technische universität
dortmund

Bachelorarbeit

Textkompression mithilfe einer Variante von LZ78

Florian Kleine
Oktober 2016

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

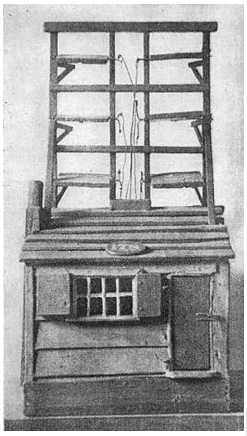
Inhaltsverzeichnis

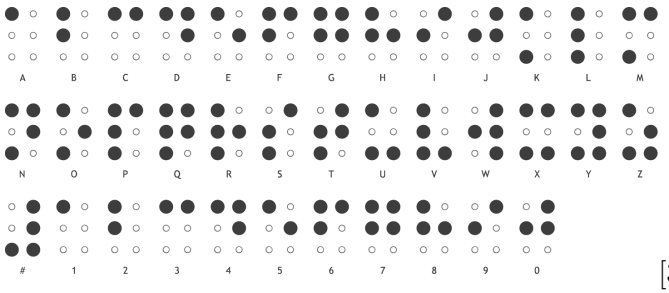
1 Einleitung

Die 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*, Abbildung 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-Alphabets (s. Abbildung 1) 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-Alphabet.

2 Theoretische Grundlagen

Im Folgenden werden Datenstrukturen, Operationen und Methoden 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 $s[x..] = s[x, n]$.

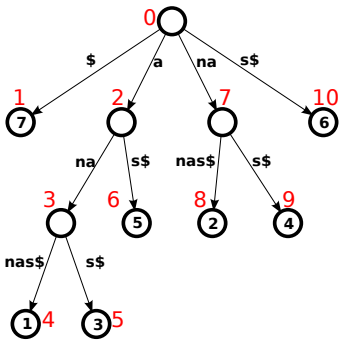
2.2 Suffix-Tree

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. Im Folgenden nennen wir diese roten Zahlen IDs, da wir einen Knoten an seiner ID eindeutig identifizieren können.

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[5] 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 (ausschließlich des gerade zu berechnenden Faktors) 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 abstrakte 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 der String auf dem Pfad von der Wurzel bis zum Knoten gleich dem Teilstring des Eingabetextes, der durch den Faktor dargestellt wird. [1, S.225]
Dieses Verfahren wird in Abbildung 3 nun am Beispiel von $T =$ `ananas$` verdeutlicht:

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

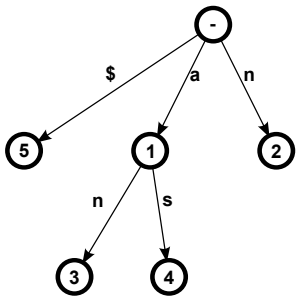


Abbildung 3: Links sieht man die Faktorisierung nach LZ78 und rechts den entsprechenden LZ78-Tree. Der i-te Faktor (x, s) wird zu einem neuen Knoten i mit x als Elternknoten und die Kante (x, i) wird mit s beschriftet. Die Wurzel des Baumes beschriften wir mit '-', da sie keinen Faktor darstellt. Ein Faktor (-,s) ist somit ein Kindknoten der Wurzel.

Den LZ78-Trie können wir zum Beispiel mithilfe [6, Kapitel 6.3] implementieren. Damit lässt sich der LZ78-Trie in $O(n)$ Zeit konstruieren und benötigt $2z + z \log \sigma$ Bits.
Die Berechnung einer LZ78 Faktorisierung benötigt jedoch viele Hilfsdatenstrukturen, zum Beispiel müssen wir speichern, bis zu welchem Buchstaben einer Kante die Faktorisierung fortgeschritten ist. An dieser Stelle zeigt sich der Vorteil der LZ78 Variante. Diese können wir einfacher berechnen, da wir zum Beispiel die Informationen über den Kantenfortschritt nicht speichern müssen.

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 4.

Den LZ78V-Tree können wir mit zwei Arrays A_1 und A_2 repräsentieren. A_1 speichert hierbei Referenzen auf vorherige Faktoren bzw. '-' für keine Referenz. A_2 speichert die Strings, die ein Faktor an den referenzierten Faktor anhängt. So wäre $A_1[i] = x$ und $A_2[i] = s$ der i-te Faktor (x, s) in der Array-Darstellung, mit $x \in \{1..z\} \cup \{-\}$ und $s \in \Sigma^+$,

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

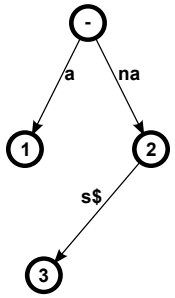


Abbildung 4: Links sieht man die Faktorisierung nach LZ78V und rechts den entsprechenden LZ78V-Tree. Der i-te Faktor (x, s) wird zu einem neuen Knoten i mit x als Elternknoten und die Kante (x, i) wird mit s beschriftet. Im Gegensatz zum LZ78-Tree kann s aber mehr als ein Zeichen beinhalten.

wenn z die Anzahl der Faktoren ist. 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 in A_2, 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 5 zu einem Faktor hinzufügt. Da `'k'` ein einzigartiges Zeichen in T ist, enthält der Faktor des gesamten Text T. Eine Lösung zum Beispiel ist, (x, s) 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 sich somit folgeneder LZ78V-Tree in der Array-Darstellung:

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

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

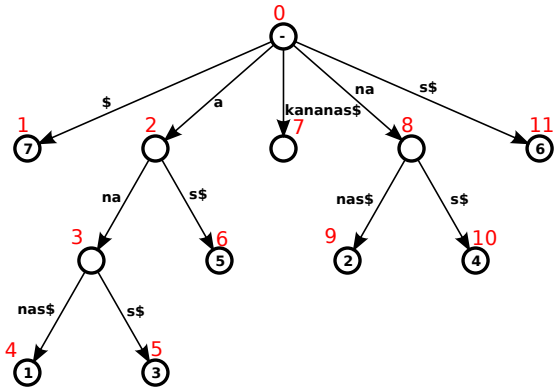


Abbildung 5: Diese Abbildung zeigt den Suffix-Tree zu $s =$ `kananas$`. Die Blätter sind hierbei nummeriert. Der String auf dem Pfad von der Wurzel zum Blatt i repräsentiert das Suffix $s[i..]$. Die roten Zahlen an den Knoten sind durch eine Pre-Order-Nummerierung entstanden und sind die IDs der Knoten.

4 Faktorisierung mit LZ78V

In diesem Kapitel entwickeln wir einen Algorithmus und die dazu notwendigen Datenstrukturen, basierend auf der Grundidee aus Abschnitt 3.2. Desweiteren definieren wir in Abschnitt 5 einen *Kodierer*, der die Faktorisierung kodiert und wir sie so effizient abspeichern können. Abschließend stellen wir einen *Dekodierer* vor, der aus einer Kodierung wieder die ursprüngliche Faktorisierung herstellt.

4.1 SDSL-lite

SDSL-lite ist eine C++11 Bibliothek, die speichereffiziente Datenstrukturen zur Verfügung stellt. Die benutzten Datenstrukturen dieser Bibliothek werden im Nachfolgenden an den passenden Stellen eingeführt[7].

Algorithm 1: LZ78V Faktorisierung mit Vektoren

Data: Eingabetext T der Länge n
Result: Vektor mit IDs(Repräsentation der Faktoren) und Vektor mit benutzzen Knoten

konstruiere Suffix-Tree aus T;
erstelle Bit-Vektor *used* und Int-Vektor $factors$;
$pos \leftarrow 1$; `// aktuelle Textposition in` T
while $pos \leq n$ **do**
    suche anhand $T[pos]$ den nächsten Knoten im Suffix-Tree;
    if *aktueller Knoten in used als benutzt markiert* **then**
        $pos \leftarrow pos + c_e(aktuellerKnoten)$;
        `/*` c_e`(Knoten) ist Anzahl Zeichen an der Kante zum Knoten */`
        continue;
    if *aktueller Knoten ist ein Blatt* **then**
        $pos \leftarrow pos + 1$;
    else
        setze Textposition entsprechend der Zeichen an der Kante weiter;
    $pos \leftarrow pos + c_e(aktuellerKnoten)$;
    markiere aktuellen Knoten in *used* als benutzt;

4.2 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.[7] 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 ist $i \in [0..x]$.

4.3 Vektoren zur Faktorisierung

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

4.3.1 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 $64 * \lceil \frac{x}{64} + 1 \rceil$ Bits.[7]

4.3.2 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.

4.4 Abbruchbedingung der Faktorisierung

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

4.5 Faktoren erstellen

Der Startknoten jedes neuen Suchdurchlaufs ist die Wurzel des Suffix-Trees. In Zeile 5 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 6 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.

4.6 Beispiel

In diesem Abschnitt betrachten wir die Faktorisierung an unserem Beispieltext $T =$ `ananas$`.
Der Suffix-Tree von T aus Abbildung 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.

Haben wir mit Algorithmus 1 die Vektoren $factors$ und $used$ gefüllt, verfügen wir über alle Informationen der Faktorisierung, die wir benötigen. Nun können wir mithilfe von Algorithmus 2 aus den beiden Vektoren konkrete Faktoren mit Referenz und anzuhängenden Symbolen entwickeln. Denn würden wir nur die Vektoren abspeichern, wäre es unmöglich daraus wieder den ursprünglichen Text zu rekonstruieren. Um die Faktoren zu speichern benutzen wir die Array-Darstellung aus Kapitel 3.2.

4.7 rank-Struktur auf used

Eine *rank*-Struktur auf einem Bit-Vektor v ist eine Funktion $rank_v : [1..|v|] \rightarrow \mathbb{N}_0$ die jedem Index i die Häufigkeit von 1-Bits im Vektor bis zum Index i zuordnet[8, S. 39].

4.8 Int-Vektor N

Der Int-Vektor N hat z Einträge. Dieser Vektor speichert für jede ID aus $factors$, eine Faktor-ID, die angibt den wievielten Faktor der entsprechende Knoten repräsentiert. Um diesen Vektor zu füllen, gehen wir $factors$ einmal von links nach rechts durch. Wir setzen dabei $N[used.rank(factors[j])] = j$ für jeden Eintrag in $factors$ an der Position j.

Algorithm 2: LZ78V-Tree

Data: Suffix-Tree, Bit-Vektor *used*, Int-Vektor $factors$
Result: Arraydarstellung des LZ78V-Tree

```
1  erstelle Int-Vektor N;
2  erstelle rank-Struktur auf used;
3  for i ←0 to z − 1 do
4  |  N[used.rank(factors[i])] = i + 1;
5  erstelle A1 und A2;
6  for j ←0 to x − 1 do
7  |  if used[j]!=0 then
   |  |  /* used[j] = 0 ⇔ Knoten mit ID j ist unbenutzt                    */
8  |  |  p ← Elternknoten im Suffix-Tree des Knotens mit der ID j;
9  |  |  if p ist Wurzel des Suffix-Trees then
10 |  |  |  A1[N[used.rank(j)]] ← 0;
11 |  |  else
12 |  |  |  A1[N[used.rank(j)]] ← N[U.rank(p.id())];
   |  |  |  /* id() gibt die ID eines Knotens zurück                       */
13 |  |  if Knoten mit ID j im Suffix-Tree ist ein Blatt then
14 |  |  |  A2[N[used.rank(j)]] ← erstes Zeichen der Kante zum Knoten mit ID j;
15 |  |  else
16 |  |  |  A2[N[used.rank(j)]] ← ganzer String der Kante zum Knoten mit ID j;
```

4.9 Füllen von A_1 und A_2

A_1 füllen wir mit Referenzen auf Faktoren, A_2 mit Strings, die an den referenzierten Faktor angehangen werden. Beide Arrays haben die Länge z. Nun können wir mithilfe von *used*, N und dem Suffix-Tree die Arrays A_1 und A_2 füllen. Dazu durchlaufen wir *used* einmal von links nach rechts durch und verfahren mit jedem Eintrag an Position j, der als nicht benutzt markiert ist, wie folgt:
Falls der Elternknoten von Knoten j die Wurzel des Suffix-Trees ist, setze $A_1[N[used.rank(j)]] = 0$. Ansonsten setze $A_1[N[used.rank(j)]] = N[U.rank(p.id())]$. $N[used.rank(j)]$ ist dabei die Faktor-ID des entsprechenden Knoten mit der Knoten-ID j. Auf diese Weise werden die Referenzen in A_1 richtig befüllt. Wollen wir A_2 füllen, müssen wir testen ob der Knoten mit der ID j im Suffix-Tree ein Blatt ist. Falls ja, speichern wir an der Position $N[U.rank(j)]$ in A_2 nur das erste Zeichen der Kante zum Knoten j. Andernfalls den gesamten String an der Kante. Dies löst das Problem von einzigartigen Zeichen im Eingabetext.

Beweis Sei $T = s_1 k s_2$ mit $s_1, s_2 \in \Sigma^+ \setminus \{k\}$ und $k \in \Sigma$.
Sind wir beim Lesen von T bei k angelangt, landen wir im Suffix-Tree als nächstes immer zu einem Blatt über eine Kante, die mit ks_2 markiert ist. Die Faktorisierung wäre somit abgeschlossen, obwohl s_2 noch viel Potenzial auf eine gute Faktorisierung haben kann. Somit nehmen wir nur k in die Faktorisierung auf und arbeiten mir s_2 weiter. □

4.9.1 Beispiel

In diesem Abschnitt betrachten wir beispielhaft die Konstruktion des LZ78V-Tree. Als Eingabe benutzen wir die Ergebnisse aus dem Beispiel in Kapitel 4.6 und konstruieren mithilfe des Algorithmus 2 die Array-Darstellung des LZ78V-Tree.

- Eingabe:
  - Suffix-Tree aus Abbildung 2
  - Bit-Vektor *used* aus dem Ergebnis von Kapitel 4.6:
    $used = [0,1,1,0,0,0,0,1,0,1,0]$
  - Int-Vektor $factors$ aus dem Ergebnis von Kapitel 4.6:
    $factors = [2,7,9,1]$
- Nach Zeile 4: $N = [3, 0, 1, 2]$
- Ergebnis:
  - $A_1 = [0, 0, 2, 0]$
  - $A_2 = [\texttt{a}, \texttt{na}, \texttt{s}, \texttt{\$}]$

5 Kodierung

Wurde aus einem Eingabetext T gemäß Algorithmus 1 eine Faktorisierung erzeugt und daraus gemäß Algorithmus 2 die Array-Darstellung des LZ78V-Trees, können wir mit der *Kodierung* beginnen. Durch die Kodierung können wir die Array-Darstellung effizient speichern.
Zuerst konstruieren wir aus der Array-Darstellung einen String. Diesen String nennen wir Ergebnisstring s. Dazu hängen wir abwechselnd die Einträge aus A_1 und A_2 hintereinander, getrennt durch ein Zeichen, das in keinem Text vorkommen kann. Daher nehmen wir als Trennzeichen das ASCII Zeichen '\x1B'. Dieses Trennzeichen wird im Folgenden als `'|'` dargestellt. In unserem Beispiel $T =$ `ananas$` erhalten wir so den Ergebnisstring $s =$ `0|a|0|na|2|s|0|$|`. Dieser Ergebnisstring s soll nun kodiert werden. Hierzu wird der nachfolgende Kodierer eingeführt.

5.1 Kanonische Huffman Kodierung

Die kanonische Huffman Kodierung ordnet jedem Symbol einen präfixfreien Code zu. Präfixfrei bedeutet, dass es keinen Code gibt, der Präfix eines anderes Codes ist. Diese Kodierung hat gegenüber der in der Literatur üblichen Huffman Kodierung[9, S. 141ff] den Vorteil, dass sie speichereffizienter ist und die Dekodierung schneller ist. [10, S. 335]

Algorithm 3: Berechne Code-Längen der Symbole [10, S. 341]

Data: Array H der Länge w mit Häufigkeiten aller w Symbole
Result: Code Längen der Symbole

```
1  /* Phase Eins */
2  erstelle ein Array A mit Länge 2n;
3  for i ← 0 to w do
4  |   lese h_i;                          /* h_i ist Häufigkeit des i-ten Symbols */
5  |   A[w + i] ← h_i;
6  |   A[i] ← w + i;
7  h ← w;
8  baue einen Min-Heap von A[1..w];
9  /*Phase zwei*/
10 while h > 1 do
11 |   m_1 ← A[1];
12 |   A[1] ← A[h];
13 |   h ← h − 1;
14 |   stelle für A[1..h] Heap-Eigenschaft wieder her;
15 |   m_2 ← A[1]
16 |   A[h + 1] ← A[m_1] + A[m_2];
17 |   A[1] ← h + 1;
18 |   A[m_1] ← A[m_2] ← h + 1;          /* speichere Elternverweise */
19 |   stelle für A[1..h] Heap-Eigenschaft wieder her;
20 /*Phase drei*/
21 for i ← w + 1 to 2w do
22 |   depth ← 0;
23 |   node ← i;
24 |   while node > 2 do
25 |   |   depth ← depth + 1;
26 |   |   node ← A[r];
```

Algorithmus 3 ist der erste Teil der Berechnung der kanonischen Huffman Kodierung. Hier berechnen wir für jedes Symbol des Eingabetextes, wie lang der entsprechende Code für das Symbol ist. Dabei bauen wir uns mithilfe eines *Min-Heaps* einen *Huffman-Tree*, an dem wir die Code-Längen für alle Symbole ermitteln können.

5.1.1 Huffman-Tree

Um einen Huffman-Tree zu einer Menge von Symbolen zu konstruieren, erstellen wir für jedes Symbol einen Knoten und ordnen diesem die Häufigkeit des Symbols zu. Dann fügen wir immer die beiden Knoten mit der geringsten Häufigkeit zu einem neuen Teilbaum zusammen. Die Wurzel dieses Baums markieren wir mit der Summe der Häufigkeiten der beiden Kindknoten. Dies wiederholen wir solange, bis nur noch ein Baum übrig ist. Dieser ist der *Huffman-Tree* der Eingabemenge[11, S. 39]. Im Beispiel von $s =$ `0|a|0|na|2|s|0|$|` sehen wir den Huffman-Tree in Abbildung 6.

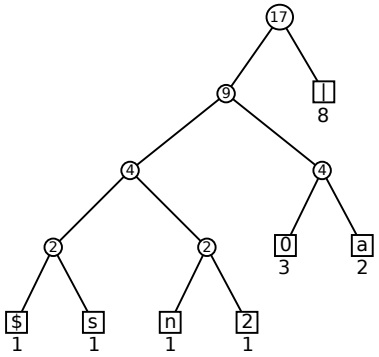


Abbildung 6: Huffman-Tree zu $s = \texttt{0|a|0|na|2|s|0|\$|}$. Die Blätter enthalten die Symbole. Darunter sehen wir die Häufigkeiten der Symbole. Die restlichen Knoten sind jeweils mit der Summe der Häufigkeiten in den entsprechenden Teilbäumen markiert.

Die Code-Längen der einzelnen Symbolen entsprechen dann der Tiefe im Huffman-Tree. So hat das Symbol '$' die Code-Länge 4.

5.1.2 Min-Heap

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

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

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

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 7 zeigt beispielhaft einen *Min-Heap* $(\mathbb{N}, \geq)$.

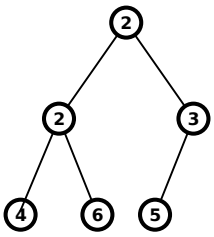


Abbildung 7: Beispiel eines *Min-Heaps* $(\mathbb{N}, \geq)$.

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 8 sehen wir die Array-Darstellung des Baums aus Abbildung 7.

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

Abbildung 8: Array-Darstellung des Baums aus Abbildung 7.

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. Da die Höhe eines Heaps logarithmisch von der Anzahl n der Elemente im Heap abhängt und wir die Eigenschaft nach dem Löschen und Einfügen auf dem Pfad von der betroffenen Stelle bis zur Wurzel wiederherstellen müssen, benötigen Einfüge- und Löschoperationen $O(\log n)$ Zeit. [12, S.115ff]

Im Folgenden betrachten wir die drei Phasen von Algorithmus 3 und erläutern, wie wir mithilfe eines Min-Heaps einen Huffman-Tree konstruieren und an diesem die Code-Längen ermitteln.
In **Phase Eins** lesen wir in die zweite Hälfte $A[w{+}1..2w]$ alle Häufigkeiten der vorkommenden Sybole ein. Diese Häufigkeiten finden wir im Array H, wobei $H[i]$ die Häufigkeit des $i-ten$ Symbols angibt. Die Werte der ersten Hälfte $A[1..w]$ dienen als Zeiger auf eine Häufigkeit aus der zweiten Hälfte. A können wir auch als eine Array-Darstellung eines Binärbaums verstehen. Betrachten wir im Beispiel den Ergebnisstring $s =$ `0|a|0|na|2|s|0|$|`. Somit haben wir eine Symbolliste $Sym =$ [`0`,`|`,`a`,`n`,`2`,`s`,`$`] und Häufigkeiten
$H = [3, 8, 2, 1, 1, 1, 1]$, sodass $H[i]$ die Häufigkeit in s von $Sym[i]$ angibt. Wir erhalten so:

$$A = [8, 9, 10, 11, 12, 13, 14, 3, 8, 2, 1, 1, 1, 1]$$

Dann bauen wir in Zeile 9 aus der ersten Häfte einen Min-Heap. Wir vergleichen jedoch nicht die Elemente direkt, sondern die Werte auf die sie referenzieren. So definieren wir einen Min-Heap $(\mathbb{N}, \succeq)$ mit $\succeq (x, y) \mapsto A[x] \succeq A[y]$.
Min-Heap mit der ersten Häfte von A:

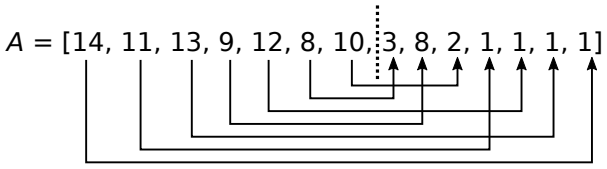


Abbildung 9: Hier sehen wir die Array-Darstellung des Min-Heaps aus dem Beispiel. Bis zu der gestrichelten Trennlinie gehören die Einträge zum Min-Heap. Die Pfeile verdeutlichen die Zeiger auf die Werte der zweiten Hälfte.

Abbildung 10 zeigt die Baumdarstellung des Min-Heaps. Die erste Hälfte von A kann man dabei als IDs der Knoten verstehen, die zweite Hälfte als Werte innerhalb der Knoten.[10, S. 342]

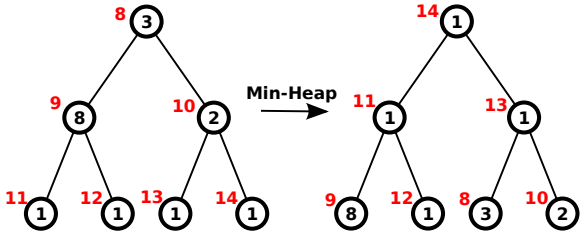


Abbildung 10: Diese Abbildung zeigt beispielhaft das Array A in der Baumdarstellung vor und nachdem wir aus der ersten Häfte ein Min-Heap konstruiert haben. Die roten Zahlen sind aus der ersten Häfte, die schwarzen aus der zweiten Hälfte von A.

In **Phase zwei** konstruieren wir aus A einen Huffman-Tree. Dazu entfernen wir die Referenz auf den kleinsten Wert aus dem Heap und speichern ihn in m_1. Diese ist gemäß der Heap-Eigenschaft in A[1] zu finden. Außerdem überrschreiben wir $A[1]$ mit $A[h]$. Wir verringern h um 1 und stellen die Heap-Eigenschaft im Bereich $A[1..h]$ wieder her. So hat sich die Größe des Heaps um 1 verringert. Nun speichern wir wieder die Referenz auf den kleinsten Wert aus dem Heap in m_2. In die freie Position $A[h+1]$ schreiben wir nun die Summe der beiden Häufigkeiten auf die m_1 und m_2 verweisen. An $A[m_1]$ und $A[m_2]$ schreiben wir $h+1$, also die Referenz auf den Elternknoten mit der Summe ihrer beiden Häufigkeiten. Nun stellen wir die Heap-Eigenschaft in $A[1..h]$ wieder her und beginnen von vorn. Dies wiederholen wir solange wie $h > 1$ gilt. Dieses Vorgehen verdeutlichen wir in Abbildung 11, indem wir im Beispiel die erste Iteration von Phase zwei illustrieren.

Als Endergebnis von Phase zwei erhalten wir in unserem Beispiel:

$$A[2, 17, 2, 3, 3, 5, 5, 7, 2, 4, 6, 4, 6, 7]$$

$A[1]$ enthält hierbei immer 2, da in $A[2]$ der einzige Knoten des Heaps gespeichert ist. Alle anderen Einträge in A sind Verweise auf Elterknoten. [10, S. 342ff] Diese Verweise nutzen wir in Phase 3, um die Code-Längen zu ermitteln.

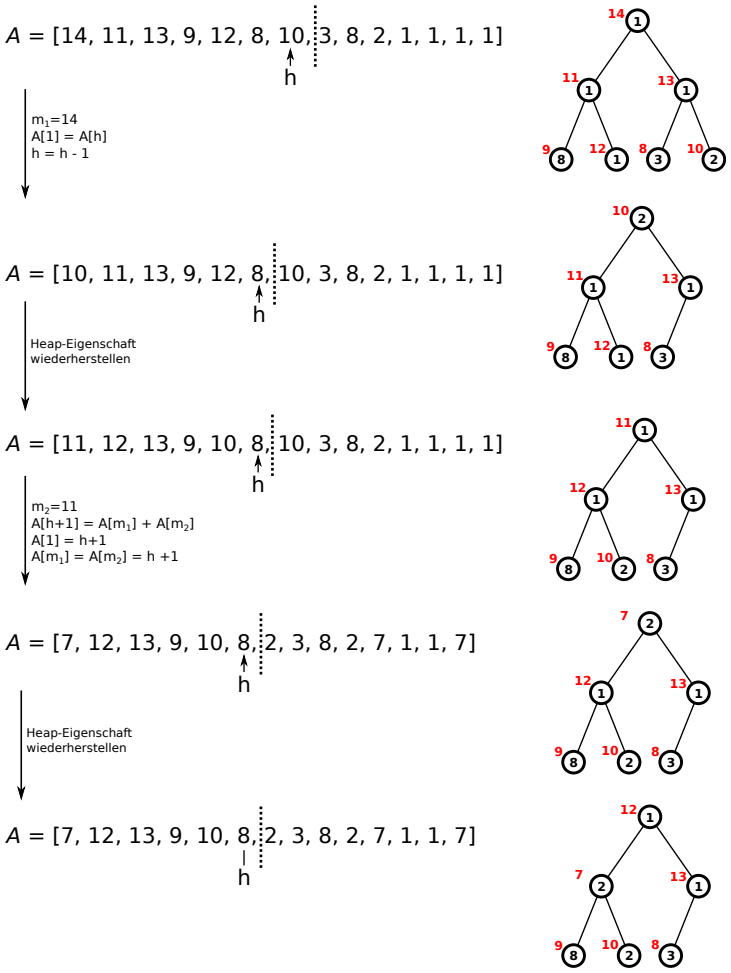


Abbildung 11: Diese Abbildung zeigt die erste Iteration von Phase zwei des Algorithmus 3. Links sehen wir die Array-Darstellung und rechts die Baumdarstellung der Heap-Struktur. An den Pfeilen sehen wir die Operationen, die in den Übergängen passieren.

In **Phase 3** ermitteln wie die Code-Länge für jedes Symbol. Dazu beginnen wir an der Stelle $w + 1$ und verfolgen die Verweise auf den Elternknoten, bis wir an der Wurzel angelangt sind. Die ermittelte Länge speichern wir an der Stelle $w+1$ ab. Dies wiederholen wir für jede Stelle aus $A[w + 2..2w]$. Am Ende befindet sich die Code-Länge für Symbol i in $A[w + i]$.[10, S. 344]

Haben wir die Code-Längen für jedes Symbol berechnet, können wir mithilfe von Algorithmus 4 für jedes Symbol einen Code in Dezimaldarstellung berechnen. Die rechten l_i Bits dieser Dezimalzahl in Binärdarstellung sind der Code für das Symbol i.
Das Array num speichert für jede Code-Länge bis max wie oft sie in L vorkommt. In $firstcode$ berechnen wir für jede Code-Länge den ersten Code. Dieser ist abhängig von $firstcode$ und num des um 1 längeren Codes. Das Ergebnis dieses Algorithmus ist in $cordword$ gespeichert. Für unser Beispiel sieht das Ergebnis so aus:

Algorithm 4: Symbolen einen kanonischen Huffman Code zuweisen [10, S. 338]

Data: Array L mit Code-Längen von w Symbolen, l_i ist Code-Länge von Symbol i
Result: Array mit Integern für Codes

```
1  max ← maximale Code-Länge;
2  for m ← 1 to max do
3  |  num[m] ← 0;
4  for i ← 1 to w do
5  |  num[l_i] ← num[l_i] + 1;
   // num[i] = Anzahl Codes der Länge i
6  firstcode[max] ← 0;
7  for m ← max − 1 downto 1 do
8  |  firstcode[m] ← (firstcode[m + 1] + num[m + 1])/2;
9  for m ← 1 to max do
10 |  nextcode[m] ← firstcode[m]
11 for i ← 1 to n do
12 |  codeword[i] ← nextcode[i];
13 |  nextcode[l_i] ← nextcode[l_i] + 1;
```

Symbol	Code-Länge l_i	*codeword*[i]	Code
`0`	3	3	011
`|`	1	1	1
`a`	3	2	010
`n`	4	1	0001
`2`	4	3	0011
`s`	4	0	0000
`$`	4	2	0010

Tabelle 1: Diese Tabelle zeigt beispielhaft *codeword* zum Ergebnisstring $s =$ `0|a|0|na|2|s|0|$|`. Die Werte in *codeword* sind Codes in Dezimaldarstellung. Der Code von Symbol i ergibt sich aus den l_i rechten Bits von *codeword*[i]

Als letzten Schritt der Kodierung müssen wir im Ergebnisstring s jedes Symbol i durch seinen Code ersetzen und Informationen für die Dekodierung abspeichern. Diese Informationen sind zum einen wie viele Codes es von welcher Code-Länge gibt und zum anderen die Symbole in aufsteigender Reihenfolge ihrer Code-Längen. Die Code-Längen sind mit ',' getrennt, dann folgt ein ';' und dann die Symbole. So erhalten wir die Ausgabe C_1C_2 der Kodierung in unseren Beispiel:

- $C_1 =$ 1,0,2,4;`|a0sn$2`
- $C_2 =$ 01110101011100010101001110000101111100101

Wie man aus dieser Ausgabe wieder den ursprünglichen Ergebnisstring s dekodiert werden wir im nachfolgenden Kapitel 6 sehen.

6 Dekodierung

In diesem Kapitel beschäftigen wir uns damit, eine Kodierung eines Ergebnisstrings s, erzeugt mit der kanonischen Huffman Kodierung aus Kapitel 5, wieder in den Ursprungszustand zu übersetzen. Diesen Vorgang nennen wir *Dekodierung*. Betrachten wir C_1 der Ausgabe der Kodierung aus dem Beispiel aus Kapitel 5 C_1 = 1,0,2,4;`|a0sn$2`.
In C_1 sind alle Informationen enthalten, um C_2 zu übersetzen.

Index i	Symbol	Code-Länge l_i
1	`|`	1
2	`a`	3
3	`0`	3
4	`s`	4
5	`n`	4
6	`$`	4
7	`2`	4

Tabelle 2: Diese Tabelle zeigt am Beispiel aus Kapitel 5 die Informationen, die in C_1 enthalten sind.

Algorithm 5: Dekodieren von C_1C_2 [10, S. 339]

Data: C_1 und C_2
Result: Ergebnisstring s

```
1  /*Initialisierung */
2  Fülle num, sodass num[i] = Anzahl Codes der Länge i;
3  max ← maximale Code-Länge;
4  firstcode[max] = 0;
5  for m ← max − 1 downto 1 do
6  |  firstcode[m] ← (firstcode[m + 1] + num[m + 1])/2;
7  for i ← 1 to max do
8  |  nextcode[i] ← firstcode[i];
9  for i ← 1 to w do
10 |  symbol_table[l_i, nextcode[l_i] − firstcode[l_i]] ← i;
11 |  nextcode[l_i] ← nextcode[l_i] + 1;      // l_i ist Länge des Codes von Symbol i
12 /*Dekodierung*/
13 result_string;
14 while Ende von C_2 nicht erreicht do
15 |  v ← read_bit();          // read_bit() gibt nächstes Bit von C_2 zurück
16 |  l ← 1;
17 |  while v < firstcode[l] do
18 |  |  v ← 2 * v + read_bit();
19 |  |  l ← l + 1;
20 |  index ← symbol_table[l, v − firstcode[l]];
21 |  result_string ← result_string + symbol[index];
22 |  v ← 0;
```

Im ersten Teil von Algorithmus 5 werden die Informationen aus C_1 verarbeitet und für das Dekodieren vorbereitet. Mithilfe von $num, firstcode$ und $nextcode$ füllen wir $symbol\_table$. Dies ist eine Tabelle, die Indizes von Symbolen enthält und lässt sich zum Beispiel mithilfe eines Vektors von Vektoren implementieren. Dieser benötigt $w * \log w$ Bits. In unserem Beispiel sieht $symbol\_table$ wie folgt aus:

Code-Länge	1	2	3	4
Symbol-Index	1		2	4
			3	5
				6
				7

Index	1	2	3	4	5	6	7
Symbol	|	a	0	s	n	$	2

Tabelle 3: Hier sehen wir die Tabelle $symbol\_table$ am Beispiel aus Kapitel 5.

Im zweiten Teil von Algorithmus 5 geht es um die Dekodierung von C_2. Dazu lesen wir nacheinander Bits von C_2 in v ein und erhöhen in jedem Durchlauf die Länge l des Codes. Dies machen wir solange, bis $v \geq firstcode[l]$ gilt, denn dann haben wir in v einen Code für ein Symbol eingelesen. Das liegt daran, dass bei einem kanonischen Huffman Code alle k-Bit Präfixe von Codes, die länger sind als k, kleiner sind als $firstcode[k]$ [10, S. 338]. Dank dieser Eigenschaft benötigt Algorithmus 5 $O(b)$ Zeit, wenn b die Länge von C_2 ist. Diese Eigenschaft verdeutlichen wir an unserem Beispiel in Tabelle 4.

Symbol	Code-Länge	Code	k-Bit Präfix			
			1	2	3	4
|	1	1	1			
a	3	010	0	1	2	
0	3	011	0	1	3	
s	4	0000	0	0	0	0
n	4	0001	0	0	0	1
$	4	0010	0	0	1	2
2	4	0011	0	0	1	3
		firstcode[k]	1	2	2	0

Tabelle 4: Diese Tabelle zeigt alle Präfixe der Codes aus Kapitel 5. Hier erkennen wir, dass alle k-Präfixe von Codes, die länger sind als k, kleiner sind als $firstcode[k]$.

Haben wir einen Code gelesen, d.h. bricht die innere Schleife ab, ist der Index des entsprechenden Symbols in $symbol\_table$ in der l-ten Spalte an der Position $v - firstcode[l]$ gespeichert. Dieses Symbol hängen wir an $result\_string$ und setzen v wieder auf 0. Dies wiederholen wir solange, bis wir alle Bits in C_2 gelesen und somit den Ergebnisstring s erfolgreich rekonstruiert haben.

7 Dekomprimierung

Haben wir mit dem Algorithmus 5 aus Kapitel 6 den Ergebnisstring s rekonstruiert, fehlt nur noch ein Schritt um wieder zum Ursprungstext zu gelangen. Wir übersetzen die Faktorisierung, die durch s dargestellt ist, wieder zurück in den Text T.

```
   Data: Ergebnisstring s
   Result: Text T
 1 /*Rekonstruktion Faktorisierung*/
 2 int-Vektor A_1;
 3 Vektor von Strings A_2;
 4 Lese abwechselnd aus s zwischen Trennzeichen '|' erst in A_1, dann in A_2 ein, bis das
   Ende von s erreicht ist;
 5 /*Konstruktion von T*/
 6 Vektor von Strings result;
 7 for i ← 1 to A_1.size() do
 8 |   if A_1[i] == 0 then
 9 |   |   result.push(A_2[i]);
10 |   else
11 |   |   tmp ← result[A_1[i]] + A_2[i];
12 |   |   result.push(tmp);
13 String T for i ← 1 to result.size() do
14 |   T ← T + result[i];
15 return T
```

Zuerst erstellen wir einen int-Vektor A_1 und einen String-Vektor A_2. A_1 speichert Referenzen auf Faktoren, A_2 die Texte, die in den Faktoren enthalten sind. Dazu lesen wir abwechselnd aus dem Ergebnisstring s erst eine Referenz in A_1, dann einen Text in A_2 ein. Referenzen und Text sind dabei durch das Trennzeichen '|' getrennt.
Haben wir die beiden Vektoren gefüllt, können wir nun den Text T rekonstruieren. Im String-Vektor *result* bauen wir Substrings von T auf. Wir durchlaufen A_1 einmal von links nach rechts. Ist am Index i in A_1 eine 0 gespeichert, so verweist der Faktor auf keinen anderen Faktor. Wir können den Text $A_2[i]$ an *result* anhängen. Ist an dem Index i keine 0 gespeichert, so hängen wir an *result* erst den Text an, der in *result* an der Position $A_1[i]$ gespeichert ist und dann $A_2[i]$. Da $A_1[i]$ nur Werte enthalten kann, die kleiner als i sind und zu diesem Zeitpunkt in *result* schon $i-1$ Strings gespeichert sind, kann es nie vorkommen, dass ein ungültiger Index in *result* adressiert wird.

8 Implementierung

In diesem Kapitel dokumentieren wir die Implementierung der in Kapitel 4, 5 und 6 beschriebenen Algorithmen.
Die Implementierung erfolgt in der Programmiersprache C++. Das ganze Projekt wurde als neuer Kompressionsalgorithmus im TuDoComp(**T**echnical **U**niversity of **DO**rtmund **COMP**ression) Framework entwickelt. In diesem Framework sind bereits eine Vielzahl von Standard-Kompressionsalgorithmen enthalten. Daher benutzen wir dieses in Kapitel 9 auch zum Vergleichen von LZ78V mit anderen Algorithmen aus dem Framework.

Um einen neuen Kompressionsalgorithmus im TuDoComp Framework zu implementieren muss man das *Compressor* Interface implementieren. Dieses besitzt zwei Methoden: *compress* und *decompress*. In *compress* faktorisieren wir den Eingabetext T mit LZ78V und kodieren ihn mit der kanonischen Huffman-Kodierung. In *decompress* dekodieren wir die Eingabe zuerst und dekomprimieren das Ergebnis dann nach dem in Kapitel 7 beschriebenen Verfahren. Dann gilt: decompress(compress(T)) = T.

8.1 Faktorisierung

Der Algorithmus 1 aus Kapitel 4 faktorisiert einen Text nach LZ78V. Dabei nutzen wir zur Implementierung des Suffix-Trees und der Vektoren die C++11 Bibliothek SDSL-lite [7].
Diese Blibliothek stellt drei verschiedene Implementierungen eines Suffix-Trees zur Verfügung. Diese benötigen verschieden viel Speicherplatz und sind bei der Konstruktion und Traversierung unterschiedlich schnell[1].

Implementierung	Speicherbedarf
cst_sct3	$3n$ + o(n) + |CSA| + |LCP| Bits
cst_sada	$4n$ + o(n) + |CSA| + |LCP| Bits
cst_fully	keine Angabe

Tabelle 5: Diese Tabelle zeigt die verschiedenen Implementierungen eines Suffix-Trees in SDSL und ihren Speicherbedarf. n ist hierbei die Länge des Textes.

Welche Implementierung sich für LZ78V am besten eignet, werden wir im Kapitel 9 testen.
In Algorithmus 2 benutzen wir eine rank-Struktur auf einem Bit-Vektor. Diese konstruieren wir mit Hilfe der Klasse "rank_support"von SDSL [7]. Diese Klasse bietet wieder verschiedene Implementierungen einer rank-Struktur an[2], die wir in Kapitel 9 testen und vergleichen werden.

[1] sdsl Cheat Sheet, `http://simongog.github.io/assets/data/sdsl-cheatsheet.pdf`
[2] siehe Fußnote 1

Implementierung	zusätzlicher Speicher zum Bit-Vektor	Zeit zum Erstellen
rank_support_v	$0,25n$ Bits	$O(1)$
rank_support_v5	$0,0625n$ Bits	$O(1)$
rank_support_scan	64 Bits	$O(n)$

Tabelle 6: Diese Tabelle zeigt die verschiedenen Implementierungen einer rank-Struktur in SDSL und ihren Speicherbedarf. n ist hierbei die Länge des Vektors.

8.2 Kodierung/Dekodierung

Nachdem man beim **Kodieren** in Kapitel 5 für jedes Symbol einen Code berechnet hat, muss C_1C_2 in den Ausgabestrom geschrieben werden. Für C_1 benutzen wir den *ostream* aus der Standard C++ Bibliothek. Für C_2 benötigen wir allerdings einen Ausgabestrom, der bitweise schreiben kann. Diese Möglichkeit bietet die Klasse *BitOStream* aus TuDoComp. Dieser Ausgabestrom besitzt eine Methode *write_int(c, l)*, die einen Integer c in Binärdarstellung mit der Länge l in die Ausgabe schreibt.

Beim **Dekodieren** verfahren wir sehr ähnlich. C_1 lesen wir mit dem *istream* byteweise aus der Standard C++ Bibliothek ein. Für C_2 benutzen wir die Klasse *BitIStream* von TuDoComp. Diese bietet die Möglichkeit mit der Methode *read_bit()* bitweise die Eingabe einzulesen.

9 Praktische Tests

In diesem Kapitel testen wir das Programm aus Kapitel 8 und dokumentieren diese Tests.

Diese Tests dienen neben den reinen Funktionstests dazu, verschiedene Implementierungen von Datenstrukturen (siehe Kapitel 8) miteinander zu vergleichen und die Besten für den Vergleich zu anderen Kompressionsverfahren auszuwählen.

Wir können mithilfe von TuDoComp Diagramme erzeugen, die Informationen über Speicherverbrauch und Geschwindigkeit der einzelnen Schritte von *compress* und *decompress* enthalten. Wir unterteilen *Compress* in folgende Schritte:

- Initialisierung
  - Eingabetext einlesen
  - Suffix-Tree konstruieren
  - Vektoren $factors$ und $used$ anlegen
  - sonstige Hilfsvariablen anlegen

- Faktorisierung
- Füllen von A_1 and A_2
- Kodierung
- Output

Decompress unterteilen wir in die Schritte:

- Dekodierung
- Text rekonstruieren

9.1 Suffix-Trees

Zuerst Testen wir, welcher der drei Suffix-Tree Implementierungen (siehe Kapitel 8) sich am Besten Für LZ78V eignet.

9.1.1 Testumgebung

Als Eingabetext T benutzen wir verschiedene Texte der Größe 1MB.

9.1.2 Durchführung

Wir faktorisieren den Eingabetext T mit LZ78V und kodieren ihn anschließend mit der kanonischen Huffman-Kodierung. Dabei führen wir den Test einmal mit dem Suffix-Tree der Klasse *cst_sct3* und einmal mit *cst_sada* durch. Den *cst_fully* können wir für LZ78V nicht anwenden, da die Knoten in dieser Implementierung keine IDs haben und LZ78V darauf aufbaut.

9.1.3 Ergebnisse

In Abbildung 12 sehen wir die Diagramme der Tests mit DNA-Sequenzen als Eingabe. Der Speicherverbrauch bei der Initialisierung liegt bei dem Test mit *cst_sada* deutlich höher als beim Test mit *cst_sct3*. Auch die Zeit zum Faktorisieren ist bei *cst_sada* deutlich höher.
Das gleiche Ergebnis erhalten wir bei den Tests mit einem englischen Text als Eingabe. (siehe Abbildung 13)

9.1.4 Auswertung

Die Tests mit beiden Implementierungen haben gezeigt, dass *compress* mit *cst_sct3* als Suffix-Tree Implementierung besser ist als mit *cst_sada*. Es zeigt sich, dass der Speicherbedarf bei *cst_sct3* um ca. 33% und die Laufzeit ca. 50% in der Initialisierung geringer ist.

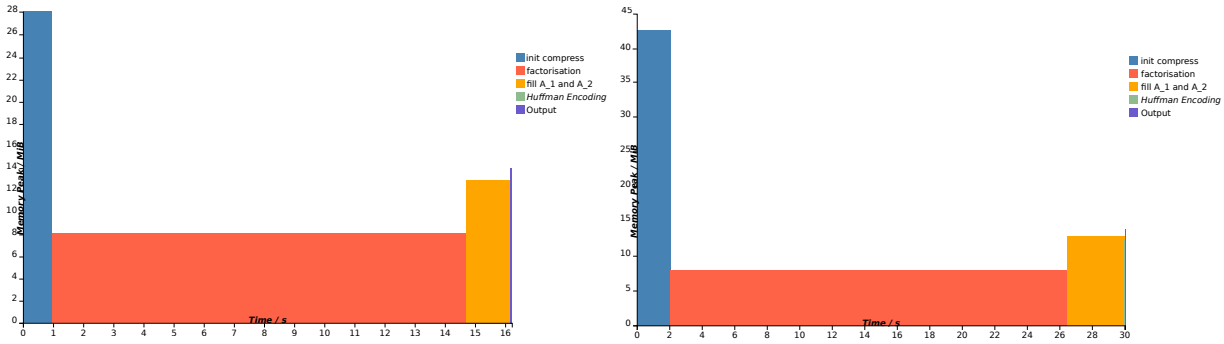


Abbildung 12: Links sehen wir das Diagramm für den Test mit *cst_sct3* und rechts mit *cst_sada*. Eingabe: 1MB DNA-Sequenz

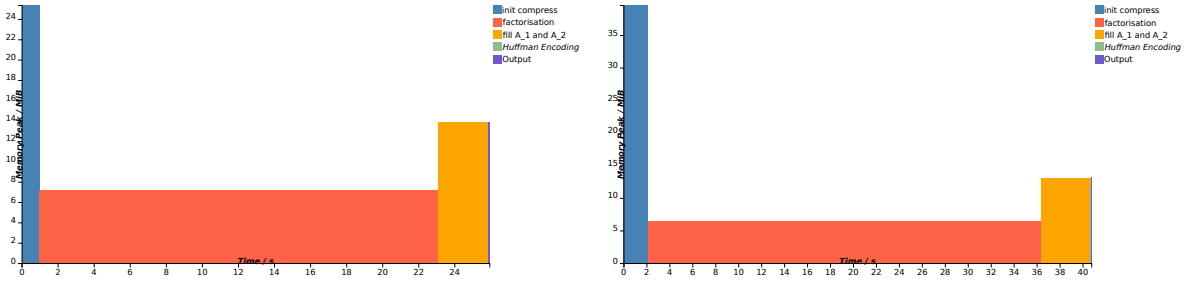


Abbildung 13: Links sehen wir das Diagramm für den Test mit *cst_sct3* und rechts mit *cst_sada*. Eingabe: 1MB Englischer Text

Bei der Faktorisierung haben beide Implementierungen ungefähr den gleichen Speicherbedarf. Bei der Laufzeit allerdings ist *cst_sada* um ca. 40% schlechter.
Basierend auf diesen Ergebnissen verwenden wir in weiteren Tests und Vergleichen mit anderen Kompressionsverfahren als Suffix-Tree *cst_sct3*.

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] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Trans. Inf. Theor.*, 24(5):530–536, September 2006.

[6] Dominik Köppl and Kunihiko Sadakane. Lempel ziv computation in compressed space (LZ-CICS). *CoRR*, abs/1510.02882, 2015.

[7] 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.

[8] J. Ian Munro. *Tables*, pages 37–42. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

[9] Gilbert Held and Thomas Marshall. *Data Compression; Techniques and Applications: Hardware and Software Considerations.* John Wiley & Sons, Inc., New York, NY, USA, 3rd edition, 1991.

[10] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images.* Van Nostrand Reinhold, New York, NY, USA, 1994.

[11] James A. Storer. *Data Compression: Methods and Theory.* Computer Science Press, Inc., New York, NY, USA, 1988.

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