i \in \Sigma$*, wobei* Σ *ein Alphabet ist, für das eine Ordnungsrelation gilt. a ist lexikografisch kleiner als b, wenn für den kleinsten Index k an dem sich die Folgen unterscheiden gilt, dass* $a_k < b_k$ *ist. Gleiches gilt auch, wenn kein solches k zu finden ist und a kürzer als b ist.*

Die Sortierung aus der Zeile 2 muss die Bedingung erfüllen, dass die Sortierung nach lexikografischer Ordnung stattfindet. Es ist dabei wichtig, dass vorher festlegt wird, ob aufsteigend oder absteigend sortiert wird. Sonst ist eine Rücktransformation nicht möglich. Wir werden die Rotationen immer aufsteigend sortieren. Die absteigende Sortierung funktioniert genauso.
Der Schritt in Zeile 3 kann verbildlicht werden, wenn die sortierten Rotationen untereinander geschrieben werden. Dann wird nur noch die letzte Spalte betrachtet und das ergibt sich die Ausgabefolge.
Die in Zeile 4 ausgegeben Daten reichen nun aus, um die die Rücktransformation durchzuführen.

3.2. Die Rücktransformation

Es werden wieder die Schritte als Algorithmus kurz vorgestellt. Danach werden einige wichtige Schritte kurz erläutert. Der Algorithmus 4 behandelt diese Transformation.

Algorithmus 4: Burrows-Wheeler-Transformation Rücktransformation.

Eingabe : Folge von Zeichen `input` und Index für die originale Zeile `index`
Ausgabe: Originale Folge von Zeichen

1 Erzeuge aus `input` eine sortierte Folge von Zeichen `sort`;
2 aktuellen Index `i` <= `index`;
3 **wiederhole**
4 Füge das Zeichen aus `sort` am Index `i` zur Ausgabefolge hinzu;
5 `n` <= # gleicher Zeichen, wie das Zeichen in `sort` am Index `i`, vor oder gleich dem Index;
6 `i` <= Index des `n`-ten Auftreten des Zeichens in `input`, welches dem Index `i` in `sort` entspricht;
7 **bis** $i \neq index$;
8 Gebe die Folge für die Ausgabe aus;

In Zeile 1 muss die Folge wieder so sortiert sein, wie die Folge der ersten Zeichen der Rotationen in der Hintransformation. Durch die Verwendung von lexikografische Ordnung ist das kein Problem. Wir sortieren die Folge aufsteigend, weil wir in der Hintransformation es so festgelegt haben.
Ein Durchlauf der Schleife lässt sich gut an einem Beispiel verdeutlichen. In Abbildung 4 ist das ganze skizziert. Die Eingabe für die Rücktransformation ist `BABABA`.
In Abbildung 4 tritt das `A` bis einschließlich Index 1 (aktueller Index) genau 2 mal in der sortierten Folge auf. Das zweite Auftreten von `A` in der Eingabefolge ist bei Index 3. Der neue aktuelle Index ist somit 3.
Es lässt sich leicht feststellen, dass die Schleife auch so oft wiederholt werden kann, wie die Folge lang ist.

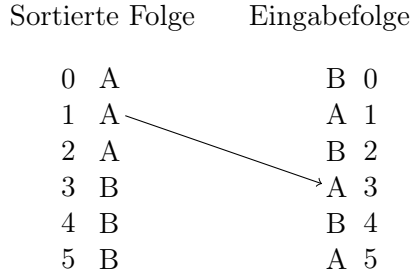

Abbildung 4: Skizze eines Durchlaufs der Schleife in der Rücktransformation mit aktuellen Index 1 und neuen Index 3.

3.3. Erklärung an einem Beispiel

Für dieses Beispiel nutzen wir verschiedene Zeichenfolgen als Eingabe. Das Hauptbeispiel ist die Zeichenfolge `HAUSMAUSLAUS`.

3.3.1. Hintransformation

Hauptbeispiel Zuerst erstellen wir alle möglichen Rotationen von dieser Zeichenfolge. Diese Rotationen werden in Abbildung 5 dargestellt. Diese können leicht erzeugt werden. Dazu wird das erste Zeichen der vorherigen Rotation vom Anfang an das Ende verschoben. Alle anderen Zeichen werden nach links geschoben. Damit ergibt sich die nächste Rotation. Gestartet wird mit der originalen Folge von Zeichen. Dieses Vorgehen muss so oft durchgeführt werden, wie lang die Folge ist.

0	HAUSMAUSLAUS
1	AUSMAUSLAUSH
2	USMAUSLAUSHA
3	SMAUSLAUSHAU
4	MAUSLAUSHAUS
5	AUSLAUSHAUSM
6	USLAUSHAUSMA
7	SLAUSHAUSMAU
8	LAUSHAUSMAUS
9	AUSHAUSMAUSL
10	USHAUSMAUSLA
11	SHAUSMAUSLAU

Abbildung 5: Alle möglichen Rotationen der Zeichenfolge `HAUSMAUSLAUS`.

Diese Rotationen werden jetzt sortiert. Diese lexikografisch sortierten Rotationen werden in Abbildung 6 dargestellt.
Die letzte Spalte von Abbildung 6 ergibt nun die transformierte Folge. Dabei werden die Zeichen von oben nach unten betrachtet. In diesem Fall lautet diese Folge `LMHSSSUUUAAA`. Weiterhin wird die Zahl 3 mit ausgegeben, da 3 der Index der Zeile ist, welche die originale Folge enthält.
Es lässt sich erkennen, dass nun alle gleichen Zeichen direkt hintereinander stehen. Diese Eigenschaft wird später ausgenutzt. Dazu wird die Folge später noch effizient kodiert,

	0	AUSHAUSMAUSL
	1	AUSLAUSHAUSM
	2	AUSMAUSLAUSH
⟶	3	HAUSMAUSLAUS
	4	LAUSHAUSMAUS
	5	MAUSLAUSHAUS
	6	SHAUSMAUSLAU
	7	SLAUSHAUSMAU
	8	SMAUSLAUSHAU
	9	USHAUSMAUSLA
	10	USLAUSHAUSMA
	11	USMAUSLAUSHA

Abbildung 6: Lexikografische Sortierung aller möglichen Rotationen der Zeichenfolge `HAUSMAUSLAUS`.

um weniger Speicher zu brauchen. Im Allgemeinen ist nach der Transformation oft der Fall, dass gleiche Zeichen beieinander stehen. Das Auftreten dieser Eigenschaft wird im Kapitel 3.5 näher erklärt.

Weiteres Beispiel Es müssen aber nicht immer alle gleiche Buchstaben beieinander stehen. Auch an verschiedenen Stellen in der Folge kann das gleiche Zeichen hintereinander auftreten. Dazu betrachten wir das Beispiel `HAASMAASLAAS`, wo das `U` im vorherigen Beispiel durch `A` ersetzt wurde. In Abbildung 7 sind alle Rotationen bereits sortiert dargestellt.

	0	AASHAASMAASL
	1	AASLAASHAASM
	2	AASMAASLAASH
	3	ASHAASMAASLA
	4	ASLAASHAASMA
	5	ASMAASLAASHA
⟶	6	HAASMAASLAAS
	7	LAASHAASMAAS
	8	MAASLAASHAAS
	9	SHAASMAASLAA
	10	SLAASHAASMAA
	11	SMAASLAASHAA

Abbildung 7: Lexikografische Sortierung aller möglichen Rotationen der Zeichenfolge `HAASMAASLAAS`.

Die transformierte Folge lautet somit `LMHAAASSSAAA` mit dem Index der originalen Folge von 6. Hier tritt das Zeichen `A` nicht komplett zusammenhängend auf, sondern in zwei Blöcken zu je drei Zeichen.

Gegenteiliges Beispiel Es gibt auch Beispiele, welche keine zusammenhängenden gleiche Zeichen bilden. Wir betrachten dazu die Zeichenfolge `AABADC`. In Abbildung 8 sind alle Rotationen sortiert dargestellt.

⟶	0	AABADC
	1	ABADCA
	2	ADCAAB
	3	BADCAA
	4	CAABAD
	5	DCAABA

Abbildung 8: Lexikografische Sortierung aller möglichen Rotationen der Zeichenfolge `AABADC`.

Die transformierte Folge lautet `CABADA` und der Index der originalen Folge ist 0. Bei diesem Beispiel sind in der Eingabe einige gleiche Zeichen beieinander, jedoch hat die Transformierte keine gleichen Zeichen nebeneinander. Dieses Beispiel zeigt somit ein gegenteiliges Ergebnis im Vergleich zu den vorherigen Beispielen.
Bevor wir die transformierte Folge weiter kodieren, werden wir die inverse Transformation an diesen Beispielen verdeutlichen.

3.3.2. Rücktransformation

Hauptbeispiel Zur Rücktransformation nutzen wir die Ausgabefolge aus der Hintransformation. Weiter nutzen wir die sortierte Folge von dieser Folge. In Abbildung 9 sind diese beiden Folgen nebeneinander dargestellt. Rechts ist die transformierte Folge (Transformierte) und links die sortierte Folge zu finden. Die Pfeile von links nach rechts zeigen die Zuordnungen an, welche für die Rücktransformation interessant ist. Es sind immer die gleichen Auftreten eines Zeichens in beiden Folgen verbunden.
Es zu beachten, dass die sortierte Folge gleich der ersten Spalte der sortierten Rotationen ist. Die Transformierte entspricht außerdem der letzten Spalte der Rotationen.

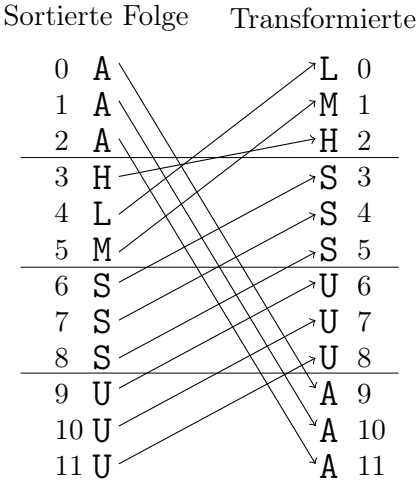

Abbildung 9: Sortierte Folge und die Ausgabefolge mit zugeordneten gleichen Auftreten des Zeichens. Das ist gleichbedeutend mit dem selben Zeichen.

Aus der Hintransformation ist bekannt, dass die originale Folge in Zeile 3 steht. Somit kann, beginnend mit Zeile 3, die originale Folge rekonstruiert werden. Das `H` ist somit das erste Zeichen der rekonstruierten Folge. Es ist das erste Auftreten des Zeichens in

der sortierten Folge, bis einschließlich Zeile 3. Das erste Auftreten des Zeichens `H` in der Transformierten ist in Zeile 2. Jetzt wird wieder das Zeichen aus der sortierten Folge in der Zeile 2 zu der rekonstruierte Folge hinzugefügt. In Abbildung 9 kann einfach den Pfeilen gefolgt werden, um den neuen aktuellen Index herauszufinden.
Die Abfolge der Schritte ist immer gleich. Zuerst wird das Zeichen am aktuellen Index zur Ausgabefolge hinzugefügt. Danach wird geschaut, wie oft das aktuelle Zeichen in der sortierten Folge vorher aufgetreten ist. Es wird dabei auch das aktuelle Zeichen mitgezählt. Danach wird geschaut, bei welchem Index dieses Zeichen genau gleich oft in der transformierten Folge aufgetreten ist. Dieser neue Index wird jetzt zum aktuellen Index. Diese Schritte beginnen jetzt wieder von vorne. Das wird solange durchgeführt bis die originale Folge wieder komplett rekonstruiert ist.
Die rekonstruierte Folge ergibt sich somit als die Folge `HAUSMAUSLAUS`. Das entspricht der originalen Folge.

Gegenteiliges Beispiel Die transformierte Folge lautete bei diesem Beispiel `CABADA`. Um die Zugehörigkeit der Zeichen darzustellen, ist in Abbildung 10 die sortierte und die transformierte Folge dargestellt. Es sind wieder die selben Zeichen verbunden.

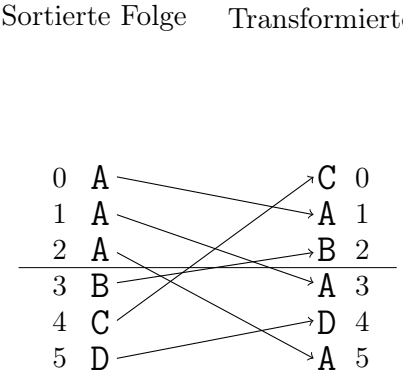

Abbildung 10: Sortierte Folge und die Ausgabefolge mit zugeordneten gleichen Auftreten des Zeichens (dasselbe Zeichen).

Nach der Durchführung der Schritte aus dem obigen Beispiel 3.3.2 ergibt sich wieder die originale Folge `AABADC`.

3.4. Korrektheit der Funktionsweise

Bei dem Nachweis der korrekten Funktionsweise sei ein besonderes Augenmerk auf die Rücktransformation gelegt. Damit wird konstruktiv gezeigt, dass die Transformation invertierbar ist. Das also die transformierte Folge und der Index ausreichen, um die originale Folge zu rekonstruieren.
Die Folge nach der Sortierung der transformierten Folge entspricht der ersten Spalte aus der Sortierung der Rotationen. In der transformierten Folge ist für jedes Zeichen der Nachfolger bekannt, da durch die Rotation die Reihenfolge bestehen bleibt. Damit steht dieser Nachfolger am gleichen Index in der sortierten Folge. Wenn immer unterschiedliche Zeichen auftauchen, ist es leicht die Folge wieder zu rekonstruieren. Dann lässt sich einfach durch die Eindeutigkeit der Zeichen und den bekannten Nachfolger die originale

Folge bestimmen. Mit dem Index lässt sich dann noch der Start der originale Folge bestimmen.
Um dieses triviale Vorgehen zu erläutern, werden wir die Beispielfolge `CHEMNITZ` betrachten. In Abbildung 11 sind zum besseren Verständnis die sortierten Rotationen dargestellt.

```
⟶ 0 CHEMNITZ
  1 EMNITZCH
  2 HEMNITZC
  3 ITZCHEMN
  4 MNITZCHE
  5 NITZCHEM
  6 TZCHEMNI
  7 ZCHEMNIT
```

Abbildung 11: Lexikografische Sortierung aller möglichen Rotationen der Zeichenfolge `CHEMNITZ`.

In Abbildung 11 ist die letzte Spalte die transformierte Folge `ZHCNEMIT`. Die erste Spalte `CEHIMNTZ` entspricht der der sortierten transformierten Folge. Aus der Abbildung kann auch die Eigenschaft des Nachfolgers jedes Zeichens erkannt werden. Der Nachfolger des Zeichen an i-ter Position in der Transformierten ist das Zeichen an i-ter Position in der sortierten Folge. Das ist in Tabelle 4 dargestellt.

aktuelles Zeichen	Z	H	C	N	E	M	I	T
nachfolgendes Zeichen	C	E	H	I	M	N	T	Z

Tabelle 4: Zuordnung des Nachfolgers für jedes Zeichen an dem Beispiel `CHEMNITZ`.

Beginnend mit dem Zeichen aus der sortierten Folge in Index 0 wird die originale Folge rekonstruiert. Die 0 erhalten wir als Ausgabe der Transformation. Das erste Zeichen ist somit ein `C`. Der Nachfolger von `C` ist laut Tabelle `H`. Der Nachfolger von `H` ist `E`, und so weiter. Somit kann ohne Probleme die originale Folge rekonstruiert werden.
Schwieriger wird es, wenn mehrere gleiche Zeichen auftreten. Damit ist eine Unterscheidung nach dem selben und nicht nur den gleichen Zeichen wichtig. Mit Hilfe der Eigenschaft der lexikografischen Ordnung ist das möglich. So lässt sich eine eindeutige Zuordnung zwischen den selben Zeichen finden.
Dazu betrachten wir die sortierten Rotationen. Alle Rotationen, welche mit den gleichen Buchstaben beginnen, stehen direkt hintereinander. Die genaue Reihenfolge dieser Rotationen wird komplett von den Buchstaben bestimmt, welche nach dem ersten Zeichen stehen. Die Rotationen, welche dieses erste Zeichen als letztes Zeichen haben, sind in genau gleicher Reihenfolge sortiert. Diese müssen aber nicht mehr direkt hintereinander stehen.
Der Grund dafür ist, dass bei gleichen ersten Zeichen die darauffolgenden Zeichen erst relevant sind. Wird dieses erste Zeichen nun entfernt, so spielt das keine Rolle für die Reihenfolge dieser Zeichenfolgen. Ohne die Reihenfolge zu ändern kann an letzter Stelle noch dieses Zeichen eingefügt werden, da die letzte Stelle kaum Relevanz hat. In Abbildung 12 wird das für einige Rotationen mit gleichen Anfangszeichen verdeutlicht. Dabei wird wieder auf das vorherige Beispiel Bezug genommen. Die Rotationen sind immer

sortiert. Zur besseren Identifizierung der selben Zeichen, sind diese nach dem Auftreten in der originalen Folge indiziert.

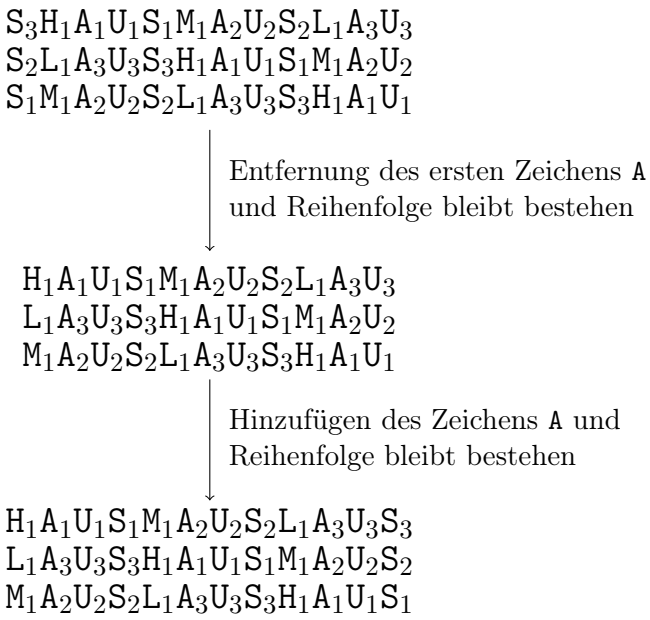

Abbildung 12: Verdeutlichung der Wichtigkeit der lexikografischen Ordnung für die Invertierbarkeit an einigen Rotationen für das Beispiel `HAUSMAUSLAUS`.

In der Abbildung 12 ergeben sich so neue Rotationen. Diese neuen Rotationen stehen natürlich auch wieder in der Liste aller Rotationen. Also ergeben sich so auch wieder neue Zuordnungen zwischen aktuellen Zeichen und Nachfolgern. Besonders ist aber der Zusammenhang des selben Zeichens in einer Zeichenfolge zu erkennen. Das erste Zeichen wurde ja nur an die letzte Stelle rotiert und der Rest bleibt gleich. Damit bleibt auch die Sortierung gleich. Das selbe Zeichen kann somit durch das gleich häufige Auftreten identifiziert werden, da diese Feststellung immer zutrifft.
Damit kann die originale Reihenfolge der Zeichen bestimmt werden. Mit dem originalen Index kann die originale Folge rekonstruiert werden.

3.5. Warum findet Kompression statt?

Ziel der BWT ist, dass gleiche Zeichen möglichst beieinander stehen. Diese Art von Zeichenfolge lässt sich dann effizient und performant komprimieren. Wir werden in Kapitel 3.6 näher betrachten, wie diese Eigenschaft für die Kompression benutzt wird. Aber wie lässt sich diese Eigenschaft eigentlich begründen.
In [BW94] haben Burrows und Wheeler als Motivation für dieses Verfahren normalen Text angegeben. Spezielle Strukturen in Texten werden für die Transformation ausgenutzt. Die Eigenschaft lässt sich am besten verdeutlichen, wenn Großbuchstaben betrachtet werden. Auf Grund der Wertigkeit der Zeichen werden alle Rotationen, welche mit einem Großbuchstaben beginnen, direkt hintereinander sortiert. Das Zeichen vor fast jedem Großbuchstaben ist ein Leerzeichen, weil es entweder ein Nomen oder ein Satzanfang ist. Das letzte Zeichen dieser Rotationen ist somit ein Leerzeichen. Viele Leerzeichen stehen damit direkt hintereinander.

Burrows und Wheeler argumentieren dann weiter, dass überall in Texten ähnlichen Wörter auftreten. Sie betrachten dabei englischen Text. In den sortierten Rotationen werden die Rotationen, welche mit dem gleichen Wörtern beginnen, hintereinander sortiert. Auch werden die Rotationen, welche mit dem zweiten Zeichen des Wortes beginnen, nah beieinander sortiert, und so weiter. Mit gewisser Wahrscheinlichkeit stehen diese direkt hintereinander.
Sie versuchen das ganze an einem Beispiel zu zeigen. So tritt vor `he␣` mit hoher Wahrscheinlichkeit ein `t` auf. Das stellt das Wort `the␣` dar. Bei den Rotationen mit `he␣` am Anfang ist das letzte Zeichen mit hoher Wahrscheinlichkeit `t`. Da dieses Wort sehr oft in englischen Texten auftritt, stehen in der Transformierten viele `t` direkt hintereinander. Allgemein tauchen solche Strukturen öfters auf. Das lässt sich leicht erkennen, weil viele Wörter im Text gleich oder ähnlich sind.
Wie lässt sich diese Eigenschaft nun verallgemeinern, damit dieses Argument nicht nur für Text gültig ist?
Das vorher beschriebene Argument nutzt Muster in Form von Worten aus. Auch in allgemeinen Zeichenfolgen treten Muster auf. Wir wollen uns nun auf Findung von Mustern konzentrieren. Diesen Prozess wollen wir im Folgenden verallgemeinern und approximieren. Wir werden am Ende sehen, dass die BWT eine Musterfindung approximiert.
Wir betrachten dazu eine feste Musterfolge und eine Zeichenfolge. Das Ziel ist, das Muster in der Zeichenfolge zu finden. Das Verfahren zum finden des Musters betrachtet die Zeichenfolge von vorne nach hinten. Bei jedem Zeichen wird überprüft, ob das Zeichen gleich dem ersten Zeichen des Musters ist. Sollte dies der Fall sein, dann werden auch die nachfolgenden Zeichen überprüft, ob sie gleich den anderen Zeichen im Muster sind. Wenn dies nicht der Fall ist, dann wird das nächste Zeichen in der Zeichenfolge betrachtet. Wenn alle Zeichen bis zur Länge des Musters gleich sind, dann wurde das Muster gefunden.
Für unser Beispiel mit der Folge `HAUSMAUSLAUS` und dem Muster `AUS` ist das Verfahren in Abbildung 13 dargestellt. Beim ersten Schritt wird bereits beim ersten Zeichen festgestellt, dass sich die Folgen unterscheiden. Beim zweiten Schritt stimmen die ersten drei Zeichen des Musters überein. Damit tritt das Muster an dieser Position auf.

```
Folge:  H A U S M A U S L A U S    ⇒ Muster stimmt ab ersten
Muster: A U S                        Zeichen nicht überein

Folge:  H A U S M A U S L A U S    ⇒ Muster stimmt überein
Muster:   A U S
```

Abbildung 13: Finden des Musters `AUS` in der Folge `HAUSMAUSLAUS`.

Ein äquivalentes Verfahren betrachtet alle möglichen Rotationen. Bei jeder Rotation werden die ersten Zeichen aus der Zeichenfolge und dem Muster betrachtet. Bei Gleichheit werden die nachfolgenden Zeichen verglichen wie vorher. In Abbildung 14 ist das vorherige Beispiel mit allen Rotationen dargestellt. Das Muster wird beginnend mit der ersten Stelle überprüft.
Dieses Verfahren kann jedoch beschleunigt werden, wenn alle Rotationen sortiert sind. Dazu muss ein lexikografische Ordnung erfüllt sein. Durch binäre Suche wird im worstcase nicht jede Rotation betrachtet, sondern nur logarithmisch viele Rotationen. Ein weiterer Vorteil ist, dass alle Auftreten dieses Muster beieinander stehen. Diese sortierten

Muster: AUS

Folge:

```
AUSHAUSMAUSL ←
AUSLAUSHAUSM ←
AUSMAUSLAUSH ←
HAUSMAUSLAUS
LAUSHAUSMAUS
MAUSLAUSHAUS
SHAUSMAUSLAU
SLAUSHAUSMAU
SMAUSLAUSHAU
USHAUSMAUSLA
USLAUSHAUSMA
USMAUSLAUSHA
```

Abbildung 14: Lexikografische Sortierung aller möglichen Rotationen der Zeichenfolge `HAUSMAUSLAUS`. Die mit Pfeilen markierten Zeilen geben an, dass eine Folge gefunden wurde.

Rotationen werden auch als *Suffix Array* bezeichnet und auch praktisch verwendet. In [LMW05] wird dieses Verfahren für Musterfindung in Genomen genutzt. Auch wird darin eine effiziente Methode zur Speicherminimierung bei der Erzeugung der *Suffix Arrays* besprochen.
Wir nutzten diese Erkenntnisse weiter, um ein Kompressionsverfahren herzuleiten, welches auch effizient ist.
Dazu betrachten wir jetzt nicht mehr ein bestimmtes Muster, sondern jedes mögliche Muster. Mit Hilfe der sortierten Rotationen lassen sich einfach alle mehrfachen Auftreten eines Musters finden. Dafür wird die aktuelle Rotation mit der davor und danach verglichen. Ein mehrfach auftretendes Muster ist dann die Folge der ersten paar Zeichen, welche gleich sind. Es wird dann die längere Folge von beiden gewählt. Damit wird das längste mögliche Muster dieser zwei Startpunkte in der Zeichenfolge gefunden. Für das Beispiel `HAUSMAUSLAUS` sind diese Muster in Abbildung 15 hervorgehoben.

```
AUSHAUSMAUSL
AUSLAUSHAUSM
AUSMAUSLAUSH
HAUSMAUSLAUS
LAUSHAUSMAUS
MAUSLAUSHAUS
SHAUSMAUSLAU
SLAUSHAUSMAU
SMAUSLAUSHAU
USHAUSMAUSLA
USLAUSHAUSMA
USMAUSLAUSHA
```

Abbildung 15: Lexikografische Sortierung aller möglichen Rotationen der Zeichenfolge `HAUSMAUSLAUS`. Der schwarze Text kennzeichnet das längste mögliche mehrfach auftretende Muster, welches an der Position beginnt.

Es kann aber möglicherweise nicht die beste Strategie sein, die längsten Muster zu kodieren. Das kann am Beispiel `ABABABABABAB` gesehen werden. So kann es sehr aufwendig sein eine optimale Strategie zu finden. Das Kodieren würde in diesem Fall durch eine Referenz auf das erste Auftreten und der Länge des Musters erfolgen. Das erste Auftreten muss dann aber normal kodiert sein.
Trotzdem können diese Erkenntnis über das Finden von mehrfachen Mustern als Approximation nutzen. So werden bei einem Mustern, welches einzigartig und lang genug ist, nicht nur die Anfänge der maximal langen Muster zusammen sortiert. Es werden auch die Rotationen, welche an der zweiten Stelle dieses Musters beginnen, beieinander sortiert. Das findet solange statt, bis die Einzigartigkeit des Musters nicht mehr erkennbar ist.
Das kann auch an dem einfachen Beispiel in Abbildung 15 gesehen werden. In Realität sieht das zwar deutlich komplizierter aus, aber das Beispiel verdeutlicht das Konzept, dass gleiche Zeichen an letzter Stelle stehen.
Wenn das ersten Zeichen eines maximalen Musters am Anfang der Rotation steht, dann ist das Zeichen davor im Allgemeinen beliebig. Von dieser Rotation aus wird jetzt rotiert. Dabei werden einige Zeichen vom Anfang an das Ende verschoben werden. Es besteht eine hohe Wahrscheinlichkeit, dass diese neuen Rotationen vom gleichen Muster direkt hintereinander stehen. Das ist abhängig von der Länge und Einzigartigkeit des Musters. Daraus folgt, dass nun gleiche Zeichen am Ende der Rotation stehen. Damit ergibt sich die gewollte Eigenschaft.
Wir werden jetzt noch kurz klären, wie die Folge von gleichen Zeichen hintereinander unerwartet unterbrochen werden kann. Ein unerwartetes Unterbrechen sei ein Ende der Folge gleicher Zeichen an letzter Position der Rotationen, obwohl es mehr gleiche Muster gibt. Das passiert, wenn ein ähnliches, aber später beginnendes Muster zwischen den anderen Mustern sortiert wird.
Ein Beispiel dafür ist die Folge `ABCDABCADBCB`. Die sortierten Rotationen dieses Beispiels sind in 16 dargestellt. Ein Muster, welches zweimal auftritt, ist `ABC`. Die Rotationen, welche mit dem Muster anfangen, stehen direkt hintereinander. Die Rotationen, welche mit dem zweiten Zeichen des Musters beginnen, stehen nicht hintereinander. Das passiert, weil eine andere Rotation auch mit `BC` beginnt und dazwischen sortiert wird.

```
ABCADBCBABCD
ABCDABCADBCB
ADBCBABCDABC
BABCDABCADBC
BCADBCBABCDA
BCBABCDABCAD
BCDABCADBCBA
CADBCBABCDAB
CBABCDABCADB
CDABCADBCBAB
DABCADBCBABC
DBCBABCDABCA
```

Abbildung 16: Lexikografische Sortierung aller möglichen Rotationen der Zeichenfolge `ABCDABCADBCB`.

So lässt sich erkennen, dass gleiche Zeichen nah beieinander stehen, aber nicht direkt

hintereinander stehen müssen.

3.6. Nachfolgende Transformation der Daten

Die Zeichen werden durch die BWT nur umsortiert und nicht komprimiert. Aus diesem Grund müssen weitere Verfahren angewendet werden um Kompression zu erreichen. Die Eigenschaft, dass gleiche Zeichen nah beieinander stehen, wollen wir dazu ausnutzen. Dazu werden wir verschiedene Verfahren behandeln.

3.6.1. Vorgehensweise nach Burrows und Wheeler

Das erste Verfahren wurde auch schon in [BW94] von Burrows und Wheeler beschrieben. Es beschreibt die einfachste Vorgehensweise.
Als erster Schritt wird jetzt ein Algorithmus genutzt, welcher die lokalen Auftreten von gleichen Zeichen anders kodiert. Es werden diese lokal gleichen Zeichen, welche hintereinander auftreten, zu einem Zeichen transformiert, welches dann global gleich ist. Lange Folgen des gleichen Zeichens werden so erzeugt. Durch dieses Vorgehen wird der Informationsgehalt der Daten herabgesenkt. Die hier verwendete Kodierung ist die Move-to-front-Kodierung (MTF). Die Länge der Folge bleibt gleich.
Der letzte Schritt dieser Transformation ist ein Entropie-Kodierer. Dieser Schritt wird bei jedem Verfahren als letzter durchgeführt. Das Ziel von diesem Kodierer besteht darin nur so viel Speicher zu verbrauchen, wie der Informationsgehalt der Daten benötigt. Auch lassen sich Kodierer, welche das können, leicht und effizient implementieren, da sie keinen Kontext betrachten. Das steht im Gegensatz zu Kodierern mit mehr Kontextbetrachtung. Diese liefern aber nur ein minimal besseres Ergebnis, welches den höheren Rechenaufwand nicht wert ist. Als Beispiel wird hier die Huffman-Kodierung oder ein arithmetischer Kodierer genutzt.
Die so erhaltenen transformierten Daten sind die Ausgabe des von Burrows und Wheeler beschriebenen Prozesses.

3.6.2. Allgemeine heutige Vorgehensweisen

Später wurden einige Modifikationen zur vorherigen Transformation gefunden, welche die Effizienz steigern. Diese werden in [Abe03] zusammengefasst. Die wichtigsten Modifikationen, welche auch oft Verwendung finden, werden wir kurz behandeln.
Zum einen kann als Alternative auch direkt die transformierte Folge komprimiert werden. Dazu kann zum Beispiel eine Lauflängenkodierung verwendet werden. Dieses Verfahren ist oft nur effizient, wenn entweder wenige verschiedene Zeichen vorhanden sind oder die Folgen gleicher hintereinander auftretender Zeichen sehr lang sind. Bei Ersterem können effizientere Verfahren als die allgemeine Lauflängenkodierung benutzt werden. Bei Letzterem kann viel Information über lange Folgen durch eine sehr kurze Zeichenfolge ausgedrückt werden.
Eine andere Modifikation setzt nach der MTF-Kodierung an. Dabei wird MTF Lauflängenkodierung auf das Zeichen `0` angewendet. Alle anderen Zeichen bleiben dabei gleich. Dieser Schritt ist in aktuellen Vorgehensweisen fast immer enthalten. Die Lauflänge wird dabei oft binär kodiert.

3.6.3. Vorgehensweise bei bzip2

Das Programm *bzip2* ist die am meisten verwendete Implementierung, welche die BWT benutzt. Im Folgenden sei kurz auf die Reihenfolge der Algorithmen eingegangen, welche vor und nach der BWT durchgeführt werden.
Die folgenden Schritte werden bei der Kompression ausgeführt. Zur Dekompression werden die Inversen der Schritte in umgekehrte Reihenfolge ausgeführt. Vor der Transformation wird die Eingabe in Blöcke gleicher Größe unterteilt. Typische Größen sind zwischen 100 und 900 Kilobyte. Für jeden Block werden die gleichen Schritte durchgeführt. Der Grund für diese Aufteilung ist, dass die BWT eine Laufzeit von $\mathcal{O}(n \log n)$ hat, wobei n die Länge der Zeichenfolge ist. So kann trotzdem eine noch vergleichsweise gute Performanz erreicht werden ohne zu viel Effizienz in der Kodierung zu verlieren.
Als erster Schritt wird mit einer Lauflängenkodierung ab Lauflänge 4 die gesamte Zeichenfolge kodiert. Diese wurde in Kapitel 2.3.3 vorgestellt. Interessant daran ist, dass dieser Schritt die Effizienz der Kompression nicht verbessert. Dieser Schritt wurde ursprünglich eingeführt, um den Sortieralgorithmus vor worst-case Eingaben zu schützen. Es hat sich aber gezeigt, dass die Sortierung damit gut umgehen kann. Der Hintergrund dazu wird in [Sew05] beschrieben.
Als nächstes werden die Zeichen im jeweilige Block mit der BWT umsortiert. Danach wird mit der normalen MTF-Kodierung der Block transformiert. Damit sollten in der Regel sehr viele Nullen in der Folge enthalten sein.
Die Folgen von direkt nacheinander auftretende Nullen werden mit MTF Lauflängenkodierung kodiert. Alle anderen Zeichen werden nicht betrachtet und bleiben gleich kodiert. Die Lauflänge der Nullen wird binäre kodiert, wodurch zwei neue Zeichen eingeführt werden. Das genaue Vorgehen wird in Kapitel 2.3.3 erklärt.
Um den benötigten Speicher nochmal nochmal deutlich zu reduzieren, wird jetzt noch ein Entropie-Kodierer verwendet. Bei *bzip2* wird dazu Huffman-Kodierung verwendet. Bei der ersten Version *bzip* wurde noch ein arithmetischer Kodierer benutzt, welcher aus patentrechtlichen Gründen nicht mehr verwendet werden durfte.
Dies ist nur ein kleiner Überblick über die Schritte, welche durchgeführt werden. Es gibt in der genauen Implementation noch einige Feinheiten, welche wir nicht betrachten. Auch werden noch kleine Schritte ausgeführt, welche die Kompression ein wenig optimieren. Hauptsächlich folgen nur noch Schritte, welche den benötigten Speicher für die Tabellen der Huffman-Kodierungen senken. Diese Tabellen werden gebraucht, um die Inverse der Huffman-Kodierung auszuführen.

3.7. Weiterführung des Beispiels

Das Hauptbeispiel aus Kapitel 3.3 wird jetzt weitergeführt, um die finale Kodierung zu illustrieren. Dazu werden wir zuerst die MTF-Kodierung auf die Ausgabefolge von der BWT anwenden. Anschließend wird das Zeichen `0` in der Folge mit MTF-Lauflängenkodierung transformiert. Dabei wird die Lauflänge binäre kodiert. Danach wird das Ergebnis noch mit einem Entropie-Kodierer kodiert. Wir werden dazu Huffman-Kodierung als Verfahren nutzen.
Die MTF-Kodierung wird in Tabelle 5 dargestellt. Die Eingabe ist die Folge der Zeichen `L,M,H,S,S,S,U,U,U,A,A,A` und das sortierte Alphabet ist `A,H,L,M,S,U`.
Die Ausgabe der Kodierung ist damit die Folge `2,3,3,4,0,0,5,0,0,5,0,0`.

Alphabet	Zeichen	gespeicherte Position	aktualisiertes Alphabet
A,H,L,M,S,U	L	2	L,A,H,M,S,U
L,A,H,M,S,U	M	3	M,L,A,H,S,U
M,L,A,H,S,U	H	3	H,M,L,A,S,U
H,M,L,A,S,U	S	4	S,H,M,L,A,U
S,H,M,L,A,U	S	0	S,H,M,L,A,U
S,H,M,L,A,U	S	0	S,H,M,L,A,U
S,H,M,L,A,U	U	5	U,S,H,M,L,A
U,S,H,M,L,A	U	0	U,S,H,M,L,A
U,S,H,M,L,A	U	0	U,S,H,M,L,A
U,S,H,M,L,A	A	5	A,U,S,H,M,L
A,U,S,H,M,L	A	0	A,U,S,H,M,L
A,U,S,H,M,L	A	0	A,U,S,H,M,L

Tabelle 5: Move-to-front-Kodierung für das HAUSMAUSLAUS Beispiel und damit für die Eingabe L,M,H,S,S,S,U,U,U,A,A,A.

Die Lauflänge des Zeichens 0 werden jetzt noch mit dem in Kapitel 2.3.3 beschriebenem Verfahren kodiert. Wir führen dazu die neuen Zeichen L1 und L2 ein. L1 hat die Wertigkeit 1 und L2 die Wertigkeit 2 in der binären Kodierung. Es gibt in der Folge drei Auftreten von hintereinander stehenden Nullen mit Lauflänge 2. Die Kodierung der Lauflänge 2 ist L2. Jetzt ersetzten wir diese Folgen von Nullen mit diesem Zeichen. Somit ergibt sich die Zeichenfolge 2,3,3,4,L2,5,L2,5,L2 nach dieser Lauflängenkodierung.
Für diese Folge wird jetzt die Huffman-Kodierung bestimmt. Dazu werden zuerst die relativen Häufigkeiten der Zeichen in der Folge ermittelt. Diese sind in Tabelle 6 angegeben.

Zeichen	relative Häufigkeit
L2	$\frac{1}{3}$
2	$\frac{1}{9}$
3	$\frac{2}{9}$
4	$\frac{1}{9}$
5	$\frac{2}{9}$

Tabelle 6: Häufigkeiten der Zeichen nach der MTF-Kodierung.

Mit diesen Häufigkeiten wird der Baum erzeugt, um die Kodierungen zu ermitteln. Der resultierende Baum ist in Abbildung 17 dargestellt. An den Kanten des Baumes sind die Kodierungen angegebenen, welche sich dann ergeben. Die Erstellung des Baums wird in Kapitel 2.3.1 beschrieben. Im Folgenden sei aber zur Vollständigkeit die Reihenfolge der Verschmelzungen angegeben.
Zuerst werden die Bäume von den Zeichen 2 und 4 zusammengefügt. Danach werden die Bäume mit den Zeichen 3 und 5 vereint. In Folge werden die entstandenen Bäume mit den Zeichen 2,4 und 3,5 zusammengefügt. Zuletzt werden noch die Bäume mit den Zeichen 2,3,4,5 und L2 verschmolzen.

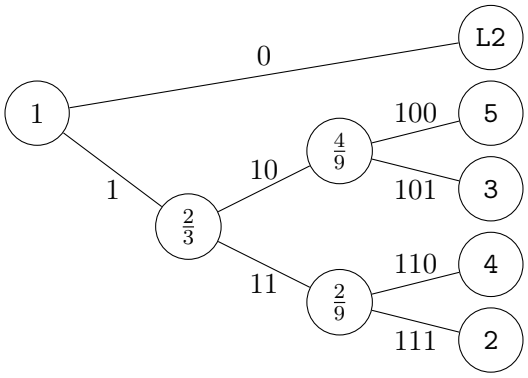

Abbildung 17: Erstellter Baum, um die Kodierungen für die Huffman-Kodierung zu ermitteln. An den Kanten sind die Kodierungen vermerkt. In den Nichtblattknoten stehen die addierten Häufigkeiten der folgenden Blätter.

Draus ergeben sich die Kodierungen für jedes Zeichen. Eine Zuordnung zwischen den Zeichen und der Kodierung ist in Tabelle 7 zu finden.

Zeichen	Kodierung
`L2`	0
`2`	111
`3`	101
`4`	110
`5`	100

Tabelle 7: Zuordnung zwischen den Zeichen und der Huffman-Kodierung.

Durch Ersetzen der Zeichen durch ihre Kodierung ergibt sich die Zeichenfolge `111101101-110010001000` als Ergebnis. Dazu muss noch noch gespeichert werden, wie die Zuordnung zwischen den Zeichen und ihrer Kodierung ist. Diese Größe wäre in Relation zur kodierten Folge recht groß, aber das liegt daran, dass die Eingabe vergleichsweise klein ist.
Die Zeichenfolge ist jetzt zwar länger, aber dafür sind die Zeichen jetzt nur noch in dualer Basis statt Basis 6. Die Zeichenkette hatte vorher Basis 6, da die sechs möglichen Zeichen `L1,L2,2,3,4,5` waren. Die benötigte Anzahl an Bits, welche für ein Zeichen der Basis 6 nötig werden, sind 3 Bits. Das ergibt sich, da $2^3 = 8 \geq 6$ ist. Somit wurden vor der Kodierung 36 Bits benötigt. Nach der Kodierung werden nur noch 21 Bits benötigt. Somit hat die komprimierte Zeichenfolge jetzt nur noch 7/12 der Größe der originalen Zeichenfolge.
Die anderen Beispiele werden wir hier nicht behandeln, da sie genauso ablaufen und die Funktionsweise nicht weiter verdeutlichen.

4. Einige Untersuchungen zur BWT

In diesem Kapitel werden wir die BWT im Zusammenhang mit anderen Algorithmen betrachten. Diese Algorithmen werden entweder vor oder nach der BWT auf die Zeichenfolge angewendet. Der Wert, welcher verglichen wird, ist die Größe der transformierten Folge. Die Größe der Daten, welche zur Rücktransformation gespeichert werden müssen,

werden wir im Allgemeinen nicht betrachten. Diese sind in Relation zu der transformierten Folge oft sehr klein.
Wir werden die BWT mit verschiedenen Vorgehensweisen zuerst experimentell untersuchen. Dazu werden Beispieldaten durch verschieden Kombinationen von Algorithmen vor und nach der BWT transformieren. Danach werden wir versuchen die Ergebnisse und Vermutungen aus diesen Untersuchungen theoretisch zu erklären.

4.1. Experimentelle Untersuchung

Zur experimentellen Untersuchung wurde die BWT und weitere Algorithmen implementiert. Diese weiteren Algorithmen werden entweder vor oder nach der BWT ausgeführt, um dann die eigentliche Kompression zu erreichen.
In Kapitel 4.1.1 werden wir einige Feinheiten und Tricks bei der Implementation behandeln. Danach wird die Methode vorgestellt, wie die experimentellen Untersuchungen durchgeführt wurden. In Kapitel 4.2 wird versucht diese experimentellen Ergebnisse auch theoretisch zu begründen.

4.1.1. Implementation

Die Implementation wurde in der Programmiersprache *Rust* durchgeführt. Es wurde die Version *1.25.0* des Standard Rust-Compilers verwendet. Zuerst wird auf die Implementation der reinen Borrows-Wheeler-Transformation eingegangen. Dabei werden wir verschiedene Probleme und Tricks behandeln. Danach werden noch Feinheiten von den anderen Transformationen betrachtet, welche in Kombination mit der BWT benutzt werden. Der gesamte Quellcode ist in Anhang A zu finden.

Erzeugung der Rotationen Das Problem ist, dass es sehr viele Rotationen der Zeichenfolge gibt und diese alle abgespeichert werden müssen. Um genau zu sein, gibt es so viele Rotationen, wie es Zeichen in der Folgen gibt. So wird für eine Folge mit n Zeichen Speicher für n^2 Zeichen benötigt. Das kann selbst bei vergleichsweise kleinen Zeichenfolgen zu viel Speicherverbrauch führen. Das ist aber nur der Fall, wenn alle Rotationen auch abspeichert werden.
Ersichtlich wird es an einem kurzen Beispiel. Wir betrachten die Länge der Folge mit 2^{20} Zeichen. Ein Zeichen hat die Größe von einem Byte. Die Zeichenkette hat damit eine Größe von einem Megabyte. Auch ergibt sich, dass insgesamt 2^{20} verschiedene Rotationen existieren. Damit ergibt sich als benötigter Speicher, dass diese 2^{20} Zeichen 2^{20}-mal gespeichert werden müssen. Das sind 2^{40} Zeichen oder ein Terabyte Speicherplatz. Das ist in etwa die Größe einer modernen Festplatte, wobei die Eingabe mit einem Megabyte relativ klein ist.
Die Lösung des Problems ist, dass nicht die Rotationen gespeichert werden, sondern die Startindizes der Rotationen. Dazu muss die originale Folge noch gespeichert werden. Die Indizes werden dann genutzt, um das Startzeichen der Rotation zu bestimmen. Die darauffolgenden Zeichen ergeben sich dann entsprechend.
In der Implementation ist das durch einer Folge realisiert, welche die originale Folge zweimal hintereinander stehen hat. Die Rotation, welche mit Index i beginnt, ist dann die Teilfolge dieser Folge von Index i bis Index $i + n - 1$. Dabei ist n die Länge der Folge. Zum Beispiel wird für die originale Folge `ABCD` die Folge `ABCDABCD` gespeichert. Die Rotation ab Index 2 ist die Teilfolge von Index 2 bis 5, welche `CDAB` ist.

Sortierung der Rotationen Die Rotationen müssen lexikografisch sortiert werden. Es sind so viele Rotationen zu sortieren, wie es Anzahl an Zeichen in der Folge gibt. Das können schnell mehrere Millionen oder mehr sein. Da die anderen Schritte der BWT lineare Laufzeit haben, ist ein schnelles Sortierverfahren wichtig. Eine Laufzeit von $\mathcal{O}(n \log n)$ ist damit wichtig, da ein allgemeines Sortierverfahren nicht besser werden kann. Das genaue Sortierverfahren, welches wir verwenden ist Quicksort mit randomisierten Pivot-Element. Die Beachtung der lexikografischen Sortierung wird durch die vorhandene Vergleichsoperation für Folgen übernommen. Diese ist in *Rust* standardmäßig vorhanden.
Bei der Sortierung werden nicht die Rotationen sortiert, sondern die Verweise durch die Indizes. Vor der Vergleichsoperation wird aus dem Index die entsprechende Rotation generiert. Die Information der Sortierung ist damit in den Indizes gespeichert.

Erzeugung der transformierten Folge Die Information der Sortierung ist durch die Indizes gespeichert, wie im vorherigen Abschnitt erwähnt. Um nun die Ausgabe der BWT zu erzeugen, muss die letzte Spalte der Rotationen und der Index der originalen Folge ermittelt werden.
Die letzte Spalte wird generiert, indem die Folge der Indizes in Reihenfolge betrachtet wird. Der aktuelle Wert in der Folge sei i. Dann wird das Zeichen in der doppelten Zeichenfolge, welches am Index $i + n - 1$ steht, der Ausgabe hinzugefügt.
Der Index der originalen Folge wird durch den Index aus der Folge der Indizes bestimmt, welcher den Wert `0` hat.

Weitere Kodierung Um verschiedene Kombinationen von Algorithmen zu testen, wird nicht nur die BWT benötigt. Es sind deswegen auch die vorherigen und nachfolgenden Transformationen implementiert. Zum Vergleich der Effizienz wurde eine Standardimplementierung umgesetzt, welche *bzip2* sehr ähnelt. Implementierte Algorithmen sind die Lauflängenkodierung in vielen verschiedenen Varianten, die MTF-Kodierung, die MTF-1-Kodierung und ein arithmetischer Kodierer.
Die Funktionsweise der Algorithmen ist in Kapitel 2.3 beschrieben. Der Quellcode von diesen Algorithmen ist in Anhang A.2 zu finden. Damit sind allen Algorithmen, außer der arithmetische Kodierer, ausreichend besprochen.
Der genaue arithmetische Kodierer, welcher verwendet wird, nutzt anfangs ein Intervall von 0 bis 1. Aus den Häufigkeiten der Zeichen wird eine Zuordnung zwischen Zeichen und entsprechenden Intervallen hergestellt. Das betrachtete Intervall wird mit jedem gelesenen Zeichen entsprechend der Zuordnung aktualisiert. Nach der Aktualisierung wird überprüft, ob die obere Intervallgrenze unter oder die untere Intervallgrenze über dem Wert $0,5$ liegt. Sollte dies der Fall sein, dann kann entsprechend eine `0` oder `1` in die Ausgabe geschrieben werden. Danach wird das Intervall entsprechend aktualisiert.
Dieser arithmetische Kodierer schreibt sofort ein Zeichen in die Ausgabe, wenn jeder Wert im Intervall mit diesem Zeichen beginnen muss. Die Funktion des arithmetischen Kodierers gibt auch die Anzahl der einzigartigen Zeichen aus. Damit wird eine Abschätzung des Speicher-Overheads durchgeführt.
Zu allen diesen Algorithmen reicht die Hintransformation und es wird keine Rücktransformation benötigt. Wir sind schließlich nur an der Qualität der Kompression interessiert und nicht an der praktischen Anwendung dieser.

4.1.2. Die Testdaten

Es werden verschiedene Testdaten verwendet um die verschieden Verfahren zu untersuchen. Zu diesen Testdaten gehören die einzelnen Dateien des *Calgary Corpus*. Auch ist der komplette *Calgary Corpus* als eine `tar` Datei in den Testdateien enthalten. Die einzelnen Dateien aus dem Corpus werden mit dem Dateinamen bezeichnet. Die komplette Datei wird als `tar` bezeichnet. Als `comb` wird die kumulative Größe der einzelnen Dateien des *Calgary Corpus* bezeichnet.
Weiterhin wurden noch die einzelnen Dateien des *Large Canterbury Corpus* betrachtet. Diese Dateien werden auch wieder durch die Dateibezeichnung identifiziert. Der *Large Canterbury Corpus* enthält zum einen zwei größere Textdateien und zum anderen eine Genom-Datei. Das besondere an der Genom-Datei ist, dass das Alphabet nur aus vier verschiedenen Zeichen besteht.
Diese 19 verschiedenen Testdaten geben somit 19 Werte für jede Kompression an. Der *Calgary Corpus* ist für viele Arten von Daten halbwegs repräsentativ. Somit haben die Ergebnisse eine gewisse Aussagekraft über alle möglichen Daten. Es können aber trotzdem andere Wert für ähnliche Daten herauskommen, weil es nur einige Beispiele sind. Deswegen sollten die Ergebnisse mit gewisser Vorsicht genossen werden. Besonders schwierig wird es eine Aussage zu treffen, wenn die Größen nah beieinander liegen. Die Aussage, welche in Kapitel 4.2 untersucht wird, lässt sich trotzdem gut erkennen und experimentell zeigen.

4.1.3. Die untersuchten Verfahren

In der Tabelle 8 werden die untersuchten Verfahren und deren Bezeichnungen dargestellt. Die neben der BWT verwendeten Algorithmen werden in Kapitel 2.3 erklärt.

Bezeichnung	Schritte des Verfahrens
`bzip2`	Das ist die Software *bzip2* [Sew07]. Die Beschreibung der Verfahrens ist in Kapitel 3.6.3 dargestellt. Es wurde zur Kompression die Option `-9` verwendet.
`M`	BWT → MTF → arithmetische Kodierung
`M1`	BWT → MTF-1 → arithmetische Kodierung
`MR`	BWT → MTF → MTF Lauflängenkodierung mit binärer Kodierung → arithmetische Kodierung
`M1R`	BWT → MTF-1 → MTF Lauflängenkodierung mit binärer Kodierung → arithmetische Kodierung
`BFRk`	Umwandlung der Folge in binäre Darstellung → BWT → binäre Lauflängenkodierung mit `k` Bits → arithmetische Kodierung
`BR`	Umwandlung der Folge in binäre Darstellung → BWT → binäre Lauflängenkodierung mit binärer Kodierung → arithmetische Kodierung
`BMR`	Umwandlung der Folge in binäre Darstellung → BWT → MTF-Kodierung → MTF Lauflängenkodierung mit binärer Kodierung → arithmetische Kodierung

`BMBR`	Umwandlung der Folge in binäre Darstellung → BWT → MTF-Kodierung → Umwandlung der Folge in Kodierung als Bytes → MTF Lauflängenkodierung mit binärer Kodierung → arithmetische Kodierung

Tabelle 8: Die untersuchte Verfahren

Es wurden auch noch andere Kombinationen dieser Schritte getestet, aber diese brachten deutlich schlechtere Ergebnisse als die hier aufgeführten.
Das Verfahren `bzip2` ist der Vergleichswert, da dies die am meisten verwendete Implementierung der BWT ist. Bei den nächsten vier Verfahren haben die Zeichen eine Größe von 8 Bits zum Zeitpunkt der BWT. Die letzten vier Verfahren haben zum Zeitpunkt der BWT eine Größe von einem Bit pro Zeichen. Es sei darauf hingewiesen, dass die Daten immer mit 8 Bits pro Zeichen eingegeben werden. Deswegen ist bei den Verfahren, welchen ein Bit Zeichengröße betrachten, der erste Schritt eine Umformung in das binäres Alphabet.
Bei den selbst implementierten Verfahren wird nie als erster Schritt eine Lauflängenkodierung durchgeführt, wie bei *bzip2*. Der Grund ist, dass damit immer schlechtere Ergebnisse erzielt werden als ohne. In *bzip2* ist dieser Schritt nur enthalten, weil er ursprünglich eingebaut wurde. Es sollten damit Fälle vermieden werden mit denen der BWT Algorithmus nicht umgehen kann. Es wurde aber festgestellt, dass solche Fälle nicht existieren. Dieses wird in [Sew05] beschrieben wird.
Es wurden auch Verfahren getestet, welche andere Anzahlen von Bits als Zeichen betrachten. Das wurde besonders als Darstellung der Zeichenfolge zum BWT-Schritt genutzt.

4.1.4. Die Ergebnisse

Bevor wir zu den Werten kommen sei noch kurz eine Feststellung zur Laufzeit erwähnt. Die Algorithmen, welche die binäre Darstellung zum BWT-Schritt nutzen, haben eine deutlich höhere Laufzeit. Das ist natürlich leicht zu erkennen, da acht mal so viele Rotationen sortiert werden müssen. Dazu kommt, dass das Sortieren eine Laufzeit von $\mathcal{O}(n \log n)$ hat.
In den Tabellen 9 und 10 sind die resultierenden Größen nach der Transformation für die verschiedenen Testdaten aufgeführt. Das Verfahren `BFRk` wurde mit dem optimalen k durchgeführt, sodass die resultierende Größe möglichst klein ist. Alle Werte in den Tabellen haben die Einheit Bits.

	`ohne`	`bzip2`	`M`	`M1`	`MR`	`M1R`
`bib`	890088	219736	254632	262175	**224867**	226946
`book1`	6150168	1860784	2121095	2081871	1954239	**1924971**
`book2`	4886848	1259544	1463919	1459635	1314789	**1305169**
`geo`	819200	455368	549960	550340	497968	**496564**
`news`	3016872	948800	1056612	1081985	**974309**	979704
`obj1`	172032	86296	93311	94515	**85612**	85795
`obj2`	1974512	611528	681891	713098	**620580**	630842
`paper1`	425288	132464	143417	147132	**134449**	135786
`paper2`	657592	200328	222732	224995	**205845**	206235

pic	4105728	398072	576594	541842	425813	**407165**
progc	316888	100352	107122	110796	**100821**	102155
progl	573168	124632	137057	145069	**126230**	128978
progp	395032	85680	92589	99673	**85860**	88545
trans	749560	143192	153244	170997	**142153**	149382
comb	25132976	6626776	7654175	7684123	6893535	**6868237**
tar	25231360	6881944	8269465	8271493	7316678	**7264555**
bible.txt	32379136	6765080	7995468	7932529	6728777	**6649565**
E.coli	37109520	10008032	9225914	**9210625**	9320679	9315239
world192.txt	19787200	3916664	4346646	4385605	3556836	**3546538**

Tabelle 9: Hier sind die Größen nach der Ausführung der untersuchten Verfahren aufgeführt, welche direkt auf Bytes angewendet werden. Alle Angaben haben die Einheit Bits. Horizontal sind die Verfahren dargestellt und vertikal die Testdaten. In Fett sind die Werte markiert, welche am kleinsten sind unter den selbst implementierten Verfahren.

	ohne	BFRk	BR	BMR	BMBR
bib	890088	254376	250892	**249615**	253969
book1	6150168	1904171	1949299	**1896867**	1901854
book2	4886848	1348120	1358621	**1339292**	1352328
geo	819200	**546523**	586833	555438	552010
news	3016872	1064080	1066393	**1061788**	1069960
obj1	172032	102651	103708	**102446**	103957
obj2	1974512	762469	**760770**	762514	766138
paper1	425288	154164	151577	**151559**	154109
paper2	657592	222046	220181	**217872**	221528
pic	4105728	441626	427263	426420	**416686**
progc	316888	120161	**118038**	118188	120483
progl	573168	147342	**143082**	143586	145659
progp	395032	107105	**104052**	104287	106217
trans	749560	183023	**180173**	180497	180829
comb	25132976	7357857	7420882	**7310369**	7345727
tar	25231360	**7505559**	7610854	7511408	7562227
bible.txt	32379136	6445883	6574161	**6438399**	6452971
E.coli	37109520	9811171	10602339	10051106	**9683059**
world192.txt	19787200	3682783	3712013	**3667625**	3714028

Tabelle 10: Hier sind die Größen nach der Ausführung der untersuchten Verfahren aufgeführt, welche zuerst in Bits umwandeln. Alle Angaben haben die Einheit Bits. Horizontal sind die Verfahren dargestellt und vertikal die Testdaten. In Fett sind die Werte markiert, welche am kleinsten sind unter den selbst implementierten Verfahren.

Im nächsten Kapitel werden wir kurz schauen, was wir aus diesen Werten erkennen

können.

4.1.5. Schlussfolgerungen aus den Ergebnissen

Zuerst betrachten wir die Verfahren, welche die BWT auf Bytes anwenden. Es lässt sich erkennen, dass `M` und `M1` fast immer schlechter sind als `MR` und `M1R`. Nur bei `E.coli` sieht es anderes aus.
Der Unterschied zwischen der Verwendung von MTF und MTF-1 ist nur marginal. In manchen Fällen ist die MTF-Kodierung besser und in anderen die MTF-1-Kodierung. Es lässt sich nicht wirklich sagen, dass ein Verfahren besser ist.
Im Vergleich zu `bzip2` sind `MR` und `M1R` zum großen Teil etwas schlechter. Besonders bei den größeren Dateien `bible.txt`, `E.coli` und `world192.txt` ist aber `bzip2` schlechter. Das liegt einfach daran, dass die maximale Blockgröße von `bzip2` 900 Kilobyte ist. Eine Verbesserung der Kompression durch größere Blockgröße lässt sich aus der Herleitung der Kompression in Kapitel 3.5 erkennen. Die BWT approximiert das Finden von Mustern und je größer der Block ist, umso mehr Möglichkeiten für Muster gibt es.
Jetzt betrachten wir noch die Verfahren, welche die BWT auf Bits anwenden. Die Ergebnisse dieser Verfahren liegen sehr nah beieinander. Die Unterschiede sind nur marginal. Es lässt sich auch nicht wirklich eine Aussage treffen, welche dieser Verfahren sehr schlecht sind. Es gibt zu jedem Verfahren immerhin mehr als ein Beispiel für welches es am besten ist.
Die beiden Verfahren `BFRk` und `BR` funktionieren beide hauptsächlich mit kompletter Lauflängenkodierung. Aus diesem Grund liegen sie sehr nah beieinander.
Das Verfahren `BMR` ist aber für viele Testdaten am besten. Auch bei den Daten, für welche das Verfahren nicht optimal ist, ist es nur minimal schlechter. Das Datum `E.coli` ist dabei eine Ausnahme, da der Wert merklich schlechter ist. Allgemein versucht das Verfahren das Vorgehen von `bzip2` möglichst gut nachzuahmen. So könnte die gute Leistung zu Stande kommen.
Das Verfahren `BMBR` ist wiederum eher mittelmäßig gut und nicht besonders zu beachten.
Wir betrachten noch den Vergleich zwischen den beiden Klassen von Verfahren. Wir unterscheiden danach, ob die Daten im BWT Schritt als Bytes oder Bits verarbeitet werden. Im Allgemeinen sind die Verfahren besser, welche mit Bytes arbeiten.
Zum Schluss sei noch auf die kleine Feststellung hingewiesen, dass `comb` immer kleiner ist als `tar`. Dabei betrachten beide Werte die gleichen grundlegenden Daten.
Die Erkenntnisse aus den Daten werden wir im nächsten Kapitel versuchen theoretisch zu begründen. Besonders werden die Sachen betrachtet, welche nicht trivial zu sehen sind.

4.2. Theoretische Untersuchung

Die folgenden Argumentationen haben keinen Anspruch ein kompletter Beweis zu sein. Besonders, weil die Aussagen hier nicht immer zutreffen, sondern nur im Großteil der Fälle. Möglich sind hier Abschätzungen für den worst-case und wünschenswert wären average-case Abschätzungen. Es ist somit schwierig zu zeigen, dass eine Aussage für alle möglichen Zeichenfolgen gilt. Tatsächlich haben wir bereits Gegenbeispiele für viele solcher Arten von Aussagen im vorherigen Kapitel gefunden.

4.2.1. Effizienz von MTF-Lauflängenkodierung

Die erste Feststellung ist, dass `M` und `M1` fast immer schlechter als `MR` und `M1R` sind. Es wird also durch das Hinzufügen der MTF-Lauflängenkodierung des Zeichens `0` eine bessere Effizienz der Kompression erreicht. Das ist eine bereits bekannte und viel untersuchte Tatsache. Deswegen existieren für diese beiden Verfahren `M` und `MR` worst-case Abschätzungen, welche diese Feststellung unterstützen.
In [Man01] sind solche Abschätzungen angegeben. Dafür benötigen wir die in Kapitel 2.2 eingeführte Informationstheorie. Mit $M(x)$ und $MR(x)$ bezeichnen wir im Folgenden die Größe nach der Kompression der Folge x.
Damit lautet die zu zeigende Aussage, dass für den Großteil der Zeichenfolgen $x \in \Sigma^*$

$$M(x) \geq MR(x)$$

gilt. Für diese beiden Werte nutzen wir die folgenden oberen Abschätzungen.
Die Abschätzung für das Verfahren ohne MTF-Lauflängenkodierung mit Eingabe $x \in \Sigma^*$ und $k \geq 0$ ist

$$M(x) \leq 8|x|H_k(x) + \left(\mu + \frac{2}{25}\right)|x| + |\Sigma|^k(2|\Sigma|\log|\Sigma| + 9).$$

Die Abschätzung für das Verfahren mit MTF-Lauflängenkodierung mit Eingabe $x \in \Sigma^*$, $k \geq 0$ und Konstante g_k ist

$$MR(x) \leq (5 + 3\mu)|x|H_k^*(x) + g_k.$$

Dabei ist $H_k^*(x)$ die modifizierte k-te empirische Entropie. Diese berücksichtigt, dass zur Darstellung einer Zeichenfolge s mit $H_0(s) = 0$ trotzdem Bits benötigt werden um die Daten zu kodieren. Sonst bleibt bei anderen Werten von $H_0(s)$ alles gleich. Der wichtige Punkt ist, dass $H_k^*(x) \geq H_k(x)$ ist, aber maximal $(1 + \lfloor\log|x|\rfloor)/|x|$. Die relevanten Terme dieser Abschätzungen sind $8|x|H_k(x)$ bzw. $(5 + 3\mu)|x|H_k^*(x)$. Das μ ist abhängig vom verwendeten Entropie-Kodierer. Für einen arithmetischen Kodierer ist $\mu \approx 10^{-2}$.
Der Beweis für diese Abschätzungen ist in [Man01] zu finden. In dem Bewies wird für das Verfahren mit MTF-Lauflängenkodierung sogar gezeigt, dass es lokal λ-optimal ist. Das sagt aus, dass für eine beliebige Untererteilung der Zeichenfolge die Kompressionsgröße durch die Entropie beschränkt werden kann. Dabei wird die Entropie für jeden Teil der Folge berechnet und dann summiert. Diese Eigenschaften erfüllen viele andere Kompressionsalgorithmen nicht. Aber darauf werden wir hier nicht weiter eingehen.
Bei diesen Abschätzungen kann nicht einfach gesagt werden, dass eine immer größer oder kleiner ist. Aber im Allgemeinen ist die Abschätzung von dem Verfahren ohne MTF-Lauflängenkodierung größer. Das zeigt die Aussage aber noch lange nicht, da es nur eine obere Schranke ist. Auch gibt es keinen Beweis dafür, dass diese Abschätzungen ein Supremum darstellt. Es gibt aber einen guten Hinweis darauf, dass die zu zeigende Aussage richtig ist. Wir werden es aber bei diesen Abschätzungen belassen, da keine besseren bekannt sind.
Viele Untersuchungen deuten darauf hin, dass die Aussage stimmt, aber das konnte bisher nicht eindeutig gezeigt werden. Auch müssten dann Bedingungen gefunden werden für die Fälle, welche die Ungleichung verletzten.

4.2.2. Unterschied der Verfahren auf Bytes und Bits

Wir betrachten nun den Unterschied zwischen den zwei Klassen von Verfahren. Die eine Klasse Y nutzt die Darstellung der Zeichenfolge in Form von Bytes in der BWT-Phase. Die andere Klasse I nutzt die Darstellung als Bits in dieser Phase.
Wir wollen zeigen, dass das beste Verfahren aus Y in den meisten Fällen besser ist als das beste Verfahren aus I. Dazu betrachten wir die Argumentation aus Kapitel 3.5. Wir werden dazu die Darstellung der Zeichen als Bitfolge betrachten und wie sich das auf die sortierten Rotationen auswirkt.
Zuerst schauen wir uns nur die Rotationen an, welche mit dem ersten Bit eines Zeichens beginnen. Nach der lexikografischen Sortierung und der binären Darstellung von Zeichen folgt, dass die Sortierung gleich bleibt. Also können in der binären Darstellung 8 Bits wieder zu einem Zeichen zurückgewandelt werden und die Rotationen sind trotzdem gleich sortiert.
Jetzt existieren zu jeder Rotation noch sieben weitere Rotationen. Diese werden irgendwo einsortiert. Dabei kann als guter Fall die Rotation so einsortiert werden, dass sie eine Folge von gleichen Zeichen in der Ausgabe nicht unterbricht. Der schlechte Fall ist die Einsortierung in der Mitte von einer Folge gleicher Zeichen in der Ausgabe. Sollte dann noch das letzte Zeichen unterschiedlich zu den anderen sein, dann wird die Folge gleicher Zeichen unterbrochen. Das lässt sich an einem Beispiel verdeutlichen. Wir betrachten die Zeichenfolge von den Zeichenwerten `100,100,118,70`. In Abbildung 18 sind die Rotationen in binär und sortiert dargestellt, welche am Anfang eines Zeichens beginnen.

```
01000110,01100100,01100100,01110110
01100100,01100100,01110110,01000110
01100100,01110110,01000110,01100100
01110110,01000110,01100100,01100100
```

Abbildung 18: Lexikografische Sortierung aller Rotationen, welche am Anfang eines Zeichens beginnen, der Zeichenfolge `100,100,118,70`. Die Zeichenfolge ist binär dargestellt. Zur besseren Darstellung sind die originalen Zeichen mit `,` getrennt.

Die Abbildung 19 demonstriert durch Hinzufügen einer weiteren Rotation, dass nun das vorherige Muster unterbrochen wird. Nach der Feststellung in Kapitel 3.5 ist das schlecht für die Kompression, weil in größeren Beispielen die Folge gleicher Zeichen so unterbrochen wird. Denn so ist die gewollte Eigenschaft, dass gleiche Zeichen möglichst beieinander stehen nicht mehr gegeben. Das kann an diesen kleinen Beispiel leider nur erahnt werden.
Die interessante Beobachtung hierbei ist die Länge der gleichen Zeichen am Anfang von zwei Rotationen. Damit eine Rotation zwischen zwei Anderen sortiert wird, muss die Länge die gleichen Zeichen zu dieser Neuen mindestens so lang sein wie zwischen den beiden Anderen. Dadurch entstehen wiederum lange Muster. Diese führen dann durch Rotation wieder zu langen Folgen gleicher Zeichen, bei der Betrachtung der letzten Spalte. Somit gibt es zwei grundlegende Effekte. Die Unterbrechung von Mustern und die Entstehung längerer Muster. Die Unterbrechung eines Musters ist aber zu häufig, sodass der andere Effekt geringer ist. Der Grund dafür ist die Vermutung, dass in der Betrachtung

```
01000110,01100100,01100100,01110110
01100100,01100100,01110110,01000110
0110,01000110,01100100,01100100,0111
01100100,01110110,01000110,01100100
01110110,01000110,01100100,01100100
```

Abbildung 19: Lexikografische Sortierung aller Rotationen, welche am Anfang eines Zeichens beginnen, der Zeichenfolge `100,100,118,70`. Es wurde eine weitere Rotation hinzugefügt, welche in der Mitte eines Bytes beginnt. Die Zeichenfolge ist binär dargestellt. Zur besseren Darstellung sind die originalen Zeichen mit `,` getrennt.

der Bits wahrscheinlich einiges an Struktur verloren geht. Besonders die Rotation, welche in der Mitte eines Bytes beginnen, haben kaum eine Aussagekraft und sind eher "zufällig". Auch ist zu beachten, dass die Zeichenfolge acht mal so viele Zeichen enthält, welche kodiert werden müssen.
Interessant ist, dass der gleiche Effekt wahrscheinlich auch die Ursache ist, dass `comb` kleiner als `tar` ist. Denn in `tar` gehen durch die vielen verschieden Arten von Dateien spezifische Struktur von jeder Datei verloren. In `comb` werden die Dateien einzeln komprimiert und damit kann die Struktur jeder Art von Datei einzeln ausgenutzt werden. Mit diesen Strukturen ist gemeint, dass Text-Dateien zum Beispiel nur aus Wörtern bestehen, während bei Bild-Dateien alles Möglich ist an Zeichenkombinationen.

4.2.3. Weitere Anmerkungen zu den theoretischen Untersuchungen

Die zuvor angebrachten Argumente sind kein Beweis. Somit wäre es wünschenswert, wenn für die experimentellen Erkenntnisse in Zukunft besser Argumente gefunden werden. Am besten wäre natürlich ein Beweis dieser Aussagen und das Finden von Bedingungen für die diese Aussagen nicht gelten.
Das würde ich aber als durchaus schwierig erachten, da dazu ein formales Verständnis für die BWT vorhanden sein sollt. Abschätzungen, wie sie in [Man01] gebracht wurden, sind auch nur für die ganze Vorgehensweise gültig und nicht für einzelne Teile von dieser. Es gibt auch keine gute Abschätzung über die Eigenschaften der Zeichenfolge direkt nach der BWT. Wichtig ist somit für zukünftige Untersuchungen, dass die BWT besser verstanden wird, um eine handliche formale Beschreibung der Transformation zu ermöglichen. Damit sind dann möglicherweise bessere Abschätzungen möglich.

Literaturverzeichnis

[Abe03] Juergen Abel. Improvements to the burrows-wheeler compression algorithm: After bwt stages. *ACM Trans. Computer Systems*, 2003.

[BW94] Michael Burrows and David J. Wheeler. A block-sorting lossless data compression algorithm. 1994.

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

[LMW05] Ross A Lippert, Clark M Mobarry, and Brian P Walenz. A space-efficient construction of the burrows–wheeler transform for genomic data. *Journal of Computational Biology*, 12(7):943–951, 2005.

[Man01] Giovanni Manzini. An analysis of the burrows—wheeler transform. *Journal of the ACM (JACM)*, 48(3):407–430, 2001.

[Sew05] Julian Seward. bzip2 Manual: 4.1. limitations of the compressed file format, 2005. abgerufen am 09.05.2018. URL: `http://www.bzip.org/1.0.3/html/misc.html`.

[Sew07] Julian Seward. bzip2 Manual, 2007. abgerufen am 09.05.2018. URL: `http://www.bzip.org/docs.html`.

[Sha01] Claude E. Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

Anhang

A. Quellcode der Implementierung

Der Quellcode ist in der Programmiersprache *Rust* geschrieben.

A.1. Hauptmodul

Der Quellcode 1 beinhaltet die Hauptfunktion `main`. Die Unterfunktionen in diesem Quellcode dienen der Strukturierung in die verschiedenen Vorgehensweisen der Transformation. In diesen Funktionen werden die Transformationen aufgerufen und die Größen der transformierten Folgen ausgegeben.

Quellcode 1: Main-Funktion und Hilfsfunktionen zum Aufrufen der Transformationen und Evaluationen von Folgen

```
1  #![allow(dead_code)]
2
3  extern crate bwt;
4
5  use bwt::trans::*;
6  use bwt::process;
7  use std::fs::File;
8  use std::io::prelude::*;
9
10 #[derive(Debug)]
11 enum CharSize {
12     Half,
13     Double,
14     Quad,
15     Octa,
16 }
17
```

```
18 const BLOCKS: [usize; 15] = [512, 1024, 2048, 4096, 8192, 16384, 32768, 65536,
       ↪ 131072, 262144, 524288, 1048576, 2097152, 4194304, 8388608];
19 const FILES: [&str; 17] = ["bib", "book1", "book2", "geo", "news", "obj1", "
       ↪ obj2", "paper1", "paper2", "pic", "progc", "progl", "progp", "trans", "
       ↪ bible.txt", "E.coli", "world192.txt"];
20
21 /// Main function to start the program, load the input data and call the
       ↪ transformations
22 fn main() {
23     for i in FILES.iter() {
24         println!("\n{:?}:", i);
25         let mut infile = File::open(i).expect("File cannot be opened!");
26         let mut input: Vec<u8> = Vec::new();
27         infile.read_to_end(&mut input).expect("Unable to read file!");
28         println!("Groesse ohne weitere Kodierung (ohne): {:?}", &input.len() *
               ↪ 8);
29         println!("Informationsgehalt der Daten: {}", process::
               ↪ evaluate_informationu8(&input));
30         //input = run_length_preprocessing(&input);
31         bytes(&input);
32         binary(&input);
33         //diff_char(&input, CharSize::Half);
34     }
35 }
36
37 /// Function that preforms the transformations, which use binary representation
       ↪  for BWT-phase
38 /// # Arguments
39 /// * ‘input‘ - Vector of the input data.
40 fn binary(input: &Vec<u8>) {
41     println!("\nGroesse von Bits:");
42     let input_bits = process::bytes_to_bits(&input);
43     let pair = transform_bit(&input_bits);
44     let (bwt, _start) = pair;
45     for i in 0..31 {
46         let (artih, overhead) = arithmetic_u64(&run_length(&bwt, i + 2));
47         println!("Groesse von Lauflaengenkodierung mit {} Bits (BFRk): {:?}", i
               ↪ + 2, &artih.len() + ((i + 2) * overhead));
48     }
49     let (artih, overhead) = arithmetic_byte(&bin_run_length(&bwt));
50     println!("Groesse von dynamischer Lauflaengenkodierung(BR): {:?}", &artih.
           ↪ len() + (8 * overhead));
51     let (artih, overhead) = arithmetic_u16(&mtf_run_length(&mtf_bit(&bwt)));
52     println!("Groesse von MTF und dann bzip2 Lauflaengenkodierung (BMR): {:?}",
           ↪ &artih.len() + (8 * overhead));
53     let (artih, overhead) = arithmetic_u16(&mtf_run_length(&process::
           ↪ bits_to_bytes(&mtf_bit(&bwt))));
54     println!("Groesse von MTF dann in Byte und dann Lauflaengenkodierung (BMBR):
           ↪  {:?}", &artih.len() + (8 * overhead));
55 }
56
57 /// Function that preforms the transformations, which use byte representation
       ↪ for BWT-phase
```

```
58 /// # Arguments
59 /// * ‘input‘ - Vector of the input data.
60 fn bytes(input: &Vec<u8>) {
61     println!("\nGroesse von Bytes:");
62     let pair = transform_byte(&input);
63     let (bwt, start) = pair;
64     let (artih, overhead) = arithmetic_byte(&mtf_byte(&bwt));
65     println!("MTF und arithmetischer Kodierung (M): {:?}", artih.len() + (
           ↪ overhead * 8));
66     let (artih, overhead) = arithmetic_byte(&mtf1_byte(&bwt));
67     println!("MTF-1 und arithmetischer Kodierung (M1): {:?}", artih.len() + (
           ↪ overhead * 8));
68     let (artih, overhead) = arithmetic_u16(&mtf_run_length(&mtf_byte(&bwt)));
69     println!("MTF dann Lauflaengenkodierung und arithmetischer Kodierung (MR):
           ↪ {:?}", artih.len() + (overhead * 8));
70     println!("Informationsgehalt der Daten: {}", process::
           ↪ evaluate_informationu16(&mtf_run_length(&mtf_byte(&bwt))));
71     let (artih, overhead) = arithmetic_u16(&mtf_run_length(&mtf1_byte(&bwt)));
72     println!("MTF-1 dann Lauflaengenkodierung und arithmetischer Kodierung (M1R)
           ↪ : {:?}", artih.len() + (overhead * 8));
73     println!("Informationsgehalt der Daten: {}", process::
           ↪ evaluate_informationu16(&mtf_run_length(&mtf1_byte(&bwt))));
74 }
75
76 /// Function that preforms the transformations, which use different sizes for
       ↪ the representation in BWT-phase
77 /// # Arguments
78 /// * ‘input‘ - Vector of the input data.
79 /// * ‘size‘ - Argument to to choose the size for the representation.
80 fn diff_char(input: &Vec<u8>, size: CharSize) {
81     let input_bits = process::bytes_to_bits(&input);
82     let mut input;
83     match size {
84         CharSize::Half => {
85             input = process::bits_to_4(&input_bits);
86             println!("\nBlockgroesse von 4 Bit:");
87         },
88         CharSize::Double => {
89             input = process::bits_to_16(&input_bits);
90             println!("\nBlockgroesse von 16 Bit:");
91         },
92         CharSize::Quad => {
93             input = process::bits_to_32(&input_bits);
94             println!("\nBlockgroesse von 32 Bit:");
95         },
96         CharSize::Octa => {
97             input = process::bits_to_64(&input_bits);
98             println!("\nBlockgroesse von 64 Bit:");
99         },
100    }
101    let pair = transform(&input);
102    let (bwt, start) = pair;
103    let (artih, overhead) = arithmetic_u64(&mtf(&bwt));
```

```
104     println!("MTF und arithmetischer Kodierung fuer Groesse {:?}: {:?}", size,
            ↪ artih.len() + (overhead * 8));
105 }
```

A.2. Modul für die Transformationen

Im Quellcode 2 sind die Transformationen, welche verwendet werden, dargestellt. Zu diesen gehört nicht nur die BWT, sondern auch alle Transformationen vor und nach dieser.

Quellcode 2: Transformationen von den Zeichenfolgen

```
1  #![allow(dead_code)]
2
3  extern crate rand;
4
5  use std::collections::HashMap;
6  use trans::rand::{Rng, thread_rng};
7
8  const POWS8: [u64; 8] = [128, 64, 32, 16, 8, 4, 2, 1];
9  const POWS32: [u32; 32] = [2147483648, 1073741824, 536870912, 268435456,
       ↪ 134217728, 67108864, 33554432, 16777216, 8388608, 4194304, 2097152,
       ↪ 1048576, 524288, 262144, 131072, 65536, 32768, 16384, 8192, 4096, 2048,
       ↪ 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1];
10
11 /// Funtion that starts the BWT transformation for 64 Bit numbers
12 /// Returns the tuple of BWT transformed Vector and the index of the original
       ↪ rotation.
13 /// It generates the Vector of the input appended by the input, which
       ↪ represents all possible rotations. Then it calls the sorting function
       ↪ and finally extracts the last column
14 /// # Arguments
15 /// * ‘input‘ - Input vector for the transformation.
16 pub fn transform(input: &Vec<u64>) -> (Vec<u64>, u64) {
17     let length = input.len();
18     let mut ds = input.clone();
19     ds.append(&mut input.clone());
20     let sorted = sort(&ds, &(length as u64));
21     let mut last_col: Vec<u64> = Vec::new();
22     let mut start: u64 = 0;
23     for (num, i) in sorted.iter().enumerate() {
24         last_col.push(ds[(*i as usize) + length - 1]);
25         if *i == 0 {
26             start = num as u64;
27         }
28     }
29     return(last_col.clone(), start);
30 }
31
32 /// Funtion that calls the sorting algorithm for 64 Bit numbers
33 /// Returns the Vector of indixes corresponding to the sorted rotations.
34 /// It generates the original Vector of indexes to represent all possible
       ↪ rotations.
```

```
35 /// # Arguments
36 /// * ‘data‘ - The input vector with the same input appended.
37 /// * ‘len‘ - The length of the original input.
38 fn sort(data: &Vec<u64>, len: &u64) -> Vec<u64> {
39     let mut index: Vec<u64> = Vec::new();
40     for i in 0..*len {
41         index.push(i);
42     }
43     return line_quicksort(data, &mut index, *len as usize);
44     //return line_insertsort(data, *len as usize);
45 }
46 
47 /// Randomized quicksort algorithm for Vectors of 64 Bit numbers
48 /// Returns the Vector of sorted indizes by recursion.
49 /// # Arguments
50 /// * ‘data‘ - The input vector with the same input appended.
51 /// * ‘index‘ - The Vector of indexes to sort.
52 /// * ‘len‘ - The length of the original input.
53 fn line_quicksort(data: &Vec<u64>, index: &Vec<u64>, len: usize) -> Vec<u64> {
54     if index.len() <= 1 {
55         return index.clone();
56     }
57     let pivot = index[index.len() / 2] as usize;
58     let mut less: Vec<u64> = Vec::new();
59     let mut greater: Vec<u64> = Vec::new();
60     let mut equal: Vec<u64> = Vec::new();
61     for i in index.iter() {
62         if &data[pivot..(pivot + len)] < &data[*i as usize..(*i as usize + len)]
               ↪ {
63             greater.push(*i);
64         } else if &data[pivot..(pivot + len)] > &data[*i as usize..(*i as usize
               ↪ + len)] {
65             less.push(*i);
66         } else {
67             equal.push(*i);
68         }
69     }
70     let mut index = line_quicksort(data, &mut less, len).clone();
71     index.append(&mut equal);
72     index.append(&mut line_quicksort(data, &mut greater, len).clone());
73     return index.clone();
74 }
75 
76 /// Insertionsort algorithm for Vectors of 64 Bit numbers
77 /// Returns the Vector of sorted indixes.
78 /// # Arguments
79 /// * ‘data‘ - The input vector with the same input appended.
80 /// * ‘len‘ - The length of the original input.
81 fn line_insertsort(data: &Vec<u64>, len: usize) -> Vec<u64> {
82     let mut sorted: Vec<u64> = Vec::new();
83     let mut new: Vec<u64>;
84     sorted.push(0 as u64);
85     for i in 1..len {
```

```
86          let pos = search_pos(&data, &sorted, i, len);
87          if pos != 0 {
88          }
89          new = Vec::new();
90          for i in sorted[..pos].iter() {
91              new.push(*i);
92          }
93          new.push(i as u64);
94          for i in sorted[pos..].iter() {
95              new.push(*i);
96          }
97          sorted = new;
98      }
99      return sorted;
100 }
101
102 /// Calculates the position where to insert the current index for the
        ↪ insertionsort algorithm
103 /// Returns the position where to insert the current index.
104 /// # Arguments
105 /// * ‘data‘ - The input vector with the same input appended.
106 /// * ‘sorted‘ - The current Vector of sorted indexes.
107 /// * ‘index‘ - The current index to insert in the Vector.
108 /// * ‘len‘ - The length of the original input.
109 fn search_pos(data: &Vec<u64>, sorted: &Vec<u64>, index: usize, len: usize) ->
        ↪ usize {
110     let mut left = 0;
111     let mut right = index;
112     while right > left {
113         let mid = (right + left) / 2;
114         if data[index..(index + len)] > data[sorted[mid] as usize..(sorted[mid]
                ↪ as usize + len)] {
115             left = mid + 1;
116         } else {
117             right = mid;
118         }
119     }
120     return right;
121 }
122
123 /// Funtion that starts the BWT transformation for bytes
124 /// Returns the tuple of BWT transformed Vector and the index of the original
        ↪ rotation.
125 /// It generates the Vector of the input appended by the input, which
        ↪ represents all possible rotations. Then it calls the sorting function
        ↪ and finally extracts the last column
126 /// # Arguments
127 /// * ‘input‘ - Input vector for the transformation.
128 pub fn transform_byte(input: &Vec<u8>) -> (Vec<u8>, u64) {
129     let length = input.len();
130     let mut ds = input.clone();
131     ds.append(&mut input.clone());
132     let sorted = sort_byte(&ds, &(length as u64));
```

```
133     let mut last_col: Vec<u8> = Vec::new();
134     let mut start: u64 = 0;
135     for (num, i) in sorted.iter().enumerate() {
136         last_col.push(ds[(*i as usize) + length - 1]);
137         if *i == 0 {
138             start = num as u64;
139         }
140     }
141     return(last_col.clone(), start);
142 }
143 
144 /// Funtion that calls the sorting algorithm for bytes
145 /// Returns the Vector of indixes corresponding to the sorted rotations.
146 /// It generates the original Vector of indexes to represent all possible
        ↪ rotations.
147 /// # Arguments
148 /// * ‘data‘ - The input vector with the same input appended.
149 /// * ‘len‘ - The length of the original input.
150 fn sort_byte(data: &Vec<u8>, len: &u64) -> Vec<u64> {
151     let mut index: Vec<u64> = Vec::new();
152     for i in 0..*len {
153         index.push(i);
154     }
155     return line_quicksort_byte(data, &mut index, *len as usize);
156     //return line_insertsort(data, *len as usize);
157 }
158 
159 /// Randomized quicksort algorithm for Vectors of bytes
160 /// Returns the Vector of sorted indizes by recursion.
161 /// # Arguments
162 /// * ‘data‘ - The input vector with the same input appended.
163 /// * ‘index‘ - The Vector of indexes to sort.
164 /// * ‘len‘ - The length of the original input.
165 fn line_quicksort_byte(data: &Vec<u8>, index: &Vec<u64>, len: usize) -> Vec<u64
        ↪ > {
166     if index.len() <= 1 {
167         return index.clone();
168     }
169     let mut rng = thread_rng();
170     let num: usize = rng.gen_range(0, index.len());
171     let pivot = index[num] as usize;
172     let mut less: Vec<u64> = Vec::new();
173     let mut greater: Vec<u64> = Vec::new();
174     let mut equal: Vec<u64> = Vec::new();
175     for i in index.iter() {
176         if &data[pivot..(pivot + len)] < &data[*i as usize..(*i as usize + len)]
                ↪  {
177             greater.push(*i);
178         } else if &data[pivot..(pivot + len)] > &data[*i as usize..(*i as usize
                ↪ + len)] {
179             less.push(*i);
180         } else {
181             equal.push(*i);
```

```
182         }
183     }
184     let mut index = line_quicksort_byte(data, &mut less, len).clone();
185     index.append(&mut equal);
186     index.append(&mut line_quicksort_byte(data, &mut greater, len).clone());
187     return index.clone();
188 }
189 
190 /// Funtion that starts the BWT transformation for bits
191 /// Returns the tuple of BWT transformed Vector and the index of the original
    ↪ rotation.
192 /// It generates the Vector of the input appended by the input, which
    ↪ represents all possible rotations. Then it calls the sorting function
    ↪ and finally extracts the last column
193 /// # Arguments
194 /// * `input` - Input vector for the transformation.
195 pub fn transform_bit(input: &Vec<u8>) -> (Vec<u8>, u64) {
196     let length = input.len();
197     let mut ds = input.clone();
198     ds.append(&mut input.clone());
199     ds.append(&mut input.clone());
200     let sorted = sort_bit(&ds, &(length as u64));
201     let mut last_col: Vec<u8> = Vec::new();
202     let mut start: u64 = 0;
203     for (num, i) in sorted.iter().enumerate() {
204         last_col.push(ds[(*i as usize) + length - 1]);
205         if *i == 0 {
206             start = num as u64;
207         }
208     }
209     return (last_col.clone(), start);
210 }
211 
212 /// Funtion that calls the sorting algorithm for bits
213 /// Returns the Vector of indixes corresponding to the sorted rotations.
214 /// It generates the original Vector of indexes to represent all possible
    ↪ rotations.
215 /// # Arguments
216 /// * `data` - The input vector with the same input appended.
217 /// * `len` - The length of the original input.
218 fn sort_bit(data: &Vec<u8>, len: &u64) -> Vec<u64> {
219     let mut index: Vec<u64> = Vec::new();
220     for i in 0..*len {
221         index.push(i);
222     }
223     //return line_quicksort_bit(data, &mut index, *len as usize);
224     return line_fast_quicksort_bit(&[bits_to_u8(&data[0..data.len() - 8]),
        ↪ bits_to_u8(&data[1..data.len() - 7]), bits_to_u8(&data[2..data.len()
        ↪ - 6]), bits_to_u8(&data[3..data.len() - 5]), bits_to_u8(&data[4..data
        ↪ .len() - 4]), bits_to_u8(&data[5..data.len() - 3]), bits_to_u8(&data
        ↪ [6..data.len() - 2]), bits_to_u8(&data[7..data.len() - 1])], &mut
        ↪ index, *len as usize / 8);
225 }
```

```
226
227 /// Randomized quicksort algorithm for Vectors of bits
228 /// Returns the Vector of sorted indizes by recursion.
229 /// # Arguments
230 /// * ‘data‘ - The input vector with the same input appended.
231 /// * ‘index‘ - The Vector of indexes to sort.
232 /// * ‘len‘ - The length of the original input.
233 fn line_quicksort_bit(data: &Vec<u8>, index: &Vec<u64>, len: usize) -> Vec<u64>
    ↪ {
234     if index.len() <= 1 {
235         return index.clone();
236     }
237     let pivot = index[index.len() / 2] as usize;
238     let mut less: Vec<u64> = Vec::new();
239     let mut greater: Vec<u64> = Vec::new();
240     let mut equal: Vec<u64> = Vec::new();
241     for i in index.iter() {
242         if &data[pivot..(pivot + len)] < &data[*i as usize..(*i as usize + len)]
            ↪ {
243             greater.push(*i);
244         } else if &data[pivot..(pivot + len)] > &data[*i as usize..(*i as usize
            ↪ + len)] {
245             less.push(*i);
246         } else {
247             equal.push(*i);
248         }
249     }
250     let mut index = line_quicksort_bit(data, &mut less, len).clone();
251     index.append(&mut equal);
252     index.append(&mut line_quicksort_bit(data, &mut greater, len).clone());
253     return index.clone();
254 }
255
256 /// Presumably faster randomized quicksort algorithm for Vectors of bits
257 /// Returns the Vector of sorted indizes by recursion.
258 /// # Arguments
259 /// * ‘data‘ - The array of 8 input vector with the same input appended, which
    ↪ begin with indexes 0 to 7 and only contain every 8th character.
260 /// * ‘index‘ - The Vector of indexes to sort.
261 /// * ‘len‘ - The length of the original input.
262 fn line_fast_quicksort_bit(data: &[Vec<u8>; 8], index: &Vec<u64>, len: usize)
    ↪ -> Vec<u64> {
263     if index.len() <= 1 {
264         return index.clone();
265     }
266     let mut rng = thread_rng();
267     let num: usize = rng.gen_range(0, index.len());
268     let pivot = index[num] as usize;
269     let mut less: Vec<u64> = Vec::new();
270     let mut greater: Vec<u64> = Vec::new();
271     let mut equal: Vec<u64> = Vec::new();
272     for i in index.iter() {
273         if &data[pivot % 8 as usize][(pivot / 8) as usize..((pivot / 8) + len)
```

```
            ↪ as usize] < &data[*i as usize % 8][*i as usize / 8..((*i as usize
            ↪  / 8) + len)] {
274             greater.push(*i);
275         } else if &data[pivot % 8 as usize][pivot / 8..((pivot / 8) + len)] > &
                ↪ data[*i as usize % 8][*i as usize / 8..((*i as usize / 8) + len)]
                ↪  {
276             less.push(*i);
277         } else {
278             equal.push(*i);
279         }
280     }
281     let mut index = line_fast_quicksort_bit(data, &mut less, len).clone();
282     index.append(&mut equal);
283     index.append(&mut line_fast_quicksort_bit(data, &mut greater, len).clone());
284     return index.clone();
285 }
286 
287 /// Reverse transformation of the BWT for 64 Bit numbers
288 /// Returns the original Vector of the BWT transformed Vector.
289 /// # Arguments
290 /// * ‘input‘ - The BWT transformed Vector.
291 /// * ‘start‘ - The index of the rotation containing the original Vector.
292 pub fn untransform(input: &Vec<u64>, start: &u64) -> Vec<u64> {
293     let mut sorted = input.clone();
294     sorted.sort();
295     let mut pos = *start as usize;
296     let mut output: Vec<u64> = Vec::new();
297     for _i in 0..input.len() {
298         output.push(sorted[pos]);
299         pos = next_pos(&input, &sorted, &pos);
300     }
301     return output;
302 }
303 
304 /// Calculates the next index for the reverse BWT of 64 Bit numbers
305 /// Returns index of the corresponding character in BWT transformed Vector.
306 /// # Arguments
307 /// * ‘last‘ - The BWT transformed Vector.
308 /// * ‘first‘ - The sorted BWT transformed Vector.
309 /// * ‘lastpos‘ - The current index of the reverse BWT.
310 fn next_pos(last: &Vec<u64>, first: &Vec<u64>, lastpos: &usize) -> usize {
311     let mut until = 0;
312     let value = first[*lastpos];
313     for i in 0..(lastpos + 1) {
314         if value == first[i] {
315             until += 1;
316         }
317     }
318     let mut next_pos = 0;
319     for i in 0..last.len() {
320         if value == last[i] {
321             until -= 1;
322         }
```

```
323         if until == 0 {
324             next_pos = i;
325             break;
326         }
327     }
328     return next_pos;
329 }
330 
331 /// Reverse transformation of the BWT for bytes
332 /// Returns the original Vector of the BWT transformed Vector.
333 /// # Arguments
334 /// * ‘input‘ - The BWT transformed Vector.
335 /// * ‘start‘ - The index of the rotation containing the original Vector.
336 pub fn untransform_byte(input: &Vec<u8>, start: &u64) -> Vec<u8> {
337     let mut sorted = input.clone();
338     sorted.sort();
339     let mut pos = *start as usize;
340     let mut output: Vec<u8> = Vec::new();
341     for _i in 0..input.len() {
342         output.push(sorted[pos]);
343         pos = next_pos_byte(&input, &sorted, &pos);
344     }
345     return output;
346 }
347 
348 /// Calculates the next index for the reverse BWT of bytes
349 /// Returns index of the corresponding character in BWT transformed Vector.
350 /// # Arguments
351 /// * ‘last‘ - The BWT transformed Vector.
352 /// * ‘first‘ - The sorted BWT transformed Vector.
353 /// * ‘lastpos‘ - The current index of the reverse BWT.
354 fn next_pos_byte(last: &Vec<u8>, first: &Vec<u8>, lastpos: &usize) -> usize {
355     let mut until = 0;
356     let value = first[*lastpos];
357     for i in 0..(lastpos + 1) {
358         if value == first[i] {
359             until += 1;
360         }
361     }
362     let mut next_pos = 0;
363     for i in 0..last.len() {
364         if value == last[i] {
365             until -= 1;
366         }
367         if until == 0 {
368             next_pos = i;
369             break;
370         }
371     }
372     return next_pos;
373 }
374 
375 /// Reverse transformation of the BWT for bits
```

```
376 /// Returns the original Vector of the BWT transformed Vector.
377 /// # Arguments
378 /// * ‘input‘ - The BWT transformed Vector.
379 /// * ‘start‘ - The index of the rotation containing the original Vector.
380 pub fn untransform_bit(input: &Vec<u8>, start: &u8) -> Vec<u8> {
381     let mut sorted = input.clone();
382     sorted.sort();
383     let mut pos = *start as usize;
384     let mut output: Vec<u8> = Vec::new();
385     for _i in 0..input.len() {
386         output.push(sorted[pos]);
387         pos = next_pos_bit(&input, &sorted, &pos);
388     }
389     return output;
390 }
391 
392 /// Calculates the next index for the reverse BWT of bits
393 /// Returns index of the corresponding character in BWT transformed Vector.
394 /// # Arguments
395 /// * ‘last‘ - The BWT transformed Vector.
396 /// * ‘first‘ - The sorted BWT transformed Vector.
397 /// * ‘lastpos‘ - The current index of the reverse BWT.
398 fn next_pos_bit(last: &Vec<u8>, first: &Vec<u8>, lastpos: &usize) -> usize {
399     let mut until = 0;
400     let value = first[*lastpos];
401     for i in 0..(lastpos + 1) {
402         if value == first[i] {
403             until += 1;
404         }
405     }
406     let mut next_pos = 0;
407     for i in 0..last.len() {
408         if value == last[i] {
409             until -= 1;
410         }
411         if until == 0 {
412             next_pos = i;
413             break;
414         }
415     }
416     return next_pos;
417 }
418 
419 /// MTF-coding for bits
420 /// Returns the MTF-coding of the Vector of bits.
421 /// # Arguments
422 /// * ‘input‘ - The input Vector of the MTF-coding.
423 pub fn mtf_bit(input: &Vec<u8>) -> Vec<u8> {
424     let mut alpha = input.clone();
425     alpha.sort();
426     alpha.dedup();
427     let mut output: Vec<u8> = Vec::new();
428     for element in input.iter() {
```

```
429         for i in 0..alpha.len() {
430             if alpha[i] == *element {
431                 output.push(i as u8);
432                 let temp = alpha[i];
433                 let mut i = i;
434                 while i >= 1 {
435                     alpha[i] = alpha[i - 1];
436                     i -= 1;
437                 }
438                 alpha[0] = temp;
439                 break;
440             }
441         }
442     }
443     return output;
444 }
445 
446 /// MTF-coding for bytes
447 /// Returns the MTF-coding of the Vector of bytes.
448 /// # Arguments
449 /// * `input` - The input Vector of the MTF-coding.
450 pub fn mtf_byte(input: &Vec<u8>) -> Vec<u8> {
451     let mut alpha = input.clone();
452     alpha.sort();
453     alpha.dedup();
454     let mut output: Vec<u8> = Vec::new();
455     for element in input.iter() {
456         for i in 0..alpha.len() {
457             if alpha[i] == *element {
458                 output.push(i as u8);
459                 let temp = alpha[i];
460                 let mut i = i;
461                 while i >= 1 {
462                     alpha[i] = alpha[i - 1];
463                     i -= 1;
464                 }
465                 alpha[0] = temp;
466                 break;
467             }
468         }
469     }
470     return output;
471 }
472 
473 /// MTF-coding for 64 Bit numbers
474 /// Returns the MTF-coding of the Vector of 64 Bit numbers.
475 /// # Arguments
476 /// * `input` - The input Vector of the MTF-coding.
477 pub fn mtf(input: &Vec<u64>) -> Vec<u64> {
478     let mut alpha = input.clone();
479     alpha.sort();
480     alpha.dedup();
481     let mut output: Vec<u64> = Vec::new();
```

```
482     for element in input.iter() {
483         for i in 0..alpha.len() {
484             if alpha[i] == *element {
485                 output.push(i as u64);
486                 let temp = alpha[i];
487                 let mut i = i;
488                 while i >= 1 {
489                     alpha[i] = alpha[i - 1];
490                     i -= 1;
491                 }
492                 alpha[0] = temp;
493                 break;
494             }
495         }
496     }
497     return output;
498 }
499 
500 /// MTF-1-coding for bytes
501 /// Returns the MTF-1-coding of the Vector of bytes.
502 /// # Arguments
503 /// * ‘input‘ - The input Vector of the MTF-1-coding.
504 pub fn mtf1_byte(input: &Vec<u8>) -> Vec<u8> {
505     let mut alpha = input.clone();
506     alpha.sort();
507     alpha.dedup();
508     let mut output: Vec<u8> = Vec::new();
509     for element in input.iter() {
510         for i in 0..alpha.len() {
511             if alpha[i] == *element {
512                 output.push(i as u8);
513                 if i == 0 {
514                     break;
515                 }
516                 if i == 1 {
517                     let temp = alpha[1];
518                     alpha[1] = alpha[0];
519                     alpha[0] = temp;
520                 } else {
521                     let temp = alpha[i];
522                     let mut i = i;
523                     while i >= 2 {
524                         alpha[i] = alpha[i - 1];
525                         i -= 1;
526                     }
527                     alpha[1] = temp;
528                 }
529                 break;
530             }
531         }
532     }
533     return output;
534 }
```

```
535
536 /// Dynamic Arithmetic coder for bits
537 /// Returns the arithmetic coding of the Vector of bits.
538 /// # Arguments
539 /// * `input` - The input Vector of the arithmetic coding.
540 pub fn arithmetic_bit(input: &Vec<u8>) -> Vec<u8> {
541     let mut histo = HashMap::new();
542     let size = input.len();
543     for i in input.iter() {
544         let count = histo.entry(i).or_insert(0);
545         *count += 1;
546     }
547     let mut histo_tuple: Vec<_> = histo.iter().collect();
548     histo_tuple.sort();
549     let mut left = 0_f64;
550     let mut right = 1_f64;
551     let mut intervals: HashMap<u8, (f64, f64)> = HashMap::new();
552     let mut last = 0_f64;
553     let mut output: Vec<u8> = Vec::new();
554     for &(key, value) in histo_tuple.iter() {
555         intervals.insert(**key, (last, last + ((*value as f64) / (size as f64)))
            ↪ );
556         last += (*value as f64) / (size as f64);
557     }
558     for i in input.iter() {
559         while left >= 0.5_f64 || right < 0.5_f64 {
560             if left >= 0.5_f64 {
561                 output.push(1);
562                 left = (left * 2_f64) - 1_f64;
563                 right = (right * 2_f64) - 1_f64;
564             } else {
565                 output.push(0);
566                 left = left * 2_f64;
567                 right = right * 2_f64;
568             }
569         }
570         let entry = intervals.entry(*i).or_insert((0.0, 0.0));
571         let (lower, upper) = *entry;
572         let dis = right - left;
573         left = left + (dis * lower);
574         right = right - (dis * (1_f64 - upper));
575     }
576     return output;
577 }
578
579 /// Dynamic Arithmetic coder for bytes
580 /// Returns the tuple of the arithmetic coding of the input Vector as a binary
    ↪ Vector and the number of unipue characters.
581 /// # Arguments
582 /// * `input` - The input Vector of the arithmetic coding.
583 pub fn arithmetic_byte(input: &Vec<u8>) -> (Vec<u8>, usize) {
584     let mut histo = HashMap::new();
585     let size = input.len();
```

```
586     for i in input.iter() {
587         let count = histo.entry(i).or_insert(0);
588         *count += 1;
589     }
590     let overhead = histo.len();
591     let mut histo_tuple: Vec<_> = histo.iter().collect();
592     histo_tuple.sort();
593     let mut left = 0_f64;
594     let mut right = 1_f64;
595     let mut intervals: HashMap<u8, (f64, f64)> = HashMap::new();
596     let mut last = 0_f64;
597     let mut output: Vec<u8> = Vec::new();
598     for &(key, value) in histo_tuple.iter() {
599         intervals.insert(**key, (last, last + ((*value as f64) / (size as f64)))
            ↪ );
600         last += (*value as f64) / (size as f64);
601     }
602     for i in input.iter() {
603         while left >= 0.5_f64 || right < 0.5_f64 {
604             if left >= 0.5_f64 {
605                 output.push(1);
606                 left = (left * 2_f64) - 1_f64;
607                 right = (right * 2_f64) - 1_f64;
608             } else {
609                 output.push(0);
610                 left = left * 2_f64;
611                 right = right * 2_f64;
612             }
613         }
614         let entry = intervals.entry(*i).or_insert((0.0, 0.0));
615         let (lower, upper) = *entry;
616         let dis = right - left;
617         left = left + (dis * lower);
618         right = right - (dis * (1_f64 - upper));
619     }
620     return (output, overhead);
621 }
622 
623 /// Dynamic Arithmetic coder for 16 Bits numbers
624 /// Returns the tuple of the arithmetic coding of the input Vector as a binary
    ↪ Vector and the number of unipue characters.
625 /// # Arguments
626 /// * ‘input‘ - The input Vector of the arithmetic coding.
627 pub fn arithmetic_u16(input: &Vec<u16>) -> (Vec<u8>, usize) {
628     let mut histo = HashMap::new();
629     let size = input.len();
630     for i in input.iter() {
631         let count = histo.entry(i).or_insert(0);
632         *count += 1;
633     }
634     let overhead = histo.len();
635     let mut histo_tuple: Vec<_> = histo.iter().collect();
636     histo_tuple.sort();
```

```
637     let mut left = 0_f64;
638     let mut right = 1_f64;
639     let mut intervals: HashMap<u16, (f64, f64)> = HashMap::new();
640     let mut last = 0_f64;
641     let mut output: Vec<u8> = Vec::new();
642     for &(key, value) in histo_tuple.iter() {
643         intervals.insert(**key, (last, last + ((*value as f64) / (size as f64)))
                ↪ );
644         last += (*value as f64) / (size as f64);
645     }
646     for i in input.iter() {
647         while left >= 0.5_f64 || right < 0.5_f64 {
648             if left >= 0.5_f64 {
649                 output.push(1);
650                 left = (left * 2_f64) - 1_f64;
651                 right = (right * 2_f64) - 1_f64;
652             } else {
653                 output.push(0);
654                 left = left * 2_f64;
655                 right = right * 2_f64;
656             }
657         }
658         let entry = intervals.entry(*i).or_insert((0.0, 0.0));
659         let (lower, upper) = *entry;
660         let dis = right - left;
661         left = left + (dis * lower);
662         right = right - (dis * (1_f64 - upper));
663     }
664     return (output, overhead);
665 }
666 
667 /// Dynamic Arithmetic coder for 64 Bits numbers
668 /// Returns the tuple of the arithmetic coding of the input Vector as a binary
        ↪ Vector and the number of unipue characters.
669 /// # Arguments
670 /// * ‘input‘ - The input Vector of the arithmetic coding.
671 pub fn arithmetic_u64(input: &Vec<u64>) -> (Vec<u8>, usize) {
672     let mut histo = HashMap::new();
673     let size = input.len();
674     for i in input.iter() {
675         let count = histo.entry(i).or_insert(0);
676         *count += 1;
677     }
678     let overhead = histo.len();
679     let mut histo_tuple: Vec<_> = histo.iter().collect();
680     histo_tuple.sort();
681     let mut left = 0_f64;
682     let mut right = 1_f64;
683     let mut intervals: HashMap<u64, (f64, f64)> = HashMap::new();
684     let mut last = 0_f64;
685     let mut output: Vec<u8> = Vec::new();
686     for &(key, value) in histo_tuple.iter() {
687         intervals.insert(**key, (last, last + ((*value as f64) / (size as f64)))
```

```
            ↪ );
688         last += (*value as f64) / (size as f64);
689     }
690     for i in input.iter() {
691         while left >= 0.5_f64 || right < 0.5_f64 {
692             if left >= 0.5_f64 {
693                 output.push(1);
694                 left = (left * 2_f64) - 1_f64;
695                 right = (right * 2_f64) - 1_f64;
696             } else {
697                 output.push(0);
698                 left = left * 2_f64;
699                 right = right * 2_f64;
700             }
701         }
702         let entry = intervals.entry(*i).or_insert((0.0, 0.0));
703         let (lower, upper) = *entry;
704         let dis = right - left;
705         left = left + (dis * lower);
706         right = right - (dis * (1_f64 - upper));
707     }
708     return (output, overhead);
709 }
710
711 /// Function that transforms a Vector of bits into a Vector of corresponding
        ↪ bytes
712 /// Returns the Vector of bytes that corresponds to the input Vector.
713 /// # Arguments
714 /// * ‘input‘ - Slice of bits.
715 fn bits_to_u8(input: &[u8]) -> Vec<u8> {
716     let mut vec: Vec<u8> = Vec::new();
717     for i in 0..(input.len() / 8) {
718         let mut temp = 0;
719         for (num, pow) in POWS8.iter().enumerate() {
720             temp += input[(i * 8) + num] as u8 * *pow as u8;
721         }
722         vec.push(temp as u8);
723     }
724     return vec;
725 }
726
727 /// Function that transforms a Vector of bits into a Vector of corresponding 32
        ↪  Bit numbers
728 /// Returns the Vector of 32 Bit numbers that corresponds to the input Vector.
729 /// # Arguments
730 /// * ‘input‘ - Slice of bits.
731 fn bits_to_u32(input: &[u8]) -> Vec<u32> {
732     let mut vec: Vec<u32> = Vec::new();
733     for i in 0..(input.len() / 32) {
734         let mut temp = 0;
735         for (num, pow) in POWS32.iter().enumerate() {
736             temp += input[(i * 32) + num] as u32 * pow;
737         }
```

```
738         vec.push(temp as u32);
739     }
740     return vec;
741 }
742 
743 /// Fixed length run-length-encoding with length ‘block‘
744 /// Returns the Vector, which corresponds to the fixed length run-length-
        ↪ encoding of length ‘block‘.
745 /// # Arguments
746 /// * ‘input‘ - The input Vector of the run-length-encoding.
747 /// * ‘block‘ - The fixed amount of bits used to encode the run-length.
748 pub fn run_length(input: &Vec<u8>, block: usize) -> Vec<u64> {
749     let block_val = (2 as u64).pow(block as u32);
750     let mut output: Vec<u64> = Vec::new();
751     let mut len = 0;
752     let mut current = 0 as u8;
753     for i in input.iter() {
754         if current != *i {
755             if current == 1 {
756                 current = 0;
757             } else {
758                 current = 1;
759             }
760             output.push(len);
761             len = 1;
762         } else {
763             len += 1;
764             if len == block_val {
765                 output.push(len - 1);
766                 output.push(0);
767                 len = 1;
768             }
769         }
770     }
771     output.push(len);
772     return output;
773 }
774 
775 /// Run-length-encoding of the character ‘0‘ with binary encoding of the run-
        ↪ length
776 /// Returns the Vector, which corresponds to the run-length-encoding of the
        ↪ character ‘0‘ with binary encoding of the run-length. All other
        ↪ characters are left untouched.
777 /// # Arguments
778 /// * ‘input‘ - The input Vector of the run-length-encoding.
779 pub fn mtf_run_length(input: &Vec<u8>) -> Vec<u16> {
780     let mut output: Vec<u16> = Vec::new();
781     let mut len: usize = 0;
782     for i in input.iter() {
783         if *i == 0 {
784             len += 1;
785         } else {
786             if len > 0 {
```

```
787                 while len != 0 {
788                     if (len % 2) == 0 {
789                         output.push(256);
790                     } else {
791                         output.push(0);
792                         len += 1;
793                     }
794                     len = len / 2;
795                     len -= 1;
796                 }
797                 len = 0;
798             }
799             output.push(*i as u16);
800         }
801     }
802     if len > 0 {
803         while len != 0 {
804             if (len % 2) == 0 {
805                 output.push(256);
806             } else {
807                 output.push(0);
808                 len += 1;
809             }
810             len = len / 2;
811             len -= 1;
812         }
813         len = 0;
814     }
815     return output;
816 }
817 
818 /// Run-length-encoding for Vectors of bits using binary encoding of the run-
    ↪ length
819 /// Returns the Vector, which corresponds to the run-length-encoding of the
    ↪ input Vector.
820 /// This function can only handle binary values. Therefor the only valid values
    ↪  of the Vector are ‘0‘ and ‘1‘.
821 /// # Arguments
822 /// * ‘input‘ - The input Vector of the run-length-encoding.
823 pub fn bin_run_length(input: &Vec<u8>) -> Vec<u8> {
824     let mut output: Vec<u8> = Vec::new();
825     let mut len: usize = 0;
826     let mut current = input[0];
827     for i in input.iter() {
828         if *i == current {
829             len += 1;
830         } else {
831             if current == 0 {
832                 while len != 0 {
833                     if (len % 2) == 0 {
834                         output.push(255);
835                     } else {
836                         output.push(0);
```

```
837                     len += 1;
838                 }
839                 len = len / 2;
840                 len -= 1;
841             }
842             len = 1;
843             current = *i;
844         } else {
845             while len != 0 {
846                 if (len % 2) == 0 {
847                     output.push(254);
848                 } else {
849                     output.push(1);
850                     len += 1;
851                 }
852                 len = len / 2;
853                 len -= 1;
854             }
855             len = 1;
856             current = *i;
857         }
858     }
859 }
860 if current == 0 {
861     while len != 0 {
862         if (len % 2) == 0 {
863             output.push(255);
864         } else {
865             output.push(0);
866             len += 1;
867         }
868         len = len / 2;
869         len -= 1;
870     }
871 } else {
872     while len != 0 {
873         if (len % 2) == 0 {
874             output.push(254);
875         } else {
876             output.push(1);
877             len += 1;
878         }
879         len = len / 2;
880         len -= 1;
881     }
882 }
883 return output;
884 }
885
886 /// Fixed length run-length-encoding of the value ‘1‘ and run-length-encoding
    ↪ of the value ‘0‘ with binary encoding of the run-length
887 /// Returns the Vector, which corresponds to the run-length-encoding of the
    ↪ input Vector.
```

```
888 /// This function can only handle binary values. Therefor the only valid values
        ↪  of the Vector are ‘0‘ and ‘1‘. The run-lengths of the ones in the
        ↪ Vector are encoded with a fixed length of bits. In this case we use 8
        ↪ Bits. The run-lengths of the zeros in the Vector are encoded by using
        ↪ binary encoding of the length.
889 /// # Arguments
890 /// * ‘input‘ - The input Vector of the run-length-encoding.
891 pub fn bin_mtf_run_length(input: &Vec<u8>) -> Vec<u8> {
892     let mut output: Vec<u8> = Vec::new();
893     let mut len: usize = 0;
894     let mut current = input[0];
895     for i in input.iter() {
896         if *i == current {
897             if current == 1 && len == 254 {
898                 output.push(254);
899                 len = 0;
900             }
901             len += 1;
902         } else {
903             if current == 0 {
904                 while len != 0 {
905                     if (len % 2) == 0 {
906                         output.push(255);
907                     } else {
908                         output.push(0);
909                         len += 1;
910                     }
911                     len = len / 2;
912                     len -= 1;
913                 }
914                 len = 1;
915                 current = *i;
916             } else {
917                 output.push(len as u8);
918                 len = 1;
919                 current = *i;
920             }
921         }
922     }
923     if current == 0 {
924         while len != 0 {
925             if (len % 2) == 0 {
926                 output.push(255);
927             } else {
928                 output.push(0);
929                 len += 1;
930             }
931             len = len / 2;
932             len -= 1;
933         }
934     } else {
935         output.push(len as u8);
936     }
```

```
937     return output;
938 }
939 
940 /// Fixed length run-length-encoding that enly encodes run-lengths greater
        ↪ equal 4
941 /// Returns the Vector, which corresponds to the run-length-encoding of the
        ↪ input Vector.
942 /// This function is usually used as the first step of the transformation. All
        ↪ run-lenghts smaller than 4 are left untouched. Run-lenghts greater equal
        ↪  4 are encoded by the first 4 values left untouched and then followed by
        ↪  the remaining run-length. The run-length is encoded by a fixed length
        ↪ number. We use 8 Bits to encode the length.
943 /// # Example
944 /// ```
945 /// let untouched = vec![5,5,5];
946 /// let edge_case = vec![5,5,5,5];
947 /// let encoded = vec![5,5,5,5,5];
948 /// assert_eq!(vec![5,5,5], run_length_preprocessing(untouched));
949 /// assert_eq!(vec![5,5,5,5,0], run_length_preprocessing(edge_case));
950 /// assert_eq!(vec![5,5,5,5,1], run_length_preprocessing(encoded));
951 /// ```
952 /// # Arguments
953 /// * `input` - The input Vector of the run-length-encoding.
954 pub fn run_length_preprocessing(input: &Vec<u8>) -> Vec<u8> {
955     let mut out: Vec<u8> = Vec::new();
956     let mut cnt = 0;
957     let mut is_run_length = false;
958     let mut current = input[0];
959     for i in input.iter() {
960         if is_run_length {
961                 if *i == current {
962                     cnt += 1;
963                 }
964                 if (*i != current) || (cnt == 255) {
965                     out.push(cnt);
966                     out.push(*i);
967                     cnt = 1;
968                     current = *i;
969                     is_run_length = false;
970                 }
971         } else {
972             if current == *i {
973                 cnt += 1;
974             } else {
975                 cnt = 1;
976                 current = *i;
977             }
978             out.push(*i);
979             if cnt == 4 {
980                 cnt = 0;
981                 is_run_length = true;
982             }
983         }
```

```
984     }
985     if is_run_length {
986         out.push(cnt);
987     }
988     return out;
989 }
990 
991 /// Function that generates the arithmetic coding of one character
992 /// Returns binary encoding of the position in the array that corresponds to
        ↪ the character.
993 /// The encoding stays the same if the bounds are the same. Thereby the
        ↪ encoding of every character is fixed. This function is used as a part of
        ↪  a arithmetic coding algorithm to determine the encoding.
994 /// # Arguments
995 /// * `lower` - This is the lower bound of the interval corresponding to the
        ↪ character in the arithmetic coding.
996 /// * `upper` - This is the upper bound of the interval corresponding to the
        ↪ character in the arithmetic coding.
997 fn get_arith_coding(lower: f64, upper: f64) -> Vec<u8> {
998     let mut pos = 0_f64;
999     let mut output: Vec<u8> = Vec::new();
1000    let mut cnt = 1_f64;
1001    while true {
1002        let dis = (1_f64 / cnt.exp2());
1003        if !(lower > pos || upper < (pos + (2.0 * dis))) {
1004            break;
1005        }
1006        if upper < (pos + dis) {
1007            output.push(0);
1008        } else {
1009            if (((pos + dis) - lower) > (upper - (pos + dis))) {
1010                output.push(0)
1011            } else {
1012                output.push(1);
1013                pos = (pos + dis);
1014            }
1015        }
1016        cnt += 1_f64;
1017    }
1018    return output;
1019 }
1020 
1021 /// Arithmetic coding algorithm with fixed coding for every character for bytes
1022 /// Returns the tuple of the arithmetic coding of the input Vector as a binary
        ↪ Vector and the number of unipue characters.
1023 /// This function calculates the intervals for every character and then calls
        ↪ the function to generate the aritmetic coding of every character and
        ↪ then appends the encodings respectively.
1024 /// # Arguments
1025 /// * `input` - The input Vector for the arithmetic coding.
1026 pub fn fixed_arithmetic_byte(input: &Vec<u8>) -> (Vec<u8>, usize) {
1027     let mut histo = HashMap::new();
1028     let size = input.len();
```

```
1029     for i in input.iter() {
1030         let count = histo.entry(i).or_insert(0);
1031         *count += 1;
1032     }
1033     let overhead = histo.len();
1034     let mut histo_tuple: Vec<_> = histo.iter().collect();
1035     histo_tuple.sort();
1036     let mut intervals: HashMap<u8, (f64, f64)> = HashMap::new();
1037     let mut last = 0_f64;
1038     let mut output: Vec<u8> = Vec::new();
1039     for &(key, value) in histo_tuple.iter() {
1040         intervals.insert(**key, (last, last + ((*value as f64) / (size as f64)))
             ↪ );
1041         last += (*value as f64) / (size as f64);
1042     }
1043     for i in input.iter() {
1044         let entry = intervals.entry(*i).or_insert((0.0, 0.0));
1045         let (lower, upper) = *entry;
1046         output.append(&mut get_arith_coding(lower, upper));
1047     }
1048     return (output, overhead);
1049 }
1050 
1051 /// Arithmetic coding algorithm with fixed coding for every character for 16
     ↪ Bit numbers
1052 /// Returns the tuple of the arithmetic coding of the input Vector as a binary
     ↪ Vector and the number of unipue characters.
1053 /// This function calculates the intervals for every character and then calls
     ↪ the function to generate the aritmetic coding of every character and
     ↪ then appends the encodings respectively.
1054 /// # Arguments
1055 /// * ‘input‘ - The input Vector for the arithmetic coding.
1056 pub fn fixed_arithmetic_u16(input: &Vec<u16>) -> (Vec<u8>, usize) {
1057     let mut histo = HashMap::new();
1058     let size = input.len();
1059     for i in input.iter() {
1060         let count = histo.entry(i).or_insert(0);
1061         *count += 1;
1062     }
1063     let overhead = histo.len();
1064     let mut histo_tuple: Vec<_> = histo.iter().collect();
1065     histo_tuple.sort();
1066     let mut intervals: HashMap<u16, (f64, f64)> = HashMap::new();
1067     let mut last = 0_f64;
1068     let mut output: Vec<u8> = Vec::new();
1069     for &(key, value) in histo_tuple.iter() {
1070         intervals.insert(**key, (last, last + ((*value as f64) / (size as f64)))
             ↪ );
1071         last += (*value as f64) / (size as f64);
1072     }
1073     for i in input.iter() {
1074         let entry = intervals.entry(*i).or_insert((0.0, 0.0));
1075         let (lower, upper) = *entry;
```

```
1076        output.append(&mut get_arith_coding(lower, upper));
1077    }
1078    return (output, overhead);
1079 }
1080
1081 /// Arithmetic coding algorithm with fixed coding for every character for 64
        ↪ Bit numbers
1082 /// Returns the tuple of the arithmetic coding of the input Vector as a binary
        ↪ Vector and the number of unipue characters.
1083 /// This function calculates the intervals for every character and then calls
        ↪ the function to generate the aritmetic coding of every character and
        ↪ then appends the encodings respectively.
1084 /// # Arguments
1085 /// * `input` - The input Vector for the arithmetic coding.
1086 pub fn fixed_arithmetic_u64(input: &Vec<u64>) -> (Vec<u8>, usize) {
1087    let mut histo = HashMap::new();
1088    let size = input.len();
1089    for i in input.iter() {
1090        let count = histo.entry(i).or_insert(0);
1091        *count += 1;
1092    }
1093    let overhead = histo.len();
1094    let mut histo_tuple: Vec<_> = histo.iter().collect();
1095    histo_tuple.sort();
1096    let mut intervals: HashMap<u64, (f64, f64)> = HashMap::new();
1097    let mut last = 0_f64;
1098    let mut output: Vec<u8> = Vec::new();
1099    for &(key, value) in histo_tuple.iter() {
1100        intervals.insert(**key, (last, last + ((*value as f64) / (size as f64)))
            ↪ );
1101        last += (*value as f64) / (size as f64);
1102    }
1103    for i in input.iter() {
1104        let entry = intervals.entry(*i).or_insert((0.0, 0.0));
1105        let (lower, upper) = *entry;
1106        output.append(&mut get_arith_coding(lower, upper));
1107    }
1108    return (output, overhead);
1109 }
```

A.3. Modul für primitive Verarbeitung der Daten

Quellcode 3 beinhaltet Funktionen, welche die Entropie der Folge ermitteln. Weitere Funktionen dienen zum umstrukturieren der Folge. Das bedeutet, dass die Folge nicht wirklich transformiert wird, sondern die Information anderes gruppiert werden, also sich die Anzahl der Zeichens verändert. So werden zum Beispiel acht Zeichen mit der Mächtigkeit des Alphabetes zwei zu einem Zeichen zusammengefasst.

Quellcode 3: Funktionen zur Evaluation der Folgen und Umformung der Folgen ohne wirkliche Transformation

```
1 extern crate num_traits;
2
```

```
3  use std::str;
4  use self::num_traits::PrimInt;
5  use std::collections::HashMap;
6
7  const POWS4: [u64; 4] = [8, 4, 2, 1];
8  const POWS8: [u8; 8] = [128, 64, 32, 16, 8, 4, 2, 1];
9  const POWS16: [u64; 16] = [32768, 16384, 8192, 4096, 2048, 1024, 512, 256, 128,
       ↪  64, 32, 16, 8, 4, 2, 1];
10 const POWS32: [u64; 32] = [2147483648, 1073741824, 536870912, 268435456,
       ↪ 134217728, 67108864, 33554432, 16777216, 8388608, 4194304, 2097152,
       ↪ 1048576, 524288, 262144, 131072, 65536, 32768, 16384, 8192, 4096, 2048,
       ↪ 1024, 512, 256, 128, 64, 32, 16, 8, 4, 2, 1];
11 const POWS64: [u64; 64] = [9223372036854775808, 4611686018427387904,
       ↪ 2305843009213693952, 1152921504606846976, 576460752303423488,
       ↪ 288230376151711744, 144115188075855872, 72057594037927936,
       ↪ 36028797018963968, 18014398509481984, 9007199254740992,
       ↪ 4503599627370496, 2251799813685248, 1125899906842624, 562949953421312,
       ↪ 281474976710656, 140737488355328, 70368744177664, 35184372088832,
       ↪ 17592186044416, 8796093022208, 4398046511104, 2199023255552,
       ↪ 1099511627776, 549755813888, 274877906944, 137438953472, 68719476736,
       ↪ 34359738368, 17179869184, 8589934592, 4294967296, 2147483648,
       ↪ 1073741824, 536870912, 268435456, 134217728, 67108864, 33554432,
       ↪ 16777216, 8388608, 4194304, 2097152, 1048576, 524288, 262144, 131072,
       ↪ 65536, 32768, 16384, 8192, 4096, 2048, 1024, 512, 256, 128, 64, 32, 16,
       ↪ 8, 4, 2, 1];
12
13 /// Function that turn the input String into the Vector of Bytes
14 /// Returns the bytes of the String as a Vector.
15 /// # Arguments
16 /// * `input` - String of the input data.
17 pub fn from_string(input: String) -> Vec<u8> {
18     let bytes = input.as_bytes();
19     let mut vec: Vec<u8> = Vec::new();
20     for b in bytes {
21         vec.push(*b);
22     }
23     return vec;
24 }
25
26 /// Function that turn a Vector of Bytes into the corresponding String
27 /// Returns the bytes of the Vector as a String.
28 /// # Arguments
29 /// * `input` - Vector of bytes.
30 pub fn to_string(input: &Vec<u8>) -> String {
31     str::from_utf8(&input).unwrap().to_string()
32 }
33
34 /// Function that calculates the amount of information or entropy of a Vector
35 /// Returns amount of information or entropy as a integer number.
36 /// # Arguments
37 /// * `input` - Vector of 64 Bit numbers.
38 pub fn evaluate_information(input: &Vec<u64>) -> usize {
39     let mut histo = HashMap::new();
```

```
40      let size = input.len();
41      for i in input.iter() {
42          let count = histo.entry(i).or_insert(0);
43          *count += 1;
44      }
45      let histo_tuple: Vec<_> = histo.iter().collect();
46      let unique = histo_tuple.len();
47      let mut information = 0_f64;
48      for &(_key, value) in histo_tuple.iter() {
49          information += (((size as f64) / (*value as f64)).log(unique as f64)) *
                ↪ (*value as f64);
50      }
51      return information.ceil() as usize;
52  }
53
54  /// Function that calculates the amount of information or entropy of a Vector
55  /// Returns amount of information or entropy as a integer number.
56  /// # Arguments
57  /// * ‘input‘ - Vector of bytes.
58  pub fn evaluate_informationu8(input: &Vec<u8>) -> usize {
59      let mut histo = HashMap::new();
60      let size = input.len();
61      for i in input.iter() {
62          let count = histo.entry(i).or_insert(0);
63          *count += 1;
64      }
65      let histo_tuple: Vec<_> = histo.iter().collect();
66      let unique = histo_tuple.len();
67      let mut information = 0_f64;
68      for &(_key, value) in histo_tuple.iter() {
69          information += (((size as f64) / (*value as f64)).log(2.0_f64)) * (*
                ↪ value as f64);
70      }
71      return information.ceil() as usize;
72  }
73
74  /// Function that calculates the amount of information or entropy of a Vector
75  /// Returns amount of information or entropy as a integer number.
76  /// # Arguments
77  /// * ‘input‘ - Vector of 16 Bit numbers.
78  pub fn evaluate_informationu16(input: &Vec<u16>) -> usize {
79      let mut histo = HashMap::new();
80      let size = input.len();
81      for i in input.iter() {
82          let count = histo.entry(i).or_insert(0);
83          *count += 1;
84      }
85      let histo_tuple: Vec<_> = histo.iter().collect();
86      let unique = histo_tuple.len();
87      let mut information = 0_f64;
88      for &(_key, value) in histo_tuple.iter() {
89          information += (((size as f64) / (*value as f64)).log(2.0_f64)) * (*
                ↪ value as f64);
```

```
90      }
91      return information.ceil() as usize;
92  }
93
94  /// Function that transforms a Vector of bytes into a Vector of corresponding
        ↪ bits
95  /// Returns the Vector of bits that corresponds to the input Vector.
96  /// # Arguments
97  /// * ‘input‘ - Vector of bytes.
98  pub fn bytes_to_bits(input: &Vec<u8>) -> Vec<u8> {
99      let mut vec: Vec<u8> = Vec::new();
100     for i in 0..input.len() {
101         let mut temp = input[i];
102         for pow in POWS8.iter() {
103             vec.push((temp / pow) as u8);
104             temp = temp % pow;
105         }
106     }
107     return vec;
108 }
109
110 /// Function that transforms a Vector of bits into a Vector of corresponding 4
        ↪ Bit numbers
111 /// Returns the Vector of 4 Bit numbers that corresponds to the input Vector.
112 /// # Arguments
113 /// * ‘input‘ - Vector of bits.
114 pub fn bits_to_4(input: &Vec<u8>) -> Vec<u64> {
115     let mut vec: Vec<u64> = Vec::new();
116     for i in 0..(input.len() / 4) {
117         let mut temp = 0;
118         for (num, pow) in POWS4.iter().enumerate() {
119             temp += input[(i * 4) + num] as u64 * pow;
120         }
121         vec.push(temp as u64);
122     }
123     return vec;
124 }
125
126 /// Function that transforms a Vector of bits into a Vector of corresponding
        ↪ bytes
127 /// Returns the Vector of bytes that corresponds to the input Vector.
128 /// # Arguments
129 /// * ‘input‘ - Vector of bits.
130 pub fn bits_to_bytes(input: &Vec<u8>) -> Vec<u8> {
131     let mut vec: Vec<u8> = Vec::new();
132     for i in 0..(input.len() / 8) {
133         let mut temp = 0;
134         for (num, pow) in POWS8.iter().enumerate() {
135             temp += input[(i * 8) + num] as u8 * pow;
136         }
137         vec.push(temp as u8);
138     }
139     return vec;
```

```
140 }
141
142 /// Function that transforms a Vector of bits into a Vector of corresponding 16
    ↪  Bit numbers
143 /// Returns the Vector of 16 Bit numbers that corresponds to the input Vector.
144 /// # Arguments
145 /// * ‘input‘ - Vector of bits.
146 pub fn bits_to_16(input: &Vec<u8>) -> Vec<u64> {
147     let mut vec: Vec<u64> = Vec::new();
148     for i in 0..(input.len() / 16) {
149         let mut temp = 0;
150         for (num, pow) in POWS16.iter().enumerate() {
151             temp += input[(i * 16) + num] as u64 * pow;
152         }
153         vec.push(temp as u64);
154     }
155     return vec;
156 }
157
158 /// Function that transforms a Vector of bits into a Vector of corresponding 32
    ↪  Bit numbers
159 /// Returns the Vector of 32 Bit numbers that corresponds to the input Vector.
160 /// # Arguments
161 /// * ‘input‘ - Vector of bits.
162 pub fn bits_to_32(input: &Vec<u8>) -> Vec<u64> {
163     let mut vec: Vec<u64> = Vec::new();
164     for i in 0..(input.len() / 32) {
165         let mut temp = 0;
166         for (num, pow) in POWS32.iter().enumerate() {
167             temp += input[(i * 32) + num] as u64 * pow;
168         }
169         vec.push(temp as u64);
170     }
171     return vec;
172 }
173
174 /// Function that transforms a Vector of bits into a Vector of corresponding 64
    ↪  Bit numbers
175 /// Returns the Vector of 64 Bit numbers that corresponds to the input Vector.
176 /// # Arguments
177 /// * ‘input‘ - Vector of bits.
178 pub fn bits_to_64(input: &Vec<u8>) -> Vec<u64> {
179     let mut vec: Vec<u64> = Vec::new();
180     for i in 0..(input.len() / 64) {
181         let mut temp = 0;
182         for (num, pow) in POWS64.iter().enumerate() {
183             temp += input[(i * 64) + num] as u64 * pow;
184         }
185         vec.push(temp as u64);
186     }
187     return vec;
188 }
```

A.4. Bibliotheksdatei der Module

Zur Vollständigkeit ist im Folgenden noch der Quellcode 4 für die Deklaration der Module angegeben.

Quellcode 4: Deklaration der Module

```
1 pub mod trans;
2
3 pub mod process;
```

TECHNISCHE UNIVERSITÄT CHEMNITZ

Studentenservice – Zentrales Prüfungsamt

Selbstständigkeitserklärung

Name: Oertel Vorname: Andy geb. am: 31.10.1996 Matr.-Nr.: 406181	**Bitte beachten:** 1. Bitte binden Sie dieses Blatt am Ende Ihrer Arbeit ein.

Selbstständigkeitserklärung*

Ich erkläre gegenüber der Technischen Universität Chemnitz, dass ich die vorliegende Bachelorarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe.

Die vorliegende Arbeit ist frei von Plagiaten. Alle Ausführungen, die wörtlich oder inhaltlich aus anderen Schriften entnommen sind, habe ich als solche kenntlich gemacht.

Diese Arbeit wurde in gleicher oder ähnlicher Form noch nicht als Prüfungsleistung eingereicht und ist auch noch nicht veröffentlicht.

Datum: 27.06.2018 Unterschrift: .....

* Statement of Authorship

I hereby certify to the Technische Universität Chemnitz that this thesis is all my own work and uses no external material other than that acknowledged in the text.

This work contains no plagiarism and all sentences or passages directly quoted from other people's work or including content derived from such work have been specifically credited to the authors and sources.

This paper has neither been submitted in the same or a similar form to any other examiner nor for the award of any other degree, nor has it previously been published.