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

21. März 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.

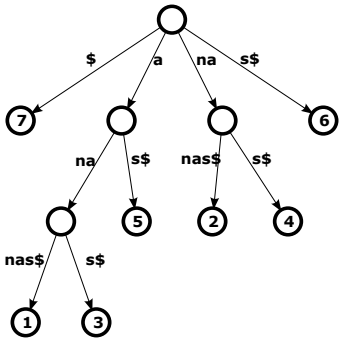
3.4 Definition (Teilstring) Für $x, y \in \mathbb{N}$ und $1 \leq x < y \leq n$ sei $s[x, y]$ die Zeichenfolge vom x-ten bis zum y-ten Zeichen aus s. s[x] und s[y] einschließlich.

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 $s[i..]$.

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.2

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

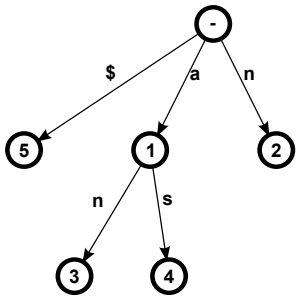


Abbildung 2: Links sieht man die Faktorisierung nach LZ78 und rechts den entsprechenden LZ78-Tree. Der i-te Faktor (x, s) wird zu einem neuen Knoten i mit x als Elternknoten und die Kante (x, i) wird mit s beschriftet.

Ein *Faktor* ist ein Tupel (x, s) mit $x \in \{1..n\} \cup \{-\}$ und $s \in \Sigma$. x bezeichnet hierbei den Elternknoten im LZ78-Tree und s die Kantenbeschriftung der Kante von x zum neu entstehenden Knoten. Die Zahlen in den Knoten stehen für die Faktoren. [2, 8-10]

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$`
Faktorzerlegung	(-,`a`)	(-,`na`)	(2,`s$`)

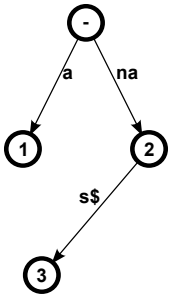


Abbildung 3: Links sieht man die Faktorisierung nach LZ78V und rechts den entsprechenden LZ78V-Tree. Der i-te Faktor (x, s) wird zu einem neuen Knoten i mit x als Elternknoten und die Kante (x, i) wird mit s beschriftet. Im Gegensatz zum LZ78-Tree kann s aber nun mehr als ein Zeichen beinhalten.

Den LZ78V-Tree können wir mit einem zweidimensionalen Array A implementieren. 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 $x \in \{1...n\} \cup \{-\}$ und $s \in \Sigma^+$ nur in die Faktorisierung aufzunehmen, wenn s mindestens zweimal im Text vorkommt. Dazu müssten wir sicherstellen, dass s ein innerer Knoten des Suffix-Tree ist. 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 factorize(T)

ST = construct_ST(T)
U = bit_vector(ST.size(), 0)
F = int_vector(0, 0, $\lfloor \log ST.size() \rfloor + 1$)
current_node = ST.root()
i = 0

Um einen Text T, wie oben verdeutlicht, in Faktoren zu zerlegen, konstruieren wir uns zuerst den Suffix-Tree. 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 im CST eine eindeutige ID aus dem Bereich 0 bis $n-1$, wenn n die Anzahl der Knoten im CST ist. Eine Eigenschaft, die man sich hier zunutze machen kann. Nachdem man den CST erstellt hat, erzeugt man wieder mit Hilfe von *sdsl* einen Bit-Vektor U mit n Einträgen und initialisiert ihn mit 0. Der Vektor U speichert für jeden Knoten v mit der ID i, ob er schon als Faktor benutzt wird. Also falls dieser Knoten v bereits als Faktor benutzt wird, ist U[i] = 1. Man setzt U[id(root())] = 1, da die Wurzel nie als Faktor verwendet wird. Außerdem erzeugt man sich mit Hilfe von *sdsl* einen sog. *int_vector* F, der für jeden Eintrag $\lfloor \log n \rfloor + 1$ Bits belegt. Nun beginnt man den Text zu lesen. Der aktuelle Knoten v im CST ist die Wurzel. Anhand des ersten Zeichens von T kann man bestimmen, welche Kante man nehmen muss und aktualisiert v mit dem entsprechenden Knoten. Außerdem setzt man die aktuelle Textposition um so viele Zeichen weiter wie an der Kante zu v stehen. Ist v ein Blatt im CST, setzt man
U[id(v)] = 1 und hängt id(v) an das Ende von F. Man kann die Faktorisierung beenden, da der Text zu Ende ist. Ist v kein Blatt, testet man U[id(v)] != 0. Falls das zutrifft, sucht man sich anhand des nächsten Zeichens den nächsten Knoten und verfährt dort analog. Falls das nicht zutrifft, setzt man U[id(v)] = 1 und hängt id(v) an das Ende von F und beginnt mit dem nächsten Zeichen des Textes von vorn.

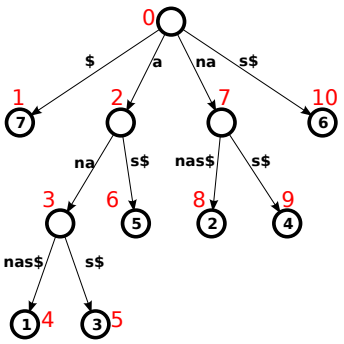


Abbildung 4: Diese Abbildung zeigt nocheinmal den Suffix-Tree zu T = `ananas$`. Die roten Zahlen an den Knoten ist eine Pre-Order-Nummerierung und stellt beispielhaft die IDs der Knoten dar.

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 im CST und y die Anzahl der Faktoren in F. Der Speicherbedarf zur Laufzeit beträgt $3n$ + o(n) + |CSA| + |LCP| Bits für den CST[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, erstellt man sich mit *sdsl* einen int_vector N mit m Einträgen, wenn m die größte ID in F ist und initialisiert ihn mit 0. Jeder Eintrag belegt $\lfloor \log z \rfloor + 1$ Bits, also insgesamt belegt der Vektor N $m * \lfloor \log z \rfloor + m$ Bits. Dieser Vektor soll für jede ID aus F speichern, den wievielten Faktor der entsprechende Knoten repräsentiert. N lässt sich in linearer Zeit mit F füllen. Dafür läuft man F einmal von links nach rechts durch und für jeden Eintrag in F an der Position j setzt man $N[F[j]] = j + 1$. Den Speicher des Vektors U kann man sogar schon vor dem Erstellen von N freigeben, den Speicher von F erst nach dem Befüllen von N. Nun lässt sich mithilfe von N und dem CST die Tabelle A in linearer Zeit füllen. Denn wird ein Knoten des CST mit der ID i als k-ter Faktor benutzt, so ist $N[i] = k$. Man erstellt in A für jeden Eintrag in N, der ungleich 0 ist, an der in N angegeben Stelle einen Eintrag, der auf $N[p]$ verweist, wenn p die ID des Vaterknotens des entsprechenden Knoten ist, und den Text enthält, der an der Kante zu diesem Knoten steht. Im Beispiel von $T =$ `ananas$` ist $N =$ [0,0,1,0,0,0,0,2,0,3].

Eine Alternative zu dem oben beschriebenen Algorithmus ist, dass man sich während der Faktorisierung den LZ78V-Tree aufbaut. Hierzu erstellt man sich einen Baum B, der nur die Wurzel als Knoten enthält. Ein Knoten enthält ein Tupel (a, b), wobei a die ID des Knotens ist und angibt, als wievielter dieser Knoten hinzugefügt wurde und b die entsprechende ID im CST. Außerdem enthält jeder Knoten eine Menge von Pointern auf die Kindknoten. Während man wie oben beschrieben anhand der gelesenen Zeichen den CST durchläuft, versucht man den gleichen Pfad auch in B zu gehen. Kommt man an den Punkt, wo dies in B nicht möglich ist, erstellt man sich den entsprechenden Knoten in B und beginnt in beiden Bäumen mit dem nächsten Zeichen von vorn. Ist man im CST 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 $z * (\lfloor \log n \rfloor + \lfloor \log z \rfloor + 2)$ Bits und die Laufzeit ist linear. Hinzu kommt noch der Speicherbedarf des CST. Mit B und dem CST kann man auch in linearer Zeit die Tabelle A füllen. Man durchläuft den Baum und für jeden Knoten außer der Wurzel mit dem Tupel (a, b) erstellt man in A an der Position a einen Eintrag der auf die ID des Elternknotens in B verweist und den Text enthält, der im CST an der Kante zum Knoten mit der ID b steht.

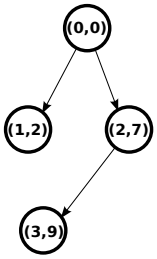


Abbildung 5: Dies ist der Baum B zu $T =$ `ananas$` nach der Faktorisierung unter Berücksichtigung der IDs aus Abbildung 4.

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 werden die Symbole nach der Häufigkeit ihres Auftretens geordnet und man erstellt für jedes Symbol einen Knoten, der die Häufigkeit des Auftretens enthält. Nun fügt man zwei Knoten (bzw. Teilbäume) zu einem neuen Teilbaum zusammen. Die Wurzel enthält dabei die Summe der beiden Häufigkeiten der Teilbäume. Dies wird solange wiederholt, bis nur noch ein Baum übrig ist. Für jedes Symbol kann man 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 muss man noch zwischen Verweisen und Text unterscheiden. Daher sind die Verweise mit einen * markiert. So ergibt sich beim Beispiel $T =$ `ananas$` der zu kodierende Text: $0^*a0^*na2^*s\$$.

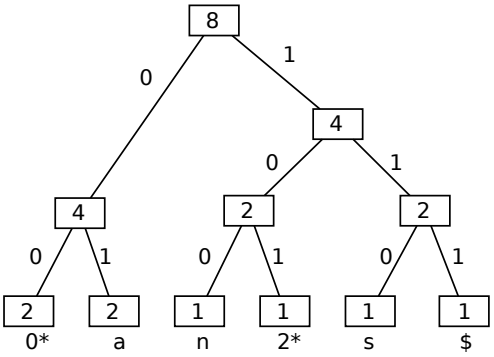


Abbildung 6: Diese Abbildung zeigt einen möglichen Huffman-Tree zum Text $0^*a0^*na2^*s\$$. Der Code für jedes Symbol ist der String auf dem Pfad bis zum Blatt des Symbols.

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.

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 März:
  - Theoretische Grundlagen zu großen Teilen fertig (bei Bedarf später ergänzen)
  - Programmbibliotheken festlegen
- Mitte März: Algorithmus implementiert
- ab April:
  - 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. AAI3274885.

[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