Berechnung der LCP-Tabelle während der Konstruktion eines Suffix-Arrays

Diplomarbeit

Universität Ulm
Fakultät für Ingenieurwissenschaften und Informatik
Institut für Theoretische Informatik

ulm university universität
u ulm

Eingereicht von: Jürgen Reiss
am: 14. Mai 2008
Erstgutachter: Prof. Dr. Enno Ohlebusch, Universität Ulm
Institut für Theoretische Informatik
Zweitgutachter: Prof. Dr. Uwe Schöning, Universität Ulm
Institut für Theoretische Informatik

Inhaltsverzeichnis

1 Einleitung

Motivation

Durch das rasante Anwachsen an digitaler Information, wie beispielsweise im Internet oder in Genomdatenbanken, steht man vor der Herausforderung, die dabei entstehenden Datenmengen so aufzubereiten, dass alle Arten von Anfragen an diese Daten zügig beantwortet werden können. Dazu sind nicht nur auf der Seite der Hardware immer leistungsfähigere Systeme bereitzustellen, sondern die logischen, auf Software basierenden Lösungen nehmen einen mindestens ebenso wichtigen Stellenwert ein. Nur mit effizienten Architekturen, Algorithmen und Datenstrukturen lässt sich die pure Rechenkraft eines Systems ausreizen, womit unter anderem auch erhebliche Kosteneinsparungen verbunden sind. Die Informatik erarbeitet auf diesem Gebiet wichtige Grundlagen und kann für viele Probleme optimale Lösungen anbieten.

Eine spezielle Strukturierungsmethode stellt die vollständige Indizierung von großen Texten dar, mit deren Hilfe sich viele Arten von Suchanfragen schnell beantworten lassen. Die Arbeitsgruppe Bioinformatik an der Universität Ulm beschäftigt sich unter anderem mit dieser Art der Textaufbereitung. Dabei steht mit dem *Suffix-Array* eine Datenstruktur im Mittelpunkt, die große Texte sehr kompakt und speicherschonend repräsentiert. Dadurch lässt sich der zur Verfügung stehende Arbeitsspeicher eines Systems effizient nutzen, sodass mehr Anfragen direkt bearbeitet werden können, ohne dabei auf langsameren Sekundärspeicher zurückzugreifen. Erweitert man das Suffix-Array um die *LCP-Tabelle*, so erreicht man für viele algorithmische Probleme optimale Laufzeiten, die allein mit der Information des Suffix-Arrays nicht möglich sind. Weil die Inhalte der LCP-Tabelle von Informationen des Suffix-Arrays abhängen, wird diese Tabelle meist erst dann konstruiert, wenn das Suffix-Array komplett erstellt ist. Dazu existiert ein Algorithmus, der die LCP-Tabelle in asymptotisch optimaler Zeitkomplexität konstruieren kann. Es stellt sich nun die Frage, ob es während der Konstruktion eines Suffix-Arrays ebenso möglich ist, die LCP-Tabelle effizient zu berechnen. Damit lassen sich zwei grundsätzliche Vorgehensweisen zur Konstruktion der LCP-Tabelle unterscheiden, die in Abb. 1.1 schematisch dargestellt sind.

Ziel der Arbeit

Diese Arbeit soll der Frage nachgehen, wie und mit welchem Aufwand die Berechnung der LCP-Tabelle während der Konstruktion eines Suffix-Arrays realisiert werden kann. Es sollen die verwendeten Konzepte und Methoden vorgestellt und diskutiert werden, sodass am Ende der Arbeit ein Fazit gezogen werden kann, ob sich mit diesen Ansätzen die parallele Berechnung der LCP-Tabelle aus praktischer Sicht lohnt.

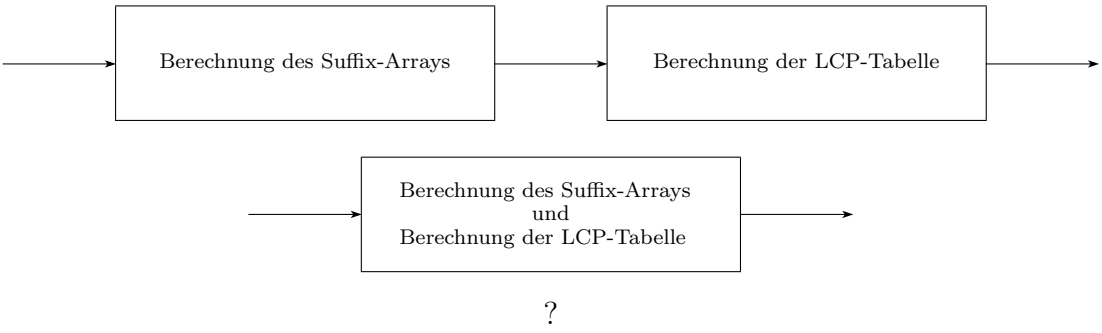


Abbildung 1.1: Die beiden Vorgehensweisen zur Konstruktion der LCP-Tabelle. In der oberen Darstellung wird zunächst das Suffix-Array berechnet, um danach die LCP-Tabelle mit dessen Hilfe zu erstellen. In der unteren Darstellung wird die LCP-Tabelle während der Berechnung des Suffix-Arrays mit aufgebaut. Diese parallele Berechnung soll in der vorliegenden Arbeit untersucht werden.

Vorgehensweise

Um die vorgestellte Fragestellung anzugehen, werden in Kapitel 2 die benötigten Begriffe und formalen Grundlagen eingeführt. Dazu gehören das Suffix-Array, die LCP-Tabelle und weitere Beziehungen im Kontext von Strings.

Die in dieser Arbeit vorgestellten Suffix-Array Konstruktionsalgorithmen, im Folgenden mit SAKA abgekürzt, verwenden intern Sortierverfahren für Strings. Die wichtigsten hiervon werden in Kapitel 3 vorgestellt und sind speziell für den Inhalt dieser Arbeit aufbereitet.

Kapitel 4 stellt die Bedeutung der LCP-Tabelle in den Mittelpunkt, die im Zusammenspiel mit dem Suffix-Array effiziente Algorithmen auf indizierte Texte ermöglicht. Es wird das *Enhanced Suffix-Array* vorgestellt, das von der Bioinformatik-Gruppe der Universität Ulm entwickelt wurde. Weiter wird ein bezüglich Laufzeit asymptotisch optimaler Algorithmus zur Konstruktion der LCP-Tabelle erläutert, der erst angewendet werden kann, nachdem das Suffix-Array vorliegt.

Mit zwei sehr ähnlichen SAKAs beginnt in Kapitel 5 die eigentliche Untersuchung der gegebenen Fragestellung. Einer davon war der chronologisch erste verfügbare SAKA und ist von den gleichen Autoren vorgestellt worden, die auch das Suffix-Array entworfen haben. Dieser benutzt eine spezielle Sortiertechnik, die eine effiziente Konstruktion des Suffix-Arrays ermöglicht. Die gleiche Technik verwendet auch der zweite untersuchte Algorithmus, der durch eine geschicktere Vorgehensweise eine aus praktischer Sicht bessere Laufzeit erzielt. Beide Algorithmen arbeiten phasenweise und es ist zu untersuchen, ob eine parallele Berechnung der LCP-Tabelle sich in dieses Schema effizient einbetten lässt.

Bis zum Jahr 2003 existierten keine Algorithmen, die ein Suffix-Array, ohne den Umweg über *Suffix-Bäume*, in optimaler linearer Zeit bezogen auf die Länge des Eingabetextes aufbauen konnten. Etwa zeitgleich erschienen in diesem Jahr drei Verfahren, die diese Zeitkomplexität erreichten. Ihr gemeinsames Merkmal ist das rekursive Vorgehen und einer der Algorithmen wird in Kapitel 6 vorgestellt. Es ist zu prüfen, inwieweit eine Berechnung der LCP-Tabelle ökonomisch in die vorgegebene rekursive Struktur des Algorithmus eingebracht werden kann.

Aus einer anderen SAKA-Klasse stammt der in Kapitel 7 vorgestellte Algorithmus, der gegenwärtig zu den praktisch schnellsten gehört, die ein Suffix-Array aufbauen, obwohl die asymptotische Laufzeit größer als quadratisch geschätzt wird. Er vereinigt viele bekannte Ideen und Heuristiken aus anderen Verfahren und es ist zu untersuchen, ob sich diese Kombination aus Ideen für eine parallele Berechnung der LCP-Tabelle ausnutzen lässt.

Für alle vorgestellen SAKAs werden vorhandene Implementierungen in C/C++ so erweitert, dass auch die LCP-Tabelle nach Terminierung der Algorithmen vorliegt. Diese erweiterten Implementierungen werden in Kapitel 8 auf verschiedene, praktisch relevante Eingaben getestet und ausgewertet. Insbesondere sollen die Unterschiede zur Vorgehensweise mit der LCP-Berechnung nach Erstellung des Suffix-Arrays beleuchtet werden.

Schließlich werden in Kapitel 9 die Testergebnisse diskutiert und beurteilt. Das gezogene Fazit soll für die benutzten Konzepte und Methoden einen Hinweis geben, inwiefern sich diese für die parallele Berechnung der LCP-Tabelle als geeignet herausgestellt haben.

Im Anhang befinden sich Anmerkungen zu den erweiterten Implementierungen, um den Einblick in technische Details zu erleichtern. Die beiliegende CD enthält den gesamten verwendeten Quellcode.

2 Grundlagen und Definitionen

2.1 Bezeichnungen

Sei Σ ein nichtleeres Alphabet. Σ^n bezeichnet die Menge aller Wörter der Länge $n \in \mathbb{N}$ über Σ. Die Vereinigung aller Mengen von Wörtern über Σ mit beliebigen Längen ist mit $\Sigma^* := \bigcup_{i=0}^{\infty} \Sigma^i$ definiert. Dabei ist $\Sigma^0 := \{\varepsilon\}$ und ε das *leere Wort* mit Länge 0. Schließlich ist $\Sigma^+ := \Sigma^* \backslash \Sigma^0$. In Definition 2.1.1 werden Notationen und Begriffe eingeführt, die im Kontext von Strings benötigt werden.

Definition 2.1.1 *Sei $s \in \Sigma^n$ ein String und $i, j \in \mathbb{N}$ mit $0 \leq i \leq j < n$. Die Indizierung der Symbole von s beginnt stets bei 0.*

1. *$s[i]$ bezeichnet das $(i+1)$-te Symbol von s*
2. *$|s| := n$ bezeichnet die Länge von s*
3. *$[i..j] := \{i, i+1, \ldots, j\}$*
4. *$[i..j) := [i..j-1]$*
5. *$s[i..j] := s[i]s[i+1]\ldots s[j]$ ist ein Teilstring von s*
6. *$s[i..j) := s[i..j-1]$*
7. *$s_i := s[i..n)$ bezeichnet das i-te Suffix von s*
8. *$s[0..i]$ bezeichnet ein Präfix von s*

Anstatt Suffix s_i wird abkürzend auch Suffix i verwendet.

2.2 Ordnungsrelationen

Um mit Strings arbeiten zu können benötigt man Vergleichsaussagen zwischen ihnen. Die in dieser Arbeit häufig verwendete *lexikographische Ordnung* basiert auf diesen Aussagen. Zunächst wird mit 2.2.1 eine strikte Ordnung auf Σ definiert.

Definition 2.2.1 *Sei* $< \subseteq \Sigma \times \Sigma$ *eine binäre Relation und* $a, b, c \in \Sigma$. $<$ *wird definiert durch:*

1. $\forall x \in \Sigma : x \not< x$ *(Irreflexivität)*
2. $a < b \Rightarrow b \not< a$ *(Asymmetrie)*
3. $(a < b) \wedge (b < c) \Rightarrow a < c$ *(Transitivität)*

Die zu $<$ *symmetrische* $>$*-Relation ergibt sich über* $a > b \Leftrightarrow b < a$.

Mit 2.2.2 wird vorstehende Definition erweitert auf Strings aus Σ^*.

Definition 2.2.2 *Sei* $< \subseteq \Sigma^* \times \Sigma^*$ *eine binäre Relation und* $s, t \in \Sigma^*$. $<$ *wird definiert durch:*

1. $s \neq \varepsilon \Rightarrow \varepsilon < s$
2. $\forall u \in \Sigma^* : (u \not< u) \wedge (u \not< \varepsilon)$
3. $s < t \Leftrightarrow s[0] < t[0] \vee (s[0] = t[0] \wedge s_1 < t_1)$

Die zu $<$ *symmetrische* $>$*-Relation ergibt sich über* $s > t \Leftrightarrow t < s$.

Um auch reflexive Aussagen zu ermöglichen, wird schließlich in 2.2.3 die lexikographische Ordnung definiert.

Definition 2.2.3 *Sei* $\leq \subseteq \Sigma^* \times \Sigma^*$ *eine binäre Relation und* $s, t \in \Sigma^*$. *Man nennt* $\leq$ *die lexikographische Ordnung und diese wird definiert durch:*

$$s \leq t \Leftrightarrow (s < t) \vee (s = t)$$

Die zu $\leq$ *symmetrische* $\geq$*-Relation ergibt sich über* $s \geq t \Leftrightarrow t \leq s$.

Abkürzend wird für die lexikographische Ordnung auch der Begriff *Lexorder* verwendet. Betrachtet man zwei Strings $s, t \in \Sigma^+$ mit Hilfe der Lexorder, so lässt sich eine Aussage $s \leq t$ oder $t \leq s$ treffen. Dies wird durch Definition 2.2.4 auch für Präfixe gleicher Länge von s und t ermöglicht.

Definition 2.2.4 *Sei* $\leq_H \subseteq \Sigma^+ \times \Sigma^+$ *eine binäre Relation mit* $H \in \mathbb{N} \backslash \{0\}$. *Weiter seien* $s, t \in \Sigma^+$ *und* $u^H := u[0..H)$ *für alle* $u \in \Sigma^+$. $\leq_H$ *wird definiert durch:*

$$s \leq_H t \Leftrightarrow (s^H \leq t^H) \vee (s^H = t^H)$$

Die zu $\leq_H$ symmetrische $\geq_H$-Relation ergibt sich über $s \geq_H t \Leftrightarrow t \leq_H s$ und anstelle von $s^H = t^H$ wird in dieser Arbeit die Notation $s =_H t$ verwendet. Für den String $s = acatacg$ ist beispielsweise $s_0 \leq_2 s_4$ wegen $s_0 =_2 s_4$.

2.3 Der Suffix-Baum

Ein Suffix-Baum ist eine Datenstruktur, die alle Suffixe eines Strings repräsentiert und in Definition 2.3.1 eingeführt wird.

Definition 2.3.1 *Sei $s \in \Sigma^n$. Der zu s gehörende Suffix-Baum ST_s ist definiert durch:*

1. *ST_s hat genau n Blätter.*
2. *Jeder innere Knoten von ST_s hat mindestens zwei Söhne.*
3. *Jede Kante von ST_s ist mit einem nichtleeren Teilstring von s beschriftet.*
4. *Alle ausgehenden Kanten eines Knotens beginnen mit einem unterschiedlichen Symbol aus Σ.*
5. *Die Konkatenation der Kantenbeschriftungen auf dem Pfad von der Wurzel zu Blatt i ergibt das i-te Suffix von s.*

Bei dieser Definition ist zu beachten, dass, falls ein Suffix s_i Präfix eines anderen Suffix s_j ist, kein Blatt mit der Beschriftung i existiert und damit die Bedingungen 1. und 5. für ein beliebiges $s \in \Sigma^n$ nicht erfüllt werden können. Suffix s_i wäre implizit auf einer Kante repräsentiert. Ein Beispiel für diese Situation ist der String $s = atcat$, bei dem Suffix $s_3 = at$ Präfix von Suffix $s_0 = atcat$ ist. Um dies zu vermeiden, erweitert man Σ um ein Extrasymbol \$, das genau einmal am Ende von s vorkommt und somit kein Suffix Präfix eines anderen Suffix sein kann. In Abb. 2.1 ist ein Beispiel für einen Suffix-Baum dargestellt, wobei im Folgenden immer $\$ < a$ für alle a aus $\Sigma\backslash\{\$\}$ gilt.

2.4 Das Suffix-Array

Ein String $s \in \Sigma^n$ besitzt genau $|s| = n$ Suffixe s_i mit $s_i = s[i..n)$ für $i \in [0..n)$. Unter Verwendung von 2.2.3 lassen sich alle Suffixe von s in aufsteigender lexikographischer Reihenfolge in ein Feld, genannt *Suffix-Array*, eintragen. Ein Suffix s_i von s ist eindeutig charakterisiert durch seinen Startindex i in s und somit genügt es diesen Index, anstelle aller Symbole des Suffixes, zu speichern[1]. Wegen 2.2.2 gilt $\varepsilon < s$ für alle $s \in \Sigma^+$, sodass sich alle Suffixe von s nur in genau einer Reihenfolge in das Suffix-Array eintragen lassen. In 2.4.1 wird das Suffix-Array formal definiert.

Definition 2.4.1 *Sei $s \in \Sigma^n$. Ein Suffix-Array $SA[0..n)$ ist ein Feld der Länge n, das alle Suffixindizes $i \in [0..n)$ von s enthält, so dass $s_{SA[0]} < s_{SA[1]} < \cdots < s_{SA[n-1]}$ gilt.*

Es gilt $SA[j] = i$ genau dann, wenn s_i das j-te Suffix von SA in aufsteigender lexikographischer Reihenfolge ist. Oft möchte man für ein gegebenes Suffix wissen, an welcher Position es im Suffix-Array zu finden ist. Dies leistet das sogenannte inverse Suffix-Array und wird in 2.4.2 definiert.

[1] Würde man explizit alle Symbole speichern, hätte man wegen $\sum_{i=0}^{n-1} |s_i| \in O(n^2)$ quadratischen Speicherbedarf bezüglich $|s|$.

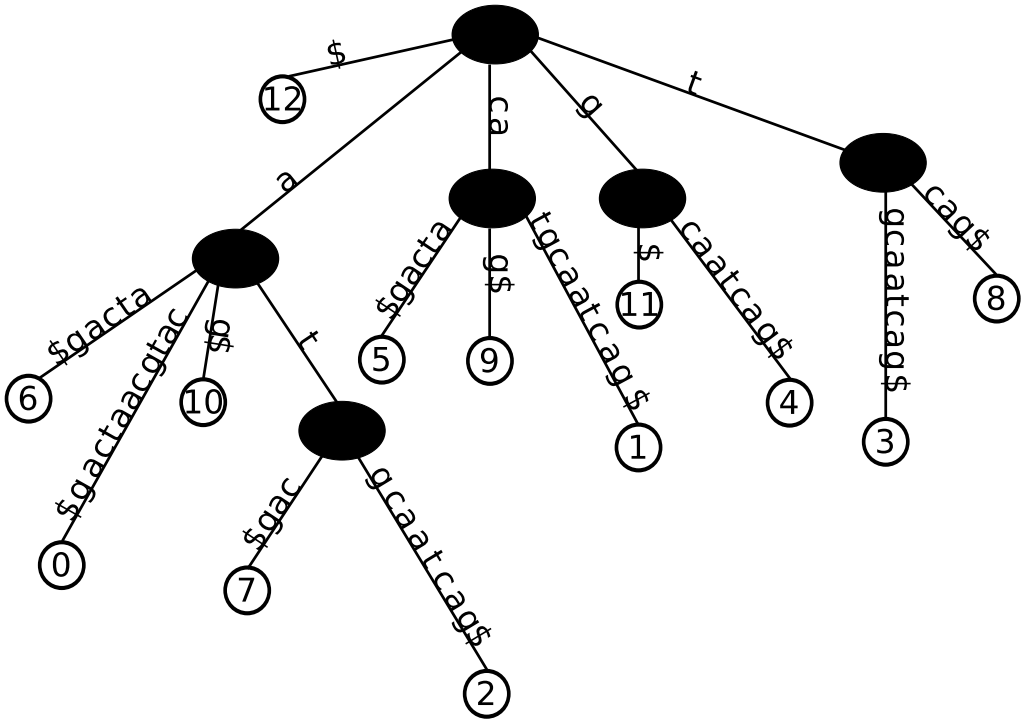


Abbildung 2.1: Der Suffix-Baum ST_s für den String $s = acatgcaatcag\$$.

Definition 2.4.2 *Sei $s \in \Sigma^n$ und SA das zu s gehörige Suffix-Array. Das inverse Suffix-Array $SA^{-1}[0..n)$ ist ein Feld der Länge n und definiert durch:*

$$SA^{-1}[i] = j \Leftrightarrow SA[j] = i$$

Das inverse Suffix-Array SA^{-1} gibt den Rang j in SA für ein Suffix i zurück, während SA das Suffix i liefert, das unter allen Suffixen von s den lexikographischen Rang j besitzt. Über $SA^{-1}[i] < SA^{-1}[j] \Leftrightarrow s_i < s_j$ lässt sich die relative Anordnung zweier Suffixe in SA feststellen. In Tabelle 2.1 ist ein Beispiel für die Felder SA und SA^{-1} dargestellt.

Es wird in dieser Arbeit oftmals ein, bezüglich einer festen Präfixlänge H, partiell geordnetes Suffix-Array vorliegen, in dem sich die Lexorder aller Suffixe in SA nur auf die ersten H Symbole bezieht. Deshalb kann es in diesem Fall sein, dass sowohl $SA^{-1}[i] < SA^{-1}[j]$ als auch $s_i > s_j$ gilt. Um dieses spezielle Suffix-Array zu kennzeichen, wird in 2.4.3 die betrachtete Präfixlänge als Index an SA angehängt.

Definition 2.4.3 *Sei $s \in \Sigma^n$ und $H \in [1..n)$. Ein bezüglich H partiell geordnetes Suffix-Array $SA_H[0..n)$ ist ein Feld der Länge n und definiert durch:*

$$s_{SA_H[0]} \leq_H s_{SA_H[1]} \leq_H \cdots \leq_H s_{SA_H[n-1]}$$

Man sagt, dass SA_H in H-Ordnung vorliegt.

Damit lässt sich analog zu 2.4.2 das inverse Suffix-Array SA_H^{-1} definieren. Das Suffix-Array SA ist für alle H auch ein partielles Suffix-Array.

i	$SA[i]$	$SA^{-1}[i]$	$s_{SA[i]}$	$LCP[i]$
0	12	2	\$	-1
1	6	8	$aatcag\$$	0
2	0	5	$acatgcaatcag\$$	1
3	10	12	$ag\$$	1
4	7	10	$atcag\$$	1
5	2	6	$atgcaatcag\$$	2
6	5	1	$caatcag\$$	0
7	9	4	$cag\$$	2
8	1	11	$catgcaatcag\$$	2
9	11	7	$g\$$	0
10	4	3	$gcaatcag\$$	1
11	8	9	$tcag\$$	0
12	3	0	$tgcaatcag\$$	1

Tabelle 2.1: Suffix-Array SA, inverses Suffix-Array SA^{-1} und die LCP-Tabelle LCP für den String $s = acatgcaatcag\$$.

2.5 Die LCP-Tabelle

Das längste gemeinsame Präfix (longest common prefix) zweier Strings $s, t \in \Sigma^+$ bezeichnet ein gemeinsames Präfix von s und t maximaler Länge. Diese Länge wird durch $lcp(s,t)$ ausgedrückt und in 2.5.1 definiert.

Definition 2.5.1 *Seien $s, t \in \Sigma^+$. Der lcp-Wert $lcp(s,t)$ von s und t wird definiert durch:*

$$lcp(s,t) := \max_{k \in [0..\min(|s|,|t|)]} \{k \mid s[0..k) = t[0..k)\}$$

Die sogenannte LCP-Tabelle enthält die lcp-Werte adjazenter Suffixe in SA und wird in 2.5.2 eingeführt.

Definition 2.5.2 *Sei $s \in \Sigma^n$ und SA das zu s gehörende Suffix-Array. Die LCP-Tabelle $LCP[0..n)$ ist ein Feld der Länge n und definiert durch:*

$$LCP[i] := \begin{cases} -1 & \text{für} \quad i = 0 \\ lcp(s_{SA[i-1]}, s_{SA[i]}) & \text{für} \quad 1 \leq i \leq n-1 \end{cases}$$

In Abb. 2.1 ist in der letzten Spalte ein Beispiel für eine LCP-Tabelle angegeben.

2.6 Das RMQ-Problem

In dieser Arbeit wird es häufig darum gehen, aus einem Feld von Zahlen das Minimum eines bestimmten Bereiches zu finden. Dieses sogenannte *Range Minimum Query Problem*, kurz RMQ-Problem, wird in 2.6.1 definiert.

Definition 2.6.1 *Sei $F[0..n)$ ein Feld der Länge n von reellen Zahlen und $i, j \in [0..n)$ mit $i \leq j$. Das Range Minimum Query Problem besteht darin, einen Index k zu finden, so dass $F[k] = \min\{F[l] \mid i \leq l \leq j\}$ ist.*

Die Suche nach einem solchen k im Intervall $F[i..j]$ wird mit $rmq_F(i, j)$ notiert. Wenn das Minimum aus $F[i..j]$ mehr als einmal in $F[i..j]$ vorkommt, so ist der zurückgegebene Index von $rmq_F(i, j)$ nicht eindeutig bestimmt. Für diesen Fall wird im Folgenden angenommen, dass $rmq_F(i, j)$ den größten Index des Minimums aus $F[i..j]$ zurückgibt, wenn nicht explizit anders angegeben.
Es existieren optimale Lösungen für dieses Problem [11, 3], die für ein beliebiges Intervall $F[i..j]$ den Index k in konstanter Zeit bestimmen und eine lineare Vorverarbeitung des Eingabefeldes F benötigen. In den Kapiteln 5 und 7 werden verschiedene andere Möglichkeiten besprochen, wie $rmq_F(i, j)$ realisiert werden kann.

3 Sortieralgorithmen für Strings

Bei der Konstruktion eines Suffix-Arrays werden häufig spezielle Sortieralgorithmen für Strings als Hilfsverfahren verwendet. Diese sollen, soweit sie sich im Kontext dieser Arbeit befinden, vorgestellt werden, um damit die benötigten Hilfsmittel für die Kapitel 5, 6 und 7 zur Verfügung zu stellen. Die Betrachtung konzentriert sich auf die lexikographische Sortierung von Strings, obwohl all diese Verfahren auch für andere zu sortierende Typen anwendbar sind.
Alle vergleichsbasierten Sortieralgorithmen, wie z.B. Heapsort, besitzen die untere Schranke $\Omega(n \log n)$ zur Sortierung von n Schlüsseln[29]. Dabei geht man von einem konstanten Aufwand für einen Vergleich der Form $x_i \leq x_j$ aus, um damit eine relative Ordnungsaussage zwischen zwei Schlüsseln x_i und x_j zu erhalten. Die beiden Verfahren aus Abschnitt 3.1 und 3.2 können dagegen n Schlüssel in $O(n)$ sortieren, da diese nicht vergleichsbasiert arbeiten und an die Schlüssel gewisse Bedingungen gestellt werden. Weiter wird in Abschnitt 3.3 ein vergleichsbasierter Algorithmus vorgestellt, der strukturell ähnlich wie Quicksort vorgeht und eine spezielle Ausrichtung auf die effiziente Sortierung von Strings besitzt.

3.1 Bucketsort

Bei vergleichsbasierten Sortieralgorithmen werden an die zu sortierenden Schlüssel keine weiteren Bedingungen gestellt. Insbesondere darf der Wertebereich der Schlüssel unendlich groß sein, wie dies z.B. bei reellen Zahlen der Fall ist. Bucketsort[1], auch Fachverteilen genannt, ist ein Sortierverfahren, das ohne explizite Schlüsselvergleiche arbeitet. Deshalb kann dieses Verfahren, in Abhängigkeit von den Typen der zu sortierenden Schlüssel, die $\Omega(n \log n)$-Schranke für vergleichsbasierte Algorithmen unterbieten. Dies geschieht auf Kosten von Einschränkungen, die an die Schlüsselwerte zu stellen sind. Um eine worst-case Laufzeit von $O(n)$ zu erreichen, darf dazu die Anzahl der von den Schlüsseln annehmbaren Werte nur endlich sein. Möchte man beispielsweise ein Feld $X[0..n)$ von n Zahlen aus $\mathbb{N}$ mit Wertebereich $[0..k)$ sortieren, legt man ein zweites Feld $Y[0..k)$ an, das die Häufigkeit jedes Wertes aus X enthält. Y wird zunächst an allen Positionen mit 0 initialisiert und anschließend durchläuft man die Eingabe X sequentiell. Für jeden dabei angetroffenen Wert $X[i]$ an Position i wird $Y[X[i]]$ inkrementiert. Damit hat man die Werte der Eingabe X als Indizes für das Feld Y benutzt und im letzten Schritt nutzt man die Häufigkeitsinformation in Y, um die sortierte Folge in X zu gewinnen. Für jeden Wert $Y[i]$ wird $Y[i]$-mal der Wert i nach $X[j]$ geschrieben, wobei j die aktuelle Position in X ist, an der ein Wert in die Sortierung eingefügt wird. Algorithmus 3.1 gibt dieses Vorgehen wieder.

[1] Bucket ≡ Eimer

Algorithmus 3.1 : Pseudocode für Bucketsort, wenn die Eingabewerte natürliche Zahlen aus $[0..k)$ sind

Eingabe : Eingabe X bestehend aus n natürlichen Zahlen im Bereich $[0..k)$
Ausgabe : Aufsteigend sortiertes Feld X

```
1 ∀i ∈ [0..k) : Y[i] ← 0
2 ∀i ∈ [0..n) : Inkrementiere Y[X[i]]
3 j ← 0
4 for i ∈ [0..k) do
5     while (Y[i] > 0) do
6         X[j] ← i
7         Inkrementiere j
8         Dekrementiere Y[i]
```

Der Aufwand für das Initialisieren von Y ist offensichtlich $O(k)$ und für das Zählen der Häufigkeiten aller Werte $O(n)$. Die geschachtelte Schleife ab Zeile 4 liegt in $O(k+n)$, da nicht für alle k Iterationen der äußeren Schleife die innere n-mal ausgeführt wird. Denn für das Zählen der Häufigkeiten in Zeile 2 wurden im Feld Y genau n Inkrementoperationen vollzogen und deshalb können in der inneren Schleife, über alle k Iterationen der äußeren Schleife auch nur genau n Dekrementoperationen vorgenommen werden. Damit ergibt sich für 3.1 eine worst-case Laufzeit von $O(k+n)$ und falls k konstant ist, $O(n)$.

Ist man an der Sortierung ganzer Datensätze interessiert, geht man ähnlich vor. Sind beispielsweise Telefonnummern zu sortieren, so will man meist auch die zugehörigen Namen oder anderweitig verknüpfte Daten mit in die Sortierung aufnehmen. Um Datensätze als Ganzes zu behandeln, legt man ein Feld Y an, das an jeder Position eine verkettete Liste enthält. Hat man allgemein n Datensätze in einem Feld L, die durch einen Schlüsselwert *key* im Bereich $[0..k)$ repräsentiert sind, so beginnt man zunächst mit der Initialisierung von Y mit k leeren Listen. Dann entnimmt man L nacheinander einen Datensatz X und hängt diesen an das Ende der Liste $Y[key(X)]$. Im letzten Schritt werden die verketteten Listen aus Y aufsteigend konkateniert, um so die sortierte Liste aller Datensätze zu erhalten. In Algorithmus 3.2 ist diese Vorgehensweise dargestellt.
Das Initialisieren von Y und die Konkatenierung aller Listen geschieht jeweils in $O(k)$. Das Durchgehen aller n Datensätze von L mit der Eintragung in eine verkettete Liste benötigt $O(n)$ viele Schritte, womit sich eine Gesamtlaufzeit von $O(k+n)$ ergibt. Ist k konstant, dann ergibt sich analog zu Algorithmus 3.1 die Komplexität $O(n)$.

Die letzte vorzustellende Variante von Bucketsort behandelt das Sortieren von Strings beliebiger Länge. In diesem Fall sind die unterschiedlichen Längen der Strings so zu verwalten, dass eine effiziente Sortierung möglich wird. Denn ein Auffüllen von kurzen Strings mit einem Wert, der eine Leerstelle repräsentiert, kann die Laufzeit extrem verschlechtern. Hat man beispielsweise $n-1$ Strings der Länge 1, die mit $n-1$ Leerstellen aufgefüllt werden, und einen String der Länge n, so führt ein n-faches Bucketsort für alle Symbolpositionen zu einem quadratischen Aufwand, obwohl die aufsummierte Länge aller Strings in $O(n)$ liegt, wenn nicht mit Leerstellen aufgefüllt wird.

Algorithmus 3.2 : Bucketsort für Datensätze unter Verwendung von verketteten Listen

Eingabe : Eingabe L bestehend aus n Datensätzen, wobei ein Datensatz X durch einen Schlüssel $key(X) \in [0..k)$ repräsentiert ist

Ausgabe : Aufsteigende Sortierung aller Datensätze nach ihren Schlüsselwerten

```
1 ∀i ∈ [0..k) : Y[i] ← leere Liste
2 while L ≠ ∅ do
3 |   Entnimm einen Datensatz X aus L
4 |   Trage X an das Ende der Liste Y[key(X)] ein
5 for i ∈ [0..k) do
6 |   Konkateniere Y[i] in L
```

Um dies zu umgehen, bildet man eine Liste von $(i, s[i])$-Paaren, deren erste Komponente die Symbolposition i und die zweite Komponente das Symbol $s[i]$ ist. Diese Paare werden mit einem Bucketsort aus 3.2 nach $s[i]$ sortiert. Anschließend werden in gleicher Weise die Positionen i sortiert und man erhält mehrere Listen $Symb[i]$, die angeben welche Buckets für jede betrachtete Symbolposition später besetzt sein werden. Dies erspart für jede Position ein komplettes Durchlaufen aller Buckets, um festzustellen ob in einem Bucket ein Wert abgelegt ist. Im letzten vorbereitenden Schritt wird ein Bucketsort auf die Längen aller Strings durchgeführt, um in die spätere Sortierung nur Strings mit entsprechender Länge aufzunehmen. Die eigentliche Sortierung beginnt dann mit den Strings der größten Länge m für die Symbole an den Positionen $s[m]$. Nach dem Aufteilen dieser Symbole in die Buckets lassen sich die letztgenannten nun effektiv mit Hilfe der zuvor gebildeten Listen $Symb$ aufsuchen. Dazu werden einfach die Buckets für die aktuelle betrachtete Position i über $Symb[i]$ konkateniert. Algorithmus 3.3 gibt dieses allgemeine Bucketsortverfahren für Strings in Pseudocode an.
Für die Laufzeitanalyse sei l_i die Länge des i-ten Strings aus L, $N := \sum_{i=1}^{n} l_i$ die Gesamtlänge aller Strings und $k := |\Sigma|$ die Alphabetgröße. Die Erzeugung der Paare geschieht in $O(N)$ mit einem Durchlauf über alle Strings und die drei vorbereitenden Bucketsort-Aufrufe haben einen Aufwand von $O(N + k)$. Wenn n_i die Anzahl der Strings ist mit einem Symbol an Position i, dann benötigt jede Iteration der While-Schleife $O(n_i)$-Zeit und wegen $\sum_{i=0}^{m-1} n_i = N$ ist der Gesamtaufwand über alle Iterationen $O(N)$. Insgesamt hat Algorithmus 3.3 damit eine worst-case Laufzeit von $O(N + k)$.

Als Beispiel sei $L = \{b, ab, acb, ac, c\}$ lexikographisch zu sortieren. Zunächst werden die Strings sequentiell durchlaufen und man erhält die Paare:

$$(0,b),(0,a),(1,b),(1,c),(2,b),(0,c)$$

Ein Bucketsort nach den Symbolen dieser Paare führt zu:

$$(0,a),(0,b),(1,b),(2,b),(1,c),(0,c)$$

Nun wird nach den Symbolpositionen wieder ein Bucketsort auf diese Paare durchgeführt und ergibt die Listen $Symb[i]$.

Algorithmus 3.3 : Pseudocode für Bucketsort mit Strings beliebiger Länge als Eingabe

Eingabe : Eingabe L bestehend aus Strings beliebiger Länge
Ausgabe : Lexikographische Sortierung der Strings in L

```
1  Erzeuge eine Liste von (i, s[i])-Paaren
2  Bucketsort auf s[i] dieser Paare
3  Bucketsort auf i dieser Paare → führt zu Listen Symb[i]
4  Bucketsort auf die Längen der Strings
5  m ← größte vorkommende Länge in L
6  L = ∅
7  while m − 1 ≥ 0 do
8      Füge alle Strings der Länge m in die Sortierung ein
9      Bucketsort auf Symbole s[m − 1] aller in der Sortierung befindlichen Strings
10     Aufsuchen aller nichtleeren Buckets über Symb[m − 1]
11     Dekrementiere m
12 Trage Sortierung in L ein
```

Symb[0]:	*(0,a)*	→	*(0,b)*	→	*(0,c)*
Symb[1]:	*(1,b)*	→	*(1,c)*		
Symb[2]:	*(2,b)*				

Als letzte Vorbereitung werden die Strings noch nach ihrer Länge sortiert:

b, c, ab, ac, acb

Da die größte Länge m aller Strings $m = 3$ ist und die Indizierung bei 0 beginnt, wird im ersten Durchlauf nur der String *acb* ausgewählt. Das letzte Symbol ist *b* und kommt in das entsprechende Bucket. Über *Symb*[2] ist nun bekannt, dass nur im *b*-Bucket ein Eintrag vorhanden ist. Im nächsten Lauf kommen die beiden Strings *ab* und *ac* mit Länge 2 hinzu. Die drei Symbole an Position 2 von den in der Sortierung befindlichen Strings verteilen sich auf die Buckets *b* und *c*. Genau dies spiegelt sich in der Liste *Symb*[1] wider, sodass man das *a*-Bucket nicht aufsuchen muss Der letzte Schritt mit *Symb*[2] verläuft dann ganz analog und man erhält die Sortierung $L = \{ab, ac, acb, b, c\}$.

Für alle vorgestellen Bucketsortvarianten sind zusätzliche Felder bereitzustellen, sodass diese Verfahren nicht in-place arbeiten. Ein weiterer Nachteil ergibt sich, wenn $n \ll k$ ist, wobei n die Anzahl und k der Wertebereich der zu sortierenden Schlüssel ist. In diesem Fall ist Bucketsort nicht mehr praktikabel, da sehr viel mehr ungenutzter Speicher zur Verfügung gestellt werden muss, als zu sortierende Schlüssel vorhanden sind. Das Verteilen und Aufsammeln aller Schlüssel in bzw. aus den Buckets geschieht offensichtlich so, dass die relative Ordnung gleicher Schlüssel in der Eingabe nicht verändert wird. Deshalb gehört Bucketsort zu den stabilen Sortierverfahren.

3.2 Radixsort

Wie in Abschnitt 3.1 erwähnt, eignet sich Bucketsort nur dann, falls die Anzahl der Schlüssel gegenüber dem Wertebereich eines Schlüssels nicht zu klein ist . In Algorithmus 3.1 wird für jeden möglichen Wert ein Bucket bereitgestellt. Für sehr große Wertebereiche ist dieses Vorgehen im Hinblick auf die Laufzeit[2]und vor allem bezüglich des bereitzustellenden Speichers nicht praktikabel. Wenn man wieder natürliche Zahlen aus einem großen Bereich sortieren will, so kann man auf Kosten der Laufzeit gegenüber Bucketsort die Bucketanzahl klein halten. Dazu stellt man die gegebenen Schlüssel geeignet über eine Basiszahl dar. Eine Zahl $x \in \mathbb{N}$ lässt sich bezüglich einer Basis b schreiben als:

$$x = \sum_{i=0}^{k} a_i b^i$$

Dabei ist k die Stelligkeit von x und a_i die Ziffer an Position i. Stellt man nun b viele Buckets bereit, so lassen sich in k vielen Bucketsort-Durchläufen auch große Werte sortieren. Die Sortierung beginnt mit der niederwertigsten Stelle $k = 0$ aller Schlüssel, wodurch diese in die b Buckets verteilt werden. Anschließend sammelt man die Schlüssel in aufsteigender Folge aus den Buckets wieder ein. Dann folgt ein Bucketsort auf die vorletzte Stelle aller Schlüssel usw. Der Übergang von Zahlen auf Strings gleicher Länge k ist somit einfach. Es werden alle Symbole bzw. deren numerischer Wert als Schlüssel verwendet und es sind $b = |\Sigma|$ viele Buckets bereitzustellen. Dieses symbolweise Vorgehen von Radixsort ist in Algorithmus 3.4 dargestellt.

Algorithmus 3.4 : Pseudocode für Radixsort

Eingabe : Eingabe L bestehend aus n Strings gleicher Länge k
Ausgabe : Lexikographische Sortierung der Strings in L

```
1 for i ∈ [0..k) do
2     Bucketsort an Postion i aller Strings
```

Die Laufzeit ergibt sich mit der k-maligen Anwendung von Bucketsort aus 3.1 zu $O(kn)$ und damit, falls k konstant ist, hat auch Radixsort eine lineare Laufzeit.

Als Beispiel seien die Strings *MON, FRE, DIE, DON* zu sortieren. Damit sind $|\Sigma| = 8$ Buckets vorzusehen und es werden drei Läufe von Bucketsort benötigt. In Abb. 3.1 sind die Strings nach jedem Bucketlauf dargestellt. Durch die symbolweise Sortierung bleibt die relative Ordnung gleicher Schlüssel erhalten, sodass auch Radixsort stabil arbeitet.

3.3 Ternary Split Quicksort

Der bekannte Quicksort-Algorithmus partitioniert das Eingabefeld bezüglich des gewählten Pivotelements in zwei Bereiche.

[2]Beim Aufsammeln der Schlüssel im letzten Schritt müssen immer alle Buckets besucht werden.

$$\begin{array}{cccccccc} MON & & FRE & & DIE & & DIE \\ FRE & \rightarrow & DIE & \rightarrow & MON & \rightarrow & DON \\ DIE & & MON & & DON & & FRE \\ DON & & DON & & FRE & & MON \end{array}$$

Abbildung 3.1: Sortierung von 4 Strings der Länge 3 mit Hilfe von Radixsort. Zuerst wird nach der letzten Position sortiert, dann nach der mittleren und schließlich nach der ersten.

Ternary Split Quicksort[6, 7], im Folgenden mit TSQS abgekürzt, teilt die Eingabe in drei Bereiche auf:

- Bereich A: Elemente die kleiner sind als das Pivotelement
- Bereich B: Elemente die gleich dem Pivotelement sind
- Bereich C: Elemente die größer sind als das Pivotelement

Diese Partitionierung ist in Abb.3.2 schematisch dargestellt. Die Bereiche A und C werden rekursiv sortiert und das weitere Vorgehen für Bereich B hängt vom Anwendungskontext ab.

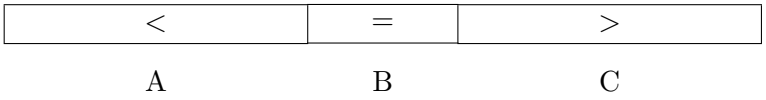


Abbildung 3.2: Partitionierungsschema bei TSQS. Elemente kleiner als das Pivotelement liegen in Partition A, Elemente gleich dem Pivot in B und Elemente größer dem Pivot in C.

Geht es um das Sortieren von Zahlen, so ist man an dieser Stelle fertig, denn Bereich B liegt schon sortiert vor. Sollen dagegen Strings sortiert werden, so werden alle Elemente aus Bereich B rekursiv weiter sortiert, indem man das nächste Symbol aller Strings aus B als Schlüssel verwendet. Die Motivation zur Verwendung von TSQS liegt darin, dass Quicksort alle Elemente die gleich dem Pivotelement sind, auf unterschiedliche Bereiche verteilt, je nach Spezifikation. Bei der Stringsortierung ist es vorteilhaft, wenn alle Strings mit identischem Symbol an der aktuell betrachteten Indexposition sequentiell im Feld gruppiert sind. TSQS leistet diese Umorganisation effizient und Algorithmus 3.5 gibt in sprachlicher Form dessen Vorgehensweise an. Als Eingabe dient ein Feld S mit n Strings als Komponenten. Der Parameter d verweist auf den Symbolindex aller Strings, an dessen Position die Sortierschlüssel entnommen werden. Schließlich ist die Konstante k der größte Index bis zu dessen Wert die Sortierung fortgeführt wird, d.h. k stellt die Länge des längsten Strings dar. Aufgerufen wird TSQS mit dem Tripel $(S, n, 0)$.
Der schwierigste Schritt stellt die Partitionierung in die drei Bereiche A, B und C dar. Bentley und McIlroy stellten hierfür in [6] ein effektives Verfahren vor. Für eine Beschreibung betrachtet man Abb. 3.3.

Algorithmus 3.5 : Pseudocode für TSQS

Eingabe : Eingabefeld $S[0..n)$, Anzahl zu sortierender Strings n, betrachteter Symbolindex d als Schlüssel

Ausgabe : Lexikographische Sortierung der Strings aus S

```
1 if n ≤ 1 OR d > k then
2     return
3 Wähle Pivotelement v.
4 Partitioniere S in die drei Bereiche S_A, S_B und S_C unter Verwendung des
  Schlüssels d und dem Pivot v. Die drei Bereiche haben die Längen n_A, n_B und n_C.
5 Sortiere S_A rekursiv mit (S_A, n_A, d)
6 Sortiere S_B rekursiv mit (S_B, n_B, d + 1)
7 Sortiere S_C rekursiv mit (S_C, n_C, d)
```

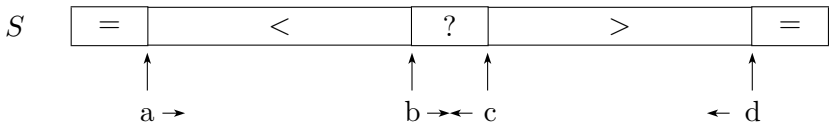


Abbildung 3.3: Momentaufnahme des Partitionierungsverfahrens von TSQS.

Es werden vier Zeiger verwendet, wobei die Zeiger a und b initial auf Position 0 und die Zeiger c und d auf Position $n-1$ des Eingabefeldes zeigen. Strukturell besteht das Partitionierungsverfahren aus einer äußeren und zwei inneren Schleifen. In der ersten inneren Schleife durchläuft Zeiger b aufsteigend die kleineren Elemente bezüglich des Pivotelements v und man tauscht die Schlüssel an den Positionen der Zeiger a und b, falls das betrachtete Element gleich v ist. Anschließend wird a inkrementiert. Dies wird solange wiederholt, bis b auf ein Element trifft, das größer als v ist. Völlig symmetrisch durchläuft der Zeiger c absteigend die größeren Elemente. Trifft man auf ein Element, das gleich v ist, tauscht man die Schlüssel an den Positionen der Zeiger c und d und dekrementiert d anschließend. In dieser Situation werden nun in der äußeren Schleife die Schlüssel an den Positionen der Zeiger b und c getauscht und der gesamte Vorgang beginnt erneut mit der ersten inneren Schleife. Die äußere Schleife wird verlassen, sobald $b > c$ gilt. Im letzten Schritt müssen noch die beiden äußeren =-Bereiche in die Mitte getauscht werden. Dies kann nun effizient ohne weitere Schlüsselvergleiche geschehen, weil die beiden Werte der Zeiger a und b bzw. c und d zur Verfügung stehen. Man berechnet $h = \min\{a, b-a\}$ und dekrementiert den Zeiger b um h. Dann werden alle Elemente ab Position 0 und $b-h$ getauscht, solange bis $h = 0$ gilt. Ganz analog verläuft dieses Umkopieren für die Zeiger c und d. Am Ende liegt die Partitionierung aus Abb. 3.2 vor und der Algorithmus fährt mit der rekursiven Sortierung der drei Bereiche fort, nachdem die neu entstandenen Bereichsgrenzen bestimmt wurden. Das komplette Verfahren der Partitionierung ist in Algorithmus 3.6 wiedergegeben.

Sedgewick und Bentley analysieren in [7] die Laufzeit von TSQS in Abhängigkeit der Methode zur Auswahl des Pivotelements. Es soll hier lediglich das Ergebnis zitiert werden, das man erhält, wenn man zur Auswahl des Medians als Pivotelement cn Vergleiche braucht. Sind dann n Strings mit jeweils k Symbolen zu sortieren, so liegt die Laufzeit in $O(cn(\log n + k))$.

Algorithmus 3.6 : Pseudocode für Partitionierung von TSQS

Eingabe : Eingabefeld S mit n Schlüsseln, vier Zeiger a, b, c, d in S, und v als gewähltes Pivotelement

Ausgabe : Partitionierung von S in die Bereiche A, B und C

```
1  a, b ← 0
2  c, d ← n − 1
3  while TRUE do
4      while (b < c ∧ S[b] ≤ v) do
5          if S[b] = v then
6              Tausche(S[a], S[b])
7              Inkrementiere a
8          Inkrementiere b
9      while (c > b ∧ S[c] ≥ v) do
10         if S[c] = v then
11             Tausche(S[d], S[c])
12             Dekrementiere c
13         Dekrementiere c
14     if b > c then
15         break
16     Tausche(S[b], S[c])
17     Inkrementiere b
18     Dekrementiere c
19 h ← min{a, b − a}
20 k ← b − h
21 for (l ← 0; h > 0; h ← h − 1) do
22     Tausche(S[l], S[k])
23     Inkrementiere l und k
24 h ← min{d − c, n − 1 − d}
25 k ← n − h
26 for (l ← b; h > 0; h ← h − 1) do
27     Tausche(S[l], S[k])
28     Inkrementiere l und k
```

4 Anwendung und Konstruktion der LCP-Tabelle

In diesem Kapitel wird in Abschnitt 4.1 eine wichtige Anwendung der LCP-Tabelle vorgestellt, mit der es möglich ist, den Suffix-Baum und die algorithmischen Verfahren auf diesem vollständig durch ein erweitertes Suffix-Array zu ersetzen. Dieses setzt sich grob beschrieben aus dem Suffix-Array und der zugehörigen LCP-Tabelle sowie weiteren, vom Anwendungskontext abhängigen Hilfsfeldern zusammen. Mit Hilfe der LCP-Information lassen sich sowohl Top-Down als auch Bottom-Up Traversierungen eines Suffix-Baums durchführen, ohne dabei den Baum explizit aufzubauen und im Speicher halten zu müssen. Die LCP-Tabelle selbst lässt sich in linearer Zeit mit dem in Abschnitt 4.2 vorgestellten Algorithmus von Kasai et al. aus dem Suffix-Array und dem inversen Suffix-Array konstruieren. Dies ist die asymptotisch optimale Laufzeit und wie zu sehen sein wird, ist der versteckte konstante Faktor auch recht klein. Den Aufbau der LCP-Tabelle nach Vorliegen des Suffix-Arrays und seinem Inversen stellt dabei die grundsätzliche andere Möglichkeit dar die LCP-Information zu gewinnen, während sich der Hauptteil dieser Arbeit mit der LCP-Berechnung während der Konstruktion des Suffix-Arrays auseinandersetzt.

4.1 Enhanced Suffix-Arrays

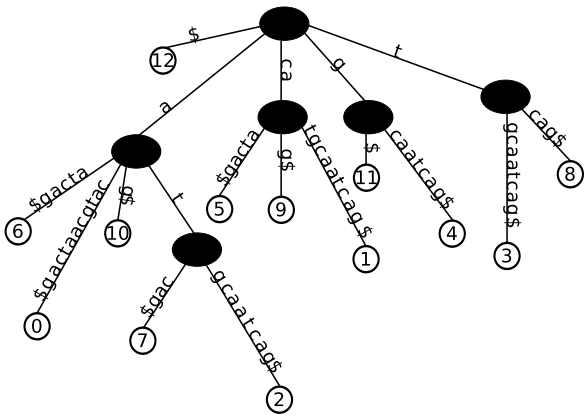
Mit Hilfe eines *Enhanced Suffix-Arrays* [2], kurz ESA, lassen sich Algorithmen für Suffix-Bäume darauf übertragen. Die Motivation für den Wechsel der Datenstruktur ist hauptsächlich in der hohen Speicheranforderung eines Suffix-Baums zu suchen. Selbst mit einer effizienten Implementierung [21] ist derzeit mit $12,5$ Bytes pro Eingabesymbol zu rechnen. Im Vergleich dazu benötigt ein Suffix-Array lediglich 4 Bytes pro Symbol. Hinzu kommt, dass durch die explizite Darstellung des Baums eine schwache Speicherlokalität gegeben ist, sodass die Effizienz von Systemen mit Cache-Architektur nicht genutzt wird. ESAs liegen dagegen in einer Größenordnung von etwa $6-9$ Bytes pro Symbol[2] und durch die Repräsentation der Informationen in Feldern wird eine blockorientierte Cache-Architektur in der Regel besser genutzt.
Bevor das ESA genauer vorgestellt wird, soll anhand eines Beispiels die Mächtigkeit und Einfachheit einer Suffix-Baum-Traversierung gezeigt werden. Die Problemstellung gibt 4.1.1 wieder.

Definition 4.1.1 *Seien $s, p \in \Sigma^+$ mit $m = |p| \leq |s| = n$ und sei ST_s der Suffix-Baum von s. String p nennt man das Muster und s den Text. Das Exact Pattern Matching Problem besteht darin, alle Vorkommen des Musters p in s zu finden, falls p in s vorhanden ist.*

Bezeichnet h die Häufigkeit des Vorkommens von p in s, so ist $O(m+h)$ die optimale Laufzeit für dieses Problem, d.h. die Suchzeit hängt nur vom Muster und dessen Häufigkeit in s, nicht aber von der Länge n des Textes ab. Durch eine Vorverarbeitung des Textes und seiner Darstellung als Suffix-Baum kann diese Unabhängigkeit von n für jeden beliebigen Text garantiert werden. Einleuchtend ist, dass sich diese Vorverarbeitung nur dann lohnt, wenn die Anzahl der Suchanfragen groß und der Text s eher von statischer Natur ist[1]. Um nun alle Vorkommen von p in s zu finden, führt man eine Top-Down-Traversierung von ST_s durch. Man beginnt bei der Wurzel und vergleicht nacheinander $p[0]$ mit dem ersten Symbol aller ausgehenden Kanten, bis entweder ein Match vorliegt oder keine Kantenbeschriftung mit $p[0]$ beginnt. Im letzten Fall kommt p in s nicht vor, da ST_s alle Suffixe von s beginnend bei der Wurzel repräsentiert. Ist dagegen eine Kante vorhanden, deren Beschriftung mit $p[0]$ beginnt, so folgt man dieser Kante weiter und vergleicht die nächsten Symbole von p mit dieser Beschriftung. Diesen Vorgang wiederholt man für alle angetroffenen Knoten. Liegt kein Mismatch vor, so endet p auf einer Kante und man folgt dieser bis zum nächsten Knoten. Mit einer Tiefensuche ab diesem Knoten gibt man alle vorgefundenen Blattbeschriftungen als Startpositionen von p in s aus. Für den ST_s aus Abb. 2.1 des Textes $s = acatgcaatcag\$$ und für das Muster $p = ca$ beginnt man bei der Wurzel und findet $p[0] = c$ als erstes Symbol der dritten Kante. Auf dieser stimmt auch das zweite Symbol $p[1] = a$ mit dem zweiten Symbol der Kantenbeschriftung überein und damit endet p an dieser Stelle. Nun geht man vor zum nächsten Knoten und führt eine Tiefensuche für alle drei vorhandenen Teilbäume durch. Man findet mit den Blattbeschriftungen 5, 9 und 1 alle Startpositionen von p in s.

Es sollte anhand des Exact Pattern Matching Problems gezeigt werden, wie einfach dieses klassische Problem der Informatik mit Hilfe eines Suffix-Baums gelöst werden kann. Viele weitere String Matching Probleme lassen sich mit dieser Datenstruktur elegant behandeln, wie z.B. in [13] gezeigt wird.
Das ESA bewahrt die Vorteile eines Suffix-Baums, indem es alle Traversierungen bei gleicher Zeitkomplexität ermöglicht, ohne dass der Baum explizit aufgebaut werden muss. Die LCP-Tabelle ist dabei die Basis, um die Suffix-Baum Algorithmen auf das ESA übertragen zu können. Vergleicht man den Suffix-Baum mit dem Suffix-Array, wie in Abb. 4.1, so erkennt man, dass lediglich die Blätter im Array repräsentiert sind. Die restliche Topologie des Suffix-Baums steht nicht zur Verfügung, sodass mit dem Suffix-Array allein eine Traversierung nicht, oder nur mit unvertretbarem Aufwand, möglich ist. Hier kommt die LCP-Tabelle ins Spiel, die diese fehlende Information platzsparend zum Suffix-Array ergänzt. Beispielsweise gehen vom Wurzelknoten $|\Sigma|$ viele Kanten aus, die einem 0-Eintrag in der LCP-Tabelle entsprechen. Geht man in Abb. 4.1 das Suffix-Array sequentiell von Position 1 bis Position 12 durch, so entsteht bei den Einträgen $1, 6, 9$ und 11 eine neue Kante aus der Wurzel[2]. Weiter lässt sich erkennen, dass an allen Teilbäumen des Wurzelknotens, genau die Suffixe an den Blättern repräsentiert sind, die zwischen zwei 0-Einträgen in der LCP-Tabelle vorhanden sind, wobei der Suffixindex des kleineren 0-Eintrags mit eingeschlossen ist.

[1] Genomdatenbanken sind ein Beispiel hierfür.
[2] Der Eintrag -1 bei Position 0 lässt ebenso eine Kante entstehen, ist aber nur per Definition von einem 0-Wert unterschieden.



i	$SA[i]$	$LCP[i]$	$s_{SA[i]}$
0	12	-1	\$
1	6	0	$aatcag\$$
2	0	1	$acatgcaatcag\$$
3	10	1	$ag\$$
4	7	1	$atcag\$$
5	2	2	$atgcaatcag\$$
6	5	0	$caatcag\$$
7	9	2	$cag\$$
8	1	2	$catgcaatcag\$$
9	11	0	$g\$$
10	4	1	$gcaatcag\$$
11	8	0	$tcag\$$
12	3	1	$tgcaatcag\$$

Abbildung 4.1: Suffix-Baum im Vergleich zum Suffix-Array für $s = acatgcaatcag\$$

Beispielsweise sind dies für den Teilbaum mit der ausgehenden Kante g die Suffixe 11 und 4, mit $LCP[9] = LCP[11] = 0$. Allgemein können bestimmte Intervalle im Suffix-Array zu Teilbäumen im Suffix-Baum zugeordnet werden. Definition 4.1.2 führt diese Vorüberlegungen mit der Einführung von *lcp*-Intervallen formal aus.

Definition 4.1.2 *Ein Intervall $[i..j]$, $0 \leq i < j \leq n$, ist ein $lcp-Intervall$ mit lcp-Wert l, wenn gilt:*

1. $LCP[i] < l$
2. $LCP[k] \geq l$ *für alle k mit $i+1 \leq k \leq j$*
3. $LCP[k] = l$ *für mindestens ein k mit $i+1 \leq k \leq j$*
4. $LCP[j+1] < l$

Positionen k mit $LCP[k] = l$ werden l-Indizes genannt.

Abkürzend wird für ein $lcp-Intervall$ mit Wert l auch l-Intervall oder $l-[i..j]$ verwendet. In Abb. 4.2 ist zum Suffix-Array und der LCP-Tabelle ein so genannter LCP-Intervallbaum dargestellt. Das $0-[0..12]$-Intervall wird eingeführt, um diesen Baum wurzeln zu können. Jedes l-Intervall enthält alle Suffixe, die ein gemeinsames Präfix der Länge l haben. Beispielsweise ist [1..5] ein 1-Intervall, denn:

1. $LCP[1] = 0 < 1$
2. $LCP[k] \geq 1$ für $2 \leq k \leq 5$
3. $LCP[3] = 1$
4. $LCP[6] = 0 < 1$

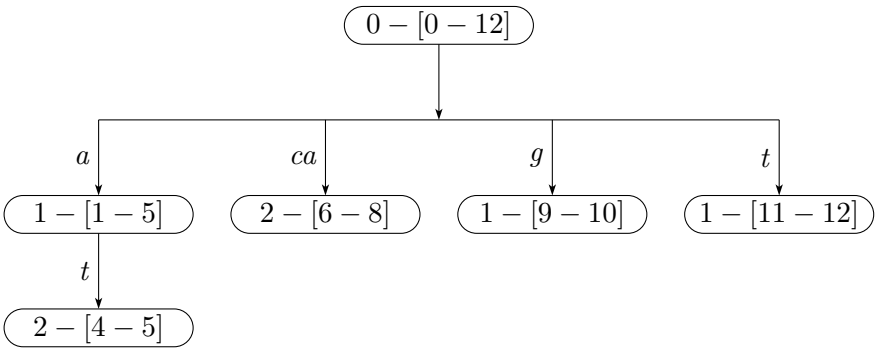


Abbildung 4.2: LCP-Intervallbaum für $s = acatgcaatcag\$$

Somit erfüllt dieses Intervall alle vier Bedingungen von Defintion 4.1.2. Man erkennt, dass die Intervalle ineinander verschachtelt sein können. So liegt z.B. $2-[4-5]$ in $1-[1-5]$ und über diese Struktur lässt sich mit 4.1.3 eine Vater-Kind Beziehung definieren.

Definition 4.1.3 *Seien $m-[l..r]$ und $q-[i..j]$ zwei lcp-Intervalle. Es werden definiert:*

1. *Gilt $i \leq l < r \leq j$ und $m > q$, so nennt man $m-[l..r]$ das eingebettete Intervall von $q-[i..j]$ und $q-[i..j]$ das umschließende Intervall von $m-[l..r]$.*
2. *Wird $m-[l..r]$ von keinem anderen eingebetteten Intervall in $q-[i..j]$ umschlossen, so nennt man $m-[l..r]$ ein Kind-Intervall von $q-[i..j]$.*
3. *Ein Intervall $[i..i]$ nennt man ein Singleton-Intervall.*

Diese formale Definition rechtfertigt die Darstellung des LCP-Intervallbaum aus Abb. 4.2, denn die Intervalle $1-[1-5], 2-[6-8], 1-[9-10]$ und $1-[11-12]$ sind Kindintervalle von $0-[0-12]$ und erfüllen die ersten beiden Bedingungen. Ebenso ist $2-[4-5]$ ein Kindintervall von $1-[1-5]$ und $1-[1-5]$ ist das umschließende Intervall von $2-[4-5]$. Kindintervalle werden untereinander durch die Positionen der l-Indizes getrennt. So ist z.B. $1-[9-10]$ von $1-[11-12]$ über die Position 11 mit $LCP[11] = 0$ getrennt. Setzt man die Kindintervalle in Beziehung zu einem Suffix-Baum, so entstehen an den l-Indizes neue ausgehende Kanten von einem Knoten. Somit entspricht ein LCP-Intervallbaum einem Suffix-Baum ohne Blätter. Die Blätter selbst werden durch die Singleton-Intervalle im Suffix-Array an allen Positionen repräsentiert. Damit liegen die in Tabelle 4.1 dargestellten Entsprechungen zwischen einem Suffix-Baum und einem LCP-Intervallbaum vor.

Im Allgemeinen wird der LCP-Intervallbaum nicht explizit konstruiert, sondern es wird eine Top-Down- oder ein Bottom-Up-Traversierung mit Hilfe von LCP simuliert. In Algorithmus 4.1 ist ein Verfahren für die Simulation einer Bottom-Up-Traversierung eines Suffixbaums mit Hilfe von SA und LCP angegeben. Dieser ist aus [2] entnommen und geht auf [18] zurück. Dabei werden alle LCP-Intervalle mit Hilfe eines Stapels berechnet, dessen Elemente Tripel der Form $\langle lcp, lb, rb\rangle$ sind. Dabei ist lcp der l-Wert des LCP-Intervalls, lb die linke Intervallgrenze und rb die rechte Intervallgrenze.

Suffix-Baum	**LCP-Intervallbaum**
Innere Knoten	*lcp*-Intervalle
Struktur	Einbettung/Umschließung
Kantenmarkierung	Gemeinsame Präfixe

Tabelle 4.1: Gegenüberstellung von Entsprechungen zwischen einem Suffix-Baum und einem LCP-Intervallbaum.

Die Stapeloperationen sind wie üblich definiert und die Vorgehensweise soll anhand der LCP-Tabelle aus Abb. 4.1 beschrieben werden.

Algorithmus 4.1 : Simulation einer Bottom-Up-Traversierung eines Suffixbaums mit Hilfe von SA und LCP

Eingabe : Suffix-Array SA und zugehörige LCP-Tabelle LCP
Ausgabe : Bottom-Up-Traversierung

```
1  push(⟨0, 0, ⊥⟩)
2  for i ∈ [1..n) do
3      lb ← i − 1
4      while LCP[i] < top.lcp do
5          top.rb ← i − 1
6          interval ← pop
7          report(interval)
8          lb ← interval.lb
9      if LCP[i] > top.lcp then
10         push(⟨LCP[i], lb, ⊥⟩)
```

Die Traversierung beginnt bei Position 0 und deren Inhalt zeigt ein 0-Intervall an, für das die rechte Grenze noch nicht bekannt ist. Der Stapel wird mit dem 0-Intervall initialisiert, wobei für dieses die linke Grenze 0 fix ist. Deswegen kommt $\langle 0, 0, \bot \rangle$ auf den Stapel. An Position 1 befindet sich erneut eine 0, die ignoriert wird. Es folgt der Wert 1 und damit beginnt ein 1-Intervall und man legt $\langle 1, 1, \bot \rangle$ auf den Stapel. Dei beiden folgenden 1-Einträge werden wieder ignoriert, da sie nicht größer sind als der oberste l-Index auf dem Stapel. An Position 5 ist der 2-Eintrag größer als der oberste l-Index, sodass dies wieder den Beginn eines neuen Intervalls anzeigt und somit $\langle 2, 4, \bot \rangle$ auf den Stapel gelegt wird. An Position 6 ist eine 0 und diese ist kleiner als der oberste l-Index auf dem Stapel, sodass nach 4.1.2 das Ende eines Intervalls angezeigt ist. Deshalb kann man die rechte Grenze 5 für dieses Intervall setzen und man nimmt Intervall $\langle 2, 4, 5 \rangle$ vom Stapel für eine weitere Verarbeitung. Weiter ist die 0 an Position 6 kleiner als der l-Index des neuen obersten Elements. Deshalb kann man nun auch dieses Intervall vervollständigen mit $\langle 1, 1, 5 \rangle$. Durch die 2 an Position 7 beginnt ein weiteres 2-Intervall mit $\langle 2, 6, \bot \rangle$ usw. Am Ende des Verfahrens wird $\langle 0, 0, 12 \rangle$ vom Stapel genommen, was der Wurzel entspricht. Es ist zu erkennen, dass sich durch einen sequentiellen Durchlauf durch LCP eine Bottom-Up-Traversierung eines Suffixbaums mit Hilfe von LCP-Intervallen durchführen lässt.

Es sollte die enorme Bedeutung der LCP-Tabelle für die Übertragung von Suffix-Baum Algorithmen auf Suffix-Arrays herausgestellt werden. Durch Anreicherung von weiteren Informationen im Suffix-Array lässt sich diese Portierung so realisieren, dass alle Algorithmen auf dem Suffix-Array in der gleichen Zeitkomplexität liegen wie die enstsprechenden Algorithmen bei den Suffix-Bäumen. Für eine detaillierte Darstellung sei auf [2] und [1] verwiesen.

4.2 LCP-Konstruktion in Linearzeit

Die LCP-Tabelle lässt sich in linearer Zeit, bezogen auf die Suffix-Anzahl eines Strings, aus dem Suffix-Array SA und dem inversen Suffix-Array SA^{-1} berechnen [18]. Um diesen Algorithmus beschreiben zu können, wird Satz 4.2.1 benötigt, der angibt, wie sich der LCP-Wert von zwei beliebigen Suffixen in SA berechnen lässt.

Satz 4.2.1 *Sei $s \in \Sigma^n$ und SA das Suffix-Array von s. Weiter seien $s_{SA[i]}$ und $s_{SA[j]}$ zwei Suffixe von s mit $i < j$. Dann gilt:*

$$lcp(s_{SA[i]}, s_{SA[j]}) = \min_{k \in [i,j-1]} \{lcp(s_{SA[k]}, s_{SA[k+1]})\}$$

Beweis: *Seien*

$$\begin{aligned} l &= lcp(s_{SA[i]}, s_{SA[j]}) \quad \textit{und} \\ m &= \min_{k \in [i,j-1]} \{lcp(s_{SA[k]}, s_{SA[k+1]})\}. \end{aligned}$$

Sei zunächst $m > l$ angenommen. Es gilt $s_{SA[q]}[0..m) = s_{SA[r]}[0..m)$ für alle $q, r \in [i..j]$ mit $q < r$. Damit gilt auch insbesondere $s_{SA[i]}[0..m) = s_{SA[j]}[0..m)$ und somit

$$lcp(s_{SA[i]}, s_{SA[j]}) \geq m > l$$

im Widerspruch zur Voraussetzung.
Sei nun $m < l$. Es existiert $p = \min_{q \in [i,j-1]} \{q \mid lcp(s_{SA[q]}, s_{SA[q+1]}) = m\}$ und somit gilt $s_{SA[p]}[m] \neq s_{SA[i]}[m]$. Da SA in Lexorder vorliegt, gilt somit auch für alle $p' \in [p+1..j]$, dass $s_{SA[p']}[m] \neq s_{SA[i]}[m]$ und somit $s_{SA[i]}[m] \neq s_{SA[j]}[m]$. Damit ist

$$lcp(s_{SA[i]}, s_{SA[j]}) \leq m < l$$

und man erhält den gewünschten Widerspruch. □

Zunächst folgt mit Korollar 4.2.2, dass die LCP-Werte zwischen adjazenten Suffixen in SA stets größer oder gleich einem LCP-Wert von zwei Suffixen sind, die diese in SA umgeben.

Korollar 4.2.2 *Aus Satz 4.2.1 folgt:*

$$lcp(s_{SA[y-1]}, s_{SA[y]}) \geq lcp(s_{SA[x]}, s_{SA[z]}) \quad mit \quad x < y \leq z$$

Beweis: *Setzt man $x = y - 1$ und $z = y$, so ist die Behauptung trivialerweise erfüllt. Für alle anderen Fällen gilt: wäre $lcp(s_{SA[y-1]}, s_{SA[y]}) < lcp(s_{SA[x]}, s_{SA[z]})$, dann ist $\min_{k \in [x,z-1]}\{lcp(s_{SA[k]}, s_{SA[k+1]})\} < lcp(s_{SA[x]}, s_{SA[z]})$ im Widerspruch zu 4.2.1.* □

Weiter lässt sich mit Lemma 4.2.3 feststellen, dass für adjazente Suffixe in SA, die einen LCP-Wert von mindestens 1 besitzen, sich die relative Ordnung dieser Suffixe in SA nicht ändert, wenn man das erste Symbol von beiden streicht.

Lemma 4.2.3 *Für zwei adjazente Suffixe $s_{SA[x-1]}$ und $s_{SA[x]}$ in SA gilt:*

$$lcp(s_{SA[x-1]}, s_{SA[x]}) \geq 1 \Rightarrow SA^{-1}[SA[x-1]+1] < SA^{-1}[SA[x]+1]$$

Beweis: *Sei $s_i = SA[x-1]$ und $s_j = SA[x]$. Da SA in Lexorder vorliegt, gilt $SA^{-1}[s_i] < SA^{-1}[s_j]$. Unter Annahme von $SA^{-1}[s_{i+1}] > SA^{-1}[s_{j+1}]$ muss $s_{i+1}[0..k] = s_{j+1}[0..k]$ und $s_{i+1}[k+1] > s_{j+1}[k+1]$ gelten für ein $k \geq 0$, wiederum weil SA in Lexorder vorliegt. Da nach Voraussetzung $s_i[0] = s_j[0]$ gilt, folgt $x - 1 = SA^{-1}[s_i] > SA^{-1}[s_j] = x$, was zum Widerspruch führt.* □

Es gilt mit Beobachtung 4.2.4 trivialerweise, dass der LCP-Wert zwischen den Suffixen $SA[x-1]+1$ und $SA[x]+1$ um genau eins kleiner ist als zwischen den Suffixen $SA[x-1]$ und $SA[x]$, wenn man das jeweils erste Symbol streicht.

Beobachtung 4.2.4

$$lcp(s_{SA[x-1]}, s_{SA[x]}) \geq 1 \Rightarrow lcp(s_{SA[x-1]+1}, s_{SA[x]+1}) = lcp(s_{SA[x-1]}, s_{SA[x]}) - 1$$

Die Berechnung der LCP-Tabelle läuft nun wie folgt ab. Unter der Annahme, dass der LCP-Wert von Suffix s_{i-1} und dem direkten Vorgängersuffix in SA bekannt ist, wird der LCP-Wert von Suffix s_i und seinem Vorgänger in SA berechnet. Seien $p = SA^{-1}[i-1]$ und $q = SA^{-1}[i]$ die Ränge der Suffixe s_{i-1} und s_i. Weiter seien $j - 1 = SA[p-1]$ und $k = SA[q-1]$ die Suffixe an den Positionen $p-1$ und $q-1$. Die Aufgabe besteht darin $LCP[q]$ zu berechnen, wenn $LCP[p]$ gegeben ist. Abb. 4.3 veranschaulicht diese Situation in SA^{-1} und SA. Lemma 4.2.5 setzt den gesuchten Wert $LCP[q] = lcp(s_k, s_i)$ in Beziehung zu $lcp(s_j, s_i)$, also zu den um ein Symbol längeren Suffixen s_{j-1} und s_{i-1}.

Lemma 4.2.5 (Kasai et al.) *Mit den vorstehenden Bezeichnungen gilt:*

$$lcp(s_{j-1}, s_{i-1}) \geq 1 \Rightarrow lcp(s_k, s_i) \geq lcp(s_j, s_i).$$

Beweis: *Mit 4.2.3 folgt aus $lcp(s_{j-1}, s_{i-1}) \geq 1$, dass $SA^{-1}[j] < SA^{-1}[i]$ gilt. Da Suffix k der direkte Vorgänger von Suffix i in SA ist, gilt $SA^{-1}[j] \leq SA^{-1}[k] = SA^{-1}[i] - 1$. Mit 4.2.2 folgt, da Suffixe k und i adjazent in SA sind, dass $lcp(s_k, s_i) \geq lcp(s_j, s_i)$.* □

Unter diesen Vorbetrachtungen lässt sich Satz 4.2.6 formulieren, der die Korrektheit des Algorithmus von Kasai feststellt.

Satz 4.2.6 (Kasai et al.)

$$\begin{aligned} LCP[p] &= lcp(s_{j-1}, s_{i-1}) \geq 1 \Rightarrow \\ LCP[q] &= lcp(s_k, s_i) \geq LCP[p] - 1. \end{aligned}$$

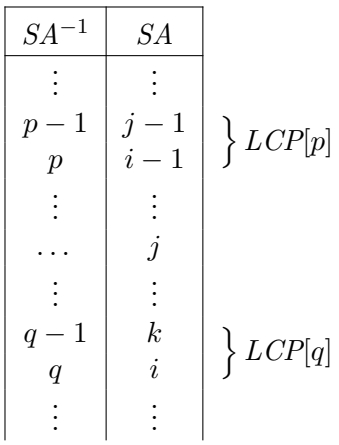


Abbildung 4.3: Situation in SA und SA^{-1} für das Berechnungsschema des Kasai-Algorithmus für $LCP[q]$, wenn $LCP[p]$ bekannt ist.

Beweis: *Wegen 4.2.5 gilt* $lcp(s_k, s_i) \geq lcp(s_j, s_i)$ *und aus 4.2.4 folgt* $lcp(s_{j-1}, s_{i-1}) - 1 = lcp(s_j, s_i)$. *Damit gilt* $LCP[q] \geq LCP[p] - 1$. □

Die Kernaussage von Satz 4.2.6 ist, dass man nicht alle Symbole zwischen adjazenten Suffixen in SA vergleichen muss, um den LCP-Wert zu erhalten. Um $lcp(s_k, s_i)$ zu berechnen, lässt man die ersten $lcp(s_{j-1}, s_{i-1}) - 1$ Symbole aus und beginnt erst dann den restlichen Vergleich. Dabei geht man sukzessive von Suffix s_0 bis Suffix s_{n-1} vor und in Algorithmus 4.2 ist das gesamte Verfahren angegeben.

Algorithmus 4.2 : Algorithmus von Kasai et al. für die Berechnung der LCP-Tabelle in Linearzeit

Eingabe : Eingabewort s, SA, SA^{-1}
Ausgabe : LCP-Tabelle

```
1  h ← 0
2  for i ∈ [0..n) do
3      j ← SA^{-1}[i]
4      if j ≥ 1 then
5          k ← SA[j − 1]
6          while s_{i+h}[0] = s_{k+h}[0] do
7              Inkrementiere h
8          LCP[j] ← h
9          if h > 0 then
10             Dekrementiere h
```

Die Laufzeit hängt von der while-Schleife in den Zeilen 6 und 7 ab. Die Variable h steht für die Anzahl an Symbolen, die man für die Berechnung eines LCP-Wertes auslassen darf. Da alle Suffixe eine unterschiedliche Länge haben und alle mit dem Symbol \$ enden, gilt stets $h < n$.

In Zeile 10 wird h maximal n mal dekrementiert, sodass h maximal $2n$ mal inkrementiert werden kann. Damit ergibt sich eine Zeitkomplexität von $O(n)$.

Der Algorithmus benötigt neben der LCP-Tabelle noch die Eingabe und die beiden Felder SA und SA^{-1}, womit sich ein Platzbedarf von $13n$-Bytes ergibt. Mit Hilfe von [25] lässt sich dies auf $9n$-Bytes verbessern, wobei auf SA^{-1} verzichtet werden kann. Es sind in dieser Arbeit auch weitere Platzreduktionen angegeben, wenn z.B. am Ende des Algorithmus SA nicht mehr benötigt wird.

Die nächsten drei Kapitel beschäftigen sich mit der LCP-Berechnung während das Suffix-Array konstruiert wird. Es werden drei typische Vertreter von SAKAs vorgestellt, wobei deren Klassifizierung auf [28] zurückgeht.

5 Prefix-Doubling

In diesem Kapitel werden die beiden SAKAs von Manber und Myers [23] sowie Sadakane und Larsson[22] vorgestellt, die beide eine worst-case Laufzeit von $O(n \log n)$ besitzen und auf einer Idee von Karp [17] basieren. Manber und Myers beschreiben in ihrer Arbeit auch ein Verfahren, um die LCP-Tabelle parallel zur Konstruktion des Suffix-Arrays zu berechnen. Die Werte der LCP-Tabelle wurden zu dieser Zeit lediglich als Hilfsmittel verwendet, um eine binäre Suche im Suffix-Array zu ermöglichen, worauf hier nicht näher eingegangen werden soll. Mit der Einführung der Enhanced Suffix-Arrays, die in Kapitel 4 beschrieben sind, hat man ein asymptotisch optimales Verfahren zur Hand, das diese Binärsuche überflüssig macht. Die Beschreibung der Autoren, wie man die LCP-Werte während des Algorithmenablaufs berechnet, ist auch für Larsson und Sadakane relevant. Diese stellten 1999 einen SAKA vor, der das gleiche Konzept wie Manber und Myers verwendet, aber durch ein geschickteres Vorgehen bessere Laufzeiten erzielt. Dabei benötigt dieser Algorithmus den gleichen Speicherbedarf wie der von Manber und Myers. Deshalb soll der Fokus in diesem Kapitel auf diesem Algorithmus beruhen und eine detailliertere Darstellung erfahren. Für die LCP-Berechnung bei beiden Verfahren werden Range Minimum Queries benötigt und zwei mögliche Realisierungen vorgestellt.

5.1 Algorithmus von Manber und Myers

Manber und Myers führten 1990 mit dem Suffix-Array eine Datenstruktur zur Repräsentation aller Suffixe eines Strings ein. Damit hat man eine alternative Darstellung gegenüber dem Suffix-Baum zur Hand, die wesentlich weniger Speicherplatz erfordert. Zudem stellten sie auch den ersten direkten Konstruktionsalgorithmus für ein Suffix-Array vor, der nun beschrieben werden soll.

Im Weiteren sei $s = s[0..n-1]$ das Eingabewort der Länge $n \geq 1$ über einem Alphabet Σ mit $\alpha = |\Sigma|$. Dieses Alphabet sei indiziert, d.h.:

- Es liegt eine totale Ordnung der Symbole $\lambda_j, j \in [1, \alpha]$ aus Σ vor:
  $\lambda_1 < \lambda_2 < \cdots < \lambda_\alpha$
- Es lässt sich ein Feld $S[\lambda_1..\lambda_\alpha]$ definieren, sodass der Zugriff auf $S[\lambda_j]$ für alle $j \in [1, \alpha]$ in konstanter Zeit erfolgen kann.

Das Feld $SA[0..n)$, das nach Ablauf des Algorithmus das Suffix-Array von s beinhaltet, wird zunächst mit allen Suffixindizes absteigend nach ihrer Größe initialisiert, also $SA[i] = n - i - 1$ für $i \in [0..n)$. Der Algorithmus arbeitet nun in Phasen, in denen jeweils ein partiell geordnetes Suffix-Array bezüglich einer festen Präfixlänge H entsteht, siehe dazu auch Definition 2.4.3.

Phase 0 beginnt mit einem Bucketsort nach dem ersten Symbol aller Suffixe, d.h. SA ist danach in 1-Ordnung und wird mit SA_1 gekennzeichnet. In Abb. 5.1 ist für das Beispielwort $s = acatgcaatcag\$$ das Ergebnis dieser Sortierung zu sehen.

$$SA_1 \;\Big|\; \underbrace{12}_{\$} \;\Big|\; \underbrace{0,2,6,7,10}_{a} \;\Big|\; \underbrace{1,5,9}_{c} \;\Big|\; \underbrace{4,11}_{g} \;\Big|\; \underbrace{3,8}_{t} \;\Big|$$

Abbildung 5.1: Bucketsort nach dem ersten Symbol für $s = acatgcaatcag\$$.

Man erhält eine Menge von Buckets, in denen sich nur Suffixe befinden, die mit dem gleichen Symbol beginnen. Um diese Sortierung zu verfeinern, könnte man rekursiv erneut Bucketsort mit allen zweiten Symbolen auf alle Buckets anwenden, die mehr als ein Suffix enthalten. Mit Hilfe der nun folgenden Prefix-Doubling Technik lässt sich die weitere Sortierung effizienter gestalten, ohne Bucketsort erneut aufzurufen.
Seien s_i, s_j mit $i \neq j$ zwei Suffixe aus einem beliebigen Bucket. Dann gilt $s_i[0] = s_j[0]$, d.h. die beiden Suffixe sind in SA_1 nach ihrem ersten Symbol sortiert. Um nun nach dem zweiten Symbol zu sortieren, muss man den Vergleich von $s_i[1]$ und $s_j[1]$ durchführen. Das Ergebnis dieses Vergleiches liegt nun aber implizit schon vor, denn die beiden Suffixe s_{i+1} und s_{j+1} liegen auch nach ihren jeweiligen ersten Symbolen $s_{i+1}[0]$ und $s_{j+1}[0]$ sortiert in SA_1 vor. Um diesen Vergleich von $s_i[1]$ und $s_j[1]$ in SA_1 durchzuführen, durchläuft man SA_1, das nach dem ersten Symbol sortiert ist, sequentiell von links nach rechts. Ist $SA_1[i] = j \in [1..n-1]$, so verschiebt man den Suffixindex $j-1$ an den aktuellen Anfang des zugehörigen Buckets und inkrementiert diesen anschließend. Ein neues Bucket entsteht mit dieser Vorgehensweise immer dann, wenn der Suffixindex $j-1$ der erste ist, der aufgrund von Suffixindex j, der aus dem Bucket an aktueller Laufposition i stammt, verschoben wurde. Diese Anfangsposition wird zu einem neuen Bucket, dessen Länge sich danach richtet, wieviel Suffixe aus dem gerade durchlaufenden Bucket eine Verschiebung in dieses neue Bucket verursachen. Als Ergebnis dieses SA_1-Durchlaufs erhält man SA_2, das nach den jeweils ersten beiden Symbolen aller Suffixe sortiert und in Abb. 5.2 dargestellt ist.

$$SA_2 \;\Big|\; \underbrace{12}_{\$} \;\Big|\; \underbrace{6}_{aa} \;\Big|\; \underbrace{0}_{ac} \;\Big|\; \underbrace{10}_{ag} \;\Big|\; \underbrace{2,7}_{at} \;\Big|\; \underbrace{1,5,9}_{ca} \;\Big|\; \underbrace{11}_{g\$} \;\Big|\; \underbrace{4}_{gc} \;\Big|\; \underbrace{8}_{tc} \;\Big|\; \underbrace{3}_{tg} \;\Big|$$

Abbildung 5.2: Sortierung von SA_2 für $s = acatgcaatcag\$$

Für die weitere Sortierung müssen nun alle Suffixe aus Buckets mit mehr als einem Element nach allen dritten Symbolen sortiert werden. Die Effizienz des gesamten Algorithmus begründet sich dadurch, dass man diesen Vergleich nicht explizit durchführen muss, sondern gleich nach dem dritten und vierten Symbol sortieren kann. Seien wieder s_i und s_j zwei Suffixe aus einem beliebigen Bucket von SA_2. Dann gilt $s_i[0..1] = s_j[0..1]$, d.h. die beiden Suffixe stimmen in ihren ersten beiden Symbolen überein. Um nun nach dem dritten und vierten Symbol zu sortieren, muss man den Vergleich von $s_i[2..3]$ und $s_j[2..3]$ durchführen. Wiederum liegt das Ergebnis dieses Vergleichs implizit in SA_2 vor, denn die beiden Suffixe s_{i+2} und s_{j+2} liegen auch nach ihren jeweiligen beiden ersten Symbolen $s_{i+2}[0..1]$ und $s_{j+2}[0..1]$ sortiert in SA_1 vor.

Analog zur Phase 1 durchläuft man SA_2, das nach den ersten beiden Symbolen sortiert ist, sequentiell von links nach rechts und wenn $SA_1[i] = j \in [1..n-1]$ gilt, so schiebt man den Suffixindex $j-2$ an den aktuellen Anfang seines Buckets und inkrementiert diesen anschließend. Als Ergebnis erhält man SA_4, das nun nach den jeweils vier ersten Symbolen aller Suffixe sortiert vorliegt und in Abb. 5.3 dargestellt ist.

SA_4	12	6	0	10	7	2	5	9	1	11	4	8	3
	$	aatc	acat	ag$	atca	atgc	caat	cag$	catg	g$	gcaa	tcag	tgca

Abbildung 5.3: Sortierung von SA_4 für $s = acatgcaatcag\$$

Für das laufende Beispiel reicht eine Präfixbetrachtung von vier Symbolen aus, um die Sortierung zu beenden, denn nun liegen genau n Buckets vor, die genau einen Suffixindex beinhalten.
Allgemein würde man in einem nächsten Schritt die ersten acht Symbole aller Suffixe zur Sortierung verwenden. Damit verdoppelt sich in jeder Phase die Anzahl der Symbole bzw. die Präfixlänge, nach denen die Suffixe sortiert sind, wodurch der Name dieses Verfahrens, Prefix-Doubling, begründet wird. Nach höchstens $\log n$ Phasen, bei denen man jeweils SA_H mit $H = 2^k$ in Phase $k+1$ von links nach rechts durchläuft, bricht das Verfahren mit der vollständigen Sortierung aller Suffixe ab. Dies ergibt eine Gesamtlaufzeit von $O(n \log n)$.

Eine Implementierung dieses Algorithmus benötigt zusätzliche Hilfsfelder. Zunächst wird ein weiteres Suffix-Array $SA'[0..n)$ benötigt, um die Sortierung der Buckets zu realisieren. Am Ende jeder Phase wird dann SA' auf SA umkopiert. Weiter wird ein Bitfeld $NBH[0..n)$ angelegt, in dem ein Eintrag $NBH[i] = 1$ anzeigt, dass an Position i in SA ein neues Bucket beginnt. Dieses Feld wird auch in Abschnitt 5.6 verwendet, um die LCP-Tabelle berechnen zu können. Für die Umsetzung der Suffix-Verschiebungen an einen Bucketanfang wird das inverse Suffix-Array SA^{-1} benötigt. Ist $SA_H[i] = j$, wobei $H = 2^k$ für Phase k, so braucht man die Position von Suffix $j - H$, um den richtigen Bucketanfang zu erhalten, also $SA_H^{-1}[j-H]$. Zu Beginn jeder Phase wird mit $SA_H^{-1}[SA_H[i]] = i$ das inverse Suffix-Array berechnet. Schließlich wird in einem Feld $L[0..n)$ die Position eines Bucketkopfs gespeichert, d.h. $L[i] = i$, falls $NBH[i] = 1$, und $L[i] = \max\{j \mid j < i \wedge NBH[j] = 1\}$ sonst. Die Organisation von Bucketköpfen ist in Abb. 5.4 dargestellt.
Es ist ein Bucket mit fünf Suffixen vorhanden, das an Position p beginnt. Um potentiell all diese Suffixe an den aktuellen Anfang p schieben zu können, müssen alle Indizes dieses Buckets einen Verweis auf diesen Anfang besitzen. Deshalb ist $L_H[i] = p$ für $i \in [p..p+4]$ und das nächste Bucket beginnt an Position $p+5$. Ist z.B. in einer Phase $SA_H[i] = j$ mit $SA_H^{-1}[j-H] = p+2$, so wird über $L_H[p+2] = p$ der aktuelle Bucketkopf angesprochen und Suffix $j - H$ an diesen Anfang verschoben. Damit das nächste Suffix nicht an die gleiche Stelle verschoben wird, inkrementiert man den Eintrag $L_H[p]$ und verweist somit an die nächste Verschiebeposition im Bucket. Schließlich zeigt das Feld NBH an den Positionen p und $p+5$ den Beginn von Buckets an. Der gesamte Algorithmus von Manber und Myers ist in 5.1 als Pseudocode angegeben.

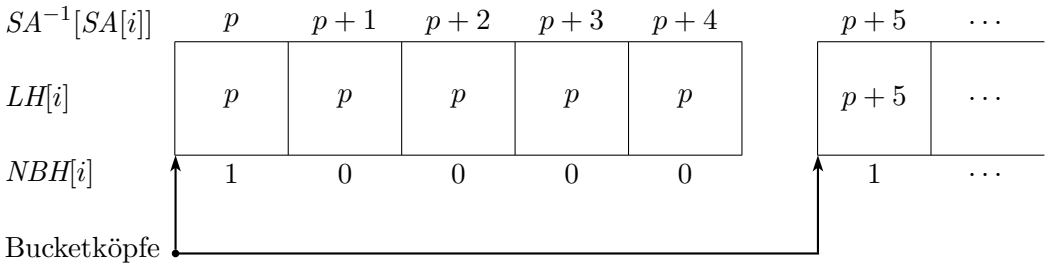


Abbildung 5.4: Organisation und Nummerierung von Bucketköpfen und Suffixen in einem Bucket bei Manber und Myers.

Neben dem Eingabewort und dem Suffix-Array ist der zusätzliche Speicherbedarf somit $12n$-Bytes für die Felder SA', SA^{-1} und L. Manber und Myers zeigten, dass sich das Bitfeld NBH über das Vorzeichenbit in den anderen Feldern unterbringen lässt und eine weitere Reduktion auf $8n$-Bytes möglich ist.

Algorithmus 5.1 : Algorithmus von Manber und Myers

Eingabe : Eingabewort s
Ausgabe : SA für s

```
1  Führe einen Bucketsort auf SA nach dem ersten Symbol durch
2  NBH[0] = TRUE
3  ∀i ∈ [0..n − 1] : B[i] ← (s[SA[i]] ≠ s[SA[i + 1]])
4  for p ∈ [0.. log p] do
5      for i ∈ [0..n − 1] do
6          SA^{-1}[SA[i]] ← i
7          SA'[i] ← 0
8          if NBH[i] = TRUE then
9              L[i] ← i
10         else
11             L[i] ← L[i − 1]
12     for i ∈ [0..n − 1] do
13         if SA[i] − 2^p > 0 then
14             j ← SA^{-1}[SA[i] − 2^p]
15             if L[j] = j then
16                 p ← L[j]
17             else
18                 p ← L[L[j]]
19             if L[i] = i then
20                 b ← i
21             else
22                 b ← L[i]
23             b' ← SA'[p]
24             SA'[p] ← SA[j]
25             if NBH[b + 1] = FALSE then
26                 SA'[p + 1] ← b
27             NBH[p] ← (NBH[p] ∨ b ≠ b')
28             if L[j] ≠ j then
29                 j ← L[j]
30             Inkrementiere L[j]
31     for i ∈ [0..n − 1] do
32         if SA'[i] > 0 then
33             SA[i] ← SA'[i]
```

5.2 Algorithmus von Larsson und Sadakane

Ebenso wie bei Manber und Myers beruht der SAKA von Larsson und Sadakane [22] auf der Idee des Prefix-Doubling. Diese wird in 5.2.1 auf einer formaleren Ebene wiedergegeben, wobei $H = 2^k$ die aktuelle Präfixlänge für Phase k darstellt.

Beobachtung 5.2.1 *Sei $s \in \Sigma^+$ und seien SA_H und SA_H^{-1} für ein $H \geq 1$ vorberechnet. Mit Hilfe der beiden Ränge*

$$(SA_H^{-1}[i], SA_H^{-1}[i + H])$$

lässt sich SA_{2H} für alle Suffixe von s berechnen, wobei $SA_H^{-1}[i]$ der Primär- und $SA_H^{-1}[i+H]$ der Sekundärschlüssel für das verwendete Sortierverfahren ist.

Der Algorithmus von Larsson und Sadakane arbeitet wie folgt. Zunächst werden alle Suffixe wieder mit einem Bucketsort nach ihrem ersten Symbol sortiert[1], indem man $S[s_i[0]]$ mit $i \in [0..n)$ als Sortierschlüssel verwendet. Als Ergebnis erhält man SA_1 und SA_1^{-1}. Danach arbeitet der Algorithmus auch in Phasen, um die Gesamtsortierung zu erhalten. Im Gegensatz zu Manber und Myers wird in Phase $k+1$ aber nicht das gesamte Feld SA_H durchlaufen, um SA_{2H} zu erhalten, sondern es werden explizit alle Suffixe eines Buckets, das mehr als ein Element beinhaltet, sortiert. Bevor darauf näher eingegangen wird, werden in 5.2.2 Begriffe eingeführt, die die Situation in SA_H nach Phase k charakterisieren.

Definition 5.2.2 *Sei das Suffix-Array SA_H nach Phase k gegeben, d.h. SA_H enthält eine partielle Sortierung aller Suffixe nach den ersten $H = 2^k$ Symbolen. Es werden definiert:*

- *Eine Gruppe ist eine maximale Sequenz von benachbarten Suffixen aus SA_H mit der Eigenschaft, dass ihre Präfixe der Länge H übereinstimmen.*
- *Besteht eine Gruppe aus mindestens zwei Suffixen, so nennt man dies eine unsortierte Gruppe.*
- *Eine Gruppe, bestehend aus genau einem Suffix, nennt man eine sortierte Gruppe.*
- *Eine verkettete, sortierte Gruppe ist eine maximale Sequenz von benachbarten sortierten Gruppen in SA_H.*

Um Beobachtung 5.2.1 zu nutzen, werden alle Gruppen nummeriert, um damit Sortierschlüssel zu erhalten. Einer Gruppe $SA_H[f..g]$ wird dabei immer die letzte Position g zugeordnet und diese wird für alle Suffixe $j \in [f..g]$ während der Sortierung abgelegt. Es bedeutet $SA_H^{-1}[j] = g$ somit, dass Suffix s_j aktuell zur Gruppe g gehört. Um nach Phase k alle entstandenen unsortierten Gruppen zu identifizieren, durchläuft man SA_H in Phase $k + 1$ von links nach rechts. Dabei merkt man sich alle Positionen von sortierten und verketteten, sortierten Gruppen, damit in den folgenden Phasen diese Positionen übersprungen werden können.

[1] Es kann auch die Situation eintreten, dass dies nicht ohne eine vorhergehende Kompaktierung des Eingabealphabets möglich ist. Im Text wird vereinfachend angenommen, dass dies schon geschehen ist. Zu den Details sei auf [22] verwiesen.

Dies ist möglich, da alle Suffixe von verketteten, sortierten Gruppen und sortierten Gruppen sich von allen anderen Suffixen in den ersten H Symbolen unterscheiden, da SA_H vor Beginn von Phase $k+1$ in H-Ordnung vorliegt. Somit behalten die Suffixe aus verketteten, sortierten Gruppen und sortierten Gruppen ihre endgültige Position in SA für alle folgenden Phasen bei. Um das Überspringen von Intervallbereichen in SA_H zu realisieren, wird in 5.2.3 ein Feld eingeführt, in dem die Längen von sortierten und verketteten, sortierten Gruppen gespeichert werden.

Definition 5.2.3 *Sei $LEN[0..n)$ ein Feld der Länge n mit Werten aus $[0..n)$ und $[f..g]$ ein Intervallereich von SA. LEN wird definiert durch:*

$$LEN[i] := \begin{cases} g-f+1 & \textit{falls} \quad i = f \textit{ und} \\ & [f..g] \textit{ eine unsortierte Gruppe ist} \\ -(g-f+1) & \textit{falls} \quad i = f \textit{ und} \\ & [f..g] \textit{ eine verkettete, sortierte Gruppe ist} \\ \bot & \textit{sonst.} \end{cases}$$

In LEN werden also die Längen von unsortierten und verketteten, sortierten Gruppen gespeichert, und zwar an der Startposition jeder Gruppe. Dabei werden die beiden Typen von Gruppen über das Vorzeichen unterschieden. Die unsortierten Gruppen werden mit TSQS sortiert, indem für jedes Suffix i aus einer unsortierten Gruppe die Gruppennummer $SA_H^{-1}[i+H]$ von Suffix $i+H$ als Schlüssel verwendet wird. Nach 5.2.1 erzielt man damit die $2H$-Ordnung von SA_H. Es entstehen neue Gruppen zwischen allen Suffixen, die unterschiedliche Schlüssel besitzen. Am Ende jeder Phase werden die Gruppennummern und die Längen der neu entstandenen Gruppen in SA_H^{-1} und LEN aktualisiert und es liegen die Felder SA_{2H} und SA_{2H}^{-1} vor. Adjazente sortierte Gruppen werden erkannt und zu einer sortierten Gruppe verschmolzen, damit diese in allen nachfolgenden Phasen überpsrungen werden kann. In Algorithmus 5.2 ist der beschriebene Ablauf auf einer sprachlichen Ebene zusammengefasst.

Für das Beispielwort $s = acatgcaatcag\$$ ergibt sich nach den ersten drei Schritten die Situation aus Abb. 5.5. Der Eintrag $LEN[0] = -1$ zeigt an, dass das Suffix 12 in einer sortierten Gruppe mit der Gruppennummer $SA^{-1}[12] = 0$ platziert wurde und somit in seiner endgültigen Position ist. In allen weiteren Phasen kann man diese Position überspringen. Es liegen $|\Sigma| = 4$ unsortierte Gruppen vor, z.B. $SA[1-5]$ für das Symbol a, und allen Suffixen aus dieser Gruppe wurde die Gruppennummer 5 zugewiesen. In der letzten Zeile sind die Schlüssel angegeben, die in der nächsten Phase zur Sortierung der unsortierten Gruppen verwendet werden. Beispielsweise ist für Suffix 5 aus Gruppe 8 der Schlüssel 5 zugewiesen, da sich Suffix 6 in der Gruppe 5 befindet bei einer Präfixvorausschau von $H = 1$. Nun werden alle unsortierten Gruppen mit TSQS sortiert und die Werte für alle beteiligten Felder aktualisiert. Der analoge Ablauf für alle weiteren Phasen ist in Abb. 5.6 zusammengefasst.
Nach Phase 1 existieren noch die beiden unsortierten Gruppen 5 und 8. Man gelangt in konstanter Zeit zur Startposition von Gruppe 5, indem man den eingetragenen Wert $LEN[0] = -4$ benutzt, um 4 Positionen in SA_1 zu überspringen.

Algorithmus 5.2 : Algorithmus von Larsson und Sadakane

Eingabe : Eingabewort s
Ausgabe : SA für s

1. Schreibe alle Suffixindizes fortlaufend nach absteigender Größe in SA. Sortiere mit Bucketsort SA nach dem ersten Symbol mit Hilfe des Schlüssels $S[\lambda_j]$ für alle $j \in [1, \alpha]$.
2. Für alle $i \in [0..n]$ schreibe die Gruppennummer von Suffix i nach $SA^{-1}[i]$, d.h. dies ist die letzte Position von SA, die ein Suffix mit identischem ersten Symbol enthält.
3. Für jede unsortierte Gruppe $SA[f..g]$ setze $LEN[f] = f - g + 1$ und für jede verkettete, sortierte Gruppe $SA[f..g]$ setze $LEN[f] = -(f - g + 1)$.
4. Sortiere mit TSQS alle unsortierten Gruppen in SA, wobei für jedes Suffix i aus einer unsortierten Gruppe $SA^{-1}[i + H]$ als Schlüssel verwendet wird.
5. Markiere alle Positionen in unsortierten Gruppen in SA, an denen neue Gruppen aufgrund von unterschiedlichen Schlüsseln enstanden sind.
6. Verdopple die Präfixvorausschau H und erzeuge neue Gruppen an den markierten Positionen. Anschließend trage in SA^{-1} die neuen Gruppennummern und in LEN die neuen Längen ein.
7. Falls SA aus genau einer verketteten, sortieren Gruppe besteht, so ist die Konstruktion von SA abgeschlossen. Falls nicht, gehe zu Schritt vier.

In dieser Gruppe werden die beiden Suffixe 2 und 7 mit Hilfe ihrer Schlüssel $SA_1^{-1}[2+2] = 4$ und $SA_1^{-1}[7+2] = 8$ sortiert. Der analoge Vorgang wird für Gruppe 8 durchgeführt und am Ende dieser Phase existiert genau eine verkettete, sortierte Gruppe, sodass der Algorithmus damit beendet ist. Angezeigt wird dies durch $LEN[0] = -12$, denn es liegen demnach genau $n = 12$ sortierte Gruppen vor. In ihrer Arbeit zeigen Larsson und Sadakane, dass mit dem Einsatz von TSQS als Hauptsortierverfahren eine worst-case Laufzeit von $O(n \log n)$ garantiert werden kann.

Es sollen im Weiteren zwei Optimierungen[2] betrachtet werden, die einen Einfluss auf die parallele Berechnung der LCP-Tabelle haben.
Zunächst lässt sich das Feld LEN komplett eliminieren. TSQS benötigt als Eingabeparameter die Anzahl der zu sortierenden Suffixe einer unsortierten Gruppe. Überspringt man in einer Phase einen schon sortierten Bereich in SA, gelangt man zur Anfangsposition einer unsortierten Gruppe. Zusammen mit der Gruppennummer des Suffixes an dieser Position lässt sich die Anzahl der zu sortierenden Suffixe dieser unsortieren Gruppe in konstanter Zeit berechnen. Es bleiben noch die negativen Einträge für sortierte Gruppen zu behandeln. Gehört ein Suffix einer sortierten Gruppe an, so wird die Position dieses Suffix in SA nicht mehr benötigt. Lediglich die Gruppennummer wird eventuell als Sortierschlüssel verlangt und diese steht in SA^{-1} zur Verfügung. Damit lassen sich die Speicherpositionen für alle Suffixe von sortierten Gruppen für andere Zwecke nutzen, ohne die Korrektheit des Algorithmus zu gefährden. Man trägt deshalb zur Kennzeichnung von verketteten, sortierten Gruppen ihre negativen Längen an den Startpositionen in SA ein.

[2] Von den Autoren selbst vorgeschlagen.

	i	0	1	2	3	4	5	6	7	8	9	10	11	12
$H=1$	$s_i[0]$	a	c	a	t	g	c	a	a	t	c	a	g	\$
$SA_1[i]$		12	0	2	6	7	10	1	5	9	4	11	3	8
$SA_1^{-1}[SA_1[i]]$		0	|5	5	5	5	5|	|8	8	8|	|10	10|	|12	12|
$LEN[i]$		-1	5					3			2		2	
$SA_1^{-1}[SA_1[i]+1]$			|8	12	5	12	10|	|5	5	5|	|8	0|	|10	8|

Abbildung 5.5: Situation nach dem initialen Bucketsort von Larsson und Sadakane für $s = acatgcaatcag\$$

	i	0	1	2	3	4	5	6	7	8	9	10	11	12
$H=2$	$s_i[0]$	a	c	a	t	g	c	a	a	t	c	a	g	\$
$SA_2[i]$		12	6	0	10	2	7	1	5	9	11	4	8	3
$SA_2^{-1}[SA_2[i]]$		0	1	2	3	|5	5|	|8	8	8|	9	10	11	12
$LEN[i]$		-4				2		3			-4			
$SA_2^{-1}[SA_2[i]+2]$						|10	8|	|12	5	9|				
$H=4$														
$SA_4[i]$		12	6	0	10	7	2	5	9	1	11	4	8	3
$SA_4^{-1}[SA_4[i]]$		0	1	2	3	4	5	6	7	8	9	10	11	12
$LEN[i]$		-12												
$SA_4^{-1}[SA_4[i]+4]$														

Abbildung 5.6: Situation nach den beiden Phasen 1 und 2 von Larsson und Sadakane.

Damit zeigt ein negativer Wert von $SA[i]$ eine sortierte Gruppe $SA[i..i - SA[i] + 1]$ und ein positiver Wert an dieser Stelle eine unsortierte Gruppe $SA[i..SA^{-1}[SA[i]]]$ an, wobei in diesem Intervall die Suffixindizes noch vorhanden sind. Durch dieses Überschreiben der Startpositionen mit Längen enthält das Suffix-Array SA am Ende des Algorithmus nicht mehr die gewünschte Information. Diese lässt sich durch das inverse Suffix-Array, das die inverse Permutation des Suffix-Arrays darstellt, in einem Durchlauf rekonstruieren. Man setzt $SA[SA^{-1}[i]] = i$ für alle $i \in [0..n-1]$. Zusammen mit noch weiteren technischen Verbesserungen kommt dieser SAKA damit auf einen Speicheraufwand von $8n$-Bytes.

Die zweite Optimierung setzt auf der Struktur von TSQS auf, um sofort die neu entstandenen Gruppennummern für alle weiteren Sortiervorgänge zu nutzen. Zur Beschreibung sei eine unsortierte Gruppe mit den drei Suffixen s_i, s_j, s_k und Gruppennummer g gegeben. Die Sortierschlüssel für diese Suffixe seien $KEY(s_i) = u$ mit $u < g$ und $KEY(s_j) = KEY(s_k) = g$, wobei $KEY(i) := SA_H^{-1}[i + H]$. Nach der in Abschnitt 3.3 beschriebenen Partitionierungsphase von TSQS wird zuerst der linke Bereich rekursiv sortiert. Der mittlere Bereich mit den Elementen gleich dem Pivot bekommt eine Gruppennummer zugewiesen und anschließend wird der rechte Bereich rekursiv sortiert.

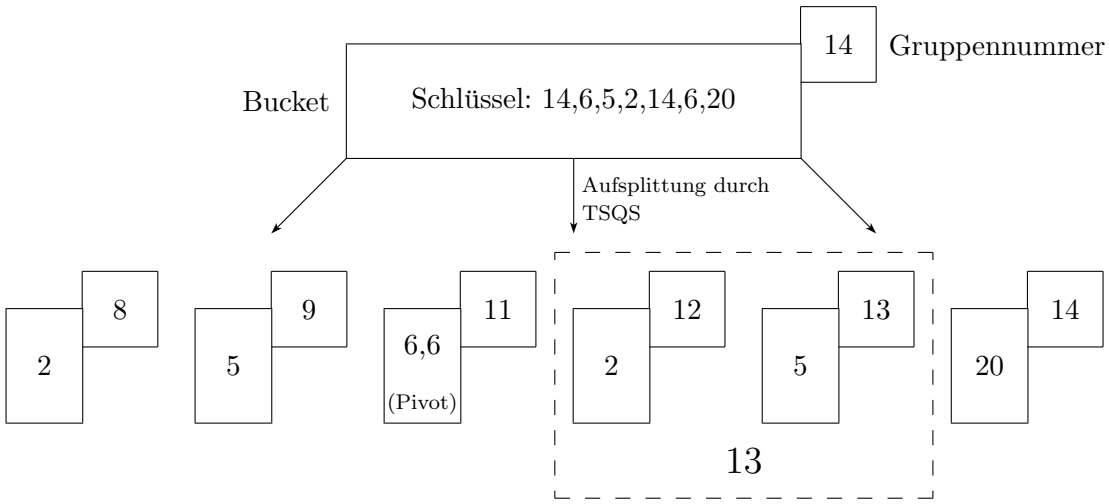


Abbildung 5.7: Beispiel für die schnellere Sortierung mit TSQS unter Ausnutzung von aktuell generierten Gruppennummern.

Sei u als Pivotelement von Suffix s_i gewählt und sei Suffix $s_{j+H} = s_i$, mit H als Präfixlänge der aktuellen Phase. Da auch s_k den Schlüssel g besitzt, muss dementsprechend $s_{k+H} = s_j$ gelten. Da es keine Schlüssel kleiner u gibt, entstehen nur zwei Partitionen der Gruppe g. Der mittlere Teil mit dem Pivot selbst wird zu einer sortierten Gruppe mit Nummer $g-2$. Diese Zuweisung hat nun den Effekt, dass bei der Sortierung des rechten Teils mit den beiden Suffixen s_j und s_k andere Schlüssel vorliegen als vor dieser Zuweisung. Ist $KEY(s_j) = SA_H^{-1}[s_{j+H}] = SA_H^{-1}[s_i] = g$ vor der Sortierung und der Zuweisung der neuen Gruppennummer $g-2$ an Suffix s_i, so greift man nun auf $KEY(s_j) = g-2$ zurück. An $SA_H^{-1}[s_j]$ selbst hat sich nichts geändert, sodass $KEY(s_k) = SA_H^{-1}[s_{k+H}] = SA_H^{-1}[s_j] = g$ immer noch gilt. Die beiden Schlüssel $g-2$ und g bewirken demnach, dass Suffix s_j in die sortierte Gruppe $g-1$ und Suffix s_k in die sortierte Gruppe g eingeteilt werden. Hätte man den aktuellen Schlüssel von s_i nicht verwendet, so wäre eine unsortierte Gruppe mit den Suffixen s_j und s_k entstanden, die in der nächsten Phase bearbeitet worden wäre. Diese Art von Schlüsseln, die sofort für die weitere Sortierung verwendet werden, nennt man *dynamische Schlüssel*. Der Einsatz von dynamischen Schlüsseln erlaubt für viele Eingabedistributionen eine schnellere Sortierung und eine Verringerung der Phasenanzahl. Abb. 5.7 soll diesen Effekt anhand eines konkreten Beispiels demonstrieren.
Es sind sieben Suffixe der unsortierten Gruppe 14 zu sortieren. Die Schlüssel sind angegeben und zwei von diesen entsprechen der Gruppennummer 14. Die beiden Suffixe, die den Schlüssel 14 liefern, haben die Schlüssel 2 und 5. TSQS wähle nun 6 als Pivotelement, sodass die Schlüssel $2, 5$ in den linken Teil, 6, 6 in den mittleren Teil und $14, 14, 20$ in den rechten Teil der Partitionierung gelangen. Die Sortierung des linken Teils ergibt die Reihenfolge $2, 5$ mit zwei neuen sortierten Gruppen 8 und 9. Die Pivotgruppe erhält Gruppennummer 11. Würde man nun den rechten Teil mit den ursprünglichen Schlüsseln verwenden, so würde sich eine unsortierte Gruppe 13 und eine sortierte Gruppe 14 ergeben. Diese unsortierte Gruppe 13 ist gestrichelt eingezeichnet.

Da sich nun aber die Gruppennummern für die Suffixe mit den Schlüsseln 2 und 5 geändert haben von jeweils 14 auf 8 und 9, so haben die beiden Suffixe, die vorher auf die 14 zugegriffen haben, nun die neuen Schlüssel 8 und 9. Genau aus diesem Grund entstehen die beiden sortierten Gruppen 12 und 13 anstatt wie beschrieben eine unsortierte Gruppe aus diesen beiden Suffixen. Dieser positive Effekt hat nicht nur wie im Beispiel Auswirkung auf die aktuelle Gruppe, sondern pflanzt sich eventuell für alle nachfolgenden Gruppen fort, sodass eine erneute Beschleunigung der Sortierung eintreten kann.

5.3 LCP-Berechnung an Bucketköpfen

Für die Berechnung der *LCP*-Werte im Algorithmusablauf geht man sowohl bei Manber und Myers als auch bei Larsson und Sadakane phasenweise vor. Den Anfang für Phase $k = 0$ bildet das initiale Sortieren nach dem ersten Symbol und der LCP-Wert an allen Positionen von Bucketköpfen ist 0, d.h. man trägt an allen Startpositionen i der Buckets $LCP[i] = 0$ ein, ansonsten $LCP[i] = \perp$. Dabei sollen die $\perp$-Werte für *LCP*-Positionen stehen, für die am Ende einer Phase der LCP-Wert noch nicht berechnet werden kann. An diesen Positionen befinden sich Suffixe, die im gleichen Bucket liegen und somit noch nicht genügend ausdifferenziert sind. Seien im Folgenden am Ende von Phase $k \geq 1$ die *LCP*-Werte an Positionen benachbarter Buckets bekannt. Während eines Durchlaufs durch SA_H entstehen neue Buckets. Die nachfolgenden Aussagen und Bedingungen haben absolute Gültigkeit nur für Manber und Myers. Für Larsson und Sadakane gelten Ausnahmen, die später genauer beschrieben werden. Lemma 5.3.1 gibt das LCP-Berechnungsschema für zwei Suffixe s_i und s_j an, die am Ende von Phase k im gleichen Bucket und am Ende von Phase $k+1$ in verschiedenen Buckets liegen.

Lemma 5.3.1 *Seien s_i und s_j zwei Suffixe, die am Ende von Phase k mit $H = 2^k$ im gleichen Bucket liegen, d.h. es gilt $SA_H^{-1}[i] = SA_H^{-1}[j]$. Am Ende von Phase $k+1$ gelte $SA_{2H}^{-1}[i] \neq SA_{2H}^{-1}[j]$, d.h. die beiden Suffixe s_i und s_j liegen in verschiedenen Buckets. Dann gilt:*

$$lcp(s_i, s_j) = H + lcp(s_{i+H}, s_{j+H}) < 2H$$

Beweis: *Am Ende von Phase k liegt SA_H vor, also ein partiell geordnetes Suffix-Array nach den ersten H-Symbolen aller Suffixe. Für zwei beliebige Suffixe i und j eines Buckets aus SA_H gilt $s_i =_H s_j$, also $lcp(s_i, s_j) \geq H$. Gilt nun $SA_{2H}^{-1}[i] \neq SA_{2H}^{-1}[j]$ am Ende von Phase $k+1$, so müssen die beiden Suffixe $i+H$ und $j+H$ am Ende von Phase k in verschiedenen Buckets gewesen sein, denn sonst würde $s_{i+H} =_H s_{j+H}$ gelten und damit auch $s_i =_{2H} s_j$. Damit wären s_i und s_j am Ende von Phase $k+1$ im gleichen Bucket, im Widerspruch zur Voraussetzung. Somit gilt $lcp(s_{i+H}, s_{j+H}) < H$ und $lcp(s_i, s_j) < 2H$. Damit ist $lcp(s_i, s_j) = H + lcp(s_i[H..2H), s_j[H..2H))$, wobei der zweite Summand $lcp(s_{i+H}, s_{j+H})$ entspricht.* □

Der LCP-Wert von Suffix $i+H$ und $j+H$ wäre am Ende von Phase k nur dann verfügbar, wenn nach Voraussetzung diese Suffixe in benachbarten Buckets liegen würden. Davon kann aber im Allgemeinen nicht ausgegangen werden und es muss eine explizite Berechnung dieses LCP-Wertes mit Hilfe von Satz 4.2.1 erfolgen. Ist ein LCP-Wert im Verlauf des Algorithmus berechnet worden, so bleibt dieser bis zur Terminierung unverändert.

Seien SA_H, SA_H^{-1} und LCP_H die berechneten Felder am Ende von Phase k, wobei $H = 2^k$. Nach Phase $k+1$ stehen SA_{2H}, SA_{2H}^{-1} zur Verfügung und Satz 5.3.2 stellt die Korrektheit des Berechnungsschemas für LCP_{2H} her.

Satz 5.3.2 (Manber & Myers) *Seien SA_{2H}, SA_{2H}^{-1} und LCP_H am Ende von Phase $k+1$ gegeben. Weiter seien*

- $a = SA_{2H}^{-1}[SA_{2H}[i-1] + H]$ *und*
- $b = SA_{2H}^{-1}[SA_{2H}[i] + H]$.

Dann gilt:

$$H \leq lcp(s_{SA_{2H}[i-1]}, s_{SA_{2H}[i]}) < 2H \Rightarrow$$
$$LCP_{2H}[i] = H + \min\{LCP_H[k] \mid k \in [a+1, b]\}$$

Beweis: *Zunächst gilt mit 5.3.1 $a < b$, denn wäre $a \geq b$, dann muss wegen $s_{SA_{2H}[i-1]} \leq_{2H} s_{SA_{2H}[i]}$ gelten, dass $s_{SA_{2H}[a]} =_H s_{SA_{2H}[b]}$ ist. Damit wäre $lcp(s_{SA_{2H}[i-1]}, s_{SA_{2H}[i]}) \geq 2H$ im Widerspruch zur Voraussetzung. Es gilt also mit 5.3.1 $LCP_{2H}[i] = H + lcp(s_u, s_v)$, wobei $u = SA_{2H}[a]$, $v = SA_{2H}[b]$ und $s_u^H < s_v^H$. Mit 4.2.1 folgt*
$lcp(s_u, s_v) = \min_{k \in [a+1,b]}\{lcp(s_{SA_{2H}[k-1]}, s_{SA_{2H}[k]})\}$. Aus $lcp(s_u, s_v) < H$ folgt, dass mindestens ein Wert aus dieser Menge kleiner als H ist. Für diese k gilt
$lcp(SA_{2H}[k-1], SA_{2H}[k]) = LCP[k] = LCP_H[k]$. □

Mit Hilfe von 5.3.2 lassen sich alle *LCP*-Werte phasenweise berechnen, indem für einen bestimmten Bereich von *LCP* das Minimum gesucht wird. Wichtig dabei ist, die Minimumsuche effizient zu gestalten und der nächste Abschnitt stellt eine Datenstruktur vor, die dies ermöglicht.

5.4 Der Intervallbaum

Wendet man Satz 5.3.2 zur Berechnung der LCP-Werte direkt an, so liegt das lineare Durchsuchen von *LCP* für das Minimum in $O(n)$ und bei genau n zu bestimmenden Minima erhält man eine quadratische Laufzeit. Dasselbe Ergebnis ergibt sich, wenn man für alle Paare von Suffixen das Minimum in einer Tabelle speichert. Man ist daran interessiert, die Berechnung der *LCP*-Tabelle so in den Ablauf einzubetten, dass sich die worst-case Laufzeit des Algorithmus nicht verschlechtert. Es ist also eine Datenstruktur gefragt, die eine Range Minimum Query effizient unterstützt. Es existieren mit [11, 3] Verfahren, die das Auffinden des Minimums eines Bereichs in konstanter Zeit ermöglichen. Dazu ist eine lineare Vorverarbeitung des gesamten Eingabefeldes nötig. Prinzipiell könnte man diese auch für die LCP-Berechnung aus Satz 5.3.2 einsetzen, es soll hier dennoch ein anderes Verfahren vorgestellt werden, das auf Manber und Myers selbst zurückgeht. Nach jeder Phase lassen sich an Bucketstartpositionen LCP-Werte berechnen und damit ist die LCP-Tabelle nur partiell mit definierten Werten ≥ 0 gefüllt. Um nun Range Minimum Queries auf dieser Tabelle in konstanter Zeit ausführen zu könnnen, müsste man nach jeder der $\log n$-Phasen die gesamte LCP-Tabelle vorverarbeiten.

Dies resultiert in einem $O(n \log n)$-Aufwand und deshalb kann man auch auf andere Verfahren zurückgreifen, die die gleichen Kosten verursachen, sich aber aus praktischer Sicht als effizient erweisen. Eine solche Datenstruktur, die eine Minimumanfrage in logarithmischer Zeit beantworten kann, ist der Intervall-Baum[3], der in 5.4.1 definiert wird.

Definition 5.4.1 *Sei $n \in \mathbb{N}$ und $a, b, c \in [0..n-1]$. Ein LCP-Intervall $LCP[a, b]$ wird definiert durch:*

$$LCP[a, b] := \min\{LCP[i] \mid i \in [a+1..b]\}$$

Der Intervall-Baum IB_n ist ein Binärbaum und wird definiert durch:

- *Jedem Knoten ist genau ein LCP-Intervall $LCP[a, b]$ zugeordnet mit $a < b$.*
- *Der Wurzel ist das LCP-Intervall $[0, n-1]$ zugeordnet.*
- *Alle Knoten $LCP[a, b]$ mit $b - a = 1$ sind Blätter.*
- *Ein Knoten $LCP[a, b]$ mit $b - a \geq 2$ hat $LCP[a, c]$ und $LCP[c, b]$ als linken bzw. rechten Sohn, falls $c - a = \max\{2^i \mid i \in \mathbb{N} \wedge 2^i < b - a\}$ ist.*

Mit Satz 4.2.1 gilt $LCP[a, b] = lcp(s_{SA[a]}, s_{SA[b]})$ und $LCP[a, b] = \min\{LCP[a, c], LCP[c, b]\}$ für ein beliebiges $c \in [a+1..b-1]$. Damit repräsentiert ein LCP-Intervall das Minimum aus dem entsprechenden Bereich.
In Abb. 5.8 ist ein Beispiel für einen Intervall-Baum IB_8 dargestellt. Die Werte an den Blättern entsprechen von links nach rechts gelesen der LCP-Tabelle und an allen inneren Knoten ist das Minimum der Werte der beiden Kindknoten eingetragen. Soll nun für einen beliebigen Bereich $[i..j]$ das Minimum mit Hilfe von IB_n bestimmt werden, muss unter anderem ein bestimmter Knoten gefunden werden. Dieser ist der kleinste gemeinsame Vorfahre der Blätter $LCP[i-1, i]$ und $LCP[j-1, j]$ und wird formal in 5.4.2 definiert.

Definition 5.4.2 *Der kleinste gemeinsame Vorfahre zweier Knoten u und v eines Baumes mit Wurzelknoten r ist der letzte gemeinsame Knoten auf den Pfaden von r zu u und von r zu v. Dieser wird mit $LCA(u, v)$ bezeichnet.*

Mit Hilfe von $LCA(i, j)$ kann das Minimum aus $[i..j]$ aus dem Minimum der folgenden Werte bestimmt werden.

- Wert des Blattes $LCP[i-1, i]$
- Sei P die Menge aller Knoten in IB_n auf dem Pfad von $LCP[i-1, i]$ zu $LCA(i, j)$, ohne $LCA(i, j)$. Dann sind alle Werte derjenigen Knoten w zu betrachten, die rechter Sohn eines Knotens v sind, wobei $w \notin P$ und $v \in P$.
- Wert des Blattes $LCP[j-1, j]$
- Sei Q die Menge aller Knoten in IB_n auf dem Pfad von $LCP[j-1, j]$ zu $LCA(i, j)$, ohne $LCA(i, j)$. Dann sind alle Werte derjenigen Knoten w zu betrachten, die linker Sohn eines Knotens v sind, wobei $w \notin Q$ und $v \in Q$.

[3] Diese Namensgebung stammt von den Autoren und ist nicht zu verwechseln mit dem LCP-Intervall-Baum aus Abschnitt 4.1.

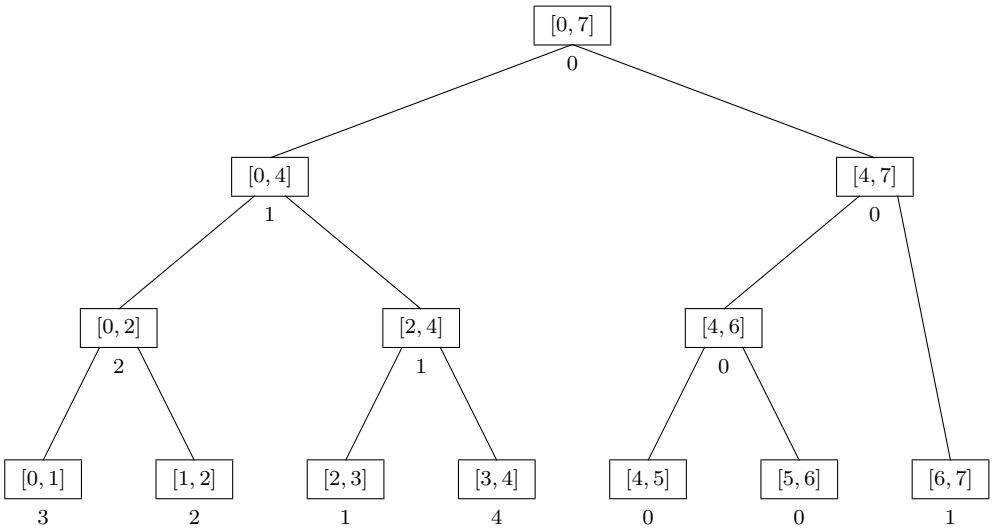


Abbildung 5.8: Beispiel eines LCP-Intervallbaums für die Realisierung einer Minimumanfrage in einem LCP-Intervall.

Da IB_n ein linksvollständiger Binärbaum ist, besitzt er die Höhe $\lceil \log_2 n \rceil$ und somit liegt die Anzahl der Knoten eines beliebigen Pfades von einem Blatt zu einem inneren Knoten in $O(\log n)$. Deshalb kann das Minimum aus den obigen Werten in $O(\log n)$-Zeit bestimmt werden. Als Beispielanfrage an den Baum aus Abb. 5.8 sei das Minimum aus dem Bereich [1..5] gesucht. Dafür ist das Minimum aus den folgenden Werten zu bilden:

- $LCP[0, 1] = 3$
- Die Knoten auf dem Pfad von $LCP[0, 1]$ zu $LCA(LCP[0, 1], LCP[4, 5]) = LCP[0, 7]$ (ohne diesen) sind $LCP[0, 1]$-$LCP[0, 2]$-$LCP[0, 4]$. Dann ist $LCP[1, 2]$ rechter Sohn von $LCP[0, 2]$ und liegt nicht auf diesem Pfad. Ebenso ist $LCP[2, 4]$ rechter Sohn von $LCP[0, 4]$ und liegt nicht auf diesem Pfad. Die relevanten Werte dieses Pfades sind also 2 und 1.
- $LCP[4, 5] = 0$
- Die Knoten auf dem Pfad von $LCP[4, 5]$ zu $LCA(LCP[0, 1], LCP[4, 5]) = LCP[0, 7]$ (ohne diesen) sind $LCP[4, 5]$-$LCP[4, 6]$-$LCP[4, 7]$. Auf diesem Pfad gibt es keine linken Söhne, die nicht selbst zum Pfad gehören, deswegen ist hier die betrachtete Wertemenge leer.

Dadurch ergibt sich $\min[1..5] = \min\{3, 2, 1, 0\} = 0$.

Werden nun LCP-Werte berechnet und gesetzt, so muss IB_n für jeden neu entstandenen Wert aktualisiert werden, sodass die Bedingungen aus Definition 5.4.1 wieder hergestellt sind. Sei $LCP[i] = k$ ein neu berechneter Wert einer beliebigen Phase. Dann trägt man an das Blatt $LCP[i - 1, i]$ den Wert k ein und betrachtet den Wert l des Vaterknotens.

Ist $l \leq k$, so ist nichts weiter zu tun. Andernfalls geht man zum Vaterknoten und trägt auch dort k ein und betrachtet dessen Vaterknoten usw. Auch diese Aktualisierungsoperation liegt offensichtlich in $O(\log n)$. In Algorithmus 5.3 ist eine Realisierung der Minimumsuche unter Verwendung von IB_n dargestellt und in Algorithmus 5.4 das Einfügen eines neuen LCP-Wertes.

Algorithmus 5.3 : Algorithmus zur Minimumbestimmung in IB_n

Eingabe : linke Grenze l, rechte Grenze r, Intervallbaum IB_n
Ausgabe : Index des Minimums aus $[l..r]$

```
1  if l = r then
2      return l
3  l_vertex ist [l − 1, l]-Knoten in IB_n
4  r_vertex ist [r − 1, r]-Knoten in IB_n
5  l_min ← l_vertex
6  r_min ← r_vertex
7  l_lastvisit ← l_vertex
8  r_lastvisit ← r_vertex
9  while TRUE do
10     if l_vertex.parent = r_vertex.parent then
11         break
12     if l_vertex.rightChild.value < l_min then
13         if l_vertex.rightChild ≠ l_lastvisit then
14             l_min ← l_vertex.rightChild
15     if r_vertex.leftChild.value < r_min then
16         if r_vertex.rightChild ≠ r_lastvisit then
17             r_min ← r_vertex.leftChild
18     l_lastvisit ← l_vertex
19     r_lastvisit ← r_vertex
20     l_vertex ← l_vertex.parent
21     r_vertex ← r_vertex.parent
22 if l_min ≤ r_min then
23     return l_min
24 else
25     return r_min
```

5.5 LCP-Berechnung bei Manber und Myers

Für die LCP-Berechnung im Algorithmus von Manber und Myers wird nach jeder Phase das Feld NBH durchlaufen und immer dann, wenn mit $NBH[i] = 1$ ein Bucketkopf festgestellt wird und an dieser Position $LCP[i] = \perp$ gilt, wird $LCP[i]$ mit Hilfe von 5.3.2 und dem Intervall-Baum IB_n berechnet.

Algorithmus 5.4 : Einfügen eines neuen LCP-Wertes in IB_n und dessen Aktualisierung

Eingabe : Intervallbaum IB_n, Einfügeposition p mit Wert v
Ausgabe : Aktualisierter Intervallbaum IB_n mit neuem Wert v

```
1 while TRUE do
2     if p_vertex.parent.value > v then
3         p_vertex.parent.value = v
4     else
5         return
6     if p_vertex.parent = root then
7         return
8     else
9         p_vertex ← p_vertex.parent
```

Hierfür werden die beiden benötigten Werte $a+1$ und b des Intervalls $[a+1, b]$ bestimmt, innerhalb derer sich das gesuchte Minimum $\min(LCP_H[k] \mid k \in [a+1, b])$ befindet. Ist der neue LCP-Wert berechnet worden, dann wird dieser konsistent in IB_n gespeichert. In Algorithmus 5.5 ist dieser Berechnungsvorgang am Ende jeder Phase in Pseudocode dargestellt, wobei *GetMin* durch Algorithmus 5.3 und *Set* durch Algorithmus 5.4 realisiert sind.

Algorithmus 5.5 : LCP-Berechnung bei Manber und Myers

Eingabe : LCP_H,SA_{2H},SA_{2H}^{-1},NBH_{2H}
Ausgabe : LCP_{2H}

```
1 for i ∈ [0..n − 1] do
2     if NBH_2H[i] = 1 ∧ LCP_H[i] = ⊥ then
3         a ← SA_2H^-1[SA_2H[i − 1] + H]
4         b ← SA_2H^-1[SA_2H[i] + H]
5         LCP_2H[i] ← H + LCP_H[GetMin(a + 1, b)]
6         Set(i, LCP_2H[i])
```

Die Laufzeit von 5.5 ist $O(n+\log n \cdot k)$, wobei k die Anzahl der Suffixindizes ist, für die am Ende einer Phase der LCP-Wert in $[H..2H)$ liegt. Da $\sum_0^{\log n} k = n$ über alle Phasen ist, ergibt sich als gesamter zusätzlicher Berechnungsaufwand für die LCP-Tabelle $O(n \log n)$. Damit ist eine "sanfte" Einbettung in den Basis-Algorithmus möglich, unter dem Aspekt, dass sich die asymptotische worst-case Laufzeit nicht verschlechtert.

5.6 Das Bucket-Head-Array

Eine effiziente Implementierung des Algorithmus von Manber und Myers in C stellte McIlroy vor [26]. Dabei wurde auch eine gegenüber dem Intervallbaum alternative Datenstruktur eingeführt, die die Minimumsuche unterstützt und in Definition 5.6.1 eingeführt wird.

Definition 5.6.1 *Ein Bucket-Head-Array, BH, ist ein Feld* $BH[0..n)$ *von Werten im Bereich* $[0..n)$. *Sei* $j = \max\{k \,|\, LCP[k] \neq \bot \wedge LCP[k] < LCP[i] \wedge k < i\}$ *für ein* $i \in [0..n)$. *BH wird definiert durch:*

$$BH[i] := \begin{cases} \bot & , \textit{falls} \quad LCP[i] = \bot \vee j = \bot \\ j & \textit{sonst.} \end{cases}$$

Mit anderen Worten sind damit die $BH[i]$-Werte nur dann definiert, also $BH[i] \neq \bot$, falls an dieser Position auch ein definierter LCP-Wert vorhanden ist und wenn eine Position $j < i$ in LCP existiert, die einen echt kleineren Wert als an Position i besitzt. In Abb. 5.9 ist ein Beispiel für ein BH-Feld dargestellt.
Um nun mit Hilfe von BH das Minimum eines Bereichs $[i..j]$ in LCP zu finden, folgt man, beginnend bei $BH[j]$ den Werten zu Positionen mit echt kleinerem LCP-Wert als an Stelle j. Abgebrochen wird dieser Vorgang, wenn $BH[k] < i$ für $k \in [i..j]$ gilt, denn dann wurde das Suchintervall komplett durchlaufen, oder wenn $BH[k] = \bot$ ist, denn dann existiert an keiner kleineren Position als k ein echt kleiner LCP-Wert. Für beide Fälle ist somit $LCP[k]$ das gesuchte Minimum aus $[i..j]$. Lemma 5.6.2 zeigt die Korrektheit dieser Vorgehensweise.

Lemma 5.6.2 *Sei* $[i..j]$ *ein Intervall aus LCP und* $M = \{a_i \,|\, i \geq 0\}$ *mit*

- $a_0 := j$
- $a_i := BH[a_{i-1}]$ *für* $i \geq 1$, *falls* $a_{i-1} \neq \bot$

Der größte Index aller Minima aus $LCP[i..j]$ *sei mit* l *bezeichnet, d.h.*
$l := \max\{m \,|\, LCP[m] = \min[i..j]\}$. *Dann gilt:*

$$l \in M$$

Beweis: *Im Fall von* $l = j$ *gilt die Behauptung wegen* $j \in M$. *Sei also* $l < j$ *und* $l' = \min\{k \,|\, k \in M \wedge k > l\}$. *Zu zeigen ist nun, dass* $BH[l'] = l$ *und somit* $l \in M$ *gilt. Es lassen sich zwei Fälle unterscheiden:*

- *Gilt* $BH[l'] = \bot$, *so existiert nach 5.6.1 kein echt kleinerer LCP-Wert im Bereich* $[0..l]$. *Damit gilt* $LCP[l] > LCP[l']$, *da Gleichheit wegen der Definition von* l *ausgeschlossen ist. Da* $l' \in [i..j]$ *kann* l *nicht der Index des Minimums sein und führt somit zum Widerspruch.*
- *Ist* $BH[l'] < l$, *so muss wieder nach Definition 5.6.1* $LCP[l] \geq LCP[l']$ *gelten. Mit der gleichen Argumentation wie im ersten Fall, folgt auch hier der Widerspruch.* □

Die LCP-Berechnung im Algorithmus von Larsson und Sadakane soll nun mit Hilfe von BH beschrieben werden, wobei auch eine Realisierung mit dem Intervall-Baum IB_n denkbar wäre.

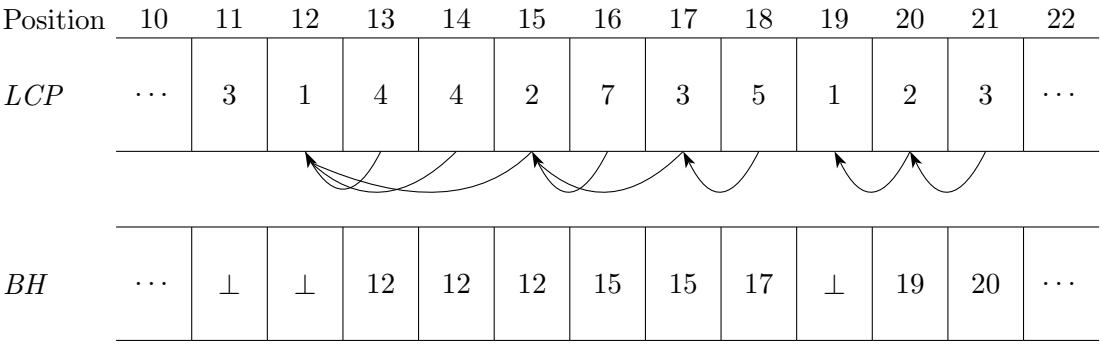


Abbildung 5.9: Beispiel für eine Verzeigerung der LCP-Werte in *BH*.

5.7 LCP-Berechnung bei Larsson und Sadakane

Die LCP-Berechnung gestaltet sich bei Larsson und Sadakane schwieriger als bei Manber und Myers, da andere Randbedingungen gelten und durch die vorgestellten Optimierungen gewisse Informationen verloren gehen, die eine einfachere Realisierung ermöglichen würden. Zunächst lässt sich feststellen, dass man zur LCP-Bestimmung in Algorithmus 5.5 die beiden Positionen a und b für das Suchintervall benötigt. Diese werden über Zugriffe auf SA und SA^{-1} in den Zeilen $3-4$ berechnet. In Abschnitt 5.1 ist angegeben, dass das Feld LEN zur Speicherung von Längen in das Suffix-Array versteckt werden kann. Das hat zur Folge, dass in SA nicht an allen Positionen Suffixindizes gespeichert sind, sondern auch Längen von unsortierten oder sortierten Gruppen. Es liegt somit nicht wie bei Manber und Myers ein konsistenter Zustand in SA vor. Möchte man zur LCP-Berechnung grundsätzlich ähnlich wie bei Manber und Myers vorgehen, führt man ein Feld $CSA[0..n-1]$ ein, das diese Suffixindizes speichert, bevor sie überschrieben werden. Mit $CSA[i] = \perp$ für $i \in [0..n-1]$ werden alle Positionen zunächst auf undefiniert gesetzt. Dies hat den Vorteil, dass damit auch konzeptionell das Feld NBH von Manber und Myers dargestellt werden kann. Denn immer dann, wenn die Gruppennummern von Suffixen eines Buckets aktualisiert werden, speichert man die aktuell an den Grenzen dieses Buckets liegenden Suffixinidizes in CSA. Somit zeigt ein Wert $CSA[i] \neq \perp$ einen Bucketanfang oder ein Bucketende an.

Eine weitere Komplikation ergibt sich durch die Tatsache, dass, im Gegensatz zu Manber und Myers, die Gruppennummern von Suffixen eines Buckets immer mit der Bucketendposition identisch sind. Für die Berechnung der Intervallgrenzen $a+1$ und b nach Satz 5.3.2 werden aber die Bucketanfangspositionen mit definiertem LCP-Wert benötigt. Um die Anfangsposition eines Buckets über ein in diesem enthaltenen Suffix in konstanter Zeit zu erhalten und um Speicherplatz zu sparen benutzt man noch undefinierte BH-Positionen. Dies geschieht so, dass an allen Positionen i eines Bucketendes $BH[i] = j$ gilt, wobei $j = \max\{k | k < i \wedge LCP[k] \neq \perp\}$ der nächstkleinere Bucketanfang mit einem definiertem LCP-Wert ist. Dies wird auch für einzelne Buckets, also sortierten Gruppen, durchgeführt, da diese in der aktuellen Phase erst entstanden sind und somit noch keinen definierten BH-Eintrag besitzen. Position j erhält man für alle ausgegebenen Buckets von TSQS über die Gruppennummer der Suffixe dieses Buckets.

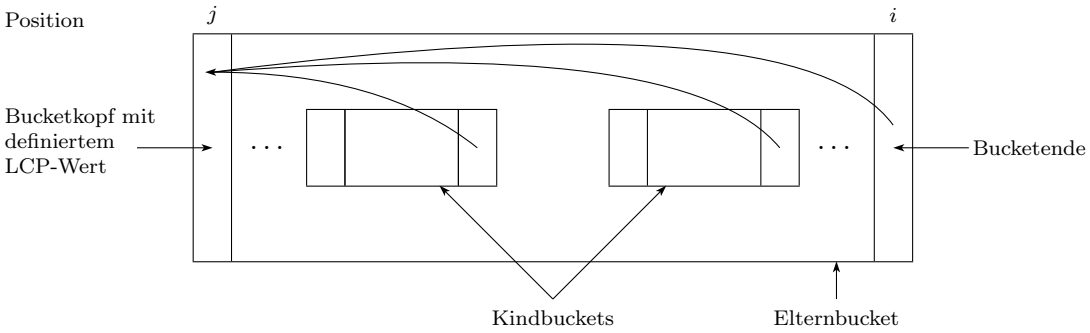


Abbildung 5.10: Verzeigerung der Endpositionen von Kindbuckets innerhalb des Elternbuckets.

Diese Nummer muss gespeichert werden, bevor für diese Suffixe die neue Gruppennummer in SA^{-1} festgelegt wird. Die Semantik von BH ist damit an allen Bucketenden verändert worden, sodass dies später bei der LCP-Berechnung berücksichtigt werden muss. In Abb. 5.10 ist diese Verzeigerung innerhalb von BH für die Endpositionen der Buckets schematisch dargestellt. Mit Elternbucket ist dasjenige Bucket gemeint, das vor der Sortierung mit TSQS vorlag und alle dadurch entstandenen Kindbuckets enthält. Der Kopf des Elternbuckets liegt an Position j und das Ende an Position i. Alle Suffixe l aus dem Elternbucket haben vor der Sortierung im inversen Suffix-Array den Eintrag $SA^{-1}[l] = i$. Wird nun für eine LCP-Berechnung mit Satz 5.3.2 in der Minimumsuche ein Suffix aus einem Kindbucket angesprochen, so muss auf die Kopfposition j des Elternbuckets zugegriffen werden können, da an dieser Position $LCP[j] < H$ gilt. Die benötigten Intervallpositionen a und b bekommt man über eine Fallunterscheidung, wobei dies nur für die obere Grenze gezeigt werden soll, denn die untere wird völlig analog ermittelt.
Ist an Position $SA^{-1}[CSA[i] + H] = k$ ein definierter Wert $LCP[k] \geq H$ vorhanden, so liegt eine sortierte Gruppe vor, also ein Bucket mit genau einem Suffix, und der LCP-Wert an dieser Position wurde in der aktuellen Phase berechnet. Ist $LCP[k] = \perp$, so liegt eine Endposition eines Kindbuckets vor. In beiden Fällen findet man die Kopfposition des Elternbuckets über $b = BH[k]$, da wie vorstehend beschrieben alle Bucketenden auf diese Kopfposition zeigen. Gilt also weder $LCP[k] \geq H$ noch $LCP[k] = \perp$, so liegt die Kopfposition des Elternbuckets selbst als sortierte Gruppe nach TSQS vor und die obere Intervallgrenze ist $b = k$.

Die nächste Schwierigkeit ergibt sich durch die zweite Optimierung aus Abschnitt 5.2, die dynamische Schlüssel zur Beschleunigung der Sortierung bei TSQS verwendet. Dazu betrachtet man zwei benachbarte Buckets während des Algorithmenablaufs mit i als Bucketkopfposition des rechten Buckets und $i-1$ als Bucketendposition des linken Buckets. Das rechte Bucket sei in der aktuellen Phase generiert worden.

Es spielt bei Manber und Myers keine Rolle, welches Suffix aus einem Bucket aktuell an der Kopf- oder Endposition steht, denn alle Suffixe des linken Buckets sind von allen Suffixen des rechten Buckets durch weniger als $2H$ Symbole unterschieden. Diese Eigenschaft stellt Lemma 5.3.1 her und ist Zusicherung für alle Phasen. Durch die Verwendung von dynamischen Schlüsseln kann es wie in Abb. 5.7 vorkommen, dass Bucketkopfpositionen i entstehen mit der Eigenschaft, dass $LCP_{2H}[i] \geq 2H$ gilt. Denn wenn zwei Suffixe i und j aus einer Gruppe g jeweils $KEY(i) = KEY(j) = g$ als Schlüssel haben und sie nach der Sortierung dennoch in benachbarten Buckets liegen, so muss ihr LCP-Wert mindestens $2H$ betragen, da die beiden Suffixe $i+H$ und $j+H$ im gleichen Bucket vor der Sortierung waren. In diesen Fällen trifft 5.3.1 nicht mehr zu und man kann damit den LCP-Wert an Position i am Ende der aktuellen Phase nicht berechnen, d.h. dass man die LCP-Berechnung an diesen Positionen zurückstellen muss. Es wird ein Kriterium benötigt, das benachbarte Buckets erkennt, die aufgrund von dynamischen Schlüsseln entstanden sind. Hier hilft die Beobachtung, dass die beiden Schlüssel $KEY(SA[i-1]+H)$ und $KEY(SA[i]+H)$ vor der Sortierung identisch gewesen sein müssen. Denn falls nicht, lagen die beiden Suffixe $SA[i-1]+H$ und $SA[i]+H$ in verschiedenen Buckets und das betrachtete Bucketpaar wäre ohne dynamischen Schlüssel entstanden. Es muss also getestet werden, ob die beiden Gruppennummern von diesen Suffixen am Ende der letzten Phase indentisch waren. Dies lässt sich indirekt über die beiden ermittelten Intervallgrenzen mit $a = b$ testen. Kann schließlich der LCP-Wert ermittelt werden, so benutzt man zur Minimumsuche das Feld BH wie beschrieben, indem man den Positionen zu immer kleineren Werten folgt, bis die untere Grenze a unterschritten wird oder $BH[i] = \perp$ gilt. In Algorithmus 5.6 ist die LCP-Berechnung am Ende einer Phase bei Larsson und Sadakane in Pseudocode dargestellt.

Nach der LCP-Berechnung einer Phase müssen die BH-Werte noch auf den aktuellen Stand gebracht und ggf. die semantisch veränderten Einträge für die Verzeigerung auf Kopfpositionen korrigiert werden. Dazu durchläuft man CSA_{2H} sequentiell. Zunächst wird geprüft, ob die aktuelle Position ein Bucketende einer unsortierten Gruppe ist. Dies lässt sich über die Bedingung $BH[i] \neq \perp \wedge LCP[i] = \perp$ feststellen, denn ein BH-Wert ohne dazugehörigen LCP-Wert kann nur ein Zeiger auf einen Bucketkopf sein. Diese BH-Positionen müssen nun auf die Positionen der zuletzt gesetzten LCP-Werte zeigen, damit ein konsistenter Zustand für die nächste Phase erreicht wird. Hierfür hält man sich eine Variable, die immer dann, wenn ein BH-Wert im Sinne von 5.6.1 definiert wird, die aktuelle Position speichert. Als nächstes prüft man den LCP-Wert an der aktuellen Position. Ist $LCP[i] \geq H$, so wurde dieser Wert in der aktuellen Phase gesetzt und es ist noch $BH[i] = \perp$. Es beginnt der Vergleich der LCP-Werte an der aktuellen Position mit der zuletzt gespeicherten, an der ein definierter LCP-Wert vorlag. Gilt hingegen $LCP[i] \neq \perp$, also $LCP[i] \in [0..H)$, so liegt zwar auch ein Bucketkopf vor, der aber in einer früheren Phase schon einen Wert zugewiesen bekommen hat. In diesem Fall merkt man sich diese Position als die aktuell letzte mit einem definierten LCP-Wert. Algorithmus 5.7 gibt dieses Aktualisierungsverfahren der BH-Werte an.
Die letzte Ausnahmebehandlung hat auch mit der Verwendung von dynamischen Schlüsseln bei TSQS zu tun, da es vorkommen kann, dass sich dadurch weniger Phasen ergeben, als dies bei Manber und Myers der Fall wäre.

Algorithmus 5.6 : LCP-Berechnung bei Larsson und Sadakane

Eingabe : LCP_H,CSA_{2H},SA_{2H}^{-1},BH_H
Ausgabe : LCP_{2H}

```
1  for i ∈ [1..n − 1] do
2      if LCP_H[i] ≠ ⊥ then
3          continue
4      if CSA_2H[i − 1] ≠ ⊥ ∧ CSA_2H[i] ≠ ⊥ then
5          g ← SA_2H^{-1}[CSA_2H[i] + H]]
6          l ← LCP_H[g]
7          if l = ⊥ ∨ l ≥ H then
8              oG ← BH_H[g]
9          else
10             oG ← g
11         g ← SA_2H^{-1}[CSA_2H[i − 1] + H]]
12         l ← LCP_H[g]
13         if l = ⊥ ∨ l ≥ H then
14             uG ← BH_H[g]
15         else
16             uG ← g
17         if og ≠ uG then
18             tmp ← oG
19             u ← n
20             while tmp > uG OR tmp ≠ ⊥ do
21                 u ← LCP_H[tmp]
22                 tmp ← BH_H[tmp]
23             LCP_2H[i] ← H + u
```

Algorithmus 5.7 : Aktualisierung der *BH*-Werte bei Larsson und Sadakane

Eingabe : LCP_{2H},CSA_{2H},SA^{-1}_{2H},BH_H
Ausgabe : BH_{2H}

```
1  last_lcp_pos ← 1
2  for i ∈ [1..n − 1] do
3      u ← LCP_2H[i]
4      if BH_H[i] ≠ ⊥ ∧ u = ⊥ then
5          BH_2H[i] ← last_lcp_pos
6          continue
7      if u ≥ H then
8          tmp ← last_lcp_pos
9          BH_2H[i] ← last_lcp_pos
10         while LCP_2H[tmp] ≥ u do
11             tmp ← BH_H[tmp]
12             BH_2H[i] ← tmp
13     else
14         if u ≠ ⊥ then
15             last_lcp_pos ← i
```

Existieren in der letzten Phase Positionen, denen noch kein LCP-Wert zugewiesen werden konnnte, weil die Schlüssel des Bucketkopfes und des vorhergehenden Bucketendes identisch sind, so muss eine Extraphase zur LCP-Bestimmung angehängt werden. In diesem letzten Durchlauf mit Algorithmus 5.6 ist garantiert, dass alle fehlenden LCP-Werte bestimmt werden können, da der Basisalgorithmus selbst genau n sortierte Gruppen und damit n verschiedene Gruppennummern generiert hat.

McIlroy hat für seine Implementierung mit dem *BH*-Konzept keine Laufzeitanalyse angegeben. Er stellte lediglich fest, dass dieses Vorgehen bei seinen Tests mindestens genauso schnell ist, wie eine garantierte $O(n \log n)$-Methode[4] und dabei sehr viel weniger Code benötigt. Es konnte während dieser Arbeit auch keine Analyse für den Aufwand des Folgens der Zeiger in *BH* gefunden werden. Die Schwierigkeit dürfte vor allem darin liegen, etwas über die Verteilung der Werte in *LCP* aussagen zu können. Diese hängt direkt mit der Struktur der Eingabe s zusammen. Sicher ist nur, dass die Minimumsuche über *BH* in $O(n)$ liegt, denn man kann offensichtlich nur maximal n vielen Zeigern in *BH* folgen. Damit liegt Algorithmus 5.6 in $O(n + n \cdot k)$, wobei k die Anzahl der Positionen ist, für die eine Range Minimum Query in *BH* durchgeführt wird. Summiert man k über alle $\log n$-Phasen auf, so erhält man n und damit liegt der Gesamtaufwand in $O(n^2)$. Algorithmus 5.7 zur Aktualisierung der *BH*-Zeiger benötigt hingegen nur linearen Aufwand.

[4] Wie z.B. der Intervall-Baum aus Abschnitt 5.4.

Soll allgemein an aktueller Position i die erste Position mit echt kleinerem LCP-Wert gefunden werden, so folgt man den Werten ab Position $i-1$, bis diese Bedingung für eine Position j erfüllt ist oder solch eine Position nicht existiert. Anschließend trägt man $BH[i] = j$ bzw. $BH[i] = \perp$ ein. Alle Positionen zwischen $[j+1..i-1]$, die dabei besucht wurden, werden zu einem späteren Zeitpunkt nie mehr betrachtet, denn diese haben alle einen LCP-Wert der größer oder gleich $LCP[i]$ ist. Damit kann $BH[i'] \in [j+1..i-1]$ für keine Position $i' > i$ gelten, denn sonst wäre $LCP[i'] > LCP[i]$ und somit im Widerspruch zur Definition 5.6.1 stehen. Dort ist gefordert, dass ein BH-Wert immer auf die größte Position zeigt, an der ein echt kleinerer LCP-Wert vorhanden ist. Deshalb liegt die Laufzeit von Algorithmus 5.7 in $O(n)$ und bei $\log n$-Phasen ergibt dies einen Gesamtaufwand von $O(n \log n)$. Zusammen mit Algorithmus 5.6 ergibt sich damit ein Aufwand von $O(n^2)$ für die parallele LCP-Berechnung. Es sei nochmals darauf hingewiesen, dass es mit dem Konzept des Intervallbaums oder mit einem RMQ-Verfahren, das Anfragen in konstanter Zeit beantworten kann, möglich ist diesen Aufwand auf $O(n \log n)$ zu reduzieren. Die Wahl für das Konzept des Feldes BH liegt darin begründet, dass in der Literatur die Implementierung von McIlroy als sehr effizient angesehen wird.

Betrachtet man den Speicherverbrauch der beiden Algorithmen 5.6 und 5.7, so müssen die zusätzlichen Felder LCP, CSA und BH berücksichtigt werden. Diese benötigen, wenn man von einer maximalen Eingabelänge von $n = 2^{32}$ ausgeht jeweils $4n$-Bytes, sodass, zusammen mit den $8n$-Bytes für den Basisalgorithmus, sich ein zusätzlicher Speicheraufwand von $20n$-Bytes ergibt.

6 Der Skew-Algorithmus

Im Jahr 2003 wurden unabhängig voneinander drei rekursive Algorithmen für die direkte[1] Konstruktion eines Suffix-Arrays entworfen[15, 19, 20], die alle die asymptotisch optimale worst-case Laufzeit $O(n)$ erzielen. Das prinzipielle Vorgehen dieser Verfahren beruht unter anderem auf [9] und lässt sich abstrakt wie folgt beschreiben:

1. Alle Suffixe werden in Klassen eingeteilt.
2. Auf die Suffixe aus einer Klasse wird ein Sortierverfahren angewendet, sodass allen Suffixen dieser Klasse ein Rang zugewiesen werden kann. Kommt ein Rang mehr als einmal vor, so wird das Sortierverfahren mit einer geeigneten Kombination von Suffixen aus dieser Klasse erneut rekursiv aufgerufen.
3. Die Sortierung der Suffixe aus dem zweiten Schritt wird dazu benutzt, um eine Sortierung der Suffixe aus den anderen Klassen zu erzielen.
4. Alle so entstandenen sortierten Suffixklassen werden in einem Verschmelzungsschritt zusammengeführt, um die endgültige Sortierung aller Suffixe zu erhalten.

In diesem Kapitel wird der sogenannte Skew-Algorithmus[15] von Kärkkäinen und Sanders vorgestellt, der sich am einfachsten von den drei Verfahren beschreiben lässt. Um hier die Einbettung der LCP-Berechnung so gestalten zu können, dass sich die asymptotische Laufzeit nicht verschlechtert, wird ein RMQ-Verfahren benötigt, das es erlaubt benötigte Werte in konstanter Zeit zu erhalten.

6.1 Klassifizierung aller Suffixe

Der Algorithmus von Kärkkäinen und Sanders beginnt mit der Einteilung aller Suffixindizes in drei verschiedene Klassen, die in Definition 6.1.1 eingeführt werden.

Definition 6.1.1 *Sei $s \in \Sigma^n$ das Eingabewort. Die Suffixindizes von s werden wie folgt in drei Klassen eingeteilt:*

$$K_m := \{i \mid i \in [0..n) \wedge i \bmod 3 = m\}$$

Weiter wird $K^{12} := K_1 \cup K_2$ gesetzt.

Im Folgenden wird von $n \bmod 3 = 0$ ausgegangen, um später unnötige Fallunterscheidungen zu vermeiden.

[1] Mit direkt meint man hier ohne den Umweg über einen Suffixbaum.

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$i \bmod 3 = 1$	a	c	a	t	g	c	a	t	t	c	a	g	$\$$	
Nummer			3			5			7			1		
$i \bmod 3 = 2$	a	c	a	t	g	c	a	t	t	c	a	g	$\$$	$\$$
Nummer				2			3			6			4	

Abbildung 6.1: Beispiel für die Tripelbildung und deren Nummerierung anhand von $s = acatgcattcag$ für Schritt 1 des Skew-Algorithmus.

6.2 Sortierung von K^{12}

Sind alle Suffixe in die drei Restklassen aus 6.1.1 eingeteilt, wird das Suffix-Array SA^{12} für die Suffixe aus K^{12} wie folgt konstruiert. Man betrachtet für alle Suffixe aus K^{12} die ersten drei Symbole, sogenannte *Tripel*, also $s[i..i+2]$ für alle $i \in K^{12}$. Um Tripel für s_{n-2} und s_{n-1} zu erhalten, hängt man an das Ende dieser Suffixe entsprechend viele \$-Symbole an, d.h. $s[n-2..n] = s[n-2..n-1]\$$ bzw. $s[n-1..n+1] = s[n-1]\$\$$. Mit Hilfe von Radix-Sort werden alle Tripel sortiert und anschließend entsprechend dem Rang nummeriert, den sie nach der Sortierung einnehmen. Damit wird allen Tripeln ein Rang $r_i \in [1..\frac{2n}{3}]$ zugeordnet, denn durch die Klassifizierung über K_m wird die betrachtete Suffixmenge auf $|K^{12}| = (2/3)n$ reduziert. Wichtig dabei ist, dass ein Rang r_i für ein Tripel und dieses wiederum für ein Suffix bzw. dessen Index steht. Das bedeutet, dass für zwei Ränge r_i und r_j gilt:

$$r_i \leq r_j \Leftrightarrow s[i..i+2] \leq s[j..j+2]$$

In Abb. 6.1 ist der Vorgang der Einteilung in die beiden Mengen K_1 und K_2 und die anschließende Rangvergabe durch Radix-Sort für den String $s = acatgcattcag$ dargestellt. Falls genau $(2/3)n$ verschiedene Ränge vergeben werden, so ist dieser erste Sortierschritt mit der Konstruktion von SA^{12} und damit mit der vollständigen Sortierung aller Suffixe aus K^{12} abgeschlossen. Falls nicht, bildet man ein neues Eingabewort s^{12} wie folgt. Für alle Tripel $s[i..i+2]$ aus dem Teilwort $s[1 \ldots n)\$$ mit $i \bmod 3 = 1$ setzt man den Rang r_i ein, sodass sich das Wort $s^1 = [r_i : i \bmod 3 = 1]$ ergibt. Analog setzt man für alle Tripel $s[j..j+2]$ aus dem Teilwort $s[2 \ldots n)\$\$$ mit $j \bmod 3 = 2$ den Rang r_j ein, sodass sich das Wort $s^2 = [r_j : j \bmod 3 = 2]$ ergibt. Anschließend werden diese beiden Wörter konkateniert und ergeben:

$$s^{12} := s^1 \circ s^2 = [r_i : i \bmod 3 = 1] \circ [r_j : j \bmod 3 = 2]$$

Dabei ist $\circ$ der Konkatenationsoperator für Strings und es gilt $|s^{12}| = (2/3)n$. Für das Beispiel aus Abb. 6.1 ergibt sich $s^{12} = 3571 \circ 2364$ und für dieses neue Wort wird der Algorithmus rekursiv aufgerufen. In Abb. 6.2 ist für diesen rekursiven Schritt die erneute Tripelbildung und Rangvergabe abgebildet.

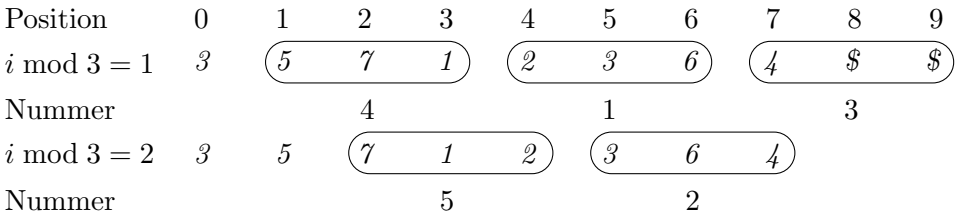


Abbildung 6.2: Beispiel für die rekursive Tripelbildung und deren Nummerierung anhand von $s^{12} = 35712364$ für Schritt 1 des Skew-Algorithmus.

i	$SA^{12}[i]$	$s^{12}_{SA^{12}[i]}$	Übersetzung von $s^{12}_{SA^{12}[i]}$
0	4	2364	$atgcattcag\$\$$
1	5	364	$cattcag\$\$$
2	7	4	$g\$\$$
3	1	5712364	$gcattcag\$$
4	2	712364	$ttcag\$$

Abbildung 6.3: Das Suffix-Array SA^{12} für den String $s^{12} = 35712364$. In der letzten Spalte ist die Rückübersetzung für die Tripelnummern aus s^{12} in Symboldarstellung von s angegeben.

Da nun alle Tripel einen eindeutigen Rang besitzen, ergibt sich das Suffix-Array SA^{12} aus Abb. 6.3, das die Sortierung aller Suffixe aus K^{12} für diesen rekursiven Schritt enthält. Wie erwähnt stehen die vergebenen Ränge für Tripel und diese wiederum für Suffixe aus s. Formal ausgedrückt bedeutet dies:

- $s^{12}[\frac{i-1}{3}..\frac{n-3}{3}]$ für $i \bmod 3 = 1$ repräsentiert Suffix $s_i = s[i..n)\$$
- $s^{12}[\frac{n+i-2}{3}..\frac{2n-3}{3}]$ für $i \bmod 3 = 2$ repräsentiert Suffix $s_i = [i..n)\$\$$

Beispielsweise sind:

- $s_4 = gcattcag$, wobei $s^{12}[\frac{4-1}{3}..\frac{12-3}{3}] = s^{12}[1..3] = 571$ und 571 für $gcattcag\$$ steht
- $s_8 = tcag$, wobei $s^{12}[\frac{12+8-2}{3}..\frac{24-3}{3}] = s^{12}[6..7] = 64$ und 64 für $tcag\$\$$ steht

In Abb. 6.3 ist diese Art der Rückübersetzung von Tripelnummern aus s^{12} in die Symboldarstellung von s angegeben. Diese Darstellung ist für $i \bmod 3 = 1$ nur bis zum ersten \$-Symbol angegeben, denn die nachfolgenden Symbole haben wegen $\$ < \lambda_j$ für alle $\lambda_j \in \Sigma \backslash \{\$\}$ mit $j \in [1..|\Sigma|]$ keinen Einfluss auf die Lexorder unter allen Suffixen aus K^{12}.

6.3 Sortierung von K_0

Der zweite Schritt des Skew-Algorithmus besteht darin, die Suffixe aus K_0 zu sortieren[2] unter Ausnutzung der eindeutigen Sortierung der Suffixe aus K^{12} in SA^{12}. Dazu betrachtet man alle Tupel $(s_i[0], s_{i+1})$ für $i \bmod 3 = 0$. Die relative Ordnung aller Suffixe s_{i+1} ist mit SA^{12} bekannt, deshalb genügt es zur Sortierung dieser Tupel $s_i[0]$ als Primär- und den lexikographischen Rang von s_{i+1} in SA^{12} als Sekundärschlüssel zu verwenden. Zunächst wird mit einem Bucketsort nach der ersten Komponente $s_i[0]$ aller Tupel sortiert und in ein Feld $SA^0[0..\frac{n}{3}]$ eingetragen. Dann wird, in Analogie zum Algorithmus von Manber und Myers, das Suffix-Array SA^{12} von links nach rechts durchlaufen und immer wenn $SA^{12}[i] \bmod 3 = 1$ gilt, wird Suffix $SA^{12}[i] - 1 \in K_0$ an den Anfang seines Buckets in SA^0 verschoben. Als Ergebnis erhält man das Suffix-Array SA^0 für alle Suffixe aus K_0.
Für das Beispielwort $s^{12} = 35712364$ sind also die drei Tupel $(3, s_1^{12})$,$(1, s_4^{12})$ und $(6, s_7^{12})$ zu sortieren und in Abb. 6.4 ist das Ergebnis SA^0 für diese drei Suffixe dargestellt. Dabei ist in der letzten Spalte wieder die Rückübersetzung in die Symboldarstellung von s angegeben.

i	$SA^0[i]$	$s^{12}_{SA^0[i]}$	Übersetzung von $s^{12}_{SA^0[i]}$
0	3	12364	$ag\$$
1	0	35712364	$catgcattcag\$\$$
2	6	64	$tcag\$\$$

Abbildung 6.4: Das Suffix-Array SA^0 für den String $s^{12} = 35712364$.

6.4 Verschmelzung von SA^0 und SA^{12}

Im letzten Schritt werden die beiden Suffix-Arrays SA^0 und SA^{12} miteinander verschmolzen, um das endgültige Suffix-Array SA zu erhalten. Dazu vergleicht man in jedem Schritt das kleinste Suffix aus SA^{12} mit dem kleinsten aus SA^0 und das kleinere von beiden wird aus seinem Array entfernt und an den Anfang von SA eingetragen. Dies führt man solange fort, bis beide Arrays leer sind. Für die Vergleiche braucht man die Ränge aller Suffixe aus SA^{12}, die über einen Suffixindex angefragt werden, sodass vorher noch das inverse Suffix-Array $\overline{SA}^{12}$ mit $\overline{SA}^{12}[SA^{12}[i]] = i$ berechnet wird[3]. Es sind in jedem Schritt zwei Suffixe s_j mit $j \bmod 3 = 0$ und s_i mit $i \bmod 3 \neq 0$ zu vergleichen, wodurch sich eine Fallunterscheidung ergibt.

- Gilt $i \bmod 3 = 1$, stellt man s_i als $(s_i[0], s_{i+1})$ und s_j als $(s_j[0], s_{j+1})$ dar. Da $i + 1 \bmod 3 = 2$ und $j + 1 \bmod 3 = 1$ gilt, lässt sich die relative Ordnung von s_{i+1} und s_{j+1} in SA^{12} mit Hilfe von $\overline{SA}^{12}$ abfragen. Man vergleicht zunächst $s_i[0]$ mit $s_j[0]$ und bei Gleichheit werden die eindeutigen Positionen $\overline{SA}^{12}[\frac{n+i-1}{3}]$ und $\overline{SA}^{12}[\frac{j}{3}]$ bestimmt. Spätestens der Vergleich dieser beiden Positionen liefert eine Entscheidung, da SA^{12} die eindeutige Sortierung aller Suffixe aus K^{12} enthält.

[2] Für die Symbole bzw. die Rangnummern der Indizes aus K_0 wird ein Feld $s^0[0..\frac{n}{3}]$ bereitgestellt.
[3] Der in dieser Arbeit sonst übliche Hochindex $^{-1}$ für SA^{-1} wird hier aus Darstellungsgründen weggelassen.

SA^{12}	$i \bmod 3$	Darstellung s_i^{12}	SA^0	Darstellung s_j^{12}	Vergleich	$SA[k]$	k
4	1	$(2,1)$	3	$(1,0)$	$2 > 1$	3	0
4	1	$(2,1)$	0	$(3,3)$	$2 < 3$	4	1
5	2	$(3,6,2)$	0	$(3,5,4)$	$3 = 3 \wedge 6 > 5$	0	2
5	2	$(3,6,2)$	6	$(6,4,\perp)$	$3 < 6$	5	3
7	1	$(4,\perp)$	6	$(6,2)$	$4 < 6$	7	4
1	1	$(5,4)$	6	$(6,2)$	$5 < 6$	1	5
2	2	$(7,1,0)$	6	$(6,4,\perp)$	$7 > 6$	6	6
2	2	$(7,1,0)$	-	-	-	2	7

Abbildung 6.5: Beispiel für das Verschmelzen von SA^0 und SA^{12} und der Erzeugung des Suffix-Arrays für den String $s^{12} = 35712364$.

- Gilt $i \bmod 3 = 2$, stellt man s_i als $(s_i[0], s_i[1], s_{i+2})$ und s_j als $(s_j[0], s_j[1], s_{j+2})$ dar. Da $i + 2 \bmod 3 = 1$ und $j + 2 \bmod 3 = 2$ gilt, ermittelt man analog die relative Ordnung von s_{i+2} und s_{j+2} in SA^{12} mit Hilfe von $\overline{SA}^{12}$. Zunächst vergleicht man die ersten beiden Symbole der beiden Suffixe und falls beide übereinstimmen, werden die Positionen $\overline{SA}^{12}[\frac{i+1}{3}]$ und $\overline{SA}^{12}[\frac{n+j}{3}]$ zur endgültigen Entscheidung herangezogen.

In Abb. 6.5 ist dieses Mischen von SA^{12} und SA^0 für das laufende Beispiel illustriert. Zur Vereinfachung ist in der jeweils letzten Komponente der 2-Tupel und 3-Tupel gleich der Wert von $\overline{SA}^{12}$ für den entsprechenden Suffixindex eingetragen. Die $\perp$-Einträge entstehen, falls versucht wird den Rang einer Position $\geq n$ abzufragen. Nach diesem Schritt ist das Suffix-Array für $s^{12} = 35712364$ erstellt und der Algorithmus kehrt aus der Rekursion zurück. Es folgen wieder die Sortierung von K_0 und das anschließende Verschmelzen von SA^0 und SA^{12}, diesmal für die ursprüngliche Eingabe $s = acatgcattcag$.

Die Laufzeit $T(n)$ des Skew-Algorithmus setzt sich aus den drei vorgestellten Schritten zusammen. Zuerst wird das Verfahren rekursiv auf $(2/3)n$ Suffixe angewendet. Bucketsort und das Durchlaufen von SA^{12} in Schritt 2 kosten jeweils $O(n)$. In der gleichen Klasse liegt Schritt 3, denn jeder Vergleich von zwei Suffixen erfordert maximal drei Einzelvergleiche und in jedem Schritt wird ein Suffix in SA festgelegt. Es ergibt sich die Rekursionsgleichung

$$T(n) = T(\lceil 2n/3 \rceil) + O(n)$$

die mit dem Mastertheorem [29] die Lösung $T(n) = O(n)$ besitzt.

Der Algorithmus legt während der Ausführung die Felder SA^{12}, SA^0, s^{12} und s^0 an, wobei $|SA^{12}| + |SA^0| = n$ und $|s^{12}| + |s^0| = n$ gilt. Alle Positionen dieser Felder benötigen 4 Bytes Speicher, sodass für jeden rekursiven Schritt $8n$-Bytes, bezogen auf die aktuelle Eingabelänge, gebraucht werden. Damit ergibt sich bei einem vollen rekursiven Abstieg ein Speicherverbrauch von etwa

$$8n \cdot \sum_{i=0}^{\infty} \left(\frac{2}{3}\right)^i = 8n \cdot \frac{1}{1 - \frac{2}{3}} = 24n\text{-Bytes}.$$

6.5 LCP-Berechnung mit Hilfe von LCP^{12}

Die Berechnung der LCP-Tabelle während des Ablaufs des Skew-Algorithmus basiert auf Satz 4.2.1 wie schon die Prefix-Doubling-Verfahren aus Kapitel 5 und wird von den Autoren zum Teil selbst angegeben[4]. Zunächst ist festzuhalten, dass sich bei Vorliegen von SA^{12} die dazugehörige LCP^{12}-Tabelle in $O(n)$ berechnen lässt. Nach Voraussetzung unterscheiden sich alle Suffixe aus SA^{12} spätestens an der dritten Stelle, da alle gebildeten Tripel einen eindeutigen Rang erhalten haben.
Seien $j = SA^{12}[i-1]$ und $k = SA^{12}[i]$ für $i \in [0..2n/3]$. Man durchläuft SA^{12} von links nach rechts und an jeder Position $i \in [1..2n/3]$ wird zunächst $s_j^{12}[0]$ mit $s_k^{12}[0]$ verglichen. Liegt Gleichheit vor, so muss noch $s_j^{12}[1]$ mit $s_k^{12}[1]$ verglichen werden. Für $i = 0$ setzt man nach Definition $LCP^{12}[0] = -1$. Dieses Vorgehen ist in Algorithmus 6.1 angegeben.

Algorithmus 6.1 : LCP^{12}-Berechnung im Skew-Algorithmus

Eingabe : s^{12}, SA^{12}
Ausgabe : LCP^{12}

```
1  LCP^12[0] ← −1
2  for i ∈ [1..2n/3] do
3      j ← SA^12[i − 1]
4      k ← SA^12[i]
5      if s_j^12[0] = s_k^12[0] then
6          if s_j^12[1] = s_k^12[1] then
7              LCP^12[i] ← 2
8          else
9              LCP^12[i] ← 1
10     else
11         LCP^12[i] ← 0
```

Liegt LCP^{12} vor, so lässt sich LCP für SA mit Hilfe der folgenden Fallunterscheidungen berechnen. Das Prinzip dabei ist, dass man die Indizes der beiden zu vergleichenden Suffixe soweit erhöht, bis sie in die Restklasse K^{12} fallen. Dann lässt sich der gesuchte LCP-Wert über die LCP^{12}-Tabelle bestimmen. Sei hierfür $j = SA[i-1]$ und $k = SA[i]$.

Fall 1: $j \bmod 3 \neq 0$ und $k \bmod 3 \neq 0$

Für diesen Fall liegen beide Suffixindizes in K^{12} und als Beispiel sei $j \bmod 3 = 1$ und $k \bmod 3 = 2$ gewählt. Seien $j' = (j-1)/3$ und $k' = (n+k-2)/3$ die entsprechenden Positionen in s^{12}. Da j und k adjazent in SA sind, müssen j' und k' auch adjazent in SA^{12} sein und somit ist $l = lcp^{12}(s_{j'}^{12}, s_{k'}^{12}) = LCP^{12}[\overline{SA}^{12}[k']]$. Da LCP^{12} LCP-Werte von Tripeln repräsentiert, multipliziert man l mit 3 und erhält damit den LCP-Wert übereinstimmender Tripel von j' und k'.

[4] Es sind nicht alle Fallunterscheidungen beschrieben.

Ab der ersten Position, an der sich die Tripelnummern unterscheiden, muss man noch maximal zwei Symbolvergleiche durchführen. Insgesamt erhält man damit

$$LCP[i] = 3l + lcp(s_{j+3l}, s_{k+3l})$$

wobei $lcp(s_{j+3l}, s_{k+3l}) \leq 2$. Für die anderen Unterfälle ändern sich nur die Positionen von j' und k' in s^{12}:

- $j \bmod 3 = 2 \wedge k \bmod 3 = 1 \rightarrow j' = (n + j - 2)/3 \wedge k' = (k - 1)/3$
- $j \bmod 3 = 1 \wedge k \bmod 3 = 1 \rightarrow j' = (j - 1)/3 \wedge k' = (k - 1)/3$
- $j \bmod 3 = 2 \wedge k \bmod 3 = 2 \rightarrow j' = (n + j - 2)/3 \wedge k' = (n + k - 2)/3$

Fall 2: ($j \bmod 3 = 0$ und $k \bmod 3 = 1$) oder ($j \bmod 3 = 1$ und $k \bmod 3 = 0$) oder ($j \bmod 3 = 0$ und $k \bmod 3 = 0$)

Hier ist mindestens ein Suffix nicht in K^{12} enthalten. Sei z.B. $j \bmod 3 = 0$ und $k \bmod 3 = 1$. In diesem Fall liegt j nicht in K^{12}, deshalb führt man zunächst den Einzelvergleich $s[j]$ mit $s[k]$ durch. Gilt $s[j] \neq s[k]$, so ist $LCP[i] = 0$. Andernfalls werden j und k inkrementiert, sodass beide Indizes in K^{12} liegen und somit ist $LCP[i] = 1 + lcp(s_{j+1}, s_{k+1})$. Dies entspricht Fall 1 und man setzt wieder $j' = ((j+1)-1)/3$ und $k' = (n+(k+1)-2)/3$ für die Positionen in s^{12}. Analog ist dann $lcp(s_{j+1}, s_{k+1}) = 3l + lcp(s_{(j+1)+3l}, s_{(k+1)+3l})$ mit $l = lcp^{12}(s^{12}_{j'}, s^{12}_{k'})$. Die Schwierigkeit hierbei ist, dass in diesem Fall $j+1$ und $k+1$ nicht benachbart in SA sein müssen und deshalb auch j' und k' nicht in SA^{12}. Man führt wegen 4.2.1 eine Range Minimum Query in LCP^{12} durch, die den Index des Minimums im Bereich $LCP^{12}[\overline{SA}^{12}[j'] + 1..\overline{SA}^{12}[k']]$ in konstanter Zeit liefert[3, 5]. Zusammenfassend erhält man somit für Fall 2

$$LCP[i] = \begin{cases} 0 & \text{falls} \quad s[j] \neq s[k] \\ 1 + 3l + lcp(s_{(j+1)+3l}, s_{(k+1)+3l}) & \text{sonst} \end{cases}$$

wobei $l = lcp^{12}(s^{12}_{j'}, s^{12}_{k'}) = LCP^{12}[rmq_{LCP^{12}}(\overline{SA}^{12}[j'] + 1, \overline{SA}^{12}[k'])]$ ist. Für die beiden anderen Unterfälle ändern sich j' und k' in s^{12} zu:

- $j \bmod 3 = 1 \wedge k \bmod 3 = 0 \rightarrow j' = (n + (j + 1) - 2)/3 \wedge k' = ((k + 1) - 1)/3$
- $j \bmod 3 = 0 \wedge k \bmod 3 = 0 \rightarrow j' = ((j + 1) - 1)/3 \wedge k' = ((k + 1) - 1)/3$

Fall 3: ($j \bmod 3 = 0$ und $k \bmod 3 = 2$) oder ($j \bmod 3 = 2$ und $k \bmod 3 = 0$)

Sei z.B. $j \bmod 3 = 0$ und $k \bmod 3 = 2$. Sehr ähnlich zu Fall 2 vergleicht man zunächst $s[j]$ mit $s[k]$. Gilt $s[j] \neq s[k]$, so ist $LCP[i] = 0$. Andernfalls muss noch der Einzelvergleich $s[j+1]$ mit $s[k+1]$ folgen, da $k + 1 \bmod 3 = 0$ ist und somit $k + 1 \notin K^{12}$.

Gilt dann $s[j+1] \neq s[k+1]$, so ist $LCP[i] = 1$. Ansonsten setzt man $j' = (n+(j+2)-2)/3$ und $k' = ((k+2)-1)/3$. Es ergibt sich ähnlich zu Fall 2

$$LCP[i] = \begin{cases} 0 & \text{falls} \quad s[j] \neq s[k] \\ 1 & \text{falls} \quad s[j] = s[k] \wedge s[j+1] \neq s[k+1] \\ 2 + 3l + lcp(s_{(j+2)+3l}, s_{(k+2)+3l}) & \text{sonst} \end{cases}$$

wobei $l = lcp^{12}(s^{12}_{j'}, s^{12}_{k'}) = LCP^{12}[rmq_{LCP^{12}}(\overline{SA}^{12}[j'] + 1, \overline{SA}^{12}[k'])]$. Für den anderen Unterfall ändern sich j' und k' in s^{12} zu:

- $j \bmod 3 = 2 \wedge k \bmod 3 = 0 \rightarrow j' = ((j+2)-1)/3 \wedge k' = (n+(k+2)-1)/3$

Der LCP-Wert wird dann mit diesen Positionen identisch zu Fall 2 berechnet.

Das komplette Berechnungsschema für die LCP-Tabelle ist in Algorithmus 6.2 angegeben. Die Funktion *get12Pos*(i_1, i_2) gibt für zwei Suffixindizes von s die entsprechenden Indizes j' und k' in s^{12} zurück und die Funktion *symbComp*(s_j, s_k) vergleicht die ersten beiden Symbole der übergebenen Suffixe und liefert einen Wert aus [0..2].

Durch die Verwendung von RMQ-Verfahren, die für eine Vorverarbeitung des Eingabefeldes $O(n)$-Zeit benötigen und eine Anfrage in konstanter Zeit beantworten, kann garantiert werden, dass die Laufzeit von Algorithmus 6.2 in $O(n)$ liegt und somit die worst-case Laufzeit des Skew-Algorithmus mit Berechnung der LCP-Tabelle auch $O(n)$ ist.
Der Speicherverbauch erhöht sich durch die Bereitstellung von LCP für jede Rekursionsstufe auf

$$24n + 4n \cdot \sum_{i=0}^{\infty} \left(\frac{2}{3}\right)^i = 24n + 4n \cdot \frac{1}{1-\frac{2}{3}} = 36n\text{-Bytes.}$$

Algorithmus 6.2 : LCP-Berechnung im Skew-Algorithmus.

Eingabe : Eingabe s, SA, SA^{12}, s^{12}, LCP^{12}
Ausgabe : LCP

```
1  LCP[0] = −1
2  Vorverarbeitung von LCP^12 für konstanten RMQ-Zugriff; benötigt O(n)
3  for i ∈ [1..n] do
4      j ← SA[i − 1]
5      k ← SA[i]
6      j_mod ← j mod 3
7      k_mod ← k mod 3
8      (j', k') ← get12Pos(k_mod, j_mod)
9      if Fall 1 then
10         l ← LCP^12[SA^12(k')]
11         LCP[i] ← 3l + symbComp(s_{j+3l}, s_{k+3l})
12     else if Fall 2 then
13         if s[j] ≠ s[k] then
14             LCP[i] ← 0
15         else
16             l ← LCP^12[rmq_{LCP^12}(SA^12[j'] + 1, SA^12[k'])]
17             LCP[i] ← 1 + 3l + symbComp(s_{j+1+3l}, s_{k+1+3l})
18     else if Fall 3 then
19         if s[j] ≠ s[k] then
20             LCP[i] ← 0
21         if s[j + 1] ≠ s[k + 1] then
22             LCP[i] ← 1
23         else
24             l ← LCP[rmq_{LCP^12}(SA^12[j'] + 1, SA^12[k'])]
25             LCP[i] ← 2 + 3l + symbComp(s_{j+2+3l}, s_{k+2+3l})
```

7 Induced Copying

In diesem Kapitel wird der Algorithmus von Puglisi und Maniscalco [24] vorgestellt, der gegenwärtig[28] zu den schnellsten und speicherplatzsparendsten SAKAs zählt. Dieser lässt sich in die sogenannte Klasse der Induced-Copying-Algorithmen einordnen, die sich dadurch auszeichnet, dass deren Algorithmen in einem ersten Schritt eine gewisse Suffixmenge bestimmen. Die ausgewählten Suffixe werden sortiert und anschließend in ihre endgültigen Positionen im Suffix-Array gebracht. Mit der Sortierung der Suffixe aus dieser Menge, im Folgenden Samplemenge genannt, lässt sich dann meist in linearer Zeit die Sortierung der Suffixe aus der Komplementärmenge herstellen. Dies hat eine gewisse Ähnlichkeit zur Vorgehensweise des Skew-Algorithmus aus Kapitel 6, der auch unter der Vorverarbeitung einer Suffixmenge die restliche Sortierung effizient erzielt.
Die Induced-Copying-Verfahren weisen eine worst-case Laufzeit auf, die nicht selten im Bereich $O(n^2 \log n)$ liegt und dennoch für nicht pathologisch konstruierte Eingaben überraschend schnell sind. Desweiteren benötigen viele Algorithmen aus dieser Klasse etwa $6n$-Bytes Speicherbedarf, eine Größenordnung die in der Literatur als *lightweight* bezeichnet wird. Die weitere Darstellung in diesem Kapitel orientiert sich an der Arbeit der beiden Autoren, wobei Details, die keinen Einfluss auf die parallele LCP-Berechnung haben, ausgelassen sind.

7.1 Bestimmung der Samplemenge

Eine zentrale Idee des Algorithmus von Puglisi und Maniscalco ist die Bestimmung einer speziellen Suffixmenge, deren Elemente mit Hilfe von TSQS sortiert werden und aufgrund von noch zu beschreibenden Eigenschaften sich auch in ihre endgültigen Positionen in SA platzieren lassen. Die Sortierung der Suffixe aus der Komplementärmenge geschieht dann unter Ausnutzung dieses ersten Schrittes und wird in den Abschnitten 7.3 und 7.4 vorgestellt. Die Literatur weist eine Evolution an Ideen zur Auswahl einer geeigneten Samplemenge aus, wobei die erste in [14] beschrieben ist. Ko und Aluru[20] zeigen in ihrer Arbeit, dass maximal $n/2$ viele Suffixe explizit zu sortieren sind, um damit die Ordnung der restlichen Suffixe effizient berechnen zu können. Dazu werden in 7.1.1 zwei Suffixmengen definiert, sodass jedem Suffix ein Typ zugeordnet werden kann, und wählt anschließend die kleinere der beiden Mengen zur Sortierung mit TSQS aus. Für das gesamte Kapitel sei \$ das letzte Symbol der Eingabe s und dieses das lexikographisch kleinste Symbol aus Σ.

Definition 7.1.1 *Sei $s \in \Sigma^n$ und s_i ein beliebiges Suffix von s mit $i \in [0..n-2]$. Es werden definiert:*

$$\begin{aligned} U &:= \{s_i \,|\, s_i < s_{i+1}\} \\ V &:= \{s_i \,|\, s_i > s_{i+1}\} \end{aligned}$$

Gilt $s_i \in U$, so hat s_i den Typ U. Gilt $s_i \in V$, so hat s_i den Typ V. Suffix $s_{n-1} = \$$ hat den Typ U und V.

Im Folgenden wird $|U| \leq |V|$ angenommen, da man den anderen Fall symmetrisch behandeln kann. Lemma 7.1.2 zeigt wie sich die Typzuteilung für jedes Suffix mit einem Durchlauf von s berechnen lässt.

Lemma 7.1.2 (Ko & Aluru) *Alle n Suffixe von $s \in \Sigma^n$ lassen sich in $O(n)$-Zeit entweder zu Typ U oder zu Typ V einteilen.*

Beweis: *Sei s_j ein beliebiges Suffix von s mit $j < n-1$. Gilt $s_j[0] < s_j[1]$, so ist s_j vom Typ U und der Aufwand ist $O(1)$. Gilt $s_j[0] > s_j[1]$, so ist s_j vom Typ V und der Aufwand ist $O(1)$. Gilt $s_j[0] = s_j[1]$, dann geht man in s weiter zur kleinsten Position $i > j$ mit $s_j[0] \neq s_i[0]$. Ist $s_j[0] < s_i[0]$, so sind alle Suffixe $s_j, s_{j+1}, \ldots, s_{i-1}$ vom Typ U. Andernfalls, wenn $s_j[0] > s_i[0]$, so sind alle Suffixe $s_j, s_{j+1}, \ldots, s_{i-1}$ vom Typ V. In beiden Fällen benötigt man linear viele Vergleiche in Bezug auf die Anzahl der Suffixe zwischen Position j und i. Damit lassen sich alle Suffixe mit einen Links-Rechts-Durchlauf von s in $O(n)$-Zeit einem Typ zuteilen.* □

Der String $s = edabdccdeedab\$$ soll als laufendes Beispiel für dieses Kapitel dienen und für s ist in Abb.7.1 die Typzuteilung dargestellt.

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13
s	e	d	a	b	d	c	c	d	e	e	d	a	b	\$
Typ	V	V	U	U	V	U	U	U	V	V	V	U	V	U/V

Abbildung 7.1: Zuteilung der Typen U und V für alle Suffixe von $s = edabdccdeedab\$$. Das letzte Suffix hat nach Definition beide Typen.

Die Samplemenge ergibt sich also für dieses Beispiel zu $U = \{s_2, s_3, s_5, s_6, s_7, s_{11}\}$. Lemma 7.1.3 stellt eine wichtige Eigenschaft fest, mit der es später möglich ist, in einem *SA*-Durchlauf die Sortierung aller Suffixe, die nicht zur Samplemenge gehören, herzustellen.

Lemma 7.1.3 (Ko & Aluru) *Sei $s_i \in U$ und $s_j \in V$. Dann gilt:*

$$s_i[0] = s_j[0] \Rightarrow s_i > s_j$$

Beweis: *Sei s_i ein Suffix vom Typ U und s_j ein Suffix vom Typ V, die beide mit dem Symbol $c \in \Sigma$ beginnen. Sei $s_i < s_j$ die gegenteilige Annahme zur Behauptung. Die beiden Suffixe lassen sich darstellen als $s_i = c\alpha c_1\beta$ und $s_j = c\alpha c_2\gamma$ mit $c_1 \neq c_2$ und α, β, γ als Teilwörter von s, die auch leer sein können.*

Fall 1: α enthält ein Symbol b mit $b \neq c$. Sei $c_3 \neq c$ das Symbol mit der kleinsten Position in α, das sich von c unterscheidet. Da s_i ein Suffix vom Typ U ist, muss $c_3 > c$ gelten. Analog, da Suffix s_j vom Typ V ist, muss $c_3 < c$ gelten, was zum Widerspruch führt.
Fall 2: α besteht ausschließlich aus dem Symbol c. Da s_i vom Typ U ist, muss $c_1 \geq c$ gelten und analog $c_2 \leq c$ für Suffix s_j. Somit ist $c_2 \leq c_1$. Nach Annahme ist aber $s_i < s_j$ und deshalb muss aufgrund der lexikographischen Ordnung $c_1 < c_2$ gelten. Auch dieser Fall führt zum Widerspruch und damit ist die Behauptung gezeigt. □

Puglisi und Maniscalco nehmen ein indiziertes Alphabet mit $|\Sigma| \in [1..255]$ an, sodass jedes Symbol ein Byte Speicher benötigt.

Für eine schnellere Sortierung und aufgrund der folgenden beschriebenen Verkleinerung der Samplemenge, werden jeweils zwei aufeinanderfolgende Symbole des Eingabeworts zu einem Symbol zusammengefasst. Im Folgenden wird ein Präfix $s_i[0..1]$ eines Suffixes von s mit Länge 2 als Zweiwort, kurz ZW, bezeichnet. Mori[27] führt die Samplingidee von Ko und Aluru noch einen Schritt weiter, sodass sich die Anzahl der explizit zu sortierenden Suffixe meist noch weiter reduziert. Dazu betrachtet man Abb. 7.1 und bemerkt, dass es bei Typ-U-Sequenzen genügt, jeweils nur das Suffix mit der größten Position in die Samplemenge aufzunehmen. Diese neue Samplemenge ist definiert durch:

$$S := \{s_i \mid s_i \in U \ \wedge \ s_{i+1} \in V\}$$

Offensichtlich ist $S \subseteq U$ und für das Beispielwort würden s_2, s_5 und s_6 nicht mehr ausgewählt werden. Damit ergibt sich die reduzierte Samplemenge zu $S = \{s_3, s_7, s_{11}\}$. Für das laufende Beispiel zeigt Abb. 7.2 diejenigen Suffixe, die zur Samplemenge S gehören.

Pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13
s	e	d	a	b	d	c	c	d	e	e	d	a	b	\$
Typ	V	V	U	U	V	U	U	U	V	V	V	U	V	U/V
S				$\in$				$\in$				$\in$		

Abbildung 7.2: Zugehörigkeit von Suffixen zur Samplemenge S für den String $s = edabdccdeedab\$$.

Die Anzahl der Suffixe aus der Samplemenge hat sich damit gegenüber Abb. 7.1 von 6 Suffixen aus U auf 3 aus S reduziert[1].
Für die meisten Eingaben kann $|S| < |U|$ angenommen werden und die Einführung von S basiert auf der Idee, dass $|\Sigma|^2 = 256^2 = 65536$ eine auf heutigen Architekturen effizient handhabbare Größe ist. Hierfür betrachtet man in Abb. 7.3 die $|\Sigma|$ vielen Buckets des später sortierten Suffix-Arrays von s, in dem alle Suffixe mit dem gleichen Symbol beginnen.

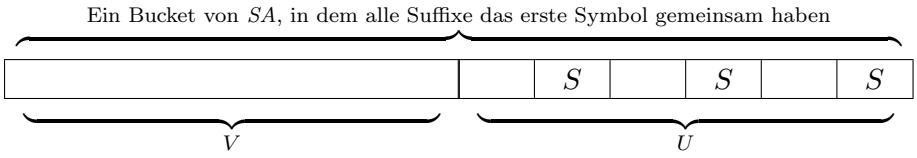


Abbildung 7.3: Bereiche eines Buckets von SA, basierend auf den Definitionen von V, U und S.

Mit Lemma 7.1.3 beginnt ein Bucket mit einem V-Bereich und anschließendem U-Bereich. Zu beachten ist, dass ein Bucket eventuell nur aus einem V- oder einem U-Bereich bestehen kann und dass in einem U- nicht zwingend ein S-Bereich existieren muss. Im U-Bereich liegen die eventuell vorhandenen S-Abschnitte, deren Suffixe mit Lemma 7.1.4 die Eigenschaft besitzen, dass deren ZW aus zwei verschiedenen Symbolen besteht.

[1] Für verschiedene prakische Eingaben geben die Autoren eine durchschnittliche Kardinalität von S mit etwa 30% von der Anzahl n aller Suffixe an.

Lemma 7.1.4 *Sei $s_i \in S$, $a, b \in \Sigma$ und $\alpha \in \Sigma^*$. Dann gilt:*

$$s_i = ab\alpha \Rightarrow a < b$$

Beweis: *Der Fall $a > b$ scheidet direkt nach Definition von S aus. Sei nun $a = b$. Weil $s_i \in S$ gilt $s_i < s_{i+1}$ und $s_{i+1} > s_{i+2}$. Sei c das Symbol mit der kleinsten Position von α für das $c \neq a$ gilt. Dann ist $c > a$, um der ersten Bedingung zu genügen. Andererseits ist damit $s_{i+1} < s_{i+2}$, was im Widerspruch zur zweiten Bedingung steht. Damit gilt $s_i \notin S$ im Widerspruch zur Voraussetzung.* □

Schließlich begründet Lemma 7.1.5, dass es nach der Sortierung der Suffixe aus S möglich ist, diese in ihre endgültigen Positionen in SA zu platzieren.

Lemma 7.1.5 *Seien $a, b \in \Sigma$, $a < b$ und $\alpha \in \Sigma^*$. Weiter seien*

- $X := \{s_i \mid s_i = ab\alpha\}$
- $X_S := \{s_i \mid s_i \in X \wedge s_i \in S\}$
- $X_{U \setminus S} := \{s_i \mid s_i \in X \wedge s_i \notin X_S\}$

Dann gilt für $s_j \in X_S$ und $s_k \in X_{U \setminus S}$: $s_j < s_k$

Beweis: *Da $s_j \in S$ und $s_k \notin S$ gilt $s_{j+1} > s_{j+2}$ und $s_{k+1} < s_{k+2}$. Nach Definition 7.1.1 gilt somit $s_{j+1} \in V$ und $s_{k+1} \in U$. Da nach Voraussetzung $s_j[0] = s_k[0]$ gilt, folgt mit Lemma 7.1.3 $s_{j+1} < s_{k+1}$ und damit die Behauptung.* □

Die Bestimmung der Samplemenge S und die Zählung aller vorkommenden ZW in der Eingabe s gibt Algorithmus 7.1 an. Die Variable *last_char* enthält das zuletzt gelesene Symbol und die Felder *SUFFIX_COUNT* und *SAMPLE_COUNT* zählen die Häufigkeiten aller vorkommenden ZW der Eingabe. In der ersten Schleife geht man in der Eingabe solange rückwärts, bis das aktuelle Symbol echt kleiner als das zuletzt gelesene ist. Damit werden Sequenzen von Typ-V Suffixen erkannt und bei Verlassen der Schleife liegt damit nach 7.1.4 ein Suffix aus S vor. In der zweiten Schleife werden analog Typ-U Sequenzen erkannt, die auf dieses Suffix aus S folgen, siehe dazu Abb. 7.1. Die Funktion $GetZW(i)$ liefert den eindeutigen ZW-Wert an Position i und wird durch $GetZW(i) := s[i+1]+256 \cdot s[i]$ berechnet. Mit dieser Kodierung lässt sich das Feld *SUFFIX_COUNT* von links nach rechts durchlaufen und man erhält die Häufigkeiten aller ZW in Lexorder. Die Laufzeit liegt offensichtlich in $O(n)$, da die Eingabe in beiden inneren Schleifen stets nach links gelesen wird.

7.2 Sortierung und Platzierung der Suffixe aus S

Nachdem die Samplemenge bestimmt ist, werden alle Suffixe aus S mit Hilfe von TSQS sortiert. Dabei wird sukzessive vorgegangen, sodass nur jeweils die Suffixe aus S in die Sortierung mit einbezogen werden, die das gleiche ZW teilen. Es werden beispielsweise S-Suffixe mit ZW ab vor denjenigen mit ZW cd sortiert.

Algorithmus 7.1 : Bestimmung der Samplemenge S und Häufigkeitszählung aller ZW

Eingabe : Eingabe s
Ausgabe : Samplemenge S und Häufigkeiten aller ZW

```
1  S = ∅
2  ∀k ∈ [0..|Σ|²] : SUFFIX_COUNT[k], SAMPLE_COUNT[k] ← 0
3  last_char ← −1
4  i ← n − 1
5  while i ≥ 0 do
6      while i ≥ 0 ∧ s[i] ≥ last_char do
7          Inkrementiere SUFFIX_COUNT[GetZW(i)]
8          last_char ← s[i]
9          Dekrementiere i
10     if i ≥ 0 then
11         Inkrementiere SUFFIX_COUNT[GetZW(i)]
12         Inkrementiere SAMPLE_COUNT[GetZW(i)]
13         Füge s_i zu S hinzu
14         last_char ← s[i]
15         Dekrementiere i
16         while i ≥ 0 ∧ s[i] ≤ last_char do
17             Inkrementiere SUFFIX_COUNT[GetZW(i)]
18             last_char ← s[i]
19             Dekrementiere i
```

Mit Hilfe des Feldes *SAMPLE_COUNT* aus Algorithmus 7.1 lassen sich diese ZW-Teilmengen aus S lexikographisch aufsteigend an TSQS übergeben. Alle Suffixe aus S werden bezüglich ihrer ZW in Lexorder in einem Feld $SP[0..|S|]$ gespeichert und haben vor der Sortierung bei einem identischen ZW einen identischen Schlüsselwert. Wie schon beim Algorithmus von Larsson und Sadakane im Abschnitt 5.2 werden auch hier für TSQS dynamische Schlüssel verwendet, um die Sortierung zu beschleunigen. Ist ein Suffix durch die Sortierung endgültig von allen anderen Suffixen differenziert worden, so kann ihm ein eindeutiger neuer Schlüsselwert zugewiesen werden und alle nachfolgenden Vergleiche können auf diesen neuen Wert zugreifen. Umgesetzt wird dies durch die Verwendung von Zählern für jedes ZW und immer dann, wenn ein Suffix seine Endposition erreicht, wird dieser Zähler inkrementiert.[2] Um die Korrektheit der Sortierung unter Verwendung von dynamischen Schlüsseln zu gewährleisten, muss immer gelten, dass, falls ein Suffix in seine Endposition i platziert wird, alle Suffixe an den Positionen $[0..i-1]$ auch schon platziert sind. Dies garantiert TSQS dadurch, dass immer die Partition mit Werten kleiner dem Pivotelement vor der mittleren Partition mit Werten gleich dem Pivotelement rekursiv sortiert wird, und analog die mittlere Partition vor der rechten Partition.

[2] Diese Zähler werden für jedes ZW geeignet initialisiert. Die Details spielen für die spätere LCP-Berechnung keine Rolle, sodass auf die Originalliteratur verwiesen sei.

Wie viele andere verwandte Algorithmen verwenden Puglisi und Maniscalco ab einer bestimmten niedrigen Schlüsselanzahl Insertion Sort, um eine bessere average-case Performance zu erreichen.

Im Anschluss an die Sortierung werden die Suffixe aus S in ihre korrekten Positionen in SA platziert. Dazu wird $SUFFIX\_COUNT$ aufsteigend durchlaufen und falls ein Eintrag an Position i größer 0 ist, werden zuerst $SAMPLE\_COUNT[i]$ viele Suffixe aus dem sortieren Feld SP aller S-Suffixe entnommen und in SA platziert. Anschließend wird die aktuelle SA-Position um den Wert $SUFFIX\_COUNT[i] - SAMPLE\_COUNT[i]$ erhöht und man fährt mit dem nächsten vorkommenden ZW fort. Die Korrektheit dieses Vorgehens folgt aus Lemma 7.1.3 und ist in Algorithmus 7.2 dargestellt.

Algorithmus 7.2 : Platzierung der Suffixe aus S in SA

Eingabe : Sortiertes Feld SP von Suffixen aus S
Ausgabe : Suffixe aus S befinden sich in endgültigen Positionen in SA

```
1 i ← 0
2 s ← 0
3 for j ∈ [0..|Σ|²] do
4     if SUFFIX_COUNT[j] > 0 then
5         for k ∈ [0..SAMPLE_COUNT[j]] do
6             SA[i] ← SP[s]
7             Inkrementiere i und s
8         Erhöhe i um SUFFIX_COUNT[j] − SAMPLE_COUNT[j]
```

Für das laufende Beispiel ergibt die Sortierung von S lexikographisch aufsteigend die Suffixindizes 11, 3 und 7. In Abb. 7.4 ist deren Platzierung in SA dargestellt, wobei die Klammern in der letzte Zeile schematisch die ZW-Buckets bzw. deren Grenzen angeben.

Pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13
SA	$\perp$	11	$\perp$	$\perp$	3	$\perp$	$\perp$	$\perp$	$\perp$	$\perp$	7	$\perp$	$\perp$	$\perp$
ZW	\$	$(ab$	)	$(b\$)$	(bd)	(cc)	(cd)	$(da$	)	(dc)	(de)	$(ed$	)	(ee)

Abbildung 7.4: Das Suffix-Array nach der Platzierung der Suffixe aus $S = \{s_{11}, s_3, s_7\}$.

Die Sortierung der Suffixe aus S mit TSQS stellt den größten Aufwand im gesamten Algorithmus dar, denn die Sortierung der restlichen Suffixe aus $U \backslash S$ und V kann nun in linearer Zeit geschehen und wird in den beiden folgenden Abschnitten beschrieben.

7.3 Sortierung der Suffixe aus $U \backslash S$

Sind alle Suffixe aus S sortiert und in ihre jeweilige Endposition in SA platziert worden, so lassen sich im nächsten Schritt mit einem Rechts-Links-Durchlauf, kurz RLD, von SA alle Suffixe aus $U \backslash S$ anordnen. Trifft man dabei auf ein Suffix $SA[i]$ an Position i, so betrachtet man den Typ von $s_{SA[i]-1}$.

Ist es vom Typ U, so wird $SA[i]-1$ an das aktuelle Ende des ZW-Buckets $s[SA[i]-1]s[SA[i]]$ gesetzt und diese Endposition anschließend dekrementiert. Lemma 7.3.1 stellt sicher, dass S das lexikographisch größte Suffix aus U enthält, sodass kein Suffix mit Typ U in diesem Durchlauf übergangen wird.

Lemma 7.3.1 *Sei $s_j \in S$ das lexikographisch größte Suffix aus S. Dann gilt:*

$$s_j > s_k \quad \textit{für alle} \quad s_k \in U \backslash \{s_j\}$$

Beweis: *Da $s_j \in S$ gilt $s_j < s_{j+1} > s_{j+2}$. Gäbe es ein Suffix $s_k \in U\backslash\{s_j\}$ mit $s_k > s_j$, so kann dieses nur der Bedingung $s_k < s_{k+1} < s_{k+2}$ genügen, denn wenn $s_{k+1} > s_{k+2}$, so wäre $s_k \in S$ und damit lexikographisch größer als s_j, was im Widerspruch zur Voraussetzung steht. Somit ist $s_{k+1} \in U$ und es muss weiter $s_{k+2} < s_{k+3}$ gelten, da ansonsten wieder mit $s_{k+2} > s_{k+3}$ ein Suffix aus S vorhanden wäre. Demnach liegt bis zur Position $n-2$ von s eine lexikographisch aufsteigende Sequenz von Suffixen vor. Es gilt dann aber $s_{n-3} < s_{n-2} > s_{n-1} = \$$ und somit ist $s_{n-3} \in S$ und damit das lexikographisch größte Suffix aus S, was erneut zum Widerspruch führt.* □

In Analogie an das noch folgende Lemma 7.4.1 von Ko & Aluru im nächsten Abschnitt stellt Lemma 7.3.2 sicher, dass während des RLD bei Erreichen eines Suffixes aus U, dieses schon in der korrekten Position in SA ist.

Lemma 7.3.2 *Wird im beschriebenen RLD durch SA ein Suffix $s_{SA[i]}$ erreicht, so ist $s_{SA[i]}$ bereits an seiner endgültigen Position in SA.*

Beweis: *Man zeigt die Behauptung durch eine Induktion über die betrachteten Positionen i in SA. Sei dazu j die Position desjenigen Suffixes, das nach Lemma 7.3.1 das lexikographisch größte aus der Menge U ist. Diese Position bildet den Induktionsanfang und es ist nach der beschriebenen Platzierung aller Suffixe aus S an seiner korrekten Position in SA. Seien nun alle Suffixe aus U im Bereich $[i..j]$ korrekt platziert. Zu zeigen ist nun, dass bei Erreichen von Position $i-1$ sich das Suffix $SA[i-1]$ bereits in der korrekten Position befindet. Dazu nimmt man das Gegenteil an, d.h. es existiert ein Suffix $s_{SA[k]}$ an Position k mit $k < i-1$, das in der endgültigen Sortierung an Position $i-1$ gehört. Folglich gilt $s_{SA[k]} > s_{SA[i-1]}$. Nach Voraussetzung sind alle Suffixe aus S bereits an ihren korrekten Positionen, sodass $s_{SA[k]}$ und $s_{SA[i-1]}$ aus $U\backslash S$ sind. Ein Suffix verlässt niemals das zugehörige Bucket, sodass die beiden Suffixe mit einem identischen Symbol c beginnen. Seien $s_{SA[i-1]} = c\alpha$ und $s_{SA[k]} = c\beta$. Da $s_{SA[k]}$ vom Typ U ist gilt $\beta > s_{SA[k]}$ und weil nach Annahme $s_{SA[k]} > s_{SA[i+1]}$ ist, gilt $\beta > \alpha$. Da die korrekte Position von $s_{SA[k]}$ an Stelle $i-1$ ist, muss β Bereich $[i..j]$ vorkommen und da $\beta > \alpha$ ist, muss $s_{SA[k]}$ an das aktuelle Bucketende verschoben worden sein, bevor dies mit $s_{SA[i-1]}$ geschehen ist. Deshalb kann $s_{SA[k]}$ nicht an einer Position, die kleiner als die Position von $s_{SA[i-1]}$ ist, auftreten. Dies ist ein Widerspruch zur Annahme $k < i-1$.* □

Nach dem beschriebenen RLD sind alle Suffixe aus U in ihren korrekten Positionen in SA und in Abb. 7.5 ist dieser Vorgang für das laufende Beispiel dargestellt.

Schritt	Suffix-Typ	Aktion
$SA[13] = \bot$	-	-
$SA[12] = \bot$	-	-
$SA[11] = \bot$	-	-
$SA[10] = 7$	$s_6 \in U$	Platziere s_6 an Position 6
$SA[9] = \bot$	-	-
$SA[8] = \bot$	-	-
$SA[7] = \bot$	-	-
$SA[6] = 6$	$s_5 \in U$	Platziere s_5 an Position 5
$SA[5] = 5$	$s_4 \notin U$	-
$SA[4] = 3$	$s_2 \in U$	Platziere s_2 an Position 2
$SA[3] = \bot$	-	-
$SA[2] = 2$	$s_1 \notin U$	-
$SA[1] = 11$	$s_{10} \notin U$	-
$SA[0] = \bot$	-	-

Pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13
SA	$\bot$	11	$\bot$	$\bot$	3	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	7	$\bot$	$\bot$	$\bot$
	$\bot$	11	$\bot$	$\bot$	3	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	7	$\bot$	$\bot$	$\bot$
	$\bot$	11	$\bot$	$\bot$	3	$\bot$	$\bot$	$\bot$	$\bot$	$\bot$	7	$\bot$	$\bot$	$\bot$
	$\bot$	11	$\bot$	$\bot$	3	$\bot$	6	$\bot$	$\bot$	$\bot$	7	$\bot$	$\bot$	$\bot$
	$\bot$	11	$\bot$	$\bot$	3	$\bot$	6	$\bot$	$\bot$	$\bot$	7	$\bot$	$\bot$	$\bot$
	$\bot$	11	$\bot$	$\bot$	3	$\bot$	6	$\bot$	$\bot$	$\bot$	7	$\bot$	$\bot$	$\bot$
	$\bot$	11	$\bot$	$\bot$	3	$\bot$	6	$\bot$	$\bot$	$\bot$	7	$\bot$	$\bot$	$\bot$
	$\bot$	11	$\bot$	$\bot$	3	5	6	$\bot$	$\bot$	$\bot$	7	$\bot$	$\bot$	$\bot$
	$\bot$	11	$\bot$	$\bot$	3	5	6	$\bot$	$\bot$	$\bot$	7	$\bot$	$\bot$	$\bot$
	$\bot$	11	2	$\bot$	3	5	6	$\bot$	$\bot$	$\bot$	7	$\bot$	$\bot$	$\bot$
	$\bot$	11	2	$\bot$	3	5	6	$\bot$	$\bot$	$\bot$	7	$\bot$	$\bot$	$\bot$
	$\bot$	11	2	$\bot$	3	5	6	$\bot$	$\bot$	$\bot$	7	$\bot$	$\bot$	$\bot$
	$\bot$	11	2	$\bot$	3	5	6	$\bot$	$\bot$	$\bot$	7	$\bot$	$\bot$	$\bot$
	$\bot$	11	2	$\bot$	3	5	6	$\bot$	$\bot$	$\bot$	7	$\bot$	$\bot$	$\bot$
ZW	\$	$(ab$	$)$	$(b\$)$	(bd)	(cc)	(cd)	$(da$	$)$	(dc)	(de)	$(ed$	$)$	(ee)

Abbildung 7.5: Sortierung der Suffixe aus $U \setminus S$. Jeweils grau hinterlegt ist die aktuell betrachtete Position und die eventuelle Platzierung des Vorgängersuffix, falls dieses aus $U \setminus S$ ist.

7.4 Sortierung der Suffixe aus V

Nach der Sortierung aller Suffixe aus U, lassen sich die Suffixe aus V in einem Links-Rechts-Durchlauf, kurz LRD, von SA sortieren. Dies entspricht der Vorgehensweise aus Abschnitt 7.3 mit entgegengesetzter Laufrichtung. Trifft man dabei auf ein Suffix $SA[i]$, so betrachtet man den Typ von $s_{SA[i]-1}$. Ist es vom Typ V, so verschiebt man $SA[i]-1$ an den aktuellen Anfang des ZW-Buckets $s[SA[i]-1]s[SA[i]]$ und inkrementiert die Anfangsposition. Am Ende dieses Schritts liegt das komplett sortierte Suffix-Array vor und Lemma 7.4.1 stellt die Korrektheit dieser Vorgehensweise her. Initial wird Suffix $s_{n-1} = \$$ an Position 0 gesetzt, da es per Definition immer das kleinste Suffix von s ist.

Lemma 7.4.1 (Ko & Aluru) *Wird im beschriebenen LRD durch SA ein Suffix $s_{SA[i]}$ erreicht, so ist $s_{SA[i]}$ bereits an seiner endgültigen Position in SA.*

Beweis: *Man zeigt die Behauptung durch eine Induktion über die betrachteten Positionen i in SA. Für $i = 0$ ist $s_{n-1} = \$$ bereits an der korrekten Position $SA[0]$. Seien $SA[0], SA[1], \ldots, SA[i]$ die ersten i Suffixe an ihrer endgültigen Position. Zu zeigen ist nun, dass bei Erreichen von $SA[i+1]$ das Suffix an dieser Position sich bereits in der korrekten Position befindet. Dazu nimmt man das Gegenteil an, d.h. es existiert ein Suffix $s_{SA[k]}$ an Position k mit $k > i+1$, das in der endgültigen Sortierung an Position $i+1$ gehört. Folglich gilt $s_{SA[k]} < s_{SA[i+1]}$. Nach Voraussetzung sind alle Typ-U Suffixe bereits in ihren korrekten Positionen, sodass $s_{SA[k]}$ und $s_{SA[i+1]}$ Typ-V Suffixe sein müssen. Ein Suffix verlässt niemals das zugehörige Bucket, sodass die beiden Suffixe mit einem identischen Symbol c beginnen. Seien $s_{SA[i+1]} = c\alpha$ und $s_{SA[k]} = c\beta$. Da $s_{SA[k]}$ vom Typ V ist, gilt $\beta < s_{SA[k]}$ und weil nach Annahme $s_{SA[k]} < s_{SA[i+1]}$ ist, gilt $\beta < \alpha$. Da die korrekte Position von $s_{SA[k]}$ an Stelle $i+1$ ist, muss β in der Folge $SA[0], SA[1], \ldots, SA[i]$ vorkommen und da $\beta < \alpha$ ist, muss $s_{SA[k]}$ an den aktuellen Bucketanfang verschoben worden sein, bevor dies mit $s_{SA[i+1]}$ geschehen ist. Deshalb kann $s_{SA[k]}$ nicht an einer Position die größer als die Position von $s_{SA[i+1]}$ ist auftreten. Dies ist ein Widerspruch zur Annahme $k > i+1$.* □

In Abb. 7.6 ist der LRD für das laufende Beispiel schematisch dargestellt.
Sowohl der RLD als auch der LRD liegen in $O(n)$, da an jeder betrachteten Position ein konstanter Aufwand entsteht. Die vorstehenden Beschreibungen setzten $|U| \leq |V|$ voraus. Im anderen Fall kann man einfach symmetrisch vorgehen oder man kehrt die Alphabetordnung zu Beginn des Algorithmus um. Dies erfordert zum einen eine Umkodierung der Eingabe und zum anderen die Umkehrung der Sortierung in SA.

Nachdem nun der Ablauf des Algorithmus von Puglisi und Maniscalco vorgestellt wurde, soll die Beschreibung einer parallelen Berechnung der LCP-Tabelle erfolgen. Die Vorgehensweise wird adaptiv sein, d.h. je mehr Struktur und Sortierinformation der Basis-Algorithmus generiert, desto mehr LCP-Werte können berechnet werden. Bevor die Sortierung der Suffixe aus S beginnt, lassen sich über die ZW-Häufigkeiten des Feldes $SUFFIX\_COUNT$ die LCP-Werte an den ZW-Buckets berechnen, wie in Abschnitt 7.5 beschrieben wird. Anschließend setzt man in 7.6 direkt an TSQS auf, um durch Partitionierungsinformationen die LCP-Werte für adjazente Suffixe aus S zu bestimmen. Schließlich wird in den Abschnitten 7.7 und 7.8 beschrieben, wie man die LCP-Berechnung für Suffixe aus U und V in den RLD bzw. LRD einbetten kann.

Schritt	Suffix-Typ	Aktion
$SA[0] = 13$	$s_{12} \in V$	Platziere s_{12} an Position 3
$SA[1] = 11$	$s_{10} \in V$	Platziere s_{10} an Position 7
$SA[2] = 2$	$s_1 \in V$	Platziere s_1 an Position 8
$SA[3] = 12$	$s_{11} \notin V$	-
$SA[4] = 3$	$s_2 \notin V$	-
$SA[5] = 5$	$s_4 \in V$	Platziere s_4 an Position 9
$SA[6] = 6$	$s_5 \notin V$	-
$SA[7] = 10$	$s_9 \in V$	Platziere s_9 an Position 11
$SA[8] = 1$	$s_0 \in V$	Platziere s_0 an Position 12
$SA[9] = 4$	$s_3 \notin V$	-
$SA[10] = 7$	$s_6 \notin V$	-
$SA[11] = 9$	$s_8 \in V$	Platziere s_8 an Position 13
$SA[12] = 0$	-	-
$SA[13] = 8$	$s_7 \notin V$	-

Pos	0	1	2	3	4	5	6	7	8	9	10	11	12	13
SA	13	11	2	12	3	5	6	⊥	⊥	⊥	7	⊥	⊥	⊥
	13	11	2	12	3	5	6	10	⊥	⊥	7	⊥	⊥	⊥
	13	11	2	12	3	5	6	10	1	⊥	7	⊥	⊥	⊥
	13	11	2	12	3	5	6	10	1	⊥	7	⊥	⊥	⊥
	13	11	2	12	3	5	6	10	1	⊥	7	⊥	⊥	⊥
	13	11	2	12	3	5	6	10	1	4	7	⊥	⊥	⊥
	13	11	2	12	3	5	6	10	1	⊥	7	⊥	⊥	⊥
	13	11	2	12	3	5	6	10	1	4	7	9	⊥	⊥
	13	11	2	12	3	5	6	10	1	4	7	9	0	⊥
	13	11	2	12	3	5	6	10	1	4	7	9	0	⊥
	13	11	2	12	3	5	6	10	1	4	7	9	0	⊥
	13	11	2	12	3	5	6	10	1	4	7	9	0	8
	13	11	2	12	3	5	6	10	1	4	7	9	0	8
	13	11	2	12	3	5	6	10	1	4	7	9	0	8
ZW	\$	(*ab*	)	(*b*\$)	(*bd*)	(*cc*)	(*cd*)	(*da*	)	(*dc*)	(*de*)	(*ed*	)	(*ee*)

Abbildung 7.6: Sortierung der Suffixe aus V. Jeweils grau hinterlegt ist die aktuell betrachtete Position und die eventuelle Platzierung des Vorgängersuffix, falls dieses aus V ist.

7.5 LCP-Berechnung an ZW-Grenzen

Der erste Schritt zur Erstellung der LCP-Tabelle beginnt mit der Ermittlung der LCP-Werte an ZW-Grenzen, die offensichtlich nur in [0..1] liegen können. Wie beschrieben ermittelt Algorithmus 7.1 in einem ersten Schritt die Häufigkeiten aller ZW, um damit die entsprechenden Bucketgrenzen festlegen zu können. Sind diese bekannt, so wird *SA* in die Bereiche aus Abb. 7.3 partitioniert, um die spätere Sortierung und Platzierung der Suffixe aus S vornehmen zu können. Während diesem Partitionierungsvorgang lassen sich die LCP-Werte an den ZW-Grenzen leicht ermitteln, denn es genügt jeweils ein Symbolvergleich zwischen zwei aufeinanderfolgenden ZW. Es werden alle $|\Sigma|^2$ verschiedene Möglichkeiten der ZW durchlaufen und falls an Position i mit
$SUFFIX\_COUNT[i] > 0$ ein positiver Wert vorliegt, lässt sich der entsprechende LCP-Wert an der *SA*-Position k bestimmen. Der Wert von k ist dabei die Summe aller bis zu diesem Zeitpunkt gefundenen ZW-Häufigkeiten. Man muss auf das zuletzt vorgekommene ZW zurückgreifen, um dessen erstes Symbol mit dem aktuellen ersten Symbol des gerade betrachteten ZW vergleichen zu können. Gilt Identität, so ist der LCP-Wert 1, ansonsten 0. Algorithmus 7.3 zeigt dieses Verfahren, wobei die Funktion $GetFirstZWChar(i)$ für einen ZW-Kodierungswert i das erste Symbol des zugehörigen ZW liefert.

Algorithmus 7.3 : Berechnung der LCP-Werte an ZW-Grenzen

Eingabe : ZW-Häufigkeit *SUFFIX_COUNT*, *LCP*
Ausgabe : *LCP* enthält alle vorkommenden Werte aus [0..1]

```
1  currPos ← 1
2  currChar, lastChar ← ⊥
3  for i ∈ [0..|Σ|²] do
4      if SUFFIX_COUNT[i] > 0 then
5          currChar ← GetFirstZWChar(i)
6          if lastChar = currChar then
7              LCP[currPos] ← 1
8          else
9              LCP[currPos] ← 0
10         lastChar ← currChar
11         Erhöhe currPos um SUFFIX_COUNT[i]
```

7.6 LCP-Berechnung während der Ausführung von TSQS

In Abschnitt 7.2 ist dargestellt wie die Suffixe aus S sortiert werden und aus Abschnitt 3.3 ist bekannt wie TSQS die Eingabe partitioniert. Diese Partitionsinformation lässt sich nutzen, um LCP-Werte ganz oder teilweise zu berechnen. Wie beschrieben werden zur Sortierung nur Suffixe aus S mit gleichem ZW in aufsteigender lexikographischer Reihenfolge herangezogen. Mit Lemma 7.1.4 folgt unmittelbar, dass zwei Suffixe aus verschiedenen S-Bereichen in *SA* einen LCP-Wert aus [0..1] besitzen.

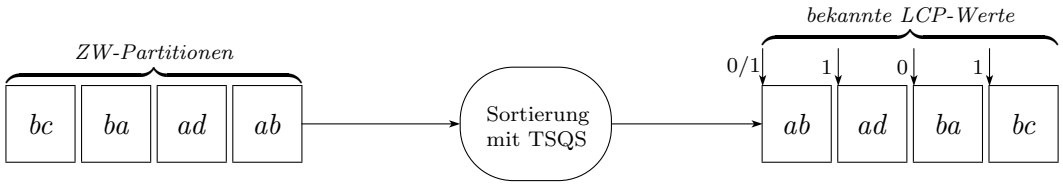


Abbildung 7.7: Die ZW-Partitionen für S-Suffixe werden lexikographisch aufsteigend TSQS zugeführt und sortiert. An den Startpositionen aller Partitionen sind die LCP-Werte schon bekannt.

Im vorherigen Abschnitt wurden alle vorkommenden LCP-Werte aus [0..1] schon berechnet, d.h. die Partition der aktuell zu sortierenden Suffixmenge besitzt an ihrer Startposition schon den korrekten LCP-Wert, unabhängig davon welches konkrete Suffix an dieser Position ist. In Abb. 7.7 ist für ein Beispiel der stufenweise Vorgang der Sortierung dargestellt. Die 0/1-Beschriftung an der ersten sortierten ab-Partition soll verdeutlichen, dass dieser LCP-Wert davon abhängt, ob noch ein kleineres ZW, also z.B. aa, existiert. Wichtig ist nur, dass auch dieser Wert durch die Berechnung an ZW-Grenzen schon bekannt ist und diese Position nicht mehr berücksichtigt werden muss.
Die eigentliche Sortierung einer ZW-Partition beginnt mit allen ZW-Schlüsseln an den Positionen 2 aller Suffixe. Es entsteht die dreiteilige Zerlegung der Eingabe, wobei im Weiteren mit $ZW_<$ die linke, mit $ZW_=$ die mittlere und mit $ZW_>$ die rechte Partition bezeichnet wird. An der Startposition von $ZW_<$ ist nichts weiter zu berechnen, wie vorstehend beschrieben. An der Startposition i von $ZW_=$ gilt $LCP[i] \in [2..3]$, denn durch die Zuführung von ZW-Partitionen stimmen alle Suffixe in den ersten beiden Positionen überein und alle Suffixe von $ZW_=$ wurden von allen Suffixen aus $ZW_<$ durch den Schlüssel an Position 2 getrennt. Die gleiche Argumentation trifft auf die Startposition von $ZW_>$ zu. Da immer ZW-Schlüssel verwendet werden, muss später das Symbol an Position 2 des ersten Suffix aus $ZW_=$ mit dem Symbol an Position 2 des letzten Suffix aus $ZW_<$ verglichen werden und analog für die beiden entsprechenden Suffixe aus $ZW_=$ und $ZW_>$. Zu beachten ist der Fall, wenn $ZW_<$ leer ist. Dann ist die Startposition von $ZW_=$ mit der von $ZW_<$ identisch und der LCP-Wert ist damit schon bekannt. Ebenso muss getestet werden, ob $ZW_>$ leer ist, denn dann kann auf diesen S-Bereich ein V-Bereich in SA folgen und der LCP-Wert an solch einer Position ist noch unklar. Dieser Fall wird in Abschnitt 7.8 besprochen und erfordert einen expliziten Vergleich der beiden beteiligten Suffixe. Für den allgemeinen Fall einer festen Rekursionstiefe von TSQS sei m die aktuell vorliegende Präfixlänge, die zur Auswahl der Sortierschlüssel dient. Da immer ZW-Schlüssel betrachtet werden, ist m ein Vielfaches von 2. Sind dann $ZW_<$ und $ZW_>$ nicht leer, so setzt man an den Startpositionen von $ZW_=$ und $ZW_>$ jeweils einen LCP-Wert von m. In den späteren Rekursionsaufrufen darf dann ein solch gesetzter LCP-Wert nicht mehr überschrieben werden, sodass noch ein Vergleich auf einen definierten LCP-Wert erfolgen muss. Abb. 7.8 zeigt eine aufgeteilte ZW-Partition für eine feste Rekursionstiefe mit betrachteter Präfixlänge m.
In Algorithmus 7.4 ist diese Vorgehensweise in Pseudocode angegeben, wobei das Feld $LCP_S[0..|S|]$ alle LCP-Werte für adjazente Suffixe aus S des Sortierfeldes SP übernimmt.

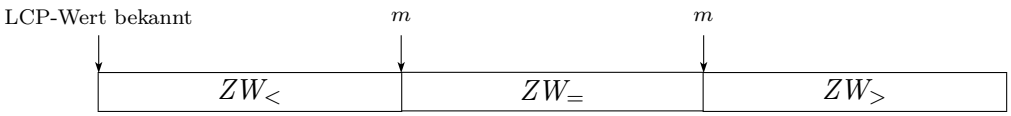


Abbildung 7.8: Festlegung der LCP-Werte an den Startpositionen von $ZW_=$ und $ZW_>$, falls $ZW_<$ und $ZW_>$ nicht leer sind.

Dieses wird dann später bei der Platzierung der Suffixe aus S genutzt, um die entsprechenden Werte in LCP eintragen zu können.

Algorithmus 7.4 : Setzen der LCP-Werte während TSQS

Eingabe : Zu sortierende ZW-Partition, $LCP_S[0..|S|]$
Ausgabe : Partielle LCP-Werte in LCP_S für alle Suffixe der ZW-Partition

```
1  TSQS sortiert ZW-Partition mit Schlüsseln an aktueller Präfixlänge m
2  |ZW_<| ist Anzahl Suffixe aus ZW_< Partition
3  |ZW_=| ist Anzahl Suffixe aus ZW_= Partition
4  |ZW_>| ist Anzahl Suffixe aus ZW_> Partition
5  i ← Startposition des zu sortierenden Feldes in LCP_S
6  if |ZW_<| > 0 then
7      if LCP_S[i + |ZW_<|] = ⊥ then
8          LCP_S[i + |ZW_<|] ← m
9  if |ZW_>| > 0 then
10     if LCP_S[i + |ZW_<| + |ZW_=|] = ⊥ then
11         LCP_S[i + |ZW_<| + |ZW_=|] ← m
12 Rekursive Aufrufe von TSQS mit Schlüsseln an Präfixposition m + 2
```

Es ist aus Abschnitt 7.2 bekannt, wie TSQS dynamische Schlüssel für eine schnellere Sortierung verwendet. Wird ein Suffix aus der ZW-Partition von allen anderen Suffixen differenziert, so wird diesem ein eindeutiger Schlüssel zugewiesen. Da nur Suffixe aus S sortiert werden, bleiben während der gesamten Ausführung von TSQS die ZW-Schlüssel von Suffixen aus V und $U \backslash S$ unverändert. Unterschreitet die Kardinalität einer ZW-Partition einen bestimmten Wert, dann übergibt TSQS diesen Bereich an Insertion Sort (IS), das für kleine Eingaben die average-case Performance verbessern kann. Dies hat zur Folge, dass dynamische Schlüssel nur in dieser Routine gesetzt werden bzw. dass ein Suffix aus S stets bei Insertion Sort seine endgültige Sortierposition einnimmt. Hat IS die Partition sortiert, werden Bereiche mit gleichen Schlüsselwerten wieder IS zugeführt, mit einer um 2 erhöhten Präfixlänge. Dies geschieht in aufsteigender Weise, sodass auch hier die Korrektheit der dynamischen Schlüsselanwendung mit der darüberliegenden TSQS-Ausführung gewährleistet ist. Wird ein Suffix endgültig platziert, so hat man nun die Möglichkeit, auf das zuletzt gesetzte Suffix zurückzugreifen, außer es ist das erste sortierte Suffix. Sei P_1 ein Bereich der an IS übergeben wurde und m wieder die Position der verwendeten ZW-Schlüssel.

Haben alle Suffixe aus P_1 den gleichen ZW-Schlüssel an Position m, dann wird IS mit Bereich P_1 und $m+2$ wieder aufgerufen. An einer bestimmten Position $m+2a$, $a \in \mathbb{N}$, wird P_1 in mindestens zwei Bereiche $P_2 \dots P_n$ mit jeweils identischen Schlüsseln aufgeteilt. Diese Bereiche werden aufsteigend auf einen Stapel gelegt und sukzessive wieder IS zugeführt. Analog zu TSQS lässt sich an den Bereichsgrenzen ein LCP-Wert von $m + 2a$ festlegen. Dieser Wert kann zu klein sein, weil dynamische Schlüssel verwendet werden. Zu Veranschaulichung sei s_i das zuletzt gesetzte Suffix an Position k in SP und s_j das aktuell platzierte Suffix an Position $k+1$. IS habe die beiden Suffixe durch eine Schlüsselbetrachtung an Position $m + 2a$ voneinander differenziert. Gehört einer der beiden Schlüssel $GetZW(i + m + 2a)$ und $GetZW(j + m + 2a)$ zu Suffixen aus V oder $U\backslash S$, so ist $lcp(s_i, s_j) = LCP_S[k+1] \in [m+2a, m+2a+1]$. Falls aber beide Schlüssel zu Suffixen aus S gehören, die beide schon ihre endgültige Sortierposition eingenommen haben, dann kann $LCP_S[k+1] = m+2a$ kleiner sein als der tatsächliche Wert. Dies ist dann der Fall, wenn beide Schlüssel zu Suffixen mit gleichem ZW gehören. Dann sind s_i und s_j durch diese beiden unterschiedlichen Schlüssel getrennt worden, obwohl $s_{i+m+2a}[0..1] = s_{j+m+2a}[0..1]$ gilt. Genau aus diesem Grund wird es im nächsten Abschnitt nötig sein, einen expliziten Suffixvergleich von s_i und s_j ab Position $m+2a$ durchzuführen. Um diese Art von teuren Vergleichen gering zu halten, wird in Abschnitt 7.9 durch eine weitere Technik versucht, den korrekten LCP-Wert auf andere Weise zu berechnen.

7.7 LCP-Werte für adjazente Suffixe aus $U\backslash S$

Für die Beschreibung der Berechnung von LCP-Werten für adjazente Suffixe aus $U\backslash S$ sei im Weiteren vorausgesetzt, dass alle LCP-Werte für Suffixe aus S und an ZW-Grenzen schon bekannt sind. Dann lässt sich die Berechnung der LCP-Werte für Suffixe aus $U\backslash S$ in den RLD aus Abschnitt 7.3 einbetten. Sei dazu $SA[i]$ das aktuell betrachtete Suffix. Gilt nun $s_{SA[i]-1} \in U$, so wird dieses Suffix in seinem Bucket an die aktuell letzte Position verschoben. Für die LCP-Berechnung bestimmt man für jedes verschobene und damit endgültig gesetzte Suffix den Eintrag in SA^{-1}. Sei $p = SA^{-1}[SA[i] - 1]$ die Position des verschobenen Suffixes und falls an Position $p + 1$ in einem früheren Schritt schon ein Suffix aus U verschoben wurde, so ist der LCP-Wert an $p + 1$ noch unbestimmt. Für diesen Fall sei $j = SA[SA^{-1}[SA[i] - 1]] = SA[p]$ das aktuell verschobene Suffix und $k = SA[SA^{-1}[SA[i]-1]+1] = SA[p+1]$ der Nachfolger von s_j in SA. Da s_k und s_j im gleichen Bucket liegen, gilt zunächst $lcp(s_k, s_j) = LCP[p + 1] \geq 1$. Es ist also noch $lcp(s_{k+1}, s_{j+1})$ zu bestimmen. Im RLD ist $s_{SA[i]} = s_{j+1}$ das aktuell betrachtete Suffix an Position i. Es gilt $s_{k+1} > s_{j+1}$ wegen $s_k[0] = s_j[0]$ und $s_k > s_j$. Die Position q von Suffix s_{k+1} bestimmt sich zu $q = SA^{-1}[SA[p + 1] + 1]$. Mit Hilfe von Satz 4.2.1 gilt damit für den LCP-Wert an Position $p + 1$:

$$LCP[p + 1] = 1 + LCP[rmq_{LCP}(i + 1, q)]$$

Dabei gibt die Funktion $rmq_{LCP}(i + 1, q)$ den größten Index des Minimums der LCP-Werte von Position $i + 1$ bis q zurück. Abbildung 7.9 veranschaulicht die vorstehenden Beziehungen.

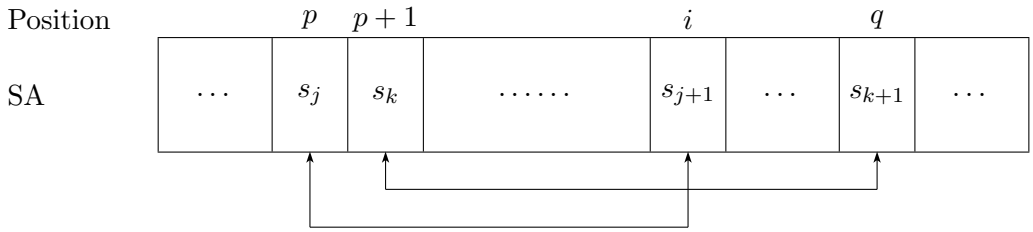


Abbildung 7.9: LCP-Berechnung während des RLD. Der Algorithmus befindet sich an Position i und bestimmt an Position $p+1$ den LCP-Wert der beiden Suffixe s_j und s_k unter Hilfe von $lcp(s_{j+1}, s_{k+1})$.

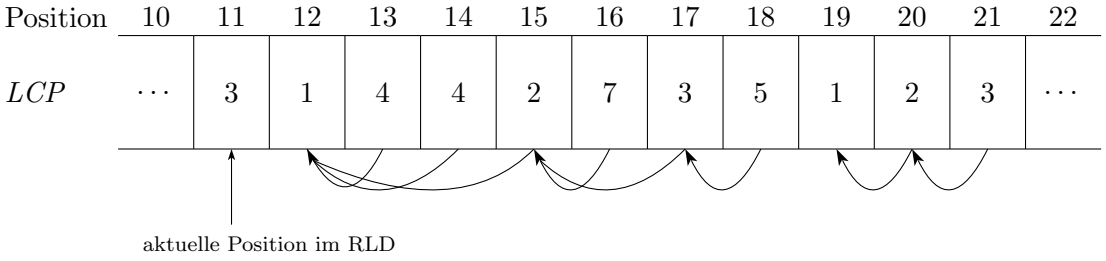


Abbildung 7.10: Verzeigerung der LCP-Werte während des RLD.

Die Realisierung der Funktion $rmq_{LCP}(i+1, q)$ ist speziell an die Vorgehensweise des Algorithmus angepasst. Der Parameter $i+1$ wird im RLD jeweils um eins dekrementiert und bei einer eventuellen Verschiebung eines Suffix aus U wird rmq_{LCP} mit einem variablen Paramter q aufgerufen. Um das Auffinden des Minimums effizient zu gestalten, arbeitet man mit einem Stapel und einem Feld $MIN\_PTR[0..n)$. Die Idee besteht darin, dass alle Positionen in der LCP-Tabelle auf eine kleinere Position zeigen, die einen echt kleineren Wert besitzen und größer als die aktuelle Position im RLD sind[3]. Abbildung 7.10 zeigt hierfür ein Beispiel, wobei zur einfacheren Beschreibung noch undefinierte LCP-Werte aus den V-Bereichen nicht berücksichtigt sind.
Der Algorithmus befindet sich dabei aktuell an Position 11 in SA und es wurde die abgebildete Verzeigerung in MIN_PTR aufgebaut. An allen Positionen aus $[11..n-1]$ der LCP-Tabelle geht ein Verweis auf eine Position mit echt kleinerem LCP-Wert, falls ein solcher existiert. Beispielsweise ist $MIN\_PTR[17] = 15$ wegen $LCP[17] = 3 > LCP[15] = 2$ und da an allen bisher gelesenen Positionen $11 \leq i \leq 19$ kein echt kleinerer Wert als $LCP[19] = 1$ vorkommt, gilt $MIN\_PTR[19] = \perp$. Die Konstruktion dieser Verzeigerung geschieht mit Hilfe eines Stapels ST. Ist man im RLD an Position i angekommen, so liegen diejenigen Positionen der LCP-Tabelle auf dem Stapel, für die noch kein Zeiger auf eine Position mit echt kleinerem Wert gesetzt werden konnte.

[3]Das Feld MIN_PTR entspricht von seiner Logik her dem Feld BH aus Abschnitt 5.6, wobei aber die noch folgende Beschreibung der Pfadkomprimierung fehlt.

Nun vergleicht man $LCP[i]$ mit dem Wert an der Position des obersten Stapelelements $ST.top$ und falls $LCP[i] < LCP[ST.top]$ gilt, so wird $MIN\_PTR[ST.top] = i$ gesetzt und dieses Element vom Stapel genommen. Nun wird wieder mit dem obersten Stapelelement verglichen usw. Dieses Vorgehen wird solange wiederholt, bis entweder der Stapel leer ist oder $LCP[i] \geq LCP[ST.top]$ gilt. Da an Position i noch kein Zeiger auf eine kleinere Position mit echt kleinerem Wert existiert, wird i auf den Stapel gelegt. Initial wird $ST.top = 0$ gesetzt, sodass diese Position immer den leeren Stapel identifiziert, da alle LCP-Werte ab Position 1 größer oder gleich 0 sind. In Definition 7.7.1 sind diese Vorüberlegungen der beschriebenen Verzeigerung formal zusammengefasst.

Definition 7.7.1 *Sei i die aktuelle Position während des RLD. $MIN\_PTR[0..n)$ sei ein Feld der Länge n und wird definiert durch:*

$$MIN\_PTR[j] := \begin{cases} g & \textit{falls} \quad \exists g : g = \max\{k \mid k \in [i..j-1] \wedge LCP[k] < LCP[j]\} \\ \bot & \textit{sonst.} \end{cases}$$

Für eine Minimumsuche im Bereich $[i+1..q]$ folgt man den Zeigern ab $MIN\_PTR[q]$ bis zu einer Position, an der kein Zeiger definiert ist. Wie schon erwähnt ist dabei zu beachten, dass die rechte Grenze q variabel und die linke Grenze $i+1$ fix ist. Beispielsweise würde man für die Anfrage $rmq_{LCP}(12, 18)$ aus Abb. 7.10 den Positionen $18-17-15-12$ folgen und mit $LCP[12] = 1$ das Minimum aus diesem Bereich finden. In Algorithmus 7.5 ist dargestellt, wie sich die LCP-Berechnung in den RLD einbetten lässt und die Bedingung für das Feld aus 7.7.1 für jede erreichte Position wieder hergestellt wird.
In den Zeilen $4-5$ werden die Positionen aus Abb. 7.9 bestimmt und falls aktuell ein Suffix aus U platziert werden kann, wird in Zeile 8 das inverse Suffix-Array mit diesem aktualisiert. Nachdem, wie vorstehend beschrieben, der entsprechende LCP-Wert in Zeile 10 berechnet ist, folgt der explizite Suffixvergleich, der wegen der Beschreibung aus Abschnitt 7.6 nötig ist. Schließlich wird in den Zeilen $14-16$ die Bedingung für das Feld MIN_PTR aus Definition 7.7.1, unter Einbeziehung der aktuellen Position i, hergestellt. Auf dem Stapel ST liegen alle Positionen aus LCP, für die bis einschließlich der aktuellen Position i kein echt kleinerer Wert existiert. In der while-Schleife wird der aktuelle LCP-Wert mit den Werten an diesen Positionen verglichen und falls diese einen echt größeren Wert besitzen, wird ein Zeiger auf Position i gesetzt. Sowohl die Herstellung der Bedingung aus 7.7.1 als auch die Eigenschaft, dass die LCP-Werte an den Positionen der Stapelelemente aufsteigend sind, lassen sich durch eine einfache Induktion zeigen.

Das Folgen der Zeiger in $MIN\_PTR[q]$ ab der rechten Grenze q für eine Anfrage $rmq_{LCP}(i+1, q)$ bis zu einem undefinierten Wert führt zur gesuchten LCP-Position. Dies ergibt sich unmittelbar aus Lemma 5.6.2. Um diese Minimumsuche effektiv zu gestalten, wird *Pfadkomprimierung* verwendet. Dabei werden bei einer Bereichsanfrage alle Zeiger der betrachteten Positionen auf dem Pfad auf die Position gesetzt, die gerade das Minimum repräsentiert. Beispielhaft sei in Abb. 7.10 die Anfrage $rmq_{LCP}(12, 18)$ gestellt. Man folgt dem Pfad $18-17-15-12$ und danach werden die beiden Zeiger an den Positionen 18 und 17 auf 12 gesetzt, sodass sich die neue Verzeigerung aus Abb. 7.11 ergibt. Erfolgt im weiteren Verlauf eine Abfrage mit 18 als rechter Grenze, so erspart man sich auf dem Pfad zum Minimum den Besuch der Position 17.

Algorithmus 7.5 : LCP-Berechnung für Suffixe aus $U \backslash S$ während des RLD

Eingabe : *MIN_PTR* und partiell definierte Felder SA, SA^{-1}, LCP
Ausgabe : LCP-Werte für adjazente Suffixe aus U vollständig gesetzt

```
1  ∀i ∈ [0..n − 1] : MIN_PTR[i] = ⊥
2  for i ∈ [n − 1..1] do
3      if SA[i] ≠ ⊥ then
4          p ← SA^{-1}[SA[i] − 1]
5          q ← SA^{-1}[SA[p + 1] + 1]
6          if SA[i] − 1 ∈ U then
7              SA[p] ← SA[i] − 1
8              SA^{-1}[SA[p]] ← p
9              if LCP[p + 1] = ⊥ then
10                 LCP[p + 1] ← 1 + LCP[rmq_LCP(i + 1, q)]
11         if SA[i − 1] ≠ ⊥ then
12             Ergänze LCP[i] durch expliziten Vergleich
13         if LCP[i] ≠ ⊥ then
14             while LCP[i] < LCP[ST.top] do
15                 MIN_PTR[ST.top] ← i
16                 ST.pop
17             ST.push(i)
```

Je mehr Anfragen und Pfadkompressionen dieser Art im Algorithmusverlauf stattfinden, desto kürzer werden die entsprechenden Pfade. Diese Strategie ist der *find*-Operation der so genannten Union-Find-Datenstruktur entnommen und die Realisierung von *find* in Algorithmus 7.6 angegeben. Nach Algorithmus 7.5 geschieht die Aktualisierung von *MIN_PTR* für Position i, nachdem eine eventuelle rmq_{LCP}-Anfrage aus Zeile 10 erfolgt ist. Daraus folgt, dass $MIN\_PTR[i+1] = \perp$ gilt und es somit für die Realisierung der Minimumsuche bei gleichzeitiger Pfadkompression genügt, nur die rechte Grenze q als Parameter zu übergeben. Die erste Schleife folgt den Zeigern zur Minimumposition und die zweite folgt diesem Pfad noch einmal. Durch einen einfachen Tauschvorgang wird jede angetroffene Position auf die zuvor gefundene Minimumsposition gesetzt.[4]
Für eine Analyse der amortisierten Kosten für eine Folge von n-*find*-Operationen mit Pfadkompression sei auf [8] verwiesen, wobei im nächsten Abschnitt ein Ergebnis daraus zitiert wird.

7.8 LCP-Werte für adjazente Suffixe aus V

Sind die Suffixe aus U durch den RLD platziert worden, wird im LRD SA mit den Suffixen aus V komplettiert.

[4] Es ist auch eine rekursive Formulierung möglich, auf die aber aus Performancegründen (z.B. Vermeidung eines Rekursionsstacks) nicht zurückgegriffen wird.

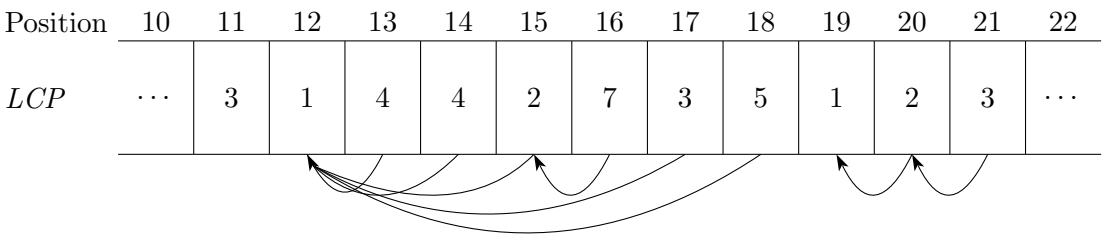


Abbildung 7.11: Pfadkomprimierung an den Positionen 18 und 17.

Algorithmus 7.6 : Minimumsuche im Bereich $[i+1, q]$, wobei i aktuelle Position im RLD ist

Eingabe : Rechte Grenze q für Minimumsuche, *MIN_PTR*
Ausgabe : Index des LCP-Minimums aus $[i+1, q]$

```
1 root ← q
2 while MIN_PTR[root] ≠ ⊥ do
3     root ← MIN_PTR[root]
4 while MIN_PTR[q] ≠ ⊥ do
5     tmp ← q
6     q ← MIN_PTR[q]
7     MIN_PTR[tmp] ← root
8 return root
```

Fast symmetrisch zum RLD erfolgt in diesem letzten Schritt die parallele LCP-Berechnung für Suffixe aus V und konstant vielen Spezialpositionen. Sei dazu $SA[i]$ das aktuell betrachtete Suffix im LRD. Gilt $s_{SA[i]-1} \in V$, so wird dieses Suffix in seinem Bucket an die aktuell vorderste Position p verschoben. Die gleiche Argumentation wie aus Abschnitt 7.7, mit unterschiedlicher Ausrichtung, führt zu folgendem LCP-Wert an Position p:

$$LCP[p] = 1 + LCP[rmq_{LCP}(q+1, i)]$$

Abb. 7.12 veranschaulicht diese Formel.
Sehr ähnlich zu Algorithmus 7.5 verläuft die Einbettung dieses Berechnungsschemas in den LRD. Zunächst werden die Zeilen $11-12$ ersetzt durch einen expliziten Suffix-Vergleich an allen Positionen i, an denen noch kein definierter LCP-Wert vorliegt. Dies ist notwendig, damit für die LCP-Berechnung immer ein konsistenter Zustand vorliegt. Es existiert eine Situation in der an der aktuellen Position i kein definierter LCP-Wert vorliegt und deshalb eine explizite Berechnung von $LCP[i]$ erfolgt. Sei dazu $s_{SA[i]} = \alpha = a^m b\gamma$ und $s_{SA[i-1]} = \beta = a^l c\delta$ mit $a, b, c \in \Sigma$, $a < b$, $a > c$, $m, l \geq 2$ und $\gamma, \delta \in \Sigma^*$. Mit Definition 7.1.1 sind damit $\alpha \in U$ und $\beta \in V$. Nach Voraussetzung sind zu diesem Zeitpunkt die LCP-Werte zwischen adjazenten Suffixen aus U und an ZW-Grenzen bestimmt und weiter werden im aktuellen Durchlauf LCP-Werte von adjazenten Suffixen aus V berechnet.

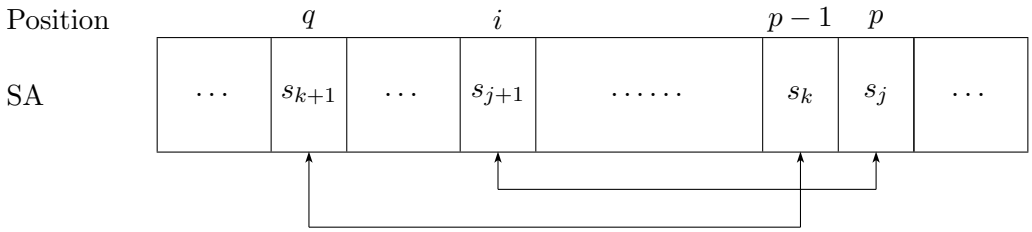


Abbildung 7.12: LCP-Berechnung während des LRD. Der Algorithmus befindet sich an Position i und bestimmt an Position p den LCP-Wert zwischen den beiden Suffixen s_j und s_k mit Hilfe von $lcp(s_{k+1}, s_{j+1})$.

An eventuell vorhandenen Übergängen von V nach U in einem Bucket kann diese Situation entstehen. Konkret würde es z.B. für $\beta = ccca\ldots$ und $\alpha = cccccd\ldots$ zu dieser Situation kommen.

Die zweite Veränderung gegenüber 7.5 betrifft den Zeitpunkt, an der die Verzeigerung in *MIN_PTR* aktualisiert wird. Im LRD muss der LCP-Wert an Position i wegen der Anfrage $rmq_{LCP}(q+1, i)$ mit berücksichtigt werden. Deshalb erfolgt die Aktualisierung und damit $MIN\_PTR[i] = \perp$ noch vor der Anfrage selbst. Die Situation ist ansonsten symmetrisch zum RLD und Algorithmus 7.7 gibt das Verfahren für den LRD an.

7.9 Verringerung von expliziten Vergleichen

Am Ende von Abschnitt 7.6 ist beschrieben wie partielle LCP-Werte für Suffixe aus S während TSQS und IS gesetzt werden können. Sind diese aufgrund von dynamischen Schlüsseln zu klein, so muss im anschließenden RLD ein expliziter Suffixvergleich erfolgen. In diesem Abschnitt soll besprochen werden, wie mit einer zusätzlichen Technik versucht werden kann, die Anzahl dieser teuren Vergleich zu verringern. Das Verfahren setzt in Algorithmus 7.2 auf, also während der Platzierung der Suffixe aus S in SA. Diese werden wie besprochen in lexikographisch aufsteigender ZW-Reihenfolge platziert. Wird nun Suffix s_k an Position $p-1$ und Suffix s_j an Position p platziert, dann lässt sich der vorberechnete partielle LCP-Wert an Position p verwenden, um eventuell den korrekten Wert wieder durch eine Range Minimum Query zu erhalten. Sei $l = LCP[p]$ und sei SA^{-1} für alle bereits platzieren Suffixe definiert besetzt und für alle anderen Suffixe noch undefiniert. Gilt dann $SA^{-1}[k+l] < p \neq \perp$ und $SA^{-1}[j+l] < p \neq \perp$, dann liegt die fast identische Situation aus Abb. 7.12 vor, mit dem einzigen Unterschied, dass nun der LCP-Wert l verwendet werden kann. Es ergibt sich in diesem Fall das Berechnungsschema:

$$LCP[p] = l + LCP[rmq_{LCP}(q+1, i)]$$

Zu diesem Zeitpunkt lässt sich das Minimum aus $[q+1..i]$ nicht berechnen, da die LCP-Werte noch unvollständig sind. Man kann aber für Position i die benötigten Informationen $q+1$ und p in einer separaten Datenstruktur abspeichern, um später mit einem LRD den LCP-Wert an Position p auf den korrekten Wert zu setzen.

Algorithmus 7.7 : LCP-Berechnung für Suffixe aus V und an V-U-Übergängen während des LRD

Eingabe : MIN_PTR und partiell definierte Felder SA, SA^{-1}, LCP
Ausgabe : Vollständige LCP-Tabelle

```
1  ∀i ∈ [0..n − 1] : MIN_PTR[i] = ⊥
2  for i ∈ [1..n − 1] do
3      if SA[i] ≠ ⊥ then
4          p ← SA^{-1}[SA[i] − 1]
5          q ← SA^{-1}[SA[p − 1] + 1]
6          if LCP[i] = ⊥ then
7              Berechne LCP[i] explizit an V-U-Übergang
8          if LCP[i] ≠ ⊥ then
9              while LCP[i] < LCP[ST.top] do
10                 MIN_PTR[ST.top] ← i
11                 ST.pop
12             ST.push(i)
13         if SA[i] − 1 ∈ V then
14             SA[p] ← SA[i] − 1
15             SA^{-1}[SA[p]] ← p
16             if LCP[p] = ⊥ then
17                 LCP[p] ← 1 + LCP[rmq_LCP(q + 1, i)]
```

Dazu ist es wiederum notwendig, dass bei diesem LRD alle LCP-Werte aus $[0..i-1]$ ihren korrekten Wert besitzen. Im Fall von $SA^{-1}[k+l] = \perp$ oder $SA^{-1}[j+l] = \perp$ wurden die beiden Suffixe s_k und s_j in der Sortierung nicht durch Suffixe aus S voneinander differenziert, sodass der LCP-Wert wegen ZW-Schlüsseln höchstens um 1 zu klein sein kann. Gilt hingegen $SA^{-1}[k+l] > p$ oder $SA^{-1}[j+l] > p$, dann ist wieder ein expliziter Suffix-Vergleich nötig, denn es entstehen Inkonsistenten, da für keinen LCP-Wert der größer als p ist dessen Korrektheit garantiert werden kann. Deshalb muss für diesen Fall der LCP-Wert an Stelle p direkt berechnet werden, damit später mit einem LRD alle Werte im Bereich $[q+1..i]$ für die Minimumsuche korrekt sind. Weiter ist zu beachten, dass es mehr als eine Minimumsuche mit der rechten Grenze i geben kann. Deswegen arbeitet man mit verketteten Listen, deren Elemente die jeweiligen Platzierungspositionen p und die entsprechenden linken Grenzen $q+1$ enthalten. Erreicht man dann in diesem LRD eine Position i, durchläuft man die eventuell vorhandene Liste und führt für alle gefundenen Elemente die Minimumsuche aus. Für die Realisierung von rmq_{LCP} wird auch hier wieder Pfadkomprimierung eingesetzt. Algorithmus 7.8 zeigt, wie sich diese Listen während der Platzierung der Suffixe aus S generieren lassen. Dabei sind $q\_pos = q+1$ und $i\_pos = i$ gesetzt, um keine Verwechslungen mit den Laufvariablen zu provozieren. Das Feld $S\_RMQ[0..n-1]$ besteht aus Zeigern auf verkettete Listen, das später genutzt wird, um die entsprechenden Minimumsanfragen zu bearbeiten.

In Algorithmus 7.9 ist schließlich der LRD dargestellt, der an einer Position alle RMQs ausführt, indem jeder vorhandenen verketteten Liste die entsprechenden Informationen entnommen werden.

Algorithmus 7.8 : Generierung von verketteten Listen für RMQ-Anfragen

Eingabe : Sortiertes Feld SP von Suffixen aus S
Ausgabe : S_RMQ enthält Informationen für nachfolgenden LRD

```
1  p ← 0
2  s ← 0
3  for j ∈ [0..|Σ|²] do
4      if SUFFIX_COUNT[j] > 0 then
5          for k ∈ [0..SAMPLE_COUNT[j]] do
6              SA[i] ← SP[s]
7              setze SA⁻¹
8              q_pos ← SA⁻¹[SA[p − 1] + LCP[p]]
9              i_pos ← SA⁻¹[SA[p] + LCP[p]]
10             if i_pos ≠ ⊥ ∧ i_pos ≠ ⊥ then
11                 if i_pos > p ∨ i_pos > p then
12                     Berechne LCP[p] explizit
13                 else
14                     Generiere Listenelement mit q_pos und p
15                     und hänge es an das Ende der Liste S_RMQ[i_pos]
16             else
17                 Erhöhe LCP[p] eventuell um 1
18             Inkrementiere p und s
19         Erhöhe p um SUFFIX_COUNT[j] − SAMPLE_COUNT[j]
```

7.10 Zeit- und Speicheraufwand

Puglisi und Maniscalco haben für ihren Algorithmus noch keine strenge Laufzeitanalyse vorgestellt. Sie geben eine $O(n^2 \log n)$-Schätzung an, die nach ihrer Meinung etwas zu hoch angesetzt ist. Der zusätzliche Aufwand für die parallele LCP-Berechnung kann hingegen angegeben werden.
Zunächst liegt die Bestimmung der LCP-Werte an ZW-Grenzen aus Abschnitt 7.5 in $O(|\Sigma|^2)$, da sie mit Algorithmus 7.3 in einer Schleife eingebettet ist, die alle $|\Sigma|^2$ vielen ZW auf Vorkommen der Eingabe überprüft. Kann von einer konstanten Alphabetgröße ausgegangen werden, so bedeutet dies einen konstanten Aufwand.
In Algorithmus 7.4 ist die Einbettung der teilweisen LCP-Berechnung in TSQS angegeben. Da hier nur zwei Abfragen mit konstantem Aufwand nach der dreiteiligen Partitionierung vorliegen, hängt der Gesamtaufwand davon ab, wieviel Knoten der Aufrufbaum von TSQS haben kann. Zur Analyse betrachtet man folgenden Fall. Sei n die Anzahl der zu sortierenden Suffixe mit $n = 2|\Sigma|^z$, $z \in \mathbb{N}\backslash\{0\}$ und $m \in O(n)$ der größte vorkommende LCP-Wert

Algorithmus 7.9 : LRD um LCP-Werte teilweise zu ergänzen

Eingabe : *S_RMQ* enthält RMQ-Informationen
Ausgabe : Ergänzte LCP-Werte, um die Anzahl expliziter Suffix-Vergleiche zu verringern

```
1 for i ∈ [0..n − 1] do
2     if LCP[i] ≠ ⊥ then
3         Aktualisiere MIN_PTR
4         if S_RMQ[i] ≠ NULL then
5             list_elem ← S_RMQ[i].first_elem
6             repeat
7                 Erhöhe LCP[list_elem.p] um LCP[rmq_LCP(list_elem.q+1, i)]
8                 list_elem ← list_elem.next
9             until list_elem.next = NULL
```

aller Suffixpaare mit $m > z$. Alle $|\Sigma|$ vielen verschiedenen Symbole bzw. Schlüssel an Symbolpostion 0 seien genau $n/|\Sigma|$ mal vorhanden. Damit entstehen mindestens $|\Sigma|$ viele Knoten, die genau $n/|\Sigma|$ viele Suffixe enthalten. Analog seien nun für all diese Knoten alle $|\Sigma|$ vielen verschiedenen Symbole an Symbolpostion 1 genau $n/(|\Sigma|^2)$ mal vorhanden. Dann entstehen für jeden der $|\Sigma|$ vielen Knoten mindestens nochmals $|\Sigma|$ viele Knoten, also insgesamt mindestens $|\Sigma|^2$ viele. Diese Aufsplittung wird nun genau z mal wiederholt, sodass sich mindestens $|\Sigma|^z = n/2$ viele Knoten ergeben, die genau zwei Suffixe beinhalten. Da man für alle diese Knoten mit zwei Suffixen s_i und s_j schon $m - z$ Symbole betrachtet hat, gilt $lcp(s_i, s_j) \leq m - 1 - z$. Deshalb kann aus jedem dieser $n/2$ vielen Knoten noch eine Liste von $O(m - z) \in O(n)$ vielen Knoten entstehen und somit liegt für diesen Fall die Gesamtanzahl der Knoten im Aufrufbaum von TSQS in $\Omega(n^2)$.
Für eine obere Schranke sei wieder n die Suffixanzahl und m der größte vorkommende LCP-Wert. Die maximale Höhe des Aufrufbaums hängt von diesen beiden Parametern ab. Aus einem Knoten kann im schlimmsten Fall ein neuer Knoten entstehen, bei dem entweder nur n oder nur m um eins kleiner geworden ist, weshalb die maximale Baumtiefe $n + m$ ist. Die maximale Breite des Baumes ist n, denn für jede Tiefe können bei n Suffixen nur maximal n verschiedene Knoten vorkommen. Mit $m < n$ bestimmt sich somit die maximale Anzahl von Knoten zu $n(n + m) \in O(n^2)$. Zusammen mit dem oben konstruierten Fall liegt die maximale Anzahl von Knoten in $\Theta(n^2)$ und somit Algorithmus 7.4 in $O(n^2)$.
Der Aufwand für die Herstellung der Verzeigerung für *MIN_PTR* und die Organisation des Stapels aus Algorithmus 7.5 für jede Position i im RLD ist $O(n)$. Denn jede Position aus $[0..n - 1]$ wird höchstens einmal auf den Stapel gelegt und höchstens einmal vom Stapel genommen. Die Kosten für das Auffinden des Minimums von Algorithmus 7.6 bei gleichzeitger Pfadkomprimierung sind in [8] analysiert. Die amortisierten Kosten für eine Folge von n vielen RMQ-Anfragen sind $O(n \log^* n)$. Dabei ist

$$log^* n = min\{k \mid \underbrace{\log \ldots \log n}_{k\text{-mal}} \leq 1\}.$$

Dies ist eine extrem langsam wachsende Funktion. So ist beispielsweise für $n \leq 10^{19728}$ der Funktionswert $\log^* n \leq 5$[29]. Zur Ergänzung der partiellen LCP-Werte müssen $O(|S|)$ viele explizite Suffixvergleiche erfolgen, die in den ersten $O(n)$ Symbolen übereinstimmen, wobei $|S| \leq n/2$ ist. Damit liegt der Algorithmus 7.5 des RLD in $O(n^2)$.

Hingegen liegt Algorithmus 7.7 in $O(n \log^* n)$, wenn man von einer konstanten Alphabetgröße ausgeht. Denn es werden wie in Abschnitt 7.8 beschrieben, zwar auch explizite Suffixvergleiche durchgeführt, deren Anzahl hängt aber nur von $|\Sigma|$ ab und nicht von n. Denn diese Situation kann maximal für jedes Symbol auftreten und bei einem maximal angenommenen Alphabet $|\Sigma| = 256$ erhält man einen Aufwand von $O(n)$.

Algorithmus 7.8 aus Abschnitt 7.9 führt in Zeile 12 explizite Suffixvergleiche durch. Deren Anzahl hängt von $|S|$ und somit von der Eingabelänge ab. Die innere Schleife wird wegen $\sum_{j=0}^{|\Sigma|^2} SAMPLE\_COUNT[j] = |S| \leq n/2$ maximal $n/2$ mal für alle Iterationen der äußeren Schleife durchlaufen und diese hängt nur von $|\Sigma|$ ab. Die Alphabetgröße wurde für den gesamten Algorithmus stets konstant angenommen, sodass sich ein Gesamtaufwand von $O(n^2)$ ergibt. Dagegen liegt der LRD aus Algorithmus 7.9 in $O(n)$, denn es können nur maximal n Listenelemente für eine RMQ generiert worden sein. Diese werden über alle n Iterationen ausgelesen, sodass sich ein amortisierter linearer Aufwand ergibt.

Insgesamt ergibt sich damit ein zusätzlicher Aufwand von $O(n^2)$ für die parallele Berechnung der LCP-Tabelle. Selbstverständlich ist aber in diesem Fall der versteckte konstante Faktor kleiner, als bei einer komplett expliziten LCP-Berechnung für alle adjazente Suffixe in SA. Dies lässt sich einfach dadurch begründen, dass $|S| \leq n/2$ gilt und man einen expliziten Vergleich für maximal $|S|$ viele Suffixe durchführen muss. Diese Vergleiche beginnen darüber hinaus meist nicht von Position 2 an[5], sondern sind ja schon durch die partiellen LCP-Werte vermindert worden.

Der SAKA von Puglisi und Maniscalco benötigt $6n$-Bytes Arbeitsspeicher. Für die LCP-Berechnung aus den Abschnitten 7.5 bis 7.8 werden zusätzlich die Felder LCP, LCP_S, SA^{-1} und MIN_PTR benötigt. Da $|S| \leq n/2$ ist, benötigt LCP_S zusätzlich $2n$-Bytes, während die übrigen $4n$-Bytes Speicher brauchen. Hinzu kommt der Stapel zur Organisation des MIN_PTR-Feldes, der im Maximalfall ebenfalls $4n$-Bytes benötigt, sodass sich insgesamt ein Speicheraufwand von $24n$-Bytes ergibt. Nimmt man die zusätzliche Technik aus Abschnitt 7.9 hinzu, so kommt noch Feld S_RMQ für die Zeiger in die verketteten Listen mit $4n$-Bytes hinzu. Da dieses Verfahren auf den Suffixen von S aufsetzt, kann es maximal $n/2$ viele Listenelemente geben, mit jeweils 8 Bytes Speicherverbrauch. Damit hat man für diese Variante insgesamt einen Aufwand von $32n$-Bytes.

[5] Es sei daran erinnert, dass alle Suffixe, die mit einem gemeinsamen ZW beginnen und somit mindestens einen LCP-Wert von 2 teilen, zusammen an TSQS übergeben werden.

8 Testergebnisse

Die Algorithmen von Larsson/Sadakane (LS), Kärkkäinen/Sanders (KS) und Puglisi-/Maniscalco (PM) wurden so erweitert, dass eine parallele Berechnung der LCP-Tabelle möglich ist. Implementierungen dieser Erweiterungen wurden in C/C++ geschrieben und sollen in diesem Kapitel gegeneinander getestet werden. Die Quellcodes der SAKAs stammen ausschließlich von den Autoren selbst[1]. Die Testdateien sind dem Corpus von Manzini entnommen[2] und Metadaten über diese Dateien entstammen aus [28]. Tabelle 8.1 gibt eine Übersicht über die verwendeten Testdateien, wobei die ersten beiden Spalten den durschnittlichen und den maximalen LCP-Wert für das zugehörige Suffix-Array der Datei angeben.

Datei	∅-LCP	max-LCP	Bytes	$\|\Sigma\|$	Beschreibung
bible	14	551	4047392	63	bible
chr22	1,979	199,999	34553758	5	Menschliches Chromosom 22
jdk13c	679	37,334	69728899	113	JDK 1.3 Dokumentation
rfc	93	3,445	116421901	120	IETF RFC Dateien
sprot34	89	7,373	109617186	66	SwissProt Datenbank
eco	17	2,815	4638690	4	Escherichia-Coli Genom

Tabelle 8.1: Die Testdateien im Überblick

Die zu testenden Algorithmen wurden unter Eclipse mit dem gcc-Compiler in Version 4.1.3 und der O3-Option übersetzt. Für die Messung des maximalen Speicherverbrauchs während der Ausführung ist die Bibliothek *memusage* mit eingebunden worden. Getestet wurde auf einem System mit zwei Dual-Core AMD Opteron Prozessoren, die eine Taktfrequenz von 3GHz besitzen. Das Betriebssystem war OpenSuse 10.3 mit Linux-Kern 2.6.22. Die gemessenen Laufzeiten sind der Durchschnitt über drei durchgeführte Läufe und beinhalten nicht das Laden der Eingabe. Die Zeitmessungen wurden mit in die Programme kompiliert. Für die Messungen mit dem Kasai-Algorithmus wurde die Berechnung des inversen Suffix-Arrays mit zur Rechenzeit einbezogen, da dieses Feld bei LS und PM am Ende der *SA*-Konstruktion nicht zur Verfügung steht.

Bevor die Testergebnisse im Einzelnen besprochen werden, gibt Tabelle 8.2 eine Übersicht über die gefundenen theoretischen Laufzeiten und Speicheranforderungen der besprochenen Algorithmen.

[1] LS: http://www.larsson.dogma.net/qsufsort.c
KS: In [16] direkt angegeben
PM: http://www.michael-maniscalco.com/downloads/MSufSort.3.1.1.zip
[2] http://www.mfn.unipmn.it/ manzini/lightweight/corpus/

Im oberen Bereich sind die Basisalgorithmen aufgeführt und im unteren Bereich die Erweiterungen mit der LCP-Berechnung. Der Speicherverbrauch ist in Bytes angegeben, bezogen auf die Länge n der Eingabe. Mit der Technik aus Abschnitt 7.9 kommt *PM_LCP* auf $32n$-Bytes Speicheranforderung.

Algorithmus	Laufzeit	Speicher
LS	$O(n \log n)$	$8n$
KS	$O(n)$	$24n$
PM	$O(n^2 \log n)$	$6n$
LS_LCP	$O(n^2)$	$20n$
KS_LCP	$O(n)$	$36n$
PM_LCP	$O(n^2 \log n)$	$24n$

Tabelle 8.2: Zusammenfassung aller theoretisch ermittelten Laufzeiten und Speicheranforderungen der besprochenen Algorithmen.

Für den Algorithmus von LS wurden drei verschiedene Programme in die Testläufe aufgenommen:

- Originalalgorithmus: *LS*
- Originalalgorithmus und Kasai-Algorithmus: *LS_Kasai*
- Algorithmus mit paralleler LCP-Berechnung: *LS_LCP*

In Tabelle 8.3 sind die Laufzeiten in Sekunden aufgelistet, während Tabelle 8.4 die Spitzenwerte im Speicherverbrauch angibt.

	bible	chr22	jdk13	rfc	sprot34	eco
LS	2, 35	25, 97	60, 48	94, 88	86, 45	2, 54
LS_Kasai	2, 45	33, 69	69, 07	112, 27	106, 75	3, 18
LS_LCP	4, 62	51, 43	103, 67	192, 15	159, 26	6, 04

Tabelle 8.3: Laufzeiten in Sekunden für LS

	bible	chr22	jdk	rfc	sprot34	eco
LS	36	311	628	1047	986	41
LS_Kasai	52	449	906	1513	1425	60
LS_LCP	84	725	1464	2444	2300	97

Tabelle 8.4: Spitzenwerte des Speicherverbrauchs in MB bei LS

Für den Skew-Algorithmus wurden vier verschiedene Implementierungen getestet:

- Originalalgorithmus: *KS*
- Originalalgorithmus und Kasai-Algorithmus: *KS_Kasai*
- Algorithmus mit paralleler LCP-Berechnung (Alstrup): *KS_Alstrup*
- Algorithmus mit paralleler LCP-Berechnung (Fischer): *KS_Fischer*

Bei der parallelen LCP-Berechnung wurden zwei verschiedene Varianten implementiert, die sich in der intern verwendeten RMQ-Implementierung unterscheiden. Zum einen das Verfahren nach Alstrup [4], *KS_Alstrup*, und zum anderen das Verfahren nach Fischer [10], *KS_Fischer*. Tabelle 8.5 gibt die Laufzeiten und Tabelle 8.6 den Speicherverbrauch für diese Programme an.

	bible	chr22	jdk13	rfc	sprot34	eco
KS	6, 30	62, 29	131, 57	248, 85	235, 9	6, 88
KS_Kasai	7, 09	72, 76	146, 13	273, 85	242, 54	8, 24
KS_Alstrup	9, 10	96, 68	188.99	–	–	10, 97
KS_Fischer	9, 60	102, 64	207, 14	–	–	12, 65

Tabelle 8.5: Laufzeiten in Sekunden für KS

	bible	chr22	jdk13	rfc	sprot34	eco
KS	123	1100	2209	3648	3456	145
KS_Kasai	139	1239	2488	4109	3894	163
KS_Alstrup	170	1514	3041	–	–	200
KS_Fischer	173	1516	3049	–	–	201

Tabelle 8.6: Spitzenwerte des Speicherverbrauchs in MB bei KS

Für die beiden Dateien *rfc* und *sprot34* konnten keine Zeiten gemessen werden, weil der Speicherverbrauch in beiden Fällen die zur Verfügung stehenden 4GB Arbeitsspeicher überschritten hat. Da alle anderen Algorithmen diesen Verbrauch nicht zeigten, bestätigt dies Ergebnisse aus [28], nachdem rekursive SAKAs sehr speicherintensiv sind.

Auch für den Algorithmus von Puglisi und Maniscalco wurden vier Varianten getestet:

- Originalalgorithmus: *PM*
- Originalalgorithmus und Kasai-Algorithmus: *PM_Kasai*
- Algorithmus mit paralleler LCP-Berechnung: *PM_LCP*
- Parallele LCP-Berechnung und Technik aus Abschnitt 7.9: *PM_LCP_2*

	bible	chr22	jdk13	rfc	sprot34	eco
PM	0,94	12,05	17,72	29,41	36,58	1,4
PM_Kasai	1,67	21,09	30,17	55,98	64,19	2,2
PM_LCP	1,74	19,89	46,16	58,86	65,70	2,43
PM_LCP_2	2,49	23,77	41,57	69,35	74,31	2,96

Tabelle 8.7: Laufzeiten in Sekunden für PM

	bible	chr22	jdk13	rfc	sprot34	eco
PM	26	209	420	700	658	29
PM_Kasai	57	484	977	1630	1535	65
PM_LCP	90	761	1535	2562	2412	103
PM_LCP_2	129	1096	2210	3651	3477	146

Tabelle 8.8: Spitzenwerte des Speicherverbrauchs in MB bei PM

Tabelle 8.7 gibt die Laufzeiten und Tabelle 8.8 den Speicherverbrauch an.
Um die absoluten Zeiten und Speicheranforderungen besser einordnen zu können, vergleicht man die Verfahren der parallelen LCP-Berechnung mit der jeweiligen Kasai-Variante. Die Differenz der Testergebnisse der beiden Algorithmen wird auf die Kasai-Variante bezogen und in % angegeben. In Tabelle 8.9 sind diese Kennwerte für die Laufzeiten dargestellt, wobei die letzte Spalte den durchschnittlichen Prozentsatz über alle Dateien angibt.

	bible	chr22	jdk13	rfc	sprot34	eco	∅
LS_LCP	+88	+53	+50	+71	+49	+90	+67
KS_Alstrup	+28	+33	+29	–	–	+33	+31
KS_Fischer	+35	+41	+42	–	–	+54	+43
PM_LCP	+4	−6	+53	+5	+2	+9	+11
PM_LCP_2	+49	+13	+38	+24	+16	+35	+29

Tabelle 8.9: Prozentualer Laufzeitvergleich mit den Kasai-Varianten

Mit durchschnittlich +67% größerer Laufzeit war *LS_LCP* im Vergleich zu *LS_Kasai* unter allen anderen Algorithmen der Langsamste. Dies dürfte sich unter anderem durch die aufwendige Verwaltung des *BH*-Feldes begründen. Die beiden Implementierungen für den Skew-Algorithmus lagen mit +31% und +43% in etwa gleich auf, wobei für *jdk13* der Unterschied zwischen den beiden mit 13% am größten ist. Da für alle Dateien *KS_Alstrup* schneller war als *KS_Fischer*, lässt darauf schließen, dass *KS_Alstrup* die bessere Wahl darstellen dürfte. Die beiden Varianten von PM waren mit +11% bzw. +29% die schnellsten im Test, wobei für *chr22* der Algorithmus *PM_LCP* sogar einmal schneller war als *PM_Kasai*. *PM_LCP_2* aus Abschnitt 7.9 war bis auf die Datei *jdk13* immer deutlich langsamer als *PM_LCP*. Nur in diesem einen Fall ergab sich ein Laufzeitplus von 28% gegenüber *PM_LCP*.

Tabelle 8.10 gibt die Anzahl von expliziten Suffixvergleichen an, die für beide Verfahren gemacht wurden.

	bible	chr22	jdk13	rfc	sprot34	eco
PM_LCP	1042513	7820241	19920374	26113431	27281592	1065563
in %	26	23	29	22	25	23
PM_LCP_2	4948	139737	8366	36490	20673	18692
in %	0,1	0,4	0,01	0,03	0,01	0,4

Tabelle 8.10: Häufigkeiten von expliziten LCP-Berechnungen für die beiden PM-Varianten. Angegeben ist auch der zugehörige Prozentwert in Bezug auf die Suffixanzahl der Testdatei.

Es ist eine deutliche Reduzierung dieser Art von Vergleichen zu erkennen. Ohne die zusätzliche Technik lag die Anzahl bei $22-29\%$ und mit dieser maximal noch im Promille-Bereich. Warum dennoch *PM_LCP* in fast allen Fällen schneller war, lässt sich ohne eine detailliertere Untersuchung nur vermuten. Zum einen muss der zusätzliche Aufwand für die beiden Algorithmen 7.8 und 7.9 in Betracht gezogen werden. Zum anderen ist der durchschnittliche LCP-Wert einer Datei zu beachten. Ist dieser klein im Vergleich zu anderen Eingaben, so könnten die expliziten Vergleiche weniger ins Gewicht fallen als der zusätzliche Verwaltungsaufwand. Ein Indiz hierfür ist das Ergebnis für *jdk13*, denn diese Eingabe hat mit 679 den zweitgrößten durchschnittlichen LCP-Wert unter allen Testdateien. Dagegen spricht allerdings das Ergebnis für *chr22*, denn hier ergab sich trotz des höchsten durchschnittlichen LCP-Werts eine längere Laufzeit. Allgemein dürfte die Laufzeit von der Anzahl der im Verlauf der Sortierung vorkommenden dynamischen Schlüssel abhängen. Werden viele Suffixe über solche Schlüssel sortiert, so sind auch viele LCP-Werte noch unvollständig und müssen explizit nachberechnet werden. Aus dieser Sicht heraus wären in *jdk13* sehr viele dynamische Schlüssel vorhanden, die mit der Technik aus Abschnitt 7.9 über eine RMQ billig ergänzt werden können, während in *chr22* die Art von Schlüsseln überwiegt, für die trotzdem ein expliziter Vergleich durchgeführt werden muss. Insgesamt dürfte die Laufzeit von der inneren Struktur der Eingabe abhängen, die die Anzahl von dynamischen Schlüssel vorgibt.

In Tabelle 8.11 sind die Kennzahlen aller Testdateien in Prozent bezüglich der Spitzenwerte im Speicherverbrauch dargestellt.

	bible	chr22	jdk13	rfc	sprot34	eco	∅
LS_LCP	+62	+61	+62	+62	+61	+62	+62
KS_Alstrup	+22	+22	+22	–	–	+23	+22
KS_Fischer	+24	+22	+23	–	–	+23	+23
PM_LCP	+58	+57	+57	+57	57	+58	+57
PM_LCP_2	+126	+126	+126	+124	+127	+125	+126

Tabelle 8.11: Prozentualer Speichervergleich mit Kasai-Varianten

Die fast konstanten Werte waren zu erwarten, da bis auf *PM_LCP_2* keine von der Eingabe abhängigen Speicherreservierungen vorgenommen werden. Die Größe der verketten Listen in *PM_LCP_2* hängt wiederum von der Struktur der Eingabe ab. Für die getesteten Dateien ergaben sich aber bis auf *rfc* keine nennenswerten Schwankungen, sodass man davon ausgehen kann, dass die Anzahl der Listenelemente für die Dateien relativ klein ist im Vergleich zu den zusätzlichen Feldern, die dafür bereitgestellt werden müssen. Die beiden Skew-Varianten bieten hier das beste Bild mit etwas mehr als 20% zusätzlichem Speicherbedarf gegenüber dem Kasai-Algorithmus. Untereinander liegen beide Varianten etwa gleich auf, sodass aus dieser Sicht heraus, zusammen mit den Ergebnissen aus Tabelle 8.9, *KS_Alstrup* die geeignetere Wahl sein dürfte. Hier könnten weitere Test mehr Aufschluss geben. Die beiden PM-Implementierungen unterscheiden sich untereinander sehr stark. Auch dies ist eine erwartete Konsequenz des zusätzlichen Verwaltungsaufwands für *PM_LCP_2*. Es wird mehr als das Doppelte an Speicher reserviert gegenüber der Kasai-Variante und zusammen mit den Laufzeitergebnissen, erscheint dies nicht erstrebenswert. Wenn man wie schon erwähnt die Struktur der Dateien bezüglich der Anzahl von vorhandenen dynamischen Schlüsseln kennen würde, könnte man die *PM_LCP_2*-Variante weiteren Tests unterziehen, um zu sehen, ob sich der zusätzliche Aufwand lohnt.

Nimmt man den Durchschnitt über alle prozentuale Kennwerte, so erhält man:

- 36% mehr Laufzeit und
- 58% mehr Speicheraufwand

gegenüber der LCP-Berechnung nach der Konstruktion des Suffix-Arrays mit dem Kasai-Algorithmus.

9 Fazit

In dieser Arbeit wurde der Frage nach der effizienten Realisierbarkeit der LCP-Tabellen-Berechnung während der Konstruktion eines Suffix-Arrays nachgegangen. Es wurden vier Algorithmen exemplarisch aus der Fülle vorhandener SAKAs ausgewählt und auf diese Frage hin untersucht. Für die Prefix-Doubling Algorithmen aus Kapitel 5 wurden die zentralen Konzepte des Intervall-Baums und des *BH*-Feldes eingesetzt, um die LCP-Werte effizient berechnen zu können. Dabei ist gezeigt worden, dass mit dem Intervall-Baum die asymptotische Laufzeit von $O(n \log n)$ erhalten werden kann, während dies für das *BH*-Feld nicht geklärt werden konnte. Es stellte sich vor allem bei Larsson und Sadakane heraus, dass durch die große Spezialisierung des Verfahrens viele Informationen nicht in einer Form vorliegen, die eine parallele LCP-Berechnung unterstützt. Dies äußert sich beispielsweise darin, dass das Suffix-Array dazu benutzt wird, um temporär andere Daten als Suffixindizes zu speichern. Dies führt zu einem semantisch inkonsistenten Zustand während der Sortierung, was die LCP-Berechnung aufwendiger macht.

Der Skew-Algorithmus aus Kapitel 6 konnte am einfachsten von allen Verfahren beschrieben werden und die LCP-Berechnung ließ sich konzeptionell einfach an ihn anpassen, sodass die optimale $O(n)$-Laufzeit mit der zusätzlichen Berechnung gehalten werden konnte. Als einen Parameter für die LCP-Berechnung lassen sich verschiedene Implementierungen eines RMQ-Verfahrens einbinden, das in konstanter Zeit das Minimum liefert. Zwei davon wurden in das Berechnungsschema eingeflochten, es ist aber auch denkbar, RMQ-Verfahren mit $O(\log n)$-Zeitkomplexität für die Minimumbestimmung zu testen. Nach [11] sind viele dieser Verfahren in der Praxis schneller als diejenigen mit $O(1)$-Zugriff, falls die Eingaben eine bestimmte Größe nicht überschreiten. Eventuell ließen sich dadurch die praktischen Laufzeiten noch verbessern.

Im Algorithmus von Puglisi und Maniscalco aus Kapitel 7 befinden sich viele Konzepte zur Beschleunigung der Suffix-Sortierung, die in geeigneter Kombination dieses Verfahren sehr schnell werden lassen. Diese Vielfalt führt andererseits dazu, dass die LCP-Berechnung aufwendiger wird. Es traten ähnliche Schwierigkeiten wie bei Larsson und Sadakane auf, weil auch hier TSQS unter Verwendung von dynamischen Sortierschlüsseln zur Anwendung kommt. Das Setzen von LCP-Werten an Partitionsgrenzen und die Minimumsuche mit Hilfe von Pfadkomprimierung waren hier die wesentlichen Konzepte, um ein bezüglich Laufzeit effizientes Verfahren generieren zu können. Die geschätzte $O(n^2 \log n)$ worst-case Laufzeit des Basisalgorithmus wurde mit einem zusätzlichen $O(n^2)$-Aufwand für die Berechnung der LCP-Werte nicht schlechter.

Mit den eingesetzten Konzepten und Methoden sowie den Testergebnissen aus Kapitel 8, ist für die untersuchten Algorithmen die LCP-Berechnung während der Konstruktion eines Suffix-Arrays nicht lohnenswert. Dies liegt vor allem am zusätzlichen Speicheraufwand, der in allen drei Verfahren dazu führt, dass der Gesamtspeicherverbrauch über dem eines Suffix-Baums liegt. Damit ist der eigentliche Hauptvorteil eines Suffix-Arrays, die speichereffiziente Textindizierung, nicht mehr gegeben. Ein Grund für dieses negative Ergebnis sind die engen Rahmenbedingungen, die es gilt einzuhalten. Einerseits ist man an einer möglichst schnellen LCP-Berechnung interessiert und andererseits soll diese kaum zusätzlichen Speicher benötigen. Hier zeigt sich sehr deutlich ein Speicher/Zeit Trade-Off, der nicht einfach aufzulösen ist. Es ist zu bemerken, dass die zur Zeit in Bezug auf Speicher sparsamsten SAKAs $5n - 6n$-Bytes benötigen. Allein die Bereitstellung des Feldes für die LCP-Tabelle kostet nochmal $4n$-Bytes, sofern dieses nicht auf irgendeine andere Art komprimiert dargestellt wird. Eine Komprimierung dürfte aber wiederum mit Laufzeiteinbußen verbunden sein. Stellt man dann nochmals $4n$-Bytes für eine weitere Datenstruktur bereit, so befindet man sich schon im Bereich eines Suffix-Baums. Arbeitet man mit Range Minimum Queries, die Anfragen in konstanter Zeit beantworten können, so werden nach [12] mindestens $2n - o(n)$ zusätzliche Bits benötigt.

Der Kasai-Algorithmus greift auf ein Suffix-Array zurück, das komplett vorliegt und an jeder Position eine konsistente Information enthält. Diese Information ist während der Konstruktion durch einen SAKA nicht immer gegeben und verlangt nach Methoden, um sie zu extrahieren. Das erklärt auch die Einfachheit des Kasai-Algorithmus gegenüber den vorgestellten Verfahren, die sehr adaptiv auf den jeweiligen Basisalgorithmus abgestimmt sind.

Es bleibt somit eine offene Frage, ob es nicht effizientere Konzepte und Methoden zur parallelen LCP-Berechnung gibt, sodass sich sowohl hinsichtlich Laufzeit als auch Speicheraufwand ein Vorteil gegenüber der LCP-Berechnung nach der Konstruktion eines Suffix-Arrays ergibt.

A Anhang

Um einen schnelleren Einblick in die verwendeten Quellcodes zu bekommen, finden sich im Folgenden hierfür ergänzende Hinweise und Kommentare. Alle Programme wurden mit Eclipse SDK Version 3.2.2 und dem C/C++ Plugin CDT (C/C++ Development Tools) entwickelt. Es wurde der C/C++ Compiler gcc in Version 4.1.3 mit O3-Option verwendet. Um die Menge an Files übersichtlich zu halten, wurden Präprozessor-Switches verwendet, mit denen sich steuern lässt, welche Programmversion übersetzt werden soll. Alle Programmteile, die zur Berechnung der LCP-Tabelle dienen, sind ebenfalls bedingt übersetzbar und mit den entsprechenden Direktiven visuell abgesetzt worden.

A.1 Larsson/Sadakane

FastSufSort.cpp

Zeile 4-24

Diese Präprozessor-Direktiven steuern verschiedene Programmabläufe und entsprechende Programmversionen. Die einzelnen Versionen lassen sich in den Zeilen 4-8 übersetzen, wobei mit *FASTSUFSORT_BASIC* der Original-SAKA übersetzt wird, mit *FASTSUFSORT_KASAI* eine Version die im Anschluss an die Konstruktion den Algorithmus von Kasai aufruft und mit *FASTSUFSORT_LCP* die Version mit der parallelen LCP-Berechnung übersetzt.

Zeile 123-135

Hier befinden sich globale Variablen, die für die LCP-Berechnung relevant sind. *BH* entspricht dem *BH*-Feld und *SA_h* dem *CSA*-Feld aus Kapitel 5.

Zeile 139-206

Dies sind die beiden Algorithmen 5.6 und 5.7 zur Berechnung der LCP-Werte und der Aktualisierung des *BH*-Feldes. In Zeile 174 wird festgelegt, ob am Ende des Algorithmus noch ein Durchlauf stattfinden muss, weil es noch undefinierte Positionen in der LCP-Tabelle gibt. Dies zeigt *openLCP* an und wird in Zeile 837 vor jeder Phase zurückgesetzt.

Zeile 208-232

Verifizierungsfunktion für die LCP-Tabelle, die über Zeile 8 gesteuert werden kann.

Zeile 333-350

Dies ist der Aufruf des Kasai-Algorithmus und dessen Zeitmessung. Das inverse Suffix-Array wurde vom Basis-Algorithmus schon bereitgestellt, deswegen erfolgt hier keine explizite Berechnung dieses Feldes.

Zeile 375-381

Speicherfreigabe aller Felder die zur LCP-Berechnung benötigt wurden.

Zeile 394-398

Mit *bh* wird die Position des aktuellen Bucketkopfes bereitgestellt und in *CSA* an dieser Position der erste Suffixindex gespeichert, bevor dieser eventuell überschrieben wird, falls ein Einzelbucket vorliegen sollte. Die Variable *act_group* speichert die alte Gruppennummer, um bei dynamischen Schlüsseln den alten Bucketkopf zu kennen. Damit lässt sich dann später testen, ob zwei Suffixe aus dem selben Bucket stammen und somit die LCP-Berecnung zurückgestellt werden muss. Siehe dazu Abschnitt 5.7.

Zeile 401-451

Dieser Abschnitt dient dazu im Falle einer Symbolaggregation die LCP-Werte für die erste Phase zu berechnen. Ob dies möglich ist hängt von der Alphabet- und Eingabegröße ab und für Details sei auf die Originalliteratur von Larsson/Sadakane verwiesen.

Zeile 468-474

Für das letzte Suffixe eines Buckets wird hier in *CSA* der Suffixindex gespeichert. In Zeile 471 wird die Verzeigerung aus Abb. 5.10 für alle entstandenen Kindbuckets hergestellt. Es wird auf den alten Bucketkopf mit definiertem LCP-Wert verwiesen.

Zeile 632-704

Hier werden während des initialen Bucketsorts die LCP-Werte an den Bucketgrenzen gesetzt. Falls mit $r > 0$ eine Symbolaggregation stattgefunden hat müssen r-Vergleiche gemacht werden, um den korrekten Wert zu erhalten. Das zu diesem Zeitpunkt noch unbenutzte, aber reservierte Feld *BH*, wird zur Zwischenspeicherung der Eingabe verwendet, um diese Vergleiche durchzuführen. In den Zeilen 647-648 wird der LCP-Wert an den Bucketgrenzen auf 0 gesetzt, da mit $r = 0$ keine Symbolaggregation vorliegt. Zeile 660 speichert an der aktuellen Position den gerade vorliegenden Suffixindex in *CSA*. Das *BH*-Feld wird nun mit den vorliegenden Werten in *LCP* initialisiert

Zeile 730-737

Die Eingabe wird nach *BH* kopiert, um bei Symbolaggregation die ersten r Symbole zweier adjazenter Suffixe vergleichen zu können, da die Eingabe danach zerstört wird.

Zeile 795-806

Speicherallokation für *CSA* und *BH*, sowie Setzen der initialien Werte für diese beiden Felder und *LCP*.

Zeile 858-870

Nach jeder Phase werden die beiden Algorithmen aus 5.6 und 5.7 aufgerufen. Falls *open-LCP* noch gesetzt ist, müssen die restlichen LCP-Werte noch in einem letzten Durchlauf gesetzt werden, siehe Zeile 867-870.

Kasai.cpp

Dies ist die direkte Implementierung des Kasai-Algorithmus 4.2.

A.2 Kärkkäinen/Sanders

skew.cpp

Zeile 13-35

Diese Präprozessor-Direktiven steuern verschiedene Programmabläufe und entsprechende Programmversionen. Die einzelnen Versionen lassen sich in den Zeilen 13-15 übersetzen, wobei mit *Basic* der Original-SAKA übersetzt wird, mit *KASAI* eine Version die im Anschluss an die Konstruktion den Algorithmus von Kasai aufruft und mit *WITH_LCP* die Version mit der parallelen LCP-Berechnung . Mit den Zeilen 17-18 lässt sich steuern welches RMQ-Verfahren verwendet werden soll, wobei genau eins aktiviert werden muss.

Zeile 40-86

Hier befinden sich Hilfsfunktionen, die im Ablauf des Algorithmus öfters aufgerufen werden. Die Funktion *smybComp* vergleicht die ersten beiden Symbole zweier übergebenen Suffixe und gibt den entsprechenen LCP-Wert zurück. Die zweite Funktion *getPos12* liefert die Indexpositionen aus s^{12} zurück, wie sie in den Fallunterscheidungen von Abschnitt 6.5 besprochen wurden.

Zeile 137-139

Speicherallokation für das LCP^{12}-Feld für jeden rekursiven Aufruf.

Zeile 143-147

Da in der Originalversion die rekursiven Aufrufe ohne das Feld LCP^{12} stattfinden, wurde die Parameterliste mit diesem Eintrag erweitert. Wird keine parallele LCP-Berechnung gewünscht, so wird als Ersatz einfach ein NULL-Pointer übergeben. Dies sollte keine weitere Auswirkung auf die Laufzeit haben.

Zeile 153-170

Direkte Implementierung des Algorithmus 6.1.

Zeile 198-281

Implementierung des Algorithmus 6.2. Zunächst wird in 202-205 das inverse Suffix-Array für SA^{12} erstellt und in 208-214 werden die Konstruktoren für den gewählten RMQ-Algorithmus initialisiert. Die Zeilen 231-238 behandeln den ersten Fall direkt, ohne auf die Funktion *getPos12* zurückzugreifen. Dies geschieht aus Performancegründen, da man hier Modulo-Berechnungen einsparen kann. In 240-272 werden die beiden anderen Fälle bearbeitet, wobei je nach gewählter RMQ-Methode der entsprechende Programmabschnitt übersetzt wird. In Zeile 273 wird schließlich noch auf das nächste Suffix umgeschaltet.

Zeile 302-327

Hier handelt es sich um eine Verifikationsroutine, die die berechnete LCP-Tabelle auf Korrektheit überprüft. Dies geschieht in naiver Weise mit einem expliziten Suffix-Vergleich und sollte deshalb nur bei nicht allzu großen Eingaben mit nicht zu großem durchschnittlichen LCP-Wert angewendet werden. In Zeile 322 lässt sich noch steuern, ob man bei eventueller Fehlerausgabe alle Fehler sehen möchte. In dieser Implementierung wird die Funktion beim ersten Fehler, falls vorhanden, verlassen. Die Aktivierung dieser Funktion lässt über den Switch aus Zeile 22 steuern.

Zeile 332-353

Direkte Umsetzung des Kasai-Algorithmus 4.2.

Zeile 358-468

Die stellt den Treiber für den Skew-Algorithmus dar. Aufgerufen wird das Programm mit *skew [file]* und das Laden der Eingabe geschieht in den Zeilen 365-385. Es folgt die Speicherreservierung für die Eingabe, das Suffix-Array und, falls gewünscht, die LCP-Tabelle. In Zeile 402 wird das Dummy-Tripel $$$ an das Ende der Eingabe gehängt. In 405-416 wird das Eingabealphabet kompaktiert, wie es der Skew-Algorithmus technisch verlangt. Diese Routine ist dem Algorithmus von Larsson und Sadakane entnommen, wo dies auch verlangt wird. Es folgen Initialisierungen für die Zeitmessung und in 424-428 erfolgt der Sortieraufruf.

Hier wird wieder unterschieden, ob die LCP-Tabelle mitberechnet werden soll. Falls nicht, wird anstelle des Feldes ein NULL-Pointer übergeben. In 434-464 erfolgt der Aufruf des Kasai-Algorithmus, wobei zunächst das inverse Suffix-Array gebildet werden muss, da dies vom Skew-Algorithmus am Ende nicht zur Verfügung steht. Es folgen noch die beiden Verifikationsverfahren für *SA* und *LCP*, falls man eine Überprüfung wünscht.

rmq.c, rmq.h

Dies ist sind die Implementierungsdateien des Alstrup-Algorithmus von [4]. Die zugrunde liegende Arbeit ist in [3] beschrieben.

RMQ_n_1_improved.cpp
RMQ_n_1_improved.hpp
RMQ_nlogn_1.cpp
RMQ_nlogn_1.hpp
RMQ.hpp

Dies ist sind die Implementierungsdateien des Fischer-Algorithmus von [10]. Die zugrunde liegende Arbeit ist in [11] beschrieben.

A.3 Puglisi/Maniscalco

MSufSort.h

Zeile 11-14

Switches zur Programmauswahl:

- *LCP_COMPUTATION* ist die Version mit LCP-Berechnung
- *LCP_BSTAR_COMPLETION* ist die Implementierung mit der Heuristik aus Abschnitt 7.9
- *LCP_KASAI* ist die Version mit anschließendem Kasai-Aufruf
- *VERIFY_LCP* für eine gewünschte Verifizierung der LCP-Tabelle

Werden alle Switches deaktiviert, wird die Originalimplementierung übersetzt.

Zeile 162-194

Deklaration aller Membervariablen und Methoden für die LCP-Berechnung. *m_LCP_RMQ* ist das Feld für LCP-Verzeigerung aus Abb. 7.10. *m_ptr_LCP_BStar* ist die temporäre LCP-Tabelle für alle Suffxie aus S. Alle Anfangspositionen einer TSQS zugeführten ZW-Partition werden in der Variablen *m_leftmost_partition_pos* gespeichert.

Zeile 389-409

Diese Methode gibt den Index des Minimums aus einem LCP-Bereich zurück mit gleichzeitiger Pfadkomprimierung und entspricht Algorithmus 7.6.

Zeile 411-423

Während der Sortierung der S-Suffixe vergleicht diese Methode das aktuell platzierte Suffix mit seinem direkten Vorgänger. Damit lässt sich eine dynamische Schlüsselverwendung feststellen und wird nur für diese Variante verwendet.

Zeile 425-450

Gibt den LCP-Wert zweier Suffixe durch einen Direktvergleich zurück. Da alle Suffixindizes einen ZW-Wert besitzen, kann man diesen Vergleich mit jeweils zwei Symbolen durchführen.

MSufSort.cpp

Zeile 36-41

Speicherbereinigung für alle verwendeten Felder.

Zeile 87-93

Die LCP-Tabelle wird mit der Länge der Eingabe initialisiert, um damit undefinierte Werte anzuzeigen und damit eine RMQ eines Bereichs den korrekten Wert zurückgibt.

Zeile 139-142, 150-160

In die umgebende Schleife wurde die Berechnung aller LCP-Werte aus [0..1] eingebettet, so wie in Algorithmus 7.3 gezeigt.

Zeile 734-745

Hier findet das Setzen der LCP-Werte an Partitionierungsgrenzen von TSQS statt, wie in Algorithmus 7.4 dargestellt. Dies wurde nur für das normale TSQS implementiert und die andere Variante von Puglisi und Maniscalco wurde deaktiviert.

Zeile 746-757

Für die Technik aus Abschnitt 7.9 muss dieses Setzen der LCP-Werte so geschehen, dass für beide adjazente Suffixe eine Präfixlänge bestimmt wird, sodass ab dieser Position Suffixe aus S stehen. Falls diese Position nicht existiert, so wurden die Suffixe durch mindestens ein Suffix aus $U \backslash S$ oder V voneinander getrennt.

Zeile 807-814

Setzen der LCP-Werte an Partitionsgrenzen bei Insertion Sort. Auch hier die gleiche Unterscheidung wie vorstehend beschrieben.

Zeile 980-998

Deklaration der Listenelemente und Initialisierung des Feldes mit den verketteten Listen aus Abschnitt 7.9.

Zeile 1015-1077

Die umgebende Schleife setzt die Suffixe aus S wie in 7.2 beschrieben. In Zeile 1017 wird für jedes gesetzte Suffix gleich der Eintrag im inversen Suffix-Array gesetzt. In 1030 wird der LCP-Wert für zwei adjazente Suffixe aus dem temporären LCP-Feld LCP_S übernommen. Es folgt der Test, ob der LCP-Wert noch zu klein ist und ob an der enstprechenden Präfix-Position die Technik aus Abschnitt 7.9 angewendet werden kann. Dies ist die Implementierung von Algorithmus 7.8.

Zeile 1083-1118

Hier werden die RMQ-Anfragen behandelt. In 1093-1099 wird die Verzeigerung für jeden definierten LCP-Wert hergestellt. In 1102-1116 wird dann abgefragt, ob die aktuelle Position in SA eine linke Grenze einer RMQ darstellt. Dies wird geprüft über das Feld mit den verketteten Listen, und falls diese nicht leer ist, werden alle Anfragen nacheinander bearbeitet.

Zeile 1146-1175

Hier wird im RLD geprüft, ob noch ein expliziter Suffixvergleich gemacht werden muss, um sicherzustellen, dass an der aktuellen Position der LCP-Wert korrekt vorliegt. Dieser wird später für folgende RMQ-Anfragen benötigt.

Zeile 1204-1211

Nachdem ein Suffix aus U an seine Position verschoben wurde, wird in Zeile 1205 der Eintrag in SA^{-1} vorgenommen. Es folgt das Berechnungsschema aus Abschnitt 7.7, falls an dieser Position noch kein LCP-Wert vorliegt.

Zeile 1254-1275

Dies ist die Behandlung der besprochenen Ausnahmesituation aus Abschnitt 7.12, in der Suffixe aus V und U gegenüberliegen und für die noch kein LCP-Wert berechnet wurde. Es erfolgt der explizite Suffixvergleich und im Anschluss die Verzeigerung der LCP-Werte für die aktuelle Position im LRD.

Zeile 1704-1710

Hier wird das temporäre LCP-Feld der Größe $|S|$ erstellt, das alle LCP-Werte adjazenter Suffixe aus S beinhalten wird, nachdem TSQS diese sortiert hat.

Zeile 1718-1721

Für jede an TSQS weitergeleitete ZW-Partition wird der Zeiger auf das temporäre LCP-Feld LCP_S neu justiert und die Anfangsposition festgehalten.

Zeile 1821-1852

Die Implementierung des Kasai-Algorithmus, wobei hier noch die Vorberechnung von SA^{-1} erfolgen muss, da am Ende das Basisalgorithmus diese Information nicht vorliegt.

Literaturverzeichnis

[1] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. The enhanced suffix array and its applications to genome analysis. In *WABI '02: Proceedings of the Second International Workshop on Algorithms in Bioinformatics*, pages 449–463, London, UK, 2002. Springer-Verlag.

[2] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *J. of Discrete Algorithms*, 2(1):53–86, 2004.

[3] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: a survey and a new distributed algorithm. In *SPAA '02: Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 258–264, New York, NY, USA, 2002. ACM.

[4] Hideo Bannai. Implementierung des RMQ-Algorithmus von Alstrup. Online im Internet, 2005. http://tlas.i.kyushu-u.ac.jp/~bannai/software/misc.

[5] Michael A. Bender and Martin Farach-Colton. The lca problem revisited. In *LATIN '00: Proceedings of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94, London, UK, 2000. Springer-Verlag.

[6] Jon L. Bentley and M. Douglas McIlroy. Engineering a sort function. 23(11):1249–1265, 1993.

[7] Jon L. Bentley and Robert Sedgewick. Fast algorithms for sorting and searching strings. In *SODA '97: Proceedings of the eighth annual ACM-SIAM symposium on Discrete algorithms*, pages 360–369, Philadelphia, PA, USA, 1997. Society for Industrial and Applied Mathematics.

[8] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[9] M. Farach. Optimal suffix tree construction with large alphabets. In *FOCS '97: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS '97)*, page 137, Washington, DC, USA, 1997. IEEE Computer Society.

[10] Johannes Fischer. Implementierung des RMQ-Algorithmus von Fischer. Online im Internet, 2006. http://www.bio.ifi.lmu.de/~fischer/newRMQ.tgz.

[11] Johannes Fischer and Volker Heun. Theoretical and practical improvements on the rmq-problem, with applications to lca and lce. In *CPM*, pages 36–48, 2006.

[12] Johannes Fischer and Volker Heun. A new succinct representation of rmq-information and improvements in the enhanced suffix array. In *ESCAPE*, pages 459–470, 2007.

[13] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, January 1997.

[14] Hideo Itoh and Hozumi Tanaka. An efficient method for in memory construction of suffix arrays. In *SPIRE '99: Proceedings of the String Processing and Information Retrieval Symposium & International Workshop on Groupware*, page 81, Washington, DC, USA, 1999. IEEE Computer Society.

[15] Juha Kärkkäinen and Peter Sanders. Simple linear work suffix array construction. In Jos C.M. Baeten, Jan Karel Lenstra, Joachim Parrow, and Gerhard J. Woeginger, editors, *Automata, languages and programming : 30th International Colloquium, ICALP 2003*, volume 2719 of *Lecture Notes in Computer Science*, pages 943–955, Eindhoven, The Netherlands, 2003. Springer.

[16] Juha Kärkkäinen, Peter Sanders, and Stefan Burkhardt. Linear work suffix array construction. *J. ACM*, 53(6):918–936, 2006.

[17] Richard M. Karp, Raymond E. Miller, and Arnold L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *STOC '72: Proceedings of the fourth annual ACM symposium on Theory of computing*, pages 125–136, New York, NY, USA, 1972. ACM.

[18] Toru Kasai, Gunho Lee, Hiroki Arimura, Setsuo Arikawa, and Kunsoo Park. Linear-time longest-common-prefix computation in suffix arrays and its applications. In *CPM '01: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, pages 181–192, London, UK, 2001. Springer-Verlag.

[19] Dong Kyue Kim, Jeong Seop Sim, Heejin Park, and Kunsoo Park. Linear-time construction of suffix arrays. In *CPM*, pages 186–199, 2003.

[20] Pang Ko and Srinivas Aluru. Space efficient linear time construction of suffix arrays. In *CPM*, pages 200–210, 2003.

[21] Stefan Kurtz. Reducing the space requirement of suffix trees. *Softw. Pract. Exper.*, 29(13):1149–1171, 1999.

[22] N. Jesper Larsson and Kunihiko Sadakane. Faster suffix sorting. Technical Report LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999), Department of Computer Science, Lund University, Sweden, May 1999.

[23] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[24] Maniscalco, M.A., Puglisi, S.J. Faster lightweight suffix array construction. In *AWOCA 2006. Proceedings of 17th Australasian Workshop on Combinatorial Algorithms*, pages 16–29, 2006.

[25] Giovanni Manzini. Two space saving tricks for linear time lcp array computation. In *SWAT*, volume 3111 of *Lecture Notes in Computer Science*, pages 372–383. Springer, 2004.

[26] M. Douglas McIlroy. ssort.c. Online im Internet, 1997. http://www.cs.dartmouth.edu/~doug/ssort.c.

[27] Yuta Mori. Divsufsort. Online im Internet, 2005. http://homepage3.nifty.com/wpage/software/libdivsufsort.html.

[28] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2):4, 2007.

[29] Uwe Schöning. *Algorithmik*. Spektrum, Akademischer Verlag, 2001.

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Selbständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe.

Ulm, den 14. Mai 2008

_____

Jürgen Reiss

Einverständniserklärung

Ich erkläre hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek der Universität Ulm ausgestellt werden darf.

Ulm, den 14. Mai 2008

_____

Jürgen Reiss