technische universität
dortmund

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

Textkompression mithilfe einer Variante von LZ78

Bachelorarbeit

Florian Kleine
Matrikelnummer: 157020

19. September 2016

Betreuer:
Prof. Dr. Johannes Fischer
Dominik Köppl

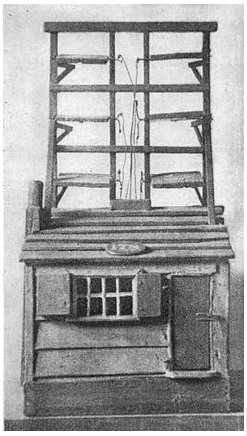
Inhaltsverzeichnis

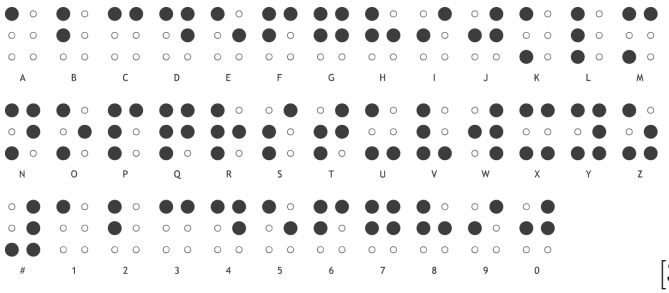
1 Einleitung

Die Datenkompression spielt in der Informatik eine große Rolle. Zwar sind die Datenträger im Vergleich zu früher um ein Vielfaches größer und vor allem günstiger geworden, stoßen bei den heute anfallenden riesigen Datenmengen aber immer noch an ihre Grenzen. Deshalb ist es sinnvoll die Daten mit geschickten Verfahren so zu komprimieren, dass sie später verlustfrei in den Ursprungszustand zurückübersetzt werden können. In dieser Arbeit soll es darum gehen, Texte mithilfe einer Variante des Lempel-Ziv78-Verfahrens in Faktoren zu zerlegen, zu kodieren und so verlustfrei zu komprimieren. Dieses Verfahren stützt sich auf der Eliminierung von Redundanzen, indem Teile des Textes durch Verweise auf vorher auftretende gleiche Teile ersetzt werden. Solche Verweise benötigen weniger Speicher als der Text, was so zu einer Kompression des kompletten Textes führt. Das aus dieser Arbeit entstehende Verfahren wird anschließend mit bereits vorhandenen Kompressionsverfahren (z.B. gzip und 7zip) verglichen.

1.1 Anekdote

Um die Bedeutung von Datenkompression zu verdeutlichen und zu zeigen, wie lange man sich über dieses Thema schon Gedanken macht, hier eine kleinen Anekdote:
Ende des 18. Jahrhunderts benötigte die britische Marine eine schnelle Kommunikation zwischen London und dem Marinestützpunkt an der Küste. Dies wurde durch eine Kette von Hütten auf Hügeln in etwa 5 Meilen Entfernung umgesetzt. Diese Hütten(sog. *Klappentelegraphen*1) hatten 6 Klappen auf dem Dach, die entweder geschlossen oder geöffnet waren. Durch diese 64 möglichen Kombinationen von offenen und geschlossenen Klappen konnte man Buchstanden des Braille-Alphabets1 darstellen. Da das Alphabet weniger als 64 Buchstaben hat, gab es freie Kombinationen von Klappenstellungen. Diese freien Kombinationen wurden dazu benutzt die Kommunikation zu beschleunigen. Oft benutzen Wörtern wie 'for' oder 'the' wurden freie Kombinationen zugewiesen. Anderen Worten wie 'father' oder 'mother' wurde die freie Kombination 'dot5' und die Kombination für 'f' bzw. 'm' zugewiesen. So erreichte man eine Kompression von 20% bei englischen Texten.[1, S.1-4]
Dieses Problem hat sich bis heute nicht geändert. Wie verringern wir die Datenmenge möglichst stark und beschleunigen somit die Übertragung, ohne Informationen zu verlieren?


[2]



[3]

Abbildung 1: Links sieht man einen Klappentelegraphen. Rechts das englische Braille-Aphabet.

2 Theoretische Grundlagen

Im Folgenden werden Datenstrukturen und Operationen eingeführt, die wir in dieser Arbeit benötigen werden.

2.1 Operationen auf Strings

Sei das Alphabet Σ definiert als eine Menge von Zeichen, dann bezeichnet Σ^* die Menge aller Worte, die aus dem Alphabet gebildet werden können. Jedes dieser Worte bezeichnet man als String. Sei s ein String mit der Länge n.

2.1 Definition (Länge) Sei $|s|$ die Länge des String s, das heißt die Anzahl von Zeichen in s.

2.2 Definition (Leerstring) Sei $\varepsilon \in \Sigma^*$ der leere String. Es gilt $|\varepsilon| = 0$.

2.3 Definition (Symbolzugriff) Für $x \in \mathbb{N}$ und $x \leq n$ sei $s[x]$ das x-te Zeichen aus s.

2.4 Definition (Teilstring) Für $x, y \in \mathbb{N}$ und $1 \leq x < y \leq n$ sei $s[x, y]$ die Zeichenfolge vom x-ten bis zum y-ten Zeichen aus s. s[x] und s[y] einschließlich.

2.5 Definition (Suffix) Für $x \in \mathbb{N}$ und $x \leq n$ sei $s[x..]$ das x-te Suffix von s. Also gilt $s[x..] = s[x, n]$.

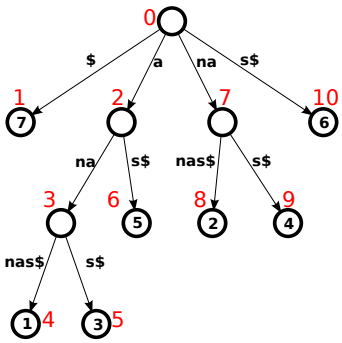
2.2 Suffix-Tree

2.6 Definition (Suffix-Tree) Ein Suffix-Tree eines Strings s ist ein Baum mit n Blättern. Alle inneren Knoten erfüllen folgende Bedingungen:

- Jeder Knoten hat mindestens 2 Kinder.
- Jede Kante ist mit einem nicht-leeren Teilstring von s markiert.

- Die Markierung ausgehender Kanten eines Knotens beginnen nicht mit dem gleichen Zeichen.
- Die Konkatenation von allen Zeichen auf dem Pfad von der Wurzel zum Blatt i ist das i-te Suffix $s[i..]$ von s.

2.7 Beispiel (Suffix-Tree) $s =$ `ananas$`



[4]

Abbildung 2: Diese Abbildung zeigt den Suffix-Tree zu $s =$ `ananas$`. Die Blätter sind hierbei nummeriert. Der String auf dem Pfad von der Wurzel zum Blatt i repräsentiert das Suffix $s[i..]$. Die roten Zahlen an den Knoten sind durch eine Pre-Order-Nummerierung entstanden und sind die IDs der Knoten.

2.3 Min-Heap

Ein **(binärer) Heap** ist ein Binärbaum, dessen Knoten je ein Element einer Menge X enthalten. Jeder Knoten erfüllt bezüglich einer totalen Ordnung $(X, \preceq)$ die *Heap-Eigenschaft*. Diese Eigenschaft ist wie folgt definiert:

2.8 Definition (Heap-Eigenschaft) Gegeben: Eine Menge X und eine Relation $\preceq$. Sei x_l der linke und x_r der rechte Kindknoten von x, so gilt:

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

Ein *Min-Heap* ist ein Heap, dessen Elemente die Relation $\geq$ erfüllen. Das heißt jeder Kindknoten enthält ein Element, das mindestens so groß ist wie das seines Elternknotens. Abbildung 3 zeigt beispielhaft einen *Min-Heap* $(\mathbb{N}, \geq)$.

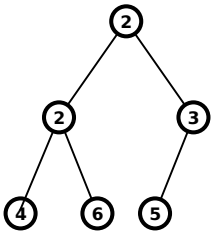


Abbildung 3: Beispiel eines *Min-Heaps* $(\mathbb{N}, \geq)$.

Eine effiziente Darstellung eines *Heaps* ist ein Array H, was folgende Eigenschaften erfüllt:

- Die erste Stelle von H ist die Wurzel des Baums.
- Der linke Kindknoten des $i - ten$ Knotens steht an der Stelle $2i$.
- Der rechte Kindknoten des $i - ten$ Knotens steht an der Stelle $2i + 1$.
- Der Vaterknoten des $i - ten$ Knotens steht an der Stelle $\lfloor \frac{i}{2} \rfloor$. $(i > 1)$

In Abbildung 4 sehen wir die Array-Darstellung des Baums aus Abbildung 3.

i	1	2	3	4	5	6
H[i]	2	2	3	4	6	5

Abbildung 4: Array-Darstellung des Baums aus Abbildung 3.

Ein Zugriff auf das kleinste Element des *Min-Heaps* ist in $O(1)$ Zeit möglich, da das kleinste Element des Heaps immer an der ersten Stelle liegt. Fügen wir ein neues Element in den Heap ein oder Löschen ein Element, so muss die *Heap-Eigenschaft* weiterhin gelten. Daher benötigen Einfüge- und Löschoperationen $O(\log n)$ Zeit, wenn der Heap n Elemente enthält. [5, S.115ff]

3 Idee der LZ78 Variante

Um die Grundidee des Verfahrens, das wir in dieser Arbeit entwickeln, zu erläutern, betrachten wir im Folgenden zunächst LZ78.

3.1 LZ78

LZ78 wurde 1978 von Jacob Ziv und Abraham Lempel erfunden und ist ein Verfahren zur Textkompression. Es benutzt dabei ein *adaptives Wörterbuch*, das heißt für jeden Eingabetext wird ein anderes Wörterbuch erzeugt. LZ78 unterteilt den Eingabetext in Faktoren. Dabei ist jeder Faktor der längste, gleiche Substring im bisher gelesenen Eingabetext plus das nächste Zeichen. Ein Faktor wird dabei als Tupel (x, c) dargestellt, wobei x der Index

des Prefix ist und c das zusätzliche Zeichen. Eine Einschränkung, wie weit der referenzierte Teilstring zurückliegen darf, gibt es hierbei nicht. Wenn p Faktoren erstellt wurden, lässt sich der Index mit $\lceil \log p \rceil$ Bits darstellen. In der Praxis kann dieses Wörterbuch nicht unendlich groß werden. Daher löscht man dieses Wörterbuch, falls kein Speicher mehr verfügbar ist, und arbeitet mit einem leeren Wörterbuch und dem restlichen Eingabetext weiter.
Eine effiziente Datenstruktur, um die Faktoren darzustellen, ist ein Trie bzw. hier der sog. LZ78-Trie. Jeder Knoten des LZ78-Tries enthält den Index des Faktors, den er repräsentiert. Hierbei ist $s_{Pfad} = s_{Faktor}$, wenn s_{Pfad} der String auf dem Pfad von der Wurzel bis zum Knoten ist und s_{Faktor} der durch den Faktor dargestellte Teilstring des Eingabetextes.[1, S.225]
Dieses Verfahren wird in Abbildung 5 nun am Beispiel von $T =$ `ananas$` verdeutlicht:

	1	2	3	4	5
Textteil	`a`	`n`	`an`	`as`	`$`
Faktorzerlegung	(-,`a`)	(-,`n`)	(1,`n`)	(1,`s`)	(-,`$`)

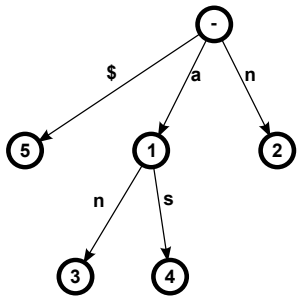


Abbildung 5: Links sieht man die Faktorisierung nach LZ78 und rechts den entsprechenden LZ78-Tree. Der i-te Faktor (x, s) wird zu einem neuen Knoten i mit x als Elternknoten und die Kante (x, i) wird mit s beschriftet.

3.2 LZ78 Variante

In dieser Arbeit wollen wir eine Variante (im Folgenden LZ78V genannt) zu LZ78 aus Kapitel 3.1 entwickeln. LZ78V unterteilt den Eingabetext auch in Faktoren. Jedoch kann ein Faktor mehr als nur das nächste Zeichen an den referenzierten Teilstring anhängen. Die Information, welche Zeichen dies sind, bezieht LZ78V aus dem Suffix-Tree. Bei unserem Beispieltext $T =$ `ananas$` folgt nach einen `'n'` immer ein `'a'`. Dies spiegelt sich im Suffix-Tree aus Abbildung 2 durch die Kante (0,7), die mit `'na'` beschriftet ist, wider. Mit LZ78V können wir nun beim ersten Lesen von `'n'` den Faktor (0, `'na'`) erstellen. Es ergibt sich die Faktorisierung aus Abbildung 6.

	1	2	3
Textteil	a	na	nas$
Faktorzerlegung	(-,a)	(-,na)	(2,s$)

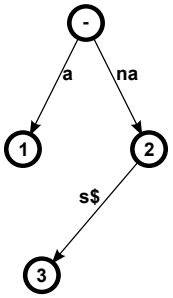


Abbildung 6: Links sieht man die Faktorisierung nach LZ78V und rechts den entsprechenden LZ78V-Tree. Der i-te Faktor (x, s) wird zu einem neuen Knoten i mit x als Elternknoten und die Kante (x, i) wird mit s beschriftet. Im Gegensatz zum LZ78-Tree kann s aber mehr als ein Zeichen beinhalten.

Den LZ78V-Tree können wir mit einem zweidimensionalen Array A repräsentieren. So wäre $A[i][1] = x$ und $A[i][2] = s$ der i-te Faktor (x, s) in der Array-Darstellung. Im Beispiel $T =$ `ananas$` ergibt sich:

	1	2	3
$A[1]$	-	-	2
$A[2]$	a	na	s$

LZ78V baut also wie LZ78 einen Tree auf. Vergleichen wir beide Trees aus unserem Beispiel, fällt auf, dass der LZ78V-Tree zwei Knoten weniger hat. Das heißt wir müssen weniger Informationen abgespeichert. Eine Kodierung des LZ78V-Trees entwickeln wir in einem späteren Abschnitt.
Gehen wir jedoch strikt nach dem oben beschriebenen Verfahren vor, so kann es auch zu einer schlechteren Faktorisierung als bei LZ78 kommen. Schlecht definieren wir hier über die Gesamtanzahl der Zeichen in der Faktorisierung. Je mehr Zeichen, desto schlechter ist die Faktorisierung. Betrachten wir beispielsweise den Text $T =$ `kananas$`. Lesen wir nun das `'k'` erstellt LZ78V direkt den Faktor $(-,$`kananas$`$)$ mit 8 Zeichen. Dies entsteht dadurch, dass sobald wir ein `'k'` lesen, LZ78V den kompletten String der Kante (0,7) des Suffix-Trees aus Abbildung 7 zu einem Faktor hinzufügt. Da `'k'` ein einzigartiges Zeichen in T ist, enthält der Faktor des gesamten Text T. In dieser Arbeit soll hierfür eine Lösung gefunden werden. Es wäre zum Beispiel denkbar, (x, s) mit $x \in \{1...z\} \cup \{-\}$ und $s \in \Sigma^+$ nur in die Faktorisierung aufzunehmen, wenn der entsprechende Knoten im Suffix-Tree kein Blatt ist. Ansonsten nehmen wir nur das erste Zeichen von s.
Am Beispiel von $T =$ `kananas$` ergibt somit folgeneder LZ78V-Tree in der Array-Darstellung:

	1	2	3	4	5
$A[1]$	-	-	-	3	-
$A[2]$	k	a	na	s	$

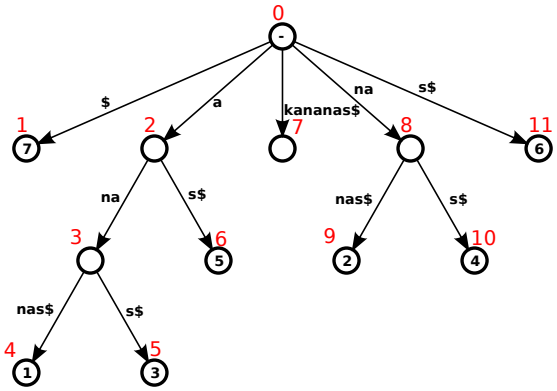


Abbildung 7: Diese Abbildung zeigt den Suffix-Tree zu $s =$ `kananas$`. Die Blätter sind hierbei nummeriert. Der String auf dem Pfad von der Wurzel zum Blatt i repräsentiert das Suffix $s[i..]$. Die roten Zahlen an den Knoten sind durch eine Pre-Order-Nummerierung entstanden und sind die IDs der Knoten.

Diese Faktorisierung hat insgesamt 6 Zeichen, ist also besser als die Faktorisierung nach dem strikten Verfahren.

4 Textkompression mit LZ78V

4.1 Faktorisierung

In diesem Kapitel entwickeln wir einen Algorithmus und die dazu notwendigen Datenstrukturen, basierend auf der Grundidee aus Abschnitt 3.2. Desweiteren definieren wir einen *Kodierer*, der die Faktorisierung kodiert und wir sie so effizient abspeichern können. Abschließend stellen wir einen *Dekodierer* vor, der aus einer Kodierung wieder eine Faktorisierung aufbaut.

Algorithm 1: LZ78V Faktorisierung mit Vektoren

```
Data: Eingabetext T
Result: Vector mit IDs, Repräsentation der Faktoren
konstruiere Suffix-Tree aus T;
erstelle Bit-Vektor used und Int-Vektor factors;
while Textende nicht erreicht do
    suche anhand des aktuellen Zeichens den nächsten Knoten im Suffix-Tree;
    if used[aktueller Knoten] then
        setze Textposition entsprechend der Zeichen an der Kante weiter;
        continue;
    end
    if aktueller Knoten ist ein Blatt then
        setze Textposition um 1 weiter;
    end
    else
        setze Textposition entsprechend der Zeichen an der Kante weiter;
    end
    füge die ID des aktuellen Knotens an factors an;
    markiere aktuellen Knoten in used als benutzt;
end
```

4.1.1 SDSL-lite

SDSL-lite ist eine C++11 Bibliothek, die speichereffiziente Datenstrukturen zur Verfügung stellt. Die benutzen Datenstrukturen dieser Bibliothek werden im Nachfolgenden an den passenden Stellen eingeführt.[6]

4.1.2 Suffix-Tree Konstruktion

Um aus dem Eingabetext einen Suffix-Tree zu konstruieren verwenden wir die Bibliothek SDSL-lite. Hiermit können wir einen sog. *Compressed Suffix Tree* (CST) erzeugen. Dieser hat zur Laufzeit einen Speicherbedarf von $3n$ + o(n) + |CSA| + |LCP| Bits .[6] Jeder Knoten besitzt eine ID i, die wir durch eine Anfrage an den Baum in konstanter Zeit erhalten. Sei x die Anzahl von Knoten im Suffix-Tree, dann gilt:

$$i \in [0..x]$$

4.1.3 Vektoren

Der Algorithmus benutzt drei Vektoren. Diese erstellen wir wieder mithilfe von SDSL-lite.

4.1.3.1 Bit-Vektor used Der Bit-Vektor *used* besteht aus x Bits, die mit 0 initialisiert werden. Dieser Vektor gibt an, ob ein Knoten mit der ID i schon als Faktor benutzt wird. Ist dies der Fall, so ist $used[i] = 1$. Dieser Vektor benötigt $64 * \lceil \frac{n}{64} + 1 \rceil$ Bits.[6]

4.1.3.2 Int-Vektor factors

Der Int-Vektor $factors$ ist zu Beginn leer. Wir füllen ihn im Laufe der Faktorisierung mit IDs von Knoten aus dem Suffix-Tree.

4.1.4 Abbruchbedingung

Der Algorithmus bricht dass ab wenn wir das Ende des Eingabetextes erreicht haben. Dies ist der Zeitpunkt, an dem die Faktorisierung abgeschlossen ist.

4.1.5 Faktoren erstellen

Der Startknoten jedes neuen Suchdurchlaufs ist die Wurzel des Suffix-Trees. In Zeile 4 des Algorithmus 1 suchen wir anhand des aktuellen Zeichens aus dem Text den nächsten Knoten k im Suffix-Tree. Diese Methode wird von SDSL-lite bereitgestellt und benötigt $O((t_{SA} + t_{SA^{-1}}) * \log \sigma + t_{LCP})$ Zeit. Hierzu sehen wir in einem späteren Abschnitt noch eine schnellere Alternative. In Zeile 5 testen wir, ob k schon als Faktor benutzt wird. Das bedeutet, wir haben den String auf dem Pfad bis k bereits durch einen Faktor dargestellt und können diesen nun referenzieren. Falls dies der Fall ist, setzen wir die Textposition um so viele Zeichen weiter, wie an der Kante zu k stehen und suchen nach dem nächsten Knoten. Sobald wir einen Knoten k gefunden haben, der noch nicht als Faktor benutzt wird, also für den $used[k] = 0$ gilt, testen wir in Zeile 9, ob der aktuelle Knoten ein Blatt ist. Falls ja, erhöhen wir die Textposition nur um 1, da nur das erste Zeichen der Kante zu k zum Faktor hinzugefügt wird. Ist k kein Blatt, setzen wir die Textposition um so viele Zeichen weiter, wie an der Kante zu k stehen. Außerdem fügen wir die ID von k an $factors$ an und markieren k in $used$ als benutzt.

4.2 Beispiel

In diesem Abschnitt betrachten wir die Faktorisierung beispielhaft an unserem Beispieltext $T =$ `ananas$`.
Der Suffix-Tree von T aus Abildung 2 hat 11 Knoten. Also ergibt sich nach der Initialisierung der Vektoren:

- $used$ = [0,0,0,0,0,0,0,0,0,0,0]
- $factors$ = []

Nun beginnen wir den Text zu lesen. Das erste Zeichen ist ein `'a'`. Benutzen wir ausgehend von der Wurzel im Suffix-Tree die entsprechende Kante, gelangen wir zum Knoten mit der ID 2. Dieser Knoten ist noch nicht benutzt, da $used[2] = 0$. Somit fügen wir 2 an $factors$ an, setzen $used[2] = 1$ und erhöhen die Textposition um 1. Es ergibt sich:

- $used$ = [0,0,1,0,0,0,0,0,0,0,0]
- $factors$ = [2]

Nächstes Zeichen: `'n'`
Gehen wir von der Wurzel des Suffix-Trees die entsprechende Kante, gelangen wir zum Knoten mit der ID 7, der auch noch nicht benutzt ist. Die Textposition erhöhen wir um 2, da an der Kante zwei Zeichen stehen.

- $used$ = [0,0,1,0,0,0,0,1,0,0,0]
- $factors$ = [2,7]

Nächstes Zeichen: `'n'`
Nun gelangen wir durch benutzen der entsprechenden Kante des Suffix-Trees von der Wurzel wieder zum Knoten mit der ID 7. Diesen haben wir im letzen Durchlauf als benutzt markiert. Das heißt wir erhöhen die Textposition um 2 und suchen anhand des nächsten Zeichens `'s'` den nächsten Knoten im Suffix-Tree. Das ist der Knoten mit der ID 9. Allerdings ist dieser Knoten ein Blatt, das heißt wir erhöhen die Textposition nur um 1. Wir fügen die ID 9 an $factors$ an und markieren den Knoten als benutzt.

- $used$ = [0,0,1,0,0,0,0,1,0,1,0]
- $factors$ = [2,7,9]

Nächstes Zeichen: `'$'` Wir gelangen zum Knoten mit der ID 1. Dieser ist noch nicht benutzt, also fügen wir die ID 1 an $factors$ an und markieren ihn als benutzt. Die Textposition erhöhen wir um 1, da der Knoten ein Blatt ist.

- $used$ = [0,1,1,0,0,0,0,1,0,1,0]
- $factors$ = [2,7,9,1]

Das Ende des Textes ist erreicht. Somit ist die Abbruchbedingung erfüllt und die Faktorisierung abgeschlossen.

Literatur

[1] Timothy C. Bell, John G. Cleary, and Ian H. Witten. *Text Compression*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[2] Bild von einem Klappentelegraphen. https://cms.sachsen.schule/typoecke2/typo-experimente/informationuebertragung-mit-dem-klappentelegraph/was-ist-ein-klappentelegraph/. abgerufen am: 24.08.2016.

[3] Braille-Aphabeth. http://www.pharmabraille.com/pharmaceutical-braille/the-braille-alphabet/. abgerufen am: 24.08.2016.

[4] Visualisierung von Suffix - Trees. http://visualgo.net/suffixtree.html. abgerufen am: 01.08.2016.

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

[6] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.