Referat für Algorithmische Anwendungen WS 2006/ 07: Verlustfreie Datenkompression mit dem Deflate-Algorithmus (LZ77- und Huffman-Codierung)

Benedikt Arnold, 11041025, ai686@gm.fh-koeln.de
Sebastian Bieker, 11038605, sebastian.bieker@gmx.de
Benedikt Malecki, 11041041, ai713@gm.fh-koeln.de

Team D_blau_Ala0607

23. Januar 2007

Inhaltsverzeichnis

1 Einleitung

Die weite Verbreitung von Rechnersystemen in allen Breichen des gesellschaftlichen und wirtschaftlichen Alltags bringt enorme Datenmengen mit sich. Diese Datenmengen müssen transferiert, bearbeitet und permanent gespeichert werden. Die Art und Beschaffenheit der Daten ist dabei sehr unterschiedlich. So werden gleichermaßen einfache ASCII-Dateien wie auch komplexe serialisierte Objekte benutzt. Allen Daten ist jedoch gemeinsam, dass sie in Bytes auf Datenspeichern gelagert werden. Um große Datenmengen möglichst Resourcen schonend behandeln zu können, ist es sinnvoll, Daten zu komprimieren. Dabei werden verschiedene Ansprüche an eine erfolgreiche Datenkompression gestellt. Die Kompression sollte möglichst hoch sein, d.h. die komprimierte Datei nimmt wenig physikalischen Speicherplatz in Anspruch. Desweiteren sollte keine entscheidende Information verloren gehen. Auch die Geschwindigkeit der Kom- und Dekompression können für die Wahl einer bestimmten Kom- oder Dekompressionsmethode entscheidend sein.

Da es eine Vielzahl verschiedener Daten gibt, gibt es auch eine Vielzahl an Kompressionsalgorithmen. Diese werden zuerst grob in verlustbehaftete und verlustfreie Algorithmen eingeteilt. Verlustfreie Verfahren kommen bei Textdateien aller Art zum Einsatz, wohingegen verlustbehaftete Verfahren meistens für audiovisuelle Daten verwendet werden. Wird ein Text verlustbehaftet gespeichert, gehen eventuell wichtige Schlüsselinformationen verloren. Wird hingegen ein Urlaubsfoto verlustbehaftet gespeichert, gehen bei einer moderaten Kompressionsrate nur geringfügig Farben verloren, aber das Motiv ist weiterhin voll erkennbar.

Ziel unseres Projekts soll die Entwicklung eines tieferen Verständnisses von verlustfreier Kompression sein. Als Algorithmus für Datenkompression haben wir den Deflate-Algorithmus, eine Kombination zweier verschiedener Algorithmen, gewählt, den wir im Folgenden untersuchen wollen.

2 Überblick über den Deflate-Algorithmus

Der Deflate-Algorithmus ist ein Daten-Kompressions-Algorithmus, der ein von Betriebssystemen und Computerarchitekturen, sowie Dateisystemen und Zeichensatz unabhängiges komprimiertes Datenformat zur Verfügung stellt.

Der Algorithmus selbst besteht aus zwei separaten Algorithmen, zum einen dem LZ77- und zum anderen dem Huffman-Algorithmus. Die Aufgabe der beiden Algorithmen ist jeweils die Kompression von Daten, wobei die Kombination der beiden Algorithmen die Effektivität der Kompression bei größer werdenden Datenmengen verbessert.

Das erste Mal tauchte der Deflate-Algorithmus 1989 auf. Entwickelt wurde er von Phil Katz, der ihn in seinem Packprogramm PKZip (Phil Katz´ Zip Program) implementierte. Wenig später machte Katz seinen Algorithmus öffentlich. Viele Firmen und Programmierer verwendeten daraufhin den Deflate-Algorithmus in ihren Projekten. So ist der Delfate-Algorithmus mittlerweile in vielen Datei-Formaten wiederzufinden. Dazu zählen unter anderem zip, gzip, png, tiff, pdf, etc..

Da der Deflate-Algorithmus keinem Patent unterliegt, ist er immer noch allgemein für Software-Entwickler interessant, die eine Kompressions- und Dekompressionsmethode mit hoher Effektivität suchen.

Der Deflate-Algorithmus ist im fünfzehnseitigen RFC1951 spezifiziert. Die Methode des Deflate-Algorithmus ermöglicht es eine Sequenz von Bytes durch eine kürzere Sequenz von Bits zu repräsentieren. Des weiteren existiert eine Methode, die erstellte Bit-Sequenz wiederum in eine Sequenz von Bytes zu entpacken.

3 Der LZ77-Algorithmus

3.1 Beschreibung des LZ77-Algorithmus

Der LZ77-Algorithmus wurde 1977 von Abraham Lempel und Jacob Ziv in dem Paper “A Universal Algorithm for Sequential Data Compression“ veröffentlicht. Der Name des Algorithmus entstammt dem Jahr der Erscheinung sowie den Anfangsbuchstaben seiner Erfinder. Diese benutzten eine Methode zur Komprimierung bei der sich wiederholende Zeichenketten allgemein zusammengefasst werden.
Das verwendete Verfahren wird als “sliding window“ bezeichnet, da immer nur ein Teil des Eingabestroms in einem Ausschnitt bearbeitet werden kann. Das sliding window wird in seiner Länge durch Parameter bestimmt, so dass diese variieren kann. Mit dem Einlesen eines Datenstroms wird das sliding window immer weiter verschoben. Es befindet sich immer an der aktuell einzulesenden Stelle. Das sliding window wird in die zwei Puffer, Vorschau- und Suchpuffer unterteilt. Der Suchpuffer entspricht einem Wörterbuch, in welchem Zeichenketten gespeichert werden, die bereits im Eingabestrom vorgekommen sind. Die Größe des Vorschau-Puffers ist überlicherweise 100 Zeichen und damit 100 Bytes. Der Such-Puffer wird in der Regel 100 mal so groß gewählt. Die Größe der Puffer, besonders des Such-Puffers beeinflußt stark die tatsächliche Laufzeit des gesamten Algorithmus. Die starke Bedeutung für die tatsächliche Laufzeit liegt in der Tatsache begründet, dass bei jedem neu eingelesenem Zeichen, das gesamte Wörterbuch, sprich der Such-Puffer durchsucht werden muss.
Der Ausgabestrom des Algorithmus enthält in seinen Grundelementen 3er-Tupel, bestehend aus einem Offset, einer Länge und einem Symbol (Offset,Länge,Symbol). Jedes Element eines Tupels belegt ein Byte, weshalb das ganze Tupel drei Byte im Speicher belegt. Die Art und Weise wie ein zu komprimierendes Zeichen in einem Tupel gespeichert wird hängt davon ab, ob das Zeichen bereits im Wörterbuch vorhanden ist, oder nicht. Ist ein Zeichen noch nicht im Wörterbuch gespeichert, so wird ein Tupel (0,0,Zeichen) gebildet. D.h., dass der Offset vom Beginn des Wörterbuchs an null und die Länge des Zeichens ebenfalls null ist. In dem Fall, in dem das zu kompremierende Zeichen oder die Zeichenkette bereits im Wörterbuch vorhanden ist wird das Tupel anders gebildet. Es wird der Offset vom Beginn des Wörterbuchs an eingetragen, sowie die Länge, die das Zeichen oder die Zeichenkette im Wörterbuch einnimmt. Anschließend wird noch das Nachfolgezeichen der Zeichenkette oder das Zeichen selbst gespeichert.
Damit bestimmt die Häufigkeit wiederholt aufretender Zeichenketten die Anzahl der Tupel im Ausgabestrom und damit die Effizienz der Kompremierung.

Der LZ77-Algorithmus kann jedoch auch den Speicherplatzbedarf eingegebener Zeichen erhöhen. Dies geschieht, wenn eine große Anzahl an unterschiedlichen einzelnen Zeichen, die kleiner als 3 Byte sind, eingegeben wird. Da der Algorithmus jedes Zeichen in einem Tupel von 3 Byte speichert, entsteht so ein Kodierverlust. Sind die eingelesenen Zeichenketten 3 Byte groß, genauso wie ein Tupel, so findet weder eine Komprimierung noch ein Komprimierverlust statt. Sind die eingelesenen Zeichenketten jedoch größer als ein Tupel, so liegt ein Kodiergewinn vor.

3.2 Ein Anwendungsbeispiel zum LZ77-Algorithmus

3.2.1 Kodierung des Wortes ANANAS

Im folgendem Beispiel wird das Wort Ananas mit Hilfe des LZ77-Algorithmus kompremiert. Die Größe des Wörterbuchs beträgt acht, die des Vorschaufensters drei Byte. Wird der String “ANANAS“ eingelesen so erscheinen zunächst nur die Zeichen “ANA“ im Vorschaufenster, da dieses nur drei Byte groß ist. Das Wörterbuch ist zunächst leer. Als erstes Zeichen wird nun das “A“ in das Wörterbuch eingetragen. Das dafür erzeugte Codewort oder Tupel erhält für den Offset wie auch für die Länge null. Das Symbol ist “A“. Das nächste Zeichen ist “N“. Dieses ist noch nicht im Wörterbuch enthalten. Der Offset und die Länge des daraus resultierenden Tupels werden wieder auf null gesetzt. Als Symbol wird das Zeichen “N“ eingetragen. Im Wörterbuch stehen nun die zwei Zeichenketten “A“ und “AN“. Im Vorschaufenster steht die Zeichenkette “ANA“. Aus dieser ist die Zeichenkette “AN“ bereits bekannt, da sie im Wörterbuch vorhanden ist. Als Offset wird nun sechs und als Länge zwei für das nun zu erstellende Tupel eingetragen. Offset und Länge identifizieren die bereits vorhandene Zeichenkette im Wörterbuch. An Position sechs beginnt die Zeichenkette im Wörterbuch und ist zwei Stellen lang. Die auf diese Art und Weise kompremierte Zeichenkette findet sich in einem einzelnen Tupel wieder. Dieses lautet nun (6,2,A). Das Symbol “A“ ist das Folgesymbol der kompremierten Zeichenkette. Anschließend ist nur noch das Zeichen “S“ im Vorschaufenster vorhanden. Da dieses Zeichen nicht im Wörterbuch vorhanden ist, wird ein neues Tupel mit einem Offset und einer Länge von Null mit dem Zeichen “S“ erzeugt.
Da das Vorschaufenster darauf hin leer ist, terminiert der Algorithmus. Das Wort “ANANAS“ ist nun in den Tupeln (0,0,A), (0,0,N), (6,2,A) und (0,0,S) kodiert.

Wörterbuch								**Vorschau**			**Codewort**
0	1	2	3	4	5	6	7				
								A	N	A	(0,0,A)
							A	N	A	N	(0,0,N)
						A	N	A	N	A	(6,2,A)
			A	N	A	N	A	S			(0,0,S)
		A	N	A	N	A	S				fertig, da Vorschau leer

Tabelle 1: Die Kodierung des Wortes ANANAS geschieht mit Hilfe des sliding window, welches in Wörterbuch und Vorschaufenster unterteilt ist.

3.2.2 Decodierung des Wortes ANANAS

Die für das Wort “ANANAS“ vorliegenden Tupel (0,0,A), (0,0,N), (6,2,A) und (0,0,S) werden im Folgenden dekodiert. Als erstes Zeichen wird A an eine leere Zeichenkette angehängt, da Offset und Länge des ersten Tupels 0 sind. Auf die selbe Art und Weise wird mit dem nächsten Tupel, bzw. Zeichen verfahren. Es wird also ein N an die vorhandene Zeichenkette angehängt. Die daraus entstehende Zeichenkette lautet nun “AN“. Das anschließende Tupel (6,2,A) wird folgerndermaßen dekodiert. Aus der erzeugten Zeichenkette des Ausgabestroms wird nun das Zeichen an Position 6 bestimmt und von da an die Zeichenkette der Länge 2 ausgelesen. Diese Zeichenkette wird wiederum an die Zeichenkette des Ausgabestroms angehangen, so dass diese Zeichenkette nun “ANAN“ lautet. An die Zeichenkette des Ausgabestroms wird nun noch das Zeichen A aus dem Tupel angehängt. Die Zeichenkette des Ausgabestroms lautet nun “ANANA“. Anschließend wird nur das Zeichen S aus dem letzten Tupel an die Zeichenkette angehängt, das Offset und Länge wieder 0 sind. Somit ist die Zeichenkette “ANANAS“ wieder rekonstruiert und die Decodierung abgeschlossen.

	0	1	2	3	4	5	6	7	
(0,0,A)								A	Da Offset und Länge Null sind wird Zeichen A an die Zeichenkette angehängt.
(0,0,N)							A	N	siehe bei Zeichen A
(6,2,A)				A	N	A	N	A	Aus der bereits vorhandenen Zeichenkette wird ab der 6. Stelle eine 2 Zeichen lange Kette genommen. Dies ist AN. Als Nachfolgezeichen wird A angehängt.
(0,0,S)			A	N	A	N	A	S	siehe bei Zeichen A

Tabelle 2: Die Decodierung erfolgt mit Hilfe der Werte in den Tupeln und dem Ausgabstrom.

3.3 Implementierung des LZ77-Algorithmus

Listing 1: LZ77 Pseudocode

```
1  while(!end_of_data )
2  {
3          index = 0;
4          length = 0;
5          /*
6           * Den Suchpuffer nach der
7           */
8          bool found = find_best_match(&index, &length, window);
9          /*
10          * Ausgabe des besten Treffers
11          */
12         if(found)
13         {
14                 output_tripel(index, length, preview[length]);
15         }
16         else
17         {
18                 output_tripel(0,0, preview[0]);
19         }
20 }
```

4 Der Huffman-Algorithmus

4.1 Shannon´s Informationstheorie als Grundlange des Huffman-Algorithmus

Die Informationstheorie von C. E. Shannon von 1948 bildet die Grundlage für das gesamte Gebiet der Datenkompression, so auch für die Huffman-Codierung.

4.1.1 Informationsgehalt

Der Informationsgehalt ist eine Größe, die über die Wahrscheinlichkeit für das Auftreten eines Ereignisses gebildet wird. Die Einheit des Informationsgehaltes ist bit. Der Informationsgehalt gibt also an, wieviele bits für eine Information benötigt werden. Die Wahrscheinlichkeit für das Auftreten eines Ereignisses liegt zwischen Null und Eins. Ein Ereignis kann beispielsweise ein einzelnes Zeichen sein. Treten zwei unterschiedliche Ereignisse mit gleicher Wahrscheinlichkeit auf, so ist der Informationsgehalt beider Ereignisse gleich. Je höher jedoch die Wahrscheinlichkeit für das Auftreten eines Ereignisses ist, desto niedriger ist der Informationsgehalt des Ereignisses. Für den umgekehrten Fall gilt je niedriger die Wahrscheinlichkeit, desto höher der Informationsgehalt eines Ereignisses.

$$\boxed{h(A) = -ld\ p(A)} \tag{1}$$

4.1.2 Entropie

Die Entropie oder auch der mittlere Informationsgehalt beschreibt den Mittelwert der Informationsgehalte aller einzelnen Ereignisse oder Zeichen.

$$\boxed{H = \sum_{i=1}^{n} p(A_i) * h(A_i)} \tag{2}$$

Die Entropie ist damit das Maß für die mittlere Anzahl der Binärzeichen, die für die Codierung einer Quelle benötigt werden.
Ist die Wahrscheinlichkeit für das Auftreten eines einzelnen Zeichens 1 so ist die Entropie 0.
Bei der Codierung einer Quelle darf die mittlere Anzahl an Bits nicht kleiner sein als die Entropie, da ansonsten ein Komprimierverlust entsteht. Damit stellt die Entropie die untere Grenze für die Kompressionsrate dar.

4.1.3 Mittlere Wortlänge

Die mittlere Wortlänge wird gebildet durch die Summe der Produkte von Wortlänge einzelner Zeichen und deren Auftrittswahrscheinlichkeit.

$$\boxed{L = \sum_{i=1}^{n} p(A_i) * l(A_i)} \tag{3}$$

4.1.4 Redundanz

Die Redundanz beschreibt praktisch den Overhead an bits, der bei einer Codierung entstehen kann. Die Redundanz wird aus der Differenz von mittlerer Wortlänge und Entropie errechnet.

$$\boxed{R = L - H} \tag{4}$$

4.1.5 Verlustfreie Kompression

Bei der verlustfreien Kompression werden häufig aufretende Zeichen mit kurzen Binärsequenzen kodiert, selten auftretende Zeichen werden mit längeren Codeworten dargestellt. Die Qualität einer Codierung kann anhand der Redundanz festgemacht werden. Ist diese 0, so liegt eine optimale Codierung vor. Deshalb werden diese Kompressionsverfahren als Entropiecodierungen bezeichnet.

4.1.6 Relevanz der Informationstheorie für Huffman-Codierung

Mit Hilfe der Shann´schen Informationtheorie kann ein optimaler Code erzeugt und überprüft werden. So ist ein Huffman-Code um so besser, wenn er eine möglichst geringe Redundanz (nach Shannon) aufweist.

4.2 Beschreibung des Huffman-Algorithmus

Der Huffman-Algorithmus erzeugt einen Präfix-Code. Bei einem Präfix-Code ist kein Codewort Präfix eines anderen Codewortes. Des weiteren werden Zeichen, die eine hohe Auftrittswahrscheinlichkeit besitzten mit kürzeren Codewörtern, Zeichen, die geringe Auftrittswahrscheinlichkeiten besitzen mit längeren Codewörtern belegt. Die beiden Zeichen mit den geringsten Auftrittswahrscheinlichkeiten werden Codewörtern mit gleicher Länge zugeordnet. Ein Präfix-Code läßt sich sehr gut in einer Baumstruktur darstellen.

4.3 Ein Anwendungsbeispiel zum Huffman-Code

Die Konstruktion eines Huffman-Codes erfordert die Kenntnis eines Quellalphabets und der Auftrittswahrscheinlichkeit der einzelnen Zeichen des Quellalphabets. Im ersten Schritt werden die einzelnen Zeichen nach ihrer Auftrittswahrscheinlichtkeit sortiert. Gegeben sei das Quellalphabets A=(a, b, c, d, e). [Tabelle 3] Die berechnete Entropie beträgt $H = 2.14$ bit. Im praktischen Fall sollten bei einem optimalen Präfix-Code die einzelnen Codewörter also nicht länger als 3 bit sein.

A	p
a	0.4
b	0.2
c	0.18
d	0.11
e	0.11

Tabelle 3: Die Zeichen des Quellalphabets sortiert nach ihren Auftrittswahrscheinlichkeiten

Im Folgenden werden sukzessiv die beiden am Seltensten auftretenden Zeichen mit ihrer Wahrscheinlichkeit zusammengefasst.

A	p
a	0.4
de	0.22
b	0.2
c	0.18

Tabelle 4: Auftrittswahrscheinlichkeiten nach Zusammenfassung von d und e

Die Zeichen d und e werden zusammengefasst und ihre Auftrittswahrscheinlichkeit summiert. [Tabelle 4] [Abbildung 1] Anhand der neu errechneten Wahrscheinlichkeit ist die Zeichenkette neu in die Tabelle einsortiert worden.

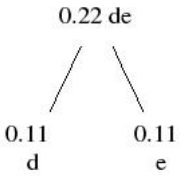

Abbildung 1: Durch die Konkatenation von 'd' und 'e' entsteht das Zeichen 'de'. Die Auftrittwahrscheinlichkeit von 'de' ist 0.22.

Die Zeichen b und c werden zusammengefasst und ihre Auftrittswahrscheinlichkeit summiert. [Tabelle 5] [Abbildung 2] Anhand der neu errechneten Wahrscheinlichkeit ist die Zeichenkette neu in die Tabelle einsortiert worden.

A	p
a	0.4
bc	0.38
de	0.22

Tabelle 5: Auftrittswahrscheinlichkeit nach Zusammenfassung von b und c

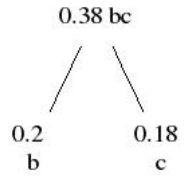

Abbildung 2: Durch die Konkatenation von 'b' und 'c' entsteht das Zeichen 'bc'. Die Auftrittwahrscheinlichkeit von 'bc' ist 0.38.

Die Zeichen 'bc' und 'de' werden zusammengefasst und ihre Auftrittswahrscheinlichkeit summiert.[Tabelle 6] [Abbildung 3] Anhand der neu errechneten Wahrscheinlichkeit ist die Zeichenkette neu in die Tabelle einsortiert worden.

A	p
a	0.4
bcde	0.6

Tabelle 6: Auftrittswahrscheinlichkeit nach Zusammenfassung von bc und de

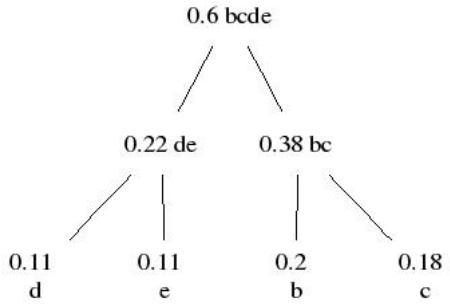

Abbildung 3: Durch die Konkatenation von 'bc' und 'de' entsteht das Zeichen 'bcde'. Die Auftrittwahrscheinlichkeit von 'bcde' ist 0.6.

Die Zeichen 'bcde' und 'a' werden zusammengefasst und ihre Auftrittswahrscheinlichkeit summiert. [Tabelle 7] [Abbildung 4] Anhand der neu errechneten Wahrscheinlichkeit ist die Zeichenkette neu in die Tabelle einsortiert worden.

A	p
abcde	1

Tabelle 7: Auftrittswahrscheinlichkeit nach Zusammenfassung von bcde und a

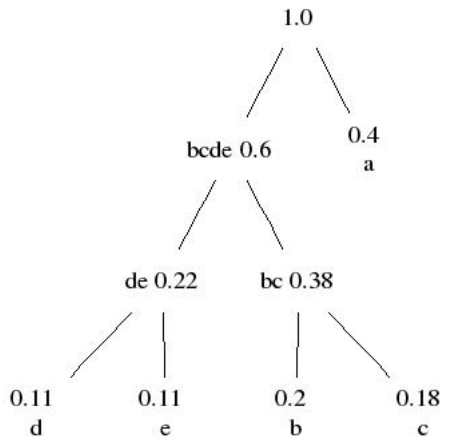

Abbildung 4: Durch die Konkatenation von 'bcde' und 'a' entsteht das Zeichen 'abcde'. Die Auftrittwahrscheinlichkeit von 'abcde' ist 1. Der Baum ist nun vollständig.

Die Kanten des fertigen Baums werden noch mit Werten aus 0, 1 gewichtet. Dabei wird die linke Kante eines Knotens mit 0, die rechte Kante mit 1 beschriftet. Anschließend kann die Codierung für ein Zeichen einfach von dem entstandenen Baum abgelesen werden. [Abbildung 5]

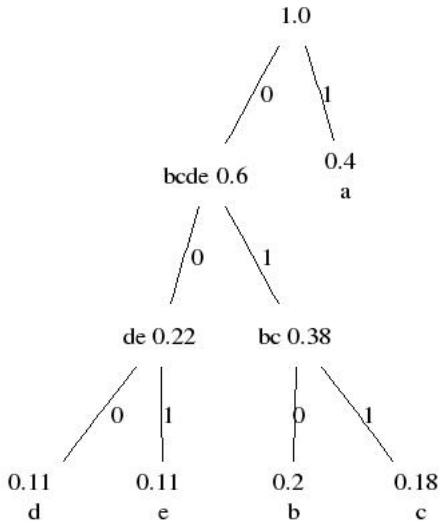

Abbildung 5: Anhand des Baumes mit den gewerteten Kanten kann nun die Kodierung für ein einzelnes Zeichen abgelesen werden. Das Zeichen 'c' wird demnach mit der Bit-Sequenz 011 codiert.

4.4 Implementierung des Huffman-Algorithmus

4.4.1 Pseudo-Code mit Erklärungen

Listing 2: Huffman Pseudocode

```
1  HUFFMAN(C)
2          n <- |C|
3          Q <- C
4          for i<-1 to n-1
5            do Allokieren eines neuen Knotens z
6               links[z] <- x <- EXTRACT-MIN(Q)
7               rechts[z] <- y <- EXTRACT-MIN(Q)
8               f[z] <- f[x] + f[y]
9               INSERT(Q, z)
10         return EXTRACT-MIN(Q)
```

C ist eine Menge von Objekten. Jedes Element dieser Menge enhält ein Zeichen und die Auftrittswahrscheinlichkeit dieses Zeichens. Der zu entwickelnde Baum mit dem Huffman-Code wird nach dem Buttom-Up-Prinzip erzeugt. Dafür wird eine Prioritätswarteschlange Q angelgt. Alle Zeichen mit ihren Häufigkeiten werden in die Warteschlange geschrieben. In $|C| - 1$ Durchläufen werden neue innere Knoten durch verschmelzen vorhandener Knoten erzeugt. Die Knoten mit den Zeichen und Auftrittswahrscheinlichkeiten werden als Blätter gespeichert. Das Verschmelzen von Knoten geschieht anhand ihrer Auftrittswahrscheinlichkeiten. Es werden immer die beiden Knoten mit der kleinsten Auftrittswahrscheinlichkeit aus der Prioritätswarteschlange geholt und miteinander verschmolzen. Anschließen wird der neue Knoten in die Prioritätswarteschlange geschrieben. Die Funktion HUFFMAN(C) gibt am Ende die Wurzel des erstellten Baums zurück.

4.4.2 Beispiel zur Implementierung des Huffman-Codes

Als erster Schritt werden die Zeichen einer Menge anhand ihrer Auftrittswahrscheinlichkeit in eine Prioritätswarteschlange gegeben. [Abbildung 6]

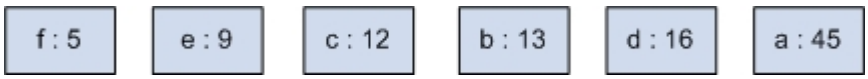

Abbildung 6: In der Prioritäts-Warteschlange befinden sich Charakter-Objekte, die nach ihrer Auftrittshäufigkeit sortiert sind.

Anschließend werden die jeweils kleinsten Knoten rekursiv miteinander

verschmolzen bis nur noch die Wurzel des Baumes in der Prioritätswarteschlange existiert. [Abbildung 7]

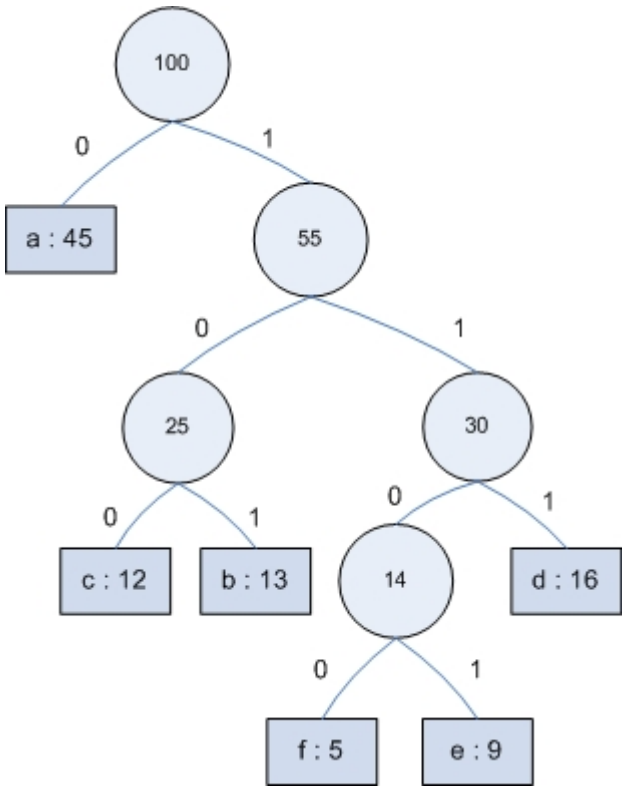

Abbildung 7: An den Kanten des fertigen Baums kann nun die Codierung für ein einzelnes Zeichen abgelesen werden.

5 CompressorTestSuite

Die CompressorTestSuite ist ein Framework zum Testen von Kompressionsalgorithmen. Es vergleicht die Größe der zu komprimierenden Datei mit der Größe der komprimierten Datei. Die Ergebnisse werden grafisch dargestellt. Die y-Achse repräsentiert die Dateigröße und die x-Achse stellt die verschiedenen Dateien dar, mit denen der Test durchgeführt werden soll. [Abbildung 8]

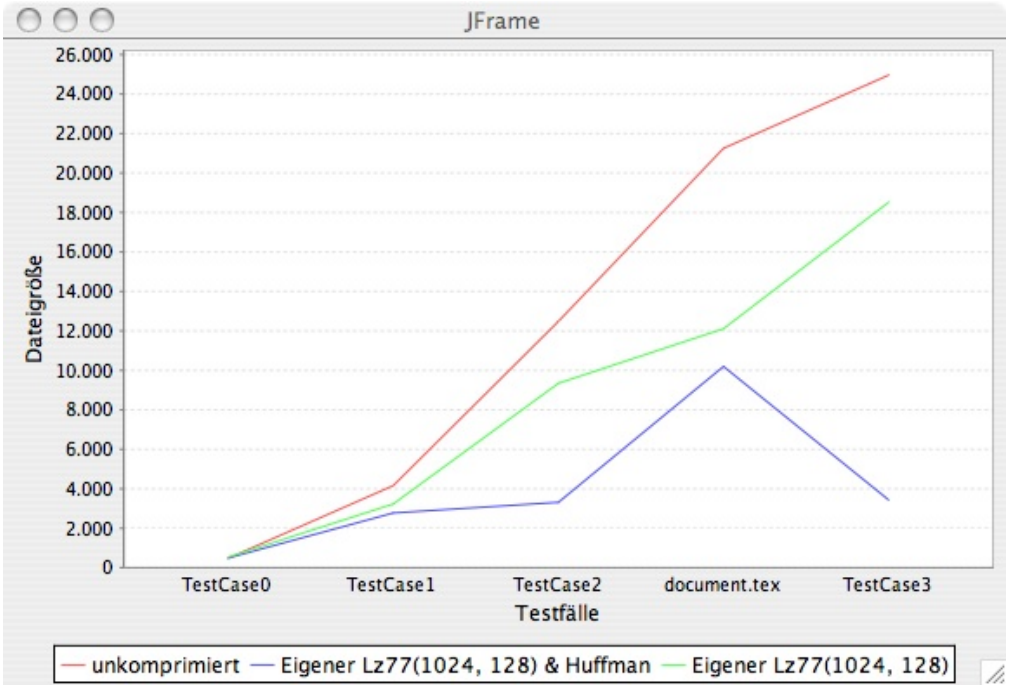

Abbildung 8: Screenshot der CompressorTestSuite

Jeder gezeichnete Graph repräsentiert dabei einen Kompressionsalgorithmus, wobei immer ein Graph der unkomprimierten Datei als Vergleich eingeblendet wird.

5.1 Konfiguration

Um einen Testlauf in der CompressorTestSuite vorzubereiten sind einige Konfigurationsschritte notwendig. Dazu legt man eine XML-Datei mit beliebigem Namen an und füllt diese mit einem Inhalt ähnlich dem in [Listing 3]. Das Wurzelelement der XML-Datei muss <TestSuite> sein [Listing 3 Zeile 2]. Darunter folgen ein oder mehrere <Tester> [Listing 3 Zeile 3] und ein oder

mehrere <TestCase> [Listing 3 Zeile 15] Elemente. Kommentare werden in normale XML Kommentare geschrieben.

<TestSuite> Diese Tag ist das Wurzelelement dieser Konfigurationsdatei.

<Tester> Diese Einträge repräsentieren die Algorithmen, die verwendet werden sollen. Bei manchen Algorithmen können Parameter angegeben werden. Folgende Algorithmen stehen zur Verfügung:

- lz77.CustomDeflateCompressor
- lz77.Lz77
- deflate.CustomDeflateWrapper
- deflate.HuffmanOnlyWrapper

Die beiden Algorithmen im lz77 Paket sind eigene Implementierungen. Im Paket deflate befinden sich Wrapper der Algorithmen aus der Java-API.

<TestCase> In diesem Tag wird ein Testfall spezifiziert. Als Parameter muss ein Name und ein Pfad zu einer Datei angegeben werden. Der Pfad wird relativ zu dieser Konfigurationsdatei angegeben. Der Name wird als Beschriftung dieses Testfalls im Diagramm genutzt.

Listing 3: XML Konfigurationsdatei

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <TestSuite>
3    <Tester param1="1024" param2="128">
4          lz77.CustomDeflateCompressor
5    </Tester>
6    <Tester param1="1024" param2="128">
7          lz77.Lz77
8    </Tester>
9    <!-- <Tester>
10         deflate.CustomDeflateWrapper
11   </Tester> -->
12   <!-- <Tester>
13         deflate.HuffmanOnlyWrapper
14   </Tester> -->
15   <TestCase name="document.tex" file="document.tex"/>
```

```
16    <TestCase name="TestCase0" file="Text0.txt"/>
17    <TestCase name="TestCase1" file="Text1.txt"/>
18    <TestCase name="TestCase2" file="Text2.txt"/>
19    <TestCase name="TestCase3" file="Text3.txt"/>
20  </TestSuite>
```

5.2 Ausführung

Nach dem Start des Programms muss in dem Datei öffnen Dialog die XML Konfigurationsdatei ausgewählt werden. Anschließend dauert es eine ganze Zeit bis ein Dialog mit den Ergebnissen angezeigt wird.

6 Experimente

In den Testfällen wird das Kompressionsverhalten von verschiedenen Datei-Typen mit den bekannten Algorithmen LZ77, Huffman und Deflate untersucht. Dabei kommen sowohl der selbst implementierte LZ77, sowie die Algorithmen Huffman und Deflate aus der Java-API und die Kombinationen von eigenem LZ77 und Huffman aus der API zum Einsatz. Es soll die Effektivität der Kompression getestet werden.
Unter der Effektivität ist das Größenverhältnis in Bytes zwischen der Orginal- und der kompremierten Datei zu verstehen. Zusätzlich wird die Effektivität abhängig von der Dateigröße der Orginaldatei untersucht. D.h, ob besser eine kleine oder eine große Datei komprimiert werden kann.

6.1 Experiment 1: Kompression von Textdateien

Als Testdateien kommen verschiedene Textdateien zum Einsatz. In diesem Testfall soll die Auswirkung der Parameter „Windowsize“ und „Previewsize“ auf die Größe der komprimierten Dateien untersucht werden.

6.1.1 LZ77 und Huffman(Java-API)

6.1.1.1 Beschreibung des Tests Bei diesem Experiment kommt der selbstimplementierte LZ77 Algorithmus kombiniert mit dem Huffman-Algorithmus aus dem java.util.zip Paket zum Einsatz

Wörterbuchgröße	Vorschaugröße	Tupellänge	Tupel
256	32	3	(W,P,D)
1024	128	4	(WW,P,D)
1024	256	4	(WW,P,D)
2048	512	5	(WW,PP,D)
8192	1024	5	(WW,PP,D)
32768	2048	5	(WW,PP,D)
32768	4096	5	(WW,PP,D)

Tabelle 8: Eingestellte Parameter für Experiment 1

[Tabelle 8] zeigt die Eingestellten Parameter für den Deflate, die bei diesem Experiment benutzt wurden. Die Wörterbuchgröße, die Vorschaugröße, sowie die Tupellänge sind in Bytes angegeben. In der Spalte Tupel sieht man die Vorlage eines Tupels, was sich aus der eingestellten Wörterbuchgröße und

der Vorschaugröße ergibt. Dabei steht ein „W“ für ein Byte zur Adressierung im Wörterbuch, ein „P“ für ein Byte der Länge des codierten Textabschnitts, dessen Maximalwert durch die Größe des Vorschaufensters bestimmt ist. Das „D“ steht für den Datenanteil des Tupels, der in unserer Implementierung ein Byte beträgt.

Alle folgenden Testfälle wurden mit jeder Parameterkonstelation aus [Tabelle 8] kombiniert.

Testfall 1 Eine kurze Datei (476 Bytes) mit nur einer Zeile und deutschem Wikipedia-Quelltext.

Testfall 2 Eine Textdatei (4160 Bytes) mit einem ganzen Artikel bestehend aus deutschem Wikipedia-Quelltext.

Testfall 3 Der gleiche Text, wie in Testfall 2, jedoch wurde dieser Artikel drei mal hintereinander in die Datei (12481 Bytes) geschrieben, so dass ein langes Teilstück der Datei Redundant ist.

Testfall 4 Bei diesem Tesfall handelt es sich um den gleichen Text, wie bei Testfall 3, jedoch wurde diese Datei nochmal verdoppelt, so dass noch mehr Redundanz besteht. Die Datei hat eine Länge von 24962 Bytes.

Testfall 5 Bei diesem Testfall handelt es sich zur Abwechslung um einen deutschen Fließtext ohne künstlich hervorgerufene Redundanz. Die Größe der Datei beträgt 24016 Bytes, ist also vergleichbar mit der Datei aus Testfall 4. Da es sich bei der Datei um Latex Quelltext handelt, sind einige Schlüsselwörter redundant, was aber im Vergleich zu einer Datei mit manipulierter Redundanz kaum ins Gewicht fällt.

6.1.1.2 Hypothese Je größer das Vorschaufenster und das Wörterbuch, desto längere Textabschnitte kann man maximal mit einem Tupel komprimieren. Das bringt natürlich nur einen Vorteil, wenn man einen entsprechenden Text hat, indem die Redundanten Stücke möglichst so lang wie das Vorschaufenster sind. Dieser Vorteil kann natürlich bei Texten mit wenig Redundanz zu einem Nachteil werden, da die Tupel unter Umständen aus mehr Bytes bestehen, obwohl kaum lange Textstellen redundant vorkommen. Da unsere Implementierung nur Tupel aus ganzen Bytes benutzt, gibt es z.B. bei einer Wörterbuchgröße bzw. Vorschaugröße von 256 solch eine Grenze, da nur ein Wert ¡= 256 in einem Byte gespeichert werden kann. Werte ¿ 256 benötigen direkt 2 Bytes, obwohl einige Bits unter Umständen ungenutzt sind.

Kaum Redundanz bedeutet nämlich, dass die Anzahl der benötigten Tupel steigt und da ein Tupel mindestens 3-mal so lang, wie ein einzelnes Datenbyte, kann dieser Fall auch zu einer höheren Dateigröße der komprimierten Datei führen, wie die ursprüngliche Datei.

6.1.1.3 Verifikation In [Abbildung 9] sieht man die Ergebnisse des Experiments. Das Bild zeigt, dass die Testfälle mit hoher Redundanz auch bei verdoppelter Dateigröße im Ergebnis nur marginal größer werden. Man kann, wenn auch nur schwach, erkennen, dass bei Testfall 1 der Nachteil der Kompression heraus kommt. Die ursprüngliche Datei ist kleiner als die komprimierte Datei. Verdeutlicht wird das in [Abbildung 10]. Die rote Linie, die die unkkomprimierte Dateigröße repräsentiert, liegt unter den Linien der komprimierten Datei.

Der Hügel bei Testfall 5 kommt daher, dass dies die Fließtextdatei ohne Redundanz ist und dadurch die Anzahl der benötigten Tupel nach oben schnellt.

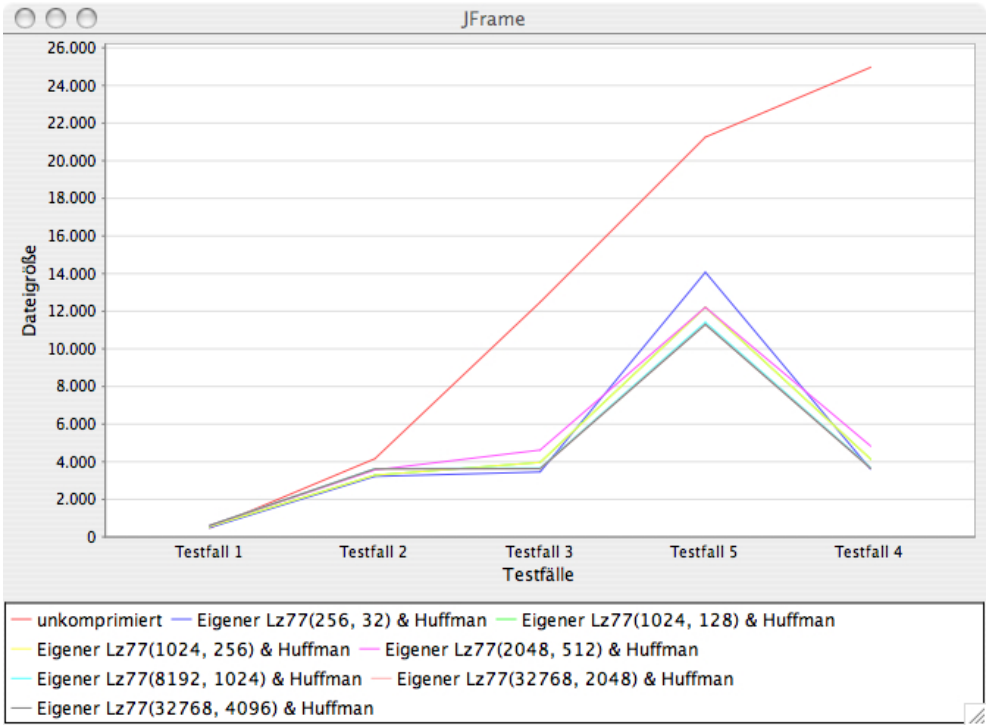

Abbildung 9: Textkompression mit unterschiedlichen Parametern

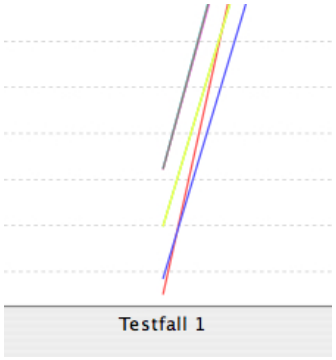

Abbildung 10: Negativerfolg bei Testfall 1

6.1.1.4 Fazit Das Ergebnis des Experiments spiegelt genau die Vermutungen der Hypothese wieder. Dass das Parameter Paar (256, 32) häufig am Besten abschneidet, liegt wie bereits angesprochen, an der Implementierung, da die Tupelgröße bei diesem Algorithmus bei mit 3 Byte mindestens ein Byte kleiner ist, als bei den anderen Algorithmen. Lässt man diesen Algorithmus außer acht und vergleicht nur die Algorithmen mit vergleichbarer Tupelgröße, dann sieht man, dass der Algorithmus mit den Parametern (32768, 4096) besser ist, als die Algorithmen mit kleineren Parametern.

6.2 Experiment 2: Kompression von Binärdateien

Als Testdateien kommen verschiedene Binärformate zum Einsatz. Des weiteren soll die Größe der Testdateien variieren. Die Binärdatei-Typen sind wav- und mp3-, bmp- und gif- und verschiedene bereits komprimierte Dateien.

6.2.1 Kompression von wav- und mp3-Dateien

6.2.1.1 Beschreibung des Experiments Es werden eine WAV-Datei der Größe 42.2 MByte und eine MP3-Datei der Größe 3.8 MByte mit dem Huffman- und dem Deflate-Algorithmus aus der Java-API komprimiert.

6.2.1.2 Hypothese Eine WAV-Datei ist eine unkomprimierte Musik-Datei. Bei dieser ist zu erwarten, daß eine Kompression mit den verschiedenen Algorithmen eine Veringerung der Byte-Größe zur Folge hat.
Eine Komprimierung der MP3-Datei erscheint nicht möglich, da diese Datei breits in einem verlustbehafteten Kompressionsformat vorliegt. Beim MP3-Format werden nämlich Tonfrequenzen, die vom menschlichen Gehör nicht wahrgenommen werden, herausgeschnitten.

6.2.1.3 Verifikation Betrachtet man die beiden Graphen [Abbildung 11] [Abbildung 12], so ist zu erkennen, dass wie erwartet keine Kompression der MP3-Datei stattfindet.
Widererwartend ist jedoch auch nur eine geringe Kompression der WAV-Datei möglich. Dies wird auch noch mal durch das Kompressionsprogramm zip überprüft. Auch dieses erreicht nur eine sehr geringe Kompression. Es wird die Konfigurationsdatei aus [Listing 4] verwendet.

Listing 4: XML Konfigurationsdatei Binärdateien Music

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <TestSuite>
3         <Tester>deflate.CustomDeflateWrapper</Tester>
4         <Tester>deflate.HuffmanOnlyWrapper</Tester>
5         <TestCase name="MP3" file="music/die_da.mp3"/>
6         <TestCase name="WAV" file="music/die_da.wav"/>
7 </TestSuite>
```

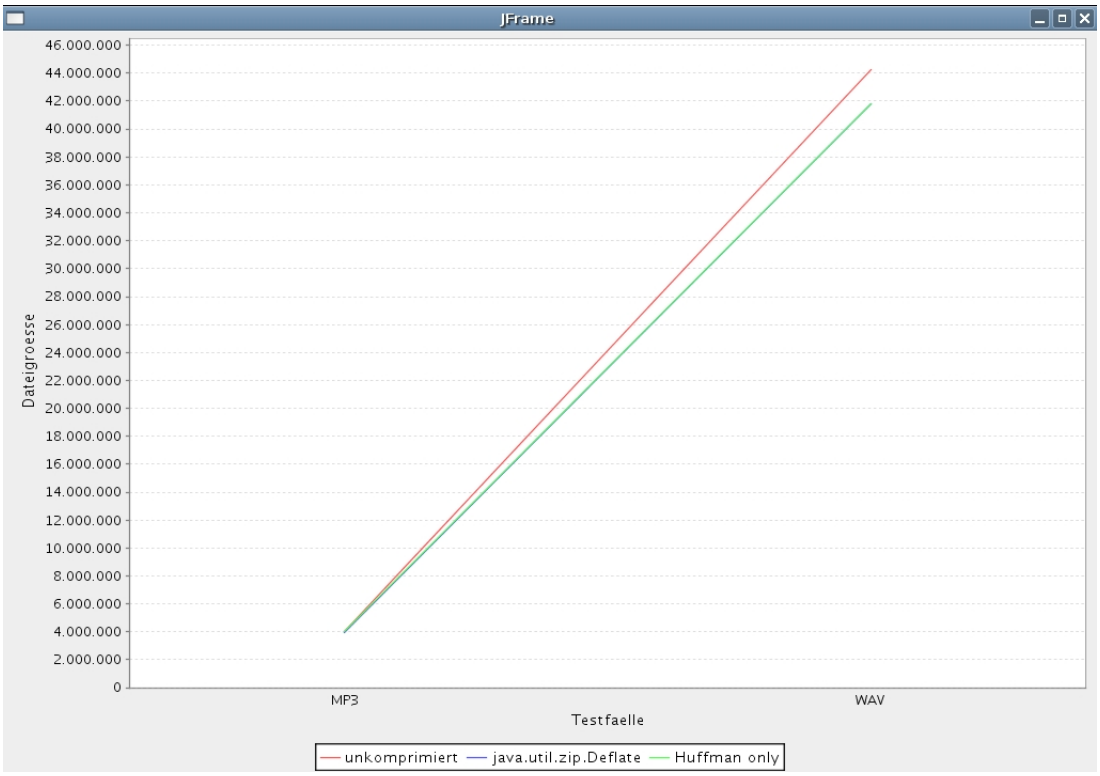

Abbildung 11: Kompression von mp3- und wav-Dateien mit dem Huffman-Algorithmus

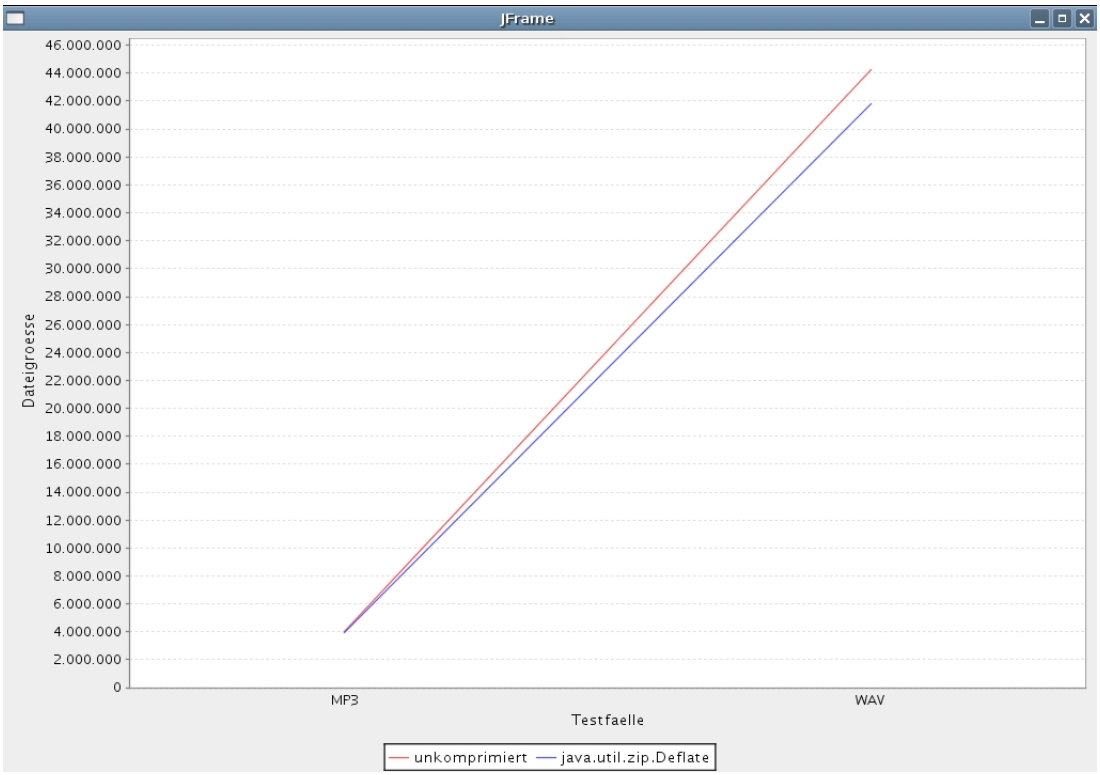

Abbildung 12: Kompression von mp3- und wav-Dateien mit dem Deflate-Algorithmus

6.2.1.4 Fazit Die Erklärung für die geringe Komprimierbarkeit der WAV-Datei ist nur dadurch zu erklären, dass das WAV-Format schon eine Komprimierung besitzt. Dies ist durchaus nach der Spezifikation für das WAV-Format möglich. Es kann sich beim WAV-Format also wie bei einer MP3-Datei um eine verlustbehaftete Kompression handeln.

6.2.2 Kompression von bmp- und gif-Dateien

6.2.2.1 Beschreibung des Experiments Es werden zwei bmp- und zwei gif-Dateien mit dem Huffman und dem Deflate-Algorithmus aus der Java-API kompremiert. Eine der bmp- und eine der gif-Dateien enthalten je nur zwei Farben. Die Größe in Pixeln der bmp-Dateien beträgt 346x435, die Größe in KByte 441.8 KByte [Abbildung 13]. Die gif-Dateien sind 350x439 Pixel groß, wobei die Größe in Byte der zweifarbigen Datei 953 Byte und die der anderen gif-Datei 128 KByte beträgt [Abbildung 14].

Abbildung 13: Bitmap 1 und 2, 441.8KByte groß

Abbildung 14: gif 1 und 2 der Größe 953 Byte und 128 KByte

6.2.2.2 Hypothese Die Kompression der bmp-Dateien sollte erwartungsgemäß eine veringerte Byte-Größe ergeben, wobei bei der zweifarbigen Datei die Kompression deutlich höher ausfallen sollte, als bei der mehrfarbigen. Dies würde an der hohen Redundanz an gleichen Farbpixeln liegen, die bei der zweifarbigen Datei natürlich sehr viel höher ist.
Bei den gif-Dateien ist keine Kompression zu erwarten, da das gif-Format intern mit dem LZW-Algorithmus arbeitet. Dieser basiert auf dem LZ78, welcher der komerzielle Nachfolger des LZ77 ist.

6.2.2.3 Verifikation Erwartungsgemäß ist keine Kompression der GIF-Dateien zu verzeichnen. Dabei ist es unerheblich, mit welchem der beiden Algorithmen die Kompremierung vorgenommen wird. Dies ist daran zu erkennen, dass alle drei Graphen für GIF 1 und GIF 2 fast identisch verlaufen. Auch die Erwartungen für die Kompression der BMP-Dateien wird erfüllt. Beide Dateien sind zu einem hohen, bzw. sehr hohen Grade komprimierbar. Auffällig ist, dass sich die Graphen von Huffman und Deflate kaum voneiander unterscheiden [Abbildung 15]. Es wird die Konfigurationsdatei aus Listing [Listing 5] verwendet.

Listing 5: XML Konfigurationsdatei Binärdateien Bilder

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <TestSuite>
3         <Tester>deflate.CustomDeflateWrapper</Tester>
4         <Tester>deflate.HuffmanOnlyWrapper</Tester>
5         <TestCase name="BMP␣1␣(zweifarbig)"
6                                 file="bilder/blau_rot.bmp"/>
7         <TestCase name="BMP␣2␣(Fritillaria)"
```

```
8                                   file="bilder/VAN_GOGH_Fritillaria_1.bmp"/>
9           <TestCase name="GIF␣1␣(zweifarbig)"
10                                  file="bilder/green_orange.gif"/>
11          <TestCase name="GIF␣2␣(Vero)"
12                                  file="bilder/Vero-Van-Gogh.gif"/>
13  </TestSuite>
```

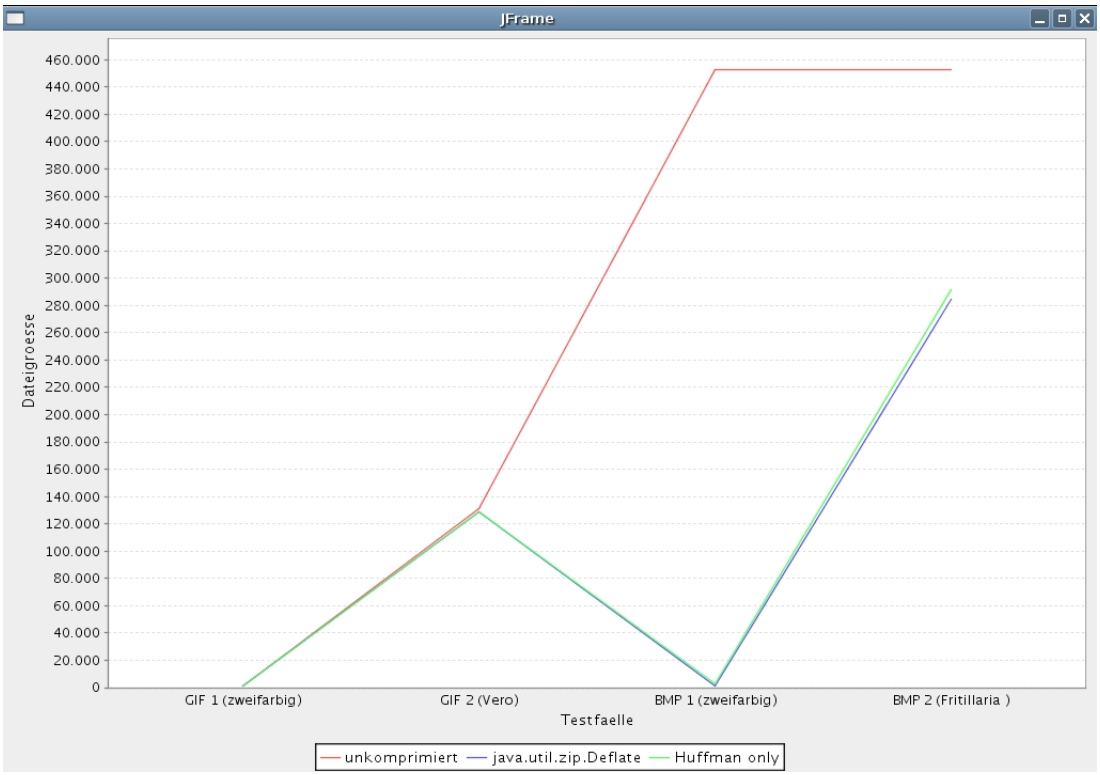

Abbildung 15: Kompression von bmp- und gif-Dateien mit Huffman- und Deflate-Algorithmus

6.2.2.4 Fazit Die Auswertung der Graphen lässt den Schluß zu, dass der Huffman-Algorithmus die hauptsächliche Kompression übernimmt. Dies kann am identischen Verlauf der Graphen für den Huffman- und Deflate-Algorithmus gesehen werden.

6.2.3 Vergleich des Deflate zu anderen Kompressionsverfahren

6.2.3.1 Beschreibung des Experiments Es wird eine Bitmap-Datei der Größe 441.8KByte (rechte Datei aus [Abbildung 13]) mit den Verfahren

bz2, jar, rar, tar, tar.gz und zip komprimiert.

6.2.3.2 Hypothese Eine weitere Kompression der diversen komprimierten Dateien sollte eigentlich nicht möglich sein. Eine Ausnahme bietet das Format tar, da dieses Dateien nicht komprimiert, sondern nur zusammenfügt. Hier ist eine Kompression zu erwarten.

6.2.3.3 Verifikation Allen Erwartungen nach ist eine weitere verlustfreie Komprimierung einer verlustfrei komprimierten Datei nicht möglich. Die Archiv-Datei tar kann jedoch noch komprimiert werden. Hierbei ist eine Verbesserung der Kompressionsrate mit dem Deflate- gegenüber dem Huffman-Algorithmus zu beobachten [Abbildung 16]. Es wird die Konfigurationsdatei aus [Listing 6] verwendet.

Listing 6: XML Konfigurationsdatei Binärdateien Kompression

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <TestSuite>
3          <Tester>deflate.CustomDeflateWrapper</Tester>
4          <Tester>deflate.HuffmanOnlyWrapper</Tester>
5          <TestCase name="bz2"
6                          file="VAN_GOGH_Fritillaria_1.bmp.bz2"/>
7          <TestCase name="jar"
8                          file="VAN_GOGH_Fritillaria_1.bmp.jar"/>
9          <TestCase name="rar"
10                         file="VAN_GOGH_Fritillaria_1.bmp.rar"/>
11         <TestCase name="tar"
12                         file="VAN_GOGH_Fritillaria_1.bmp.tar"/>
13         <TestCase name="tar.gz"
14                         file="VAN_GOGH_Fritillaria_1.bmp.tar.gz"/>
15         <TestCase name="zip"
16                         file="VAN_GOGH_Fritillaria_1.bmp.zip"/>
17 </TestSuite>
```

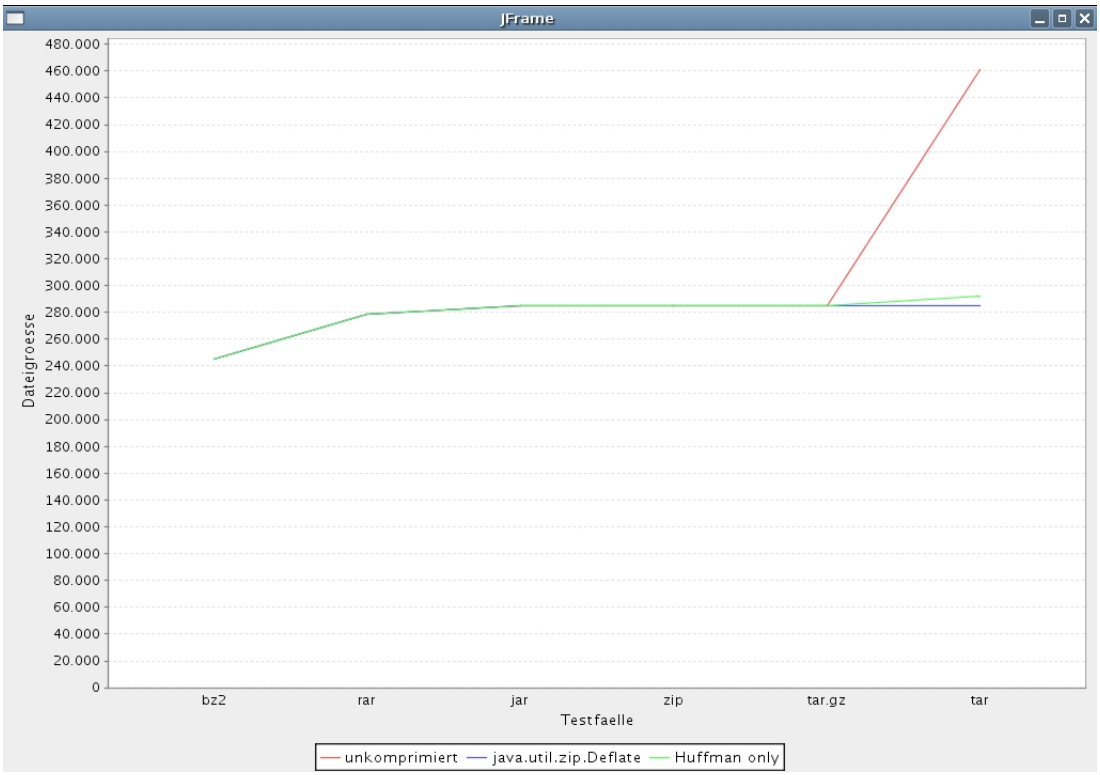

Abbildung 16: Kompression von komprimierten Dateien mit Huffman- und Deflate-Algorithmus

6.2.3.4 Fazit Die verlustfreie Komprimierung einer verlustfrei komprimierten Datei ist nicht möglich, da schon alle Redundanzen aus der Datei entfernt wurden. Die verbesserte Kompression durch den Deflate-Algorithmus wäre dadurch zu erklären, dass eine tar-Datei Redundanzen enthält. Diese würden durch den LZ77 mit dem Wörterbuch zusätzlich entfernt werden.

6.3 Experiment 3: Kompression von Textdateien

In unserem Testfall 3 haben wir uns entschieden die Komprimierung von Textdateien verschiedener Größen sowie die Komprimierung verschiedener Textdateien mit und ohne Redundanz zu Vergleichen. Es soll verdeutlicht werden in wie weit sich die Kompressionsrate verändert wenn entweder nur der LZ77 Algorithmus oder nur der Huffman Algorithmus, eine eigene Implementierung oder die Implementierung der Java API verwendet werden.

6.3.1 LZ77 und Huffman(Java-API)

6.3.1.1 Beschreibung des Experiments Es soll die Kompressionsrate einer Textdatei in Verbindung mit dem selbst implementierten LZ77 Algorithmus und dem Huffman Algorithmus der Java API verglichen werden. Es werden verschiedene Textdateien unterschiedlicher Größe einmal mit unserer Implementierung des LZ77 und einmal mit dem Huffman Algorithmus komprimiert.

6.3.1.2 Hypothese Bei einer kleinen Datei erreicht der LZ77 eine geringere Kompressionsrate als der Huffman Algorithmus, da der LZ77 auf Redundanzen im Text angewiesen ist und der Huffman jedes Zeichen codiert. Bei einer großen Datei erwarten wir das der LZ77 Algorithmus eine höhere Kompressionsrate erreicht, da in einer größeren Datei häufiger Redundanzen auftreten.

6.3.1.3 Verifikation Insgesamt ist in den Graphen [Abbildung 17] [Abbildung 18] zu sehen das die Huffman Algorithmus sowohl bei den kleinen als auch bei den großen Dateien eine bessere Kompressionsrate erziehlt als unser LZ77 Algorithmus. Bei den sehr kleinen Dateien ist aber zu sehen das der LZ77 einen Overhead produziert, so dass die Datei nach der "Komprimierung" größer ist als vorher. Es wird die Konfigurationsdatei aus [Listing 7] verwendet.

Listing 7: XML Konfigurationsdatei TextDateien LZ77 und Huffman(Java-API)

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <TestSuite>
3          <Tester param1="32768" param2="4096">lz77.Lz77</Tester>
4          <Tester>deflate.HuffmanOnlyWrapper</Tester>
5          <TestCase name="TestCase31␣:␣476␣␣bytes"
6                                      file="Text0.txt"/>
7          <TestCase name="TestCase31␣:␣4160␣␣bytes"
8                                      file="Text1.txt"/>
9          <TestCase name="TestCase31␣:␣24962␣␣bytes"
10                                     file="Text3.txt"/>
11         <TestCase name="TestCase31␣:␣954368␣␣bytes"
12                                     file="thebigone1.txt"/>
13 </TestSuite>
```

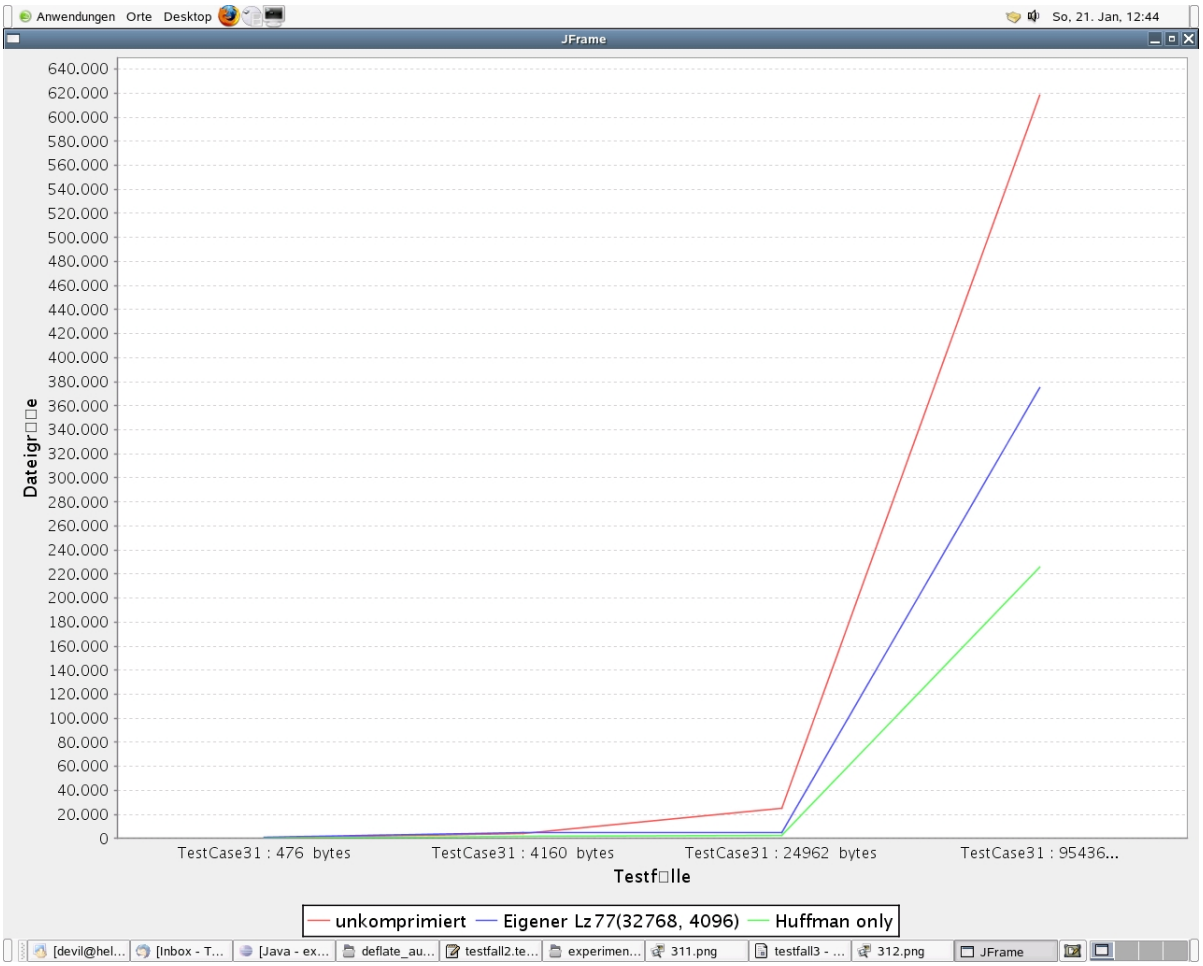

Abbildung 17: Kompression von Textdateien mit LZ77 und Huffman(Java-API)

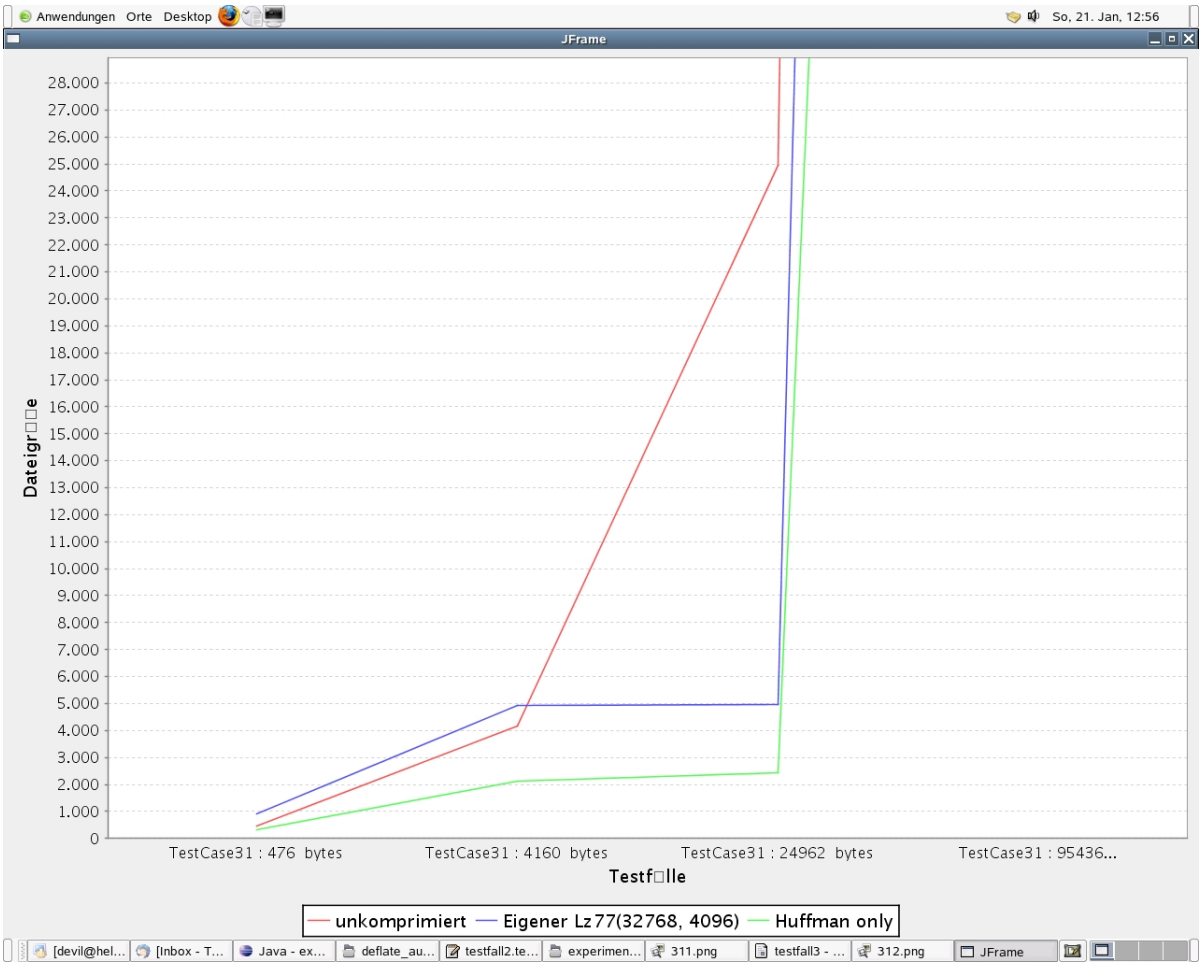

Abbildung 18: Kompression von Textdateien mit LZ77 und Huffman(Java-API)

6.3.1.4 Fazit Wir gehen davon aus das die wesentlich schlechteren Kompressionsraten unseres Algorithmus gegenüber dem Huffman-Algorithmus an unserer Implementierung lagen. Es ist aber zu erkennen das der LZ77 Algorithmus eine bessere Kompressionsrate erziehlt um so größer die Datei wird und um so mehr Redundanzen im Text vorkommen.

6.3.2 LZ77 und Huffman(Java-API)

6.3.2.1 Beschreibung des Experiments Es werden zwei gleich große Text Dateien mit unsere Implementierung des LZ77 und danach mit dem Huffman-Algorithmus komprimiert. Eine Datei ist eine beliebige Textdatei,

die andere Datei hat sehr viele Redundanzen.

6.3.2.2 Hypothese Aufgrund der beiden Kompressionsverfahren erwarten wir, dass obwohl die Dateien gleich gro“s sind, der LZ77 bei der Datei mit vielen Redundanzen eine höhere Kompressionsrate erzielt als der Huffman Algorithmus.

6.3.2.3 Verifikation Es ist in Graph [Abbildung 19] zu erkennen, dass der LZ77 Algorithmus bei der Textdatei mit wenigen Redundanzen (TestCase 312a) eine schlechtere Kompressionsrate erzielt als der Huffman Algorithmus. In TestCase 312b ist jedoch zu erkennen das die beiden Geraden sich schneiden und der LZ77 Algorithmus eine höhere Kompressionsrate erzielt. Es wird die Konfigurationsdatei aus [Listing 8] verwendet.

Listing 8: XML Konfigurationsdatei TextDateien LZ77 und Huffman(Java-API)

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <TestSuite>
3         <Tester param1="32768" param2="4096">lz77.Lz77</Tester>
4         <Tester>deflate.HuffmanOnlyWrapper</Tester>
5         <TestCase name="TestCase312a" file="Text3.txt"/>
6         <TestCase name="TestCase312b" file="redundanzen.txt"/>
7 </TestSuite>
```

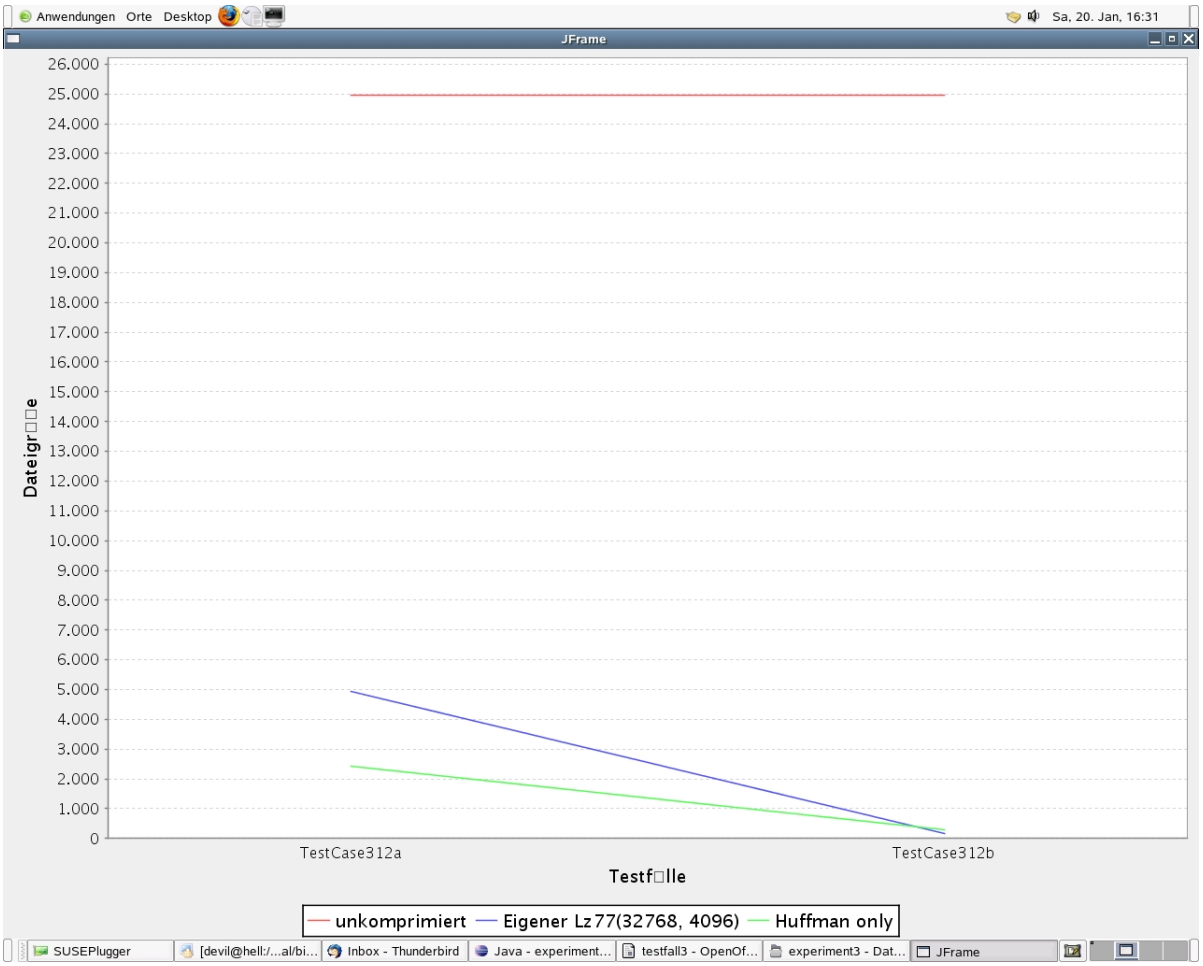

Abbildung 19: Kompression von Textdateien mit und ohne Redundanz mit LZ77 und Huffman(Java-API)

6.3.2.4 Fazit Unsere Hypothese das der LZ77 bei Dateien mit vielen Redundanzen eine höhere Kompressionsrate erziehlt ist somit bestätigt. Dies erklärt nun auch die Reihenfolge der Algorithmen im Deflate Algorithmus. In der noch nicht komprimierten Datei sind wesentlich mehr Redundanzen als in einer vom Huffman Algorithmus komprimierten Datei, da dieser eine Binärdatei als Ausgabe verwendet.

6.3.3 Deflate(Java-API) und LZ77 + Huffman(Java-API)

6.3.3.1 Beschreibung des Experiments In diesem Experiment sollen die Kompressionsraten unserer LZ77 Implementierung in Verbindung mit

dem Huffman-Algorithmus der Java Api mit dem Deflater der Java Api verglichen werden. Hierzu werden einmal Dateien unterschiedlicher Größe und einmal zwei Text Dateien gleicher Größe mit unterschiedlichen Redundanzen verwendet.
Es werden Text Dateien unterschiedlicher Größe gewählt. Alle Dateien werden einmal mit unserem LZ77 in Verbindung mit dem Huffman-Algorithmus der Java Api und einmal mit dem Deflater komprimiert.

6.3.3.2 Hypothese Wir erwarten bei den größeren Text Dateien eine höhere Kompressionsrate als bei den kleineren Dateien, da hier mehr Redundanzen vorkommen und der LZ77 eine bessere Kompression erreicht. Insgesamt erwarten wir das der Deflater der Java API eine bessere Kompressionsrate erzielt als die Kombination unsere Implementierung mit dem Huffman Algorithmus der Java Api.

6.3.3.3 Verifikation Es ist in dem Graph [Abbildung 20] zu erkennen das die Implementierung der Java API in allen Testfällen ein bessere Kompressionsrate erreicht als die Kombination mit unserer eigenen Implementierung. Die Kompressionsrate vergrössert sich weiterhin um so größer die zu komprimierende Datei wird. Während die komprimierte Datei im zweiten Testfall des Deflate Algorithmus der Java API 36,7 Prozent des Orginals beträgt, beträgt sie im letzten Testfall nur noch 31,3 Prozent des Originals. Es wird die Konfigurationsdatei aus [Listing 9] verwendet.

Listing 9: XML Konfigurationsdatei Deflate(Java-API) und LZ77 + Huffman(Java-API))

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <TestSuite>
3          <Tester>deflate.CustomDeflateWrapper</Tester>
4          <Tester param1="32768"
5                  param2="4096">lz77.CustomDeflateCompressor</Tester>
6          <TestCase name="TestCase32␣:␣476␣␣bytes"
7                                  file="Text0.txt"/>
8          <TestCase name="TestCase32␣:␣77649␣␣bytes"
9                                  file="Text1a.txt"/>
10         <TestCase name="TestCase32␣:␣113488␣␣bytes"
11                                 file="Text2a.txt"/>
12         <TestCase name="TestCase32␣:␣204945␣␣bytes"
13                                 file="Text3a.txt"/>
14         <TestCase name="TestCase32␣:␣618496␣␣bytes"
15                                 file="thebigone1.txt"/>
```

```
16 </TestSuite>
```

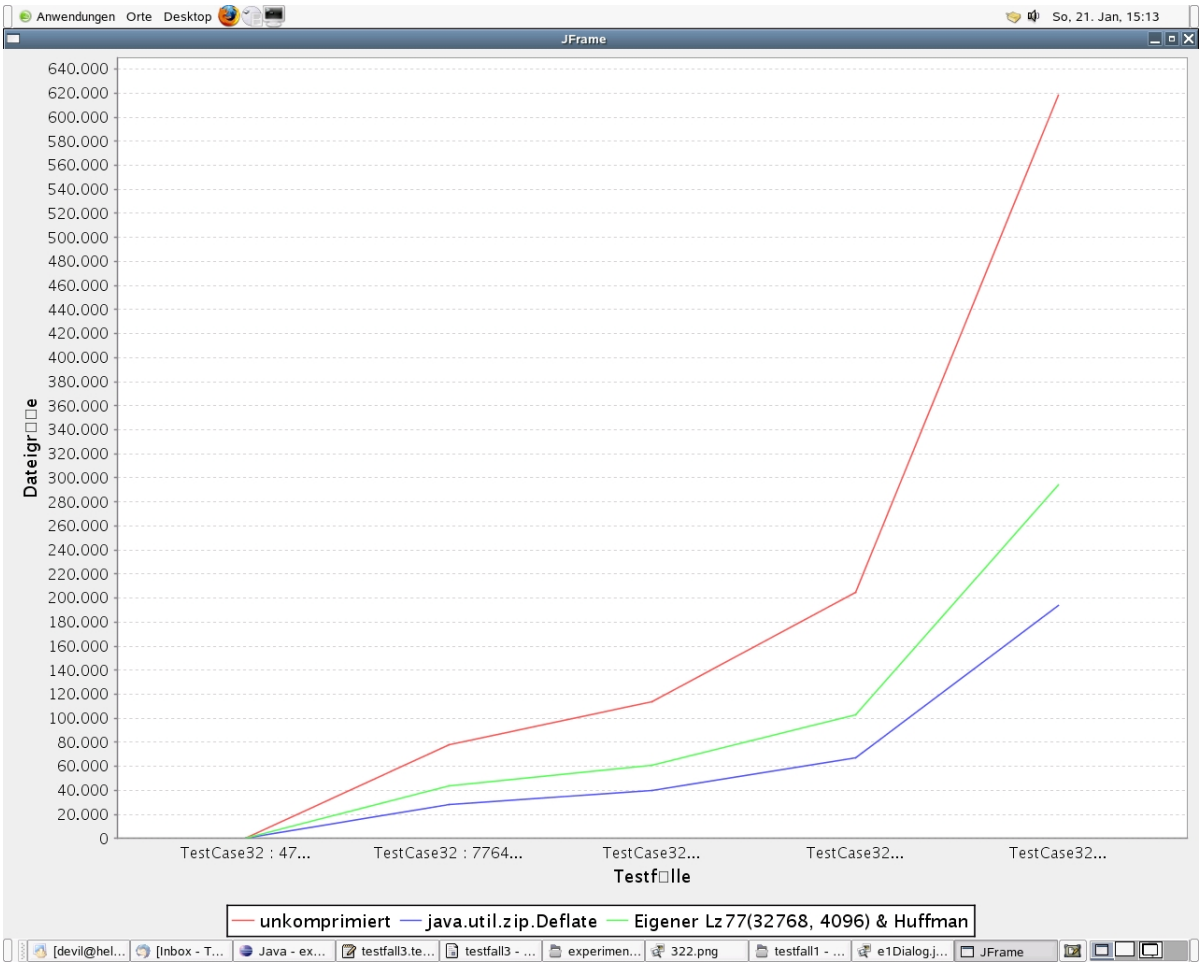

Abbildung 20: Kompression von Textdateien mit Deflate(Java-API) und LZ77 + Huffman(Java-API)

6.3.3.4 Fazit Unsere Hypothese das die Implementierung der Java Api eine bessere Komporessionsrate erzielt ist bestätigt worden. Es ist also möglich unsere Implementierung noch zu verbessern. Die Hypothese das sich die Kompressionsrate vergrößert je größer die Textdatei ist und somit mehr Redundanzen auftreten können ist auch bestätigt worden. Dies ist der Implementierung des LZ77 in dem Deflate Algorithmus zu verdanken.

6.3.4 Deflate(Java-API) und LZ77 + Huffman(Java-API)

6.3.4.1 Beschreibung des Experiments Es werden zwei Text Dateien gleicher Größe gewählt. Ein Textdatei ist ein Auszug aus Wikipedia, die zweite Datei besitzt viele Redundanzen.

6.3.4.2 Hypothese Die Kompressionsrate der Datei mit vielen Redundanzen ist höher als die Kompressionsrate der beliebigen Textdatei, da wir erwarten das der LZ77 eine höhere Kompressionrate erzielt. Insgesamt erwarten wir das der Deflater der Java API eine bessere Kompressionsrate erzieht als die Kombination unserer Implementierung mit dem Huffman-Algorithmus der Java Api.

6.3.4.3 Verifikation Es ist in Graph [Abbildung 21] zu erkennen das die Algorithmen bei der Textdatei mit vielen Redundanzen eine weit aus bessere Komprimierungsrate erzielen. In [Abbildung 22] ist zu sehen das unsere Implementierung bei einer Datei mit vielen Redundanzen durchaus mit der Implementierung der Java Api vergleichbar ist bzw. je nach dem wie die Parameter gewählt werden eine bessere Komprimierungsrate erreicht. Es wird die Konfigurationsdatei aus [Listing 10] verwendet.

Listing 10: XML Konfigurationsdatei TextDateien LZ77 und Huffman(Java-API)

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <TestSuite>
3          <Tester>deflate.CustomDeflateWrapper</Tester>
4          <Tester param1="32768"
5                  param2="4096">lz77.CustomDeflateCompressor</Tester>
6          <Tester param1="1024"
7                  param2="128">lz77.CustomDeflateCompressor</Tester>
8          <TestCase name="TestCase322a:normal"
9                  file="Text3.txt"/>
10         <TestCase name="TestCase322b:redundanz"
11                 file="redundanzen.txt"/>
12 </TestSuite>
```

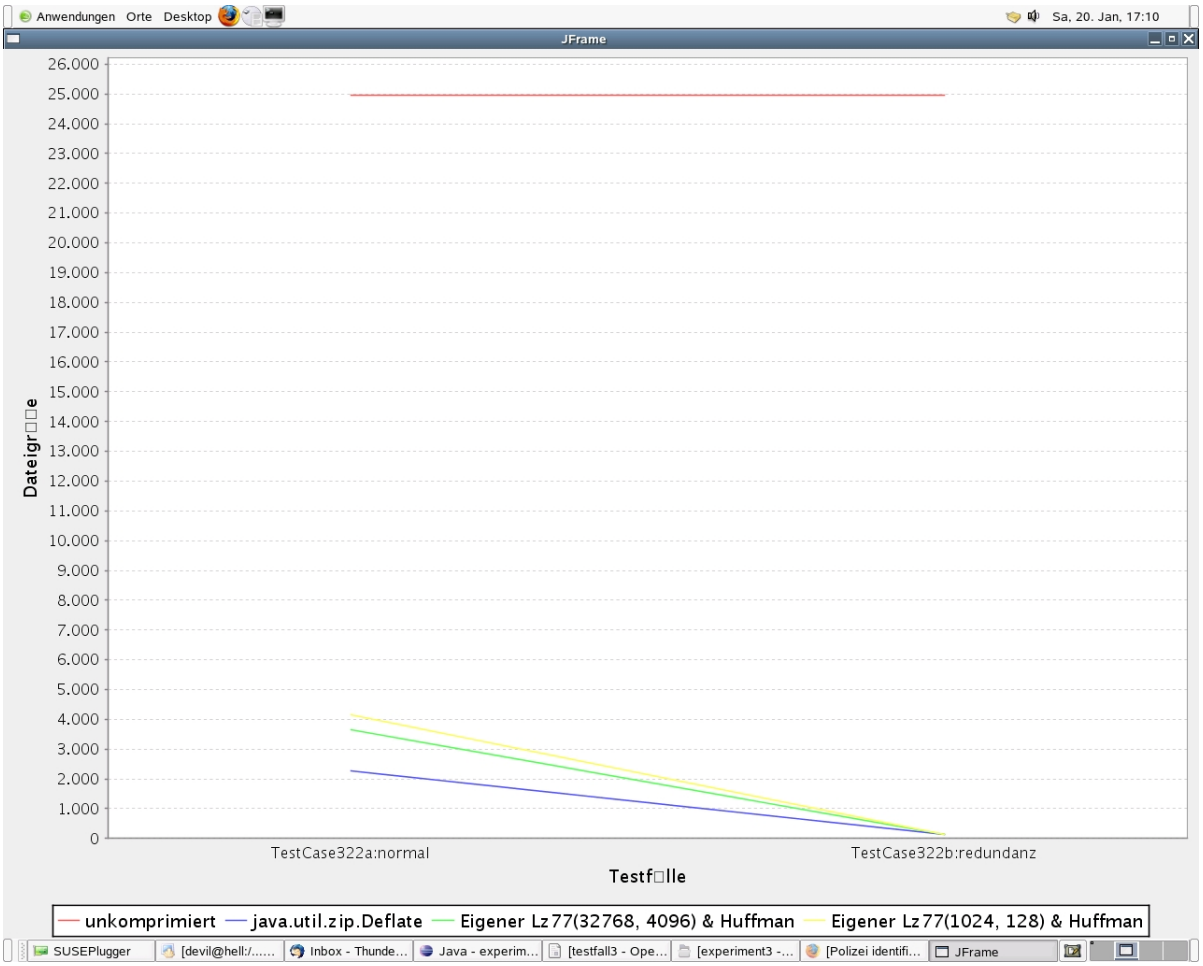

Abbildung 21: Kompression von Textdateien mit und ohne Redundanz, Deflate(Java-API) und LZ77 + Huffman(Java-API)

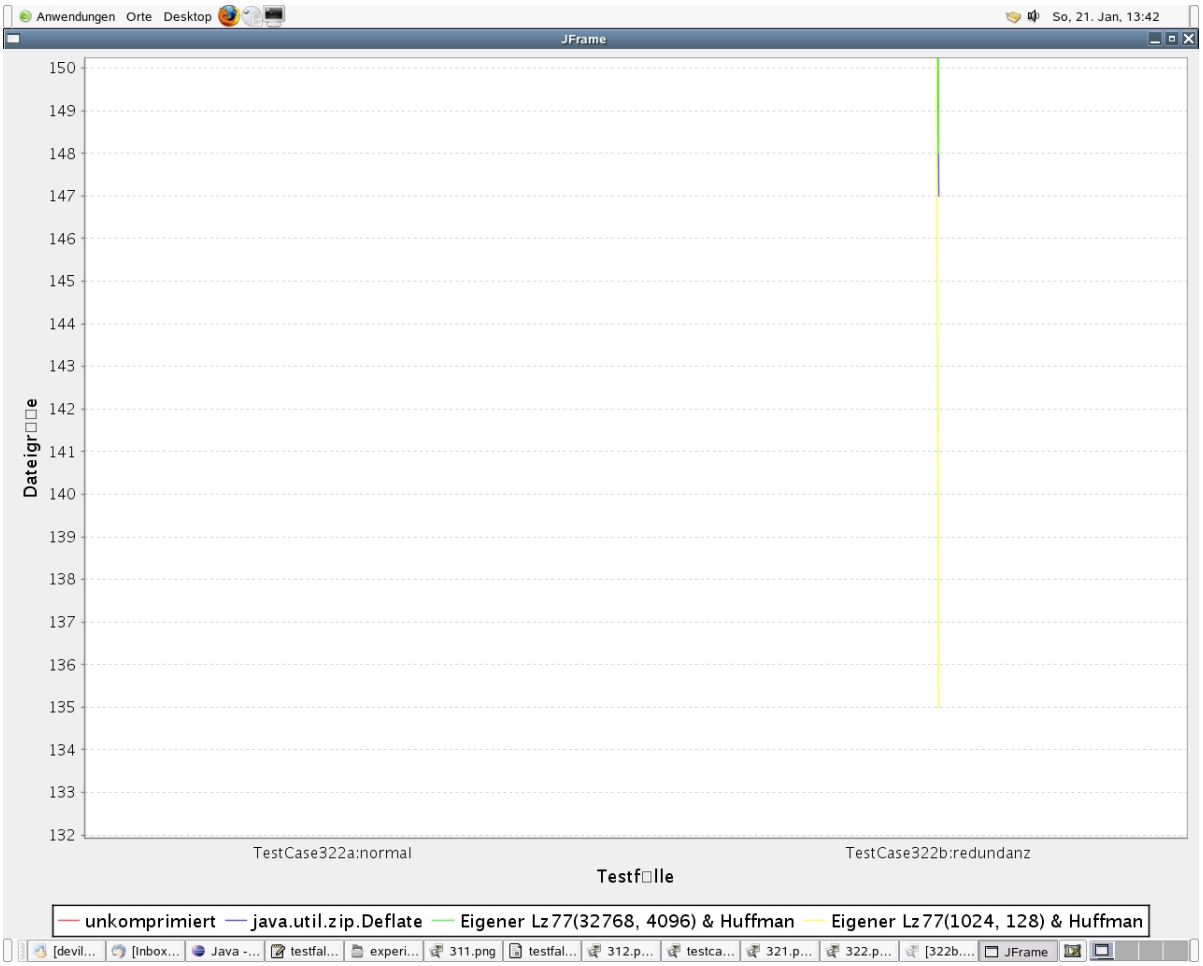

Abbildung 22: Kompression von Textdateien mit und ohne Redundanz, Deflate(Java-API) und LZ77 + Huffman(Java-API)

6.3.4.4 Fazit Unsere Hypothese das eine Datei mit vielen Redundanzen besser komprimiert werden kann ist somit bestätigt. Dies verdeutlicht das der LZ77 im Deflate einen wesentlichen Anteil an der Komprimierung insbesondere von Textdateien hat. Unsere Hypothese das der Deflate Algorithmus der Java Api durchweg ein besseres Ergbebniss erzielt, hat sich nicht bestätigt. Gerade bei Dateien mit vielen Redundanzen ist unser Algorithmus nicht unterlegen.

6.4 Fazit aller Testfälle

Bei verlustfreier Kompression wird in jedem Fall versucht, Speicherplatz möglichst ökonomisch zu nutzen. Dabei kommmen verschiedene Verfahren zum Einsatz.
So werden zum Einen vorhandene Redundanzen eleminiert, indem sie so zu sagen ausgeschnitten werden. Dies übernimmt beim Deflate-Algorithmus der Wörterbuch basierende LZ77. Redundanzen werden durch Offsets und Längenbestimmungen beseitigt.
Eine andere Möglichkeit, Daten zu komprimieren besteht in einer veränderten Darstellung der enthaltenen Zeichen oder Bytes. Dabei können Codewörter durch kürzere Codesequenzen ersetzt werden. Innerhalb des Deflate-Algorithmus übernimmt der Huffman-Algorithmus diese Aufgabe.
Das Ziel verlustfreier Kompression ist damit die Elemenierung von Redundanzen sowie die optimale Darstellung von Zeichen, im Sinne von speichersparender Darstellung.

Literatur

[1] Aladdin Enterprises, *DEFLATE Compressed Data Format Specification version 1.3*, RFC1951, May 1996

[2] Jacob Ziv, Abraham Lempel, *A Universal Algorithm for Sequential Data Compression*, IEEE Transactions On Information Theory, May 1977

[3] Deflate, *http://www.arikah.com/enzyklopadie/Deflate*

[4] Phil Katz, *http://www.arikah.com/enzyklopadie/Phil_Katz*

[5] WAV-Format, *http://www.kgw.tu-berlin.de/Studio/ProTools/audio-formate/wav/overview.html*

Tabellenverzeichnis

Abbildungsverzeichnis

Listings