Entwurf und Implementierung eines generischen Substring-Index

Diplomarbeit

Humboldt-Universität zu Berlin
Mathematisch-Naturwissenschaftliche Fakultät II
Institut für Informatik

eingereicht von: David Weese

Betreuer: Prof. Dr. Knut Reinert
Institut für Informatik, Freie Universität Berlin
Prof. Dr. Ulf Leser
Institut für Informatik, Humboldt-Universität zu Berlin

Berlin, den 2. Mai 2006

Inhaltsverzeichnis

Kapitel 1

Einleitung

Motivation

Mit der Sequenzierung kompletter Genome in den letzten Jahren wurden erhebliche Datenmengen veröffentlicht, die nun weiter analysiert werden müssen. Viele der dafür benötigten Algorithmen suchen sehr oft nach Vorkommen bestimmter Teilsequenzen innerhalb dieser Genome, so dass es Datenstrukturen und Algorithmen bedarf, die allgemein das Auffinden von Mustern in großen Datensätzen effizient realisieren. In Datenbanken häufig verwendete B-Bäume oder Invertierte Listen sind dabei nur bedingt geeignet, da sie ganze Wörter speichern und die Suche nach beliebigen Teilsequenzen nicht zulassen.

Für die Suche innerhalb ganzer Genome besser geeignet sind so genannte Substring-Indizes. Ein Substring-Index ist eine Datenstruktur, welche Informationen zur lexikographischen Ordnung aller Teilsequenzen eines Strings enthält und Methoden zur Verfügung stellt, effizient nach Vorkommen beliebiger Strings zu suchen.

In der Arbeitsgruppe Algorithmische Bioinformatik der Freien Universität Berlin wird die Software-Bibliothek Seqan entwickelt. Seqan ist eine generische Software-Bibliothek, die Algorithmen und Datenstrukturen zur Analyse von Genom- oder Proteinsequenzen bereit stellt und die Entwicklung von eigenen Algorithmen erleichtern soll.

Ziel der Arbeit

Das Ziel dieser Arbeit ist der Entwurf, die Implementierung und die Analyse eines generischen Substring-Index für große Strings. Er soll in die Software-Bibliothek Seqan integriert werden und die effiziente Suche nach Vorkommen beliebiger Teilsequenzen ermöglichen.

Vorgehen und Aufbau der Arbeit

Für die exakte Substring-Suche innerhalb sehr großer Datenmengen haben sich Indizes wie Suffix-Arrays [20], Suffix-Arrays mit erweiterten LCP-Tabellen und Enhanced Suffix-Arrays [1] als geeignet erwiesen. Mit ihnen lassen sich die Vorkommen eines Substrings der Länge m innerhalb eines festen Strings s der Länge n in $O(m \log n)$, $O(m + \log n)$ bzw. $O(m)$ finden [2]. Ihre Struktur ist einfacher und kompakter als die der Suffixbäume und viele auf Suffixbäumen basierende Algorithmen lassen sich auf Suffix-Arrays oder Enhanced Suffix-Arrays übertragen [1]. Der Substring-Index dieser Arbeit basiert auf einem Suffix-Array und optional einer erweiterten LCP-Tabelle.

Kapitel 3 definiert zunächst grundlegende Begriffe und beschreibt den Aufbau von Suffix-Arrays und LCP-Tabellen sowie verschiedene Speichermodelle. In Kapitel 4 werden bekannte interne und externe Sortierverfahren und ein neuer externer Permutationsalgorithmus vorgestellt. Sie bilden das Fundament des in Kapitel 5 beschriebenen Konzepts des Pipelinings [14] für externe Algorithmen. Es werden außerdem Grundlagen zur Entwicklung eigener externer Algorithmen und deren theoretischer Analyse geschaffen.

In Kapitel 6 wird der 2003 von Kärkkäinen und Sanders vorgestellte Skew-Algorithmus [13], [14] beschrieben. Er ist einer der ersten Linearzeitalgorithmen zum Aufbau von Suffix-Arrays. Im Anschluß wird eine Erweiterung des Algorithmus um so genannte Difference Covers [12] vorgenommen, durch die sowohl die Laufzeit als auch der Speicherbedarf des Algorithmus reduziert werden konnten. Da die Algorithmen trotzdem jeweils mindestens $4n$ Byte zusätzlichen Hauptspeicher benötigen, wird in Abschnitt 6.5 eine externe Variante des Skew-Algorithmus sowie dessen Erweiterung um Difference Covers vorgestellt. Am Ende des Kapitels erfolgt eine allgemeine theoretische Analyse der I/O-Komplexitäten von externen Skew-Algorithmen mit Difference Covers verschiedener Größen.

Kapitel 7 konzentriert sich auf die Konstruktion von LCP-Tabellen. Es wird der 2001 von Kasai et al. [15] veröffentlichte LCP-Algorithmus beschrieben und eine Verbesserung vorgestellt, die für die Konstruktion keinen zusätzlichen Speicher mehr benötigt. Am Ende des Kapitels wird eine externe Variante des Algorithmus entworfen, umgesetzt und analysiert.

In Kapitel 8 wird schließlich gezeigt, wie effizient im Suffix-Array gesucht werden kann. Um die worst-case-Laufzeit der Substring-Suche zu verbessern wird zusätzlich eine erweiterte LCP-Tabelle benötigt. Der Aufbau und ein optimaler Konstruktionsalgorithmus dieser Tabelle, sowie die entsprechend angepasste Binärsuche sind in Abschnitt 8.2 angegeben.

Die in den Kapiteln 6 bis 8 entwickelten, implementierten und theoretisch analysierten Algorithmen werden in Kapitel 9 innerhalb einer generischen Datenstruktur, dem generischen Substring-Index, zusammengefasst. Dazu werden

zunächst Abstrakte Datentypen für eine allgemeine exakte Substring-Suche konzipiert, die anschließend um den Abstrakten Datentyp 'Substring-Index' erweitert werden. Er ermöglicht einen schnellen Zugriff auf alle Substrings einer Menge von Strings und stellt Methoden zur Verfügung zum Aufbau des Index zu einem oder mehreren Strings und zum Suchen nach Teilstrings.

In Kapitel 10 werden die verschiedenen Implementierungen an biologischen und anderen praktischen Daten getestet und mit bekannten Algorithmen verglichen. Genauer untersucht werden Ausführungszeiten, Speicherbedarf und I/O-Zugriffe, die für den Aufbau des Index und die Suche von Substrings benötigt werden. Die entwickelten theoretischen Abschätzungen werden daraufhin mit den empirischen verglichen.

Die letzten beiden Kapitel fassen die Arbeit schließlich zusammen und geben einen Ausblick auf mögliche Erweiterungen und Verbesserungen.

Kapitel 2

Stand der Entwicklung

Suffixbäume wurden erstmals 1973 zusammen mit einem $\mathcal{O}(n)$[1]-Konstruktionsalgorithmus von Weiner vorgestellt [28]. McCreight verbesserte 1976 die Speichereffizienz [23] und Ukkonen fand 1995 einen besser analysierbaren und s sequentiell lesenden Konstruktionsalgorithmus [26]. Mit Suffixbäumen wurde eine exakte Substing-Suche in $\mathcal{O}(m)$ möglich. Wegen des geringeren Speicherbedarfs eignen sich Suffix-Arrays allerdings besser für Substring-Indizes.

Suffix-Arrays und ein direkter $\mathcal{O}(n \log n)$-Konstruktionsalgorithmus wurden erstmals 1990 von Manber und Myers vorgestellt [20],[21]. In ihrer Arbeit veröffentlichten sie außerdem Algorithmen für die exakte Substring-Suche mit einem Suffix-Array in $\mathcal{O}(m \log n)$ bzw. mit einem Suffix-Array und zusätzlichem LCP-Intervallbaum in $\mathcal{O}(m+\log n)$. Seitdem erschienen zahlreiche Publikationen und Algorithmen zum Aufbau von Suffix-Arrays, von denen die wichtigsten vergleichend in [25] beschrieben sind. Im Jahr 2003 wurden von Kärkkäinen und Sanders, Kim et al. und von Ko und Aluru unabhängig und fast gleichzeitig drei Algorithmen veröffentlicht, mit denen erstmals Suffix-Arrays asymptotisch optimal und direkt in $\mathcal{O}(n)$ konstruiert werden konnten [13],[16],[17].

Crauser und Ferragina stellten 2002 erste Implementationen für Sekundärspeicher vor, die aber noch asymptotisch suboptimal waren [7]. Den ersten asymptotisch optimalen externen Algorithmus, eine für Sekundärspeicher modifizierte Version des Algorithmus von Kärkkäinen und Sanders, veröffentlichten 2005 Dementiev et al. [8]. Er hat eine I/O-Komplexität von $\mathcal{O}\left(\text{sort}(n)\right)$ Zugriffen[2] und benötigt zum Sortieren des menschlichen Genoms etwa $8\frac{1}{2}$ Stunden auf einem Testsystem mit 4 Festplatten und einer 2 GHz CPU.

LCP-Tabellen konnten erstmals 1990 im Algorithmus von Manber und Myers [20] und in vielen danach veröffentlichten Algorithmen zusammen mit dem Suffix-Array konstruiert werden. Im Jahr 2001 gelang es Kasai et al. [15], einen

[1] Es wird ein festes Alphabet Σ angenommen.

[2] $\text{sort}(N) = \Theta\left(\frac{N}{PB} \log_{\frac{M}{B}} \frac{N}{B}\right)$ ist die Zahl der I/O-Zugriffe, die zum externen Sortieren der Zahlen 1,. . . ,n benötigt werden [27]. Siehe Kapitel 3.6.

direkten LCP-Algorithmus zu entwickeln mit einer Laufzeit von $\mathcal{O}(n)$. Der Speicherbedarf des Algorithmus konnte 2004 von Manzini [22] mit Hilfe der Burrows-Wheeler-Transformation [6] von 13n auf optimal[3] 9n Byte reduziert werden.

[3] ohne Kompression oder Überschreiben des Suffix-Array

Kapitel 3

Grundlagen und Definitionen

3.1 Bezeichnungen

Es sei Σ ein nichtleeres Alphabet. Σ^n bezeichnet die Menge aller Wörter der Länge n über dem Alphabet Σ. Weiter sei $\Sigma^* := \bigcup_{i=0}^{\infty} \Sigma^i$, wobei $\Sigma^0 = \{\epsilon\}$ ist und ϵ das leere Wort bezeichnet. Für $s \in \Sigma^n$ bezeichnet $|s| = n$ die Länge von s. Im weiteren beginnen alle Strings mit dem Index 0 und es gilt:

Definition 3.1.1
Sei $s \in \Sigma^n$ ein String und $i, j \in \mathbb{N}$.

- $s[i]$ *bezeichnet das* $(i+1)$*-te Zeichen von* s*.*
- $[i..j] := \{i, i+1, \ldots, j\}$
- $[i..j) := [i..j-1]$
- $s[i..j] := s[i]s[i+1]\ldots s[j]$ *heißt Substring oder Teilstring von* s
- $s[i..j) := s[i..j-1]$
- $s_i := s[i..n)$ *heißt Suffix von* s*.*
- $s[0..i)$ *heißt Präfix von* s*.*

Die Bezeichnungen Felder und Tabellen werden synonym für Strings benutzt. Das entsprechende Alphabet geht dann entweder aus dem Kontext hervor oder ist ohne weitere Angaben $\mathbb{N}$, die Menge der nichtnegativen ganzen Zahlen. Strings der Länge k werden im Folgenden auch k-lets genannt. Strings können auch als Mengen aufgefasst werden, wobei ein Element x genau dann in s enthalten ist, wenn x als Zeichen in s vorkommt.

3.2 Relationen

Auf Σ sei eine Striktordnung $<$ definiert, das heißt, für zwei Buchstaben $a, b \in \Sigma$ mit $a \neq b$ gilt entweder $a < b$ oder $b < a$, nicht aber beides. Die Relation $>$ ergibt sich aus $<$ mit $a > b \Leftrightarrow b < a$ Die Ordnung der Buchstaben in Σ ist nun übertragbar auf Ordnungen auf Strings aus Σ^*.

Definition 3.2.1
Es seien $s, t \in \Sigma^* \setminus \{\epsilon\}$ zwei Strings. $<$ wird folgendermaßen definiert:

- $\epsilon < s$
- $\neg\,(\epsilon < \epsilon) \;\wedge\; \neg\,(s < \epsilon)$
- $s < t \Leftrightarrow s[0] < t[0] \;\vee\; (s[0] = t[0] \;\wedge\; s_1 < t_1)$

Diese rekursive Definition macht $<$ zu einer Striktordnung auf Σ^*. Im Weiteren seien die Relationen $>$, $\leq$ und $\geq$ für beliebige $s, t \in \Sigma^*$ definiert als:

- $s > t \Leftrightarrow t < s$
- $s \leq t \Leftrightarrow s < t \;\vee\; s = t$
- $s \geq t \Leftrightarrow s > t \;\vee\; s = t$

Die so entstehende Totalordnung $\leq$ wird auch *lexikographische Ordnung* genannt. Es folgt die Definition der lexikographischen Namen, mit denen zwei beliebig lange Strings s und t in konstanter Zeit verglichen werden, ohne auf die einzelne Zeichen der Strings zuzugreifen.

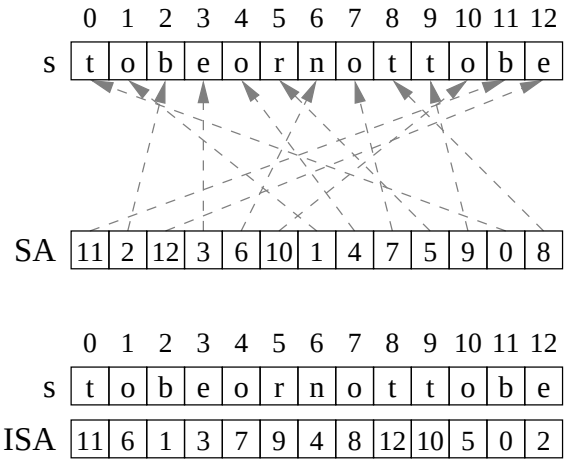
Definition 3.2.2
Gegeben sei eine nichtleere, endliche Menge $S \subseteq \Sigma^*$ von Strings. Eine Funktion $l : S \to [0, |S|)$ heißt lexikographische Benennung, wenn für beliebige $s, t \in S$ gilt:

$$s \leq t \Leftrightarrow l(s) \leq l(t).$$

Die Funktion l ist bijektiv und durch die lexikographische Reihenfolge der Strings in S eindeutig bestimmt. Zu einem String $s \in S$ nennen wir den Wert $l(s)$ auch lexikographischen Namen von s und schreiben kurz s'.

3.3 Suffix-Array

Durch die in 3.2.1 definierte Ordnung $<$ lässt sich eine endliche Menge von paarweise verschiedenen Strings auf genau eine Art und Weise mit dem kleinsten beginnend der Größe nach anordnen. Diese Reihenfolge wird auch lexikographische Reihenfolge genannt.
Nach 3.2.1 sind Strings unterschiedlicher Länge verschieden. Ein String $s \in \Sigma^n$ besitzt genau n paarweise verschiedene Suffixe $s_i = s[i..n)$ mit $i \in [0..n)$.



11	be
2	beornottobe
12	e
3	eornottobe
6	nottobe
10	obe
1	obeornottobe
4	ornottobe
7	ottobe
5	rnottobe
9	tobe
0	tobeornottobe
8	ttobe

Abbildung 3.1: Suffix-Array SA und inverses Suffix-Array ISA zum String s

Definition 3.3.1
Es sei $s \in \Sigma^n$. Ein Suffix-Array für s ist ein Feld Länge n, das die Startpositionen i der Suffixe s_i in lexikographischer Reihenfolge enthält.

Das Suffix-Array eines Strings ist mit dieser Definition eindeutig bestimmt. Ein Beispiel für s = tobeornottobe ist in Abbildung 3.1 angegeben. Ist SA das Suffix-Array eines Strings $s \in \Sigma^n$, dann hat s_i den *Rang* j, wenn $\mathrm{SA}[j] = i$ gilt. Das Suffix-Array SA kann auch als Permutation der Menge $[0..n)$ aufgefasst werden. Darum ist folgende Definition sinnvoll:

Definition 3.3.2
Es sei SA ein Suffix-Array der Länge n. Das inverse Suffix-Array ISA ist ein Feld Länge n, für das gilt:

$$\forall_{i \in [0..n)} \mathrm{SA}[\mathrm{ISA}[i]] = i.$$

Offensichtlich ist das inverse Suffix-Array so wohldefiniert. Das inverse Suffix-Array gibt nun zu einem Suffix s_i dessen Rang $\mathrm{ISA}[i]$ an.

3.4 LCP-Tabelle

Für zwei Strings s und t wird der längste gemeinsame Präfix auch als *longest common prefix* bezeichnet. Der lcp-Wert von s und t gibt die Länge dieses gemeinsamen Präfix an.

Definition 3.4.1
Seien $s, t \in \Sigma^*$. Dann ist der lcp-Wert von s und t definiert als:

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

Die so genannte LCP-Tabelle enthält die lcp-Werte lexikographisch aufeinanderfolgender Suffixe.

Definition 3.4.2
Es sei $s \in \Sigma^n$ und SA *das zugehörige Suffix-Array. Die zu s und* SA *gehörige LCP-Tabelle* LCP *ist ein Feld Länge n, für das gilt:*

$$\mathrm{LCP}[i] = \begin{cases} \mathrm{lcp}(s_{\mathrm{SA}[i-1]}, s_{\mathrm{SA}[i]}) & \text{für } i > 0, \\ 0 & \text{für } i = 0. \end{cases}$$

0	be
2	beornottobe
0	e
1	eornottobe
0	nottobe
0	obe
3	obeornottobe
1	ornottobe
1	ottobe
0	rnottobe
0	tobe
4	tobeornottobe
1	ttobe

Abbildung 3.2: LCP-Tabelle (linke Spalte) zum String $s =$ tobeornottobe

Ein Beispiel einer LCP-Tabelle für $s =$ tobeornottobe ist in Abbildung 3.2 angegeben.
Eine wichtige Eigenschaft der LCP-Tabelle ist, dass der lcp-Wert zweier beliebiger Suffixe das Minimum der lcp-Werte aller benachbarten Suffixpaare zwischen den beiden Suffixen im Suffix-Array ist:

$$\mathrm{lcp}(s_{\mathrm{SA}[i]}, s_{\mathrm{SA}[j]}) = \min_{k \in [i+1,j]} \mathrm{LCP}[k] \quad \text{, für } i < j.$$

3.5 Konventionen

In den folgenden Kapiteln werden Algorithmen vorgestellt, deren Eingabe Strings oder Felder sind. Σ sei dann ein beliebiges, endliches Alphabet, $s \in \Sigma^*$ ein beliebiger String. Ohne weitere Bemerkungen gilt $n = N = |s|$, SA ist das zu s

gehörige Suffix-Array, ISA das zu SA inverse Suffix-Array und LCP die zu s und SA gehörige LCP-Tabelle.

Geordnete Paare, Tripel oder Tupel werden durch spitze Klammern $\langle\ \rangle$ gekennzeichnet. Um einen String als solchen hervorzuheben, werden seine Zeichen in eckige Klammern [ ] gefasst. Sind die Elemente von einem Index abhängig, sind sie nach aufsteigendem Index geordnet. $\left[2^i : i \in [0..n)\right]$ bspw. ergibt den String $[1, 2, 4, \ldots, 2^n]$. Ist die Reihenfolge der Elemente irrelevant, werden geschweifte Klammern { } verwendet.

Für die internen Algorithmen und speziell ihren Speicherbedarf wird angenommen, dass der Typ zum Speichern einer Zahl aus $[0..n)$ genau 4 Byte lang ist. Unter dieser Annahme kann n höchstens 2^{32} sein. Tatsächlich sind die Implementierungen dieser Arbeit aber generisch und mit beliebigen skalaren Typen spezialisierbar. Der Speicherbedarf wird in einigen Fällen daher auch in *Worten* angegeben, d.h. bezüglich des Typs der Zeichen des Suffix-Array, ohne Festlegung der Wortgröße.

3.6 Speichermodelle

Die Komplexität eines Algorithmus beschreibt in der Komplexitätstheorie seinen maximalen Ressourcenbedarf in Abhängigkeit einer Problemgröße n. Die betrachteten Ressourcen sind unter anderem die benötigten Rechenschritte und der Speicherbedarf. Theoretische Analysen benutzen dabei meist das RAM-Modell [4]. In diesem Modell ist der Speicher unbegrenzt und ein Zugriff auf beliebige Speicherzellen kostet einheitlich viel. Das RAM-Modell wird hauptsächlich benutzt, um Primärspeicher-Algorithmen zu analysieren. Sekundärspeicher hat allerdings andere Eigenschaften. So ist ein Zugriff 1000-100.000 mal langsamer als ein Hauptspeicher- oder Cache-Zugriff. Weiterhin liefert ein Zugriff immer einen Block von Daten zurück. Sekundärspeicher-Algorithmen benutzen also zwei Speicherhierarchien mit sehr verschiedenen Eigenschaften. Ein einheitliches Kostenmaß modelliert den Speicherzugriff nur schlecht.

Ein erstes Modell, welches die Eigenschaften von Sekundärspeicher mitberücksichtigt, wurde von Aggrawal und Vitter [3] veröffentlicht. In diesem Modell besitzt ein Computer einen beschränkten Primärspeicher der Größe M und unbegrenzten Sekundärspeicher. Einzelne Sekundärspeicherzugriffe (I/O-Zugriffe) übertragen immer einen Block von Daten der Größe B. Eine Festplatte besteht aus $P \geq 1$ unabhängigen Schreib-/Lese-Köpfen, so dass parallel bis zu $P \cdot B$ Daten innerhalb eines I/O-Zugriffs übertragen werden können.

Tatsächlich besitzen heutige Festplatten mehrere Köpfe, diese können aber weder unabhängig voneinander bewegt werden, noch haben sie Zugriff auf dieselben Daten. Eine Weiterentwicklung erfolgte von Vitter und Shriver [27], die in ihrem I/O-Modell den Sekundärspeicher als $P \geq 1$ unabhängige Festplatten

mit jeweils einem Kopf betrachten. In diesem Modell besteht der Computer aus $P' \geq 1$ Prozessoren. In dieser Arbeit wird $P' = 1$ angenommen.

Nachteil des Modells [27] ist, dass nicht zwischen wahlfreien (engl. random I/O accesses) und sequentiellen I/O-Zugriffen (engl. bulk I/O accesses) unterschieden wird. Sequentielle Zugriffe sind in der Praxis schneller als wahlfreie Zugriffe. Das Modell von Farach et al. [11] erweitert das Modell von Vitter und Shriver [27] um eben diese Unterscheidung und schließt die Zahl der wahlfreien Zugriffe in die Analyse mit ein. Als sequentiell werden $c \cdot \frac{M}{B}$ zusammenhängende I/O-Zugriffe definiert für ein $c \geq 1$.

In dieser Arbeit werden externe Algorithmen, also Algorithmen, die Sekundärspeicher verwenden, mit dem Modell von Farach et al. [11] analysiert. Ressourcen dieses Modells sind:

1. Zahl der I/O-Zugriffe
2. Zahl der CPU-Instruktionen nach dem RAM-Modell
3. Sekundärspeicherbedarf (I/O-Bedarf)

Bedingungen sind:

- $1 \leq B \leq \frac{M}{2}$
- $1 \leq P \leq \left\lfloor \frac{M}{B} \right\rfloor$

Für interne Algorithmen, also Algorithmen, die ausschließlich Primärspeicher verwenden, wird das klassische RAM-Modell [4] verwendet.

Die wichtigsten Ergebnisse von Vitter und Shriver [27] sind asymptotische obere und untere Schranken für die Zahl der I/O-Zugriffe die zum Sortieren oder Permutieren eines Feldes mit N Elementen benötigt werden. So ist die Zahl der benötigten I/O-Zugriffe für das vergleichsbasierte Sortieren von N Elementen in ihrem Modell

$$\text{sort}(N) = \Theta\left(\frac{N}{PB} \log_{\frac{M}{B}} \frac{N}{B}\right).$$

Das Permutieren von N Elementen benötigt

$$\text{permute}(N) = \Theta\left(\min\left\{\frac{N}{P}, \frac{N}{PB} \log_{\frac{M}{B}} \frac{N}{B}\right\}\right),$$

das sequentielle Lesen bzw. Schreiben von N Elementen

$$\text{scan}(N) = \Theta\left(\frac{N}{PB}\right)$$

und die Suche in einem externen Suchbaum

$$\text{search}(N) = \Theta\left(\log_{PB} N\right)$$

I/O-Zugriffe.

Kapitel 4

Sortierverfahren

In diesem Kapitel werden zwei klassische Sortierverfahren und ein neuer laufzeitoptimaler externer Permutationsalgorithmus vorgestellt. Alle drei Algorithmen werden in dieser Arbeit verwendet, wobei die beiden externen Algorithmen das Fundament des im folgenden Kapitel beschriebenen Pipelinings bilden. Doch zunächst soll das Sortierproblem näher definiert werden.

Gegeben ist ein Feld F der Länge n über dem Alphabet Ψ. Weiter sei eine Totalordnung $\leq_\Psi$ definiert über Ψ. Gesucht ist ein Feld F', so dass $F'[i] \leq_\Psi F'[j]$ gilt für alle $0 \leq i < j < n$. Ein Sortierverfahren ist ein Algorithmus, der F in F' überführt. Die Elemente von Ψ nennen wir Datensätze. Meist bezieht sich die Relation $\leq_\Psi$ nur auf einen Teil des Datensatzes (sortiert man bspw. eine Menge von Personen dem Alter entsprechend), diesen Teil nennen wir Sortierschlüssel oder Schlüsselwert.

4.1 Radixsort

Wir betrachten den Fall $\Psi = \Sigma^k$ für kleines k, Σ sei endlich und $\leq_\Psi$ die lexikographische Ordnung. Stabile Sortierverfahren sind Sortierverfahren, welche die Reihenfolge der Datensätze, deren Sortierschlüssel gleich sind, erhalten. Mit Hilfe stabiler Sortierverfahren kann man die k-lets der Menge Ψ auch sortieren, indem man sie zuerst stabil nach dem letzten Zeichen, dann stabil nach dem vorletzten Zeichen usw. und zuletzt stabil nach dem ersten Zeichen sortiert. Nach den k Durchläufen ist die Menge vollständig sortiert.

Radixsort oder auch Distributionsort funktioniert auf diese Art und Weise und ruft zum stabilen Sortieren nach einer bestimmten Stelle (Radix) das eigentliche Sortierverfahren Radixpass k-mal auf. Ein Beispiel für $n = 4$, $k = 3$ und $\Sigma = \{A, C, G, T\}$ ist in Abbildung 4.1 angegeben.

Radixsort ist auch für den Fall $\Psi = \Sigma^1 \cup \Sigma^2 \cup \ldots \cup \Sigma^k$ geeignet. In diesem Fall sei \$ ein nicht in Σ vorkommendes Zeichen, $\Sigma' := \Sigma \cup \$$ und $\forall_{x \in \Sigma} \$ < x$. Strings,

deren Länge kleiner ist als k, werden mit \$-Zeichen aufgefüllt, so dass wir das Problem auf $\Psi' = \Sigma'^k$ reduzieren können.

Radixpass ist nicht-vergleichsbasiert und hat eine Laufzeit von $\mathcal{O}(n + |\Sigma|)$ und einen zusätzlichen Speicherbedarf von $\mathcal{O}(|\Sigma|)$. Voraussetzung für den Aufruf von Radixpass ist eine Abbildung $\varphi : \Sigma \to [0, m)$ mit $m = \mathcal{O}(|\Sigma|)$, die ordnungserhaltend ist. Radixsort hat eine Laufzeit von $\mathcal{O}(k \cdot n + k \cdot |\Sigma|)$ und einen zusätzlichen Speicherbedarf von $\mathcal{O}(|\Sigma|)$.

CAT		CT**C**		T**A**G		**A**CT
ACT	→	TA**G**	→	C**A**T	→	**C**AT
TAG		CA**T**		A**C**T		**C**TC
CTC		AC**T**		C**T**C		**T**AG

Abbildung 4.1: Radixsort-Beispiel

4.2 Externes Mergesort

Es seien $N := n$ und M, B, P die Parameter aus 3.6 und $3 \cdot B \leq M$. Zum Sortieren großer Datenmengen ($N > M$ oder $|\Sigma| > M$) ist das in 4.1 vorgestellte Sortierverfahren ungeeignet. Besser geeignet sind externe Sortierverfahren wie bspw. das Externe Mergesort. Das Externe Mergesort besteht aus 2 Phasen:

- Sortierphase
- Verschmelzungsphase

Zur Vereinfachung der Erklärung nehmen wir $B|M$, $M|N$ und $P = 1$ an. Ein Element von F benötige genau eine Speichereinheit. In Phase 1 wird das Feld F in $\frac{N}{M}$ Teile der Größe M partitioniert, die einzeln mit einem internen Sortierverfahren sortiert werden. Diese in sich sortierten Teilsequenzen werden auch *Runs* genannt.

In Phase 2 werden nun in mehreren Durchläufen so lange mehrere ($k \geq 2$) Runs zu einem neuen Run vereinigt, bis nur noch ein Run existiert. Die Verschmelzung von k Runs (engl. k-way Merging) funktioniert so, dass von allen k Teilsequenzen jeweils der Block mit dem kleinsten Element in den Hauptspeicher geladen wird. Nun werden die Elemente der Blöcke der Größe nach vereinigt, bis ein Block eines Runs im Speicher kein ungelesenes Element mehr enthält. Für diesen Run wird der nachfolgende Block geladen und die Vereinigung fortgesetzt. Im Speicher sind so immer die aktuell kleinsten Elemente enthalten. Das kleinste ungelesene Element der k Blöcke kann effizient mit einer Priority Queue der Größe k ermittelt werden.

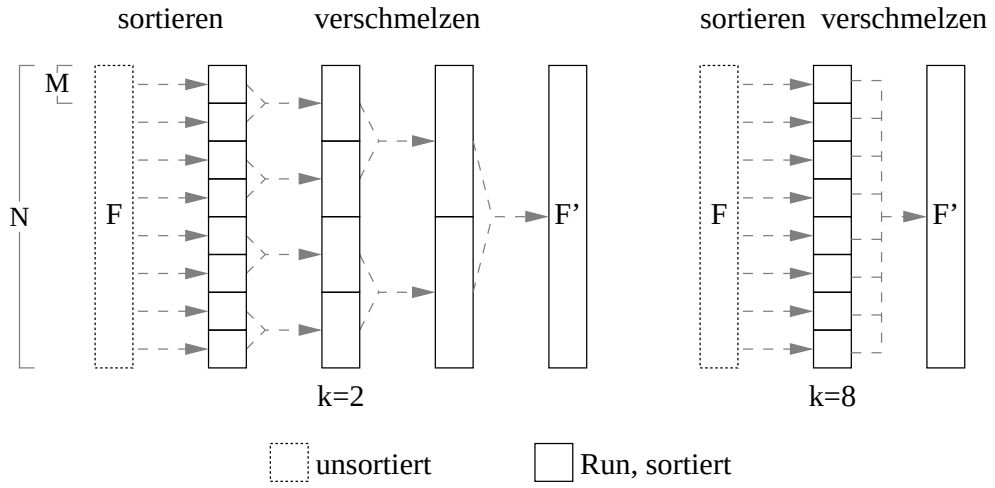


Abbildung 4.2: Externes Mergesort mit $k = 2$ (links) und $k = \frac{N}{M}$ (rechts)

Ist k die Zahl der Runs, die jeweils zu einem neuen Run verschmolzen werden, so besteht Phase 2 aus $\lceil \log_k \frac{N}{M} \rceil$ Durchläufen. Der Ablauf von Mergesort ist schematisch in Abbildung 4.2 für verschiedene k dargestellt. Phase 1 benötigt M und Phase 2 $(k+1) \cdot B$ Speicher. Die Zahl der I/O-Zugriffe ist $2\frac{N}{B} + 2\frac{N}{B} \lceil \log_k \frac{N}{M} \rceil = \mathcal{O}(\frac{N}{B} \log_k \frac{N}{M})$. Offensichtlich reduziert sich mit wachsendem k die Zahl der I/O-Zugriffe. Ein maximales k wäre $k := \min(\frac{M}{B} - 1, \frac{N}{M})$. Im Fall $k = \frac{M}{B} - 1 \leq \frac{N}{M}$ benötigt Mergesort $\mathcal{O}(\frac{N}{B} \log_{\frac{M}{B}-1} \frac{N}{M}) = \mathcal{O}(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{M}) = \mathcal{O}(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ und im Fall $k = \frac{N}{M} \leq \frac{M}{B} - 1$ benötigt Mergesort $\mathcal{O}(\frac{N}{B}) \subseteq \mathcal{O}(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/O-Zugriffe.

Für $P = 1$ ist Mergesort also asymptotisch I/O-optimal. Für $P > 1$ lässt sich der Algorithmus so modifizieren, dass er mit Blöcken der Größe $P \cdot B$ arbeitet, die in parallelen Streifen auf die P Platten verteilt werden (engl. Striping). Diese Variante (DSM = disk-striped mergesort) ist aber bei mehr als einem Durchlauf in Phase 2 nicht mehr I/O-optimal ($\Theta(\frac{N}{PB} \log_{\frac{M}{PB}} \frac{N}{PB})$ anstatt $\Theta(\frac{N}{PB} \log_{\frac{M}{B}} \frac{N}{B})$). In [5] und [9] werden Mergesort-Varianten vorgestellt, die mit Forecasting, Overlapping und Randomisierung bessere Ergebnisse erzielen als DSM.

In dieser Arbeit beschränken wir uns auf DSM für $P > 1$.

4.3 Externes Permutieren

Wir nehmen an, die Schlüsselwerte der Elemente in F seien eindeutig aus $[0..n)$ und $\leq_\Psi$ ordnet die Elemente aufsteigend nach ihren Schlüsselwerten. Das Problem, F zu sortieren, reduziert sich zu dem Problem, F zu permutieren, wobei der Schlüsselwert eines Datensatzes dessen Zielposition in F' angibt.

Wie in 3.6 bereits vorgestellt wurde, ist die asymptotische untere Schranke der I/O-Zugriffe für das externe Permutieren $\Omega\left(\min\left\{\frac{N}{P}, \frac{N}{PB} \log_{\frac{M}{B}} \frac{N}{B}\right\}\right)$. Diese

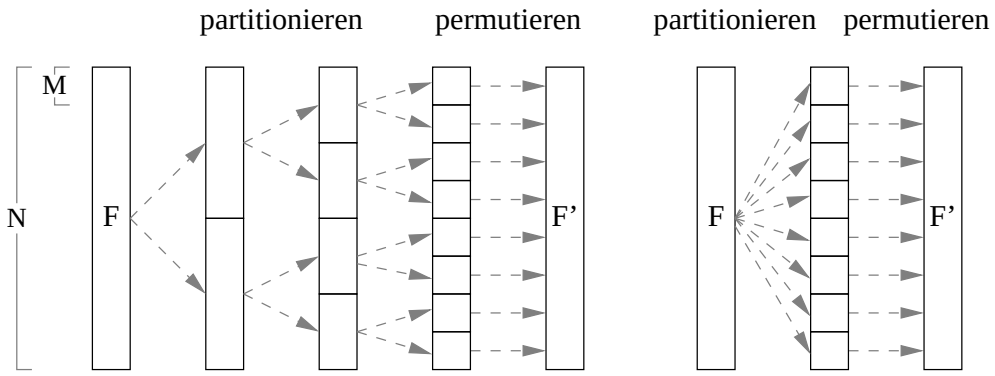


Abbildung 4.3: Externer Permutationsalgorithmus mit $k = 2$ (links) und $k = \frac{N}{M}$ (rechts)

Abschätzung ist scharf und wird erreicht, wenn man zum Permutieren ein I/O-optimales Sortierverfahren verwendet und im Fall $\frac{N}{P} < \frac{N}{PB} \log_{\frac{M}{B}} \frac{N}{B}$ (tritt ein für kleine B und P) die Datensätze auf naive Weise wie in [27] permutiert.

Verwendet man ein Sortierverfahren zum Permutieren benötigt dieses mindestens $\Omega(N \log N)$ interne Laufzeit. Hier soll nun ein einfacher Permutationsalgorithmus vorgestellt werden, mit der optimalen Laufzeit von $\mathcal{O}(N \log_{\frac{M}{B}} \frac{N}{B})$ bzw. $\mathcal{O}(N)$ für kleine B und P.

Der Algorithmus funktioniert ähnlich wie eine Umkehrung des Externen Mergesort oder das Externe Distributionsort, mit dem Unterschied, dass die Partitionen equidistant sind und im letzten Durchlauf im Speicher nicht sortiert sondern permutiert wird. Es sei $\alpha : \Psi \to [0..n)$ die Abbildung, die den Elementen von F ihren Schlüsselwert zuordnet. Der Algorithmus besteht aus den folgenden 2 Phasen:

- Partitionierungsphase
- Permutationsphase

Zu Beginn von Phase 1 partitionieren wir die Menge $[0, n)$ der Schlüsselwerte in mehrere ($k \geq 2$) zusammenhängende, möglichst equidistante Teile. Die Grenzen dieser Partition seien $0 = s_0, s_1, \cdots, s_k = n$. Ein Bucket T (engl. Bucket = Eimer) ist eine Teilmenge von Ψ. Für $j \in [0, k)$ enthalte das Bucket T_j von F alle Elemente $x \in F$ für die $s_j \leq \alpha(x) < s_{j+1}$ gilt. Als nächstes sollen nun sequentiell alle Elemente aus F auf die entsprechenden Buckets verteilt werden, Bucket T_i beginnt dabei an Position s_i der Zieldatei. Dazu werden für F und die k Buckets insgesamt $k + 1$ Puffer der Größe B im Hauptspeicher reserviert. Der erste Block von F wird in den Hauptspeicher geladen und dessen Elemente den Puffern der jeweiligen Zielbuckets angefügt. Dannach wird mit dem nächsten Block von F fortgefahren. Sollte ein Bucketpuffer voll laufen, wird er auf die Festplatte geschrieben und kann erneut gefüllt werden. Am Ende

dieses Durchgangs ist die Folge der Schlüsselwerte der Elemente im Bucket T_j eine Permutation der Menge $[s_j, s_{j+1})$. In den folgenden Durchläufen werden nun die so entstandenen Buckets jeweils wieder in k Buckets zerlegt, bis kein Bucket mehr größer als M ist.

In Phase 2 können nun die $t \geq \lceil \frac{N}{M} \rceil$ entstandenen Buckets jeweils nacheinander im Hauptspeicher permutiert werden. Für $j = 0, \cdots, t-1$ wird dazu Bucket T_j komplett in den Hauptspeicher geladen und mit einem Algorithmus linearer Laufzeit intern permutiert. In Abbildung 1 ist ein solcher Algorithmus angegeben, der T_j ohne zusätzlichen Speicher in $\mathcal{O}(|T_j|)$ permutiert. Der Algorithmus geht linear durch T_j und kopiert für $i = 0, \cdots, |T_j| - 1$ das Element $x = T_j[i]$ an dessen Zielposition $T_j[\alpha(x) - \mathit{offset}]$. Das Element an dieser Position wird aber vor dem Überschreiben gesichert, um dann im nächsten Schritt an dessen Zielposition bewegt zu werden usw. bis ein Kreis geschlossen wird. Da alle so auftretenden echten Kreise disjunkt sind, wird die innere repeat-Schleife insgesamt höchstens $|T_j|$-mal durchlaufen.

Algorithmus 1 PermuteInPlace(T, offset)

Benötigt: $[\mathit{offset}, \mathit{offset} + |T|) = \{\alpha(x) | x \in T\}$

```
for i := 0 to |T| − 1 do
    x_prev := T[i]
    j := α(x_prev) − offset
    if j ≠ i then
        repeat
            x_next := T[j]
            T[j] := x_prev
            x_prev := x_next
            j := α(x_prev) − offset
        until j = i
        T[i] := x_prev
```

Ist k die Zahl der Buckets, in die ein Bucket jeweils partitioniert wird, so besteht Phase 1 aus $\lceil \log_k \frac{N}{M} \rceil$ Durchläufen. Der Ablauf des Permutationsalgorithmus ist schematisch in Abbildung 4.3 für verschiedene k dargestellt. Bis auf die interne Laufzeit entspricht die Komplexität dieses Algorithmus in Phase 1 der Komplexität von Phase 2 des Externen Mergesort. Ebenso entspricht Phase 2 der Komplexität von Phase 1 des Externen Mergesort. Der Algorithmus ist also I/O-optimal für $P = 1$. Für $P > 1$ bedarf es noch einer geschickten Strategie beim Schreiben der Bucketpuffer, um I/O-Optimalität bei mehr als einem Durchlauf in Phase 1 zu erreichen, die aber nicht Teil dieser Arbeit sein soll. Wir begnügen uns mit Disk-Striping für $P > 1$. Die interne Laufzeit von Phase 1 ist $\mathcal{O}(N \log_k \frac{N}{M})$ und von Phase 2 $\mathcal{O}(N)$. Mit größtmöglichem k ist die gesamte interne Laufzeit also $\mathcal{O}(N \log_{\frac{M}{B}} \frac{N}{B})$. Der Algorithmus ist optimal unter

allen I/O-optimalen Permutationsalgorithmen, da die Menge der in einem I/O-optimalen Algorithmus transferierten und verarbeiteten Daten $\Omega(N \log_{\frac{M}{B}} \frac{N}{B})$ ist.

Der Aufbau des Algorithmus ermöglicht auf einfache Weise eine effiziente Implementierung mit überlappenden I/O-Zugriffen, Prefetching und Write Buffering.

Kapitel 5

Pipelining

In den folgenden Abschnitten soll das Pipelining-Konzept beschrieben werden, welches aus der digitalen Signalverarbeitung und dem Mikroprozessorentwurf bekannt ist. In [8] wird es erstmals für externe Algorithmen vorgestellt und benutzt. Pipelining kann die Entwicklung von I/O-effizienten externen Algorithmen stark vereinfachen. Es werden die Umsetzung des Konzepts in dieser Arbeit beschrieben und anschließend Grundlagen zur theoretischen Analyse von externen Algorithmen geschaffen.

5.1 Das Konzept

Wir nehmen an, wir haben eine große Eingabedatenmenge und eine Reihe von Algorithmen, die nacheinander ausgeführt werden müssen. Die Ausgabe eines Algorithmus sei die Eingabe des nachfolgenden Algorithmus in der Kette. Um zu verhindern, dass zwischen zwei aufeinanderfolgenden Algorithmen jeweils der komplette Datensatz auf die Festplatte geschrieben und von ihr gelesen werden muss, könnte man alle Algorithmen zu einem einzigen großen Algorithmus vereinen. Auf Kosten der Modularität und Wiederverwendbarkeit der einzelnen Algorithmen erhielte man einen großen und für die Fehlersuche unübersichtlichen Algorithmus.

Entsprechen die einzelnen Algorithmen dem Pipelining-Konzept, lassen sie sich zu einem großen Algorithmus zusammensetzen, ohne dass auf Modularität und Korrektheit der einzelnen Algorithmen verzichtet werden muss. Über generische Schnittstellen werden die Daten dabei direkt und ohne Sekundärspeicher-Umwege ausgetauscht.

5.2 Pipes und Pools

Das primäre Paradigma des Konzepts ist, das Speichern von Daten möglichst zu vermeiden. Betrachtet man in einem externen Algorithmus die einzelnen

Berechnungschritte (Module) als Knoten und den Datenfluss zwischen ihnen als Kanten erhält man einen gerichteten Flussgraph $G = (V, E)$ und $V = P_i \cup P_o \cup P_u$.

Das in dieser Arbeit verwendete Pipelining-Konzept unterscheidet zwei Hauptklassen von Modulen:

- Pipes P_i
- Pools P_o

Pipes können einen beliebigen eingehenden Knotengrad und höchstens eine ausgehende Kante haben. Es sind einfache Module, die zu jedem Zeitpunkt nur einen kleinen Teil des Eingabestroms zwischenspeichern und bereits während des Einlesens Daten verarbeiten und ausgeben (vgl. streaming node [8]).

Pools können höchstens eine eingehende Kante und eine ausgehende Kante haben. Pools greifen auf die Gesamtheit der Eingabedaten zu und speichern bzw. verarbeiten diese dazu ggfs. im Sekundärspeicher. Meist hängt jedes einzelne Zeichen der Ausgabe vom kompletten Eingabestrom ab (bspw. beim Sortieren), so dass Pools erst nach dem vollständigen Verarbeiten aller Eingabedaten, lesbar sind. Zu den wichtigsten Unterklassen des Pools zählen der *Simple Pool* zum Speichern bzw. Lesen der kompletten Daten (vgl. file node [8]), der *Sorter* zum Sortieren (vgl. sorting node [8]) und der *Mapper* zum Permutieren von Eingabedaten.

Daten werden zwischen den einzelnen Modulen immer nur sequentiell und zeichenweise entlang der Kanten ausgetauscht. Den Kantenursprung nennen wir Sender und das Ziel Empfänger.

Ist G zusammenhängend und azyklisch und enthält Pools nur ohne eingehende Kanten, bezeichnen wir G auch als *Pipeline*. Pipes nennen wir *pull-aktiv*, d.h. eine Pipe liest die Daten seiner Sender erst bei Bedarf. In diesem Fall sind die Sender also passiv (*push-passiv*) und die Pipe aktiv. Da diese Pipe auch erst aktiv wird, wenn von ihr gelesen wird, ist die gesamte Pipeline pull-aktiv. Pipes mit verschiedenen Empfängern, die parallel lesen sind so nicht möglich. Pools nennen wir *pull-passiv*, da die Sender des Pools den Transfer von Daten aktiv (*push-aktiv*) initiieren müssen. Bisher haben wir aber noch keine push-aktiven Module kennengelernt.

5.3 Pumps

Um Pipes mit mehr als einem Empfänger zuzulassen bzw. Pools füllen zu können, müssen wir das Modell erweitern. Wir bezeichnen ein Modul als *Pump*, wenn es pull-aktiv und push-aktiv ist. Es kann mehrere Empfänger haben, diese müssen aber Pools sein. Pumps stellen die Verbindungsstücke der Pipelines dar

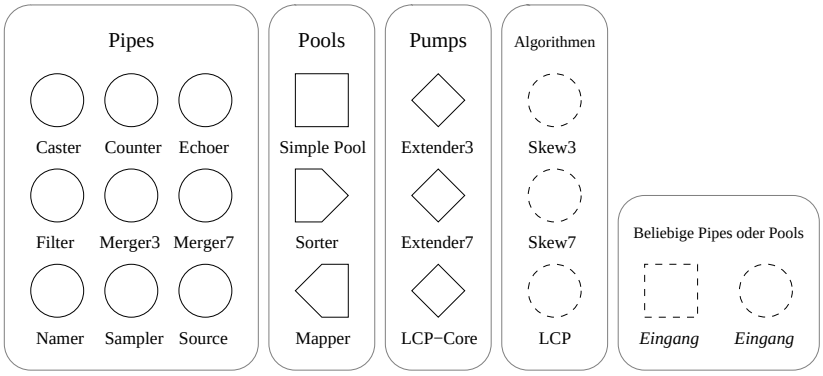


Abbildung 5.1: Symbole der Pipelining-Module

und die einfachste Pump (triviale Pump) verbindet genau zwei Pipelines und leitet den Eingabestrom einfach in den Empfängerpool weiter. Algorithmische Fallunterscheidungen oder bedingte Datentransfers können mit Hilfe von Pumps realisiert werden, indem je nach Fall dynamisch der entsprechende Empfänger gefüllt oder der entsprechende Sender gelesen wird. Da Pipelines azyklisch sind, können Schleifen nicht direkt innerhalb einer Pipeline, wohl aber mit Pumps modelliert werden. Zum Überprüfen der Schleifenbedingung wird eine Pump mit mindestens 2 Ein- und Ausgängen benutzt, die den Eingabestrom in eine Pipeline für den Schleifenkörper oder im Fall des Schleifenendes zum anderen Ausgang durchleitet. Die Pipeline des Schleifenkörpers ist gleichzeitig ein Sender der Pump und wird im zweiten und den darauffolgenden Durchläufen als Eingabestrom der Pump gewählt. Notwendig ist, dass der Schleifenkörper mindestens 2 Pools enthält, da ein Pool nicht gleichzeitig gelesen und geschrieben werden kann. Mit dieser Einschränkung können wir nun also auch Schleifen in G zulassen. Wir können auch Pipes oder Pools mit mehr als einer ausgehenden Kante zulassen, was bedeutet, dass diese Module mehrere Male sequentiell gelesen werden. Pools dürfen sogar parallel gelesen werden.

Ist G zusammenhängend, sind in G die Sender aller Pools Pumps und enthält jeder Kreis mindestens 2 Pools bezeichnen wir G als *Aquädukt*. Jede triviale Pump hat in G genau eine Pipe als Sender und einen Pool als Empfänger. Zur Darstellung von G werden die Symbole aus Abbildung 5.1 verwendet. Sind in einer Darstellung von G eine Pipe und ein Pool direkt durch genau eine Kante verbunden, so seien die beiden Module in G aber durch eine triviale Pump verbunden. Zur besseren Übersicht soll im Weiteren diese vereinfachte Darstellung gewählt werden.

5.4 Die Umsetzung

Alle pull- oder push-passiven Pipeline-Module (Pipes und Pools) haben eine einheitliche Schnittstelle für Datentransfer und Flusskontrolle (Kapitel 9.3.3).

Ist ein Algorithmus in Form eines Aquädukts gegeben, können die einzelnen Pipeline-Module mit dieser Schnittstelle unabhängig implementiert, getestet und zusammengesetzt werden. Tabelle 5.4 zeigt die in den folgenden Kapiteln benutzten Pipes und eine vereinfachte Darstellung ihrer Semantik. Alle Pipes haben einen Ausgang und bis auf source (keinen Eingang), merger3 und merger7 (2 bzw. 5 Eingänge) genau einen Eingang. x_i bezeichnet das zuletzt gelesene Zeichen des Eingabestroms $x_0x_1 \dots x_{n-1} \in \Psi^n$. Der dann am Ausgang anliegende Wert ist in der rechten Spalte ersichtlich.

Die in Kapitel 4 vorgestellten externen Sortier- und Permutieralgorithmen wurden als Pool-Module Sorter und Mapper implementiert mit der Relation $\leq_\Psi$ bzw. der Abbildung α als Parameter und genau einem Durchlauf in der Verschmelzungs- bzw. Partionierungsphase, um auch für $P > 1$ mit DSM I/O-Optimalität zu erreichen. Die Zahl der Durchläufe ist 1, wenn $k = \frac{N}{M}$ bzw. $M \geq \sqrt{2NPB}$ gilt ($(N + M)PB \leq M^2 \Rightarrow \frac{N}{M} \leq \frac{M}{PB} - 1$). Diese Bedingung ist in den meisten praktischen Anwendungsfällen erfüllbar, da bspw. für derzeit gängige Blockgrößen von $B = 4\text{KB}$, $P = 1$ und $d \cdot N = d \cdot 2\text{GB}$ Daten der Speicherbedarf $d \cdot M = d \cdot 2\text{MB}$ ist, wenn ein Datensatz d Byte groß ist. Sollte diese Bedingung nicht erfüllbar sein, arbeiten die Pool-Module nicht mehr I/O-zugriffsoptimal. Alle implementierten Pool-Module benutzen Prefetching und Write Buffering Techniken und führen asynchrone, parallele I/O-Zugriffe aus, um einen möglichst hohen Durchsatz zu erzielen. Die zwei Phasen der Algorithmen werden während des Füllens bzw. Lesens der Pools ausgeführt.

Pumps müssen keine Schnittstelle zur Verfügung stellen, da sie den Datentransfer ausschließlich aktiv übernehmen. Sie können als einfache Methoden implementiert werden, deren Argumente Sender und Empfänger sind. Aquädukte können auch selbst wieder als Pipes oder Pumps gekapselt werden und so auf einfache Weise rekursiv in sich selbst oder in anderen Aquädukten benutzt werden. Damit ein Aquädukt an den Senken ausgelesen werden kann, müssen vorher nacheinander alle enthaltenen Pumps in topologischer Reihenfolge aktiviert werden. Am Datenaustausch beteiligt sind ausschließlich die gerade aktivierte Pump und die in Hin- und Rückrichtung als nächstes erreichbaren Pools einschließlich aller dazwischenliegenden Pipes. Die Eingangsdaten fließen so von Quellen zu Senken und werden auf dem Weg dorthin in Pools temporär zwischengespeichert.

5.5 Komplexität

In den folgendenen Kapiteln werden die Komplexitätsgrößen in Zeit- bzw. Speicherkomplexitäten unterteilt. Es zählen dabei CPU-Laufzeit, I/O-Zugriffe und wahlfreie I/Os zu den Zeitkomplexitäten und Speicher- und I/O-Bedarf zu den

[1] Typkonvertierung

Pipe	Semantik
caster(T)	$x_i \to {}^1(T)x_i$
counter	$x_i \to \langle x_i, i\rangle$
echoer(e)	$x_i \to \langle i, x_i x_{i-1} \dots x_{i-e+1}\rangle$
filter(φ)	$x_i \to \varphi(x_i)$
merger3	leitet kleinstes Element von 2 Sendern weiter (Skew)
merger7	leitet kleinstes Element von 5 Sendern weiter (Skew7)
namer($=_\Psi$)	$x_i \to \langle i, x'_{i-1}\rangle$ für $x_i =_\Psi x_{i-1}$, sonst $x_i \to \langle i, x'_{i-1}+1\rangle$
sampler($D \subseteq \mathbb{Z}_v$)	$x_{i+v-1} \to \langle n-i, x_i x_{i+1} \dots x_{i+v-1}\rangle$ wenn $n-i \in D$
source(s)	$\to s[i]$

Pool	Semantik
simple_pool	$x_0 x_1 \dots x_{n-1} \Rightarrow x_0 x_1 \dots x_{n-1}$
sorter($\leq_\Psi$)	$x_0 x_1 \dots x_{n-1} \Rightarrow x_{\pi(0)} x_{\pi(1)} \dots x_{\pi(n-1)}$ mit $x_{\pi(0)} \leq_\Psi \dots \leq_\Psi x_{\pi(n-1)}$
mapper(α)	$x_0 x_1 \dots x_{n-1} \Rightarrow x_{\pi(0)} x_{\pi(1)} \dots x_{\pi(n-1)}$ mit $\alpha(x_{\pi(i)}) = i$

Tabelle 5.1: Pipeline-Module mit zugehörigen Ausgabeströmen

Speicherkomplexitäten. Die Zeitkomplexität eines externen Algorithmus ergibt sich aus der Summe und die Speicherkomplexität aus dem Maximum der Zeit- bzw. Speicherkomplexitäten aller nacheinander aktivierten Pumps im entsprechenden Aquädukt. Die Komplexitäten einer Pump setzen sich wiederum aus der Summe bzw. dem Maximum der Komplexitäten der beteiligten Pools und Pipes zusammen. Zum I/O-Bedarf einer Pump kommt dabei zusätzlich noch der I/O-Bedarf von Pools, die vorher gefüllt und später gelesen werden, hinzu.

Definition 5.5.1
Es sei ein beliebiger Algorithmus gegeben. Die Funktion $t : \mathbb{R} \to \mathbb{R}$ sei eine Komplexitätsgröße in Abhängigkeit der Problemgröße n im worst-case. Wir nennen t reduzierbar, wenn folgende Ungleichungen erfüllt sind:

$$\begin{aligned} t(a) + t(b) &\leq t(a+b) \\ t_d(x) &\leq t(d \cdot x), \end{aligned}$$

wobei t_d der Aufwand des Problems mit d-facher Datensatzgröße sei.

Zu einem beliebigen Pipelining-Modul $p \in P_i \cup P_o$ sei $N(p)$ die Länge des Ausgabestroms. Ist $p \in P_o$ so sei $M(p)$ die Größe des dem Modul zur Verfügung stehenden Speichers, gemessen in Datensätzen. Tabelle 5.2 zeigt zu den in dieser Arbeit implementierten Pipes und Pools die Komplexitäten eines kompletten Schreib- bzw. Lesevorgangs. Als Pipes gekapselte Algorithmen oder Pumps sind nicht in der Tabelle aufgeführt und werden einzeln betrachtet.

Geht man davon aus, dass M und B absolute Speichergrößen beschreiben und deshalb antiproportional zu d sind, und verzichtet zur Vereinfachung auf Gauß-

klammern[2], so sind die I/O-Zugriffe und wahlfreien I/Os der Pools in Tabelle 5.2 reduzierbare Komplexitätsgrößen. Der Begriff reduzierbar soll nun auf Funktionsmengen erweitert werden.

Definition 5.5.2
Es sei ein beliebiger Algorithmus mit einer Komplexitätsgröße $t(n) = \mathcal{O}(T(n))$ für eine Funktion $T : \mathbb{R} \to \mathbb{R}$ gegeben. Wir nennen t asymptotisch reduzierbar, wenn folgende Bedingungen erfüllt sind:

$$\begin{aligned} \mathcal{O}(T(a) + T(b)) &\subseteq \mathcal{O}(T(a+b)) \\ \mathcal{O}(T_d(x)) &\subseteq \mathcal{O}(T(d \cdot x)), \end{aligned}$$

wobei $t(n) = \mathcal{O}(T_d(x))$ der Aufwand des Problems mit d-facher Datensatzgröße sei (d fest).

Ohne Beweis soll hier nur angegeben werden, dass die Laufzeiten der Pool-Klassen und die I/O-Zugriffe sort, scan und permute aus Kapitel 3.6 ebenfalls asymptotisch reduzierbar sind.

Bei der oberen Abschätzung der Zeitkomplexitäten von Aquädukten kann nun entsprechend der Regeln aus den Definitionen 5.5.1 und 5.5.2 gerechnet werden, wobei die Komplexitätsgrößen der Sorter, Mapper und Simple Pools mit Funktionen sort, permute und scan ausgedrückt werden, hinter denen sich die Größen für CPU-Laufzeit, I/O-Zugriffe und wahlfreie I/Os aus Tabelle 5.2 verbergen. Ebenso können aber auch allgemein die Abschätzungen Kapitel 3.6 eingesetzt werden. Gegebenenfalls wird die Schreib- oder Lesekomplexität sort_{write} bzw. sort_{read} einzeln betrachtet. Es gilt $\text{sort} = \text{sort}_{write} + \text{sort}_{read}$. Im Gegensatz zu sort und permute sei scan die Zeitkomplexität genau *eines* Lese- oder Schreibvorgangs.

Ein einfaches Beispiel

Das Beispiel in Abbildung 5.2 zeigt ein Aquädukt zum externen Sortieren und anschließenden Speichern eines Feldes von n Zahlen. Das Feld liegt im Hauptspeicher und kann über das Modul 'Source' gelesen werden. Im ersten Schritt werden die n Elemente in das Modul 'Sorter' und im zweiten Schritt aus dem Sorter in das Modul 'Simple Pool' *gepumpt*. Nach Beendigung des Algorithmus liegen die Elemente sortiert in der zum 'Simple Pool' gehörenden Datei vor. Für die Analyse dieses Algorithmus ist nur von Bedeutung, wieviele Daten in die Pools einfließen und wie oft sie gelesen werden. Für die Zeitkomplexität $T(n)$ gilt:

$$T(n) = \text{sort}_1(n) + \text{scan}_1(n) = \text{sort}(1 \cdot n) + \text{scan}(1 \cdot n)$$

[2]Durch den Verzicht kann es zu einer Differenz von einem I/O-Zugriff pro Summand kommen.

Modul	Laufzeit	Speicher	I/O-Zugriffe	wahlfreie I/Os	I/O-Vol.
Simple Pool	$\mathcal{O}(N)$	$\mathcal{O}(PB)$	$\lceil \frac{N}{PB} \rceil$	0	N
Sorter	$\mathcal{O}(N \log M)$	$\mathcal{O}(M)$	$\lceil \frac{N}{PB} \rceil$	0	N
Mapper	$\mathcal{O}(N)$	$\mathcal{O}(M)$	$\lceil \frac{N}{PB} \rceil$	$\lceil \frac{N}{PB} \rceil$	N

Modul	Laufzeit	Speicher	I/O-Zugriffe	wahlfreie I/Os	I/O-Vol.
Pipe	$\mathcal{O}(N)$	$\mathcal{O}(1)$	0	0	0
Simple Pool	$\mathcal{O}(N)$	$\mathcal{O}(PB)$	$\lceil \frac{N}{PB} \rceil$	0	N
Sorter	$\mathcal{O}(N \log \frac{N}{M})$	$\mathcal{O}(M)$	$\lceil \frac{N}{PB} \rceil$	$\lceil \frac{N}{PB} \rceil$	N
Mapper	$\mathcal{O}(N)$	$\mathcal{O}(M)$	$\lceil \frac{N}{PB} \rceil$	0	N

Tabelle 5.2: Pipeline-Module und ihre Schreib- (oben) und Lese-Komplexitäten (unten) für $M \geq \sqrt{2NPB}$

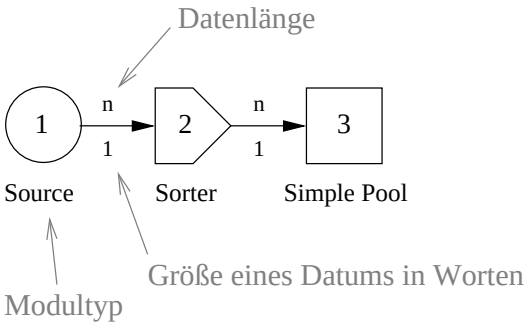


Abbildung 5.2: Beispielaquädukt

Für die Pools bedeuten die Terme ober- und unterhalb der eingehenden Pfeile, wieviele Elemente einfließen und wie groß ein einzelnes Element gemessen in Worten ist. Es sei B in Worten gegeben. Mit den Angaben aus Tabelle 5.2 können nun die I/O-Zugriffe genau bestimmt werden:

$$T_{I/O}(n) = 2\left\lceil \frac{n}{PB} \right\rceil + \left\lceil \frac{n}{PB} \right\rceil = 3\left\lceil \frac{n}{PB} \right\rceil .$$

Kapitel 6

Das Suffix-Array

Der Skew-Algorithmus wurde 2003 von Kärkkäinen und Sanders vorgestellt [13]. Er gehört zur Klasse der rekursiven Suffix-Array-Algorithmen [25] und hat eine Laufzeit von $\mathcal{O}(n)$. Dementiev et al. veröffentlichten 2005 erstmals eine externe asymptotisch I/O-optimale Variante dieses Algorithmus [8].

In den folgenden Abschnitten wird der Skew-Algorithmus zunächst vorgestellt und für eine generische Implementierung modifiziert. Es wird anschließend gezeigt, wie sich die Grundidee des Algorithmus auf beliebige so genannte Difference Covers verallgemeinern lässt. Abschließend wird eine externe Pipelining-Variante des einfachen und des um Difference Covers erweiterten Skew-Algorithmus vorgestellt und analysiert.

6.1 Der Skew-Algorithmus

Die Idee des Skew-Algorithmus und auch anderer rekursiver Linearzeitalgorithmen [17] besteht darin, die Menge der Suffixe s_i in zwei Mengen zu partitionieren und in folgenden Schritten vorzugehen:

1. Sortiere die Suffixe der ersten Menge rekursiv
2. Sortiere die Suffixe der zweiten Menge mit Hilfe der ersten Menge
3. Verschmelze beide Mengen.

Der Algorithmus soll zunächst anhand des Beispiels aus Abbildung 6.1 erklärt werden.

Schritt 1: Sortieren der Suffixe s_i, $i \in I_1 \cup I_2$

Wir nehmen zunächst eine Einteilung der Indizes i entsprechend ihrer Restklasse modulo 3 vor:

$$I_j := \{i | i \in [0..n) \ \wedge \ i \bmod 3 = j\} .$$

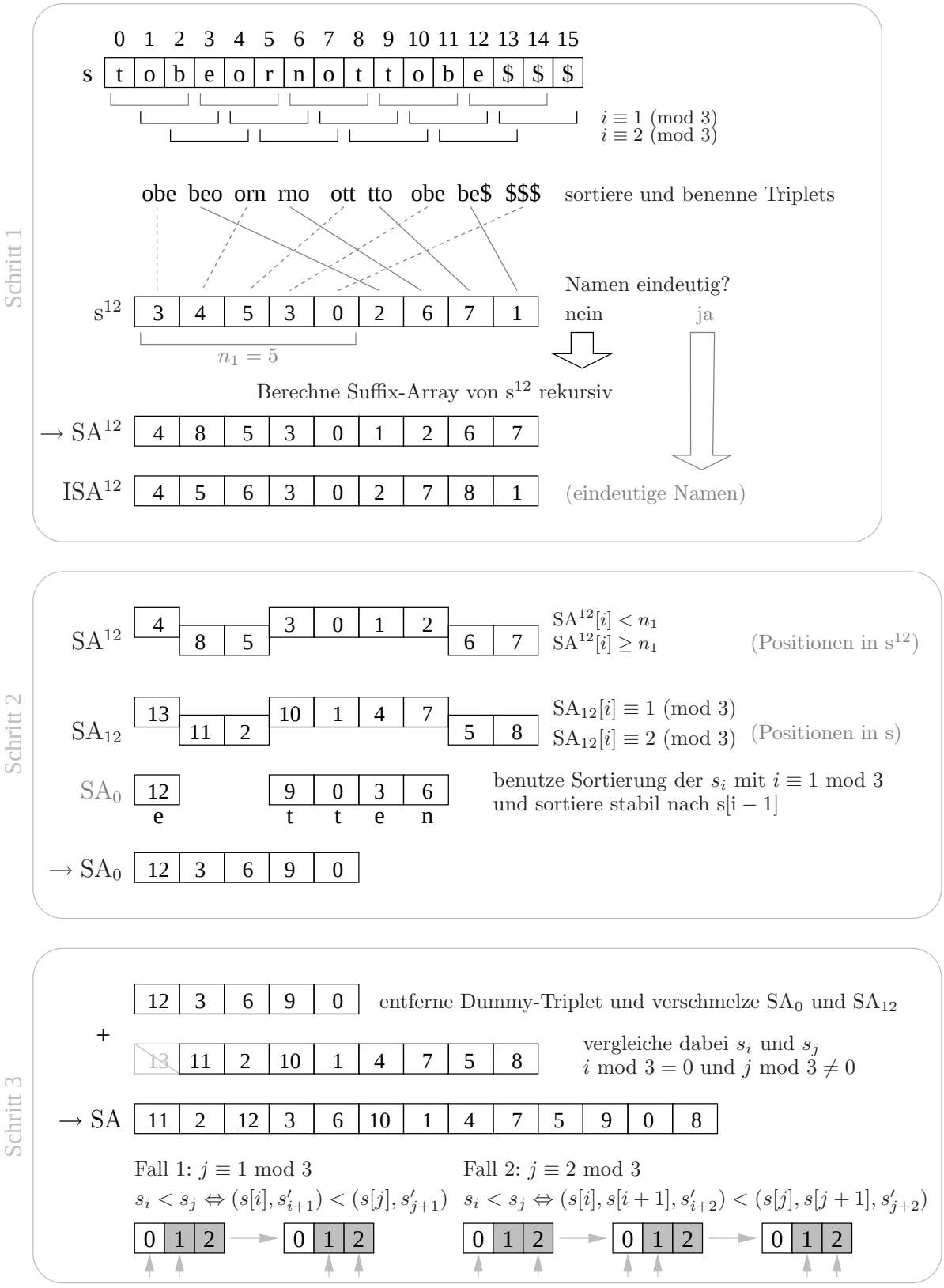


Abbildung 6.1: Klassicher Skew-Algorithmus [13]

Wir fügen 3 Dollarzeichen, ein so genanntes Dummy-Triplet, an das Ende von s an und betrachten die Triplets $t_i := s[i..i+2]$ mit $i \in I_1 \cup I_2$. Die Triplets werden mit Radixsort sortiert, wobei das Dollarzeichen kleiner sei als alle anderen Zeichen von s. Anschließend werden die Triplets t_i nach ihrer Ordnung mit Zahlen $t'_i \geq 1$ benannt.

Aus den Namen der Triplets werden zwei Strings konstruiert

$$\begin{array}{rclcrcl} s^1 & = & t'_1 t'_4 t'_7 \ldots & \text{und} & s^2 & = & t'_2 t'_5 t'_8 \ldots \\ & = & 3453 & & & = & 2671, \end{array}$$

die zusammen mit 0, dem Namen des Dummy-Triplets, zu einem neuen String verknüpft werden:

$$\begin{array}{rcl} s^{12} & = & s^1 \cdot 0 \cdot s^2 \\ & = & 345302671. \end{array}$$

Es sei $n_1 := |s^1 \cdot 0|$ die Länge des linken Teilstring einschließlich Dummy-Namen. Wir berechnen nun das Suffix-Array SA^{12} von s^{12} rekursiv und erhalten eine Ordnung der Suffixe von s^{12} die gleichzeitig einer Ordnung der Suffixe s_i, $i \in I_1 \cup I_2$ entspricht:

$$\begin{array}{lcllcl} \mathrm{SA}^{12}[0] & = & 4 & \hat{=} & 02671 & \hat{=} & \$\$\$\ldots \\ \mathrm{SA}^{12}[1] & = & 8 & \hat{=} & 1 & \hat{=} & \text{be\$} \\ \mathrm{SA}^{12}[2] & = & 5 & \hat{=} & 2671 & \hat{=} & \text{beo rno tto be\$} \\ \mathrm{SA}^{12}[3] & = & 3 & \hat{=} & 302671 & \hat{=} & \text{obe } \$\$\$\ldots \\ \mathrm{SA}^{12}[4] & = & 0 & \hat{=} & 345302671 & \hat{=} & \text{obe orn ott obe } \$\$\$\ldots \\ \mathrm{SA}^{12}[5] & = & 1 & \hat{=} & 45302671 & \hat{=} & \text{orn ott obe } \$\$\$\ldots \\ \mathrm{SA}^{12}[6] & = & 2 & \hat{=} & 5302671 & \hat{=} & \text{ott obe } \$\$\$\ldots \\ \mathrm{SA}^{12}[7] & = & 6 & \hat{=} & 671 & \hat{=} & \text{rno tto be\$} \\ \mathrm{SA}^{12}[8] & = & 7 & \hat{=} & 71 & \hat{=} & \text{tto be\$.} \end{array}$$

Das Dummy-Triplet dient zum Trennen der Worte s^1 und s^2. Da es den kleinsten Namen unter allen Triplets trägt, müssen alle Zeichenvergleiche in s^{12} spätestens dort enden. Die Werte $\mathrm{SA}_{12}[i]$ ergeben sich direkt aus den Werten $3 \cdot \mathrm{SA}^{12}[i] + 1$ bzw. $3 \cdot (\mathrm{SA}^{12}[i] - n_1) + 2$ für $\mathrm{SA}^{12}[i] \geq n_1$ und dienen hier nur der Veranschaulichung.

Schritt 2: Sortieren der Suffixe s_i, $i \in I_0$

In diesem Schritt werden alle Suffixe s_i, $i \in I_0$ sortiert. Es gilt $s_i = s[i] \cdot s_{i+1}$ und das Sortieren der s_i kommt dem Sortieren der Menge $(s[i], s_{i+1})$ gleich. Da gilt $i + 1 \in I_1$, ist die Ordnung der s_{i+1} bereits mit SA^{12} gegeben. Wir können also SA^{12} von links nach rechts nach Werten j kleiner n_1 durchsuchen, da sie die Suffixe s_{3j+1} repräsentieren, und in der Reihenfolge die Werte $3j$ an SA_0 anfügen. Anschließend wird SA_0 mit Radixpass nach den Zeichen $s[3j]$ stabil sortiert und wir erhalten in SA_0 die Sortierung der Suffixe s_i, $i \in I_0$.

Schritt 3: Verschmelzen beider Mengen

Aus dem im ersten Schritt erhaltenen Suffix-Array SA^{12} berechnen wir das inverse Suffix-Array und erhalten Namen s'_j für die Suffixe s_j, $j \in I_1 \cup I_2$. Der dritte Schritt besteht nun aus dem Verschmelzen der Mengen SA_0 und SA_{12}. Der dafür notwendige Vergleich der Suffixe s_i, $i \in I_0$ und s_j, $j \in I_1 \cup I_2$ wird in zwei Fälle unterteilt:

Fall 1: $j \in I_1$
In diesem Fall gilt $i+1 \in I_1$ und $j+1 \in I_2$. Wir können daher an Stelle von s_i und s_j auch die Paare $(s[i], s'_{i+1})$ und $(s[j], s'_{j+1})$ vergleichen, da die Relation von s'_{i+1} und s'_{j+1} der Relation von s_{i+1} und s_{j+1} entspricht.

Fall 2: $j \in I_2$
In diesem Fall gilt $i+2 \in I_2$ und $j+2 \in I_1$ und wir können statt s_i und s_j auch die Paare $(s[i], s[i+1], s'_{i+2})$ und $(s[j], s[j+1], s'_{j+2})$ vergleichen.

So wird sukzessive das jeweils kleinste Element der beiden Mengen an SA angefügt, wobei das Dummy-Triplet übersprungen wird.

Die Elemente der Mengen S_0 und $S_1 \cup S_2$ können also in jedem Fall in Relation gebracht werden, in dem auf schon bekannte Relationen von Suffixen aus $S_1 \cup S_2$ zurückgegriffen wird. Entsprechend der Indexverschiebung beider Suffixe nennen wir die Reduktion in Fall 1 *Shift um 1* und die Reduktion in Fall 2 *Shift um 2*.

6.2 Modifikationen

Der Skew-Algorithmus, wie er im vorherigen Abschnitt vorgestellt wurde, macht einige Einschränkungen an den String s. So wird vorausgesetzt, dass dem String s mindestens 3 Nullen folgen und diese nicht im Alphabet Σ enthalten sind. Diese Bedingungen an s erschweren eine generische Implementierung des Algorithmus. Wir werden sehen, dass es nicht schwierig ist, die beschränkenden Bedingungen zu entschärfen.

Das obligatorische Nulltriplet am Ende von s

Um auf die an s angehängten Nullen verzichten zu können, müssen wir zunächst Radix Sort dahingehend modifizieren, dass Digits, die hinter dem Stringende liegen, in einem extra Bucket gezählt werden und beim Zusammenfügen zuerst benutzt werden. Beim Nummerieren der Triplets können wir die Tatsache benutzen, dass sich die letzten 2 Triplets von s immer von allen anderen unterscheiden und auf Clipping beim Vergleichen verzichten. Beim Vereinigen der Suffixmengen im dritten Schritt des Skew-Algorithmus muss lediglich der Vergleich von $(s[i..i+1], s'_{i+2})$ mit $(s[j..j+1], s'_{j+2})$ entsprechend angepasst werden.

Das Dummy-Triplet

Das Dummy-Triplet wird im Fall $n \equiv 1 \bmod 3$ eingefügt, damit gesichert ist, dass das letzte Zeichen von s^1 nie mehrmals in s^{12} vorkommt. Gleichermaßen könnten statt dessen im Fall $n \equiv 1 \bmod 3$ beide Hälften von s^{12} vertauscht werden:

$$s^{12} = s^2 \cdot s^1$$

Um gesonderte Fallunterscheidungen zu umgehen, werden wir die Suffixe s_i nicht mehr nach den Restklassen von i sondern den Restklassen von $(n - i)$ modulo 3 einteilen:

$$I_j := \{i | i \in [0..n) \ \wedge \ (n - i) \bmod 3 = j\}.$$

Daraus ergeben sich folgende Änderungen für den String s^{12}:

$$s^1 = \ldots t'_{n-7} t'_{n-4} t'_{n-1} \quad \text{und} \quad s^2 = \ldots t'_{n-8} t'_{n-5} t'_{n-2}$$

$$s^{12} = s^2 \cdot s^1.$$

Mit dieser Einteilung haben die letzten Triplets von Suffixen s_i, $i \in I_1$ immer zwei Nullen und die letzten Triplets von s_i, $i \in I_2$ immer eine Null am Ende. Das letzte Zeichen von s^1 ist daher eindeutig und wir können auf das Dummy-Triplet verzichten.

Der Ausschluss der Null aus Σ

Die Null hat im Skew-Algorithmus zwei Funktionen. Zum einen übernimmt sie die Funktion eines Trennzeichens in s^{12} und zum anderen sorgt sie dafür, das Triplets die über das Stringende hinausragen, definiert sind. Nach Anwendung der beiden eben genannten Modifikationen benötigen wir kein solches Zeichen mehr, können bei der Bestimmung der lexikographischen Namen mit Null anfangen zu zählen und Null als Element von Σ zulassen.

Abbildung 6.2 zeigt, wie sich die Modifikationen auf den Algorithmus auswirken. Änderungen gegenüber dem klassischen Skew-Algorithmus [13] ergeben sich in der Auswahl der Triplets und der Anordnung der Namen in s^{12}, sowie der Fallunterscheidung beim Verschmelzen. Die Dollarzeichen wurden nur zur Veranschaulichung an s angefügt und sind mit einem modifizierten Radixsort nicht erforderlich.

6.3 Difference Covers

Die Essenz des Skew-Algorithmus liegt im Vereinigen der beiden in sich sortierten Suffixmengen. Durch die Dreiteilung der Suffixe in Restklassen und die

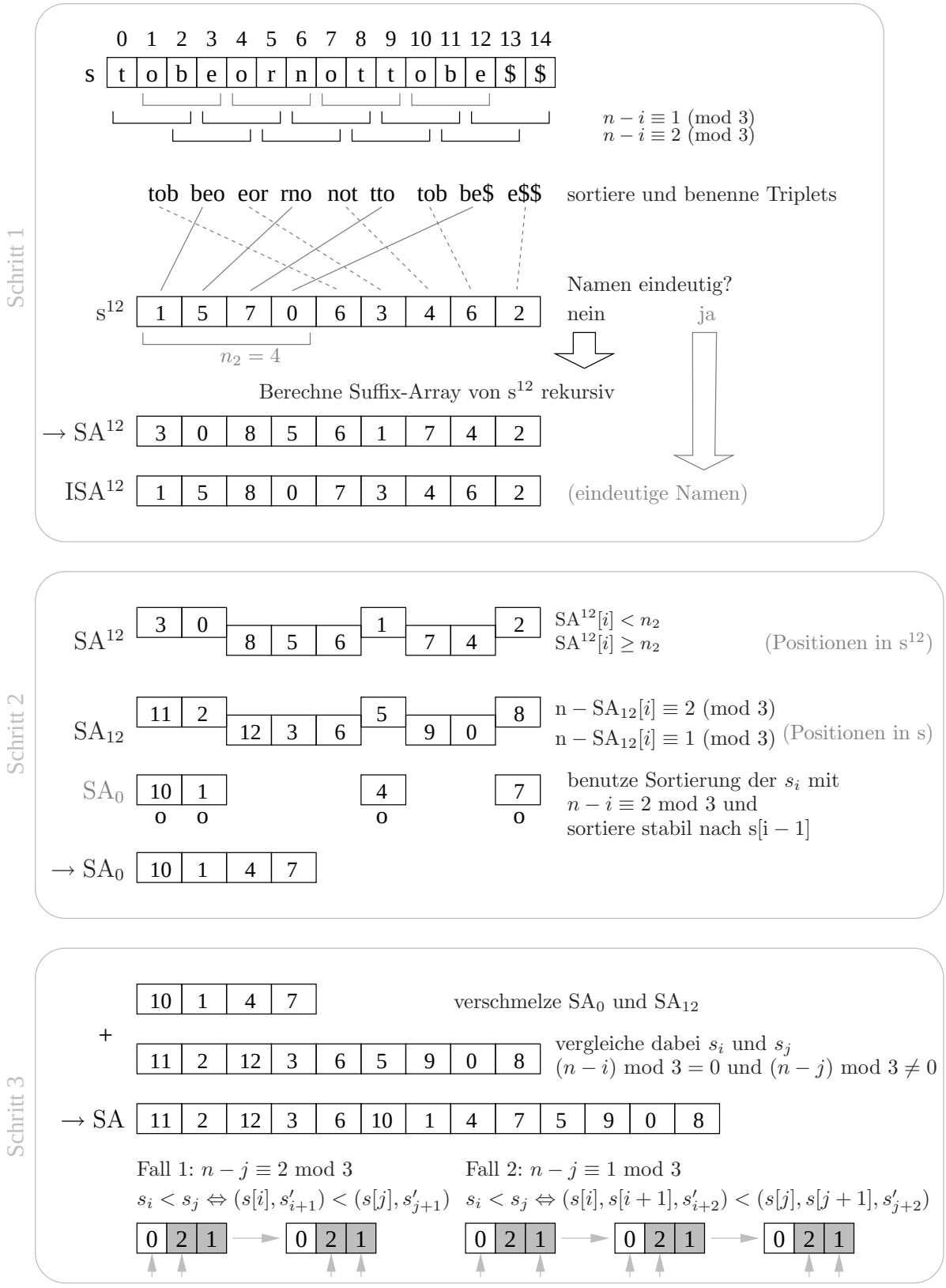


Abbildung 6.2: Modifizierter Skew-Algorithmus

Sortierung von s_j, $j \in I_1 \cup I_2$ ist es im dritten Schritt immer möglich, zwei Suffixe s_i, $i \in I_0$ und s_j, $j \in I_1 \cup I_2$ durch einen geeigneten Shift zu vergleichen. Statt einer Dreiteilung der Suffixe ist aber ebenso eine Einteilung der Suffixe in Restklassen modulo $v \in \mathbb{N}$ mit $v > 3$ vorstellbar, $D \subseteq [0, v)$ entspricht der Menge der Restklassen $I_j, j \in D$ die dann im ersten Schritt sortiert würden. Wenn nun zu jedem $d \in [0, v)$ ein $i \in D$ und ein $j \in D$ existieren mit $i - j \equiv d \bmod v$, könnte man im dritten Schritt je zwei Suffixe durch einen geeigneten Shift miteinander vergleichen. In diesem Fall nennen wir D ein *Difference Cover*. Für den Skew-Algorithmus sind dann $v = 3$ und $D = \{1, 2\}$.

Abbildung 6.3: $d = 2 \Rightarrow i = 4$ und $j = 2$ (Shift um 3)

Definition 6.3.1
Sei G eine endliche abelsche Gruppe und D Teilmenge von G. Man nennt D ein **Difference Cover**, *wenn gilt:*

$$\forall_{d \in G} \exists_{i,j \in D}\ i - j = d.$$

In unserem Fall ist $G = \mathbb{Z}_v$. Großen Einfluss auf Geschwindigkeit und Speicherbedarf des Algorithmus hat der Faktor

$$\lambda = \frac{|D|}{v},$$

da sich die Datenmenge beim rekursiven Abstieg in Schritt 1 mit dem Faktor λ verkleinert. Interessant sind also Difference Covers D, mit kleinem λ.
Tabelle 6.1 zeigt zu gegebenen Mächtigkeiten $|D|$ größtmögliche Gruppen $G = \mathbb{Z}_v$, so dass D noch Difference Cover von G ist. Die Gruppenmächtigkeit von G ist beschränkt durch:

$$|G| \leq 2 \cdot \binom{|D|}{2} + 1 = |D|^2 - |D| + 1.$$

Gilt $|G| = |D|^2 - |D| + 1$, so nennen wir D ein *perfektes Difference Cover* für G. Die in der Tabelle hervorgehobenen Mengen D sind perfekt. Unter allen Difference Covers fester Mächtigkeit $|D|$ wird λ minimal, wenn D perfekt ist. Für eine Modifikation des Algorithmus eignen sich bspw. $D = \{1, 2, 4\}$ mit $v = 7$ und $D = \{1, 2, 4, 10\}$ mit $v = 13$.

Umsetzung

Ist ein Difference Cover D modulo v gegeben, so definieren wir:

$$\begin{aligned} S_r &:= \{s_i | i \in [0..n) \ \wedge\ n - i \equiv r \bmod v\} \\ S_D &:= \bigcup_{r \in D} S_r. \end{aligned}$$

$\|D\|$	G	ein minimales Difference Cover	λ
2	$\mathbb{Z}_3$	$\{\mathbf{1}, \mathbf{2}\}$	0,6666...
3	$\mathbb{Z}_7$	$\{\mathbf{1}, \mathbf{2}, \mathbf{4}\}$	0,4285...
4	$\mathbb{Z}_{13}$	$\{\mathbf{1}, \mathbf{2}, \mathbf{4}, \mathbf{10}\}$	0,3076...
5	$\mathbb{Z}_{21}$	$\{\mathbf{1}, \mathbf{2}, \mathbf{7}, \mathbf{9}, \mathbf{19}\}$	0,2380...
6	$\mathbb{Z}_{31}$	$\{\mathbf{1}, \mathbf{2}, \mathbf{4}, \mathbf{9}, \mathbf{13}, \mathbf{19}\}$	0,1935...
7	$\mathbb{Z}_{39}$	$\{1, 2, 17, 21, 23, 28, 31\}$	0,1794...
8	$\mathbb{Z}_{57}$	$\{\mathbf{1}, \mathbf{2}, \mathbf{10}, \mathbf{12}, \mathbf{15}, \mathbf{36}, \mathbf{40}, \mathbf{52}\}$	0,1403...
9	$\mathbb{Z}_{73}$	$\{\mathbf{1}, \mathbf{2}, \mathbf{4}, \mathbf{8}, \mathbf{16}, \mathbf{32}, \mathbf{37}, \mathbf{55}, \mathbf{64}\}$	0,1232...
10	$\mathbb{Z}_{91}$	$\{\mathbf{1}, \mathbf{2}, \mathbf{8}, \mathbf{17}, \mathbf{28}, \mathbf{57}, \mathbf{61}, \mathbf{69}, \mathbf{71}, \mathbf{74}\}$	0,1098...
11	$\mathbb{Z}_{95}$	$\{1, 2, 6, 9, 19, 21, 30, 32, 46, 62, 68\}$	0,1157...
12	$\mathbb{Z}_{133}$	$\{\mathbf{1}, \mathbf{2}, \mathbf{33}, \mathbf{43}, \mathbf{45}, \mathbf{49}, \mathbf{52}, \mathbf{60}, \mathbf{73}, \mathbf{78}, \mathbf{98}, \mathbf{112}\}$	0,0902...

Tabelle 6.1: Minimale Difference Covers [12]

Zu beliebigen Suffixen s_i, s_j mit $s_i \in S_a$ und $s_j \in S_b$ ergibt sich die Länge des Shifts, der notwendig ist, um beide in Schritt 3 vergleichen zu können, folgendermaßen:

$$\delta_{D,v}(a,b) := \min_{t \in [0..v)} \{t | (a-t) \bmod v \in D \ \wedge \ (b-t) \bmod v \in D\}.$$

Da D ein Difference Cover ist, gibt es $a', b' \in D$ mit $a-b \equiv a'-b' \bmod v$ und ein $t \in [0..v), t \equiv a-a' \bmod v$. Folglich gilt $a-t \equiv a' \bmod v$ und $b-t \equiv b' \bmod v$. Das Minimum der Definition existiert also immer, außerdem gilt $s_{i+t} \in S_{a'} \subseteq S_D$ und $s_{j+t} \in S_{b'} \subseteq S_D$. Die Relation von s_i und s_j lässt sich also mit einem Shift um t, also durch einen Vergleich von $(s[i..i+t), s'_{i+t})$ und $(s[j..j+t), s'_{j+t})$ ermitteln. Zu beliebigen i, j lässt sich der Wert $\delta_{D,v}(a,b)$ in $\mathcal{O}(|D|)$ berechnen. Eine Vorberechnung der Tabelle aller Shiftwerte ermöglicht einen schnelleren Zugriff und lässt sich für optimale Difference Covers in $\mathcal{O}(v^2)$ realisieren (siehe Algorithmus 2). Tabelle 6.2 zeigt die Shiftwerte für $D = \{1, 2, 4\}$ und $v = 7$. Der maximal notwendige Shift der benötigt wird, um $s_i \in S_a$ mit einem anderen Suffix zu vergleichen ist:

$$\Delta_{D,v}(a) := \max_{b \in [0..v)} \delta_{D,v}(a,b).$$

Der mittlere Shiftwert einer Menge $M \subseteq D$ sei definiert als:

$$\bar{\Delta}_{D,v}(M) := \frac{1}{|M|} \cdot \sum_{a \in M} \Delta_{D,v}(a).$$

Der D entsprechend angepasste Skew-Algorithmus besteht aus folgenden Schritten:

a, b	0	1	2	3	4	5	6	$\Delta_{D,v}(a)$
0	0	6	5	6	3	3	5	6
1	6	0	0	6	0	4	4	6
2	5	0	0	1	0	1	5	5
3	6	6	1	0	2	1	2	6
4	3	0	0	2	0	3	2	3
5	3	4	1	1	3	0	4	4
6	5	4	5	2	2	4	0	5

Tabelle 6.2: Shiftwerte für $D = \{1, 2, 4\}$ und $v = 7$

Algorithmus 2 Berechnung der Shiftwert-Tabelle

Benötigt: Feld D ist optimales Difference Cover von $\mathbb{Z}_v$

```
for i := 0 to v − 1 do
  for j := 0 to v − 1 do
    δ_{D,v}[i, j] := 0
for i := 0 to |D| − 2 do
  for j := i + 1 to |D| − 1 do
    for k := 0 to v − 1 do
      δ_{D,v}[(D[i] + k) mod v, (D[j] + k) mod v] := k
      δ_{D,v}[(D[j] + k) mod v, (D[i] + k) mod v] := k
```

1. Sortieren der Suffixmenge S_D
2. Sortieren der verbleibenden Suffixmengen S_r mit $r \notin D$
3. Verschmelzen der $v - |D| + 1$ sortierten Suffixmengen

Schritt 1: Sortieren der Suffixmenge S_D

Schritt 1 lässt sich analog zum klassischen Skew-Algorithmus implementieren, wobei wir v-Tupel $\hat{s}_i := s[i..i + v)$ mit $s_i \in S_D$ mit Radixsort in $\mathcal{O}(|D| \cdot n)$ lexikographisch benennen. Gibt es keine zwei $s_i, s_j \in S_D$, $i \neq j$ und $\hat{s}_i = \hat{s}_j$, so ist die Benennung der v-Tupel auch eine Benennung der entsprechenden Suffixe in S_D. Andernfalls berechnen wir rekursiv das Suffix-Array SA^D des folgenden Strings über dem Alphabet $\left[0..\left\lceil \frac{|D|}{v} \cdot n \right\rceil\right)$ der Namen:

$$s^D := \prod_{r \in D} \left[\hat{s}'_i | s_i \in S_r\right] \qquad \left(\text{mit } \left[\hat{s}'_i | s_i \in S_r\right] = \ldots \hat{s}'_{n-r-2v} \hat{s}'_{n-r-v} \hat{s}'_{n-r}\right)$$

Für $D = \{1, 2, 4\}$ und $v = 7$ bspw. ist $s^D = s^{124}$ definiert als:

$$s^{124} := \left[\hat{s}'_i | s_i \in S_4\right] \cdot \left[\hat{s}'_i | s_i \in S_2\right] \cdot \left[\hat{s}'_i | s_i \in S_1\right].$$

Die Reihenfolge der Teilstrings $[\hat{s}'_i | s_i \in S_r]$, $r \in D$ in s^D ist willkürlich für $\{0\} \notin D$ und muss lediglich beim Umrechnen von Positionen i' aus SA^D in Positionen i der Suffixe s_i beachtet werden. Im Fall $\{0\} \in D$ muss $[\hat{s}'_i | s_i \in S_0]$ in s^D am Ende stehen. Die letzten Zeichen $\hat{s}'_{n-r}$ der Mengen $[\hat{s}'_i | s_i \in S_r]$, $r \in D \setminus \{0\}$ kommen dann jeweils nur einmal in s^D vor und Zeichenvergleiche brechen spätestens bei diesen Zeichen ab. Daraus folgt, dass eine Benennung der Suffixe von s^D auch eine Benennung der entsprechenden Suffixe in S_D ist. Das zu SA^D inverse Suffix-Array ISA^D lässt sich in $\mathcal{O}(\frac{|D|}{v} \cdot n)$ berechnen und ergibt die lexikographische Benennung von S_D.

Schritt 2: Sortieren der Suffixmengen aus S_r mit $r \notin D$

Die nun noch fehlenden Suffixmengen S_r mit $r \notin D$ lassen sich sukzessive ausgehend von der Sortierung der Menge S_{r-1} stabil mit Radixsort in insgesamt $\mathcal{O}(n)$ sortieren. Für $D = \{1, 2, 4\}$ und $v = 7$ ergeben sich bspw. folgende Abhängigkeiten:

$$\begin{array}{ccccccc} S_2 & \rightarrow & S_3 & & & & \\ S_4 & \rightarrow & S_5 & \rightarrow & S_6 & \rightarrow & S_0. \end{array}$$

Schritt 3: Verschmelzen der sortierten Suffixmengen

Im letzten Schritt sollen die $v - |D| + 1$ in sich sortierte Suffixmengen zu einer verschmolzen werden. Dazu führen wir ein Multiway Merge analog zu Mergesort (siehe 4.2) durch, der in jedem Schritt die Menge mit dem kleinsten Suffix ermittelt, diesen entfernt und dem Suffix-Array anfügt. Um das kleinste Element zu ermitteln, ist es notwendig, zwei beliebige Suffixe $s_i \in S_a$ und $s_j \in S_b$ miteinander vergleichen zu können. Wie oben gezeigt, ist dies durch einen Shift um $t := \delta_{D,v}(a, b)$ möglich und die Relation von s_i und s_j entspricht der Relation von $(s[i..i+t), s'_{i+t})$ und $(s[j..j+t), s'_{j+t})$, wobei die Namen s'_{i+t} und s'_{j+t} in Schritt 1 berechnet wurden. Zum effizienten Ermitteln des kleinsten Elements verwenden wir eine Priority Queue mit höchstens $v - |D| + 1$ Elementen. Die Elemente der Queue sind zu jeder Zeit die jeweils kleinsten noch übrigen Elemente der Suffixmengen. In jedem Schritt wird das kleinste Element der Queue entfernt und, falls möglich, das nächstgrößere seiner Suffixmenge eingefügt, so lange bis die Queue leer ist.

6.4 Speicherreduktion und Analyse

Wir bezeichnen im Folgenden die Skew-Algorithmen mit den perfekten Difference Covers $v = 3, D = \{1, 2\}$ als Skew3 und $v = 7, D = \{1, 2, 4\}$ als Skew7. Ist $T(n)$ die Laufzeit bzw. der Speicherbedarf des Skew-Algorithmus mit dem

Difference Cover D modulo v für die Stringlänge n, so gilt mit $\lambda = \frac{|D|}{v} < 1$:

$$T(n) = T\left(\lceil \lambda n \rceil\right) + \mathcal{O}(n) \quad \text{und} \quad T(n) = \mathcal{O}(1) \text{ für } n \leq v.$$

Daraus folgt $T(n) = \mathcal{O}(\frac{1}{1-\lambda} \cdot n) = \mathcal{O}(n)$. Der Algorithmus hat also eine lineare Laufzeit- und Speicherkomplexität.

Der ursprüngliche Skew-Algorithmus [13] reserviert in jedem Rekursionsschritt zu einem String der Länge n temporäre Felder mit einer Größe von $2(n+3)$ Einträgen. Insgesamt wird dann im schlechtesten Fall, also bei vollem rekursiven Abstieg, für einen String der Länge n temporärer Speicher von etwa

$$\sum_{i=0}^{\infty} \left(\frac{2}{3}\right)^i \cdot 2n = \frac{1}{1-\frac{2}{3}} \cdot 2n = 6n$$

Worten (entspricht $24n$ Byte) benötigt. Die temporären Felder sind s^{12}, SA^{12}, s^0 und SA^0. Da im dritten Schritt des Algorithmus die sortierten Felder SA^0 und SA^{12} zum Suffix-Array SA vereinigt werden, kann der hintere Teil des bereits für SA reservierten Speichers für SA^{12} verwendet werden, ohne dass beim Verschmelzen im dritten Schritt noch ungelesene Zeichen in SA^{12} überschrieben werden. s^0 wird nur im zweiten Schritt für Radix Sort benötigt und kann im vorderen Teil von SA untergebracht werden, da s^0 und SA^{12} zusammen genau n Einträge haben (kein \$\$\$-Triplet am Ende von s^{12}). So halbiert sich der Speicherbedarf auf $3n$ Worte. SA^0 wird erst nach der Rekursion benötigt, so dass durch einfaches Verlagern der Anforderung weiterer Speicher gespart werden kann. Diese Reduktion kann analog auch auf den Algorithmus mit Difference Covers übertragen werden.

Der Speicherbedarf dieser Optimierung soll nun noch genauer untersucht werden. Es bezeichne Σ_m das Eingabealphabet des Algorithmus im m-ten Rekursionsschritt. Zur Vereinfachung habe $\Sigma_1 := \Sigma$ die Form $[0..|\Sigma|)$ und es gelte $|\Sigma| \leq n$. Ist $S_{(m)}(n)$ der zusätzlich zu s und SA von der Implementierung benötigte Speicher in Rekursionsschritt m vor und nach dem rekursiven Abstieg gemessen in Zeichen des Suffix-Array, dann gilt:

$$\begin{aligned} S_{(m)}(n) &=^1 \overbrace{\sum_{i=1}^{m} \lambda^i n}^{s^D \text{ rekursiv}} + \overbrace{(1-\lambda) \cdot \lambda^{m-1} n}^{\bigcup_{j \notin D} \mathrm{SA}^j} + \overbrace{|\Sigma_m|}^{\text{Radixsort}} \\ &= \left(\frac{\lambda - \lambda^{m+1}}{1-\lambda} + (1-\lambda) \cdot \lambda^{m-1}\right) \cdot n + |\Sigma_m| \\ &= \left(\frac{\lambda}{1-\lambda} + \frac{1-2\lambda}{1-\lambda} \cdot \lambda^{m-1}\right) \cdot n + |\Sigma_m|. \end{aligned} \tag{6.1}$$

[1] In der Formel wurde auf Gaußklammern verzichtet. Die dadurch maximal entstehende Abweichung ist im Anhang in Korollar A.1.3 angegeben.

Für die Alphabetgröße $|\Sigma_m|$ in Rekursionsschritt $m \geq 2$ gilt folgende Invarianz:

$$|\Sigma_m| \leq n_m - 1 \leq \lambda^{m-1} n.$$

Bei einer maximalen Rekursionstiefe von r benötigt der Skew-Algorithmus insgesamt also höchstens:

$$\begin{aligned} S(n) &\leq \max_{m \in [1..r]} \left\{ \frac{\lambda}{1-\lambda} + \frac{1-2\lambda}{1-\lambda} \cdot \lambda^{m-1} + \lambda^{m-1} \right\} \cdot n \\ &= \max_{m \in [1..r]} \left\{ \frac{2-3\lambda}{1-\lambda} \cdot \lambda^{m-1} \right\} \cdot n + \frac{\lambda}{1-\lambda} \cdot n \\ &\leq \begin{cases} \frac{\lambda}{1-\lambda} \cdot n & \text{, falls } \lambda \geq \frac{2}{3} \\ 2n & \text{, sonst.} \end{cases} \end{aligned} \tag{6.2}$$

Die Abschätzung ist scharf und kann im bspw. mit $|\Sigma| = n$ und $s[i] = i$ erreicht werden. Nach unten lässt sich der Speicherbedarf durch n abschätzen ($S_{(1)}(n) > n$). Für alle perfekten Difference Covers $|D| \geq 2$ gilt:

$$\lambda = \frac{|D|}{v} = \frac{|D|}{|D|^2 - |D| + 1} \leq \frac{2}{3}.$$

Folgerung 6.4.1 *Skew3 und Skew7 benötigen bei einer Wortgröße von 4 Byte beide mindestens* $4n$ *und höchstens* $8n$ *Byte zusätzlichen Speicher während der Suffix-Array-Konstruktion.*

Für Skew7 kann unter bestimmten Einschränkungen für $|\Sigma|$ und n die obere Schranke für den zusätzlichen Speicher verschärft werden. Nach den Sätzen A.2.1 und A.2.2 im Anhang gilt:

$$\begin{aligned} |\Sigma| \leq \tfrac{2}{7} \cdot n &\Rightarrow S(n) \leq \tfrac{9}{7} \cdot n \quad \hat{=} 5\tfrac{1}{7} \cdot n \text{ Byte} \\ |\Sigma| \leq 5 \ \wedge\ 1,25 \cdot 10^6 \leq n &\Rightarrow S(n) \leq n + 5 \quad \hat{=} 4n + 20 \text{ Byte.} \end{aligned}$$

Eine Abschätzung der maximalen Rekursionstiefe in Abhängigkeit des maximalen lcp-Werts liefert der nachfolgende Satz.

Satz 6.4.2
Hat die Rekursion des Skew-Algorithmus zum Difference Cover v, D für einen String s mit dem maximalen lcp-Wert $\mathrm{lcp}_{\max}$ die Tiefe r, dann gilt:

$$\log_v \mathrm{lcp}_{\max} < r.$$

Beweis
In Rekursionsebene r verkörpern die Namen Substrings der Länge v^r. Wären die Namen in dieser Ebene nicht eindeutig, gäbe es zwei gleiche Substrings der Länge v^r und es wäre $\mathrm{lcp}_{\max} \geq v^r$. □

6.5 Externalisierung

Eine asymptotisch I/O-optimale externe Implementierung des Skew-Algorithmus wurde erstmals in [8] vorgestellt. Die Implementierung benutzt Pipelining und ihre Laufzeit wird dominiert durch die Laufzeit von 6 Sorter-Modulen pro Rekursionsebene. Tatsächlich handelt es sich aber in 4 von 6 Fällen um Permutationen, bei denen sich anhand eines Datensatzes dessen Zielposition berechnen lässt. Mapper haben eine geringere interne Laufzeitkomplexität als Sorter und lassen sich einfacher I/O-effizient implementieren, da bspw. auf Prediction-Techniken [9], wie sie für das externe Mergesort benötigt werden, verzichtet werden kann. In dieser Arbeit wird eine ebenfalls asymptotisch I/O-optimale externe Variante des Skew-Algorithmus aus 6.1 vorgestellt, bei der alle nicht notwendigen Sorter durch Mapper ersetzt wurden.

6.5.1 Pipelining

Externer Skew3

Algorithmus 3 und das zugehörige Aquädukt in Abbildung 6.4 stellen die externe Implementierung des Algorithmus aus Kapitel 6.1 vor. In der Implementierung werden die vom Algorithmus benutzten Felder derart umsortiert, dass ehemals wahlfreie zu sequentiellen Zugriffen werden. Die einzigen wahlfreien Zugriffe werden so in das Innere der Sorter bzw. Mapper verlagert. Die Schritte 1 bis 3 nehmen die lexikographische Benennung der Triplets vor, welche wenn nötig in den Schritten 4 bis 8 rekursiv zu einer Benennung der Suffixe $S_1 \cup S_2$ erweitert wird. In Schritt 10 werden die Namen s'_i der Suffixe nach i sortiert, so dass s in Schritt 11 erneut sequentiell gelesen werden kann und alle für den Verschmelzungsschritt 14 notwendigen Daten wie $s[i], s[i+1], s'_i, s'_{i+1}, s'_{i+2}$ gesammelt werden können. Schritt 12 berechnet das inverse Suffix-Array von SA^{12} bzw. ordnet die Suffixe in $S_1 \cup S_2$ lexikographisch. Die Erweiterung der Sortierung von S_2 zu einer Sortierung von S_0 erfolgt in Schritt 13. Schritt 14 kann nun die in sich sortierten Mengen $S_1 \cup S_2$ und S_0 sequentiell vereinigen, da die Tupel alle Informationen für einen geeigneten Shift enthalten.

Externer Skew7

Das Aquädukt einer Externalisierung des Skew-Algorithmus mit dem Difference Cover $D = \{1, 2, 4\}$ und $v = 7$ ist in Abbildung 6.5 angegeben. Der externe Algorithmus ergibt sich analog zu Algorithmus 3. Unterschiede gibt es lediglich in der Länge der Substrings (v-Tupel), die in Schritt 1 bis 3 benannt werden, dem Rekursionsfaktor $\lambda = \frac{|D|}{v}$ und den Knoten 11 bis 17 mit denen die in sich sortierten Suffixmengen verschmolzen werden. Die Datensätze der Pools 12-16 müssen alle erforderlichen Eingabezeichen und Namen für einen geeigneten Shift

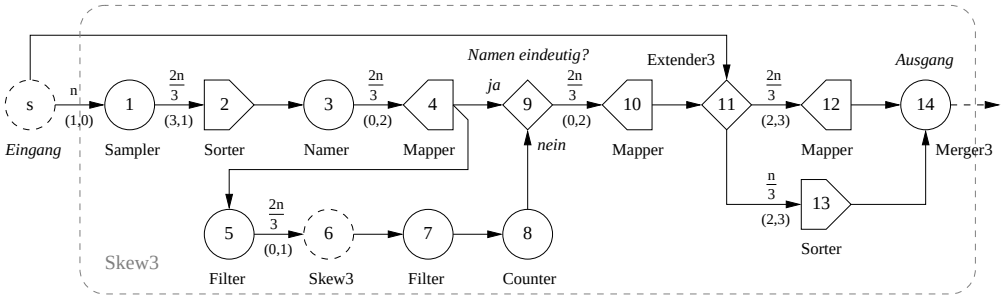


Abbildung 6.4: Aquädukt des externen Skew-Algorithmus (Skew3)

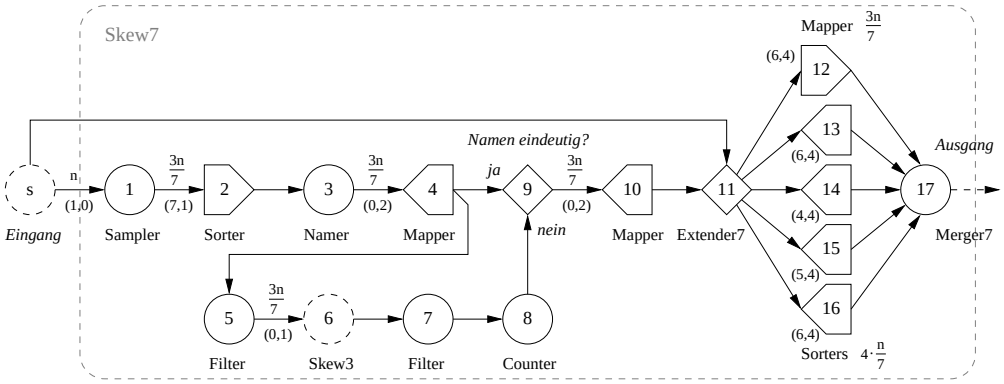


Abbildung 6.5: Aquädukt des externen Skew-Algorithmus mit $D = \{1, 2, 4\}$ und $v = 7$ (Skew7)

enthalten. Der Datensatztyp eines Pools, der die Menge S_a enthält, besteht also aus $\Delta(a)$ Eingabezeichen, $|D|$ Namen und den Index i für $s_i \in S_a$.

Für kleine Alphabete kann die I/O-Komplexität des externen Skew-Algorithmus reduziert werden, indem die Zeichentupel, die für die Benennung in Schritt 1 und für das Verschmelzen in Schritt 3 benötigt werden, komprimiert werden. Benutzt man eine einfache Bitkompression werden nur $\lceil \log_2 |\Sigma| \rceil$ Bit pro Zeichen benötigt. Auf den externen Skew7-Algorithmus angewendet können die v-Tupel und $\Delta(a)$-Tupel so im Fall $|\Sigma| \leq 16$ in 16 und 32 Bit langen Worten und im Fall $|\Sigma| \leq 4$ in 8 und 16 Bit langen Worten gespeichert werden.

In dieser Arbeit wurde diese Bitkompression in die externe Skew7-Implementierung integriert und wird automatisch für Strings wie DNA-Sequenzen mit Alphabeten $|\Sigma| < 16$ verwendet. Für die effiziente Umsetzung wurden Template-Metaprogramming-Techniken (Anhang B) angewandt und die Module Sampler und Extender7 entsprechend modifiziert. Zur Kompilationszeit werden von 8, 16 und 32 Bit großen Datentypen die am besten zur Alphabet- und Tupelgröße der einzelnen Pools passenden ausgewählt.

6.5.2 Analyse

Der Analyse des Skew-Algorithmus liegt der Datentyp 'Wort' (Kapitel 3.5) zu Grunde und die Modellgrößen B, M und N seien in Worten angegeben. Außer im ersten Rekursionsschritt arbeiten die Skew-Algorithmen ausschließlich mit Worten. Der erste Rekursionsschritt wird getrennt betrachtet, da der Datentyp zum Speichern von Eingabezeichen in der Regel kleiner ist als ein Wort. Er habe die relative Größe $\sigma \leq 1$ eines Textzeichens gemessen in Worten. Ist bspw. ein Textzeichen 1 Byte und ein Wort 4 Byte groß, sei $\sigma = \frac{1}{4}$. Für alle anderen Rekursionsebenen $m > 1$ definieren wir $\sigma_m = 1$ und $\sigma_1 = \sigma$.

Da zwischen den Modulen des externen Skew-Algorithmus Tupel ausgetauscht werden, die ausschließlich Eingabezeichen oder Positionen enthalten, sind die Größen der Moduldatentypen Linearkombinationen von σ_m und 1 mit den ganzzahligen Koeffizienten x und y. Die konkreten Koeffizienten sind als Paare (x, y) für die Pool-Module in Abbildung 6.4 und 6.5 unterhalb der einführenden Pfeile angegeben. Die I/O-Komplexitäten der externen Skew-Algorithmen können nun mit den Regeln aus Abschnitt 5.5 berechnet werden. Die recht umfangreichen Rechnungen für perfekte Difference Covers sind in Anhang A ausgeführt. Im Folgenden sollen nur die Ergebnisse vorgestellt werden:

Es seien $T_3(n)$ bzw. $T_7(n)$ die Zeitkomplexitäten des externen Skew3- und Skew7-Algorithmus. Es gilt:

$$\begin{aligned} T_3(n) &\leq \text{sort}\left(\left(2\tfrac{2}{3} \cdot \sigma + 10\tfrac{1}{3}\right) \cdot n\right) + \text{permute}\left(\left(1\tfrac{1}{3} \cdot \sigma + 16\tfrac{2}{3}\right) \cdot n\right) + \text{scan}\left((2\sigma + 4) \cdot n\right) \\ T_7(n) &\leq \text{sort}\left(\left(6\sigma + 9\tfrac{1}{4}\right) \cdot n\right) + \text{permute}\left(\left(2\tfrac{4}{7} \cdot \sigma + 7\tfrac{13}{14}\right) \cdot n\right) + \text{scan}\left(\left(2\sigma + 1\tfrac{1}{2}\right) \cdot n\right). \end{aligned}$$

Ist $T_3(n)$ bzw. $T_7(n)$ der Sekundärspeicherbedarf des externen Skew3- und Skew7-Algorithmus, dann gilt:

$$\begin{aligned} \left(2\sigma + 4\tfrac{1}{3}\right) \cdot n &\leq S_3(n) \leq \max\left\{2\sigma + 4\tfrac{1}{3}\,,\, 5\tfrac{5}{9}\right\} \cdot n \\ \left(5\tfrac{4}{7} \cdot \sigma + 4\tfrac{6}{7}\right) \cdot n &\leq S_7(n) \leq \max\left\{5\tfrac{4}{7} \cdot \sigma + 4\tfrac{6}{7}\,,\, 5\tfrac{16}{49}\right\} \cdot n. \end{aligned}$$

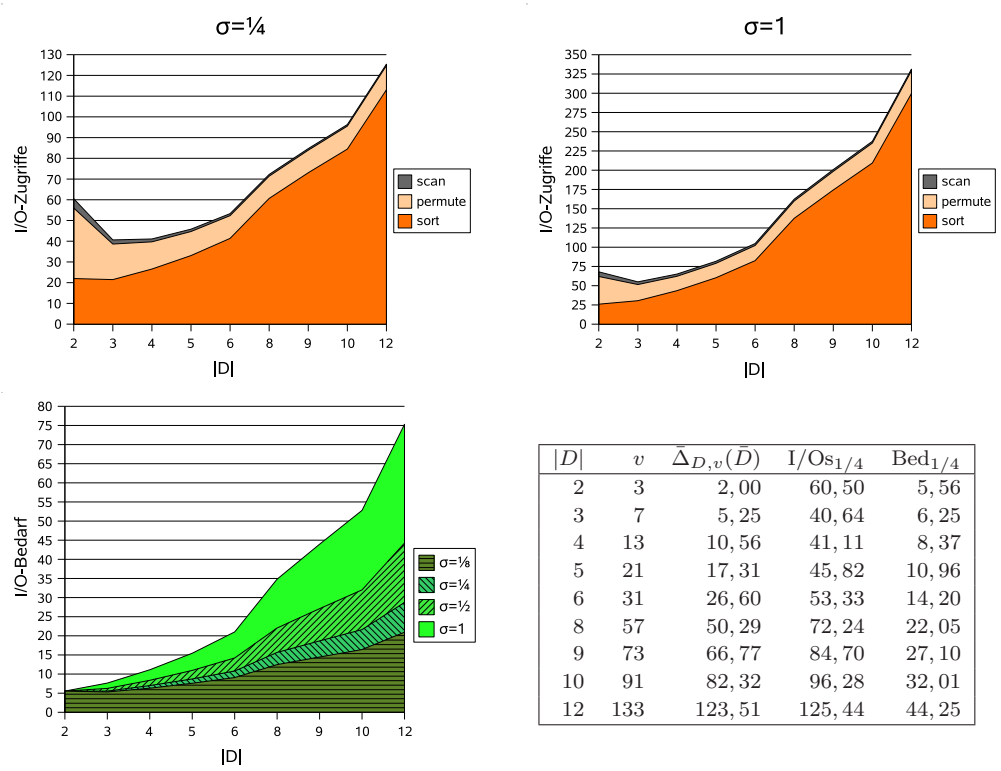
Skew3, Skew7 und allgemein alle Implementierungen mit perfekten Difference Covers sind asymptotisch I/O-optimal, wie folgender Satz zeigt.

Satz 6.5.1
Die externen Skew-Implementierungen sind asymptotisch optimal in der Zahl der I/O-Zugriffe.

Beweis
Nach den Abschätzungen aus Anhang A gilt für die Zeitkomplexität des externen Skew-Algorithmus mit perfektem Difference Cover D.

$$T^D(n) = \mathcal{O}(\text{sort}(n)).$$

Da sich das Sortierproblem in Linearzeit auf das Suffix-Array-Problem reduzieren lässt, gilt die Behauptung. □



$\|D\|$	v	$\bar{\Delta}_{D,v}(\bar{D})$	$\mathrm{I/Os}_{1/4}$	$\mathrm{Bed}_{1/4}$
2	3	2,00	60,50	5,56
3	7	5,25	40,64	6,25
4	13	10,56	41,11	8,37
5	21	17,31	45,82	10,96
6	31	26,60	53,33	14,20
8	57	50,29	72,24	22,05
9	73	66,77	84,70	27,10
10	91	82,32	96,28	32,01
12	133	123,51	125,44	44,25

Abbildung 6.6: I/O-Zugriffe (gestapelt) und I/O-Bedarf (gestaffelt)

Wie die I/O-Zugriffe und der zusätzliche Sekundärspeicherbedarf im worst-case konkret von der Größe des Difference Covers abhängen zeigt Abbildung 6.5.2 in Vielfachen von $\mathrm{scan}(n)$ Zugriffen[2] bzw. in Vielfachen der Größe eines n-elementigen Suffix-Arrays. Es wurden dabei verschiedene relative Zeichengrößen σ angenommen (bspw. $\sigma = \frac{1}{4}$ für $2^{16} < n \leq 2^{32}$ und $|\Sigma| = 256$). Gut zu erkennen ist, dass Skew7 offenbar unter allen Difference-Cover-Algorithmen optimal ist in der Zahl der I/O-Zugriffe und Skew3 und Skew7 den geringsten Sekundärspeicherbedarf haben.

[2]Für Pools gilt: $\mathrm{sort}(n) = \mathrm{permute}(n) = 2 \cdot \mathrm{scan}(n)$.

Algorithmus 3 Skew3Ext(s)

1: $S := [\langle n-i, s[i..i+3)\rangle : i \in [0..n) \wedge (n-i) \bmod 3 \neq 0]$
2: Sortiere S nach der 2. Komponente
3: Ersetze die 2. Komponente durch lex. Namen (es entsteht P)
if Namen sind nicht eindeutig **then**
4: Bewege $\langle j, r\rangle \in P$ nach $|P| - \frac{j+2}{3}$ für $j \bmod 3 = 1$ bzw. nach $\lfloor \frac{|P|}{2} \rfloor - \frac{j+1}{3}$ für $j \bmod 3 = 2$
5: Entferne die erste Komponente von P (es entsteht s^{12})
6: Berechne rekursiv das Suffix-Array $\mathrm{SA}^{12} := \mathrm{Skew3Ext}(s^{12})$
7: Konvertiere $x \in \mathrm{SA}^{12}$ zu $j := 3(|P| - x) - 2$ für $x \geq \lfloor \frac{|P|}{2} \rfloor$ und sonst zu $j := 3(\lfloor \frac{|P|}{2} \rfloor - x) - 1$
8: Nummeriere zu $P := [\langle j, r\rangle : r \in [0..|P|)]$
9: **end if**
10: Bewege $\langle j, r\rangle \in P$ an Position $|P| - \lceil \frac{2j}{3} \rceil$ und es entsteht $S' := [\langle n-i, s'_i\rangle : i \in [0..n) \wedge (n-i) \bmod 3 \neq 0]$
11: $S_0 := \left\{\left\langle n-i, s[i], s[i+1], s'_{i+1}, s'_{i+2}\right\rangle : i \in [0..n) \wedge (n-i) \bmod 3 = 0\right\}$ und
$S_{12} := \left\{\left\langle n-i, s[i], s[i+1], s'_i, s'_{i+2}\right\rangle : i \in [0..n) \wedge (n-i) \bmod 3 = 1\right\} \cup \left\{\left\langle n-i, s[i], [i+1], s'_i, s'_{i+1}\right\rangle : i \in [0..n) \wedge (n-i) \bmod 3 = 2\right\}$
12: Bewege $\left\langle n-i, s[i], s[i+1], s'_i, s'_{i+2}\right\rangle \in S_{12}$ an Position s'_i
13: Sortiere S_0 nach den Komponenten 2 und 4
14: Verschmelze S_0 und S_{12} mit $\langle i, t, t', c, c'\rangle \in S_0 < \langle j, u, u', d, d'\rangle \in S_{12} \Leftrightarrow (j \bmod 3 = 1 \wedge \langle t, t', c'\rangle < \langle u, u', d'\rangle) \vee (j \bmod 3 = 2 \wedge \langle t, c\rangle < \langle u, d'\rangle)$

Kapitel 7

Die LCP-Tabelle

Die folgenden Abschnitte stellen zwei Linearzeitalgorithmen und einen externen Algorithmus zur Konstruktion der LCP-Tabelle vor. Die Algorithmen des zweiten und dritten Abschnitts basieren beide auf der Idee des ersten Algorithmus, welcher 2001 von Kasai et al. [15] veröffentlicht wurde.

7.1 Konstruktion der LCP-Tabelle

Zum Erzeugen der LCP-Tabelle wird der Algorithmus aus [15] verwendet, da dieser unabhängig von der Erzeugung des Suffix-Array ist und die optimale Laufzeit von $\mathcal{O}(n)$ hat. Algorithmus 4 zeigt eine Umsetzung in strukturiertem Pseudocode.

Algorithmus 4 CreateLCPTable(s, SA, LCP)

```
1:  for i := 0 to n − 1 do
2:      ISA[SA[i]] := i
3:  h := 0
4:  for j := 0 to n − 1 do
5:      if ISA[j] > 0 then
6:          i := SA[ISA[j] − 1]
7:          while s[i + h] = s[j + h] do
8:              h := h + 1
9:          LCP[ISA[j]] := h
10:         if h > 0 then
11:             h := h − 1
```

Der Algorithmus von Kasai et al. macht sich die Eigenschaft zu Nutze, dass zwei Strings, die im ersten Zeichen übereinstimmen, nach Abschneiden dieses Zeichens noch immer die gleiche Relation zueinander haben. Für zwei Suffixe

s_i und s_j bedeutet dies:

$$\begin{array}{rrcl} & s_i < s_j \wedge s[i] = s[j] & \rightarrow & s_{i+1} < s_{j+1} \\ (\Leftrightarrow) & \mathrm{ISA}[i] < \mathrm{ISA}[j] \wedge s[i] = s[j] & \rightarrow & \mathrm{ISA}[i+1] < \mathrm{ISA}[j+1]. \end{array}$$

Ebenso verringert sich der lcp-Wert der beiden Suffix-Paare nach Abschneiden des ersten Zeichens um genau 1:

$$s[i] = s[j] \quad \rightarrow \quad \mathrm{lcp}(s_{i+1}, s_{j+1}) = \mathrm{lcp}(s_i, s_j) - 1.$$

Der lcp-Wert von s_{i+1} und s_{j+1} ist höchstens so groß wie der von s_{j+1} und seinem direkten lexikographischen Vorgänger $s_{\mathrm{SA[ISA}[j+1]-1]}$ im Suffix-Array:

$$s_{i+1} < s_{j+1} \quad \rightarrow \quad \mathrm{lcp}(s_{i+1}, s_{j+1}) \leq \mathrm{lcp}(s_{\mathrm{SA[ISA}[j+1]-1]}, s_{j+1}).$$

Setzt man nun $i := \mathrm{SA[ISA}[j] - 1]$, so dass s_i im Suffix-Array der direkte Vorgänger von s_j ist, erhält man unter Ausnutzung der drei oben genannten Eigenschaften:

$$\begin{array}{rrcl} & \mathrm{lcp}(s_{\mathrm{SA[ISA}[j]-1]}, s_j) - 1 & \leq & \mathrm{lcp}(s_{\mathrm{SA[ISA}[j+1]-1]}, s_{j+1}). \\ (\Leftrightarrow) & \mathrm{LCP[ISA}[j]] - 1 & \leq & \mathrm{LCP[ISA}[j+1]], \end{array}$$

unter der Bedingung, dass $\mathrm{ISA}[j] > 0$ gilt. Daraus folgt für die Berechnung von $\mathrm{LCP[ISA}[j+1]]$, dass beim Vergleich von s_{j+1} mit seinem Vorgänger die ersten $h := \mathrm{LCP[ISA}[j]] - 1$ Zeichen übersprungen werden können, wenn $h > 0$ ist. Der Algorithmus durchläuft den Text s sequentiell, die amortisierte Laufzeit ist deshalb linear.

7.2 Verbesserung

Zusätzlich zum Speicher für s, SA und LCP benötigt der Algorithmus in 7.1 im Allgemeinen $4n$ Byte für das temporäre inverse Suffix-Array. Dieser Speicher kann durch eine Modifikation eingespart werden. Diese wird in Algorithmus 5 vorgestellt.

Im Unterschied zum LCP-Algorithmus in [15] wird der Speicher der LCP-Tabelle hierbei zugleich für das inverse Suffix-Array verwendet. Das ist möglich, da die Einträge des inversen Suffix-Array sequentiell gelesen werden und der Speicher des j-ten Elements nach dessen letztmaligen Auslesens in Zeile 6 zum Beschreiben zur Verfügung steht. Nach Beendigung der Hauptschleife in Zeile 4 sind in LCP alle lcp-Werte enthalten, nur anders indiziert, diese Tabelle sei nun mit LCP_{pre} bezeichnet. LCP_{final} bezeichne die tatsächliche zu s und SA gehörende LCP-Tabelle. Es besteht nun folgender Zusammenhang:

$$\begin{array}{rrll} & \forall_{i \in [0,n)} & \mathrm{LCP}_{final}[\mathrm{ISA}[i]] & = \mathrm{LCP}_{pre}[i] \\ (\Leftrightarrow) & \forall_{j \in [0,n)} & \mathrm{LCP}_{final}[j] & = \mathrm{LCP}_{pre}[\mathrm{SA}[j]] \end{array}$$

Algorithmus 5 CreateLCPTableInPlace(s, SA, LCP)

```
1:  for i := 0 to n − 1 do
2:    LCP[SA[i]] := i
3:  h := 0
4:  for j := 0 to n − 1 do
5:    if LCP[j] > 0 then
6:      i := SA[LCP[j] − 1]
7:      while s[i + h] = s[j + h] do
8:        h := h + 1
9:      LCP[j] := −(h + 1)
10:     if h > 0 then
11:       h := h − 1
12: LCP[SA[0]] = −1
13: for i := 0 to n − 1 do
14:   if LCP[i] < 0 then
15:     j := i
16:     tmp := LCP[j]
17:     while SA[j] ≠ i do
18:       LCP[j] := −(LCP[SA[j]] + 1)
19:       j := SA[j]
20:     LCP[j] := −(tmp + 1)
```

Die Umsortierung, die LCP_{pre} in LCP_{final} überführt, findet in der Schleife in Zeile 13 statt. Es werden sukzessive Elemente entlang des Pfades i, SA[i], SA[SA[i]], SA[SA[SA[i]]], ... getauscht, bis ein Kreis geschlossen wird. Das Vorzeichen der Einträge in LCP wird verwendet, um sicherzustellen, dass jeder Kreis nur einmal iteriert wird.

Die Korrektheit folgt aus der Korrektheit des LCP-Algorithmus in [15] und dem Zusammenhang zwischen LCP_{pre} und LCP_{final}. Alle auftretenden Kreise sind disjunkt, jeder Index aus [0,n) ist also höchstens in einem Kreis enthalten und die while-Schleife in Zeile 17 wird höchstens n-mal durchlaufen. Die zum LCP-Algorithmus [15] zusätzliche Laufzeit ist also $\mathcal{O}(n)$ und die Gesamtlaufzeit daher ebenfalls $\mathcal{O}(n)$.

7.3 Externalisierung

Der in 7.1 vorgestellte Algorithmus durchläuft zur Berechnung der LCP-Tabelle den String s und das inverse Suffix-Array sequentiell über den Zeiger j bzw. $j + h$, ist aber nur schwach räumlich lokal im Zugriff auf LCP, SA und beim zeichenweisen Vergleich auf s mit dem Zeiger $i + h$. Für die Benutzung von Sekundärspeicher ist eine naive Implementierung des Algorithmus daher ungeeignet.

Im Folgenden wird nun eine Variante des LCP-Algorithmus [15] vorgestellt, die einen Speicherpuffer und das Pipelining-Konzept nutzt, um die Zahl der I/O-Zugriffe zu reduzieren.

7.3.1 Überlappungen

Der Algorithmus von Kasai et al. vergleicht für aufsteigende j jeweils s_j mit dessen direktem Vorgänger $s_{\mathrm{SA[ISA}[j-1]]}$ im Suffix-Array. Durch Ausnutzen der Eigenschaften des Suffix-Array kann auf den String s beim zeichenweisen Vergleich von $s[i+h]$ und $s[j+h]$ über den Zeiger $j+h$ sequentiell zugegriffen werden. Im Allgemeinen sind die Werte $i := \mathrm{SA[ISA}[j-1]]$ für aufsteigende j allerdings ziemlich zufällig und nicht aufeinanderfolgend. Die Zugriffe auf s über den Zeiger $i+h$ sind daher schwach räumlich lokal.

In Algorithmus 7 wird nun eine Modifikation vorgestellt, die mit einem Speicherpuffer w arbeitet. w ist ein Substring von s auf den wahlfrei und schnell zugegriffen werden kann. Der Algorithmus von Kasai et al. wird hierbei nacheinander für disjunkte aufeinanderfolgende Ausschnitte $w_k = s[b_k, e_k)$ durchlaufen, wobei gilt:

$$[b_1, e_1) \dot\cup [b_2, e_2) \dot\cup \ldots \dot\cup [b_l, e_l) = [0, n).$$

Innerhalb eines Durchlaufs mit $w = s[b, e)$ werden die zeichenweisen Vergleiche von $s_i[h]$ und $s_j[h]$ nur dann durchgeführt, wenn $b \leq i + h < e$ gilt.

Gesonderte Betrachtung erfordert der Fall $l := \mathrm{lcp}(s_i, s_j)$ und $i < e \leq i + l$, da hierbei nur die ersten $e-i$ Zeichen $s[i], s[i], \ldots, s[e-1]$ verglichen werden können. Auf die Zeichen $s[e], s[e+1], \ldots$ kann erst im nächsten Durchlauf zugegriffen werden. Der lcp-Wert von s_i und s_j kann also auch erst in einem nachfolgenden Durchlauf $w_k = s[b_k, e_k)$ mit $i + l \in [b_k, e_k)$ bestimmt werden. Im Folgenden wird gezeigt, wie das ohne nochmaliges Lesen von $s[i, b_k)$ realisiert werden kann. Dazu wird das folgende Lemma benötigt.

Lemma 7.3.1 *Für* $i + 1 < n$, $\mathrm{ISA}[i] + 1 < n$ *und* $\mathrm{ISA}[i+1] + 1 < n$ *gilt:*

$$\mathrm{LCP[ISA}[i] + 1] - 1 \leq \mathrm{LCP[ISA}[i + 1] + 1].$$

Beweis
$\mathrm{ISA}[i]$ ist die Position von s_i im Suffix-Array SA. Der direkte Nachfolger von s_i ist daher s_j mit $j := \mathrm{SA[ISA}[i] + 1]$. Aus Eigenschaft 2 folgt:

$$\mathrm{lcp}(s_i, s_j) - 1 \leq \mathrm{lcp}(s_{i+1}, s_{j+1}).$$

O.B.d.A. sei $\mathrm{lcp}(s_i, s_j) \geq 1$. Im Fall $\mathrm{lcp}(s_i, s_j) = 0$ gilt die Behauptung ohnehin. Aus $s_i < s_j$ folgt $s_{i+1} < s_{j+1}$. $s_{\mathrm{SA[ISA}[i+1]+1]}$ ist der direkte Nachfolger von s_{i+1}. Wegen $s_i < s_{\mathrm{SA[ISA}[i+1]+1]} < s_j$ folgt nun analog zu Eigenschaft 3:

$$\mathrm{lcp}(s_{i+1}, s_{j+1}) \leq \mathrm{lcp}(s_{i+1}, s_{\mathrm{SA[ISA}[i+1]+1]}) = \mathrm{LCP[ISA}[i + 1] + 1].$$

Nach Definition ist $\text{LCP}[\text{ISA}[i]+1] = \text{lcp}(s_i, s_j)$. Damit gilt die Behauptung. □

Korollar 7.3.2 *Es sei* $\text{LCP}[\text{ISA}[i]+1] \geq h$. *Dann gilt für* $i' \in [i, i+h]$*:*

$$\text{LCP}[\text{ISA}[i']+1] \geq h - (i' - i).$$

Korollar 7.3.2 bedeutet, wenn für ein festes e s_i und sein Nachfolger in den ersten $e-i$ Zeichen $s[i], s[i+1], \ldots, s[e-1]$ übereinstimmen, stimmen für $i' \in [i, e)$ $s_{i'}$ und dessen Nachfolger in den ersten $e-i'$ Zeichen überein. Der Substring $s[i', e)$ muss zur Bestimmung des lcp-Werts demnach nicht nochmal gelesen werden.

Definition 7.3.3
Für $e \in [0, n)$ *wird* $\text{overlap}(e)$ *definiert als:*

$$\text{overlap}(e) := \min\{i | i \leq e \leq i + \text{LCP}[\text{ISA}[i]+1]\}.$$

Zu einer festen Position e sind s_i mit $i \in [\text{overlap}(e), e]$ genau die Strings, die zur Bestimmung des lcp-Werts mit ihren Nachfolgern erst ab dem Zeichen $s[e]$ verglichen werden müssen. Bei Strings s_i mit $i \notin [\text{overlap}(e), e]$ bricht die Zeichengleichheit zu ihren Nachfolgern entweder vor e ab oder beginnt erst ab $e+1$.

Algorithmus 7 benutzt diese Eigenschaft. Für jeden Durchlauf $w_k = s[b_k, e_k)$ wird $\text{overlap}(e_k)$ in Zeile 18 dynamisch ermittelt. Der im vorherigen Durchlauf ermittelte Wert $\text{overlap}(e_{l-1})$ wird in den Zeilen 10 und 12 benutzt, um Vergleiche, die an der Grenze e_{l-1} abbrechen mussten, im aktuellen Durchlauf l fortzusetzen.

Die Anzahl der Zeichen, die mit dem Aufruf in Zeile 4 gelesen werden bzw. die Länge von w, bestimmt die Zahl der Durchläufe des Algorithmus und ist dominierend für dessen Laufzeit. Wird der komplette String s in w gelesen, verhält sich der Algorithmus exakt wie in [15].

7.3.2 Pipelining

Lemma 7.3.4 *Wird ein Feld* F *von Tripeln mit* $F[i] = (i, \text{SA}[i], \text{SA}[i-1])$ *aufsteigend nach der zweiten Komponente sortiert, erhält man ein Feld* F' *mit* $F'[j] = (\text{ISA}[j], j, \text{SA}[\text{ISA}[j]-1])$.

Beweis
Substituiert man i durch $\text{ISA}[j]$, so sind die Tripel für $j = 0, \ldots, n-1$ aufsteigend nach der zweiten Komponente sortiert. □

Durch vorherige Konstruktion eines F' aus Lemma 7.3.4 kann auf die Werte $\text{SA}[\text{ISA}[j]-1]$ und $\text{ISA}[j]$ für aufsteigendes j, die in den Algorithmen benötigt

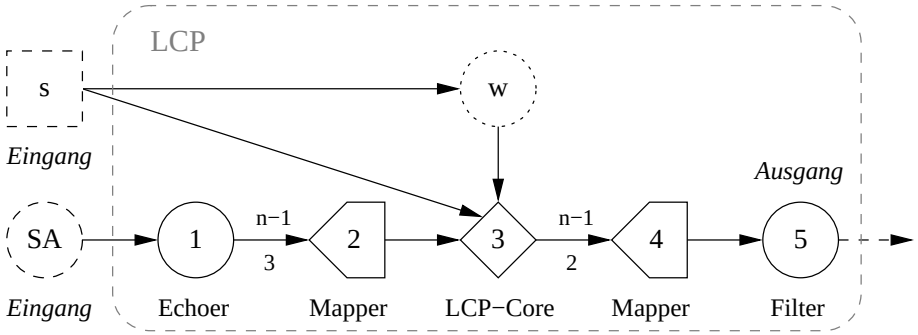


Abbildung 7.1: Aquädukt des externen LCP-Algorithmus

werden, sequentiell zugegriffen werden. Algorithmus 7 kann nun einfach als Pump (LCP-Core) implementiert werden, mit dem Eingabestrom F' und einem Mapper als Empfänger, der die Paare $\langle \mathrm{ISA}[j], \mathrm{LCP}[\mathrm{ISA}[j]] \rangle$ aus Zeile 16 aufsteigend nach der ersten Komponente sortiert. Aquädukt und Pseudo-Code dieses externen Algorithmus sind in Abbildung 7.1 und Algorithmus 6 angegeben. In den Schritten 1 und 2 wird F' erzeugt. Schritt 3 berechnet die Werte $\langle i, \mathrm{LCP}[i] \rangle$. s wird dabei einmal in Blöcken der Größe $|w|$ und parallel dazu mehrere Male zusammen mit F' zeichenweise sequentiell gelesen. Schritt 4 sortiert die in P ungeordneten Elemente $\langle i, \mathrm{LCP}[i] \rangle$ nach i und nach Entfernen der ersten Komponente entsteht die LCP-Tabelle $\mathrm{LCP}[1..n)$ ohne das erste Element $\mathrm{LCP}[0] = 0$.

Algorithmus 6 LCPExt(s)

1: $F := [\langle i, \mathrm{SA}[i], \mathrm{SA}[i-1] \rangle : i \in [1..n)]$
2: Bewege $\langle i, \mathrm{SA}[i], \mathrm{SA}[i-1] \rangle \in F$ an Position $\mathrm{SA}[i]$
3: Berechne mit modifiziertem Algorithmus 7 $P := \{\langle i, \mathrm{LCP}[i] \rangle : i \in [1..n)\}$
4: Bewege $\langle i, j \rangle \in P$ an Position $i - 1$
5: Entferne die erste Komponente von P

7.3.3 Analyse

Bestimmend für die Laufzeit des externen LCP-Algorithmus ist die Zahl der Schleifendurchläufe in Zeile 4 von Algorithmus 7. Ist $W := |w|$ die Größe des Speicherpuffers w, wird der Schleifenkörper $\lceil \frac{n}{W} \rceil$-mal wiederholt. Insgesamt wird s also $\lceil \frac{n}{W} + 1 \rceil$-mal und F' $\lceil \frac{n}{W} \rceil$-mal gelesen. Es seien σ und 1 die Größen der Datentypen für Zeichen von s bzw. für Indizes $i \in \mathrm{SA}$ und $T^W(n)$ die Zeitkomplexität des externen LCP-Algorithmus in Abhängigkeit der Puffergröße

W, dann gilt:

$$\begin{aligned} T^W(n) &\leq \text{scan}(n) + \text{permute}_{write,3}(n-1) + \\ &\quad \left\lceil \tfrac{n}{W} + 1 \right\rceil \cdot \text{scan}_\sigma(n) + \left\lceil \tfrac{n}{W} \right\rceil \cdot \text{permute}_{read,3}(n-1) + \text{permute}_2(n-1) \\ &\leq \text{scan}\left(\left\lceil \tfrac{n}{W} + 1 \right\rceil \cdot \sigma n + n\right) + \text{permute}_{read}\left(\left\lceil \tfrac{n}{W} - 1 \right\rceil \cdot 3n\right) + \text{permute}(5n) \\ &\leq^{1} \text{scan}\left(\left\lceil \tfrac{n}{W} - 1 \right\rceil \cdot (\sigma+3) \cdot n + (2\sigma+1) \cdot n\right) + \text{permute}(5n). \end{aligned}$$

Für den zusätzlichen Sekundärspeicherbedarf $S^W(n)$ der Implementierung gilt:

$$\begin{aligned} S^W(n) &= \max\{3,\, 3+2,\, 2\} \cdot (n-1) \\ &= 5(n-1). \end{aligned}$$

Algorithmus 7 CreateLCPTableExt(s, SA, LCP)

```
1:  for i := 0 to n − 1 do
2:    ISA[SA[i]] := i
3:  overlap_prev := 0
4:  while readnext(w) do                    // w = s[b, e)
5:    h := 0
6:    overlap := e
7:    for j := 0 to n − 1 do
8:      if ISA[j] > 0 then
9:        i := SA[ISA[j] − 1]
10:       if overlap_prev ≤ i and i + h ≤ e then
11:         if i + h < b then
12:           h := b − i
13:         while i + h ∈ [b, e) and w[i + h − b] = s[j + h] do
14:           h := h + 1
15:         if i + h < e or e = n then
16:           LCP[ISA[j]] := h
17:       if i + h ≥ e then
18:         overlap = min{overlap, i}
19:       if h > 0 then
20:         h := h − 1
21:   overlap_prev := overlap
```

[1] für den Algorithmus aus Kapitel 4.3 gilt: $\text{permute}_{read}(x) = \text{scan}(x)$

Kapitel 8

Die erweiterte LCP-Tabelle

In diesem Kapitel soll die Suche nach Teilstrings eines gegebenen Strings s der Länge n beschrieben werden. Ist zu s das entsprechende Suffix-Array gegeben, können zu einem Teilstring t der Länge m alle Vorkommen in s in $\mathcal{O}(m \log n)$ gefunden werden. Mit einer Erweiterung [21] der zugehörigen LCP-Tabelle lässt sich diese Laufzeit zu $\mathcal{O}(m + \log n)$ verbessern.

8.1 Binärsuche im Suffix-Array

Ein String t kommt genau dann an Position i in s vor, wenn t ein Präfix des Suffixes s_i ist. Die Menge aller Suffixe von s ist durch das Suffix-Array SA lexikographisch geordnet. Die Menge der Suffixe mit Präfix t ist in SA daher zusammenhängend, es gibt also $l, r \in [0..n)$ mit $i \in [l..r) \Leftrightarrow t$ ist Präfix von $s_{\mathrm{SA}[i]}$. Ziel ist nun, l und r zu beliebigen t effizient zu bestimmen.

Definition 8.1.1
Es seien $r \in \Sigma^*$ und $m \in \mathbb{N}$ beliebig. Dann sei

$$\mathrm{prefix}_m(r) := \begin{cases} r & \text{für } |r| \leq m, \\ r[0..m) & \text{für } |r| > m. \end{cases}$$

Wir definieren die Relationen $<_m$, $=_m$ und $>_m$ nun entspechend der ersten m Zeichen.

Definition 8.1.2
Es seien $a, b \in \Sigma^*$ und $m \in \mathbb{N}$ beliebig. Die Relationen $<_m$, $=_m$ und $>_m$ sind wie folgt definiert:

$$\begin{aligned} a <_m b &\Leftrightarrow \mathrm{prefix}_m(a) < \mathrm{prefix}_m(b), \\ a =_m b &\Leftrightarrow \mathrm{prefix}_m(a) = \mathrm{prefix}_m(b), \\ a >_m b &\Leftrightarrow \mathrm{prefix}_m(a) > \mathrm{prefix}_m(b). \end{aligned}$$

Die Relationen $\leq_m$ bzw. $\geq_m$ seien die Vereinigungen der Relationen $<_m$ bzw. $>_m$ und $=_m$. Sucht man im Suffix-Array nun mittels Binärsuche und entsprechend der Relationen $<_m$ bzw. $\leq_m$ nach t, erhält man die Indizes l bzw. r und $[l..r)$ ist das gesuchte Intervall (siehe Algorithmen 8).

Algorithmus 8 Finde l bzw. r zu gegebenen s, SA und t

```
1: l_1 := 0                          1: r_1 := 0
2: l_2 := |s|                        2: r_2 := |s|
3: while l_1 ≠ l_2 do                3: while r_1 ≠ r_2 do
4:    i := ⌊(l_1+l_2)/2⌋             4:    i := ⌊(r_1+r_2)/2⌋
5:    if s_SA[i] <_|t| t then        5:    if s_SA[i] ≤_|t| t then
6:       l_1 := i + 1                6:       r_1 := i + 1
7:    else                           7:    else
8:       l_2 := i                    8:       r_2 := i
9: return l_1                        9: return r_1
```

Für die in den Algorithmen 8 verwendeten Variablen $l_{1/2}$ und $r_{1/2}$ gelten die Invarianzen $s_{\mathrm{SA}[l_1-1]} <_{|t|} t \leq_{|t|} s_{\mathrm{SA}[l_2]}$ und $s_{\mathrm{SA}[r_1-1]} \leq_{|t|} t <_{|t|} s_{\mathrm{SA}[r_2]}$, wobei wir zur Vereinfachung $s_{\mathrm{SA}[-1]}$ und $s_{\mathrm{SA}[n]}$ als kleinstes bzw. größtes Wort ungleich aller Suffixe s_i definieren. Unter Einhaltung dieser Invarianzen werden l_1/r_1 bzw. l_2/r_2 schrittweise vergrößert bzw. verkleinert, bis $l_1 = l_2$ bzw. $r_1 = r_2$ gilt. Für die zurückgegebenen Positionen l und r gilt dann $s_{\mathrm{SA}[l-1]} <_m t \leq_m s_{\mathrm{SA}[l]}$ und $s_{\mathrm{SA}[r-1]} \leq_m t <_m s_{\mathrm{SA}[r]}$. Im Falle $l < r$ gilt $t =_m s_{\mathrm{SA}[l]} =_m s_{\mathrm{SA}[l+1]} =_m \ldots =_m s_{\mathrm{SA}[r-1]} =_m t$ und $[l..r)$ ist das gesuchte Intervall. Andernfalls gilt $l = r \Rightarrow s_{\mathrm{SA}[l-1]} <_m t <_m s_{\mathrm{SA}[l]}$ und t ist kein Substring von s. $[l..r) = \emptyset$ ist auch in diesem Fall das gesuchte Intervall.

Die Laufzeit der l,r-Suche (Algorithmus 8) ist $\mathcal{O}(m \log n)$, da das Suchintervall nach jedem Durchlauf der Schleife in Zeile 3 mindestens halbiert wird und der Vergleich in Zeile 5 eine worst-case Laufzeit von $\mathcal{O}(m)$ hat.

8.2 Verbesserte Binärsuche

Die in 8.1 vorgestellte Binärsuche lässt sich verbessern durch Optimierung des Vergleichs in Zeile 5. Offensichtlich können die ersten $\mathrm{lcp}(s_{\mathrm{SA}[i]}, t)$ Zeichen übersprungen werden, da sie gleich sind und erst das $(\mathrm{lcp}(s_{\mathrm{SA}[i]}, t) + 1)$-te Zeichen über die Relation von $s_{\mathrm{SA}[i]}$ und t entscheidet. Der Wert $\mathrm{lcp}(s_{\mathrm{SA}[i]}, t)$ lässt sich leider nicht ohne weiteres in $\mathcal{O}(1)$ bestimmen, in der l-Suche aber bspw. durch

$$\mathrm{lcp}(s_{\mathrm{SA}[i]}, t) \geq \min\left\{\mathrm{lcp}(s_{\mathrm{SA}[l_1-1]}, t), \mathrm{lcp}(s_{\mathrm{SA}[l_2]}, t)\right\}$$

für $0 < l_1$ und $l_2 < n$ abschätzen. Nach Abschluss des Vergleichs in Zeile 5 ist die Anzahl der erfolgreichen Zeichenvergleiche $k := \mathrm{lcp}(s_{\mathrm{SA}[i]}, t)$ bekannt

und je nach Relation gilt im nächsten Durchlauf $\text{lcp}(s_{\text{SA}[l_1-1]}, t) = k$ oder $\text{lcp}(s_{\text{SA}[l_2]}, t) = k$. Durch die Benutzung von Variablen k_1 bzw. k_2 für die Anzahl der Zeichenvergleiche, die mit $s_{\text{SA}[l_1-1]}$ bzw. $s_{\text{SA}[l_2]}$ erfolgreich durchgeführt wurden, und $\text{lcp}(s_{\text{SA}[i]}, t) \geq \min\{k_1, k_2\}$ kann der zeichenweise Vergleich also optimiert werden. In der Praxis verbessert sich die Laufzeit dadurch auf $\mathcal{O}(m + \log n)$, die worst-case Laufzeit ist aber immer noch $\mathcal{O}(m \log n)$.

Die schlechte worst-case Laufzeit beruht darauf, dass Zeichenvergleiche ab Position $\min\{k_1, k_2\}$ beginnen. Zeichen mit den Position $[\min\{k_1, k_2\}.. \max\{k_1, k_2\})$ werden dabei erneut verglichen. Würde man den Algorithmus so modifizieren, dass alle Zeichenvergleiche ab Position $\max\{k_1, k_2\}$ beginnen, hätte dieser eine worst-case Laufzeit von $\mathcal{O}(m + \log n)$.

Diese Modifikation ist möglich und soll im Folgenden für die l-Suche (r-Suche analog) vorgestellt werden. Voraussetzung ist, dass die lcp-Werte der auftretenden Paare $(s_{\text{SA}[l_1-1]}, s_{\text{SA}[i]})$ und $(s_{\text{SA}[l_2]}, s_{\text{SA}[i]})$ bekannt sind. Es seien k_1, k_2 die durch erfolgreich durchgeführte Zeichenvergleiche bekannten lcp-Werte von t mit $s_{\text{SA}[l_1-1]}$ bzw. $s_{\text{SA}[l_2]}$. Vor Beginn der Hauptschleife setzen wir $k_1 := k_2 := -1$, was bedeutet, dass die lcp-Werte der Paare $(s_{\text{SA}[l_1-1]}, t)$ und $(s_{\text{SA}[l_2]}, t)$ noch unbekannt sind.

In jedem Schleifendurchlauf unterscheiden wir drei Hauptfälle je nach Relation von k_1 und k_2:

Fall 1: $k_1 = k_2$

Im Fall $k_1 = k_2$ vergleichen wir $s_{\text{SA}[i]}$ und t ab dem $(\max\{0, k_1\}+1)$-ten Zeichen und passen je nach Relation entweder k_1 oder k_2 an.

Fall 2: $k_1 > k_2$

Es werden die Invarianzen $s_{\text{SA}[l_1-1]} \leq_m t$ und $\text{lcp}(s_{\text{SA}[l_1-1]}, t) = k_1 > -1$ benutzt und folgende drei Fälle unterschieden:

Fall 2.1: $\text{lcp}(s_{\text{SA}[l_1-1]}, s_{\text{SA}[i]}) > k_1$
Der gemeinsame Präfix von $s_{\text{SA}[l_1-1]}$ und $s_{\text{SA}[i]}$ ist also länger als der gemeinsame Präfix von $s_{\text{SA}[l_1-1]}$ und t. $s_{\text{SA}[i]}$ und t stimmen in genau den ersten $k := k_1$ Zeichen überein und es gilt $s_{\text{SA}[l_1-1]}[k] = s_{\text{SA}[i]}[k] < t[k]$, falls $k < m$. Demnach gilt $s_{\text{SA}[i]} \leq_m t$ und $s_{\text{SA}[i]} =_m t \Leftrightarrow k_1 = m$.

Fall 2.2: $\text{lcp}(s_{\text{SA}[l_1-1]}, s_{\text{SA}[i]}) < k_1$
Der gemeinsame Präfix von $s_{\text{SA}[l_1-1]}$ und $s_{\text{SA}[i]}$ ist kürzer als der gemeinsame Präfix von $s_{\text{SA}[l_1-1]}$ und t. $s_{\text{SA}[i]}$ und t stimmen in genau den ersten $k := \text{lcp}(s_{\text{SA}[l_1-1]}, s_{\text{SA}[i]})$ Zeichen überein und es gilt $s_{\text{SA}[l_1-1]}[k] = t[k] < s_{\text{SA}[i]}[k]$. Demnach gilt $s_{\text{SA}[i]} >_m t$ und wir setzen $k_2 := k$.

Fall 2.3: $\text{lcp}(s_{\text{SA}[l_1-1]}, s_{\text{SA}[i]}) = k_1$
$s_{\text{SA}[l_1-1]}$, $s_{\text{SA}[i]}$ und t stimmen in diesem Fall in den ersten k_1 Zeichen überein, so

dass erst ab dem (k_1+1)-ten Zeichen verglichen werden muss. Je nach Relation wird entweder k_1 oder k_2 entsprechend angepasst.

Fall 3: $k_1 < k_2$

Es werden die Invarianzen $s_{\mathrm{SA}[l_2]} \geq_m t$ und $\mathrm{lcp}(s_{\mathrm{SA}[l_2]}, t) = k_2 > -1$ benutzt und drei analoge Fälle unterschieden:

Fall 3.1: $\mathrm{lcp}(s_{\mathrm{SA}[l_2]}, s_{\mathrm{SA}[i]}) > k_2$
$s_{\mathrm{SA}[i]}$ und t stimmen in genau den ersten $k := k_2$ Zeichen überein und es gilt $s_{\mathrm{SA}[l_2]}[k] = s_{\mathrm{SA}[i]}[k] > t[k]$, falls $k < m$. Demnach gilt $s_{\mathrm{SA}[i]} \geq_m t$ und $s_{\mathrm{SA}[i]} =_m t \Leftrightarrow k_2 = m$.

Fall 3.2: $\mathrm{lcp}(s_{\mathrm{SA}[l_2]}, s_{\mathrm{SA}[i]}) < k_2$
$s_{\mathrm{SA}[i]}$ und t stimmen in genau den ersten $k := \mathrm{lcp}(s_{\mathrm{SA}[l_2]}, s_{\mathrm{SA}[i]})$ Zeichen überein und es gilt $s_{\mathrm{SA}[l_2]}[k] = t[k] > s_{\mathrm{SA}[i]}[k]$. Demnach gilt $s_{\mathrm{SA}[i]} <_m t$ und wir setzen $k_1 := k$.

Fall 3.3: $\mathrm{lcp}(s_{\mathrm{SA}[l_2]}, s_{\mathrm{SA}[i]}) = k_2$
$s_{\mathrm{SA}[l_2]}$, $s_{\mathrm{SA}[i]}$ und t stimmen in diesem Fall in den ersten k_2 Zeichen überein, so dass erst ab dem (k_2+1)-ten Zeichen verglichen werden muss. Je nach Relation wird entweder k_2 oder k_1 entsprechend angepasst.

In allen Fällen ist k der durch Zeichenvergleiche oder Fallunterscheidungen ermittelte Wert $k = \mathrm{lcp}(s_{\mathrm{SA}[i]}, t)$. Die Relation von $\mathrm{prefix}_m(s_{\mathrm{SA}[i]})$ und t entscheidet je nach Suchalgorithmus darüber, in welcher Hälfte des Suchintervalls fortgefahren wird und ob $k_1 := k$ oder $k_2 := k$ gesetzt wird. Es gilt $k_1 = -1 \Leftrightarrow l_1 = 0$ und $k_2 = -1 \Leftrightarrow l_2 = n$. Der Wert -1 für k_1 bzw. k_2 bedeutet, dass der lcp-Wert von t und $s_{\mathrm{SA}[-1]}$ bzw. $s_{\mathrm{SA}[n]}$ nicht definiert ist. Die Fälle 2 oder 3 treten nur für definierte lcp-Werte $k_1 > -1$ bzw. $k_2 > -1$ in Kraft. k_1 und k_2 können auch mit 0 initialisiert werden, die Fälle 2 oder 3 treten dann für $k_1 > 0$ bzw. $k_2 > 0$ in Kraft, die dann auch definiert sind. Der Wert -1 diente der besseren Veranschaulichung.

Zeichenvergleiche sind in den Fällen 1, 2.3 und 3.3 notwendig und beginnen alle an Position $\max\{k_1, k_2, 0\}$. Da weder k_1 noch k_2 während der Suche verringert werden, ergibt sich die Gesamtlaufzeit von $\mathcal{O}(m+\log n)$. Tabelle 8.1 zeigt die im Vergleich zur verbesserten Suffix-Array-Suche eingesparten Zeichenvergleiche.

Fall	1	2	3
.1		$k_1 - k_2$	$k_2 - k_1$
.2	0	$\mathrm{lcp}(s_{\mathrm{SA}[l_1-1]}, s_{\mathrm{SA}[i]}) - k_2$	$\mathrm{lcp}(s_{\mathrm{SA}[l_2]}, s_{\mathrm{SA}[i]}) - k_1$
.3		$k_1 - k_2$	$k_2 - k_1$

Tabelle 8.1: Eingesparte Zeichenvergleiche der verbesserten Suche (pro Schritt)

8.3 LCP-Intervallbaum

Der in 8.2 vorgestellte Algorithmus setzt voraus, dass auf die lcp-Werte der Suffixpaare $(s_{\mathrm{SA}[l_1-1]}, s_{\mathrm{SA}[i]})$ und $(s_{\mathrm{SA}[l_2]}, s_{\mathrm{SA}[i]})$ in $\mathcal{O}(1)$ zugegriffen werden kann. Ausgehend von der LCP-Tabelle LCP ist dies aber nur in $\mathcal{O}(i - l_1 + 1)$ bzw. $\mathcal{O}(l_2 - i)$ möglich. Benutzt man einen balancierten LCP-Intervallbaum [21], der alle in der Binärsuche evtl. benötigten lcp-Werte enthält, sind diese Zugriffe jeweils in $\mathcal{O}(1)$ möglich.

Im Folgenden soll ein LCP-Intervallbaum vorgestellt werden, der leicht zu konstruieren ist und dessen Serialisierung in Breitensuche die LCP-Tabelle als Teilstring enthält. Wir definieren das LCP-Intervall $\mathrm{LCP}[a, b]$ als

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

Dann gilt $\mathrm{LCP}[a, b] = \mathrm{lcp}(s_{\mathrm{SA}[a]}, s_{\mathrm{SA}[b]})$. Für beliebige $c \in (a..b)$ gilt $\mathrm{LCP}[a, b] = \min \{\mathrm{LCP}[a, c], \mathrm{LCP}[c, b]\}$. Ein Intervallbaum ist ein Baum dessen Knoten Intervalle sind.

Definition 8.3.1
Es sei $n \in \mathbb{N}$ gegeben. Der (linksvollständige) Intervallbaum T_n ist ein Binärbaum und durch folgende Eigenschaften definiert:

- *Jedem Knoten ist ein Wert zugeordnet.*
- *Der Wert ist ein Intervall $[a..b]$ mit $a, b \in [0..n)$ und $a < b$.*
- *Die Wurzel erhält das Intervall $[0..n - 1]$.*
- *Ein Knoten $[a..b]$ mit $b - a \geq 2$ hat $[a, c]$ und $[c, b]$ als linkes bzw. rechtes Kind, wenn $c - a$ die größte Zweierpotenz kleiner $b - a$ ist.*
- *Ein Knoten $[a..b]$ mit $b - a = 1$ ist ein Blatt.*

Definition 8.3.2
Zu einem Intervallbaum T und einer LCP-Tabelle LCP definieren wir den LCP-Intervallbaum $T(\mathrm{LCP})$ durch Ersetzen der Intervalle $[a..b]$ durch lcp-Werte $\mathrm{LCP}[a, b]$.

T_n ist balanciert und hat die Eigenschaft, dass alle linken Kinder vollständige Binärbäume sind. Im Folgenden bezeichnen wir $T_{\mathrm{LCP}} := T_{|\mathrm{LCP}|}(\mathrm{LCP})$ auch als LCP-Baum zu LCP. Der LCP-Baum zu $s =$ tobeornottobe und der zugehörigen LCP-Tabelle (siehe Abb. 3.2) ist in Abbildung 8.1 dargestellt.

Damit der LCP-Baum in der verbesserten Binärsuche benutzt werden kann, muss in Algorithmus 8 die Wahl von i das Setzen der Intervallgrenzen geändert werden. Algorithmus 9 zeigt eine vereinfachte Implementierung in pseudo-code. Die Relation $<_{|t|}^{k_1}$ soll einen Zeichenvergleich der Zeichen $[k_1..|t|)$ verdeutlichen.

simpleL ist die am Anfang von 8.2 beschriebene $\mathcal{O}(m \log n)$ l-Suche, die in diesem Fall eine Laufzeit von $\mathcal{O}(m)$ hat. Durch die Wahl von i werden in der verbesserten Binärsuche ausschließlich Intervalle des Intervallbaums benötigt. Ist $[l_1, l_2]$ das aktuelle Suchintervall wird für $k_1 > k_2$ oder $k_1 < k_2$ das linke bzw. rechte Kind des Knotens LCP$[l_1, l_2]$ verwendet.

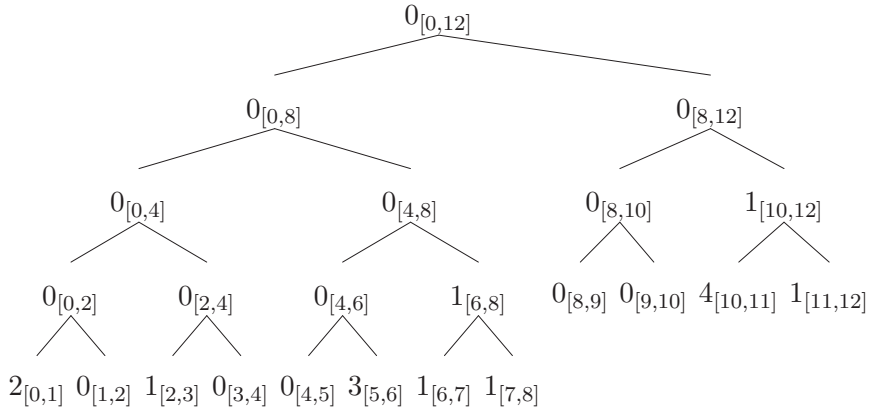


Abbildung 8.1: linksvollständiger LCP-Baum für $s =$ tobeornottobe

Konstruktion

Die Form des Binärbaums T_{LCP} hängt nicht von den Einträgen in LCP sondern nur von $n = |\text{LCP}|$ ab. Durch eine geeignete Bijektion der Knoten in Indizes, lässt sich T_{LCP} daher leicht in einem Feld LCPE speichern. Diese Bijektion nennen wir auch Serialisierung. Da die Blätter genau die Einträge LCP$[i]$ für $i = 1, \ldots, n-1$ sind, streben wir eine Serialisierung an, mit der einfach und in $\mathcal{O}(1)$ auf LCP$[i]$ zugegriffen werden kann. Wir definieren zunächst den expandierten Intervallbaum $\bar{T}_n$.

Definition 8.3.3
Es sei $n \in \mathbb{N}$ gegeben und $N := 2^{\lceil \log_2(n-1) \rceil}$. Der Intervallbaum $\bar{T}_n$, der entsteht, wenn ausgehend von T_N alle Knoten $[a..b]$ mit $[a..b] \cap [0..n) = \emptyset$ entfernt und alle übrigen $[a..b]$ in $[a..n)$ umgewandelt werden, heißt expandierter Intervallbaum.

Im expandierten Intervallbaum haben alle Blätter die gleiche Tiefe (Abstand zur Wurzel). Außerdem ist T_n eine Unterteilung von $\bar{T}_n$. Wir bezeichnen den LCP-Intervallbaum $\bar{T}_{\text{LCP}} := \bar{T}_{|\text{LCP}|}(\text{LCP})$ auch kurz als expandierten LCP-Baum zu LCP. Der expandierte Intervallbaum $\bar{T}_n$ hat die Tiefe $d := \lceil \log_2(n-1) \rceil$. Die Knoten der Tiefe i, mit $i \in [0..d]$ beliebig gewählt, tragen von links nach rechts die Intervalle $[a_0..a_1], [a_1..a_2], \ldots, [a_{k_i-2}..a_{k_i-1}], [a_{k_i-1}..n)$ mit $a_j = j \cdot l_i$, wobei $l_i := 2^{d-i}$ die maximale Intervallbreite und $k_i := \left\lceil \frac{n-1}{l_i} \right\rceil$ die Anzahl der Knoten in Tiefe i sind.

Algorithmus 9 verbesserte l-Suche

```
1:  l_1 := k_1 := k_2 := 0
2:  l_2 := |s| - 1
3:  while l_2 - l_1 > 1 do
4:    i := l_1 + 2^⌊log_2(l_2 - l_1 - 1)⌋
5:    if k_1 > k_2 then
6:      if LCP[l_1, i] > k_1 then                  // Fall 2.1
7:        l_1 := i
8:      else if LCP[l_1, i] < k_1 then             // Fall 2.2
9:        (l_2, k_2) := (i, LCP[l_1, i])
10:     else                                       // Fall 2.3
11:       if s_SA[i] <^{k_1}_{|t|} t then
12:         (l_1, k_1) := (i, lcp(s_i, t))
13:       else
14:         (l_2, k_2) := (i, lcp(s_i, t))
15:   else if k_1 < k_2 then
16:     if LCP[i, l_2] > k_2 then                  // Fall 3.1
17:       l_2 := i
18:     else if LCP[i, l_2] < k_2 then             // Fall 3.2
19:       (l_1, k_1) := (i, LCP[i, l_2])
20:     else                                       // Fall 3.3
21:       if s_SA[i] <^{k_2}_{|t|} t then
22:         (l_1, k_1) := (i, lcp(s_i, t))
23:       else
24:         (l_2, k_2) := (i, lcp(s_i, t))
25:   else                                         // Fall 1
26:     if s_SA[i] <^{k_1}_{|t|} t then
27:       (l_1, k_1) := (i, lcp(s_i, t))
28:     else
29:       (l_2, k_2) := (i, lcp(s_i, t))
30: return simpleL(s,t,SA,l_1,l_2)                 // einfache l-Suche im Intervall [l_1, l_2]
```

Es sei $f_i := \Sigma_{j=0}^{i-1} k_j$. $\bar{T}_{\mathrm{LCP}}$ wird nun derart serialisiert, dass der Wert der Wurzel an Position $f_0 = 0$, die Werte der Knoten in Tiefe 1 von links nach rechts an den Positionen $[f_1..f_2)$, die Werte der Knoten in Tiefe 2 an den Positionen $[f_2..f_3)$ usw. im Feld LCPE der Länge f_{d+1} gespeichert werden. Da $l_d = 1$ gilt, tragen in $\bar{T}_{\mathrm{LCP}}$ die Blätter von links nach rechts die Einträge 1 bis n von LCP. Es gilt also $\mathrm{LCPE}[f_d..|\mathrm{LCPE}|) = \mathrm{LCP}[1..n)$ und LCPE enthält die LCP-Tabelle als Teilstring. Für die Länge von LCPE gilt:

$$2n - 3 \leq |\mathrm{LCPE}| < 2n - 3 + d.$$

LCPE lässt sich konstruieren, indem LCP linear gelesen wird und parallel für alle $i \in [0..d]$ jeweils nach l_i Lesezyklen das Minimum der letzten l_i Werte in F geschrieben wird, beginnend an Position f_i. Ist das Auslesen von LCP beendet und liegt das letzte Schreiben für ein i nur $0 < k < l_i$ Zyklen zurück, wird das Minimum dieser k Werte gebildet und an Position $f_{i+1} - 1$ geschrieben.

Nutzt man die Tatsache, dass für ein beliebiges $i \in [0..d)$ das gesuchte Minimum in Tiefe i dem Minimum der letzten beiden Minima in Tiefe $i + 1$ entspricht, ist diese Konstruktion in $\mathcal{O}(n)$ möglich.

8.4 Externalisierung und Analyse

Die interne Konstruktion von LCPE ist in linearer Zeit mit zusätzlichem Speicher von $\mathcal{O}(\log n)$ möglich. Sie lässt sich leicht externalisieren und mit höchstens $d \cdot P$ Puffern der Größe B und

$$T_{\mathrm{LCPE}}(n) = \mathrm{scan}(n) = \Theta\left(\frac{n}{PB}\right)$$

I/O-Zugriffen kann LCPE I/O-optimal erzeugt werden. Der interne Speicherbedarf des externen Algorithmus ist dann $\mathcal{O}\left(PB \cdot \log n\right)$.

Die Binärsuche benötigt $\mathcal{O}(\log n)$ Zugriffe auf die Felder SA bzw. LCPE. Die Zugriffe auf LCPE und SA sind zum Ende bzw. zu Beginn der Suche ortslokal. Befinden sich SA bzw. LCPE im Sekundärspeicher ergibt sich für die maximale Anzahl der I/O-Zugriffe $\mathrm{binary}(n)$ auf die Felder:

$$\mathrm{binary}(n) = \Theta\left(\log \frac{n}{PB}\right).$$

Die maximale Zahl der I/O-Zugriffe $\bar{T}_{\mathrm{Search_{SA}}}(n, m)$ der externen Binärsuche kann nun folgendermaßen abgeschätzt werden:

$$\begin{aligned} \bar{T}_{\mathrm{Search_{SA}}}(n, m) &= \overbrace{\lceil \log_2 n \rceil \cdot (\mathrm{scan}(m) + 1)}^{s} + \overbrace{\mathrm{binary}(n)}^{\mathrm{SA}} \\ &= \Theta\left(\tfrac{m \log n}{PB} + \log \tfrac{n}{PB}\right) \end{aligned}$$

und für die externe Binärsuche mit zusätzlicher LCPE-Tabelle gilt:

$$\begin{aligned}\bar{T}_{\text{Search}_{\text{LCPE}}}(n,m) &= \overbrace{2\lceil\log_2 n\rceil + \text{scan}(m)}^{s} + \overbrace{\text{binary}(n)}^{\text{SA}} + \overbrace{\text{binary}(n)}^{\text{LCPE}} \\ &= \Theta\left(\tfrac{m}{PB} + \log n\right).\end{aligned}$$

Die Terme für die Zugriffe auf s ergeben sich aus den worst-case Abschätzungen der internen Algorithmen, wobei zu beachten ist, dass das Lesen eines Substrings der Länge m nicht $\text{scan}(m)$ sondern schlimmstenfalls $\text{scan}(m)+1$ I/O-Zugriffe kosten kann, wenn er nicht an einer Blockgrenze beginnt. Außerdem gilt genaugenommen für das Zusammenfassen zweier scan-Summanden $\text{scan}(a+b) \leq \text{scan}(a) + \text{scan}(b) \leq \text{scan}(a+b)+1$. Aus diesen beiden Eigenschaften resultiert der Term $2\lceil\log_2 n\rceil$.

Mögliche Verbesserungen

Die im Abschnitt 8.3 vorgestellte Serialisierung hat die Eigenschaft, dass jeder Pfad von der Wurzel bis zu einem Blatt in T_{LCP} eine zugehörige Folge von streng monoton steigenden Indizes in LCPE hat. Betrachtet man unter allen Pfaden die jeweiligen Einträge in LCPE, werden Elemente mit geringeren Indizes häufiger besucht. Für einen externen Suchalgorithmus ist es bei wiederholten Suchanfragen deshalb sinnvoll, einen Präfix von LCPE im Speicher vorzuhalten. Hat dieser Präfix die Länge $M > PB$, kann die Zahl der I/O-Zugriffe auf LCPE reduziert werden zu:

$$\text{binary}_M(n) = \Theta\left(\log \frac{n}{M}\right).$$

In dieser Arbeit nicht implementiert, für die externe Suche in den I/O-Zugriffen aber noch effizienter, ist die Speicherung von T_{LCP} als B-Baum. Dabei sind die Knoten des B-Baums disjunkte Teilbäume von T_{LCP} möglichst und höchstens der Größe $P \cdot B$. Der Algorithmus zur Konstruktion der LCPE-Tabelle kann so modifiziert werden, dass er den B-Baum mit $\mathcal{O}(\log_{PB} n)$ Puffern der Größe $P \cdot B$ und ebenfalls I/O-optimal in $\Theta\left(\frac{n}{PB}\right)$ erzeugt. Die externe Suche mit Algorithmus 9 benötigt dann $\text{search}(n) = \Theta\left(\log_{PB} n\right)$ I/O-Zugriffe auf die lcp-Werte.

Kapitel 9

Der Index

In diesem Kapitel werden Abstrakte Datentypen (ADT) entwickelt, welche allgemein für die exakte Substring-Suche benötigt werden. Anschließend wird der ADT 'Substring-Index' als Erweiterung einer dieser Klassen entworfen. Die Spezifikationen der Abstrakten Datentypen ergeben sich aus den Anforderungen an einen generischen Substring-Index bzw. einer allgemeinen Suchschnittstelle. Der dritte Abschnitt geht schließlich näher auf die Implementierung des Index in Seqan ein.

9.1 Anforderungen

Die wichtigste Funktion eines Substring-Index zu einem String s ist die effiziente Suche nach Vorkommen beliebiger Strings t als Substrings in s. Von einem generischen Substring-Index soll Folgendes erwartet werden:

1. Erzeugung des Index eines oder mehrerer Strings s_i mit $i \in [0..I)$
2. Suche nach einem oder mehreren Substrings t_j mit $j \in [0...J)$
3. Unterstützung verschiedener Sucharten, wie die Suche nach Existenz, Anzahl oder Positionen der Treffer
4. Unterstützung verschiedener Algorithmen zum Aufbau des Index und verschiedener Indexkomplexitäten
5. Funktionen zur Serialisierung des Index

Die Funktionen zum Suchen von Substrings in Strings sollen einer generischen Schnittstelle entsprechen, der Query-Schnittstelle. Die Anforderungen 1 bis 3 lassen sich allgemein auf beliebige Suchklassen übertragen. Um eine spätere Integration von Online-Suchalgorithmen mit identischer Schnittstelle in Seqan zu ermöglichen, soll die folgende Anforderung hinzugefügt werden:

6. Die Query-Schnittstelle des Index sei eine Erweiterung einer einheitlichen Query-Schnittstelle für beliebige Suchalgorithmen

Die Online-Suche unterscheidet sich von der Index-Suche in der Art der Daten-Vorverarbeitung, so werden bei der Index-Suche Informationen des Textes im Index und bei der Online-Suche Informationen der Suchmuster einmalig vorberechnet, um die Suche zu beschleunigen. Für eine allgemeine Query-Schnittstelle sollen daher die Datentypen *Haystack* und *Needle* verwendet werden, die jeweils eine einheitliche Schnittstelle haben, aber je nach Suchklasse verschieden implementiert werden können. Die approximative Mustersuche soll in dieser Arbeit nicht weiter betrachtet werden, sie lässt sich aber ohne Weiteres in die Schnittstellen integrieren. Aus den oben aufgeführten Anforderungen sollen nun abstrakte Datentypen der exakten Substring-Suche und des Substring-Index entwickelt werden und in Seqan mit den in den vorherigen Kapiteln vorgestellten Algorithmen modelliert werden.

9.2 Abstrakte Datentypen

Ein Abstrakter Datentyp [10] ist definiert durch eine Σ-Signatur $\Sigma = (\mathrm{S}, \mathrm{OP})$ mit Variablen X zu den Sorten S und Operationen OP und einer Menge von Gleichungen (Axiome) E bzgl. (Σ, X). $\mathrm{SP} = (\Sigma, \mathrm{E})$ wird auch als Algebraische Spezifikation bezeichnet, deren Modellmenge der entsprechende Abstrakte Datentyp ist. Abbildung 9.1 zeigt die für die exakte Mustersuche benötigten Abstrakten Datentypen String, Multi, Haystack, Needle und Result. Zur Vereinfachung ist nur ein abgeschlossener Teil der benötigten Funktionalität angegeben und nicht alle Axiome sind auf Termgleichungen beschränkt (siehe Result). Der ADT String (oder auch String_Σ) spezifiziert einen String des Alphabets Σ. Multi wird für die Suche in bzw. nach mehreren Strings verwendet und kann eine Kette von mehreren Strings enthalten. Haystack und Needle sind Typen, die die Texte s_i bzw. Suchmuster t_j beinhalten und aus denen der Typ Result erzeugt werden kann, welcher die Suchergebnisse enthält. Die Suchergebnisse in Result können analog zur Iteratorklasse aus C++ mit den Funktionen $*$ und ++ iteriert werden und mit rewind bzw. eof erneut gelesen und auf weitere Treffer überprüft werden. Ein konkreter Suchalgorithmus implementiert nun Verfeinerungen der Typen Haystack, Needle und Result.

Der ADT Index aus Abbildung 9.2 ist eine Verfeinerung von Haystack und spezifiziert einen Substring-Index. Er basiert auf einem Suffix-Array ($\mathrm{Index}_{\mathrm{SA}}$) und bei Bedarf zusätzlich auf einer erweiterten LCP-Tabelle ($\mathrm{Index}_{\mathrm{LCPE}}$), welche die Binärsuche im Suffix-Array beschleunigen soll. Im Fall $I = 1$ enthält der Index das Suffix-Array von $s = s_0$ und im Fall $I > 1$ das Suffix-Array der Konkatenation $s = \prod_{i=0}^{I-1} s_i$ aller Texte s_i. Für $i \in [0..I)$ und $j \in [0..J)$ kommt

der String t_j kommt genau dann in einem s_i vor, wenn er in s an Stelle p vorkommt und $\sum_{k=0}^{i-1} |s_k| \leq p \leq \sum_{k=0}^{i} |s_k| - |t_j|$ gilt. Die Suche nach J Mustern in I Texten lässt sich so zu einer Suche nach J Mustern in einem Text, also J unabhängigen Binärsuchen im Suffix-Array reduzieren. Mit einem Binärbaum oder einer Skip-Liste können in $O(\log I)$ die Treffer in s auf etwaige Treffer in den $s_i, i \in I$ abgebildet werden.

9.3 Implementierung

Im Folgenden soll die Umsetzung des Abstrakten Datentyps Index in Seqan beschrieben werden. Um einen Index für große Strings zu implementieren, die mitunter nicht mehr vollständig im Hauptspeicher Platz finden, mussten die generischen String-Klassen in Seqan um eine weitere ergänzt werden, der External-String-Klasse, einer Container-Klasse für Strings im Haupt- und Sekundärspeicher. Die Schnittstelle soll eine einfache Integration dieser Klasse ermöglichen, ohne dass bestehende Algorithmen in Seqan angepasst werden müssen.

Die in Kapitel 5 vorgestellten Pipeline-Module wurden zusammen mit einer generischen und erweiterbaren Pipeline-Schnittstelle implementiert und erlauben das einfache Komponieren komplexer Algorithmen, wie den externen Skew-Algorithmen oder den externen LCP-Algorithmus.

Zum Zugriff auf den Sekundärspeicher wurde eine plattformunabhängige und generische I/O-Schnittstelle entwickelt, die synchrone und asynchrone I/O-Zugriffe unterstützt. Sie kapselt die Interna der Plattform und des Dateisystems und liegt den Pool-Modulen und der External-String-Klasse zu Grunde.

Nach der Umsetzung der in dieser Arbeit vorgestellten bzw. entwickelten internen und externen Algorithmen wurde die generische Index-Klasse entsprechend des ADT implementiert. Der Benutzer kann den Index in verschiedenenen Komplexitäten (Suffix-Array, Suffix-Arraymit LCP-Tabelle, ...) spezialisieren und manuell in die Auswahl der Konstruktionsalgorithmen eingreifen. Die Suche in oder mit mehreren Strings kann über die implementierte Klasse Multiple realisiert werden.

9.3.1 I/O-Schnittstelle

Im Wesentlichen besteht die I/O-Schnittstelle aus einer generischen File-Klasse. Sie stellt Funktionen zur Dateiverwaltung und sowohl synchrone als auch asynchrone Funktionen zum Lesen und Schreiben von Daten zur Verfügung. Die File-Klasse unterstützt transparent verschiedene System-I/O-Schnittstellen darunter: POSIX IEEE[1] 1003.1-2001, Win32 API und ANSI C. Über Tags (Kapitel B) kann auch eine ganz bestimmte Schnittstelle verwendet werden. So ist es bspw.

[1] http://standards.ieee.org/

Typ	**String/String$_\Sigma$**
Sorten	Σ, String, $\mathbb{N}$
Operat.	create :$\longrightarrow$ String
	append : String $\times\ \Sigma \rightarrow$ String
	substr : String $\times\ \mathbb{N} \times \mathbb{N} \rightarrow$ String
	suffix : String $\times\ \mathbb{N} \rightarrow$ String
	len : String $\rightarrow \mathbb{N}$
	at : String $\times\ \mathbb{N} \rightarrow \Sigma$
Axiome	$(x, S, i, j, l) \in \Sigma \times$ String $\times\ \mathbb{N}^3$,
	$i < \text{len}(S), j < \min(l, \text{len}(S) - i)$:
	$\text{len}(\text{create}) = 0$
	$\text{len}(\text{append}(S, x)) = \text{len}(S) + 1$
	$\text{len}(\text{substr}(S, i, l)) = \min(l, \text{len}(S) - i)$
	$\text{at}(\text{append}(S, x), \text{len}(S)) = x$
	$\text{at}(\text{append}(S, x), i) = \text{at}(S, i)$
	$\text{at}(\text{substr}(S, i, l), j) = \text{at}(S, i + j)$
	$\text{suffix}(S, i) = \text{substr}(S, i, \text{len}(S) - i)$

Typ	**Multi**
Sorten	Σ, String, Multi, $\mathbb{N}$
Operat.	create :$\longrightarrow$ Multi
	append : Multi $\times$ String $\rightarrow$ Multi
	len : Multi $\rightarrow \mathbb{N}$
	at : Multi $\times\ \mathbb{N} \rightarrow$ String
	serial : Multi $\rightarrow$ String
Axiome	$(S, M, i, j_1, j_2, l) \in$ String $\times$ Multi $\times\ \mathbb{N}^4$,
	$k < \text{len}(M), i_1 < \text{len}(M) \leq i_2$
	$\text{len}(\text{create}) = 0$
	$\text{len}(\text{append}(M, S)) = \text{len}(M) + 1$
	$\text{at}(\text{append}(M, S), \text{len}(M)) = S$
	$\text{at}(\text{append}(M, S), k) = \text{at}(M, k)$
	$\text{len}(\text{serial}(\text{create})) = 0$
	$\text{len}(\text{serial}(\textit{append}(M, S))) = \text{len}(\text{serial}(M)) + \text{len}(S)$
	$\text{at}(\text{serial}(\text{append}(M, S)), i_1) = \text{at}(\text{serial}(M), i_1)$
	$\text{at}(\text{serial}(\text{append}(M, S)), i_2) = \text{at}(S, i_2 - \text{len}(\text{serial}(M)))$

Typ	**Haystack**
Sorten	Haystack, Alg, String, Multi, $\mathbb{N}$
Operat.	create : String $\times$ Alg $\rightarrow$ Haystack
	create : Multi $\times$ Alg $\rightarrow$ Haystack
	at : Haystack $\times\ \mathbb{N} \rightarrow$ String
Axiome	$(S, M, i) \in$ String $\times$ Multi $\times\ \mathbb{N}, i < \text{len}(M)$:
	$\text{at}(\text{create}(S), 1) = S$
	$\text{at}(\text{create}(M), i) = \text{at}(M, i)$

Typ	**Needle**
Sorten	Needle, Alg, String, Multi, $\mathbb{N}$
Operat.	create : String $\times$ Alg $\rightarrow$ Needle
	create : Multi $\times$ Alg $\rightarrow$ Needle
	at : Needle $\times\ \mathbb{N} \rightarrow$ String
Axiome	$(S, M, i) \in$ String $\times$ Multi $\times\ \mathbb{N}, i < \text{len}(M)$:
	$\text{at}(\text{create}(S), 1) = S$
	$\text{at}(\text{create}(M), i) = \text{at}(M, i)$

Typ	**Result**
Sorten	Haystack, Needle, Result, $\mathbb{N}$, $\mathbb{N}^3$, Bool
Operat.	query : Haystack $\times$ Needle $\rightarrow$ Result
	queryCount : Haystack $\times$ Needle $\rightarrow \mathbb{N}$
	queryExist : Haystack $\times$ Needle $\rightarrow$ Bool
	hayAt : Result $\times\ \mathbb{N} \rightarrow$ String
	ndlAt : Result $\times\ \mathbb{N} \rightarrow$ String
	$*$: Result $\rightarrow \mathbb{N}^3$
	$++$: Result $\rightarrow$ Result
	rewind : Result $\rightarrow$ Result
	eof : Result $\rightarrow$ Bool
Axiome	$(H, N, i) \in$ Haystack $\times$ Needle $\times\ \mathbb{N}$,
	$M_{\text{hit}} := \left\{(p, h, n) \in \mathbb{N}^3 \mid \text{substr}\left(\text{hayAt}(h), p, \text{len}(\text{ndlAt}(n))\right) = \text{ndlAt}(n)\right\}$:
	$\text{hayAt}(\text{query}(H, N), i) = \text{at}(H, i)$
	$\text{ndlAt}(\text{query}(H, N), i) = \text{at}(N, i)$
	$\text{queryCount}(H, N) = \lvert M_{\text{hit}} \rvert$
	$\text{queryExist}(H, N) = (\text{queryCount}(H, N) > 0)$
	$\text{rewind}\left(\text{query}(H, N)++^i\right) = \text{query}(H, N)$
	$\text{eof}\left(\text{query}(H, N)++^i\right) = (i \geq \text{queryCount}(H, N))$
	$*\left(\text{query}(H, N)++^i\right) \in M_{\text{hit}} \quad \mid \text{eof}\left(\text{query}(H, N)++^i\right) = \mathit{false}$
	$*\left(\text{query}(H, N)++^i\right) = *\left(\text{query}(H, N)++^j\right) \Rightarrow i = j$

Abbildung 9.1: Abstrakte Datentypen der exakten Substring-Suche

Typ	**Index**$_{\mathrm{SA}}$ ($\supset$ Haystack)
Sorten	Index$_{\mathrm{SA}}$, Alg, String, String$_{\mathbb{N}}$, Multiple, $\mathbb{N}$
Operat.	create : String $\times$ Alg $\rightarrow$ Index$_{\mathrm{SA}}$
	create : Multi $\times$ Alg $\rightarrow$ Index$_{\mathrm{SA}}$
	at : Index$_{\mathrm{SA}}$ $\times$ $\mathbb{N}$ $\rightarrow$ String
	text : Index$_{\mathrm{SA}}$ $\rightarrow$ String
	sa : Index$_{\mathrm{SA}}$ $\rightarrow$ String$_{\mathbb{N}}$
Axiome	$(I, S, M, i, j) \in \mathrm{Index_{SA}} \times \mathrm{String} \times \mathrm{Multi} \times \mathbb{N}^2$,
	$i < \mathrm{len}(M), j < \mathrm{len}(\mathrm{text}(M)) - 1$:
	$\mathrm{at}(\mathrm{create}(S), 1) = S$
	$\mathrm{at}(\mathrm{create}(M), i) = \mathrm{at}(M, i)$
	$\mathrm{text}(\mathrm{create}(S)) = S$
	$\mathrm{text}(\mathrm{create}(M)) = \mathrm{serial}(M)$
	$\mathrm{len}(\mathrm{sa}(I)) = \mathrm{len}(\mathrm{text}(I))$
	$\mathrm{suffix}(\mathrm{text}(I), \mathrm{at}(sa(I), j)) <$
	$\mathrm{suffix}(\mathrm{text}(I), \mathrm{at}(sa(I), j+1))$

Typ	**Index**$_{\mathrm{LCPE}}$ ($\supset$ Index$_{\mathrm{SA}}$ $\supset$ Haystack)
Sorten	Index$_{\mathrm{LCPE}}$, Alg, String, String$_{\mathbb{N}}$, Multiple, $\mathbb{N}$
Operat.	create : String $\times$ Alg $\rightarrow$ Index$_{\mathrm{LCPE}}$
	create : Multi $\times$ Alg $\rightarrow$ Index$_{\mathrm{LCPE}}$
	create : Index$_{\mathrm{SA}}$ $\times$ Alg $\rightarrow$ Index$_{\mathrm{LCPE}}$
	at : Index$_{\mathrm{LCPE}}$ $\times$ $\mathbb{N}$ $\rightarrow$ String
	text : Index$_{\mathrm{SA}}$ $\rightarrow$ String
	sa : Index$_{\mathrm{LCPE}}$ $\rightarrow$ String$_{\mathbb{N}}$
	lcpe : Index$_{\mathrm{LCPE}}$ $\rightarrow$ String$_{\mathbb{N}}$
	lcp : Index$_{\mathrm{LCPE}}$ $\rightarrow$ String$_{\mathbb{N}}$
Axiome	$(I, S, M, i, j) \in \mathrm{Index_{LCPE}} \times \mathrm{String} \times \mathrm{Multi} \times \mathbb{N}^2$,
	$i < \mathrm{len}(M), j < \mathrm{len}(\mathrm{text}(M)) - 1$:
	$\mathrm{at}(\mathrm{create}(S), 1) = S$
	$\mathrm{at}(\mathrm{create}(M), i) = \mathrm{at}(M, i)$
	$\mathrm{text}(\mathrm{create}(S)) = S$
	$\mathrm{text}(\mathrm{create}(M)) = \mathrm{serial}(M)$
	$\mathrm{len}(\mathrm{sa}(I)) = \mathrm{len}(\mathrm{text}(I))$
	$\mathrm{suffix}(\mathrm{text}(I), \mathrm{at}(sa(I), j)) <$
	$\mathrm{suffix}(\mathrm{text}(I), \mathrm{at}(sa(I), j+1))$
	$\mathrm{lcp}(I) = \mathrm{substr}(\ \mathrm{lcpe}(I),$
	$\mathrm{len}(\mathrm{lcpe}(I)) - (\mathrm{len}(\mathrm{sa}(I)) - 1),$
	$\mathrm{len}(\mathrm{sa}(I)) - 1\ )$
	$\vdots$

Abbildung 9.2: Abstrakter Datentyp Substring-Index mit Suffix-Array (links) und zusätzlicher erweiterter LCP-Tabelle (rechts)

möglich, die externe String-Klasse oder den externen Skew-Algorithmus mit den ANSI-C-Funktionen zu spezialisieren. Die asynchronen I/O-Funktionen werden dann auf synchrone abgebildet. Tabelle 9.1 zeigt im Vergleich einen Ausschnitt der Funktionen für synchrone und asynchrone Zugriffe.

Die I/O-Schnittstelle unterstützt mehrere Festplatten $P > 1$ über File-Striping (vgl. DSM Seite 16) und unterstützt lange Dateien ($>$ 4GB) sowie virtuell lange Dateien, die tatsächlich aus mehreren kleineren Dateien bestehen, um Dateisystembeschränkungen zu umgehen.

9.3.2 Externer String

Der externe String funktioniert ähnlich wie der Paging-Mechanismus der virtuellen Speicherverwaltung eines Betriebssystems. Dem String wird eine Datei zugeordnet und er wird unterteilt in zusammenhängende Seiten (engl. Pages) der Größe P, von denen sich im Allgemeinen nur ein kleiner Teil im Hauptspeicher befindet. Eine Seitentabelle enthält zu einer Seite die Information ob und wo sich eine Seite im Hauptspeicher befindet. Damit auf eine Seite außerhalb des Hauptspeichers zugegriffen werden kann, muss gegebenenfalls vorher eine andere Seite im Hauptspeicher durch diese ersetzt werden. Entscheidend für die Effizienz der externen String-Klasse ist dabei die Seitenersetzungsstrategie. Die

[2] siehe Kapitel B

Notation	
F	eine File-Klasse
f	ein Objekt der Klasse F
p_o	Objekt des Typs Size<P>::Type
c	ein Objekt des Typs aRequest<P>::Type

Metafunktion[2]	**Rückgabe**
Size<F>::Type	Typ für die Position Datei
aRequest<F>::Type	Kontexttyp asynchroner I/O-Operationen

Funktion	**Semantik**
write(...)	synchrones Schreiben
writeAt(..., p_o)	synchrones Schreiben an Position p_o
awriteAt(..., p_o, c)	startet asynchrones Schreiben an Position p_o
waitFor(c)	wartet auf den Abschluss von c

Tabelle 9.1: I/O-Schnittstelle von Seqan (Auszug)

meisten Betriebssysteme verwenden dabei eine LRU-Strategie, welche die Seite ersetzt, auf die am längsten nicht zugegriffen wurde.

Für die Implementierung des externen Strings wurde die prioritätenbasierte LRU-Strategie, eine Erweiterung der LRU-Strategie, benutzt. Dabei können den eingelagerten Seiten Prioritäten zugeordnet werden, so dass Seiten zuerst nach Priorität und dann nach dem letzten Zugriff geordnet werden. Außerdem werden unversehrte Seiten, also jene, die beim Auslagern nicht auf die Festplatte geschrieben werden müssen, bevorzugt ausgelagert.

Seiten, die permanent eingelagert werden sollen, bspw. die Wurzel und dicht darunterliegende Knoten eines serialisierten Suchbaums, erhalten die höchste Priorität. Seiten, auf die mit hoher Wahrscheinlichkeit kurz hintereinander zugegriffen wird, bspw. mit einen sequentiellen Iterator vergleichbar mit STL[3]-Forward-Iteratoren, erhalten eine mittlere Priorität und Seiten auf die mit Random-Iteratoren oder direkt zugegriffen wird, erhalten die niedrigste Priorität. Treffen mehrere Bedingungen zu, gilt das Maximum der Prioritäten. Der externe String benutzt außerdem asynchrone Prefetching- und Write-Through-Techniken, wenn ein sequentielles Zugriffsmuster erkennbar ist, um das Ein- und Auslagern zu parallelisieren.

9.3.3 Pipeline-Schnittstelle

Zu den in Kapitel 5 vorgestellten Pipeline-Modulen wurden die generische Klassen Pipe und Pool entwickelt, die mit der Art des Moduls (bspw. Echoer, Map-

[3] C++ Standard Template Library `http://www.sgi.com/tech/stl/`

Notation	
$\mathcal{P}$	Menge der Pipe-Klassen
$\mathcal{P}_o$	Menge der Pool-Klassen ($P \subseteq P_o$)
P, P_o	Klassen mit $P \in \mathcal{P}$ und $P_o \in \mathcal{P}_o$
p, p_o	Objekte der Klassen P bzw. P_o
n	die Länge des Ausgabestroms
j	ein Zähler aus $[0..n)$ zur Verdeutlichung
e_j, a_j	Zeichen des Ein- bzw. Ausgabestroms
x	ein Objekt des Typs Value<P>::Type

Metafunktion	**Rückgabe**
Value<P>::Type	Typ der Ausgabezeichen
Size<P>::Type	Typ für die Länge des Ausgabestroms

Funktion	**Semantik**	**Rückgabe**
size(p)		n
beginRead(p)	$0 \rightarrow j$	*true*, wenn erfolgreich
pop(p)	$j + 1 \rightarrow j$	
front(p, x)	$a_j \rightarrow x$	
pop(p, x)	front(p, x); pop(p)	
eof(p)		$true \Leftrightarrow j \geq n$
endRead(p)		*true*, wenn erfolgreich
resize(j)	$j \rightarrow n$	
beginWrite(p_o)	$0 \rightarrow j$	*true*, wenn erfolgreich
push(p_o, x)	$x \rightarrow e_j, j + 1 \rightarrow j$	
endWrite(p_o)		*true*, wenn erfolgreich

Tabelle 9.2: Pipeline-Schnittstelle von Seqan (Auszug)

per, ...) und mit der Klasse des Senders spezialisiert werden können. Pipes, die von mehreren Pipes lesen, werden mit einer Adapterklasse spezialisiert, die selbst mit bspw. 2 oder 5 Sendertypen spezialisiert wird. So entsteht basierend auf Templates eine Verkettung mehrerer Pipes zu einer Pipeline, die vom Compiler gut optimiert werden kann. Die Schnittstelle ähnelt der STL-Schnittstelle der Queue-Klasse und ist als Ausschnitt in Tabelle 9.2 dargestellt. Pools besitzen dieselbe Schnittstelle wie Pipes, werden aber nicht mit einem Sender sondern mit ihrem Zeichen- und Größentyp und im Fall der Sorter oder Mapper u.a. mit Vergleichs- bzw. Abbildungsoperatoren spezialisiert. Sie können u.a. so konfiguriert werden, dass die Eingabedaten nicht auf die Festplatte geschrieben, sondern im Speicher sortiert oder zwischenspeichert werden. Die Funktionen beginRead/endRead und beginWrite/endWrite ermöglichen das mehrfache Lesen oder Scheiben von Daten.

Kapitel 10

Experimente

In diesem Kapitel sollen experimentelle Ergebnisse vorgestellt werden, die es erlauben, die Qualität der implementierten Algorithmen einzuschätzen. Untersucht wurden die in den Kapiteln 6 bis 8 beschriebenen Algorithmen zur internen und externen Konstruktion des Index mit Suffix-Array bzw. mit zusätzlicher erweiterter LCP-Tabelle und den zugehörigen Suchalgorithmen.

Die Tests wurden durchgeführt auf einer x86-Architektur mit einem 1,8 GHz Prozessor AMD Athlon XP, 1 GB Primärspeicher, einer 160 GB Festplatte Samsung SP1614N und dem Betriebssystem SuSE Linux 9.3. Kompiliert wurde mit g++ Version 4.0.2 (Optimierungsstufe -O3).

10.1 Interne Algorithmen

Tabelle 10.1 zeigt die internen Algorithmen, die näher untersucht und verglichen wurden. Die Algorithmen Skew3, Skew7, KA, MM und LS sind Suffix-Array-Algorithmen zur Erstellung eines einfachen Index (Index_{SA}). LCP und LCP_{IP} jeweils in Verbindung mit LCPE sind Algorithmen zur Konstruktion des erweiterten Index ($\text{Index}_{\text{LCPE}}$). Find_{SA} und $\text{Find}_{\text{LCPE}}$ sind die Suchalgorithmen des Index mit Suffix-Array bzw. Suffix-Array und erweiterter[1] LCP-Tabelle. Gemessen wurden die CPU-Laufzeit und der temporär für die Konstruktionen benötigte Speicher.

Skew3 ist eine Implementierung des Skew-Algorithmus von Kärkkäinen und Sanders [13] mit den Modifikationen aus Kapitel 6.1. Skew7 ist eine Erweiterung des Skew3 um Difference Covers mit $D = \{1, 2, 4\}$, $v = 7$. Zum Vergleich wurden die Algorithmen MM von Manber und Myers [21], LS von Larsson und Sadakane [18] sowie KA von Ko und Aluru [17] herangezogen. MM hat eine Laufzeit von $\mathcal{O}(n \log n)$ und ist ein Doubling-Algorithmus, welcher die Länge der Präfixe, nach dem das Suffix-Array sortiert ist, schrittweise verdoppelt. Für

[1] Die erweiterte LCP-Tabelle ist ein serialisierter LCP-Intervallbaum (Kapitel 8.3, [21])

die Tests wurde die Implementierung von McIlroy und McIlroy[2] verwendet. LS ist ebenfalls ein $\mathcal{O}(n \log n)$-Doubling-Algorithmus, der in der Praxis aber deutlich schneller ist als MM. KA ist ein $\mathcal{O}(n)$-Algorithmus, der effizient von Lee und Park [19] implementiert wurde und zur Klasse der rekursiven Suffix-Array-Algorithmen gehört, die ähnlich dem Skew-Algorithmus einen Teil der Suffixe rekursiv sortieren und mit dem Rest vereinen. Auf die einzelnen Funktionsweisen und Komplexitäten der Algorithmen MM, LS, KA und des einfachen Skew-Algorithmus KS [13] im Vergleich wird u.a. in den Arbeiten [19], [24] und [25] näher eingegangen.

LCP ist die direkte Implementierung des $\mathcal{O}(n)$-LCP-Algorithmus von Kasai [15]. LCP_{IP} funktioniert ähnlich und hat ebenfalls eine Laufzeit von $\mathcal{O}(n)$, benötigt aber für die Konstruktion keinen zusätzlichen Speicher (Kapitel 7.2). LCPE ist ein $\mathcal{O}(n)$-Algorithmus zur direkten Konstruktion der erweiterten LCP-Tabelle (LCP-Intervallbaum).

Für die Suchalgorithmen wurden zu den Testdaten s verschiedene Mengen $T_{s,m}$ mit $m \in \{1, 2, \ldots, 200, 500\}$ von jeweils 10.000 in s vorkommenden Substrings generiert. Die Strings in $T_{s,m}$ haben jeweils die Länge m und beginnen in s an zufälligen Positionen. Für Strings mit $|\Sigma| > 5$, wurden nur Substrings verwendet, die an Wortgrenzen beginnen und möglichst die Länge m haben. Die Strings aus $T_{s,m}$ wurden nacheinander gesucht und diese Suche für verschiedene m (Tabelle 10.6, linke Spalte) ausgeführt.

Skew3	Skew-Algorithmus mit Difference Cover $D = \{1, 2\}$, $v = 3$ (Kap. 6.1)
Skew7	Skew-Algorithmus mit Difference Cover $D = \{1, 2, 4\}$, $v = 7$ (Kap. 6.3)
KA	Suffix-Array-Algorithmus von Ko und Aluru [17]
LS	Suffix-Array-Algorithmus von Larsson und Sadakane [18]
MM	Suffix-Array-Algorithmus von Manber und Myers [21]
LCP	LCP-Algortihmus von Kasai (Kapitel 7.1)
$\mathbf{LCP_{IP}}$	Verbesserter LCP-Algortihmus (Kapitel 7.2)
LCPE	Konstruktion der Erweiterten LCP-Tabelle (Kapitel 8.3)
$\mathbf{Find_{SA}}$	Substring-Suche mit Suffix-Array
$\mathbf{Find_{LCPE}}$	Substring-Suche mit Suffix-Array und erweiterter LCP-Tabelle

Tabelle 10.1: Untersuchte interne Algorithmen (in dieser Arbeit implementierte sind hervorgehoben)

[2] `http://www.cs.dartmouth.edu/~doug/sarray/`

10.2 Testdaten

Zum Testen der internen Algorithmen wurden Texte und DNA-Sequenzen des Korpus von Manzini[3] und Ferragina und des Canterbury corpus[4] sowie Chromosom 19 des menschlichen Genoms[5] und der deutschsprachige Bibeltext[6] verwendet. Außerdem wurden synthetische pseudozufällige Strings mit den Alphabetgrößen 5 und 95 sowie der triviale String mit Alphabetgröße 1 getestet. Die Alphabete der Zufallsstrings sind DNA-Sequenzen und deutschsprachigen Texten entsprechend gewählt. Die Häufigkeit und die Positionen der Buchstaben sind allerdings gleichverteilt. Die Eigenschaften der Datensätze sind in Tabelle 10.2 aufgeführt. $\text{lcp}_{\max}$ und $\overline{\text{lcp}}$ sind der maximale bzw. über aufeinanderfolgende Suffixe gemittelte lcp-Wert und $|\Sigma|$ die Alphabetgröße.

	n	$\text{lcp}_{\max}$	$\overline{\text{lcp}}$	$\lvert\Sigma\rvert$	Beschreibung
world192	2.473.400	559	23	94	CIA World Factbook
bibel	4.272.089	286	12	95	Luther-Bibel
E.coli	4.638.690	2.815	17	4	Escherichia-coli-Genom
aaa	33.554.432	33.554.431	16.777.216	1	Trivialtext $s = \text{aaa}\ldots$
random5	33.554.432	20	10	5	Zufallsfolge DNA5
random95	33.554.432	7	3	95	Zufallsfolge Text
chr22	34.553.758	199.999	1.979	5	Human-Chromosom 22
howto	39.422.105	70.720	268	197	Linux Howto Texte
chr19	63.790.860	7.999.999	501.657	5	Human-Chromosom 19
jdk13c	69.728.899	37.334	679	113	JDK 1.3 Dokumentation
etext99	105.277.340	286.352	1.109	146	Project Gutenberg Texte
sprot34	109.617.186	7.373	89	66	SwissProt-Datenbank
linux-2.4.5	116.254.720	136.035	479	256	Linux 2.4.5 Quelltexte
rfc	116.421.901	3.445	93	120	IETF RFC Texte

Tabelle 10.2: Testdaten für die internen Algorithmen

10.3 Ergebnisse

Laufzeit

Tabelle 10.3 zeigt, dass Skew7 für alle nicht-synthetischen Datensätze etwa 20% weniger Zeit benötigt als Skew3. Auf den Zufallsstrings und dem Trivialstrings erreicht Skew3 eine bessere Laufzeit. Unter den Linearzeitalgorithmen kann

[3] `http://www.mfn.unipmn.it/~manzini/lightweight/corpus/`
[4] `http://www.cosc.canterbury.ac.nz/corpus/`
[5] Homo sapiens, build 33, 14. April 2003, `http://ncbi.nih.gov`
[6] Neues und Altes Testament nach Luther, `http://www.onlinebible.org`

	world	bibel	Ecoli	aaa	rn5	rn95	chr22	howto	chr19	jdk	etext	sprot	linux	rfc
Skew3	8	15	17	16	114	119	178	218	334	340	758	680	692	720
Skew7	7	13	14	20	116	147	141	179	259	281	597	549	557	580
KA	5	10	9	5	88	128	78	104	139	165	355	330	305	323
MM	20	34	49	95	176	145	876	838	2.140	1.249	856	1.959	1.758	1.848
LS	2	3	3	26	41	38	40	59	82	163	251	234	200	250
LCP	1	2	2	1	36	37	33	29	60	44				
LCP_{IP}	1	3	3	1	42	43	40	40	78	70	140	133	127	139
LCPE	0	0	0	1	1	1	1	1	2	2	4	4	5	5

Tabelle 10.3: CPU-Laufzeit in Sekunden

	world	bibel	Ecoli	aaa	rn5	rn95	chr22	howto	chr19	jdk	etext	sprot	linux	rfc
Skew3	19	33	35	256	207	174	264	301	487	532	803	836	887	888
Skew7	9	16	18	128	128	165	132	150	243	266	402	418	443	444
KA	19	33	35	128	256	256	264	301	517	532	803	836	887	888
MM	19	33	35	256	256	256	264	301	487	532	803	836	887	888
LS	9	16	18	128	128	128	132	150	243	266	402	418	443	444
LCP	9	16	18	128	128	128	132	150	243	266				
LCP_{IP}	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LCPE	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Tabelle 10.4: Temporärer Speicherbedarf in MB

	world	bibel	Ecoli	aaa	rn5	rn95	chr22	howto	chr19	jdk	etext	sprot	linux	rfc
Skew3	6	6	8	16	3	2	12	11	15	10	12	9	11	8
Skew7	4	3	4	9	2	2	7	6	9	6	7	5	7	5

Tabelle 10.5: Maximale Rekursionstiefe

m	world	bibel	Ecoli	aaa	rn5	rn95	chr22	howto	chr19	jdk	etext	sprot	linux	rfc
1	0,67	0,70	0,40	0,64	1,11	3,15	0,52	1,13	0,58	1,37	1,23	1,32	1,50	1,37
2	1,63	1,27	0,67	0,71	2,70	5,49	0,99	3,58	1,06	3,72	3,77	5,41	5,29	4,72
3	2,10	1,81	0,91	0,78	3,97	5,49	1,38	4,53	1,54	3,86	5,34	6,10	5,93	5,78
4	2,21	2,27	2,14	0,95	5,90	5,37	3,77	4,77	4,16	3,88	6,38	6,18	6,17	6,46
5	2,24	2,51	3,33	1,09	6,42	5,20	5,13	4,87	5,40	4,30	6,65	6,18	6,26	6,58
10	2,24	2,83	3,69	1,48	6,42	5,15	5,69	4,93	6,03	4,95	6,88	6,04	6,62	6,76
20	2,32	2,85	3,81	1,97	6,48	4,99	5,73	4,96	6,18	5,12	6,89	6,42	6,81	6,76
50	2,41	2,85	4,00	3,32	6,68	4,98	5,97	4,96	6,40	5,43	6,89	7,02	7,00	6,80
100	2,48	2,98	4,28	5,51	6,98	5,24	6,32	5,13	6,88	6,11	7,07	7,24	7,22	7,00
200	2,75	3,30	4,80	9,62	7,51	5,74	6,92	5,60	7,63	6,88	7,59	7,71	7,83	7,44
500	3,67	4,40	6,27	21,25	8,90	7,08	8,45	6,96	9,99	8,88	9,01	9,24	9,23	8,75

Tabelle 10.6: Durchschnittliche CPU-Laufzeit der Substring-Suche im Suffix-Array mit verschiedenen Suchlängen in Mikrosekunden

m	world	bibel	Ecoli	aaa	rn5	rn95	chr22	howto	chr19	jdk	etext	sprot	linux	rfc
1	1,89	1,73	1,32	1,45	1,52	2,79	1,47	2,20	1,46	2,39	2,22	2,11	2,64	2,39
2	2,84	2,40	1,38	1,47	1,76	6,28	1,66	4,31	1,66	4,00	3,89	5,14	5,62	4,85
3	3,50	2,84	1,80	1,47	2,50	6,82	2,11	5,77	2,13	4,35	5,78	6,72	6,63	6,23
4	3,73	3,50	2,40	1,45	5,40	6,82	3,49	6,21	3,59	4,42	7,14	7,13	7,06	7,00
5	3,76	3,88	3,98	1,45	6,54	6,66	5,56	6,47	5,51	4,92	7,66	7,16	7,31	7,22
10	3,76	4,61	5,58	1,48	7,98	6,61	7,59	6,97	7,61	5,94	8,67	7,29	8,30	8,00
20	3,85	4,64	5,73	1,49	8,09	6,32	7,92	7,09	7,98	6,40	8,92	7,97	8,68	8,47
50	3,90	4,64	5,93	1,57	8,29	6,20	8,14	7,09	8,24	6,64	8,92	8,73	8,82	8,59
100	3,91	4,62	6,22	1,68	8,60	6,49	8,48	7,14	8,56	7,06	8,94	8,73	8,95	8,74
200	4,08	4,88	6,67	1,93	9,13	7,02	9,01	7,54	9,10	7,72	9,32	8,72	9,42	9,07
500	4,72	5,82	7,93	2,68	10,50	8,38	10,39	8,72	10,43	9,06	10,41	9,54	10,54	10,19

Tabelle 10.7: Durchschnittliche CPU-Laufzeit der Substring-Suche im Suffix-Array mit LCPE-Tabelle und verschiedenen Suchlängen in Mikrosekunden

KA in allen Fällen die beste Laufzeit erzielen und ist dabei ungefähr doppelt so schnell wie Skew3. LS erzielt mit Ausnahme des Trivialstrings für alle Datensätze und unter allen Algorithmen die beste Laufzeit. MM ist teilweise bis zu viermal langsamer als Skew3. Er benötigt für DNA-Sequenzen deutlich mehr Zeit als für gleichgroße Zufallsstrings oder Texte.

Die Laufzeiten von LCP, LCP_{IP} und LCPE verhalten sich annähernd linear zu n. Im Durchschnitt benötigt LCP_{IP} 130% der Zeit von LCP. Die Tests für LCP mit $n > 10^8$ mussten nach 2 Stunden Laufzeit leider erfolglos abgebrochen werden, da der Primärspeicher des Testsystems nicht ausreichte.

LCPE erzielt sehr gute Laufzeiten und benötigt zur Konstruktion der erweiterten LCP-Tabelle nur ein Dreisigstel der Zeit von LCP_{IP} zur Konstruktion der LCP-Tabelle.

Die Substring-Suche im Index mit Suffix-Array benötigt im Durchschnitt mit fast allen Datensätzen und Suchlängen etwa 1-2 Mikrosekunden weniger Zeit als die Substring-Suche im Index mit Suffix-Array und erweiterter LCP-Tabelle. Lediglich der Trivialstring kann von $\text{Find}_{\text{LCPE}}$ schneller durchsucht werden als von Find_{SA}.

Speicherbedarf

Skew3, KA und MM haben für fast alle Datensätze einen temporären Speicherbedarf von $8n$ Byte. Skew7 und LS hingegen benötigen mit $4n$ Byte fast überall nur halb so viel Speicher. Ausnahmen gibt es für die Skew-Algorithmen mit den Zufallstrings. Skew3 benötigt für die beiden Strings etwa $6,5n$ bzw. $5,4n$ Byte und Skew7 für random95 etwa $5n$ Byte Speicher. Weitere Ausnahmen gibt es für KA mit dem Trivialstring ($4n$ Byte) und Chromosom 19 ($\approx 8,5n$ Byte).

LCP benötigt genau $4n$ Byte und LCP_{IP} keinen zusätzlichen Speicher. LCPE hat genaugenommen einen zusätzlichen Speicherbedarf von $\lceil 4 \log_2 n \rceil$ Byte, der aber vernachlässigt werden kann.

10.4 Auswertung

10.4.1 Skew-Algorithmen

Laufzeit

Die Erweiterung des einfachen Skew-Algorithmus Skew3 um Difference Covers bringt eine Laufzeitverbesserung von durchschnittlich 30%. Die Ursache liegt dabei im kleineren Rekursionsfaktor $\lambda_{\text{Skew7}} = \frac{3}{7} < \frac{2}{3} = \lambda_{\text{Skew3}}$, durch den die Problemgrößen n_i und die von n_i beschränkten Alphabetgrößen beim rekursiven Abstieg schneller klein werden. Außerdem ist die maximale Rekursionstiefe r in Skew7 wegen der höheren Tupelgröße v im Allgemeinen geringer. Die untere Schranke des Satzes 6.4.2 liefert in der Praxis gleichzeitig auch eine gute Näherung des eigentlichen Wertes r.

Eine allgemeine theoretische Analyse des average-case der internen Skew-Algorithmen ist schwierig, da Laufzeit und Speicherbedarf der Algorithmen im konkreten Fall von mehreren orthogonalen Faktoren abhängen:

- Difference Cover v, D
- Art des Wachstums der Folge $|\Sigma_i|$ der Alphabetgrößen
- Rekursionstiefe r

Alphabetwachstum
Die Laufzeit der Skew-Algorithmen hängt stark von den Alphabetgrößen und den davon abhängigen Laufzeiten von Radixpass ab. Mit den meisten Datensätzen wachsen die Alphabetgrößen zunächst wie $|\Sigma|^{v^{(i-1)}}$ und fallen dann ab wie $\lambda^{i-1}n$. Für die Geschwindigkeit des Anstiegs ist dabei die Vielfalt der v-Tupel entscheidend. DNA-Sequenzen, Texte und Zufallsstrings bspw. lassen die Alphabete sehr schnell wachsen. Für den Trivialstring aaa wachsen die Alphabete so langsam wie nur möglich, nämlich wie $|D|^{(i-1)}$.

Rekursionstiefe
Für die Rekursionstiefe ist in erster Linie die Länge und Häufigkeit sich wiederholender Teilsequenzen entscheidend. DNA-Sequenzen enthalten im Bereich der Zentromere oder Telomere kurze hochrepetitive Teilsequenzen, die im String meist als Folge von 'N'-Zeichen dargestellt sind. Durch die Länge dieser Folge, wird der Wert $\text{lcp}_{\max}$ und nach Satz 6.4.2 auch r bestimmt. In Zufallsstrings ist die Länge und Häufigkeit von Wiederholungen gering und die Namen werden in der Rekursion schnell eindeutig. Tabelle 10.5 zeigt zu den Testdaten die gemessenen maximalen Rekursionstiefen r für Skew3 und Skew7.

Welchen Einfluß die Testdatensätze auf die Skew-Algorithmen haben, zeigt Tabelle 10.8. Die Laufzeit nimmt nach rechts unten hin zu. Ein worst-case-Datensatz muss also möglichst zufällig und trotzdem häufige oder lange Wiederholungen besitzen. DNA-Sequenzen erfüllen beide Eigenschaften.

	Σ-Wachstum gering	Σ-Wachstum hoch
r gering		rn5, rn95
r hoch	aaa	DNA, Texte

Tabelle 10.8: Verhalten der Skew-Algorithmen

Die einzigen Datensätze für die Skew3 schneller ist als Skew7 sind die Zufallsstrings und der Trivialstring. Hier kann Skew7 nicht davon profitieren, dass für Skew3 die Alphabetgrößen langsamer abfallen, denn in diesen Fällen bleibt das Alphabet für Skew3 klein, entweder weil $|\Sigma| = 1$ oder r sehr klein ist. Würde man die Laufzeiten von Skew3 und Skew7 bei gleichem Alphabet und Stringlänge n ohne den rekursiven Abstieg vergleichen, wäre Skew7 langsamer, da im ersten Schritt mit Radixsort eine Menge von λn v-Tupeln sortiert werden muss. Dabei werden die innersten Schleifen von Radixsort mindestens $\lambda nv = \frac{|D|}{v} nv = |D|n$ mal durchlaufen. Desweiteren benötigt das Einfügen und Entfernen der Namen in der Priority-Queue des Skew7 mehr Zeit als das direkte Vergleichen bei Skew3. Im average-case wird dieser Mehraufwand für Skew7 durch schneller fallende n_i und kleinere Rekursionstiefen kompensiert.

Speicherbedarf

Die Messungen zum Speicherbedarf der Skew-Algorithmen decken sich mit den in Folgerung 6.4.1 gemachten Abschätzungen. Der theoretische Maximalspeicherbedarf von $8n$ Byte wird von Skew3 in fast allen Fällen erreicht. Die schärferen oberen Schranken für Skew7 aus den Sätzen A.2.1 und A.2.2 des Anhangs gemessen in Worten können ebenfalls in den Experimenten bestätigt werden. Skew7 benötigt im Allgemeinen etwa $4n$ Byte bzw. für random95 $165 \cdot 2^{20} \approx \frac{9}{7} \cdot 4n$ Byte temporären Speicher.

Die einzige Ausnahme besteht bei Datensätzen mit kleinen maximalen Rekursionstiefen r in Skew3, wie random5 ($r = 3$) und random95 ($r = 2$). Dort sind die Namen der Triplets wegen der Gleichverteilung der Buchstaben schon sehr früh eindeutig. Die Alphabetgrößen nähern sich nicht an n_i an und der zusätzliche Speicherbedarf bleibt unterhalb von $8n$ bei etwa $6,4n$ (random5) bzw. $5,6n$ (random95) Byte.

10.4.2 LCP-Algorithmen

Vor dem Ausführen der LCP-Algorithmen wird $9n$ Byte Speicher für die Felder s, SA und LCP benötigt. Der Algorithmus LCP benötigt temporär $4n$ Byte insgesamt also $13n$ Byte Speicher. Für $n > 10^8$ konnten nicht mehr alle Felder im Primärspeicher gehalten werden. Das nicht-ortslokale Zugriffsverhalten verursacht ein Ein- und Auslagern von Speicherseiten durch das Betriebssy-

stem während der gesamten Ausführungszeit. LCP wird dadurch unzumutbar langsam.

Die gemessenen Laufzeiten des Algorithmus LCP_{IP} sind im Mittel etwa ein Drittel höher als die von LCP. Die auffällig geringe Laufzeit für *'aaa'* ist auf die Ortslokalität der Zugriffe auf die Felder SA und LCP und das Wirken der Speicher-Caches zurückzuführen.

Im Algorithmus LCPE erfolgen alle Zugriffe auf die Felder LCP und LCPE sequentiell. Sie können, wie Tabelle 10.3 zeigt, gut von den Speicher-Caches beschleunigt werden.

10.4.3 Substring-Suche

Die Ergebnisse in den Tabellen 10.6 und 10.7 zeigen deutlich, dass $\text{Find}_{\text{LCPE}}$ für nicht-synthetische Datensätze trotz der kleineren Laufzeitkomplexität in der Praxis langsamer ist als Find_{SA}. Die Laufzeit, die $\text{Find}_{\text{LCPE}}$ durch das Überspringen redundanter Zeichenvergleiche einspart, ist also im Mittel kleiner als die hinzukommende Laufzeit beim Traverserieren der erweiterten LCP-Tabelle. Der Quelltext von $\text{Find}_{\text{LCPE}}$ ist komplexer und erfordert mehr bedingte Sprünge im Maschinencode als der von Find_{SA}. Sieht man von den Laufzeiten der Zeichenvergleiche ab, kann Find_{SA} schneller vom Prozessor ausgeführt werden.

Die beiden Binärsuchalgorithmen unterscheiden sich zwar in der Wahl des mittleren Elements, für die folgende Betrachtung soll aber angenommen werden, dass beide zu einem Substring t die gleiche Intervallschachtelung vornehmen. Es seien k_1 und k_2 die in Find_{SA} durch erfolgreiche Zeichenvergleiche ermittelten lcp-Werte von t und der oberen bzw. unteren Intervallgrenze. Die Zahl der Zeichenvergleiche, die $\text{Find}_{\text{LCPE}}$ gegenüber Find_{SA} einsparen kann (Tabelle 8.1), beträgt in der l-Suche in jedem Schritt dann höchstens:

$$\max\{k_1, k_2\} - \min\{k_1, k_2\} = |k_1 - k_2|.$$

Es sei l_t das Ergebnis der l-Suche nach t. Die Absolutdifferenz von k_1 und k_2 wird im Allgemeinen größer, je dichter sich nur eine der beiden Intervallgrenzen an l_t annähert. Eine theoretische Modellierung des Problems ist schwierig und hängt ab von l_t und der Funktion $f : i \in [0..n) \to \text{lcp}(t, s_i)$. Die Aufsummierung der Werte $|k_1 - k_2|$ aus jedem Schritt ergibt die Anzahl der Zeichenvergleiche, die insgesamt von $\text{Find}_{\text{LCPE}}$ in der Suche eingespart werden können.

Für fast alle Testdatensätze bleiben die Werte $|k_1 - k_2|$ so klein, dass die zusätzlichen Vergleiche in Find_{SA} effizienter ausgeführt werden können als die Fallunterscheidung und der Zugriff auf die erweiterte LCP-Tabelle in $\text{Find}_{\text{LCPE}}$. Hinzu kommt, dass die Zugriffe auf s und t in jedem Schritt ortslokal sind und vom Prozessor-Cache beschleunigt werden. Ein zweiter Zeichenvergleich kostet im Allgemeinen weniger Zeit als der erste.

Der einzige Datensatz, bei dem Find_{SA} durchschnittlich mehr Zeit für die Suche benötigt, ist der Trivialstring. Die l-Suche liefert zu einem Substring t der Länge m die Position $l = m - 1$ im Suffix-Array. Während der Suche gilt ab dem zweiten Schritt $k_2 = m$ und für fast alle Schritte $k_1 = 0$. Erst innerhalb der letzten $\lceil \log_2 m \rceil$ Schritte wächst auch k_1. Find_{SA} führt also in fast allen Schritten m Zeichenvergleiche durch. Der erste Schritt in $\text{Find}_{\text{LCPE}}$ verläuft analog zu Find_{SA}. In allen weiteren Schritten muss jeweils höchstens noch ein Zeichen verglichen werden. Das erklärt, warum Find_{SA} für 'aaa' auffällig höhere Suchlaufzeiten hat als $\text{Find}_{\text{LCPE}}$.

m	aaa4	aaa32	aaa64	aaa110
1	0,42	0,76	0,75	0,58
2	0,44	0,77	0,82	0,61
4	0,51	0,85	0,89	0,71
8	0,64	1,08	1,05	0,97
16	0,84	1,35	1,38	1,22
32	1,19	1,80	1,86	1,66
64	1,82	2,58	2,62	2,45
128	3,00	3,97	4,07	3,94
256	5,10	6,50	6,73	6,68
512	9,15	11,46	11,99	12,09
1024	16,62	20,75	21,87	22,29
2048	30,95	38,95	41,44	41,90
4096	56,71	72,24	77,25	78,42
8192	102,91	133,87	143,14	147,33

m	aaa4	aaa32	aaa64	aaa110
1	1,25	1,42	1,47	1,52
2	1,27	1,43	1,49	1,53
4	1,27	1,44	1,49	1,53
8	1,28	1,44	1,50	1,54
16	1,28	1,45	1,50	1,55
32	1,30	1,46	1,52	1,56
64	1,33	1,50	1,55	1,60
128	1,41	1,57	1,63	1,68
256	1,56	1,72	1,78	1,83
512	1,89	2,05	2,10	2,15
1024	2,18	2,34	2,40	2,44
2048	2,32	2,48	2,54	2,58
4096	2,40	2,55	2,61	2,65
8192	2,43	2,59	2,64	2,69

Tabelle 10.9: Durchschnittliche CPU-Laufzeit von Find_{SA} (links) und $\text{Find}_{\text{LCPE}}$ (rechts) für Trivialstrings ($|\text{aaa}X| = X \cdot 2^{20}$) in Mikrosekunden

Tabelle 10.9 zeigt die Laufzeiten der beiden Suchalgorithmen im Vergleich für verschieden lange Trivialstrings und Suchlängen. In diesen Tests wurde die Laufzeit von 10.000 Einzelsuchen nach Substrings der zufälligen Länge zwischen 1 und m gemessen und gemittelt. Gut zu erkennen ist, dass die sich im worstcase die Laufzeit von Find_{SA} proportional m zu verhält. $\text{Find}_{\text{LCPE}}$ wird von m dagegen nur wenig beeinflußt.

10.5 Externe Algorithmen

Getestet wurden die zu den in Kapitel 10.1 beschriebenen internen Algorithmen gehörenden externen Implementierungen Skew3, Skew7, LCP und LCPE. Hinzu kommt Skew7c, eine um Bitkompression erweiterte Variante des Skew7. Zum Vergleich wurde die Implementierung[7] des externen Skew-Algorithmus von Dementiev et al. [8] herangezogen. Er benutzt die Bibliothek STXXL[8], ei-

[7] `http://i10www.ira.uka.de/dementiev/esuffix/docu/install.html`

[8] `http://stxxl.sourceforge.net/`

ne Software-Bibliothek die u.a. Algorithmen und Datenstrukturen der STL[9] für die Benutzung von Sekundärspeicher erweitert.

Skew3 funktioniert im Wesentlichen wie DC3. Die beiden Algorithmen unterscheiden sich in der Art der Verschmelzung in Schritt 3 des Skew-Algorithmus und im zugrundeliegenden Pipelining-System. In den Implementierungen dieser Arbeit wurden außerdem nicht benötigte Sorter-Module durch Mapper-Module ersetzt.

Die Skew-Algorithmen und LCP wurden entsprechend der vorgestellten Aquädukte als Pipelining-Algorithmen implementiert. Für die Testdaten gilt $n \leq 2^{32}$ und ein Wort hat die Größe von 4 Byte. Die Eingabezeichen sind jeweils ein Byte lang, so dass $\sigma = \frac{1}{4}$ gilt. Standardmäßig wurde jedem Pool-Modul Speicher der Größe 64 MB zugeteilt, es gilt also $M = \frac{64 \text{ MB}}{4 \text{ Byte}} = 2^{24}$. Dieser wird nur während der Schreib- oder Lesephase benötigt und angefordert. Der Puffer des externen LCP-Algorithmus hat eine Größe von 512 MB also $W = \frac{512 \text{ MB}}{4 \text{ Byte}} = 2^{27}$.

Bei LCPE, Find_{SA} und $\text{Find}_{\text{LCPE}}$ handelt es sich um dieselben generischen Implementationen wie in Kapitel 10.1 getestet, sie wurde für die externen Tests lediglich mit der externen String-Klasse spezialisert. Die externen Strings zum Speichern der SA- und LCPE-Tabellen in Find_{SA} und $\text{Find}_{\text{LCPE}}$ wurden auf die Blockgröße von $B = 1024$ Worten angepasst. Der externe String der erweiterten LCP-Tabelle wurde für die permanente Pufferung der ersten 1024 Blöcke parametrisiert. Auf diese Art und Weise werden die Elemente der oberen 20 Ebenen im LCP-Intervallbaum nach dem erstmaligen Zugriff zwischengespeichert.

Die Tests wurden mit nur einer Festplatte und einer Blockgröße von 4 KB durchgeführt. Es gilt $P = 1$ und $B = \frac{4 \text{ KB}}{4 \text{ Byte}} = 1024$.

Skew3	externer Skew3 (Kapitel 6.5)
Skew7	externer Skew7
Skew7c	externer Skew7 mit Bitkompression
DC3	externer Skew-Algorithmus von Dementiev et al. [8]
LCP	externer LCP-Algortihmus (Kapitel 7.3)
LCPE	externer LCPE-Algorithmus (Kapitel 8.4)
$\textbf{Find}_{\textbf{SA}}$	Substring-Suche mit Suffix-Array
$\textbf{Find}_{\textbf{LCPE}}$	Substring-Suche mit Suffix-Array und erweiterter LCP-Tabelle

Tabelle 10.10: Untersuchte externe Algorithmen

[9] C++ Standard Template Library `http://www.sgi.com/tech/stl/`

10.6 Testdaten

Die Experimente mit den externen Algorithmen wurden auf der menschlichen DNA-Sequenz des Korpus[10] von Dementiev et al. [8] durchgeführt. Der Datensatz entsteht aus der Humangenomsequenz des UCSC[11], Stand Mai 2005, durch die Projektion der Zeichen {a,c,g,t,n,A,C,G,T,N} auf die Zeichen {A,C,G,T,N} und der anschließenden ordnungserhaltenden Bijektion auf die Menge [0..5). Die ermittelten Eigenschaften zeigt Tabelle 10.11.

	n	$\text{lcp}_{\max}$	$\overline{\text{lcp}}$	$\lvert\Sigma\rvert$	Beschreibung
genome	3.070.128.193	21.999.999	454.111	5	Humangenom

Tabelle 10.11: Testdatum für die externen Algorithmen

	Laufzeit	Speicherbedarf	I/O-Zugriffe	I/O-Bedarf	Rek.tiefe
Skew3	15h 17m 53s	423 MB	184 Mio.	63,5 GB	16
Skew7	12h 01m 25s	446 MB	124 Mio.	71,5 GB	9
Skew7c	10h 41m 21s	446 MB	119 Mio.	65,4 GB	9
DC3	14h 34m 26s	256 MB	2,8 Mio[12]	80,0 GB	
LCP	14h 56m 57s	576 MB	86 Mio.	57,2 GB	6
LCPE	23m 20s	33 MB	12 Mio.	0,0 GB	

Tabelle 10.12: Aufwand der externen Algorithmen für 'genome'

10.7 Ergebnisse

Die Ergebnisse in Tabelle 10.12 zeigen, dass Skew7 über 20% und Skew7c über 30% weniger Zeit zur Berechnung des Suffix-Array benötigen als Skew3. Skew3 benötigt für das Humangenom etwa 45 Minuten also 5% mehr Zeit als DC3.

LCP benötigt für das Genom etwa ebenso viel Zeit wie Skew3 und LCPE etwa ein Vierzigstel der Laufzeit von LCP.

Skew3 benötigt etwa $0,060n$ I/O-Zugriffe. Skew7 und Skew7c benötigen mit jeweils $0,040n$ etwa $\frac{2}{3}$ der I/O-Zugriffe von Skew3. Für LCP hängen Laufzeit und I/O-Zugriffe nicht nur von n sondern auch von W ab, der Größe des Pufferspeichers in Worten. In diesem Fall ist die Zahl der I/O-Zugriffe etwa $0,0012 \cdot \frac{n^2}{W}$.

[10] `ftp://ftp.mpi-sb.mpg.de/pub/outgoing/sanders/`

[11] `http://genome.ucsc.edu/downloads.html`

[12] DC3 benutzt eine Blockgröße von $B = 256$ KB. Mit einer Blockgröße von $B = 4$ KB würde DC3 also geschätzte 179 Mio. Zugriffe benötigen.

m	Find_{SA}	I/O-Zugriffe	$\text{Find}_{\text{LCPE}}$	I/O-Zugriffe
1	0,38	0,02	0,21	0,01
2	1,11	0,08	0,60	0,04
3	3,33	6,09	2,78	0,88
4	9,69	23,17	7,18	10,19
5	28,45	28,59	23,39	16,23
10	171,49	31,00	165,03	28,00
20	174,84	30,95	216,93	30,85
50	174,20	30,90	220,07	31,14
100	173,61	30,87	221,22	31,24
200	173,68	30,78	220,98	31,23
500	173,98	30,53	220,98	31,12

Tabelle 10.13: Durchschnittliche Laufzeit in Mikrosekunden und Anzahl der I/O-Zugriffe der externen Substring-Suche mit verschiedenen Suchlängen

Für die externe Substring-Suche im Humangenom benötigen Find_{SA} und $\text{Find}_{\text{LCPE}}$ ab einer Suchwortlänge von 10 Zeichen etwa 170 bzw. 220 Millisekunden, das entspricht einer Laufzeit von etwa $7,8 \cdot \log n$ bzw. $10 \cdot \log n$ Millisekunden. Die Zahl der I/O-Zugriffe beträgt für beide Algorithmen im Mittel höchstens 32 also etwa $\log_2 n$. $\text{Find}_{\text{LCPE}}$ ist in den Test höchstens für Suchwörter mit weniger als 10 Zeichen geringfügig schneller als Find_{SA}.

10.8 Auswertung

10.8.1 Skew- und LCP-Algorithmen

I/O-Zugriffe

In Kapitel 6.5.2 wurden Abschätzung für externe Skew-Implementationen mit perfekten Difference Covers gefunden. Für $\sigma = \frac{1}{4}$ und Skew3 und Skew7 gilt dementsprechend:

$$\begin{aligned} T^{\{1,2\}}(n) &\leq \text{sort}\left((2\tfrac{2}{3} \cdot \sigma + 10\tfrac{1}{3}) \cdot n\right) + \text{permute}\left((1\tfrac{1}{3} \cdot \sigma + 16\tfrac{2}{3}) \cdot n\right) + \text{scan}\left((2\sigma + 4) \cdot n\right) \\ T^{\{1,2,4\}}(n) &\leq \text{sort}\left(\left(6\sigma + 9\tfrac{1}{4}\right) \cdot n\right) + \text{permute}\left(\left(2\tfrac{4}{7} \cdot \sigma + 7\tfrac{13}{14}\right) \cdot n\right) + \text{scan}\left(\left(2\sigma + 1\tfrac{1}{2}\right) \cdot n\right) \end{aligned}$$

$\Rightarrow$

$$\begin{aligned} T^{\{1,2\}}(n) &\leq \text{sort}\left(11n\right) + \text{permute}\left(17n\right) + \text{scan}\left(4\tfrac{1}{2} \cdot n\right) \\ T^{\{1,2,4\}}(n) &\leq \text{sort}\left(10\tfrac{3}{4} \cdot n\right) + \text{permute}\left(8\tfrac{4}{7} \cdot n\right) + \text{scan}\left(2n\right) \end{aligned}$$

$\Rightarrow$ [13]

$$\begin{aligned} T^{\{1,2\}}(n) &\leq \text{scan}\left(60\tfrac{1}{2} \cdot n\right) = \left\lceil 60\tfrac{1}{2} \cdot \tfrac{n}{1024} \right\rceil \approx \mathbf{181,4} \text{ Mio.} \\ T^{\{1,2,4\}}(n) &\leq \text{scan}\left(40\tfrac{9}{14} \cdot n\right) = \left\lceil 40\tfrac{9}{14} \cdot \tfrac{n}{1024} \right\rceil \approx \mathbf{121,9} \text{ Mio..} \end{aligned}$$

[13] Für die I/O-Zugriffe der Pool-Module dieser Arbeit gilt: $\text{permute}(n) = 2 \cdot \text{scan}(n)$ und $\text{sort}(n) = 2 \cdot \text{scan}(n)$.

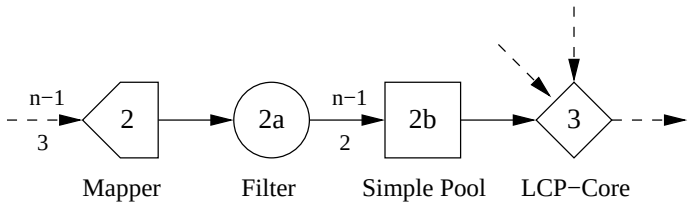


Abbildung 10.1: Optimierung im Aquädukt des externen LCP-Algorithmus

Die Abschätzung der I/O-Zugriffe des LCP-Algorithmus aus Kapitel 7.3.3 liefert:

$$\begin{aligned} T^W(n) &\leq \text{scan}\left(\left\lceil \tfrac{n}{W} - 1 \right\rceil \cdot (\sigma+3) \cdot n + (2\sigma+1) \cdot n\right) + \text{permute}(5n) \\ &\leq \text{scan}\left(5 \cdot 3\tfrac{1}{4} \cdot n + 11\tfrac{1}{2} \cdot n\right) \\ &\leq \text{scan}\left(27\tfrac{3}{4} \cdot n\right) = \left\lceil 27\tfrac{3}{4} \cdot \tfrac{n}{1024} \right\rceil \quad \approx \mathbf{83,2} \text{ Mio.} \end{aligned}$$

Vergleicht man die theoretischen Werte mit den gemessenen in Tabelle 10.12 fällt eine Differenz von bis zu 3 Mio. Zugriffen auf. Ursache dafür ist, dass in den Tests auch die Zugriffe zum Beschreiben der Ergebnistabellen SA bzw. LCP gemessen wurden. Diese sind aber absichtlich nicht in den Abschätzungen enthalten. Zieht man also von den gemessenen Werten $\text{scan}(n) = \left\lceil \frac{n}{1024} \right\rceil \approx 3,0$ Mio. Zugriffe ab, werden die theoretischen und experimentellen Werte annähernd gleich.

Bei mehr als 5 Durchläufen kann LCP optimiert werden, indem im Aquädukt zwischen den Modulen Mapper (2) und LCP-Core (3) die Module Filter und Simple Pool einfügt werden (Abb. 10.1), um die nicht benötigte zweite Komponente aus F' zu entfernen. Die Zahl der I/O-Zugriffe kann so um

$$\left\lceil \frac{n}{W} - 1 \right\rceil \cdot \text{permute}_{read,3}(n) - \left\lceil \frac{n}{W} + 1 \right\rceil \cdot \text{scan}_2(n) \hat{=} \text{scan}\left(\left\lceil \frac{n}{W} - 5 \right\rceil \cdot n\right)$$

reduziert werden.

Sekundärspeicherbedarf

Die in Kapitel 6.5.2 gefundenen Abschätzung für den temporären Sekundärspeicherbedarf der externen Skew-Algorithmen in Worten reduzieren sich zu:

$$\begin{aligned} \left(2\sigma + 4\tfrac{1}{3}\right) \cdot n &\leq S^{\{1,2\}}(n) \leq \max\left\{2\sigma + 4\tfrac{1}{3}\,,\, 5\tfrac{5}{9}\right\} \cdot n \\ \left(5\tfrac{4}{7} \cdot \sigma + 4\tfrac{6}{7}\right) \cdot n &\leq S^{\{1,2,4\}}(n) \leq \max\left\{5\tfrac{4}{7} \cdot \sigma + 4\tfrac{6}{7}\,,\, 5\tfrac{16}{49}\right\} \cdot n. \end{aligned}$$

$\Rightarrow$

$$\begin{aligned} \mathbf{55,3} \text{ GB} \hat{=} \quad 4\tfrac{5}{6} \cdot n \leq\ & S^{\{1,2\}}(n) \leq 5\tfrac{5}{9} \cdot n \hat{=} \mathbf{63,5} \text{ GB} \\ & S^{\{1,2,4\}}(n) = 6\tfrac{1}{4} \cdot n \hat{=} \mathbf{71,5} \text{ GB}. \end{aligned}$$

Die Abschätzung für den externen LCP-Algorithmus aus Kapitel 7.3.3 ergibt:

$$S^W(n) = 5(n-1) \hat{=} \mathbf{57,2} \text{ GB}.$$

LCPE benötigt während der Konstruktion keinen zusätzlichen Sekundärspeicher, sondern arbeitet direkt auf den Eingabe- und Ausgabefeldern. Die theoretischen Vorhersagen für den maximal benötigten zusätzlichen Externspeicher konnten also durch Experimente bestätigt werden.

10.8.2 Substring-Suche

Die Ergebnisse in den Tabellen 10.13 zeigen keine allgemeine Verbesserung der Laufzeit von $\text{Find}_{\text{LCPE}}$ gegenüber Find_{SA}. Die wesentlichen Gründe dafür wurden bereits in Abschnitt 10.4.3 beschrieben.

Entscheidend für die Laufzeiten der Algorithmen sind die Anzahl und Zufälligkeit der I/O-Zugriffe. Innerhalb einer Suche mit Find_{SA} liegen zwei aufeinanderfolgende Zugriffe auf das Suffix-Array räumlich weit auseinander und fallen für $n = 2^{32}$ frühestens nach etwa $\log_2 \frac{n}{B} = 22$ Schritten in denselben Block des externen Strings. $\text{Find}_{\text{LCPE}}$ verhält sich ähnlich, innerhalb einer Substring-Suche werden hier die Abstände zweier Zugriffe auf die erweiterte LCP-Tabelle fortlaufend größer und können nur zu Beginn durch den externen String gepuffert werden.

Die in beiden Suchen benötigten Zugriffe auf den externen String s können dagegen sehr gut gepuffert werden, da sie ortslokal sind und in jedem Schritt höchstens $m \leq 500$ Zeichen verglichen werden müssen. Der Zugriff auf s erfordert in jedem Schritt also nur 1-2 I/O-Zugriffe und beeinflusst die Gesamtlaufzeit für verschiedene m nicht wesentlich. Da $\text{Find}_{\text{LCPE}}$ lediglich die Zahl der Zugriffe auf s verringert, kann damit die Laufzeit nicht erheblich verbessert werden.

Die permantente Pufferung der ersten Blöcke der erweiterten LCP-Tabelle verbesserte die Laufzeit von $\text{Find}_{\text{LCPE}}$ nur für kurze Suchstrings. Da es in s für $|\Sigma| = 5$ und $m \leq 5$ nur maximal 3125 verschiedene Substrings der Länge m geben kann, werden einige der 10.000 Substrings innerhalb eines Testlaufs mindestens 3-mal gesucht und die entsprechenden Werte aus SA und LCPE können ohne zusätzliche I/O-Zugriffe gelesen werden. Das erklärt die besseren Laufzeiten von $\text{Find}_{\text{LCPE}}$ gegenüber Find_{SA}.

Auf Grund der geringen Ortslokalität der Zugriffe auf SA und LCPE sollten diese Tabellen extern besser in Form von B-Bäumen abgelegt werden. Mit ihnen könnte die Zahl der pro Suche benötigten I/O-Zugriffe auf SA und LCPE auf ein Zehntel reduziert werden. Mit einer Blockgröße von $B = 1024$ und dem Abspeichern von Teilbäumen innerhalb eines Blocks, können so $\log_2 B = 10$ Zugriffe der Binärsuche mit nur einem I/O-Zugriff ausgeführt werden.

Kapitel 11

Zusammenfassung

In dieser Arbeit wurde ein generischer Substring-Index entworfen, implementiert und analysiert. Er wurde in Seqan integriert und stellt Methoden zur Verfügung, effizient nach Vorkommen von Strings zu suchen. Er basiert auf Suffix-Arrays und LCP-Tabellen, die neben der exakten Substring-Suche in späteren Erweiterungen auch für andere Probleme eingesetzt werden können, bspw. zur Konstruktion von Enhanced Suffix-Arrays oder zur Bestimmung von maximalen oder supermaximalen Repeats [1]. In dieser Arbeit konnten außerdem bekannte Algorithmen zur Konstruktion von Suffix-Arrays und LCP-Tabellen in Theorie und Praxis verbessert werden.

In Kapitel 4 wurden zunächst bekannte Algorithmen vorgestellt zum internen und externen Sortieren von Feldern F. In Abschnitt 4.3 wurde ein Algorithmus entwickelt, der im Spezialfall, dass F permutiert werden soll, laufzeitoptimal ist. Mit nur einer Festplatte ($P = 1$) ist dieser auch optimal in der Zahl der I/O-Zugriffe. Er lässt sich sehr einfach effizient implementieren und soll bislang zum externen Permutieren eingesetzte laufzeitsuboptimale Sortierverfahren ersetzen.

Kapitel 5 stellt das Konzept des Pipelinings [14] vor. Es wurden die drei Modulklassen *Pipes*, *Pools* und *Pumps* eingeführt und Module entwickelt, mit denen externe Algorithmen einfach zusammengesetzt werden können. Unter anderem wurden dabei die Algorithmen aus Kapitel 4 als Pools implementiert. In Abschnitt 5.5 wurde erläutert, wie Komplexitätsgrößen eines Pipelining-Algorithmus, bspw. der Sekundärspeicherbedarf oder die Zahl der I/O-Zugriffe, theoretisch analysiert werden können.

Im 6. Kapitel wurde der Skew-Algorithmus [13] zur Linearzeit-Konstruktion von Suffix-Arrays vorgestellt. Er wurde um Difference Covers erweitert und sein zusätzlicher Speicherbedarf konnte durch geringfügige Modifikationen reduziert werden. Der einfache Skew-Algorithmus, Skew3, benötigt so etwa $8n$ Byte zusätzlichen Speicher. Die Difference-Cover-Erweiterung, Skew7, konnte den zusätzlichen Speicherbedarf im Allgemeinen auf $4n$ Byte halbieren und benötigt

etwa 20% weniger Zeit für die Suffix-Array-Konstruktion. In Abschnitt 6.5 wurden der Skew-Algorithmus und die Difference-Cover-Variante als asymptotisch I/O-optimale Pipelining-Algorithmen entwickelt. In der Theorie konnte gezeigt werden, dass im worst-case der externe Skew7-Algorithmus etwa ein Drittel weniger I/O-Zugriffe als Skew3 benötigt. Diese Verbesserung konnte durch die Praxis am Beispiel der DNA-Sequenz des menschlichen Genoms bestätigt werden.

In Kapitel 7 wurde ein Linearzeit-Algorithmus [15] zur Konstruktion der LCP-Tabelle beschrieben und im zweiten Abschnitt eine Verbesserung vorgestellt, die für die Konstruktion keinen zusätzlichen Speicher mehr benötigt. Der LCP-Algorithmus [15] konnte anschließend als externer Algorithmus entwickelt werden, welcher Pipelining benutzt und die LCP-Tabelle innerhalb mehrerer Durchläufe konstruiert.

Wie effizient im Suffix-Array gesucht werden kann, wurde im 8. Kapitel gezeigt. Die Benutzung einer Erweiterung der LCP-Tabelle verbessert die theoretische Laufzeit der Suche im worst-case. Es wurde gezeigt, wie diese Tabelle optimal konstruiert werden kann und der Suchalgorithmus entsprechend angepasst. In den Experimenten aus Kapitel 10 konnte für average-case Datensätze allerdings keine Laufzeitverbesserung gegenüber der Suffix-Array-Suche festgestellt werden.

In Kapitel 9 schließlich wurde der generische Substring-Index zunächst als Abstrakter Datentyp entwickelt und anschließend zusammen mit den in den vorherigen Kapiteln entwickelten Algorithmen implementiert und in die Softwarebibliothek Seqan integriert.

Die Experimente in Kapitel 10 konnten die entwickelten theoretischen Abschätzungen der Zeit-, Speicher- und I/O-Komplexitäten der Implementationen an verschiedenen praktischen Datensätzen bestätigen. Wichtigstes Ergebnis ist die Laufzeitverbesserung der externen Suffix-Array-Konstruktion des menschlichen Genoms um fast 4 Stunden auf 10,5 Stunden auf einem einfachen PC mit nur einer Festplatte.

Kapitel 12

Ausblick

Als erstes sollte das Pipelining-Modul Sorter um Forecast-Techniken erweitert werden, um beim Auslesen in der Verschmelzungsphase eine höhere Parallelität zu erreichen. Alternative, für kleine Alphabete Σ gut geeignete Sorter-Module sollten implementiert werden, wie bspw. eine externe Version des in [14] vorgestellten stabilen Distributionsort oder ein externes Radixsort. Diese könnten im externen Skew-Algorithmus in den ersten Rekursionsschritten zum Sortieren der v-Tupel in Schritt 1 und zum Erweitern der Sortierung auf die Mengen $S_i, i \notin D$ in Schritt 3 getestet und bei Erfolg eingesetzt werden. Das Sorter-Modul dieser Arbeit benötigt zum Sortieren einer Permutation mit $n = 2^{27}$ bislang noch doppelt soviel Zeit wie das Modul Mapper.

Der externe Skew-Algorithmus könnte so modifiziert werden, dass in der Rekursion ab einer parametrisierbaren Textgröße auf interne Suffix-Array-Algorithmen zurückgegriffen wird. Skew7 hat s im 5. Rekursionsschritt bereits auf ein Dreistigstel seiner Länge reduziert, für das menschliche Genom eine Länge von $n \approx 10^8$. Auf diese Weise könnten unter Benutzung des Algorithmus LS auf dem Testsystem etwa 25min Laufzeit eingespart werden.

In [24] wird eine Erweiterung des Skew-Algorithmus zur Reduktion der Textlänge beschrieben und getestet, durch die viele der nach der Benennung eindeutigen Namen nicht weiter sortiert werden müssen. Der Text der nächsten Rekursionebene ist dann höchstens doppelt so lang wie die Zahl der aktuell mehrdeutigen Namen. Interessant wäre eine Integration in Skew7.

Die LCP-Tabelle und insbesondere die LCPE-Tabelle enthalten im Allgemeinen bezogen auf ihre Größe nur relativ wenig Einträge die größer als 255 sind. Mit einer Kompression, wie in [2] beschrieben, kann die Größe beider Tabellen in vielen Fällen etwa gedrittelt werden.

Wie am Ende von Kapitel 8.4 beschrieben, sollten Suffix-Arrays und LCP-Tabellen, die ausschließlich für die externe Binärsuche verwendet werden, in Form von B-Bäumen sekundär gespeichert werden. So wird eine asymptotisch I/O-optimale externe Suche in $\text{search}(n) = \Theta\left(\log_{PB} n\right)$ realisierbar.

Anhang A

Nebenrechnungen

A.1 Gaußklammern

Die folgenden Sätze liefern obere Schranken für die Differenzen zwischen rekursiv definierten Funktionen mit und ohne Gaußklammern.

Satz A.1.1
Es sei eine Folge (a_i) *definiert als* $a_{i+1} = \lceil \lambda a_i \rceil$ *und* $a_0 = n$ *für* $n, i \in \mathbb{N}$, $\lambda \in \mathbb{Q}$ *und* $0 < \lambda = \frac{p}{q} < 1$. *Für beliebige* $r \in \mathbb{N}$ *gilt dann:*

$$\sum_{i=0}^{r} a_i \leq \frac{1-\lambda^r}{1-\lambda} \cdot n \; + \; \frac{q-1}{q-p} \cdot r$$

Beweis
Es sei $\delta_i = \lceil \lambda a_i \rceil - \lambda a_i$. Dann gilt:

$$\begin{array}{rcll} a_0 & = & n & \\ a_1 & = & \lambda a_0 + \delta_1 & = \lambda n + \delta_1 \\ a_2 & = & \lambda a_1 + \delta_2 & = \lambda\left(\lambda n + \delta_1\right) + \delta_2 \\ \vdots & & \vdots & \vdots \\ a_i & = & \lambda a_{i-1} + \delta_i & = \lambda^i n + \displaystyle\sum_{j=0}^{i-1} \lambda^j \delta_{i-j}. \end{array}$$

Für beliebige $a_i \in \mathbb{N}$ mit $pa_i = cq + d$ und $0 < d \leq q$ für geeignete $c, d \in \mathbb{Z}$ gilt:

$$\lceil \lambda a_i \rceil - \lambda a_i = \left\lceil \frac{pa_i}{q} \right\rceil - \frac{pa_i}{q} = \left\lceil \frac{cq+d}{q} \right\rceil - \frac{cq+d}{q} = \left\lceil \frac{d}{q} \right\rceil - \frac{d}{q} = 1 - \frac{d}{q} \leq \frac{q-1}{q}.$$

Daraus folgt:

$$a_i \leq \lambda^i n + \frac{q-1}{q} \sum_{j=0}^{i-1} \lambda^j \leq \lambda^i n + \frac{q-1}{q} \cdot \frac{1}{1-\frac{p}{q}} = \lambda^i n + \frac{q-1}{q} \cdot \frac{q}{q-p}$$

$$\Rightarrow \quad \sum_{i=0}^{r} a_i \leq \sum_{i=0}^{r} \lambda^i n + \frac{q-1}{q-p} \cdot r.$$

□

Satz A.1.2
Es sei $g : \mathbb{R} \to \mathbb{R}$ eine lineare Funktion und $\lambda = \frac{p}{q} \in (0,1)_{\mathbb{Q}}$. Weiter sei f eine Funktion $f : \mathbb{R} \to \mathbb{R}$ mit $f(x) = g(x) + f(\lceil \lambda x \rceil)$ für $x \in R$, $x > q$ und $f(x) = 0$ für $x \leq q$. Dann gilt:

$$f(n) \leq g\left(\frac{1}{1-\lambda} n + \delta\right)$$

mit $\delta \leq \frac{q-1}{q-p} \cdot \left(\log_{\frac{q}{p}} n - 1\right)$.

Beweis
Ist (a_i) die entprechende Folge aus Satz A.1.1 mit $a_0 = n$, dann gilt:

$$a_i \leq \lambda^i n + \frac{q-1}{q-p} \leq q - 1.$$

Weiter gilt: $\log_{\frac{q}{p}} n \leq i \Rightarrow n \leq \frac{1}{\lambda^i} \Rightarrow \lambda^i n \leq 1 \Rightarrow a_i \leq q \Rightarrow f(a_i) = 0$. Daraus folgt:

$$f(n) = \sum_{i=0}^{\left\lceil \log_{\frac{q}{p}} n \right\rceil - 1} g(a_i) \leq g\left(\frac{1}{1-\lambda} n + \frac{q-1}{q-p} \cdot \left(\log_{\frac{q}{p}} n - 1\right)\right).$$

□

Korollar A.1.3 *Für $n \leq 2^{64}$ und $\lambda = \frac{2}{3}$ bzw. $\lambda = \frac{3}{7}$ gilt in Satz A.1.2: $\delta < 217$ bzw. $\delta < 78$.*

A.2 Schärfere Speicherbedarfsschranken für Skew7

Die obere Speicherbedarfsschranke von $2n$ Worten aus Gleichung 6.2 wird für Skew7 in den seltensten Fällen angenommen. Der Grund dafür ist, dass in Gleichung 6.1 für jede Rekursionsebene die maximal mögliche Alphabetgröße angenommen wurde. Für viele Datensätze gilt aber $|\Sigma_1 = \Sigma| \ll n$. $|\Sigma_i|$ nähert sich dem theoretischen Maximum $\lambda^{m-1} n$ erst in tieferen Ebenen m an. Für $m \geq 1$ kann $|\Sigma_m|$ folgendermaßen noch besser nach oben abschätzt werden, wobei n_i die Stringlänge in Rekursionsebene m ist:

$$\begin{aligned} |\Sigma_1| &\leq n \\ |\Sigma_{m+1}| &\leq \min\left\{n_{m+1} - 1, |\Sigma_m|^v\right\} \\ \Rightarrow \quad |\Sigma_m| &\leq \min\left\{\lambda^{m-1} n, |\Sigma|^{v^{m-1}}\right\}. \end{aligned}$$

i	$\lvert\Sigma\rvert^{3^{i-1}}$	$\lvert\Sigma\rvert^{7^{i-1}}$
1	4	4
2	64	16384
3	262144	$3 \cdot 10^{29}$
4	$2 \cdot 10^{16}$	

i	$\lvert\Sigma\rvert^{3^{i-1}}$	$\lvert\Sigma\rvert^{7^{i-1}}$
1	5	5
2	125	78125
3	1953125	$2 \cdot 10^{34}$
4	$8 \cdot 10^{18}$	

i	$\lvert\Sigma\rvert^{3^{i-1}}$	$\lvert\Sigma\rvert^{7^{i-1}}$
1	95	95
2	857375	$7 \cdot 10^{13}$
3	$7 \cdot 10^{17}$	

Tabelle A.1: Maximales Wachstum der $|\Sigma_i|$ für verschiedene $|\Sigma|$

Es ist $m_0 := \min_{m \in \mathbb{N} \setminus \{0\}} \left\{ m \,\middle|\, |\Sigma|^{v^{m-1}} \geq \lambda^{m-1} n \right\}$ die Rekursionsebene, in der $|\Sigma_m|$ das theoretische Maximum frühestens erreichen kann. Nach Gleichung 6.1 ergibt sich für den temporären Speicherbedarf $\bar{S}_{(m)}(n)$ im worst-case in Rekursionsebene m dann:

$$\bar{S}_{(m)}(n) = \left(\frac{\lambda}{1-\lambda} + \frac{1-2\lambda}{1-\lambda} \cdot \lambda^{m-1} \right) \cdot n + \begin{cases} |\Sigma|^{v^{m-1}} & \text{, für } m < m_0 \\ \lambda^{m-1} n & \text{, sonst.} \end{cases}$$

Unter allen $\bar{S}_{(m)}(n)$ für $m \geq m_0$ ist $\bar{S}_{(m_0)}(n)$ maximal mit:

$$\bar{S}_{(m_0)}(n) = \left(\frac{2-3\lambda}{1-\lambda} \cdot \lambda^{m_0-1} + \frac{\lambda}{1-\lambda} \right) \cdot n.$$

Tabelle A.1 zeigt, wie schnell die Alphabetgrößen in Skew3 und Skew7 ausgehend von $|\Sigma| = 5$ oder $|\Sigma| = 95$ höchstens wachsen können. Im Folgenden betrachten wir speziell Skew7. Für Strings mit $|\Sigma| \geq 4$ gilt unter der Annahme $n \leq 2^{32}$ offensichtlich $m_0 \leq 3$. Für $\bar{S}_{(m_0)}(n)$ und $m_0 = 1, 2, 3$ gilt:

$$\begin{aligned} \bar{S}_{(m_0)|m_0=1}(n) &= 2n. \\ \bar{S}_{(m_0)|m_0=2}(n) &= \tfrac{9}{7} \cdot n \\ \bar{S}_{(m_0)|m_0=3}(n) &= \tfrac{48}{49} \cdot n, \end{aligned}$$

und für $m < m_0$:

$$\begin{aligned} \bar{S}_{(m)}(n) &= \left(\tfrac{\lambda}{1-\lambda} + \tfrac{1-2\lambda}{1-\lambda} \cdot \lambda^{m-1} \right) \cdot n + |\Sigma|^{v^{m-1}} \\ &= \left(\tfrac{3}{4} + \tfrac{1}{4} \cdot \left(\tfrac{3}{4} \right)^{m-1} \right) \cdot n + |\Sigma|^{7^{m-1}} \\ \bar{S}_{(1)|m_0>1}(n) &= n + |\Sigma| \\ \bar{S}_{(2)|m_0>2}(n) &= \tfrac{15}{16} \cdot n + |\Sigma|^7. \end{aligned}$$

Der insgesamt benötigte Speicherbedarf $S(n)$ ist höchstens so groß, wie das Maximum der $\bar{S}_{(m)}(n)$ über alle $m \in [1..r]$:

$$\begin{aligned} S_{|m_0 \geq 1}(n) &\leq 2n \\ S_{|m_0 \geq 2}(n) &\leq \max\left\{ n + |\Sigma|, \tfrac{9}{7} \cdot n \right\} \\ S_{|m_0 \geq 3}(n) &\leq \max\left\{ n + |\Sigma|, \tfrac{15}{16} \cdot n + |\Sigma|^7, \tfrac{48}{49} \cdot n \right\}. \end{aligned}$$

Satz A.2.1
Sei s ein String mit $|\Sigma| \leq 5$ und $1,25 \cdot 10^6 \leq n$. Für den temporär von Skew7 benötigten Speicher $S(n)$ in Worten gilt:

$$S(n) \leq n + 5.$$

Beweis
Es gilt $|\Sigma| \leq 5$ und $5^7 = 78125 < \frac{3}{7} \cdot n = \lambda n \Rightarrow m_0 \geq 3$. Wie oben gezeigt gilt:

$$\begin{array}{rcll} S(n) & \leq & \max\left\{n+5, \frac{15}{16} \cdot n + 5^7, \frac{48}{49} \cdot n\right\} & \\ & \leq & \max\left\{n+5, \frac{15}{16} \cdot n + 5^7\right\} & \mid \quad 5^7 \leq \frac{n}{16} \\ & \leq & n+5. & \end{array}$$

□

Satz A.2.2
Sei s ein String mit $|\Sigma| \leq \frac{2}{7} \cdot n$. Für den temporär von Skew7 benötigten Speicher $S(n)$ in Worten gilt:

$$S(n) \leq \frac{9}{7} \cdot n.$$

Beweis
Aus $|\Sigma| \leq \frac{2}{7} \cdot n$ folgt $m_0 \geq 2$ und wie oben gezeigt gilt:

$$\begin{array}{rcl} S(n) & \leq & \max\left\{n + |\Sigma|, \frac{9}{7} \cdot n\right\} \\ & \leq & \frac{9}{7} \cdot n. \end{array}$$

□

A.3 I/O-Analyse des externen Skew-Algorithmus

Es seien σ_m und τ_m die Größen der Datentypen, die in der Implementation in Rekursionschritt m zum Speichern von Eingabezeichen bzw. Indizes $[0..n)$ verwendet werden. Es gilt $\sigma_{m+1} = \tau_m$ und $\tau_{m+1} = \tau_m$. O.B.d.A. sei $\tau_m = 1$, $\sigma_{m+1} = 1$ für alle $m \in \mathbb{N} \setminus \{0\}$ und $\sigma := \sigma_1$ die relative Größe eines Eingabezeichens in Bezug auf ein Indexzeichen. Die Modellgrößen B, M und N beziehen sich alle auf den Datentyp der Indizes (Wort).

Da zwischen den Modulen des externen Skew-Algorithmus Tupel ausgetauscht werden, die ausschließlich Eingabezeichen oder Positionen enthalten, sind die Größen der Moduldatentypen Linearkombinationen von σ_m und 1 mit den ganzzahligen Koeffizienten x und y (siehe Paare (x, y) in Abb. 6.4 und 6.5). D sei ein perfektes Difference Cover, $v = |D|^2 - |D| + 1$ und $T^D(n)$ die Zeitkomplexität

des zugehörigen externen Skew-Algorithmus. Dann gilt für Skew3:

$$\begin{aligned}
T^{\{1,2\}}(n) &\leq^{1} \text{scan}_\sigma(n) + \text{sort}_{3\sigma+1}\left(\tfrac{2n}{3}\right) + \\
&\quad \text{permute}_2\left(\tfrac{2n}{3}\right) + T^{\{1,2\}}_{\text{rek}}\left(\tfrac{2n}{3}\right) + \text{permute}_2\left(\tfrac{2n}{3}\right) + \\
&\quad \text{scan}_\sigma(n) + \text{permute}_{2\sigma+3}\left(\tfrac{2n}{3}\right) + \text{sort}_{2\sigma+3}\left(\tfrac{n}{3}\right) \\
&\leq \text{sort}\left((3\sigma+1)\tfrac{2n}{3} + (2\sigma+3)\tfrac{n}{3}\right) + \text{permute}\left((2\sigma+7)\tfrac{2n}{3}\right) + \\
&\quad \text{scan}(2\sigma n) + T^{\{1,2\}}_{\text{rek}}\left(\tfrac{2n}{3}\right) \\
&\leq \text{sort}\left((8\sigma+5)\tfrac{n}{3}\right) + \text{permute}\left((4\sigma+14)\tfrac{n}{3}\right) + \\
&\quad \text{scan}(2\sigma n) + T^{\{1,2\}}_{\text{rek}}\left(\tfrac{2n}{3}\right) \\
T^{\{1,2\}}_{\text{rek}}(n) &\leq \text{sort}\left(\tfrac{8n}{3}\right) + \text{permute}\left(\tfrac{4n}{3}\right) + T^{\{1,2\}}_{\text{rek}}\left(\tfrac{2n}{3}\right) + \text{permute}\left(\tfrac{4n}{3}\right) + \\
&\quad \text{permute}_{read}(2n) + \text{permute}\left(\tfrac{10n}{3}\right) + \text{sort}\left(\tfrac{5n}{3}\right) \\
&\leq \text{sort}(13n) + \text{permute}(18n) + \text{permute}_{read}(6n)
\end{aligned}$$

und allgemein für ein perfektes Difference Cover D modulo v und $\lambda := \frac{|D|}{v}$:

$$\begin{aligned}
T^{D}(n) &\leq \text{scan}_\sigma(n) + \text{sort}_{v\sigma+1}(\lambda n) + \text{permute}_2(\lambda n) + \\
&\quad T^{D}_{\text{rek}}(\lambda n) + \text{permute}_2(\lambda n) + \text{scan}_\sigma(n) + \\
&\quad \text{permute}_{(v-1)\cdot\sigma+|D|+1}(\lambda n) + \textstyle\sum_{i\notin D} \text{sort}_{\Delta_{D,v}(i)\cdot\sigma+|D|+1}\left(\frac{n}{v}\right) \\
&\leq \text{sort}\left(|D|\sigma n + \lambda n + \left(\bar{\Delta}_{D,|D|^2-|D|+1}(\bar{D})\cdot\sigma + |D| + 1\right)\cdot(1-\lambda)\cdot n\right) + \\
&\quad \text{permute}\left(\left(|D|^2 - |D|\right)\cdot\sigma + |D| + 5\right)\cdot\lambda n\right) + \\
&\quad \text{scan}_{read}(2\sigma n) + T^{D}_{\text{rek}}(\lambda n) \\
T^{D}_{\text{rek}}(n) &\leq \text{sort}\left(|D|n + \lambda n + \left(\bar{\Delta}_{D,|D|^2-|D|+1}(\bar{D}) + |D| + 1\right)\cdot(1-\lambda)\cdot n\right) + \\
&\quad \text{permute}\left(\left(|D|^2 + 5\right)\cdot\lambda n\right) + \\
&\quad \text{permute}_{read}(2n) + T^{D}_{\text{rek}}(\lambda n) \\
&\leq \text{sort}\left(\left(\tfrac{|D|+\lambda}{1-\lambda} + \bar{\Delta}_{D,|D|^2-|D|+1}(\bar{D}) + |D| + 1\right)\cdot n\right) + \\
&\quad \text{permute}\left(\left(|D|^2 + 5\right)\cdot\tfrac{\lambda}{1-\lambda}\cdot n\right) + \\
&\quad \text{permute}_{read}\left(\tfrac{2}{1-\lambda}\cdot n\right)
\end{aligned}$$

$$\begin{aligned}
T^{D}(n) &\leq^{2} \text{sort}\left(\left(|D|\sigma + \bar{\Delta}_{D,|D|^2-|D|+1}(\bar{D})\cdot(\sigma - \sigma\lambda + \lambda) + (|D| + \lambda)\cdot\tfrac{\lambda}{1-\lambda} + |D| + \lambda + 1\right)\cdot n\right) + \\
&\quad \text{permute}\left(\left(\left(|D|^2 - |D|\right)\cdot\sigma + \left(|D|^2 + 5\right)\cdot\tfrac{\lambda}{1-\lambda} + |D| + 5\right)\cdot\lambda n\right) + \\
&\quad \text{scan}\left(\left(\sigma + \tfrac{\lambda}{1-\lambda}\right)\cdot 2n\right)
\end{aligned}$$

$$\begin{aligned}
T^{\{1,2\}}(n) &\leq \text{sort}\left(\left(2\tfrac{2}{3}\cdot\sigma + 10\tfrac{1}{3}\right)\cdot n\right) + \text{permute}\left(\left(1\tfrac{1}{3}\cdot\sigma + 16\tfrac{2}{3}\right)\cdot n\right) + \text{scan}\left((2\sigma+4)\cdot n\right) \\
T^{\{1,2,4\}}(n) &\leq \text{sort}\left(\left(6\sigma + 9\tfrac{1}{4}\right)\cdot n\right) + \text{permute}\left(\left(2\tfrac{4}{7}\cdot\sigma + 7\tfrac{13}{14}\right)\cdot n\right) + \text{scan}\left(\left(2\sigma + 1\tfrac{1}{2}\right)\cdot n\right).
\end{aligned}$$

Ist $S^D(n)$ der Sekundärspeicherbedarf des externen Skew-Algorithmus und $S^D_{(m)}(n)$

[1] In der Formel wurde auf Gaußklammern verzichtet. Die dadurch maximal entstehende Abweichung ist im Anhang in Korollar A.1.3 angegeben.

[2] für den Algorithmus aus Kapitel 4.3 gilt: $\text{permute}_{read}(x) = \text{scan}(x)$

der Bedarf in Rekursionschritt m vor und nach dem rekursiven Abstieg, dann gilt:

$$
\begin{aligned}
S^D_{(m)}(n) &= \overbrace{\sum_{i=1}^{m-1} \lambda^i \cdot 2n}^{\text{Mappers rekursiv (4)}} \\
&\quad + \max \left\{ \begin{array}{l} (v+1)\,\lambda, \\ (v+1)\,\lambda + 2\lambda, \\ 4\lambda, \\ \left(\lambda\,(v+|D|) + (1-\lambda)\cdot\left(\bar{\Delta}_{D,v}(\bar{D}) + |D|+1\right)\right)\lambda + 2\lambda, \\ 2\lambda + \underbrace{\lambda\,(v+|D|)}_{\text{Mapper (12)}} + \underbrace{(1-\lambda)\cdot\left(\bar{\Delta}_{D,v}(\bar{D}) + |D|+1\right)}_{\text{Sorters (13–16)}} \end{array} \right\} \cdot \lambda^{m-1} n \\
&= \sum_{i=1}^{m} \lambda^i \cdot 2n + \left(\lambda\,(v+|D|) + (1-\lambda)\cdot\left(\bar{\Delta}_{D,v}(\bar{D}) + |D|+1\right)\right)\cdot\lambda^{m-1} n \\
&= \tfrac{\lambda-\lambda^{m+1}}{1-\lambda} \cdot 2n + \left(\lambda\,(v+|D|) + (1-\lambda)\cdot\left(\bar{\Delta}_{D,v}(\bar{D}) + |D|+1\right)\right)\cdot\lambda^{m-1} n \\
&= \tfrac{2\lambda}{1-\lambda} \cdot n + \left(\lambda \cdot \underbrace{\left(v + |D| - \frac{2\lambda}{1-\lambda}\right)}_{\geq 1 \text{ für}^3\ |D|\geq 2} + (1-\lambda)\cdot\left(\bar{\Delta}_{D,v}(\bar{D}) + |D|+1\right) \right) \cdot \lambda^{m-1} n
\end{aligned}
$$

$$
\begin{aligned}
S^D_{(1)}(n) &= \max \left\{ \begin{array}{l} (v\sigma+1)\,\lambda, \\ (v\sigma+1)\,\lambda + 2\lambda, \\ 4\lambda, \\ \left(\lambda\,(v+|D|) + (1-\lambda)\cdot\left(\bar{\Delta}_{D,v}(\bar{D}) + |D|+1\right)\right)\lambda + 2\lambda, \\ 2\lambda + \lambda\,((v-1)\sigma + |D|+1) + (1-\lambda)\cdot\left(\bar{\Delta}_{D,v}(\bar{D})\sigma + |D|+1\right) \end{array} \right\} \cdot n \\
&= \max \left\{ \begin{array}{l} S^D_{(2)}(n) - 2\lambda^2 n, \\ 2\lambda n + ((v-1)\sigma+1)\,\lambda n \\ \quad + \max \left\{ \begin{array}{l} \sigma\lambda, \\ \lambda|D| + (1-\lambda)\cdot\left(\bar{\Delta}_{D,v}(\bar{D})\sigma + |D|+1\right) \end{array} \right\} \end{array} \right\} \cdot n \\
&=^4 \max \left\{ \begin{array}{l} S^D_{(2)}(n) - 2\lambda^2 n, \\ \left(\lambda\,((v-1)\sigma + |D| + 3) + (1-\lambda)\cdot\left(\bar{\Delta}_{D,v}(\bar{D})\sigma + |D|+1\right)\right)\cdot n \end{array} \right\}
\end{aligned}
$$

$$
\begin{aligned}
S^D(n) &\leq \max_{m\in[1..r]} \left\{ S^D_{(m)}(n) \right\} = \max\left\{ S^D_{(1)}(n), S^D_{(2)}(n) \right\} \\
&= \max \left\{ \begin{array}{l} \lambda\,((v-1)\sigma + |D| + 3) + (1-\lambda)\cdot\left(\bar{\Delta}_{D,v}(\bar{D})\sigma + |D|+1\right), \\ 2\lambda + \lambda^2\cdot(v+|D|+2) + \lambda(1-\lambda)\cdot\left(\bar{\Delta}_{D,v}(\bar{D}) + |D|+1\right) \end{array} \right\} \cdot n
\end{aligned}
$$

Die obere Schranke für S^D ist eine Abschätzung des worst-case bei maximaler Rekursionstiefe. Im best-case sind die Namen der Tupel bereits im ersten Schritt

---

[3] $|D|\geq 2 \quad\Rightarrow\quad \left(v\geq 3 \wedge \lambda \leq \frac{2}{3}\right) \quad\Rightarrow\quad \frac{2\lambda}{1-\lambda} \leq 4$

[4] $|D| > 2 \quad\Rightarrow\quad \left(1-\lambda > \lambda \wedge \bar{\Delta}_{D,v} \geq 1\right) \quad\Rightarrow\quad (1-\lambda)\cdot\bar{\Delta}_{D,v}(\bar{D})\sigma \geq \sigma\lambda$
$|D| = 2 \quad\Rightarrow\quad \left(\lambda = \frac{2}{3} \wedge \bar{\Delta}_{D,v} = 2\right) \quad\Rightarrow\quad (1-\lambda)\cdot\bar{\Delta}_{D,v}(\bar{D})\sigma \geq \sigma\lambda$

eindeutig und es gilt:

$$\begin{aligned} S^D(n) &\geq \max \left\{ \begin{array}{l} (v\sigma+1)\,\lambda, \\ (v\sigma+1)\,\lambda + 2\lambda, \\ 4\lambda, \\ 2\lambda + \lambda\,((v-1)\sigma + |D| + 1) + (1-\lambda)\cdot\left(\bar{\Delta}_{D,v}(\bar{D})\sigma + |D| + 1\right) \end{array} \right\} \cdot n \\ &= \left(\lambda\,((v-1)\sigma + |D| + 3) + (1-\lambda)\cdot\left(\bar{\Delta}_{D,v}(\bar{D})\sigma + |D| + 1\right)\right) \cdot n. \end{aligned}$$

Anhang B

Seqan

Seqan ist eine generische Software-Bibliothek für die Programmiersprache C++, die Datenstrukturen und Algorithmen für Anwendungen der Genom- und Proteinanalyse zur Verfügung stellt. Einige der wichtigsten Design-Prinzipien von Seqan sind:

- **Performanz**
  Die starke Streuung der Anwendungsdaten erfordert performante Algorithmen mit guter Laufzeit- und Speichereffizienz.
- **Spezialisierung**
  Kontextabhängige Optimierungen sind allgemeinen Lösungen vorzuziehen.
- **Erweiterbarkeit**
  Die Funktionalität der Bibliothek soll erweiterbar sein, ohne dass Quelltexte geändert werden müssen.
- **Generalisierung**
  Sämtliche Datenstrukturen und Algorithmen sollen weitestgehend typ- und problemunabhängig sein.
- **Integration**
  Die Algorithmen und Datenstrukturen sollen mit anderen Bibliotheken interagieren können.
- **Einfachheit**
  Die Benutzerschnittstelle und Interna sollen intuitiv bzw. verständlich sein und einen Kompromiss zwischen Einfachheit und Perfektion darstellen.

Um diesen Anforderungen gerecht zu werden, benutzt Seqan C++ Templates und eine Reihe von Paradigmen, um effizient und generisch zugleich zu sein. Ein Auszug davon soll im Folgenden vorgestellt werden.

Generalisierung und Konzepte

Sämtliche Algorithmen und Datenstrukturen sind Datentyp unabhängig und können vom Benutzer der Bibliothek nach Bedarf spezialisiert werden. Der Alphabettyp eines Strings bspw. wird bei dessen Definition als Template-Argument übergeben.

Algorithmen werden so implementiert, dass sie auf eine Vielzahl von Datenstrukturen anwendbar sind und zugleich auf spezielle inherente Eigenschaften dieser eingehen können. Eine Menge von Eigenschaften eines Datentyps wird auch als Konzept bezeichnet. In Seqan kann ein Algorithmus verschiedene Implementierungen haben, die speziell für bestimmte Konzepte optimiert sind und zur Kompilationszeit automatisch ausgewählt werden.

Globale Funktionen und Erweiterbarkeit

Ein wichtiges Paradigma in Seqan ist, die Definition von Member-Funktionen zu vermeiden und soweit wie möglich durch globale Funktionen zu ersetzen. So weicht die Objektorientierung der Orientierung auf globale Algorithmen, die möglichst generisch auf Datenstrukturen arbeiten. Dadurch lässt sich ein höherer Grad der Erweiterbarkeit und Generalisation erreichen. Neue Funktionen können hinzugefügt werden, ohne in den Quelltext der Bibliothek eingreifen zu müssen. Um bspw. eine allgemeine Funktion durch eine für eigene Datenstrukturen optimierte zu ersetzen, genügt es, diese mit gleichem Namen für die Datenstruktur zu spezialisieren, da der Compiler die am stärksten für die Aufrufparamter spezialisierte Funktion auswählt. Globale Funktionen können generisch auch für Typen definiert werden, die einem sehr schwachen Konzept entsprechen (eingebaute Datentypen).

Template-Vererbung

Sind die Klassen B und C jeweils Verfeinerungen der Klasse A, können diese in C++ einfach als Kindklassen von A definiert werden. Polymorphie wird dabei über die Definition virtueller Funktionen erreicht. Leider sind die so definierten Funktionen nur über indirekte Sprünge erreichbar und nicht inline-fähig und damit nicht vom Compiler entsprechend des Kontexts optimierbar.

In Seqan wird dieses Paradigma vermieden und durch ein anderes ersetzt. A, B und C werden als Spezialisierungen einer einzigen Stammklasse S deklariert, die einen zusätzlichen Template-Parameter (TSpec) erhält. A wird ohne Einschränkungen und B und C mit (Teil-)Spezialisierungen für den Typ TSpec deklariert. Weitere Verzweigungen der Unterklassen können über verschieden starke Spezialisierungen von TSpec realisiert werden. Member-Funk-

tionen können nun, sofern möglich[1], als globale Funktionen definiert werden, die für eine einzelne oder einen Unterbaum von Klassen (teil-)spezialisiert sind. Es handelt sich hierbei um eine konzeptionelle Vererbung, bei der im Gegensatz zur C++-Vererbung nicht automatisch alle Membervariablen übernommen werden müssen.

Template-Polymorphie

Polymorphie kann über die Template-Delegation erreicht werden. Wird für A und B eine Funktion f benötigt, die für beide Klassen gleich deklariert aber verschieden definiert sein soll, kann das einfach durch verschiedene globale Funktionen erreicht werden, die mit S aufgerufen werden und unterschiedlich stark im Parameter TSpec spezialisiert sind. Andere Funktionen, die bspw. allgemein auf allen Unterklassen von S arbeiten, können f aufrufen, ohne zu wissen, welche konkrete (ehemals virtuelle) Definition verwendet wird. Da in C++ bereits zur Zeit der Kompilierung feststehen muss, welche Implementierung von f verwendet wird, lässt sich so allerdings nur Kompilationszeit-Polymorphie erreichen. Laufzeit-Polymorphie erfordert die klassischen C++-Konzepte. In vielen Fällen wird aber nur Kompilationszeit-Polymorphie benötigt.

Metafunktionen und Template Meta Programming

Mit Templates ist es in C++ möglich, nicht nur Funktionen zu implementieren, deren Ergebnisse Werte oder Objekte sind, sondern Metafunktionen zu definieren, die auch Typen oder Klassen zurückliefern. Diese Metafunktionen lassen sich in Form von Datenstrukturen implementieren, welche abhängig von ihren Template-Argumenten, entsprechende Rückgabetypen oder -konstanten definieren. Die Argumente müssen Typen oder Konstanten sein und zur Kompilationszeit feststehen. Mit Metafunktionen lassen sich bspw. der Alphabettyp oder Längentyp eines Strings in einem Algorithmus verwenden, welcher nur mit dem String spezialisiert wurde.
Template-Metaprogrammierung ist eine Weiterführung dieses Konzepts. Mit Hilfe der Template-Spezialisierung lassen sich Metaprogramme schreiben, die bereits während der Kompilationszeit ausgeführt werden oder Loop-Unrolling und komplexe typ- oder wertabhängige Case-Anweisungen realisieren.

Tag-Klassen

Tag-Klassen sind Klassen, die als Schalter für andere Klassen oder Funktionen dienen. Sie enthalten meist keine Variablen oder Funktionen, sondern eventu-

[1] Konstruktoren, Destruktoren und einige Operatoren können nur als Member-Funktionen definiert werden.

ell Typdefinitionen oder Konstanten. Gibt es mehrere Definitionen einer Klasse oder Funktion, die sich in nur einem Template-Parameter unterscheiden, wählt der Compiler diejenige Spezialisierung, die den Template-Argumenten am stärksten entspricht. Steht der Template-Parameter für eine Tag-Klasse, kann durch die Angabe eines entsprechenden Tags eine bestimmte Definition einer Standard-Defintion vorgezogen werden.

Ein einfaches Beispielprogramm

```
001 #include "seqan/sequence.h"
002 #include "index/index.h"
003 #include <iostream>
004
005 using namespace std;
006 using namespace seqan;
007
008 int main()
009 {
010     typedef Index<String<char> > TIndex;
011
012     String<char>        text    = "tobeornottobe";
013     String<char>        needle  = "be";
014     TIndex              idx;
015     QueryResult<TIndex> result;
016
017     assign(idx, text);
018     query(idx, needle, result);
019
020     if (!eof(result))
021         cout << "found '" << needle << "' in '" << text << "' at: ";
022     else
022         cout << "'" << needle << "' not found in '" << text << "'";
023
024     while (!eof(result)) {
025         cout << *result << " ";
026         ++result;
027     }
028     cout << endl;
030     return 0;
031 }
```

Ausgabe:

```
found 'be' in 'tobeornottobe' at: 11 2
```

Die Quelltexte, Beispiele und Ergebnisse dieser Arbeit sind im Internet unter `http://www.informatik.hu-berlin.de/~weese/genindex/` verfügbar.

Literaturverzeichnis

[1] Abouelhoda, M.I. ; Kurtz, S. ; Ohlebusch, E.: The Enhanced Suffix Array and its Applications to Genome Analysis. In: *Proceedings of the Second Workshop on Algorithms in Bioinformatics*, Lecture Notes in Computer Science 2452, Springer-Verlag, 449-463

[2] Abouelhoda, M.I. ; Ohlebusch, E. ; Kurtz, S.: Optimal Exact String Matching Based on Suffix Arrays. In: *Proceedings of the Ninth International Symposium on String Processing and Information Retrieval*, Lecture Notes in Computer Science 2476, Springer-Verlag, 31-43

[3] Aggarwal, A. ; Vitter, J.S.: The Input/Output Complexity of Sorting and Related Problems. In: *Communications of the ACM*, 1988

[4] Aho, A.V. ; Hopcroft, J.E. ; Ullman, J.D.: *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. – ISBN 0–20100029 –6

[5] Barve, R.D. ; Vitter, J.S.: A Simple and Efficient Parallel Disk Mergesort. In: *ACM Symposium on Parallel Algorithms and Architectures*, 1999, S. 232–241

[6] Burrows, M. ; Wheeler, D. J.: A block-sorting lossless data compression algorithm. Version: 1994. `citeseer.ist.psu.edu/76182.html` (124). – Elektronische Ressource

[7] Crauser, A. ; Ferragina, P.: A theoretical and experimental study on the construction of suffix arrays in external memory. In: *Algorithmica* 32 (2002), January, Nr. 1, S. 1–35. ISBN 0178–4617

[8] Dementiev, R. ; Kärkkäinen, J. ; Mehnert, J. ; Sanders, P.: Better external memory suffix array construction. In: *Workshop on Algorithm Engineering & Experiments, Vancouver*

[9] Dementiev, R. ; Sanders, P.: Asynchronous parallel disk sorting. In: *Proc. 15th Annual Symposium on Parallelism in Algorithms and Architectures*, ACM, 2003

[10] Ehrig, H. ; Mahr, B.: *Monographs in Theoretical Computer Science. An EATCS Series*. Bd. 6: *Fundamentals of Algebraic Specification I: Equations und Initial Semantics*. Springer, 1985. – ISBN 3–540–13718–1

[11] Farach, M. ; Ferragina, P. ; Muthukrishnan, S.: Overcoming the Memory Bottleneck in Suffix Tree Construction. In: *FOCS '98: Proceedings of the 39th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1998. – ISBN 0–8186–9172–7, S. 174

[12] Haanpää, H.: Minimum Sum and Difference Covers of Abelian Groups. In: *Journal of Integer Sequences* 7 (2004)

[13] Kärkkäinen, J. ; Sanders, P.: Simple linear work suffix array construction. In: *Proc. 13th International Conference on Automata, Languages and Programming*, Springer, 2003

[14] Kärkkäinen, J. ; Sanders, P. ; Burkhardt, S.: Linear work suffix array construction. In: *Journal of the ACM* (2005)

[15] Kasai, T. ; Lee, G. ; Arimura, H. ; Arikawa, S. ; Park, K.: Linear-Time Longest-Common-Prefix Computation in Suffix Arrays and Its Applications. In: *CPM '01: Proceedings of the 12th Annual Symposium on Combinatorial Pattern Matching*. London, UK: Springer-Verlag, 2001. – ISBN 3–540–42271–4, S. 181–192

[16] Kim, D.K. ; Sim, J.S. ; Park, H. ; Park, K.: Constructing suffix arrays in linear time. In: *J. Discrete Algorithms* 3 (2005), Nr. 2-4, S. 126–142

[17] Ko, P. ; Aluru, S.: Space Efficient Linear Time Construction of Suffix Arrays. In: *CPM*, 2003, S. 200–210

[18] Larsson, N.J. ; Sadakane, K.: Faster Suffix Sorting / Department of Computer Science, Lund University. Sweden, Mai 1999 (LU-CS-TR:99-214, LUNDFD6/(NFCS-3140)/1–20/(1999))

[19] Lee, S. ; Park, K.: Efficient Implementations of Suffix Array Construction Algorithms. In: *15th Australasian Workshop on Combinatorial Algorithms (AWOCA)*, 2004, S. 64–72

[20] Manber, U. ; Myers, E.W.: Suffix arrays: a new method for on-line string searches. In: *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, Society for Industrial and Applied Mathematics, 1990. – ISBN 0–89871–251–3, S. 319–327

[21] Manber, U. ; Myers, E.W.: Suffix Arrays: A New Method for On-Line String Searches. In: *SIAM J. Comput.* 22 (1993), Nr. 5, S. 935–948

[22] Manzini, Giovanni: Two Space Saving Tricks for Linear Time LCP Array Computation. In: *SWAT*, 2004, S. 372–383

[23] McCreight, E.M.: A space-economical suffix tree construction algorithm. In: *Journal of the ACM* (1976), Nr. 23(2):262-272

[24] Puglisi, S. ; Smyth, W.F. ; Turpin, A.: The Performance of Linear Time Suffix Sorting Algorithms. In: *DCC*, 2005, S. 358–367

[25] Puglisi, S. ; Smyth, W.F. ; Turpin, A.: A Taxonomy of Suffix Array Construction Algorithms. In: *Proc. Prague Stringology Conference '05, Jan Holub (ed.)*. Prague, Czech Republic, August 2005, S. 1–30

[26] Ukkonen, E.: On-Line Construction of Suffix Trees. In: *Algorithmica* 14 (1995), Nr. 3, S. 249–260

[27] Vitter, J.S. ; Shriver, E.A.M.: Algorithms for Parallel Memory I: Two-Level Memories. 1993 (Technical report DUKE–TR–1993–01)

[28] Weiner, P.: Linear pattern matching algorithms. In: *Proceedings of the 14th IEEE Symposium on Switching and Automata Theory*, 1973, S. 1–11

Selbständigkeitserklärung

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

Berlin, den 2. Mai 2006

Einverständniserklärung

Ich erkläre hiermit mein Einverständnis, dass die vorliegende Arbeit in der Bibliothek des Instituts für Informatik der Humboldt-Universität zu Berlin ausgestellt werden darf.

Berlin, den 2. Mai 2006