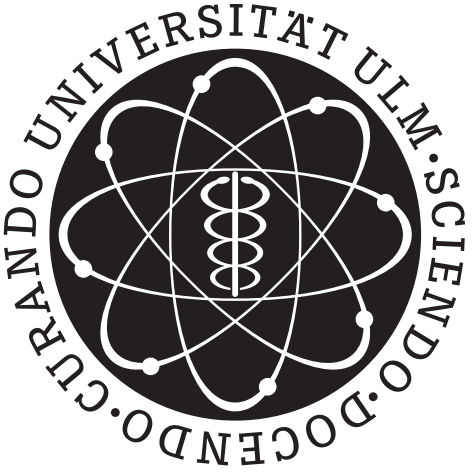
Konstruktion des LCP-Arrays aus der Burrows-Wheeler-Transformierten

Diplomarbeit

Institut für Theoretische Informatik
Fakultät für Ingenieurwissenschaften und Informatik
Universität Ulm



vorgelegt von: Timo Beller

Matrikelnummer: 629727

Gutachter: Prof. Dr. Enno Ohlebusch
Prof. Dr. Uwe Schöning

Betreuer: Prof. Dr. Enno Ohlebusch
Dipl.-Inf. Simon Gog

27. Juli 2011

Inhaltsverzeichnis

1

Einleitung

Die Menge digital abgespeicherter Daten wächst seit Jahren stetig. So gibt es beispielsweise eine infolge erheblicher Fortschritte in der Sequenzierung schnell anwachsende DNA-Datenmenge. Oft möchte man diese Datenmengen nicht nur speichern, sondern auch effizient durchsuchen können. Im Falle von DNA-Daten werden meist viele verschiedene Suchanfragen auf denselben Daten durchgeführt. In diesem Fall ist es sinnvoll, diese einmalig so vorzuverarbeiten, dass anschließend Suchanfragen schnell beantwortet werden können.

Das Suffix-Array ist eine sehr häufig eingesetzte Indexstruktur, welche das schnelle Durchsuchen sehr großer Datenmengen erlaubt. In den letzten Jahren wurden verschiedene Suffix-Array-Konstruktionsalgorithmen entwickelt, sodass das Suffix-Array inzwischen sehr schnell erstellt werden kann. Jedoch ist es bisher nicht gelungen, besonders speichersparende Algorithmen zur Konstruktion des Suffix-Arrays zu entwerfen.

Um komplexere Suchanfragen beantworten zu können, wird häufig zusätzlich zum Suffix-Array das sogenannte LCP-Array benötigt. Nachdem bei der Entwicklung von Algorithmen zur LCP-Array-Konstruktion längere Zeit keine Fortschritte zu verzeichnen waren, wurden 2009 und 2010 zwei neue, besonderes schnelle und speichereffiziente LCP-Konstruktionsalgorithmen veröffentlicht. Allerdings basieren diese alle auf dem Suffix-Array. Ist der Text, für den das LCP-Array benötigt wird, so groß, dass sein Suffix-Array mangels speichersparender Suffix-Array-Konstruktionsalgorithmen nicht berechnet werden kann, so scheitert auch die Konstruktion seines LCP-Arrays.

Da nicht nur die Konstruktion des Suffix-Arrays, sondern auch das Suffix-Array selbst viel Platz benötigt, wird zunehmend versucht, das Suffix-Array durch andere Indexstrukturen zu ersetzen. So können manche Suchanfragen beispielsweise auch mit der Burrows-Wheeler-Transformierten (kurz BWT) beantwortet werden. Diese

ist wesentlich speichersparender, da hierbei – im Gegensatz zum Suffix-Array – nur die Buchstaben gespeichert werden. Die BWT kann leicht aus dem Suffix-Array berechnet werden, allerdings ist auch eine direkte Berechnung aus dem Text möglich. So wird in [OS09] ein speichereffizienter und linearer Algorithmus zur Konstruktion der BWT vorgestellt, welcher nicht auf dem Suffix-Array beruht. Die beschriebene Situation ist in Abbildung 1.1 dargestellt.

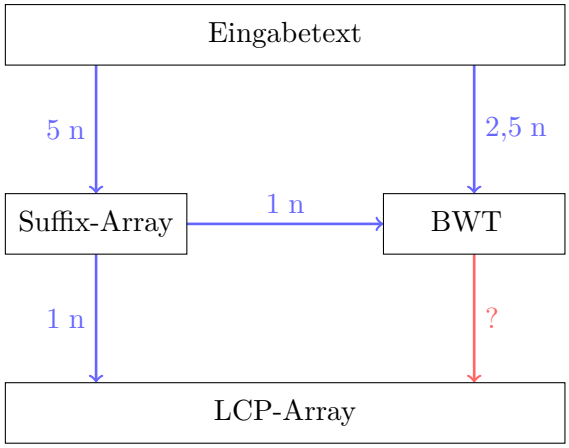


Abbildung 1.1: Die Berechnungsmöglichkeiten mit dem jeweils benötigten Speicher für das Suffix-Array, die BWT und das LCP-Array. In dieser Arbeit wird die Konstruktion des LCP-Arrays aus der BWT untersucht, im Schaubild durch den roten Pfeil angedeutet.

Ziel dieser Arbeit ist es, einen Algorithmus zur Konstruktion des LCP-Arrays zu entwerfen, welcher nicht auf dem Suffix-Array, sondern auf der BWT basiert. Gelingt es darüber hinaus, den Algorithmus ebenso speichereffizient wie die direkte Konstruktion der BWT zu entwerfen, so ist – mit der gleichen Hardware – das Erzeugen des LCP-Arrays von größeren Texten möglich.

Diese Diplomarbeit ist wie folgt gegliedert. Zunächst werden im anschließenden Kapitel die benutzten Notationen, notwendigen Definitionen und grundlegenden Datenstrukturen eingeführt. In Kapitel 3 werden dann einige der bisherigen LCP-Konstruktionsalgorithmen betrachtet. Anschließend wird in Kapitel 4 ein Algorithmus zur Konstruktion des LCP-Arrays vorgestellt, welcher auf der BWT basiert und das Suffix-Array nicht benötigt. Die Laufzeit des neuen Algorithmus wird in Kapitel 5 experimentell bestimmt und mit anderen Algorithmen verglichen. In Kapitel 6 wird gezeigt, wie das Prinzip des neuen Algorithmus zur Lösung zweier weiterer Probleme genutzt werden kann. Zuletzt werden in Kapitel 7 die Ergebnisse der Arbeit zusammengefasst und ein Ausblick auf mögliche Entwicklungen gegeben.

2

Grundlagen

In diesem Kapitel werden zunächst die in dieser Arbeit benötigten Definitionen und Notationen eingeführt. Anschließend werden einige wichtige Datenstrukturen vorgestellt. Die verwendeten Beispiele basieren dabei durchgehend auf dem Text `annasanannas$`.

2.1 Grundlegende Definitionen

Während dieser Arbeit bezeichne Σ ein Alphabet, welches aus σ vielen verschiedenen Buchstaben bestehe und den besonderen Buchstaben `$` enthalte. Auf den Buchstaben aus Σ sei eine totale Ordnung $<$ definiert, wobei `$` der kleinste Buchstabe sei. Reiht man n Buchstaben hintereinander, so erhält man einen Text oder einen String $S = S[1 \dots n]$ der Länge n. Der String der Länge 0 wird in dieser Arbeit durch ϵ repräsentiert. Die Menge Σ^n enthalte alle Strings der Länge n über dem Alphabet Σ. $S[i]$ bezeichne den i-ten Buchstaben und $|S|$ die Länge eines Strings S.

Definition 2.1 (Suffix und Präfix) *Sei S ein String der Länge n. Wir nennen für alle $i \in \{1, \dots, n\}$ den String $S_i = S[i \dots n]$ das i-te Suffix von S und bezeichnen den Teilstring $S[1 \dots i]$ als Präfix von S. Darüber hinaus ist ϵ ebenfalls ein Präfix von S.*

In dieser Arbeit wird für jeden String über Σ vorausgesetzt, dass er mit dem Buchstabe `$` endet; es gilt also $S[n] =$ `$`. Darüber hinaus tritt `$` an keiner anderen Position auf. Folglich gilt für alle Suffixe eines Strings S, dass S_i kein Präfix von S_j für $i \neq j$ ist.

Sollen aus einem String S über Σ nur bestimmte Buchstaben betrachtet werden, so schreiben wir S^M, wobei $M \subseteq \Sigma$ die Menge der gewünschten Buchstaben ist. Für

Aussagen über die Buchstabenhäufigkeit in einem String soll das C-Array dienen. Es liefert für $c \in \Sigma$, wie viele kleinere Buchstaben im gesamten String vorkommen. Anders ausgedrückt ist $C[c] = |S^{\{d \in \Sigma | d < c\}}|$.

Beispiel 2.2 *`annasanannas$` ist ein String der Länge* $|S| = 13$ *über* $\Sigma = \{\texttt{\$}, \texttt{a}, \texttt{n}, \texttt{s}\}$ *mit* $\sigma = 4$. *Es gilt* $S[2] = \texttt{n}$, $S[1 \dots 4] = \texttt{anna}$ *und* $S_{10} = \texttt{nas\$}$. *Zudem ist* $S^{\{\texttt{n},\texttt{s}\}} = \texttt{nnsnnns}$ *und* $C[\texttt{\$}] = 0$, $C[\texttt{a}] = 1$, $C[\texttt{n}] = 6$, $C[\texttt{s}] = 11$.

Aufbauend auf der Ordnung der Buchstaben in Σ kann die lexikografische Ordnung auf Strings definiert werden.

Definition 2.3 (Lexikografische Ordnung) *Sind S und T Strings über Σ, so ist S lexikografisch kleiner als T ($S <_{lex} T$), falls eine der folgenden Bedingungen zutrifft:*

- *S ist ein Präfix von T*
- $S[1] < T[1]$
- $S[1] = T[1]$ *und* $S[2 \dots |S|] <_{lex} T[2 \dots |T|]$

In dieser Arbeit wird die alphabetische Reihenfolge als totale Ordnung für die Buchstaben aus Σ verwendet. Es gilt also $\texttt{\$} < \texttt{a} < \texttt{b} < \cdots < \texttt{z}$. Damit entspricht die lexikografische Ordnung der gewohnten alphabetischen Sortierung.

2.2 Suffix-Array und ω-Intervall

Jeder String S der Länge n hat genau n verschiedene Suffixe S_1, S_2, S_3, …, S_n. Diese Suffixe können gemäß Definition 2.3 sortiert werden, was zur Definition des Suffix-Arrays führt.

Definition 2.4 (Suffix-Array) *Das Suffix-Array SA eines Strings S ist eine Permutation der Zahlen von* 1 *bis n, so dass gilt:* $S_{SA[1]} <_{lex} S_{SA[2]} <_{lex} \cdots <_{lex} S_{SA[n]}$.

Da `$` der kleinste Buchstabe ist und nur am Ende eines Strings vorkommt, gilt für jedes Suffix-Array $SA[1] = n$. Das Suffix-Array benötigt unkomprimiert $n \log_2 n$ Bits, da n Zahlen mit je $\log_2 n$ Bits abgespeichert werden müssen.

Um für ein gegebenes Suffix S_i seine Position im Suffix-Array und somit seine lexikografische Ordnung zu bestimmen, kann das inverse Suffix-Array verwendet werden. Dieses ist wie folgt definiert:

Definition 2.5 (Inverses Suffix-Array) *Ist SA das Suffix-Array eines Strings S, so heißt SA^{-1} das inverse Suffix-Array, falls für $1 \leq i \leq n$ gilt $SA^{-1}[SA[i]] = i$.*

Bei dem inversen Suffix-Array handelt es sich somit um die inverse Permutation des Suffix-Arrays.

i	$SA[i]$	$SA[i]^{-1}$	$S_{SA[i]}$
1	13	4	`$`
2	6	11	`annas$`
3	8	9	`nas$`
4	1	6	`annasanannas$`
5	11	13	`as$`
6	4	2	`asanannas$`
7	7	7	`nannas$`
8	10	3	`nas$`
9	3	10	`nasanannas$`
10	9	8	`nnas$`
11	2	5	`nnasanannas$`
12	12	12	`s$`
13	5	1	`sanannas$`

Tabelle 2.1: Das Suffix-Array, das inverse Suffix-Array sowie alle Suffixe des Strings `annasanannas$`. Zusätzlich sind vier ω-Intervalle eingezeichnet.

Beispiel 2.6 *Tabelle 2.1 zeigt in der ersten Spalte das Suffix-Array*

$$SA[1 \ldots 13] = (13, 6, 8, 1, 11, 4, 7, 10, 3, 9, 2, 12, 5)$$

des Strings $S =$ `annasanannas$`. Das inverse Suffix-Array

$$SA^{-1}[1 \ldots 13] = (4, 11, 9, 6, 13, 2, 7, 3, 10, 8, 5, 12, 1)$$

steht in der zweiten Spalte. Aus dem Suffix-Array kann abgelesen werden, dass das zweitkleinste Suffix von S an Position $6 = SA[2]$ beginnt. Dagegen gibt es zehn kleinere Suffixe als S_2, welches daher an Position $11 = SA^{-1}[2]$ steht.

Kommt ein Teilstring mehrfach in einem String vor, so stehen diese Teilstrings aufgrund der lexikografischen Ordnung der Suffixe im Suffix-Array hintereinander.

Beispiel 2.7 *Im String* **annasanannas$** *kommt der Teilstring* **an** *an den Positionen* 1, 6 *und* 8 *vor. Die drei zugehörenden Teilstrings* S_1, S_6 *und* S_8 *stehen im Suffix-Array entsprechend hintereinander an Position* 2, 3 *und* 4.

Diese Beobachtung führt zu der folgenden Definition.

Definition 2.8 (ω-Intervall) *Sei* S *ein String der Länge* n *und* SA *das zugehörige Suffix-Array. Ein Intervall* $[i \dots j]$ *heißt* ω*-Intervall, falls die folgenden drei Bedingungen erfüllt sind:*

1. $\exists k \in \{1, \dots, n\} : \omega$ *ist ein Präfix von* $S_{SA[k]}$.
2. $i = \min\{k \mid 1 \leq k \leq n$ *und* ω *ist ein Präfix von* $S_{SA[k]}\}$
3. $j = \max\{k \mid 1 \leq k \leq n$ *und* ω *ist ein Präfix von* $S_{SA[k]}\}$

Unter der Präfixlänge eines ω-Intervalls $[i \dots j]$ verstehen wir $|\omega|$, wohingegen wir die Anzahl der $j - i + 1$ Teilstrings im Intervall als Intervallgröße bezeichnen. Definition 2.8 gibt für jeden Teilstring aus S ein eindeutiges Intervall an. Andererseits ist nicht jedes Intervall ein ω-Intervall und einem Intervall können auch mehrere ω-Intervalle zugeordnet sein.

Beispiel 2.9 *Das* **an***-Intervall des Strings* $S =$ **annasanannas$** *ist das Intervall* $[2 \dots 4]$. *Die Präfixlänge beträgt* $2 = |$**an**$|$ *und die Intervallgröße ist* $3 = 4 - 2 + 1$. *Das Intervall* $[9 \dots 11]$ *ist in* S *kein* ω*-Intervall, wohingegen* $[10 \dots 11]$ *sowohl ein* **nn***-Intervall als auch ein* **nna***-Intervall und ein* **nnas***-Intervall ist. In Tabelle 2.1 sind die* ω*-Intervalle für* $[10 \dots 11]$ *und das* **an***-Intervall* $[2 \dots 4]$ *eingezeichnet.*

2.3 BWT und LCP-Array

In [BW94] beschreiben Burrows und Wheeler erstmals die nach ihnen benannte Burrows-Wheeler-Transformierte (BWT). Die Transformation hat die Eigenschaft, oft gleiche Buchstaben hintereinander zu reihen und wurde daher zunächst als Vorstufe von Kompressionsalgorithmen vorgeschlagen.

Definition 2.10 (Burrows-Wheeler-Transformierte) *Ist S ein String und SA das zugehörige Suffix-Array, so ist die Burrows-Wheeler-Transformierte definiert als:*

$$BWT[i] = \begin{cases} S[SA[i]-1] & \text{falls } SA[i] \neq 1 \\ \$ & \text{falls } SA[i] = 1 \end{cases}$$

Die BWT eines Strings S ist eine Permutation von S und benötigt daher mit $n \log_2 \sigma$ Bits wesentlich weniger Speicherplatz als das Suffix-Array.

i	$SA[i]$	$SA[i]^{-1}$	$BWT[i]$	$LCP[i]$	$S_{SA[i]}$
1	13	4	s	-1	$
2	6	11	s	0	anannas$
3	8	9	n	2	annas$
4	1	6	$	5	annasanannas$
5	11	13	n	1	as$
6	4	2	n	2	asanannas$
7	7	7	a	0	nannas$
8	10	3	n	2	nas$
9	3	10	n	3	nasanannas$
10	9	8	a	1	nnas$
11	2	5	a	4	nnasanannas$
12	12	12	a	0	s$
13	5	1	a	1	sanannas$
14				-1	

Tabelle 2.2: Das Suffix-Array, das inverse Suffix-Array, die BWT und das LCP-Array des Strings `annasanannas$`.

Beispiel 2.11 *Die Tabelle 2.2 ist gegenüber der Tabelle 2.1 um die BWT und das LCP-Array des Strings $S =$ **annasanannas$** erweitert. Die BWT von S ist **ssn$nnannaaaa**, wie an der vierten Spalte abgelesen werden kann.*

Das längste gemeinsame Präfix (englisch: *longest common prefix*, kurz *lcp*) zweier Strings S und T ist das längste Präfix von S, welches auch Präfix von T ist. Die *lcp*-Funktion berechnet die Länge des längsten gemeinsamen Präfixes zweier Strings und wird in 2.12 definiert.

Definition 2.12 (*lcp*-Funktion) *Für beliebige Strings S und T ist die lcp-Funktion wie folgt definiert:*

$$lcp(S,T) = \begin{cases} 0 & \text{falls } S = \epsilon \text{ oder } T = \epsilon \text{ oder } S[1] \neq T[1] \\ 1 + lcp(S_2, T_2) & \text{sonst} \end{cases}$$

Im LCP-Array stehen jeweils die Ergebnisse der *lcp*-Funktion von zwei im Suffix-Array aufeinanderfolgenden Suffixen. Wir geben nachfolgend eine formale Definition.

Definition 2.13 (LCP-Array) *Sei S ein String mit Suffix-Array SA. Ein Array LCP heißt LCP-Array, falls die folgenden Bedingungen erfüllt sind:*

1. $LCP[1] = -1$
2. $LCP[i] = lcp(S_{SA[i-1]}, S_{SA[i]})$ *für* $1 < i \leq n$
3. $LCP[n+1] = -1$

Im Gegensatz zum Suffix-Array und inverse Suffix-Array ist das LCP-Array keine Permutation der Zahlen von 1 bis n. Da aber nicht ausgeschlossen werden kann, dass es sehr große LCP-Werte gibt, benötigt die Speicherung des LCP-Arrays unkomprimiert ebenfalls $n \log_2 n$ Bits.

Beispiel 2.14 *Für den String $S =$ `annasanannas$` ist das LCP-Array in der fünften Spalte von Tabelle 2.2 aufgeführt. Die zwei Suffixe S_6 und S_8 stehen im Suffix-Array unmittelbar nebeneinander an den Positionen 2 und 3. Da `an` das längste gemeinsame Präfix von $S_6 =$ `annas$` und $S_8 =$ `nas$` ist, gilt $LCP[3] = |$`an`$| = 2$.*

Abschließend zeigt Lemma 2.15, wie mittels dem LCP-Array die Länge des längsten gemeinsamen Präfixes zweier beliebiger Suffixe berechnet werden kann.

Lemma 2.15 *Ist S ein String der Länge n, so gilt für alle $i, j \in \{1, \ldots, n\}$ mit $i < j$:*

$$lcp(S_{SA[i]}, S_{SA[j]}) = \min\{LCP[k] \mid i < k \leq j\}$$

Beweis: Für $i = j-1$ ist die Aussage des Lemmas $lcp(S_{SA[j-1]}, S_{SA[j]}) = LCP[j]$ und entspricht damit gerade der Definition 2.13. Sei im Folgenden also $i < j-1$ sowie $A = S_{SA[i]}$ und $C = S_{SA[j]}$. Für Buchstaben $a, c \in \Sigma$, $a \neq c$ und entsprechenden Strings ω, A', C' lassen sich A und C darstellen als $A = \omega a A'$ und $C = \omega c C'$. Da für $B = S_{SA[i+1]}$ gilt $A <_{lex} B <_{lex} C$, muss ω auch ein Präfix von B sein. Folglich kann B zerlegt werden in $B = \omega b B'$ und es gilt:

1. $lcp(A, B) \geq lcp(A, C)$
2. $lcp(B, C) \geq lcp(A, C)$
3. Falls $a \neq b$, so gilt $lcp(A, B) = lcp(A, C)$.
4. Falls $a = b$, so gilt $b \neq c$ und damit $lcp(B, C) = lcp(A, C)$.

Aus diesen Bedingungen folgt $lcp(A, C) = \min\{lcp(A, B), lcp(B, C)\}$.

Somit gilt $lcp(S_{SA[i]}, S_{SA[j]}) = \min\{\overbrace{lcp(S_{SA[i]}, S_{SA[i+1]})}^{LCP[i+1]}, lcp(S_{SA[i+1]}, S_{SA[j]})\}$, woraus sich die folgenden Gleichungen ergeben:

$$\begin{aligned}
lcp(S_{SA[i]}, S_{SA[j]}) &= \min\{LCP[i+1], lcp(S_{SA[i+1]}, S_{SA[j]})\} \\
lcp(S_{SA[i+1]}, S_{SA[j]}) &= \min\{LCP[i+2], lcp(S_{SA[i+2]}, S_{SA[j]})\} \\
lcp(S_{SA[i+2]}, S_{SA[j]}) &= \min\{LCP[i+3], lcp(S_{SA[i+3]}, S_{SA[j]})\} \\
&\vdots \\
lcp(S_{SA[j-2]}, S_{SA[j]}) &= \min\{LCP[j-1], \underbrace{lcp(S_{SA[j-1]}, S_{SA[j]})}_{LCP[j]}\}
\end{aligned}$$

Aufgrund der Tatsache, dass $\min\{a, \min\{b, c\}\} = \min\{a, b, c\}$ ist, können die obigen Gleichungen nun ineinander eingesetzt werden, woraus das Lemma 2.15 folgt.

□

Beispiel 2.16 *Die zwei Suffixe S_7 und S_2 des Strings* `annasanannas$` *stehen im Suffix-Array an den Positionen* 7 *und* 11*, wie Tabelle 2.2 zeigt. Somit ist $lcp(S_7, S_2)$ das Minimum von $\{LCP[8], LCP[9], LCP[10], LCP[11]\} = \{2, 3, 1, 4\}$ also* 1*.*

2.4 Wavelet Tree

Der Wavelet Tree ist ein Binärbaum, welcher einen String in einer speziellen Baumstruktur speichert und dadurch verschiedene Abfragen auf diesem ermöglicht. Er wurde erstmals in [GGV03] vorgestellt. In [MN05] wird eine speichereffiziente Variante vorgeschlagen. Die folgende Definition gilt in beiden Fällen.

Definition 2.17 (Wavelet Tree) *Sei S ein String über dem Alphabet Σ. Ein Wavelet Tree ist ein Binärbaum, bei dem jedem Knoten eine Buchstabenmenge $M \subseteq \Sigma$ zugeordnet ist. Darüber hinaus gilt:*

1. *Dem Wurzelknoten ist Σ zugeordnet.*
2. *Falls $|M| > 1$ ist, so sei $M_0 \neq \emptyset$ und $M_1 \neq \emptyset$ eine disjunkte Zerlegung von M. Der Knoten enthält dann einen Bitarray B, bei dem $B[i] = 0$ ist, falls $S^M[i] \in M_0$ ist; andernfalls ist $B[i] = 1$. Der Knoten hat zudem zwei Kinder, denen M_0 bzw. M_1 zugeordnet sind.*
3. *Falls $|M| = 1$ ist, so enthält der Knoten nur den Buchstaben aus M und hat keine Kinder.*

Die Definition 2.17 lässt offen, wie die Zerlegung der Buchstabenmenge erfolgen soll. Abbildung 2.1 zeigt einen möglichen Wavelet Tree für den String `annasanannas$`.

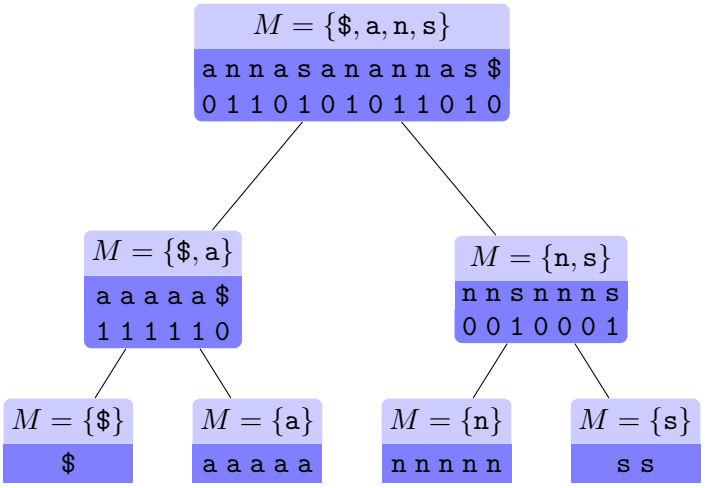


Abbildung 2.1: Balancierter Wavelet Tree für den String `annasanannas$`. Der Text in den inneren Knoten dient nur dem Verständnis und ist nicht Bestandteil des Wavelet Trees.

Bei dem Wavelet Tree in Abbildung 2.1 wurde M in jedem Knoten halbiert. Jeder Wavelet Tree mit dieser Zerlegungsart hat eine Tiefe von $\lceil \log_2 \sigma \rceil$ und einen Speicherbedarf von $n \log_2 \sigma$ Bits, da in jeder Ebene n Bits gespeichert werden.

Die Rank-Anfrage $rank_c(i)$ liefert die Anzahl der Vorkommen des Buchstabens c in $S[1 \dots i]$. Dies kann mit dem Wavelet Tree des Strings S und der Funktion *rank*(c, *node*, i) effizient berechnet werden. Hierbei ermittelt die Funktion *rank*(c, *node*, i), wie oft der Buchstabe c in $S^M[1 \dots i]$ vorkommt, wobei M die zugeordnete Buchstabenmenge von *node*, einem Knoten des Wavelet Trees, ist. Da dem Wurzelknoten *root* des Wavelet Trees gerade Σ zugeordnet und $S^\Sigma = S$ ist, kann $rank_c(i)$ mithilfe des Funktionsaufrufs *rank*(c, *root*, i) berechnet werden.

Algorithmus 1
Sei WT der Wavelet Tree des Strings S und *node* ein Knoten von WT mit zugeordneter Buchstabenmenge M. Dann berechnet die Funktion *rank*(c, *node*, i), wie oft c in $S^M[1 \dots i]$ vorkommt.

```
if |M| = 1 then
  return i
else
  if c ∈ M_0 then
    i ← rank_0(node, i)
    node ← leftChild(node)
    return rank(c, node, i)
  else
    j ← rank_1(node, i)
    node ← rightChild(node)
    return rank(c, node, i)
  end if
end if
```

Algorithmus 1 zeigt die Implementierung der Funktion *rank*(c, *node*, i), während Beispiel 2.18 die Funktionsweise von Algorithmus 1 anhand einer konkreten Rank-Anfrage veranschaulicht.

Beispiel 2.18 *Es soll die Anfrage* $\text{rank}_{\texttt{n}}(6)$ *mittels Algorithmus 1 auf dem Wavelet Tree aus Abbildung 2.1 beantwortet werden, das heißt man zählt die Vorkommen von* **n** *im String* $S =$ **annasanannas$** *bis zur Position* 6. *Da sich* **n** *in* M_1 *des Wurzelknotens befindet, wird mit* $\text{rank}_1(root, 6)$ *zunächst berechnet, wie viele Einsen im Bitarray bis zu der Stelle* 6 *vorkommen. Dies ist dreimal der Fall. Nun erfolgt der rekursive Abstieg in das rechte Kind. Hier befindet sich* **n** *in der Buchstabenmenge* M_0. *Es werden daher die Nullen bis zu Position* 3 *gezählt. Da es zwei Nullen gibt und das linke Kind ein Blatt ist, kommt* **n** *in* $S[1 \dots 6]$ *genau zweimal vor.*

Von Algorithmus 1 werden zur Berechnung einer $rank_c(i)$-Anfrage alle Knoten von der Wurzel bis zu demjenigen Kind, welchem der Buchstabe c zugeordnet ist, besucht. Dabei wird bei jedem Knoten mit der Funktion $rank_b(node, i)$ die Anzahl des Bits b im Bitarray des Knotens $node$ bis zur Position i gezählt. Die Bitarrays in den inneren Knoten des Wavelet Trees werden daher initial so vorverarbeitet, dass die Funktion $rank_b(node, i)$ in konstanter Zeit beantwortet werden kann [Vig08]. Ist die Tiefe durch $\lceil \log_2 \sigma \rceil$ beschränkt, so beträgt die Zeitkomplexität $\lceil \log_2 \sigma \rceil$, um die Vorkommen eines beliebigen Buchstabens bis zu einer beliebigen Position zu zählen.

In manchen Fällen werden Rank-Anfragen für Buchstaben, die oft im String auftreten, häufiger gestellt als für Buchstaben, die selten vorkommen. In diesem Fall ist es günstig, wenn die häufig auftretenden Buchstaben einen besonders kurzen Pfad im Wavelet Tree haben. Da Definition 2.17 die Aufteilung der Buchstaben nicht festlegt, kann diese nach dem Huffman-Schema [Huf52] erfolgen. Dieses minimiert die Summe $\sum_{c \in \Sigma} C[c] \cdot T(c)$ und damit die Laufzeit, für den Fall, dass für jedes Vorkommen von c eine Rank-Anfrage für c gestellt wird. Die Funktion $T(c)$ gibt dabei die Tiefe des Buchstabens c im Wavelet Tree an. Abbildung 2.2 zeigt eine solche Knotenaufteilung für den String `annasanannas$`.

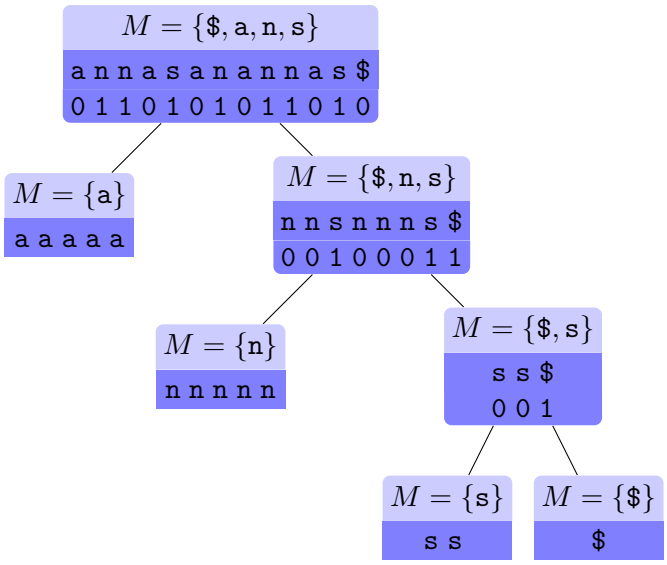


Abbildung 2.2: Der Wavelet Tree für den String `annasanannas$` nach dem Huffman-Schema. Der Text in den inneren Knoten dient nur dem Verständnis und wird in einer praktischen Implementierung nicht gespeichert.

Der Wavelet Tree unterstützt neben der $rank_c(i)$-Anfrage normalerweise auch die $select_c(i)$-Abfrage, welche die Position zurückgibt, an welcher der Buchstabe c das i-te Mal auftritt. Diese Funktionalität wird im Laufe dieser Arbeit allerdings nicht benötigt. Wir gehen stattdessen auf die LF-Funktion ein, welche mit einer $rank_c(i)$-Anfrage berechnet werden kann.

2.5 *LF*-Funktion und Ψ-Funktion

Die LF-Funktion gibt für ein Suffix S_i diejenige Stelle im Suffix-Array zurück, an welcher das um einen Buchstaben längere Suffix S_{i-1} steht. Wir geben zunächst eine formale Definition dieser Funktion.

Definition 2.19 (*LF*-Funktion) *Ist SA das Suffix-Array eines Strings S der Länge n. Dann ist die LF-Funktion wie folgt definiert:*

$$LF(i) = \begin{cases} SA^{-1}[SA[i]-1] & \textit{falls } SA[i] > 1 \\ SA^{-1}[n] & \textit{falls } SA[i] = 1 \end{cases}$$

LF steht hierbei für *Last-To-Front*. Diese Bezeichnung wird deutlich, indem man die Suffixe S_i und $S_{LF(i)}$ betrachtet. Man erhält das Suffix $S_{LF(i)}$ nämlich, indem man den Buchstaben $BWT[i]$ an das Suffix S_i voranstellt. Dabei entspricht $BWT[i]$ dem letzten Buchstaben des um $SA[i]$ Positionen zyklisch rotierten Strings S.

Da die LF-Funktion nur auf den Zahlen von 1 bis n definiert ist, ist das LF-Array mit $LF[i] = LF(i)$ äquivalent zur LF-Funktion. Im Folgenden wird daher nicht weiter zwischen der LF-Funktion und dem LF-Array unterschieden. Die vollständige Berechnung des LF-Arrays ist – wie Algorithmus 2 zeigt – in linearer Zeit möglich.

Algorithmus 2
Die Berechnung des LF-Arrays aus der BWT in $\mathcal{O}(n)$.

```
for i ← 1 to n do
    c ← BWT[i]
    C[c] ← C[c] + 1
    LF[i] ← C[c]
end for
```

Möchte man hingegen den Funktionswert von $LF(i)$ berechnen, ohne zuvor alle LF-Werte von 1 bis $i-1$ bestimmen zu müssen, kann dies mit Hilfe des Wavelet Trees für

die BWT in einer Zeitkomplexität von $\mathcal{O}(\log \sigma)$ pro Eintrag erfolgen. Aus Algorithmus 2 folgt nämlich unmittelbar, dass $LF(i) = C[c]+1+rank_{BWT[i]}(i)$ ist. Berechnet man auf diese Weise alle n Einträge des LF-Arrays, so beträgt der Rechenaufwand allerdings $\mathcal{O}(n \log \sigma)$.

Beispiel 2.20 *Das Suffix $S_{11} =$ `as$` steht – wie man Tabelle 2.3 entnehmen kann – im Suffix-Array an der Stelle 5. Im LF-Array steht an Position 5 nun, dass das Suffix $S_{10} =$ `nas$` an Position $8 = LF[5]$ gefunden werden kann. Analog gibt $LF[8] = 10$ die Stelle des Suffixes $S_9 =$ `nnas$` an. Folgt man also dem LF-Array, so erhält man nacheinander die Positionen der Suffixe S_n, S_{n-1}, S_{n-2}, ..., S_2, S_1, S_n, S_{n-1}, ...*

i	$SA[i]$	$SA^{-1}[i]$	$BWT[i]$	$LCP[i]$	$LF[i]$	$\Psi[i]$	$S_{SA[i]}$
1	13	4	s	-1	12	4	$
2	6	11	s	0	13	7	anannas$
3	8	9	n	2	7	10	annas$
4	1	6	$	5	1	11	annasanannas$
5	11	13	n	1	8	12	as$
6	4	2	n	2	9	13	asanannas$
7	7	7	a	0	2	3	nannas$
8	10	3	n	2	10	5	nas$
9	3	10	n	3	11	6	nasanannas$
10	9	8	a	1	3	8	nnas$
11	2	5	a	4	4	9	nnasanannas$
12	12	12	a	0	5	1	s$
13	5	1	a	1	6	2	sanannas$
14				-1			

Tabelle 2.3: Das Suffix-Array, das inverse Suffix-Array, die BWT, das LCP-Array, das LF-Array und das Ψ-Array des Strings `annasanannas$`.

Mittels der LF-Funktion lassen sich reduzible LCP-Werte einfach berechnen. Die nachfolgende Definition erklärt zunächst diesen Begriff.

Definition 2.21 (reduzibler LCP-Wert) *Gilt $BWT[i] = BWT[i-1]$, so heißt $LCP[i]$ reduzibel, andernfalls heißt $LCP[i]$ irreduzibel.*

Während die Anzahl reduzibler LCP-Werte von String zu String stark variieren kann, konnte in [KMP09] bewiesen werden, dass die Summe aller irreduzibelen LCP-Werte höchsten $2n \log_2 n$ beträgt.

Beispiel 2.22 *In Tabelle 2.3 sind innerhalb des LCP-Arrays alle reduzible LCP-Werte markiert. Beispielsweise ist* $LCP[11]$ *reduzibel, weil* $BWT[11] = BWT[10] =$ `a` *ist. Analog ist* $LCP[13]$ *reduzibel, da* $BWT[13] = BWT[12]$ *gilt.*

Das folgende Lemma zeigt nun, wie reduzible LCP-Werte berechnet werden können.

Lemma 2.23 *Ist* $LCP[i]$ *reduzibel, so gilt* $LCP[i] = LCP[LF[i]] - 1$.

Anschaulich ist die Aussage des Lemmas klar. Denn $LCP[LF[i]]$ gibt die Länge des längsten gemeinsamen Suffixes von $S_{SA[LF[i]-1]}$ und $S_{SA[LF[i]]}$ an. Die um einen Buchstaben kürzeren Suffixe $S_{SA[LF[i]]+1} = S_{SA[i]}$ und $S_{SA[LF[i]-1]+1}$ haben folglich $LCP[LF[i]] - 1$ Buchstaben gemeinsam. Da $BWT[i-1] = BWT[i]$ gilt, stehen diese verkürzten Suffixe im Suffix Array direkt hintereinander, womit die Aussage des Lemmas folgt.

Beweis: Für den Beweis wird zunächst die Gleichung $S_{SA[LF[i]]} = BWT[i]S_{SA[i]}$ benötigt. Diese folgt für $SA[i] > 1$ unmittelbar aus der Definition der LF-Funktion und der BWT:

$$\begin{aligned} S_{SA[LF[i]]} &= S_{SA[SA^{-1}[SA[i]-1]]} \\ &= S_{SA[i]-1} \\ &= S[SA[i]-1]S_{SA[i]} \\ &= BWT[i]S_{SA[i]} \end{aligned}$$

Im Folgenden soll gezeigt werden, dass aus $BWT[i-1] = BWT[i]$ die Gleichung $SA[LF[i-1]] = SA[LF[i]-1]$ folgt, also dass das Suffix $SA[LF[i-1]]$ im Suffix-Array direkt vor dem Suffix $SA[LF[i]]$ steht. Dies ist der Fall, wenn es kein Suffix S_k mit $S_{SA[LF[i-1]]} <_{lex} S_k <_{lex} S_{SA[LF[i]]}$ gibt. Wenn es ein solches Suffix S_k gäbe, so würde gelten:

$$\begin{aligned} S_{SA[LF[i-1]]} &<_{lex} S_k &<_{lex}& S_{SA[LF[i]]} \\ BWT[i-1]S_{SA[i-1]} &<_{lex} S_k &<_{lex}& BWT[i]S_{SA[i]} \\ S_{SA[i-1]} &<_{lex} S_{k+1} &<_{lex}& S_{SA[i]} \end{aligned}$$

Die letzte Zeile steht aber im Widerspruch zur Definition des Suffix-Arrays. Somit muss $SA[LF[i-1]] = SA[LF[i]-1]$ gelten.

Nun kann $LCP[LF[i]]$ wie folgt umgeformt werden:

$$\begin{aligned} LCP[LF[i]] &= lcp(S_{SA[LF[i]-1]}, S_{SA[LF[i]]}) \\ &= lcp(S_{SA[LF[i-1]]}, S_{SA[LF[i]]}) \\ &= lcp(BWT[i-1]S_{SA[i-1]}, BWT[i]S_{SA[i]}) \\ &= 1 + lcp(S_{SA[i-1]}, S_{SA[i]}) \\ &= 1 + LCP[i] \end{aligned}$$

Daraus folgt die Aussage des Lemmas. □

Beispiel 2.24 *In Tabelle 2.3 ist die Aussage des Lemmas 2.23 an den beiden reduziblen LCP-Werten* $LCP[11]$ *und* $LCP[13]$ *dargestellt. Es gilt also* $LCP[11] = LCP[LF[11]] - 1 = LCP[4] - 1 = 5 - 1 = 4$ *bzw.* $LCP[13] = LCP[LF[13]] - 1 = LCP[6] - 1 = 2 - 1 = 1$.

Die LF-Funktion gibt für ein Suffix S_i die Stelle im Suffix-Array zurück, an welcher das um einen Buchstaben längere Suffix S_{i-1} steht. Im Gegensatz dazu gibt die Ψ-Funktion die Stelle im Suffix-Array zurück, an welcher das um einen Buchstaben kürzere Suffix S_{i+1} steht.

Definition 2.25 (Ψ-Funktion) *Sei SA das Suffix-Array eines Strings S der Länge n. Dann ist die Ψ-Funktion wie folgt definiert:*

$$\Psi(i) = \begin{cases} SA^{-1}[SA[i]+1] & \text{falls } SA[i] < n \\ SA^{-1}[1] & \text{falls } SA[i] = n \end{cases}$$

Die Ψ-Funktion ist wie die LF-Funktion einer Permutation der Zahlen von 1 bis n. Genauer gesagt, ist die Ψ-Funktion die inverse Permutation der LF-Funktion, wie folgendes Lemma zeigt.

Lemma 2.26 *Für alle $i \in \{1, \ldots, n\}$ gilt $\Psi[LF[i]] = i$.*

Das Lemma kann durch einfaches einsetzen der Definitionen 2.19 und 2.25 nachgerechnet werden.

Beweis: Fall $SA[i] > 1$:

$$\begin{aligned}\Psi[LF[i]] &= \Psi[SA^{-1}[SA[i]-1]] \\ &= SA^{-1}[SA[SA^{-1}[SA[i]-1]]+1] \\ &= SA^{-1}[SA[i]-1+1] \\ &= i\end{aligned}$$

Fall $SA[i] = 1$:

$$\begin{aligned}\Psi[LF[i]] &= \Psi[SA^{-1}[n]] \\ &= SA^{-1}[1] \\ &= i\end{aligned}$$

□

Wie die LF-Funktion kann auch die Ψ-Funktion mittels Wavelet Tree in $\mathcal{O}(\log \sigma)$ berechnet werden. Hierzu wird auf [NM07] verwiesen.

3

Herkömmliche Berechnung des Longest-Common-Prefix-Arrays

Manber und Myers führten 1990 in [MM90] nicht nur das Suffix-Array, sondern auch das LCP-Array ein. Sie schlugen einen Suffix-Array-Konstruktionsalgorithmus vor, der eine Zeitkomplexität von $\mathcal{O}(n \log n)$ besitzt und das LCP-Array als Nebenprodukt berechnen kann.

2003 stellten drei Forschergruppen unabhängig voneinander einen linearen Algorithmus zur Konstruktion des Suffix-Arrays vor [KS03, KSPP03, KA03]. Der in [KS03] vorgeschlagene Algorithmus kann ebenfalls so modifiziert werden, dass er das LCP-Array ebenfalls zusammen mit dem Suffix-Array berechnen.

Die derzeit schnellsten LCP-Array Konstruktionsalgorithmen berechnen allerdings das LCP-Array nicht gleichzeitig mit, sondern – in einem zweiten Schritt – aus dem Suffix-Array. In diesem Kapitel werden drei der derzeit schnellsten LCP-Array Konstruktionsalgorithmen vorgestellt.

3.1 *KLAAP*-Algorithmus

Bereits 2001 stellten Kasai et al. in [KLA+01] mit *KLAAP* einen schnellen und einfachen Algorithmus zur Berechnung des LCP-Arrays aus dem Suffix-Array vor. Dabei wird das LCP-Array nicht sequenziell, sondern in Suffix-Array Reihenfolge berechnet. Der *KLAAP*-Algorithmus vergleicht also die Suffixe S_1, S_2, S_3, ..., S_n Buchstabenweise mit den im Suffix-Array unmittelbar davor stehenden Suffixen. Hierzu wird zusätzlich zum Suffix-Array und dem String, auch das inverse Suffix-Array benötigt.

Algorithmus 3 zeigt die ursprüngliche Formulierung dieses Algorithmus.

Algorithmus 3
Der *KLAAP*-Algorithmus zur Berechnung des LCP-Arrays.

```
for i ← 1 to n do
  SA^{-1}[SA[i]] ← i
end for
lcp ← 0
for i ← 1 to n do
  if SA^{-1}[i] > 1 then
    k ← SA[SA^{-1}[i] − 1]
    while S[i + lcp] = S[k + lcp] do
      lcp ← lcp + 1
    end while
    LCP[SA^{-1}[i]] ← lcp
    if lcp > 0 then
      lcp ← lcp − 1
    end if
  end if
end for
```

Algorithmus 3 berechnet zunächst das inverse Suffix-Array. Anschließend durchläuft die FOR-Schleife die Suffixe S_1, S_2, S_3, ..., S_n. In der WHILE-Schleife wird dabei jeweils gezählt, wie viele Buchstaben des Suffixes S_i mit dem im Suffix-Array unmittelbar davor stehenden Suffix $S_{SA[SA^{-1}[i]-1]}$ übereinstimmen. Die Schlüsselidee des *KLAAP*-Algorithmus besteht darin, dass hierbei die ersten $LCP[SA^{-1}[i-1]] - 1$ Buchstaben übersprungen werden können. Diesem Trick verdankt der Algorithmus seine lineare Laufzeit. Lemma 3.1 erklärt, warum dies möglich ist.

Lemma 3.1 *Sei SA das Suffix-Array eines Strings S der Länge n. Dann gilt für alle $i \in \{1, \ldots, n-1\}$ die Ungleichung:*

$$LCP[SA^{-1}[i+1]] \geq LCP[SA^{-1}[i]] - 1$$

Für den Beweis des Lemmas setzen wir $b = SA^{-1}[i]$ und $a = b-1$. Die Suffixe $S_{SA[a]}$ und $S_{SA[b]} = S_i$ folgen im Suffix-Array also unmittelbar aufeinander und stimmen in den ersten $l = LCP[b]$ Buchstaben überein. Ferner setzen wir $b' = SA^{-1}[i+1]$ und $a' = SA[SA^{-1}[a] - 1]$. Die Suffixe $S_{SA[b']}$ und $S_{SA[a']}$ erhält man so aus $S_{SA[b]}$ und $S_{SA[a]}$ durch das Entfernen des jeweils ersten Buchstabens.

Beweis: Für $l = 0$ bzw. $l = 1$ ist $LCP[SA^{-1}[i+1]] \geq -1$ bzw. $LCP[SA^{-1}[i+1]] \geq 0$ und damit die Aussage des Lemmas trivialerweise erfüllt. Sei also $l > 1$. Dann stimmen die Suffixe $S_{SA[a]}$ und $S_{SA[b]}$ an den ersten l Positionen überein. Folglich

ist $lcp(S_{SA[a']}, S_{SA[b']}) = l - 1$. Darüber hinaus gilt aufgrund der lexikografischen Ordnung, dass $S_{SA[a']} <_{lex} S_{SA[b']}$ und somit $a' < b'$ ist. Es kann also Lemma 2.15 angewandt werden, womit $LCP[b'] \geq l - 1$ folgt. Ersetzt man in dieser Gleichung b' durch $SA^{-1}[i+1]$ und l durch $LCP[b] = LCP[SA^{-1}[i]]$, so folgt das Lemma. □

In Beispiel 3.2 wird das Lemma 3.1 bzw. Algorithmus 3 an einem konkreten String erläutert.

i	$SA[i]$	$SA^{-1}[i]$	$LCP[i]$	$S_{SA[i]}$
1	13	4	-1	`$`
2	6	11	0	`anannas$`
3	8	9	2	`annas$`
4	1	6	5	`annasanannas$`
5	11	13	1	`as$`
6	4	2	2	`asanannas$`
7	7	7	0	`nannas$`
8	10	3	2	`nas$`
9	3	10	3	`nasanannas$`
10	9	8	1	`nnas$`
11	2	5	4	`nnasanannas$`
12	12	12	0	`s$`
13	5	1	1	`sanannas$`
14			-1	

Tabelle 3.1: Das Suffix-Array, das inverse Suffix-Array und das LCP-Array des Strings `annasanannas$`.

Beispiel 3.2 *Tabelle 3.1 zeigt die von Algorithmus 3 benötigten Datenstrukturen für den String* `annasanannas$`*. Nachdem der* KLAAP*-Algorithmus das inverse Suffix-Array SA^{-1} berechnet hat, wird der LCP-Eintrag von $SA^{-1}[1] = 4$ bestimmt. Hierzu werden die Suffixe S_1 und S_8 in der* While*-Schleife Zeichen für Zeichen verglichen, bis an der sechsten Position die zwei unterschiedlichen Buchstaben* `$` *und* `a` *festgestellt werden. Damit steht $LCP[4] = 5$ fest und gemäß Lemma 3.1 ist der nächste LCP-Wert mindestens vier. Anschließend wird die Variable lcp um eins verringert. Im nächsten Durchlauf der* For*-Schleife wird der LCP-Wert an der Stelle $SA^{-1}[2] = 11$ berechnet. Beim Vergleichen der Suffixe S_2 und S_9 können dann die ersten vier Zeichen übersprungen werden.*

Wir erläutern nun, warum der *KLAAP*-Algorithmus eine lineare Laufzeit besitzt. Die erste For-Schleife, welche das inverse Suffix-Array berechnet, wird ebenso wie die zweite For-Schleife genau n-mal durchlaufen. Interessanterweise hat auch die

While-Schleife im Innern der zweiten For-Schleife eine lineare Laufzeit. Dies liegt daran, dass die Variable *lcp* stets kleiner als n sein muss. Da *lcp* darüber hinaus höchstens n mal dekrementiert wird, kann die Variable höchstens $2n$-mal inkrementiert werden. Somit wird die While-Schleife während des gesamten Algorithmus höchstens $2n$ mal durchlaufen. Folglich benötigt der *KLAAP*-Algorithmus nur $2n$ Buchstabenvergleiche, um in einer Zeitkomplexität von $\mathcal{O}(n)$ das gesamte LCP-Array zu berechnen.

Die Nachteile des *KLAAP*-Algorithmus sind sowohl sein hoher Speicherbedarf von $13n$ Bytes als auch seine schlechte Lokalität. Zwar wurde in [Man04] gezeigt, dass der Speicherverbrauch etwas gesenkt werden kann, indem die Werte des inversen Suffix-Arrays mit den LCP-Werten überschrieben werden – der benötigte Speicher bleibt mit $9n$ Bytes jedoch noch immer erheblich.

3.2 Φ-Algorithmen

Kärkkäinen et al. schlugen 2009 in [KMP09] eine Familie neuer LCP-Konstruktionsalgorithmen vor – die sogenannten Φ-Algorithmen. Diese zeichnet allesamt aus, dass sie zuerst das PLCP-Array (englisch: *permuted longest common prefix*, kurz *plcp*) berechnen. Dieses ist für $1 \leq j \leq n$ wie folgt definiert: $PLCP[j] = LCP[SA^{-1}[j]]$ bzw. $PLCP[SA[j]] = LCP[j]$. Es handelt sich beim PLCP-Array also – wie der Name bereits vermuten lässt – um eine Permutation des LCP-Arrays.

Im ersten Schritt erzeugen die Φ-Algorithmen zunächst das Φ-Array, dem die Algorithmenfamilie ihren Namen verdankt. Im nächsten Schritt wird das PLCP-Array berechnet, wobei diese Berechnung ähnlich dem *KLAAP*-Algorithmus ist. Zuletzt wird aus dem PLCP-Array das LCP-Array konstruiert. Ein Pseudocode dieses Grundschemas findet sich in Algorithmus 4.

Der Hauptunterschied zwischen den Φ-Algorithmen und dem *KLAAP*-Algorithmus besteht darin, dass bei ersteren die berechneten LCP-Werte sequenziell abgespeichert werden können. Es wird also zunächst das PLCP-Array erzeugt, welches abschließend in die richtige Reihenfolge gebracht werden muss. Dieses Vorgehen hat Geschwindigkeitsvorteile, es ermöglicht darüber hinaus aber auch eine Reduzierung des Hauptspeicherbedarfs.

Algorithmus 4 verwendet neben dem Text und dem Suffix-Array zusätzlich das Φ-Array, das PLCP-Array und das LCP-Array. Allerdings kann das Φ-Array mit dem PLCP-Array überschrieben werden, da nach Berechnung von $PLCP[i]$ der Wert

Algorithmus 4
Das Grundschema der Φ-Algorithmen zur Berechnung des LCP-Arrays eines Strings S. SA bezeichnet dabei das Suffix-Array von S.

```
/* Berechnung des Φ-Arrays */
for i ← 1 to n do
  Φ[SA[i]] ← SA[i − 1]
end for

/* Berechnung des PLCP-Arrays */
lcp ← 0
for i ← 1 to n do
  k ← Φ[i]
  while S[i + lcp] = S[k + lcp] do
    lcp ← lcp + 1
  end while
  PLCP[i] ← lcp
  if lcp > 0 then
    lcp ← lcp − 1
  end if
end for

/* Berechnung des LCP-Arrays */
for i ← 1 to n do
  LCP[i] ← PLCP[SA[i]]
end for
```

$\Phi[i]$ nicht mehr benötigt wird. Ebenso kann das Suffix-Array mit dem LCP-Array überschrieben werden. Insgesamt werden also nur zwei Arrays sowie der Text im Hauptspeicher gehalten. Der Speicherbedarf beträgt folglich $9n$ Bytes.

Um den Speicherverbrauch zu senken, wird beim Φ/PLCP-Array nur jeder q-te Wert gespeichert. Damit benötigt ein Φ-Algorithmus nur noch $5n + \frac{4n}{q}$ Bytes, allerdings müssen nun im letzten Schritt die fehlenden Werte im PLCP-Array berechnet werden. Dies kann amortisiert in einer Zeitkomplexität von $\mathcal{O}(q)$ pro Eintrag erfolgen, wie die Autoren von [KMP09] zeigen.

Mit Φq wird nun also ein Φ-Algorithmus bezeichnet, welcher initial nur jeden q-ten Wert berechnet. Bei Algorithmus 4 handelt es sich demnach um den Φ1-Algorithmus, der – wie dargestellt – das Φ/PLCP-Array mit der Länge n erzeugt und jeden Wert darin berechnet. Bei Φ4 wird dagegen nur jeder vierte Wert bestimmt. Es genügt also, das Φ/PLCP-Array mit $\frac{n}{4}$ Einträgen anzulegen, wodurch der Speicherverbrauch auf $6n$ Bytes sinkt. Zugleich ist aber zu erwarten, dass Φ4 langsamer als Φ1 ist, da

$\Phi 4$ im letzten Schritt $3n$ Buchstabenvergleiche benötigt, um die $\frac{3}{4}n$ fehlenden Werte zu berechnen.

Wird Algorithmus 4 näher betrachtet, so stellt man fest, dass nur beim Φ/PLCP-Array ein *random access* notwendig ist. Auf das Suffix/LCP-Array wird dagegen nur sequenziell zugegriffen. Dies sorgt nicht nur für einen Geschwindigkeitsvorteil, sondern erlaubt es auch, das Suffix/LCP-Array auszulagern.

Wird das Suffix/LCP-Array nicht im Hauptspeicher gehalten, sondern von der Festplatte gelesen, so bezeichnen wir den Algorithmus als Φq-*Semi*. Der Hauptspeicherbedarf sinkt dadurch auf $n + \frac{4n}{q}$ Bytes; somit benötigt beispielsweise der $\Phi 4$-*Semi*-Algorithmus nur noch $2n$ Bytes. Da das Suffix/LCP-Array stets sequenziell gelesen wird, verschlechtert sich die Laufzeit dadurch nicht wesentlich.

In [KMP09] wird eine weitere Berechnungsmöglichkeit des PCLP-Arrays aufgezeigt, die auf irreduziblen LCP-Werten basiert. Allerdings ist diese langsamer als die hier vorgestellte Methode und wird daher in dieser Arbeit nicht erläutert. Kärkkäinen et al. weisen darauf hin, dass in manchen Fällen – etwa um den größten oder durchschnittlichen LCP-Wert zu bestimmen – das PLCP-Array genügt. In diesem Fall kann der letzte Schritt entfallen. Die Autoren geben für diesen Fall an, dass der $\Phi 1$-Algorithmus etwa $2,2$-mal schneller als der *KLAAP*-Algorithmus ist. Aber auch mit dem letzten Schritt sei $\Phi 1$ immer noch ungefähr $1,5$mal schneller als *KLAAP*, was eigene Messungen in Abschnitt 5.3 bestätigen konnten.

3.3 *go*-Φ-Algorithmus

In [GO10] wird der *go*-Φ-Algorithmus vorgestellt. Hierbei handelt es sich um einen schnellen LCP-Konstruktionsalgorithmus, welcher für nahezu alle Eingaben einen Hauptspeicherbedarf von nur $2n$ Bytes besitzt. *go*-Φ benötigt – anders als *KLAAP* und Φ – zusätzlich zum Suffix-Array die BWT des Strings. Die Berechnung der LCP-Werte geschieht darüber hinaus mit zwei verschiedenen Verfahren.

Das erste Verfahren berechnet alle LCP-Werte, welche kleiner als 255 sind. Hierfür werden die Lemmata 3.4 und 3.5 sowie die Definition 3.3 benötigt.

Definition 3.3 (*prev*-Funktion) *Für einen String S der Länge n und $1 \leq i \leq n$ ist die Funktion prev wie folgt definiert:*

$$prev(i) = \begin{cases} 0 & \text{falls } \forall j < i : S[j] \neq S[i] \\ \max\{j < i \mid S[j] = S[i]\} & \text{sonst} \end{cases}$$

Für einen String S und eine Position i ermittelt die *prev*-Funktion folglich die Stelle, an der das Zeichen $S[i]$ zuletzt vor i auftrat. Das folgende Lemma macht hiervon Gebrauch.

Lemma 3.4 *Sei S ein String der Länge n. Dann gilt für alle $i \in \{1, \ldots, n\}$:*

$$LCP[LF[i]] = \begin{cases} 0 & \text{falls } prev(i) = 0 \\ 1 + \min\{LCP[j] \mid prev(i) < j \leq i\} & \text{falls } prev(i) \neq 0 \end{cases}$$

Beweis: Falls $prev(i) \neq 0$, so beginnen die beiden Suffixe $S_{SA[LF[i]-1]}$ und $S_{SA[LF[i]]}$ mit dem gleichen Buchstaben und es gilt $SA[LF[i] - 1] + 1 = prev(i)$. $LCP[LF[i]]$ kann dann wie folgt umgeformt werden:

$$\begin{aligned} LCP[LF[i]] &= lcp(S_{SA[LF[i]-1]}, S_{SA[LF[i]]}) && \text{Definition 2.13} \\ &= 1 + lcp(S_{SA[LF[i]-1]+1}, S_{SA[LF[i]]+1}) && \text{Definition 2.12} \\ &= 1 + lcp(S_{prev(i)}, S_{SA[LF[i]]+1}) \\ &= 1 + lcp(S_{prev(i)}, S_{SA[SA^{-1}[SA[i]-1]]+1}) && \text{Definition 2.19} \\ &= 1 + lcp(S_{prev(i)}, S_{SA[i]-1+1}) \\ &= 1 + \min\{LCP[j] \mid prev(i) < j \leq i\} && \text{Lemma 2.15} \end{aligned}$$

Gilt andernfalls $prev(i) = 0$, so ist $S_{SA[LF[i]]}$ das lexikografisch kleinste Suffix, welches mit $BWT[i]$ beginnt. Folglich beginnt das Suffix $S_{SA[LF[i]-1]}$, welches im Suffix-Array vor $S_{SA[LF[i]]}$ steht, mit einem anderen Buchstaben. In diesem Fall ist $LCP[LF[i]] = 0$. □

Sind alle LCP-Werte zwischen $prev(i)$ und i bekannt, so kann der LCP-Wert an der Stelle $LF[i]$ mittels Lemma 3.4 bestimmt werden.

Ist hingegen $LCP[LF[j]]$ bereits berechnet, so kann der LCP-Wert an Stelle j nach unten abgeschätzt werden. Dies zeigt das anschließende Lemma.

Lemma 3.5 *Für alle $i \in \{1, \ldots, n\}$ gilt: $LCP[j] \geq LCP[LF[j]] - 1$.*

Beweis: Lemma 3.5 folgt aus Lemma 3.1 und aus der Definition des LF-Arrays.

$$\begin{aligned} LCP[SA^{-1}[i+1]] &\geq LCP[SA^{-1}[i]] - 1 && \text{Lemma 3.1} \\ &= LCP[SA^{-1}[i+1-1]] - 1 \\ &= LCP[SA^{-1}[SA[SA^{-1}[i+1]] - 1]] - 1 \\ &= LCP[LF[SA^{-1}[i+1]]] - 1 && \text{Definition 2.19} \end{aligned}$$

Setzt man nun $j = SA^{-1}[i+1]$, so ergibt sich Lemma 3.5 für alle $j \in \{1, \ldots, n\} \setminus \{SA^{-1}[1]\}$. Lemma 3.1 schließt den Fall $SA^{-1}[1]$ aus. Daher bleibt zu zeigen, dass die Aussage auch für $j = SA^{-1}[1]$ gilt. Da $LF[SA^{-1}[1]] = SA^{-1}[n] = 1$ gilt, ist in diesem Fall die Aussage des Lemmas $LCP[j] \geq LCP[1] - 1 = -1 - 1 = -2$ und daher trivialerweise erfüllt. □

Algorithmus 5 zeigt nun, wie in der ersten Phase des *go-Phi*-Algorithmus alle LCP-Werte kleiner als 255 berechnet und die anderen LCP-Werte auf 255 gesetzt werden. Das LCP-Array wird hierzu sequentiell durchlaufen.

Dabei wird zunächst überprüft, ob der LCP-Wert an der i-ten Position bereits berechnet ist. Muss $LCP[i]$ noch berechnet werden, so wird der LCP-Wert an dieser Stelle ermittelt, indem die entsprechenden Suffixe buchstabenweise verglichen werden. Zuvor wird jedoch mittels $LF[i] < i$ überprüft, ob der LCP-Wert an der Stelle $LF[i]$ schon bestimmt wurde. In diesem Fall müssen aufgrund von Lemma 3.5 die ersten $LCP[LF[i]] - 1$ Buchstaben nicht verglichen werden. Ist $BWT[i] = BWT[i-1]$ so folgt darüber hinaus aus Lemma 2.23, dass der LCP-Wert reduzibel und damit genau $LCP[LF[i]] - 1$ ist.

Nach der Berechnung von $LCP[i]$ sind alle LCP-Werte von 1 bis i bekannt. Daher kann $LCP[LF[i]]$ über Lemma 3.5 bestimmt werden. Folglich wird im letzten Schritt der For-Schleife mit $LF[i] > i$ überprüft, ob $LCP[LF[i]]$ noch nicht berechnet wurde. Falls dies zutrifft, wird der LCP-Wert ermittelt und an die Stelle $LF[i]$ geschrieben.

In [GO10] wird gezeigt, dass das Minimum der LCP-Werte zwischen $prev(i)$ und i amortisiert mit einer Zeitkomplexität von $\mathcal{O}(\sigma)$ bestimmt werden kann. Da die inneren While-Schleifen höchstens 256 mal durchlaufen werden, ergibt sich – wenn σ als konstant angenommen wird – insgesamt eine Laufzeitkomplexität von $\mathcal{O}(n)$.

Obwohl in der erste Phase des *go*-Φ-Algorithmus das Suffix-Array, das LCP-Array, die BWT und der String benötigt werden, liegt der Hauptspeicherbedarf nur bei

Algorithmus 5
Die erste Phase des *go*-Φ-Algorithmus. Es werden alle LCP-Werte kleiner als 255 berechnet und in das LCP-Array eingetragen. Alle anderen LCP-Werte werden auf 255 gesetzt.

```
LCP[LF[1]] ← 0
for i ← 2 to n do
  if LCP[i] = ⊥ then
    l ← 0
    if LF[i] < i then
      /* Lemma 3.5 */
      l ← max{LCP[LF[i]] − 1, 0}
      /* Lemma 2.23 */
      if BWT[i] ≠ BWT[i − 1] then
        while S[SA[i] + l] = S[SA[i − 1] + l] and l < 255 do
          l ← l + 1
        end while
      end if
    else
      while S[SA[i] + l] = S[SA[i − 1] + l] and l < 255 do
        l ← l + 1
      end while
    end if
    LCP[i] ← l
  end if
  if LF[i] > i then
    /* Lemma 3.4 */
    LCP[LF[i]] ← min{255, 1 + min{LCP[j] | prev(i) < j ≤ i}}
  end if
end for
```

$2n$ Bytes. Der Grund dafür ist, dass auf das Suffix-Array und die BWT nur sequenziell zugegriffen wird. Daher können beide Datenstrukturen von der Festplatte gelesen werden und müssen nicht im Hauptspeicher gehalten werden. Darüber hinaus werden im LCP-Array nur 256 verschiedene Werte – die Zahlen von 0 bis 255 – abgespeichert, es benötigt daher ebenso wie der String nur n Bytes.

Für die zweite Phase verwendet *go*-Φ die Grundidee des Φ-Algorithmus mit einigen Modifikationen. Zunächst wird das Φ-Array und das PLCP-Array nur für die noch nicht berechneten LCP-Werte erzeugt. Außerdem müssen die ersten 255 Buchstaben nicht verglichen werden, da alle verbliebenen LCP-Werte größer als 254 sein müssen. Zuletzt wird zusätzlich das Konzept der irreduziblen LCP-Werte hinzugenommen, um weitere Buchstabenvergleiche einsparen zu können.

Die zweite Phase hat, wie die erste Phase, eine lineare Laufzeit. Daher ist die Zeitkomplexität des gesamten *go*-Φ-Algorithmus linear. Darüber hinaus besitzt der Algorithmus eine hohe Lokalität, wodurch *go*-Φ in der Praxis sehr schnell ist.

Wie bei dem Φ*q*-*Semi*-Algorithmen wird in der zweiten Phase neben dem String mit n Bytes nur das PLCP-Array im Hauptspeicher gehalten. Da die erste Phase stets einen Speicherbedarf von $2n$ Bytes besitzt, die zweite Phase jedoch bis zu $5n$ Bytes benötigt, liegt der Worst-Case-Speicherbedarf des *go*-Φ-Algorithmus bei $5n$ Bytes. In den meisten Fällen werden jedoch bereits in der ersten Phase mehr als 75% der LCP-Werte berechnet. In diesem Fall hat das PLCP-Array weniger als $\frac{n}{4}$ Einträge und benötigt daher weniger als n Bytes. Im Normalfall liegt der Speicherbedarf von *go*-Φ somit bei $2n$ Bytes.

4

Berechnung des LCP-Arrays aus der Burrows-Wheeler-Transformierten

In diesem Kapitel wird ein Algorithmus vorgestellt, welcher allein aus der BWT das LCP-Array eines Strings S berechnet. Zunächst wird in Abschnitt 4.1 die Grundidee und die Korrektheit des Algorithmus aufgezeigt. Dabei werden die Funktionen *getIntervals*, *saveInterval*, *getNextInterval* und *saveLCPValue* eingeführt. Die Implementierungen dieser Funktionen werden in den Abschnitten 4.2, 4.3 und 4.4 diskutiert. Abschließend werden die Ergebnisse in 4.5 zusammengefasst.

4.1 Grundidee des Algorithmus

Der im Folgenden vorgestellte Algorithmus basiert auf dem – in Abschnitt 2.8 definierten – Begriff des ω-Intervalls. Das folgende Lemma stellt dabei den wichtigen Zusammenhang zwischen den ω-Intervallen und dem LCP-Array her.

Lemma 4.1 *Ist* $[i \dots j]$ *ein* ω*-Intervall, so gilt* $LCP[j+1] < |\omega|$.

Beweis: Falls $LCP[j+1] \geq |\omega|$ gilt, dann stimmen die Suffixe $S_{SA[j+1]}$ und $S_{SA[j]}$ mindestens in den ersten $|\omega|$ Buchstaben überein. Da j im Intervall $[i \dots j]$ liegt, muss das Suffix $S_{SA[j]}$ mit ω beginnen. Somit muss ω auch Präfix des Suffixes $S_{SA[j+1]}$ sein. Gemäß Definition 2.8 muss dann das Intervallende größer oder gleich $j+1$ sein, das heißt, es muss $j \geq j+1$ gelten. Widerspruch. □

In Tabelle 4.1 sind die BWT und das LCP-Array des Strings `annasanannas$` dargestellt. Zudem sind einige ω-Intervalle markiert.

i	$BWT[i]$	$LCP[i]$	$S_{SA[i]}$
1	s	-1	$
2	s	0	anannas$
3	n	2	annas$
4	$	5	annasanannas$
5	n	1	as$
6	n	2	asanannas$
7	a	0	nannas$
8	n	2	nas$
9	n	3	nasanannas$
10	a	1	nnas$
11	a	4	nnasanannas$
12	a	0	s$
13	a	1	sanannas$
14		-1	

Tabelle 4.1: Die BWT und das LCP-Array des Strings `annasanannas$`. Zusätzlich sind fünf ω-Intervalle eingezeichnet. Für jedes eingezeichnete ω-Intervall $[i \dots j]$ ist der LCP-Wert an der Stelle $j+1$ markiert.

Beispiel 4.2 *Das* **na***-Intervall* $[7 \dots 9]$ *mit einer Präfixlänge von zwei ist in Tabelle 4.1 grün eingezeichnet. Der LCP-Wert an der zehnten Position ist eins und somit kleiner als die Präfixlänge.*

Algorithmus 6 erzeugt nacheinander – in aufsteigender Präfixlänge – alle ω-Intervalle $[i \dots j]$ des Strings. Dabei wird jeweils überprüft, ob der LCP-Wert an Position $j+1$ noch nicht berechnet wurde. Ist dies der Fall, so wird $LCP[j+1]$ auf $|\omega|-1$ gesetzt.

Algorithmus 6
Die Grundidee zur Berechnung des LCP-Arrays mittels BWT.

```
LCP[i] ← ⊥  ∀i ∈ {1, ..., n}
saveInterval([1 ... n], ϵ)
repeat
  ([i' ... j'], ω') ← getNextInterval()
  list ← getIntervals([i' ... j'], ω')
  for all ([i ... j], ω) in list do
    saveLCPValue(j + 1, |ω| − 1)
    saveInterval([i ... j], ω)
  end for
until all lcp-values are calculated
```

In Algorithmus 6 werden die folgenden Funktionen verwendet; auf deren Implementierung jedoch erst in den folgenden Abschnitten eingegangen wird:

- *saveInterval*($[i \dots j]$, ω): Speichert das übergebene ω-Intervall $[i \dots j]$.
- *getNextInterval*(): Liefert ein ω-Intervall, welches unter allen gespeicherten Intervallen die kürzeste Präfixlänge hat.
- *saveLCPValue*(i, *lcp*): Speichert, falls $LCP[i]$ noch nicht berechnet wurde oder größer als *lcp* ist, den Wert *lcp* an Position i des LCP-Arrays.
- *getIntervals*($[i \dots j]$, ω): Gibt alle vorkommenden $c\omega'$-Intervalle zurück.

Um die Korrektheit von Algorithmus 6 zu zeigen, wird folgendes Lemma benötigt:

Lemma 4.3 *Sei S ein String mit zugehörigem Suffix-Array SA und $k \in \{1, \dots, n\}$ so gewählt, dass $LCP[k+1] = l > 0$ ist. Dann kann das längste gemeinsame Präfix der Suffixe $S_{SA[k]}$ und $S_{SA[k+1]}$ durch $c\omega$, mit $c \in \Sigma$ und $\omega \in \Sigma^{l-1}$ dargestellt werden. Mit $p = SA^{-1}[SA[k]+1]$ und $q = SA^{-1}[SA[k+1]+1]$ folgt:*

1. $LCP[i] \geq |\omega|$ *für alle* i *mit* $p+1 \leq i \leq q$
2. $\exists r' \in \{p+1, \dots, q\} : LCP[r'] = |\omega|$
3. $BWT[i] \neq c$ *für alle* i *mit* $p < i < q$

Tabelle 4.2 zeigt die von Lemma 4.3 beschriebene Situation.

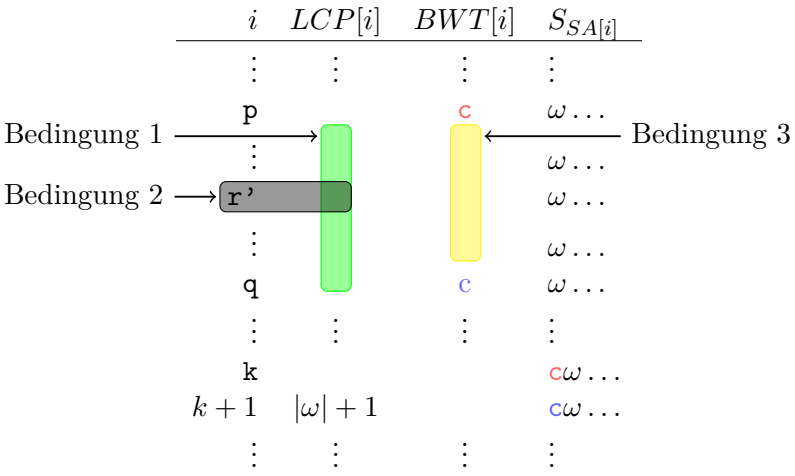


Tabelle 4.2: Gemäß Aussage 1 des Lemmas 4.3 müssen alle LCP-Werte im grünen Bereich größer oder gleich $|\omega|$ sein, wobei an mindestens einer Stelle – schwarz markiert – nach Bedingung 2 der LCP-Wert genau $|\omega|$ ist. Zusätzlich besagt Aussage 3, dass alle BWT-Werte im gelben Bereich ungleich c sein müssen.

Beweis: p und q sind derart gewählt, dass $S_{SA[k]} = cS_{SA[p]}$ und $S_{SA[k+1]} = cS_{SA[q]}$ gilt. Da $c\omega$ das längste gemeinsame Präfix der Suffixe $S_{SA[k]}$ und $S_{SA[k+1]}$ ist, muss ω das längste gemeinsame Präfix der Suffixe $S_{SA[p]}$ und $S_{SA[q]}$ sein. Es gilt somit $lcp(S_{SA[p]}, S_{SA[q]}) = |\omega|$. Nach Lemma 2.15 gilt die Gleichung:

$$lcp(S_{SA[p]}, S_{SA[q]}) = \min\{LCP[k] \mid p < k \leq q\}$$

Folglich ist kein LCP-Wert zwischen p und q kleiner als $|\omega|$ und die Aussage 1 ist bewiesen. Außerdem folgt daraus die Aussage 2 des Lemmas, da andernfalls $lcp(S_{SA[p]}, S_{SA[q]}) > |\omega|$ wäre.

Um die Aussage 3 zu zeigen, sei $i \in \{p+1, \ldots, q-1\}$ mit $BWT[i] = c$. Aufgrund der lexikografischen Sortierung des Suffix-Arrays gilt $S_{SA[p]} <_{lex} S_{SA[i]} <_{lex} S_{SA[q]}$. Die lexikografische Ordnung ändert sich durch das Voranstellen desselben Buchstabens nicht; daher gilt ebenso $S_{SA[k]} = cS_{SA[p]} <_{lex} cS_{SA[i]} <_{lex} cS_{SA[q]} = S_{SA[k+1]}$. Da $cS_{SA[i]} = BWT[i]S_{SA[i]} = S[SA[i]-1]S_{SA[i]} = S_{SA[i]-1}$ ist, muss $S_{SA[k']} = cS_{SA[i]}$ ein Suffix von S sein. Aufgrund der Definition des Suffix-Arrays muss somit $k < k' < k+1$ gelten. Widerspruch. □

Mittels Lemma 4.3 kann nun der folgende Satz bewiesen werden.

Satz 4.4 *Algorithmus 6 berechnet das LCP-Array korrekt.*

Der Beweis dieses Satzes basiert darauf, dass die LCP-Werte aufsteigend berechnet werden. Dies wird in dem Algorithmus durch die Funktion *getNextInterval* sichergestellt, welche stets ein ω-Intervall mit der kürzesten Präfixlänge liefert. Da sich bei den daraus erzeugten $c\omega$-Intervallen die Präfixlänge um eins erhöht, sind alle nachfolgenden LCP-Werte größer oder gleich dem aktuellen.

Beweis: Wir beweisen Satz 4.4 per Induktion über die LCP-Werte l.

Basisfall $l = 0$: Aus dem ϵ-Intervall werden mittels der Methode *getIntervals* alle in S vorkommenden c-Intervalle $[C[c]+1 \ldots C[c]+|S^{\{c\}}|+1]$ berechnet. Der Algorithmus setzt nun – abgesehen von dem größten Buchstaben – den jeweiligen LCP-Wert an der Stelle $C[c]+|S^{\{c\}}|+1$ auf null. Dies ist korrekt, da das Suffix $S_{SA[C[c]+|S^{\{c\}}|]}$ mit dem Buchstaben c, das Suffix $S_{SA[C[c]+|S^{\{c\}}|+1]}$ dagegen mit dem nächst größeren Buchstaben beginnt.

Gibt es σ' viele verschiedene Buchstaben in S, so tritt die Null genau $\sigma' - 1$ mal im LCP-Array auf. Dies liegt daran, dass die Suffixe lexikografisch sortiert sind

und daher der Fall, dass zwei aufeinanderfolgende Suffixe mit unterschiedlichem Buchstaben beginnen, nur $\sigma' - 1$ mal auftreten kann. Andererseits gibt es genau σ' viele c-Intervalle, denn ein c-Intervall existiert genau dann, wenn c in S auftritt. Da für jedes c-Intervall – bis auf das mit dem größten vorkommenden Buchstaben – eine Null im LCP-Array eingetragen wird, werden alle Nullwerte im LCP-Array gesetzt.

Fall $l > 0$: Sei $[i \dots j]$ ein ω-Intervall mit $|\omega| = l + 1$. Ist der LCP-Wert an der Stelle $j + 1$ noch nicht berechnet, so folgt $LCP[j + 1] \geq l$, da nach Induktionshypothese alle LCP-Werte kleiner l bereits berechnet sind. Zugleich gilt aber nach Lemma 4.1, dass $LCP[j + 1] < |\omega| = l + 1$ ist. Somit muss $LCP[j + 1] = l$ sein.

Es bleibt zu zeigen, dass alle l-Werte des LCP-Arrays berechnet werden. Sei hierzu $LCP[k + 1] = l$. Dann lässt sich das längste gemeinsame Präfix der Suffixe $S_{SA[k]}$ und $S_{SA[k+1]}$ als $c\omega$ mit $c \in \Sigma$ und $\omega \in \Sigma^{l-1}$ schreiben. Seien $SA[p] = SA[k] + 1$ und $SA[q] = SA[k + 1] + 1$. Dann ist ω das längste gemeinsame Präfix der Suffixe $S_{SA[p]}$ und $S_{SA[q]}$. Aus Lemma 4.3 folgt somit:

1. $LCP[i] \geq |\omega|$ für alle i mit $p + 1 \leq i \leq q$
2. $\exists r' \in \{p + 1, \dots, q\} : LCP[r'] = |\omega| = l - 1$
3. $BWT[i] \neq c$ für alle i mit $p < i < q$

Sei r die kleinste Zahl, welche Bedingung 2 erfüllt. Da $LCP[r] = l - 1$ also bereits berechnet ist, gibt es nach der Induktionshypothese ein ω'-Intervall $[t \dots r - 1]$ mit $|\omega'| = l$. Aufgrund von Bedingung 1 muss p im ω'-Intervall liegen, das heißt es muss $t \leq p \leq r - 1$ gelten. Da zusätzlich $BWT[p] = c$ ist, existiert ein $c\omega'$-Intervall $[t' \dots s']$, wobei aus Bedingung 3 $s' = k$ folgt. Das ω'-Intervall induziert folglich das $c\omega'$-Intervall $[t' \dots k]$ mit $|c\omega'| = l + 1$, wodurch sichergestellt wird, dass der Algorithmus $LCP[k + 1]$ auf l setzt. □

Sieht man sich den Beweis von Satz 4.4 genau an, stellt man fest, dass nur diejenigen ω-Intervalle gespeichert werden müssen, durch die ein LCP-Wert auf $|\omega| - 1$ gesetzt wurde. Algorithmus 7 nutzt diese Beobachtung aus. Hierzu wird die Funktion *saveLCPValue* um einen booleschen Rückgabewert ergänzt, welcher genau dann *true* ist, wenn der LCP-Wert an der übergebenen Stelle noch nicht berechnet wurde. Das Intervall wird nur in diesem Fall gespeichert.

Da es genau n LCP-Werte gibt, liefert *saveLCPValue* genau n-mal *true* zurück; somit werden nur n Intervalle gespeichert. Die Funktionen *getNextInterval*, *saveInterval* und *getIntervals* werden daher insgesamt n-mal ausgeführt. Aufgrund der Tatsache,

dass ein ω-Intervall höchstens σ viele Teilintervalle $c\omega$ besitzt, werden in der FOR-Schleife höchstens σ viele Elemente durchlaufen. Daher ist σn eine obere Schranke für die Anzahl der Funktionsaufrufe von *saveLCPValue*.

Algorithmus 7
Optimierte Version von Algorithmus 6: Die Verbesserung des Speicherverbrauchs ist in Rot dargestellt während die Laufzeitoptimierung Blau markiert ist.

```
LCP[i] ← ⊥ ∀i ∈ {1, ..., n}
saveInterval([1...n])
(counter, size, lcp) ← (1, 0, 0)
repeat
  [i...j] ← getNextInterval()
  counter ← counter − 1
  list ← getIntervals([i...j])
  for all [i...j] in list do
    if saveLCPValue(j + 1, lcp) then
      saveInterval([i...j])
      size ← size + 1
    end if
  end for
  if counter = 0 then
    (counter, size, lcp) ← (size, 0, lcp + 1)
  end if
until all lcp-values are calculated
```

Algorithmus 7 wurde gegenüber Algorithmus 6 nicht nur in der Zeitkomplexität, sondern auch hinsichtlich des Speicherbedarfs verbessert. Da es für die Methode *getIntervals* genügt, die Intervallgrenzen i und j zu kennen, und die Funktion *saveLCPValue* nur die Präfixlänge des Intervalls benötigt, ist es nicht nötig, ω explizit abzuspeichern. Vielmehr ist es ausreichend, die Präfixlänge des aktuellen Intervalls zu kennen. Da die Intervalle nach Präfixlängen abgearbeitet werden, erhöht sich diese um eins, nachdem alle ω-Intervalle der gleichen Präfixlänge abgearbeitet sind. Es genügt folglich die Verwendung einer Variablen *lcp*, welche inkrementiert wird, nachdem alle Intervalle der gleichen Präfixlänge abgearbeitet wurden. Hierzu wird in Algorithmus 7 mit den Variablen *counter* und *size* gezählt, wie viele Intervalle der aktuellen und der nächstgrößeren Präfixlänge gespeichert sind.

Beispiel 4.5 *Die Tabellen 4.3 und 4.4 veranschaulichen die Berechnung von Algorithmus 7 auf dem String* `annasanannas$`. *Initial wird das ϵ-Intervall* $[1 \dots 13]$ *gespeichert. Über* getIntervals *werden die Intervalle* `$`, `a`, `n` *und* `s` *in einer Liste zurückgeliefert. Diese wird anschließend durchlaufen:*

i	$BWT[i]$	$LCP[i]$	$S_{SA[i]}$
1	s	-1	$
2	s	0	ananannas$
3	n	2	annas$
4	$	5	annasanannas$
5	n	1	as$
6	n	2	asanannas$
7	a	0	nannas$
8	n	2	nas$
9	n	3	nasanannas$
10	a	1	nnas$
11	a	4	nnasanannas$
12	a	0	s$
13	a	1	sanannas$
14		-1	

Tabelle 4.3: Die BWT und das LCP-Array des Strings `annasanannas$`. Die ω-Intervalle, welche einen Eintrag im LCP-Array festlegen, sind farbig eingezeichnet.

Zuerst wird das `$`*-Intervall* $[1 \ldots 1]$ *bearbeitet. Da* $LCP[2]$ *noch nicht berechnet ist, wird der LCP-Wert an Position* 2 *auf Null gesetzt und das Intervall* $[1 \ldots 1]$ *gespeichert. Dasselbe geschieht anschließend bei dem* `a`*-Intervall und bei dem* `n`*-Intervall. Für das* `s`*-Intervall* $[12 \ldots 13]$ *liefert die Funktion* saveLCPValue *hingegen false zurück, daher wird das* `s`*-Intervall nicht gespeichert.*

Damit ist das ϵ*-Intervall abgearbeitet und die Funktion* getNextInterval *kann ein beliebiges der drei gespeicherten Intervalle zurückgeben, da diese alle die gleiche Präfixlänge haben. In Tabelle 4.4 liefert die Funktion zuerst das* `$`*-Intervall zurück. Die Funktion* getIntervals *findet hierfür nur das* `s$`*-Intervall* $[12 \ldots 12]$*, denn die Intervalle* `$$`*,* `a$` *und* `n$` *kommen in* S *nicht vor. Der LCP-Wert an Position* 13 *ist noch nicht berechnet, daher muss er größer null sein, denn die Nullwerte wurden bereits vollständig berechnet. Außerdem folgt aus Lemma 4.1, dass* $LCP[13] < 2$ *ist. Daher wird der LCP-Wert auf eins gesetzt und das Intervall* $[12 \ldots 12]$ *gespeichert.*

Das `$`*-Intervall ist damit abgearbeitet und die Funktion* getNextInterval *kann nun entweder das* `a`*-Intervall oder das* `n`*-Intervall zurückliefern. Das* `s$`*-Intervall kann hingegen erst zurückgegeben werden, wenn keine* ω*-Intervalle mit* $|\omega| = 1$ *mehr gespeichert sind.*

i	Aktuelles Intervall		Gespeicherte Intervalle	1	2	3	4	5	6	7	8	9	10	11	12	13	14
1	[1 . . . 13]	ϵ		-1													-1
2	[1 . . . 13]	ϵ	[1 . . . 1]	-1	0												-1
3	[1 . . . 13]	ϵ	[1 . . . 1], [2 . . . 6]	-1	0					0							-1
4	[1 . . . 13]	ϵ	[1 . . . 1], [2 . . . 6], [7 . . . 11]	-1	0					0					0		-1
5	[1 . . . 1]	`$`	[2 . . . 6], [7 . . . 11], [12 . . . 12]	-1	0					0					0	1	-1
6	[2 . . . 6]	`a`	[7 . . . 11], [12 . . . 12], [7 . . . 9]	-1	0					0			1		0	1	-1
7	[7 . . . 11]	`n`	[12 . . . 12], [7 . . . 9], [2 . . . 4]	-1	0			1		0			1		0	1	-1
8	[12 . . . 12]	`s$`	[7 . . . 9], [2 . . . 4], [5 . . . 5]	-1	0			1	2	0			1		0	1	-1
9	[7 . . . 9]	`na`	[2 . . . 4], [5 . . . 5], [2 . . . 2]	-1	0	2		1	2	0			1		0	1	-1
10	[2 . . . 4]	`an`	[5 . . . 5], [2 . . . 2], [7 . . . 7]	-1	0	2		1	2	0	2		1		0	1	-1
11	[5 . . . 5]	`as$`	[2 . . . 2], [7 . . . 7], [8 . . . 8]	-1	0	2		1	2	0	2	3	1		0	1	-1
12	[2 . . . 2]	`ana`	[7 . . . 7], [8 . . . 8]	-1	0	2		1	2	0	2	3	1		0	1	-1
13	[7 . . . 7]	`nan`	[8 . . . 8]	-1	0	2		1	2	0	2	3	1		0	1	-1
14	[8 . . . 8]	`nas$`	[10 . . . 10]	-1	0	2		1	2	0	2	3	1	4	0	1	-1
15	[10 . . . 10]	`nnas$`	[3 . . . 3]	-1	0	2	5	1	2	0	2	3	1	4	0	1	-1

Tabelle 4.4: Berechnung des LCP-Arrays gemäß Algorithmus 7 auf dem String `annasanannas$`.

4.2 Berechnung der Intervalle

Die in 4.1 eingeführte Funktion *getIntervals* soll zu einem ω-Intervall sämtliche in S vorkommenden $c\omega$-Intervalle zurückliefern. In diesem Abschnitt wird hierzu zunächst eine Methode vorgestellt, welche auf dem Wavelet Tree der BWT basiert und dessen Prinzip bereits in [Sch10] und in [CNPT10] beschrieben ist. Anschließend wird eine alternative Methode aufgezeigt, welche auf der LF-Funktion beruht.

4.2.1 Intervallberechnung mittels Wavelet Tree

In einem String S existiert ein $c\omega$-Intervall genau dann, wenn vor einem Suffix, welches mit ω beginnt, der Buchstabe c steht. Alle Suffixe, welche mit ω beginnen, befinden sich nach Definition gerade im ω-Intervall. Welcher Buchstabe vor dem Suffix steht, kann der BWT entnommen werden. Die Anzahl l der Vorkommen eines Buchstabens c im ω-Intervall bestimmt also, ob das $c\omega$-Intervall $[i \dots j]$ existiert $(l > 0)$ und wie groß dieses ist $(j - i + 1 = l)$.

Beispiel 4.6 $[2 \dots 6]$ *ist das* `a`*-Intervall des Strings* `annasanannas$`*. In* `sn$nn`*, also* $BWT[2 \dots 6]$*, kommen die Buchstaben* `$` *und* `s` *einmal,* `n` *dreimal und* `a` *keinmal vor. Es gibt daher kein* `aa`*-Intervall, aber ein* `$a`*-Intervall und ein* `sa`*-Intervall der Intervallgröße eins sowie ein* `na`*-Intervall mit Intervallgröße drei.*

Um das $c\omega$-Intervall eindeutig bestimmen zu können, benötigt man zusätzlich zur Intervallgröße auch den Intervallanfang. Hier kann folgende Beobachtung ausgenutzt werden: Ist $S_i <_{lex} S_j$, dann gilt auch $cS_i <_{lex} cS_j$, denn durch Voranstellen desselben Buchstabens ändert sich die lexikografische Ordnung zweier Strings nicht. Ist also $c = BWT[k]$ und i die Anzahl der Vorkommen von c in $BWT[1 \dots k-1]$, so gibt es i Suffixe, die mit c beginnen und kleiner als das Suffix $cS_{SA[k]}$ sind. Da es zusätzlich $C[c]$ Suffixe gibt, welche mit einem kleineren Buchstaben als c beginnen, gibt es unter allen Suffixen von S genau $C[c] + i$ Suffixe, welche kleiner als $cS_{SA[k]}$ sind. Damit steht $cS_{SA[k]}$ an Position $C[c] + i + 1$.

Beispiel 4.7 *Der Buchstabe* `n` *steht an Position* 5 *der BWT des Strings* $S =$ `annasanannas$`*. Gesucht sei nun die Position des Suffixes* $\texttt{n}S_{SA[5]} =$ `nas$`*. Da* `n` *in* $BWT[1 \dots 4] =$ `ssn$` *einmal vorkommt und es in* S *genau sechs Vorkommen von kleineren Buchstaben als* `n` *gibt, ist die gesuchte Position* $6 + 1 + 1 = 8$.

Die Frage nach allen $c\omega$-Intervallen lässt sich aus dem ω-Intervall $[i \dots j]$ folglich leicht beantworten, wenn für jeden Buchstaben c, der in $BWT[i \dots j]$ vorkommt, die Häufigkeit in $BWT[i \dots j]$ und $BWT[1 \dots i-1]$ bekannt ist. Mit anderen Worten: Es genügt also zu zählen, wie oft c bis zur Position $i-1$ bzw. j in der BWT vorkommt. Denn sei a die Häufigkeit von c in $BWT[1 \dots i-1]$ und b die Häufigkeit von c in $BWT[1 \dots j]$, so ist $[C[c]+a+1 \dots C[c]+b]$ das gesuchte $c\omega$-Intervall.

Algorithmus 8 zeigt nun, wie diese Häufigkeiten mit Hilfe des Wavelet Trees der BWT von S bestimmt werden können.

Algorithmus 8
Sei WT der Wavelet Tree des Strings S und *node* ein Knoten von WT mit zugeordneter Buchstabenmenge M. Dann zählt die Funktion *getIntervals*(*node*, i, j, *list*) für alle $c \in M$, wie oft c in $S^M[1 \dots i]$ und in $S^M[1 \dots j]$ vorkommt. Falls sich die Anzahl der Vorkommen unterscheiden, so werden diese zu der Liste *list* hinzugefügt.

```
if |M| = 1 then
  c ←getElement(M)
  add(list, [C[c] + i + 1 ... C[c] + j])
else
  (i', j') ← (rank_0(node, i), rank_0(node, j))
  if i' ≠ j' then
    getIntervals(leftChild(node), i', j', list)
  end if
  (i', j') ← (rank_1(node, i), rank_1(node, j))
  if i' ≠ j' then
    getIntervals(rightChild(node), i', j', list)
  end if
end if
```

Besteht ein String aus nur einem Buchstaben c, so kommt c bis zur Position i bzw. j trivialerweise gerade i- bzw. j-mal vor. Da ein Blatt im Wavelet Tree mit zugehöriger Buchstabenmenge $M = \{c\}$ alle Vorkommen von c im zugehörigen String repräsentiert, kann *getIntervals*(*node*, i, j, *list*) leicht berechnet werden, wenn es sich bei *node* um ein Blatt handelt. Enthält die Buchstabenmenge, welche *node* zugeordnet ist, mehr als einen Buchstaben, so handelt es sich um einen inneren Knoten. Hier wird zunächst i' als die Anzahl an Nullen bis zur Position i und j' als die Anzahl der Nullen bis zur Position j gezählt. Ist $i' = j'$, so kommt zwischen i und j keine Null vor und folglich gibt es auch kein $c\omega$-Intervall für $c \in M_0$. Andernfalls gibt es $j' - i'$ Vorkommen, welche rekursiv berechnet werden können. Dasselbe wird anschließend für die Anzahl an Einsen wiederholt.

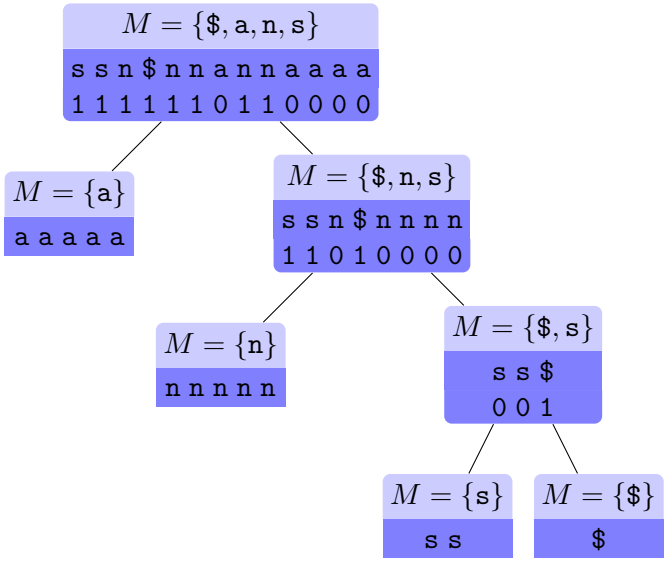


Abbildung 4.1: Wavelet Tree nach dem Huffman Schema für die BWT des Strings `annasanannas$`. Der Text in den inneren Knoten dient nur dem Verständnis und ist nicht Bestandteil des Wavelet Trees.

Beispiel 4.8 erläutert die Funktionsweise von Algorithmus 8 anhand des Strings `annasanannas$`. Ein Wavelet Tree der BWT von S ist in Abbildung 4.1 dargestellt.

Beispiel 4.8 *Mit* getIntervals*(root,* 6*,* 10*, list) werden, unter Verwendung des* **n***-Intervalls* $[7 \dots 11]$*, die in S vorkommenden c***n***-Intervalle an die Liste list angefügt.*

Da der Wurzelknoten kein Blatt ist und bis zu Position 7 *keine Null, bis zu Position* 11 *jedoch drei Nullen auftreten, findet ein rekursiver Abstieg in den* **a***-Knoten statt. Auch die Anzahl der Einsen bis zu den Positionen* 6 *und* 11 *unterscheidet sich mit sechs bzw. acht. Daher erfolgt zusätzlich ein rekursiver Abstieg in das rechte Kind.*

Der **a***-Knoten ist ein Blatt, folglich wird beim Aufruf von* getIntervals*(***a***-Knoten,* 0*,* 3*, list), mit* $[C[\mathbf{a}]+0+1 \dots C[\mathbf{a}]+3] = [2 \dots 4]$ *das* **an***-Intervall an die Liste angefügt.*

Im rechten Kind wird anschließend die Anzahl der Nullen bzw. Einsen bis zu den Positionen 6 *bzw.* 8 *ermittelt. Die Anzahl der Nullen ist mit drei bzw. fünf unterschiedlich, die Anzahl der Einsen mit jeweils drei dagegen gleich. Somit findet mit* getIntervals*(***n***-Knoten,* 3*,* 5*, list) nur ein rekursiver Abstieg in das linke Kind statt.*

Der **n***-Knoten ist wiederum ein Blatt, daher wird dort mit* $[C[\mathbf{n}]+3+1 \dots C[\mathbf{n}]+5] = [10 \dots 11]$ *das* **nn***-Intervall an die Liste angefügt.*

4.2.2 Intervallberechnung mittels *LF*-Funktion

Die Aufgabe, aus einem gegebenen ω-Intervall alle $c\omega$-Intervalle zu berechnen, kann auch unter Verwendung der LF-Funktion gelöst werden. Diese gibt für jeden Index k an, an welcher Position das Suffix steht, welches man durch Voranstellen von $BWT[k]$ an $S_{SA[k]}$ erhält.

Algorithmus 9 ermittelt daher die $c\omega$-Intervalle aus dem ω-Intervall $[i \dots j]$, indem die BWT von i bis j durchlaufen und für jeden Buchstaben c der LF-Wert beim ersten und beim letzten Auftreten von c gespeichert wird. Da alle Suffixe, welche mit ω beginnen, gerade im ω-Intervall liegen, muss nicht die gesamte BWT durchlaufen werden. Des Weiteren wird ausgenutzt, dass die lexikografische Ordnung zweier Strings bei Voranstellen des gleichen Buchstabens erhalten bleibt. Somit gibt der LF-Wert beim ersten Auftreten des Buchstabens c die Position des kleinsten Suffixes, welches mit $c\omega$ beginnt, und damit den Intervallanfang des $c\omega$-Intervalls an. Analog ist der LF-Wert an der Stelle des letzten Vorkommens von c in BWT$[i \dots j]$ gerade das Intervallende des $c\omega$-Intervalls.

Algorithmus 9
Die Funktion *getIntervals*(i, j, *list*) ermittelt für alle $c \in \Sigma$, wie oft c in $S[1 \dots i]$ und in $S[1 \dots j]$ vorkommt. Falls sich die Anzahl der Vorkommen unterscheiden, so werden diese zu der Liste *list* hinzugefügt.

```
M ← ∅
for k ← i to j do
  c ← BWT[k]
  if c ∉ M then
    M ← M ∪ {c}
    First[c] ← LF[k]
  end if
  Last[c] ← LF[k]
end for
for all c ∈ M do
  add(list, [First[c] ... Last[c]])
end for
```

Beispiel 4.9 erläutert die Funktionsweise von Algorithmus 9 anhand des Strings `annasanannas$`. Die BWT und das LF-Array von S sind in Tabelle 2.3 auf Seite 14 dargestellt.

Beispiel 4.9 *Mit* getIntervals*(2, 6, list) werden – unter Verwendung des* `a`*-Intervalls* $[2 \dots 6]$ *– die in S vorkommenden c*`a`*-Intervalle an die Liste list angefügt. Wie*

	First				*Last*				
k	$	a	n	s	$	a	n	s	M
	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	$\emptyset$
2	⊥	⊥	⊥	13	⊥	⊥	⊥	13	{s}
3	⊥	⊥	7	13	⊥	⊥	7	13	{s, n}
4	1	⊥	7	13	1	⊥	7	13	{s, n, $}
5	1	⊥	7	13	1	⊥	8	13	{s, n, $}
6	1	⊥	7	13	1	⊥	9	13	{s, n, $}

Tabelle 4.5: Berechnung von Algorithmus 9 auf dem a-Intervall $[2 \dots 6]$ des Strings annasanannas$.

in der ersten Zeile von Tabelle 4.5 dargestellt, sind zu Beginn die beiden Arrays First und Last undefiniert und die Menge M leer. Nun wird die BWT von Position 2 *bis Position* 6 *– also* **sn$nn** *– durchlaufen.*

Da der Buchstabe $\mathtt{s} = BWT[2]$ *noch nicht in der Menge M ist, wird er hinzugefügt und* $First[\mathtt{s}] = LF[2] = 13$ *gesetzt. Zudem wird* $Last[\mathtt{s}]$ *ebenfalls auf* 13 *gesetzt. Anschließend wird der Buchstabe* $\mathtt{n} = BWT[3]$ *in die Menge M aufgenommen und sowohl* $First[\mathtt{n}]$ *als auch* $Last[\mathtt{n}]$ *auf* $LF[3] = 7$ *gesetzt. Analog wird* $\$ = BWT[4]$ *zu M hinzugefügt und anschließend werden* $First[\$]$ *und* $Last[\$]$ *auf* $LF[4] = 1$ *gesetzt. In den letzten beiden Schritten steht an Position* 5 *und* 6 *der BWT jeweils der Buchstabe* **n**. *Da* **n** *bereits in der Menge M ist, wird nur* $Last[\mathtt{n}]$ *zuerst auf* $LF[5] = 8$ *und schließlich auf* $LF[6] = 9$ *geändert. Tabelle 4.5 zeigt in der letzten Zeile den endgültigen Inhalt von First, Last und M. Da M die Buchstaben* **s**, **n**, **$** *enthält, gibt es ein* **sa**, **na** *und* **$a***-Intervall. Die Intervallgrenzen stehen dabei an den entsprechenden Positionen von First und Last. Demnach ist beispielsweise* $[First[\mathtt{n}] \dots Last[\mathtt{n}]] = [7 \dots 9]$ *das* **na***-Intervall.*

4.2.3 Vergleich der Intervallberechnungsalgorithmen

Beim Berechnen der Intervalle nach Algorithmus 8 wird im Worst Case der gesamte Wavelet Tree durchlaufen. Da es sich beim Wavelet Tree um einen Binärbaum mit σ vielen Blättern handelt, besitzt der Wavelet Tree $\sigma - 1$ viele innere Knoten. In jedem der $2\sigma - 1$ Knoten werden zwei Rank-Anfragen gestellt, welche in $\mathcal{O}(1)$ beantwortet werden können. Damit beträgt die Zeitkomplexität $\mathcal{O}(\sigma)$.

Algorithmus 9 durchläuft zum Ermitteln der $c\omega$-Intervalle die BWT von i bis j, wobei $[i \dots j]$ gerade das ω-Intervall ist. Da sämtliche Operationen während des BWT-

Durchlaufs in konstanter Zeit durchgeführt werden können, besitzt die Berechnung der Intervalle mittels LF-Funktion eine Zeitkomplexität von $\mathcal{O}(j - i)$.

Somit ist zu vermuten, dass bei sehr kleinen Intervallen die Berechnung mittels LF-Funktion schneller ist, als die Berechnung mittels Wavelet Tree. Dies liegt nicht allein an der Alphabetgröße σ sondern zusätzlich an den Rank-Abfragen. Denn diese benötigen – obwohl in der $\mathcal{O}$-Notation nur ein konstanter Faktor – in der Praxis deutlich mehr Zeit als beispielsweise ein Array-Zugriff. Andererseits ist Algorithmus 9 bei großen Intervallen langsamer als Algorithmus 8.

Neben der Laufzeit spielt der Speicherbedarf eine entscheidende Rolle. Die Berechnung mittels Wavelet Tree offenbart hier zunächst deutliche Vorteile gegenüber der Berechnung mittels LF-Funktion. Denn während Algorithmus 8 nur die BWT in Form des Wavelet Trees benötigt, setzt Algorithmus 9 zusätzlich das LF-Array voraus. Da das LF-Array $n \log_2 n$ Bits Speicher benötigt, während für die BWT nur $n \log_2 \sigma$ benötigt werden, ist das LF-Array in der Praxis ungefähr vier mal größer als die BWT. Darüber hinaus kann durch Einsatz eines Huffman-Schemas die BWT im Wavelet Tree komprimiert abgespeichert werden.

Andererseits erlaubt Algorithmus 9, dass die benötigten Datenstrukturen, die BWT und das LF-Array, während der Berechnung sequentiell von der Festplatte gelesen werden. Wird die Funktion *getIntervals* oft aufgerufen und liegen die Intervalle dabei in aufsteigender Reihenfolge vor, so kann der Festplattenzugriff gepuffert erfolgen und so den Geschwindigkeitsnachteil der Festplatte gegenüber dem Hauptspeicher merklich abmildern. Somit benötigt in diesem Szenario die Berechnung mittels LF-Funktion weniger Hauptspeicher als mittels Wavelet Tree.

4.3 Speicherung der Intervalle

In diesem Abschnitt werden zwei Möglichkeiten zur Implementierung der von Algorithmus 7 benötigten Funktionen *getNextInterval* und *saveInterval* vorgestellt. Die Funktion *saveInterval* soll den Intervallanfang i und das Intervallende j eines ω-Intervalls $[i \dots j]$ speichern. Parallel dazu soll *getNextInterval* unter allen gespeicherten ω-Intervallen ein Intervall mit kürzester Präfixlänge zurückliefern.

Da aus einem ω-Intervall die $c\omega$-Intervalle berechnet werden, steigt die Präfixlänge der Intervalle jeweils um den Wert 1. Die Datenstruktur *Queue* bietet somit eine

einfache Möglichkeit zur Realisierung der Funktionen *saveInterval* und *getNextInterval*. Algorithmus 10 und 11 zeigen eine mögliche Implementierung, welche pro Aufruf nur konstante Zeit benötigt.

Algorithmus 10
Die Funktion *saveInterval*($[i \dots j]$) speichert das Intervall $[i \dots j]$ mittels *Queue*.

```
enqueue(i)
enqueue(j)
```

Algorithmus 11
Die Funktion *getNextInterval* liefert ein Intervall $[i \dots j]$ zurück, welches unter allen gespeicherten Intervallen die kürzeste Präfixlänge hat.

```
i ← dequeue()
j ← dequeue()
return [i ... j]
```

Die Realisierung mittels *Queue* ist zwar einfach und schnell, allerdings ist der Speicherverbrauch abhängig von der Anzahl gespeicherter Intervalle. Da der Intervallanfang und das Intervallende im Zahlenbereich $[1 \dots n]$ liegt, werden pro Intervall $2 \log_2 n$ Bits benötigt. Ist beispielsweise $n < 2^{32}$ so werden zum Speichern von $\frac{n}{8}$ Intervallen $\frac{n}{8} \cdot 2 \log_2 2^{32} = 8n$ Bits benötigt.

Eine alternative Speichermöglichkeit stellt das Speichern der Intervalle in einem Bitarray dar. Hierbei markiert eine 1 den Anfang bzw. das Ende eines Intervalls. Damit eine Anfangsmarkierung nicht mit einer Endmarkierung zusammenfällt, werden die Markierungen so verschachtelt, dass die Anfangsmarkierungen an ungeraden Positionen und die Endmarkierungen an geraden Positionen liegen. Ein ω-Interval $[i \dots j]$ wird also, wie Algorithmus 12 zeigt, dadurch gespeichert, dass die Position $2i - 1$ und $2j$ des Bitarrays auf 1 gesetzt werden. Abbildung 4.2 veranschaulicht dies.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	0	0	0	0	1	1	0	0

Abbildung 4.2: Im Bitarray ist das `s$`-Intervall $[12 \dots 12]$, das `na`-Intervall $[7 \dots 9]$ und das `an`-Intervall $[2 \dots 4]$ des Strings `annasanannas$` gespeichert.

Algorithmus 13 zeigt, wie ein auf diese Weise gespeicherters Intervall rekonstruiert werden kann. Dazu wird das Bitarray von vorne nach hinten durchlaufen. Trifft man an Stelle k_1 auf die erste 1, so beginnt das Intervall bei $\frac{k_1+1}{2}$. Durchsucht man das Array weiter und trifft an der Stelle k_2 auf die zweite 1, so steht mit $\frac{k_2}{2}$ auch

das Intervallende fest. Dies setzt voraus, dass die gefundenen Einsen bei k_1 und k_2 den Intervallanfang bzw. das Intervallende des gleichen Intervalls markieren. Anders ausgedrückt dürfen sich die Intervalle im Bitarray nicht überlappen. Da dies nur für ω-Intervalle mit gleicher Präfixlänge erfüllt ist, können in einem Bitarray nur ω-Intervalle mit gleicher Präfixlänge gespeichert werden.

Algorithmus 12
Die Funktion *saveInterval*(B, $[i \dots j]$) speichert das Intervall $[i \dots j]$ im Bitarray B.

$B[2i-1] \leftarrow 1$
$B[2j] \leftarrow 1$

Algorithmus 13
Die Funktion *getNextInterval*(B, k) liefert ein Intervall $[i \dots j]$ zurück, welches in B gespeichert ist und für das $k \leq i$ gilt.

```
i ← 2k − 1
while B[i] = 0 and i ≤ 2n do
  i ← i + 2
end while
if i > 2n then
  return ⊥
end if
j ← i + 1
i ← (i + 1)/2
while B[j] = 0 do
  j ← j + 2
end while
j ← (j − 1)/2
return [i . . . j]
```

Um die Intervalle mittels Bitarray speichern zu können, werden also zwei Bitarrays B_1 und B_2 benötigt. Initial sind in B_1 alle Intervalle der Präfixlänge l gespeichert. Nun wird B_1 durchlaufen, um die ω-Intervalle zu rekonstruieren. Die dabei erzeugten $c\omega$-Intervalle haben die Präfixlänge $l+1$ und werden in B_2 gespeichert.

Sind alle Intervalle aus B_1 abgearbeitet, werden die Markierungen in B_1 gelöscht und das Bitarray B_2 durchlaufen. Die dann erzeugten Intervalle haben die Präfixlänge $l+2$ und können in B_1 gespeichert werden. Auf diese Weise enthalten beide Bitarrays nur Intervalle mit gleicher Präfixlänge und die Funktion *getNextInterval* liefert stets ein Intervall kleinster Präfixlänge zurück.

Bemerkenswert ist, dass bei dieser Variante die Intervalle gleicher Präfixlänge in aufsteigender Reihenfolge zurückgeliefert werden. Beim Bitarray aus Abbildung 4.2 wird

daher zuerst das `an`-Intervall $[2 \dots 4]$, dann das `na`-Intervall $[7 \dots 9]$ und schließlich das `s$`-Intervall $[12 \dots 12]$ zurückgeliefert, unabhängig davon in welcher Reihenfolge diese Intervalle gespeichert wurden.

Da B_1 und B_2 eine Länge von $2n$ haben, beträgt der Speicherverbrauch insgesamt $4n$ Bits, unabhängig davon wie viele Intervalle gespeichert sind. Wie bei der Speichermethode mittels *Queue* hat die Methode *saveInterval* konstanten Aufwand. Allerdings erhöht sich der Aufwand für *getNextInterval*: Sind im Bitarray i Intervalle gespeichert, so können diese mit einem Durchlauf des Bitarrays, also mit $2n$ Operationen, gefunden werden. Der Aufwand für einen *getNextInterval*-Aufruf beträgt somit amortisiert $\frac{2n}{i}$.

Betrachtet man ausschließlich die Laufzeit, so ist die Speicherung mittels *Queue* vorzuziehen. Anders sieht es beim Speicherbedarf aus. Da die *Queue* pro gespeichertem Intervall $2 \log_2 n$ Bits benötigt, während das Speichern mittels Bitarray konstant $4n$ Bits benötigt, ist bei mehr als $\frac{2n}{\log_2 n}$ Intervallen das Bitarray speichereffizienter.

4.4 Speicherung der LCP-Werte

Da alle LCP-Werte eines Strings S kleiner als $n = |S|$ sind, kann jeder LCP-Wert mit $\lceil \log_2 n \rceil$ Bits gespeichert werden. Algorithmus 7 benötigt jedoch *random access* auf das LCP-Array, weshalb das gesamte LCP-Array der Länge n im Hauptspeicher liegen muss und somit ein Hauptspeicherbedarf von $n \log_2 n$ Bits besitzt. Im Folgenden wird ein Verfahren vorgestellt, welches den Hauptspeicherbedarf auf $\mathcal{O}(n)$ Bits senkt. Dabei wird ausgenutzt, dass die LCP-Werte von Algorithmus 7 in aufsteigender Reihenfolge berechnet werden.

Das LCP-Array mit einem Speicherbedarf von $n \log_2 n$ Bits wird nicht im Hauptspeicher, sondern auf der Festplatte erzeugt. Im Hauptspeicher wird dagegen ein Array A der Länge n mit nur k Bits pro Eintrag angelegt. Damit kann jeder Eintrag nur 2^k viele verschiedene Werte speichern, wobei davon ein Wert zum Kennzeichnen noch nicht berechneter LCP-Werte benötigt wird. Daher werden jeweils $2^k - 1$ viele verschiedene LCP-Werte in A gespeichert, bevor diese in das Array auf der Festplatte eingefügt und aus A gelöscht werden. Dies wird solange wiederholt, bis alle LCP-Werte berechnet sind.

Ist lcp_{max} der größte LCP-Wert, so muss A allerdings $\frac{lcp_{max}}{2^k - 1}$ mal auf die Festplatte geschrieben werden. Damit ergibt sich eine Worst-Case-Zeitkomplexität von $\mathcal{O}(n^2)$. Diese tritt beispielsweise bei einem String auf, der nur aus einem Buchstaben besteht

Abbildung 4.3: Darstellung des Hauptspeicherinhalts jeweils nach vollständiger Berechnung aller LCP-Werte derselben Größe. Das LCP-Array auf der Festplatte ist grau dargestellt, jeweils nachdem der Hauptspeicherinhalt eingefügt wurde.

und dessen LCP-Array folglich alle Werte von 0 bis $n-1$ enthält. Dies führt dazu, dass das Array $\mathcal{O}(n)$ mal ausgelagert werden muss, wobei jede Auslagerung eine Zeitkomplexität von $\mathcal{O}(n)$ erfodert.

Beispiel 4.10 *Abbildung 4.3 demonstriert das Speichern des LCP-Arrays für den String* `annasanannas$` *mit einem Hauptspeicherbedarf von* $2n = 26$ *Bits. Das sich im Hauptspeicher befindende Array hat daher* 2 *Bits pro Eintrag und kann damit* 4 *verschiedene Werte speichern. Zum Markieren nicht berechneter LCP-Werte wird das Symbol* $\perp$ *benutzt. Nun werden die ersten drei Werte* -1, 0 *und* 1 *als* 0, 1 *und* 2 *kodiert gespeichert. Anschließend werden die Werte auf die Festplatte verschoben. Nun können die nächsten drei LCP-Werte wiederum als* 0, 1 *und* 2 *kodiert gespeichert und anschließend in das LCP-Array auf der Festplatte eingefügt werden. Im letzten Schritt wird der LCP-Wert* 5 *als* 0 *kodiert eingetragen und in das LCP-Array auf der Festplatte gespeichert.*

Das gerade vorgestellte Prinzip kann verbessert werden, indem für bereits berechnete LCP-Werte kein Speicherplatz bereitgehalten wird. Gibt es also l viele Werte, welche noch nicht berechnet wurden, so genügt es, A mit dieser Länge anzulegen. Dann können für jeden Eintrag $b = \left\lfloor \frac{k \cdot n}{l} \right\rfloor$ Bits vorgesehen werden ohne insgesamt mehr als $k \cdot n$ Bits zu benötigen.

Allerdings kann der LCP-Wert an Stelle i nun nicht mehr an die Position $A[i]$ geschrieben werden. Vielmehr muss die korrekte Position j als die Anzahl nicht berechneter LCP-Werte bis zur Stelle i ermittelt werden. Hierzu kann ein Bitarray B der Länge n angelegt werden, bei dem $B[i] = 0$ ist, falls $LCP[i]$ noch nicht berechnet wurde. Wird B so vorverarbeitet, dass Rank-Anfragen in konstanter Zeit beantwortet werden können, dann kann j mittels $rank_0(B,\ i)$ in konstanter Zeit ermittelt werden.

Algorithmus 14 zeigt abschließend eine mögliche Implementierung der Funktion *saveLCPValue*. Im Wesentlichen wird mit $j \leftarrow \text{rank}_0(B,\ i)$ die Position im Array A ermittelt. Wurde der LCP-Wert bereits berechnet, ist also entweder $B[i] = 1$ oder $A[j] \neq \perp$, so wird *false* zurückgegeben. Andernfalls wird v in das A-Array geschrieben und *true* zurückgegeben. Hierbei gibt v an, wie viele verschiedene LCP-Werte seit dem letzten Auslagern des A-Arrays gespeichert wurden. Unterscheidet sich der aktuell zu speichernde LCP-Wert V vom zuletzt gespeicherten LCP-Wert V_{last}, so wird v erhöht. Kann v nicht mehr in A gespeichert werden ($v = v_{max}$), so wird das Bitarray B aktualisiert und A auf die Festplatte geschrieben. Anschließend wird

Algorithmus 14
Die Funktion *saveLCPValue*(V, i) speichert den LCP-Wert V an Position i. V_{last} gibt dabei den letzten gespeicherten LCP-Wert an, während die Variable v zählt, wie viel verschiedene LCP-Werte seit dem letzten Auslagern des A-Arrays gespeichert wurden. v_{max} ist der kleinste Wert, der nicht mehr in A gespeichert werden kann. Das Bitarray B ist an Stelle j genau dann 1, wenn der LCP-Wert an Stelle j bereits im endgültigen LCP-Array auf der Festplatte steht.

```
if V ≠ V_last then
  V_last ← V
  v ← v + 1
  if v = v_max then
    B[i] ← 1  ∀i : A[i] ≠ ⊥
    save A to disk
    l ← rank_0(B, n)
    b ← ⌊k · n/l⌋
    v ← 0
    v_max ← 2^b − 1
    create A as Array of length l and b bits per entry
  end if
end if
j ← rank_0(B, i)
if B[i] = 1 or A[j] = ⊥ then
  A[j] ← v
  return true
else
  return false
end if
```

die Anzahl nicht berechneter LCP-Werte gezählt und ermittelt, wie viele Bits pro Eintrag zur Verfügung stehen. Zuletzt wird A mit diesen Parametern neu angelegt und v sowie v_{max} aktualisiert.

Diese Variante hat einen Hauptspeicherbedarf von $k \cdot n$ Bits für das A-Array, n Bits für das B-Array und $0,25n$ Bits um die Rank-Anfrage auf dem B-Array in konstanter Zeit zu beantworten. Insgesamt werden also $(k + 1,25) \cdot n$ Bits mit $k \geq 1$ benötigt. Wie zuvor muss das A-Array umso öfter auf die Festplatte geschrieben werden, je kleiner k gewählt wird. An der Worst-Case-Zeitkomplexität von $\mathcal{O}(n^2)$ ändert sich daher nichts.

Allerdings kommt dieser Variante zu Gute, dass in der Praxis kleine LCP-Werte viel häufiger vorkommen als große. So gilt beispielsweise für alle in 4.4 dargestellten Testdateien, dass mindestens 70% der LCP-Werte kleiner als 63 sind. Wurde das A-Array zunächst mit 6 Bits pro Eintrag angelegt, so sind nach dem ersten Auslagern

aller LCP-Werte ≤ 63 nur noch 30% der Werte nicht berechnet. Das A-Array kann dann mit 20 Bits pro Eintrag neu angelegt werden und alle LCP-Werte zwischen 63 und $63 + 2^{20} - 1 = 1.048.638$ speichern.

4.5 Zusammenfassung

In diesem Abschnitt soll es abschließend darum gehen, welche Implementeriungsvariante von *getIntervals* und *saveInterval* bzw. *getNextInterval* bevorzugt werden sollte. Zudem wird die Laufzeit- und Speicherplatzkomplexität des in diesem Kapitel vorgestellten Algorithmus analysiert.

Wie bereits in 4.3 erläutert, ist die Implementierung von *getNextInterval* mittels *Queue* der Realisierung mittels Bitarray zu bevorzugen, wenn sich wenige Intervalle im Speicher befinden. In diesem Fall ist zur Berechnung von *getIntervals* der Wavelet Tree dem LF-Array vorzuziehen. Denn da das LF-Array nicht im Hauptspeicher sondern auf der Festplatte gespeichert werden soll, muss der Zugriff gepuffert erfolgen um den Geschwindigkeitsnachteil der Festplatte abzumildern. Dies wird sowohl durch eine geringe Anzahl an Intervallen mit gleicher Präfixlänge, als auch durch die nicht sequenzielle Rückgabe der Intervalle, wie sie bei der Speicherung in einer *Queue* geschieht, erschwert.

Müssen dagegen viele ω-Intervalle gleichzeitig gespeichert werden, so ist zur Berechnung von *getNextInterval* aus Speicherplatzgründen die Verwendungs des Bitarrays zu bevorzugen. Da dieses die Intervalle in aufsteigender Reihenfolge zurückgibt, bietet sich dabei das LF-Array zur Realisierung der Funktion *getIntervals* an.

Es liegt folglich nahe, die beiden Varianten zu kombinieren. Gibt es wenige Vorkommen eines LCP-Wertes, so werden die zugehörigen ω-Intervalle in einer *Queue* gespeichert und die Funktion *getIntervals* mittels Wavelet Tree realisiert. Bei häufigem Auftreten eines LCP-Wertes werden die ω-Intervalle hingegen in Bitarrays gespeichert und *getIntervals* über das LF-Array berechnet.

In der Praxis gibt es oft sehr wenig kleine und wenig große LCP-Werte. Dazwischen kommt es zu einer Häufung der LCP-Werte, wie Abbildung 4.4 zeigt. Dies begünstigt eine Kombination der beiden Varianten, da somit nur zwei Wechsel nötig sind. Ab welcher Intervallanzahl in der Praxis gewechselt werden sollte, wird in Abschnitt 5.2 empirisch ermittelt.

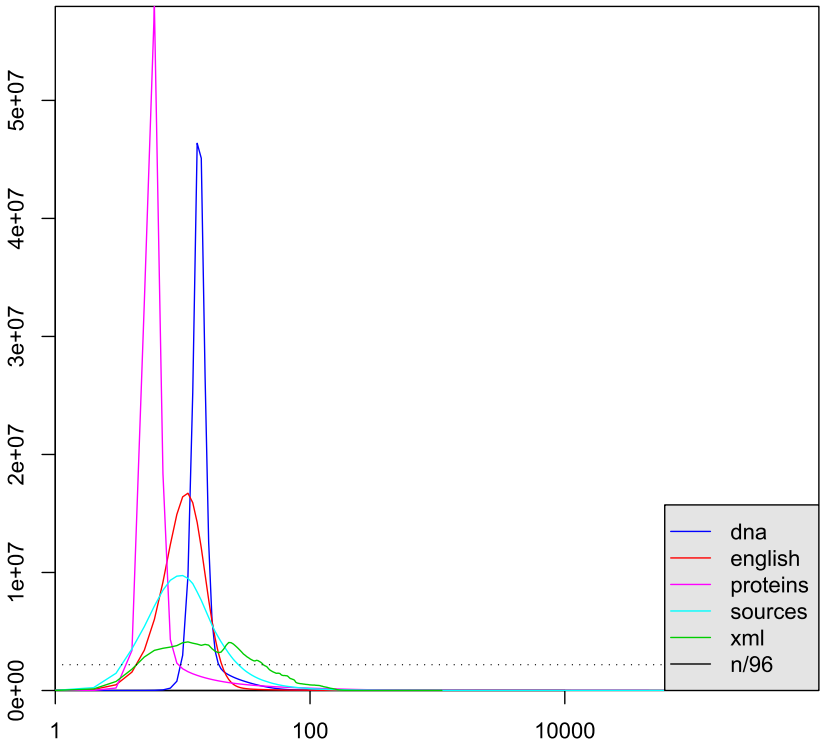


Abbildung 4.4: LCP-Häufigkeiten der Testdateien des Pizza&Chili Corpus. Auf der logarithmischen X-Achse ist der LCP-Wert, auf der Y-Achse die Häufigkeit des LCP-Wertes aufgetragen.

Wir betrachten nun den gesamten Speicherverbrauch des Algorithmus. Dieser ist abhängig von k, der Anzahl an Bits die initial pro LCP-Wert zur Verfügung stehen und s, der maximalen Anzahl an Intervallen, welche in der *Queue* gespeichert werden. Für das Abspeichern der LCP-Werte werden $(k + 1,25) \cdot n$ Bits benötigt. Werden die Intervalle in der *Queue* gespeichert, so werden $2 \log_2 n$ Bits pro Intervall benötigt, insgesamt also $2s \log_2 n$ Bits. Die Berechnung von *getIntervals* wird in diesem Fall mittels Wavelet Tree durchgeführt. Der Wavelet Tree benötigt maximal $n \log_2 \sigma$ Bits, zuzüglich $0,25(n \log_2 \sigma)$ Bits um die Rank-Anfrage in den inneren Knoten des Wavelet Trees in konstanter Zeit beantworten zu können. Werden die Intervalle hingegen in zwei Bitarrays der Länge $2n$ gespeichert, so werden $4n$ Bits zur Speicherung benötigt. Da die Berechnung von *getIntervals* dann über die *LF*-Funktion erfolgt, wird hierfür kein Hauptspeicher benötigt. Der Speicherbedarf beträgt somit $(k+1,25) \cdot n + \max\{4n\ ;\ 1,25(n \log_2 \sigma) + 2s \log_2 n\}$. Wird beispielsweise $k = 8$ und $s = \frac{1}{4 \log_2 n}$ gewählt, so liegt der Speicherbedarf selbst bei $\sigma = 2^8$ bei knapp unter $20n$ Bits, also $2,5n$ Bytes.

Die Zeitkomplexität zum Berechnen der Intervalle kann mit $\mathcal{O}(n \log \sigma)$ abgeschätzt werden. Die Funktion *getNextInterval* benötigt, wenn sie mittels Bitarray implementiert ist, $\frac{2n}{i}$ Operationen pro Intervall, wobei i die Anzahl gespeicherter Intervalle ist. Da die Intervalle nur in Bitarrays gespeichert sind, falls mindestens $\mathcal{O}(\frac{n}{\log n})$ Intervalle gespeichert werden, beträgt die Laufzeitkomplexität $\mathcal{O}(\log n)$ pro Intervall insgesamt also $\mathcal{O}(n \log n)$. Das Speichern der LCP-Werte hat im Worst Case eine Laufzeitkomplexität von $\mathcal{O}(n^2)$. Bei einer rein theoretischen Betrachtung dominiert dieser Wert, daher hat der Algorithmus insgesamt eine Laufzeitkomplexität von $\mathcal{O}(n^2)$. In der Praxis sind die LCP-Werte meist so verteilt, dass die LCP-Werte nur wenige Male ausgelagert werden müssen und die Zeit dafür vernachlässigt werden kann. Unter dieser Annahme, oder auch beim Speichern der LCP-Werte in ein Array mit $n \log_2 n$ Bits, beträgt die Laufzeit daher $\mathcal{O}(n \log n)$.

5

Experimentelle Messungen

Um die Praxistauglichkeit des in Kapitel 4 vorgestellten Algorithmus zu testen, wurde dieser implementiert und seine Laufzeit sowie sein Speicherbedarf gemessen. In diesem Kapitel wird zunächst die verwendete Testumgebung erläutert und auf die eigene Implementierung eingegangen. Abschließend werden die Testergebnisse vorgestellt und bewertet.

5.1 Testumgebung

Zum Vergleich wurden die in Kapitel 3 vorgestellten Algorithmen eingesetzt. Der in [KMP09] vorgestellte *KLAAP*-Algorithmus wird meist in seiner speicheroptimierten Variante genutzt, daher wurde diese auch hier verwendet. Aus den Φ-Algorithmen wurde der auf Geschwindigkeit optimierte Φ1, sowie die speichersparsamen Varianten Φ4, Φ64, Φ4-*Semi* und Φ64-*Semi* ausgewählt. Schließlich wurde mit dem in [GO10] vorgestellten *go*-Φ ein weiterer speichereffizienter LCP-Algorithmus ins Testfeld aufgenommen.

Zum Erstellen des Suffix-Arrays wurde der Divsufsort-Algorithmus benutzt. Obwohl dieser eine Worst-Case-Laufzeit von $\mathcal{O}(n \log n)$ besitzt, ist er in der Praxis einer der schnellsten Suffix-Array-Konstruktionsalgorithmen. Die Implementierung[1] stammt von Yuta Mori. Um die BWT direkt aus dem Text – also ohne Verwendung des Suffix-Arrays – zu erstellen, wurde der in [OS09] vorgestellte Algorithmus benutzt.

Der Pizza&Chili Corpus[2] bietet Testdateien für verschiedene Anwendungsgebiete in unterschiedlicher Größe an. Für den Test wurden jeweils die 200 MB Varianten der folgenden Dateien ausgewählt:

[1] http://code.google.com/p/libdivsufsort/
[2] http://pizzachili.dcc.uchile.cl/texts.html

1. DNA (16 / 97.979 / 59, 96439)
2. ENGLISH (239 / 987.770 / 9390, 247)
3. PROTEINS (27 / 45.704 / 278, 4906)
4. SOURCES (230 / 307.871 / 373, 2572)
5. XML (97 / 1.084 / 44, 15651)

Die Zahlen in Klammern geben dabei jeweils die Alphabetgröße / den maximalen LCP-Wert / den durschnittlichen LCP-Wert an.

Da der betrachtete LCP-Konstruktionsalgorithmus insbesondere für große DNA-Daten interessant ist, wurden die Algorithmen zusätzlich auf den folgenden DNA-Dateien getestet:

1. Gasterosteus_aculeatus.BROADS1.62.dna.toplevel.fa.gz:
   Die 446 MB große DNA des Dreistachligen Stichlings.
2. Gallus_gallus.WASHUC2.62.dna.toplevel.fa.gz:
   Die 1.050 MB große DNA des Haushuhns.
3. Choloepus_hoffmanni.choHof1.62.dna.toplevel.fa.gz:
   Die 2.060 MB große DNA des Zweifinger-Faultieres.
4. Pongo_abelii.PPYG2.62.dna.toplevel.fa.gz:
   Die 3.093 MB große DNA des Sumatra-Orang-Utans.

Die Daten wurden im FASTA-Format[3] heruntergeladen und so bereinigt, dass darin nur noch die vier Buchstaben A, C, G und T vorkommen, welche für die vier organischen Basen Adenin, Cytosin, Guanin und Thymin stehen.

Gemessen wurde die mittels dem Unix-Kommando *time* ermittelte *realtime*. Um die damit verbundenen Messungenauigkeiten auszugleichen, wurde jeweils der Mittelwert aus zehn verschiedenen Läufen ermittelt.

Sämtliche Messungen wurden auf einem mit 32 GB Arbeitsspeicher und zwei Six-Core AMD Opteron™ 2431 mit 2, 4 GHz ausgestatteten Rechner durchgeführt. Die Algorithmen wurden mit dem *gcc*-Compiler (Version 4.4.3) und den Übersetzoptionen *-O9 -DNDEBUG* ausgeführt. Als Betriebssystem kam die 64-Bit Version von *Ubuntu* mit der Kernelversion 2.6.32 zum Einsatz.

[3] ftp://ftp.ensembl.org/pub/release-62/fasta/

5.2 Eigene Implementierung

Die Implementierung des Algorithmus erfolgte in der Programmiersprache C++, da diese eine maschinennahe und damit besonders effiziente Programmierung ermöglicht. Zudem sind die Vergleichsalgorithmen ebenfalls in C++ bzw. C geschrieben, was einen fairen Vergleich ermöglicht.

Die erstellte Implementierung basiert auf der von Simon Gog am Institut für theoretische Informatik der Universität Ulm in C++ entwickelten *succinct data structure library* [Gog09]. Insbesondere wurde die enthaltene Implementierung eines Wavelet Trees, die Implementierung von Vektoren für Ganzzahlen mit beliebiger aber fester Bitbreite zwischen 1 und 64 Bit sowie Algorithmen zur Rank-Anfrage verwendet.

In Abschnitt 4.4 wurde der Parameter k eingeführt. Dieser gibt an, wie viele Bits pro LCP-Eintrag initial zur Verfügung stehen. In der in diesem Abschnitt getesteten Implementierung wurde k auf den Wert 8 gesetzt. Dies scheint in der Praxis ausreichend, da bei allen verwendeten Testdaten bereits bei $k = 6$, die Auslagerung der LCP-Werte nur zweimal erfolgen muss.

Die Speicherung der Intervalle kann sowohl mittels *Queue* als auch mittels Bitarray erfolgen (Abschnitt 4.3). Wie in Abschnitt 4.5 dargestellt, ist es vorteilhaft, die Berechnung von *getIntervals* mithilfe der *LF*-Funktion durchzuführen, wenn die Intervalle im Bitarray gespeichert sind. Andernfalls ist die Berechnung über den Wavelet Tree sowohl speichersparender als auch schneller. Um zu ermitteln, ab welcher Anzahl es effizienter ist die Intervalle im Bitarray anstatt in der *Queue* zu speichern, wurde die Laufzeit und der Speicherbedarf für verschiedene Grenzen experimentell ermittelt.

	DNA		ENGLISH		PROTEINS		SOURCES		XML	
$\frac{n}{16}$	70	2,7	162	2,9	138	2,8	324	2,8	228	2,3
$\frac{n}{32}$	67	2,2	131	2,4	138	2,3	165	2,5	227	2,3
$\frac{n}{64}$	65	1,9	126	2,1	134	2,1	136	2,3	106	2,2
$\frac{n}{96}$	66	1,8	123	2	129	2	132	2,2	101	2,1
$\frac{n}{128}$	66	1,8	125	2	137	1,9	130	2,1	103	2,1
$\frac{n}{160}$	69	1,7	123	2	134	1,9	130	2,1	105	2,1
$\frac{n}{192}$	69	1,7	125	2	137	1,9	130	2,1	107	2,1
$\frac{n}{224}$	70	1,7	125	1,9	136	1,9	126	2,1	109	2,1
$\frac{n}{256}$	71	1,7	128	1,9	133	1,9	132	2,1	110	2

Tabelle 5.1: Für jede Datei steht in der ersten Spalte die Laufzeit in Sekunden und in der zweiten Spalte der Speicherverbrauch pro Eingabezeichen in Byte.

Tabelle 5.1 zeigt die Laufzeit und den Speicherbedarf bei verschiedenen Grenzen zwischen $\frac{n}{16}$ und $\frac{n}{256}$ auf den Dateien des Pizza&Chili Corpus. Wie erwartet nimmt der Speicherbedarf mit kleiner werdender Grenze ab. Bis ca. $\frac{n}{96}$ verbessert sich zudem auch die Laufzeit. Wird die Schranke noch kleiner gewählt, so nimmt die Laufzeit wieder zu. Daher wurde der Algorithmus so implementiert, dass die Intervalle im Bitarray gespeichert werden, sobald ihre Anzahl größer als $\frac{n}{96}$ ist.

5.3 Testergebnisse

Die Testergebnisse für die Dateien des Pizza&Chili Corpus sind in der Tabelle 5.2 dargestellt, während sich in Tabelle 5.3 die Ergebnisse für die DNA-Dateien befinden. Die Tabellen sind jeweils in drei Bereiche aufgeteilt. Im obersten Abschnitt ist die Laufzeit und der Speicherverbrauch für die Konstruktion des Suffix-Arrays bzw. der BWT aufgeführt. Im mittleren Bereich sind die Laufzeiten und der Speicherverbrauch für die LCP-Konstruktionsalgorithmen dargestellt. Diese Werte beziehen sich nur auf die Konstruktion des LCP-Arrays ohne die Berechnung der dafür nötigen Datenstruktur – also des Suffix-Arrays bzw. der BWT – zu berücksichtigen. Dies ist im letzten Abschnitt aufgeführt. Dort kann abgelesen werden wie viel Zeit insgesamt und wie viel Speicher maximal benötigt wird, um für einen Text das LCP-Array zu erstellen.

Wie man Tabelle 5.2 entnehmen kann, ist die Suffix-Array-Konstruktion in allen Testfällen schneller als das direkte Erstellen der BWT. Der Geschwindigkeitsvorteil bewegt sich dabei zwischen Faktor $1,3$ bei DNA und $2,1$ bei PROTEINS.

Zur Konstruktion des LCP-Arrays benötigt der Φ1-Algorithmus in allen Testfällen am wenigsten Zeit. Er ist ungefähr $1,5$ mal schneller als der nächstschnellere *KLAAP*-Algorithmus. Allerdings haben beide Algorithmen mit $9n$ Bytes den höchsten Speicherverbrauch. Mit $6n$ bzw. $5,1n$ Bytes benötigt der Φ4-Algorithmus bzw. Φ64-Algorithmus weniger Speicher. Allerdings sind die Varianten Φ4-*Semi* und Φ64-*Semi* etwas schneller, obwohl diese durch das Auslagern des Suffix-Arrays weitere $4n$ Bytes einsparen. Hieran erkennt man, dass sich in manchen Fällen die Reduktion des Hauptspeicherbedarfs auch positiv auf die Geschwindigkeit auswirkt. Der Algorithmus *go*-Φ ist – bei einem Speicherverbrauch von $2n$ Bytes – zum Teil deutlich schneller als die speichereffizienten Φ-Algorithmen.

Der hier vorgestellte Algorithmus ist, bis auf DNA, bei allen Testfällen der langsamste LCP-Konstruktionsalgorithmus. Gegenüber *go*-Φ benötigt er bis zu $2,6$ mal

	DNA 200 MB		ENGLISH 200 MB		PROTEINS 200 MB		SOURCES 200 MB		XML 200 MB	
SA Konstr.	71	5	64	5	72	5	45	5	49	5
dBWT Konstr.	93	1,9	109	2,2	150	2,6	87	2,2	83	2,2
KLAAP	58	9	48	9	48	9	33	9	32	9
Φ1	37	9	30	9	30	9	22	9	22	9
Φ4	83	6	74	6	78	6	60	6	63	6
Φ64	80	5,1	76	5,1	78	5,1	64	5,1	75	5,1
Φ4-*Semi*	78	2	72	2	72	2	59	2	63	2
Φ64-*Semi*	76	1,1	70	1,1	70	1,1	56	1,1	73	1,1
go-Φ	53	2	74	2	70	2	51	2	49	2
Neuer Algo.	66	1,8	124	2	137	2	131	2,2	99	2,1
KLAAP	129	9	112	9	120	9	78	9	81	9
Φ1	108	9	94	9	102	9	67	9	71	9
Φ4	154	6	138	6	150	6	105	6	112	6
Φ64	151	5,1	140	5,1	150	5,1	109	5,1	124	5,1
Φ4-*Semi*	149	5	136	5	144	5	104	5	112	5
Φ64-*Semi*	147	5	134	5	142	5	101	5	122	5
go-Φ	124	5	138	5	142	5	96	5	98	5
Neuer Algo.	159	1,9	233	2,2	287	2,6	218	2,2	182	2,2

Tabelle 5.2: Für jede Datei ist in der ersten Spalte die Laufzeit in Sekunden und in der zweiten Spalte der Speicherverbrauch pro Eingabezeichen in Byte angegeben.

länger, gegenüber Φ1 sogar bis zu sechsmal länger. Allerdings schwankt die Laufzeit des neuen Algorithmus stark. So ist er auf den Testfällen mit kleiner Alphabetgröße – zum Beispiel DNA und XML – bis zu doppelt so schnell wie bei größerer Alphabetgröße. Dies liegt daran, dass mit kleiner Alphabetgröße auch der Wavelet Tree kleiner wird und sich dadurch die Pfade von der Wurzel bis zu den Blättern verkürzen. Damit sinkt der Rechenaufwand wenn die Methode *getIntervals* mittels Wavelet Tree berechnet wird.

Der Speicherbedarf des neuen Algorithmus liegt zwischen $1,8n$ Bytes bei DNA und $2,2n$ Byte bei SOURCES. Diese Schwankung wird ebenfalls vom Wavelet Tree verursacht: Aufgrund der geringen Alphabetgröße der Testdatei DNA, wird weniger Speicherplatz für die Datei benötigt. Dass der Speicherverbrauch bei ENGLISH trotz größerem Alphabet geringer ist als bei SOURCES, liegt am Huffman-Schema des Wavelet Trees. Da die Buchstabenhäufigkeit bei der Testdatei ENGLISH stärker schwankt als bei SOURCES, kann der Wavelet Tree ENGLISH besser komprimieren und benötigt daher weniger Speicherplatz.

Für sich allein betrachtet, hat der neue Algorithmus daher gegenüber den Algorithmen Φ4-*Semi* und *go*-Φ keine Vorteile, da er langsamer ist und mit $2n$ Bytes ungefähr den gleichen Speicherbedarf hat. Allerdings setzen alle anderen Algorithmen das Suffix-Array voraus, für dessen Konstruktion $5n$ Bytes benötigt werden. Daher benötigt der Ansatz, das LCP-Array aus dem Suffix-Array zu berechnen mindestens $5n$ Bytes, unabhänig davon, ob der LCP-Konstruktionsalgorithmus $1,1n$ oder $5n$ Bytes benötigt.

Der in dieser Arbeit vorgestellte Algorithmus berechnet das LCP-Array dagegen allein aus der BWT, welche mit ungefähr $2n$ Bytes berechnet werden kann. Um zu einem gegebenen Text das LCP-Array zu berechnen, benötigt dieser Ansatz folglich nur zwischen $1,9n$ und $2,6n$ Bytes, wie die letzte Zeile von Tabelle 5.2 zeigt. Damit verbunden ist eine bis zu doppelt so lange Laufzeit.

In Tabelle 5.3 sind die Ergebnisse der Laufzeit- und Speichermessung auf den in Abschnitt 5.1 beschriebenen DNA-Daten dargestellt. Auf diesen benötigt das direkte Erzeugen der BWT nicht nur weniger Speicher als die Suffix-Array-Konstruktion, sondern ist für große Daten auch schneller. Dies wird insbesondere auf der 3 GB großen DNA-Datei deutlich, für welche die BWT fast 20% schneller erzeugt werden kann als das Suffix-Array. Da dieser Testfall aus mehr als 2^{31} Zeichen besteht, können die Werte nicht mehr mit vorzeichenbehafteten 32-Bit Integer-Werten dargestellt werden. Folglich musste auf die 64-Bit Implemenierung des Divsufsort-Algorithmus zurückgegriffen werden, wodurch allerdings der Speicherbedarf auf $9n$ Bytes ansteigt. Für die Algorithmen Φ4 und Φ64 lag, im Gegensatz zu den anderen Φ-Algorithmen, nur eine vorzeichenbehafteten 32-Bit Implemenierung vor, sodass diese nicht auf der DNA des Orang-Utans getestet werden konnte.

Bei der LCP-Array-Konstruktion ist auf den DNA-Daten der *go*-Φ-Algorithmus am schnellsten. Wie sich bereits bei der DNA-Datei des Pizza&Chili Corpus abzeichnete, profitiert der neue Algorithmus von einer kleinen Alphabetgröße. Bei allen ausgewählten DNA-Daten ist er nach *go*-Φ und Φ1 der drittschnellste Algorithmus. Er ist – bei einem über viermal geringeren Speicherverbrauch – schneller als der *KLAAP*-Algorithmus und deutlich schneller als die Speichereffizienten Φ-Algorithmen. Beim Speicherverbrauch ändert sich im Vergleich zur DNA Datei des Pizza&Chili Corpus dagegen nichts.

Wie auch schon bei den Dateien des Pizza&Chili Corpus ist der *go*-Φ-Algorithmus dem neuen Algorithmus vorzuziehen, wenn allein die Konstruktion des LCP-Arrays betrachtet wird. Da *go*-Φ aber das Suffix-Array benötigt ist dieses Verfahren nur dann sinnvoll, wenn $5n$ Bytes Hauptspeicher für die Konstruktion des Suffix-Arrays

	Stichling 446 MB		Haushuhn 1.050 MB		Faultier 2.060 MB		Orang-Utan 3.093 MB	
SA Konstr.	171	5	471	5	1.100	5	2.013	9
dBWT Konstr.	204	2	549	1,9	1.062	1,9	1.686	1,9
KLAAP	150	9	454	9	951	9	1.527	9
Φ1	98	9	318	9	756	9	1.183	9
Φ4	187	6	534	6	1.236	6	-	-
Φ64	193	5,1	522	5,1	1.163	5,1	-	-
Φ4-*Semi*	182	2	523	2	1.183	2	1.786	2
Φ64-*Semi*	180	1,1	454	1,1	1.064	1,1	1.648	1,1
go-Φ	117	2	316	2	685	2	1.041	2
Neuer Algo.	141	1,8	338	1,8	800	1,8	1.270	1,8
KLAAP	321	9	925	9	2.051	9	3.540	9
Φ1	269	9	789	9	1.856	9	3.196	9
Φ4	358	6	1.005	6	2.336	6	-	-
Φ64	364	5,1	993	5,1	2.263	5,1	-	-
Φ4-*Semi*	353	5	994	5	2.283	5	3.799	9
Φ64-*Semi*	351	5	925	5	2.164	5	3.661	9
go-Φ	288	5	787	5	1.785	5	3.054	9
Neuer Algo.	345	2	887	1,9	1.862	1,9	2.956	1,9

Tabelle 5.3: Für jede Datei ist in der ersten Spalte die Laufzeit in Sekunden und in der zweiten Spalte der Speicherverbrauch pro Eingabezeichen in Byte aufgeführt.

zur Verfügung stehen. Demgegenüber kann das LCP-Array mit nur $2n$ Bytes berechnet werden, indem die BWT direkt aus dem Text berechnet und anschließend mit dem neuen Algorithmus das LCP-Array allein aus der BWT erstellt wird. Wie Tabelle 5.3 im untersten Abschnitt zeigt, spart dieses Verfahren $3n$ Bytes und ist nur geringfügig langsamer.

Die Messergebnisse zeigen somit, dass der in Kapitel 4 vorgestellte Algorithmus wie vermutet mit $2n$ Bytes sehr speichersparsam ist. Darüber hinaus weisen die Laufzeitmessungen nach, dass die Geschwindigkeit des neuen Algorithmus in der Praxis nicht dramatisch schlechter ist, als die bisherigen LCP-Array-Konstruktionsalgorithmen. Vielmehr ist er in manchen Fällen, insbesondere bei kleiner Alphabetgröße wie sie beispielsweise bei DNA-Daten auftritt, konkurrenzfähig. Er eröffnet somit die Möglichkeit, über das direkte Berechnen der BWT, das LCP-Array mit einem Hauptspeicherbedarf von nur $2n$ Bytes zu erstellen, ohne drastische Geschwindkeitseinbußen in Kauf nehmen zu müssen.

6

Weitere Anwendungen

Der in dieser Arbeit vorgestellte Algorithmus berechnet das LCP-Array direkt aus der BWT. Im Gegensatz zu den LCP-Konstruktionsalgorithmen aus Kapitel 3 werden die LCP-Werte dabei in aufsteigender Reihenfolge berechnet. Diese Eigenschaft ermöglicht es, den Algorithmus so anzupassen, dass er auf andere Problemstellungen angewandt werden kann. In diesem Kapitel werden zwei davon vorgestellt.

Im Folgenden werden auch Strings benötigt, welche nicht mit dem Buchstaben `$` enden. Daher sei in diesem Kapitel $\Sigma' = \Sigma \setminus \{\$\}$ und es gilt, dass ein String über Σ' den Buchstaben `$` nicht enthält. Dagegen kommt `$` bei einem String über Σ weiterhin am Ende und nur dort vor.

6.1 Kürzester fehlender Teilstring

In diesem Abschnitt soll nun das Problem betrachtet werden, für einen gegebenen String über Σ, seine kürzesten fehlenden Teilstrings zu finden. Definition 6.1 erklärt zunächst den Begriff des fehlenden Teilstrings.

Definition 6.1 (Fehlender Teilstring) *Sei S ein String über Σ und ω ein String über Σ'. Dann ist ω genau dann ein fehlender Teilstring, wenn ω kein Teilstring von S ist. ω heißt kürzester fehlender Teilstring, wenn ω ein fehlender Teilstring ist und es kein fehlenden Teilstring ω' mit $|\omega'| < |\omega|$ gibt.*

Für einen String der Länge n über Σ, ist jeder String über Σ' der Länge $n' \geq n$ ein fehlender Teilstring. Von der Berechnung dieser fehlenden Teilstrings soll im Folgenden genauso abgesehen werden, wie von Eingabestrings über einem zweielementigen Alphabet, da es dort nur solche triviale fehlende Teilstrings gibt.

Beispiel 6.2 *Für den String* `annasanannas$` *über* $\Sigma = \{\$, a, n, s\}$ *sind* `aa`, `ns`, `sn` *und* `ss` *kürzeste fehlende Teilstrings. Darüber hinaus sind auch* `ananas`, `sas` *und* `nnnnn` *fehlende Teilstrings, jedoch keine kürzeste fehlende Teilstrings. Im Gegensatz dazu sind* `n$s`, `a$` *und* `$$` *keine fehlende Teilstrings, da sie keine Strings über* Σ' *darstellen. Da* `a`, `sa` *oder* `anna` *in* S *vorkommen, sind diese ebenfalls keine fehlende Teilstrings.*

Fehlende Teilstrings spielen unter anderem in der Bioinformatik eine Rolle und wurden bereits in [HA07] und [HKG08] berechnet. So wird in [HA07] ein Zusammenhang zwischen fehlenden Teilstrings und negativer Selektion vermutet, wenngleich dieser umstritten ist [APCK07]. Daneben können fehlende Teilstrings in der Praxis als Markierung dienen [PFGR09] indem sie in die DNA eingefügt werden.

Eine naheliegende Methode, um (kürzeste) fehlende Teilstrings zu berechnen, ist, für alle möglichen Wörter über Σ' der Länge 1, 2, 3, ... zu überprüfen, ob sie in S vorkommen. Algorithmus 6 kann so modifiziert werden, dass er auf diese Weise die kürzesten fehlenden Teilstrings eines Strings aus seiner BWT berechnet.

Algorithmus 15
Algorithmus zur Berechnung der kürzesten fehlenden Teilstrings eines Strings.

```
([i' ... j'], ω') ← ([1 ... n], ϵ)
length ← n
result ← ∅
repeat
  list ← getIntervals([i' ... j'], ω')
  M ← Σ'
  for all ([i ... j], cω) in list do
    if c ≠ $ then
      M ← M \ {c}
      saveInterval([i ... j], cω)
    end if
  end for
  for all c ∈ M do
    length ← |cω'|
    add(result, cω')
  end for
  ([i' ... j'], ω') ← getNextInterval()
until |ω'| ≥ length
return result
```

Algorithmus 15 erzeugt mit der Funktion *getIntervals* die im String vorkommenden $c\omega'$-Intervalle. Beim anschließenden Durchlaufen der vorkommenden $c\omega$-Intervalle,

werden die Buchstaben c aus der Menge M entfernt und die Intervalle mittels saveInterval gespeichert. Die zweite FOR-Schleife durchläuft anschließend die Buchstaben $c \neq \$$ für die kein $c\omega'$-Intervall existiert. Für diese Buchstaben kommt der String $c\omega'$ folglich nicht im String vor und damit ist $c\omega'$ ein fehlender Teilstring. Die auf diese Weise gefundenen fehlenden Teilstrings werden in der Liste *result* gespeichert. Abschließend gibt *getNextInterval* ein ω'-Intervall mit kürzester Präfixlänge zurück.

Ist ω' nicht kürzer als ein bisher gefundener fehlender Teilstring, so kann ω' nicht zu einem kürzesten fehlenden Teilstring verlängert werden. Da ω' ein Intervall mit kürzester Präfixlänge ist, trifft dies auch auf alle anderen gespeicherten Intervalle zu. In diesem Fall sind alle kürzeste fehlende Teilstrings gefunden und in der Liste *result* gespeichert.

Algorithmus 15 kann als Breitensuche aufgefasst werden. Jedes im String vorkommende ω-Intervall entspricht dabei einem Knoten, welcher die Kinder $c\omega$ hat. Abbildung 6.1 zeigt, wie der Baum für den String `annasanannas$`, von Algorithmus 15 durchlaufen wird.

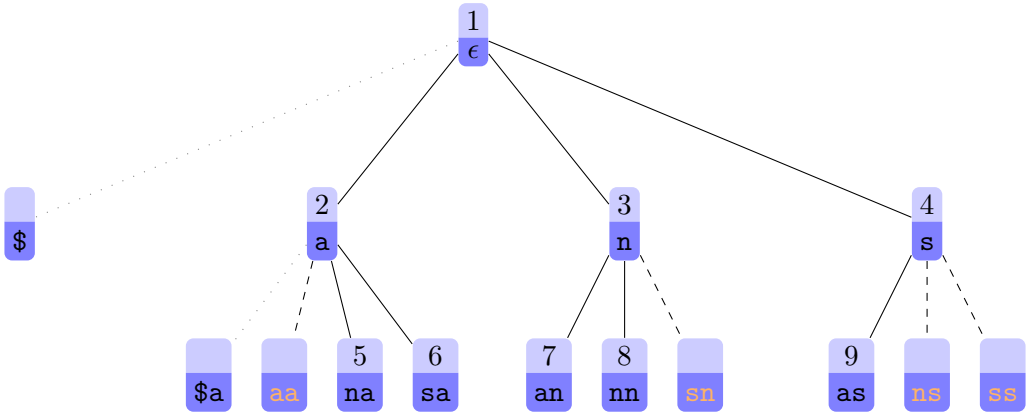


Abbildung 6.1: Visualisierung von Algorithmus 15 auf dem String `annasanannas$`. Nur Knoten zu denen eine schwarze Kante führt werden besucht. Die Zahl im oberen Teil des Knotens gibt die Besuchsreihenfolge an. Knoten zu denen eine gepunktete Linie führt, werden nicht besucht, da fehlende Teilstrings den Buchstaben `$` nicht enthalten dürfen. Die gestrichelten Kanten deuten nicht vorhandene Knoten an – die gesuchten fehlenden Teilstrings.

Die Laufzeit von Algorithmus 15 hängt direkt von der Anzahl der besuchten Knoten – also den erzeugten Intervallen – ab. Da der Algorithmus nacheinander alle σ^i viele Intervalle der Länge i erzeugt, bis ein kürzester fehlender Teilstring gefunden ist, wird eine Abschätzung für die Länge des kürzesten fehlenden Teilstrings benötigt.

In einem String der Länge n, gibt es genau $n - k + 1$ Teilstrings der Länge k. Allerdings gibt es, mit $\tau = |\Sigma'|$, gerade τ^k Strings über Σ' der Länge k. Es müssen daher fehlende Teilstrings der Länge k existieren, wenn $n - k + 1 < \tau^k$ gilt. Daraus ergibt sich, dass für alle $k \geq \log_\tau n$ ein fehlender Teilstring gefunden werden kann.

Folglich werden insgesamt höchstens $\sum_{i=0}^{\lceil \log_\tau n \rceil} \sigma^i$ viele Intervalle erzeugt, bis ein kürzester fehlender Teilstring gefunden wird. Da der Buchstabe `$` nur am Ende der Strings vorkommt und die Intervalle, welche `$` enthalten, nicht gespeichert werden, werden nur k viele Intervalle mit `$` erzeugt. Vernachlässigt man diese, kann die Intervallanzahl mit Hilfe der geometrischen Summe wie folgt abgeschätzt werden:

$$\sum_{i=0}^{\lceil \log_\tau n \rceil} \tau^i = \frac{\tau^{\lceil \log_\tau n \rceil + 1} - 1}{\tau - 1} \leq \frac{\tau^{\log_\tau n + 2}}{\tau - 1} = \frac{n\tau^2}{\tau - 1}$$

Die Worst-Case-Laufzeitkomplexität von Algorithmus 15 liegt daher bei $\mathcal{O}(|\Sigma'| \cdot n)$.

Anders als bei Algorithmus 6, werden die Präfixe der Intervalle für die Ausgabe benötigt. Dies wirkt sich negativ auf den Speicherbedarf von Algorithmus 15 aus, denn im Worst Case müssen bis zu $\tau^{\lceil \log_\tau n \rceil} = n\tau$ viele Intervalle gespeichert werden. Jedes Intervall benötigt bis zu $\log_\tau n$ Bytes für das höchsten $\log_\tau n$ lange Präfix und $2 \log_2 n$ Bits für die Intervallgrenzen. Damit ergibt sich allein für die Intervalle ein Speicherbedarf von $(\log_\tau n + 0,5 \log_2 n) \cdot n\tau$ Bytes.

Um den Speicherbedarf zu reduzieren liegt es nahe, eine Tiefensuche anstatt einer Breitensuche zu verwenden. Algorithmus 16 zeigt einen möglichen Pseudocode.

Anders als bei der Breitensuche ist bei einer Tiefensuche nicht garantiert, dass der erste gefundene fehlende Teilstring ein kürzester fehlender Teilstring ist. In Algorithmus 16 gibt die Variable *length* daher stets die Länge des bisher gefundenen, kürzesten fehlenden Teilstrings an. Initial kann *length* auf den Wert $\lceil \log_\tau n \rceil$ gesetzt werden, da es mindestens ein fehlender Teilstring mit dieser Länge geben muss.

Die Methode *getFirstInterval* gibt das zuletzt gespeicherte Intervall zurück. Es muss somit zunächst überprüft werden, ob dieses Intervall zu einem kürzesten fehlenden Teilstring verlängert werden kann. Ist dies der Fall ($\omega' < length$) so werden – wie zuvor – sämtliche im String vorkommenden Intervalle $c\omega'$ erzeugt und anschließend gespeichert. Schließlich werden die Buchstaben durchlaufen, für welche es kein $c\omega'$-Intervall im String gibt. Hierbei handelt es sich um fehlende Teilstrings, welche in der Liste *result* gespeichert werden. Ist $length > |c\omega'|$, so werden zuvor die bisherigen fehlenden Teilstrings der Länge *length* aus der Liste *result* gelöscht und die Variable *length* auf $|c\omega'|$ gesetzt.

Algorithmus 16
Speichereffizienter Algorithmus zur Berechnung der fehlenden Teilstrings eines Strings S.

```
([i' ... j'], ω') ← ([1 ... n], ϵ)
length ← ⌈log_τ n⌉
result ← ∅
while there are saved intervals do
  ([i' ... j'], ω') ← getFirstInterval()
  if ω' < length then
    list ← getIntervals([i' ... j'], ω')
    M ← Σ'
    for all ([i ... j], cω) in list do
      M ← M \ {c}
      saveInterval([i ... j], cω)
    end for
    for all c ∈ M do
      if length > |cω'| then
        result ← ∅
        length ← |cω'|
      end if
      add(result, cω')
    end for
  end if
end while
```

Algorithmus 16 hat die gleiche Worst-Case-Laufzeit wie Algorithmus 15, da im Worst Case die gleichen Operation – lediglich in anderer Reihenfolge – durchgeführt werden. Es gibt allerdings Eingaben, auf denen die Tiefensuche aufwändiger ist als eine Breitensuche. Dies liegt daran, dass bei der Breitensuche der erste gefundene fehlende Teilstring ein kürzester fehlender Teilstring ist. Im Gegensatz dazu ist es möglich, dass eine Tiefensuche erst sehr spät einen kürzesten fehlenden Teilstring findet und davor eine Vielzahl unnötiger Operationen durchführt.

Der Vorteil von Algorithmus 16 liegt im geringeren Speicherverbrauch. Dies wird deutlich, wenn die Berechnung als Tiefensuche aufgefasst wird. Im Worst Case sind alle Knoten von der Wurzel bis zu einem Knoten der Tiefe $\lceil \log_\tau n \rceil$ sowie deren Kindknoten gespeichert. Da jeder Knoten ein Intervall repräsentiert sind dies $\tau \cdot \lceil \log_\tau n \rceil$ viele Intervalle. Jedes gespeicherte Intervall benötigt $\log_\tau n + 0,5 \log_2 n$ Bytes, womit sich für die Intervalle ein Speicherbedarf von $(\log_\tau n + 0,5 \log_2 n) \cdot \tau \cdot \lceil \log_\tau n \rceil$ Bytes ergibt. Die Speicherkomplexität sinkt daher von $\mathcal{O}(|\Sigma'| \cdot n \cdot \log n)$ bei der Breitensuche auf $\mathcal{O}(|\Sigma'| \cdot \log^2 n)$ bei der Tiefensuche.

6.2 Kürzester eindeutiger Teilstring

Wie die kürzesten fehlenden Teilstrings sind auch kürzeste eindeutige Teilstrings unter anderem für Molekularbiologen interessant und wurden bereits in [HPMW05] bestimmt. So sind sie beispielsweise beim Design von Primer relevant [Gus97].

Definition 6.3 (Eindeutiger Teilstring) *Sei S ein String über dem Alphabet Σ und ω ein String über Σ'. Dann ist ω genau dann ein eindeutiger Teilstring, wenn ω genau einmal in S vorkommt. ω heißt kürzester eindeutiger Teilstring, wenn ω ein eindeutiger Teilstring ist und es keinen eindeutigen Teilstring ω' mit $|\omega'| < |\omega|$ gibt.*

Anders ausgedrückt, ist ω über Σ' genau dann ein eindeutiger Teilstring, wenn das zugehörige ω-Intervalle eine Intervallgröße von 1 hat.

Beispiel 6.4 *Für den String $S =$ `annasanannas$` über $\Sigma = \{$`$`, `a`, `n`, `s`$\}$ sind unter anderem `ana`, `asa` und `asan` eindeutige Teilstrings. Da eindeutige Teilstrings Strings über Σ' sein müssen, sind Suffixe wie beispielsweise `$`, `s$` und `as$` keine eindeutigen Teilstrings. Des Weiteren sind auch die Strings `an`, `nn` und `na` keine eindeutigen Teilstrings, da sie mehrmals in S auftreten. Der kürzeste eindeutige Teilstring von S ist `sa`.*

Das folgende Lemma zeigt, wie sich mithilfe des LCP-Arrays alle eindeutigen Teilstrings eines Strings berechnen lassen.

Lemma 6.5 *Ein $k+1$ langer Teilstring $S[SA[i] \dots SA[i]+k]$ eines Strings S ist genau dann ein eindeutiger Teilstring, wenn gilt:*

1. $SA[i] + k < n$
2. $k \geq \max\{LCP[i], LCP[i+1]\}$

Beweis: Sei $\omega = S[SA[i] \dots SA[i]+k]$ ein eindeutiger Teilstring, dann enthält ω kein \$. Folglich muss $SA[i]+k < n$ und damit Bedingung 1 erfüllt sein. Da ω kein Präfix von $S_{SA[i-1]}$ ist – andernfalls würde ω an zwei Positionen, $SA[i]$ und $SA[i-1]$, in S vorkommen – muss $lcp(S_{SA[i-1]}, S_{SA[i]}) = LCP[i] < |\omega| = k+1$ sein. Außerdem ist ω auch kein Präfix von $S_{SA[i+1]}$, woraus analog folgt, dass $lcp(S_{SA[i]}, S_{SA[i+1]}) = LCP[i+1] < |\omega| = k+1$ ist. Aus $k+1 > LCP[i]$ und $k+1 > LCP[i+1]$ folgt Bedingung 2 des Lemmas.

Sei nun für $\omega = S[SA[i] \dots SA[i] + k]$ die Bedingungen 1 und 2 des Lemmas erfüllt. Aufgrund von $SA[i] + k < n$ enthält $S[SA[i] \dots SA[i] + k]$ den Buchstaben \$ nicht und ist damit ein String über Σ'. Für $j \in \{1, \dots, i-1\}$ folgt mit Lemma 2.15 $lcp(S_{SA[j]}, S_{SA[i]}) = \min\{LCP[k] \mid j < k \leq i\} \leq LCP[i] \leq k$. Da $S_{SA[j]}$ und $S_{SA[i]}$ in weniger als $|\omega| = k+1$ Buchstaben übereinstimmen, ist ω kein Präfix von $S_{SA[j]}$. Analog gilt für $j \in \{i+1, \dots, n\}$, dass $lcp(S_{SA[i]}, S_{SA[j]}) = \min\{LCP[k] \mid i < k \leq j\} \leq LCP[i+1] \leq k$ ist. Damit können $S_{SA[j]}$ und $S_{SA[i]}$ nicht in $|\omega| = k+1$ Buchstaben übereinstimmen; ω kann also kein Präfix von $S_{SA[j]}$ sein.

ω ist also für $j \neq i$ kein Präfix von $S_{SA[j]}$ und kommt deshalb nur einmal – an Position $SA[i]$ – in S vor. □

Die eindeutigen Teilstrings, welche an Position $SA[i]$ beginnen, können folglich leicht bestimmt werden, wenn der LCP-Wert an den Stellen i und $i+1$ bekannt ist. Anders herum können eindeutige Teilstrings bereits während der Berechnung des LCP-Arrays gefunden werden:

Nach dem Eintragen eines LCP-Wertes an Stelle i, wird zusätzlich überprüft, ob der LCP-Wert an Position $i+1$ bereits berechnet ist. Ist dies der Fall, so kann das Maximum der beiden Werte – und damit die eindeutigen Teilstrings, welche an Position $SA[i]$ beginnen – bestimmt werden. Danach wird überprüft, ob der LCP-Wert an Position $i-1$ bereits bekannt ist. In diesem Fall kann das Maximum von $LCP[i-1]$ und $LCP[i]$ und die eindeutigen Teilstrings, welche an Position $SA[i-1]$ beginnen, berechnet werden.

Der in Kapitel 4 vorgestellte Algorithmus 7 kann also leicht modifiziert werden, um zusätzlich zum LCP-Array auch die eindeutigen Teilstrings eines Strings zu berechnen. Soll nicht das LCP-Array, sondern nur die eindeutigen Teilstrings berechnet werden, so kann das LCP-Array durch ein Bitarray ersetzt werden, in welchem das Bit an Position i angibt, ob der LCP-Wert an Position i bereits berechnet wurde.

Da die LCP-Werte in Algorithmus 17 in aufsteigender Reihenfolge berechnet werden, ist ein neu eingetragener Wert nie kleiner als alle bisher eingetragenen Werte – die Berechnung des Maximums kann also entfallen. Darüber hinaus kann die Berechnung nach dem ersten gefundenen eindeutigen Teilstrings abgebrochen werden, wenn man lediglich an einem kürzesten eindeutigen Teilstring interessiert ist.

Zuletzt soll die Frage geklärt werden, wie die Strings $S[SA[i] \dots SA[i] + k]$ für $\max\{LCP[i], LCP[i+1]\} \leq k \leq n - SA[j+1] - 1$ ohne das Suffix-Array SA ausgegeben werden können. Da die Suffixe lexikografisch sortiert sind, kann der erste

Algorithmus 17
Algorithmus, welcher neben dem LCP-Array auch eindeutige Teilstrings berechnet. Die Änderungen gegenüber Algorithmus 7 sind rot markiert.

```
LCP[i] ← ⊥ ∀i ∈ {1, ..., n}
saveInterval([1 ... n])
(counter, size, lcp) ← (1, 0, 0)
repeat
  [i ... j] ← getNextInterval()
  counter ← counter − 1
  list ← getIntervals([i ... j])
  for all [i ... j] in list do
    if saveLCPValue(j + 1, lcp) then
      saveInterval([i ... j])
      size ← size + 1
      if LCP[j] ≠ ⊥ and i ≠ idx then
        for k ← lcp to n − SA[j] − 1 do
          add(uniquewords, S[SA[j] ... SA[j] + k])
        end for
      end if
      if LCP[j + 2] ≠ ⊥ and i ≠ idx then
        for k ← lcp to n − SA[j + 1] − 1 do
          add(uniquewords, S[SA[j + 1] ... SA[j + 1] + k])
        end for
      end if
    end if
  end for
  if counter = 0 then
    (counter, size, lcp) ← (size, 0, lcp + 1)
  end if
until all lcp-values are calculated
```

Buchstabe – also $S[SA[i]]$ – mittels binärer Suche auf dem C-Array in $\log_2 \sigma$ Schritten ermittelt werden. Anschließend muss der restliche String $S[SA[i]+1 \dots SA[i]+k]$ ausgegeben werden. Da $\Psi(i) = SA^{-1}[SA[i]+1]$ ist, gilt $S[SA[i]+1 \dots SA[i]+k] = S[SA[\Psi(i)] \dots SA[i]+k]$. Dessen erster Buchstabe kann wieder mittels binärer Suche ermittelt werden. Somit können die Strings rekonstruiert werden, indem k mal eine binäre Suche und die Ψ-Funktion angewandt wird. Das Ende des Strings kann an $S[SA[i]] = \$$ bzw. $i = 1$ erkannt werden kann. Da die Ψ-Funktion mittels Wavelet Tree in $\mathcal{O}(\log \sigma)$ berechnet werden kann, beträgt die Laufzeitkomplexität zur Ausgabe $\mathcal{O}(k \log \sigma)$.

7

Zusammenfassung und Ausblick

In Kapitel 3 wurden die bisherigen Standardalgorithmen zur Konstruktion des LCP-Arrays eines Strings betrachtet. Sie alle benötigen das entsprechende Suffix-Array, um daraus das LCP-Array zu erstellen. Im Zentrum dieser Arbeit stand der in Kapitel 4 entworfene neue LCP-Konstruktionsalgorithmus, welcher im Gegensatz dazu nur die Burrows-Wheeler-Transformierte des Strings verwendet. Da die Konstruktion der BWT nur die Hälfte des für die Suffix-Array-Konstruktion benötigen Speicherplatzes erfordert, wird es durch den neuen Algorithmus möglich, auf gleicher Hardware das LCP-Array eines bis zu doppelt so langen Strings zu erstellen.

Der vorgestellte Algorithmus hat eine Laufzeitkomplexität von $\mathcal{O}(n \log n)$ und ist, wie die Messergebnisse in Kapitel 5 zeigen, in der Praxis etwa zwei- bis dreimal langsamer als die LCP-Konstruktionsalgorithmen mit vergleichbarem Speicherbedarf. Auf Eingaben mit geringer Alphabetgröße – wie beispielsweise bei DNA-Sequenzen – ist der Geschwindigkeitsnachteil jedoch wesentlich geringer.

Zudem kann der Algorithmus möglicherweise an verschiedenen Stellen noch verbessert werden.

- Für den sehr unwahrscheinlichen Fall, dass die LCP-Werte eines Strings gleichmäßig verteilt sind, ist das derzeitige Speicherverfahren für das LCP-Array sehr langsam. Der Algorithmus könnte daher um eine, auf diesen Fall spezialisierte, Datenstruktur zur Speicherung der LCP-Werte ergänzt werden.

- Im Algorithmus werden Intervalle in zwei verschiedenen Datenstrukturen gespeichert. Bis zu einer Intervallanzahl von s werden die Intervalle in einer *Queue*, andernfalls in einem Bitarray gespeichert. Zwar wurde s in Abschnitt 5.2 experimentell ermittelt, allerdings wurde dabei die Alphabetgröße des Testfalls nicht berücksichtigt. Möglicherweise kann die Laufzeit jedoch verbessert werden, wenn s abhängig von der Alphabetgröße gewählt wird.

- Die Speicherung von Intervallen in einem Bitarray kann vermutlich mittels einer Baumstruktur mit Verzweigungsgrad k beschleunigt werden. Hierzu wird für jeweils k Bits des Bitarrays ein weiteres Bit erzeugt, welches genau dann den Wert null hat, falls alle zugeordneten Bits null sind. Diese neu erzeugten Bits werden solange nach dem gleichen Schema zusammengefasst, bis nur noch k Bits übrig sind, welche die Wurzel bilden. Abbildung 7.1 zeigt dieses Schema für ein Bitarray der Länge 27 und einem Verzweigungsgrad von $k = 3$.

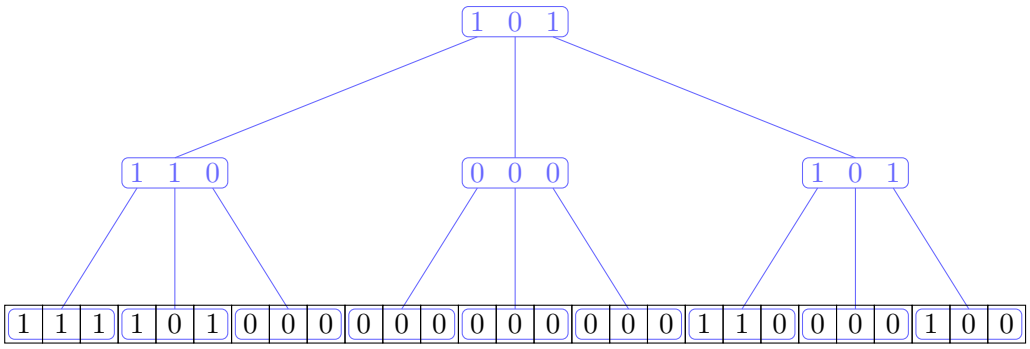


Abbildung 7.1: Ein Bitarray zur Speicherung von Intervallen. In Blau ist die vorgeschlagene Baumstruktur mit einem Verzweigungsgrad von $k = 3$ dargestellt.

  Diese Baumstruktur ermöglicht es nun, unmarkierte Bereiche im Bitarray zu überspringen und dadurch die Positionen der markierten Einträge schneller zu finden. In der Praxis bietet sich ein Verzweigungsgrad von 64 an, da dies der Wortbreite moderner Prozessoren entspricht.

Da das Suffix-Array eine sehr wichtige Datenstruktur ist, wird in Zukunft sicherlich an Algorithmen geforscht, welche dieses mit einem geringeren Speicherverbedarf erstellen können. So ist es beispielsweise denkbar, das Suffix-Array aus der BWT oder blockweise [FGM10] zu erstellen. Aufgrund des in dieser Arbeit vorgestellten Algorithmus ist es jedoch bereits jetzt möglich, Probleme, welche mittels der BWT und dem LCP-Array gelöst werden können, mit geringem Speicherbedarf zu bewältigen. In Kapitel 6 wurden bereits zwei dieser Probleme vorgestellt. In einem weiteren Schritt könnte daher untersucht werden, welche anderen Probleme sich mit dem neuen LCP-Array-Konstruktionsalgorithmus speichereffizienter lösen lassen.

Literaturverzeichnis

[APCK07] Acquisti, Claudia ; Poste, George ; Curtiss, David ; Kumar, Sudhir: Nullomers: Really a Matter of Natural Selection? In: *PLoS ONE* 2 (2007), 10, Nr. 10, e1022. http://dx.doi.org/10.1371/journal.pone.0001022. – DOI 10.1371/journal.pone.0001022

[BW94] Burrows, Michael ; Wheeler, David J.: A Block-sorting Lossless Data Compression Algorithm. 1994 (124). – Forschungsbericht

[CNPT10] Culpepper, J. S. ; Navarro, Gonzalo ; Puglisi, Simon J. ; Turpin, Andrew: Top-k Ranked Document Search in General Text Databases. In: *Proceedings of the 18th annual European conference on Algorithms: Part II*. Berlin, Heidelberg : Springer-Verlag, 2010 (ESA'10). – ISBN 3–642–15780–7, 978–3–642–15780–6, 194–205

[FGM10] Ferragina, Paolo ; Gagie, Travis ; Manzini, Giovanni: Lightweight Data Indexing and Compression in External Memory. Version: 2010. http://dx.doi.org/10.1007/978-3-642-12200-2_60. In: López-Ortiz, Alejandro (Hrsg.): *LATIN 2010: Theoretical Informatics* Bd. 6034. Springer Berlin / Heidelberg, 2010, 697-710. – 10.1007/978-3-642-12200-2_60

[GGV03] Grossi, Roberto ; Gupta, Ankur ; Vitter, Jeffrey S.: High-Order Entropy-Compressed Text Indexes. In: *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 2003 (SODA '03). – ISBN 0–89871–538–5, 841–850

[GO10] Gog, Simon ; Ohlebusch, Enno: Lightweight LCP-Array Construction in Linear Time. In: *CoRR* abs/1012.4263 (2010)

[Gog09] Gog, Simon: Broadword Computing and Fibonacci Code Speed Up Compressed Suffix Arrays. Version: 2009. http://dx.doi.org/10.1007/978-3-642-02011-7_16. In: Vahrenhold, Jan (Hrsg.): *Experimental Algorithms* Bd. 5526. Springer Berlin / Heidelberg, 2009, 161-172. – 10.1007/978-3-642-02011-7_16

[Gus97] Gusfield, Dan: Algorithms on Stings, Trees, and Sequences: Computer Science and Computational Biology. In: *SIGACT News* 28 (1997), December, 41–60. http://dx.doi.org/http://doi.acm.org/10.1145/270563.571472. – DOI http://doi.acm.org/10.1145/270563.571472. – ISSN 0163–5700

[HA07] Hampikian, Greg ; Andersen, Tim: Absent Sequences: Nullomers and Primes. In: *Pacific Symposium on Biocomputing* Bd. 12 Stanford University, Stanford University, 2007, S. 355–366

[HKG08] Herold, Julia ; Kurtz, Stefan ; Giegerich, Robert: Efficient computation of absent words in genomic sequences. In: *BMC Bioinformatics* 9 (2008), Nr. 1, 167. http://dx.doi.org/10.1186/1471-2105-9-167. – DOI 10.1186/1471–2105–9–167. – ISSN 1471–2105

[HPMW05] Haubold, Bernhard ; Pierstorff, Nora ; Moller, Friedrich ; Wiehe, Thomas: Genome comparison without alignment using shortest unique substrings. In: *BMC Bioinformatics* 6 (2005), Nr. 1, 123. http://dx.doi.org/10.1186/1471-2105-6-123. – DOI 10.1186/1471–2105–6–123. – ISSN 1471–2105

[Huf52] Huffman, David A.: A Method for the Construction of Minimum-Redundancy Codes. In: *Proceedings of the IRE* 40 (1952), sept., Nr. 9, S. 1098–1101. http://dx.doi.org/10.1109/JRPROC.1952.273898. – DOI 10.1109/JRPROC.1952.273898. – ISSN 0096–8390

[KA03] Ko, Pang ; Aluru, Srinivas: Space Efficient Linear Time Construction of Suffix Arrays. In: *Proceedings of the 14th annual conference on Combinatorial pattern matching*. Berlin, Heidelberg : Springer-Verlag, 2003 (CPM'03). – ISBN 3–540–40311–6, 200–210

[KLA+01] Kasai, Toru ; Lee, Gunho ; Arimura, Hiroki ; Arikawa, Setsuo ; Park, Kunsoo: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: *Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*. London, UK, UK : Springer-Verlag, 2001 (CPM '01). – ISBN 3–540–42271–4, 181–192

[KMP09] Kärkkäinen, Juha ; Manzini, Giovanni ; Puglisi, Simon: Permuted Longest-Common-Prefix Array. Version: 2009. http://dx.doi.org/10.1007/978-3-642-02441-2_17. In: Kucherov, Gregory (Hrsg.) ; Ukkonen, Esko (Hrsg.): *Combinatorial Pattern Matching* Bd. 5577.

Springer Berlin / Heidelberg, 2009, 181-192. – 10.1007/978-3-642-02441-2_17

[KS03] Kärkkäinen, Juha ; Sanders, Peter: Simple Linear Work Suffix Array Construction. Version: 2003. http://dx.doi.org/10.1007/3-540-45061-0_73. In: Baeten, Jos (Hrsg.) ; Lenstra, Jan (Hrsg.) ; Parrow, Joachim (Hrsg.) ; Woeginger, Gerhard (Hrsg.): *Automata, Languages and Programming* Bd. 2719. Springer Berlin / Heidelberg, 2003, 187-187. – 10.1007/3-540-45061-0_73

[KSPP03] Kim, Dong K. ; Sim, Jeong S. ; Park, Heejin ; Park, Kunsoo: Linear-Time Construction of Suffix Arrays. In: *Proceedings of the 14th annual conference on Combinatorial pattern matching*. Berlin, Heidelberg : Springer-Verlag, 2003 (CPM'03). – ISBN 3–540–40311–6, 186–199

[Man04] Manzini, Giovanni: Two Space Saving Tricks for Linear Time LCP Array Computation. Version: 2004. http://dx.doi.org/10.1007/978-3-540-27810-8_32. In: Hagerup, Torben (Hrsg.) ; Katajainen, Jyrki (Hrsg.): *Algorithm Theory - SWAT 2004* Bd. 3111. Springer Berlin / Heidelberg, 2004, 372-383. – 10.1007/978-3-540-27810-8_32

[MM90] Manber, Udi ; Myers, Gene: Suffix Arrays: A New Method for On-Line String Searches. In: *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*. Philadelphia, PA, USA : Society for Industrial and Applied Mathematics, 1990 (SODA '90). – ISBN 0–89871–251–3, 319–327

[MN05] Mäkinen, Veli ; Navarro, Gonzalo: Succinct Suffix Arrays Based on Run-Length Encoding. Version: 2005. http://dx.doi.org/10.1007/11496656_5. In: Apostolico, Alberto (Hrsg.) ; Crochemore, Maxime (Hrsg.) ; Park, Kunsoo (Hrsg.): *Combinatorial Pattern Matching* Bd. 3537. Springer Berlin / Heidelberg, 2005, 121-137. – 10.1007/11496656_5

[NM07] Navarro, Gonzalo ; Mäkinen, Veli: Compressed Full-Text Indexes. In: *ACM Comput. Surv.* 39 (2007), April. http://doi.acm.org/10.1145/1216370.1216372. – ISSN 0360–0300

[OS09] Okanohara, Daisuke ; Sadakane, Kunihiko: A Linear-Time Burrows-Wheeler Transform Using Induced Sorting. In: *Proceedings of the 16th*

International Symposium on String Processing and Information Retrieval. Berlin, Heidelberg : Springer-Verlag, 2009 (SPIRE '09). – ISBN 978–3–642–03783–2, 90–101

[PFGR09] Pinho, Armando ; Ferreira, Paulo ; Garcia, Sara ; Rodrigues, Joao: On finding minimal absent words. In: *BMC Bioinformatics* 10 (2009), Nr. 1, 137. `http://dx.doi.org/10.1186/1471-2105-10-137`. – DOI 10.1186/1471–2105–10–137. – ISSN 1471–2105

[Sch10] Schnattinger, Thomas: *Bidirektionale indexbasierte Suche in Texten*. Deutschland, Universität Ulm, Diplomarbeit, 2010

[Vig08] Vigna, Sebastiano: Broadword Implementation of Rank/Select Queries. In: McGeoch, Catherine (Hrsg.): *Experimental Algorithms* Bd. 5038. Springer Berlin / Heidelberg, 2008, S. 154–168

Erklärung

Hiermit versichere ich, Timo Beller, Matrikelnummer 629727, dass ich die vorliegende Diplomarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ulm, den 27. Juli 2011

Timo Beller