

Universität Ulm

Fakultät für Informatik

Abteilung Theoretische Informatik

Exact Optimal String Matching

based on Suffix Arrays

Seminar Bioinformatik

Ausarbeitung von Chi Tai Dang

Sommersemester 2006

{Vorname1Vorname2.Nachname}@uni-ulm.de

Betreuer: Dr. Mohamed I. Abouelhoda

Abgabe: 15.06.2006

Inhaltsverzeichnis

1	Einleitung	1
2	Exact Pattern Matching	1
2.1	Bedingungen in der Bioinformatik	1
3	Bemerkungen zur Notation	2
4	Enhanced Suffix Array	2
4.1	Suffix Tree	3
4.2	Suffix Array	4
4.3	Vom Suffix Tree zum Suffix Array	5
4.4	lcp-Tabelle	5
4.4.1	lcp-Intervall-Baum	8
4.5	Child-Tabelle	9
4.5.1	Konstruktion der Child-Tabelle	10
4.5.1.1	Erzeugung der up/down-Einträge	10
4.5.1.2	Erzeugung der nextIndex-Einträge	12
4.6	Finden von Kind-Intervallen	13
5	Exact Pattern Matching - Enhanced Suffix Array	14
6	Reduzierung der Speicheranforderung	17
6.1	lcp-Tabelle	17
6.2	Child-Tabelle	17
7	Zusammenfassung	20
	Literaturverzeichnis	21

1 Einleitung

Die Suche in einem Text ist eine der traditionellen Aufgabenstellungen in der Informatik, wobei die Randbedingungen sich von Zeit zu Zeit ändern und die Aufgabe dadurch immer wieder neu auftaucht. Je nachdem welche Bedingungen die Aufgabe begleiten, bieten sich unterschiedliche Algorithmen zur Problemlösung an. Für die Anwendung in Browsern oder Texteditoren eignen sich einfache Algorithmen wie Knuth-Morris-Pratt oder Boyer-Moore. Hierbei ist der Eingabetext meist relativ klein und von Suche zu Suche variabel, d.h. von Suche zu Suche ist der Eingabetext mehr oder weniger stark verändert. Anders verhält es sich bei Anwendungen in der Bioinformatik mit DNA-Sequenzen oder auch Proteinsequenzen. Solche Daten können sehr große Dimensionen annehmen, welche sich dann aber nicht mehr ändern. Für eine wiederholte Suche unterschiedlicher Muster in solchen Daten gibt es Algorithmen sowohl zur Indizierung als auch für die eigentliche Suche. Eine sehr bekannte und beliebte Datenstruktur für solche Problemstellungen ist der Suffix Tree und ebenfalls bekannt sind auch Suffix Arrays, wobei letztere i. A. den Ruf haben, dass sie kompliziert und schwer zu verstehen sind. Suffix Arrays benötigen aber im Vergleich zu den Suffix Trees weniger Speicher und genau das macht sie für Anwendungen auf große Datenmengen interessanter. Ganz allgemein kann man einen Suffix Tree sehr effizient sowohl Bottom-Up als auch Top-Down traversieren. Suffix Arrays bieten ebenfalls beide Möglichkeiten mit effizienter Laufzeit an, sodass die Algorithmen auf Suffix Trees systematisch auf Suffix Arrays umgesetzt werden können, wie in [2] gezeigt. Die Anwendungen profitieren dabei von den Vorteilen der Suffix Arrays, wobei in der Bioinformatik die geringere Speicheranforderung der ausschlaggebende Grund sein dürfte. Für einen Top-Down-Traversal mit logarithmischer Laufzeit ist bereits von U. Manber und G. Myers in [5] ein Algorithmus bekannt. Diese Ausarbeitung beschäftigt sich mit der speicherplatzeffizienten Erweiterung eines Suffix Arrays aus [1], um ein Top-Down-Traversal in linearer Laufzeit zu ermöglichen.

2 Exact Pattern Matching

Bei der Problemstellung *Exact Pattern Matching* ist ein String S der Länge n gegeben, in welchem ein Muster P der Länge m gesucht werden soll. Das Muster P kann genau einmal, mehrmals oder gar nicht in S vorhanden sein. Kommt P in S mehr als einmal vor, so bekommt die Häufigkeit den Buchstaben z zugeordnet. Meistens ist man auch an den Positionen in S interessiert an denen das Muster gefunden wurde. Die optimale Laufzeit für dieses Problem ist dabei $O(m + z)$, d.h. das Ergebnis wird unabhängig von der Länge n des Strings S und nur abhängig von der Länge m und der Häufigkeit z des Musters P erwartet.

2.1 Bedingungen in der Bioinformatik

In der Bioinformatik geht es u.a. um Algorithmen und Datenstrukturen auf Daten aus der molekularen Biologie, wie z.B. Genen, Genomen, also DNA¹-Sequenzen oder Protein-

¹DNA ist ein Akronym für *deoxyribonucleic acid*.

sequenzen. Die Daten werden dabei durch einen String aus einer sehr langen Sequenz von Buchstaben eines endlichen Alphabets \mathcal{A} repräsentiert. Bei DNA-Sequenzen werden meist die Anfangsbuchstaben der Nukleobasen Adenin (A), Guanin (G), Cytosin (C), Thymin (T) oder Uracil (U) als Buchstaben von \mathcal{A} verwendet. Analog dazu könnten bei Proteinen die Anfangsbuchstaben der 20 Aminosäuren verwendet werden. Die Größe des Alphabets ist also bei Genomen relativ klein im Vergleich zu Proteinsequenzen, aber in beiden Fällen konstant.

Das Problem *Exact Pattern Matching* findet sich z.B. bei der Suche nach bekannten Genen eines Genoms in einem anderen Genom wieder. Das Ergebnis solch einer Suche kann helfen, die biologische Funktion eines Gens oder Gruppen von Genen zu bestimmen oder auch bekannte Funktionen zu bestätigen. Auf das Genom wird dabei meist eine wiederholte Suche mit mehreren Genen, also Mustern, durchgeführt. Eine besondere Eigenschaft bei vollständig sequenzierten Genomen (Summe der Basenpaare über alle Chromosomen) ist deren Länge. Wie die Tabelle 1 exemplarisch zeigt, können sie sehr große Dimensionen annehmen.

Lebewesen	Basenpaare
Darmbakterium (<i>Escherichia coli</i>)	> 4.600.000
Backhefe (<i>Saccharomyces cerevisiae</i>)	> 20.000.000
Zebrafisch (<i>Danio rerio</i>)	> 1.050.000.000
Mensch (<i>Homo sapiens</i>)	> 3.000.000.000

Tabelle 1: Genomgrößen, NCBI-Datenbank

Bei den gegebenen Bedingungen ist eine vorherige Indizierung des Strings sehr vorteilhaft, da dies den Suchvorgang von der Länge n des Strings unabhängig macht und dadurch stark beschleunigt. Geeignete Algorithmen für das Problem müssen daher die enorme Länge des Eingabestrings berücksichtigen und damit auch geeignete Datenstrukturen mit möglichst niedriger Speicheranforderung für die vorherige Indizierung unterstützen.

3 Bemerkungen zur Notation

Zur Vereinfachung bzw. Abkürzung wird in dieser Arbeit der Begriff String als Bezeichnung für eine Folge von Buchstaben oder Zeichen verwendet. Weiterhin entspricht $S[i]$ dem Buchstaben an der i -ten Stelle im String S , wobei die Stellen in S von Anfang bis Ende mit 0 bis $|S| - 1$ durchnummeriert sind. Ein Suffix S_i ist der Teilstring von S , der ab der Stelle i beginnt und am Ende von S , an der Stelle $|S| - 1$ aufhört. Eine weitere Möglichkeit einen Teilstring anzugeben ist $S[i..i + 5]$, welcher dem Teilstring in S ab der Stelle i mit der Länge 5 entspricht.

4 Enhanced Suffix Array

Suffix Trees und Algorithmen darauf sind schon sehr lange bekannt und sie werden oft eingesetzt um Problemstellungen auf Strings effizient zu lösen. Allerdings ist die hohe

Speicheranforderung von Suffix Trees problematisch für Anwendungen auf sehr großen DNA-Sequenzen. An dieser Stelle kommen die Suffix Arrays ins Spiel, da sie wesentlich kompakter sind und weniger Speicher benötigen. In diesem Abschnitt wird der Suffix Tree und das Suffix Array grob beschrieben. Durch die Erweiterung des Suffix Arrays wird sowohl der Übergang zum Enhanced Suffix Array als auch die dadurch erreichte Analogie zum Suffix Tree dargestellt. Es ist hilfreich sich bereits Vorwissen über Suffix Trees und Suffix Arrays durch weitere Literatur anzueignen.

4.1 Suffix Tree

Formal ist ein Suffix Tree wie folgt definiert.

Definition 1 (Suffix Tree) Sei \mathcal{A} ein Alphabet und sei S ein String der Länge $|S| = n$ über \mathcal{A} . Sei der Buchstabe $\$ \in \mathcal{A}$ und lexikographisch größer als alle anderen Buchstaben in \mathcal{A} . Außerdem kommt der Buchstabe $\$$ nicht im String S vor und bildet als Begrenzungszeichen am Ende von S den String $S\$$. Ein Suffix Tree T für $S\$$ ist ein gerichteter gewurzelter Baum mit folgenden Eigenschaften:

- Jede Kante ist mit einem nicht leeren Teilstring aus $S\$$ markiert.
- Jeder innere Knoten besitzt max. k Kinder mit $2 \leq k \leq |\mathcal{A}|$, deren ausgehende Kanten mit voneinander unterschiedlichen Buchstaben beginnen.
- T besitzt genau $n + 1$ Blätter.
- Die Aneinanderreihung der Teilstrings von der Wurzel zu einem Blatt i entspricht dem Suffix, der in $S\$$ an der Position i beginnt.

Die Abbildung 1 zeigt ein einfaches Beispiel für den String $S\$ = abbab\$$ über dem Alphabet $\mathcal{A} = \{a, b, \$\}$. Der Suffix $S_0 = abbab\$$ entspricht der Konkatination der Kantenmarkierungen von der Wurzel bis zum Blatt mit der Markierung 0. Der Suffix $S_1 = bbab\$$ entspricht der Konkatination der Kantenmarkierungen von der Wurzel bis zum Blatt mit der Markierung 1, usf..

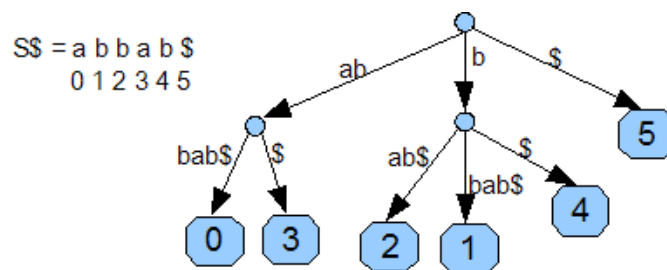


Abbildung 1: Suffix Tree

Für die Konstruktion eines Suffix Trees in linearer Zeit gibt es verschiedene Algorithmen, z.B. in [8], [9] oder [10], die sich alle im zusätzlich benötigten Speicher unterscheiden. Nach der Konstruktion ist für einen Suffix Tree in der Theorie $20n$ Bytes erforderlich, aber

in der Praxis kommt man bei Anwendungen auf DNA-Sequenzen mit $12,5n$ Bytes aus, wie Kurtz 1998 in [11] zeigte. Mit Hilfe eines Top-Down-Traversals ermöglichen Suffix Trees die Suche nach einem Muster in der optimalen Laufzeit von $O(m + z)$. Allerdings wirkt sich die Speicheranforderung mit $20n$ Bytes und die Fragmentierung der Daten² im Speicher bei Anwendungen auf DNA-Sequenzen negativ auf die Performance aus. Da DNA-Sequenzen extrem lang sein können, ist auch der bei der Konstruktion zusätzlich benötigte Speicher nicht unerheblich. Letztgenannte Punkte sind bei Suffix Arrays wesentlich optimaler, zumindest aber nicht so stark ausgeprägt.

4.2 Suffix Array

Das Suffix Array wurde erstmals 1993 von U. Manber und G. Myers in [5] als neue und konzeptionell einfache Datenstruktur zur Lösung des *Exact Pattern Matching* Problems vorgestellt. Im Grunde genommen stellt das Suffix Array eine sortierte Liste aller Suffixe eines Strings $S\$$ dar, welche mit Tabellen für weitere Informationen erweitert werden kann.

Definition 2 Ein Suffix Array **suftab** für einen String S ist ein Array aus ganzzahligen Werten zwischen 0 und n , welche die Startpositionen der lexikographisch sortierten $(n+1)$ Suffixe von $S\$$ (String S mit Begrenzungszeichen $\$$) darstellen.

Die linke Tabelle in 2 zeigt ein einfaches Beispiel für den dort rechts abgebildeten Suffix Tree. Unterhalb der Blätter des Suffix Trees sind die Indexe i der *suftab* des Suffix Arrays skizziert. Für die Konstruktion des Suffix Arrays gibt es Algorithmen, z.B. in [4], [5] oder

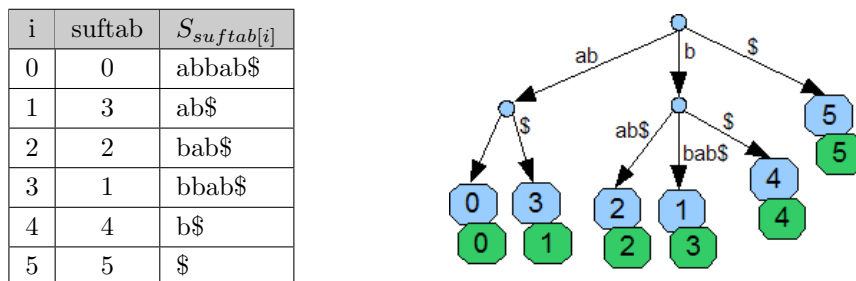


Tabelle 2: Suffix Array

[6], die eine lineare Laufzeit aufweisen. Das Suffix Array benötigt lediglich $4n$ Bytes für die *suftab* und im Vergleich zu Suffix Trees damit um den Faktor 3 bis 5 weniger Speicherplatz. Diese Eigenschaft ist natürlich sehr vorteilhaft bei großen Datenmengen, da es trotz moderner und schneller Computersysteme noch einen großen Unterschied ausmacht, ob eine Suche auf 1GB oder auf 3-5GB durchgeführt wird. Bei solch großen Datenmengen ist es meistens nicht zu umgehen nur einen Teil der Daten im Hauptspeicher zu halten und den Rest auf einem wesentlich langsameren Sekundärspeicher auszulagern. Man kann davon ausgehen, dass der Zugriff auf den Hauptspeicher im Vergleich zum Sekundärspeicher in den meisten Fällen vernachlässigbar klein ist, sodass ein Sekundärspeicherzugriff zum

²wegen der Verkettung von Elementen

merklichen Faktor wird. Je seltener daher auf den Sekundärspeicher zugegriffen werden muss, desto schneller erfolgt auch die Suche im Allgemeinen.

In der Regel ist auch die Zugriffseffizienz auf Suffix Arrays besser. Moderne Computer besitzen meistens eine mehrschichtige Cache-Architektur, bei der die Hardware versucht genügend benötigte Daten aus dem langsamen Hauptspeicher rechtzeitig in einen viel schnelleren Cache³ zu laden. Im Optimalfall wird der Prozessor ununterbrochen mit Instruktionen und Daten aus dem Cache versorgt und muss keine Wartezyklen einlegen. Bei Suffix Trees werden die Elemente mit Zeigern verknüpft und sind deswegen anfällig für eine starke Fragmentierung im Speicher. Im *worst – case* führen sie zu häufigen Cache-Misses⁴. Bei Suffix Arrays liegen die Elemente des Arrays meist dicht beieinander, was bei einem sequentiellen Zugriff zu weniger Cache-Misses führen kann. Allerdings ist kein optimaler Suchalgorithmus für Suffix Arrays bekannt. Manber und Myers legten 1993 einen Algorithmus vor, der das Problem *Exact Pattern Matching* mit einer Laufzeit von $O(m * \log n + z)$ löst. Mit entsprechenden Erweiterungen ist aber auch eine optimale Laufzeit von $O(m + z)$ erreichbar, wobei der Suffix Tree als Vorlage dient.

4.3 Vom Suffix Tree zum Suffix Array

Betrachtet man einen Suffix Tree und das dazugehörige Suffix Array, dann fällt sofort auf, dass lediglich die Blätter des Suffix Trees im Suffix Array abgebildet sind.

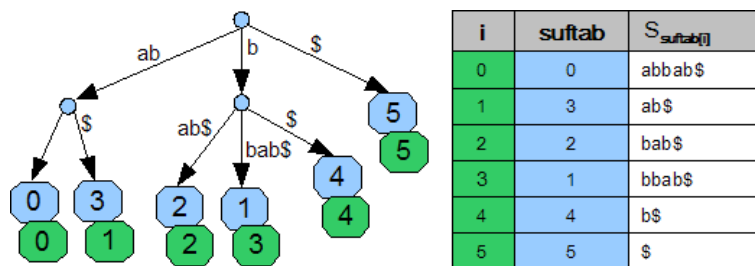


Abbildung 2: Suffix Tree - Suffix Array

Es ist aus der Gegenüberstellung von Suffix Tree und Suffix Array in Abbildung 2 leicht zu sehen, dass die inneren Knoten und Kanten des Suffix Trees nicht im Suffix Array repräsentiert sind. Alle in dieser Ausarbeitung dargestellten Erweiterungen des Suffix Arrays, widmen sich diesem Umstand und bilden die Informationen der inneren Knoten und deren Kanten so platzsparend wie möglich auf geeignete Datenstrukturen ab.

4.4 lcp-Tabelle

Eine erste Erweiterung des Suffix Arrays ist das Array $lcptab$, welches als lcp-Tabelle (lcp = longest common prefix) bezeichnet wird. Jeder Eintrag $lcptab[i]$ speichert die Länge des längsten gemeinsamen Präfix zwischen zwei aufeinander folgenden Suffixen $S_{sufstab[i-1]}$ und

³Abkürzung für Cache-Speicher

⁴Ein Cache-Miss tritt auf, wenn Daten nicht im Cache vorhanden sind und aus dem Hauptspeicher in den Cache befördert werden müssen.

$S_{sufstab[i]}$. Da der erste Suffix im Suffix Array keinen Vorgänger besitzt wird $lcptab[0] = 0$ definiert. Die Abbildung 3 zeigt dies am String $S\$ = acaacatat\$$, welcher von hier an für alle weiteren Betrachtungen als Beispiel beibehalten wird.

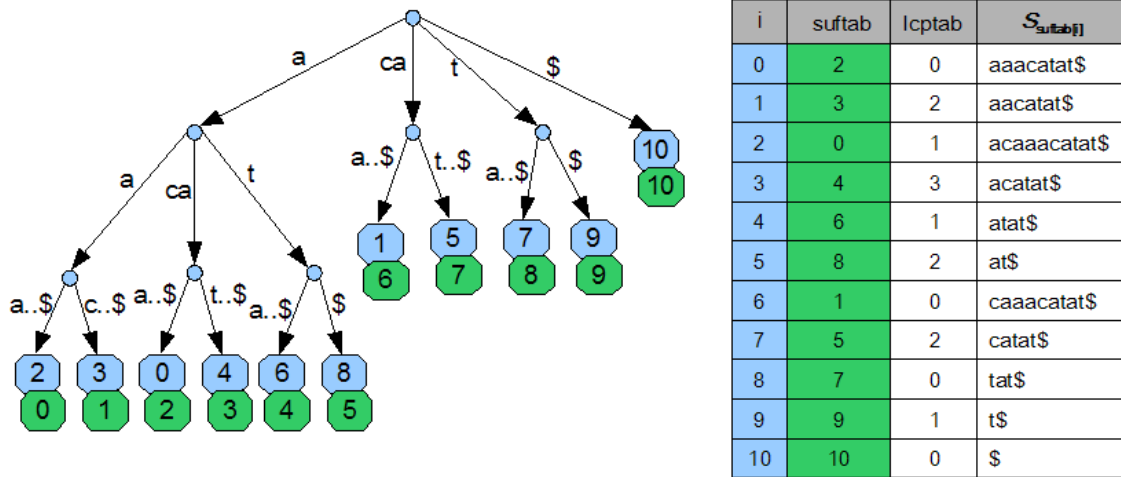


Abbildung 3: Erweiterung lcp-Tabelle

Wenn man $i = 1$ annimmt und die Suffixe an Position 0 ($S_{sufstab[i-1]}$) und an Position 1 ($S_{sufstab[i]}$) betrachtet, so ist klar, dass die Suffixe $aaacatat\$$ und $aacatat\$$ einen gemeinsamen Präfix aa der Länge 2 besitzen und somit $lcptab[1]$ den Wert 2 erhalten muss. Die lcp-Tabelle kann entweder während der Konstruktion des Suffix Arrays oder auch danach in linearer Zeit erzeugt werden, wie in [12] gezeigt.

Ein lcp-Wert von 0 wie an den Positionen 0, 6, 8 und 10 bedeutet, dass das Suffix an der Position mit dem vorherigen Suffix (falls vorhanden) kein einziges Zeichen als Präfix gemeinsam haben. Nachdem die Suffixe im Array lexikographisch sortiert sind, kann im Array $lcptab$ die Anzahl der lcp-Werte = 0 höchstens $|A|$ betragen. Bezieht man diese Betrachtung auf einen Suffix Tree, würde bei jedem lcp-Wert von 0 eine neue Kante vom Knoten ausgehen, da laut Definition alle Kanten im Suffix Tree mit voneinander unterschiedlichen Zeichen beginnen. Wie man in Abbildung 3 sehen kann, gehen von der Wurzel aus genauso viele Kanten ab, wie 0-Werte in $lcptab$ vorhanden sind. Betrachtet man weiterhin die Teilbäume, die (vom Wurzelknoten aus) an den abgehenden Kanten hängen, so wird man feststellen, dass die Blätter eines Teilbaums diejenigen Suffixe enthalten, die im Suffix Array vom „zugehörigen“ lcp-Wert = 0 zum nächsten lcp-Wert = 0 aufeinander folgen. z.B. zwischen $lcptab[8] = 0$ bis $lcptab[10] = 0$ liegen die Suffixe $tat\$$ und $t\$$, welche auch die Blätter des Teilbaums sind, der an der mit t markierten abgehenden Kante hängt.

Genauer gesagt gehören im Suffix Array die Suffixe eines bestimmten Bereichs zu einem zuordbaren Teilbaum im Suffix Tree. Außerdem werden die Bereiche von lcp-Werten getrennt und das führt uns zur Definition der lcp-Intervalle.

Definition 3 (lcp-Intervalle) Ein Intervall $[i..j]$ der lcp-Tabelle wird als lcp-Intervall des Wertes l bezeichnet, wenn

- $lcptab[i] < l$

- $lcptab[k] \geq l, \forall k \text{ mit } i < k < j + 1$
- $lcptab[k] = l, \exists k \text{ mit } i < k < j + 1$. Diese werden als **l -Indices** bezeichnet.
- $lcptab[j + 1] < l$

Das lcp-Intervall wird $l-[i..j]$ geschrieben.

Als Beispiel betrachten wir die in Abbildung 4 farbig hinterlegten Intervalle. Sowohl das Intervall $1-[0..5]$ als auch die Intervalle $2-[0..1]$, $3-[2..3]$ und $2-[4..5]$ erfüllen die Bedingungen der Definition. Es ist auch leicht zu sehen, dass letztere von den Intervallen $0-[0..10]$ und $1-[0..5]$ eingeschlossen werden und die Suffixe einen gemeinsamen Präfix mit einer Länge größer 1 besitzen. Ein Intervall umfasst somit Suffixe, die einen gemeinsamen Präfix der Länge l besitzen. Das gemeinsame Präfix entspricht im Suffix Tree den Kantenmarkierungen von der Wurzel bis zu einem Teilbaum.

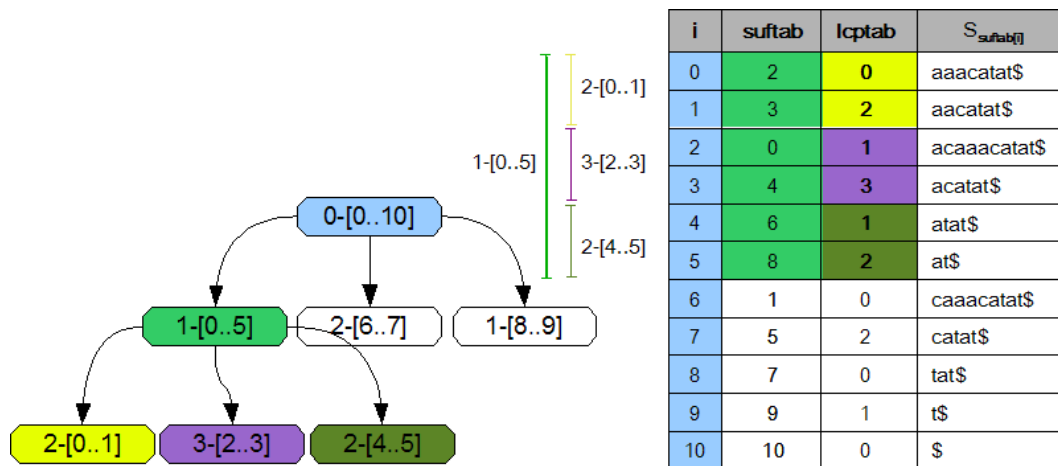


Abbildung 4: lcp-Intervalle

Die lexikographische Sortierung und die Definition eines lcp-Intervalls lässt also eine bestimmte Schachtelung der Intervalle zu, wobei eine genaue Eltern-Kind-Beziehung folgendermaßen definiert wird.

Definition 4 (Kind-Intervalle) Seien $m-[l..r]$ und $q-[i..j]$ zwei lcp-Intervalle.

- Das Intervall $m-[l..r]$ wird als eingebettetes Intervall von $q-[i..j]$ oder $q-[i..j]$ als das umschließende Intervall von $m-[l..r]$ bezeichnet, wenn $i \leq l < r \leq j$ und $m > q$.
- Das Intervall $m-[l..r]$ ist ein Kind-Intervall von $q-[i..j]$, wenn $m-[l..r]$ von keinem anderen eingebetteten Intervall in $q-[i..j]$ umschlossen wird.
- Ein Intervall $[i..i]$ wird als Singleton-Intervall bezeichnet.

Anhand dieser Definition können wir in Abbildung 4 genau drei Kind-Intervalle vom Intervall $1-[0..5]$ identifizieren. Diese Kind-Intervalle werden durch die l -Indices an den Positionen 2 und 4 getrennt und die Zeile mit einem l -Index bildet das erste Element

eines Kind-Intervalls. Letzteres gilt nicht für das erste Kind-Intervall. Die vorherige Betrachtung bei den Intervallen gilt analog auch bei den Kind-Intervallen, d.h. bezieht man die Kind-Intervalle auf einen Suffix Tree, würde bei einem l -Index eine neue ausgehende Kante entstehen. Wenn wir das Intervall $1-[0..5]$ als inneren Knoten ansehen, würden somit von diesem drei Kanten entsprechend der drei Kind-Intervalle ausgehen, welche mit dem gemeinsamen Präfixzeichen der Kind-Intervalle (ohne die bereits „abgelegten“ gemeinsamen Präfixzeichen bis zur Position l im Suffix) markiert sind. Eine besonders wichtige Feststellung der bisherigen Betrachtungen wird durch folgendes Lemma beschrieben.

Lemma 1 Sei $[i..j]$ ein l -Intervall und $i_1 < i_2 < \dots < i_k$ die l -Indices in aufsteigender Anordnung, dann sind die Kind-Intervalle von $[i..j]$ gleich $[i..i_1 - 1], [i_1..i_2 - 1], \dots, [i_k..j]$.

Beweis 1 Sei $[l..r]$ eines der Kind-Intervalle $i_1 < i_2 < \dots < i_k$. Wenn $[l..r]$ ein Singleton-Intervall ist, dann ist $[l..r]$ auch ein Kind-Intervall von $[i..j]$. Angenommen $[l..r]$ ist ein eingebettetes m -Intervall laut Definition 4, d.h. $[l..r]$ enthält keinen l -Index von $[i..j]$ außer an l . Da die l -Indices gleich $lcptab[i_1] = lcptab[i_2] = \dots = lcptab[i_k] = l$ sind, gibt es kein eingebettetes Intervall in $[i..j]$, das $[l..r]$ umschließt, d.h. $[l..r]$ ist ein Kind-Intervall von $[i..j]$. Wir betrachten z.B. das Intervall $[6..7]$ aus der Abbildung 4. Ein mögliches umschließendes Intervall wäre $[0..8]$, aber $[0..8]$ ist laut Definition 4 kein Kind-Intervall von $[0..10]$, da die erste Bedingung ($m > q$) verletzt wird. Es ergeben sich zu $[i..j]$ also nur die Kind-Intervalle mit $[i..i_1 - 1], [i_1..i_2 - 1], \dots, [i_k..j]$, da es keine anderen geben kann.

4.4.1 lcp-Intervall-Baum

Die bisherigen Definitionen erlauben das Herausstellen der Kind-Intervalle, deren Einbettungsbeziehungen zueinander und somit eine baumartige Darstellung mit den einzelnen Intervallen wie in Abbildung 4 auf der linken Seite zu sehen ist. Diese wird als lcp-Intervall-Baum bezeichnet und entspricht praktisch einem Suffix Tree ohne die Blätter wie die Gegenüberstellung in Abbildung 5 zeigt.

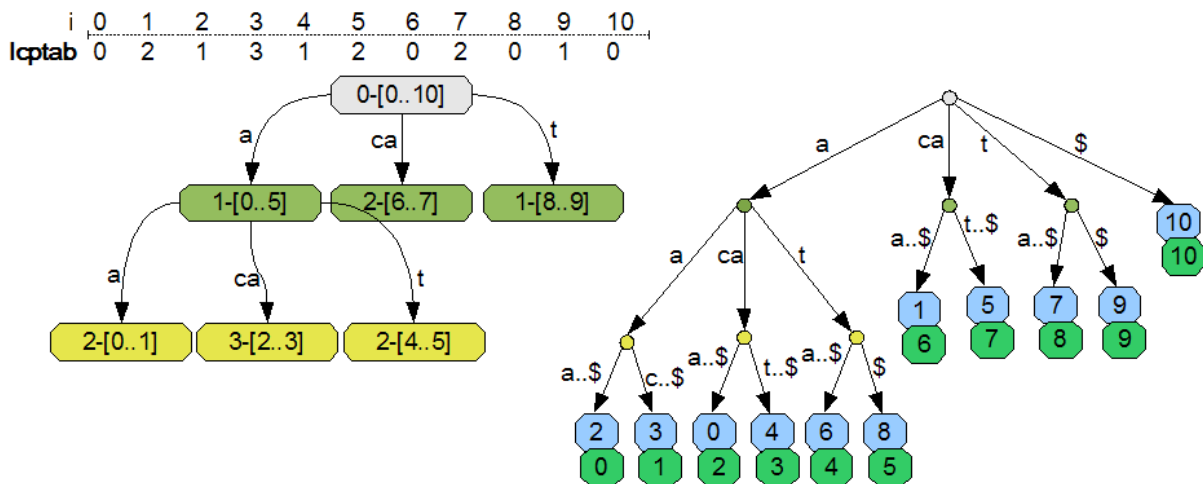


Abbildung 5: lcp-Intervalle

Die Blätter des Suffix Trees sind jedoch implizit durch die Singleton-Intervalle gegeben. Für die Konstruktion des lcp-Intervall-Baums wird als Wurzelknoten das Intervall

0-[0.. n] definiert und die Einbettungen der Kind-Intervalle durch Kanten weitergeführt. Im lcp-Intervall-Baum werden vom Suffix Tree die inneren Knoten durch lcp-Intervalle, die Eltern-Kind-Beziehungen durch deren Einbettung und die Kantenmarkierungen durch Teilstrings gemeinsamer Präfixe repräsentiert.

In der Regel wird der lcp-Intervall-Baum nicht direkt konstruiert, sondern es wird ein Bottom-Up-Traversal des Baums anhand der lcp-Tabelle simuliert, wie in [7] im Algorithmus 3 gezeigt. Beim Durchlaufen der lcp-Tabelle in Abbildung 5 von Position 0 bis n unter Beobachtung der Intervallgrenzen wird zuerst das Kind-Intervall 2-[0..1] und danach das Kind-Intervall 3-[2..3] gefunden. Frühestens nach dem letzten Kind-Intervall 2-[4..5] kann das umschließende Intervall 1-[0..5] gefunden werden, da erst bei Position 6 ein lcp-Wert kleiner als 1 gefunden wird. Laut Definition der lcp-Intervalle hört das Intervall genau dann auf, wenn die Bedingung $lcptab[j + 1] < l$ erfüllt ist. Insgesamt führt das Verfahren daher zu einem Bottom-Up-Traversal. Im Algorithmus wird dabei eine Methode implementiert, die für das gefundene (Kind-)Intervall entsprechende Anweisungen durchführt. Die lcp-Tabelle genügt also für ein Bottom-Up-Traversal durch den zugehörigen Suffix Tree, aber für manche Probleme wie *Exact Pattern Matching* wäre ein Top-Down-Traversal wünschenswerter. U. Manber und G. Myers stellten 1993 in [5] einen Algorithmus vor, der anhand einer binären Suche einen Top-Down-Traversal mit einer logarithmischen Laufzeit ermöglicht.

4.5 Child-Tabelle

Um in linearer Laufzeit den lcp-Intervall-Baum und damit den zugehörigen Suffix Tree mit einem Top-Down-Verfahren traversieren zu können, sind die Eltern-Kind-Beziehungen vom Elternknoten aus notwendig. Bei der Betrachtung der lcp-Intervalle wurde bereits in Lemma 1 festgestellt, dass die l -Indices eines Intervalls die eingeschlossenen Kind-Intervalle trennen, sodass alle Kind-Intervalle eines gegebenen Intervalls mit dessen l -Indices identifiziert werden können. Daher wird das Suffix Array um die Child-Tabelle $cldtab$ der Größe n erweitert, welche 3 Einträge für jede Position i zwischen 0 und n zur Verfügung stellt und Verweise auf die l -Indices aufnimmt.

Definition 5 Für ein lcp-Intervall q -[$i..j$] enthält $cldtab[i].down$ und $cldtab[j+1].up$ die Position des ersten l -Index und $cldtab[i].nextlIndex$ die Position des nächsten l -Index. Die formale Definition dieser Werte lautet:

$$\begin{aligned} cldtab[i].up &= \min\{q \in [0..i - 1] \mid lcptab[q] > lcptab[i] \wedge \forall k \in [q + 1..i - 1] : lcptab[k] \geq lcptab[q]\} \\ cldtab[i].down &= \max\{q \in [i + 1..n] \mid lcptab[q] > lcptab[i] \wedge \forall k \in [i + 1..q - 1] : lcptab[k] > lcptab[q]\} \\ cldtab[i].nextlIndex &= \min\{q \in [i + 1..n] \mid lcptab[q] = lcptab[i] \wedge \forall k \in [i + 1..q - 1] : lcptab[k] \geq lcptab[i]\} \end{aligned}$$

Man beachte, dass sich der Eintrag $cldtab[j + 1]$ nicht im gleichen Intervall q -[$i..j$] sondern im nächsten Intervall befindet.

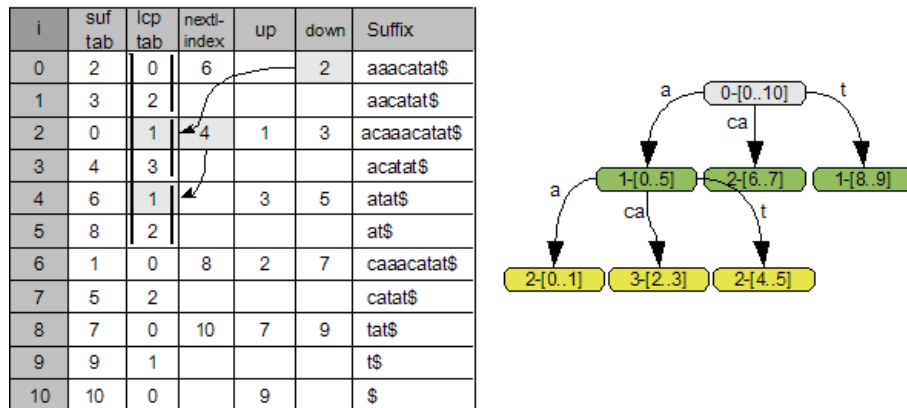


Abbildung 6: Child-Tabelle

Die Abbildung 6 zeigt das Suffix Array mit den bisherigen Erweiterungen und dem dazugehörigen lcp-Intervall-Baum. Für eine exemplarische Erklärung der Werte betrachten wir das Intervall 1-[0..5] und deren Kind-Intervalle. Die Grenzen des ersten Kind-Intervalls 2-[0..1] sind mit dem ersten l -Index durch $cldtab[0].down = cldtab[5 + 1].up = 2$ bekannt, da mit dem ersten l -Index das zweite Kind-Intervall 3-[2..3] anfängt (i. A. falls vorhanden). Durch den Eintrag $cldtab[2].nextlIndex = 4$ ist auch der zweite bzw. nächste l -Index bekannt und somit sind auch die Grenzen des nächsten Kind-Intervalls 3-[2..3] bekannt. Auf diese Weise wird solange fortgesetzt bis der Eintrag $nextlIndex$ keinen Wert mehr enthält und daraufhin das letzte Kind-Intervall bis j identifiziert ist. Somit kann also der lcp-Intervall-Baum beginnend mit dem Wurzelknoten in linearer Laufzeit Top-Down traversiert werden.

4.5.1 Konstruktion der Child-Tabelle

Die Tabelle $cldtab$ kann mit Hilfe eines Stacks und der lcp-Tabelle in linearer Laufzeit konstruiert werden. Als Stack verstehen wir einen Stapelspeicher nach dem LIFO⁵-Prinzip, der die zwei Methoden $push$ und pop bereitstellt. Mit der Anweisung $push(i)$ wird das Element i auf den Stack gelegt und mit pop wird das höchste Element vom Stack entfernt. Mit top wird auf das höchste Element des Stacks zugegriffen, ohne dass es dabei entfernt wird. Dieser Abschnitt beschreibt einen Algorithmus zur Konstruktion der $up/down$ -Werte und einen Algorithmus zur Konstruktion der $nextlIndex$ -Werte. Beide Algorithmen durchlaufen die lcp-Tabelle von 0 bis n und traversieren den lcp-Intervall-Baum Bottom-Up.

4.5.1.1 Erzeugung der $up/down$ -Einträge

Bei den $up/down$ -Werten handelt es sich immer um den ersten l -Index eines Intervalls. Wenn während des Bottom-Up-Traversals ein Intervall $q-[i..j]$ gefunden wird, muss also der Eintrag $cldtab[i].down$ mit dem ersten l -Index von $q-[i..j]$ ergänzt werden. Dieser l -Index muss auch im Eintrag $cldtab[j + 1].up$ des „benachbarten nächsten“ Intervalls $r-[j + 1..k]$ ergänzt werden. Der Algorithmus muss daher feststellen können, wann es sich um einen ersten l -Index handelt. Prinzipiell durchläuft der Algorithmus die lcp-Tabelle von

⁵Last-In-First-Out

der Position $i = 0$ bis n und überprüft den aktuellen lcp-Wert $lcptab[i]$ mit dem lcp-Wert des höchsten Stackelementes $lcptab[top]$. Solange die Bedingung $lcptab[i] < lcptab[top]$ erfüllt ist, wird mit pop das höchste Stackelement entfernt, d.h. die lcp-Werte eines soeben abgeschlossenen Intervalls werden vom Stack geräumt. Im Anschluss daran gilt die Bedingung $lcptab[i] \geq lcptab[top]$ und das aktuelle Element wird mit $push(i)$ auf den Stack gelegt, d.h. ein l -Index eines noch nicht abgeschlossenen Intervalls wird auf den Stack gelegt. Beim Bottom-Up-Traversal nach diesem Verfahren wird sozusagen ein Intervall nach dem anderen und damit deren l -Indices auf den Stack gestapelt bzw. vom Stack entfernt. Anders ausgedrückt befinden sich die l -Indices zu einem Intervall direkt aufeinander folgend auf dem Stack. Wenn wir das Verfahren auf die lcp-Werte aus Tabelle 3 anwenden so entsteht der dort nebenstehende Stackverlauf. Unterhalb des Stackverlaufs sind die dazugehörigen lcp-Werte notiert, wobei die ersten l -Indices eines Intervalls grau hinterlegt sind.

i	suf	lcp	n	u	d	$S_{sufstab[i]}$
0	2	0	6		2	aaacatat\$
1	3	2				aacatat\$
2	0	1	4	1	3	acaaacatat\$
3	4	3				acatat\$
4	6	1		3	5	atat\$
5	8	2				at\$
6	1	0	8	2	7	caaacatat\$
7	5	2				catat\$
8	7	0	10	7	9	tat\$
9	9	1				t\$
10	10	0		9		\$

i	0	1	2	3	4	5	6	7	8	9	10
S						5				9	
t				3	4	4		7	8	8	
a		1	2	2	2	2	6	6	6	6	
ck	0	0	0	0	0	0	0	0	0	0	0
l						2				1	
c		2	1	1	1	1	0	2	0	0	
p	0	0	0	0	0	0	0	0	0	0	0
i	0	1	2	3	4	5	6	7	8	9	10

Tabelle 3: Enhanced Suffix Array und Stackverlauf zu Algorithmus 1

Die Abbildung 7 zeigt eine andere Darstellung des Stacks bzw. deren lcp-Werte, d.h. für jede Position i wird nur der lcp-Wert des höchsten Stackelementes angegeben. In dieser Darstellung können nun die Intervalle (graue Boxen) und der Verlauf des Stacks skizziert werden. Beginnend ab der Position 0 bis n sieht man leicht die Reihenfolge der beim Bottom-Up-Traversal gefundenen bzw. angefangenen und noch nicht abgeschlossenen Intervalle, d.h. $0-[0..undef]$, $2-[0..1]$, $1-[0..undef]$, $3-[2..3]$, $2-[4..5]$, update $1-[0..5]$, ..., update $0-[0..10]$.

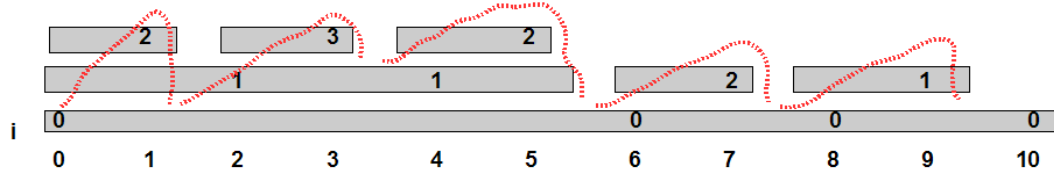


Abbildung 7: „Flachgeklopfter“ Verlauf des Stacks

Betrachtet man den Verlauf des Stacks und die Reihenfolge der gefundenen Intervalle, so ergibt sich folgende Feststellung. Immer wenn der Stack abgebaut wird, hören Intervalle

oder nur ein Intervall auf und ein oder mehrere neue Intervalle beginnen (Positionen 2, 4, 6, 8, 10). Beim Abbauen des Stacks wird daher in einem abgeschlossenen Intervall der *down*-Eintrag und in einem neu beginnenden Intervall der *up*-Eintrag aktualisiert. Es muss dabei lediglich der erste *l*-Index gefunden werden. Nachdem auf dem Stack die *l*-Indices eines Intervalls direkt aufeinander folgend gestapelt sind, kann der erste *l*-Index durch die Bedingung $lcptab[top] \neq lcptab[top - 1]$ identifiziert werden. Wenn die letzten beiden *l*-Indices auf dem Stack gleich sind, also die Bedingung nicht erfüllt ist, dann gehören sie zum gleichen Intervall und es ist nichts zu tun. Erfüllen sie die Bedingung, dann ist der Algorithmus beim Abbauen von oben nach unten auf den ersten *l*-Index des Intervalls gestoßen, denn $lcptab[top - 1]$ gehört nicht mehr zu dem Intervall. Der Eintrag $cldtab[j + 1].up$ hingegen kann erst eingetragen werden, wenn der Stack nicht mehr weiter abgebaut wird. Solange der Stack abgebaut wird, handelt es sich nicht um einen ersten *l*-Index des vorhergehenden „benachbarten“ Intervalls. Entweder ist es noch nicht der erste *l*-Index oder es ist ein *l*-Index eines Kind-Intervalls. Als Beispiel betrachten wir die Abbildung 7 an der Position 6. Der *up*-Eintrag für die Position 6 darf laut Definition nicht auf das Intervall 2-[4..5] verweisen, sondern auf den ersten *l*-Index des benachbarten Intervalls 1-[0..5].

Algorithmus 1

```
1: lastIndex := 1, push(0)
2: for i := 1 to n do
3:   while lcptab[i] < lcptab[top]
4:     lastIndex := pop
5:     if (lcptab[i] <= lcptab[top])
6:       && (lcptab[top] != lcptab[lastIndex]) then
7:         cldtab[top].down := lastIndex
8:   if lcptab[i] >= lcptab[top] then
9:     if lastIndex != -1 then
10:      cldtab[i].up := lastIndex
11:      lastIndex := -1
12:   push(i)
```

Die Laufzeit von Algorithmus 1 gehört zur Komplexitätsklasse $O(n)$, da die for-Schleife bei Zeile 2 von 0 bis n läuft und dabei in Zeile 12 höchstens n Elemente auf den Stack legt. Daraus folgt, dass die while-Schleife bei Zeile 3 insgesamt höchstens n Elemente bearbeitet.

4.5.1.2 Erzeugung der nextlIndex-Einträge

Der Algorithmus 2 durchläuft die lcp-Tabelle wie in Algorithmus 1 und führt dabei ein Bottom-Up-Traversal des lcp-Intervall-Baums durch. Beim *nextlIndex*-Wert handelt es sich um den nächsten *l*-Index eines Intervalls, weshalb der Algorithmus nur den letzten gefundenen *l*-Index auf dem Stack speichert. Im Detail bedeutet das, dass im Vergleich zum Algorithmus 1 das höchste Stackelement auch entfernt wird, wenn die Bedingung $lcptab[i] = lcptab[top]$ erfüllt ist. Es entsteht dann ein Stackverlauf wie er in Tabelle 4 dargestellt ist. Wenn die Bedingung $lcptab[i] = lcptab[top]$ erfüllt ist, dann wurde ein nächster *l*-Index gefunden und der Eintrag *nextlIndex* des *l*-Index auf dem Stack kann

i	suf	lcp	n	u	d	$S_{sufstab[i]}$
0	2	0	6		2	aaacatat\$
1	3	2				aacatat\$
2	0	1	4	1	3	acaacatat\$
3	4	3				acatat\$
4	6	1		3	5	atat\$
5	8	2				at\$
6	1	0	8	2	7	caaacatat\$
7	5	2				catat\$
8	7	0	10	7	9	tat\$
9	9	1				t\$
10	10	0		9		\$

i	0	1	2	3	4	5	6	7	8	9	10
St				3		5					
ac		1	2	2	4	4		7		9	
k	0	0	0	0	0	0	0	0	0	0	0
l				3		2					
c		2	1	1	1	1		2		1	
p	0	0	0	0	0	0	0	0	0	0	0
i	0	1	2	3	4	5	6	7	8	9	10

Tabelle 4: Enhanced Suffix Array und Stackverlauf zu Algorithmus 2

in $cldtab$ aktualisiert werden. Im Detail bedeutet das, wenn $lcptab[top] = l$ -Index, dann $cldtab[top].nextlIndex = i$.

Algorithmus 2

```

1: push(0)
2: for i := 1 to n do
3:   while lcptab[i] < lcptab[top]
4:     pop
5:   if lcptab[i] = lcptab[top] then
6:     lastIndex := pop
7:     cldtab[lastIndex].nextlIndex := i
8:   push(i)

```

Der Algorithmus 2 besitzt eine Laufzeit von $O(n)$, da der Aufbau prinzipiell dem von Algorithmus 1 gleicht.

4.6 Finden von Kind-Intervallen

Durch die Tabelle $cldtab$ ist es nun möglich die Kind-Intervalle eines gegebenen Intervalls in linearer Laufzeit zu finden. Hierzu definieren wir eine Funktion $getChildIntervals(i, j)$, welche die Intervallgrenzen des Intervalls $q[i..j]$ entgegen nimmt und eine Liste der Kind-Intervalle in $q[i..j]$ ausgibt. Zuerst bestimmt die Funktion entweder durch den Eintrag $cldtab[i].down$ oder $cldtab[j + 1].up$ den ersten l -Index i_1 und kann somit das erste Kind-Intervall $k_1-[i..i_1 - 1]$ ausgeben. Im Anschluss daran können alle weiteren Kind-Intervalle durch die $nextlIndex$ -Einträge ermittelt werden, da ein l -Index ein Kind-Intervall abschließt und gleichzeitig das erste Element eines neuen Kind-Intervalls ist, d.h. $k_2-[i_1..i_2 - 1], \dots, k_r-[i_{r-1}..j]$ mit $0 < r < |\mathcal{A}|$ (Lemma 1). Wenn kein $nextlIndex$ -Wert mehr vorhanden ist, kann das letzte Kind-Intervall ausgegeben werden. Die Realisierung eines Top-Down-Traversals durch den lcp-Intervall-Baum kann nun durch $getChildIntervals(i, j)$ in linearer Laufzeit erfolgen. Für das Problem *Exact Pattern Matching* macht es Sinn eine Funktion $getChildInterval(i, j, b)$ zu definieren, die zusätzlich zu den Intervallgrenzen

einen Buchstaben $b \in \mathcal{A}$ akzeptiert und nur das Kind-Intervall ausgibt, welches mit dem Buchstaben b beginnt. Die Funktion geht genauso vor wie $getChildIntervals(i, j)$ und führt zusätzlich bei jedem Kind-Intervall einen Zeichenvergleich mit dem l -ten Zeichen des Suffixes beim l -Index durch. Dadurch kann in konstanter Zeit das gesuchte Kind-Intervall gefunden werden, wenn das Alphabet \mathcal{A} konstant ist, d.h. beide Funktionen haben eine Laufzeit von $O(|\mathcal{A}|)$. Außerdem ist es auch sinnvoll die Länge des längsten gemeinsamen Präfix eines Kind-Intervalls zu ermitteln. Dafür wird eine Funktion $getlcp(i, j)$ definiert, die lediglich den lcp-Wert entweder durch $cldtab[i].down$ oder durch $cldtab[j + 1].up$ ermittelt.

5 Exact Pattern Matching - Enhanced Suffix Array

Das Problem *Exact Pattern Matching* lässt sich nun mit Hilfe des Enhanced Suffix Arrays und den definierten Funktionen mit dem Algorithmus 3 in linearer Laufzeit lösen. Prinzipiell wird ein Suchintervall über dem Suffix Array durch eine Schleife und Zeichenvergleich mit dem Muster P Schritt für Schritt verkleinert. Zu Beginn wird das Suchintervall auf das gesamte Suffix Array von 0 bis n festgelegt und durch $getChildInterval()$ dasjenige Kind-Intervall \mathcal{I} abgefragt, dessen erstes Zeichen mit dem des Musters P übereinstimmt. Wir betrachten also im lcp-Intervall-Baum bzw. im Suffix Tree die Kantenmarkierungen am Wurzelknoten und wählen den Teilbaum, bei dem das erste Zeichen der Kantenmarkierung mit dem des Musters übereinstimmt. Den Fortschritt im Muster P erhalten wir mit einer Variablen pos . Betrachten wir die Abbildung 8, bei der wir das Muster $P = at$ der Länge $m = 2$ im String $S\$ = acaaacatat\$$ suchen. Das Intervall $0-[0..n]$ besitzt 3 Kind-Intervalle, wovon nur die Suffixe des ersten Intervalls $1-[0..5]$ das erste Zeichen mit P gemeinsam haben. Die Suffixe im Intervall beginnen alle mit dem Zeichen 'a', also dem ersten Zeichen von P .

i	suf tab	lcp tab	next-index	up	down	Suffix
0	2	0	6		2	aaacatat\$
1	3	2				aacatat\$
2	0	1	4	1	3	acaacatat\$
3	4	3				acatat\$
4	6	1		3	5	atat\$
5	8	2				at\$
6	1	0	8	2	7	caaacatat\$
7	5	2				catat\$
8	7	0	10	7	9	tat\$
9	9	1				t\$
10	10	0		9		\$

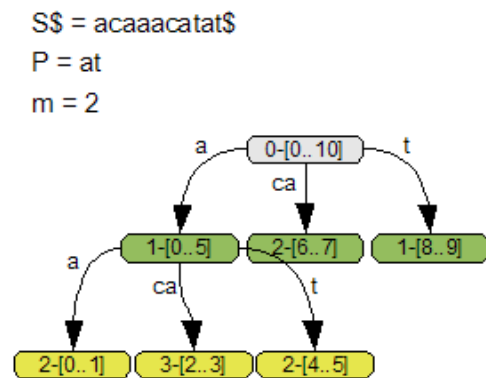


Abbildung 8: Exact Pattern Matching mit dem Enhanced Suffix Array

Ganz allgemein müssen wir hier eine erste Fallunterscheidung (1) machen, ob das gefundene Intervall \mathcal{I} ein leeres Intervall, ein Singleton-Intervall oder ein Kind-Intervall ist.

Bei einem leeren Intervall ist das Muster P nicht in S vorhanden, d.h. im lcp-Intervall-Baum geht keine „passende“ Kante weg. Wurde ein Singleton-Intervall gefunden, so liegt ein expliziter Suffix vor und es muss nur noch überprüft werden ob der restliche Teil von P im gefundenen Suffix vorhanden ist, d.h. $S[suftab[i] + pos..suftab[i] + m - 1] == P[pos..m - 1]$.

z.B. $S\$ = acaaacatat\$$, $P = catc$
führt schließlich zum Intervall [7..7] mit dem Suffix $S_{suftab[7]} = catat\$$

Wenn nicht, so ist P nicht in S vorhanden und ansonsten ist P genau einmal in S vorhanden. Im Suffix Tree wäre die Suche bei einem Blatt angekommen. Für den Fall, dass bei (1) ein Kind-Intervall vorliegt, erfolgt eine weitere Fallunterscheidung (2). Hierfür wird zuerst der Wert lcp durch die Funktion $getlcp(\mathcal{I})$ abgefragt und damit dann das längste gemeinsame Präfix w der Suffixe im Kind-Intervall \mathcal{I} ermittelt. Dies entspricht im lcp-Intervall-Baum der Aneinanderreihung der Kantenmarkierungen von der Wurzel bis zum Intervall \mathcal{I} .

z.B. $\mathcal{I} = 3-[2..3]$ mit $lcp = 3$, dann ist $w = aca$ vgl. Abbildung 8.

Mit der Länge lcp von w wird dann bei (2) ein Vergleich mit der Länge m des Musters P durchgeführt. Wenn das Muster weniger oder gleich viele Zeichen besitzt als w (das Präfix der Suffixe im Intervall), dann muss nur überprüft werden ob der restliche Teil des Musters in w vorhanden ist, d.h. wenn die Länge m des Musters kleiner oder gleich lcp ist, dann prüfe $S[suftab[i] + pos..suftab[i] + m - 1] == P[pos..m - 1]$.

z.B. $S\$ = acaaacatat\$$, $P = ac$, $m = 2$
führt schließlich zum Intervall [2..3] mit $lcp = 3$ und $w = aca$.
Hier besitzt P weniger Zeichen als w und das Muster ist vollständig im Präfix vorhanden.

Wenn P nicht in w vorhanden ist, dann ist P auch nicht in S vorhanden. Andernfalls kann die Suche erfolgreich mit dem Intervall \mathcal{I} beendet werden.

Im anderen Fall bei (2) ist das Muster P länger als das Präfix w und es muss festgestellt werden, ob (3) das Präfix im Muster vorhanden ist. Falls bei (3) w nicht in P ist, dann wurde das Muster P nicht in S gefunden, d.h. jedes der Suffixe im Kind-Intervall \mathcal{I} stimmt schon beim Präfix w nicht überein. Übertragen auf den lcp-Intervall-Baum bedeutet dies, dass eine Kante mit übereinstimmendem erstem Zeichen gefunden wurde, aber bei einem der weiteren Zeichen auf der Kante nicht mehr.

z.B. $S\$ = acaaacatat\$$, $P = acct$, $m = 4$

führt schließlich zum Intervall $[2..3]$ mit $lcp = 3$ und $w = aca$.

Hier besitzt P mehr Zeichen als w und das Muster stimmt mit dem Präfix ab dem 3.ten Zeichen nicht überein.

Wenn bei (3) das Präfix im Muster vorhanden ist, dann wird ab dem nächsten Zeichen im Muster die Suche mit dem Kind-Intervall \mathcal{I} fortgesetzt. Der Fortschritt wird also mit $pos := lcp$ aktualisiert und die Suche wird wie bisher beschrieben mit dem Kind-Intervall \mathcal{I} fortgesetzt.

Dieses Verfahren ermittelt also ein Kind-Intervall $l-[i..j]$, wenn das Muster P im String S vorhanden ist. Möchte man wissen wie oft das Muster P in S vorkommt, müssen nur die Intervallgrenzen voneinander abgezogen werden, d.h. $z := j - i$. Die Positionen von P in S können mit Hilfe der Tabelle $suftab$ ermittelt werden, d.h. $suftab[i]$, ..., $suftab[j]$.

Algorithmus 3

```

1: pos := 0
2: bFound := TRUE
3: (i,j) := getChildInterval(0, n, P[pos])
4: while (i,j) != EMPTY && pos < m && bFound = TRUE
5:   if i != j then
6:     lcp := getlcp(i, j)
7:     min := min { lcp, m }
8:     bFound := S[suftab[i]+pos .. suftab[i]+min-1] = P[pos .. min-1]
9:     pos := min
10:    (i,j) := getChildInterval(i, j, P[pos])
11:   else bFound := S[suftab[i]+pos .. suftab[i]+m-1] = P[pos .. m-1]
12: if bFound then
13:   print "Interval [i..j]"
14: else print "Pattern P not found"

```

Der Algorithmus 3 liefert das Ergebnis mit einer Laufzeit von $O(m)$, da die while-Schleife ab Zeile 4 spätestens mit der Bedingung $pos \geq m$ abbricht. Der lcp -Wert lcp in der Schleife bei Zeile 6 wächst streng monoton mit jedem gefundenen Kind-Intervall an und nachdem lcp der Variablen pos zugewiesen wird, muss eine existierende Lösung nach höchstens m Iterationen gefunden worden sein. Das Ausgeben der Positionen von P in S ist mit $O(z)$ Zeit möglich, sodass für das Problem *Exact Pattern Matching* mit Ausgabe der Positionen eine optimale Laufzeit von $O(m + z)$ entsteht.

Allerdings ist diese Laufzeit auch noch von der Größe des Alphabets abhängig. Diese Konstante steckt in $getChildInterval()$, da dort der Zeichenvergleich im *worst - case* bei allen Kind-Intervallen, also höchstens $|\mathcal{A}|$ mal, durchgeführt wird. Konkret entsteht dadurch eine Laufzeit von $O(|\mathcal{A}| * m + z)$.

6 Reduzierung der Speicheranforderung

Bei Anwendungen auf großen Datenmengen ist eine geringe Speicheranforderung der Algorithmen und Datenstrukturen enorm wichtig wie Anfangs bereits erwähnt. Dieser Abschnitt erläutert, wie die lcp-Tabelle auf n Bytes und die Child-Tabelle auf $4n$ Bytes reduziert werden kann und somit nur $6n$ Bytes für das gesamte Enhanced Suffix Array benötigt wird. Im Vergleich zu den $20n$ Bytes für einen Suffix Tree ist die Speicheranforderung um den Faktor 3 geringer bzw. im Vergleich zu den $12.5n$ Bytes in der Praxis um den Faktor 2 geringer.

6.1 lcp-Tabelle

Ein lcp-Wert kann theoretisch zwischen 0 und $n - 1$ betragen und bei einem großen $n < 2^{32}$ wären 8 Bytes für einen lcp-Eintrag notwendig. In der Praxis jedoch sind lcp-Werte größer als 255 bei Anwendungen auf DNA-Sequenzen eher selten, z.B. $0.0078n$ bei *E.coli* (bakterielles Genom) oder $0.071n$ bei *Yeast* (eukariontisches Genom) vgl. [1]. Mit dieser Beobachtung kann die Speicheranforderung für die lcp-Tabelle auf n Bytes reduziert werden, um lcp-Werte von 0 bis 254 zu speichern. In einer zusätzlichen Tabelle *llvtab* werden dann die selten auftretenden lcp-Werte größer als 254 zusammen mit der Position i sortiert abgespeichert. Wenn man in *lcptab* auf ein Wert von 255 trifft, wird eine binäre Suche mit $O(\log|llvtab|)$ auf *llvtab* durchgeführt, welche nach $ld|llvtab|^6$ Schritten anhand von i den lcp-Wert liefert.

6.2 Child-Tabelle

Die Child-Tabelle benötigt in der Theorie zusätzlich $12n$ Bytes, aber in der Praxis kann diese um das zwölfwache auf n Bytes reduziert werden, indem Redundanzen entfernt und nicht belegte Einträge genutzt werden. Betrachtet man die *nextlIndex*-Spalte von *clddb* so stellt man fest, dass *clddb*[i].*nextlIndex* keinen Wert enthält, wenn die Bedingung *lcptab*[i] > *lcptab*[$i + 1$] erfüllt ist. Wenn der lcp-Wert von i auf $i + 1$ abnimmt, dann hört ein lcp-Intervall bei i auf und es kann keinen nächsten l -Index geben. Gleichzeitig fängt auch ein neues Intervall an, der einen *up*-Eintrag besitzen muss. Daher kann der *up*-Eintrag des neu beginnenden Intervalls in den *nextlIndex*-Eintrag des abgeschlossenen Intervalls verschoben werden vgl. Abbildung 9.

⁶*ld* entspricht einem Logarithmus Dualis, also \log_2

i	suf tab	lcp tab	nextl- index	up	down	Suffix
0	2	0	6		2	aaacatat\$
1	3	2				aacatat\$
2	0	1	4	1	3	acaaacatat\$
3	4	3				acatat\$
4	6	1		3	5	atat\$
5	8	2				at\$
6	1	0	8	2	7	caaacatat\$
7	5	2				catat\$
8	7	0	10	7	9	tat\$
9	9	1				t\$
10	10	0		9		\$

Abbildung 9: Reduzierung der Speicheranforderung

Weiterhin sind die meisten *down*-Einträge bereits durch *up*-Einträge redundant vorhanden. Lediglich der letzte *down*-Eintrag eines Intervalls wird nicht durch einen *up*-Eintrag abgedeckt, z.B. in Abbildung 9 an Position 4. Der Grund dafür ist, dass der „dazugehörige“ *up*-Eintrag sich nicht auf die Kind-Intervalle sondern auf das umschließende Intervall bezieht. Als Beispiel betrachten wir in Abbildung 9 die Position 6. Der dortige *up*-Eintrag verweist auf den ersten *l*-Index des Intervalls 1-[0..5] und nicht auf das Kind-Intervall 2-[4..5]. Allerdings liegt der *down*-Eintrag eines Intervalls auch auf einem *l*-Index und der *nextlIndex*-Eintrag des letzten *l*-Index ist immer leer. Es gibt nach dem letzten *l*-Index keinen weiteren und der entsprechende Eintrag $cldtab[i].down$ kann in den Eintrag $cldtab[i].nextlIndex$ verschoben werden. Hier entsteht kein Konflikt mit den *nextlIndex*-Einträgen der verschobenen *up*-Einträge, da es sich hier um das umschließende Intervall der dort erwähnten Kind-Intervalle handelt.

Um bei einer gegebenen Position i eine Unterscheidung nach *up/down/nextlIndex* durchzuführen, wird zuerst die Bedingung $lcptab[cldtab[i]] == lcptab[i]$ geprüft. Ist diese erfüllt, so handelt es sich um einen *nextlIndex*-Eintrag, d.h. es werden die zwei beteiligten lcp-Werte auf Gleichheit geprüft. Falls die zwei beteiligten lcp-Werte nicht gleich sind, so handelt es sich um einen *up/down*-Eintrag. Ist die Bedingung $lcptab[i] > lcptab[i + 1]$ erfüllt, so handelt es sich beim Eintrag $cldtab[i]$ um den *up*-Eintrag von Position $i + 1$. Auf diese Weise können die Einträge unterschieden werden und die Tabelle *cldtab* kommt mit $4n$ Bytes für alle 3 Einträge aus.

Eine weitere Reduzierung wird erreicht, indem relative Werte abgelegt werden, d.h. aus $j := cldtab[i]$ wird $cldtab[i] := j - i$ vgl. Tabelle 5. Negative Werte werden als positive Werte abgelegt, entsprechen also dem Absolutbetrag, wobei durch *up* oder *down/nextlIndex* bereits festgelegt ist, ob ein Wert negativ oder positiv interpretiert wird.

Anschließend sind die Werte in *cldtab* insgesamt sehr viel kleiner und für die Tabelle *cldtab* sind lediglich n Bytes notwendig. Es werden also nur Werte bis 254 gespeichert und Werte, die größer als 254 sind werden nicht mehr abgespeichert. Der Wert 255 dient wie bei der lcp-Tabelle als Auslöser für eine binäre Suche nach dem *l*-Index.

Enhanced Suffix Array				
i	suftab	lcptab	cldtab	$S_{suftab[i]}$
0	2	0	6	aaacatat\$
1	3	2	0	aacatat\$
2	0	1	2	acaaacatat\$
3	4	3	0	acatat\$
4	6	1	1	atat\$
5	8	2	3	at\$
6	1	0	2	caaacatat\$
7	5	2	0	catat\$
8	7	0	2	tat\$
9	9	1	0	t\$
10	10	0		\$

Tabelle 5: Relative Werte für *cldtab*

Ein Wert von 255 in der Tabelle *cldtab* wird im Vergleich zur lcp-Tabelle viel öfter angetroffen, da es sich in *cldtab* um Verweise auf andere Einträge in der Tabelle handelt und nicht um eine Eigenschaft zweier aufeinander folgender Einträge. In den Anfängen einer Suche kommt es bei einem großen String oft vor, dass die Größe der Kind-Intervalle den Wert 255 überschreitet. Wenn beispielsweise das Alphabet aus drei Buchstaben, also $\mathcal{A} = \{a, b, c\}$, $|\mathcal{A}| = 3$, besteht und der String mehr als $3 * 255$ Buchstaben aufweist, also $|S| > 3 * 255$, dann besitzt mindestens eines der Kind-Intervalle mehr als 255 Suffixe, d.h. mindestens einmal findet sich ein Wert von 255 in der Tabelle *cldtab* wieder. Bei einer Gleichverteilung der Buchstaben in S besitzt auch jedes Kind-Intervall mindestens ein Kind-Intervall mit mehr als 255 Suffixen, wenn $|S| > 3 * 255 * 255$, usw.. Daraus folgt, dass die binäre Suche für die ersten Kind-Intervalle je häufiger durchgeführt wird, umso länger der Strings S ist. Die binäre Suche gehört zur Komplexitätsklasse $O(\log n)$ und ist daher nicht optimal, d.h. konkret werden bis zu $ld n$ Schritte für eine Lösung benötigt.

Wünschenswert ist eine lineare Laufzeit natürlich auch bei der Suche durch die ersten Kind-Intervalle der Größe > 254 , welche so gut wie bei jeder Suche durchlaufen werden. Deshalb wird hierfür eine zusätzliche Bucket-Tabelle eingeführt. Sie speichert bei einem gegebenen Parameter q für jeden String w der Länge $|w| = q$, das kleinste i , so dass $S[suftab[i]..suftab[i] + q - 1] = w$. Man betrachtet also ein Intervall $l-[i..j]$ mit Suffixen, die sich mindestens den String w als Präfix teilen. Die Bucket-Tabelle enthält dann für jeden String w die linke Intervallgrenze i der betrachteten Intervalle.

Auf diese Art und Weise wird die binäre Suche für die ersten Zeichen w der Länge q und damit den großen Kind-Intervallen am Anfang vermieden. Bei geschickter Implementierung durch ein Hash-Verfahren liefert die Bucket-Tabelle in linearer Zeit für w das entsprechende Kind-Intervall. Der Suchalgorithmus mit *getChildInterval()* arbeitet dadurch meist nur auf einem kleineren Teil des Suffix Arrays.

Insgesamt werden also für das Enhanced Suffix Array lediglich $6n$ Bytes ($4n$ Bytes *suftab*, $1n$ Bytes *lcptab*, $1n$ Bytes *cldtab*) benötigt. Die Bucket-Tabelle und die Tabelle *llvtab* sind im Vergleich zum Enhanced Suffix Array vernachlässigbar klein.

7 Zusammenfassung

Dass die hier besprochenen Erweiterungen des Suffix Arrays nicht nur von theoretischer Natur sind, zeigen experimentelle Ergebnisse aus [1]. Dort wurde das Enhanced Suffix Array im Programm *esamatch* implementiert und mit zwei weiteren Programmen verglichen, die Suffix Arrays nach [5] und Suffix Trees verwenden. Bei Anwendungen in der Bioinformatik, wo die Größe des Alphabets meist klein ist, wird eine Suche mit Enhanced Suffix Arrays mindestens so schnell, meistens aber etwas schneller als mit Suffix Arrays nach [5] durchgeführt. Im Vergleich zu Suffix Trees erreichen Enhanced Suffix Arrays meist mindestens einen Geschwindigkeitsgewinn um den Faktor 2.

Je größer aber das Alphabet wird, desto mehr macht sich die Konstante $|\mathcal{A}|$ in der Laufzeit negativ bemerkbar. Letzteres erklärt sich aus der Optimierung der Laufzeit von $O(m * \log n + z)$ auf $O(m * |\mathcal{A}| + z)$. Wenn die Größe des Alphabets den Wert $\ln n$ erreicht, dann beginnt sich die Laufzeit gegenüber den Suffix Arrays nach [5] zu verschlechtern. Bei DNA-Sequenzen ist das Alphabet mit der Länge 4 allerdings klein genug, sodass sich die Konstante $|\mathcal{A}|$ in der Laufzeit nicht so stark bemerkbar macht.

Literaturverzeichnis

- [1] M. I. Abouelhoda, E. Ohlebusch and S. Kurtz. Optimal Exact String Matching Based on Suffix Arrays, Faculty of Technology, University of Bielefeld, A.H.F. Laender and A.L. Oliveira (Eds.): *SPIRE 2002*, pp. 31-43, Springer-Verlag Berlin Heidelberg 2002
- [2] M. I. Abouelhoda, S. Kurtz., E. Ohlebusch, Replacing suffix trees with enhanced suffix arrays, *Journal of Discrete Algorithms 2* (2004) 53-86
- [3] M. I. Abouelhoda, S. Kurtz., E. Ohlebusch, The enhanced suffix array and its applications to genome analysis, *Proceedings of the second workshop on algorithms in bioinformatics*, Springer-Verlag, Lecture notes in computer science, 2002
- [4] J. Kärkkäinen and P. Sanders, Simpler Linear Work Suffix Array Construction, Max-Planck-Institut für Informatik, *ICALP 2003, LNCS 2719*, pp. 943-955, Springer-Verlag Berlin Heidelberg 2003
- [5] U. Manber, G. Myers, A new method for on-line string searches, Department of Computer Science, University of Arizona, *SIAM Journal on Computing*, 22(5), pp. 935-948, 1993
- [6] P. Ko and S. Aluru, Space Efficient Linear Time Construction of Suffix Arrays, Department of Electrical and Computer Engineering, Iowa State University, *Proceedings of the 14th Annual Symposium on Combinatorial Pattern Matching*, pp. 200-210, 2003
- [7] M. I. Abouelhoda, S. Kurtz., E. Ohlebusch, The Enhanced Suffix Array and Its Applications to Genome Analysis, Faculty of Technology, University of Bielefeld, *WABI 2002, LNCS 2452*, pp. 449-463, Springer-Verlag Berlin Heidelberg 2002
- [8] P. Weiner, Linear pattern matching algorithms, *Proceedings of the 14th Symposium on Switching and Automata Theory*, pp. 1-11, IEEE, 1973
- [9] E. M. McCreight, A space-economical suffix tree construction algorithm, *In Journal of the ACM 23*, pp. 262-272, 1976
- [10] E. Ukkonen, On-line construction of suffix trees, Department of Computer Science, University Helsinki, *Algorithmica 14(3)*, pp. 249-260, 1995
- [11] S. Kurtz, Reducing the Space Requirement of Suffix Trees, *Software - Practice and Experience*, 29(13),:1149-1171, 1999
- [12] T. Kasai, G. Lee, H. Arimura, S. Arikawa, and K. Park. Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and its Applications, *In Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*, pp. 181-192, LNCS 2089, Springer-Verlag Berlin Heidelberg 2001
- [13] E. Ohlebusch, Algorithmen zur Sequenzanalyse, Skript zur Vorlesung im Sommersemester 2004