technische universität
dortmund

Bachelorarbeit

Approximation der LZ-Zerlegung

Christoph Darms
175259
12. Mai 2021

Gutachter:
Prof. Dr. Johannes Fischer
M.Sc. Patrick Dinklage

Technische Universität Dortmund
Fakultät für Informatik
Algorithmic Foundations and Education in
Computer Science
http://ls11-www.cs.tu-dortmund.de

Inhaltsverzeichnis

1. Einleitung

Das Ziel dieser Arbeit war die Implementation des in *Approximation LZ77 via Small-Space Multiple-Pattern Matching* beschriebenen Algorithmus zur Approximierung der *LZ77-Faktorisierung* [8]. Außerdem sollte eine Auswertung der entstandenen Implementation erfolgen. Den Abschluss der Arbeit bilden ein Fazit zum Verlauf der Arbeit und ein Ausblick hinsichtlich der Verbesserungen die an der Implementation noch möglich sind.

1.1 Motivation und Relevanz

Kompression ist ein zentraler Bestandteil moderner Computersysteme. Das Komprimieren von Dateien spart Speicherplatz und dadurch Hardwarekosten. An Videodaten ist dies besonders deutlich zu erkennen. Eine Sekunde unkomprimiertes Videomaterial in einer üblichen Auflösung von 1920 x 1080 Pixel mit Standard 60 *frames-per-second* und einer Farbtiefe von 24Bit belegt alleine 2.98GB. Komprimiert mit dem *H.264*-Standard belegen die gleichen Informationen nur 0.01GB Speicherplatz [1].
Ein weiterer wichtiger Anwendungsfall für die Kompression ist die Übertragung von Daten. Zugänge zu Netzwerken sind meist in ihrer Geschwindigkeit oder in ihrem Datenvolumen begrenzt. Die Möglichkeit Daten vor dem Übertragen zu komprimieren erhöht damit die Menge an Informationen, die wir über ein Netzwerk übertragen können. Für mobile Geräte und Streamingdienste ist dies von enormer Bedeutung.
IoT-Geräte wie *smartsensors* und *embedded systems* produzieren Daten und senden diese über ein Netzwerk. Um die Auslastung des Netzwerkes zu verringern ist es sinnvoll, dass bereits diese ihre Daten komprimieren [25] [14]. Diese Geräte sind aber oft in Rechenleistung und Speicher begrenzt, um dennoch eine Kompression ausführen zu können, braucht man spezielle Kompressionsalgorithmen die dies berücksichtigen.
LZ77 ist ein verlustfreier Kompressionsalgorithmus mit weitem Einsatzgebiet. *LZ77* eliminiert sich wiederholende Zeichenketten und ersetzt diese durch Verweise auf identische vorherige Ketten. Die so entstandene Struktur aus Verweisen und Zeichenketten ermöglicht es die gleichen Daten komprimiert darzustellen [27].

Seit mehr als 40 Jahren ist *LZ77* die Grundlage für eine ganze Familie an klassischen und modernen Kompressionsverfahren [22] [2] [9]. So basieren zum Beispiel die bekannten Formate *png*, *zip* darauf, dass *LZ77* als Teilschritt angewendet worden ist [6] [5]. Die effiziente Berechnung der *LZ77-Faktorisierung* ist daher von großer Bedeutung.
Es existiert eine Vielzahl an Möglichkeiten eine *LZ77-Faktorisierung* zu generieren. Bekannte Ansätze sind, die Eingabe in *chunks* aufzuteilen, ein *sliding-window* zu benutzen, der Einsatz von *suffix-trees* oder Graphentheorie [5] [20] [21] [17]. Des Weiteren existieren Designs für parallele und verteilte Algorithmen [4] und Konzepte die besonders sparsam mit Ressourcen umgehen [12][26].
Der in *Approximation LZ77 via Small-Space Multiple-Pattern Matching* vorgestellte Algorithmus ist besonders sparsam im Hinblick auf den zur Laufzeit benötigten Speicher [8]. Da der Speicher zur Laufzeit ein allgemeiner Flaschenhals heutiger Systeme darstellt, ist eine Approximation hier sinnvoll, um bei limitierten Ressourcen ein bestmögliches Ergebnis zu erzielen [8].

1.2 Ziele und Arbeitsverlauf

Innerhalb dieser Arbeit beschreibe ich meine Umsetzung des Algorithmus in *C++* [24]. Die erarbeitete Implementation ist in das *TU Dortmund Compression Framework* integriert.
Diese Arbeit konzentriert sich auf die ersten beiden Phasen des Algorithmus. Die dritte Phase des Algorithmus basiert auf einer komplexen Suche, die den Rahmen einer Bachelorarbeit überschreiten würde.
Zunächst führe ich angewendete Konzepte und Fachbegriffe ein.
Danach folgt eine Beschreibung der Transformation des Algorithmus in funktionierenden C++ Code. Im gleichen Abschnitt erkläre ich im Detail an Hand von Pseudocode wie die Implementation funktioniert.
Nach den Implementationsabschnitt erfolgt eine statistische Aufarbeitung, Auswertung und Visualisierung der Testergebnisse. Zunächst werden Testumgebung und als Eingabe dienende Datensätze gewählt. Darauf erfolgt eine Auswertung der Ergebnisse von allen Datensätze hinsichtlich Laufzeit, Speicher und Kompressionsgrad. Verschiedene Versionen werden miteinander verglichen und ausgewertet.
Als Vergleichswerte werden ebenfalls Daten zu den im *TU Dortmund Compression Framework* vorhandenen Algorithmus *lzlss_cp* und den Drittprogramm *gzip* erhoben.
Eine Fazit und ein Ausblick bildet den Abschluss der Arbeit.

2. Grundlagen

In diesem Kapitel werden Grundbegriffe und Zusammenhänge der Kompression, Approximation und insbesondere der LZ-Zerlegung erläutert.

2.1 Zahlen

Im Laufe de Arbeit steht $PRIME$ für die Menge der Primzahlen. Eine Zahl x ist eine Primzahl wenn die einzigen beiden Teiler von x die Zahl 1 und x selber sind. Eine Zahl n steht für eine beliebige natürliche Zahl $n \in \mathbb{N}$.

2.2 Kompression

Kompression ist die Transformation von Daten in eine kleinere Darstellung. Es gibt zwei Möglichkeiten diese Transformation auszuführen, verlustfrei oder verlustbehaftet. Alle verlustfreien Ansätze basieren darauf, Redundanzen innerhalb eines Datensatzes zu eliminieren, während verlustbehaftete Methoden ausgewählte Daten löschen. Der bedeutende Unterschied zwischen diesen beiden Ansätzen ist, dass sich nur aus einer verlustfreien Kompression das Original wiederherstellen lässt.
Die Güte einer Kompression, der *Kompressionsgrad*, ist definiert als:

$$\text{Kompressionsgrad} = \frac{\text{Größe der Ausgabe}}{\text{Größe der Eingabe}}$$

Während der *Kompressionsfaktor* das Inverse des Kompressionsgrades ist:

$$\text{Kompressionsfaktor} = \frac{\text{Größe der Eingabe}}{\text{Größe der Ausgabe}}$$

Keine verlustfreie Kompressionsmethode kann einen Kompressionsgrad < 1 garantieren, ein Kompressionsgrad ist immer von der Eingabe abhängig. Sollte theoretisch eine solche Methode existieren, könnte man sie immer wieder auf ihre eigenen Ausgaben anwenden und jegliche Datenmenge in nur einem einzelnen Bit codieren, was offensichtlich unmöglich ist [19].

In dieser Arbeit beschäftige ich mich ausschließlich mit verlustfreien Verfahren, der Begriff der Kompression bezieht sich deshalb immer auf die verlustfreie Kompression.

2.3 Strings

Eine fundamentale Art Informationen darzustellen ist der *String*. Ein *String* w der Länge n ist eine Folge von Symbolen $w[1], w[2], \cdots, w[n-1]$ aus einem Alphabet Σ.
Ein String u der Länge m ist Substring von w, wenn für ein $i < n - m$ gilt:
$u[1]u[2]\cdots u[m-1] = w[i]w[i+1]\cdots w[m-1]$.
Der Substring u besitzt einen *vorherigen Substring* in w, wenn ein $x < i$ existiert mit:
$w[i+0]w[i+1]\cdots w[m-1] = w[x+0]w[x+1]\cdots w[x-1]$.
Der folgende/nächste Substring von $w[i,j]$ ist $u[i+1, j+1]$.
Für Substrings existieren zwei Kategorien mit besonderer Bedeutung, Präfixe und Suffixe. Präfixe sind Substrings, die an Index 1 beginnen, während Suffixe Substrings sind, die an Index n enden. Als *echte Substrings* bezeichnet man Substrings die kürzer sind als der String in dem sie liegen. Im Folgenden soll *Substring*$[i,j]$ den Substring der an Index i beginnt und an Index j endet bezeichnen.
In *C++* existieren zwei Konzepte, die als String bezeichnet werden, der *C-style char array* und die *std::string* Klasse. Das *C-style char array* ist ein kontinuierlicher Block an Speicher mit fester Größe. In diesem Block liegen *chars*, die Symbole darstellen, direkt hintereinander. Die Größe des *arrays* kann nachträglich nicht verändert werden.
Die *std::string* Klasse spezifiziert Objekte, die ein *char array* enthalten. Objekte der Klasse String bieten weitere Funktionalitäten und die Möglichkeit, die Länge des Strings zu verändern [24].

2.4 LZ-Zerlegung

Eine LZ-Zerlegung ist eine Aufteilung eines String T in sich nicht überschneidende Substrings, diese Substrings bezeichnet man als Faktoren. Das heißt für jeden Faktor $f[i,j]$, also jeden Substring$[i,j]$ in der LZ-Zerlegung $LZ()$ der Eingabe T gilt:

- Die Faktoren überschneiden sich nicht: $T = f_1 f_2 f_3 \cdots f_n$
- Die Faktoren sind einzelne Symbole oder besitzen vorherige Substrings
- Die Faktoren sind in ihrer möglichen Länge maximal, d.h. keine zwei aufeinander folgenden Faktoren können zusammen ein Faktor sein

Die LZ-Zerlegung erlaubt es einen String komprimiert darzustellen. Diese Kompression wird erreicht, indem in der LZ-Zerlegung Faktoren durch Referenzen ersetzt werden. Jede Referenz ist ein Tupel aus zwei Zahlen, das einen vorher liegenden Substring identifiziert.

Das erste Element des Tupels enthält Informationen über die Position des vorher liegenden Substrings, während das zweite Element die Länge des Substrings angibt. Die Positionsangabe kann dabei ein *offset* oder ein absoluter Index über den faktorisierten String sein. Ein *offset* ist meist aber kleiner, da für größere Strings ein absoluter Index $\lceil \log_2 n \rceil$ Bits belegt (wobei $n = |T|$), wohingegen ein offset nur im *worst case* genau so groß ist.
Allerdings werden nicht alle Substrings durch Referenzen ersetzt. Wenn die zugehörige Referenz mehr Speicher benötigt als der Faktor selbst, lohnt es sich nicht den Substring zu ersetzten. Deshalb wird auch keine Referenz gespeichert, wenn der Faktor ein einzelnes Symbol ist [27].

2.5 Approximation

Zu jedem Optimierungsproblem existiert eine Menge an richtigen Lösungen und in dieser eine Teilmenge an optimale Lösungen. Ein Approximationsalgorithmus erzeugt für eine Eingabe eine richtige Lösung, die aber nicht zwangsläufig optimal sein muss.
Die *Approximationsgüte* ρ ist eine Garantie, wie sehr die erzeugte Lösung s_e maximal von einer optimalen Lösung s_{opt} abweicht. Um nichtnumerische Ergebnisse vergleichen zu können, evaluiert man beide durch eine Bewertungsfunktion *f(s)* [11]. Dann ist

$$\rho = max \left\{ \frac{f(s_e)}{f(s_{opt})}, \frac{f(s_{opt})}{f(s_e)} \right\}$$

Da eine Abweichung vom Optimum sowohl größer als auch kleiner als eins sein kann, ist der größere der beiden Brüche die Güte.

2.6 LZ-Approximation

Zu jedem beliebigen String existiert genau eine LZ-Zerlegung. Eine Approximation dieser Zerlegung besteht, wenn die Faktoren in ihrer Länge nicht zwingend maximal sind. Die Faktoren der Approximation sind weiterhin entweder einzelne Symbole oder besitzen vorherige Substrings.
Eine *c-optimalen Approximation* besteht dann, wenn keine c aufeinander folgenden Faktoren selber einen Faktor bilden. Daraus folgt, dass in einem Faktor der LZ-Zerlegung maximal c-1 ganze Faktoren liegen können. Es kann aber sein, dass sowohl ein echtes Suffix und ein echtes Präfix der umliegenden Faktoren ebenfalls in dem Faktor der LZ-Zerlegung enthalten sind. Im ungünstigsten Fall wird damit ein Faktor vollkommen zwischen zwei Faktoren der LZ-Zerlegung aufgeteilt (vgl. Abb. 2.1).
Das bedeutet, dass wir maximal c-mal mehr Faktoren in der *c-optimale* LZ-Approximation finden als in der LZ-Zerlegung. Die Anzahl der Faktoren in der Approximation ist also durch

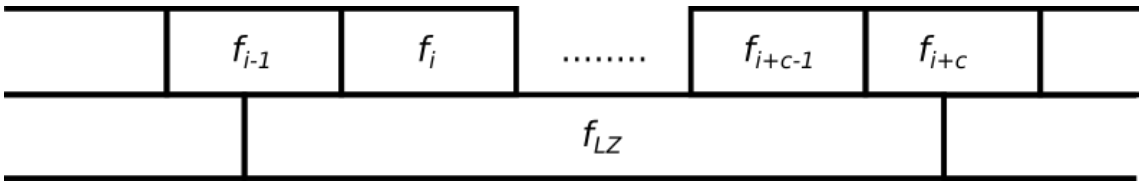

Abbildung 2.1: Der Faktor f_{LZ} erstreckt sich von f_{i-1} bis f_{i+c}

die Anzahl der Faktoren in der LZ-Zerlegung multipliziert mit c nach oben beschränkt.

$$\rho = max \left\{ \frac{c * f(s_{opt})}{f(s_{opt})}, \frac{f(s_{opt})}{c * f(s_{opt})} \right\} = c$$

Aus einer *c-Optimalität* der Approximation folgt eine Güte der Approximation von c [8].

2.7 Hashfunktionen

Eine *Hashfunktion* h ist eine Funktion von einem *Universum* U auf eine beschränkte Teilmenge der natürlichen Zahlen $h : U \to \{0, 1, 2 \cdots, n\}$. Die Ausgaben bezeichnet man als Schlüssel oder auch als *Hashwert/Hashcode*, eine Hashfunktion definiert damit *key value pairs*. Wenn unterschiedliche Werte den gleichen Hashcode besitzen, bezeichnet man dies als *Kollision*. Wenn die Menge der Eingaben größer ist als die Menge der Hashcodes sind Kollisionen unvermeidlich.
Eine Hashfunktion wird anhand folgender zentraler Kriterien bewertet:

- Verteilung: Alle möglichen Hashwerte sollen gleich wahrscheinlich getroffen werden
- Dichte: Die Menge aller Hashwerte soll möglichst klein sein
- Kollisionswahrscheinlichkeit: Kollisionen sollen möglichst gering sein

Diese Güten einer Hashfunktion müssen allerdings im Hinblick auf die Eingabewerte betrachtet werden. Für Eingabemengen mit sehr kleinem Alphabet z.B. DNA mit {G,A,T,C} können Funktionen gut funktionieren, die für Alphabete mit z.B. allen möglichen Werten pro Byte sehr oft zu Kollisionen führen[15].

2.8 Hashmaps

Die *Hashmap* ist eine Datenstruktur die *Lookups*, das Einfügen und das Löschen in erwartet konstanter Zeit ermöglicht. Dies ist einer der häufigsten Anwendungsfälle der Hashfunktionen. Beim Einfügen eines Elementes e wird der Hashwert von e, $h(e)$, berechnet und e unter Index $h(e)$ abgespeichert, dabei kann es hier zu einer Kollision kommen. Um

zu überprüfen, ob ein Element e in der Hashmap enthalten ist, genügt es den Hashwert zu berechnen und nur den so erhaltenen Index zu untersuchen. Das Löschen eines Elementes verläuft ebenso, außer dass nach dem Auffinden das entsprechende Element gelöscht wird. Ein Speicherplatz in einer Hashmap bezeichnet man auch als *Bucket*.
Um mit Kollisionen effizient umzugehen existieren zahlreiche Ansätze, zu den bekanntesten zählen:

- *chaining*: die verschiedenen Elemente einfach in einer Liste zu verketten
- *double hashing*: hinter jedem Index eine zweite Hashmap zu bilden
- *linear probing*: das Element unter dem nächsten freien Index zu speichern
- *quadratic probing*: das Element unter dem doppelten Index zu speichern

Alle Kollisionsresolutionsmethoden haben ihre eigenen Vor- und Nachteile [15][23].

2.9 Fingerprint

Als *Rabin-Karp-Fingerprint*, im Folgenden einfach Fingerprint, bezeichnet man den Hashcode eines Strings. Fingerprints erlauben es, Strings schneller zu vergleichen. Um herauszufinden ob zwei Strings s_1, s_2 gleich sind müssen wir alle Symbole miteinander vergleichen. Sollten die beiden Strings sich nur im letzten Symbol unterscheiden, so haben wir alle vorherigen Symbole umsonst überprüft.
Vergleichen wir aber vorher beide Fingerprints können wir ungleiche Strings direkt daran erkennen. Dies ist wesentlich effizienter, da Fingerprints meist deutlich kürzer sind als die Strings selber. Es kann aber sein, dass durch eine Kollision unterschiedliche Strings den gleichen *Fingerprint* besitzen. Daher reicht es nicht aus, nur die Fingerprints zu vergleichen. Wenn zwei Fingerprints gleich sind, müssen danach alle Symbole der Strings untersucht werden[13][15].

2.10 rolling Hash

Ein *rolling Hash* ist eine rekursive Methode, um aus dem Fingerprint eines Substrings w den Fingerprint des nachfolgenden Substrings u zu erhalten.
Wenn wir einen Text T durchsuchen, müssen wir für jede Position den Hashwert des Substrings bilden. Da die benötigte Zeit um einen Fingerprint zu berechnen von der Länge des Substrings abhängt, kann dies für lange Substrings zeitintensiv sein. Ein rolling Hash beschleunigt die Berechnung von Hashwerten aufeinander folgender Substrings, indem er ausnutzt, dass sich die Substrings nur in zwei chars unterscheiden. In einem rolling Hash entfernen wir sozusagen den nicht mehr vorhandenen char aus dem Hashwert und fügen den neuen char hinzu [13].

Allerdings eignen sich nicht alle Hashmethoden für einen rolling Hash. Um einen rolling Hash zu ermöglichen, muss die Hashfunktion folgende Eigenschaften aufweisen:

$$h(u) = h(w) - f(w_0) + f(u_m)$$

Wobei m die Länge der Substrings und $f()$ die Transformation der einzelnen *chars* ist. Diese Eigenschaft erfüllen nicht alle Hashfunktionen. Im Pearson Hash wird aus einem String lediglich ein Index für eine Substitutionstabelle berechnet, die die eigentlichen Hashwerte enthält [18]. Da diese Substitution nicht rückführbar ist, ohne die ganze Tabelle zu durchsuchen, eignet sich diese Hashfunktion nicht als rolling Hash.

2.10.1 Polynomen Hash

Ein *Polynom Hash* ist eine intuitive Methode, um den Fingerprint eines Strings zu berechnen. Ein Polynomen Hash besteht aus drei Parametern:

- eine Funktion die jedem Symbol des Eingabealphabetes einen numerischen Wert zuordnet $f()$
- eine Primzahl p als Potenzbasis
- die Länge der zu hashenden Strings l

Der Hashwert eines Strings w wird wie folgt berechnet:

$$Hash(w) = \sum_{0}^{l} f(w[i]) * p^{l-i}$$

So lässt sich der Fingerprint des nächsten Substrings u wie folgt aus dem Fingerprint des aktuellen Substrings w berechnen:

$$Hash(u) = Hash(w) - f(w[0]) * p^{l} + f(u[l])$$

Da bei langen Substrings *Overflows* (wir erhalten eine Zahl die nicht in unsere Bitdarstellung passt) auftreten können, wenden wir nach jedem Berechnungsschritt eine Modulofunktion an. Zur Vermeidung von Kollisionen ist die Zahl mit der wir Modulo rechnen meist auch eine Primzahl $p_m \in PRIME$ [15][13].

$$Hash(w) = \sum_{0}^{l} (f(w[i]) * p^{l-i}) \% p_m$$

Man muss ebenfalls beachten, dass die Primzahl mit der Länge als Exponent keinen Overflow verursacht. Diesen *Exponentenwert* berechnen wir daher wie folgt:

$$exp_0 = p$$
$$exp_n = (exp_{n-1} \% p_m) * p$$

Die Berechnung endet wenn $n = l$ ist.

Wahl der Primzahlen

Die Wahl geeigneter Primzahlen ist für einen Polynomen Hash sehr wichtig. Sie haben Einfluss auf die Wahrscheinlichkeit einer Kollision und die Streuung der Hashwerte über den Wertebereich. Folgende Grundregeln sind bei der Wahl der Primzahlen zu beachten:

- sei p eine Primzahl die größer ist als der maximale Wert von $f()$
- die Primzahl mit der man Modulo rechnet p_m sollte so groß wie möglich sein
- sei l die Länge der Strings, dann soll gelten dass $p^l * max(f())$ keinen Overflow verursachen

[15][8]

Mersenne und Fermat Primzahlen

Mersenne Primzahlen sind Primzahlen die genau eins weniger sind als eine Zweierpotenz

$$p \in MERSENNEPRIME.p \in PRIME \wedge p = 2^n - 1 \wedge n \in \mathbb{N}$$

Mit diesen Zahlen lässt sich die relativ rechenzeitintensive Modulooperation vereinfachen.

$$z \bmod p = \begin{cases} (z\&p) + (z >> n) - p, p < (z\&p) + (z >> n) \\ (z\&p) + (z >> n), p \leq (z\&p) + (z >> n) \end{cases}$$

Dieses Vorgehen kann schneller sein da keine Division ausgeführt werden muss.
Fermat Primzahlen können besonders schnell multipliziert werden. Fermat Primzahlen sind immer genau eins mehr als eine Zweierpotenz. Daher gilt:

$$x \in FERMATPRIME, y \in \mathbb{N}.2^y + 1 = x$$
$$s * x = (s << y) + s$$

2.10.2 Buzhash

Ein *Buzhash* ist eine Variante des Tabellenhashens die lediglich Substitutionen, Verundung, Bitshifts und Antivalenzen anwendet. Dies macht die Berechnung besonders schnell.
Die Grundidee des Buzhashing ist es, den vorhandenen Hash zu rotieren und aus der Rotation und dem neuen char die Antivalenz zu bilden. Die n-te Rotation $rotatation^n(x)$ oder der n-te *cyclic shift* von x: entspricht dem n-fachen Tauschen aller Bits mit ihrem Vorgänger, wobei der Vorgänger des ersten Bits das letzte Bit ist. Um weniger Kollisionen zu erhalten, kann man eine Substitutionsfunktion $f()$ auf die chars anwenden [3][15].

2.10.3 ntHash

Der *ntHash* ist eine Hashfunktion die speziell für das Alphabet der DNA {A,C,G,T} konzipiert wurde [16]. Ein ntHash Fingerprint einer Nukleotidsequenz w mit Länge l wird in drei Schritten berechnet. Als erstes wird jedes Symbol durch ein 64Bit Wert ersetzt, danach rotieren wir diese Werte anhand ihrer Position in der Sequenz. Zum Schluss bilden wir die Antivalenz aus all diesen Werten.

$$Fingerprint(w) =$$
$$rotation^{l-1}(h(w[0])) \oplus rotation^{l-2}(h(w[1])) \oplus \cdots \oplus rotation^{1}(h(w[l-1])) \oplus h(w[l])$$

Aus einem ntHash Fingerprint eines Substrings $T[i,k]$ mit Länge $l = k - i$ von Text T lässt sich leicht der Fingerprint des folgenden Substrings $T[i+1,k+1]$ berechnen. Wir berechnen die l-te Rotation der Substitution des Symbols an Stelle i und die Substitution des Symbols an Stelle $k+1$. Aus diesen drei bilden wir die Antivalenz:

$$Fingerprint(T[i+1,k+1]) =$$
$$rotation^{1}(Fingerprint(T[i,sk])) \oplus rotation^{k}(h(T[i])) \oplus h(T[i+k])$$

Diese Hashfunktion zeichnet sich dadurch aus, dass keine Addition, Multiplikation, Modulo oder Subtraktion ausgeführt werden muss und dass sie besonders wenig Kollisionen erzeugt [16].

3. Technische Beschreibung

In diesem Kapitel beschreibe ich die technische Umsetzung des Algorithmus, in ein konkretes C++ Programm.
Zunächst beginne ich mit einer allgemeinen Beschreibung der Implementation, gefolgt von einer Erklärung aller *Objekte* und der Konzepte die diese umsetzen. Danach erfolgt eine genaue Beschreibung der Phasen und ihrer Teilschritte anhand von Pseudocode. Aus Gründen der Lesbarkeit und um unnötige Wiederholungen zu vermeiden, sind die Erläuterungen zum Umgang mit Kollisionen in den Hashmaps nicht in den Sektionen der Teilschritte, sondern in einem extra Abschnitt am Ende dieses Kapitels.
Die Begriffe ***chain*** und ***group*** bezeichnen die in *Approximation LZ77 via Small-Space Multiple-Pattern Matching* beschriebenen theoretischen Konzepte.
Dagegen beschreiben die Begriffe Chain, Group und Factor die C++ Objekte.

3.1 Implementation

Als Eingabe erhält die Implementation drei Werte:

- einen zu komprimierenden Text T
- eine zweier Potenz als maximale Größe der Faktoren m
- eine zweier Potenz als minimal Größe der Faktoren θ kleiner gleich m

Der Algorithmus erstellt einen vollen Binärbaum über T, wobei jeder Knoten für einen Substring steht, den wir auf vorheriges Vorkommen untersuchen. Knoten mit vorherigen Substrings haben nie Kinder, während Knoten ohne vorherigen Substring immer zwei Kinder haben. Die Kinder eines Knoten symbolisieren das Prä- bzw. Suffix mit je halber Länge des Strings des Elternknotens. Aus diesen zwei Regeln ergibt sich die finale Struktur des Baumes.
In der Implementation beginnen wir damit, die Eingabe T in die maximal größte gerade Anzahl an Teilstücke der Länge m aufzuteilen. Zu jedem dieser Teilstücke erzeugen wir ein Chain Objekt. Diese repräsentieren die Knoten des Baumes auf der Ebene von der aus die weitere Bearbeitung erfolgt. Chains mit geradem Index sind immer linke Kinder und Chains mit ungeradem Index sind immer rechte Kinder. Die maximale Länge dieser Chains

ist 32768, da sich dass Bereitstellen von mehr Speicher für größere Längen experimentell als nicht sinnvoll erwiesen hat.
Die Implementation untersucht alle Chains auf vorherige Substrings. Chains, die nicht gefunden wurden, ersetzen wir durch ihre zwei Kinder, die die Prä- bzw. Suffixe halber Länge darstellen. Ist eine Chain gefunden, löschen wir sie und speichern uns die gefundene Länge am verbleibenden Kind der Eltern Chain. Sollten beide Kinder gefunden worden sein, finden sie bei der weiteren Generierung des Baumes keine weitere Beachtung. Das Programm bricht die Generierung des Baumes ab, sobald die Länge der Chains θ erreicht hat. In der Runde, in der wir Substrings der Länge θ untersuchen, generieren wir keine neuen Chains.
Der Algorithmus teilt nun alle Blätter des Baumes in die sogenannten ***chain*** auf. Als Grenzen der ***chain*** dienen die cherrys, zwei Blätter mit gleichem Eltern Knoten. Eine aufsteigende ***chain*** besteht aus dem rechten Blatt der linken cherry und allen rechten Blättern des größtmöglichen Teilbaumes mit nur einer cherry. Eine absteigende ***chain*** besteht aus dem linken Blatt der rechten cherry und allen linken Blättern des größtmögliche Teilbaumes mit nur einer cherry.

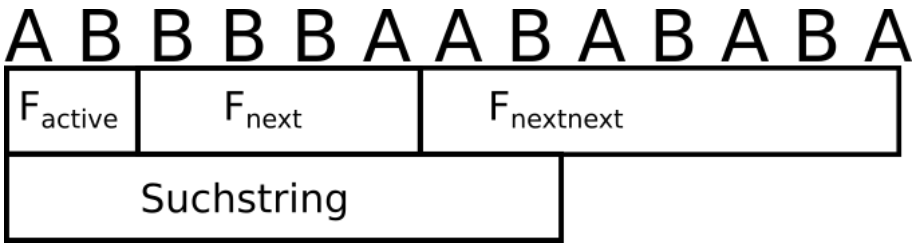

Abbildung 3.1: Entstehung des Suchstrings mit: $|F_{active}|$=2, $|F_{next}|$=4,|Suchstring|=8

Die Implementation generiert aus jeder Chain, zu der mehr als ein gefundener Faktor assoziiert ist, ein Group Objekt.
Danach versucht der Algorithmus die Faktoren innerhalb der ***chain*** ausgehend von dem cherry Blatt zu mergen. Dazu markiert der Algorithmus in jeder ***chain*** das cherry Blatt als aktiv und testet ob es mit dem nächst größeren Faktor vereinigt wird. Das hängt davon ab ob ein vorheriger Substring für den Suchstring existiert. Der Suchstring besteht aus dem aktiven Faktor, dem nächsten Faktor und einem nicht leerem Prä- bzw. Suffix des übernächsten Faktors. Der Suchstring ist immer doppelt solang wie der nächste Faktor.
Sollte so ein vorheriger Substring existieren, mergen wir den aktiven und den nächsten Faktor, dieser neue Faktor ist nun aktiv. Existiert kein solcher Substring markieren wir den nächsten Faktor als aktiv.
Die Implementation folgt genau der zuvor beschriebenen Vorgehensweise. Wir markieren den kleinsten Faktor einer Group als aktiv und versuchen ihn mit dem nächsten Faktor zu mergen. Wir entscheiden dies genauso mit Hilfe eines Suchstrings der doppelt so lang ist wie der nächste Faktor.

Sowohl aus dem Algorithmus, als auch der Implementation erhalten wir so eine Menge an Faktoren, die eine 5-optimale LZ-Approximation darstellt.

3.2 Objekte

Im Folgenden beschreibe ich alle im Laufe der Bachelorarbeit entstandenen oder verwendeten Klassen. C++ ist eine objektorientierte Programmiersprache, deren Vorzüge nur dann verständlich und nachvollziehbar erkennbar sind, wenn darin verwendete Objekt und Klassen transparent dokumentiert werden. Die folgenden Beschreibungen beinhalten auch den in der Schreibweise ausgedrückten Bezug zur Implementation. Doppelbezeichnungen und Begriffe, die sowohl innerhalb der Implementation, wie auch im zu zugrundeliegenden Algorithmus verwendet werden, sind durch verschiedene Schreibweisen kenntlich gemacht.

3.2.1 Chain

Chain Objekte stellen die im Algorithmus beschriebenen aktiven Knoten dar. Jedes Chain Objekt enthält eine Position und eine Zahl die als Bitvektor interpretiert wird. Die Position der Chain gibt immer den Anfang des zu untersuchenden Substrings an. Der Bitvektor gibt an welche gefundenen Längen mit der Chain assoziiert wurden. Zum Beispiel bedeutet ein gesetztes Bit an vierter Stelle, dass ein Faktor der Länge 16 gefunden wurde.
Eine Zahl mit einer Chain zu assoziieren oder eine Zahl zu einer Chain zu addieren, beschreibt das Setzten des entsprechenden Bits im Bitvektor. Dies wird durch einfache Veroderung ausgeführt.
Ein Chain Objekt ist damit 6 Byte groß.

3.2.2 Group

Ein *Group* Objekt stellt die im Algorithmus beschriebenen ***chain*** und einen aktiven Faktor dar. Jedes Group Objekt enthält die Position, die Länge und die Quellposition des aktiven Faktors. Außerdem besitzt eine Group eine Zahl die als Bitvektor interpretiert wird und die die vor- bzw. zurückliegenden Faktoren darstellt. Dabei steht ein gesetztes Bit für einen Faktor der Länge wenn nur dieses Bit 1 wäre. Zusätzlich hat eine Group ein Flagbit, welches kennzeichnet ob es sich um eine aufsteigende oder absteigende ***chain*** handelt.
Eine Group kann sich auf zwei Arten transformieren:

- der nächste Faktor geht in die aktive Gruppe mit ein
- der aktive Faktor wird ausgegeben und der nächste Faktor wird aktiv

Ein Group Objekt ist 14 Byte groß.

3.2.3 Factor

Ein *Factor* ist ein Triple aus drei Zahlen: der Position, der Länge und der Quellposition. Ein Faktor benötigt 12 Byte.

3.2.4 Rollinghash

Ein *Rollinghash* Objekt besteht aus vier Werten: dem Hashwert, der Position, der Länge und dem Exponentenwert. Der Hashwert enthält den Fingerprint des an Position beginnenden Substrings der gespeicherten Länge.
Ein Rollinghash dient dazu während der Suche über die Eingabe alle benötigten Werte zu verwalten.
Ein Rollinghash Objekt ist 24 Byte groß.

3.3 Ablauf

3.3.1 Initialisation

Die Initialisation beginnt damit die Fenstergröße anzupassen. Sollte die maximale Faktorlänge, *windowsize*, größer als der Text sein, so halbieren wir sie solange bis sie kleiner gleich der Textlänge ist. Ist die minimale Faktorlänge, *threshold* θ, größer als *windowsize* m brechen wir das ganze Programm ab. Man könnte zwar einfach die Eingabe T unkomprimiert ausgeben, aber da die Eingabewerte offensichtlich unlogisch sind, ist es sinnvoll hier das Programm mit einem Errorcode zu beenden.
Als nächstes erstellen wir die benötigten Container: einen Vektor für die zu bearbeitenden Chains $chainVec$, einen Vektor für die abgeschlossenen Chains $p2\_buffer$, eine Hashmap für die Suche *hmap*, eine Hashmap für das Speichern der Quellpositionen $temp\_hmap\_store$ und einen Vektor, der diese zweiten Hashmaps speichert $hmap\_storage$.
Danach teilen wir die Eingabe T in Teilstücke der Größe *windowsize* auf. Für jedes dieser Teilstücke, ausgenommen die letzten beiden, erzeugen wir ein Objekt vom Typ Chain und fügen es zu *chainVec* hinzu.

3.3.2 Phase 1

Die erste Phase der Implementation erhält einen Vektor von Chains, die maximale Länge der Faktoren und die minimale Länge der Faktoren.
Als Ausgabe aus der ersten Phase erhalten wir zwei Vektoren aus Chains und einen Vektor von Hashmaps, die mit allen gefundenen Quellpositionen gefüllt sind.

In Phase 1 simulieren wir das Erstellen des Baumes und der Bitvektoren in einer Schleife. Diese Schleife führen wir $\log_2 windowsize - \log_2 threshold$ mal aus. Zwei aufeinander folgende Chains im curr_Chains stellen einen Knoten auf der momentanen Ebene des Baumes dar. Wir füllen eine Hashmap *hmap* mit key-value-pairs aus den Fingerprints und Indizes der Chains Zeile 2. Danach suchen wir mit einem Rollinghash Objekt über den Eingabetext T und markieren gefundene Substrings 3 Zeile 3. Als letzten Schritt in der Schleife entfernen wir Chains die *cherrys* bilden und passen die noch zu bearbeitenden Chains an Seite 17. In der letzten Ausführung der Schleife überspringen wir diesen Schritt, da wir die Chains nicht für einen nächsten Durchlauf anpassen müssen Seite 17.

Algorithmus 1 : Phase 1

Data : vector<Chain> curr_Chains, vector<Chain>phase2_buffer, len_t size,
len_t threshold, unordered_map<uint64_t, len_t> hmap,
unordered_map<uint64_t, len_t> temp_hmap_store,
vector<unordered_map<uint64_t, len_t>> hmap_store

```
1 while size>=threshold do
2     make_hash_map(hmap, temp_hmap_storage, curr_Chains, size);
3     phase1_search(hmap, curr_Chains, size, temp_hmap_storage);
4     move temp_hmap_storage into hmap_storage;
5     clear temp_hmap_storage and hmap;
6     if size>threshold;
7      then
8         new_and_old_chains(curr_Chains, size, phase2_buffer);
9     size=size/2;
```

3.3.3 make_hash_map

Die Funktion make_hash_map erhält eine Länge size, einen Vector mit Chains, einen Buffer für die zweite Phase, eine Hashmap die später zum Suchen benötigt wird und eine Hashmap, die die Quellpositionen abspeichert.
Die Funktion füllt die Hashmap mit den Fingerprints aller noch zu untersuchenden Chains und speichert alle bereits gefundenen Quellpositionen in einer zweiten Hashmap ab. Des Weiteren werden alle bereits gefundenen Chains mit der size assoziiert.

Für jede Chain versuchen wir einen Eintrag in der Hashmap mit dem Fingerprint als Schlüssel und als Wert dessen Index im Vektor zu erstellen. Sollte aber schon ein Eintrag mit gleichem *key* vorhanden sein, vergleichen wir zunächst unsere aktuelle Chain_i mit der in der Hashmap vorhandenen Seite 18. Wir addieren *size* auf die Chain mit höherer

Position und ändern den Hashmapeintrag auf den Index der Chain mit niedriger Position (Algo.2 L:6,8,9). Dies geschieht, weil ein vorher liegender Substring eine Quellposition des späteren Faktors ist. Daher müssen wir nur noch nach dem vorherigen Substring suchen, da wir den anderen bereits gefunden haben. Die gefundene Quellposition tragen wir in temp_hmap_storage unter dem entsprechenden Fingerprint ein (Algo.2 L:6,10), damit später nach der Quellposition nicht noch einmal gesucht werden muss.

Algorithmus 2 : make_hash_map

Input : vector<Chain> chainVec, unordered_map<uint64_t, len_t> hashmap, unordered_map<uint64_t, len_t> temp_hmap_storage, len_t size

```
1  foreach Chain_i:chainVec do
2      if hashmap contains entry for fingerprint(Chain_i);
3       then
4          hmap_chain = chainVec[hashmap[fingerprint(Chain_i)]];
5          if hmap_chain.position < Chain_i.position then
6              Chain_i.add(size);
7              temp_hmap_storage[fingerprint(hmap_chain)]=hmap_chain.position
8          else
9              hmap_chain.add(size);
10             hashmap[fingerprint(Chain_i)]=i;
11             temp_hmap_storage[fingerprint(Chain_i)]=Chain_i.position;
12     else
13         add {fingerprint(Chain_i), i} to hashmap;
```

3.3.4 phase1_search

Die Funktion phase1_search erhält die Liste aller Chains und eine Hashmap gefüllt mit den Fingerprints und Indizes der zu überprüfenden Chains. Außerdem erhält die Funktion die Länge der Chains und eine leere Hashmap zum Speichern der Quellpositionen.
Am Ende der Funktion ist die zweite Hashmap mit allen gefundenen Quellpositionen gefüllt und alle Cherrys mit gefundener Quellposition wurden mit der size assoziiert.

In phase1_search suchen wir mit einem Rollinghash Objekt über den Text T. Sollte die Hashmap einen Eintrag enthalten der dem Hashwert des Rollinghashes entspricht (Algo.3 L:3), so überprüfen wir ob der Rollinghash noch vor dem Substring liegt (Algo.3 L:5). Wenn dem so ist können wir *size* auf die entsprechende Chain addieren.
Da wir als erstes das linkeste Vorkommen finden, liegt dies entweder vor dem Substring oder es ist der Substring selber. Daher müssen wir spätere Vorkommen nicht mehr betrachten

und können deshalb den Eintrag aus der Hashmap entfernen(Algo.3 L:8).
Aus dem selben Grund fügen wir die Quellposition nur zu temp_hmap_store hinzu, wenn wir vor dem Faktor liegen, da wir sonst keine Quellposition gefunden haben (Algo.3 L:7). Damit haben wir jede Chain deren aktueller Substring einen vorherigen Substring besitzt mit der aktuellen size markiert.

Algorithmus 3 : phase1_search

Input : vector<Chain> chainVec, unordered_map<uint64_t, len_t> hashmap, len_t size

```
1 create Rollinghash Object of length size over the text T;
2 while rollinghash does not reach beyond T do
3     if hashmap contains entry for rollinghash.hashvalue then
4         chain = hashmap[rollinghash.hashvalue];
5         if rollinghash.position<chain.position then
6             chain.add(size);
7             temp_hmap_store[rollinghash.hashvalue]=rollinghash.position;
8         hashmap.erase(rollinghash.hashvalue);
9     advance rollinghash;
```

3.3.5 new_chains

Die Funktion new_chains erhält die Liste aller Chains und die Länge der untersuchten Substrings.
Am Ende der Funktion sind alle Chains die eine cherry bilden aus der Liste entfernt, für nicht gefundene Chains sind neue Chains eingefügt und gefundene Chains wurden entfernt. Alle Chains repräsentieren nun Substrings die halb so lang sind wie zuvor Algorithmus 4.

Nachdem die Suche abgeschlossen ist müssen wir alle Chains untersuchen und feststellen, wie sie zu transformieren sind. Dies erfolgt immer in Zweierschritten, um die zusammengehörigen ab- und aufsteigenden Chains (dec, inc) gleichzeitig zu betrachten. Das heißt, wir betrachten immer eine Chain mit geradem Index und eine Chain mit ungeradem Index (0:1,2:3,4:5...).
Wir verunden bitweise die Bitvektoren von inc und dec mit der übergebenen Länge und betrachten die Ergebnisse als *Bool*werte.

Für die Werte dec_found und inc_found gibt es damit vier mögliche Kombinationen:

`dec_found` $\wedge$ `inc_found`:
Dieses Ergebnis bezeichnen wir als *cherry*.
Beide Textstellen besitzen vorherige Substrings und beide Chains müssen in Phase 1 nicht weiter betrachtet werden. Wir verschieben inc und dec an das Ende von phase2_buffer und ersetzen ihre Positionen durch die beiden letzten Chains im Vektor. Dies erspart uns, gegenüber dem Löschen, dass alle Chains im Vektor hinter inc und dec auch noch verschoben werden müssen. Stammen diese beiden Chains bereits aus der vorherigen Runde (oder der Initialisation) so betrachten wir diese beiden als nächstes.

`!dec_found` $\wedge$ `!inc_found`:
In diesem Fall wurde kein Textstück gefunden das heißt, dass wir vier Kinder Chains haben müssen. Da wir bereits zwei Chains zur Verfügung haben (inc, dec) passen wir diese einfach an und erstellen nur zwei passende neue Chains. Wir verschieben inc an das Ende des Vektors und fügen eine Kopie von inc am Ende des Vektors ein. Die Position dieser Kopie erhöhen wir um $size/2$. Die alte Position von inc füllen wir durch eine Kopie von dec. Die Position dieser Kopie erhöhen wir um $size/2$.
Damit stehen die beiden Kinder von inc am Ende des Vektors und die beiden Kinder von dec an den ursprünglichen Positionen von dec und inc.

`dec_found` $\wedge$ `!inc_found`:
Wurde nur das linke Textstück gefunden, so ändern wir lediglich die Positionen von inc und dec. Inc nimmt die Position von dec an und wir erhöhen die Position von dec um $size/2$.
Damit stehen die beiden Kinder von inc an den ursprünglichen Positionen von dec und inc.

`!dec_found` $\wedge$ `inc_found`:
Wurde nur das rechte Textstück gefunden, so ändern wir lediglich die Positionen von dec. Dec nimmt die Position von inc an und wir erhöhen die Position von dec um $size/2$.
Damit stehen die beiden Kinder von dec an den alten Positionen von dec und inc.

Algorithmus 4 : new_chains

Input : vector<Chain> ChainVector, len_t size

```
1  old_size = ChainVector.size();
2  for i=0;i<old_size;i=i+2 do
3      dec_found = ChainVector[i].getChain() & size;
4      inc_found = ChainVector[i+1].getChain() & size;
5      if dec_found ∧ inc_found then
6          remove ChainVector[i] and ChainVector[i+1] from ChainVector and add
            them to the Phase2Buffer;
7          replace ChainVector[i] and ChainVector[i+1] with the last two Chains in
            the Vector;
8          if ChainVector.size()<=old_size then
9              i=i-2
10         continue
11     if !dec_found ∧ !inc_found then
12         add a new Chain(ChainVector[i+1]) to ChainVector;
13         add a copy of ChainVector[i+1] to ChainVector and increase its position by
            size/2;
14         replace ChainVector[i+1] with a new Chain(ChainVector[i]) and increase its
            position by size/2;
15         continue
16     if dec_found then
17         change ChainVector[i] position to the position of ChainVector[i+1];
18         increase ChainVector[i+1] position by size/2;
19         continue
20     if inc_found then
21         change ChainVector[i+1] position to the position of ChainVector[i]+size/2;
22         continue
```

3.3.6 Transferphase

Der Übergang von Phase 1 zu Phase 2 oder die Transferphase erhält zwei Vektoren aus Chains.
Aus der Transferphase erhalten wir einen Vektor aus Groups und einen Vektor aus Faktoren.

Aus Phase 1 erhalten wir zwei Vektoren aus *Chains*, aber nicht alle sind für Phase 2 relevant. Chains, die kein positives Bit enthalten können wir einfach löschen, da sie keine Faktoren darstellen. Chains, die nur ein positives Bit enthalten können Phase 2 überspringen, da sie nur einen Faktor darstellen.
Zu jeder Chain die nicht Null ist und auch nicht nur einen Faktor darstellt, erstellen wir eine Group.
Während die Positionen der Chains den Anfang eines eventuell gefundenen Substrings der Länge θ angibt, ist die Position der cherry Chains immer der Anfang eines gefundenen Faktors mit beliebiger Länge. Daher müssen wir diese beiden Vektoren auch unterschiedlich bearbeiten. Um in der Group korrekte Positionen zu erstellen, übergeben wir im Konstruktor die Länge des zuletzt untersuchten Faktors. Für Chains entspricht diese Länge immer dem threshold, während sie den cherrys Chains dem *least significant bit* gleich ist.

Algorithmus 5 : void cherrys_to_groups

Input : vector<Group> groupVec, vector<Factor> factorVec, vector<Chain> phase2_buffer, len_t threshold

```
1  Chain c;
2  bool inc = true;
3  while !phase2_buffer.empty() do
4      c = phase2_buffer.back();
5      if c.get_chain() has only 1 bit set to 1 then
6          factorVec.push_back( Factor(c.get_position(), c.get_position(),
             c.get_chain(), false));
7          inc = !inc;
8          phase2_buffer.pop_back();
9          continue;
10     if c.get_chain() then
11         len_t lc = 1 « __builtin_ctz(c.get_chain());
12         groupVec.push_back(Group(c, inc, lc));
13         inc = !inc; phase2_buffer.pop_back();
14         continue;
15 phase2_buffer.clear();
```

Algorithmus 6 : void chains_to_groups

```
Input : vector<Group> groupVec, vector<Factor> factorVec, vector<Chain>
        phase2_buffer, len_t threshold
1  Chain c;
2  bool inc = true;
3  while !phase2_buffer.empty() do
4      c = phase2_buffer.back();
5      if c.get_chain() has only 1 bit set to 1 then
6          factorVec.push_back( Factor(c.get_position(), c.get_position(),
            c.get_chain(), false));
7          inc = !inc;
8          phase2_buffer.pop_back();
9          continue;
10     if c.get_chain() then
11         len_t lc = 1 « __builtin_ctz(c.get_chain());
12         groupVec.push_back(Group(c, inc, lc));
13         inc = !inc; phase2_buffer.pop_back();
14         continue;
15 phase2_buffer.clear();
```

3.3.7 Phase 2

Phase 2 erhält einen Vektor aus Groups.
Aus Phase 2 erhalten wir einen Vektor aus Faktoren.
In Phase 2 versuchen wir Faktoren zu verschmelzen. Dabei beschränken wir uns auf die im

Algorithmus 7 : Phase 2

```
1 while !groupVec.empty() do
2     find_next_group();
3     fill_hashmap();
4     phase2_search();
5     remove_empty_groups();
```

Bitvektor einer Group kodierten Faktoren, wir versuchen nicht Faktoren über eine Group hinaus zu verschmelzen.
Die Schleife beginnt damit alle Groups zu finden deren nächster Faktor den kleinsten Wert hat. Das heißt, wenn z.B. 32 der kleinste Wert aller nächsten Faktoren ist, dann erstellen wir uns eine Liste aus allen diesen entsprechenden Groups. Für alle diese Groups tragen

wir den entsprechenden Hashwert und Index in eine Hashmap ein. Dann suchen wir mit einem Rollinghash Object über die Eingabe T und verschmelzen den nächsten Faktor der Group mit dem aktiven Faktor. Oder fügen den aktiven Faktor in den Faktoren Vektor ein und deklarieren den nächsten Faktor als aktiv. Im letzten Schritt der Schleife entfernen wir alle Groups, die keinen nächsten Faktor mehr besitzen, aus dem groupVec und fügen ihre aktiven Faktoren dem Faktoren Vektor hinzufügen.

3.3.8 find_next_search_groups

Die Funktion find_next_search_groups erhält den Vektor aller Groups.
Als Ergebnis erhalten wir einen Vektor mit den Indizes aller Groups mit kleinstem nächsten Faktor.
Um alle Groups zu finden, die wir in dieser Runde der Phase 2 betrachten, iterieren wir über

Algorithmus 8 : find_next_search_groups

Input : std::vector <Group> groupVec, std::vector <len_compact_t> active_index

```
1 size = length of the next factor of the first group;
2 foreach Group_i : groupVec do
3     if length of the next factor of Group_i < size then
4         size = length of the next factor of Group_i;
5         clear active_index;
6         active_index.push_back(i);
7         continue;
8     if length of the next factor of Group_i == size then
9         active_index.push_back(i);
```

alle Groups. Dabei speichern wir den Index aller Groups dessen nächster Faktor kleiner gleich allen anderen nächsten Faktoren ist.

3.3.9 fill_hmap

Die Funktion fill_hmap erhält eine leere Hashmap, den Vektor aller Groups, einen Vektor über die Indizes aller Groups mit kleinstem nächsten Faktor und dessen Länge. Aus dieser Funktion erhalten wir die Hashmap, gefüllt mit den Fingerprints und Indizes, der noch zu untersuchenden Groups. Des Weiteren haben die Groups, dessen Suchstring bereits gefunden wurde, schon ihren nächsten Faktor absorbiert.
Für jede Group, die wir diese Runde betrachten, erstellen wir einen Eintrag in der Hashmap mit dem Fingerprint des Substrings beginnend bzw. endend an der Position der Group der doppelt so lang ist wie der nächste Faktor. Sollte bereits ein Fingerprint eines vorherigen

Algorithmus 9 : fill_hmap

Input : vector<Group> groupVec, unordered_map<uint64_t, len_t> hashmap, vector<len_t> active_groups, len_t size

```
1  foreach i : active_groups do
2      if hashmap contains entry for fingerprint(groupVec[i]) then
3          hmap_group = groupVec[hashmap[fingerprint(groupVec[i])]];
4          hmap_ss =hmap_group.get_start_of_search();
5          gv_ss =groupVec[i].get_start_of_search();
6          if hmap_ss < gv_ss then
7              groupVec[i].absorp_next(hmap_ss));
8          else
9              hmap_group.absorp_next(gv_ss));
10             hashmap[fingerprint(groupVec[i])]=i;
11     else
12         add {fingerprint(groupVec[i]), i} to hashmap;
```

Substrings in der Hashmap enthalten sein, so absorbiert die folgende Group ihren nächsten Faktor.

3.3.10 phase2_search

Die Funktion phase2_search erhält eine Hashmap gefüllt mit den Fingerprints und Indizes der noch zu untersuchenden Groups, einen Vektor aus Faktoren und den Vector aller Groups.
Als Ausgabe erhalten wir einen Vektor von Faktoren erweitert um alle alten aktiven Faktoren der Groups die wir nicht gefunden haben. Außerdem haben alle Groups dessen Searchstring wir gefunden haben ihren nächsten Faktor absorbiert.

In dieser Funktion suchen wir mit einem Rollinghash Objekt über die Eingabe T. Sollten wir einen vorherigen Substring finden, so absorbiert die Group ihren nächsten Faktor. Wenn der gefundene Substring nicht vor dem Suchstring liegt, so fügen wir den aktiven Faktor der Group in den Faktoren Vektor ein und setzen den nächsten Faktor als aktive Gruppe der Group. Danach entfernen wir den Eintrag aus der Hashmap, um ihn später nicht noch einmal betrachten zu müssen.

Algorithmus 10 : phase2_search

Input : len_t size, std::vector <Group> groupVec, std::unordered_map <uint64_t, len_compact_t> hmap, std::vector <lz77Aprox::Factor> factorVec, io::InputView input_view, hash_interface *hash_provider)

```
1  create rollinghash of length size over the text T;
2  while rollinghash does not reach beyond T do
3      if hashmap contains entry for rollinghash.hashvalue then
4          hmap_group = groupVec[hashmap[rollinghash.hashvalue]];
5          if rollinghash.position<chain.position then
6              hmap_group.absorp_next(rollinghash.position);
7          else
8              factorVec.push_back(hmap_group.advance_group());
9          hashmap.erase(rollinghash.hashvalue);
10     advance rollinghash;
```

3.3.11 check_groups

Die Funktion check_groups erhält einen Vektor von Indizes aller Groups die in diesem Durchlauf der Phase 2 getestet wurden, den Vektor aller Groups und einen Vektor von Faktoren.
Als Ergebnis erhalten wir den Vektor aller Groups die wir weiter betrachten müssen. Alle aktiven Faktoren der Groups die keinen nächsten Faktor hatten sind in den Vektor aus Faktoren verschoben worden.

Nach der Suche überprüfen wir ob die durchsuchten Groups weiter betrachtet werden müssen. Sollten sie keine weiteren nächsten Faktoren besitzen, so können wir keine weiteren Faktoren verschmelzen. Deshalb fügen wir die aktiven Faktoren der Groups dem Faktor Vektor hinzu und löschen die entsprechenden Groups. Um zu vermeiden, dass alle Groups hinter der gelöschten verschoben werden müssen, ersetzen wir die zu löschenden Groups einfach durch die letzte Group im Vektor. Wenn die zu löschende Group an letzter Position im Vektor steht so löschen wir diese einfach.

Algorithmus 11 : check_groups

Input : (len_t size, std::vector <Group> groupVec, std::vector <len_compact_t> active_index, std::vector <lz77Aprox::Factor> factorVec

```
1  while !active_index.empty() do
2      int index = active_index.back();
3      Group g = groupVec[index];
4      active_index.pop_back();
5      if !g.has_next() then
6          factorVec.push_back(g.advance_Group());
7          if groupVec.size() - 1 == index then
8              groupVec.pop_back();
9          else
10             groupVec[index] = groupVec.back();
11             groupVec.pop_back();
```

3.4 Kollisionsresolution

Das Programm benutzt lineares Sondieren, um für kollidierende Hashwerte freie Buckets zu finden. Dies ist eine sehr robuste Methode die nur dann versagt, wenn wirklich alle Buckets gefüllt sind.
Sollten zwei Hashcodes gleich sein überprüfen wir zunächst ob die beiden entsprechenden Strings gleich sind. Wenn sie übereinstimmen haben wir keine Kollision. Sollten beide Strings unterschiedlich sein so versuchen wir den String an der Stelle Schlüssel + 1 einzufügen. Ist dieser Bucket auch gefüllt überprüfen wir, ob es sich hier ebenfalls um eine Kollision handelt. Ist dem so, erhöhen wir den Wert immer weiter bis wir einen leeren Bucket finden. In diesem speichern wir der Wert dann ab.
Sollte es sich bei einem Lookup herausstellen, dass eine Kollisionen vorhanden ist müssen wir auch alle folgenden Hashwerte überprüfen bis wir keine Kollision mehr haben oder einen leeren Bucket finden.
Beim Löschen eines Eintrages müssen wir, nachdem wir den Eintrag entfernt haben, alle folgenden Hashwerte untersuchen bis wir einen leeren Bucket finden. Wollen wir also den Eintrag e_0 mit Hashwert h löschen so entfernen wir den Eintrag zu h und untersuchen als nächstes $h+1$. Für den Eintrag e_1 in $h+1$ bilden wir den originalen Hashwert. Wenn dieser mit h übereinstimmt tragen wir e_1 unter h ein und entfernen den Eintrag für $h+1$. Danach untersuchen wir alle folgenden Hashwerte $h+2, h+3, ...$ bis wir einen leeren Bucket finden.

4. Evaluation

Die experimentell erhobenen Daten werden auf zwei Arten repräsentiert, als Liste für kleine Datenmengen von *gzip* und *lzss_lcp* und als Tabellen mit einer Heatmap für über die Implementation erhobene Daten.
Für die Färbung der Heatmaps werden zwei Muster verwendet:

- In Tabellen die den Speicherbedarf darstellen richten sich die Farben danach welche Multiplikation der Eingabegröße n der Wert in der Tabelle x überschritten hat.
  $x < 2*n$ →Grün, $2*n \leq x < 3*n$ →Orange, $3*n \leq x < 4*n$ →Rot, $4*2 \leq x < 5*n$ →Violett, $5 * m \leq x$ →Blau
- Für alle anderen Tabellen teilen wir den präsentierten Wertebereich in fünf Intervalle auf und weisen ihnen Farben in aufsteigender Reihenfolge zu: Grün, Orange, Rot, Violett, Blau

In den Tabellen steht x-Achse, beschriftet mit *window*, für die maximale Faktorlänge und die y-Achse, beschriftet mit *threshold*, für die minimale Faktorlänge.

4.1 Auswahl der Hashfunktionen

Das Programm wurde mit vier verschiedenen Hashfunktionen getestet.

- Rabin-Karp-Hash: Fingerprint mit maximalen Primzahlen für 64Bit Hashwerte
- djb2: ein optimierter Polynomen Hash mit Mersenne und Fermat Primzahlen der auf 32Bit Hashwerten arbeitet
- Buzhashing: Tabellenhash mit fester Substitutionstabelle für 64Bit Hashwerte
- ntHash: ein für das DNA-Alphabet optimierter Tabellenhash für 64Bit Hashwerte

4.2 Versionen

Zur Auswertung wurden vier Versionen des Programmes erstellt, eine mit jeder Hashfunktion.

- `tdc-djb2`
- `tdc-buz`
- `tdc-rk`
- `tdc-nthash`

Als Vergleichswerte wurden Werte gzip und lzss_lcp erzeugt. Zum Vergleich der Kompressionsrate wurden ebenfalls Werte für den verwendeten Coder ohne vorherige LZ-Approximation erhoben.

4.3 Auswahl der Testdaten

Die verwendeten Testdaten stammen mit einer Ausnahme alle aus dem Pizza & Chilli Corpus [7]. Der randomisierte Datensatz wurde mit tudocomp generiert mit $seed = 16112020$ für 95.4MB. Die Datei sdna.200MB entspricht der Datei dna.200MB wenn alle Bytes die nicht A,C,G,T entsprechen gelöscht wurden.

- english.200MB
- dblp.xml.200MB
- proteins.200MB
- sdna.200MB
- random.95MB
- einstein.de.txt

4.4 Testumgebung

Alle Tests und Versionen wurden auf einem System mit folgender Hard- und Software generiert und ausgeführt:

Software:

`Kernel`: 5.9.1-arch1-1

`OS`: Arch Linux x86_64

`Compiler`: gcc 10.2.0

`SDSL`: SDLSL 2.1.1

`glog`: google-glog 0.4.0

Hardware:

`CPU`: Intel Xeon E3-1231v3

`GPU`: AMD R9 270

`Mainboard`: ASRock H97M Pro4

`Memory`: 2x 8GB DDR3-1600 Crucial Ballistix Sport

`Storage`: Western Digital 1TB Blue 7200RPM

4.5 Codierung

Als Coder wurden der *binary*-Coder für Referenzen *ref* und Längen *len* gewählt, während die Literale *lit* mit *huff*-Coder codiert wurden.

4.6 Tabellenwerte

Die Werte in den Tabellen bezüglich des Speichers sind das Maximum aus den Speicherwerten von Phase 1 und Phase 2. Die Transferphase beachte ich nicht, da wir hier eigentlich nur aus Chains Groups erzeugen und die verbrauchten Chains dann löschen. In dieser Phase treten manchmal Speicherspitzen auf die die Ergebnisse verzerren und für die keine Erklärung vorliegt. Daher ist die Transferphase nicht in den Tabellen enthalten.

4.7 Testparameter

Wir schließen aus, dass die minimale und maximale Faktorlänge gleich sind. Wir testen diese Paare nicht, da wir mit ihnen das Ausführen der zweiten Phase immer komplett ausschließen würden und wir hier versuchen das gesamte Programm zu bewerten.
Die Version des ntHash wurde nur für die Eingabe sdna.200MB getestet, da die Hashfunktion nur für das DNA-Alphabet konzipiert wurde.
Djb2 wurde nicht für die Eingabe random.100MB getestet da hier so viele Kollisionen auftreten, dass ein einzelner Test über eine Stunde dauert. Ich sah es daher als nicht sinnvoll an einen vollen Testdurchlauf auszuführen.

4.8 exemplarischer Programmdurchlauf

Hier zu sehen sind die beiden Phasen des Programms und ihr Speicherbedarf. Die dunkle Färbung eines Balken zeigt den Speicherbedarf bei Beginn der Funktion, während der helle Teil des Balken den maximalen Speicherbedarf zur Zeit der Funktion angibt.
Deutlich zu erkennen sind auch die Subphasen, die für Phase 1 aus make_hash_map, phase1_search, und new_chains besteht. Wir sehen den Speicheranstieg in jeder Runde der ersten Phase der zeigt, dass wir neue Chain Objekte erzeugen. Der Anstieg erinnert an eine gestauchte Parabel. Der maximale Speicher ist durch die Größe der Hashmap bedingt und sinkt sogar nach dem die Suche abgeschlossen ist. Die Laufzeit ist fast nur durch die Laufzeit der Suche bedingt, des Weiteren ist auch zu sehen, dass das Erstellen der Hashmap von der Anzahl der Chain Objekte abhängt.
Die Subphasen der Phase 2 sind weniger gut zu erkennen, diese sind
find_next_search_groups, fill_hmap, phase2_search und check_groups. Die zweite Phase beginnt mit weniger Speicherbedarf als die erste endet, da wir alle Chain Objekte, die

keinen Faktor darstellen einfach löschen können. Den Speicheranstieg in Runde 0 der zweiten Phase resultiert daraus, dass wir in der Suche bereits Factor Objekte erzeugen, die Group Objekte aber nicht direkt löschen können.

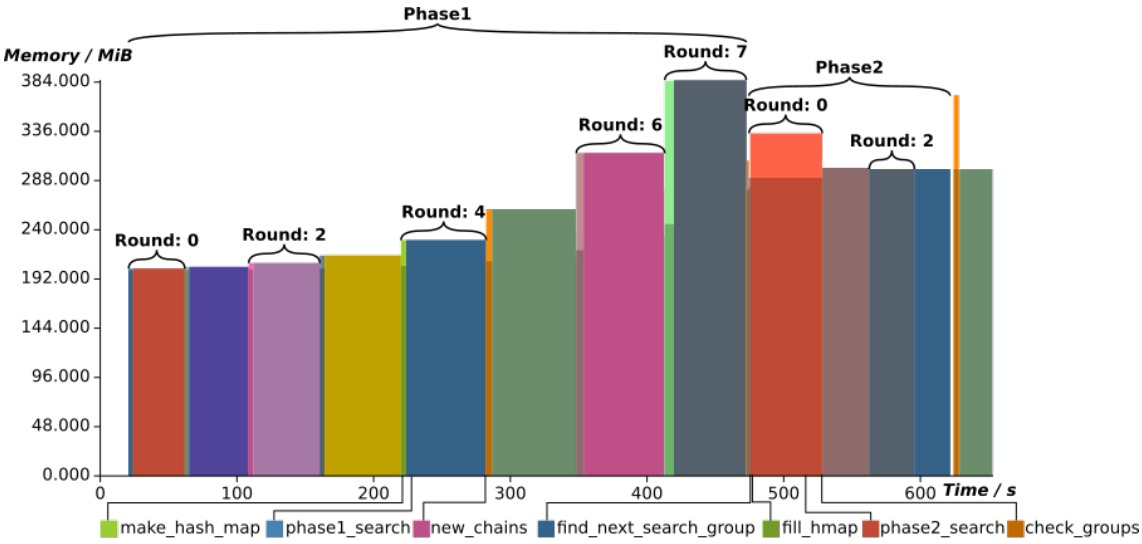

Abbildung 4.1: Speicherbedarf während des Programmdurchlaufs für dblp.xml.200MB von 4096 bis 32 *farbig*

4.9 Speicherbedarf

Speichertechnisch zeigt das Programm gute Ergebnisse, nur selten überschreiten wir die vierfache Eingabegröße. Im allgemeinen folgt der Speicherbedarf dem Muster je kleiner die minimaler Faktorlänge desto mehr Speicher benötigen wir.
Besonders gute Ergebnisse erzielen wir für den sehr komprimierbaren einstein.de.txt.
Djb2 benötigt, bedingt durch die kleineren Hashwerte, manchmal weniger Speicher. Die Ersparnis zeigt sich aber erst deutlicher je mehr die minimale Faktorlänge sinkt und wir damit mehr Faktoren haben. Dadurch steigt natürlich auch die Anzahl der Einträge in der Hashmap 4.14.54.34.2.
Besonders auffällig ist jedoch der Speicherbedarf für proteins.200MB bei Faktoren kleiner 32 für 64Bit Hashwerte 4.4 und wenn die minimale Faktorlänge unter 8 sinkt bei 32-Bit Hashwerte 4.3. Ich vermute, dass der höhere Speicherbedarf für 32Bit Hashwerten gegenüber 64Bit Werten aus der Speicherallokation der Hashmap resultieren. Es ist als ein entscheidendes Ergebnis zu bewerten da es zeigt, dass kleinere Hashwerte nicht unbedingt in kleinerem Speicherbedarf resultiert.
Des Weiteren überschreiten wir das vierfache der Eingabegröße nur wenn die minimale Faktorlänge unter 16 Byte sinkt. Dies entspricht auch meinen theoretischen Abschätzungen, für den worstcase:

$$n \leq (n/threshold) * 6Byte * 2 \rightarrow threshold \leq 12$$

Wobei „2“ der Overhead einer Hashmap ist [10].
Groups müssen wir hier nicht betrachten, da eine Group immer mindestens zwei Faktoren darstellt oder wenn sie nur einen darstellt im nächsten Schritt gelöscht wird. Daher ist der Speicherbedarf für Groups immer kleiner gleich dem Speicherbedarf der Chains. Es ist zu beachten, dass dies nur im worstcase gilt wenn jede Chain nur einen Faktor darstellt, sollten Chains mehrere Faktoren enthalten kann der Speicherbedarf der Groups größer sein. Wenn wir die Factors mit in Betracht ziehen erhalten wir das gleiche Ergebnis. Ein Factor benötigt 12Byte, da wir sie aber nie in etwas anderem als einen Vektor halten und damit konstanten Overhead haben, gilt hier auch:

$$n \leq (n/threshold) * 12Byte \rightarrow threshold \leq 12$$

Daraus folgt, dass wir für minimale Faktorlängen kleiner 16 den gewünschten Speicherbedarf nicht mehr garantieren können.
Gzip's benötigter Speicher liegt deutlich unter allen anderen Werten mit weniger als 1MB für alle Eingaben. Für gzip scheinen alle getesteten Eingaben gleich viel Speicher zu benötigen. Außerdem scheint die Eingabegröße nicht mit einzugehen.
Lzss_lcp benötigt minimal soviel Speicher wie die Implementation maximal. Damit liegen wir immer deutlich unter lzss_lcp.

Eine interessante Beobachtung ist es wie viele Faktoren bereits vor der Suche gefunden werden. Betrachten wir die Eingabe proteins.200MB mit maximaler Faktorlänge von 8 und minimaler Länge von 4, so sehen wir, dass wir in der Initialisierung 26214398 Chains generieren. Die Hashmap mit der wir suchen enthält aber nur 20010332 Einträge. Daraus können wir schließen das 6204066 Faktoren als Quellposition die Position vorheriger Chains haben.

`gzip:`

sdna.200MB: 7,432MB	dblp.xml.200MB: 7,432MB
proteins.200MB: 7,432MB	einstein.de.txt: 7,432MB
english.200MB: 7,432MB	random.100MB: 7,432MB

`lzss_lcp:`

sdna.200MB: 3067MB	dblp.xml.200MB: 2909MB
proteins.200MB: 3040MB	einstein.de.txt: 1356MB
english.200MB: 3145MB	random.100MB: 1310MB

Tabelle 4.1: 32Bit-Hash Speicherbedarf sdna.200MB in MB *farbig*

threshold													
8192													210
4096												211	211
2048											212	212	212
1024										215	215	215	215
512									222	222	222	222	222
256								234	234	234	234	234	234
128							259	259	259	259	259	259	259
64						309	309	309	309	309	309	309	309
32					408	408	408	408	408	408	408	408	408
16				591	591	591	591	591	591	591	591	591	591
8			659	661	661	661	661	661	661	661	661	661	661
4		714	804	806	806	806	806	806	806	806	806	806	806
2	1216	714	804	806	806	806	806	806	806	806	806	806	806
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.2: 64Bit Hash Speicherbedarf sdna.200MB in MB *farbig*

threshold													
8192													210
4096												211	211
2048											213	213	213
1024										217	217	217	217
512									225	225	225	225	225
256								240	240	240	240	240	240
128							272	272	272	272	272	272	272
64						335	335	334	334	334	334	334	334
32					455	454	454	454	454	454	454	454	454
16				687	687	686	686	685	685	685	685	685	685
8			790	795	796	795	795	795	795	795	795	795	795
4		715	820	825	825	825	824	824	824	824	824	824	824
2	1216	715	820	825	825	825	824	824	824	824	824	824	824
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.3: 32Bit-Hash maximaler Speicherbedarf proteins.200MB *farbig*

threshold													
8192													210
4096												211	211
2048											212	212	212
1024										215	215	215	215
512									222	222	222	222	222
256								234	234	234	234	234	234
128							259	259	259	259	259	259	259
64						308	308	308	308	308	308	308	308
32					402	402	402	402	402	402	402	402	402
16				579	580	578	578	578	578	578	578	578	578
8			928	925	924	924	925	924	924	925	925	925	925
4		1123	1144	1148	1149	1149	1149	1149	1149	1149	1149	1149	1149
2	1222	1355	1363	1367	1368	1368	1368	1368	1368	1368	1368	1368	1368
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.4: 64Bit Hash Speicherbedarf proteins.200MB in MB *farbig*

threshold													
8192													210
4096												211	211
2048											213	213	213
1024										217	217	217	217
512									225	224	224	224	224
256								239	239	239	239	239	239
128							268	266	266	265	265	265	265
64						320	317	315	315	315	315	315	315
32					425	409	405	403	403	403	403	403	403
16				617	589	573	570	568	567	567	567	567	567
8			1004	922	893	878	874	873	873	873	873	873	873
4		1386	1659	922	893	878	874	873	873	873	873	873	873
2	1224	1418	1657	1003	981	969	967	965	965	965	965	965	965
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.5: 32Bit-Hash maximaler Speicherbedarf english.200MB in MB *farbig*

threshold													
8192													210
4096												211	211
2048											212	212	212
1024										215	215	215	215
512									222	222	222	222	222
256								234	234	234	234	234	234
128							259	259	259	259	259	259	259
64						309	309	309	309	309	309	309	309
32					409	409	409	409	409	409	409	409	409
16				602	602	602	602	602	602	602	602	602	602
8			730	729	729	729	729	729	729	729	729	729	729
4		869	937	937	937	937	937	937	937	937	937	937	937
2	1223	868	889	889	889	889	889	889	889	889	889	889	889
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.6: 64Bit Hash Speicherbedarf english.200MB in MB *farbig*

threshold													
8192													210
4096												211	211
2048											213	213	213
1024										217	216	216	216
512									224	223	223	223	223
256								238	237	236	236	236	236
128							266	263	262	261	261	261	261
64						323	317	314	312	311	311	311	311
32					437	424	417	414	412	412	411	411	411
16				657	629	616	609	606	604	604	603	603	603
8			1162	1084	745	728	723	718	717	716	716	716	716
4		878	1024	1154	1096	1087	1084	1081	1080	1080	1080	1080	1080
2	1226	877	1013	1148	1109	1076	1071	1070	1069	1069	1069	1069	1069
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.7: 32Bit-Hash Speicherbedarf dblp.xml.200MB in MB *farbig*

threshold													
8192													210
4096												211	211
2048											212	212	212
1024										215	215	215	215
512									222	222	222	222	222
256								234	234	234	234	234	234
128							259	259	259	259	259	259	259
64						308	308	308	308	308	308	308	308
32					378	379	379	379	379	379	379	379	378
16				452	449	449	449	449	449	449	449	449	449
8			528	636	640	640	640	640	640	640	640	640	640
4		783	978	759	759	759	759	759	759	759	759	759	759
2	1229	760	979	924	909	909	909	909	909	909	909	909	909
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.8: 64Bit Hash Speicherbedarf dblp.xml.200MB in MB *farbig*

threshold													
8192													210
4096												211	211
2048											213	213	213
1024										217	217	217	217
512									225	225	225	225	225
256								240	240	240	240	240	240
128							272	272	272	272	272	272	272
64						329	329	329	329	329	329	329	329
32					405	404	404	404	404	404	404	404	404
16				466	500	488	488	488	488	488	488	488	488
8			596	675	663	665	663	665	665	665	665	665	665
4		784	638	723	732	719	721	710	710	710	733	719	710
2	1233	784	639	724	718	721	715	721	721	721	715	721	733
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.9: 32Bit Hash Speicherbedarf für einstein.de.txt in MB *farbig*

threshold													
16384													
8192													93
4096												93	93
2048											94	94	94
1024										95	95	95	95
512									98	98	98	98	98
256								103	103	103	103	103	103
128							109	109	109	109	109	109	109
64						115	123	122	122	122	122	122	122
32					130	121	122	124	124	124	124	124	124
16				161	135	150	149	149	148	148	148	148	148
8			222	161	139	153	154	153	153	153	153	153	153
4		347	222	161	187	143	155	155	156	156	156	156	156
2	596	347	222	161	187	144	155	155	155	155	155	155	155
window	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384

Tabelle 4.10: 64Bit Hash Speicherbedarf einstein.de.txt in MB *farbig*

threshold													
8192													93
4096												93	93
2048											93	93	93
1024										95	94	93	93
512									97	95	94	93	93
256								101	97	95	94	93	93
128							108	101	97	95	94	93	93
64						118	108	101	97	95	94	94	93
32					132	118	108	101	97	95	94	94	94
16				163	132	118	108	101	97	95	95	94	94
8			223	163	132	118	108	102	98	96	95	95	94
4		348	223	163	132	118	108	102	98	98	96	95	95
2	597	348	223	163	132	118	108	102	101	98	97	96	95
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.11: 64Bit Hash Speicherbedarf für random.100MB in MB *farbig*

threshold													
16384													
8192													101
4096												101	101
2048											102	102	102
1024										104	104	104	104
512									108	108	108	108	108
256								115	115	115	115	115	115
128							131	131	131	131	131	131	131
64						161	161	161	161	161	161	161	161
32					220	220	220	220	220	220	220	220	220
16				341	341	341	341	341	341	341	341	341	341
8			583	583	583	583	583	583	583	583	583	583	583
4		1069	1069	1069	1069	1069	1069	1069	1069	1069	1069	1069	1069
2	1119	1069	1069	1069	1069	1069	1069	1069	1069	1069	1069	1069	1069
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

4.10 Kompressionsfaktor

Die erzeugten Kompressionsraten sind mit denen von lzss_lcp und gzip vergleichbar, für einstein.de.txt sind sie sogar deutlich besser.
Dies ist sehr erstaunlich für eine Approximation. Ich vermute, dass dies daher resultiert, dass gzip auf DEFLATE aufbaut, welches die Länge eines Faktors in acht Bit kodiert und damit eine maximale Faktorlänge von 258 Byte begrenzt [5]. Und natürlich auch daher, dass einstein.de.txt eine extrem komprimierbare Eingabe ist, die lange Faktoren zulässt.
Generell erkennen wir einen Trend dahin, dass wenn die Faktoren kleiner gleich 8 sein können sie die Kompression verschlechtern während alle größeren Faktoren sie verbessern. Brechen wir die Suche aber zu früh ab, also mit minimalen Längen größer als 16, erhalten wir ebenfalls eine schlechtere Kompression.
Generell betrachtet ist die Kompressionsrate für eine 5-optimale Approximation erstaunlich gut.

`gzip:`

sdna.200MB: 28.2%	dblp.xml.200MB: 17.5%
protein.200MB: 46.5%	einstein.de.txt: 31.2%
english.200MB: 37.8%	random.100MB: 0%

`lzss_lcp:`

sdna.200MB: 27.46%	dblp.xml.200MB: 16.16%
proteins.200MB: 54.17%	einstein.de.txt: 0.34%
english.200MB: 39.64%	random.100MB: 100.68%

`Huffman coder:`

sdna.200MB: 27.53%	dblp.xml.200MB: 16.16%
proteins.200MB: 52.94%	einstein.de.txt: 63.23%
english.200MB: 57.00%	random.100MB: 100.68%

Tabelle 4.12: Kompressionsfaktor für sdna.200MB *farbig*

threshold	window 4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384
8192													0.2753
4096												0.2753	0.2753
2048											0.2753	0.2753	0.2753
1024										0.2753	0.2753	0.2753	0.2753
512									0.2753	0.2753	0.2753	0.2753	0.2753
256								0.2752	0.2752	0.2752	0.2752	0.2752	0.2752
128							0.2752	0.2752	0.2752	0.2752	0.2752	0.2752	0.2752
64						0.2747	0.2748	0.2748	0.2748	0.2748	0.2748	0.2748	0.2748
32					0.2727	0.2728	0.2728	0.2729	0.2730	0.2730	0.2730	0.2730	0.2730
16				0.2770	0.2776	0.2783	0.2783	0.2791	0.2798	0.2798	0.2798	0.2798	0.2798
8			0.4817	0.4929	0.5073	0.5218	0.5218	0.5364	0.5510	0.5510	0.5510	0.5510	0.5510
4		0.4966	0.4827	0.4939	0.5084	0.5230	0.5230	0.5377	0.5524	0.5524	0.5524	0.5524	0.5523
2	0.9600	0.4966	0.4827	0.4940	0.5084	0.5231	0.5231	0.5377	0.5524	0.5524	0.5524	0.5524	0.5523

Tabelle 4.13: Kompressionsfaktor für proteins.200MB *farbig*

threshold	window 4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384
8192													0.5294
4096												0.5294	0.5294
2048											0.5294	0.5294	0.5294
1024										0.5289	0.5289	0.5289	0.5289
512									0.5269	0.5269	0.5269	0.5269	0.5269
256								0.5235	0.5235	0.5235	0.5235	0.5235	0.5235
128							0.5172	0.5171	0.5172	0.5172	0.5172	0.5172	0.5172
64						0.5063	0.5061	0.5061	0.5061	0.5061	0.5061	0.5061	0.5061
32					0.4912	0.4906	0.4906	0.4908	0.4911	0.4911	0.4911	0.4911	0.4911
16				0.4806	0.4780	0.4778	0.4783	0.4790	0.4796	0.4796	0.4796	0.4796	0.4796
8			0.4962	0.4862	0.4848	0.4860	0.4878	0.4898	0.4918	0.4918	0.4918	0.4918	0.4918
4		0.8723	0.8621	0.8740	0.8964	0.9213	0.9469	0.9726	0.9984	0.9984	0.9984	0.9985	0.9985
2	0.9610	0.8734	0.8614	0.8752	0.8977	0.9226	0.9483	0.9741	0.9999	0.9999	0.9999	0.9999	0.9998

Tabelle 4.14: Kompressionsfaktor für english.200MB *farbig*

threshold	window 4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384
8192													0.5700
4096												0.5700	0.5700
2048											0.6322	0.5700	0.5700
1024										0.5700	0.5700	0.5700	0.5700
512									0.4104	0.5699	0.5699	0.5699	0.5699
256								0.5698	0.5698	0.5698	0.5698	0.5698	0.5698
128							0.5694	0.5694	0.5694	0.5694	0.5694	0.5694	0.5694
64						0.5685	0.5685	0.5685	0.5685	0.5685	0.5685	0.5685	0.5685
32					0.5657	0.5656	0.5656	0.5657	0.5657	0.5657	0.5657	0.5657	0.5657
16				1.1143	0.5541	0.5546	0.5546	0.5551	0.5557	0.5557	0.5557	0.5557	0.5557
8			0.5505	0.5605	0.5715	0.5827	0.5827	0.5939	0.6051	0.6051	0.6051	0.6051	0.6051
4		0.6119	1.2267	1.2618	0.6476	0.6662	0.6662	0.6848	0.7034	0.7034	0.7034	0.7034	1.4110
2	0.9630	1.1654	0.6149	0.6326	0.6512	0.6700	0.6700	1.0992	0.7076	0.7076	0.7076	0.7076	0.7076

Tabelle 4.15: Kompressionsfaktor für dblp.xml.200MB *farbig*

threshold													
8192													0.6608
4096												0.6608	0.6608
2048											0.6608	0.6608	0.6608
1024										0.6608	0.6608	0.6608	0.6608
512									0.6608	0.6608	0.6608	0.6608	0.6608
256								0.6608	0.6608	0.6608	0.6608	0.6608	0.6608
128							0.6583	0.6583	0.6583	0.6583	0.6583	0.6583	0.6583
64						0.6124	0.6124	0.6124	0.6124	0.6124	0.6124	0.6124	0.6124
32					0.4912	0.4922	0.4922	0.4922	0.4922	0.4922	0.4922	0.4929	0.4937
16				0.3784	0.3756	0.3790	0.3790	0.3790	0.3804	0.3819	0.3834	0.3849	0.3863
8			0.3753	0.3321	0.3324	0.3389	0.3403	0.3418	0.3432	0.3447	0.3462	0.3477	0.3491
4		0.5420	0.3810	0.3402	0.3436	0.3534	0.3535	0.3537	0.3539	0.3541	0.3543	0.3544	0.3546
2	0.9643	0.5456	0.3849	0.3444	0.3481	0.3580	0.3580	0.3580	0.3580	0.3580	0.3580	0.3581	0.3581
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.16: Kompressionsfaktor für einstein.de.txt *farbig*

threshold													
8192													0.1129
4096												0.0632	0.0630
2048											0.0363	0.0358	0.0357
1024										0.0225	0.0214	0.0209	0.0208
512									0.0165	0.0143	0.0132	0.0127	0.0125
256								0.0161	0.0118	0.0096	0.0085	0.0080	0.0079
128							0.0218	0.0135	0.0092	0.0070	0.0059	0.0054	0.0052
64						0.0366	0.0204	0.0120	0.0077	0.0055	0.0044	0.0039	0.0038
32					0.0674	0.0358	0.0195	0.0111	0.0068	0.0046	0.0035	0.0030	0.0029
16				0.1284	0.0670	0.0353	0.0190	0.0107	0.0064	0.0042	0.0031	0.0026	0.0025
8			0.2473	0.1283	0.0667	0.0351	0.0188	0.0104	0.0061	0.0040	0.0029	0.0024	0.0023
4		0.4779	0.2474	0.1284	0.0669	0.0353	0.0190	0.0107	0.0065	0.0043	0.0032	0.0028	0.0027
2	0.9238	0.4781	0.2477	0.1286	0.0672	0.0356	0.0195	0.0112	0.0068	0.0047	0.0036	0.0032	0.0031
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.17: Kompressionsfaktor für random *farbig*

threshold													
8192													1.0068
4096												1.0068	1.0068
2048											1.0068	1.0068	1.0068
1024										1.0068	1.0068	1.0068	1.0068
512									1.0068	1.0068	1.0068	1.0068	1.0068
256								1.0068	1.0068	1.0068	1.0068	1.0068	1.0068
128							1.0068	1.0068	1.0068	1.0068	1.0068	1.0068	1.0068
64						1.0068	1.0068	1.0068	1.0068	1.0068	1.0068	1.0068	1.0068
32					1.0068	1.0068	1.0068	1.0068	1.0068	1.0068	1.0068	1.0068	1.0068
16				1.0068	1.0068	1.0068	1.0068	1.0068	1.0068	1.0068	1.0068	1.0068	1.0068
8			1.0068	1.0068	1.0068	1.0068	1.0068	1.0068	1.0068	1.0068	1.0068	1.0068	1.0068
4		1.0189	1.0189	1.0189	1.0189	1.0189	1.0189	1.0189	1.0189	1.0189	1.0189	1.0189	1.0189
2	1.7796	1.7796	1.7796	1.7796	1.7796	1.7796	1.7796	1.7796	1.7797	1.7797	1.7797	1.7797	1.7796
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

4.11 Laufzeit

Deutlich zu sehen ist der Zeitunterschied zwischen den verschiedenen Hashfunktionen. Wobei im Durchschnitt das Buzhashing am schnellsten war, gefolgt vom Rabin-Karp-Hash und djb2 als langsamster.
Alle drei sind deutlich langsamer als gzip, mit lediglich 34 Sekunden, jedoch deutlich schneller als lzss_lcp.
Die Version mit dem Buzhash ist die schnellste, da sie gegenüber dem Rabin-Karp-Hash kein Modulo oder Subtraktion durchführen muss. Kollisionen treten für beide Versionen selten auf, sind jedoch für das Buzhashing häufiger. Der ntHash weist eine minimale Beschleunigung auf. Die längere Laufzeit von djb2 lässt sich durch die häufigeren Kollisionen erklären. Da djb2 nur Hashwerte in 32Bit Größe erzeugt, müssen wir in fast jedem Suchdurchlauf Kollisionen beachten. Dieses Problem spitzt sich zu je mehr Chains bzw. Groups existieren.
Im Allgemeinen folgt die Laufzeit dem zu erwartenden Muster, je weiter der Abstand zwischen maximaler und minimaler Faktorgröße ist desto länger ist die Laufzeit.

gzip:

- sdna.200MB 34.3s
- proteins.200MB 7.6s
- english.200MB 12.8s
- einstein.de.txt 4.2s
- random.100MB 2.7s
- dblp.xml.200MB: 4.1s

lzss_lcp:

- sdna.200MB: 5279s
- proteins.200MB: 2725s
- english.200MB: 2833s
- dblp.xml.200MB: 2519s
- einstein.de.txt: 1531s
- random.100MB: 1548s

Tabelle 4.18: djb2 Laufzeit für sdna.200MB in s *farbig*

threshold													
8192													276
4096												338	451
2048											458	580	699
1024										557	756	900	1012
512									698	1000	1068	1203	1348
256								797	1103	1361	1536	1687	1843
128							828	1249	1554	1825	1994	2070	2283
64						951	1327	1689	2027	2271	2381	2587	2642
32					1062	1550	1886	2285	2576	2841	3056	3104	3173
16				1649	2336	2672	2916	3196	3575	3787	3742	4172	4266
8			2776	3372	3949	4294	4614	4945	5303	5445	5519	5626	5987
4		565	1496	2506	3083	3357	3740	4104	4365	4429	4815	4920	5120
2	325	664	1526	2451	3028	3371	3766	3890	4271	4573	5060	4672	5231
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.19: Buzhashing Laufzeit für sdna.200MB in s *farbig*

threshold													
8192													179
4096												174	259
2048											191	258	349
1024										196	271	340	444
512									209	289	360	456	551
256								223	307	390	464	585	636
128							232	324	410	492	558	683	731
64						237	334	427	514	591	677	799	874
32					245	344	442	541	619	709	783	927	987
16				272	387	485	582	676	767	833	911	1096	1126
8			302	419	533	636	739	831	914	989	1072	1240	1361
4		227	357	480	581	694	780	878	969	1061	1122	1210	1396
2	359	291	411	534	660	748	836	939	1016	1117	1183	1360	1393
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.20: Rabin-Karp-Hash Laufzeit für sdna.200MB in s *farbig*

threshold													
8192													201
4096												259	293
2048											223	316	401
1024										234	324	421	597
512									291	409	518	621	722
256								309	360	554	562	660	861
128							319	459	577	593	804	906	860
64						269	387	503	701	719	811	907	987
32					279	481	519	639	736	851	1115	1039	1328
16				364	439	667	677	924	909	1188	1310	1478	1312
8			339	488	612	740	849	970	1081	1394	1500	1412	1809
4		348	502	555	830	806	1109	1253	1160	1255	1354	1731	1538
2	517	403	627	629	879	890	987	1200	1429	1360	1525	1730	1516
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.21: ntHash Laufzeit für sdna.200MB in s *farbig*

threshold													
8192													175
4096												180	251
2048											185	261	333
1024										208	270	340	412
512									200	315	353	429	502
256								252	296	430	450	526	599
128							252	355	409	516	555	632	704
64						239	361	488	529	609	665	742	815
32					251	353	464	594	664	721	782	858	931
16				273	381	485	600	700	811	836	904	983	1055
8			299	416	527	703	788	870	1002	1003	1049	1125	1197
4		214	359	477	582	770	856	935	1037	1059	1107	1184	1258
2	295	270	417	534	680	822	927	973	1099	1113	1164	1241	1315
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.22: djb2 Laufzeit für proteins.200MB in s *farbig*

threshold													
8192													280
4096												347	452
2048											469	587	708
1024										610	722	894	1017
512									717	1016	1288	1286	1421
256								828	1168	1422	1586	1708	1782
128							840	1300	1625	1861	2031	2172	2334
64						947	1347	1824	2159	2399	2533	2652	2823
32					1083	1593	1922	2347	2664	2917	3071	3263	3396
16				1589	2276	2771	2995	3428	3793	3967	4062	4234	4474
8			2747	3387	4110	4468	4742	5113	5528	5780	5800	5875	6224
4		5316	5971	6772	7164	7364	7888	8122	8273	8518	8958	8997	9515
2	643	1689	2066	3183	4141	4198	4856	4772	5405	5562	6014	5775	6038
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.23: Buzhashing Laufzeit für proteins.200MB in s *farbig*

threshold													
8192													185
4096												177	266
2048											181	254	344
1024										196	266	339	432
512									206	290	364	459	557
256								213	297	373	450	570	630
128							220	313	400	473	557	679	709
64						218	313	402	489	572	633	780	879
32					227	328	425	522	597	676	759	903	984
16				251	360	455	556	650	737	827	894	1089	1114
8			291	392	511	605	701	796	887	961	1041	1204	1310
4		376	473	607	683	812	902	992	1086	1172	1235	1326	1522
2	409	431	530	636	744	865	925	1036	1107	1228	1309	1507	1517
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.24: Rabin-Karp-Hash Laufzeit in s proteins.200MB in s *farbig*

threshold													
8192													261
4096												212	301
2048											265	370	401
1024										280	321	493	508
512									245	346	443	536	627
256								245	350	449	645	635	736
128							253	367	483	692	679	770	870
64						255	449	482	622	700	786	1048	1155
32					321	383	606	619	730	980	933	1207	1120
16				286	494	535	663	921	1050	1161	1085	1457	1279
8			325	461	692	841	819	945	1241	1156	1269	1596	1674
4		404	528	665	791	1066	1033	1148	1256	1584	1457	1554	1980
2	445	434	598	731	856	1004	1121	1236	1360	1557	1567	1921	1690
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.25: djb2 Laufzeit für english.200MB in s *farbig*

threshold													
8192													280
4096												345	449
2048											468	580	702
1024										606	761	892	1014
512									743	991	1111	1256	1381
256								815	1125	1387	1538	1644	1728
128							834	1279	1617	1828	1980	2143	2291
64						925	1313	1772	2046	2308	2479	2567	2689
32					1068	1546	1869	2298	2541	2826	3010	3158	3285
16				1530	2033	2547	2749	3160	3472	3736	3939	3983	4134
8			2948	3433	3978	4307	4637	4935	5459	5720	5774	5847	6039
4		1533	2561	3155	3591	3910	4229	4685	5032	5289	5576	5654	5496
2	765	1606	2227	3216	3844	4001	4349	4589	5076	5193	5460	5775	6066
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.26: Buzhashing Laufzeit für english.200MB in s *farbig*

threshold													
8192													186
4096												171	261
2048											179	245	351
1024										186	254	321	433
512									204	274	343	434	535
256								210	284	371	435	548	615
128							222	308	390	459	536	650	693
64						222	318	403	487	555	618	763	835
32					235	330	424	514	599	670	737	856	930
16				258	361	454	548	629	715	797	865	1030	1064
8			301	402	505	596	699	791	871	937	1028	1154	1308
4		296	416	519	616	721	811	906	986	1071	1129	1194	1394
2	376	359	473	579	681	781	868	964	1052	1128	1173	1377	1396
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.27: Rabin-Karp-Hash Laufzeit für english.200MB in s *farbig*

threshold													
8192													208
4096												255	297
2048											207	303	397
1024										223	389	404	494
512									238	332	425	515	605
256								244	426	441	538	620	716
128							253	364	463	557	659	748	827
64						260	373	580	583	680	773	864	948
32					267	385	497	611	872	815	903	994	1074
16				293	410	640	724	754	859	949	1060	1225	1251
8			334	460	692	694	989	923	1034	1123	1195	1634	1663
4		329	561	606	712	832	1139	1061	1414	1268	1364	1712	1696
2	415	506	543	673	805	932	1047	1151	1306	1401	1457	1722	1587
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.28: djb2 Laufzeit für dblp.xml.200MB in s *farbig*

threshold													
8192													279
4096												332	451
2048											451	588	700
1024										557	753	891	1024
512									679	1008	1065	1252	1343
256								757	1060	1283	1437	1622	1751
128							803	1208	1438	1733	1892	1988	2110
64						926	1261	1590	1871	2145	2307	2423	2542
32					1107	1599	1898	2252	2477	2733	2923	3053	3185
16				1075	2011	2727	2928	3271	3471	3723	4050	4001	4154
8			933	1684	2572	3405	3848	4060	4582	4757	4487	4599	5021
4		1040	1306	2520	3145	3052	3650	4028	4290	4193	5028	4872	5286
2	938	1200	1564	2410	3664	3918	4412	4086	4581	4577	5016	4817	5464
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.29: Buzhashing Laufzeit für dblp.xml.200MB in s *farbig*

threshold													
8192													154
4096												145	203
2048											151	191	262
1024										166	208	246	321
512									179	226	269	322	392
256								215	271	310	346	444	467
128							224	316	368	416	453	532	569
64						226	319	407	461	509	556	632	698
32					231	333	430	509	574	612	653	759	792
16				218	336	436	536	628	677	727	767	896	910
8			227	323	446	553	640	743	789	838	877	1000	1054
4		251	331	424	534	665	718	815	873	924	971	994	1139
2	356	343	362	448	618	698	807	878	942	966	1064	1160	1184
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.30: Rabin-Karp-Hash Laufzeit für dblp.xml.200MB in s *farbig*

threshold													
8192													224
4096												174	313
2048											180	230	401
1024										193	332	295	349
512									206	272	320	369	422
256								253	400	385	429	645	538
128							263	374	432	500	543	605	860
64						318	373	590	549	770	850	720	778
32					266	392	507	738	828	738	804	852	903
16				254	390	518	781	752	815	871	926	1218	1325
8			346	487	526	654	939	1079	949	1011	1071	1330	1205
4		285	378	637	628	784	854	986	1289	1104	1467	1211	1508
2	392	384	433	538	731	870	973	1073	1163	1215	1327	1483	1325
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.31: djb Zeit für einstein.de.txt in s

threshold													
8192													76
4096												90	127
2048											99	134	171
1024										116	151	175	220
512									106	176	191	220	267
256								111	152	225	249	279	306
128							109	157	205	285	285	321	356
64						99	158	200	246	337	326	359	403
32					85	146	206	258	286	398	366	403	451
16				96	134	201	269	300	347	457	409	450	492
8			96	134	173	264	320	347	400	511	464	500	532
4		116	132	183	212	258	377	400	437	568	541	542	577
2	163	150	211	207	250	322	359	401	483	514	552	573	616
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.32: Buzhashing Laufzeit in s für einstein.de.txt *farbig*

threshold	window 4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384
8192													58
4096												59	86
2048											61	88	112
1024										59	88	117	141
512									58	88	121	148	169
256								92	89	114	146	176	192
128							89	139	119	148	176	206	221
64						85	135	191	146	178	210	237	281
32					78	124	158	229	181	209	237	267	288
16				83	115	163	224	259	202	234	262	294	309
8			76	134	149	200	244	224	225	266	290	324	330
4		110	109	176	203	241	291	289	255	289	321	351	356
2	161	137	195	220	238	285	342	326	286	323	347	361	390

Tabelle 4.33: Rabin-Karp-Hash Laufzeit in s für einstein *farbig*

threshold	window 4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384
8192													72
4096												73	109
2048											77	112	142
1024										74	110	149	180
512									72	111	151	188	217
256								91	112	145	185	223	246
128							95	156	152	187	225	263	283
64						75	139	187	186	226	267	302	347
32					95	110	199	233	228	265	300	341	366
16				96	138	144	259	289	260	299	334	376	394
8			90	145	144	177	291	275	290	338	375	412	422
4		123	128	194	178	212	304	345	323	369	409	447	456
2	174	157	214	246	210	327	383	329	363	405	445	466	496

Tabelle 4.34: Buzhashing Laufzeit in s für random *farbig*

threshold													
8192													58
4096												68	72
2048											70	85	103
1024										81	93	105	122
512									75	110	113	130	154
256								86	103	144	156	162	177
128							96	116	138	184	176	192	212
64						97	126	147	165	215	201	221	246
32					99	137	169	190	210	254	237	257	283
16				114	142	182	213	226	250	308	285	299	324
8			123	163	192	241	269	286	306	365	337	353	377
4		159	195	254	269	303	360	373	388	456	441	434	406
2	395	395	486	506	508	554	602	594	654	661	681	692	707
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

Tabelle 4.35: Rabin-Karp-Hash Laufzeit für random.100MB in s

threshold													
8192													79
4096												80	105
2048											82	103	129
1024										95	113	128	150
512									87	130	140	160	188
256								98	124	173	185	200	221
128							110	134	162	218	210	240	257
64						109	145	174	201	255	245	268	297
32					110	156	191	226	238	300	285	309	340
16				129	160	209	246	264	294	354	330	356	382
8			133	181	215	271	313	327	354	426	395	417	446
4		169	212	274	296	339	401	411	441	520	584	505	533
2	392	398	503	527	529	586	638	646	699	722	758	742	770
window	4	8	16	32	64	128	256	512	1024	2048	4096	8129	16384

5. Fazit

Innerhalb dieser Arbeit wurden die ersten beiden Phasen des in *Approximation LZ77 via Small-Space Multiple-Pattern Matching* beschriebenen Algorithmus zur Approximierung der *LZ77-Faktorisierung* umgesetzt [8].
Die Anwendung bietet eine bessere Laufzeit und geringeren Speicherbedarf gegenüber *lzss_lcp*. Allerdings resultiert daraus eine schlechtere Kompression.
Die erstellte Implementation ist in das *TU Dortmund Compression Framework* integriert. Trotz der fehlenden dritten Phase zeigt sich eine bessere Bilanz für den Speicherbedarf, die vermutlich auch mit einer dritten Phase bestehend bleibt.
Im Allgemeinen konnte gezeigt werden, dass die Approximation der LZ-Faktorisierung auch in der praktischen Umsetzung akzeptable Resultate liefert.
Die Implementation ist eng angelegt an die theoretische Beschreibung. Das Programm weicht darin ab, dass wir eine Schranke für die maximale Faktorlänge haben. Es hat sich experimentell gezeigt, dass dies speichertechnisch besser ist und auch die Kompression nicht stark beeinflussen sollte, da wir Faktoren mergen. Trotzdem stellt dies eine deutliche Abweichung der Implementation dar.
Als sinnvolle Grenzen stellten sich 16 Byte als minimale Faktorlänge und eine maximale Länge von 4096 Byte heraus. Dabei ist die maximale Länge eine eher schwächere Aussage, da die Werte um 4096 Byte ähnliche Ergebnisse erzielen. Man könnte zwar größere Werte wählen und so für manche Eingaben bessere Kompression und besseren Speicherbedarf erzielen, aber für die hier getesteten Eingaben traf dies selten zu. Oft führten größere Werte nur zur längerer Laufzeit.
Es hat sich gezeigt, dass 64Bit Hashwerte zu Hashmaps mit weniger Speicherverbrauch führen können, als die Verwendung von 32Bit Hashwerte. Und das klassische Rabin-Karp-Fingerprints langsamer sind als andere Hashmethoden ohne diesen gegenüber keine ersichtlichen Vorteile zubringen.

6. Ausblick

In dieser Arbeit wurden nur die ersten beiden Phasen des Algorithmus umgesetzt. Die Implementation einer dritten Phase würde die Kompressionsrate natürlich verbessern, wenn auch vermutlich verlangsamen. Es existieren jedoch auch einige Möglichkeiten zu weiteren Verbesserung des vorhandenen Programms.

Bessere Quellpositionen Die Implementation erzeugt als Quellposition oft das erste Vorkommen des Substrings. Um eine bessere Kompression zu erreichen, könnte man stattdessen immer das letzte Vorkommen vor dem String selber als Quellposition benutzen. Dies würde vermutlich aber das Programm verlangsamen.

Redundanz zwischen Objekten Ein Chain Objekt hat nur aus einem Grund eine Position, um während der Suche überprüfen zu können, ob der rolling Hash vor dem Substring der Chain liegt. Da wir in der Liste der Chains sowieso die zusammengehörigen auf- und absteigenden Chains hintereinander speichern müssen, lässt sich die Position des einen immer aus der Position des anderen errechnen. Daher benötigt jedes zweite Chain Objekt keine Position mit zwei Byte, so könnten wir ein Byte pro Chain Objekt sparen.

Hashfunktionen Es existiert eine fast unbegrenzte Menge an Hashfunktionen. Im Laufe dieser Arbeit habe ich eine kleine Auswahl im Zusammenhang mit der Implementation getestet. Es ist sehr wahrscheinlich das bessere Hashfunktionen existieren. Diese könnten das Programm beschleunigen.

Relaxing der Suchparameter In Phase 2 des Algorithmus versuchen wir immer wieder zwei Faktoren zu mergen. Dazu suchen wir nach einem String der doppelt solang ist wie der längere Faktor. Wir vermeiden damit, dass drei aufeinanderfolgende Faktoren selber einen Faktor bilden können. Während wir es erlauben, dass zwei aufeinander folgende immer noch einen Faktor bilden können. Daher reicht es eigentlich aus wenn der Suchstring nur ein nicht leeres Präfix des dritten Faktors enthält. Der Suchstring muss also nur eine Länge haben die im Intervall $[|f_1| + |f_2| + 1 \cdots |f_1| + |f_2| + |f_3|]$ liegt. Solche Intervalle können sich für aufeinander folgende Runden überschneiden. Wenn wir dann die Länge

des Suchstrings so wählen, dass sie in beiden Intervallen liegt, können wir beide Runden gleichzeitig bearbeiten. Dies würde die Laufzeit von Phase 2 verkürzen.

longest common extension Während der Suche müssen wir oft überprüfen ob zwei Strings gleich sind. Mit Fingerprints schließen wir einen Großteil aus, aber wir können nicht garantieren, dass die Strings gleich sind. Das Programm überprüft dies mit der C++ internen *compare* Methode. Eine bessere Lösung für das Vergleichen von zwei Strings könnte die Implementation beschleunigen.
Die Größe der Hashmap ist für die meisten Eingaben der entscheidende Faktor für den Speicherbedarf des Programms.

Hashmaps Eine speichersparende Implementation oder auch nur eine andere Allokationsstrategie würde den Speicherbedarf senken. Vielleicht könnten diese auch 32Bit Hashwerte wieder interessant machen.

Literaturverzeichnis

[1] 14496-10:2014, ISO/IEC: *Coding of audio-visual objects — Part 10: Advanced Video Coding*. Technischer Bericht.

[2] Alakuijala, Jyrki, Andrea Farruggia, Paolo Ferragina, Eugene Kliuchnikov, Robert Obryk, Zoltan Szabadka und Lode Vandevenne: *Brotli: A General-Purpose Data Compressor*. ACM Transactions on Information Systems, 37:1–30, 12 2018.

[3] Cohen, Jonathan D.: *Recursive Hashing Functions for n-Grams*. ACM Trans. Inf. Syst., 15(3):291–320, Juli 1997.

[4] De Agostino, Sergio: *Lempel–Ziv Data Compression on Parallel and Distributed Systems*. Algorithms, 4, 09 2011.

[5] Deutsch, Peter: *DEFLATE Compressed Data Format Specification version 1.3*. RFC, 1951:1–17, 1996.

[6] Duce, David: *Portable Network Graphics (PNG) Specification (Second Edition)*. W3C Recommendation, W3C, November 2003. http://www.w3.org/TR/2003/REC-PNG-20031110/.

[7] Ferragina, Paolo und Gonzalo Navarro: *Pizza and Chili Corpus Compressed Indexes and their Testbeds*, 2005 (zugriff am 4-10-2020). `http://pizzachili.dcc.uchile.cl/index.html`.

[8] Fischer, Johannes, Travis Gagie, Paweł Gawrychowski und Tomasz Kociumaka: *Approximating LZ77 via Small-Space Multiple-Pattern Matching*. Seiten 533–544, 2015.

[9] Geldreich, Richard: *LZHAM - Lossless Data Compression Codec*, 2015 (zugriff am 20-Mai-2020). `https://github.com/richgel999/lzham_codec`.

[10] jsteemann, 2016 (zugriff am 1-November-2020).

[11] Kann, Viggo: *On the Approximability of NP-complete Optimization Problems*. 01 1992.

[12] Kärkkäinen, Juha, Dominik Kempa und Simon J. Puglisi: *Lightweight Lempel-Ziv Parsing.* In: Bonifaci, Vincenzo, Camil Demetrescu und Alberto Marchetti-Spaccamela (Herausgeber): *Experimental Algorithms*, Seiten 139–150, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[13] Karp, Richard M. und Michael O. Rabin: *Efficient randomized pattern-matching algorithms.* IBM Journal of Research and Development, 31(2):249–260, März 1987.

[14] Khairi, Nor Asilah und Bahari Jambek, Asral: *Study on data compression algorithm and its implementation in portable electronic device for Internet of Things applications.* EPJ Web Conf., 162:01073, 2017.

[15] Mailund, Thomas: *The Joys of Hashing: Hash Table Programming with C.* APress, 1st Auflage, 2019.

[16] Mohamadi, Hamid, Justin Chu, Benjamin P. Vandervalk und Inanc Birol: *ntHash: recursive nucleotide hashing.* Bioinformatics, 32(22):3492–3494, 07 2016.

[17] Nitto, Igor: *Parsing Algorithms for Data Compression.* Seiten 42–45, 2010.

[18] Pearson, Peter K.: *Fast Hashing of Variable-Length Text Strings.* Commun. ACM, 33(6):677–680, 1990.

[19] Salomon, David: *Data Compression: The Complete Reference*, Seiten 1–15. Springer-Verlag, 2006.

[20] Shmuel Tomi Klein, Yair Wiseman: *Parallel Lempel Ziv coding.* Discrete Applied Mathematics, 146:180–191, 2005.

[21] Shun, J. und F. Zhao: *Practical Parallel Lempel-Ziv Factorization.* In: *2013 Data Compression Conference*, Seiten 123–132, 2013.

[22] Storer, James A. und Thomas G. Szymanski: *Data Compression via Textual Substitution.* J. ACM, 29(4):928–951, Oktober 1982.

[23] Streib, James T. und Takako Soma: *Hashing*, Seiten 339–362. Springer International Publishing, Cham, 2017.

[24] Stroustrup, Bjarne: *The C++ Programming Language.* Addison-Wesley Professional, 4th Auflage, 2013.

[25] Ukil, A., S. Bandyopadhyay und A. Pal: *IoT Data Compression: Sensor-Agnostic Approach.* In: *2015 Data Compression Conference*, Seiten 303–312, 2015.

[26] Valenzuela, Daniel, Dmitry Kosolobov, Gonzalo Navarro und Simon Puglisi: *Lempel-Ziv-like Parsing in Small Space*, 03 2019.

[27] Ziv, J. und A. Lempel: *A universal algorithm for sequential data compression.* IEEE Transactions on Information Theory, 23(3):337–343, 1977.