technische universität
dortmund

Bachelorarbeit

Textkompression mithilfe von Enhanced Suffix Arrays

Patrick Dinklage
Februar 2015

Gutachter:
Prof. Dr. Johannes Fischer
Dominik Köppl

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl für Algorithm Engineering (LS11)
http://ls11-www.cs.tu-dortmund.de

Inhaltsverzeichnis

1 Einleitung

Die Kompression von Daten spielt in der Informatik eine wichtige Rolle. Es wird Rechenaufwand betrieben, um die Größe von Daten auf dem Datenträger zu verkleinern – etwa weil nur begrenzt Speicherplatz zur Verfügung steht, weil Transferzeiten verkürzt werden sollen usw.

Die grundlegende Idee hinter der Kompression ist die Eliminierung von Redundanzen in der Menge der zugrundeliegen Daten. Im Folgenden wird speziell die Kompression von Texten betrachtet. Diese muss *verlustfrei* arbeiten, d.h. der Text muss nach der Kompression wieder in die Originalfassung überführt werden können (während z.B. bei der Kompression von Bild und Ton ein gewisser Informationsverlust oft tolerierbar ist).

Die klassischen *Lempel-Ziv-Verfahren* indexieren den zu komprimierenden Text und ersetzen redundante Teile durch Referenzen auf zuvor gefundene Vorkommen [30, S. 75 ff.] [31]. Dadurch, dass diese Referenzen weniger Speicherplatz auf dem Datenträger erfordern, wird der Text komprimiert.

In dieser Arbeit wird ein Verfahren zur Kompression von Texten entwickelt, implementiert und bezüglich seiner Effizienz mit dem Lempel-Ziv-Verfahren *LZ77* verglichen.

1.1 Gliederung der Arbeit

In Abschnitt 1.2 folgt eine grobe Vorstellung der Idee für das in dieser Arbeit entwickelte Kompressionsverfahren.

Die vorausgesetzten *theoretischen Grundlagen* werden in Kapitel 2 eingeführt, in Kapitel 3 folgt dann die detaillierte Beschreibung des *Kompressionsverfahrens mit Enhanced Suffix Arrays* in Form der entwickelten Datenstrukturen und Algorithmen.

Kapitel 4 beschreibt die *Implementierung* eines Java-Programms, welches das entwickelte Verfahren in die Praxis umsetzt. Hiermit werden die *praktischen Tests* durchgeführt, die in Kapitel 5 spezifiziert und ausgewertet werden.

In Kapitel 6 wird abschließend ein Fazit aus den gewonnenen Erkentnissen gezogen und ein Ausblick auf mögliche Weiterentwicklungen geworfen.

Der Arbeit folgen zwei Anhänge. Anhang A enthält eine detaillierte Ausführung aller Ergebnisse des praktischen Tests sowie einige graphische Darstellungen dieser.

Anhang B beschreibt die Verwendung des entwickelten Programms, dessen Quellcode und ausführbare Version auf einem Datenträger (CD-R) beiliegen.

1.2 Idee für das Kompressionsverfahren

Das hier zu untersuchende Verfahren soll Redundanzen mithilfe des *Enhanced Suffix Arrays* (vgl. [3]; hier: *Suffix-Array* mit *LCP-Tabelle*) des zu komprimierenden Textes auffinden und eliminieren.

Da das Suffix-Array lexikographisch sortiert ist, liegen sich wiederholende Textphrasen darin benachbart vor. Jeder in der LCP-Tabelle enthaltene Präfix stellt somit unmittelbar eine Redundanz im Text dar. Ähnlich wie bei den Lempel-Ziv-Verfahren (siehe Abschnitt 2.5) sollen Redundanzen dieser Art durch Referenzen auf bereits gefundene Vorkommen ersetzt werden.

Während bei den Lempel-Ziv-Verfahren ausschließlich Rückwärtsreferenzen möglich sind (d. h. nur auf Textteile, die vor dem ersetzten Textteil stehen), können bei diesem Verfahren auch Referenzen auf spätere Textteile entstehen – je nach Ordnung des Suffix-Arrays.

Die Präfixe aus der LCP-Tabelle werden aufgrund ihrer Länge priorisiert. Lange Präfixe werden hierdurch bevorzugt und zuerst verarbeitet, längere redundante Teile werden demnach zuerst eliminiert.

Bei der Ersetzung eines Textteils kann es dazu kommen, dass ein Teil eines anderen in der LCP-Tabelle enthaltenen Präfixes eliminiert wird. Im Folgenden sprechen wir bei diesen Fällen von *Überschneidungen*. Die möglicherweise von Überschneidungen betroffenen Präfixe müssen nach einer Ersetzung daraufhin untersucht und ggf. verkürzt werden.

Abbildung 1.1 zeigt beispielhaft das Enhanced Suffix Array des Textes „banana". Man kann hieran die Eigenschaft erkennen, dass redundante Textteile direkt benachbart sind und dessen Länge der LCP-Tabelle zu entnehmen ist. Zum Beispiel haben der zweite und der vierte Suffix des Textes genau drei anfängliche Zeichen gemeinsam, und zwar `ana`.

Suffix-#	Suffix	LCP
6	`a`	$\perp$
4	`ana`	1
2	`anana`	3
1	`banana`	0
5	`na`	0
3	`nana`	2

Abbildung 1.1: Enhanced Suffix Array zum Beispieltext „banana".

Das Verfahren zur Findung der jeweils größten Redundanz, die Ermittlung von *Ersetzungsregeln* sowie die Korrektur der LCP-Tabelle bei etwaigen Überschneidungen wird in Kapitel 3 vorgestellt.

2 Theoretische Grundlagen

Dieses Kapitel dient zur Einführung der Datenstrukturen, Operationen und Methoden, welche dieser Arbeit zugrundeliegen.

2.1 Operationen auf Strings

Im Folgenden werden die in dieser Arbeit verwendendeten Operationen auf Strings definiert.

Sei Σ ein Alphabet. Die Elemente von Σ nennen wir *Symbole* (oder *Zeichen*). Es sei Σ^* die Menge der *Strings* (oder *Texte*) über Σ. Für ein $n \in \mathbb{N}_0$ sei $w \in \Sigma^n$ ein String der *Länge* $|w| = n$.

Definition 2.1 (Leerstring) Es sei $\epsilon \in \Sigma^*$ der *leere String* mit $|\epsilon| = 0$.

Definition 2.2 (Wiederholung) Für ein $x \in \Sigma$ sei $x^n \in \Sigma^*$ mit $n \in \mathbb{N}$ der String, der die n-malige Wiederholung des Symbols x darstellt.

Definition 2.3 (Symbolzugriff) Für ein $i \in \mathbb{N}$ mit $1 \leq i \leq n$ bezeichne $w[i] \stackrel{\text{df}}{=} x_i$ das i-te Symbol des Strings w.

Definition 2.4 (Teilstring) Für $i,j \in \mathbb{N}$ mit $1 \leq i < j \leq n$ bezeichne

$$w[i \ldots j] \stackrel{\text{df}}{=} w[i]w[i+1] \cdots w[j-1]w[j]$$

den *Teilstring* von w von der Stelle i bis zur Stelle j (jeweils einschließlich).

Definition 2.5 (Suffix) Für ein $k \in \mathbb{N}$ sei $w[k \ldots] \stackrel{\text{df}}{=} w[k \ \ldots \ n]$ mit $1 \leq k \leq n$ das *k-te Suffix* von w.

Definition 2.6 (Präfix) Für ein $k \in \mathbb{N}$ sei $w[\ldots k] \stackrel{\text{df}}{=} w[1 \ \ldots \ k]$ mit $1 \leq k \leq n$ der *Präfix der Länge k* von w.

Definition 2.7 (Konkatenation) Seien $w \in \Sigma^n$ und $v \in \Sigma^m$ Strings. Durch die *Konkatenation* von w und v entsteht der aus beiden Strings zusammengesetzte String

$$w \circ v \stackrel{\text{df}}{=} w[1] \cdots w[n]v[1] \cdots v[m]$$

der Länge $|w \circ v| = n + m$.

Definition 2.8 (Konkatenation mehrerer Strings) Seien w_1,w_2,...,$w_k \in \Sigma^*$ Strings. Wir führen für die Konkatenation mehrerer Strings folgende Schreibweise ein:

$$\bigodot_{i=1}^{k} w_i \stackrel{\text{df}}{=} w_1 \circ w_2 \circ \cdots \circ w_k.$$

Definition 2.9 (Lexikographische Ordnung) Eine *lexikographische Ordnung* $(\Sigma, \prec)$ bezeichnet eine totale Ordnung über die Symbole in Σ.

Für das Alphabet der englischen Großbuchstaben $\Sigma_{\text{engl}} = \{\texttt{A}, \texttt{B}, \ldots, \texttt{Z}\}$ ist zum Beispiel die *alphabetische Ordnung* $(\Sigma_{\text{engl}}, \prec_\alpha)$ mit $\texttt{A} \prec_\alpha \texttt{B} \prec_\alpha \cdots \prec_\alpha \texttt{Z}$ eine lexikographische Ordnung.

Definition 2.10 (Lexikographische Ordnung über Strings) Seien $w \in \Sigma^n$ und $v \in \Sigma^m$ Strings.

Eine lexikographische Ordnung $(\Sigma, \prec)$ induziert die lexikographische Ordnung $(\Sigma^*, \prec)$ über die Strings über Σ. Es ist $w \prec v$ genau dann, wenn gilt:

- $n < m$ und $w = v[\ldots n]$ oder
- es gibt ein $i \in \mathbb{N}$ mit $1 \leq i \leq \min(n,m)$, so dass gilt:
  - $w[\ldots i-1] = v[\ldots i-1]$ und
  - $w[i] \prec v[i]$.

2.2 Doppelt verkettete Listen

Definition 2.11 (Doppelt verkettete Liste) Sei X eine Menge und $X_\perp = X \cup \{\perp\}$. Das 5-Tupel

$$\begin{aligned} L^X = (&X_\perp, \\ &\text{head}_L \in X_\perp, \\ &\text{next}_L : X_\perp \to X_\perp, \\ &\text{prev}_L : X_\perp \to X_\perp, \\ &\text{tail}_L \in X_\perp) \end{aligned}$$

ist eine *doppelt verkettete Liste*[1] über Elemente aus X.

Das erste Element der Liste ist head_L, das letzte Element ist tail_L. Für ein Element $x \in X$ ist $\text{next}_L(x)$ der Nachfolger und $\text{prev}_L(x)$ der Vorgänger von x.

L^X heißt *leer*, falls $\text{head}_L = \text{tail}_L = \perp$.

[1] Die hier aufgestellte Definition der doppelt verketteten Liste ist eine Anlehnung an jene aus [9, S. 236 f.] mit einigen Anpassungen für diese Arbeit.

Folgende Eigenschaften sind stets erfüllt:

1. $\text{next}_L(\perp) = \text{prev}_L(\perp) = \perp$.
2. $\text{next}_L(\text{tail}_L) = \text{prev}_L(\text{head}_L) = \perp$ *(Das letzte Element hat keinen Nachfolger und das erste keinen Vorgänger).*
3. $\text{head}_L = \perp \Leftrightarrow \text{tail}_L = \perp$ *(Integrität der leeren Liste).*
4. $\text{head}_L \neq \perp \Leftrightarrow \text{tail}_L \neq \perp$ *(Integrität der nichtleeren Liste).*
5. $\text{next}_L(x_1) = x_2 \Leftrightarrow \text{prev}_L(x_2) = x_1$ für alle $x_1, x_2 \in X$ *(Integrität der Elemente).*

Statt $\text{next}_L(x_1) = x_2$ bzw. $\text{prev}_L(x_2) = x_1$ schreiben wir auch: $x_1 \overset{L}{\leftrightarrow} x_2$.

Wir betrachten im Rahmen der Arbeit nur *azyklische* doppelt verkettete Listen, d. h. es gibt kein Element, dessen Nachfolger es selbst oder einer seiner Vorgänger innerhalb der Liste ist:

$$\nexists x_i, x_j : \; x_i \overset{L}{\leftrightarrow} \cdots \overset{L}{\leftrightarrow} x_j \; \wedge \; x_j \overset{L}{\leftrightarrow} x_i.$$

2.3 Max-Heaps

Ein (binärer) *Heap* ist die Array-Repräsentation eines binären Baums, dessen Knoten mit Elementen aus einer Menge X beschriftet sind und welcher bezüglich einer totalen Ordnung $(X, \preceq)$ die *Heap-Eigenschaft* erfüllt. Diese verlangt, dass für einen Knoten x im Baum und seine Kinder x_l und x_r gilt:

$$x \preceq x_l \text{ und } x \preceq x_r.$$

Abbildung 2.1 zeigt beispielhaft die Baumstruktur eines Heaps über $(\mathbb{N}, \geq)$, welcher auch als *Max-Heap* bezeichnet wird [9, S. 151 f.]. In einem Max-Heap ist jeder Knoten demnach mit einem größerem Wert beschriftet als seine Kinder.

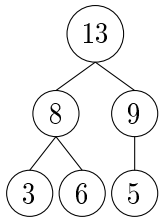


Abbildung 2.1: Beispiel für die Baumstruktur eines Max-Heaps.

Der Baum wird durch ein Array A repräsentiert, welches folgende Eigenschaften erfüllt:

1. An erster Stelle des Arrays steht die Wurzel des repräsentierten Baums.
2. Das linke Kind des i-ten Knotens steht an Stelle $2i$.
3. Das rechte Kind des i-ten Knotens steht an Stelle $2i + 1$.
4. Für $i > 1$ steht der Vaterknoten des i-ten Knotens an Stelle $\lfloor \frac{i}{2} \rfloor$.

Abbildung 2.2 zeigt die Array-Repräsentation des Baums aus Abbildung 2.1.

i:	1	2	3	4	5	6	7
A:	13	8	9	3	6	5	$\perp$

Abbildung 2.2: Array-Repräsentation des Max-Heaps aus Abbildung 2.1.

Aufgrund der Heap-Eigenschaft ist in einem Max-Heap die Wurzel des Baums größer als alle anderen Elemente im Heap. Hierdurch wird der Zugriff auf das größte Element in $O(1)$ Zeit ermöglicht (vgl. Punkt 1).

Die Einfüge-, Änderungs- und Löschoperationen eines Heaps müssen jeweils die Heap-Eigenschaft aufrechterhalten. Dies ist grundsätzlich in $O(\log n)$ Zeit bezüglich der Anzahl n von enthaltenen Elementen möglich [9, S. 153]. Für spezielle Anwendungen existieren Varianten von Heaps, die ein Einfügen in linearer Zeit ermöglichen, dafür jedoch ggf. Anforderungen an die darunterliegende Menge von Elementen stellen [6, 7, 12] [9, S. 505 ff.].

2.4 Enhanced Suffix Arrays

Als *Suffix-Array* wird eine lexikographisch sortierte Auflistung aller Suffixe[2] eines Strings bezeichnet [21, S. 2] (eine alternative Bezeichnung ist das *PAT-Array* [15]). Die *LCP-Tabelle* enthält die Längen des *LCP* (längster gemeinsamer Präfix, engl. „longest common prefix") je zweier benachbarter Einträge im Suffix-Array [21, S. 2]. Die Erweiterung eines Suffix-Arrays um weitere Informationen, wie etwa der LCP-Tabelle, wird als *Enhanced Suffix Array* bezeichnet [3].

Sei Σ ein Alphabet und $w \in \Sigma^n$ ein String über Σ.

Definition 2.12 (Suffix-Array) $\mathrm{SA}_w = (s_1, \ldots, s_n)$ ist das *Suffix-Array* zu w genau dann, wenn für alle Paare s_i,s_j mit $1 \leq i < j \leq n$ gilt: $w[s_i \ldots] \prec w[s_j \ldots]$.

Statt s_i schreiben wir fortan $\mathrm{SA}[i]$.

Definition 2.13 (LCP) Seien w,$v \in \Sigma^*$ Strings. $\mathrm{lcp}(w,v)$ ist die Länge des *längsten gemeinsamen Präfix* (LCP) von w und v genau dann, wenn gilt:

- $w[\ldots \mathrm{lcp}(w,v)] = v[\ldots \mathrm{lcp}(w,v)]$ und
- $w[\mathrm{lcp}(w,v)+1] \neq v[\mathrm{lcp}(w,v)+1]$.

Definition 2.14 (LCP-Tabelle) $\mathrm{LCP}_w = (p_1, \ldots, p_n)$ ist die *LCP-Tabelle* zu SA_w genau dann, wenn gilt:

- $p_1 = \perp$ und
- $p_i = \mathrm{lcp}(w[s_{i-1} \ldots], w[s_i \ldots])$ für alle $i \in \mathbb{N}$ mit $1 < i \leq n$.

Statt p_i schreiben wir fortan $\mathrm{LCP}[i]$.

[2] Der leere String ϵ wird hier nicht als Suffix eines Strings betrachtet.

Abbildung 2.3 zeigt beispielhaft das Suffix-Array sowie die LCP-Tabelle zum String „banana".

i:	1	2	3	4	5	6
$w[i]$:	b	a	n	a	n	a
SA[i]:	6	4	2	1	5	3
LCP[i]:	$\perp$	1	3	0	0	2

Abbildung 2.3: Suffix-Array zum String „banana" und die zugehörige LCP-Tabelle.

Es gibt verschiedenartige Verfahren zur Berechnung von SA_w und LCP_w, [11,14,17–20,23, 24,29], einige lösen dieses Problem in linearer Zeit bezüglich der Länge des Eingabetextes [17,20].

Ein Verfahren zur Konstruktion des Suffix-Arrays SA_w in lineaer Zeit ist der in [17] entwickelte und in [26, S. 61 ff.] ausgeführte *Skew*-Algorithmus. Dieser arbeitet nach dem Prinzip „Divide & Conquer": Er teilt den Eingabestring in zwei Teile mit einem Längenverhältnis 2:1 auf (wobei der kleinere Teil jedes dritte Symbol enthält, der größere den Rest), berechnet rekursiv die Suffix-Arrays der einzelnen Teile und führt die Ergebnisse schließlich zusammen.

Der Skew-Algorithmus kann so erweitert werden, dass die LCP-Tabelle LCP_w als Nebenprodukt des Suffix-Arrays berechnet wird [17] [26, S. 79 ff.]. Überdies kann sie mit dem einfachen Algorithmus *GetHeight* in linearer Zeit über das Suffix-Array ermittelt werden [18].

Definition 2.15 (Inverses Suffix-Array) Das *inverse Suffix-Array* ist die Abbildung $\mathrm{SA}_w^{-1} : \mathbb{N} \to \mathbb{N}$, welche einem Suffix (repräsentiert durch seine Position im Text) seine Position im Suffix-Array zuordnet. Für ein $i \in \mathbb{N}$ gilt:

$$\mathrm{SA}_w[\mathrm{SA}_w^{-1}(i)] = w[i \ldots].$$

2.5 Textkompression mit adaptiven Wörterbuchmethoden

Die *Lempel-Ziv-Verfahren zur Textkompression* basieren auf dem 1977 von Jacob Ziv und Abraham Lempel entwickelten Verfahren *LZ77*. Dieses wird in Unterabschnitt 2.5.1 näher betrachtet.

Es fällt in die Kategorie der *adaptiven Wörterbuchmethoden* zur Kompression von Texten. Bei dieser Art von Kompressoren wird, basierend auf dem Eingabetext, eine Menge mehrfach vorkommender *Phrasen* ermittelt, die in ein *Wörterbuch* aufgenommen werden. Im Eingabetext werden diese redundanten Phrasen durch Referenzen auf die jeweiligen Einträge im Wörterbuch ersetzt [30, S. 74 ff.].

Dadurch, dass die Wörterbuchreferenzen auf dem Datenträger weniger Speicherplatz erfordern als die (mehrfach) ersetzten Phrasen, entsteht die *Kompression*.

Bei den Lempel-Ziv-Verfahren dient der Eingabetext selbst als Wörterbuch. Das erste Vorkommen einer Phrase wird gespeichert, alle Folgevorkommen werden durch eine Referenz auf das erste Vorkommen ersetzt [30, S. 74 ff.].

Im Folgenden stellen wir diese Phrasen durch *Ersetzungsregeln* dar. Wir trennen überdies die Schritte *Kompression* und *Kodierung*.

Definition 2.16 (Ersetzungsregel) Eine *Ersetzungsregel* ist ein Tripel der Form $(z,q,n) \in \mathbb{N}^3$. Wir nennen z die *Zielposition*, q die *Quellposition* und n die *Länge* der Ersetzung.

Die Semantik einer Ersetzungsregel kann als eine Anweisung für den Kodierer verstanden werden: „Ersetze an Stelle z im Text n Symbole durch eine Referenz auf Stelle q."

Definition 2.17 (Kompressor) Ein *Kompressor* ist eine Funktion $\text{compress} : \Sigma^* \to \mathcal{P}(\mathbb{N}^3)$, die für einen Eingabetext eine Menge von Ersetzungsregeln ermittelt[3].

Definition 2.18 (Kodierer) Ein *Kodierer* ist eine Funktion $\text{code} : \Sigma^* \times \mathcal{P}(\mathbb{N}^3) \to \Gamma^*$, die einen Text mithilfe einer Menge von Ersetzungsregeln kodiert, indem es diese Regeln anwendet. Die Ausgabe kann in einem beliebigen Alphabet Γ vorliegen.

Definition 2.19 (Kompressionsrate) Sei $t \in \Sigma^*$ ein Text und $t' = \text{code}(t,\text{compress}(t))$ eine komprimierte Fassung von t. Wir nennen das Längenverhältnis zwischen der komprimierten Fassung und des Originaltextes $\frac{|t'|}{|t|}$ die erzielte *Kompressionsrate*.

Definition 2.20 (Dekodierer) Ein *Dekodierer* ist eine Funktion $\text{decode}_{\text{code}} : \Gamma^* \to \Sigma^*$, die einen durch code kodierten Text wieder in die Ursprungsform zurücküberführt. Für einen Text $t \in \Sigma^*$ und einen beliebigen Kompressor gilt demnach:

$$\text{decode}_{\text{code}}(\text{code}(t,\text{compress}(t))) = t.$$

Definition 2.21 (Quellposition) Sei $t \in \Sigma^*$ ein Text, $R = \text{compress}(t)$ eine Menge von für t ermittelten Ersetzungsregeln und $t' = \text{code}(t, R)$ eine komprimierte Fassung von t. Die *Quellposition* $q_i \in \mathbb{N}$ ist die Position, von der ein Dekodierer das Symbol $t[i]$ aus t' lesen muss:

$$q_i = \begin{cases} q + (i - z) & \text{, falls es eine Ersetzungsregel } (z,q,n) \in R \text{ gibt mit } z \leq i < z + n, \\ i & \text{sonst.} \end{cases}$$

Liegt i im Bereich einer Ersetzungsregel, zeigt q_i auf die entsprechende Stelle relativ zur Quellposition der Ersetzung. Andernfalls gilt für nicht ersetzte Symbole $q_i = i$.

Definition 2.22 (Ersetzungstiefe) Für das i-te Symbol in t sei $d_i \in \mathbb{N}$ die *Ersetzungstiefe*. Dies ist die Anzahl der Quellpositionen, die ein Dekodierer rekursiv verfolgen muss, um auf ein nicht ersetztes Symbol zu stoßen. Sie ist induktiv wie folgt definiert:

$$d_i = \begin{cases} d_{q_i} + 1 & \text{, falls } q_i \neq i, \\ 0 & \text{sonst.} \end{cases}$$

[3] Für eine Menge X bezeichne $\mathcal{P}(X)$ die *Potenzmenge* von X.

Für nicht ersetzte Symbole ($q_i = i$) ist die Ersetzungstiefe demnach stets 0, da keine Quellpositionen aufgelöst werden müssen.

2.5.1 LZ77

Das klassiche Verfahren *LZ77* ermittelt die Menge der Phrasen in einem Durchlauf über den Eingabetext.

Während des Durchlaufs wird ein Teil des bereits verarbeiteten Textes nach einem *Sliding-Window*-Prinzip vorgehalten, um in dem noch folgenden Text redundante Phrasen zu erkennen [30, S. 75 ff.] [31].

Bei LZ77 liegen die Phrasen in der Form $(\delta,n,x) \in \mathbb{N}\times\mathbb{N}\times\Sigma$ vor. Dabei ist δ der momentane Abstand zur Quellposition, n die Länge der Ersetzung und x das Folgesymbol aus dem Eingabealphabet. Die Semantik einer solchen Phrase ist demnach eine Anweisung an einen etwaigen Dekodierer, der bereits i Symbole dekodiert hat:

1. Falls $n > 0$, kopiere n Symbole von Position $i - \delta$ ans Ende des Ausgabestroms.
2. Schreibe x ans Ende des Ausgabestroms.

Abbildung 2.4 zeigt beispielhaft die durch LZ77 vorgesehene Kodierung des Eingabetextes „abababababc".

t:	`a`	`b`	`abababc`
t':	`(0,0,a)`	`(0,0,b)`	`(2,6,c)`

Abbildung 2.4: Durch LZ77 kodierter Text „abababababc".

Auffallend ist, dass auch einzelne Symbole durch Phrasen der Länge 0 ersetzt werden.

Das noch heute weiterentwickelte und weit verbreitete Verfahren *gzip* basiert auf LZ77 und bringt einige Optimierungen mit sich, wie die Indexierung des Textes während der Verarbeitung sowie die gesonderte Kodierung einzelner Symbole [30, S. 78 f.].

2.6 LZ-Faktorisierung mithilfe des Enhanced Suffix Array

Die Phrasenmenge nach *LZ77* lässt sich auch über das Enhanced Suffix Array ermitteln. Dies wurde erstmals durch [8] beschrieben, in [26, S. 125 ff.] wird ein effizientes und einfaches Verfahren hierzu vorgestellt.

Eine für diese Arbeit angepasste Variante hiervon ist Algorithmus 2.5. Für einen Text $t \in \Sigma^*$ ermittelt er die LZ-Faktorisierung in Form von Ersetzungsregeln. Damit ist er ein *Kompressor* gemäß Definition 2.17. Es werden dabei nur Ersetzungsregeln aufgenommen, die mindestens θ Symbole ersetzen.

2.6.1 Vorgehensweise des Algorithmus

Zu jedem Suffix i wird seine Umgebung im Suffix-Array betrachtet. Zunächst muss also der Index h im Suffix-Array ermittelt werden, der auf i zeigt. Hierzu wird das inverse Suffix-Array herangezogen.

In der Umgebung von h im Suffix-Array wird nun nach „links" ($h_l < h$) und nach „rechts" ($h_r > h$) jeweils der erste Eintrag gesucht, der auf einen Suffix zeigt, der im Text vor $\mathrm{SA}[h] = i$ steht (also $\mathrm{SA}[h_l] < i$ bzw. $\mathrm{SA}[h_r] < i$).

Sofern diese existieren, wird während der Suche die Länge der gemeinsamen LCPs p_l und p_r zwischen i und h_l bzw. h_r ermittelt – dieser entspricht genau dem Minimum der LCP-Längen, die „auf dem Weg" von h zu h_l bzw. h_r in der LCP-Tabelle liegen.

Ist das Maximum p der beiden gemeinsamen Präfixe mindestens so groß wie der Schwellwert θ, so wird eine Ersetzungsregel eingeführt: an Stelle i werden p Symbole mit einer Referenz auf $\mathrm{SA}[h_l]$ bzw. $\mathrm{SA}[h_r]$ ersetzt.

Eingabe: Text $t \in \Sigma^*$ mit $|t| = n$, Schwellwert $\theta \in \mathbb{N}$.
Ausgabe: Menge von Ersetzungsregeln $R \in \mathbb{N}^3$.

1. Berechne SA_t, SA_t^{-1} und LCP_t.
   Initialisiere $R := \emptyset$ und $i := 1$.
2. Falls $i > n$, brich ab.
3. Wähle $h := \mathrm{SA}_t^{-1}(i)$ (dann ist $\mathrm{SA}[h] = i$).
4. Finde das größte $h_l < h$ mit $\mathrm{SA}[h_l] < i$ und setze
   $p_l := \min\{\mathrm{LCP}[h_l], \ldots, \mathrm{LCP}[h]\}$. Existiert h_l nicht, setze $p_l := 0$.
5. Finde das kleinste $h_r > h$ mit $\mathrm{SA}[h_r] < i$ und setze
   $p_r := \min\{\mathrm{LCP}[h+1], \ldots, \mathrm{LCP}[h_r]\}$. Existiert h_r nicht, setze $p_r := 0$.
6. $p := \max\{p_l, p_r\}$.
   - Falls $p \geq \theta$:
     - $q := \begin{cases} \mathrm{SA}[h_l], \text{ falls } p = p_l \\ \mathrm{SA}[h_r], \text{ falls } p = p_r \end{cases}$.
     - $R := R \cup \{(i, q, p)\}$
   - $i := i + p$.
7. Springe zu Punkt 2.

Algorithmus 2.5: LZ-Faktorisierung mithilfe des Enhanced Suffix Arrays.

3 Textkompression mit Enhanced Suffix Arrays

In diesem Kapitel wird aufgrund der in Abschnitt 1.2 vorgestellten Idee für das Kompressionsverfahren mit dem Enhanced Suffix Array (im Folgenden *ESAComp)* ein Algorithmus und die dazu notwendigen Datenstrukturen entworfen. Darauf folgt die Dokumentation eines beispielhaften Durchlaufs.

Des Weiteren wird ein allgemeines Verfahren zur *Kodierung* des Eingabetextes mithilfe der ermittelten Ersetzungsregeln eingeführt und zwei verschiedene Kodierer definiert.

Abschließend wird ein allgemeiner Ansatz zur *Dekodierung* vorgestellt.

3.1 LCP-sortierte Suffix-Liste

Für das Kompressionsverfahren wird es nötig sein, in jedem Schritt Zugriff auf den Suffix im Suffix-Array zu haben, der den längsten LCP in der LCP-Tabelle aufweist, und diesen zu entfernen.

Wir definieren hierfür eine Datenstruktur, die diesen Zugriff in konstanter Zeit ermöglicht und deren Operationen zur Änderung und Aufrechterhaltung in nahezu konstanter Zeit durchgeführt werden können: Die *LCP-sortierte Suffix-Liste*.

3.1.1 Listenstruktur

Dabei handelt es sich um eine doppelt verkettete Liste L_t^{LCP} (fortan kurz L) über den Suffixen eines Textes $t \in \Sigma^*$ (repräsentiert durch ihre Positionen im Text). Diese sind darin absteigend nach ihren zugehörigen LCP-Längen in der LCP-Tabelle sortiert, d. h. für je zwei Suffixe i,$j \in \mathbb{N}$ mit $i \overset{L}{\leftrightarrow} j$ gilt: $\mathrm{LCP}[\mathrm{SA}_t^{-1}(i)] \geq \mathrm{LCP}[\mathrm{SA}_t^{-1}(j)]$.

Damit kann über head_L stets in konstanter Zeit auf das Suffix mit dem längsten LCP zugegriffen werden.

3.1.2 LCP-Index

Die Operationen zum Einfügen und Entfernen von Elementen arbeiten mit dem *LCP-Index* $I_L : \mathbb{N} \to \mathbb{N} \cup \{\bot\}$. Für ein $p \in \mathbb{N}$ ist $I_L(p)$ das erste Suffix in L, dessen LCP gleich p ist. Gibt es keinen solchen Suffix in L, so ist $I_L(p) = \bot$.

Für alle $k = I_L(p)$ mit $k \neq \bot$ gelten demnach folgende Eigenschaften:

1. $\mathrm{LCP}[\mathrm{SA}_t^{-1}(k)] = p$
   (Der LCP des Suffix k ist genau p lang).
2. $\mathrm{prev}_L(k) \neq \bot \;\Rightarrow\; \mathrm{LCP}[\mathrm{SA}_t^{-1}(\mathrm{prev}_L(k))] > p$
   (Hat k einen Vorgänger in L, so ist dessen LCP länger als jener von k).

Für das Einfügen in die Liste (siehe Unterabschnitt 3.1.3) ist es notwendig, für einen Wert p den ersten Suffix in L ausfindig zu machen, dessen LCP gleich *oder kürzer* ist als p. Hierzu führen wir die Funktion $\mathrm{lookup}_{I_L} : \mathbb{N}_0 \to \mathbb{N} \cup \bot$ ein, die für p jenen Suffix zurückgibt oder $\bot$, wenn kein solcher Suffix in L existiert.

Die Funktion ist folgendermaßen rekursiv definiert:

$$\mathrm{lookup}_{I_L}(p) = \begin{cases} I_L(p) & \text{, falls } I_L(p) \neq \bot, \\ \mathrm{lookup}_{I_L}(p-1) & \text{, falls } I_L(p) = \bot \text{ und } p > 0, \\ \bot & \text{, falls } p = 0. \end{cases}$$

3.1.3 Einfügen von Elementen

Wir nehmen an, ein Suffix i mit der LCP-Länge p soll in die Liste eingefügt werden. Sei $k = lookup_{I_L}(p)$.

Ist $k \neq \bot$, so bezeichnet k die Einfügeposition. Das Suffix i wird so in die Liste eingefügt, dass unmittelbar danach gilt: $i \overset{L}{\leftrightarrow} k$. Im Falle $k = \bot$ wird i ans Ende der Liste angefügt, so dass $\mathrm{tail}_L = i$ ist. In beiden Fällen bleiben die in Unterabschnitt 3.1.2 geforderten Eigenschaften erhalten.

Da i nun das erste Suffix in der Liste mit einem LCP kürzer oder gleich p ist, wird der LCP-Index für p auf i geändert, so dass nach der Einfügeoperation gilt: $I_L(p) = i$.

Beispiel

Gegeben sei eine LCP-sortierte Suffix Liste L mit den Einträgen $a\langle 4\rangle \overset{L}{\leftrightarrow} b\langle 2\rangle \overset{L}{\leftrightarrow} c\langle 2\rangle$ (in den eckigen Klammern stehen dabei jeweils die zugehörigen LCP-Längen) und dem LCP-Index $I_L = \{4 \mapsto a,\ 2 \mapsto b\}$ (für alle nicht aufgeführten LCP-Werte zeigt der Index auf $\bot$).

Es soll nun der Eintrag $x\langle 3\rangle$ in L eingefügt werden. Die Einfügeposition erhalten wir durch $\text{lookup}_{I_L}(3) = b$ (wegen $I_L(3) = \perp$ und $I_L(2) = b$ wird genau ein Rekursionsschritt ausgeführt).

Demnach wird x unmittelbar vor b in L eingefügt und der Index aktualisiert, so dass nach der Einfügeoperation gilt: $a\langle 4\rangle \overset{L}{\leftrightarrow} x\langle 3\rangle \overset{L}{\leftrightarrow} b\langle 2\rangle \overset{L}{\leftrightarrow} c\langle 2\rangle$ und $I_L = \{4 \mapsto a,\ 3 \mapsto x,\ 2 \mapsto b\}$.

3.1.4 Entfernen von Elementen

Wird ein Suffix i mit der LCP-Länge p aus der Liste entfernt, muss der Index für den Fall, dass er für p gerade auf i zeigt ($I_L(p) = i$), aktualisiert werden.

Hat i einen Nachfolger in der Liste ($\text{next}_L(i) \neq \perp$) und ist dessen LCP-Länge gleich p, so wird $I_L(p)$ auf $\text{next}_L(i)$ aktualisiert. Gibt es keinen Nachfolger oder ist dessen LCP ungleich p, so wird der LCP-Index für p invalidiert, so dass $I_L(p) = \perp$ gilt.

Beispiel

Gegeben sei die Liste L aus dem Einfügebeispiel mit $a\langle 4\rangle \overset{L}{\leftrightarrow} b\langle 2\rangle \overset{L}{\leftrightarrow} c\langle 2\rangle$ und $I_L = \{4 \mapsto a,\ 2 \mapsto b\}$.

Der Eintrag $a\langle 4\rangle$ soll aus L entfernt werden. Wegen $I_L(4) = a$ muss der Index aktualisiert werden. Der LCP des Nachfolgers ($b\langle 2\rangle$) ist nicht 4, daher wird der Index für den Wert 4 invalidiert. Nach dem Entfernen gilt also: $b\langle 2\rangle \overset{L}{\leftrightarrow} c\langle 2\rangle$ und $I_L = \{2 \mapsto b\}$.

3.2 LCP-Heap

Eine deutlich einfachere Alternative zur LCP-sortierten Suffix-Liste stellt der *LCP-Heap* dar.

Dieser ist ein Max-Heap, dessen Knoten mit den Suffixen des Eingabetextes beschriftet und nach den jeweiligen LCP-Längen sortiert sind. Aufgrund der Heap-Eigenschaft des Max-Heaps steht jener Suffix mit dem längsten LCP immer an erster Stelle.

Die Operationen zur Aufrechterhaltung der Struktur (Initialisierung, Einfügen oder Entfernen von Elementen) unterliegen jedoch, im Gegensatz zur LCP-sortierten Suffix-Liste, einer logarithmischen Laufzeit bezüglich der Anzahl der Elemente.

3.3 Kompression

Gemäß Definition 2.17 ist es die Aufgabe des Kompressors, Ersetzungsregeln zu ermitteln, mit dessen Hilfe ein Kodierer den Text in eine kürzere Fassung überführen kann.

Algorithmus 3.1 ermittelt eine Menge von Ersetzungsregeln für t mithilfe des Enhanced Suffix Arrays von t so, dass nach Ausführung keine gemeinsamen Präfixe der Länge $\theta \in \mathbb{N}$ (*Schwellwert*) oder länger zwischen zwei Suffixen mehr in der LCP-Tabelle erkennbar sind[4].

Die einzelnen Schritte des Algorithmus werden in den folgenden Unterabschnitten näher erläutert.

Eingabe: Text $t \in \Sigma^*$ mit $|t| = n$, Schwellwert $\theta \in \mathbb{N}$.
Ausgabe: Menge von Ersetzungsregeln $R \subseteq \mathbb{N}^3$.

1. Berechne SA_t und LCP_t und konstruiere die initiale *LCP-sortierte Suffix-Liste* L aus allen $\mathrm{SA}[i]$ mit $\mathrm{LCP}[i] \geq \theta$. Initialisiere $R := \emptyset$.
2. Ist L leer, brich ab.
3. Wähle m so, dass $\mathrm{SA}[m] = \mathrm{head}_L$ ist (also $\mathrm{LCP}[m]$ längster LCP ist).
   Führe eine neue Ersetzungsregel ein: $R := R \cup (\mathrm{SA}[m], \mathrm{SA}[m-1], \mathrm{LCP}[m])$.
4. Entferne alle Suffixe $\sigma_r \in [\mathrm{SA}[m], \mathrm{SA}[m] + \mathrm{LCP}[m])$ aus L.
5. Für alle Suffixe $\sigma_l \in [\mathrm{SA}[m] - \mathrm{LCP}[m] - 1, \mathrm{SA}[m])$ mit $\sigma_l > 1$ und $\sigma_l + \mathrm{LCP}[\mathrm{SA}_t^{-1}(\sigma_l)] \geq \mathrm{SA}[m]$:
   - Falls $\mathrm{SA}[m] - \sigma_l \geq \theta$,
     setze $\mathrm{LCP}[\mathrm{SA}_t^{-1}(\sigma_l)] := \mathrm{SA}[m] - \sigma_l$ und aktualisiere L.
   - Sonst entferne σ_l aus L.
6. Gehe zurück zu Punkt 2.

Algorithmus 3.1: Kompression mithilfe des Enhanced Suffix Arrays (*ESAComp*).

3.3.1 Initialisierung

Die Berechnung des Enhanced Suffix Arrays (SA_t und LCP_t) wird hier nicht weiter betrachtet. In die *LCP-sortierte Suffix-Liste* L werden alle Suffixe von t eingetragen, deren LCP mindestens θ lang ist. Die Menge der Ersetzungsregeln ist zunächst leer.

3.3.2 Abbruchbedingung

Der Algorithmus bricht ab, wenn L leer ist. Dies ist der Fall, wenn es keine gemeinsamen Präfixe der Länge θ oder länger zwischen zwei Suffixen mehr gibt.

[4] Der Algorithmus liefert jedoch keine *optimale* Menge von Ersetzungsregeln. In Unterabschnitt 6.2.7 wird ferner ein Beispiel gezeigt, dass es in gewissen Fällen bessere Faktorisierungen gäbe.

3.3.3 Einführen einer Ersetzungsregel

Ist L nichtleer, bezeichnet head_L jenen Suffix SA[m], welcher momentan eine der längsten Redundanzen im Text beinhaltet. Diese teilt er mit dem Suffix SA[$m-1$] und ist LCP[m] Symbole lang.

Es wird die Ersetzungsregel (SA[m], SA[$m-1$], LCP[m]) eingeführt („Ersetze LCP[m] Symbole an Stelle SA[m] durch eine Referenz auf SA[$m-1$].").

3.3.4 Entfernen von Teilredundanzen

Die Symbole SA[m] bis SA[m]+LCP[m] (exkl.) entfallen durch die eingeführte Ersetzung. Aus diesem Grunde können die jeweiligen Suffixe, die mit diesen Symbolen beginnen, nicht mehr Ziel von zukünftigen Ersetzungen sein. Sie werden daher, sofern vorhanden, aus L entfernt.

Wir betrachten diesen Fall am Beispieltext $t =$ `banana` mit dem Schwellwert $\theta = 2$. Durch den Algorithmus wird eine Ersetzungsregel eingeführt, welche den Teilstring t[2,4] = `ana` der Länge 3 durch eine Referenz auf Position $t[4\ldots]$ = `ana` ersetzt. Die Symbole 2-4 entfallen durch diese Ersetzung, also werden die entsprechenden Suffixe (2, 3 und 4) von einer zukünftigen Betrachtung ausgeschlossen.

3.3.5 Korrektur von Linksüberschneidungen

Durch die Ersetzung eines Textteils nach Punkt 3 können eventuell Teile eines anderen, noch nicht verarbeiteten LCP ersetzt werden, so dass dieser ungültig wird.

Die LCP-Längen dieser Suffixe müssen also nach der Ersetzung nach unten korrigiert werden. Die LCP-sortierte Suffix-Liste L muss entsprechend aktualisiert werden, damit ihre Eigenschaften bestehen bleiben. Würde der LCP eines Suffix mit seinem Vorgänger durch die Korrektur kürzer als der Schwellwert θ, wird der Suffix stattdessen einfach aus L entfernt.

Die betroffenen Überschneidungen können maximal LCP[m] lang sein. Wären sie länger, stünde das entsprechende Suffix an erster Stelle in L und wäre zuerst verarbeitet worden.

Es müssen nur die Linksüberschneidungen mit dem ersetzten Teil betrachtet werden. Alle Rechtsüberschneidungen wurden bereits vom in Unterabschnitt 3.3.4 beschriebenen Schritt eliminiert.

Wir betrachten wieder den Beispieltext $t =$ `banana` mit dem Schwellwert $\theta = 2$. Diesmal nehmen wir an, der Algorithmus ersetze den Teilstring t[4,6] = `ana` der Länge 3 durch eine Referenz auf Position $t[2\ldots]$ = `anana`, wie in Abbildung 3.2 verdeutlicht wird. Der Teilstring $t[2\ldots4]$ = `ana` stellt eine wie oben beschriebene Linksüberschneidung dar, da sein Suffix `a` auch einen Teil des ersetzten Textteils darstellt. Die LCP-Längen der Suffixe

$t[2\ldots] =$ `anana` und $t[3\ldots] =$ `nana` müssen nach der Ersetzung also jeweils um 1 nach unten korrigiert werden, da ihr Suffix `a` wegfällt.

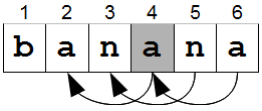


Abbildung 3.2: Beispiel für eine Linksüberschneidung bei einer Ersetzung.

3.4 Beispiel

Wir betrachten einen beispielhaften Durchlauf von Algorithmus 3.1 für den Eingabetext $t =$ `abcdebcdeabc` und den Schwellwert $\theta = 2$. Zur Übersicht ist t mit Nummerierung seiner Suffixe in Abbildung 3.3 abgebildet. Das zu t ermittelte Enhanced Suffix Array ist Abbildung 3.4 zu entnehmen.

k	1	2	3	4	5	6	7	8	9	10	11	12
$t[k]$	a	b	c	d	e	b	c	d	e	a	b	c

Abbildung 3.3: Der Eingabetext „abcdebcdeabc".

i	SA[i]	Suffix	LCP[i]
1	10	`abc`	$\perp$
2	1	`abcdebcdeabc`	3
3	11	`bc`	0
4	6	`bcdeabc`	2
5	2	`bcdebcdeabc`	4
6	12	`c`	0
7	7	`cdeabc`	1
8	3	`cdebcdeabc`	3
9	8	`deabc`	0
10	4	`debcdeabc`	2
11	9	`eabc`	0
12	5	`ebcdeabc`	1

Abbildung 3.4: Das Enhanced Suffix Array zum Text „abcdebcdeabc".

Abbildung 3.5 zeigt die initialen LCP-sortierte Suffix-Liste. Gemäß Algorithmus 3.1 sind nur diejenigen Suffixe SA[i] enthalten, für die LCP[i] $\geq \theta$ gilt. Dies trifft auf die Suffixe 2, 3, 1, 6 und 4 zu. Mit LCP[5] = 4 ist SA[5] = 2 das größte Element in der Liste.

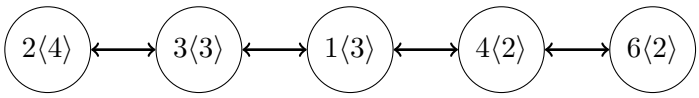


Abbildung 3.5: Initiale LCP-sortierte Suffix-Liste zum Text „abcdebcdeabc".

Weil die Liste nichtleer ist, fügen wir gemäß Punkt 3 des Algorithmus die erste Regel $(\mathrm{SA}[5], \mathrm{SA}[4], \mathrm{LCP}[5]) = (2,6,4)$ in die bislang noch leere Regelmenge R ein.

Nach Punkt 4 des Algorithmus werden nun die Suffixe 2, 3, 4 und 5 – sofern vorhanden – aus der Liste entfernt. Abbildung 3.6 zeigt die Liste nach dieser Operation.

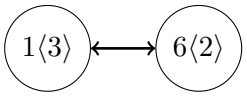


Abbildung 3.6: LCP-sortiere Suffix-Liste nach Entfernen von Teilredundanzen.

Des Weiteren gibt es eine Linksüberschneidung des ersetzten Textteils durch den Suffix $\mathrm{SA}[2] = 1$, dessen LCP $\mathrm{LCP}[2] = 3$ lang ist. Gemäß Punkt 5 korrigieren wir diesen auf $\mathrm{LCP}[2] := \mathrm{SA}[5] - \mathrm{SA}[2] = 2 - 1 = 1$. Wegen $1 < \theta$ wird das Suffix $\mathrm{SA}[2] = 1$ aus der LCP-sortierten Suffix-Liste entfernt, so dass diese nur noch den Suffix 6 enthält (Abbildung 3.7).

Abbildung 3.7: LCP-sortiere Suffix-Liste nach Korrektur von Linksüberschneidungen.

Der zweite Durchlauf des Algorithmus ist trivial. Gemäß Punkt 3 wird die neue Regel (SA[4],SA[3],LCP[4]) = (6,11,2) eingeführt. Durch Punkt 4 des Algorithmus wird das Suffix 6 sodann aus der Liste entfernt, welche daraufhin leer ist.

Der Algorithmus terminiert schließlich mit der Regelmenge $R = \{(2,6,4), (6,11,2)\}$.

Der Eingabetext „abcdebcdeabc" könnte mithilfe dieser Regeln beispielsweise wie in Abbildung 3.8 kodiert werden.

k	1	2	3	4	5	6	7	8	9	10	11	12
t'_k	a	[6,4]				[11,2]		d	e	a	b	c

Abbildung 3.8: Beispielkodierung des Textes „abcdebcdeabc" mithilfe der ermittelten Regeln.

3.5 Kodierung

Wurde zum Text $t \in \Sigma^*$ die Menge von Ersetzungsregeln $R \subseteq \mathbb{N}^3$ gemäß Algorithmus 3.1 ermittelt, kann der Text kodiert werden. Wird der Schwellwert θ passend zu einer geschickten Repräsentation der Regeln in R gewählt, wird durch die Kodierung eine *Kompression* von t erzielt.

Wir betrachten nun ein abstraktes Kodierungsverfahren, welches beliebig variiert werden kann, um eine bessere Kompressionsrate zu erzielen.

Die Grundidee ist hierbei die, dass auf den Text t alle Ersetzungsregeln in R angewendet werden. Das Resultat ist eine Mischung aus nicht ersetzten (jedoch ggf. gesondert kodierten) *Rohsymbolen* aus Σ und aus Repräsentationen der Ersetzungsregeln. Um für einen Dekodierer Rohsymbole und Regeln unterscheidbar zu machen, führen wir die *Ankündigungssymbole* $\rho,\bar{\rho} \notin \Sigma$ ein, die jeweils vor ein Rohsymbol bzw. eine kodierte Regel geschrieben werden.

Ferner definieren wir die Funktionen

- encodeRaw : $\Sigma \to \Gamma^*$,
- encodeSource : $\mathbb{N} \to \Gamma^*$ und
- encodeLength : $\mathbb{N} \to \Gamma^*$.

Sie kodieren jeweils Rohsymbole (*encodeRaw*), die Quellposition einer Regel (*encodeSource*) und die Länge einer Ersetzung (*encodeLength*) in einen String über ein frei wählbares Ausgabealphabet Γ. In dieser Kodierung muss jeweils die Information über ihre eigene Länge enthalten sein oder sie muss fix definiert werden. Die Zielposition der Regel ist durch die Position im kodierten Text bereits implizit gegeben und muss nicht kodiert werden.

Algorithmus 3.9 kodiert t unter Verwendung dieser Definitionen.

Im Folgenden werden darauf basierend zwei Referenzkodierer definiert, die im praktischen Test (Kapitel 5) herangezogen werden.

3.5.1 code1

Der Kodierer *code1* soll eine einfache Kodierung des Textes erzeugen, ohne irgendwelche Eigenschaften der zugrundeliegenden Sprache o. ä. auszunutzen. Hierdurch werden Kodierungen eines Textes mit verschiedenen Regelmengen leicht vergleichbar.

code1 überführt den zu kodierenden Text in eine Folge von *Bytes*, so dass wir das Ausgabealphabet $\Gamma = [0{,}255]$ festlegen[5]. Das Symbol $\bar{\rho} = 0$ wird dabei als Ankündigungssymbol

[5] Es wird vorausgesetzt, dass der Text einem Zeichensatz vorliegt, in welchem jedes Zeichen durch 8 Bit kodiert werden kann, wie z. B. ASCII. Für erweiterte Zeichensätze muss die Kodierung entsprechend angepasst werden.

Eingabe: Text $t \in \Sigma^*$, Menge $R \subseteq \mathbb{N}^3$ von Ersetzungsregeln.

Ausgabe: Kodierter Text $t' \in (\Gamma \cup \{\rho,\bar{\rho}\})^*$.

1. Initialisiere $t' := \text{encodeLength}(|t|)$ und den Zeiger $p := 1$.
2. Ist $R = \emptyset$, springe zu Punkt 8.
3. Wähle und entferne die Regel (z,q,n) mit der kleinsten Zielposition z aus R.
4. Falls $p < z$: $\quad t' := t' \circ \left(\bigodot_{i=p}^{z-1} \rho \circ \text{encodeRaw}(t[i]) \right)$.
   (Kodierung der Rohsymbole bis zur nächsten Ersetzung)
5. $t' := t' \circ \bar{\rho} \circ \text{encodeSource}(q) \circ \text{encodeLength}(n)$.
   (Kodierung der Ersetzung)
6. $p := z + n$.
7. Gehe zurück zu Punkt 2.
8. Falls $p < |t|$: $\quad t' := t' \circ \left(\bigodot_{i=p}^{|t|} \rho \circ \text{encodeRaw}(t[i]) \right)$.
   (Kodierung der übrigen Rohsymbole nach der letzten Ersetzung)

Algorithmus 3.9: Kodierung eines Textes mithilfe der Ersetzungsregeln.

für eine Ersetzungsregel reserviert[6]. Für Rohsymbole wird auf ein Akündigungssymbol verzichtet, daher wählen wir $\rho = \epsilon$.

Sei $ord : \Sigma \to [0{,}255]$ die *Ordinalfunktion*, die jedem Zeichen seine Repräsentation als Byte zuweist. Dann wählen wir $\text{encodeRaw} = ord$.

Sei $int_n : \mathbb{N}_0 \to [0{,}255]^n$ überdies die Funktion, die eine positive Ganzzahl in eine Darstellung durch n Bytes überführt[7]. Wir wählen $\text{encodeSource} = int_{\lceil \log_{256}(|t|) \rceil}$. Die Quellposition einer Ersetzungsregel wird also durch so viele Bytes dargestellt, wie für die Darstellung der Länge des Textes notwendig wäre. Des Weiteren wählen wir $\text{encodeLength} = int_{\lceil \log_{256}(n_{R,\max}) \rceil}$. Dabei ist $n_{R,\max}$ die Länge der längsten Ersetzung in R.

Damit sind alle notwendigen Definitionen für die Anwendung von Algorithmus 3.9 gegeben. Für einen Dekodierer müsste die Länge der längsten Ersetzung $n_{R,\max}$ zusätzlich gespeichert werden.

3.5.2 code2

Der Kodierer *code2* hat das Ziel, unter Ausnutzung geeigneter Kriterien eine möglichst kleine Darstellung des Quelltextes zu erzeugen. Hierzu wurden die Ergebnisse des Kodierers *code1* (vgl. Unterabschnitt 3.5.1) beobachtet und experimentiell eine deutlich effizientere Kodierung für längere Texte ermittelt.

[6] Es ist anzumerken, dass hierdurch die maximale Alphabetgröße des zugrundeliegenden Textes auf 255 beschränkt wird. Binärdateien über dem Alphabet [0,255] können mit *code1* daher nicht kodiert werden.

[7] Hierbei ist es willkürlich, ob die Darstellung im Format *little endian* oder *big endian* erfolgt, solange ein etwaiger Dekodierer dasselbe verwendet.

Im Gegensatz zu *code1* arbeitet *code2* auf Bit-Ebene. Als Ausgabealphabet wählen wir daher $\Gamma = \{0,1\}$. Jedes kodierte Rohsymbol bzw. jede kodierte Regel folgt auf ein *Ankündigungsbit*, anhand dessen ein Dekodierer diese auseinanderhalten kann. So definieren wir $\rho = 0$ für Rohsymbole und $\bar{\rho} = 1$ für Regeln.

Sei $int_n : \mathbb{N}_0 \to \{0,1\}^n$ nun die Funktion, die eine positive Ganzzahl in eine Darstellung durch n Bits überführt.

Kodierung von Ersetzungsregeln

Eine Untersuchung der mithilfe von Algorithmus 3.1 erzeugten Regeln ergab, dass kürzere Ersetzungen deutlich häufiger vorkommen als längere.

Demnach bietet sich für die Länge der Ersetzungen eine Kodierung variabler Länge an. Um dies weiter zu begünstigen, kann noch ausgenutzt werden, dass alle Ersetzungen mindestens θ lang sind. Statt der Länge n einer Ersetzung kann also die Differenz $\Delta_\theta^n = n - \theta$ kodiert werden.

Die besten Ergebnisse lieferte eine Unterteilung des Wertebereichs für Δ_θ^n in die Intervalle [0,7], [8,15], [16,31] und [32,$\Delta_\theta^{n_{R,\max}}$]. Zwei Bits dienen der Einteilung in eines der Intervalle, eine fixe Anzahl von Folgebits kodiert den eigentlichen Wert innerhalb des Intervalls. Tabelle 3.10 zeigt diese Kodierung für die einzelnen Intervalle. Die 32 häufigsten Ersetzungslängen können so jeweils in insgesamt 6 bzw. 5 (die 16 häufigsten) kodiert werden. Die Funktion encodeLength kodiert die Länge einer Ersetzung in eine entsprechende Bitfolge.

Intervall für Δ_θ^n	Einteilungsbits	Anzahl Folgebits
[0,7]	00	3
[8,15]	01	3
[16,31]	10	4
[32,$\Delta_\theta^{n_{R,\max}}$]	11	$\lceil \log_2(\Delta_\theta^{n_{R,\max}} - 32) \rceil$

Tabelle 3.10: Variable Kodierung der Ersetzungslängen bei *code2*.

Die Quellpositionen der erzeugten Ersetzungsregeln sind sehr gleichmäßig über den Text verteilt, so dass hier keine Eigenschaft ausgenutzt werden kann, um die Kodierung weiter zu optimieren[8]. Wir definieren daher einfach encodeSource $= int_{\lceil \log_2(|t|) \rceil}$.

Kodierung von Rohsymbolen

Vor allem in Texten natürlicher Sprachen ist zu beobachten, dass Symbole ungleichmäßig verteilt sind. Zum Beispiel kommt in deutschsprachigen Texten der Buchstabe *e* am häufig-

[8] Es ist anzumerken, dass die Kodierung der Quellpositionen das größte Problem bei der Findung einer minimalen Kodierung des Textes darstellt. Trotz großer Bemühungen konnte keine effizientere Variante ermittelt werden.

sten vor [5]. Auch für die nicht ersetzten Rohsymbole liegt daher eine Kodierung variabler Länge nahe.

Hierzu wird zunächt eine Rangordnung $rank_t : \Sigma \to \mathbb{N}_0$ ermittelt, die jedem Symbol $\varphi \in \Sigma$ einen Rang bezüglich der Häufigkeit im Text zuordnet. Das häufigste Symbol erhält den Rang 0, das zweithäufigste 1, usw.

Anstelle eines Rohsymbols $\varphi \in \Sigma$ wird also $rank_t(\varphi)$ gemäß Tabelle 3.11 mit variabler Länge kodiert. Die genaue Kodierung wird zusätzlich von der Größe des Eingabealphabets Σ abhängig gemacht. Für Texte mit kleinen Alphabeten (z.B. DNA-Sequenzen) lohnt sich beispielsweise keine Kodierung, welche pro Symbol mindestens fünf Bits erfordert. Währenddessen ist es jedoch sinnvoll, für Texte mit großen Alphabeten (z.B. englischsprachige Texte) den Wertebereich für $rank_t(\varphi)$ in mehrere Intervalle aufzuteilen. Die Funktion encodeRaw erzeugt eine entsprechende Kodierung.

$\lceil \log_2(\lvert\Sigma\rvert) \rceil$	$rank_t(\varphi)$	Einteilungsbits	Anzahl Folgebits
[0,5]	[0,3]	`0`	2
	[4,$\lvert\Sigma\rvert$]	`1`	$\lceil \log_2(\lvert\Sigma\rvert) \rceil$
6	[0,7]	`00`	3
	[8,15]	`01`	3
	[16,31]	`10`	4
	[32,$\lvert\Sigma\rvert$]	`11`	$\lceil \log_2(\lvert\Sigma\rvert) \rceil$
[7,∞)	[0,3]	`000`	2
	[4,7]	`001`	2
	[8,11]	`010`	2
	[12,15]	`011`	2
	[16,23]	`100`	3
	[24,31]	`101`	3
	[32,39]	`110`	3
	[40,$\lvert\Sigma\rvert$]	`111`	$\lceil \log_2(\lvert\Sigma\rvert) \rceil$

Tabelle 3.11: Variable Kodierung der Symbolränge bei *code2*.

Das Alphabet muss einem Dekodierer mitsamt seiner Rangzuordnung bekannt sein und zusätzlich abgespeichert werden.

Kodierung von Trigrammen

Man kann feststellen, dass es abhängig von der Länge des Quelltextes Schwellwerte θ gibt, für welche sich eine Kodierung nicht lohnt. In diesen Fällen wird die Kodierung der entsprechenden Ersetzungsregeln länger als die Kodierung der Folge von Rohsymbolen.

Für längere Texte kann man allgemein einen lohnenswerten Schwellwert von $\theta > 3$ annehmen. Dies bewirkt, dass einige sehr häufige *Trigramme*[9] (wie z. B. die Artikel „der", „die", „das") nur innerhalb längerer Phrasen ersetzt werden, also durchaus noch einige Vorkommen im übrigen (nicht ersetzten) Rohtext stehenbleiben. Es hat sich als sinnvoll erwiesen, die η häufigsten übrigen Trigramme durch ein jeweils neues Symbol zu ersetzen, welches dann in das Alphabet aufgenommen wird. Hierdurch entsteht das *erweiterte Alphabet* $\Sigma' \subseteq (\Sigma \cup \Sigma^3)$ der Größe $|\Sigma'| = |\Sigma| + \eta$. Die Kodierung der neu eingeführten Symbole geschieht wie in Tabelle 3.11, allerdings dann im Bezug auf Σ'.

η sollte von der Größe des Eingabealphabets Σ abhängig gemacht werden. Als effizient erwies sich die Wahl von η so, dass gilt:

$$\log_2(|\Sigma'|) \leq \log_2(|\Sigma|) + 2.$$

Die binäre Darstellung der unkodierten Rohsymbole soll sich durch Erweiterung des Alphabets also nur um höchtens 2 Bits verlängern. Für *code2* wird gewählt:

$$\eta = \begin{cases} |\Sigma| & \text{, falls } |\Sigma| \text{ Zweierpotenz ist } (\exists a \in \mathbb{N} : 2^a = |\Sigma|), \\ 2^{b+2} - |\Sigma| \text{ mit } b = \lceil \log_2(|\Sigma|) \rceil & \text{sonst.} \end{cases}$$

3.6 Dekodierung

Um einen durch Algorithmus 3.9 kodierten Text $t' \in (\Gamma \cup \{\rho,\bar{\rho}\})^*$ in den ursprünglichen Text $t \in \Sigma$ zu überführen, muss ein passender Dekodierer definiert werden.

Er besteht entsprechend der Definition des Kodierers in Abschnitt 3.5 aus den Funktionen

- decodeRaw : $(\Gamma \cup \{\rho, \bar{\rho}\})^* \to \Sigma \times \mathbb{N}$,
- decodeSource : $(\Gamma \cup \{\rho, \bar{\rho}\})^* \to \mathbb{N} \times \mathbb{N}$ und
- decodeLength : $(\Gamma \cup \{\rho, \bar{\rho}\})^* \to \mathbb{N} \times \mathbb{N}$.

Diese dekodieren ein Rohsymbol, die Quellposition einer Ersetzung bzw. dessen Länge. Sie geben ferner in Form einer zweiten Ausgabe Auskunft darüber, wie viele Zeichen aus der Eingabe gelesen wurden, um die entsprechende Komponente zu dekodieren.

Die Funktionen müssen zu den entsprechenden Kodierungsfunktionen *passend* sein, d. h. es muss gelten:

- Für alle $x \in \Sigma$: $x = y$ mit $(y,l) = \text{decodeRaw}(\text{encodeRaw}(x))$ und
- für alle $k \in \mathbb{N}$: $k = y$ mit $(y,l) = \text{decodeSource}(\text{encodeSource}(k))$ und
  $k = z$ mit $(z,m) = \text{decodeLength}(\text{encodeLength}(k))$.

Ferner müssen dem Dekodierer die Ankündigungssymbole ρ und $\bar{\rho}$ bekannt sein.

Algorithmus 3.12 dekodiert einen kodierten Text t' mithilfe dieser Informationen. In den folgenden Unterabschnitten werden die einzelnen Schritte des Algorithmus genauer beschrieben.

[9] k-gramme sind Phrasen der Länge k, Trigramme sind 3-gramme.

Eingabe: Kodierter Text $t' \in (\Gamma \cup \{\rho,\bar{\rho}\})^*$
Ausgabe: Ursprungstext $t \in \Sigma^*$.

1. Sei $(l,a) = \text{decodeLength}(t')$.
   - Initialisiere die Länge des Ursprungstextes $|t| := l$ und den Lese-Cursor $p := a + 1$.
   - Initialisiere den dekodierten Text $t := \bot^{|t|}$ und den Schreib-Cursor $k := 1$.
   - Initialisiere die leere *Callback-Abbildung* $C : \mathbb{N} \to \mathcal{P}(\mathbb{N})$.
2. Ist $p > |t'|$, brich ab.
3. Ist $t'[p] = \bar{\rho}$, erhöhe p um $|\bar{\rho}|$ und springe zu Punkt 5 (Dekodierung einer Regel), sonst erhöhe p um $|\rho|$ und springe zu Punkt 4 (Dekodierung eines Rohsymbols).
4. Sei $(x,l) = \text{decodeRaw}(t'[p\ldots])$.
   - $t[k] := x$.
   - Führe *callback*(t,C,k) aus (siehe unten).
   - $k := k + 1$, $p := p + l$.
   - Springe zu Punkt 2.
5. Sei $(q,l) = \text{decodeSource}(t'[p\ldots])$ und $(n,m) = \text{decodeLength}(t'[p + l\ldots])$.
   - $p := p + l + m$, $i := 0$.
   - Solange $i < n$:
     a) Falls $t[q + i] \neq \bot$:
        – $t[k + i] := t[q + i]$.
        – Führe *callback*$(t,C,k + i)$ aus (siehe unten).
     b) Falls $t[q + i] = \bot$:
        – $C := C \cup \{q + i \mapsto k + i\}$.
     c) $i := i + 1$.
   - $k := k + n$.
   - Springe zu Punkt 2.

Funktion *callback*(t,C,x):

1. Für alle $c_x \in C(x)$:
   a) $t[c_x] := t[x]$.
   b) Führe callback(t,C,c_x) rekursiv aus.

Algorithmus 3.12: Dekodierung eines kodierten Textes.

3.6.1 Initialisierung

Durch den Initialisierungsschritt wird zunächst die Länge des Originaltextes gelesen und der dekodierte Text t mit $\perp^{|t|}$ initialisiert. Dabei sei $\perp$ ein nicht weiter spezifiziertes Platzhaltersymbol.

Der *Lese-Cursor* p zeigt auf die aktuelle Leseposition im kodierten Text, der *Schreib-Cursor* k auf die nächste zu schreibende Position im dekodierten Text.

Ferner wird die (zunächst leere) *Callback-Abbildung* $C : \mathbb{N} \to \mathcal{P}(\mathbb{N})$ initialisiert. Wird eine Ersetzungsregel dekodiert, kann es sein, dass die Symbole an der Quellposition noch nicht dekodiert wurden, so dass die Rückersetzung noch nicht durchgeführt werden kann. In diesem Fall muss sich der Dekodierer „merken", dass an diese Positionen die entsprechenden Symbole geschrieben werden müssen, sobald sie an der Quellposition aufgelöst werden. Für eine Textposition $i \in \mathbb{N}$ enthält $C(i)$ also jene Textpositionen, die darauf „warten", dass das Symbol an Stelle i dekodiert wird. Bei der Dekodierung von i wird das entsprechende Symbol dann auch an alle Stellen in $C(i)$ geschrieben (siehe Unterabschnitt 3.6.6).

3.6.2 Abbruchbedingung

Die Dekodierung wird solange fortgeführt, bis der Lese-Cursor p die Länge des kodierten Textes t' überschreitet.

3.6.3 Unterscheidung zwischen Rohsymbolen und Ersetzungsregeln

An der aktuellen Leseposition p wird ein Ankündigungssymbol erwartet. Ist dieses $\bar{\rho}$, so folgt eine kodierte Ersetzungsregel, die durch Punkt 5 aufgelöst wird. Ansonsten wird ein kodiertes Rohsymbol erwartet, welches von Punkt 4 dekodiert wird.

Der Lese-Cursor wird nach Lesen des Ankündigungssymbols entsprechend erhöht.

3.6.4 Dekodierung eines Rohsymbols

Mit der Funktion decodeRaw wird ein kodiertes Rohsymbol dekodiert und im dekodierten Text an die aktuelle Schreibposition k geschrieben.

Es wird unmittelbar die *Callback-Funktion* (Unterabschnitt 3.6.6) für k aufgerufen, die „wartende" Textpositionen auflöst.

Schreib- sowie Lese-Cursor werden nach der Operation entsprechend weitergeschoben.

3.6.5 Dekodierung einer Ersetzungsregel

Durch den Kodierer durchgeführte Ersetzungen müssen durch den Dekodierer rückgängig gemacht werden, indem die Symbole von den Quellpositionen ausgelesen werden. Die Zielposition ist die aktuelle Schreibposition k.

Hierzu wird zunächst die Quellposition q und die Länge n der Ersetzung dekodiert und der Lese-Cursor entsprechend erhöht. Für alle n Symbole wird nun versucht, Symbole ab der Position q im dekodierten Text an die Stellen ab k im dekodierten Text zu kopieren. Hierzu wird die Laufvariable i verwendet.

Wurde das Quellsymbol an Stelle $q+i$ bereits dekodiert ($t[q+i] \neq \perp$), wird es an die Stelle $k+i$ kopiert und die *Callback-Funktion* (Unterabschnitt 3.6.6) dafür aufgerufen. Wurde das Quellsymbol noch nicht dekodiert ($t[q+i] = \perp$), wird die zu schreibende Position $k+i$ wird zur „Warteliste" $C(q+i)$ hinzugefügt.

Schreib- sowie Lese-Cursor werden nach der Operation entsprechend weitergeschoben.

3.6.6 Callback-Funktion

Die *Callback-Funktion* wird für eine Stelle x aufgerufen, wenn diese dekodiert wurde. Das entsprechende Symbol wird hierdurch auch an alle „wartenden" Positionen geschrieben, welche in der Menge $C(x)$ enthalten sind.

Da auf jede dieser wartenden Positionen wieder weitere Positionen warten können, muss die Funktion rekursiv arbeiten und sich nach jeder Schreiboperation für die jeweilige Stelle selbst erneut aufrufen.

4 Implementierung

In diesem Kapitel wird die Implementierung der in Kapitel 3 entwickelten Algorithmen dokumentiert.

Ziel der Implementierung im Rahmen dieser Arbeit ist es, ein lauffähiges Programm zur Kompression von Texten mit dem vorgestellten Verfahren zu entwickeln. Die Effizienz der Implementierung bezüglich Laufzeit und Speicheraufwand ist dabei zweitrangig.

Für einen Eingabetext, welcher direkt eingegeben wird oder in einer Datei vorliegt, sollen Ersetzungsregeln mit dem Verfahren *ESAComp* (siehe Abschnitt 3.3) ermittelt und anschließend eine der Kodierungen *code1* bzw. *code2* (siehe Abschnitt 3.5) angewendet werden. Der kodierte Text wird schließlich ausgegeben.

Zwecks der später folgenden praktischen Tests (siehe Kapitel 5) soll das Programm überdies einige Statistiken über den Verlauf und das Ergebnis der Kompression ausgeben. Damit ein Vergleich mit dem Verfahren LZ77 möglich wird, soll alternativ zum Verfahren *ESAComp* die LZ-Faktorisierung (siehe Abschnitt 2.6) zur Ermittlung der Ersetzungsregeln ausgewählt werden können.

Die Implementierung erfolgt in der Programmiersprache *Java 7*.

In Anhang B werden die Ausführung und die Ausgaben des Programms beschrieben. Der gesamte Quelltext sowie das ausführbare Programm befindet sich auf dem beliegenden Medium.

4.1 jSuffixArrays

Zur Ermittlung des Enhanced Suffix Arrays zu einem Text wird die freie Bibliothek *jSuffixArrays* [28] in der Version *0.10* eingesetzt. Diese implementiert unter anderem das Verfahren *BPR* zur Konstruktion des Suffix Arrays nach [29] und *GetHeight* zur Ermittlung der LCP-Tabelle aus dem Suffix Array nach [18].

4.2 Java-Klassen

Im Folgenden werden die Aufgaben der implementierten Java-Klassen kurz erläutert. Abbildung 4.1 zeigt das zugehörige UML-Klassendiagramm, worin die relevanten Felder und Operationen aufgeführt sind (nicht jedoch Hilfsvariablen oder Hilfsfunktionen).

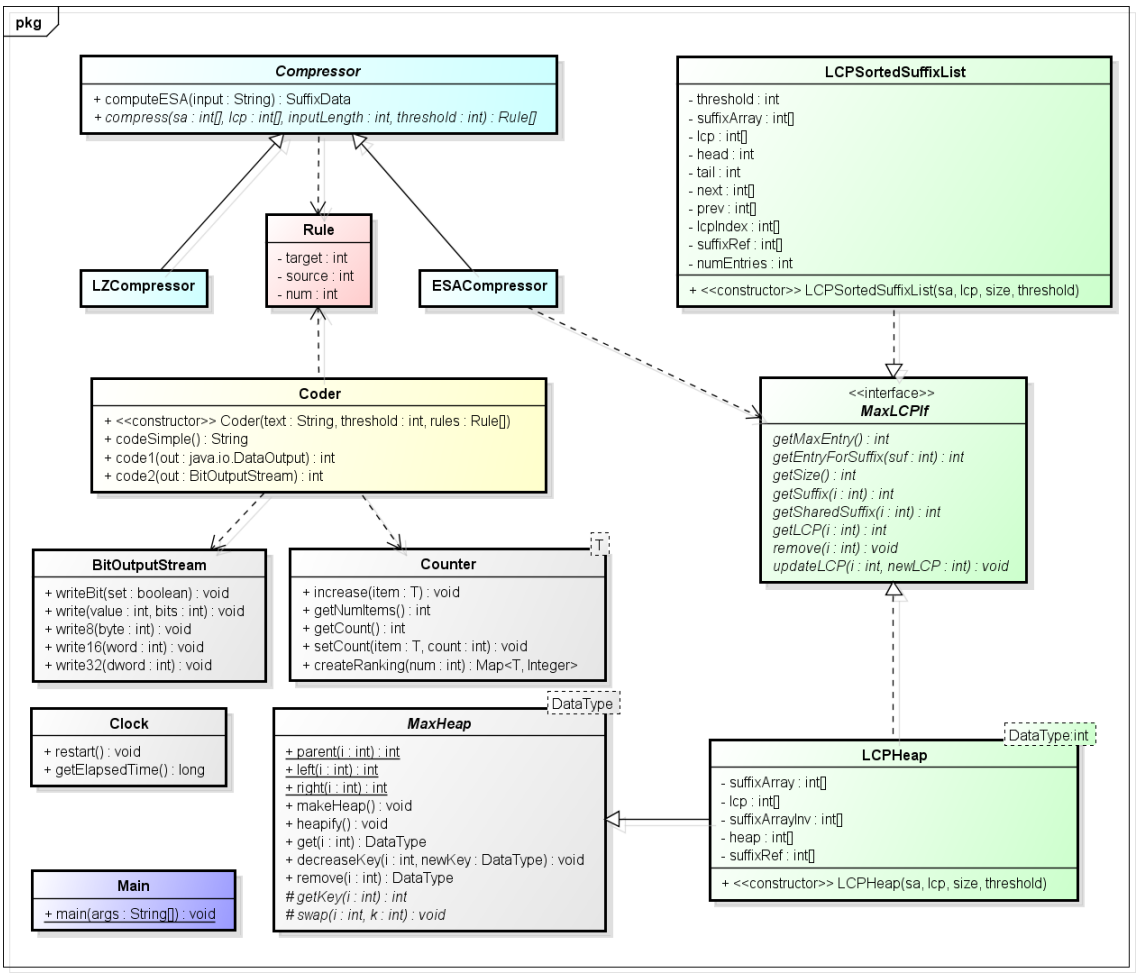


Abbildung 4.1: UML-Klassendiagramm für die Implementierung.

4.2.1 Main

Die Klasse *Main* enthält die *main*-Methode und dient als Einstiegspunkt für das Programm. Sie steuert den Ablauf des Programms wie folgt:

1. Interpretation der Kommandozeilenparameter
2. Falls erforderlich: Einlesen des Eingabetextes aus einer Datei
3. Ermittlung von Ersetzungsregeln für den Eingabetext mit dem gewählten Kompressor (*ESAComp* oder *LZ-Faktorisierung*)
4. Ausgabe statistischer Messgrößen zur Menge der Ersetzungsregeln (weitere Details siehe Kapitel 5)
5. Kodierung des Eingabetextes mit den gewählten Kodierern und Ausgabe der Kompressionsraten.

4.2.2 Rule

Gemäß Definition 2.16 repräsentiert eine Instanz der Klasse *Rule* eine Ersetzungsregel. Sie enthält Zielposition (*target*), Quellposition (*source*) und Länge (*num*) einer Ersetzung.

4.2.3 Compressor

Die Klasse *Compressor* dient als abstrakte Basis für die in Unterabschnitt 4.2.4 beschriebenen Implementierungen der Kompressionsverfahren.

Sie bietet einerseits eine vereinfachte Schnittstelle zur Ermittlung des Enhanced Suffix Arrays (*computeESA*, welche intern die Bibliothek *jSuffixArrays* verwendet). Die Rückgabe erfolgt über ein *SuffixData*-Objekt, welches Suffix-Array und LCP-Tabelle für den Eingabetext enthält.

Des Weiteren wird die abstrakte Methode *compress* definiert. Diese ermittelt mithilfe eines Enhanced Suffix Arrays (*sa* und *lcp*) für den gegebenen Schwellwert (*threshold*) eine Menge von Ersetzungsregeln. Die Implementierung des Kompressors muss in einer abgeleiteten Klasse erfolgen.

4.2.4 ESACompressor und LZCompressor

Gemäß den Anforderungen werden zwei Kompressoren implementiert, welche jeweils von der Klasse *Compressor* erben und die darin definierte abstrakte Methode *compress* implementieren.

Die Klasse *ESACompressor* dient zur Kompression mit dem Verfahren *ESAComp* (Algorithmus 3.1) unter Verwendung einer LCP-sortierten Suffix-Liste (siehe Unterabschnitt 4.2.6).

Die Lempel-Ziv-Faktorisierung (Algorithmus 2.5) wird durch die Klasse *LZCompressor* realisiert. In der Implementierung ist auch die Berechnung des inversen Suffix-Arrays (Definition 2.15) des Eingabetextes enthalten.

4.2.5 MaxLCPIf

Der Kompressor *ESAComp* benötigt eine Datenstruktur, welche es ermöglicht, auf das Suffix mit dem (momentan) längsten LCP zuzugreifen, dieses aus der Struktur zu entfernen und ggf. den LCP eines Suffix zu ändern. In Kapitel 3 wurden zwei derartige Strukturen eingeführt: der *LCP-Heap* und die *LCP-sortierte Suffix-Liste*.

Für die Implementierung erscheint es sinnvoll, die Schnittstelle *MaxLCPIf* zu definieren, die die von *ESAComp* benötigten Operationen spezifiziert und von den entsprechenden Strukturen implementiert wird.

Für Einträge in die Datenstruktur werden ganzzahlige Nummern vergeben. Mit *getMaxEntry* kann die Nummer des Eintrags geholt werden, die auf den Suffix mit dem derzeit längsten LCP zeigt. *getEntryForSuffix* gibt für den i-ten Suffix der Eingabe die Num-

mer des zugehörigen Eintrags in der Datenstruktur zurück[10]. Die Operationen *getSuffix*, *getSharedSuffix*[11] und *getLCP* ermöglichen den Zugriff auf das Enhanced Suffix Array über Eintragsnummern. Mit *remove* und *updateLCP* können Einträge aus der Struktur entfernt werden bzw. kann der LCP eines Eintrags aktualisiert werden.

4.2.6 LCPSortedSuffixList

Die in Abschnitt 3.1 spezifizierte LCP-sortierte Suffix-Liste wird von der Klasse *LCPSortedSuffixList* implementiert. Die Elemente der verketteten Liste und des LCP-Index sind Zeilennummern im Suffix-Array.

Die Felder *head* und *tail* zeigen auf den ersten bzw. letzten Suffix der verketteten Liste, die Arrays *next* und *prev* realisieren jeweils die Verkettung zwischen Vorgänger- bzw. Nachfolgeelement. Der LCP-Index wird als Array der Länge max LCP realisiert, da sein Definitionsbereich dem Wertebereich der LCP-Tabelle entspricht.

Durch den Konstruktor wird die Liste intial aufgebaut. Zur weiteren Verwendung implementiert die Klasse die Operationen der Schnittstelle *MaxLCPIf* (siehe Unterabschnitt 4.2.5).

4.2.7 MaxHeap

Die abstrakte Klasse *MaxHeap* dient als Rahmenimplementierung für Max-Heaps, in denen an den Knoten noch weitere Daten des generischen Typs *DataType* hinterlegt werden können.

Die Operationen *makeHeap* (initiales Herstellen der Heap-Bedingung), *heapify* (Herstellung der Heap-Bedingung für einen bestimmten Knoten), *decreaseKey* (Veränderung eines Knotengewichtes) und *remove* (Entfernen eines Knotens) werden in dieser Klasse allgemein implementiert, ohne an einen bestimmten Datentyp gebunden zu sein oder die Art der Datenhaltung vorzugeben. Dies wird dadurch erreicht, dass die Methoden *getKey* (Rückgabe des Wertes eines Knotens), *get* (Rückgabe der hinterlegten Daten für einen Knoten) und *swap* (Vertauschen zweier Knoten) als *abstrakt* deklariert sind, also durch erbende Klassen implementiert werden müssen.

Lediglich die Indexierung der Knoten wird wie in Abschnitt 2.3 verlangt: Die statischen Methoden *parent*, *left* und *right* bestimmen für den Index eines Knotens jeweils den Index des Vaterknotens, linken Kindes bzw. rechten Kindes. Für den Index 0 wird das Wurzelelement erwartet.

[10] Diese Operation ist für die Eliminierung von Überschneidungen notwendig und wird intern mit einer Suffix-Referenztabelle realisiert.

[11] Als „shared suffix" wird hier der Vorgänger eines Suffix im Suffix-Array bezeichnet, mit welchem er den LCP teilt.

4.2.8 LCPHeap

Der in Abschnitt 3.2 eingeführte *LCP-Heap* wird durch die Klasse *LCPHeap* realisiert. Sie erbt von der Klasse *MaxHeap* und implementiert die entsprechenden Operationen.

Durch den Konstruktor wird der Heap intial aufgebaut. Zur weiteren Verwendung implementiert die Klasse die Operationen der Schnittstelle *MaxLCPIf* (siehe Unterabschnitt 4.2.5).

4.2.9 Coder

Die Klasse *Coder* stellt Implementierungen der in Unterabschnitt 3.5.1 und Unterabschnitt 3.5.2 spezifizierten Kodierer *code1* und *code2* bereit. Eine Instanz erhält hierzu die Eingabewerte und die ermittelte Menge von Ersetzungsregeln (repräsentiert durch die Klasse *Rule*).

Die zusätzliche Kodierungsfunkton *codeSimple* erzeugt eine einfache, für den Menschen lesbare Kodierung wie in Abbildung 3.8.

code1 schreibt direkt in einen Java-Ausgabestrom, während *code2* bitweise arbeitet und eine spezielle Lösung erfordert, welche in Unterabschnitt 4.2.10 dokumentiert wird. Die Methode *codeSimple* erzeugt eine für den Menschen lesbare Kodierung, welche jener aus Abbildung 3.8 ähnelt.

Im Konstruktor dieser Klasse werden auch einige statistische Daten über die Menge der Ersetzungsregeln ermittelt, da diese teilweise für *code2* verwendet werden. Details hierzu sind Abschnitt 4.3 zu entnehmen.

4.2.10 BitOutputStream

Die von Java bereitgestellte Schnittstelle für Ausgabeströme arbeitet byteweise, d. h. sie stellt Methoden zur Ausgabe einzelner Bytes bereit.

Der in Unterabschnitt 3.5.2 definierte Kodierer *code2* arbeitet jedoch bitweise, d. h. es müssen einzelne Bits ausgegeben werden können. Hierzu wird die Klasse *BitOutputStream* eingeführt, welche bitweise Schreiboperationen bereitstellt und die geschriebenen Bits schließlich über einen Java-Ausgabestrom ausgibt.

BitOutputStream arbeitet auf einem *Puffer-Byte* b (mit dem initialen Wert 0) und dem *Bitzeiger* $p \in [0,7]$, welcher das nächste zu schreibende Bit innerhalb von b markiert (mit dem initialen Wert 0, welcher das höchstwertigste Bit markiert).

Die Schreiboperation *writeBit* nimmt ein Bit $x \in [0,1]$ entgegen und arbeitet gemäß Algorithmus 4.2.

In Punkt 1 des Algorithmus wird, falls erforderlich, das derzeit markierte Bit mithilfe des bitweisen `OR`-Operators gesetzt. Durch Punkt 2 wird der Zeiger erhöht. Ist das Puffer-

Eingabe: Bit $x \in [0,1]$.

1. Falls $x = 1$: $b := b$ `OR` 2^p.
2. $p := p + 1$.
3. Falls $p \geq 8$:
   a) Schreibe b in Java-Ausgabestrom.
   b) $b := 0$, $p := 0$.

Algorithmus 4.2: Bitweise Schreiboperation in Java über ein Puffer-Byte.

Byte b ausgefüllt, wird es durch Punkt 3 in den Java-Ausgabestrom geschrieben und alle Parameter zurückgesetzt.

Der Sonderfall, dass der Ausgabestrom geschlossen wird, bevor das Puffer-Byte ausgefüllt ist, muss ebenfalls beachtet werden. In diesem Fall muss es, sofern es teilweise befüllt wurde, vor der Schließung des Stroms noch darin ausgegeben werden.

Die Funktionen *write8*, *write16* und *write32* sind Hilfsfunktionen, welche die einzelnen Bits von ganzzahligen 8-, 16-, bzw. 32-Bit-Werten über *writeBit* ausgibt.

4.2.11 Counter

Der Kodierer *code2* verwendet zur effizienten Kodierung von Rohsymbolen bzw. Trigrammen dessen Häufigkeit im Text (vgl. Abschnitt 3.5.2). Die Klasse *Counter* bietet eine Hilfe zum Zählen beliebiger Elemente des generischen Typs T und zur anschließenden Erzeugung einer Rangordnung dieser Elemente.

Durch die Methode *increase* wird der Zähler für ein Element um 1 erhöht. *getNumItems* liefert die Anzahl der Ausprägungen (d.h. die Anzahl der Elemente, die mindestens einmal gezählt wurden).

Die Methode *createRanking* erzeugt eine Rangzuordnung $rank : T \to \mathbb{N}_0$ für die (höchstens) *num* häufigsten Elemente. Das häufigste Element erhält hierbei den niedrigsten Rangwert 0.

4.2.12 Clock

Die Klasse *Clock* dient zur Messung von Zeitintervallen.

Wird eine Instanz dieser Klasse erzeugt, so merkt sie sich den aktuellen Zeitstempel t_0 des Systems. Die Methode *getElapsedTime* gibt die Anzahl an Millisekunden zurück, die seit t_0 vergangen sind. Die Methode *reset* überschreibt t_0 wieder mit dem aktuellen Zeitstempel.

4.3 Ermittlung statistischer Daten

Für die in Kapitel 5 spezifizierten Tests und für den Kodierer *code2* müssen einige statistische Größen ermittelt werden, nachdem der Kompressor eine Menge von Ersetzungsregeln $R \subseteq \mathbb{N}^3$ ermittelt hat.

In diesem Abschnitt werden die nicht-trivialen Fälle beschrieben.

4.3.1 Ersetzungstiefe der Symbole

Die Ersetzungstiefe d_i des i-ten Symbols im Eingabetext erfordert die jeweilige Quellposition q_i (vgl. Definition 2.22) und wird daher in zwei Arbeitsschritten ermittelt.

Zunächst wird für jede Regel $(z,q,n) \in R$ die Quellpositionen $q_i = q + (i - z)$ für jedes ersetzte Symbol mit $z \leq i < z + n$ berechnet und die Tiefe zu $d_i = \perp = -1$ initialisiert. Alle nicht ersetzten Symbole erhalten gemäß Definition die Ersetzungstiefe 0.

Sind alle q_i bekannt, werden alle d_i nach der Definition rekursiv berechnet. Die Ergebnisse der Rekursionsschritte werden dabei direkt gespeichert. So können die Ersetzungstiefen aller Symbole in insgesamt linearer Zeit bezüglich der Länge des Eingabetextes ermittelt werden.

4.3.2 Rangordnung von Alphabet und Trigrammen

code2 arbeitet mithilfe einer Rangordnung, die die Häufigkeit von Einzelsymbolen bzw. Trigrammen aus dem Eingabealphabet wiederspiegelt (vgl. Unterabschnitt 3.5.2).

Hierfür muss der Text einmal durchlaufen werden. Mithilfe der Klasse *Counter* (siehe Unterabschnitt 4.2.11) werden dabei alle nicht ersetzten Einzelsymbole und alle Trigramme gezählt, welche kein ersetztes Symbol enthalten.

Nach dem Durchlauf ermittelt *Counter* die zugehörige Rangordnung.

5 Praktische Tests

Das in Kapitel 4 entwickelte Programm wird für die praktischen Tests verwendet, die in diesem Kapitel dokumentiert werden.

Die Tests haben neben der reinen Überprüfung auf Funktionalität ferner das Ziel, statistische Größen über die durch das Verfahren erzielte Kompression zu ermitteln, um dessen Brauchbarkeit für verschiedene Eingabetexte zu überprüfen und einen Vergleich mit anderen Verfahren zu ermöglichen.

Vor dem *Haupttest*, in welchem das Verfahren *ESAComp* an verschiedenartigen Texten erprobt und mit gängigen Verfahren verglichen wird, wird ein vorbereitender Test durchgeführt. Anhand dessen soll entschieden werden, welche Datenstruktur zum Zugriff auf den maximalen LCP im Haupttest verwendet wird.

5.1 LCP-sortierte Suffix-Liste versus LCP-Heap

In dem vorgeschalteten Test werden die beiden in Kapitel 3 vorgestellten Datenstrukturen miteinander verglichen, die stets Zugriff auf den Suffix mit dem derzeit längsten LCP ermöglichen: die *LCP-sortierte Suffix-Liste* (fortan kurz „Liste"; siehe Abschnitt 3.1) und der *LCP-Heap* (fortan kurz „Heap"; siehe Abschnitt 3.2).

5.1.1 Testumgebung

Für diesen Test wird ein Text der Länge $n = 4 \cdot 2^{20}$ (4 MiB) herangezogen.

5.1.2 Durchführung

Der Text wird mit dem Verfahren *ESAComp* komprimiert und anschließend mit *code2* kodiert. Für *ESAComp* wird dabei abwechselnd die Liste bzw. der Heap als Datenstruktur verwendet. Für beide Datenstrukturen wird der Test zehnmal durchgeführt.

Bei jedem Durchlauf werden dabei die Teillaufzeiten der folgenden Schritte gemessen:

1. Berechnung des Enhanced Suffix Arrays (mit *DivSufSort* [22])
2. Initialer Aufbau der Datenstruktur (Liste bzw. Heap)
3. Kompression mit *ESAComp* (mit Liste bzw. Heap)
4. Kodierung mit *code2*.

Für die ersten bzw. letzten Teillaufzeiten wird erwartet, dass sie unabhängig von der Datenstruktur ähnlich ausfallen. Lediglich bei den Laufzeiten der Kodierung kann es sehr kleine Abweichungen geben, da die erzeugte Menge von Ersetzungsregeln genau genommen von der Datenstruktur abhängig ist und daher leicht variieren kann[12].

5.1.3 Ergebnisse

Die Aufstellung der Testergebnisse ist Abschnitt A.1 (Anhang A) zu entnehmen.

5.1.4 Auswertung

Die Vergleichswerte in Tabelle A.1 (Anhang A) zeigen, dass sowohl die Initialisierung als auch die Kompression unter Verwendung der LCP-sortierten Suffix-Liste im Durchschnitt um den Faktor 2,3 schneller abläuft als mit dem LCP-Heap.

Aus diesem Grunde wird im Haupttest ausschließlich die LCP-sortierte Suffix-Liste verwendet.

5.2 Haupttest

Im folgenden Haupttest soll das in Kapitel 3 entwickelte Verfahren *ESAComp* an verschiedenartigen Texten erprobt und hinsichtlich der erzielten Kompression mit der *LZ-Faktorisierung* (siehe Abschnitt 2.6) und alltagsüblichen Kompressoren verglichen werden.

5.2.1 Testumgebung

Als Grundlage für die folgenden Tests wird die Textkollektion aus [10] herangezogen. Diese enthält verschiedene Arten von Texten[13]. Tabelle 5.1 zeigt die Textproben, welche für die praktischen Tests eingesetzt werden. Hierzu werden jeweils auch die Alphabetgröße[14] sowie eine kurze Beschreibung der Textart aufgeführt. Genauere Informationen zu den Textproben sind [10] zu entnehmen. Zwecks Vergleichbarkeit werden all diese Texte auf die gleiche Länge $n = 50 \cdot 2^{20}$ (50 MiB) gekürzt.

Die Textproben dienen als Eingabe für das nach Kapitel 4 entwickelte Testprogramm.

[12] Zwar enthalten sowohl Liste als auch Heap initial die selben Einträge, die Reihenfolge der Einträge mit gleich langen LCP-Längen ist jedoch ggf. unterschiedlich. Dies kann bei langen Texten zu leicht unterschiedlichen Mengen von Ersetzungsregeln führen.

[13] d. h. auf Grundlage verschiedener Alphabete und Sprachen.

[14] Das Alphabet umfasst jeweils auch Sonderzeichen wie Leerzeichen, Zeilenumbrüche usw.

Text	Alphabetgröße	Kurzbeschreibung
Englisch	239	Englische Textkollektion aus [2].
Quellcode	230	C-/Java-Quelltext und -Header.
XML	97	Strukturierter Text im XML-Format.
Protein-Sequenzen	27	Protein-Sequenzen aus [25].
DNA-Sequenzen	16	DNA-Sequenzen aus [2, Human Genome Project] (s.a. [10]).

Tabelle 5.1: Textproben der Länge $n = 50 \cdot 2^{20}$ (50 MiB) für praktische Tests.

5.2.2 Durchführung

Für jeden in Tabelle 5.1 aufgeführten Text t und für jeden Schwellwert $\theta \in [2{,}16]$ wird folgender Komprimiererlauf ausgeführt:

1. Berechne SA_t und LCP_t.
2. Ermittle die Regelmenge R_{LCP} nach Algorithmus 3.1 mit dem Schwellwert θ.
3. Ermittle die Phrasenmenge R_{LZ} nach [8] (LZ-Faktorisierung), wobei nur Phrasen aufgenommen werden, deren Länge mindestens θ ist.
4. Wende *code*1 jeweils mit R_{LCP} und R_{LZ} auf t an.
5. Wende *code*2 jeweils mit R_{LCP} und R_{LZ} auf t an.

Es werden dabei für jeden Durchlauf folgende Kennzahlen gemessen:

- Die Kompressorlaufzeit in Sekunden,
- die Anzahl der erzeugten Regeln bzw. Phrasen,
- die Anzahl der insgesamt ersetzten Symbole (Summe der Längen aller Ersetzungen),
- die durchschnittliche Ersetzungstiefe je Symbol.
- die Länge des durch *code1* kodierten Strings (mit *ESAComp* und *LZ*),
- die Länge des durch *code2* kodierten Strings (mit *ESAComp* und *LZ*) und
- die Initiale Anzahl von Elementen in der LCP-sortierten Suffix-Liste.

Die ersten sechs Kennzahlen werden jeweils für beide Kompressoren ermittelt. Aus ihnen lassen sich Vergleichswerte wie die durschnittliche Anzahl ersetzter Symbole je Regel oder die Kompressionsrate ermitteln.

Die *initiale Anzahl von Elementen in der LCP-sortierten Suffix-Liste* dient der Information über die Menge von Einträgen im Suffix-Array, die *ESAComp* verarbeiten muss. Hiervon hängt maßgeblich die Laufzeit des Kompressors ab. Für die Sprache des zugrundeliegenden Textes dient diese Anzahl außerdem als Indikator für die Wahrscheinlichkeit dafür, dass eine Phrase der Länge $k \geq \theta$ im Text mehrfach vorkommt.

Die *durchschnittliche Ersetzungstiefe je Symbol* $\varnothing_d$ sagt aus, wie viele ersetzte Symbole ein Dekodierer für die Dekodierung eines beliebigen Symbols im Schnitt auflösen muss, um im kodierten Text auf ein Rohsymbol zu stoßen. Sei d_i die Ersetzungstiefe des i-ten Symbols

(vgl. Definition 2.22), dann ist $\varnothing_d = \frac{1}{n} \sum_{i=1}^{n} d_i$. Diese Kennzahl dient zur Bewertung des Aufwands für *Random Access*[15] im kodierten Text.

Für die *Anzahl der erzeugten Regeln* ist zu beachten, dass diese auch genau die Anzahl der Durchläufe des Verfahrens *ESAComp* wiederspiegelt.

5.2.3 Ergebnisse

Die Aufstellung der Testergebnisse ist Abschnitt A.2 (Anhang A) zu entnehmen.

5.3 Auswertung des Haupttests und Vergleich

In diesem Abschnitt werden zweierlei Vergleiche hinsichtlich der Testergebnisse vorgenommen. Es liegen dabei folgende Fragestellungen zugrunde:

1. Wie „gut" komprimiert *ESAComp* Texte einer bestimmten Art?
2. Wie „gut" komprimiert *ESAComp* im Vergleich zu *LZ*?

Mit *LZ* wird im Folgenden die Kompression durch die in Abschnitt 2.6 beschriebene LZ-Faktorisierung bezeichnet.

Hinsichtlich der „Güte" eines Kompressors wird hier lediglich die Kompressionsrate betrachtet. Auf eine Betrachtung der Laufzeiten wird hier verzichtet, da bei der Implementierung im Rahmen dieser Arbeit kein Fokus darauf gelegt wird.

5.3.1 Vergleich zwischen Textarten bezüglich ESAComp

Wir betrachten zunächst, wie *ESAComp* die verschiedenen Arten von Texten verarbeitet. Hierzu werden folgende Fragen gestellt:

1. Welche Art von Texten kann *ESAComp* am besten komprimieren?
2. Gibt es einen optimalen Schwellwert θ, für welchen stets die beste Kompressionsrate erzielt wird?

[15] *Random Access* meint die Dekodierung eines bestimmten Ausschnittes des kodierten Textes.

Rangfolge

Im Bezug auf die besten Kompressionsraten unterliegen die Texte unabhängig von der Kodierungsmethode folgender Rangfolge:

1. XML
2. Quellcode
3. Englisch
4. DNA
5. Proteine.

Für unnatürliche, strukturierte Sprachen (XML, Quellcode) ist *ESAComp* demnach besser geeignet als für menschliche Sprachen (Englisch) oder natürliche Sprachen (DNA, Proteine). Dies entspricht genau der Rangfolge für andere Kompressoren [10].

Optimaler Schwellwert

Wir betrachten im Folgenden die Kompressionsrate im Bezug auf den Kodierer *code1*, welcher im Gegensatz zu *code2* direkt kodiert, ohne weitere spezielle Kodierungsschritte durchzuführen.

Die besten Kompressionsraten werden für alle Texte für die Schwellwerte 7 bis 9 erzielt. Zur Ergründung dieser Beobachtung berechnen wir für jeden Text t die Länge l_R (in Bytes) einer kodierten Ersetzungsregel für den jeweiligen Text gemäß *code1*:

$$l_R = |\bar{\rho}| + \lceil \log_{256}(|t|) \rceil + \lceil \log_{256}(n_{R,\max}) \rceil \,.$$

Wir nehmen $|\bar{\rho}| = 1$ und $|t| = 50 \cdot 2^{20}$ an. $n_{R,\max}$ (die Länge der längsten Ersetzung) wurde bei den Tests für jeden Text gemessen. Θ_{best} bezeichnet die Schwellwerte, für welche die besten Kompressionsraten bezüglich *code1* erzielt wurden[16].

Text	$n_{R,\max}$	l_R	Θ_{best}	Kompressionsrate
Englisch	109.394	8	{8,9}	45,05%
Quellcode	71.651	8	{8,9}	39,83%
XML	1.005	7	{7,8}	21,80%
Proteine	25.822	7	{7,8}	68,67%
DNA	14.836	7	{7,8}	50,55%

Tabelle 5.2: Vergleich zwischen kodierter Länge von Ersetzungsregeln und optimalem Schwellwert.

[16] Es wurde überraschend festgestellt, dass es für alle Texte zwei Schwellwerte gibt, für die exakt gleich lange und zugleich beste Kodierungen mit *code1* erzielt wurden. Dieser Umstand wurde im Rahmen der Arbeit nicht weiter verfolgt.

Die jeweiligen Werte von l_R sind Tabelle 5.2 zu entnehmen. Es ist festzustellen, dass in allen Fällen $l_R = \min\{\Theta_{\text{best}}\}$ gilt. Der optimale Schwellwert für *ESAComp* ist demnach jener, für welchen alle Ersetzungen gerade so lang sind wie eine kodierte Regel.

Demnach bietet $\theta = l_R$ eine sehr gute Approximation des optimalen Schwellwertes für *code1*. Die Länge der längsten Ersetzung ist das Maximum der LCP-Tabelle und kann entsprechend einfach hieraus abgelesen werden.

Für den Kodierer *code2* ist die Berechnung von l_R deutlich komplexer, da die Länge einer kodierten Ersetzungsregel von der Länge der entsprechenden Ersetzung sowie vom Schwellwert selbst abhängig ist, den es ja zu bestimmen gilt (vgl. Abschnitt 3.5.2). Eine solche Berechnung könnte also unter Verwendung der LCP-Tabelle approximativ erfolgen. Dies soll jedoch nicht Bestandteil dieser Arbeit sein.

5.3.2 Vergleich zwischen ESAComp und LZ

Zum Vergleich zwischen *ESAComp* und *LZ* betrachten wir folgende Kriterien:

1. Durch welche Kompressionsmethode werden mehr Symbole ersetzt?
2. Wie wirkt sich dies jeweils auf die Kompressionsrate aus?
3. Welche Kompression eignet sich im Nachhinein besser für Random Access?

Ersetzte Symbole

Den Wertetabellen in Unterabschnitt A.2.1 (Anhang A) ist zu entnehmen, dass der *LZ*-Kompressor in fast allen Fällen mehr Ersetzungsregeln erzeugt und hierdurch insgesamt mehr Rohsymbole ersetzt werden als durch *ESAComp*.

Jedoch ist für all diese Fälle die durschnittliche Anzahl der ersetzten Symbole je Regel bei *ESAComp* höher als bei *LZ* – die durch *ESAComp* erzeugte Menge von Regeln ist also konzentrierter, während die durch *LZ* ermittelten Ersetzungen sich weiter über den Text streuen.

Kompressionsrate

Wie Abbildung A.15 und Abbildung A.16 (Anhang A) zeigen, gibt es für beide Kodierer mindestens einen Schwellwert θ, für welchen *ESAComp* eine bessere Kompressionsrate liefert als *LZ* ($\frac{\text{ESA}}{\text{LZ}} < 1$).

Korrelation

Es scheint eine (nicht-proportionale) Korrelation zwischen der durschnittlichen Ersetzungslänge und der Kompressionsrate zu geben: Je näher sich die durschnittliche Ersetzungs-

länge für *ESAComp* der von *LZ* nach unten nähert, desto schlechter wird im Vergleich die Kompressionsrate.

Dies ist insbesondere in Kombination mit *code2* zu beobachten. Die höhere Konzentration der Ersetzungen bei *ESAComp* bewirkt vermutlich, dass längere, nicht ersetzte Rohpassagen im Text stehenbleiben. Für $\theta > 3$ bedeutet dies auch eine wachsende Wahrscheinlichkeit für redundante Trigramme, die durch *code2* speziell kodiert werden.

Random Access

In den Tabellen in Unterabschnitt A.2.1 (Anhang A) ist erkennbar, dass die Vergleichswerte für die durchschnittlichen Ersetzungstiefen je Symbol bei den verschiedenen Texten stark variieren. Wir legen den Fokus für die folgende Betrachtung auf die Schwellwerte θ im Bereich von 6-11, da hierfür die besten Kompressionsraten erzielt werden.

Für englische Texte ist die durschnittliche Tiefe für diesen Schwellwertbereich in der durch *ESAComp* erzeugten Regelmenge nicht wesentlich höher als in der *LZ*-Phrasenmenge (bis Faktor 1,5; im Schnitt jedoch niedriger), für Protein- und DNA-Sequenzen teilweise sogar niedriger (bis zu Faktor 0,83). Einzig bei XML scheinen die *ESAComp*-Regeln deutlich komplexer ineinanderzugreifen als die *LZ*-Regeln; die durschnittliche Tiefe ist hier bis um den Faktor 5 höher.

Im Regelfall wird der Aufwand für Random Access in durch *ESAComp* komprimierten Texten ähnlich ausfallen wie für Texte, die durch *LZ* komprimiert wurden. Lediglich bei XML ist der Aufwand hinsichtlich Random Access deutlich höher.

5.3.3 Vergleich zwischen ESAComp/code2 und anderen Verfahren

Wir betrachten nun die besten Kompressionsraten, die durch die Kombination des Verfahrens *ESAComp* in Kombination mit der Kodierung *code2* (im Folgenden *ESAComp/code2*) erzielt wurden und vergleichen diese mit den Kompressionsraten zweier gängigen, allgemein gebräuchlichen Kompressionsprogrammen: *gzip* und *bzip2* [1, 13].

Tabelle 5.3 stellt die Kompressionsraten dieser Programme mit höchster Kompressionsstufe nach [10] den gemessenen Kompressionsraten von *ESAComp/code2* gegenüber.

Text	`gzip -9`	`bzip -9`	*ESAComp/code2*
Englisch	37,52%	28,40%	24,85%
Quelltext	23,29%	19,79%	23,62%
XML	17,23%	11,22%	14,37%
Proteine	47,39%	45,56%	45,82%
DNA	27,05%	25,98%	27,67%

Tabelle 5.3: Aufstellung der besten Kompressionsraten je Text nach [10] und *ESAComp* in Kombination mit *code2*.

Es ist erkennbar, dass *ESAComp/code2* die beste Kompressionsrate für den Text in englischer Sprache liefert. Bei den Proteinsequenzen und XML schneidet es zumindest gegenüber *gzip* besser ab. Für Quelltexte und DNA-Sequenzen konnten keine besseren Ergebnisse erzielt werden als durch die alltagsüblichen Kompressoren.

6 Fazit

In dieser Arbeit wurde das Verfahren *ESAComp* zur Kompression von Texten mithilfe des Enhanced Suffix Arrays entwickelt, implementiert, getestet und mit anderen Verfahren und alltäglich gebräuchlichen Kompressoren verglichen.

6.1 Bewertung

Im Vergleich mit *gzip* und *bzip2* erzielt *ESAComp* in Kombination mit dem in dieser Arbeit entwickelten Kodierer *code2* sehr gute Kompressionsraten für Texte in englischer Sprache (vgl. Unterabschnitt 5.3.3) und gute Ergebnisse für XML und Proteinsequenzen. Für Quelltexte und DNA-Sequenzen schneidet *ESAComp*/*code2* etwas schlechter ab.

Gegenüber der LZ-Faktorisierung erzeugt *ESAComp* eine höher konzentrierte Menge von Ersetzungsregeln (Phrasen), d. h. im Durchschnitt ersetzt jede Regel mehr Symbole (vgl. Unterabschnitt 5.3.2). Hierdurch kann für alle Texte mit geschickter Wahl des Schwellwertes für Ersetzungslängen eine bessere Kompression erzielt werden als mit der LZ-Faktorisierung. Ein optimaler Schwellwert kann bestimmt werden, wenn entsprechende Details über den Kodierer bekannt sind.

Die durchschnittliche Ersetzungstiefe je Symbol ist in den durch *ESAComp* ermittelten Ersetzungsregeln vergleichbar mit der in der Menge von LZ77-Phrasen, so dass der Aufwand für Random Access ähnlich ausfällt. Lediglich für XML muss mit einem deutlichen Mehraufwand gerechnet werden.

Es muss bedacht werden, dass in dieser Arbeit in begrenzter Zeit ein neues Kompressionsverfahren entwickelt und mit etablierten, über lange Zeit weiterentwickelten Verfahren verglichen wurde. Es schneidet bezüglich der Kompressionsrate in keinem Fall deutlich schlechter ab, in bestimmten Fällen sogar besser. Unter Beachtung der in Abschnitt 6.2 diskutierten Punkte besteht demnach möglicherweise ein gutes Ausbaupotenzial.

6.2 Ausblick

Da der Bearbeitungsrahmen zeitlich begrenzt war, konnten einige Punkte nicht weiter untersucht werden, die eine bessere Bewertung ermöglichen oder zur Verbesserung der Laufzeit, des Speicheraufwands und der Kompressionsrate beitragen könnten. Diese Punkte

werden in diesem Abschnitt diskutiert und können als Ausgangspunkt für Folgearbeiten dienen.

6.2.1 Feldtest

In dieser Arbeit wurde für den Test lediglich ein bestimmer Text jeder Sprache herangezogen, in welchen teilweise recht lange redundante Sequenzen vorkommen. Um eine ausführlichere Bewertung zu ermöglichen, sollte *ESAComp* an einer breiteren Auswahl von Texten mit anderen Kompressoren verglichen werden.

6.2.2 Beschleunigung des Initialisierungsprozesses

Die derzeitige Implementierung bereitet die Ermittlung der Ersetzungsregeln schrittweise wie folgt vor:

1. Ermittlung des Enhanced Suffix Arrays
   a) Berechnung des Suffix-Arrays mit dem Verfahren *DivSufSort* [22] (Java-Implementierung [28])
   b) Ermittlung der LCP-Tabelle mit dem Algorithmus *GetHeight* [18] (vgl. [28])
2. Erzeugen der initialen LCP-sortierte Suffix-Liste.

Diese Schritte können ggf. in einen einzigen Schritt zusammengefasst werden, um die Initialisierungszeit zu verbessern. Es sind bereits Verfahren zur Berechnung des Suffix-Arrays bekannt, die die LCP-Tabelle als Nebenprodukt erzeugen [11, 17]. Denkbar wäre es demnach, auch den Aufbau der Suffix-Liste hier zu verankern.

6.2.3 Eliminierung von Debug- und Statistikfunktionen

Die derzeitige Implementierung erfolgte speziell für diese Arbeit und erhebt zwecks Messungen und Vergleichen viele Informationen, die in der Praxis zur eigentlichen Kompression nicht benötigt würden, jedoch sowohl Rechen- als auch Speicheraufwand beanspruchen.

Eine praxistaugliche Implementierung sollte Funktionen dieser Art nicht oder nur bedingt unterstützen.

6.2.4 Systemnähere Implementierung

Die Implementierung erfolgte im Rahmen dieser Arbeit in der Programmiersprache Java.

Java-Implementierungen sind zwar aufgrund der Optimierungen, die die JVM (*Java Virtual Machine*) im Rahmen der *Just-In-Time*-Kompilierung durchführt [27, Kapitel 2], im Allgemeinen nicht mehr deutlich langsamer als systemnahe Programme. Die JVM erledigt allerdings neben der Ausführung des eigentlichen Programms noch einige weitere Aufgaben, u.a.

- die automatische Initialisierung von Arrays [16, Abschnitt 10.6],
- die automatische Initialisierung von erzeugten Objekten [16, Abschnitt 8.3.2] und
- die Speicherverwaltung inklusive Garbage-Collection [27, Kapitel 4].

Diese Punkte fallen vor allem dann ins Gewicht, wenn große Arrays und viele Objekte erzeugt werden, wie es bei der derzeitigen Implementierung von *ESAComp* der Fall ist (mehrere Arrays der Länge n, $O(n)$ Objekte für Ersetzungsregeln).

Des Weiteren war bei den Tests ein erheblicher Speicheraufwand zu beobachten. So wird für die Kompression eines Textes der Länge 10^8 (das entspricht 95,3 MiB) derzeit bis zu 3,33 GiB Speicher benötigt. Dies liegt sicherlich auch an den bereits angesprochenen Debug- und Statistikfunktionen, jedoch liegt zudem die Vermutung nahe, dass viel Speicher-Mehraufwand durch die interne JVM-Verwaltung entsteht.

Eine systemnähere Implementierung (z.B. in der Programmiersprache *C*) könnte die Laufzeit und insbesondere den Speicherbedarf des Verfahrens deutlich verbessern.

6.2.5 Blockweise Kompression

Es ist denkbar, den Eingabetext blockweise mit *ESAComp* zu komprimieren. Hierzu wird der Text in Blöcke aufgeteilt und pro Block werden Ersetzungsregeln ermittelt. Der erhoffte Effekt ist die Beschleunigung des Gesamtprozesses, da das Verfahren keiner strikt linearen Laufzeit unterliegt. Es würde außerdem eine verteilte Verarbeitung ermöglicht.

Um dabei nicht zu viel Kompressionsrate einbüßen zu müssen, könnte ggf. ein Verfahren nach dem Prinzip „Divide & Conquer" entwickelt werden, welches die Ersetzungsregeln der einzelnen Blöcke abschließend zusammenführt.

6.2.6 Verbesserung der Kodierung

Die Quellpositionen der Ersetzungsregeln sind ein wichtiger Informationsträger für die Dekodierung. Es wurde beobachtet, dass diese sich quer über den Ursprungstext verteilen und daher keine offensichtlichen Eigenschaften aufweisen, die eine bessere Kodierung ermöglichen.

Für einen Text der Länge n werden Quellpositionen daher von *code2* einfach als gannzahlige Werte durch $\lceil \log_2(n) \rceil$ Bits kodiert. Eine bessere Kodierung könnte, sofern sie existiert, die Kompressionsraten deutlich verbessern.

6.2.7 Bessere Ersetzungen

Die Menge der durch Algorithmus 3.1 ermittelten Ersetzungen ist derzeit nicht optimal in dem Sinne, dass weder alle Redundanzen der Länge θ oder länger eliminiert werden, noch möglichst viele Symbole im Text ersetzt werden.

Ein Beispiel für bessere Ersetzungsmöglichkeiten stellt der Text $t =$ `abaaabbababb$` dar. Der Algorithmus ermittelt für t die Ersetzungsregeln $r_1 = (8,1,3)$ und $r_2 = (5,10,3)$. Eine beispielhafte Kodierung wäre demnach $t' =$ `abaa[10,3][1,3]bb$`.

Eine größere „Ausbeute" könnte jedoch erzielt werden, wenn statt $r_1 = (8,1,3)$ die Regel $r_1' = (1,8,3)$ gewählt würde, Ziel- und Quellposition also vertauscht würden. Hierdurch würde sich dann die Möglichkeit ergeben, zusätzlich eine Regel $r_3 = (9,7,3)$ einzuführen. Die Beispielkodierung würde sich dann zu $t'' =$ `[8,3]a[10,3]a[7,3]b$` ändern – es würden also mehr Symbole ersetzt, ohne die durchschnittliche Ersetzungslänge je Regel zu verringern.

Es scheint demnach in bestimmten Fällen lohnenswert, Ziel- und Quellposition einer Ersetzungsregel zu vertauschen, um „Platz" für weitere Ersetzungsregeln zu schaffen. Es konnte im Rahmen dieser Arbeit keine Methode gefunden werden, diese Fälle zu erkennen und auszunutzen. Das bedingte Vertauschen der Positionen hätte überdies deutliche Auswirkungen auf die Komplexität der Eliminerung von Überschneidungen.

A Testergebnisse

Dieser Anhang stellt die Ergebnisse der in Kapitel 5 spezifizierten Testdurchläufe auf.

A.1 LCP-sortierte Suffix-Liste versus LCP-Heap

Dieser Abschnitt enthält die Ergebnisse für den in Abschnitt 5.1 spezifizierten Test zum Laufzeitvergleich der *LCP-sortierten Suffix-Liste* („Liste") mit dem *LCP-Heap* („Heap").

Tabelle A.1 zeigt die gemessen Laufzeiten und Vergleichswerte je Durchlauf. Die letzte Zeile bildet den Mittelwert aller zehn Durchläufe.

Die Spalten sind in drei Bereiche unterteilt. Die Werte im Spaltenbereich *Liste* beziehen sich auf die Durchläufe mit der Liste, während sich der Bereich *Heap* auf die Durchläufe mit dem Heap beziehen. Der Bereich *Vergleich* bildet jeweils Verhältnisse von Heap zu Liste.

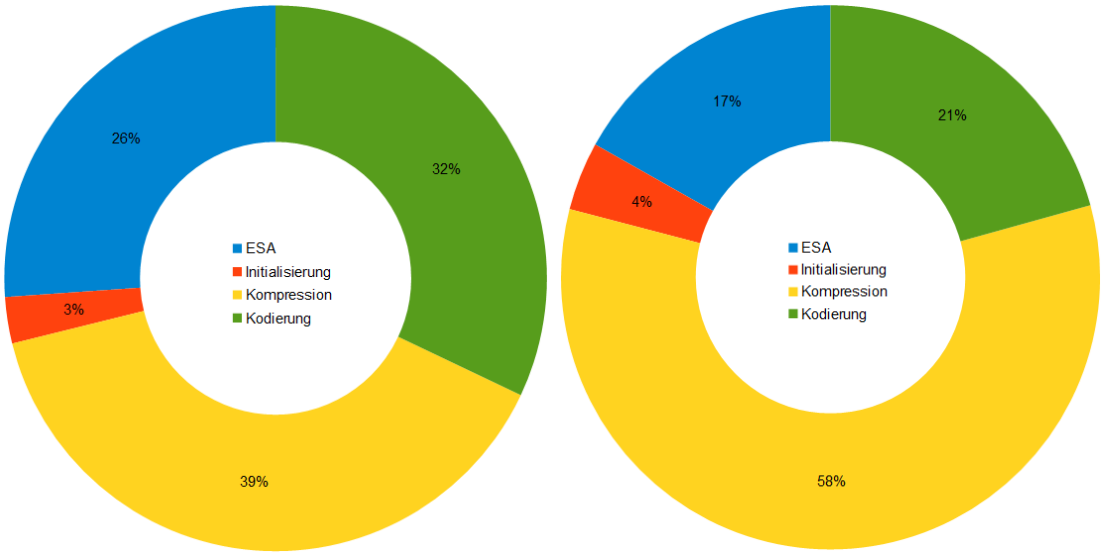
Je Bereich sind folgende Spalten aufgeführt:

- *ESA*: Zeit für den Aufbau des Enhanced Suffix Arrays mit *DivSufSort* [22],
- *Init.*: Zeit für den initialen Aufbau der jeweiligen Datenstruktur,
- *Kompr.*: Zeit für die Kompression mit *ESAComp* und der jeweiligen Datenstruktur und
- *Kod.*: Zeit für die Kodierung mit *code2*.

	Liste				Heap				Vergleich $\frac{\text{Heap}}{\text{Liste}}$	
#	ESA	Init.	Kompr.	Kod.	ESA	Init.	Kompr.	Kod.	Init.	Kompr.
1	967 ms	101 ms	1.440 ms	1.166 ms	968 ms	231 ms	3.326 ms	1.176 ms	2,29	2,31
2	966 ms	102 ms	1.450 ms	1.169 ms	972 ms	232 ms	3.352 ms	1.204 ms	2,27	2,31
3	969 ms	101 ms	1.440 ms	1.193 ms	955 ms	232 ms	3.312 ms	1.173 ms	2,30	2,30
4	959 ms	102 ms	1.449 ms	1.149 ms	962 ms	235 ms	3.320 ms	1.165 ms	2,30	2,29
5	963 ms	101 ms	1.440 ms	1.212 ms	964 ms	232 ms	3.316 ms	1.180 ms	2,30	2,30
6	959 ms	101 ms	1.441 ms	1.177 ms	954 ms	232 ms	3.319 ms	1.187 ms	2,30	2,30
7	960 ms	102 ms	1.440 ms	1.199 ms	950 ms	231 ms	3.308 ms	1.195 ms	2,26	2,30
8	960 ms	101 ms	1.437 ms	1.229 ms	969 ms	232 ms	3.333 ms	1.164 ms	2,30	2,32
9	959 ms	101 ms	1.440 ms	1.204 ms	968 ms	234 ms	3.304 ms	1.178 ms	2,32	2,29
10	958 ms	101 ms	1.440 ms	1.167 ms	954 ms	234 ms	3.326 ms	1.130 ms	2,32	2,31
∅	962 ms	101 ms	1.442 ms	1.187 ms	962 ms	233 ms	3.322 ms	1.175 ms	2,30	2,30

Tabelle A.1: Laufzeitmessungen für Liste und Heap.

Abbildung A.2 zeigt die durchschnittliche Laufzeitverteilung je Durchlauf mit der Liste (Abbildung A.2a) bzw. dem Heap (Abbildung A.2b). Die Zuordnung von Farbe zu Schritt erfolgt jeweils durch die beistehende Legende.



(a) Durschschn. Laufzeitverteilung mit Liste. **(b)** Durschschn. Laufzeitverteilung mit Heap.

Abbildung A.2: Durchschnittliche Laufzeitverteilugen.

A.2 Haupttest

In diesem Abschnitt werden die Ergebnisse des in Abschnitt 5.2 spezifizierten Haupttests erfasst.

A.2.1 Tabellen

Die nachstehenden Tabellen (Folgeseiten) enthalten die einzelnen Messwerte zeilenweise nach Schwellwert θ. Sie beziehen sich jeweils auf einen Text (siehe Tabelle 5.1) und sind in vier Spaltenbereiche unterteilt:

- Eingabe (Schwellwert θ),
- Messwerte bzgl. *ESAComp*,
- Messwerte bzgl. *LZ* und
- Vergleichswerte zwischen *ESAComp* und *LZ*.

Die einzelnen Spalten sind in Tabelle A.3 geschlüsselt.

Abschnitt	Spalte	Beschreibung
ESAComp	Liste	Initiale Anzahl der Elemente in der LCP-sortierten Suffix-Liste.
	Regeln	Anzahl der ermittelten Ersetzungsregeln.
	Ersetzt	Gesamtzahl der dadurch ersetzten Symbole.
	∅/Regel	Durschn. Anzahl der ersetzten Symbole je Regel.
	$\varnothing_d$	Durchschn. Ersetzungstiefe je Symbol.
	code1	Kompressionsrate bei Anwendung von *code1*.
	code2	Kompressionsrate bei Anwendung von *code2*.
LZ	Regeln	Anzahl der ermittelten Regeln.
	Ersetzt	Gesamtzahl der dadurch ersetzten Symbole.
	∅/Regel	Durschn. Anzahl der ersetzten Symbole je Regel.
	$\varnothing_d$	Durchschn. Ersetzungstiefe je Symbol.
	code1	Kompressionsrate bei Anwendung von *code1*.
	code2	Kompressionsrate bei Anwendung von *code2*.
Vergleich $\frac{\text{ESA}}{\text{LZ}}$	$\varnothing_d$	Verhältnis zwischen den durchschn. Ersetzungstiefen je Symbol.
	code1	Verhältnis zwischen den Kompressionsraten bzgl. *code1*.
	code2	Verhältnis zwischen den Kompressionsraten bzgl. *code2*.

Tabelle A.3: Schlüsselung der Spalten in den Ergebnistabellen.

	ESAComp							LZ						Vergleich $\frac{\text{ESA}}{\text{LZ}}$		
θ	Liste	Regeln	Ersetzt	∅/Regel	$\varnothing_d$	code1	code2	Regeln	Ersetzt	∅/Regel	$\varnothing_d$	code1	code2	$\varnothing_d$	code1	code2
2	52.422.804	3.476.953	52.114.936	14,99	51,40	53,65%	27,26%	3.562.850	52.425.940	14,71	15,74	54,37%	27,34%	3,27	98,68%	99,71%
3	52.370.078	3.238.273	51.637.576	15,95	28,73	50,92%	26,18%	3.551.047	52.408.642	14,76	13,80	54,22%	27,27%	2,08	93,91%	96,00%
4	52.121.087	3.014.578	50.966.491	16,91	19,03	48,79%	25,49%	3.516.200	52.336.707	14,88	12,15	53,83%	27,14%	1,57	90,64%	93,92%
5	51.420.613	2.797.540	50.098.339	17,91	14,02	47,13%	24,99%	3.448.382	52.149.429	15,12	10,87	53,15%	26,92%	1,29	88,67%	92,83%
6	50.009.476	2.611.145	49.166.364	18,83	11,18	46,07%	24,82%	3.337.794	51.766.847	15,51	9,43	52,19%	26,66%	1,19	88,27%	93,10%
7	47.736.003	2.434.508	48.106.542	19,76	9,25	45,39%	24,88%	3.176.515	51.090.287	16,08	8,09	51,02%	26,41%	1,14	88,97%	94,21%
8	44.622.564	2.254.889	46.849.209	20,78	7,79	45,05%	25,13%	2.962.469	50.024.841	16,89	6,84	49,79%	26,26%	1,14	90,48%	95,70%
9	40.863.624	2.070.864	45.377.009	21,91	6,55	45,05%	25,61%	2.702.943	48.499.718	17,94	5,75	48,74%	26,31%	1,14	92,43%	97,34%
10	36.802.559	1.881.690	43.674.443	23,21	5,55	45,41%	26,30%	2.409.974	46.498.844	19,29	4,79	48,08%	26,68%	1,16	94,45%	98,58%
11	32.788.996	1.687.053	41.728.073	24,73	4,72	46,15%	27,20%	2.100.835	44.073.514	20,98	3,95	47,99%	27,36%	1,19	96,17%	99,42%
12	29.090.917	1.486.812	39.525.422	26,58	3,98	47,30%	28,32%	1.791.706	41.313.348	23,06	3,24	48,54%	28,39%	1,23	97,45%	99,75%
13	25.890.741	1.284.386	37.096.310	28,88	3,37	48,84%	29,67%	1.496.100	38.343.931	25,63	2,65	49,69%	29,72%	1,27	98,29%	99,83%
14	23.273.133	1.086.282	34.520.958	31,78	2,84	50,73%	31,20%	1.223.736	35.287.245	28,84	2,16	51,37%	31,31%	1,31	98,75%	99,65%
15	21.220.770	898.070	31.885.990	35,51	2,39	52,89%	32,87%	982.674	32.298.493	32,87	1,77	53,39%	33,01%	1,35	99,06%	99,58%
16	19.661.462	725.735	29.300.965	40,37	2,03	55,19%	34,57%	776.782	29.494.817	37,97	1,47	55,60%	34,84%	1,38	99,26%	99,23%

Tabelle A.4: Testergebnisse für englischen Text.

	ESAComp							LZ						Vergleich $\frac{\text{ESA}}{\text{LZ}}$		
θ	Liste	Regeln	Ersetzt	∅/Regel	$\varnothing_d$	code1	code2	Regeln	Ersetzt	∅/Regel	$\varnothing_d$	code1	code2	$\varnothing_d$	code1	code2
2	52.420.267	3.086.060	52.062.999	16,87	59,00	47,79%	25,27%	3.302.421	52.425.337	15,87	25,53	44,10%	26,04%	2,31	108,37%	97,04%
3	52.266.927	2.854.406	51.599.691	18,08	44,65	45,14%	24,35%	3.275.924	52.384.394	15,99	23,06	43,82%	25,88%	1,94	103,01%	94,09%
4	51.555.881	2.649.391	50.984.646	19,24	35,97	43,18%	23,87%	3.196.467	52.207.193	16,33	20,77	43,10%	25,57%	1,73	100,19%	93,35%
5	50.148.354	2.461.046	50.231.266	20,41	30,88	41,74%	23,59%	3.070.313	51.839.564	16,88	18,26	42,12%	25,24%	1,69	99,10%	93,46%
6	48.200.340	2.287.641	49.364.241	21,58	23,85	40,75%	23,59%	2.913.170	51.279.422	17,60	16,52	41,09%	24,98%	1,44	99,17%	94,44%
7	45.821.465	2.123.567	48.379.797	22,78	20,83	40,13%	23,79%	2.730.418	50.493.131	18,49	14,52	40,15%	24,84%	1,43	99,95%	95,77%
8	43.176.991	1.966.960	47.283.548	24,04	18,55	39,83%	24,20%	2.533.311	49.470.083	19,53	13,36	39,47%	24,90%	1,39	100,91%	97,19%
9	40.466.094	1.821.479	46.119.700	25,32	17,15	39,83%	24,72%	2.334.384	48.260.644	20,67	12,13	39,12%	25,18%	1,41	101,81%	98,17%
10	37.782.150	1.684.461	44.886.538	26,65	15,95	40,09%	25,38%	2.134.250	46.878.513	21,96	10,79	39,08%	25,67%	1,48	102,58%	98,87%
11	35.201.787	1.551.832	43.560.248	28,07	14,86	40,59%	26,18%	1.939.306	45.365.959	23,39	9,92	39,36%	26,33%	1,50	103,13%	99,43%
12	32.802.463	1.428.829	42.207.215	29,54	13,77	41,30%	27,07%	1.758.785	43.793.199	24,90	8,15	39,95%	27,13%	1,69	103,38%	99,78%
13	30.601.180	1.313.624	40.824.755	31,08	13,00	42,18%	28,11%	1.592.020	42.193.210	26,50	7,46	40,78%	28,07%	1,74	103,43%	100,14%
14	28.616.341	1.204.166	39.401.801	32,72	12,34	43,22%	29,20%	1.434.841	40.528.631	28,25	6,88	41,85%	29,15%	1,79	103,27%	100,17%
15	26.840.242	1.100.823	37.954.999	34,48	11,71	44,40%	30,35%	1.292.690	38.890.859	30,09	6,45	43,08%	30,31%	1,82	103,06%	100,13%
16	25.257.862	1.008.855	36.575.479	36,25	10,96	45,63%	31,52%	1.169.954	37.360.987	31,93	6,04	44,36%	31,48%	1,81	102,86%	100,13%

Tabelle A.5: Testergebnisse für Quelltexte.

	ESAComp							LZ						Vergleich $\frac{\text{ESA}}{\text{LZ}}$		
θ	Liste	Regeln	Ersetzt	∅/Regel	$\varnothing_d$	code1	code2	Regeln	Ersetzt	∅/Regel	$\varnothing_d$	code1	code2	$\varnothing_d$	code1	code2
2	52.422.685	1.864.950	52.203.253	27,99	164,53	25,33%	15,48%	1.987.650	52.425.944	26,38	24,57	26,54%	16,04%	6,70	95,44%	96,51%
3	52.335.418	1.742.454	51.958.261	29,82	141,29	24,16%	14,97%	1.969.256	52.396.613	26,61	22,53	26,35%	15,93%	6,27	91,69%	93,97%
4	51.968.143	1.617.447	51.583.240	31,89	125,17	23,21%	14,63%	1.918.598	52.280.729	27,25	21,22	25,90%	15,72%	5,90	89,61%	93,07%
5	51.224.035	1.491.680	51.080.172	34,24	114,21	22,49%	14,42%	1.847.559	52.065.500	28,18	19,95	25,36%	15,51%	5,72	88,68%	92,97%
6	50.205.873	1.367.469	50.459.117	36,90	105,14	22,01%	14,36%	1.767.740	51.762.385	29,28	19,26	24,87%	15,34%	5,46	88,50%	93,61%
7	49.088.605	1.255.358	49.786.451	39,66	98,62	21,80%	14,45%	1.690.520	51.389.957	30,40	18,25	24,55%	15,27%	5,40	88,80%	94,63%
8	47.955.173	1.165.725	49.159.020	42,17	88,93	21,80%	14,65%	1.620.331	50.970.595	31,46	17,87	24,42%	15,31%	4,98	89,27%	95,69%
9	46.815.364	1.100.821	48.639.788	44,19	85,19	21,92%	14,87%	1.554.955	50.499.753	32,48	17,19	24,44%	15,45%	4,96	89,69%	96,25%
10	45.641.031	1.046.649	48.152.240	46,01	79,87	22,13%	15,13%	1.493.029	49.984.622	33,48	16,58	24,60%	15,66%	4,82	89,96%	96,62%
11	44.421.812	1.001.070	47.696.450	47,65	76,22	22,39%	15,40%	1.433.704	49.441.392	34,49	15,96	24,84%	15,93%	4,78	90,14%	96,67%
12	43.258.146	961.241	47.258.331	49,16	70,84	22,70%	15,70%	1.379.853	48.901.233	35,44	15,75	25,15%	16,24%	4,50	90,26%	96,67%
13	42.178.341	926.869	46.845.867	50,54	70,44	23,02%	16,01%	1.331.557	48.372.162	36,33	15,29	25,52%	16,57%	4,61	90,20%	96,62%
14	41.160.092	896.982	46.457.336	51,79	67,08	23,37%	16,31%	1.288.222	47.864.928	37,16	15,17	25,90%	16,91%	4,42	90,23%	96,45%
15	40.163.256	870.483	46.086.350	52,94	64,62	23,72%	16,60%	1.246.253	47.373.890	38,01	14,99	26,28%	17,25%	4,31	90,26%	96,23%
16	39.129.213	844.197	45.692.060	54,12	63,68	24,12%	16,96%	1.198.419	46.832.346	39,08	14,85	26,68%	17,66%	4,29	90,40%	96,04%

Tabelle A.6: Testergebnisse für XML.

	ESAComp							LZ						Vergleich $\frac{\text{ESA}}{\text{LZ}}$		
θ	Liste	Regeln	Ersetzt	∅/Regel	$\varnothing_d$	code1	code2	Regeln	Ersetzt	∅/Regel	$\varnothing_d$	code1	code2	$\varnothing_d$	code1	code2
2	52.428.214	6.613.738	51.371.096	7,77	19,99	90,32%	52,26%	6.787.630	52.428.438	7,72	16,50	90,63%	52,04%	1,21	99,66%	100,42%
3	52.417.860	5.910.541	49.964.702	8,45	11,10	83,61%	48,95%	6.784.617	52.423.539	7,73	13,77	90,59%	52,02%	0,81	92,29%	94,10%
4	52.222.159	5.367.085	48.334.334	9,01	7,66	79,47%	47,18%	6.750.290	52.345.196	7,75	10,41	90,29%	51,87%	0,74	88,02%	90,96%
5	49.320.193	4.872.199	46.354.790	9,51	5,77	76,64%	46,25%	6.387.844	51.244.606	8,02	6,88	87,55%	50,72%	0,84	87,54%	91,19%
6	32.953.680	3.849.552	41.241.555	10,71	4,11	72,74%	45,76%	4.430.902	43.168.734	9,74	4,24	76,82%	47,40%	0,97	94,69%	96,54%
7	19.773.788	1.715.856	28.439.379	16,57	2,69	68,67%	47,00%	1.754.558	28.258.398	16,11	2,90	69,53%	47,58%	0,93	98,76%	98,78%
8	17.095.653	653.197	21.000.766	32,15	2,25	68,67%	48,54%	689.941	21.042.506	30,50	2,51	69,08%	48,72%	0,90	99,41%	99,63%
9	16.365.182	440.184	19.296.662	43,84	2,11	69,07%	49,10%	470.848	19.353.949	41,10	2,39	69,37%	49,12%	0,88	99,57%	99,96%
10	15.899.223	366.717	18.635.459	50,82	2,02	69,35%	49,46%	392.742	18.690.233	47,59	2,33	69,59%	49,40%	0,87	99,66%	100,12%
11	15.517.375	318.402	18.152.309	57,01	1,95	69,63%	49,77%	341.816	18.212.880	53,28	2,26	69,83%	49,66%	0,86	99,71%	100,22%
12	15.188.435	282.381	17.756.078	62,88	1,88	69,90%	50,06%	303.847	17.822.409	58,66	2,20	70,07%	49,93%	0,85	99,76%	100,26%
13	14.898.218	253.943	17.414.822	68,58	1,82	70,17%	50,31%	273.946	17.486.640	63,83	2,14	70,30%	50,18%	0,85	99,82%	100,26%
14	14.638.208	230.766	17.113.521	74,16	1,74	70,44%	50,55%	249.358	17.188.074	68,93	2,09	70,55%	50,43%	0,83	99,84%	100,24%
15	14.402.477	211.231	16.840.031	79,72	1,67	70,70%	50,82%	228.527	16.916.447	74,02	2,04	70,79%	50,63%	0,82	99,87%	100,38%
16	14.186.994	195.090	16.597.916	85,08	1,62	70,95%	51,10%	211.170	16.674.381	78,96	1,99	71,02%	50,85%	0,81	99,90%	100,49%

Tabelle A.7: Testergebnisse für Protein-Sequenzen.

	ESAComp							LZ						Vergleich $\frac{\text{ESA}}{\text{LZ}}$		
θ	Liste	Regeln	Ersetzt	∅/Regel	$\varnothing_d$	code1	code2	Regeln	Ersetzt	∅/Regel	$\varnothing_d$	code1	code2	$\varnothing_d$	code1	code2
2	52.428.682	4.202.804	52.051.171	12,38	30,96	56,83%	32,71%	3.852.232	52.428.734	13,61	16,74	51,43%	29,58%	1,85	110,50%	110,58%
3	52.428.312	3.929.495	51.504.553	13,11	22,37	54,23%	31,11%	3.852.130	52.428.579	13,61	15,07	51,43%	29,56%	1,48	105,44%	105,24%
4	52.427.402	3.707.293	50.837.947	13,71	16,31	52,53%	30,03%	3.851.956	52.428.267	13,61	13,84	51,43%	29,55%	1,18	102,14%	101,62%
5	52.425.494	3.518.475	50.082.675	14,23	12,83	51,45%	29,18%	3.851.681	52.427.562	13,61	12,83	51,43%	29,53%	1,00	100,04%	98,81%
6	52.421.438	3.354.629	49.263.445	14,69	10,33	50,83%	28,57%	3.851.288	52.426.145	13,61	11,39	51,43%	29,52%	0,91	98,83%	96,78%
7	52.408.331	3.210.305	48.397.501	15,08	8,61	50,55%	28,15%	3.850.363	52.422.232	13,61	10,22	51,42%	29,51%	0,84	98,31%	95,39%
8	52.358.406	3.080.406	47.488.208	15,42	7,34	50,55%	27,87%	3.847.463	52.407.776	13,62	8,89	51,41%	29,49%	0,83	98,33%	94,51%
9	52.161.329	2.961.787	46.539.256	15,71	6,38	50,78%	27,69%	3.838.409	52.356.508	13,64	7,56	51,39%	29,45%	0,84	98,81%	94,02%
10	51.402.023	2.851.564	45.547.249	15,97	5,58	51,20%	27,61%	3.808.655	52.167.395	13,70	6,33	51,35%	29,38%	0,88	99,71%	93,98%
11	48.857.373	2.740.761	44.439.219	16,21	4,91	51,83%	27,61%	3.718.248	51.522.335	13,86	5,13	51,37%	29,23%	0,96	100,90%	94,46%
12	42.070.084	2.607.410	42.972.358	16,48	4,32	52,85%	27,70%	3.479.904	49.598.835	14,25	3,93	51,86%	28,98%	1,10	101,91%	95,58%
13	29.565.024	2.394.906	40.422.310	16,88	3,62	54,88%	27,97%	2.986.447	44.984.342	15,06	2,88	54,07%	28,88%	1,26	101,50%	96,85%
14	17.812.318	1.998.498	35.269.006	17,65	2,77	59,41%	28,64%	2.258.225	36.866.034	16,33	2,02	59,83%	29,40%	1,37	99,30%	97,41%
15	11.293.156	1.395.518	26.827.286	19,22	1,96	67,46%	29,99%	1.485.049	26.919.059	18,13	1,42	68,48%	30,53%	1,38	98,51%	98,23%
16	8.397.716	836.064	18.435.476	22,05	1,43	76,00%	31,57%	899.574	18.561.993	20,63	1,07	76,61%	31,91%	1,34	99,20%	98,93%

Tabelle A.8: Testergebnisse für DNA-Sequenzen.

A.2.2 Graphen

Die folgenden Abbildungen (Folgeseiten) geben eine grafische Übersicht über einige ausgewählte Testergebnisse.

Auf die X-Achse wird hierbei jeweils der eingegebene Schwellwert θ gelegt, während auf der Y-Achse der jewelige Messwert abgetragen ist. Für jeden der in Tabelle 5.1 aufgeführten Texte erhält jedes Diagramm einen unterschiedlich gefärbten Graphen. Zur Zuordnung von Farbe zu Text dient jeweils eine Legende auf der rechten Seite.

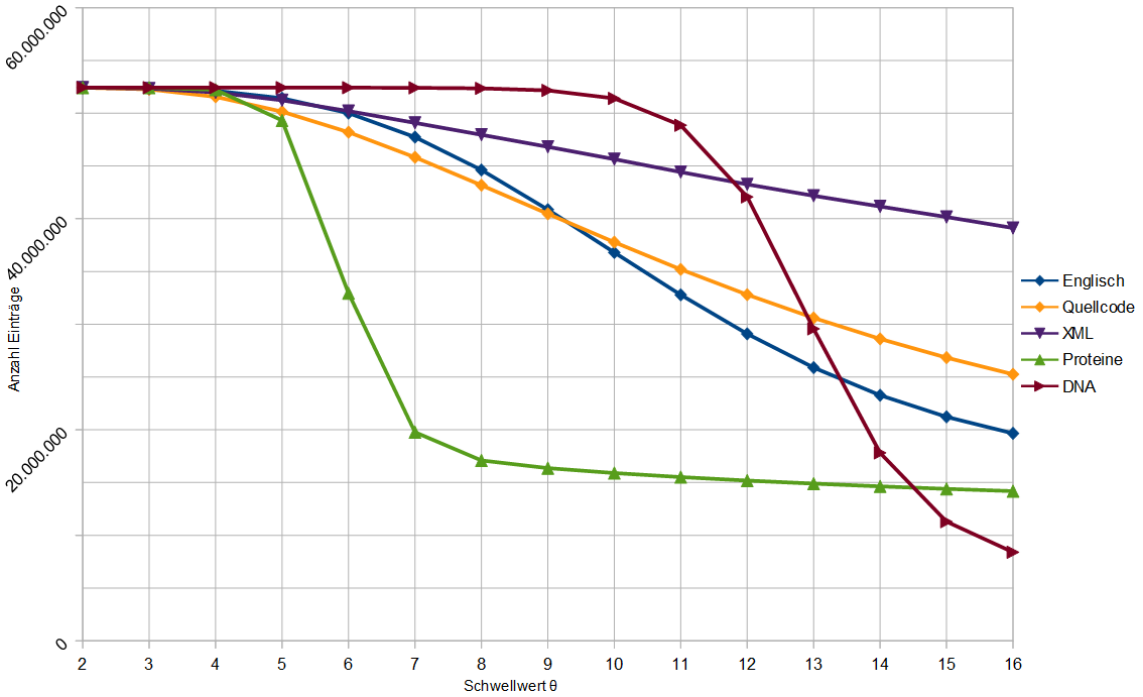


Abbildung A.9: Initiale Länge der LCP-sortierten Suffix-Liste, je Text nach Schwellwert θ.

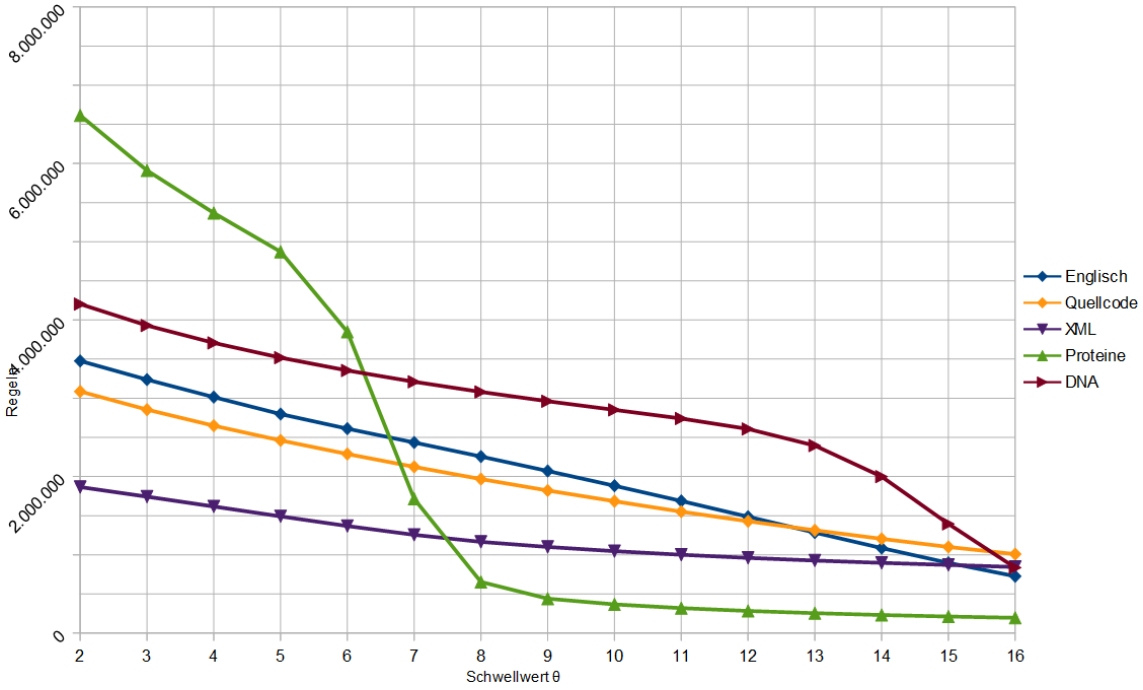


Abbildung A.10: Durch *ESAComp* erzeugte Anzahl Ersetzungsregeln, je Text nach Schwellwert θ.

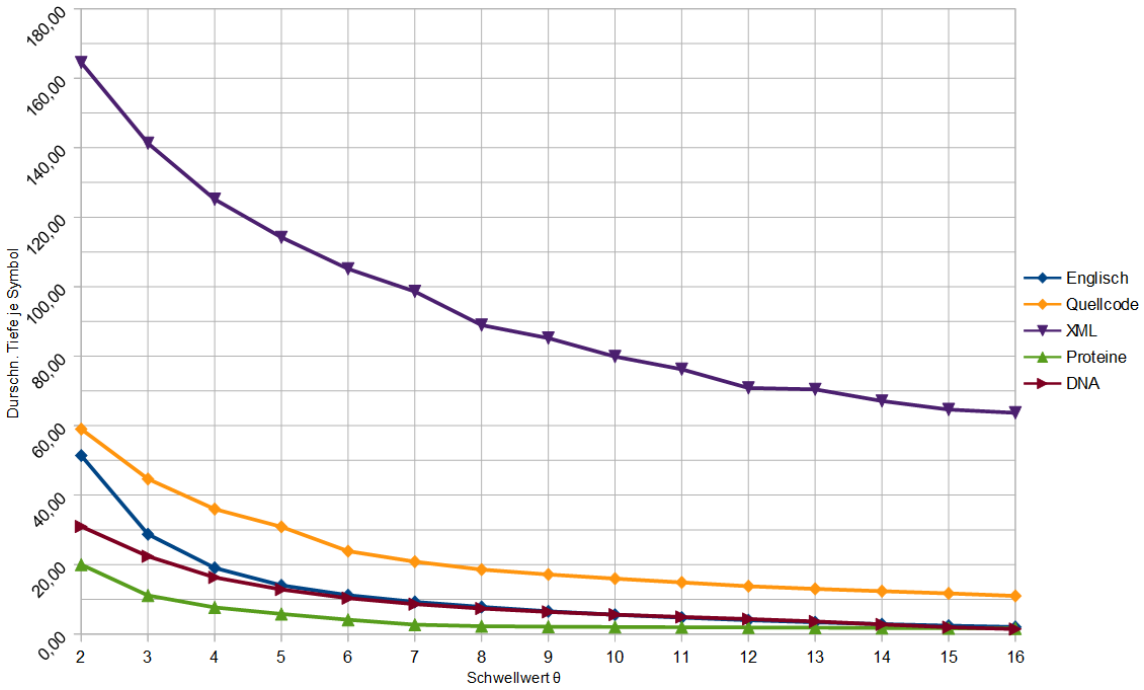


Abbildung A.11: Durschn. Ersetzungstiefe je Symbol, je Text nach Schwellwert θ.

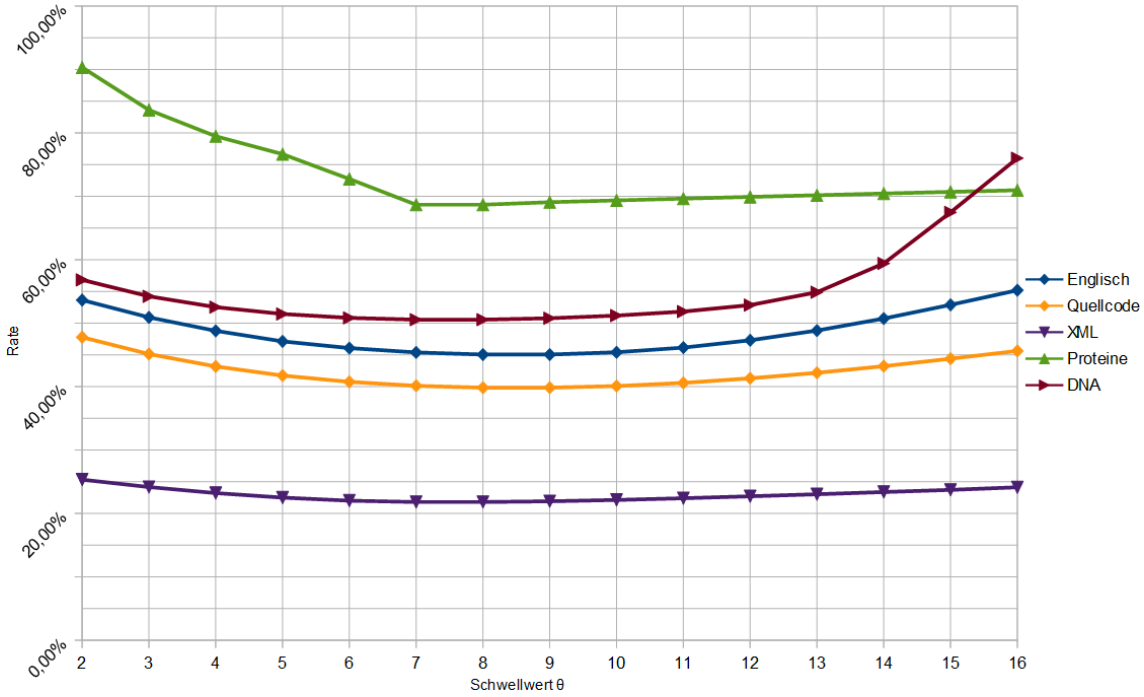


Abbildung A.12: Erzielte Kompressionsrate durch die Kombination *ESAComp* und *code1*, je Text nach Schwellwert θ.

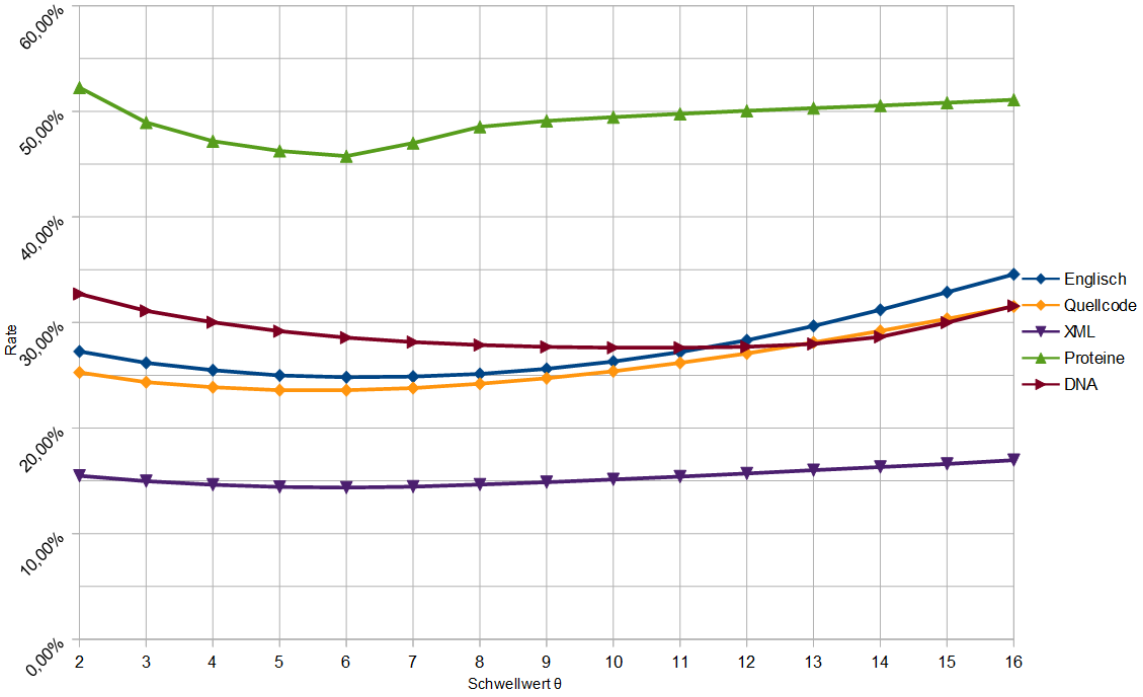


Abbildung A.13: Erzielte Kompressionsrate durch die Kombination *ESAComp* und *code2*, je Text nach Schwellwert θ.

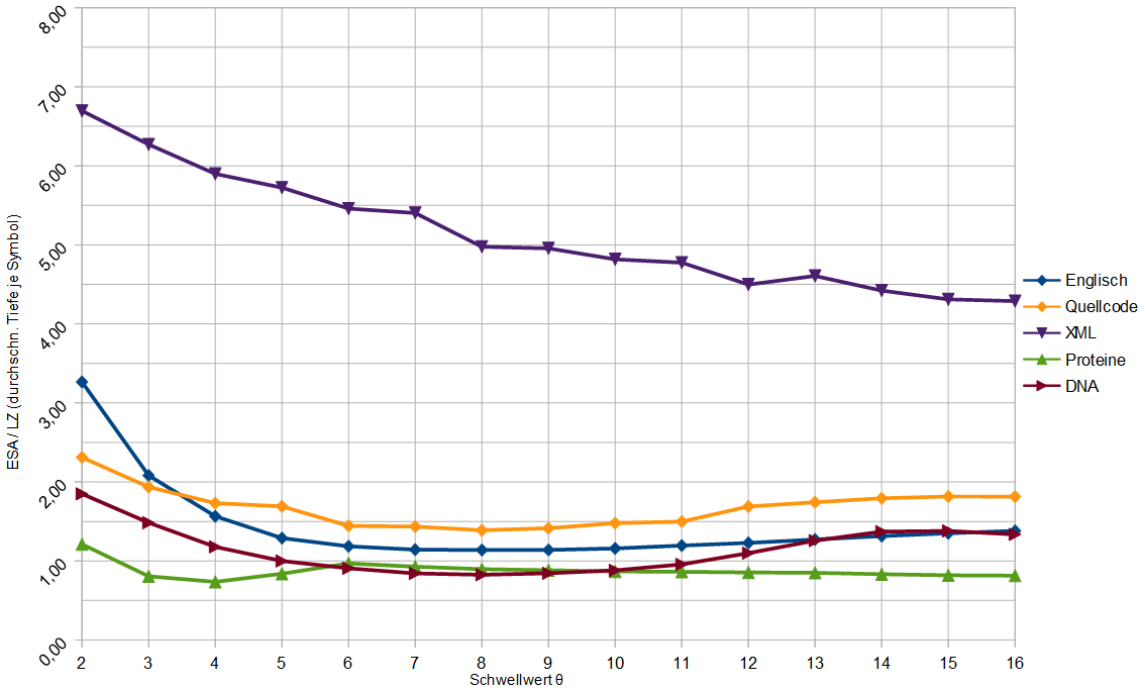


Abbildung A.14: Verhältnis $\frac{\text{ESA}}{\text{LZ}}$ der durschn. Ersetzungstiefe je Symbol in der durch *ESAComp* bzw. *LZ* ermittelten Regelmenge, je Text nach Schwellwert θ.

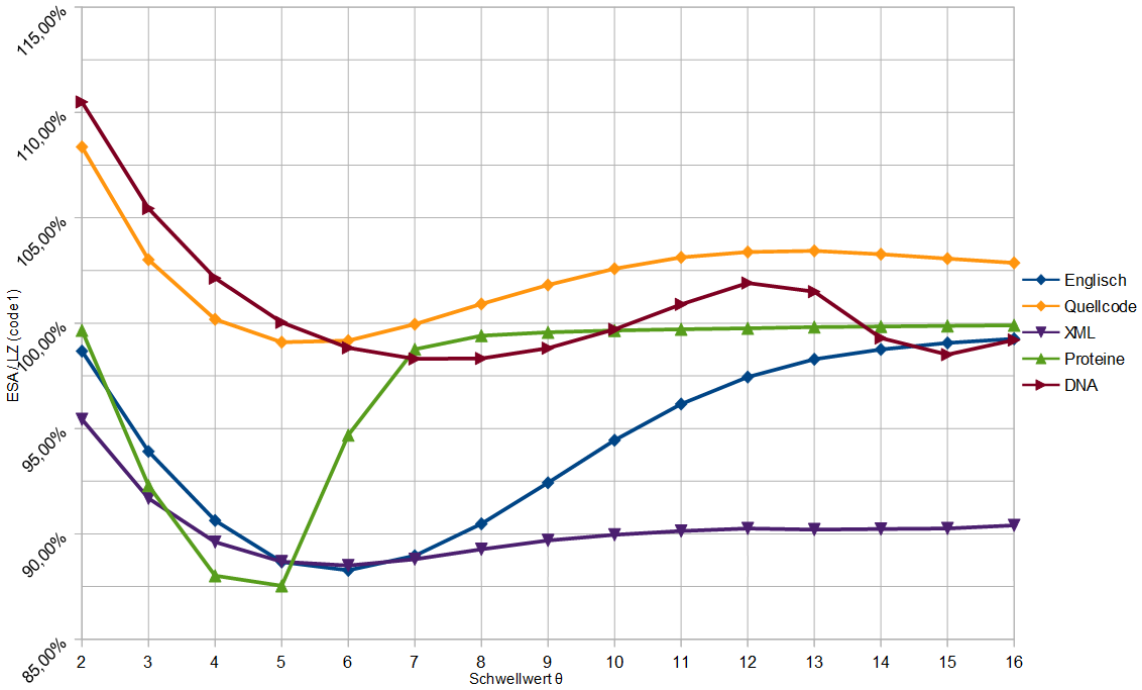


Abbildung A.15: Verhältnis $\frac{\text{ESA}}{\text{LZ}}$ der durch *ESAComp* bzw. *LZ* erzielten Kompressionsraten bzgl. *code1*, je Text nach Schwellwert θ.

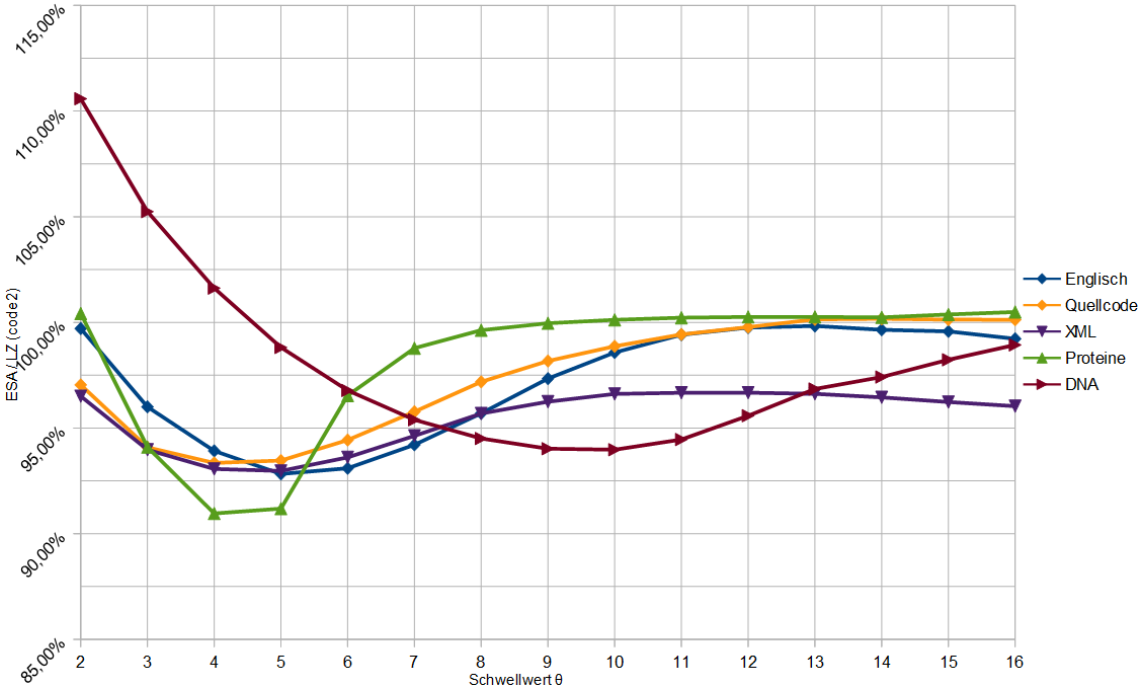


Abbildung A.16: Verhältnis $\frac{\text{ESA}}{\text{LZ}}$ der durch *ESAComp* bzw. *LZ* erzielten Kompressionsraten bzgl. *code2*, je Text nach Schwellwert θ.

B Quelltext und Programm

In diesem Anhang wird das beiliegende Medium und die Verwendung des in Kapitel 4 entwickelten Programms dokumentiert.

B.1 Aufbau des Verzeichnissystems

Das Verzeichnissystems des beiliegenden Mediums ist wie in Tabelle B.1 aufgebaut.

Dateiname	Beschreibung
`BA_Textkompression.pdf`	PDF-Version dieser Arbeit.
`esacomp/lib`	Enthält die verwendete Bibliothek *jSuffixArrays*.
`esacomp/src`	Enthält die Quelldateien.
`esacomp/build.xml`	Buildfile für *Apache Ant* zum automatischen Bauen der Quellen.
`esacomp/esacomp`	Unix-Shell-Script zur Ausführung auf Unix-basierten Systemen.
`esacomp/esacomp.bat`	Batch-Script zur Ausführung auf Windows-Systemen.
`esacomp/esacomp.jar`	Ausführbares JAR-Archiv des Programms.

Tabelle B.1: Aufbau des Dateisystems des beiliegenden Mediums.

B.2 Kompilierung

Zur automatischen Kompilierung der Quelltexte und Generierung des ausführbaren JAR-Archivs wird ein XML-Buildfile für *Apache Ant* [4] beigelegt.

Es wird ein *Java Development Kit 7* oder höher zum Kompilieren benötigt.

B.3 Systemvoraussetzungen

Für die Ausführung des Programms wird *Java 7* oder höher benötigt.

Die Verarbeitung von Texten länger als 50 MiB erfordert derzeit mehr als 2 GiB Heapspeicher, was nur die 64-Bit-Versionen von Java unterstützen.

Es wird außerdem vorausgesetzt, dass sich der Pfad zum Programm `java` in der Umgebungsvariable `PATH` befindet (aktuelle Java-Installationen sorgen automatisch dafür).

B.4 Ausführung

Das Programm kann mit dem beiliegenden Batch- oder Shell-Script ausgeführt werden.

B.4.1 Optionen

Die durchzuführenden Aktionen müssen dem Programm über Optionen mitgeteilt werden. Diese folgen der Java-Property-Syntax und werden daher mit `-Dname=wert` übergeben.

Wird weder Eingabetext noch Einagbedatei angegeben, schreibt das Programm eine Nutzungsanweisung mit den verfügbaren Optionen. Tabelle B.2 listet die verfügbaren Optionen mit Beschreibung auf.

Option	Beschreibung
`-Dtext=<text>`	Der Eingabetext.
`-Dfile=<path>`	Der Pfad zur Eingabedatei.
`-Dtheta=<int>`	Der Schwellwert θ (Standardwert 2).
`-Dcompressor=<esa|lz>`	Auswahl des Kompressors: *ESAComp* oder *LZ* (Standard `esa`).
`-Dmaxlcp=<list|heap>`	Auswahl der MaxLCP-Datenstruktur: *LCP-sortierte Suffix-Liste* oder *LCP-Heap* (Standard `list`).
`-Dcode1=[path]`	Kodierung mit *code1*, optional Ausgabe in Datei.
`-Dcode2=[path]`	Kodierung mit *code2*, optional Ausgabe in Datei.
`-Dkgram=<int>`	Länge der k-gramme, die *code2* gesondert kodiert (Standard 3) *(zum Deaktivieren `-Dkgram=0` angeben).*
`-DcacheESA`	Enhanced Suffix Array in Datei cachen bzw. aus Cache lesen.
`-DcacheRules`	Ersetzungsregeln in Datei cachen bzw. aus Cache lesen.
`-DprintESA`	Ausgabe des Enhanced Suffix Arrays für den Eingabetext.
`-DprintSufList`	Ausgabe der initialen LCP-sortierten Suffix-Liste (nur *ESAComp*).
`-DprintProgress`	Ausgabe von Fortschrittsinformationen.
`-DprintRules`	Ausgabe aller ermittelten Ersetzungsregeln.
`-DprintStats`	Ausgabe statistischer Messgrößen.
`-DprintTime`	Ausgabe von Laufzeiten.
`-DprintCode`	Ausgabe einer einfachen Kodierung.

Tabelle B.2: Verfügbare Kommandozeilen-Optionen.

Es ist zu beachten, dass standardmäßig eine Kompression mit dem Schwellwert $\theta = 2$, jedoch keinerlei Kodierung oder Ausgabe erfolgt. Alle gewünschten Ausgaben müssen über die entsprechenden Optionen angefordert werden.

Die Kompression von beliebigen Binärdateien (ohne Textinhalt) ist möglich, für diese ist jedoch nur *code2* ohne Kodierung von k-grammen (`-Dkgram=0`) sinnvoll.

B.4.2 Beispielaufrufe

Einfache Kodierung einer Eingabe

Abbildung B.3 zeigt einen Aufruf zur einfachen Kodierung des Textes „abcdebcdeabc" mit Ausgabe des Enhanced Suffix Arrays, der initialien LCP-sortierten Suffix-Liste und aller erzeugten Regeln.

```
$ esacomp -Dtext=abcdebcdeabc
          -DprintESA -DprintSufList -DprintRules -DprintCode

Enhanced Suffix Array:
 S | Suffix       | LCP
---------------------------
10 | abc          |  -1
 1 | abcdebcdeabc |   3
11 | bc           |   0
 6 | bcdeabc      |   2
 2 | bcdebcdeabc  |   4
12 | c            |   0
 7 | cdeabc       |   1
 3 | cdebcdeabc   |   3
 8 | deabc        |   0
 4 | debcdeabc    |   2
 9 | eabc         |   0
 5 | ebcdeabc     |   1


Initial LCP-sorted suffix list (5):
[2, 3, 1, 4, 6, ]

Computing rules ...
... adding rule: (2, 6, 4)
... adding rule: (6, 11, 2)

Coded text:
12:a{6,4}{11,2}deabc
```

Abbildung B.3: Beispielaufruf: Einfache Kodierung des Textes „abcdebcdeabc".

Kodierung einer großen Textdatei

Abbildung B.4 zeigt einen Aufruf zur Verarbeitung der Datei `english50.txt` (50 MiB), für welche *code1* simuliert wird und *code2* in eine Datei ausgegeben wird. Außerdem werden Fortschrittsinformationen, Zeitmessungen und Statistiken über die Kompression ausgegeben.

```
$ esacomp -Xmx2G
          -Dfile=english50.txt
          -Dcode1 -Dcode2=english50.code2
          -DprintProgress -DprintTime -DprintStats

Computing ESA ... took 19214 ms.
Creating initial LCP list ... took 3377 ms.
Computing rules ...
... 5.02%
... 10.04%
... 15.05%
... 20.05%
... 25.05%
... 30.05%
... 35.05%
... 40.05%
... 45.05%
... 50.05%
... 55.05%
... 60.05%
... 65.05%
... 70.05%
... 75.05%
... 80.05%
... 85.05%
... 90.05%
... 95.05%
... done after 45541 ms.
Total time: 68365 ms
Gathering rule statistics ...

Statistics:
Input text length: 52428800
Theta: 2
Initial suffix list size: 52422804
Rules (phrases): 3476953
Symbols substituted: 52114936
Average per rule: 14
Longest substitution: 109394
Raw symbols left: 313864
Maximum depth: 1114
Average depth per symbol: 51.40080579757691

Simulating code1  ...
Coded length (code1): 28129493 (53.65%)

Writing code2 to english50.code2 ...
Coded length (code2): 14292537 (27.26%) (took 67315 ms)
```

Abbildung B.4: Beispielaufruf: Kodierung einer Datei und Ausgabe von Statistiken.

Kodierung einer Binärdatei

Abbildung B.5 zeigt einen Aufruf zur Kodierung einer Binärdatei `audio.wav` mit *code2* mit dem Schwellwert $\theta = 3$ und Deaktivierung der k-gramm-Kodierung.

```
$ esacomp -Dfile=audio.wav -Dtheta=3
          -Dcode2=audio.code2 -Dkgram=0

Statistics:
Input text length: 76376
Theta: 3
Initial suffix list size: 38174
Rules (phrases): 10100
Symbols substituted: 52739
Average per rule: 5
Longest substitution: 4045
Raw symbols left: 23637
Maximum depth: 4649
Average depth per symbol: 89.0456295171258

Writing code2 to audio.code2 ...
Coded length (code2): 61299 (80.26%) (took 299 ms)
```

Abbildung B.5: Beispielaufruf: Kodierung einer Binärdatei und Ausgabe von Statistiken.

B.4.3 Verarbeitung langer Texte

Für die Verarbeitung von längeren Texten muss Java mehr Heapspeicher zur Verfügung gestellt werden. Diese kann mit dem Kommandozeilenargument `-Xmx` festgelegt werden.

Empfehlenswert ist für Texte bis 50 MiB ein maximaler Heapspeicher von 2 GiB (`-Xmx2G`), für längere Texte möglichst viel. Ein Indikator für zu wenig Heapspeicher ist die Ausgabe der Fehlermeldung „`java.lang.OutOfMemoryError: Java heap space`".

Literaturverzeichnis

[1] bzip2: Home. `http://www.bzip.org/`. The current version is 1.0.6, released 20 Sept 2010.

[2] Project Gutenberg. `https://www.gutenberg.org/`. last modified on 11 October 2014.

[3] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms*, 2(1):53–86, March 2004.

[4] The Apache Software Foundation. Apache ant. `http://ant.apache.org/`. Last Published: 05/26/2014 02:39:30.

[5] Albrecht Beutelspacher. *Kryptologie* -. Vieweg Friedr. + Sohn Ver, Braunschweig, Wiesbaden, 7, verb. aufl. 2005 edition, 2005.

[6] Gerth Stølting Brodal. Worst-case efficient priority queues. In *Proceedings of the Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '96, pages 52–58, Philadelphia, PA, USA, 1996. Society for Industrial and Applied Mathematics.

[7] Gerth Stølting Brodal, George Lagogiannis, and Robert E. Tarjan. Strict fibonacci heaps. In *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing*, STOC '12, pages 1177–1184, New York, NY, USA, 2012. ACM.

[8] Gang Chen, Simon J. Puglisi, and William F. Smyth. Lempel-Ziv factorization using less time & space. *Mathematics in Computer Science*, 1(4):605–623, 2008.

[9] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[10] P. Ferragina and G. Navarro. Pizza & chili corpus – compressed indexes and their testbeds. `http://pizzachili.dcc.uchile.cl/texts.html`. Last update: September, 2005.

[11] Johannes Fischer. Inducing the lcp-array. In Frank Dehne, John Iacono, and Jörg-Rüdiger Sack, editors, *Algorithms and Data Structures*, volume 6844 of *Lecture Notes in Computer Science*, pages 374–385. Springer Berlin Heidelberg, 2011.

[12] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. In *Proceedings of the 25th Annual Symposium onFoundations of Computer Science, 1984*, SFCS '84, pages 338–346, Washington, DC, USA, 1984. IEEE Computer Society.

[13] Jean-loup Gailly and Mark Alder. The gzip home page. `http://www.gzip.org/`. Last modification: July 27th, 2003.

[14] Simon Gog and Enno Ohlebusch. Compressed suffix trees: Efficient computation and storage of lcp-values. *J. Exp. Algorithmics*, 18:2.1:2.1–2.1:2.31, May 2013.

[15] Gaston H. Gonnet, Ricardo A. Baeza-Yates, and Tim Snider. Information retrieval. chapter New Indices for Text: PAT Trees and PAT Arrays, pages 66–82. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[16] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, and Alex Buckley. The java language specification. `http://docs.oracle.com/javase/specs/jls/se7/html/index.html`. Java SE 7 Edition, 2013-02-28.

[17] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In *Proceedings of the 30th International Conference on Automata, Languages and Programming*, ICALP'03, pages 943–955, Berlin, Heidelberg, 2003. Springer-Verlag.

[18] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, CPM '01, pages 181–192, London, UK, UK, 2001. Springer-Verlag.

[19] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching*, CPM'03, pages 186–199, Berlin, Heidelberg, 2003. Springer-Verlag.

[20] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *Proceedings of the 14th Annual Conference on Combinatorial Pattern Matching*, CPM'03, pages 200–210, Berlin, Heidelberg, 2003. Springer-Verlag.

[21] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. In *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[22] Yuta Mori. libdivsufsort: A lightweight suffix-sorting library. `https://code.google.com/p/libdivsufsort/`. libdivsufsort version 2.0.1, uploaded Nov 2012.

[23] Ge Nong. Practical linear-time o(1)-workspace suffix sorting for constant alphabets. *ACM Trans. Inf. Syst.*, 31(3):15:1–15:15, August 2013.

[24] Ge Nong, Sen Zhang, and Wai Hong Chan. Linear suffix array construction by almost pure induced-sorting. In *Proceedings of the 2009 Data Compression Conference*, DCC '09, pages 193–202, Washington, DC, USA, 2009. IEEE Computer Society.

[25] SIB Swiss Institute of Bioinformatics. Uniprotkb/swiss-prot. `http://web.expasy.org/docs/swiss-prot_guideline.html`.

[26] Enno Ohlebusch. *Bioinformatics Algorithms: Sequence Analysis, Genome Rearrangements, and Phylogenetic Reconstruction.* Oldenbusch Verlag, 2013.

[27] Oracle. Oracle jrockit introduction. `http://docs.oracle.com/cd/E15289_01/doc.40/e15058/toc.htm`. Release R28.

[28] Stanisław Osiński and Dawid Weiss. jsuffixarrays: Suffix arrays for java. `http://labs.carrotsearch.com/jsuffixarrays.html`. Stand: 04.01.2015.

[29] Klaus-Bernd Schürmann and Jens Stoye. An incomplex algorithm for fast suffix array construction. *Softw. Pract. Exper.*, 37(3):309–329, March 2007.

[30] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes (2Nd Ed.): Compressing and Indexing Documents and Images.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[31] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE TRANSACTIONS ON INFORMATION THEORY*, 23(3):337–343, 1977.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet sowie Zitate kenntlich gemacht habe.

Dortmund, den 4. Februar 2015

Patrick Dinklage