Technische Universität Dortmund
Fakultät für Informatik

Das Enhanced Suffix Array

Seminar *Algorithm Engineering 2009*

Veranstalter:
Prof. Dr. Petra Mutzel
Bernd Zey

Betreuer:
Wolfgang Paul

von
Kai Winnekens
Sommersemester 2009

Abstract

Die Bioinformatik erzeugt in ihren Problemstellungen hohe Datenaufkommen, darunter viele Sequenzvergleiche (die als lange Zeichenketten interprtiert werden können), und stellt damit hohe Anforderungen an Algorithmen und Datenstrukturen, die dieser Herausforderung gerecht werden müssen. Für diesen Zweck wurden anfangs der Suffix-Baum und später das Suffix Array entwickelt. Jedoch hat der Suffix-Baum im Vergleich zum Suffix Array keine gute Platzkomplexität, und das Suffix Array hat im Vergleich zum Suffix-Baum keine gute Zeitkomplexität.

Diese Ausarbeitung stellt das *Enhanced Suffix Array* vor. Es versucht die Vorteile des Suffix-Baumes und des Suffix Arrays zu vereinigen. Wir werden sehen, wie sich eine effiziente Index-Datenstruktur aufbauen lässt, die – analog zum Suffix-Baum – Suchanfragen auf einer Zeichenkette S für ein Muster P in Zeit $O(|P| \cdot \log |\Sigma|)$ beantwortet und dabei nur $O(|S|)$ Platz braucht (mit kleinen Konstanten).

Inhaltsverzeichnis

1 Einleitung

Der Suffix-Baum hat sich in der Vergangenheit als effiziente Datenstruktur zur Lösung vieler Probleme auf Zeichenketten hervorgetan. Er findet Anwendung in vielen verschiedenen Anwendungsgebieten der Informatik, beispielweise bei der Beantwortung der Frage *„Ist P eine Teilzeichenkette von S?“ (Pattern-Matching)*, beim Finden der längsten gemeinsamen Teilzeichenkette zweier Zeichenketten oder auch beim Finden sich wiederholender Teilzeichenketten. Letzteres findet direkte Anwendung bei der Kompression von Daten, ersteres beim Suchen von interessanten Stellen in einem Text. Ein besonderes Anwendungsgebiet mit ebenso besonderen Anforderungen stellt jedoch die Bioinformatik dar: Sie stellt die Aufgabe, Muster in DNA- bzw. Proteinsequenzen zu finden, die man als lange Zeichenketten darstellen kann. Die enorme Länge solcher DNA-Sequenzen (mehrere Millarden Zeichen) stellt besondere Anforderungen an eine geeignete Datenstruktur. Sie muss sowohl eine gute Zeit-, als auch eine gute Platzkomplexität aufweisen.

Der Suffix-Baum einer Zeichenkette S, $n = |S|$, kann in $O(n)$ berechnet und gespeichert werden. Auf einer solchen Datenstruktur kann das *Pattern-Matching-Problem* für ein Muster P, $m = |P|$, in Zeit $O(m)$ gelöst werden. Außerdem können alle z Vorkommen eines Musters P in Zeit $O(m + z)$ berechnet werden. Obwohl diese Zeiten asymptotisch optimal sind, findet der Suffix-Baum in der Praxis keine große Anwendung: Erstens ist die Konstante in der Platzkomplexität mit $20n$ (s. [1]) ziemlich groß, zweitens hat der Suffix-Baum den Nachteil, dass er auf Prozessoren mit Cache an Effizienz verliert.

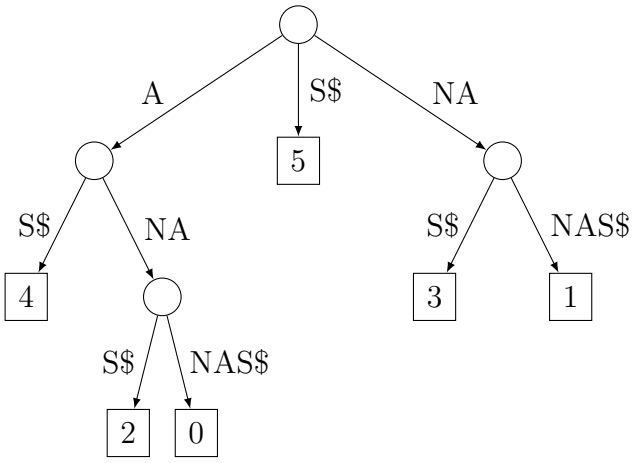


Abbildung 1: Suffix-Baum zur Zeichenkette S = ANANAS$. Die Pfade von der Wurzel zu den Blättern entsprechen den Suffixen von S, die Blätter sind mit den Anfangsindizes der entsprechenden Suffixe in S versehen.

Als eine Platz-effizientere Alternative bietet sich das *Suffix Array* an. Es benötigt in seiner grundlegenden Form nur $4n$ Byte. Es hat jedoch auf den ersten Blick zwei Nachteile: Erstens benötigt die direkte Konstruktion eines Suffix Arrays $O(n \log n)$ Zeit. Zweitens braucht es Zeit $O(m \log n)$, um das *Pattern-Matching-Problem* zu lösen.

Indem wir erst den Suffix-Baum und daraus das Suffix Array konstruieren, können wir für die Konstruktion auch eine Zeitkomplexität von $O(n)$ erreichen. Trotzdem zeigen Experimente, dass der verbesserte $O(n \log n)$ Algorithmus nach Larsson und Sadakane [3] zur direkten Konstruktion des Suffix Arrays in der Praxis effizienter ist als die indirekte $O(n)$ Konstruktion über den Suffix-Baum. Manber and Myers konnten 1993 zeigen [4], dass das *Pattern-Matching-Problem* mit einem Suffix Array in Zeit $O(m + \log n)$ gelöst werden kann.

i	`suftab`	entspr. Suffix
0	**0**	ANANAS$
1	**2**	ANAS$
2	**4**	AS$
3	**1**	NANAS$
4	**3**	NAS$
5	**5**	S$
6	**6**	$

Tabelle 1: Das *Suffix Array* zu $S =$ ANANAS$

Das *Enhanced Suffix Array (ESA)* vereint schließlich die Vorteile beider Datenstrukturen. M. I. Abouelhoda, E. Ohlebusch und S. Kurtz konnten 2002 zeigen, dass für das *Pattern-Matching-Problem* bzw. das Suchen aller Vorkommen eines Musters P in einem Text mit Hilfe eines um zwei weitere Tabellen erweiterten *Suffix Arrays* eine *worst-case* Zeit $O(m)$ respektive $O(m + z)$ erreicht werden kann. Dabei kann jedes Array in linearer Zeit berechnet werden und benötigt nur einen Platz von $4n$ Byte. Weiterhin konnte gezeigt werden, dass jeder Top-Down-Durchlauf eines Suffix-Baumes durch einen Durchlauf eines ESAs ersetzt werden kann. Damit ist das Verfahren nicht nur auf das *Pattern-Matching-Problem* beschränkt.

Was wir unter einem *Enhanced Suffix Array* zu verstehen haben und wie es in Anwendungen eingesetzt werden kann, soll Gegenstand der nachfolgenden Betrachtungen sein.

2 Grundlegende Begriffe

Sei S eine Zeichenkette über einem geordneten Alphabet Σ. Wir bezeichnen die Länge von S mit $n = |S|$. Um die weiteren Betrachtungen zu erleichtern, nehmen wir $n < 2^{32}$ an. Somit können wir jede Position $i \in [0, n-1]$ in S in einem Integer (4 Byte) speichern. Wir nehmen weiter an, dass es ein Zeichen \$ gibt, das lexikographisch größer ist als alle anderen Zeichen und nicht in Σ vorkommt, somit $\Sigma \cap \{\$\} = \emptyset$. Mit $S[i]$ bezeichnen wir das Zeichen in S an

Position i. Für eine Teilzeichenkette von S, die an Position $i \geq 0$ startet und bei j mit $i \leq j < n$ endet, schreiben wir $S[i \ldots j]$.

Das Suffix Array `suftab` speichert die Startpositionen aller $n+1$ Suffixe von $S\$$, und zwar in lexikographisch aufsteigender Reihenfolge. Es ist daher vom Typ *Integer* und kann mit $4n$ Byte gespeichert werden. Den leeren Suffix haben wir bisher ignoriert. Dies ist gerechtfertigt, steht er in der lexikographischen Ordnung doch immer vor allen anderen Suffixen und würde damit in `suftab` immer den gleichen Platz einnehmen. Schreiben wir $S_i = S[i \ldots n-1]$ für den i-ten nicht-leeren Suffix von $S\$$, so erhalten wir mit $S_{\texttt{suftab[0]}}$, $S_{\texttt{suftab[1]}}$, $\ldots$, $S_{\texttt{suftab[n]}}$ also alle Suffixe von $S\$$ in lexikographisch aufsteigender Reihenfolge. Für die Konstruktion des Suffix Arrays sind grundsätzlich zwei verschiedene Ansätze bekannt: Die Berechnung erfolgt (1) direkt in Zeit $O(n \log n)$ [4] oder (2) indirekt über den Aufbau des Suffix-Baumes in Zeit $O(n)$ [2].

Zusätzlich zum grundlegenden Suffix Array `subtab` speichern wir eine Tabelle der längsten gemeinsamen Präfixe (engl. *longest common prefix*), `lcptab`. Dazu definieren wir $\texttt{lcptab}[0] = 0$, für $1 \leq i \leq n$ ist $\texttt{lcptab}[i]$ die Länge des längsten gemeinsamen Präfixes von $S_{\texttt{suftab}[i-1]}$ und $S_{\texttt{suftab}[i]}$. Wir setzen auch $\texttt{lcptab}[n] = 0$, da ohnehin immer $S_{\texttt{suftab}[n]} = \$$ ist. Da jeder Suffix aus weniger als 2^{32} Zeichen besteht, können wir auch `lcptab` mit insgesamt $4n$ Byte speichern. Die Konstruktion von `lcptab` erhalten wir als Nebenprodukt während der Konstruktion von `suftab`.

3 lcp-Intervalle

Wir haben am Ende von Kapitel 2 das Array `lcptab` der längsten gemeinsamen Präfixe kennengelernt. Da die Anfangsindizes der Suffixe in `suftab` in lexikographischer Reihenfolge gepeichert sind, können wir das Gesamtintervall [0,n] so in Teilintervalle aufteilen, dass alle darin enthaltenen Suffixe einen gemeinsamen Präfix haben. Jedes dieser Teilintervalle könnte weiteraufgeteilt werden, so dass sich weitere Teilintervalle von Suffixen mit längerem gemeinsamen Präfix ergeben. Insgesamt führt uns dieses Vorgehen zum so genannten *lcp-Intervall Baum*, der dem bereits bekannten Suffix-Baum ähnelt. Jedoch bleiben [i,i]-Intervalle in einem lcp-Intervall Baum implizit, so dass diesem die Blätter des entsprechenden Suffix-Baumes fehlen.

Definition 1. *Ein Intervall* $[i,j]$, $0 \leq i < j \leq n$, *heißt lcp-Intervall mit lcp-Wert* α, *falls gilt:*

1. $\texttt{lcptab}[i] < \alpha$
2. $\texttt{lcptab}[k] \geq \alpha$ *für alle* k *mit* $i+1 \leq k \leq j$

3. $\texttt{lcptab}[k] = \alpha$ *für mindestens ein* $k \in [i+1,j]$

4. $\texttt{lcptab}[j+1] < \alpha$

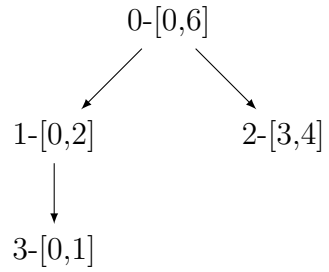
Im Folgenden schreiben wir für ein lcp-Intervall $[i,j]$ mit lcp-Wert α auch einfach α-Intervall oder α-$[i,j]$. Jeder Index $k \in [i+1,j]$ mit $\texttt{lcptab}[k] = l$ heißt α-Index. Ein α-Intervall $[i,j]$ heißt ω-Intervall, falls $\omega = S[\texttt{subtab}[i] \ldots \texttt{subtab}[i] + \alpha - 1]$ der längste gemeinsame Präfix von $S_{\texttt{subtab}[i]}, S_{\texttt{subtab}[i+1]}, \ldots, S_{\texttt{subtab}[j]}$ ist.

Die Beziehungen zwischen den lcp-Intervallen definieren gemäß der vorangegangenen Überlegungen einen *lcp-Intervall Baum*.

Definition 2. *Ein β-Intervall $[l,r]$ heißt enthalten in einem α-Intervall $[i,j]$, falls es ein Teilintervall von $[i,j]$ ist (also $i \leq l < r \leq j$) und $\beta > \alpha$. Das α-Intervall $[i,j]$ heißt dann das $[l,r]$ umschließende Intervall. Falls $[i,j]$ das Intervall $[l,r]$ umschließt und es kein Intervall gibt, das sowohl in $[i,j]$ enthalten ist als auch $[l,r]$ umschließt, dann heißt $[l,r]$ ein Kind-Intervall von $[i,j]$.*

i	suftab	lcptab	entspr. Suffix
0	0	**0**	ANANAS$
1	2	**3**	ANAS$
2	4	**1**	AS$
3	1	**0**	NANAS$
4	3	**2**	NAS$
5	5	**0**	S$
6	6	**0**	$

(a) Das Suffix Array mit den längsten gemeinsamen Präfixen



(b) Der aus lcptab resultierende lcp-Intervall Baum

Abbildung 2: Suffix Array mit lcp-Tabelle und dem resultierenden lcp-Intervall Baum

Abbildung 2(b) zeigt den lcp-Intervall Baum, der aus Definition 2 hervorgeht. Dieser Baum ist jedoch ein virtueller Baum. Seine explizite Berechnung ist nicht Teil des Konzeptes von Enhanced Suffix Arrays. Wir werden später sehen, wie wir zu jedem Intervall stets in konstanter Zeit seine Kind-Intervalle berechnen können.

Die Wurzel des lcp-Intervall Baumes bildet immer das 0-Intervall $[0,n]$. Er entspricht im Wesentlichen dem bekannten Suffix-Baum, jedoch fehlen ihm im Gegensatz dazu die Blätter, sie bleiben hier nur implizit. In einem

Suffix-Baum entspricht ein Blatt einem Suffix $S_{\texttt{suftab}[l]}$. Projiziert auf den entsprechenden lcp-Intervall Baum würde das dem Intervall $[l,l]$ entsprechen. Dessen Elter-Intervall ist klar definiert: Es ist das kleinste lcp-Intervall $[i,j]$ mit $l \in [i,j]$. Allgemein können die Kind-Intervalle eines Intervalls gemäß dem folgenden Lemma berechnet werden.

Lemma 3. *Sei $[i,j]$ ein α-Intervall. Sind $i_1 < i_2 < \ldots < i_k$ die α-Indizes in aufsteigender Reihenfolge, dann sind die Kind-Intervalle von $[i,j]$ gerade $[i,i_1-1], [i_1,i_2-1], \ldots, [i_k,j]$. Einige der Kind-Intervalle können auch einelementig sein.*

Beweis. Sei $[l,r]$ eines der Intervalle $[i,i_1-1], [i_1,i_2-1], \ldots, [i_k,j]$. Falls $l = r$ ist, dann muss $l = r = i$ oder $l = r = j$ gelten und somit ist $[l,r]$ ein Kind-Intervall von $[i,j]$. Ist hingegen $[l,r]$ ein β-Intervall, dann muss es in $[i,j]$ enthalten sein, da $[l,r]$ keinen α-Index enthält. Da $\texttt{lcptab}[i_1] = \texttt{lcptab}[i_2] = \ldots = \texttt{lcptab}[i_k] = \alpha$ kann es kein Intervall geben, das sowohl in $[i,j]$ enthalten ist als auch $[l,r]$ umschließt. Also ist $[l,r]$ nach Definition 2 ein Kind-Intervall von $[i,j]$. Folglich sind die Intervalle $[i,i_1-1], [i_1,i_2-1], \ldots, [i_k,j]$ auch genau die Kind-Intervalle von $[i,j]$. □

Wir betrachten hier noch einmal Abbildung 2 und darin das 1-Intervall [0,2]. In diesem Intervall gibt es nur einen 1-Index, nämlich 2. Nach Lemma 3 sind die Kind-Intervalle daher [0,1] und [2,2]. Da wir einelementige Intervalle im lcp-Intervall Baum nicht darstellen, entspricht dieses Ergebnis der Darstellung in Abbildung 2(b).

Wir haben nun die Analogie zwischen dem Suffix-Baum und dem lcp-Intervall Baum kennengelernt. Diese Analogie besteht jedoch erst einmal nur in den Äußerlichkeiten. Aber können wir damit auch ähnlich gute Ergebnisse erzielen, was die Zeitkomplexität betrifft? Die Antwort ist „Ja“, falls wir zu einem Intervall dessen Kindintervalle in konstanter Zeit bestimmen können. Dazu benötigen wir eine weitere Tabelle, die Gegenstand des nächsten Kapitels sein soll.

4 Das Enhanced Suffix Array

Im Folgenden sollen die Unterschiede zum bisher bekannten Suffix Array näher beleuchtet werden.

4.1 Die Kinder-Tabelle

Das *Enhanced Suffix Array* unterscheidet sich von seiner Standard-Version in zwei Tabellen, nämlich (1) der Tabelle der längsten gemeinsamen Präfixe `lcptab` (bereits in Kapitel 2 kennengelernt) und (2) der Kind-Tabelle `cldtab`. Diese Kind-Tabelle umfasst $n+1$ Einträge, die über einen Index $i \in [0,n]$ adressiert werden können. Jeder dieser Einträge speichert die drei Werte `up`, `down` und `next`α`Index`. Für jedes der Felder brauchen wir insgesamt $4n$ Byte, da wir Inidizes in S darin speichern wollen. Wir werden später sehen, dass wir trotzdem alles in einem Array speichern können und so mit $4n$ statt $12n$ Byte für alle drei Felder zusammen auskommen.

Sei $[i,j]$ das längste lcp-Intervall, das an Position j endet und sei α dessen lcp-Wert. Dann speichern `up`$[j+1]$ und `down`$[i]$ den kleinsten α-Index von $[i,j]$. In `next`α`Index`$[i]$ speichern wir den Anfangsindex des nächsten Geschwisterintervalls, und zwar nur, falls $[i,j]$ nicht das letzte Kind des Elter-Intervalls ist. Formal können wir dies wie folgt definieren:

$$\begin{aligned}
\texttt{cldtab}[i]\texttt{.up} &= \min\{q \in [0,i-1] \mid \texttt{lcptab}[q] > \texttt{lcptab}[i] \\
&\quad \text{und } \forall k \in [q+1,i-1] : \texttt{lcptab}[k] \geq \texttt{lcptab}[q]\} \\
\texttt{cldtab}[i]\texttt{.down} &= \max\{q \in [i+1,n] \mid \texttt{lcptab}[q] > \texttt{lcptab}[i] \\
&\quad \text{und } \forall k \in [i+1,q-1] : \texttt{lcptab}[k] > \texttt{lcptab}[q]\} \\
\texttt{cldtab}[i]\texttt{.next}\alpha\texttt{Index} &= \min\{q \in [i+1,n] \mid \texttt{lcptab}[q] = \texttt{lcptab}[i] \\
&\quad \text{und } \forall k \in [i+1,q-1] : \texttt{lcptab}[k] > \texttt{lcptab}[i]\}
\end{aligned}$$

Für jedes α-Intervall mit α-Indizes $i_1 < i_2 < \ldots < i_k$ können wir also mittels `up` bzw. `down` den ersten α-Index i_1 bestimmen und mittels `next`α`Index` auch die restlichen Indizes $i_2, \ldots, i_k$. Also können wir gemäß Lemma 3 zu einem Intervall alle seine Kindintervalle bestimmen. Zur Verdeutlichung der Zusammenhänge erweitern wir unser Beispiel aus Abbildung 2.

i	`suftab`	`lcptab`	1^1	2^1	3^1
0	0	**0**		2	3
1	2	**3**			
2	4	**1**	1		
3	1	**0**	2	4	5
4	3	**2**			
5	5	**0**	4		6
6	6	**0**			

Tabelle 2: Das Enhanced Suffix Array mit `cldtab` zu $S =$ ANANAS.

Um die Kind-Intervalle des 1-Intervalls [0,2] zu bestimmen, schauen wir nun in `cldtab`[0]`.down` oder `cldtab`[3]`.up`. Wir erhalten als ersten 1-Index den Index 2. Da `cldtab`[2]`.next`α`Index` leer ist, gibt es keine weiteren Kind-Intervalle. Nach Lemma 3 sind die Kind-Intervalle also [0,1] und [2,2], was unseren vorheri-

[1] Die Zahlen bezeichnen die Komponenten der Tabelle `cldtab`: 1 steht für `up`, 2 steht für `down` und 3 für `next`α`Index`.

gen Überlegungen entspricht. Das nächste Kapitel wird klären, wie die Kind-Tabelle effizient berechnet werden kann.

4.2 Berechnung der Kinder-Tabelle

Im letzten Kapitel wurde die Kinder-Tabelle als Bestandteil von Enhanced Suffix Arrays eingeführt. Ziel dieses Kapitels ist es, die Berechnung dieser Tabelle zu klären. Dazu betrachten wir in Analogie zu [1] zwei Algorithmen jeweils für die Werte `up/down` und `next`α`Index`. In beiden Algorithmen kommt ein Stack zum Einsatz. Im Folgenden bezeichnen die Funktionen *push* und *pop* die bei Stacks bekannten Operationen, d.h. *push* legt ein Element auf dem Stack ab und *pop* nimmt das oberste Element vom Stack und gibt es zurück. Mit *peek* bezeichnen wir das oberste Element auf dem Stack.

Listing 1: Berechnen der `up`- und `down`-Werte (nach [1]) in Java

```
1  Stack<Integer> stack = new Stack<Integer>();
2  int lastIndex = -1;
3  stack.push(0);
4  for(int i=1; i<lcptab.length; i++) {
5    while(lcptab[i] < lcptab[stack.peek()]) {
6      lastIndex = stack.pop();
7      if(lcptab[i] <= lcptab[stack.peek()] && lcptab[stack.
           peek()] != lcptab[lastIndex])
8        down[stack.peek()] = lastIndex;
9    }
10   if(lcptab[i] >= lcptab[stack.peek()]) {
11     if(lastIndex != -1) {
12       up[i] = lastIndex;
13       lastIndex = -1;
14     }
15     stack.push(i);
16   }
17 }
```

Algorithmus 1 durchläuft das Array `lcptab` in aufsteigender Reihenfolge. Da das erste Intervall immer bei $i = 0$ beginnt, legen wir in Zeile 3 die 0 auf den Stack. Falls `lcptab`$[i] \geq$ `lcptab`$[top]$ ist wissen wir, dass wir uns aktuell innerhalb eines Intervalls befinden und legen i für spätere Vergleiche auf den Stack. Ist gleichzeitig noch `lastIndex` $\neq -1$, so haben wir nach Definition 1 eine rechte Intervallgrenze überschritten (Zeilen 10 – 15): Für jedes α-Intervall $[i,j]$ muss `lcptab`$[k] \geq \alpha$ gelten für alle $i+1 \leq k \leq j$. Da in Zeile 10 `lcptab`$[i] \geq$ `lcptab`$[top]$ ist können wir folgern, dass mit dem Index

i ein neues Geschwisterintervall beginnt und setzen folglich `cldtab`$[i]$`.up` = `lastIndex`.

Wir kommen zu dem Teil, der die `down`-Werte berechnet: Falls in Zeile 5 `lcptab`$[i]$ < `lcptab`[`top`] wahr ist, haben wir wieder eine Intervallgrenze überschritten und nehmen den obersten Index vom Stack, der gemäß Fall Nr. 4 aus Definition 1 ein α-Index ist. Ist zusätzlich `lcptab`$[i] \leq$ `lcptab`[`top`], so wissen wir, dass wir nicht die Grenze zu einem weiteren Kind-Intervall überschritten haben, sondern zu einem neuen Geschwister-Intervall und setzen folglich `cldtab`[`top`]`.down` = `lastIndex`. Abouelhoda et al. präsentieren in [1] einen längeren, formalen Beweis auf den wir hier seiner Länge wegen nicht näher eingehen wollen.

Es bleibt noch die Berechnung der `next`α`Index`-Werte zu klären. Diese gestaltet sich etwas einfacher.

Listing 2: Berechnen der `next`α`Index`-Werte (nach [1]) in Java

```
1  Stack<Integer> stack = new Stack<Integer>();
2  int lastIndex = -1;
3
4  stack.push(0);
5  for(int i=1; i<lcptab.length; i++) {
6    while(lcptab[i] < lcptab[stack.peek()])
7      stack.pop();
8    if(lcptab[i] == lcptab[stack.peek()]) {
9      lastIndex = stack.pop();
10     nextlIndex[lastIndex] = i;
11   }
12   stack.push(i);
13 }
```

Auch hier legen wir anfangs wieder eine 0 auf den Stack und laufen in aufsteigender Reihenfolge durch `lcptab`. Zu jedem Index i mit `lcptab`$[i] = 0$ suchen wir einen Index $j > i$ mit `lcptab`$[j] = 0$ und setzen `cldtab`$[i]$`.next`α`Index` = j. Ebenso verfahren wir für alle l und r mit $i < l < r < j$.

Wie bereits in Kapitel 4.1 angedeutet, können wir die gesamte Tabelle `cldtab` mit nur $4n$ Byte speichern. Dazu betrachten wir zuerst die `up`- und `down`-Felder. Wir erzeugen Redundanz, wenn wir beide vollständig abspeichern, da viele der `up`- und `down`-Einträge auf die gleichen Indizes verweisen. Man betrachte dazu ein α-Intervall $[i,j]$ mit k α-Indizes. Ein solches Intervall kann höchstens $k+1$ Kind-Intervalle $[l_q,r_q]$ $(1 \leq q \leq k+1)$ haben, wobei $[l_q,r_q]$ ein α_q-Intervall und i_q den ersten α_q-Index eines solchen Intervalls bezeichnen. Wir können den Index i_q in `cldtab`$[r_q + 1]$`.up` speichern. Somit bleibt nur noch i_{k+1} übrig, was wir in `cldtab`$[r_k + 1]$`.down` speichern müssen.

Hier kommt aber zum Tragen, dass $[l_{k+1},r_{k+1}]$ das letzte Kind-Intervall von $[i,j]$ ist und somit speichern wir für $r_k + 1 = l_{k+1}$ kein `next`α`Index`, gemäß der Definition von `next`α`Index`. Bleibt nun also noch sicherzustellen, dass wir auch entscheiden können, ob ein `next`α`Index`-Eintrag wirklich den nächsten α-Index bezeichnet, oder aber den Wert für `cldtab`$[i]$`.down`. Aber auch hier können wir wieder auf die Definition von `next`α`Index` zurückgreifen. Falls `cldtab`$[i]$`.next`α`Index` den nächsten α-Index speichert, muss demnach nämlich `lcptab`[`cldtab`$[i]$`.next`α`Index`]= `lcptab`$[i]$ gelten und andernfalls `lcptab`[`cldtab`$[i]$`.next`α`Index`] > `lcptab`$[i]$. Außerdem können wir die `cldtab`$[i+1]$`.up` in `cldtab`$[i]$`.next`α`Index` speichern, da letzteres immer leer ist, falls `cldtab`$[i+1]$`.up` nicht leer ist. Es verbleibt zu entscheiden, ob ein in `next`α`Index` gespeicherter Wert den nächsten α-Index oder aber einen `up`-Wert bezeichnet. `cldtab`$[i+1]$`.up` ist jedoch nur dann nicht leer, falls `lcptab`$[i]$ > `lcptab`$[i+1]$.

4.3 Berechnung der Kind-Intervalle in konstanter Zeit

Abouelhoda et al. präsentieren in [1] ein wichtiges Lemma, welches die Grundlage zur Berechnung von Kind-Intervallen in konstanter Zeit bildet. Für einen Beweis von Lemma 4, siehe [1].

Lemma 4. Für jedes α-Intervall $[i,j]$ gelten die folgenden Aussagen:

1. $i <$ `cldtab`$[j+1]$`.up` $\leq j$ oder $i <$ `cldtab`$[i]$`.down` $\leq j$
2. `cldtab`$[j+1]$`.up` speichert den ersten α-Index in $[i,j]$, falls $i <$ `cldtab`$[j+1]$`.up` $\leq j$
3. `cldtab`$[i]$`.down` speichert den ersten α-Index in $[i,j]$, falls $i <$ `cldtab`$[i]$`.down` $\leq j$

Mit Hilfe von Lemma 4 können wir also für ein α-Intervall I in konstanter Zeit den ersten α-Index i_1 von I berechnen. Weiter erhalten wir über `cldtab`$[i_1]$`.next`α`Index` den nächsten α-Index i_2. Wir können dieses Vorgehen für alle α-Indizes $i_1 < i_2 < \ldots < i_k$ fortführen und erhalten so gemäß Lemma 3 die Kind-Intervalle von I, nämlich $[i,i_1-1],[i_1,i_2-1],\ldots,[i_k,j]$.

Bei der Berechnung von Kind-Intervallen genügt es jedoch nicht, nur die Intervallgrenzen zu berechnen. Für ein Kind-Intervall $[i,j]$ brauchen wir ebenso eine Funktion $getlcp(i,j)$, die den lcp-Wert des Intervalls berechnet. Aber auch das gestaltet sich mit Hilfe von Lemma 4 einfach: Falls $i <$ `cldtab`$[j+1]$`.down`, dann gibt $getlcp(i,j)$ den Wert in `lcptab`[`cldtab`$[j+1]$`.up`] zurück, sonst gibt die Funktion den Wert in `lcptab`[`cldtab`$[i]$`.down`] zurück. Im fol-

Listing 3: Die Funktion *getlcp*(i,j) berechnet für ein Kind-Intervall [i,j] dessen lcp-Wert (in Java).

```
1  if(i < up[j+1] && up[j+1] <= j)
2    return lcptab[up[j+1]];
3  else
4    return lcptab[down[i]];
```

genden Kapitel wird es um die Lösung des *Pattern-Matching-Problems* gehen. Dazu benötigen wir zu einem Intervall jedoch nicht immer die gesamte Liste aller Kind-Intervalle, sondern nur das Intervall, dessen Suffixe dem Suchmuster entsprechende Präfixe haben. Wir brauchen eine Funktion, die zu einem α-Intervall [i,j] und einem Zeichen $a \in \Sigma$ genau das Kind-Intervall [l,r] von [i,j] berechnet, dessen Suffixe an der Stelle α das Zeichen a aufweisen. Dabei ist zu beachten, dass alle Suffixe von [l,r] die gleichen Präfixe der Länge $\alpha + 1$ aufweisen, da [l,r] ja ein Kind-Intervall von [i,j] ist. Im Falle [i,j] = [0,n] geben wir das Kind-Intervall von [0,n] zurück, dessen Suffixe an Position 0 das Zeichen a aufweisen, da [0,n] immer ein 0-Intervall ist. Ist hingegen [i,j] $\neq$ [0,n], dann berechnen wir mit Hilfe von Lemma 4 und der soeben beschriebenen Vorgehensweise sukzessive die Kind-Intervalle von [i,j] und testen jeweils die Suffixe der Kind-Intervalle α-[l,r] auf Übereinstimmung des Zeichens an der Position α mit dem Zeichen a.

Listing 4: Die Funktion *getInterval*($i, j, a \in \Sigma$) in Java.

```
1  private LcpInterval getInterval(int i, int j, char a) {
2    int n = 0, m = 0, lcp = 0;
3    if(i < 0 || i >= j) return null;
4    if(i==0 && j==suftab.length) {
5      n=0;
6      while(nextlIndex[n] != 0) {
7        m=nextlIndex[n];
8        if(s.substring(suftab[n]).charAt(0) == a)
9          return new LcpInterval(getLcp(n,m-1), n, m-1);
10       n=m;
11     }
12   } else {
13     //Betrachte erstes Kind-Intervall
14     lcp = getLcp(i, j);
15     if(up[j+1] <= j)
16       n = up[j+1];
17     else
18       n = down[i];
```

```
19      if(s.substring(suftab[i]).charAt(lcp) == a)
20        return new LcpInterval(getLcp(i, n-1), i, n-1);
21      //Sonst suche in den uebrigen Kind-Intervallen weiter
22      while(nextlIndex[n] != 0) {
23        m = nextlIndex[n];
24        if(s.substring(suftab[n]).charAt(lcp) == a)
25          return new LcpInterval(getLcp(n, m-1), n, m-1);
26        n=m;
27      }
28      if(s.substring(suftab[n]).charAt(lcp) == a)
29        return new LcpInterval(getLcp(n, j), n, j);
30    }
31    return null;
32  }
```

5 Pattern-Matching mit einem Enhanced Suffix Array

Die Frage, ob eine Zeichenkette in einer anderen Zeichenkette enthalten ist, lässt sich mit dem „normalen“ Suffix Array in Zeit $O(m \log n)$ beantworten. Manber und Myers [4] konnten zeigen, wie diese Zeit auf $O(m + \log n)$ verbessert werden kann. In beiden Verfahren ist der logarithmische Term auf eine binäre Suche im Suffix Array zurückzuführen. Im Folgenden soll gezeigt werden, dass das Pattern-Matching-Problem auf einem Enhanced Suffix Array in Zeit $O(m)$ und das Auffinden von z Vorkommen einer Zeichenkette in Zeit $O(m + z)$ gelöst werden können.

Listing 5: Die Java-Funktion *search_pattern*($p \in \Sigma^*$) findet alle z Vorkommen einer Zeichenkette in Zeit $O(m + z)$ (Java).

```
1  public void search_pattern(String p) {
2    int c = 0, min = 0, i=0, j=0, lcp=0, m = p.length();
3    LcpInterval interval = getInterval(0, suftab.length, p.
       charAt(c));
4    boolean queryFound = (interval != null);
5
6    while(interval != null && c<m && queryFound) {
7      i = interval.getLb();
8      j = interval.getRb();
9      if(i != j) {
10       lcp = interval.getLcp();
11       min = Math.min(lcp, m);
```

```
12          interval = (min != m ? getInterval(i, j, p.charAt(min
                )) : null);
13          queryFound = s.substring(suftab[i]+c, suftab[i]+min).
                equals(p.substring(c, min));
14          c = min;
15        } else { /* i == j */
16          queryFound = s.substring(suftab[i]+c, suftab[i]+m).
                equals(p.substring(c, m));
17          interval = null;
18        }
19      }
20      if(queryFound) /* Ausgabe aller suftab[k], i <= k <= j */
21  }
```

Algorithmus 5 findet alle z Vorkommen eines Textmusters in einer Zeichenkette wie folgt: Der erste Schritt ist die Bestimmung des Kind-Intervalls I vom Wurzelintervall 0-$[0,n]$, dessen Suffixe auf dem ersten Zeichen (also an Position 0) mit dem ersten Zeichen des Suchmusters P übereinstimmen (Zeile 3). Dabei kann die Funktion $getInterval(p \in \Sigma^*)$ als Ergebnis auch $I = null$ liefern, nämlich dann, wenn es keine Suffixe gibt, deren erstes Zeichen mit dem ersten Zeichen von P übereinstimmen. Zeile 4 stellt sicher, dass die Suche in diesem Fall erfolglos bleibt.

Wir betrachten nun den Fall $I = [i,j]$. Im Falle $i = j$ erhalten wir ein einelementiges Intervall und somit ist P nur dann in S enthalten, wenn P ein Präfix von S_i ist. Falls das einelementige Intervall I kein direktes Kind des Wurzelintervalls ist, gibt es ein α-Intervall I' mit $\alpha > 0$, das sich im Intervall-Baum zwischen dem Wurzelintervall und dem Blatt I befindet. Das heißt, dass wir schon vorher einen Präfix von P mit einem Teil von S verglichen haben. Wieviel bereits verglichen wurde, merken wir uns in der Variablen c und vergleichen nur noch den Suffix von P, der noch nicht abgeglichen wurde (Zeile 16).

Sei nun $I = [i,j]$ ein α-Intervall mit $i < j$ (Zeile 9) und $m = |P|$. Im Falle $\alpha < m$ gleichen wir den gemeinsamen Präfix der Suffixe in I mit $P[c \ldots \alpha - 1]$ ab, wobei wir in c die Anzahl der Zeichen speichern, die wir bereits abgeglichen haben. Stimmen die Präfixe überein, so berechnen wir das Kind-Intervall $I' = \alpha'$-$[i',j']$ von I. Dabei gilt immer $\alpha' > \alpha$, da die lcp-Werte der Intervalle mit zunehmender Tiefe im lcp-Intervall-Baum nur größer werden können. Wir vergleichen dann wieder die verbleibenden Zeichen der gemeinsamen (jetzt längeren) Präfixe der Suffixe mit den verbleibenden Zeichen von P, also $P[c \ldots \alpha' - 1]$. Dieses Verfahren treiben wir so lange weiter, bis schließlich $\alpha > m$ oder $I = null$. Im Falle $\alpha > m$ können wir den verbleibenden Suffix von P mit den verbleibenden Zeichen der Präfixe der Suffixe in I abgleichen.

Entsprechen sich diese, so sind $\texttt{suftab}[i], \texttt{suftab}[i+1], \ldots, \texttt{suftab}[j]$ gerade die Anfangsindizes der Vorkommen von P in S. Ist hingegen $I = null$, dann gibt es keinen Suffix von S, dessen Präfix gerade P entspricht und somit kommt P nicht in S vor.

6 Ein verbessertes Enhanced Suffix Array

In Kapitel 4.3 haben wir bereits die Funktion $getInterval(i, j, a \in \Sigma)$ (Algorithmus 4) kennengelernt und eingesehen, dass diese nur konstante Zeit braucht. Dies gilt jedoch nur, so lange die Alphabetgröße eine Konstante ist. Für ein Alphabet Σ ist $|\Sigma|$ eine obere Schranke für die Anzahl der Kind-Intervalle eines lcp-Intervalls. Dabei tritt der *Worst-Case* ein, falls alle Suffixe an der ersten Stelle mit unterschiedlichen Zeichen beginnen (beispielweise $\Sigma = \{a, b, c, d\}$ und $S =$ abcd). Genauer ist die Laufzeit der Funktion $getIntervall(i, j, a \in \Sigma)$ also $O(|\Sigma|)$ und damit die nötige Zeit zur Lösung des Pattern-Matching-Problems $O(m|\Sigma|)$. Im Gegensatz dazu ist die Suchzeit auf einem Suffix-Baum nur $O(m \log |\Sigma|)$. Dieser Nachteil des Enhanced Suffix Arrays gegenüber dem Suffix Baum macht sich in Experimenten, in denen die Alphabetgröße nicht mehr als vernachlässigbar angesehen werden kann, bemerkbar.

Dong Kyue Kim, Jeong Eun Jeon und Heejin Park konnten 2004 zeigen, wie die Funktion $getInterval(i, j, a \in \Sigma)$ in Zeit $O(\log |\Sigma|)$ ausgeführt werden kann (siehe [5]). Damit konnten sie das Enhanced Suffix Array auch bei großen Alphabeten praktikabel machen. Wir diskutieren im folgenden Kapitel die Unterschiede zu dem Enhanced Suffix Array, das wir bereits kennengelernt haben.

6.1 Die neue Kinder-Tabelle

Die neue Kinder-Tabelle `cldtab` besteht aus vier Arrays der Länge n. Diese sind `up` und `down` und die beiden Arrays `lchild` und `rchild`, durch die wir das Array `next`α`Index` ersetzen. Wir hatten das Array `next`α`Index` dazu benutzt, die Kind-Intervalle in einer Listenstruktur zu verwalten. Wir wollen die beiden neuen Arrays `lchild` und `rchild` nun dazu benutzen, diese Informationen in einer Baumstruktur zu verwalten. Damit bräuchten wir im schlimmsten Fall nicht mehr die gesamte Liste der Kind-Intervalle durchlaufen und somit Zeit $O(|\Sigma|)$, sondern nur noch Zeit $O(\log |\Sigma|)$ zum Durchlaufen der Baumstruktur.

Mit *Wurzel-Kind* eines Intervalls $[i,j]$ bezeichnen wir das Wurzel-lcp-Intervall des Baumes, der durch die Kinder von $[i,j]$ entsteht. Abbildung

3 zeigt den neuen (konzeptuellen) Suffix-Baum nach Park et al. Darin verbindet eine durchgezogene Kante ein lcp-Intervall mit seinem Wurzel-Kind und eine gestrichelte Kante die Geschwister-Intervalle. Projeziert auf unsere Kind-Tabelle `cldtab` heißt das, dass die Arrays `up` und `down` gerade die durchgezogenen Kanten speichern und die Arrays `lchild` und `rchild` die gestrichelten Kanten speichern.

Die einzelnen Arrays sind wie folg definiert: `up`$[i]$ speichert den ersten lcp-Index vom Wurzel-Kind des längsten Intervalls, das bei Index $i-1$ endet. `down`$[i]$ speichert den ersten Index vom Wurzel-Kind des längsten Intervalls, das bei Index i startet. In `lchild`$[i]$ bzw. `rchild`$[i]$ speichern den ersten lcp-Index des linken bzw. rechten Kindes des längsten Intervalls, das bei Index i beginnt. Die Arrays `lchild` und `rchild` speichern also gerade die Geschwisterbeziehungen im Suffix-Baum. Wie schon zuvor brauchen wir auch für die neue Kinder-Tabelle insgesamt nur $4n$ Byte. Um dies zu beweisen reicht es zu zeigen, dass die Anzahl der durchgezogenen und gestrichelten Kanten im Suffix-Baum n beträgt.

Theorem 5. Nur n der insgesamt $4n$ Elemente der Arrays `up`, `down`, `lchild` und `rchild` sind nötig, um den Suffix-Baum zu repräsentieren.

Theorem 6. Die neue Kinder-Tabelle `cldtab` kann in Zeit $O(n)$ berechnet werden.

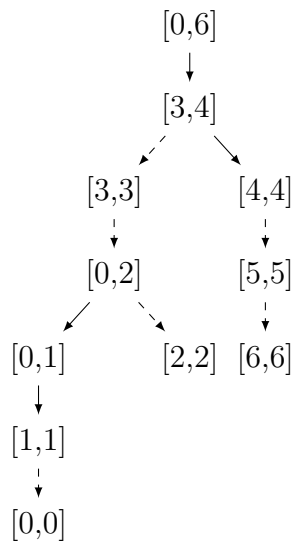


Abbildung 3: Der Suffix-Baum nach Park et al.

Für den etwas länglichen Beweis sei an dieser Stelle auf [5] verwiesen. Um das Pattern-Matching-Problem wirklich in Zeit $O(m \log |\Sigma|)$ lösen zu können, müssen wir nun etwas anders vorgehen als in Algorithmus 4. Die Suche startet hier mit dem Wurzel-Kind $[i,j]$ von $[0,n-1]$. Falls der längste gemeinsame Präfix von $[i,j]$ übereinstimmt mit einem Präfix des Suchmusters, dann steigen wir im Baum ab und fahren mit dem Wurzel-Kind von $[i,j]$ fort. Den entsprechenden Index dazu finden wir in `up`$[j+1]$ oder in `down`$[i]$. Stellen wir hingegen keine Übereinstimmung fest, so machen wir mit einem der Geschwisterintervalle weiter, indem wir `lchild`$[i]$ oder `rchild`$[i]$ benutzen. Dieses Verfahren iterieren wir solange bis wir das Muster gefunden haben oder sicher sind, dass es nicht im Text enthalten ist. Um feststellen zu können, ob ein Element der Kinder-Tabelle zu `up` oder `down` bzw. zu `lchild` oder `rchild` gehört, greifen wir auf die lcp-Tabelle zurück: Wenn `lcptab`$[i]$ = `lcptab`[`cldtab`$[i]$] ist, dann gehört `cldtab`$[i]$ zu `lchild`

oder `rchild`, sonst gehört es zu `up` oder `down`. Somit kann das Pattern-Matching-Problem mit einem Enhanced Suffix Array in Zeit $O(m \log |\Sigma|)$ gelöst werden.

Theorem 7. Mit Hilfe der neuen Kinder-Tabelle `cldtab` und des Arrays `lcptab` lässt sich das Pattern-Matching-Problem in Zeit $O(m \log |\Sigma|)$ lösen.

7 Experimentelle Ergebnisse

Im letzten Kapitel wollen wir die experimentellen Ergebnisse aus [1] und [5] diskutieren. Abouelhoda et al. führen ihre Experimente auf den folgenden Eingaben durch:

1. *ecoli*, eine DNA-Sequenz mit einer Länge von 4.639.221 Zeichen und Alphabetgröße 4.

2. *yeast*, eine DNA-Sequenz mit einer Länge von 12.156.300 und Alphabetgröße 4.

3. *swiss*, eine Sammlung von Proteinsequenzen, deren Gesamtlänge 2.683.054 Zeichen beträgt. Die Alphabetgröße ist 20.

4. *shaks*, eine Sammlung der Werke von William Shakespeare mit einer Gesamtlänge von 5.582.655 Byte. Die Alphabetgröße ist 92.

Abouelhoda et al. schickten ihr Programm *esamatch* ins Rennen. Sie benutzten Algorithmus 5 zur Ausführung der Suche in Zeit $O(m+z)$ und benötigten insgesamt $6n$ Byte. Zum Vergleich wählten sie

1. *stree*, das auf einen Suffix-Baum zurückgreift und damit eine Anfrage in Zeit $O(m + z)$ beantworten kann. Sein Platzverbrauch variiert von 9,6n Byte für *shaks* bis 12,6n Byte für *ecoli*.

2. *mamy*, das auf Suffix Arrays basiert und eine Suchanfrage in Zeit $O(m \log n + z)$ beantwortet. Sein Speicherverbrauch liegt bei insgesamt $4n$ Byte.

Als Testsystem wurde ein Linux-Computer mit 933 MHz Pentium PIII CPU und 512 MB RAM verwendet. Die Programme *stree* und *many* arbeiten ausschließlich im Hauptspeicher, *esamatch* hingegen speichert die einzelnen Tabellen auf der Festplatte und lädt sie bei Bedarf in den Hauptspeicher. Tabelle 3 zeigt die Laufzeiten für 1 Millionen Aufzählunsanfragen, wobei die

	stree	*mamy*	*esamatch*
Datei	Zeit/s	Zeit/s	Zeit/s
ecoli	7,47	5,00	3,32
yeast	9,16	5,35	3,53
swiss	10,47	3,53	3,40
shaks	18,45	3,47	27,14

Tabelle 3: Die Laufzeiten für 1 Millionen Aufzählungsanfragen. Die Eingabelängen lagen gleichverteilt zwischen 30 und 40 Zeichen.

Eingabelänge gleichverteilt zwischen 30 und 40 Zeichen betrug. Das Preprocessing wurde dabei nicht beachtet. Diese Daten bestätigen, dass die Laufzeiten von *stree* und *esamatch* stark von der Alphabetgröße anhängen, im Gegensatz zu *mamy*. Bei kleineren Alphabeten ist *esamatch* jedoch immer mehr als doppelt so schnell wie *stree* und etwas schneller als *mamy*.

Länge	$\|\Sigma\| = 2$	$\|\Sigma\| = 20$	$\|\Sigma\| = 64$	$\|\Sigma\| = 128$
1M	5,05	8,95	16,89	26,80
30M	6,48	17,13	26,47	46,78
50M	6,77	18,02	35,81	54,34

(a) Laufzeiten für das Pattern-Matching im ESA nach Abouelhoda et al. in [s]

Länge	$\|\Sigma\| = 2$	$\|\Sigma\| = 20$	$\|\Sigma\| = 64$	$\|\Sigma\| = 128$
1M	6,60	6,31	6,59	6,41
30M	8,77	10,63	9,50	10,59
50M	8,74	11,56	13,19	12,22

(b) Laufzeiten für das Pattern-Matching im ESA nach Park et al. [s]

Abbildung 4: Vergleich der Laufzeiten für das Pattern-Matching im ESA nach Abouelhoda et al. und im verbesserten ESA nach Park et al.

Schließlich bleibt noch ein Vergleich des verbesserten Enhanced Suffix Arrays von Park et al. mit dem von Abouelhoda et al. (mit der Kind-Tabelle in ihrer ursprünglichen Form). Für den Vergleich benutzen Park et al. zufällig generierte Zeichenketten der Längen 1M, 30M und 50M bei Alphabetgrößen von 2, 4, 20, 64 und 128 Zeichen. Die Tests wurden auf einem Computer mit einem 2,8 GHz Pentium IV Prozessor und 2 GB RAM ausgeführt. Analog zu Abouelhoda et al. wurden auch hier 1 Millionen Suchanfragen gestellt. Aus Abbildung 4 geht deutlich hervor, dass das Pattern-Matching mit dem verbesserte ESA nach Park et al. nahezu unabhängig von der Alphabetgröße

ist, wohingegen die Laufzeiten mit dem ESA nach Abouelhoda et al. mit zunehmender Alphabetgröße deutlich ansteigen.

8 Diskussion

Seit der Entwicklung des Suffix-Baumes und schließlich des Suffix Arrays hat die Entwicklung von Datenstrukturen zur Indizierung von Zeichenketten einen interessanten Verlauf genommen. Das Enhanced Suffix Array stellt besonders für die Bioinformatik eine lohnende Entwicklung dar, benötigt seine Speicherung doch viel weniger Platz als die eines Suffix-Baumes. Für die eher kleinen Alphabete, die in der Bioinformatik üblich sind, vereint es gerade die Vorteile von Suffix-Baum und Suffix Array. Das verbesserte Enhanced Suffix Array erlaubt auch bei großen Alphabeten ähnlich schnelle Algorithmen wie auf einem Suffix-Baum, obwohl es nur etwa 4% mehr Speicherplatz braucht als das „normale“ Enhanced Suffix Array. Die Entwicklung der eigentlichen Datenstrukturen sorgte außerdem für die Entwicklung neuer, effizienter Algorithmen, beispielweise für die Sortierung von Suffixen oder für verschiedenen Aufgaben auf Zeichenketten.

Literatur

[1] Abouelhoda, M. I., E. Ohlebusch und S. Kurtz: *Optimal Exact String Matching Based on Suffix Arrays.* In: *String Processing and Information Retrieval*, Band 2467 der Reihe *Lecture Notes in Computer Science*, Seiten 175 – 180. Springer Verlag Berlin/Heidelberg, 2002.

[2] Gusfield, D.: *Algorithms on Strings, Trees and Sequences.* Cambridge University Press, 1997.

[3] Larsson, N. J. und K. Sadakane: *Faster Suffix Sorting.* Technical Report LU-CS-TR: 99 – 214, Dept. of Computer-Science, Lund University, 1999.

[4] Manber, U. und E. W. Myers: *A New Method for On-Line String Searches.* SIAM Journal on Computing, 22:935 – 948, 1993.

[5] Park, H., D. K. Kim und J. E. Jeon: *An Efficient Index Data Structure with the Capabilities of Suffix Trees and Suffix Arrays for Alphabets of Non-negligible Size.* In: *String Processing and Information Retrieval*, Band 3246 der Reihe *Lecture Notes in Computer Science*, Seiten 138 – 149. Springer Verlag Berlin/Heidelberg, 2004.