technische universität
dortmund

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

Textkompression mithilfe einer Variante von LZ78

Exposee

Florian Kleine
Matrikelnummer: 157020

27. Mai 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.

2 Entwurf einer Gliederung

- Einleitung
- Theoretische Grundlagen
- Entworfene Algorithmen
- Implementierung
- Praktische Tests
- Fazit

3 Leseprobe

3.1 Theoretische Grundlagen

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

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

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

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

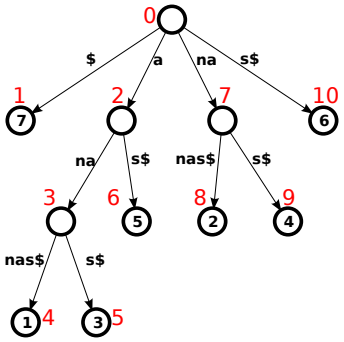
3.4 Definition (Teilstring) Für $x, y \in \mathbb{N}$ und $1 \leq x < y \leq n$ sei $s[x, y]$ die Zeichenfolge vom x-ten bis zum y-ten Zeichen aus s. s[x] und s[y] einschließlich.

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

3.6 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.

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



[1]

Abbildung 1: 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.

3.2 Idee der LZ78 Variante

3.2.1 LZ78

LZ78 wurde 1978 von Jacob Ziv und Abraham Lempel erfunden und ist ein Verfahren zur Datenkompression. Der (naive) Algorithmus durchläuft den String bzw. den Text T und ersetzt Redundanzen durch Verweise auf das längste vorherige Vorkommen des selben Teils und hängt das nächste Zeichen an diesen Verweis an. Dadurch wird ein Baum erzeugt, der sog. LZ78-Trie. Dieses Verfahren wird nun am Beispiel von $T =$ `ananas$` verdeutlicht.

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

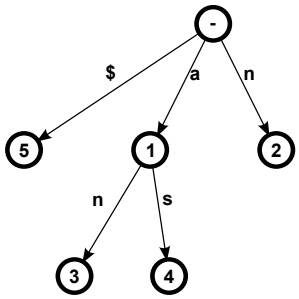


Abbildung 2: Links sieht man die Faktorisierung nach LZ78 und rechts den entsprechenden LZ78-Tree. Der i-te Faktor (x, s) wird zu einem neuen Knoten i mit x als Elternknoten und die Kante (x, i) wird mit s beschriftet.

Ein *Faktor* ist ein Tupel (x, s) mit $x \in \{1..z\} \cup \{-\}$ und $s \in \Sigma$. x bezeichnet hierbei den Elternknoten im LZ78-Tree und s die Kantenbeschriftung der Kante von x zum neu entstehenden Knoten. z ist die Anzahl der Faktoren. Die Zahlen in den Knoten stehen für die Faktoren.

3.2.2 LZ78 Variante

Im Gegensatz zu LZ78 kann diese Variante (im Folgenden LZ78V genannt) auch mehr Zeichen an eine Ersetzung anhängen. Im Beispieltext $T =$ `ananas$` folgt nach einem `'n'` immer ein `'a'`. Dies spiegelt sich im Suffix-Tree durch die Kantenbeschriftung `'na'` wider. Mit LZ78V können wir nun beim ersten Lesen von `'n'` den Faktor (-,na) erstellen, denn nach `'n'` kann nichts anderes kommen. Es ergibt sich folgende Faktorzerlegung:

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

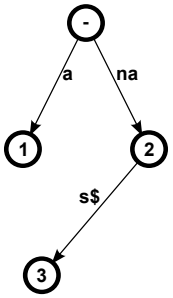


Abbildung 3: Links sieht man die Faktorisierung nach LZ78V und rechts den entsprechenden LZ78V-Tree. Der i-te Faktor (x, s) wird zu einem neuen Knoten i mit x als Elternknoten und die Kante (x, i) wird mit s beschriftet. Im Gegensatz zum LZ78-Tree kann s aber mehr als ein Zeichen beinhalten.

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$

Gehen wir jedoch strikt nach diesem Verfahren vor, kann es aber auch zu einer schlechteren Faktorisierung kommen. Schlecht definieren wir hier über die Gesamtanzahl der Buchstaben in der Faktorisierung. Je mehr Buchstaben, desto schlechter ist die Faktorisierung. Beispielweise beim Text $T =$ `abrakabrabra$` würde LZ78V den Text in 3 Faktoren zerlegen: (-,a)(-,bra)(-,kabrabra) mit insgesamt 12 Buchstaben.
Dies entsteht dadurch, dass sobald man ein `'k'` liest, die restlichen Zeichen eindeutig durch den Suffix-Tree bestimmt werden können und somit zu einem Faktor zusammengefasst werden. 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 mindestens 2 Kinder hat. Ansonsten nehmen wir nur das erste Zeichen von s.
Am Beispiel $T =$ `abrakabrabra$` ergibt sich dann:
(-,a)(-,bra)(-,k)(1,bra)(2,$) mit insgesamt 9 Buchstaben.

3.3 Idee des Algorithmus

Algorithm 1 Faktorisierung von Text T nach LZ78V mit Vektoren

```
ST = constructST(T)                                   // mithilfe von sdsl
x = ST.size()
U = bit_vector(x, 0)                                  // Bitvektor mit x Einträgen, mit 0 initialisiert
F = int_vector(0, 0, ⌊log x⌋ + 1)                     // leerer Vektor mit ⌊log x⌋ + 1 Bits für jeden Eintrag
U[ST.root().id()] = 1                                 // markiere die Wurzel als benutzt
current_node = ST.root()                              // Startknoten ist die Wurzel von ST
t = 0
while not current_node.is_leaf() do
    current_node = ST.nextNode(current_node, T[t])    // nächsten Knoten in ST finden
    t = t + ST.string_on_edge(current_node).length()  // Textposition entsprechend verändern
    if U[current_node.id()] == 1 then
        continue                                      // suche mit dem nächsten Zeichen den nächsten Knoten
    else
        F.add(current_node.id())                      // ID des aktuellen Knotens an F anhängen
        U[current_node] = 1                           // aktuellen Knoten als benutzt markieren
        current_node = ST.root()                      // beginne im nächsten Durchlauf wieder an der Wurzel
    end if
end while
```

Um einen Text T, wie in Kapitel 3.2.2 in Faktoren zu zerlegen, konstruieren wir uns zuerst den Suffix-Tree ST. Hierzu wird die C++ Bibliothek *sdsl* verwendet. Mit Hilfe dieser Bibliothek lassen sich sog. *Compressed Suffix Trees* (CST) leicht aus Texten erzeugen. Außerdem besitzt jeder Knoten in ST eine eindeutige ID aus dem Bereich 0 bis $x-1$, wenn x die Anzahl der Knoten in ST ist. Nachdem wir ST erstellt haben, erzeugen wir wieder mit Hilfe von *sdsl* einen Bit-Vektor U mit x Einträgen und initialisieren ihn mit 0. Der Vektor U speichert für jeden Knoten v mit der ID i, ob er schon als Faktor benutzt wird. Falls dieser Knoten v bereits als Faktor benutzt wird, ist U[i] = 1. Wir setzen U[id(root())] = 1, da die Wurzel nie als Faktor verwendet wird. Außerdem erzeugen wir mit Hilfe von *sdsl* einen sog. *int_vector* F, der für jeden Eintrag $\lfloor \log x \rfloor + 1$ Bits belegt. Nun beginnen wir den Text zu lesen. Der aktuelle Knoten *current_node* in ST ist die Wurzel. Anhand des ersten Zeichens von T können wir bestimmen, welche Kante man nehmen muss und aktualisieren *current_node* mit dem entsprechenden Knoten. Außerdem setzen wir die aktuelle Textposition t um so viele Zeichen weiter wie an der Kante zu *current_node* stehen. Wir testen, ob *current_node* schon als Faktor benutzt wird. Das heißt wir testen, ob U[*current_node*] = 1 ist. Wenn ja, suchen wir uns anhand des nächsten Zeichens in T den nächsten Knoten in ST und verfahren mit diesem Knoten analog. Wenn nein, setzen wir U[*current_node.id*()] = 1 und hängen *current_node.id*() an F an. Dann beginnen

wir mit dem nächsten Zeichen von T an der Wurzel von ST erneut. Dies wiederholen wir solange, bis der *current_node* ein Blatt von ST ist. Dann können wir die Faktorisierung beenden, da der Text zu Ende ist.

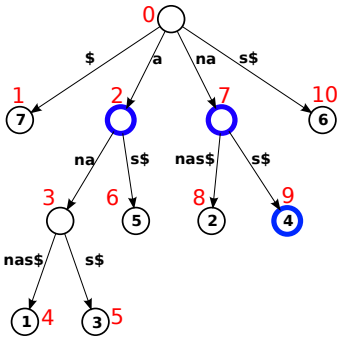


Abbildung 4: Die blauen, dicken Kreise visualisieren die Knoten, die wir in F als Faktoren benutzen. Sonstige Nummerierungen entsprechen den Angaben aus Abbildung 1.

Im Beispiel von $T =$ `ananas$` und den IDs aus Abbildung 4 ergibt sich:
U = [1,0,1,0,0,0,0,1,0,1,0] und F = [2,7,9].

Der beschriebene Algorithmus hat lineare Laufzeit. Im Folgenden sei n die Länge des Textes T, x die Anzahl der Knoten in ST und y die Anzahl der Faktoren in F. Der Speicherbedarf zur Laufzeit beträgt $3n$ + o(n) + |CSA| + |LCP| Bits für ST[3], x Bits für den Bit-Vektor U und $y * (\lfloor \log(x) \rfloor + 1)$ Bits für F.
Also insgesamt $3n$ + o(n) + |CSA| + |LCP| + x + y * $\lfloor \log x \rfloor$ + y Bits.
Um die Tabelle A nun mit Faktoren zu füllen, erstellen wir uns mit *sdsl* einen int_vector N mit z Einträgen. Jeder Eintrag belegt $\lfloor \log z \rfloor + 1$ Bits, also insgesamt belegt der Vektor N $z * \lfloor \log z \rfloor + z$ Bits. Dieser Vektor soll für jede ID aus F speichern, den wievielten Faktor der entsprechende Knoten repräsentiert. Mithilfe von *sdsl* bauen wir auf U eine *rank* - Struktur auf. Diese hilft uns N zu füllen. Dafür laufen wir F einmal von links nach rechts durch und setzen $N[U.rank[F[j]]] = j$ für jeden Eintrag in F an der Position j. Im Beispiel von $T =$ `ananas$` ist $N = [1, 2, 3]$. Den Speicher von F können wir jetzt freigeben. Nun lässt sich mithilfe von U, N und ST die Tabelle A in linearer Zeit füllen. Dazu laufen wir U einmal von links nach rechts durch und erstellen für jeden Eintrag in U mit dem Index i, der ungleich 0 ist, einen Faktor in A an der Stelle $N[U.rank[i]]$. Ist der Vaterknoten p des entsprechenden Knoten in ST die Wurzel verweist der Faktor auf 0. Ansonsten auf $N[U.rank[p.id()]]$. Außerdem enthält der Faktor den Text, der an der Kante zu diesem Knoten steht.

Algorithm 2 Faktorisierung von Text T nach LZ78V mit Baumstruktur

```
ST = constructST(T)                                   // mithilfe von sdsl
B.root = node(0, ST.root().id())
ST_current_node = ST.root()
B_current_node = B.root()
t = 0, k = 0          // t ist die aktelle Textposition, k die aktuelle Anzahl der Faktoren
while not ST_current_node.isLeaf() do
  ST_current_node = ST.nextNode(ST_current_node, T[t])        // nächster Knoten in ST
  if B_current_node.dict.exist(T[t]) then
    B_current_node = B_current_node.dict.find(T[t])           // gehe gleichen Pfad wie in ST
    t = t + ST.string_on_edge(ST_current_node, T[t]) + 1
    continue
  else
    new_node = node(k, ST_current_node.id())                  // es muss ein Knoten eingefügt werden
    B_current_node.dict.insert(T[t], *new_node)
    t = t + ST.string_on_edge(ST_current_node, T[t]) + 1
    B_current_node = B.root()          // beginne wieder an beiden Wuzeln mit dem nächsten Zeichen
    ST_current_node = ST.root()
    k = k + 1
  end if
end while
```

Eine Alternative zu Algorithmus 1 ist, dass wir während der Faktorisierung den LZ78V-Tree aufbauen. Hierzu erstellen wir einen Baum B, der nur die Wurzel als Knoten enthält. Ein Knoten repräsentiert ein Tupel (a, b), wobei a die ID des Knotens ist und angibt, als wievielter dieser Knoten hinzugefügt wurde. b ist die ID des entsprechenden Knotens in ST. Außerdem enthält jeder Knoten ein dynamisches Dictionary, das Zeichen mit Pointern verknüpft. Diese Pointer zeigen auf die Kindknoten eines Knotens. Das Dictionary benötigt $s_{dict}(m)$ zum Speichern von m Elementen. Die Laufzeit eines Look-Ups beträgt $t_{lookup}(m)$ und das Einfügen $t_{insert}(m)$. Während wir, wie in Algorithmus 1 beschrieben, anhand der gelesenen Zeichen ST durchlaufen, versuchen wir den gleichen Pfad auch in B zu gehen. Kommen wir an den Punkt, wo dies in B nicht möglich ist, erstellen wir den entsprechenden Knoten in B und beginnen in beiden Bäumen mit dem nächsten Zeichen von vorn. Sind wir in ST an einem Blattknoten angelangt, kann man die Faktorisierung beenden, da der Text zu Ende ist. So ergibt sich im Beispiel $T =$ `ananas$` der Baum aus Abbildung 5. Diese Alternative ist besonders gut, wenn die Anzahl z der Faktoren klein ist. Denn der Speicherbedarf des Baums beträgt $z * (\lfloor \log x \rfloor + \lfloor \log z \rfloor + 2) + \sum_{k=1}^{z} s_{dict}^{(k)}$ Bits und die Laufzeit ist $O(n * t_{lookup}(z))$. Hinzu kommt noch der Speicherbedarf von ST. Mit B und ST können wir auch in linearer Zeit die Tabelle A füllen. Wir durchlaufen den Baum z.B. in Pre-Order. Für jeden Knoten, außer der Wurzel, erstellen wir in A an der Position a

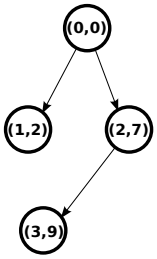


Abbildung 5: Dies ist der Baum B zu $T =$ `ananas$` nach der Faktorisierung unter Berücksichtigung der IDs aus Abbildung 4.

einen Faktor. Dieser verweist auf die ID des Elternknotens in B und enthält den Text, der in ST an der Kante zum Knoten mit der ID b steht.

3.4 Mögliche Kodierung

Eine Möglichkeit die Tabelle A zu kodieren wäre die Huffman Kodierung. Diese Kodierung ordnet jedem Symbol einen präfixfreien Code zu. Dazu ordnen wir die Symbole nach der Häufigkeit ihres Auftretens und erstellen für jedes Symbol einen Knoten, der die Häufigkeit des Auftretens enthält. Nun fügen wir zwei Knoten (bzw. Teilbäume) mit kleinster Häufigkeit zu einem neuen Teilbaum zusammen. Die Wurzel enthält dabei die Summe der beiden Häufigkeiten der Teilbäume. Dies wiederholen wir solange, bis nur noch ein Baum übrig ist. Für jedes Symbol können wir nun auf dem Pfad von der Wurzel zum Blattknoten(Blätter enthalten die Symbole) einen Code ablesen: 0 für das linke Kind und 1 für das rechte Kind.
In der Kodierung der Tabelle A müssen wir noch zwischen Verweisen und Zeichen des Textes unterscheiden. Daher markieren wir die Verweise mit einen *. So ergibt sich beim Beispiel $T =$ `ananas$` der zu kodierende Text: $0^*a0^*na2^*s\$$.

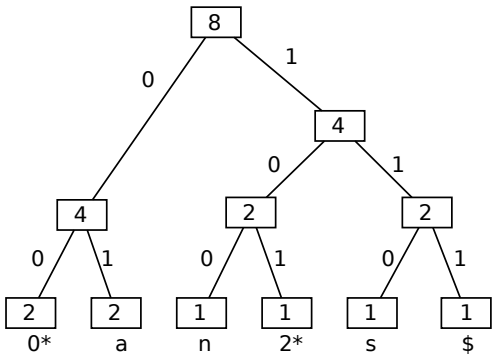


Abbildung 6: Diese Abbildung zeigt einen möglichen Huffman-Tree zum Text $0^*a0^*na2^*s\$$. Der Code für jedes Symbol ist der String auf dem Pfad bis zum Blatt des Symbols.

Mit dem Huffman-Tree aus Abbildung 6 ergibt sich folgende Kodierung für $T =$ `ananas$`:

00010010001101110111. Diese Art der Kodierung hat den Vorteil, dass Symbole, die häufig vorkommen, einen kürzeren Code bekommen als Symbole mit niedriger Häufigkeit.
Um den Huffman-Tree abzuspeichern, starten wir bei der Wurzel und gehen wie folgt vor:

- bei einem Knoten speichere 1 und das Zeichen am Blatt
- sonst speichere 0 und verfahre mit den Kindknoten auf die gleiche Weise (erst links, dann rechts)

So ergibt sich für den Huffman-Tree aus dem Beispiel: $0010^*1a001n12^*01s1\$$

4 Ziele der Arbeit

In dieser Arbeit soll eine Variante des LZ78-Verfahrens entwickelt und in C++, unter der Verwendung der C++ Bibliothek *sdsl*, implementiert werden. Dieses soll Texte und Dateien besser komprimieren als LZ78. Außerdem soll ein gutes Verfahren gefunden oder selbst entwickelt werden, um die Faktorisierung nach dieser LZ78-Variante zu kodieren. Denn nur so lässt sich diese Kompression mit Kompressionen anderer Verfahren, wie z.B. gzip oder 7zip, vergleichen. Kriterien für den Vergleich sind die relative Anzahl von Bits der Ausgabe und die Geschwindigkeit beim Komprimieren bzw. Dekomprimieren. Testdaten entnehme ich hierbei aus [12]. Enthalten sind Source-Code in C und Java, Proteinsequenzen, DNA, englische Texte und XML-Dateien mit verschiedenen Alphabetgrößen.

5 Zeitplan

Meilensteine:

- bis April:
  - Theoretische Grundlagen zu großen Teilen fertig (bei Bedarf später ergänzen)
  - Programmbibliotheken festlegen
- Mitte April: Algorithmus implementiert
- ab Mai:
  - Optimierung von Speicherbedarf und Geschwindigkeit
  - Mit anderen Kompressionsverfahren vergleichen und Ergebnisse dokumentieren

Außerdem finden nach Absprache alle 2 Wochen Treffen mit dem Betreuer statt um eventuell auftretende Fragen zu klären und den aktuellen Stand der Abschlussarbeit sicherzustellen.

Literatur

[1] Visualisierung von Suffix - Trees. `http://visualgo.net/suffixtree.html`. abgerufen am: 10.12.2015.

[2] Dominik Köppl, Johannes Fischer, Tomohiro I. Lempel Ziv Computation in Small Space(LZ-CISS). *Combinatorial Pattern Matching*, Seiten 172–184, 2015.

[3] Simon Gog. sdsl Cheet Sheet. `http://simongog.github.io/assets/data/sdsl-cheatsheet.pdf` abgerufen am: 31.01.2016.

[4] R. Nigel Horspool. Improving LZW. In *Proceedings of the IEEE Data Compression Conference, DCC 1991, Snowbird, Utah, April 8-11, 1991.*, Seiten 332–341, 1991.

[5] Mamta Sharma. Compression using huffman coding. *IJCSNS International Journal of Computer Science and Network Security*, 10:133–141, 2010.

[6] Yuto Nakashima, Tomohiro I, Shunsuke Inenaga, Hideo Bannai, and Masayuki Takeda. Constructing LZ78 tries and position heaps in linear time for large alphabets. *CoRR*, abs/1501.06619, 2015.

[7] Dominik Köppl and Kunihiko Sadakane. Lempel Ziv Computation in Compressed Space (LZ-CICS). *CoRR*, abs/1510.02882, 2015.

[8] Pang Ko. *Suffix Trees and Suffix Arrays in Primary and Secondary Storage*. PhD thesis, Ames, IA, USA, 2007.

[9] Keisuke Goto and Hideo Bannai. Simpler and faster lempel ziv factorization. *CoRR*, abs/1211.3642, 2012.

[10] Simon Gog. Text indexing: Lecture 2. Lecture notes. `http://algo2.iti.kit.edu/gog/2_text_indexing.pdf` abgerufen am: 24.01.2016.

[11] Guy Blelloch. Introduction to data compression. Lecture notes. `https://www.cs.cmu.edu/~guyb/realworld/compression.pdf` abgerufen am: 24.01.2016

[12] P. Ferragina and G. Navarro. Pizza & chili corpus - compressed indexes and their testbeds. `http://pizzachili.dcc.uchile.cl/texts.html` Letztes Update: September, 2005