Karlsruher Institut für Technologie

Bachelorarbeit

Kompressionstechniken für Beschreibungen von SAT Formeln

Jens Manig

Abgabedatum: 28.3.2018

Betreuer: Prof. Dr. Peter Sanders
M.Sc. Dominik Schreiber

Institut für Theoretische Informatik, Algorithmik
Fakultät für Informatik
Karlsruher Institut für Technologie

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Ort, den Datum

Zusammenfassung

In dieser Arbeit werden verschiedene Ansätze für ein Kompressionsverfahren des DIMACS-Dateiformat vorgestellt. Das DIMACS-Dateiformat findet beim SAT-Solving Verwendung und enthält Logik-Formeln, welche als Zahlen repräsentiert, aber als Strings gespeichert werden. Daher beschäftigt sich die vorliegende Arbeit damit, Vorwissen über dieses Format für eine verbesserte Kompression zu verwenden. Der erste einwickelte Ansatz ist eine Umwandlung der Strings in Zahlen mit einer festen Anzahl Bytes pro Zahl, der zweite Ansatz extrahiert die Vorzeichen in einem Bitvektor und nutzt eine variable Byteanzahl pro Zahl, der dritte nutzt einen Move-to-Front Ansatz, der vierte versucht eine Kompression über Präfixmatching der Zeilen zu erreichen und der letzte Ansatz ist eine Kombination aus Move-to-Front und Präfix. Von den vorgestellten Ansätzen ist die Kombination der beste bezüglich des Kompressionsfaktors. Bei der Laufzeit ähneln sich die entwickelten Ansätze stark, sowohl in der Kodierung als auch Dekodierung. Im Vergleich zu ZIP ist die Kodierungszeit bedingt besser, während die Dekodierungszeiten von ZIP noch deutlich geringer sind. Beim Kompressionsfaktor ist die Kombination nicht weit von ZIP entfernt, und eine Kombination aus ZIP und den neuartigen Verfahren führt zu den insgesamt besten Kompressionsraten.

Abstract

In this thesis we present different approaches for a copmression of the DIMACS file format. The DIMACS file format is used for SAT solving and contains logic formulas that are represented as numbers but stored as strings. Therefore, this paper is engaged with exploiting prior knowledge about the file format when compressing it. The first presented approach is a conversion of the strings into numbers with a fixed number of bytes for each number, the second approach extracts the number's signs into a bit vector and uses a variable number of bytes per number. The third approach uses a move-to-front approach, the fourth achieves additional compression by prefix matching the rows and the last approach is a combination of move-to-front and prefixmatching. Beneath the presented approaches, the combination performs best in terms of compression factor. In terms of runtime, the presented approaches are very similiar, both in terms of encoding and decoding. Compared to ZIP, the encoding time is in some cases better, while the decoding times of ZIP are significantly shorter. The compression factor of the combined approach is similar to ZIP's and re-compressing the file resulting from the combined approach leads to the best overall compression rates.

Inhaltsverzeichnis

1 Einleitung **7**
1.1 Motivation . . . 7
1.2 Beiträge der Bachelorarbeit . . . 7
1.3 Überblick . . . 8

2 Verwandte Arbeiten **9**
2.1 SAT-Solving . . . 9
2.2 Kompressionsverfahren . . . 9
2.3 Lempel-Ziv . . . 11
2.4 Move-to-Front . . . 11
2.5 Variable-Length Quantity . . . 12

3 Voruntersuchung **13**
3.1 Datenerhebung und Auswertung . . . 13
3.2 Diskussion . . . 13

4 Entwurf **19**
4.1 Basis-Algorithmus . . . 19
4.2 Variable-Length Quantity . . . 20
4.3 Move-to-Front . . . 20
4.4 Präfix . . . 21
4.5 Kombination . . . 22

5 Implementierung **23**
5.1 Basis-Algorithmus . . . 23
5.2 VLQ . . . 23
5.3 Move-to-Front . . . 24
5.4 Präfix . . . 24
5.5 Kombination . . . 24

6 Evaluation **25**
6.1 Kompressionsrate . . . 25
6.2 Laufzeit der Algorithmen . . . 30
6.3 Move-to-Front Listenlängen . . . 31
6.4 Doppelte Kodierung . . . 33

7 Schlussfolgerungen **37**
7.1 Zusammenfassung . . . 37
7.2 Ausblick . . . 38

Abbildungsverzeichnis

1 Histogramm über die Zahlen . . . 13
2 Histogramm über die Beträge . . . 14
3 Histogramm über die Klausellängen . . . 15
4 Differenz der Zahlen . . . 16
5 Differenz innerhalb einer Klausel . . . 16
6 Beispieldatei . . . 19
7 Beispieldatei Basis-Algorithmus . . . 19
8 Beispieldatei VLQ . . . 20
9 Beispieldatei Move-to-Front . . . 21
10 Beispieldatei Präfix . . . 22
11 Beispieldatei Kombination . . . 22
12 Kompressionsrate Basis VLQ . . . 25
13 Kompressionsrate Move-to-Front Basis VLQ . . . 26
14 Kompressionsrate Präfix Basis VLQ . . . 26
15 Kompressionsrate Präfix Move-to-Front . . . 27
16 Kompressionsrate Präfix Move-to-Front Kombination . . . 28
17 Kompressionsrate Vergleich Kombination ZIP . . . 29
18 Kompressionsraten aller Algorithmen . . . 29
19 Laufzeit Kodierung Kombination ZIP . . . 30
20 Alle Kodierungszeiten . . . 31
21 Alle Dekodierungszeiten . . . 31
22 Kompressionsraten Move-To-Front, verschiedene Listenlängen . . . 32
23 Laufzeit Move-To-Front, verschiedene Listenlängen . . . 33
24 Doppel Kodierung Kompressionsraten . . . 34
25 Laufzeit zweite Kodierung . . . 35

Tabellenverzeichnis

1 Symbole nach Häufigkeit geordnet und ihre Kodierung . . . 10
2 Symbole und eine mögliche Huffmankodierung derselben . . . 10
3 Simulation des Speicherverbrauchs der Kodierungen . . . 17
4 Kompressionsraten Durchschnitt Median . . . 28
5 Listenlänge und dazugehörige durchschnittliche Kompressionsrate, Kodierungszeit und Dekodierungszeit . . . 32
6 Doppel Kodierung Durchschnitt Median . . . 34

1 Einleitung

Im folgendem wird die Motivation dieser Arbeit dargelegt und die grundlegende Problemstellung diskutiert.

1.1 Motivation

In vielen Anwendungsbereichen lassen sich anspruchsvolle Probleme in Aussagenlogik überführen. Beispiele hierfür sind Planungsprobleme, Verifikation, Theorem-Beweise, Scheduling oder auch Kryptographie.
Aussagenlogik bedeutet, dass eine Aussage in Elementaraussagen zerteilt wird, diese haben einen Wahrheitswert „wahr“ oder „falsch“. Diese Elementaraussagen werden mit Junktoren, und/oder/nicht, verknüpft, dadurch kann der Wahrheitswert der Aussage aus den Elementaraussagen bestimmt werden. Eine Menge dieser logischen Elementaraussagen wird als (SAT-) Formel oder SAT- Instanz bezeichnet. SAT bezieht sich auf Boolean Satisfiability Problem, zu deutsch Erfüllbarkeitsproblem der Aussagenlogik, welches feststellt, ob eine gegebene Formel erfüllbar ist. SAT ist NP-vollständig und daher lassen sich viele Probleme auch in SAT darstellen. SAT-Instanzen können mit sogenannten SAT-Solvern automatisch gelöst werden. Dieser Vorgang wird auch SAT-Solving genannt.
SAT-Instanzen liegen meistens im DIMACS-CNF-Format[6] vor. Eine Zeile beschreibt eine einzelne Klausel, beispielsweise „1 -2 0“ repräsentiert die logische Klausel $x_1 \vee \neg x_2$, wobei die „0“ als Trennzeichen dient. Zeilen dieser Art werden in einer normalen Textdatei gespeichert und mit der Endung „.cnf“ gekennzeichnet.

SAT-Solving kann auch auf großen Parallelrechnern, beispielsweise HPC-Systemen, ausgeführt werden. Für die Verteilung und Kommunikation der einzelnen Recheneinheiten werden die Aufträge über das Netzwerk verschickt. In einem HPC-System sollen so wenig Daten wie möglich das Netzwerk verstopfen, deshalb werden die Daten komprimiert. Für SAT-Solving bedeutet dies, dass Formel über das Netzwerk verschickt werden: wie oft, hängt von der SAT-Instanz ab. Es ist deshalb von Vorteil ein gutes Kompressionsverfahren zu verwenden, welche die zu verschickende Datenmenge reduziert. Ebenfalls sollte es ein schnelles Kompressionsverfahren sein, damit die Zeit minimiert wird, die mit Komprimieren und Dekomprimieren verbracht wird. Bisher wird dafür kein spezifisches Kompressionsverfahren verwendet, sondern stattdessen wird herkömmliche generische Datenkompression verwendet.

1.2 Beiträge der Bachelorarbeit

In dieser Arbeit soll untersucht werden, ob ein verbessertes Kompressionsverfahren für das DIMACS-Format möglich ist. Das Kompressionsverfahren sollte mit gängigen Verfahren vergleichbar sein. Es soll ein speziellerer Ansatz für die Kompression genutzt werden, als bei gängigen Verfahren: Das Wissen über das Dateiformat und den Aufbau der Datei bei der Kompression Verwendung finden. Gängige Kompressionsverfahren nutzen kein vorheriges Wissen über den Aufbau der Dateien, können dafür aber alle Dateien komprimieren. Im Umkehrschluss können mit den entwickelten Verfahren nur DIMACS Dateien komprimiert werden. Das Verfahren wird im Anschluss mit ZIP verglichen werden.
Als erster Algorithmus wird eine Umwandlung der Textdateien in ein Binärformat mit fester Länge für eine Zahl vorgestellt. Daruaf folgt ein Algorithmus mit variabler Bytelänge und separierten Negationen. Als nächstes kommt ein Move-to-Frontalgorithmus und ein Algorithmus der Präfixe der Zeilen auf Übereinstimmungen prüft, vorgestellt. Der letzte Algorithmus kombiniert

den Move-to-Front-Algorithmus und den Präfix-Algorithmus. Des weiteren wird auch eine doppelte Komprimierung getestet, welche zuerst die entwickelten Algorithmen und anschließend ZIP verwendet.
Das zentrale Ergebnis der Arbeit ist ein Verfahren das bezüglich der Kodierungszeit schneller ist als ZIP, bei der Kompressionsrate fast auf Höhe mit ZIP, aber bei der Dekodierung langsamer ist; wird das entwickelte Verfahren als ersten Schritt einer doppelte Kompression genutzt, lassen sich höhere Kompressionsraten erzielen, als mit einer doppelten ZIP-Kodierung.

1.3 Überblick

Im zweiten Kapitel wird auf die Grundlagen SAT-Solving und Kompressionsverfahren eingegangen.
Danach wird eine Voruntersuchung und Datenerhebung über DIMACS Benchmarks vorgestellt.
Darauf folgt die Vorstellung der eigenen Algorithmen.
Im nächsten Kapitel folgen Implementierungsdetails, wie zum Beispiel, welche Datenstrukturen verwendet werden.
Im Anschluss werden die durchgeführten Evaluationen vorgestellt und ausgewertet.
Im letzten Kapitel erfolgt eine Zusammenfassung der Ergebnisse und ein Ausblick auf zukünftige Verwendungsmöglichkeiten.

2 Verwandte Arbeiten

In diesem Kapitel wird SAT-Solving und und verschiedene Ansätze für Kompressionsverfahren vorgestellt.

2.1 SAT-Solving

SAT-Solving bezeichnet den Vorgang des Lösens einer SAT-Instanz.
Eine SAT-Instanz oder auch SAT-Formel besteht aus einer Menge von Klauseln. Eine Klausel wiederum besteht aus einem oder mehreren Literalen, welche mit Junktoren (und/oder/nicht) verknüpft sind. Ein Literal repräsentiert eine Variable, welche die Werte „wahr“ oder „falsch“ annehmen kann. SAT-Instanzen sind erfüllbar, wenn eine Belegung für alle Variablen existiert, das die Formel zu „wahr“ ausgewertet werden kann, deshalb wird es auch Erfüllbarkeitsproblem genannt. Somit fällt SAT-Solving in den Bereich der Aussagenlogik. Das Lösen einer solchen SAT-Instanz ist NP-schwierig.
Ein gängiges Verfahren zum lösen von SAT-Instanzen ist DPLL(Davis-Putnam-Logemann-Loveland)[5]. DPLL ist ein rekursiver Algorithmus, welcher die Belegung für ein Literal aus der Ausgangsinstanz festlegt und diese neue entstandene Formel zu lösen versucht. Es gibt zwei Kriterien, nach denen die Belegung vereinfacht wird: Zum einen die Einheitsklauseln, bestehend aus nur einem Literal, auf „wahr“ gesetzt; zum anderen werden reine Literale, welche nur positiv oder negativ vorkommen so belegt, dass alle Klauseln mit diesem Literal erfüllbar sind.
Das heute am meisten verbreitete Verfahren ist CDCL (Conflict-driven clause learning)[14], welches auf DPLL aufbaut.
SAT-Solver wie MiniSAT[8] oder Glucose[2] können automatisch SAT-Instanzen lösen, beide verwenden das DIMACS-Format.
Das DIMACS-Format repräsentiert eine SAT-Instanz und ist einfach aufgebaut.
Als erstes besitzt es einen Header, welcher angibt, wie viele Variablen und Klauseln die Datei hat. Ein Beispielheader wäre „p cnf 4 2“: Die größte Variable hat den Wert Vier und es sind zwei Klauseln vorhanden. Die Formel in einer DIMACS-Datei ist in der Konjunktiven Normalform. Es folgen die Klauseln, jede Klausel in seinen eigenen Zeile. Die Literale werden als Zahlen dargestellt. Beispielsweise repräsentieren die beiden Zeilen „1 -2 0“ und „2 3 4 0“, die aussagenlogische Formel $(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3 \vee x_4)$. Die „0“ dient als Trennzeichen zwischen den Klauseln. Die Negationen werden als „-“ vor die negierten Literale geschrieben. Ein „c“ am Anfang der Zeile stellt einen Kommentar dar.

2.2 Kompressionsverfahren

Mit Kompressionsverfahren sind Verfahren gemeint, welche Daten komprimieren können(vgl. [4]). Diese werden genutzt, um Datenmengen kompakter abzuspeichern. Dazu ist es notwendig, dass die Komprimierung wieder umgekehrt werden kann, mathematisch gesehen invertierbar ist. Eine bekanntes Kompressionsverfahren ist das ZIP-Format [13], welches auch für Stringkompression benutzt werden kann. Ein Beispiel für eine einfache Kompression wäre bei einem Text die Leerzeichen wegzulassen. Also aus „Dies ist ein Beispiel“ wird „DiesisteinBeispiel“. Allerdings ist die Umkehrung problematisch, da erkannt werden muss, wo ein Wort beginnt und endet. Hieran ist auch erkennbar, dass die Umkehrung der Kompression im allgemeinen schwieriger ist als das Komprimieren.
Eine weitere Möglichkeit zur Kompression ist eine variable Bitanzahl für die einzelnen Symbole. Dazu kann eine Häufigkeitsanalyse der Symbole durchgeführt werden und mit dieser wird bestimmt, welche Symbole am häufigsten vorkommen: Diesen wird die kleinstmögliche Bitanzahl

zugewiesen.

Symbol	Anzahl	Kodiert
i	5	1
e	4	01
s	3	11
' '	3	10
D	1	001
t	1	011
B	1	010
p	1	0111
l	1	0001
n	1	0010

Tabelle 1: Symbole nach Häufigkeit geordnet und ihre Kodierung

Der komprimierte String, mit der Kodierung in Tab. 1, würde mit 00110111 starten. Die Dekompression ist allerdings nicht mehr eindeutig: 001 wird zu 'D', als nächstes kommt 10111, entweder wird nur eine 1 als i übersetzt oder eine 10 als ' '. Also ist die Zuweisung von Symbol zu Bits nicht einfach ohne Regeln machbar. Damit dies möglich ist, muss es sich bei der Zuweisung um einen Präfixcode handeln. Bei einem Präfixcode ist kein Symbol Präfix eines anderen, in diesem Beispiel wird diese Bedingung unter anderem von i = 1 und ' ' = 10 verletzt, da 1 ein Präfix von 10 ist.
Eine bekannter Präfixcode ist die Huffmankodierung.

Symbol	mögliche Huffmankodierung
i	00
e	010
s	011
' '	100
D	1010
t	1011
B	1100
p	1101
l	1110
n	1111

Tabelle 2: Symbole und eine mögliche Huffmankodierung derselben

Die Huffmankodierung [10] nutzt einen statistischen Ansatz für verlustfreie Kompression. Die Basis für die Kodierung bildet ein Binärbaum: Die Blätter sind die Symbole mit ihrer Wahrscheinlichkeit, dass sie im Text vorkommen, als Gewicht. Dann werden die beiden Blattknoten mit der niedrigsten Wahrscheinlichkeit mit einem Elternknoten verbunden; das neue Gewicht ist die Summe der vorherigen. Dies wird solange durchgeführt, bis der Baum eine Wurzel hat. Eine allgemeine Konvention ist, dass der linke Kindknoten die Null bekommt und der rechte die Eins, beim traversieren des Baumes. Jetzt ist die Kodierung eines Symbols der Pfad von

der Wurzel zum Blatt.
Der Anfang der Huffman-Kodierung angewandt auf das vorherige Beispiel ist 101000010011...; diese Kodierung ist eindeutig und es findet immer noch eine Kompression statt, siehe Tab. 2. Vor der Kompression hat jedes Symbol acht Bits benötigt, mit der Huffmankodierung nur noch maximal 4 Bits, somit wurde in diesem Fall eine Kompression von etwas über 50% erreicht. Dies ist nur ein mögliches Verfahren zum Komprimieren von Daten: Im folgenden werden weitere Methoden und Algorithmen vorgestellt, diese können und werden auch in der Praxis kombiniert.
Ein Beispiel für ein Kompressionsverfahren, das Huffmankodierung nutzt, wäre der DEFLATE Algorithmus[7], welcher bei ZIP verwendet wird.

2.3 Lempel-Ziv

LZ1 und LZ2, erstmals in [11] und [12] vorgestellt sind zwei verlustfreie Datenkompressionsalgorithmen. Diese bilden die Basis für einige Kompressionsverfahren, unter anderem dem DEFLATE Algorithmus.
Die Grundidee ist, den Eingabestring über einem endlichen Alphabet A in Substrings oder Wörter zu zerlegen, welche eine vorher definierte Länge L_s nicht überschreiten. Diese werden dann auf einen einzigartiges, entschlüsselbares Codewort L_c fixer Länge über dem gleichen Alphabet A abgebildet.
Umgesetzt wird dies mithilfe eines Buffers. Dieser hat eine Länge n und wird initialisiert mit $n - L_s$ Nullen und L_s ersten Symbolen des Eingabestrings. Dann wird im Buffer der längste Präfix der Länge l gesucht und dann das Codewort generiert. Jetzt wird der Buffer aktualisiert, indem die ersten l Symbole gelöscht werden und am Ende die nächsten l Symbole des Eingabestrings hinzugefügt werden. Danach wird im Buffer wieder der längste Präfix gesucht. Dieser Vorgang wiederholt sich solange, bis der Eingabestring zu Ende ist. Manchmal wird dieses Vorgehen auch Sliding-Window-Encoding genannt.
Zum Dekodieren wird der Kodierungsprozess umgedreht. Der Buffer hat jetzt eine Länge $n - L_s$, wird mit Null initialisiert und speichert die dekodierten Symbole.

2.4 Move-to-Front

Die Grundidee bei Move-to-Front ist es ein Alphabet A zu führen, in welchem sich die am häufigsten genutzten Symbole weit vorne in der Liste befinden.
Erstmals wurde der Move-to-Front Ansatz in [15] für Datenkompression verwendet und seitdem häufig in Literatur zur Datenkompression gefunden, wie in [4]. Ein Symbol wird ans vordere Ende der Liste gesetzt, wenn es erkannt wird (daher auch der Name Move-to-Front). Es werden die Indizes gespeichert anstatt das eigentlichen Symbols. Außerdem muss das Alphabet noch in irgendeiner Form gespeichert werden: Die Kompression erfolgt dadurch, dass häufig benutzten Symbolen niedrige Nummern zugewiesen werden und diese dann gespeichert werden.
Wenn beispielsweise $A = (h, a, l, o)$ ist und das nächste Symbol ein l ist, wird dieses mit 2 kodiert. Danach wird das l an die erste Stelle des von A gesetzt, also $A = (l, h, a, o)$. Wenn ein Symbol häufig vorkommt, bleibt es somit am Anfang der Liste, während seltene Symbole eher am Ende sind.
Bei obigem Alphabet würde das Wort *hallo* zur Zahlenfolge 01203, diese Folge könnte beispielsweise mit einer Huffman-Kodierung gespeichert werden. Es können auch andere Präfixcodes verwendet werden. Diese Verfahren funktioniert dann gut, wenn viele Wiederholungen von Symbolen oder Symbolfolgen auftreten, die nahe beieinander liegen, da dann die Zahlenfolge aus kleinen Zahlen besteht und diese entsprechend komprimiert werden können.

Die Wahl des Alphabets hat ebenfalls Auswirkungen auf die Kompression. Das Alphabet könnte zum Beispiel der ASCII Code sein, es kann aber auch ein eigenes Alphabet genutzt werden, beispielsweise alle Kleinbuchstaben. Hier tritt das Problem auf, wie der Dekompressor feststellt, welches Alphabet verwendet wurde; bei vorher festgelegten Alphabeten muss diese Information mitgereicht werden. Es gibt aber auch die Möglichkeit, das Alphabet dynamisch beim Komprimieren zu erstellen, allerdings muss auch hier sichergestellt werden, dass beim Dekomprimieren das gleiche Alphabet verwendet wird.

2.5 Variable-Length Quantity

Das Variable-Length Quantity (VLQ) Verfahren ist eine Variable-Length Kodierung für positive Zahlen und wurde ursprünglich für das MIDI-Format definiert[1].
Dieses Verfahren oder Abwandlungen davon wrerden auch in der Praxis, beispielsweise von Google Protocol Buffers[1] oder im .NET Framework[2], genutzt.
Variable-Length Quantity nutzt die letzten sieben Bits pro Byte für Daten, das erste Bit dient als Indikator. Das erste Bit gibt an, ob das nächste Byte noch zur Zahl gehört oder nicht. Ein Nachteil dieser Methode ist, dass Zahlen, welche exakt ein Vielfaches von acht Bits nutzen immer noch ein weiteres Byte benutzen. Beispielsweise wird aus 128 anstatt 10000000 die Kodierung 1000000100000000, somit können aus den 4 Byte, die ein Integer benötigt, 5 Byte in kodierter Form werden. Allerdings brauchen die meisten Zahlen weniger als wenn sie als 32Bit Integer gespeichert werden.

[1] https://developers.google.com/protocol-buffers/docs/encoding?csw=1

[2] https://docs.microsoft.com/en-us/dotnet/api/system.io.binarywriter.write7bitencodedint?redirectedfrom=MSDN&view=4.7.2#System_IO_BinaryWriter_Write7BitEncodedInt_System_Int32_

3 Voruntersuchung

In diesem Kapitel geht es um die Voruntersuchung gängiger SAT-Instanzen und ihrer Analyse.

3.1 Datenerhebung und Auswertung

Über verschiedene Benchmarkdateien [9] wurden automatisch Daten erhoben und in Graphen visualisiert. Die Datenerhebung und darauffolgende Analyse soll dazu dienen, um potentielle Ansätze zur Kompression zu diskutieren. Es wurde pro Benchmark-Typ eine zufällige Datei zur Analyse gewählt, um vielfältige Formeltypen zu betrachten. Es wurde unter anderem das Vorkommen einzelner Zahlen und die Beträge selbiger erfasst, um zu sehen, wie der genutzte Zahlenbereich verteilt ist. Ebenfalls wurde die Differenzen der Zahlen erfasst, einmal pro Klausel und einmal klauselübergreifend. Die Null wurde nicht erfasst, da sie als Trennzeichen zwischen den einzelnen Klauseln dient. Des weiteren wurden die Differenzen auch normiert bezüglich der maximalen Differenz innerhalb einer Datei erfasst. Als letztes wurden die Verteilung über die Klausellängen erfasst.

3.2 Diskussion

Wie in Abb. 1 zu sehen ist, umfassen die Zahlen, welche zu kodieren sind, einen großen Bereich. Es ist folglich nicht möglich, die Zahlenwerte auf einen starren Bereich einzuschränken. Der Durchschnitt und Median liegen eher nahe beieinander und sind kleiner Null, also ist es wahrscheinlich, dass eine knappe Mehrheit der Zahlen ein negatives Vorzeichen besitzt. Bei

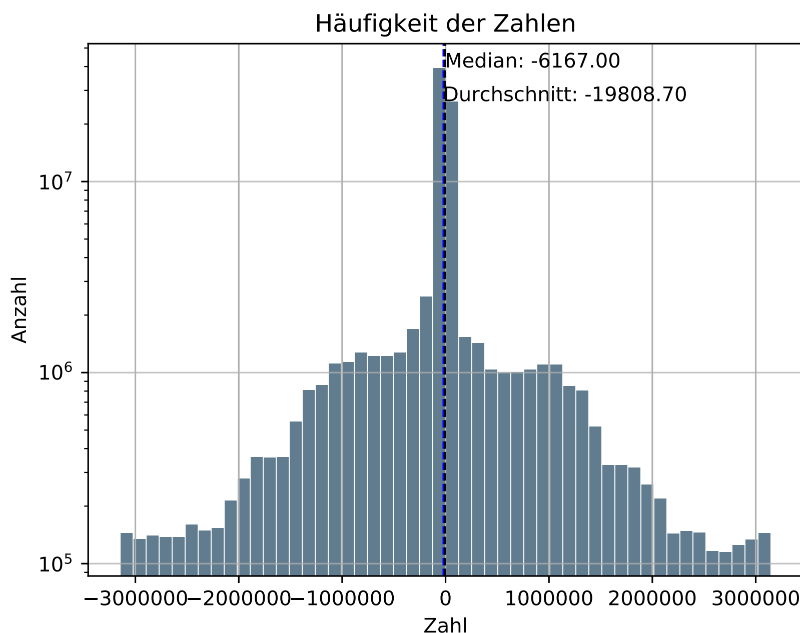

Abbildung 1: Histogramm über alle vorkommenden Zahlen in den Daten

Abb. 2 wurden nur die Beträge der Zahlen betrachtet. Hier ist eine Verschiebung von Durchschnitt und Median weit in positiver Richtung auf der x-Achse zu sehen. Dies war zu erwarten, da keine negativen Zahlen mehr vorkommen, allerdings ist der Abstand zwischen Median und

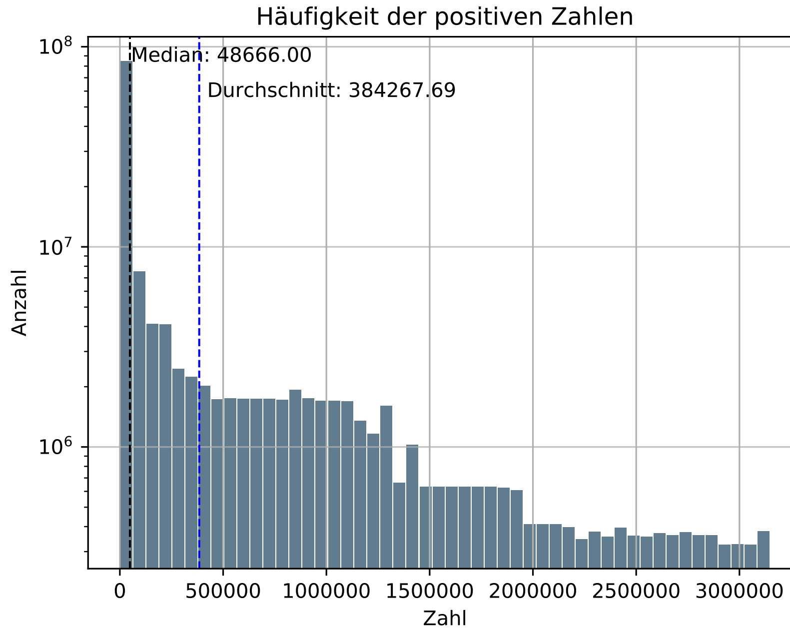

Abbildung 2: Histogramm über alle vorkommenden Beträge der Zahlen in den Daten

Durchschnitt weitaus größer als zuvor. Der Median ist weitaus niedriger und deshalb sollte vielleicht für Werte unter dem Median eine effizientere Kodierung verwendet werden, falls möglich. Ebenfalls könnte es sich lohnen die Vorzeichen zu extrahieren, beispielsweise in einen separaten Bitvektor, damit ein Variable-Length Quantity Verfahren möglich ist.
Für so einen Bitvektor ist relevant, wie lange die Klauseln sind. Deshalb wurde in Abb. 3 die Länge der vorkommenden Klauseln untersucht. Dabei ist zu erkennen, dass vor allem kleine Klauseln vorkommen: der Durchschnitt liegt bei einer Klausellänge von knapp drei. Lange Klauseln kommen verhältnismäßig selten vor. Für einen Bitvektor sollte also ein Verfahren gewählt werden, aas vor allem kleine Bitvektoren effizient kodiert.
Des weiteren wurde erwägt, ein Differential Encoding zu realisieren. Dafür wurde die Differenz der Zahlen erfasst. Einmal über die gesamte Datei, also klauselübergreifend (Abb. 4) und einmal innerhalb der Klauseln (Abb. 5). Es wurde festgestellt, dass es keinen relevanten Unterschied macht, ob die Differenz klauselübergreifend betrachtet wurde oder nicht. Die erhoffte Häufung der Zahlen um die Null ist zwar eingetreten, allerdings sind auch andere Effekte sichtbar.
Zum einen hat sich der Bereich im Vergleich zu Abb. 1 verdoppelt, von ca -3 Millionen bis 3 Millionen auf ca -6 Millionen bis 6 Millionen. Es gibt folglich auch Differenzen zwischen kleinsten Literalen und dem Größten, die in einer Datei vorkommen. Eine Beispielklausel wäre: 1024 − 10251026, hier ist die Differenz von 1024 zu -1025 größer als der Maximalwert 1026. Solche Klauseln können häufiger in einer Datei vorkommen, wenn zum Beispiel verschiedene Kombinationen vom Literale getestet werden.
Zum anderen ist die Häufung um die Null geringer als zuvor.
Neben der Kodierung mit Variable-Length Quantity und dem Differential Encoding wurden noch eine Move-to-Front Kodierung und ein Präfixmatcher Ansatz erwägt. Um zu überprüfen, welche Ansätze Erfolg versprechen wurde eine grobe Simulation des Speicherverbrauchs in Bytes nach der Kodierung erstelle(Tab. 3).
Aus der Simulation ging hervor, dass ein Differential Encoding keinen Erfolg verspricht, da es nicht effektiver scheint als die VLQ, deshalb wurde dieser Ansatz nicht weiter verfolgt. Der Move-to-Front- und Präfixansatz wurden als geeignete Kandidaten angesehen, um weiter auf

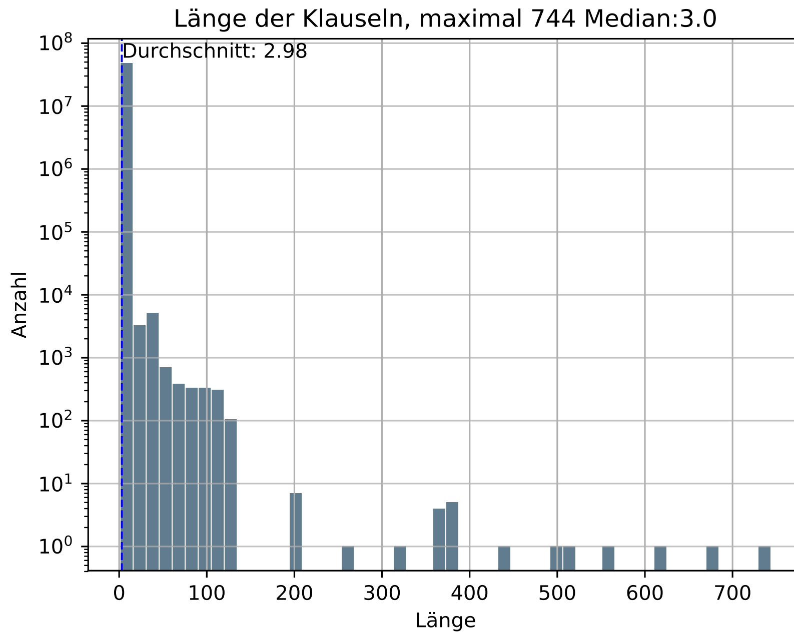

Abbildung 3: Histogramm über die vorkommenden Klausellängen

der VLQ-Kodierung aufzubauen. Dadurch entstand die Idee beide Ansätze zu verbinden; diese Kombination wurde auch simuliert und führte zu vielversprechenden Ergebnissen. Aufgrund dieser Simulation wurde entschieden die Verfahren Move-to-Front, Präfix und die Kombination weiter zu verfolgen.

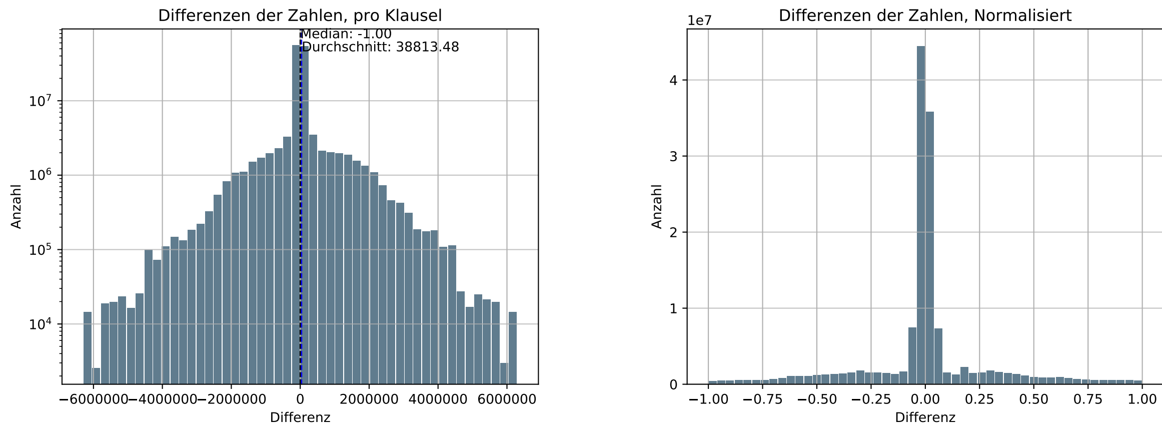

Abbildung 4: Differenzen über alle Zahlen, rechts normiert bezüglich der maximalen Differenz

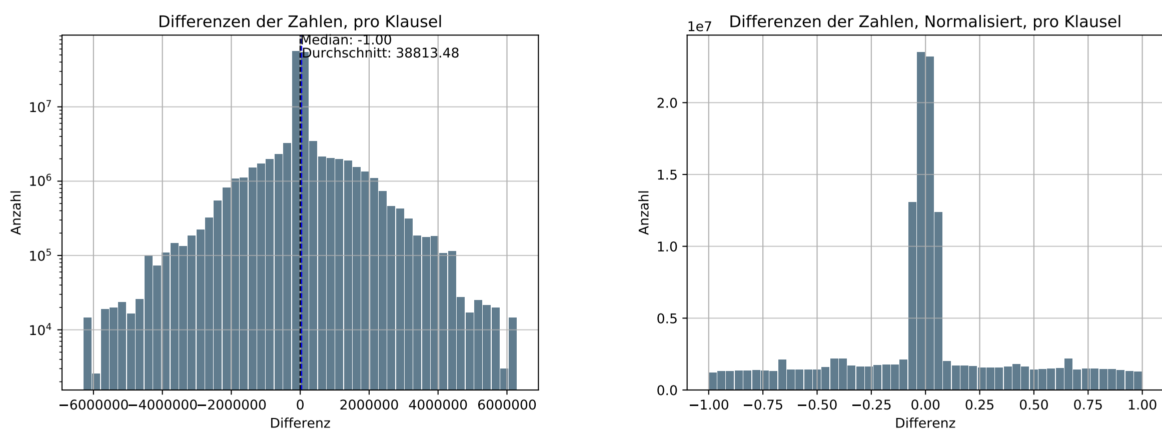

Abbildung 5: Differenzen über alle Zahlen innerhalb einer Klausel, rechts normiert

Originalgröße	Basis	VLQ	Differential	MtF	Präfix	Kombi
16234	17920	5490	6644	6318	**5446**	5554
25426	26848	7676	10057	7809	**7443**	7828
44989	48600	17464	18938	15217	20228	**14530**
290200	266560	89427	108676	75964	59641	**51066**
366266	336344	**108164**	120356	105982	111060	112387
397759	364168	**100576**	143562	129386	118239	117566
576546	594528	212890	222986	258673	215694	**182818**
718883	623192	198367	231770	**193499**	200315	206012
873012	747864	245042	302660	**242190**	251838	253107
1163966	1099000	325540	480796	309223	250818	**242979**
1253925	1052856	360657	431283	296337	327852	**275639**
1443453	1317008	**332065**	488363	550643	332075	438863
1536964	1259456	438209	528257	353900	407152	**335242**
2232062	1717488	605431	826999	483452	401970	**362537**
2262808	2127440	**566137**	868463	1019871	566142	710281
2271733	1347624	512848	667948	439115	394309	**379984**
3386619	2988160	1068925	1464132	860942	914431	**623788**
5377970	4132248	1698530	2027090	1251838	1473390	**1019877**
5865195	5039784	**1271969**	**1271969**	2129205	1271976	1674175
14384818	10404056	4429288	5130687	3136134	3603850	**2624642**
16522623	11884076	5036053	5589498	3492475	4639287	**3261459**
16920923	12329792	5398256	6213206	3803858	4698076	**3078867**
17776014	15122688	5790882	8291485	4436168	2698758	**2332858**
21634016	18546272	6456407	8144180	7074411	7783388	**5167289**
32795466	21796280	7989407	12607036	8482230	11635180	**6027187**
35429310	22873104	9377898	12753815	7454039	7577067	**5887837**
60555579	38803592	15426170	21274730	17965028	16119333	**13960603**
65709554	51209524	20554130	27710618	13343802	21254542	**13077184**
122486652	78560000	33427607	40055679	23712306	25548805	**19617735**
244536572	155289180	72764842	77963380	55588102	62948643	**45722004**
415182722	310219304	126266408	163078615	166128384	162675516	**83013593**

Tabelle 3: Simulation des Speicherverbrauchs der potentiellen Kodierungen; Ergebnisse in Bytes; Basis ist die Umwandlung der Zeichenketten in Zahlen

4 Entwurf

In diesem Kapitel werden die verschiedenen entwickelten Algorithmen vorgestellt.

4.1 Basis-Algorithmus

Da ein bestimmtes Datenformat, das DIMACS-Format, vorliegt, war die erste Überlegung die Zahlen nicht als Zeichenketten sondern direkt als Zahlen zu speichern.
Dies ist eine Fixed-Length Kodierung; dies bedeutet, dass jede Zahl gleich viel Speicherplatz benötigt, egal wie groß die Zahl ist; in diesem Fall ist es eine feste Byteanzahl. Um die Byteanzahl zu bestimmen wird die maximale Variablengröße aus dem Header verwendet, entsprechend dieses Wertes werden alle Zahlen entweder in einen 16Bit oder 32Bit Integer umgewandelt. Der ursprüngliche Header mit „p cnf“ wird nicht gespeichert, stattdessen steht als erstes die Byteanzahl der Zahlen. Danach kommen Variablen und Klauselzahl, wie bei der Ursprungsdatei und als letztes die Klauseln. Bei den Klauseln wird weiterhin die „0“ als Trennzeichen zwischen den Klauseln verwendet. Die eventuell vorhandenen Kommentare werden nicht mit in die kodierte Datei übernommen und gehen somit verloren, dies ist bei späteren Tests zu berücksichtigen. Um die Vorgehensweise zu verdeutlichen, wird folgende Beispieldatei (Abb. 6) mit einer Größe von 59 Byte kodiert. Der maximale Wert in diesem Beispiel beträgt 8192, somit können alle

```
p cnf 8192 3
-1 2 -3 4 5 6 -7 0
8192 -10 1024 0
-8192 1024 5 0
```

Abbildung 6: Beispieldatei

Zahlen mit 16Bit dargestellt werden. Jetzt geht der Algorithmus jede Zeile durch und wandelt die Zahlen von Zeichenketten in Integers um und gibt sie aus. Die Ausgabedatei hat dadurch eine Größe von 36 Byte, was niedriger ist als die der Ausgangsdatei. Im Abb. 7 ist die Formel Abb. 6 mit diesem Algorithmus kodiert.
Bei ersten Tests zeigte dieser Algorithmus bereits eine Komprimierungsrate von bis zu 50%.

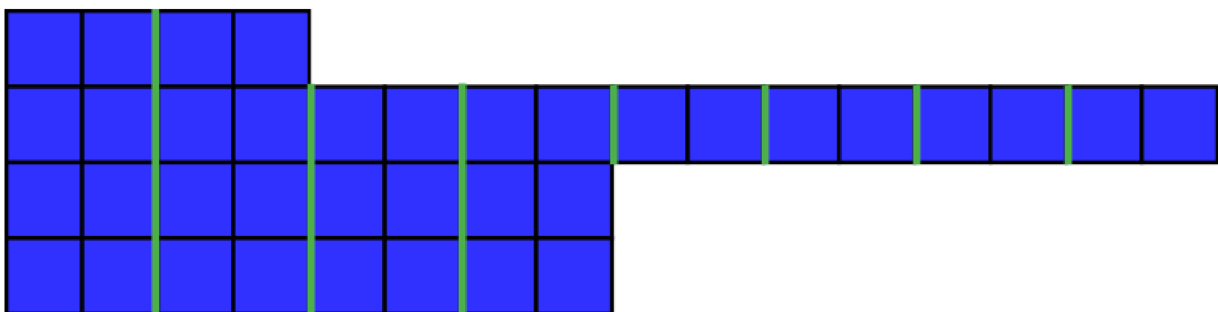

Abbildung 7: Beispieldatei mit Basisalgorithmus kodiert. Jeder Kasten ist ein Byte. Der grüne strich trennt einzelne Zahlen.

Wird danach noch eine ZIP-Komprimierung angewendet ist es möglich noch weiter zu komprimieren. Dies liegt unter anderem daran, das sehr viele Byte, welche in allen Bits den Wert Null haben, vor allem bei kleinen Zahlen, angefallen sind und dies eingespart werden kann.

4.2 Variable-Length Quantity

Der nächste Algorithmus, im folgenden VLQ genannt, baut auf dem vorherigen auf und extrahiert zudem die Vorzeichen in einen separaten Bitvektor.
Das erste Bit des Bitvektors gibt an, um welche Art von Bitvektor es sich handelt. Die Länge wird angegeben um sicher zu sein, dass die richtige Anzahl an Zahlen entschlüsselt wird, dadurch wird kein explizites Trennzeichen mehr benötigt. Bei einer „0“ im ersten Bit geben die nächsten drei Bit die Länge des Bitvektors an und die letzten vier enthalten die eigentlichen Vorzeichen. Somit können Klauseln bis zur Länge vier mit so wenig wie möglich Speicherplatz kodiert werden. Bei einer „1“ im ersten Bit ist auch noch das zweite Bit für die Art des Bitvektor entscheidend. Bei einer „0“ geben die nächsten sechs Bits die Länge des Bitvektors an. Die nachfolgenden Bits sind der eigentliche Bitvektor, somit werden mit diesem Bitvektor Klauseln der Länge fünf bis 32 kodiert. Ist das zweite Bit eine „1“, ist die Länge des Bitvektors mit einer Art Variable-Length Quantity kodiert. Im Vergleich zur normalen Variable-Length Quantity tragen hier im ersten Byte nur die letzten sechs, statt der üblichen sieben, zu den Daten bei. Danach kommt der eigentliche Bitvektor, wieder bitweise. Beim eigentlichen Bitvektor bedeutet eine „0“ ein positives Literal und eine „1“ eine Negation.
Die Zahlen werden mit Variable-Length Quantity kodiert. Bei dieser Methode entfällt das Speichern des Trennzeichens, sowohl zwischen Klauseln als auch zwischen den Zahlen, da die Länge der Klausel bzw. Zahl bekannt ist. Der Bitvektor ersetzt das Trennzeichen zwischen den Klauseln und für kleine Klauseln braucht er nicht mehr Speicher als eine kodierte Null, wie sie vorher genutzt wurde; dies war nach der Analyse der Daten der häufigste Fall.
Zur Veranschaulichung ist in Abb. 8 ist 6 mit der VLQ-Kodierung kodiert. Der Bitvektor in

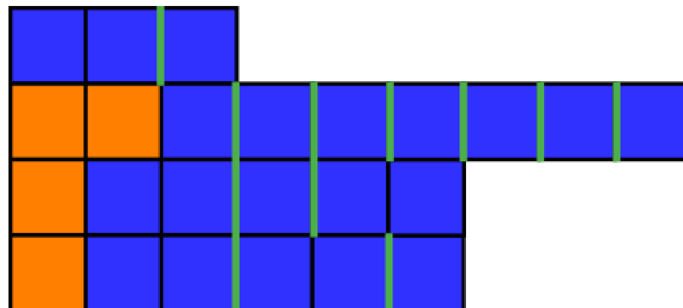

Abbildung 8: Beispieldatei mit der VLQ kodiert. Jeder Kasten ist ein Byte. Blau sind Zahlen, Grün trennt einzelne Zahlen, Orange der Bitvektor

der zweiten Zeile beginnt mit „10“ im ersten Byte, um die entsprechende Art des Bitvektors festzulegen. Das nächste Byte beinhaltet die eigentlichen Vorzeichen. Es ist zu erkennen, dass der Speicherverbrauch reduziert wurde auf 24 Byte.

4.3 Move-to-Front

Im Anschluss wird ein Move-to-Front Ansatz mit einem begrenzten und dynamischen Alphabet vorgestellt.
Zunächst muss das genutzte Alphabet definiert.
Eine Möglichkeit ist es alle vorkommenden Zahlen in das Alphabet aufzunehmen; es ist allerdings offensichtlich, dass damit nichts gewonnen wird, da die Indizes sehr groß werden können. Auch das Alphabet mit den Ziffern von 0...9 und -,+ ist nicht verwendbar: Die meisten Zahlen würden mehr Speicher verbrauchen als nötig. Beispielsweise würde die Zahl „100“ mit dieser Methode 12Bit benötigen, mit Variable-Length Quantity aber nur 8Bit. Dieser Effekt trifft vor

allem für große Zahlen auf. Also wird die Größe des Alphabets auf 255 festgelegt, somit verbraucht der Index immer ein Byte. Um zu unterschieden, wann es ein Index ist und wann eine Zahl welche noch nicht im Alphabet steht, wird der Wert 255 benutzt, ist dieser gesetzt folgt eine Zahl, welche noch nicht im Alphabet vorkommt und wird an den Anfang des Alphabets gesetzt. Die 255 dient somit als Erkennungswert, ob es sich um eine Zahl handelt oder einen Index. Des weiteren wird wieder der Bitvektor verwendet, sodass nur positive Zahlen in der Liste stehen und kodiert werden.
Allerdings verbraucht jede Zahl, welche nicht im Alphabet vorkommt, ein Byte mehr als nötig. Generell ist dieser Ansatz für Zahlen kleiner gleich 127 kein Gewinn, da diese mehr Speicher verbrauchen als vorher.
Eine natürliche Optimierung ist daher, das Alphabet weiter zu beschränken und den Erkennungswert zu verändern. Zahlen unter 127 werden nicht ins Alphabet aufgenommen, da dies keinen Vorteil bringt. Das Alphabet wird auf Größe 64 beschränkt. Der Erkennungswert wird so verändert, dass er der Trennung der verschiedenen Bedeutungen des Bitvektor ähnelt. Ist das erste Bit „0,“ ist das restliche Byte eine Zahl. Ist das erste Bit „1“, und das zweite eine „0“ sind die restlichen Bits der Index im Alphabet. Sind die beiden vorderen Bits „1“ beginnt mit den nächsten sechs Bits eine Zahl, welche nicht in der Liste ist; diese Zahl ist wieder mit Variable-Length Quantity kodiert. Somit werden bei der Verbesserung die ehemals leeren Bits des Erkennungswertes benutzt und damit eine weitere Komprimierung erreicht.
Zur Veranschaulichung wird jetzt wieder das Beispiel aus Abb. 6 benutzt. In Abb. 9 ist zu

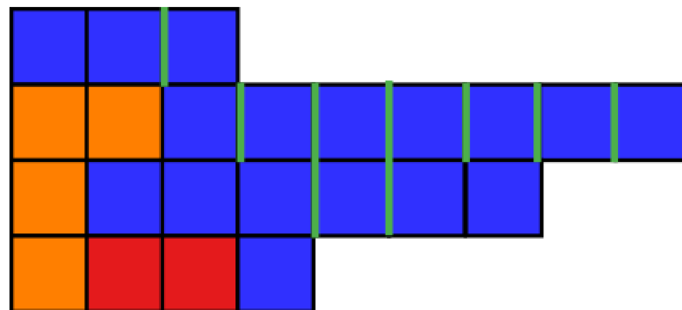

Abbildung 9: Beispieldatei mit Move-to-Front kodiert. Jeder Kasten ist ein Byte. Blau sind Zahlen,Grün trennt Zahlen, Orange der Bitvektor, Rot ist ein Index im Alphabet

erkennen, dass es sich nur in den letzten beiden Zeile von der VLQ-Kodierung unterscheidet. Dort wurden 8192 und 1024, welche in der vorletzten erkannt wurden, als Indizes gespeichert. Die vorletzte Zeile verbraucht jetzt mehr Bytes als vorher, da die 8192 ein Grenzfall ist, welche mit dieser Kodierung drei Bytes benötigt. Es wurden insgesamt 23 Bytes benötigt.

4.4 Präfix

Als letztes wird ein Verfahren basierend auf Präfixmatching vorgestellt, kurz Präfix. Dieses speichert immer die letzte Klausel ohne Vorzeichen und überprüft inwieweit die aktuelle Klausel mit der letzten übereinstimmt. Gibt es mehr als einen Match mit der letzten Klausel, wird ein Erkennungsbyte vor den Bitvektor geschrieben. Das Erkennungsbyte beginnt mit „0000“, damit es nicht vom Decoder mit einem Bitvektor verwechselt wird. Die letzten vier Bits geben die Anzahl der Übereinstimmungen an, somit können auch maximal 16 Literale übereinstimmen. Ansonsten ist dieser Algorithmus identisch zur VLQ-Kodierung, d.h. es wird ein Bitvektoren extrahiert und die Zahlen werden mit Variable-Length-Quantity kodiert.
Das Beispiel aus Abb. 6 mit Präfix kodiert Abb. 10. Es werden 23 Byte benötigt und es ist zu erkennen, dass anstatt zwei Bytes nur eines für 8192 verbraucht wird.

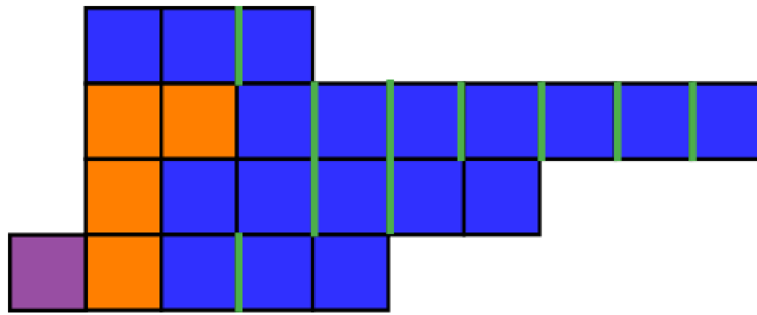

Abbildung 10: Beispieldatei mit Präfix kodiert. Jeder Kasten ist ein Byte. Blau sind Zahlen,Grün trennt Zahlen, Gelb der Bitvektor, Violett ist das Erkennungsbyte für den Präfix

4.5 Kombination

Das zuletzt entwickelte Verfahren ist eine Kombination aus Präfix und Move-to-Front. Das Erkennungsbyte von Präfix bleibt vor dem Bitvektor und somit unverändert. Das Kodieren der Zahlen ändert sich nur insofern, dass, falls das Erkennungsbyte gesetzt ist, entsprechend weniger Zahlen kodiert werden; ansonsten wird wie bei Move-to-Front kodiert.
Das Beispiel aus Abb. 6 kodiert mit der Kombination ist in Abb. 11 dargestellt. Es werden 23 Byte verbraucht. Die dritte Zeile ist identisch mit der dritten aus Abb. 9, hier tritt ebenfalls der Grenzwert auf. In der letzten arbeiten sowohl Präfix als auch Move-to-Front zusammen.

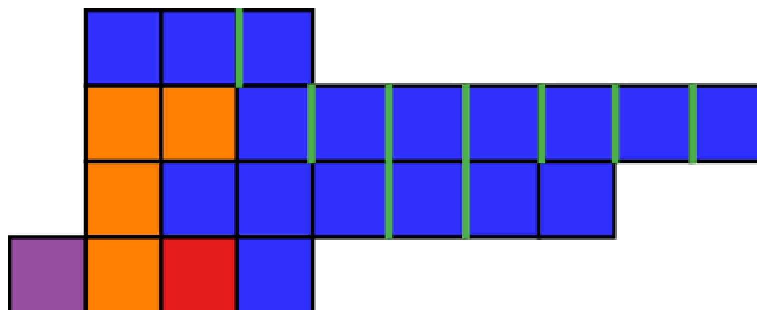

Abbildung 11: Beispieldatei mit Variable kodiert. Jeder Kasten ist ein Byte. Blau sind Zahlen,Grün trennt Zahlen, Gelb der Bitvektor, Violett ist das Erkennungsbyte für den Präfix, Rot ein Indexbyte im Alphabet

5 Implementierung

In diesem Kapitel geht es um die eigentliche Implementierung der Algorithmen und welche Datenstrukturen verwendet werden. Es wurde immer sowohl ein Kodierer als auch ein Dekodierer implementiert. Die Korrektheit wird durch Tests mit verschiedenen Benchmarkdateien überprüft. Falls nicht anders angegeben, nutzen die Dekodierer die gleichen Datenstrukturen wie die Kodierer.

5.1 Basis-Algorithmus

Da der Basis-Algorithmus eine einfache Umwandlung von Zeichenketten in Integer ist, wird der Maximalwert im Header genommen und damit bestimmt, ob 16Bit oder 32Bit Integer benutzt werden. Diese Information in Form der Byteanzahl wurde als erstes ausgegeben, danach der Maximalwert und die Länge. Dann wird die Datei Zeile für Zeile eingelesen, die Literale, welche durch Leerzeichen getrennt sind, in 16Bit oder 32Bit Integers umgewandelt und ausgegeben. Beim Dekodieren werden die eingelesenen Bytes wieder in ihren Ursprungswert umgewandelt.

5.2 VLQ

Zunächst wird die Umsetzung von Variable-Length Quantity und der Kodierung eines Integers erklärt. Wie zuvor erwähnt werden bei Variable-Length Quantity sieben von acht Bits effektiv genutzt. Um dies zu erreichen wird als erstes ein char-Array der Länge fünf, da dies die maximale Anzahl an Bytes ist, die benötigt werden, mit Null in jedem Eintrag initialisiert. Dieses Array wird von hinten nach vorne befüllt. An die letzte Stelle werden die hintersten sieben Bits der Zahl geschrieben. Danach wird eine Schleife durchlaufen: Falls der Wert größer als 127 ist, wird der Integer um sieben nach rechts verschoben und danach wird mit den neuen letzten sieben Bits die nächste Stelle des Arrays gefüllt; gleichzeitig wird das erste Bit auf „1“ gesetzt, in dem das Byte mit 128 addiert wird.
Beim Dekodieren wird der Vorgang umgekehrt. Der neue Integer bekommt die letzten sieben Bits des ersten Zahlenbytes. Falls die Zahl weitere Bytes besitzt, wird der Integer um sieben Bits nach links verschoben und dann solange die nächsten Bytes gelesen, bis ein Byte mit „0“ anfängt.
Jetzt zum eigentlichen Bitvektor, angefangen mit dem für kleine Klauseln. Wird festgestellt, dass die Klausellänge kleiner gleich vier ist, wird das erste Bit null gesetzt, die nächsten entsprechen der Länge der Klausel. Die letzten vier Bits werden entsprechen der Vorzeichen der Klausel gesetzt, eine „1“ bei negativen Vorzeichen und eine „0“ bei positiven. Gesetzt werden nur die „1“, da alle Bytes mit dem Wert „0x00“ initialisierst werden. Mit `|= 1ul « n` wird das n-te Bit eines Bytes auf „1“ gestellt, $0 \leq n \leq 7$. Die Bits werden von Stelle fünf an beschrieben, d.h. wenn eine Klausel kürzer ist, bleiben die letzten Bits auf „0“. Somit sind alle Informationen in einem Byte gespeichert.
Bei den nächst längeren Bitvektoren mir einer Klausellänge zwischen fünf und 64, wird das erste Bit auf „1“ gesetzt und das zweite auf „0“; in den restlichen Bits wird die Länge gespeichert. Danach folgt der eigentliche Bitvektor, in einem Byte werden bis zu acht Vorzeichen gespeichert. Falls der Bitvektor vor Ende eines Byte zu Ende geht, bleiben die restlichen Bits auf „0“.
Bei den längsten Bitvektoren wird nach „11“ im ersten Byte und in den folgenden Bytes die Länge des Bitvektors gespeichert. Dies geschieht mit einer abgewandelten Form von Variable-Length Quantity. Im ersten Byte werden nur sechs statt sieben Bits für die Information genutzt. Zuerst wird dazu die Zahl wieder mit Variable-Length Quantity kodiert, danach wird überprüft,

ob sich im zweiten Bit eine „1“ befindet oder nicht. Ist die zweite Stelle „0“, wird sie zu einer „1“ und damit ist die Kennzeichnung vollständig. Bei einer „1“ wird eines neues Byte vor die eigentliche Zahl geschrieben mit den ersten beiden Bytes auf „1“, um die Kennzeichnung zu erhalten. Dies tritt nur bei den Grenzfällen auf.
Der Algorithmus geht wieder Zeile für Zeile vor und speichert alle Zahlen einer Zeile und den Bitvektor in jeweils einem `std::vector<int>`, bevor sie ausgegeben werden.

5.3 Move-to-Front

Im Wesentlichen unterscheidet sich Move-to-Front von der VLQ-Kodierung nur beim Kodieren der Zahlen. Die Liste des Alphabets, im weiteren Movelist genannt, ist auf die Größe von 64 Einträgen begrenzt. Als Datenstruktur wird `std::list<int>` verwendet. Die Kodierung der Zahlen findet wieder nach dem Herausschreiben des Bitvektors statt. Die Verarbeitung des Bitvektors ist identisch mit der VLQ-Kodierung; Zahlen und Bitvektor werden wieder in `std::vector<int>` zwischengespeichert. Es wird für jede Zahl erst einmal überprüft, welchen Wert sie hat: Ist sie kleiner gleich 127, wird sie direkt kodiert. Bei den Zahlen größer 127 wird überprüft, ob die Zahl schon in der Movelist vorhanden ist, mithilfe von `std::find`. Falls sie vorhanden ist, wird der Index in einen Integer umgewandelt und dann das erste Bit auf „1“ gesetzt, um einen Index für den Decoder zu markieren.
Eine Liste als Datenstruktur wurde aufgrund des häufigen Einfügens an die erste Position der Movelist verwendet, was beim einem Array weitaus aufwändiger ist.
Kodiert werden die Zahlen, welche nicht in der Liste vorkommen, mit einem abgewandelten Variable-Length Quantity. Es ist dieselbe Abwandlung, welche schon bei den längsten Bitvektoren vorkam, um deren Länge zu speichern, und wurde dementsprechend implementiert.

5.4 Präfix

Dieser Algorithmus hat große Überschneidungen mit der VLQ-Kodierung.
Der Präfix Encoder verfügt über einen zusätzlichen `std::vector<int>` im Vergleich zur VLQ-Kodierung, welcher die Beträge der Literale der vorherigen Klausel speichert. Das Kodieren geschieht wieder Zeile für Zeile mit Variable-Length Quantity und es gibt einen Bitvektor. Stimmt der Anfang der aktuellen Zeile mit der letzten überein, wird das Erkennungsbyte gesetzt und erst ab der ersten unterschiedlichen Zahl kodiert.

5.5 Kombination

Der kombinierte Ansatz wurde analog zu den bereits vorgestellten Ansätzen implementiert.
Der Präfixteil wird vor dem Bitvektor eingefügt und der Move-to-Front-Algorithmus nach dem Bitvektor.

6 Evaluation

In diesem Kapitel erfolgt die Evaluation der Algorithmen und der Vergleich mit dem ZIP-Format.
Vor der Evaluation wurden jegliche Kommentare aus den Dateien entfernt, da die Algorithmen keine Kommentare unterstützen und um eine bessere Vergleichbarkeit zu erreichen. Die Messung der verschiedenen Größen wurde automatisch vorgenommen. Alle Datenpunkte in den Graphen entsprechen einer Datei, welche komprimiert wurde. Als Testdateien wurden Benchmarks aus der SAT Competition 2018[9] und Planungsinstanzen, generiert durch das Framework Aquaplanning[3], genutzt.

6.1 Kompressionsrate

Mit der einfachen Umwandlung wird eine erreicht eine Kompressionsrate von 1,55 erreicht. Bei kleinen Dateien schwankt die Kompressionsrate zwischen 1 und 2,5 bei großen Dateien, über 100mb zwischen 1,6 und 1,3. Es ist zu erkennen, dass die Kompressionsrate relativ stabil bleibt, auch für größere Dateien (Abb. 12). Die Messpunkte die eine schlechtere Kompressionsrate haben, ca. 1,4, unabhängig von der Größe der Datei, gehören zu einem einzigen Problemtyp gehören. Die Kompressionsrate wurde berechnet, indem die Größe der Ausgangsdatei durch die Größe der komprimierten Datei geteilt wurde. Diese Werte sollen als Grundlage dienen, um festzustellen, ob die anderen Algorithmen eine Verbesserung darstellen.

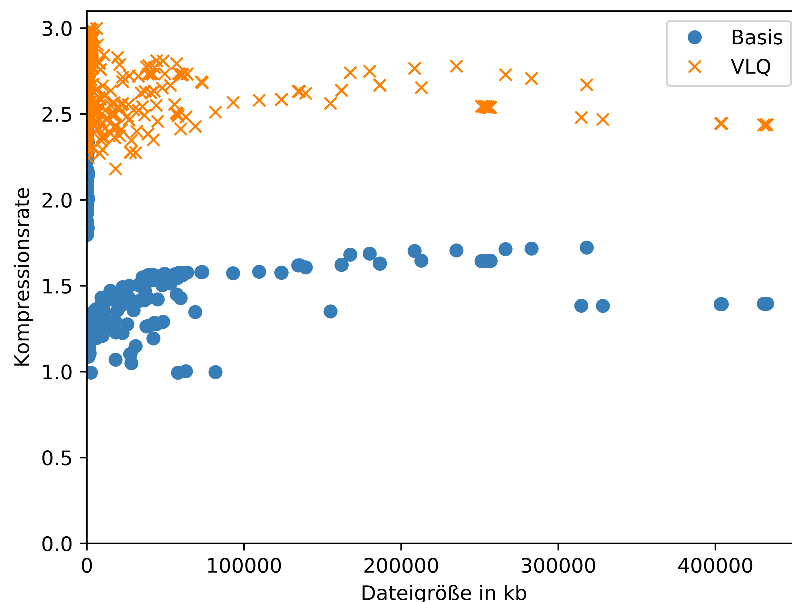

Abbildung 12: Basis und VLQ im Vergleich

Jetzt zum ersten Algorithmus, welcher Variable-Length Quantity nutzt, der VLQ-Kodierung. Hier liegt der Durchschnitt bei 2,61, siehe Tab. 4, was deutlich besser ist als der Basisalgorithmus. Allerdings schwankt auch hier die Kompressionsrate bei kleinen Dateien zwischen 2.25 und 3(Abb. 12). Bei großen Dateien pendelt sie zwischen 2,5 und 2,75. Der Verlauf in Abb. 12 ist ziemlich ähnlich, wobei der VLQ besser abschneidet als die Basis und es ist kein Trend

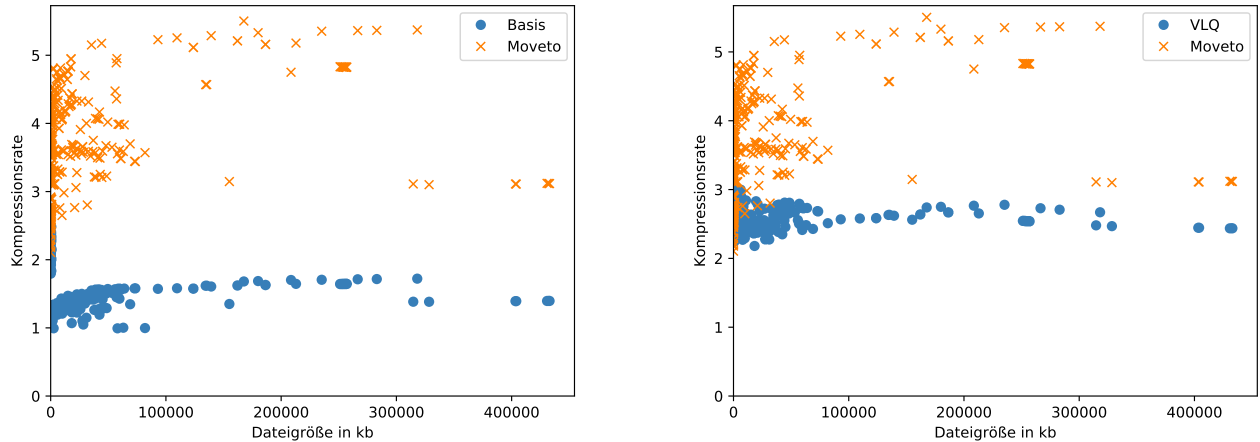

Abbildung 13: Move-to-Front im Vergleich mit Basis links und VLQ rechts

erkennbar. Bei der VLQ-Kodierung haben die beiden größten Dateien eine schlechtere Kompressionsrate als die direkt vorhergegangenen: es sind die gleichen Datenpunkte wie bei dem Basisalgorithmus.

Als nächstes wird der Move-to-Front-Algorithmus in Abb. 13 betrachtet.
Hier sind die Schwankungen weitaus größer, zwischen 2,5 und 5. Bei großen Dateien ist die

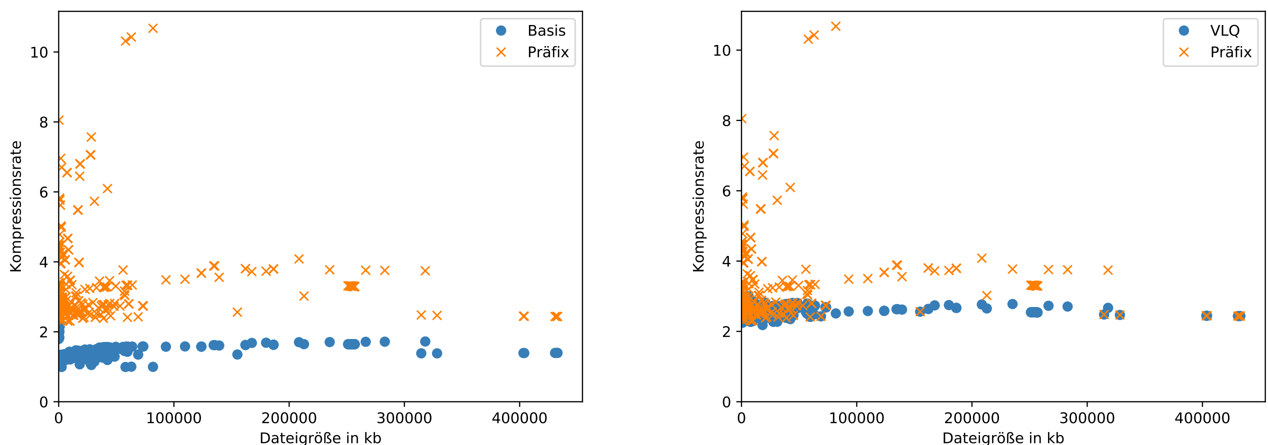

Abbildung 14: Präfix im Vergleich mit Basis links und VLQ rechts

Kompressionsrate relative konstant bei 5 mit kleinen Ausreißern nach unten. Der Durchschnitt beträgt 3,82, siehe Tab. 4. Bei den größten Dateien fällt die Kompression schlechter aus, wie schon bei den vorherigen Algorithmen, und es sind wieder die gleichen Messpunkte. Das Verfahren ist durchgehend besser als der Basis-Algorithmus und die VLQ.

Die Kompressionsrate des Präfix-Algorithmus schwankt zwischen 2 und ca 11 (Abb. 14). Bei den niedrigen Werten ist die Kompressionsrate beinahe bis komplett identisch mit dem Variablen. Dies war zu erwarten, da Präfix auf der VLQ aufbaut und diese auch nicht verschlechtern sollte. Im Durchschnitt ist er, mit 3,35, besser als der Variable, somit auch besser als die Basis, siehe Tab. 4. Es gibt drei Extreme mit einer hohen Kompressionsrate über 10, diese Dateien

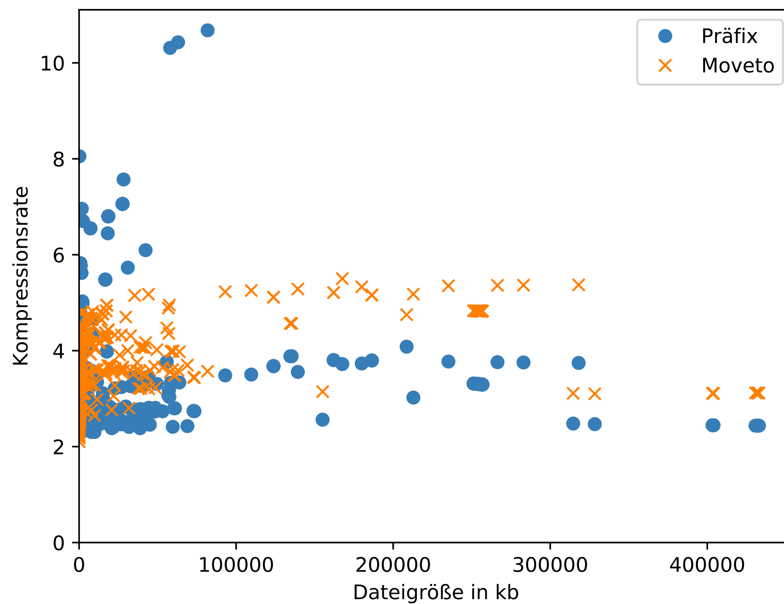

Abbildung 15: Präfix und Move-to-Front im Vergleich

enthalten eine sehr hohe Wiederholung von ähnlichen Klauseln. Es sind wieder die gleichen Messpunkte, die der größten Dateien, welche eine schlechtere Kompressionsrate haben als die vorherigen. Hier sind Präfix und VLQ identisch. Dies bedeutet, innerhalb dieser Dateien gibt es wenige Wiederholungen, was sich auf die Kompressionsrate auswirkt.
Beim Vergleich zwischen Präfix und Move-to-Front Abb. 15 ist erkennbar, dass Move-to-Front im Durchschnitt besser ist als der Präfix. Allerdings sind vor allem die Ausreißer bei Präfix auffällig, bei welchem Präfix Kompressionsrate weitaus besser ist als Move-to-Front.

Als letztes ist der Kombinationsansatz übrig.
Dieser hat einen durchschnittliche Kompressionsrate von 4.51 und manche Datenpunkte sind identisch mit dem Move-to-Front. Dies ist beispielsweise bei den größten Dateien der Fall und war zu erwarten, da der Move-to-Front der Worstcase. Ebenfalls hat die Kombination die Stärken des Präfix übernommen. Die Ausreißer sind hier sogar noch besser, als beim Präfix-Algorithmus, bei ca 12.5. Dadurch hat die Kombination genau das gemacht, was erwartet wurde, die Stärken des Präfix-Algorithmus mit denen von Move-to-Front verbunden Abb. 16.

Nun zum Vergleich zwischen ZIP und der Kombinationslösung (Abb. 17).
Es ist zu erkennen, dass bei kleinen Dateien ZIP deutlich besser ist als die Kombination. Bei größeren Dateien ist die Kombination nur knapp schlechter als ZIP. Ausgenommen hiervon sind die größten Dateien, bei welchem die Kombination wieder abfällt, ZIP wiederum stabil bleibt. Ebenfalls hat ZIP die größeren Ausreißer bei ca 20, anstatt wie bei der Kombination bei 12.5.
Des weiteren hat ZIP mehr Ausreißer und höhere nach oben als die Kombinationslösung, zu sehen in Abb. 18. Präfix, Kombination und ZIP haben alle signifikante Ausreißer nach oben. Präfix hat hiervon die meisten, kommt aber nicht an die Ausreißer der Kombination heran; ZIP hat die höchsten Ausreißer. Es ist ebenfalls zu erkennen, dass der Worstcase aller Algorithmen (außer der Basis), der Bestcase der Basis ist; dies war zu erwarten, da alle anderen Algorithmen auf der Basis aufbauen und diese verbessern. Interessant ist auch, dass nur ZIP Ausreißer nach unten vorweist, scheinbar gibt es ein paar Dateien, die selbst ZIP nicht gut komprimieren kann.

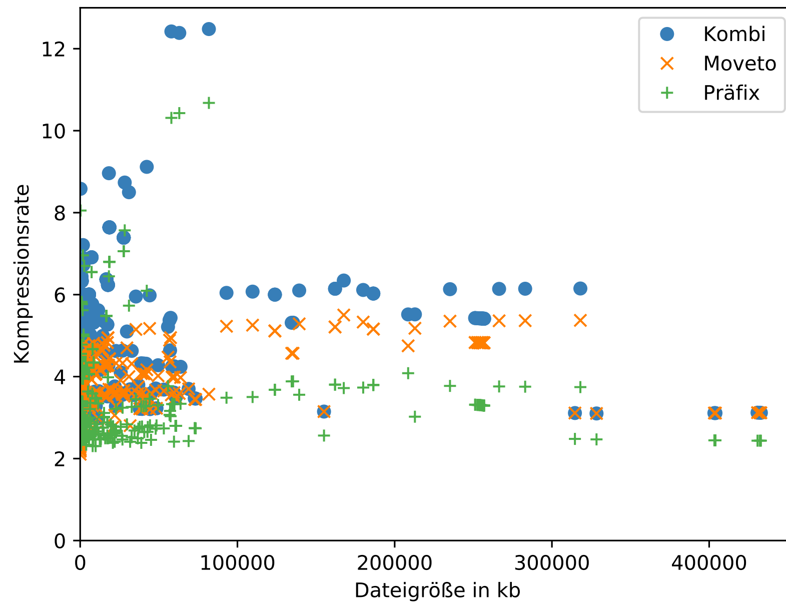

Abbildung 16: Vergleich zwischen Präfix, Move-to-Front und Kombination

Die Kombination ist zwar schlechter als ZIP, ungefähr um Faktor 1,5 (Tab. 4), aber dieser Abstand ist nicht sehr groß. Mit Abb. 18 lässt sich eine Rangliste der Algorithmen bezüglich der Kompressionsrate aufstellen (aufsteigend): Basis, VLQ, Präfix, Move-to-Front, Kombination und ZIP. Das Vorwissen über das Dateiformat hat im Schnitt nicht zu einer besseren Kompressionsrate geführt im Vergleich zu einem universellen Kompressionsverfahren.

Algorithmus	Durchschnitt	Median
Basis	1,55	1,43
VLQ	2,60	2,59
Move-to-Front	3,82	3,81
Präfix	3,35	2,91
Kombination	4,51	4,1
ZIP	6,02	6,0

Tabelle 4: Median und Durchschnitt der Algorithmen bzgl. Kompressionsraten

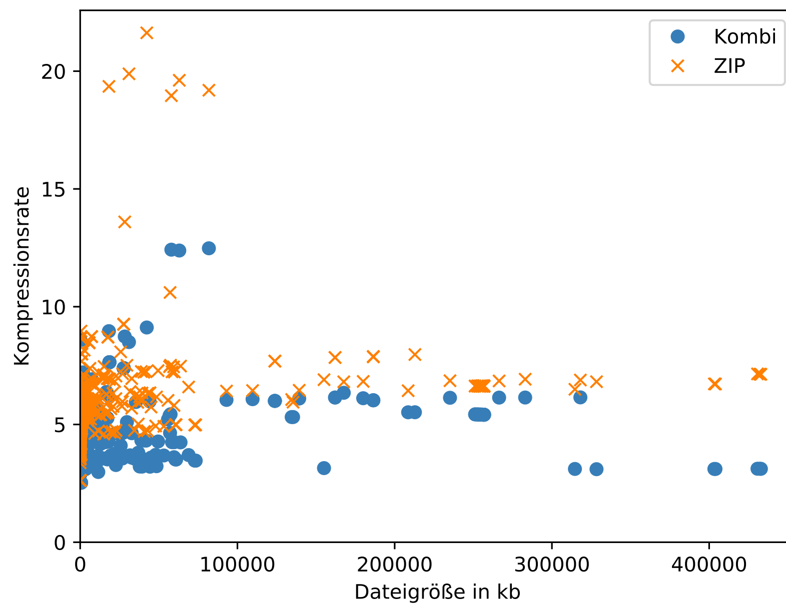

Abbildung 17: Kombination und ZIP im Vergleich

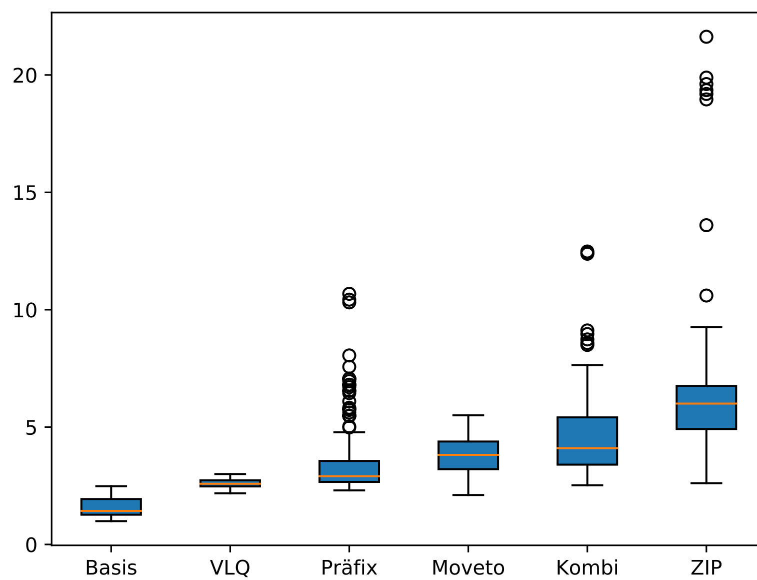

Abbildung 18: Kompressionsraten aller Algorithmen, Gelb Median

6.2 Laufzeit der Algorithmen

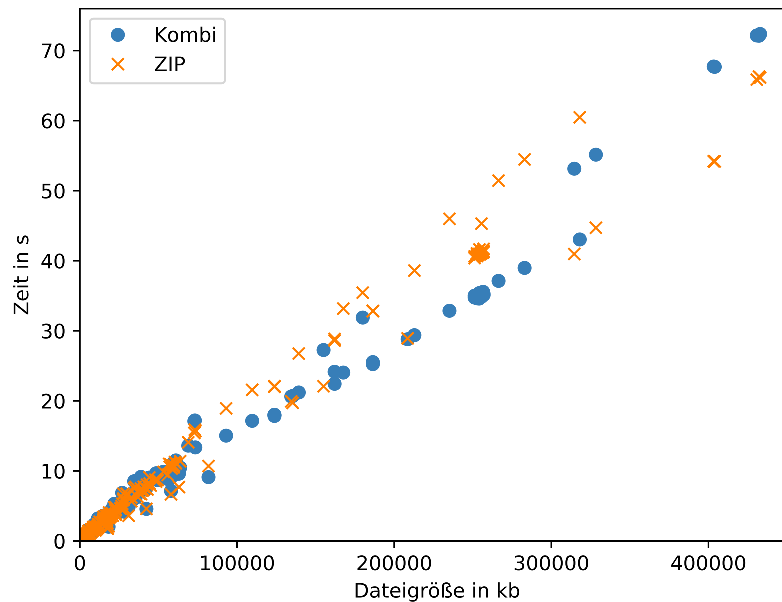

Abbildung 19: Kombination im Vergleich mit ZIP bzgl Kodierungszeit

Da die Kompressionsrate der Kombination am besten war, wird nur noch diese einzeln mit dem ZIP-Format verglichen (Abb. 19). Die meiste Zeit ist die Kombination gleich bis schneller als ZIP. Bei großen Dateien ist ZIP minimal schneller. Insgesamt kann gesagt werden, das beide Algorithmen ungefähr gleich schnell beim Kodieren sind.
Werden jetzt alle Algorithmen verglichen, fällt auf, das ZIP eher zu den langsamen gehört (Abb. 20), während die Kombinationslösung meist relativ nahe an ihren Bestandteilen ist, also Move-to-Front und Präfix. Es auffällig, dass ZIP teilweise langsamer ist, im Bereich zwischen ca. 100000kb und 350000kb, als alle anderen Algorithmen. Das Vorwissen hat sich hier ausgezahlt und zumindest die Kodierungszeit reduziert.

Nach der Kodierung wird die Dekodierung betrachtet, hier wurden ebenfalls die Zeiten gemessen (Abb. 21). Es ist zu erkennen, dass bei allen vorgestellten Algorithmen die Dekodierungszeit linear zur Dateigröße zunimmt. Es ist zu sehen, dass die zunehmende Komplexität Auswirkungen auf die Dekodierungszeit hat. Je komplexer einer der Algorithmen ist, desto länger braucht er beim Dekodieren. Das ZIP-Format hat die mit Abstand schnellste Dekodierung, sie liegt relativ konstant bei wenigen Sekunden, selbst für die größten Dateien.
Kombination und Präfix waren am langsamsten und sind Move-to-Front und Variable liegen dazwischen. Hier hat die zunehmende Komplexität zu einer steigenden Dekodierungszeit geführt.

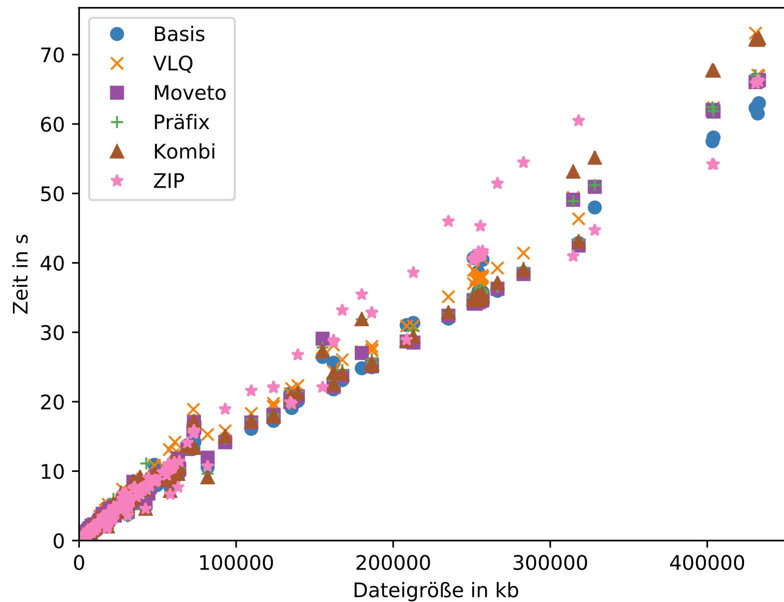

Abbildung 20: Alle Kodierungszeiten im Vergleich

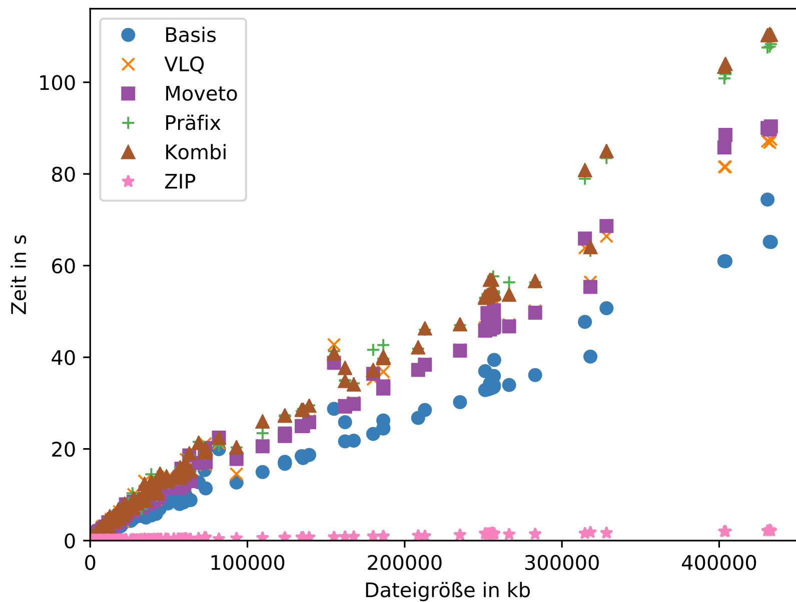

Abbildung 21: Alle Dekodierungszeiten im Vergleich

6.3 Move-to-Front Listenlängen

Es wurden verschieden lange Listen von Move-to-Front getestet. Eine frühere Version wurde verwendet, vor der Optimierung, da dort die Länge der Liste einfacher verändert werden konnte. Diese Tests sollten herausfinden, wie groß der Unterschied in der Kompressionsrate von der Listenlänge abhängig ist. Es wurde eine kleinere Menge von Testdaten verwendet.

In Tab. 5 sind die durchschnittlichen Kompressionsraten dargestellt. Es ist zu erkennen, dass

Listenlänge	Kompressionsrate	Kodierungszeit	Dekodierungszeit
16	3,36	6,44	8,68
32	3,45	6,55	8,16
64	3,61	6,56	8,02
128	3,71	6,34	8,19
255	3,85	6,45	7,91

Tabelle 5: Listenlänge und dazugehörige durchschnittliche Kompressionsrate, Kodierungszeit und Dekodierungszeit

eine längere Liste eine höhere Kompressionsrate mit sich bringt. Allerdings steigt die Kompressionsrate viel langsamer als die Listengröße: Während die Kompressionsrate sich maximal um 0,5 unterscheidet, hat die Listenlänge einen Faktor von 16. Die beste Kompressionsrate ist immer noch niedriger als der Durchschnitt der verbesserten Version, welcher 3,98 ist, somit hat es auch Auswirkungen auf die Kompression, was in die Liste aufgenommen wird.

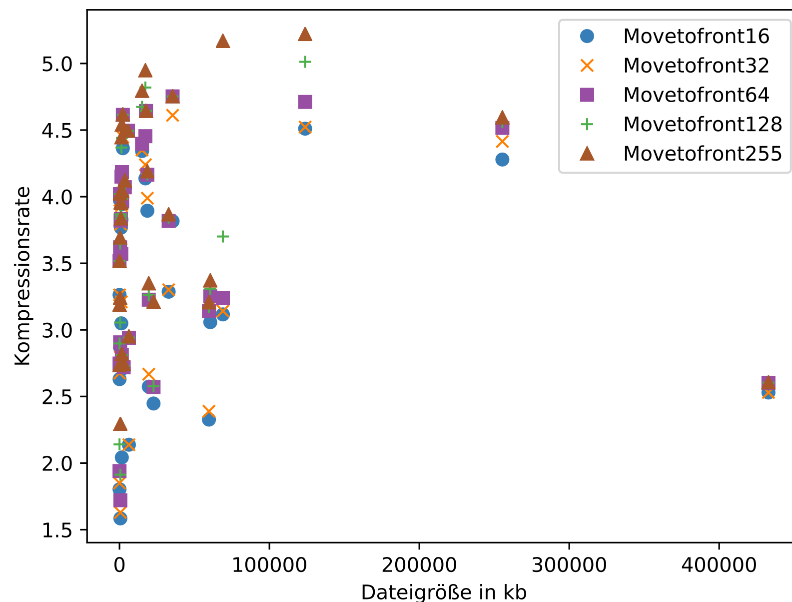

Abbildung 22: Kompressionsraten für einzelnen Dateien, bei unterschiedlicher Listenlänge

Im direkten Verglich zwischen allen Listenlängen in Abb. 22 fällt auf, dass der Unterschied sehr von Datei zu Datei schwankt. Bei der größten getesteten Datei ist fast kein Unterschied zu erkennen, diese Datei gehört zu dem Problemtyp, welcher schon in Kapitel 6.1 schlechte Kompressionsraten hatte. Währenddessen verdoppeln sich bei manchen Dateien die Kompressionsrate bei einer Verlängerung der Liste von 16 auf 254.
In Abb. 23 und Tab. 5 ist zu erkennen, dass die Länge der Liste keinen signifikanten Unterschied bei den Kodierungs und Dekodierungszeiten spielt. Wird jetzt der Move-to-Front mit Listenlänge 255 mit dem verbesserten Move-to-Front, Listenlänge 64, siehe Kapitel 4.3, verglichen, ist zu erkennen, dass der verbesserte leicht besser ist (Tab. 4, Tab. 5). Trotz der weitaus kleineren Liste erreicht der verbesserte Move-to-Front eine höhere Kompression; dies liegt daran, dass nur Zahlen in die Liste aufgenommen werden, bei welchen eine Kompression stattfinden kann. Kleine Zahlen für die keine Kompression mit Move-to-Front erzielt werden kann, werden nicht

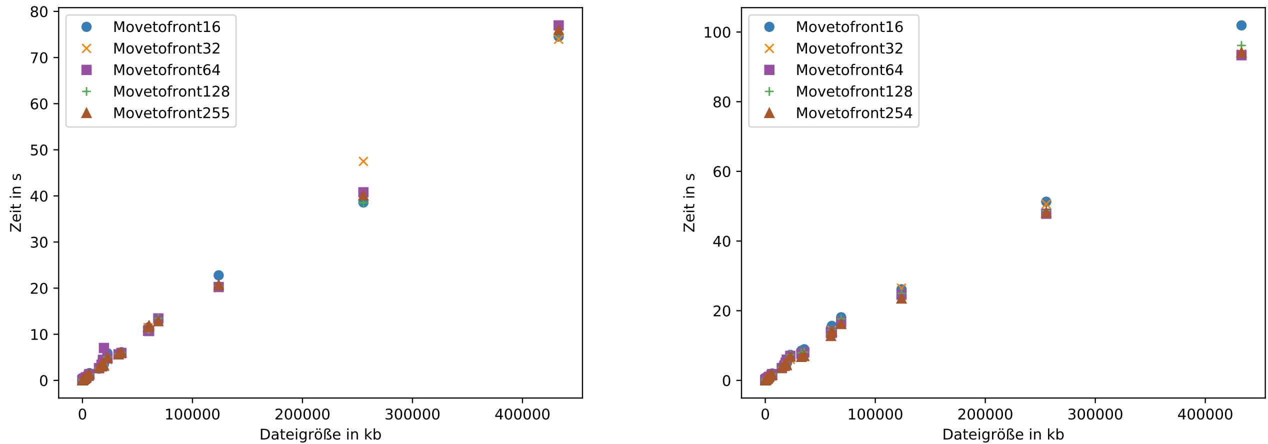

Abbildung 23: Links Kodierungszeiten, rechts Dekodierungszeiten in Abhängigkeit von der Listenlänge

in der Liste gespeichert. Somit ist bei einem Move-to-Front-Algorithmus sowohl die Listengröße als auch der Inhalt in der Liste entscheidend.

6.4 Doppelte Kodierung

Doppelte Kodierung heißt in diesem Fall, dass die Ursprungsdatei als erstes mit einem der hier diskutierten Verfahren kodiert wird und danach nochmals mit ZIP. Dieser Test soll herausfinden, wie es am sinnvollsten ist, die Dateien langfristig zu speichern. Die Benchmarkdateien werden beispielsweise in einem Archiv gespeichert. In diesem Archiv sind die einzelnen Dateien ebenfalls archiviert.
Die Test wurden nur auf den SAT-Competion [9] Dateien ausgeführt. Mit welchen Algorithmus zuerst kodiert wurde, ist den Grafiken und Tabellen zu entnehmen. In Abb. 24 ist zu erkennen, dass alle Algorithmen zu einer Steigerung in der Kompressionsrate führen. Die Verbesserungen liegen im Bereich von ca. Faktor zwei bis ca. Faktor vier, siehe Tab. 4 und Tab. 6. Der Präfix-Algorithmus ist auch noch nahe an VLQ, allerdings hat er sehr viele hohe Ausreißer. Kombination und Move-to-Front verhalten sich bei doppelter Kodierung ziemlich ähnlich bezüglich der Kompressionsrate. Interessant ist, das ZIP schlechter abschneidet als Move-to-Front oder die Kombination. Dies ist naheliegend, da dasselbe Kompressionsverfahren zweimal angewendet wird. ZIP hat allerdings ein paar Ausreißer, welche signifikant höher sind als alle anderen Ausreißer. Es ergibt somit Sinn bei einer doppelten Kodierung zwei unterschiedliche Algorithmen zu verwenden. Auffällig ist, das sich Algorithmen, die Move-to-Front verwenden, besonders gut durch ZIP komprimieren lassen. Dies kann unter anderen mit den gespeicherten Indizes der Listen zu tun haben, was ZIP scheinbar sehr gut komprimieren kann. VLQ und die Bitvektoren lassen sich kaum besser mit ZIP kodieren als Zahlen, wie aus dem Abstand von Basis und Variable heraus geht.

Bei den Laufzeiten, siehe Abb. 25, ist zu erkennen, dass diese sich deutlich von der ersten Kodierung unterschieden, siehe Abb. 20 und Abb. 21. Im ersten Durchlauf der Kodierung steigen alle Zeiten ungefähr gleich schnell. Im zweiten Durchlauf ist das nicht mehr der Fall, sie steigen weitaus langsamer. Bei den Kodierungszeiten und Dekodierungszeiten sticht vor allem der Basisalgorithmus heraus, da dieser durchweg länger benötigt um nochmal kodiert bzw. dekodiert zu

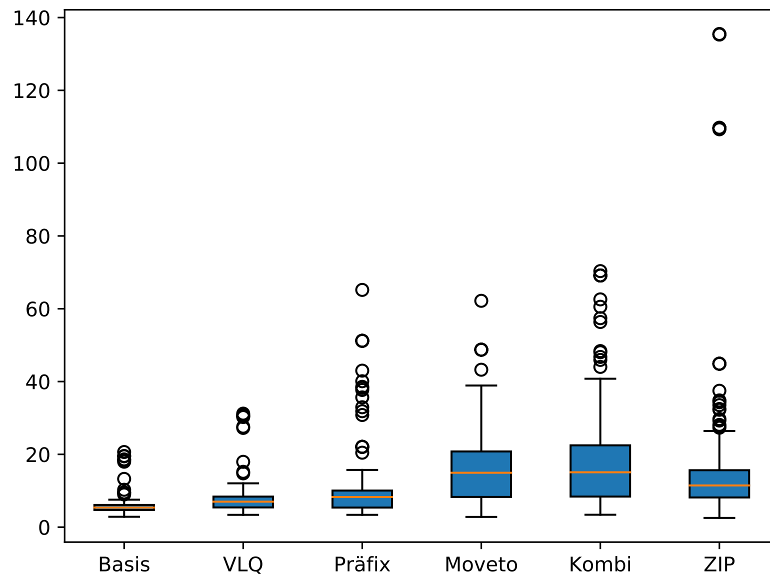

Abbildung 24: Kompressionsraten der doppelten Kompression im Vergleich zur Originaldatei

Algorithmus	Durchschnitt	Median
Basis	5,61	5,38
VLQ	7,29	6,99
Move-to-Front	16,19	14,94
Präfix	9,33	8,28
Kombination	17,77	15,1
ZIP	15,41	11,46

Tabelle 6: Median und Durchschnitt der Algorithmen bzgl. Kompressionsraten

werden. Des weiteren sind Move-to-Front und Kombination beim zweiten Kodieren langsamer, allerdings sind sie beim Dekodieren gleich bis schneller als ZIP, dies trifft hauptsächlich auf den Bereich von 10000 bis 30000kb zu.
Eine doppelte Kodierung verringert somit den benötigten Speicherplatz, vor allem wenn zwei unterschiedliche Algorithmen genutzt werden, allerdings geht dies auf Kosten der Laufzeit.

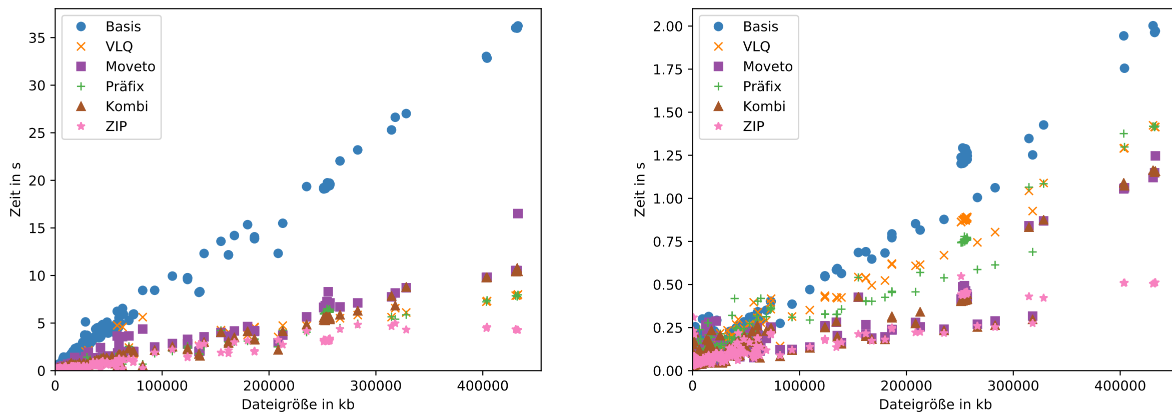

Abbildung 25: Links Kodierungszeiten, rechts die Dekodierungszeiten bei doppelter Kompression, Zeiten sind exklusive der Zeiten für die erste Kodierung, die Dateigrößen sind die der unkodierten Dateien

7 Schlussfolgerungen

In diesem Kapitel wird die Arbeit abschließend zusammengefasst, die Evaluation bewertet und ein Ausblick gegeben.

7.1 Zusammenfassung

In dieser Arbeit wurden verschiedene Kompressionsalgorithmen für das DIMACS-Format vorgestellt, implementiert und mit ZIP verglichen. Es wurden Tests mit verschiedenen Benchmarkdateien, verschiedenen Listengrößen bei Move-to-Front und einer doppelten Kodierung durchgeführt.
Die Algorithmen sind: eine einfache Umwandlung der Eingabedatei in Integers mit fixed-length Kodierung, genannt Basis, dann ein Algorithmus, welcher die Zahlen mit Variable-Length Quantity speichert und die Negationen in einem Bitvektor auslagert, genannt VLQ, darauf aufbauend ein Algorithmus der gleiche Präfixe der Zeilen effizienter kodiert, danach, ebenfalls auf VLQ aufbauend ein Move-to-Front-Algorithmus und als letztes eine Kombination aus dem Präfix- und Move-to-Front-Algorithmus.
Diese wurden dann mit ZIP verglichen bezüglich der Kompressionsrate, Kodierungszeit und Dekodierungszeit.
Von den vorgestellten Algorithmen hat die Kombination die beste Kompressionsrate, danach Move-to-Front, Präfix, Variable und als letztes Basis. Mit zunehmender Komplexität ist die Kompressionsrate gestiegen. Aber es ist auch zu erkennen, dass die Datei an sich entscheidend ist für die Kompression. Die Kompressionsraten von ZIP werden mit den vorgestellten Ansätzen im Allgemeinen nicht erreicht; das Vorwissen über das Dateiformat hat nicht zu einer unmittelbar besseren Kompression geführt. Bei den Kodierungszeiten sind alle Algorithmen relative nahe beieinander. Beim Dekodieren sieht das Bild anders aus, hier war ZIP der schnellste, die anderen erheblich langsamer; das Vorwissen konnte also teilweise genutzt werden, um zumindest die Kodierungszeit zu verringern.
Bei den Tests mit verschiedenen Listengrößen bei Move-to-Front hat die längste Liste die beste Kompressionsrate. Bei den Laufzeiten sind alle gleich auf. Unterschiede zwischen den verschiedenen Listengrößen, bezüglich der Kompressionsrate sind vorhanden und sehr von der Datei abhängig, bei manchen Dateien gab es keinen Unterschiede. Ebenfalls gibt es einen Unterschied bei den Laufzeiten, allerdings ist dieser nicht signifikant. Bei dem Vergleich mit dem verbesserten Move-to-Front zeigt sich, dass es auch von Bedeutung ist, was in der Liste gespeichert wird.
Als letztes wurde eine doppelte Kodierung durchgeführt, bei welcher einer der Ansätze als primäre Kodierung und anschließend ZIP als erneute Kodierung verwendet wurde. Hier haben Move-to-Front und die Kombination die besten Kompressionsraten, eine doppelte ZIP-Kompression bewegt sich unterhalb des Medians der beiden. Bei den Kodierungszeiten benötigt die Basis am längsten, am schnellsten ist hier wieder ZIP. Bei der Dekodierung ist die Laufzeit von Basis wieder am langsamsten, am schnellsten sind ZIP und die Kombination, der Rest liegt dazwischen. Somit hat die Vorarbeit der Verfahren Kombination und Move-to-Front bei einer doppelten Kodierung Vorteile bezüglich der Kompressionsrate, im Vergleich zu einer (doppelten) ZIP-Kompression.
In dieser Arbeit ist also ein Kompressionsverfahren entstanden, das unter bestimmten Bedingungen schneller als ZIP komprimiert, aber nicht ganz dessen Kompression erreicht. Bei einer doppelten Kodierung ist das Verfahren leistungsfähiger als ZIP.
In der vorliegenden Form könnten zwei der Algorithmen verwendet werden.
Und zwar mit einer doppelten Kodierung können sowohl Move-to-Front als auch die Kombina-

tion genutzt werden. Es ist allerdings besser, wenn nur die Kombination verwendet wird, da diese im gesamten besser ist als Move-to-Front. Dies kann beispielsweise für die langfristige Speicherung der DIMACS-Dateien verwendet werden. Ebenfalls können die Algorithmen für die Datenübertragung genutzt werden, da dort die Größe der zu verschickende Daten entscheidend ist, zum Beispiel innerhalb eines Netzwerks in einem HPC-Systems. Somit ist ein Teil der ursprünglichen Motivation erfüllt worden.

7.2 Ausblick

In zukünftigen Arbeiten könnte der Kombinations-Ansatz verbessert werden. Die Dekodierung kann optimiert werden, sodass die Laufzeit näher an aktuell genutzten Verfahren ist. Des weiteren könnte auch ein anderer Ansatz genutzt werden um die Integers zu kodieren, als Variable-Length Quantity. Das Einbinden eines neuen Verfahrens stellt für den allgemeinen Ansatz kein Problem dar. Des weiteren kann erprobt werden, ob durch eine Umordnung der Variablen und Klauseln eine höhere Kompressionsrate erreicht wird.
Für die praktische Verwendung der entwickelten Verfahren bietet sich eine direkte Einbindung in SAT-Solver an. Dies bedeutet, dass die SAT-Solver ihre Ausgabedateien anstatt in DIMACS direkt komprimiert ausgeben und so auch wieder einlesen können. Das kann entweder direkt im SAT-Solver geschehen oder als Plugin eingebunden werden.

Literatur

[1] Association, The International MIDI: *Standard MIDI-File Format Spec 1.1.* http://www.music.mcgill.ca/~ich/classes/mumt306/midiformat.pdf.

[2] Audemard, Gilles und Laurent Simon: *Predicting Learnt Clauses Quality in Modern SAT Solvers.* IJCAI International Joint Conference on Artificial Intelligence, Seiten 399–404, 07 2009.

[3] Balyo, Tomáš und Dominik Schreiber: *Aquaplanning: Quick Automated Planning.* https://github.com/domschrei/aquaplanning. Abgerufen: 2019-03-15.

[4] Daivd Salomon, Giovanni Motta: *Handbook of Data Compression.* Springer-Verlag London, 2010.

[5] Davis, Martin und Hilary Putnam: *A Computing Procedure for Quantification Theory.* J. ACM, 7(3):201–215, Juli 1960.

[6] Deutsch, Peter: *Satisfiability Suggested Format.* http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf.

[7] DIMACS: *DEFLATE Compressed Data Format Specification version 1.3.* https://tools.ietf.org/html/rfc1951#section-Abstract.

[8] Eén, Niklas und Niklas Sörensson: *An Extensible SAT-solver.* Springer Berlin Heidelberg, 2004.

[9] Heule, Marijn JH, Matti Juhani Järvisalo, Martin Suda et al.: *Proceedings of SAT Competition 2018*, 2018.

[10] Huffman, David A.: *A Method of the Construction of Minimum-Redundancy Codes.* Proceedings of the I.R.E., Seiten 1098 – 1101, 1952.

[11] Jacob Ziv, Abraham Lempel: *A Universal Algorithm for Sequential Data Compression.* IEEE Transactions on Information Theory, Seiten 337 – 343, 1977.

[12] Jacob Ziv, Abraham Lempel: *Compression of Individual Sequences via Variable-Rate Coding.* IEEE Transactions on Information Theory, Seiten 5330 – 536, 1978.

[13] Lindner, Paul: *Technische Spezifikation ZIP.* https://www.iana.org/assignments/media-types/application/zip.

[14] Marques Silva, J. P. und K. A. Sakallah: *GRASP-A new search algorithm for satisfiability.* In: *Proceedings of International Conference on Computer Aided Design*, Seiten 220–227, Nov 1996.

[15] Wei, Jon Louis Bentley; Daniel D. Sleator; Rober E. Tarjan; Victor K.: *Adaptive-data-compression.* Communications of the ACM Volume 29, Seiten 320–330, 1986.