

LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

**Grundlagenpraktikum: Rechnerarchitektur**

Gruppe 120 – Abgabe zu Aufgabe A402

Wintersemester 2022/23

Christian Schmucker

David Sievers

Simon Seeberger

## 1 Einleitung

Mediendateien haben historisch betrachtet eine enorme Wichtigkeit bezüglich der Popularität von Computersystemen. Mit der Entwicklung von barrierefreien Anwendung für Endnutzer ohne Informatikerfahrung wurden auch Bild-/Ton- und später Videodateiformate stetig weiterentwickelt. Dabei war ein großes Problem die Repräsentation der Daten, da effizientes kodieren von Informationen bei immer weiter wachsenden Datenmengen eine Notwendigkeit wurde.

Um jene Problematik von wachsenden Dateigrößen zu lösen, wurden viele Kompressionsalgorithmen entwickelt. Diese haben zum Ziel, dass (teilweise verlustbehaftet) die Dateigröße verringert werden kann, indem die Daten auf unterschiedliche Weise zusammengefasst werden und somit die abzusichernden Informationen verringert werden.

Das Ziel dieser Arbeit ist es, anhand des Verfahrens der *Lauf längencodierung* Bilddateien im PBM-Format zu komprimieren und dekomprimieren. Im Folgenden wird das erarbeitete Vorgehen beschrieben, zudem wird auf die Korrektheit unserer Implementierung und die Performanzanalyse von Selbiger eingegangen.

## 2 Lösungsansatz

Die Aufgabe der Implementierung lässt sich im Grunde zu drei wesentlichen Punkten zusammenfassen. Zunächst musste das Lesen und Schreiben der PBM-Dateien möglich sein, zweitens musste ein Format für die komprimierten Informationen festgelegt werden und schließlich wurden verschiedene Ansätze für die Implementierung der eigentlichen Kompression beziehungsweise Dekompression realisiert.

### 2.1 Das PBM-Bildformat

Das Portable-Bit-Map-Format, im Folgenden PBM, ist ein vergleichsweise einfaches Format zum abspeichern von Graphiken. Das Format enthält folgende Informationen:

1. **Eine Magic Number:** Zur Abgrenzung von anderen Formaten mit gleicher Dateierweiterung werden stets die beiden Zeichen „P4“ gesetzt.
2. **Whitespace**
3. **Die Länge des Pixelarrays:** Repräsentiert als ASCII Zeichen in Dezimaldarstellung.

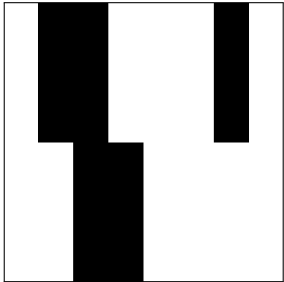
---

4. **Whitespace**
5. **Die Höhe des Pixelarrays:** Repräsentiert als ASCII Zeichen in Dezimaldarstellung.
6. **Ein einzelner Whitespace-Character**
7. **Ein Array von von horizontalen Pixelreihen:** Jede Reihe ist so lang wie vorher mit der Länge spezifiziert. Falls die Bildbreite modulo 8  $\neq$  0 ist, werden die übrigen Bits eines Bytes mit Nullen aufgefüllt. Die Pixel werden von links nach rechts gespeichert. Ein Bit repräsentiert einen schwarzen Pixel, falls 1 gesetzt ist, ansonsten ein weißes Pixel.

Die Spezifikationen wurden [1] entnommen. Die Werte für eine Reihe sind als ASCII-Character kodiert.

## 2.2 Das verwendete Datenformat

Um die komprimierten Pixelinformationen abzuspeichern, findet folgendes Format Verwendung: Zunächst werden die Höhe, anschließend die Breite des ursprünglichen Bildes gespeichert. Es folgt die Größe des enkodierten Abschnittes, und schließlich wird Selbiger abgespeichert. Die Datei ist ungültig, wenn die Größe des kodierten Bildes nicht mit der geschriebenen Datenmenge übereinstimmt.

Unkomprimiert	Bild	Komprimiert
P4 8 8 bbbb0000		00000000: 0800 0000 0000 0000 0800 0000 0000 0000 00000010: 1900 0000 0000 0000 0182 0381 0282 0381 00000020: 0282 0381 0282 0381 0382 0682 0682 0682 00000030: 04

Das unkomprimierte Format verwendet ASCII-Zeichen für die Repräsentation einer Reihe. Da der Kleinbuchstabe b der 98<sub>10</sub>ste Character im ASCII-Standard ist, wird dieser binär 01100010<sub>2</sub> kodiert. Daraus ergeben sich die im Bild darstellten oberen 4 Reihen. Die Ziffer 0 hat in ASCII binär den Code 00110000<sub>2</sub>, entsprechend ergeben sich die weiteren 4 Reihen. Die Zahl wird von links nach rechts gelesen, wobei die 0 einem weißem, die 1 einem schwarzem Pixel entspricht.

Im komprimierten Format stellen also, wie in der Tabelle dargestellt, die ersten 4 Byte die Breite des Bildes, die nächsten 4 Byte die Höhe, weitere 4 Byte die Länge des kodierten Teils und schließlich die restlichen Daten die Pixelinformationen dar.

Der eigentliche Datenteil weicht bei den Implementierungen ab. Das bedeutet insbesondere, dass die Huffman-Kodierung nicht von den beiden anderen Versionen dekodiert werden kann, beziehungsweise die Huffman-Version auch die komprimierten Daten der anderen Versionen nicht dekodieren kann.

Der in der Tabelle dargestellte Hexdump wurde von Version 0, also unserer Standardimplementierung, ausgegeben. Im Folgenden Kapitel werden die 3 Versionen besprochen und verglichen, dabei wird auch auf die Informationsrepräsentation eingegangen werden.

## 2.3 Die verwendeten Kompressionsalgorithmen

Im Zuge dieser Arbeit wurden drei unterschiedliche Implementierungen für das Lauf­längenkodierungsproblem angefertigt. Die Standardimplementierung und die zweite Implementierung sind von der Arbeitsweise gleich, V1 verwendet im Vergleich zu V0 allerdings Multithreading. Der dritte Ansatz in V2 ist anders aufgebaut und verwendet Huffmancodes. Im Folgenden werden die drei Ansätze vorgestellt.

### 2.3.1 Die Standardimplementierung (V0)

Die Hauptimplementierung ist ein sequenzieller, single-threaded Ansatz. Das Datenarray wird Byteweise betrachtet. Für die enkodierte Speicherung wird zunächst ein Farbbit gesetzt, anschließend folgen mehrere Bits mit der Anzahl der gleichfarbigen Bits. Wieviele Bits für die Abspeicherung der Anzahl zur Verfügung stehen, kann mit der Variable *LENGTH\_BIT\_SIZE* festgelegt werden, welche standardmäßig auf 7 gesetzt ist. Innerhalb eines Bytes wird bitweise geprüft, ob sich eine Änderung der Pixelfarbe ereignet, ansonsten wird eine Variable mit der Anzahl der gleichfarbigen bits inkrementiert, solange diese nicht den Maximalwert von *LENGTH\_BIT\_SIZE* überschreitet. Wenn ein neues Byte betrachtet wird, und dieses den Wert  $00_{16}$  bzw.  $ff_{16}$  hat, wird ohne Betrachtung der Bits einfach der Counter um 8 erhöht. Sobald der Farbwert wechselt, wird für die vorangegangenen gleichfarbigen Bits Farbe und Anzahl geschrieben. Die maximale Länge der kodierten Daten haben wir bei V0 und V1 auf  $LENGTH\_BIT\_SIZE \cdot bits\_to\_encode$  geschätzt. Dies stellt den Fall dar in welchem die Farbe mit jedem Bit wechselt.

### 2.3.2 Implementierung mithilfe von Threads (V1)

Die V1 ist eine Modifizierung der Standardimplementierung. Grundsätzlich werden die Funktionen von V0 verwendet, allerdings wird hier nicht mehr nur mit einem, sondern mit mehreren Threads gearbeitet.

Sei  $n$  die Anzahl der vom Programm benutzten Threads. Diese wurde in der Implementierung statisch auf 4 festgelegt. Grundsätzlich sei jedoch angemerkt, dass die Wahl eines optimalen Wertes vor allem von der Anzahl tatsächlicher Prozessorkerne sowie von der Bildgröße der zu komprimierenden Datei abhängt.

---

Das übergebene Daten-Array wird für die Enkodierung in ebenfalls  $n$  Abschnitte unterteilt. Jeder Thread übernimmt demnach  $\frac{1}{n}$  der ursprünglichen Datenmenge. Das eigentliche Enkodieren erfolgt mit der entsprechenden Funktion aus V0. Zunächst wird ein eigenes Outputarray je Thread beschrieben. Zuletzt werden jene Teilarrays zu einem großen Array zusammengefasst und die Daten geschrieben.

Beim Dekodieren ist das grundsätzliche Vorgehen wie beim Enkodieren. Hier ist notwendig, dass `LENGTH_BIT_SIZE = 7` gilt, das entspricht 1 Bit Pixelfarbe, 7 Bit Länge, weshalb bei abweichender Enkodierung mit anderem Wert die Dekodierung mit V1 nicht funktioniert.

Für die Konkatenation der nun wieder dekodierten Teilarrays muss allerdings Bitshifting betrieben werden, was die Dekodierung deutlich ineffizienter macht als bei V0.

### 2.3.3 Implementierung mit Huffmancodes (V2)

Für die dritte Implementierung wurde der Huffman Coding Algorithm [3, S. 13f.] von David Huffman verwendet. Dieser funktioniert wie folgt:

- Jedes Element ist zunächst ein eigener Baum aus einem einzigen Knoten bestehend. Ein solcher hat das Startgewicht  $w_i = p_i$ .
- Wiederhole, bis nur noch ein einziger Baum übrig ist:
  - Wähle die zwei Bäume mit den niedrigsten Gewichten  $w_1$  und  $w_2$ .
  - Kombiniere beide Bäume zu einem neuen Baum. Die Wurzel hat das Gewicht  $w_1 + w_2$ . Die beiden ursprünglichen Bäume werden die beiden Kinder des neuen Baums. Falls  $w_1 \neq w_2$ , so sei das linke Kind dasjenige mit dem geringeren Gewicht.

Der Algorithmus wird in der Abgabe zu Datenkomprimierung verwendet. Zunächst werden alle Pixel durchlaufen und die Häufigkeit der Länge von Pixeln wird temporär abgespeichert. Der Huffman-Baum wird aus der Länge der Pixel aufgebaut, wobei jeder Baum des Waldes am Anfang als Gewicht je die Häufigkeit des Auftretens der Länge bekommt. Wie im Huffman Coding Algorithm beschrieben, werden nun Paare mit den niedrigsten Häufigkeiten zu einem neuen Baum zusammengefügt. Das wird mit den neuen Bäumen wiederholt, bis der Huffmanbaum fertig ist.

Um den Baum zu durchlaufen, wird eine rekursive Tiefensuche verwendet, welche Knoten für Knoten abarbeitet. Sobald ein Blatt gefunden wurde, wird für den in dem Blatt gespeicherten Längenwert der zugehörige Huffman-Code in einer Tabelle abgespeichert.

Danach wird der generierte Huffman-Baum in eine Bitstream-Repräsentation umgewandelt und am Anfang der Ausgabedatei abgespeichert. Der aus dem Huffman-Baum generierte Bitstream lässt sich wie folgt beschreiben:

- Es wird mit der Wurzel des Baumes gestartet.
-

- Ein Bit gibt an, welcher Knotentyp kodiert ist. Ist das Bit gesetzt, handelt es sich um einen inneren Knoten, andernfalls um ein Blatt.
- Ist der Knoten ein innerer Knoten, folgt zunächst die kodierte Repräsentation vom ersten, dann vom zweiten Kinderknoten.
- Ist der Knoten ein Blatt, folgt eine vordefinierte Anzahl an Bits, welche den zum Blatt gehörigen Längenwert kodieren.

Es folgen, wie auch in V0 und V1, die Tupel des Run-Length-Encodings, nur mit dem Unterschied, dass hier die Länge statt mit einer festen Anzahl an Bits mit einem Huffman-Code variabler Länge angegeben wird. Dieser kann dann beim Dekodieren mithilfe des am Anfang des Formates abgespeicherten Huffman-Baums wieder in die tatsächliche Länge umgewandelt werden. Die maximale Länge der kodierten Daten haben wir als  $(tree\_inner\_nodes\_amount + (1 + HUFFMAN\_LENGTH\_BIT\_SIZE) \cdot tree\_leaves\_amount) + (bits\_to\_encode \cdot (max\_huffman\_code\_length + 1))$  geschätzt. Hierbei ist:

- $bits\_to\_encode$  die Anzahl der zu kodierenden Bits
- $HUFFMAN\_LENGTH\_BIT\_SIZE$  die Größe der im Huffman-Baum abgespeicherten Längen in Bits
- $tree\_leaves\_amount = 2^{HUFFMAN\_LENGTH\_BIT\_SIZE}$
- $tree\_inner\_nodes\_amount = tree\_leaves\_amount - 1$
- $max\_huffman\_code\_length = tree\_inner\_nodes\_amount$

### 3 Korrektheit

Um die Korrektheit der Implementierung zu gewährleisten, haben wir mit einem automatischen Testprogramm ("Fuzzer") unsere Programmkomponenten untersucht. Auch wenn die hier verwendete Methode des Grey-Box Fuzzings im Gegensatz zu formeller Verifikation nicht die Abwesenheit von ungewolltem/inkorrektem Verhalten beweisen kann, hat sie sich in der Vergangenheit als effiziente und schnell zu realisierende Methode zur Untersuchung von Programmverhalten auf einer großen Menge an möglichen Eingaben herausgestellt. Es ist also, wenn wir bei der Konzeption unserer Fuzzer keine falschen Annahmen gemacht haben, relativ unwahrscheinlich, dass eine Eingabe existiert, welche beispielsweise zu einer inkorrekten Kodierung/Dekodierung oder einem ungültigen Speicherzugriff führt.

### 4 Performanzanalyse

Im Folgenden werden die Benchmarkergebnisse der verschiedenen Implementierungen für Kompression und Dekompression verglichen und ausgewertet. Alle Berechnungen

---

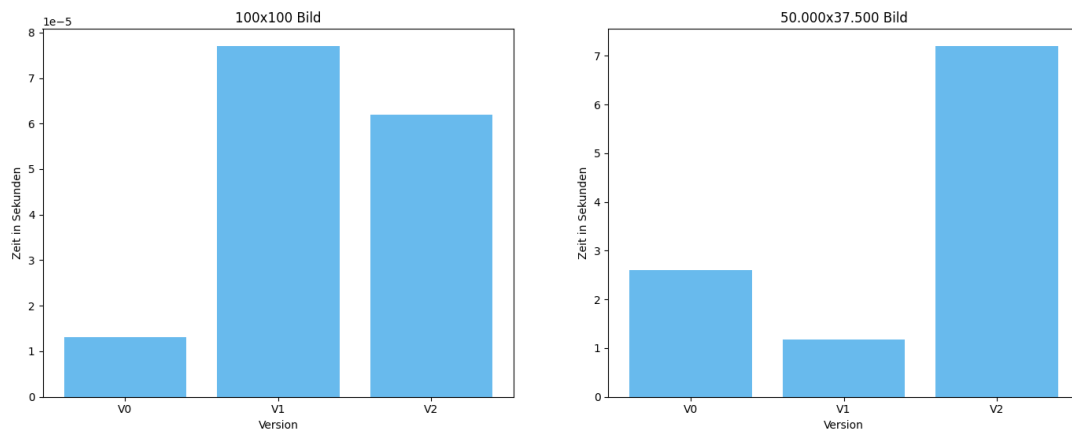


Abbildung 1: Kompression: Performanz unterschiedlicher Versionen.

wurden auf einem System mit 16 logischen Kernen durchgeführt. Kompiliert wurde mit Optimierungsstufe -O2. Der Parameter LENGTH\_BIT\_SIZE wurde darüber hinaus bei allen Tests auf 7 Bit festgelegt und die Anzahl an Threads bei der Multithreading-Variante (V1) wurde auf 4 gesetzt.

Abbildung 1 stellt die Kompressionsdauer der unterschiedlichen Versionen gegenüber. Im linken Diagramm wurde ein kleines Bild der Dimension 100 x 100 Pixel komprimiert. Die große Differenz zwischen V0 und den anderen Versionen ist hier vor allem bedingt durch den hohen initialen Overhead von V1 und V2. Die Erzeugung von Threads, das Allokieren einzelner Arrays für jeden Thread und die anschließende Konkatenation der Daten in das Ausgabe Array beim Multithreadingansatz erhöht die Ausführungszeit, sodass diese Version für kleine Bilddateien im Gegensatz zur sequenziellen Verarbeitung nicht empfehlenswert ist. Auch die Serialisierung der Baum-Struktur bei der Huffman-Enkodierung benötigt mehr Rechenzeit. Dennoch befindet sich alles noch in einem Bereich von 0,1 bis 0,8 Millisekunden.

Das Diagramm auf der rechten Seite von Abb. 1 liefert ein anderes Ergebnis. Der Zeitunterschied zwischen den Varianten ist hier deutlich bemerkbar. Für die Komprimierung der Testdatei von 50.000 x 37.500 Pixel benötigt die Multithreadingvariante nur ca. 1,18 Sekunden. Dies ist mehr als doppelt so schnell wie V0 mit 2,60s und knapp siebenmal so schnell wie die Huffman-Variante mit 7,19s.

In Abbildung 2 zeigt sich, ab welcher Bildgröße sich die Performanzunterschiede der unterschiedlichen Versionen voneinander absetzen. Bereits für eine Bildgröße von 400 x 400 Pixel benötigt die Huffman-Enkodierung mehr als doppelt so viel Rechenzeit wie die anderen Varianten. Außerdem sieht man, dass bei einer Bildgröße von 1920 x 1080 Pixel die Laufzeit von V0 und V1 fast identisch ist. Würde man eine noch größere Datei komprimieren, wäre folglich V1 die schnellste Variante.

Bei der Dekomprimierung ändern sich die Geschwindigkeitsverhältnisse der Versionen im Vergleich zur Komprimierung. Hier zeigt sich deutlich, dass unsere Standardim-

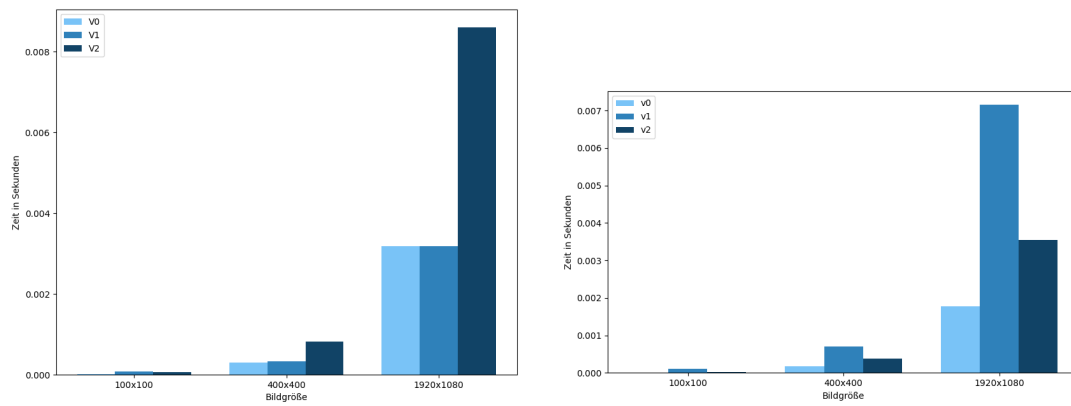


Abbildung 2: Performanz der unterschiedlichen Varianten: Kompression (links) und Dekompression (rechts)

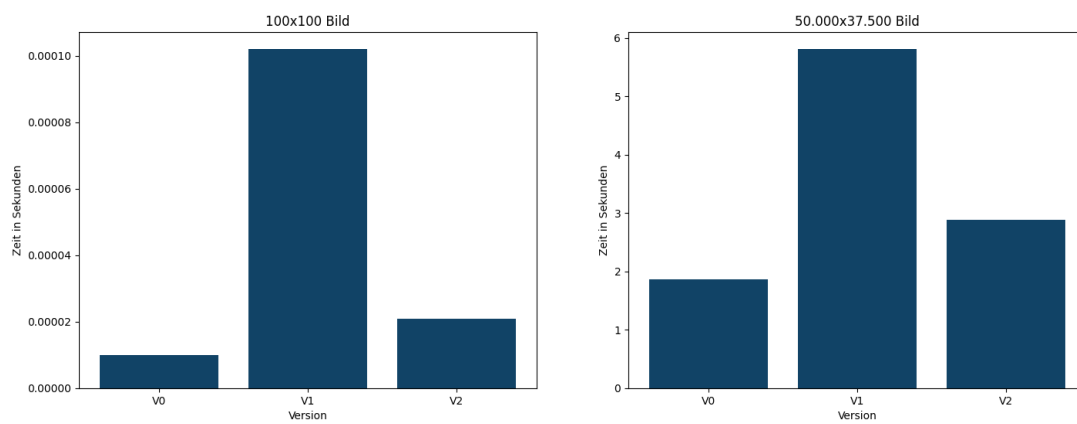


Abbildung 3: Dekompression: Performanz unterschiedlicher Versionen

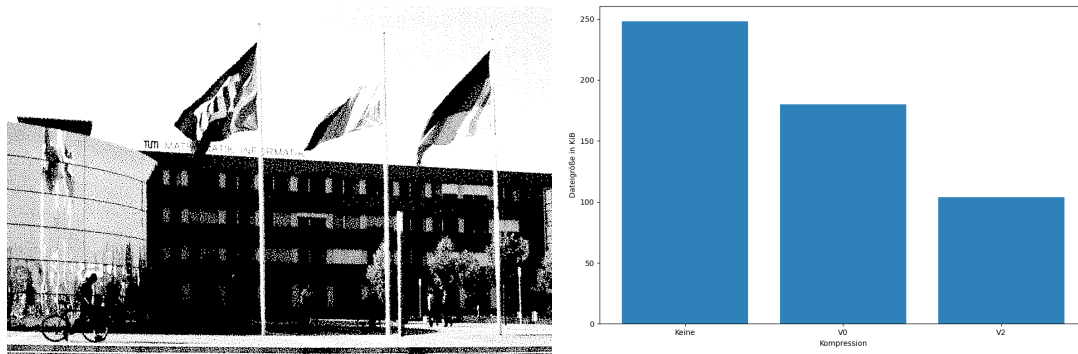


Abbildung 4: Dateigröße vor und nach Kompression (Bild [2]:  $1920 \times 1080$  px)

plementierung schneller als die anderen beiden ist und V1 wiederum langsamer. Der Zeitverlust in der Version mit Threads lässt sich auf das aufwendige, mehrmalige Bitshifting zurückführen, welches benötigt wird, um die Teilarrays der jeweiligen Threads am Ende zu konkatenieren.

Wie in Abbildungen 2 und 3 zu sehen ist, sind die Unterschiede in der Laufzeit bei kleinen wie großen Bilddateien etwas kleiner, die Reihenfolge der Versionen bleibt untereinander jedoch erhalten.

Abschließend stellte sich jedoch nun die Frage, wie erfolgreich die Run-Length-Enkodierung tatsächlich ist, speziell die Variante mit der Huffman-Enkodierung, welche mit Abstand am meisten Rechendauer benötigt. Abbildung 4 zeigt die unterschiedlichen Dateigrößen einer  $1920 \times 1080$  Pixel großen PBM Datei vor und nach der Kompression. Im unkomprimierten Format ist die Datei 248 KiB groß, nach der Kompression mit V0 lediglich nur noch 180 KiB und V2 reduziert sie sogar mit der Huffmann Kodierung auf 104 KiB. V1 benötigt ca. gleich viel Byte wie V0, im schlechtesten Fall jedoch Anzahl-an-Threads-Byte mehr, falls das letzte Bit, das jeder Thread enkodiert mitten in einer Folge zusammenhängender Nullen oder Einsen endet. Dies ist jedoch vernachlässigbar. Im Allgemeinen verringert die Run-Length-Enkodierung jedoch nicht zwangsläufig die Dateigröße. Bei unserer Implementierung benötigt die Enkodierung der Pixelwerte im schlimmsten Fall  $length \cdot height \cdot (1 + LENGTH\_BIT\_SIZE)$  Bit. Dies ist der Fall, wenn wir eine alternierende Folge von Pixelwerten haben (1010101...).

Bei normalen Bildern hingegen, wie das in Abb. 4, mit längerer Folgen gleicher Pixelwerte, schafft es die Run-Length-Enkodierung die Dateigröße deutlich zu komprimieren. Besonders die Huffman-Variante (V2) kann große Bilddateien noch besser komprimieren, sodass sich die längere Laufzeit unter Umständen durchaus lohnt.

## 5 Zusammenfassung und Ausblick

Das Projekt hat gezeigt, dass es unterschiedliche Ansätze gibt, wie man Lauflängencodierung umsetzen kann. Nach einer Einarbeitung in das pbm-Dateiformat und in die grundsätzliche Funktionsweise hinter dem Kodierungsalgorithmus wurden 3 Versionen



implementiert, welche es uns ermöglicht haben, die unterschiedlichen Ansätze gegenüberzustellen. Die Performance von einem Kompressionsalgorithmus ist bei seltener Nutzung nicht besonders relevant, aber macht einen großen Unterschied, sobald beispielsweise auf einer Webseite sehr viele Daten auf einmal verarbeitet werden müssen.

Die Optimierung kann nicht nur hinsichtlich weniger Laufzeit, sondern auch im Bezug auf die abzusichernde Dateigröße vorgenommen werden. Die Huffman-Version (V2) benötigt mehr Zeit als die Standardimplementierung (V0), allerdings zeigt sich auch eine deutliche Verbesserung in der Dateigröße, somit ist die Kompression erfolgreicher. Eine Erkenntnis ist auch, dass unterschiedliche Verfahren für das Dekodieren und Enkodieren verwendbar sind, solange die Datenrepräsentation im Hintergrund die Selbe bleibt. Somit kann ein Algorithmus mit guter Enkodierzeit mit einem mit guter Dekodierzeit kombiniert werden.

Auch im Punkt Korrektheit scheint unser Programm richtig zu funktionieren. Nach unseren eigenen Tests sind die Dateien, welche zunächst komprimiert und anschließend wieder dekomprimiert wurden, identisch.

Das Projekt war für sich erfolgreich, es wäre aber durchaus möglich, weitere Verbesserungen vorzunehmen. Von einer Implementierung mit SIMD-Befehlen wurde nach einigen Versuchen abgesehen. Da sequenziell die Bits eingelesen werden müssen und SIMD-Instruktionen eher auf Bytes ausgelegt sind, hätte aufwendig mit Bitshifts gearbeitet werden müssen, wodurch eine Performanceverschlechterung das Ergebnis gewesen wäre. Deshalb wurde dieses Vorgehen wieder verworfen.

Eine Adaption des Vorgehens mit Farbwerten, welche unser pbm-Format nicht unterstützt, aber beispielsweise in den verwandten ppm-Dateien Anwendung finden, ist auch denkbar. Hier müssten nur die Farbwerte als Zahl statt unserem einem Bit für Schwarz oder Weiß ersetzt werden (und natürlich das Einlesen und Ausgeben der Datei leicht verändert werden).

Abschließend lässt sich sagen, dass das verwendete Verfahren natürlich eher für ältere Bildformate anwendbar ist. Modernere Dateien sind oft von sich aus schon stark komprimiert und das Feld der Datenkompression findet in der Informatik immer wieder neue Wege, um Daten auf ein Minimum reduzieren. Der Algorithmus ist aber natürlich nicht nur auf Bilder anwendbar, entsprechend sind auch andere Anwendungen denkbar, solange Daten im Spiel sind, welche sich in irgendeiner Form wahrscheinlich wiederholen.

## Literatur

- [1] pbm. <https://netpbm.sourceforge.net/doc/pbm.html>, 2013. Zuletzt geöffnet am 02. Feb. 2023.
  - [2] Haupteingang gebäude der fakultät für informatik. [https://www.in.tum.de/fileadmin/\\_processed\\_/5/5/csm\\_2006\\_1015Bild0136\\_4386718267.jpg](https://www.in.tum.de/fileadmin/_processed_/5/5/csm_2006_1015Bild0136_4386718267.jpg), 2022. Zuletzt geöffnet am 05. Feb. 2023.
  - [3] Guy E. Blelloch. Introduction to data compression. <http://www.cs.cmu.edu/afs/>
-

[cs/project/pscico-guyb/realworld/www/compression.pdf](https://www.cs.cmu.edu/project/pscico-guyb/realworld/www/compression.pdf), Carnegie Mellon University. Computer Science Department., 2013.