Technische Universität München
Zentrum Mathematik

Ein Konstruktionsalgorithmus einer Indexstruktur für Genome

Gabriele Witterstein

Vollständiger Abdruck der von der Fakultät für Mathematik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:		Univ.-Prof. Dr. Peter Gritzmann
Prüfer der Dissertation:	1.	Univ.-Prof. Dr. Dr. h.c. mult. Karl-Heinz Hoffmann
	2.	Univ.-Prof. Dr. Stefan Kramer
	3.	Univ.-Prof. Dr. Ulf Leser Humboldt-Universität zu Berlin (schriftliche Beurteilung)

Die Dissertation wurde am 06.12.2004 bei der Technischen Universität eingereicht und durch die Fakultät für Mathematik am 23.06.2005 angenommen.

Inhaltsverzeichnis

Abstract **5**

Kurzzusammenfassung **5**

1 Einleitung **7**

2 Datenstrukturen zur Unterstützung einer schnellen exakten Suche **13**

2.1 Grundlegende Notationen 13

2.2 Definition Baumartiger Datenstrukturen 14

2.3 Suchoperationen 16

2.4 Persistente Datenstrukturen 17

3 Repräsentation **19**

3.1 Logische Darstellung 19

3.2 Physikalische Darstellung - Baumkodierungsstruktur 23

4 Konstruktionsalgorithmen für persistente Indexstrukturen **31**

4.1 Konstruktionsalgorithmen für persistente Suffix Trees 31

4.2 Evaluation linearer Algorithmen 34

4.3 Wege der Partitionierung 35

4.4 Selektive Partitionierung – Ein tatsächlich linearer Konstruktionsalgorithmus und dessen Nichtverwendbarkeit 37

4.5 Präfix Clusterung 42

5 Datenpartitionierung – Ein neuer Konstruktionsalgorithmus **47**

5.1 Schritt 1 : Partitionierung der Daten 49

5.1.1 Beschreibung 49

5.1.2 Lineare in-memory Algorithmen 49

5.1.3 Der modifizierte Algorithmus von Ukkonen – *mukk* 50
5.1.4 Persistente Speicherung 55
5.1.5 Analyse von Schritt 1 56
5.2 Schritt 2 : Mischen der Bäume 57
5.2.1 Beschreibung 57
5.2.2 Partitionierung eines Suffix Tree 58
5.2.3 Optimale Segmentierung 60
5.2.4 Festlegung eines Muster geeignet bei DNA Sequenzen 62
5.3 Externes Mischen 65
5.3.1 Beschreibung des tatsächlichen Algorithmus 75
5.4 Inneres Mischen 76
5.5 Stochastische Analyse 78
5.6 Schritt 3 : Endmischen 86

6 Experimentelle Ergebnisse **87**
6.1 Konstruktionszeiten 87
6.1.1 Zusammenfassung und Interpretation 95
6.2 Meßzeiten der exakten Suche 95

Dank **99**

Literaturverzeichnis **101**

Abstract

Due to world-wide Genome sequence projects, large quantities of DNA raw data arise. The fundamental task emerging is their textual analysis. In the frame of this work, an index structure over DNA sequences is introduced, which supports important string analysis tasks, such as fast approximative string matching and fast exact string matching, and so on.

Because such structures are very main-memory expensiv, a design of a variant of a suffix tree is discussed. This design is orientated on the most important biological needs as well as on the usage of a relatively slow, but unlimited storage medium. The goal is a preferably space-saving representation with a maximum efficiency in the access.

Beside that, the main scientific question is the construction of such a data structure under a 'Two-Level Memory Model' with small main storage. A construction algorithm is presented, which is based on the partitioning of the data set. In the context of a stochastic analysis, under certain model assumptions to the DNA sequences, it can be shown that the complexity of the algorithm is of order $O(n \log n)$ in average.

An implementation and test runs with real sequences demonstrate the applicability of the algorithm in biology. Comparisons with similar algorithms in this field document a considerable performance benefit.

Kurzzusammenfassung

Infolge weltweiter Genom-Sequenzierungsprojekte entstehen große Mengen an DNA Rohdaten. Eine grundlegende Fragestellung ist deren textuelle Analyse. Im Rahmen dieser Arbeit wird eine Indexstruktur über DNA-Sequenzen vorgestellt, welche grundlegende Aufgaben, wie schnelles approximatives Stringmatching, exaktes Stringmatching usw. unterstützt.

Da derartige Strukturen sehr speicherintensiv sind, wird ein Design einer Suffix Tree Variante besprochen, das sich sowohl an den wichtigsten biologischen Anwendungen ausrichtet, als auch an den Gebrauch eines relativ langsamen, dafür unbegrenzten Speichermediums. Das Ziel ist eine möglichst platzsparende Repräsentation bei maximaler Effizienz im Zugriff.

Die hauptsächliche wissenschaftliche Fragestellung ist die Konstruktion einer solchen Datenstruktur unter einem 'Two-Level-Memory Model' mit geringem Hauptspeicher. Es wird ein Konstruktionsalgorithmus vorgestellt, welcher auf der Partitionierung der Datenmenge basiert. Im Rahmen einer stochastischen Analyse, unter gewissen Modell-

annahmen an die DNA Sequenzen, kann gezeigt werden, daß die Komplexität des Algorithmus durchschnittlich von der Ordnung $O(n \log n)$ ist.

Eine Implementierung und Testläufe mit realen Sequenzen belegen die Anwendbarkeit des Algorithmus in der Biologie. Vergleiche mit ähnlichen Algorithmen auf diesem Gebiet dokumentieren einen beträchtlichen Performancegewinn.

Kapitel 1

Einleitung

Eine DNA Sequenz ist ein Wort über einem Alphabet, bestehend aus vier Buchstaben – A, T, G und C. Diese Buchstaben stehen für die Nukleotide *bzw.* Basen: Adenin, Thymin, Guanin und Cytosin. Innerhalb der letzten Jahre wurden die Genome verschiedener Organismen, das ist die Gesamtheit der genetischen Erbinformation, vollständig entschlüsselt. Es können dabei zwei Arten der Zugehörigkeit unterschieden werden. *Eukaryonten*, zu ihnen zählen, zum Beispiel Tiere und Pflanzen, sind Organismen deren Zellen, Membran umhüllte Gebiete enthalten. *Prokaryonten* sind Organismen, deren Zellen etwas derartiges fehlt. Zu ihnen gehören Bakterien und Archaea.

Bis heute sind 32 eukaryontische Genome bestimmt. Deren Sequenzen wurden unter anderem unter *http://www.genomesonline.org* veröffentlicht. Beispiele sind: die Bäcker Hefe *Saccharomyces cerevisiae* (1997), der Wurm *Caenorhabditis Elegans* (1998), die Fruchtfliege *Drosophila Melanogaster* (2000), die Modellpflanze *Arabidopsis Thaliana*, der Mensch (2001), die Malaria Mücke *Anopheles Gambiae* (2002) und die Ratte (2004). Gleichzeitig sind nahezu 1100 Sequenzierungsprojekte derzeit in Bearbeitung, siehe dazu *http://www.genomesonline.org/Gold_statistics.html*. Um einiges größer sind diese Zahlen für Prokaryonten. Denn die Genomlänge einfachster Archaea ist wesentlich kürzer, als bei Eukaryonten durchschnittlich.

All diese Anstrengungen erzeugen eine große Menge an Rohdaten in Form von DNA Sequenzen. Die DNA Sequenz eines Eukaryonten ist oftmals länger als 100 Millionen Basen Paare (bp = "base pairs"). Das Genom des Menschen hat ungefähr eine Länge von $3,2 \cdot 10^9$ bp. In derselben Größenordnung liegen die Genome der Maus und der Ratte. Doch das ist keine obere Grenze – Gerste hat etwa 5 Gbp, Weizen sogar 16 Gbp. Eine Vielzahl anderer Organismen werden noch zu entschlüsseln sein. Betrachtet man die bisherige Entwicklung der Gesamtheit der identifizierten Nukleotide, ob als komplettes Genom oder viele Teilstücke von Genomen, so war deren Wachstum von 1996 – 2004 von 1 Gbp auf 70,1 Gbp exponentiell und wird es in Anbetracht vielzähliger noch zu sequenzierender Spezies auch bleiben (siehe: EMBL Statistik, *http://www3.ebi.ac.uk/Services/DBStats*).

Doch das komplette Sequenzieren ganzer Genome ist aufwendig, und daher nicht im-

mer praktisch, gerade für Genomgrößen, die weit hinter dem des Menschen liegen. Eine bessere Lösung zur Untersuchung von Genen ist die Sequenzierung von ESTs. ESTs (Expressed Sequence Tags) sind kurze Teilstücke der mRNA, welche durch den Prozeß der Transkription der Gene (DNA) entstehen. Das heißt, ESTs sind kurze Wörter, ebenfalls gebildet aus einem 4-buchstabigem Alphabet, $\{A,G,U,C\}$. Derzeit sind laut EMBL Statistik [EStat] ungefähr $10,6$ Gbp an EST Nukleotiden in den öffentlichen Datenbanken gespeichert.

Eine Fülle weiterer genomischer Stringdaten wird durch das 'Genotyping' entstehen. Um die molekularen Ursachen komplexer Phänotypen einer Spezies zu erkennen, versucht man geringe genomische Variationen unter vielen Individuen dieser Spezies zu entdecken. Diese genomischen Variationen drücken sich in SNPs aus. SNPs (Single Nucleotide Polymorphisms) sind einzelne Positionen in der DNA, welche Unterschiede in den Nukleotiden in Bezug zu verschiedenen Individuen aufweisen. Derzeit sind mehr als 2 Millionen menschlicher SNPs identifiziert (siehe [Ken03]).

Die Hauptziele der Genomforschung sind: Die Analyse und der Vergleich von Genomen unterschiedlicher Spezies, genannt 'Comparative Genomics', um das Verhältnis der Verwandschaft zwischen den Arten zu klären. Der Vergleich der Variation des Genoms in ein und derselben Spezies, um die Ursachen unterschiedlicher Ausprägungen der Phänotypen zu verstehen, und damit auch die Ursachen von Krankheiten zu finden. Dafür ist unter anderem zu klären, was tatsächlich die kodierenden Bereiche in einem Genom sind, und welche Funktionen die einzelnen Abschnitte erfüllen. Das ist das Ziel der 'Functional Genomics'.

Aus diesen Zielrichtungen der Genomforschung ergeben sich die Aufgaben der Sequenzanalyse. Die Themen der Sequenzanalyse umfassen folgende Anforderungen: die Suche ähnlicher Teilwörter innerhalb verschiedener DNA Sequenzen ([Alt90], [Alt97], [Ken02]); paarweises und mehrfaches Alignment ([Pei03]), global oder lokal; die exakte oder geringfügig inexakte Patternsuche ([Bur99]); die Suche nach sich identisch wiederholenden Teilabschnitten ([Ben99]); sowie zahlreiche Methoden zur Genvorhersage ([Bes01], [Sta03]). Eine Vielzahl weiterer Analysen wurden übersichtsartig in [Gus97] beschrieben.

Diese Methoden der Suche basieren alle auf einer guten Stringverarbeitung. Bei gegebenen fast statischen Texten können sie durch eine Vorverarbeitung (Preprocessing) des Textes in der Abarbeitungszeit verbessert werden. Resultat eines solchen Preprocessing ist ein Index. Ein Index ist eine Datenstruktur, welche die wesentlichen Merkmale des Textes bei einer Suchanfrage für den Algorithmus schon bereitstellt. Im günstigstem Fall brauchen nur noch die Parameter der Anfrage 'online' berücksichtigt zu werden.

Bei obigem prognostiziertem exponentiellem Wachstum des Datenvolumens, in Form von DNA, ESTs und SNPs, und bei zunehmender Datenlänge, zum Beispiel von Genomen, ist eine schnelle Stringanalyse *im herkömmlichen Sinne* weder mit einem Preprocessing noch ohne zu gewährleisten. Das Hauptproblem besteht darin, daß derzeitige Datenstrukturen in ihrer Anwendung, als auch bei ihrer Konstruktion, einen völlig willkürlichen Zugriff auf jede Speicherposition benötigen. Das bedeutet, sie benötigen

'random access'. Damit sind sie allein auf eine Hauptspeichernutzung zugeschnitten. Für die Algorithmen der Stringanalyse, welche über diesen Datenstrukturen operieren, als auch für die Algorithmen der Stringanalyse, welche ohne Preprocessing arbeiten, gilt das gleiche.

Doch das in dieser Anwendung gegebene Datenvolumen selbst, hat schon eine Größenordnung weit jenseits gebräuchlicher Hauptspeichergrößen erreicht. Erwägt man ein wesentlich langsameres Speichermedium mit zu benutzen, so verkürzt je nach der Art des Mediums ein Index die Bearbeitungszeit der Analyse wesentlich *bzw.* macht zusammenhängende Analysen über dem gesamten Datenvolumen erst möglich.

Die vorliegende Arbeit stellt eine Datenstruktur vor, die auf dem Sekundärmedium liegt und für den Gebrauch von Suchalgorithmen der exakten Teilstringsuche optimiert wurde. Da die Suche nach ähnlichen Pattern auf der exakten Suche basiert, ist die Nutzung dieses Index für die zentrale Fragestellung der biomolekularen Analyse, nämlich der vergleichenden DNA Sequenzanalyse, geeignet. Und somit in 'Comparative' und 'Functional Genomics' einsetzbar.

Das Hauptproblem besteht neben dem Design für ein Sekundärmedium, vor allem in der Möglichkeit der Konstruktion. Wie oben erwähnt, benötigen bisherige Konstruktionsalgorithmen 'random access' (siehe in [Wei73], [MCr76], [Ukk95]). Dieses Kriterium ist wesentlich, und läßt sich nicht einfach durch einen 'serial access', wie er vom Sekundärmedium fokussiert wird, austauschen.

Aufbau der Arbeit:

Kapitel 2 gibt eine Einführung in bisher gebräuchliche Datenstrukturen und stellt grundlegende Definitionen bereit. Es wurden die wichtigsten Suchoperationen bzgl. DNA Sequenzen beschrieben, und daraus die Anforderungen an die Beschaffenheit des Index abgeleitet. Da die Indexierung von Texten bei weltweit immer größer werdender Datenmenge ein Problem darstellt, werden die unterschiedlichen Ansätze schon existierender Datenstrukturen erläutert, als auch die Forschungsbestrebungen zu deren Verbesserung.

In dieser Arbeit wird eine neuere Bestrebung verfolgt, nämlich die Ausnutzung mehrerer Speichermedien – insbesondere Speichermedien, welche nach derzeitigem Stand unbegrenzt verfügbar sind. Es wird von Anfang an davon ausgegangen, daß gerade bei der Fülle an Daten in den Life Sciences, eine Analyse mit begrenzter Speicherkapazität nicht mehr möglich ist und Minimierungsstrategien die Erhöhung des Datenvolumens nicht auffangen können.

In Kapitel 3 werden Darstellungstechniken einer solchen Baumstruktur entworfen. Sowohl in abstrakter Hinsicht zur Unterstützung einer optimalen Stringanalyse, als auch zum Erreichen einer minimalen Repräsentation. Das heißt, redundante Elemente werden herausgestrichen, insbesondere in Bezug zur Verwendung eines 4 elementigen Alphabetes. Desweiteren wird deren tatsächliche Implementierung beschrieben. Da der Index Disk-resident ist, wird seine Implementierung in Hinblick auf einen möglichst geringen Diskzugriff optimiert.

Kapitel 4 faßt verschiedene Möglichkeiten zusammen, einen solchen Index zu konstruieren. Es werden bisher bekannte in-memory Algorithmen vorgestellt, und ihre Komplexität für lange Sequenzen angegeben. Desweiteren werden prinzipielle Richtungen angegeben, wie sich für lange Sequenzen die in Kapitel 3 entworfene Graphenstruktur in einzelnen Teilschritten aufbauen läßt. Ich entwerfe hier drei verschiedene Szenarien einen Ansatz zu machen. Es wird die Selektive Partitionierung, die Präfix Partitionierung und die Datenpartitionierung unterschieden.

Es wird bewiesen, daß man beim Ansatz der Selektiven Partitionierung theoretisch die bestmögliche obere Schranke erhalten kann. Ein Baum läßt sich in linearer Zeit konstruieren. Bei dem Beweis wird die Tatsache ausgenutzt, daß Suffixe eines Strings, welche in einem festen Abstand zueinander stehen, auf ihren jeweiligen wohldefinierten Vorgänger verweisen können. Das heißt, zum Aufbau eines Baumes über einer solchen selektiven Suffixmenge läßt sich ein linearer Algorithmus verwenden. Aber auch zum Postprocessing der entstandenen Bäume kann durch mehrere Scans, aufgrund des festen Abstandes, eine lineare Zeit erreicht werden. Dieser Beweis beruht auf einer wesentlichen Verallgemeinerung der Beweisidee eingeführt von [Far97]. Es wird weiter erläutert, warum dieser Ansatz in der Praxis nicht anwendbar ist, obwohl wir theoretisch ein bestmögliches Ergebnis erhalten.

Der zweite Ansatz der Präfix Partitionierung liefert hingegen schon theoretisch kein befriedigendes Ergebnis. Kapitel 4 motiviert den in dieser Arbeit eingeführten neuen Konstruktionsalgorithmus, welcher den Ansatz der Datenpartitionierung verfolgt.

In Kapitel 5 wird die Methode ausführlich beschrieben und in den einzelnen Teilschritten analysiert. Die Grundidee des neuen Algorithmus besteht in der Teilung der Datensequenz und dem anschließenden Mischen der resultierenden Bäume.

Im ersten Schritt wird eine modifizierte Variante des Algorithmus von Ukkonen eingeführt. Trotz dieser Änderung bleibt dieser Schritt in der Zeit linear, da die einzelnen Suffixpartitionen nur hintereinanderliegende Strings enthalten. Der zweite Schritt besteht aus dem Mischen der erhaltenen Bäume. Hierfür müssen die aus Schritt 1 resultierenden Bäume segmentiert und die Segmente anschließend vereinigt.

Die Wahl der Segmentierung ist entscheidend für eine minimale Zeitkomplexität. Es wird eine optimale Segmentierung definiert, als auch die Abhängigkeit der Segmentierung vom Text am Beispiel von DNA Sequenzen gezeigt. Es wird untersucht, für welche Segmentierungsparameter die Anzahl der “inneren” Mischvorgänge minimal ist. Es werden die bestmöglichen Parameter im Fall von DNA Sequenzen angegeben.

Aus diesem bisherigen Vorgehen resultiert, daß die Komplexität der inneren Mischprozedur nicht linear sein kann. Es wird jedoch gezeigt, daß im Fall von DNA Sequenzen die Komplexität ’durchschnittlich’ von der Ordnung $O(n \log n)$ ist.

Diese Arbeit fokussiert nicht nur auf der Gewinnung einer theoretischen Verbesserung der Komplexität bezüglich einer neu gewonnenen Methode, sondern auf die Möglichkeit der praktischen Umsetzung, welche in vielen Referenzarbeiten nicht gegeben ist und auch nicht versucht wurde.

Die Implementierung des eingeführten Konstruktionsalgorithmus bestätigt, daß der vorgestellte Algorithmus für die angestrebten Textlängen anwendbar ist und damit weit hinter den bisher genutzten Varianten skaliert. Auf diese Weise wird der tatsächliche Aufbau eines persistenten Indexes für relevante, anwendungsnahe Größenordnungen erst ermöglicht. Die umfangreichen Experimente unter gleichen Bedingungen mit Referenzalgorithmen, wie *wotd*, Enhanced Suffix Array oder Ukkonen's Algorithmus, dokumentieren dies. Auch Vergleiche mit Veröffentlichungen auf diesem Gebiet bestätigen, daß derzeit keine performantere Methode für diese Fragestellung existiert.

Kapitel 2

Datenstrukturen zur Unterstützung einer schnellen exakten Suche

Bei Datenstrukturen über Texten *oder* Indizes unterscheidet man Wortindizes und Volltextindizes. Im Falle von Wortindizes wird der Text als eine Sequenz betrachtet, deren kleinste syntaktische Einheit ein Wort ist. Daher wird jede Position eines Wortes im Text ausgezeichnet, das heißt indexiert. Beispiele sind die Inverted List [Bae99], String-B-Tree [Fer99] oder Signature Files, eingeführt in [Fal84]. Der Index ist optimiert für Anfragen nach einem oder mehreren Wörtern, und wird daher vorwiegend im 'Document Retrieval' eingesetzt. Bekannteste Vertreter hierfür sind die Suchmaschinen (z.B. Google) im World Wide Web.

Die Teilwort- oder Ähnlichkeitssuche, wie zum Beispiel bei der Rechtschreibkorrektur benötigt, ist aufwendiger. Für solche Anwendungen sind Volltextindizes besser geeignet. Desweiteren gibt es Texte, die sich gerade nicht in Wörter unterbrechen lassen, zum Beispiel Video- oder DNA Sequenzen. Im Fall von DNA Sequenzen könnte man jedes Chromosom als ein Wort betrachten. Ein ganzes Genom wäre dann ein Text mit sehr wenigen, dafür sehr langen Wörtern.

Bei einem Volltextindex wird jede Position *innerhalb* eines Wortes als Indexierungsparameter betrachtet, ebenso die Leerzeichen dazwischen. Das heißt, jeder Buchstabe und jedes Leerzeichen werden indexiert.

2.1 Grundlegende Notationen

Sei Σ ein endliches Alphabet von fester Größe. Eine Sequenz von Buchstaben aus Σ nennt man Text *oder* String über dem Alphabet Σ. Die Menge aller Strings über Σ wird mit Σ^* bezeichnet. Sei $t \in \Sigma^*$, dann ist $|t|$ die Länge des Textes. Der leere String hat die Länge 0 und wird mit ϵ bezeichnet. $\Sigma^m \subset \Sigma^*$ ist die Menge aller Strings $t = t_1 t_2 t_3 \cdots t_m, t_i \in \Sigma$ mit der Länge m. Sei $t^+ = t\$$ die Erweiterung des Strings t durch ein Zeichen, welches kein Element des Alphabetes ist, hier \$. Für $1 \leq i \leq n+1$, sei $s_i(t^+) = t_i \cdots t_n\$$ der i-te nicht-leere Suffix von t^+. Ein Teilstring aus t^+ wird notiert durch $t(i, j) = t_i t_{i+1} \ldots t_j \in \Sigma^*$.

Konvention: In dieser Arbeit bezeichnen a, b, c, x, y Buchstaben aus dem Alphabet Σ, wohingegen s, t, u, v, w Elemente aus Σ^* sind. Für einen gegebenen Text $t \in \Sigma^*$ wird das i-te Zeichen des Textes durch $t_i \in \Sigma$ notiert.

2.2 Definition Baumartiger Datenstrukturen

Sei $\mathcal{S} \subset \Sigma^* \setminus \{\epsilon\}$ eine gegebene Menge von Strings.

Definition: (*Allgemeiner Baum - Trie*)
Ein Trie $\mathcal{T}(\mathcal{S})$ ist ein Wurzelbaum, dessen Kanten mit Teilstrings aus $\mathcal{S}$ beschriftet sind. Jeder Knoten in $\mathcal{T}$ hat höchstens eine ausgehende Kante, dessen Beschriftung mit dem Buchstaben $a \in \Sigma$ startet. Jedem Blatt von $\mathcal{T}(\mathcal{S})$ kann eineindeutig ein String aus $\mathcal{S}$ zugeordnet werden.

BEMERKUNG: Ein Knoten mit mindestens einer ausgehenden Kante wird innerer Knoten genannt, andernfalls ist es ein Blattknoten.

Gegeben sei ein Text t über dem Alphabet Σ der Länge n.

Definition: (*Suffix Baum - Suffix Tree*)
Ein Suffix Tree $\mathcal{T}_S(t)$ über einen Text t ist ein Trie über der Menge aller Suffixe $\mathcal{S} = \mathcal{S}(t) := \{s_i(t^+)^1 : 1 \leq i \leq n+1\}$, für welchen gilt: Jeder innere Knoten muß mindestens zwei ausgehende Kanten besitzen.

Konvention: Wenn nicht anders definiert, wird im folgenden ein Suffix Tree kurz mit $\mathcal{T}(t)$ bezeichnet, statt $\mathcal{T}_S(t)$.

$\mathcal{V}_{\mathcal{T}(t)}$ bezeichnet die Menge der Knoten und $\mathcal{E}_{\mathcal{T}(t)}$ die Menge der Kanten von $\mathcal{T}(t)$. R bezeichne den Wurzelknoten. Jeder innere Knoten eines Suffix Trees ist ein Verzweigungsknoten, jede Kante ist beschriftet mit einem nicht-leeren Teilstring von t^+. Die ausgehenden Kanten jedes inneren Knotens haben unterschiedliche Anfangsbuchstaben. Jedes der $(n+1)$ Blätter ist eindeutig mit einem Index i, $1 \leq i \leq n+1$ bezeichnet, wobei sich durch das Zusammenfügen der Kantenbezeichnungen von der Wurzel bis zum Blatt der String $s_i(t^+)$ ergibt. Der Pfad $path(u) \in \Sigma^*$ eines Knotens $u \in \mathcal{V}_{\mathcal{T}(t)}$, ist definiert als die gerichtete Vereinigung der Kantenbeschriftungen (Konkatenation) von der Wurzel des Baumes $\mathcal{T}(t)$ bis zum Knoten u. Wir bezeichnen u durch $\overrightarrow{w}$ dann und nur dann, wenn $path(u) = w$ ist. Für jeden Knoten $\overrightarrow{w}$ in $\mathcal{T}(t)$ bezeichnet $depth(\overrightarrow{w})$ die Tiefe von $\overrightarrow{w}$ im Baum. Das ist die Anzahl der inneren Knoten von R bis $\overrightarrow{w}$. Ein Knoten $u \in \mathcal{V}_{\mathcal{T}(t)}$ gehört zum Level *bzw.* zur Schicht i, falls $depth(u) = i$. Sei $u \in \mathcal{V}_{\mathcal{T}(t)}$, mit $deg(u)$ wird die Anzahl der Geschwisterknoten bezeichnet.

Das größte Problem bei der Anwendung von Suffix Trees ist ihr enormer Platzbedarf.

[1] auch einfach s_i bezeichnet

Grundsätzlich besteht ein allgemeiner Baum aus 2 Komponenten:

- der Tree Topologie, das ist die Menge aller Knoten und Kanten $\mathcal{V}_{\mathcal{T}(t)} \cup \mathcal{E}_{\mathcal{T}(t)}$;
- und der Kantenbeschriftung.

Diese 2 Bestandteile müssen in Bezug auf die Speicherplatz Komplexität optimiert werden.

Die Tree Topologie wird ausgedrückt durch Pointer Strukturen oder Skip + Branching Faktoren. Die Kantenbeschriftung kann ausgedrückt werden durch Teilstrings aus dem Text oder durch String Pointer in den Text.

Das heißt, es müssen folgende 3 Komponenten so minimal wie möglich dargestellt werden:

1. der Text,
2. Skip + Branching Faktoren,
3. String Pointer in den Text.

Es gelten, als untere Schranken des Platzbedarfes: Der Text t der Länge n läßt sich verschlüsseln in $n \lg |\Sigma|$ Bits. Die Tree Topologie ist definiert als die Menge der Eltern-Kind-Beziehungen der Knoten eines Baumes und besteht aus $O(n)$ Zeigern (Pointern). Diese lassen sich mit $O(n \log n)$ Bits darstellen. Die jeweiligen Kantenlabel werden durch Pointer in den Text dargestellt und mit $n \log n$ Bits repräsentiert. Die Menge aller Blätter benötigt ebenfalls $n \log n$ Bits.

Bekannte Datenstrukturen:

Allgemein bekannt als Datenstrukturen für Volltextindizes sind das Suffix Array, eingeführt von [Man93], der Suffix Tree von [MCr76], der Suffix Binary Search Tree [Irv03] und der AVL Tree [Irv03]. All diese Strukturen sind baumartig und unterscheiden sich nur in der Verzweigungsvielfalt und der Pfadkodierung (Kantenbeschriftung). Ausführlich beschrieben sind diese Strukturen auch in [Bae99]. Als Verbindung zwischen einem Tree und einem Array, um Vor- und Nachteile beider Strukturen gegeneinander auszuwägen – der Suffix Cactus [Kär95]. Da Volltextindizes im allgmeinen einen hohen Platzbedarf haben, wurde mit dem Suffix Vektor [Mon02] versucht, diesen zu minimieren. Dies wird durch das Wegstreichen redundanter Pfade erreicht. Weitere Reduktionsstrategien liegen in der Anwendung von Kompressionsalgorithmen auf den Text oder auf den gesamten Index, oder auf beides. Der erste Beitrag in dieser Richtung kam von [Gro99], welcher das Compressed Suffix Array und den Compressed Suffix Tree einführte. Weiter verfolgt und verbessert wurde diese Struktur durch [Sad00], [Fer00], [Sad02]. In die gleiche Richtung ging auch [Nar02] mit der Einführung des Ziv-Lempel Trie.

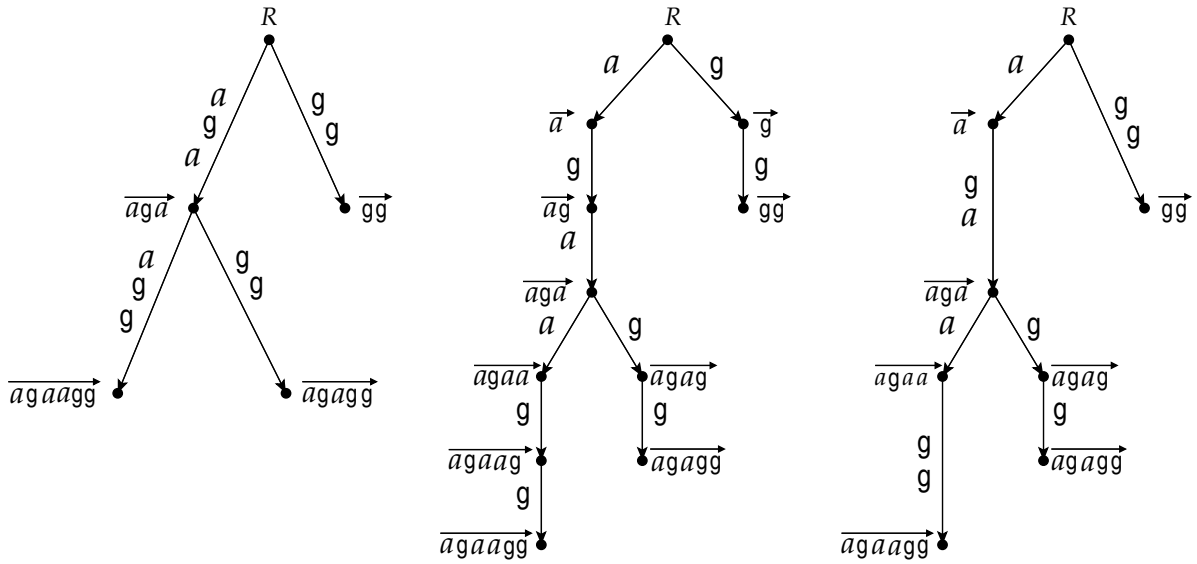


Abbildung 2.1: Unterschiedliche Varianten eines Tries über einer Menge von Strings $\mathcal{S} = \{agaagg, agagg, gg\}$

2.3 Suchoperationen

Die schnelle exakte Suche eines Patterns *bzw.* Musters ist definiert durch:

Definition: Sei $p = p_1 \dots p_m$ ein Pattern und $t = t_1 \dots t_n$ eine Textdatenbank. Das Problem des *Exact String Matching* besteht im Finden der Textposition i, wobei $t(i, i + m - 1) = p$.

Bekannte Algorithmen des Exact String Matching ohne Index, genannt 'On-line' Exact String Matching, sind von Knuth, Morris und Pratt [KMP77], Boyer und Moore [BM77] oder Apostolico und Giancarlo [AG86]. Diese Algorithmen haben die untere Schranke $O(n/m)$ und im 'worst case' eine Komplexität von $O(n)$.

Beim Indexed Exact String Matching wird der Text vorverarbeitet. Die Struktur und der Aufbau des Textes fließen in eine Datenstruktur, genannt Index. Das Wissen des Index verringert die Komplexität der Suche. Ist der Index zum Beispiel ein Suffix Tree, so hat das Exact String Matching Problem eine Komplexität von $O(m)$. In diesem Fall konnte die Komplexität der Suche unabhängig von der Länge des Textes gemacht werden. Bei sehr langen Texten, wie unter anderem DNA Sequenzen, und wesentlich kürzeren Anfragepattern, ist dies notwendig und erwünscht. Für einfache Suffix Arrays ([Man93]) gilt dies mit $O(m \log n)$ nicht.

Damit sich der Aufbau eines Index gegen eine sequentielle, 'on-line' Suche rentiert, muß erstens der Text groß genug sein und zweitens der Text in einem gewissen Sinne statisch sein. Das Verhältnis von Änderungen am Text und Anfragen an den Text muß beachtet

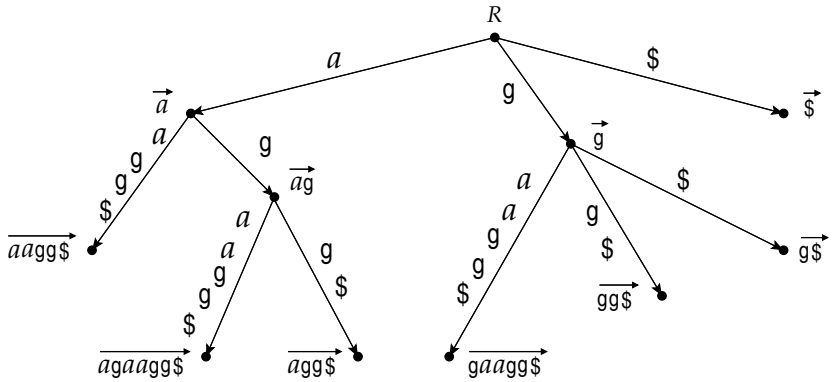


Abbildung 2.2: Suffix Tree für den Text $t^+ = agaagg\$$

werden. Es gilt, je statischer der Text und umso mehr Anfragen, desto sinnvoller ist die Konstruktion eines Indexes.

Die bisherigen vorherrschenden Bewertungskriterien zur Beurteilung über den Nutzen eines Indexes, sind Speicherplatzbedarf und Zugriffsgeschwindigkeit. Die allgemeine Regel ist, je schneller der Zugriff, um so größer der Speicherplatzbedarf. Dies ist um so wichtiger, falls man nur Indizes betrachtet, welche größtenteils im Hauptspeicher liegen. Im allgemeinen sind Volltextindizes sehr platzintensiv.

2.4 Persistente Datenstrukturen

Die oben betrachteten Strukturen wurden meist für den Gebrauch im Hauptspeicher entwickelt. Die typische Anwendungssituation bestand aus einem mittellangen gegebenen Text. Der Index war innerhalb weniger Minuten konstruiert (in der Praxis nur Sekunden) und wurde dann für mehrere Anfragen (Query Packs) verwendet. Die Abarbeitungszeit für mehrere Anfragen und die Konstruktionszeit zusammengerechnet, war geringer als die gleiche Anzahl an Suchanfragen mit einem 'online' Algorithmus bearbeitet. Da der Index nur im Hauptspeicher liegt, wurde er meist bei einer anderen Situation verworfen oder gelöscht, falls der Computer zum Beispiel heruntergefahren oder das Programm abgebrochen wurde. Daher nennt man solche Strukturen *transient*.

Indizes, welche nicht nur für eine einmalige Reihe von Suchoperationen kreiert, zu diesem Zweck auf der Festplatte gespeichert und bei Bedarf geladen werden, nennt man *persistent*. Solche *persistenten* Datenstrukturen werden vorwiegend bei Datenbanken zur Unterstützung der Suche eingesetzt. Denn, die zu indexierenden Daten, sind hier meist so groß, daß sie auf externen Speichermedien gehalten werden müssen. Indizes, die das Datenvolumen um ein mehrfaches überschreiten, können erst recht nur extern gehalten werden.

Die meisten konventionellen, existierenden Datenbanksysteme folgen dem relationa-

lem *bzw.* objekt-relationalem Paradigma. Die Suche wurde optimiert, um einzelne Einträge in Relationen (Tabellen) zu finden. Das können Zahlen oder ganze Wörter sein. Texte werden als atomare Einheiten gesehen und im Datentyp CLOB gehalten. Sie werden nur über wenige Identifizierer aus dem Titel mit in einen Index einbezogen (siehe die Grundlagen dazu in [Sil01]).

Anders Textdatenbanken, definiert gerade als Systeme, welche auf großen Mengen an Textdaten einen schnellen Zugriff auch bei anspruchsvollen Analysen im Text sicherstellen. Zu solchen Texten zählen neben natürlichsprachlichen Texten (in Büchern, in Zeitungen, oder als semi-strukturierte Daten) und biomolekulare Sequenzen, wie in Kapitel 1 beschrieben, auch jede Form von fortlaufenden Signalen, wie sie zum Beispiel bei Zeitfunktionen oder Audio-Sequenzen auftreten.

Indizes für Textdatenbanken zur Unterstützung der Teiltextsuche in diesen, sind zur Zeit Gegenstand der Forschung. Prototypen bzw. Implementierungen hierzu existieren wenige. Die Erfordernisse wurden von [Nav04] in einem Übersichtsartikel aus dem Jahr 2004 zusammengetragen – unter anderem gefordert sind Mechanismen zur Auslagerung eines solchen Indexes in den Sekundärspeicher. Dieser ist billig und damit fast unbegrenzt verfügbar. Weiterhin läßt sich davon ausgehen, daß die Menge an Information, welche in Texten gespeichert ist, in den nächsten Jahren exponentiell anwächst und dies allein durch eine Größenzunahme des Hauptspeichers nicht abgefangen werden kann. Die Texte werden zwar auch in der zusammenhängenden Länge wachsen, in erster Linie wird jedoch ein exponentieller Anstieg in der Textanzahl erwartet.

Persistente baumartige Datenstrukturen sind, unter vielen, der String-B-Tree von [Fer99] und Index Fabric [Coo01]. Ersterer wurde entwickelt für Texte, die sich in Wörter unterteilen lassen. Letzterer agiert über semi-strukturierten Daten, zum Beispiel XML. Der String-B-Tree ist ein B-Tree, bei welchem in jedem Blatt ein Wort gespeichert und durch einen Patricia Tree kodiert wird. Index Fabric stellt eine Schichtungsvariante eines Patricia Trees dar (siehe [Coo01]).

Weitere Ansätze, insbesondere auf dem Gebiet der Genominformatik, finden sich in [Bur99], [Kah01], [Kah03], [Nar00], oder als Erweiterung eines Suffix Arrays, das Enhanced Suffix Array von [Abo02].

Zielrichtung

In dieser Arbeit wird eine Variante eines Suffix Trees vorgestellt. Hierbei liegt das Hauptaugenmerk in erster Linie nicht in der Ausnutzung bestimmter Kompressions- und Reduktionsstrategien, mit dem Ziel für immer größer werdende Texte eine minimale Hauptspeicherdarstellung zu erhalten. Sondern, es wird von Anfang an davon ausgegangen, daß dies bei der Fülle an Daten nicht möglich ist. Es wird eine platzsparende Repräsentation auf der Festplatte gesucht. Diese wird im nächsten Kapitel vorgestellt. Neben dieser Darstellung ist das eigentliche wissenschaftliche Problem die Konstruktion einer solchen Datenstruktur (Kapitel 4).

Kapitel 3

Repräsentation

Gegeben sei ein beliebiges endliches Alphabet Σ.

In diesem Kapitel werden Implementierungstechniken eines persistenten Suffix Trees beschrieben. Gesucht ist eine Darstellungsvariante,

- die es einerseits erlaubt, einen Suffix Tree auf einem relativ langsamem Speichermedium (es wird normalerweise die Festplatte betrachtet) überhaupt darzustellen,
- und andererseits, eine bezüglich verschiedener Speichermedien möglichst unabhängige Repräsentation ist. Das bedeutet, die Darstellung soll auch auf schnelleren Medien performant sein.
- weiterhin Speicherplatz effizient ist,
- Zugriffszeiten bezüglich der Grundoperationen des Suchens minimiert, unter der Voraussetzung, daß sowohl die Datenstruktur als auch der Text selbst auf einem langsamen externen Speicher liegen,
- aber ebenso einen quasi-linearen Konstruktionsalgorithmus für einen persistenten Tree ermöglichen (siehe Kapitel 4).

Im weiteren wird zuerst besprochen, wie die theoretische Struktur eines solchen Index aussehen muß, damit sich ein optimales Suchverhalten auf der Festplatte ergibt.

Danach werden konkrete Implementierungstechniken erläutert, die eine weitgehend Speicherplatz sparende Abbildung ermöglichen.

3.1 Logische Darstellung

In diesem Kapitel geht es darum, das Design eines Trees auf ein langsames Hintergrundmedium auszurichten. In meinem betrachteten Fall ist dies die Festplatte. Wie im

Kapitel 2.4 beschrieben, läßt sich die Annahme machen, daß man ein von der Größe her unbegrenztes Speichermedium hat, welches im Zugriff langsam ist.

Die Anzahl der Knoten $|\mathcal{V}_{\mathcal{T}(t)}|$ eines Suffix Trees $\mathcal{T}_S(t)$ wird zusammen mit der Anzahl der Kanten $|\mathcal{E}_{\mathcal{T}(t)}|$ als die Größe von $\mathcal{T}_S(t)$ bezeichnet, kurz $|\mathcal{T}_S(t)|$. Es gilt die Größe von $\mathcal{T}_S(t)$ zu minimieren. Wenn sich auch davon ausgehen läßt, daß unbegrenzter Speicherplatz zur Verfügung steht, so gilt es doch die gesamte Zugriffszeit minimal zu halten, das heißt die Anzahl Zugriffe zu reduzieren. Die kleinste Einheit eines Zugriffes auf die Festplatte ist eine Speicherseite. Je kleiner der Tree, um so weniger Speicherseiten belegt dieser. Und um so weniger Speicherseiten müssen bei der Suche in den Hauptspeicher geladen werden. Hierfür sind alle Komponenten, aus denen sich der Baum zusammensetzt und deren Darstellungen zu reduzieren.

1. Reduktion der Pfade

Die Reduktion eines Pfades ergibt sich schon aus der Definition eines Suffix Trees. Innere Knoten mit nur einem Kind werden zusammengefaßt und mit der Konkatenation ihrer Kantenbeschriftungen ausgezeichnet. Auf diese Weise hat man die kompakteste Form eines allgemeinen Tries.

Eine weitere Reduktion ergibt sich aus der Tatsache, daß bei einem gegebenen Text nicht die ganzen Suffixe in den Index einzufügen sind, sondern nur die Präfixe der Suffixe, welche jeden Suffix in der Menge aller Suffixe eindeutig darstellen.

Definition : Sei s ein beliebiger Suffix eines Textes t^+. Der *Präsuffix* von s ist $presuffix(s) := w$, falls w ein Präfix von s und w kein Präfix von $s' \in \mathcal{S} \setminus \{s\}$.

In den meisten Fällen gilt $|s| \gg |presuffix(s)|$. Sei $\mathcal{S}' = \{presuffix(s) : s \in \mathcal{S}\}$. Dann ist $|\mathcal{T}_S| \gg |\mathcal{T}_{S'}|$.

Beispiel:

Sei der Text $t^+ = agaagg\$$ gegeben, so lauten die jeweiligen Präsuffixe und der dazugehörige Suffix Tree mit reduzierten Pfaden:

$presuffix(agaagg\$) = ag \in \Sigma^* \cup \{\$\}$
$presuffix(gaagg\$) = ga$
$presuffix(aagg\$) = aa$
$presuffix(agg\$) = ag$
$presuffix(gg\$) = gg$
$presuffix(g\$) = g\$$
$presuffix(\$) = \$$

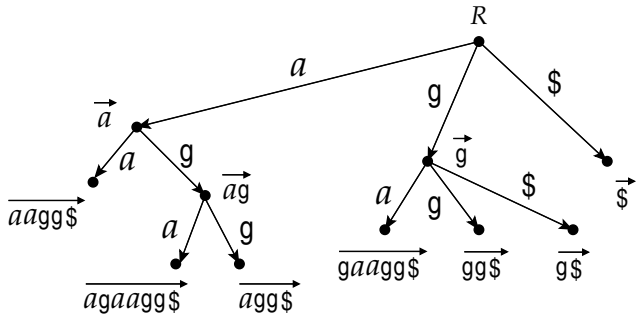


Abbildung 3.1: Suffix Tree mit reduzierten Pfaden für den Text $t^+ = agaagg\$$

---

2. Reduktion der Tiefe

Sei k die Größe des betrachteten Alphabetes. Sind die i-ersten Schichten ab der Wurzel

eines Baumes komplett belegt, so nennt man dies eine maximale Belegung *bis zur Stufe i*. Dann gilt für alle $u \in \mathcal{V}_{T(t)}$ mit $depth(u) \leq i$, daß $deg(u) = k$ ist. Der Teil eines Baumes für dessen Knoten u gilt $depth(u) \leq i$ wird 'Trunk' des Baumes bis zur Stufe i bezeichnet. Bei einer maximalen Belegung ergibt sich die Größe des Trunkes mit $k^1 + k^2 + \ldots + k^i$. Durch das Streichen aller inneren Knoten bis zur Ebene i läßt sich die Anzahl der Knoten auf k^i reduzieren. Die Tiefe des Trunks wird damit von i auf 1 herabgesetzt. Diese Ersetzung läßt sich für jeden inneren Knoten u durchführen, für welchen eine maximale Belegung der direkt darunter liegenden Schichten existiert. Auf diese Weise erhalten wir eine Struktur, welche sich der Verteilung der Buchstaben des Eingabetextes perfekt anpaßt. Das heißt, durch eine Umkodierung des Alphabetes erreicht man einen ausbalancierteren Baum.

3. Reduktion, der zu indexierenden Textgröße

Ein gegebener Text wird gewöhnlich im ASCII-Code dargestellt. Jeder Buchstabe benötigt 8 Bit. Bei den meisten Anwendungen ist es unwahrscheinlich, daß alle 256 Zeichen tatsächlich genutzt werden. Und wenn dieser Fall trotzdem eintritt, dann haben nicht alle Buchstaben die gleiche Häufigkeit. Die überfälligen Bits im ASCII-Code der Kantenmarkierungen erhöhen die Suchzeit. Daher sind Strategien zur Datenkompression sinnvoll, um schon die Eingabedaten geeignet zu reduzieren. Die wohl bekannteste und einfachste Methode ist die Huffmann Codierung.

Anwendung dieser Strategien auf DNA Sequenzen

Diese drei Strategien lassen sich auf Tries über beliebigen Texten anwenden. Es folgt eine Bewertung für DNA Sequenzen:

Erstens: Die geringe Größe des Alphabetes mit 4 Buchstaben bewirkt, daß viele Schichten mit einer maximalen Belegung existieren. Das heißt, die Tiefe einer Variante eines Suffix Trees reduziert sich durch Strategie 2 wesentlich. Bei einer DNA Sequenz von ungefähr 33 MB zeigt die Erfahrung, daß die ersten 6 Level vollständig belegt sind. Bei 220 MB sind es meist die ersten 10 Level. Desweiteren sind gerade Teilbäume unterhalb jeglicher A-Pfade häufig maximal belegt. Daher verringert sich auch bei ausladenden Teilbäumen, innerhalb des Baumes weit unterhalb der Wurzel, noch einmal das Knotenvolumen.

Zweitens: Eine DNA Sequenz benötigt pro Buchstabe nur 2 Bits zur Kodierung. Diese Annahme bezieht sich auf bereinigte DNA Sequenzen, die aus dem Alphabet $\{A, T, G, C\}$ bestehen. Ungenau sequenzierte Strings werden aus dem Alphabet $\{A, T, G, C, N\}$ gebildet. Das N steht für eine Base, die bisher experimentell noch nicht identifiziert werden konnte. Hier werden dann höchstens 3 Bits benötigt.

Aber in der Praxis können und werden DNA Sequenzen mit noch größeren Alphabeten beschrieben. Für die Base N, welche experimentell zwar nicht bestimmbar ist, läßt sich in einigen Fällen angeben, daß sie nur die Basen A oder T annehmen kann. Daher können auch Alphabete der Form $\{A, T, G, C, A|T, T|G, \ldots, N\}$ auftreten.

Die erste Strategie bestand in der Minimierung der Anzahl der Knoten. Die zweite Strategie besteht in der Minimierung der Kantenbeschriftung.

Minimierung der Pfadlänge

Eine Besonderheit bei DNA Sequenzen sind sich identisch wiederholende Teilstrings, welche sehr lang sein können. Im menschlichem Genom können solche Teilstrings bis zu 150 Basen betragen. Das würde im schlechtesten Fall Kantenmarkierungen von 150 Byte ergeben.

Definition: Sei $w \in \Sigma^*$. Es ist $\mathit{suffixpos}(w) := i$ für ein gewisses $i \in [1, n+1]$, für welches w ein Präfix vom Suffix s_i ist.

Beobachtung: Sei $\overrightarrow{w} \xrightarrow{u} \overrightarrow{wu}$ eine Kante beschriftet mit dem String u, dann läßt sich u darstellen durch $u = t(i, i+\ell-1)$, wobei $i = \mathit{suffixpos}(u)$ und $\ell = |u|$.

Das heißt, String u läßt sich durch zwei Zahlen i und $i+\ell-1$ darstellen. Diese zwei Zahlen sind nicht eindeutig. In den meisten Fällen, wird für u jeweils minimales i gewählt. Das Tupel $(i, i+\ell-1)$ wird als *Referenzierung* von u auf den Text t bezeichnet. Deren Anwendung in der Datenstruktur bewirkt beim Exact String Matching einen willkürlichen Zugriff auf das Speichermedium. Daher ist auf dem Sekundärspeicher eine derartige Referenzierung nur sinnvoll, falls die Pfadlänge eine gewisse Größe überschreitet. Diese Größe ist von der Zugriffsgeschwindigkeit und der Darstellungseffizienz der Pfade abhängig. Lange repetitive Elemente, wie sie in der DNA auftreten, haben in der Praxis eine solche Länge.

Bei der hier vorgestellten logischen Darstellung einer Variante eines Suffix Trees, werden nur Präsuffixe kodiert. Die Blattknoten des Baumes sind mit den Pointer $\mathit{suffixpos}(s_i)$ in den Text beschriftet.

Weiterhin ist, für die nach Punkt 3 beschriebene Schichtenreduktion, eine Implementierung als Hash-Tabelle am sinnvollsten.

Suffix Tree aus mehreren Strings

Der Suffix Tree ist in erster Linie eine Datenstruktur, die einen kontinuierlichen Text indexiert, und es dadurch gestattet Teilwörter in diesem zu finden. Sequenzdatenbanken sind zumeist in semantische Einheiten, den Texten, unterteilt, welche die einzelnen Datenbankeinträge darstellen. Typischerweise sind Anfragen an Beziehungen zwischen den einzelnen Datenbankeinträgen gerichtet. Das Ziel ist es, einen Suffix Baum über einer Kollektion von Sequenzen aufbauen zu können. [Hui92] führt den *Generalised Suffix Tree* ein, welcher ein kompakter allgemeiner Trie über der Menge von Strings $\mathcal{S} = \{s_i \in \Sigma^* : s \in \mathcal{D}, s_i \text{ Suffix von } s, 1 \leq i \leq |s|+1\}$ ist. Hierbei ist $\mathcal{D} = \{s^1, \ldots, s^m\}$ eine Textdatenbank. Um die Positionen unterschiedlicher Strings voneinander unabhängig zu machen, werden sie mit einem eindeutigen Stringidentifikator gelabelt.

Abbildung 3.2 demonstriert die tatsächliche Darstellung einer solchen Struktur am Beispiel einer Datenbank bestehend aus mehreren Sequenzen. Da nur Präsuffixe kodiert werden, besteht der Index aus der beschriebenen Datenstruktur *und* dem Text.

DNA Sequenz 1, s1

AATGAAGGTA TGAATTGCTA AGCTTCAAAA TGCGACATGA TTAGGCCAAT
AGCAACAATT ATCGACATTT CTCGACACCT ATCGACATTT ……AAC$

DNA Sequenz 2, s2

CAAGAAGGTA CCAAAGGCTA AACGACATTT ATCGGCACAC ATAGATAAAA
GTCGACAAGG TTACACATAT ATCTGGAGTT TTAGCAAATT ……TAG$

DNA Sequenz 3, s3

……

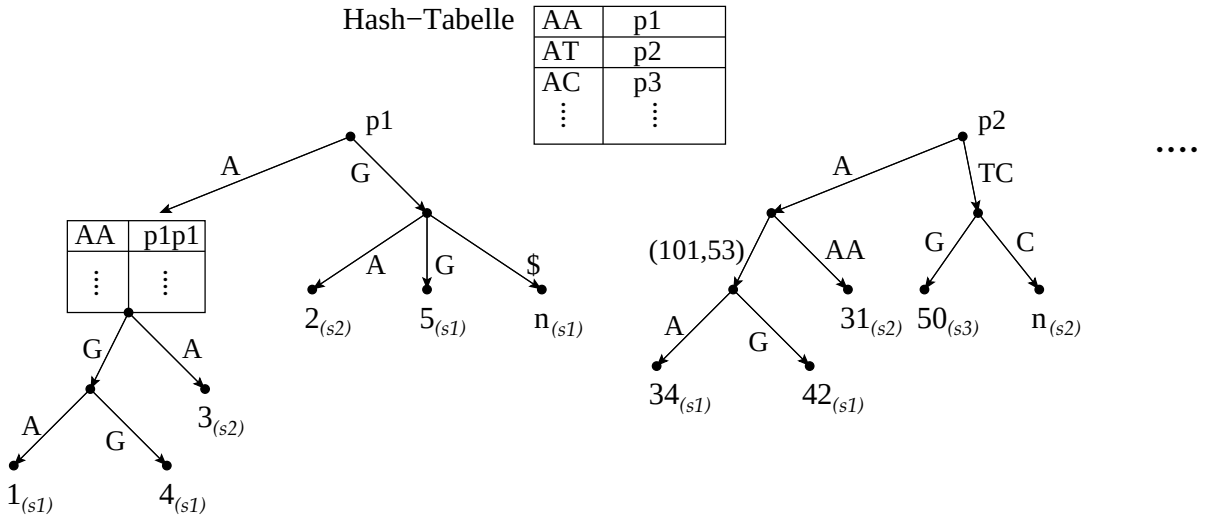


Abbildung 3.2: Beispiel der logischen Darstellung eines Baumes für mehrere Strings einer DNA Datenbank, ohne Huffmann Codierung

3.2 Physikalische Darstellung - Baumkodierungsstruktur

Für den Vergleich verschiedener Bäume wird nicht nur auf konzeptionelle Unterschiede geachtet, sondern auch auf Implementierungsvarianten. Die folgenden drei Implementierungsvarianten für Bäume sind gebräuchlich:

- Kantenbaum,
- Knotenbaum,
- Array.

Abbildung 3.3 zeigt schematisch die unterschiedlichen Implementierungsvarianten.

Das Grundelement des Kantenbaumes ist die einzelne Kante. Eine Kante beinhaltet ihre Kantenbeschriftung, und Zeiger auf ihre Nachfolger und Geschwister. In einer trivialen Implementierung ist die Größe einer Kante konstant. Platzsparender ist die kompakte Implementierung, in welcher alle Pointer mit Leerverweisen weggelassen werden.

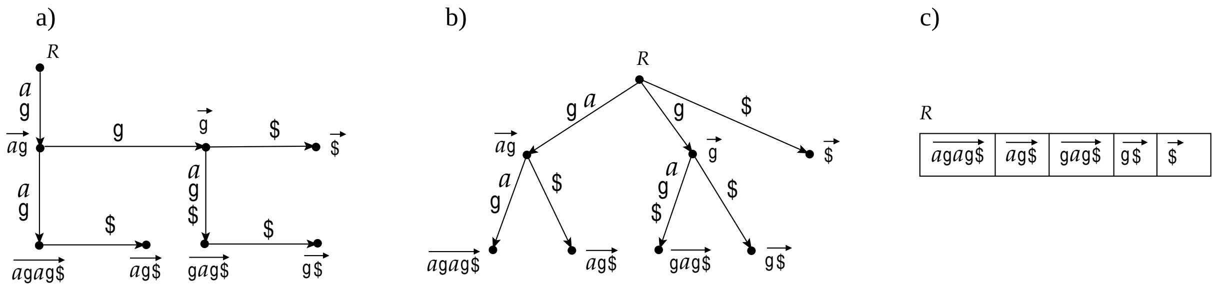


Abbildung 3.3: Darstellung der Implementierungsvarianten a) Kantenbaum b) Knotenbaum c) Array für den String *agag*$

Die Implementierungsvariante als Kantenbaum ist gut anwendbar, wenn ein Knoten im Tree wenige Nachfolger hat. Weiterhin ergeben sich Vorteile bei der Aktualisierung des Baumes. Die atomaren Einheiten, die reorganisiert werden müssen, sind hier kleiner als bei anderen Varianten. Der klassische Suffix Baum wird als Kantenbaum implementiert.

Die Implementierung als Knotenbaum folgt der logischen Darstellung eines Baumes. Es ist das intuitive Implementierungskonzept. Es wird eine Knotendatenstruktur definiert. Jeder Knoten enthält die jeweilige Kantenbeschriftung und die Verweise auf seine Nachfolger. Die Folge der Geschwisterkanten im Kantenbaum entspricht einem Knoten im Knotenbaum. Je geringer der Verzweigungsgrad des Baumes, um so kleiner die Größe eines Knotens. Ist der Baum gut balanciert, so ist diese Implementierung besonders geeignet. Für einen Suffix Baum über DNA Sequenzen läßt sich das, wegen der Ungleichverteilung der jeweiligen Nukleotide in der Sequenz, nicht erwarten.

Das Array ist eine linearisierte Implementierung des Baumes. Es werden die Knotenebenen des Baumes so formatiert, daß ihnen Array Abschnitte zugeordnet werden können. Oder es werden innere Knotenschichten völlig weggelassen. Ist der Baum ausgeglichen und gleichmäßig besetzt, so enstehen aus diesem Vorgehen am wenigsten Nachteile. Orientiert man sich bei unausgeglichenen Bäumen an der größten Knotenstruktur, so wird Speicherplatz verschwendet. Kompakter gespeicherte Arrays sind ungünstig beim Aktualisieren. Im schlechtesten Fall muß das Array vollständig neu aufgebaut werden. Suffix- oder PAT-Arrays ([Man93]) sind Beispiele für diese Implementierungsvarianten. Neuer ist die Datenstruktur von [Abo02].

□

Bei der Speicherung auf dem Sekundärspeicher muß auf eine gewisse Sequentialität des Zugriffes geachtet werden. Es kann nicht wie beim Hauptspeicher ein 'bad locality of memory references' hingenommen werden. Um gerade dies zu verhindern und dem sequentiellen Zugriff Rechnung zu tragen, bevorzugen wir in dieser Arbeit, infolge der betrachteten Anwendung, die Darstellung als ein Array.

Definition : Die lokale Tree Topologie ist die Verzweigungsstruktur des Baumes.

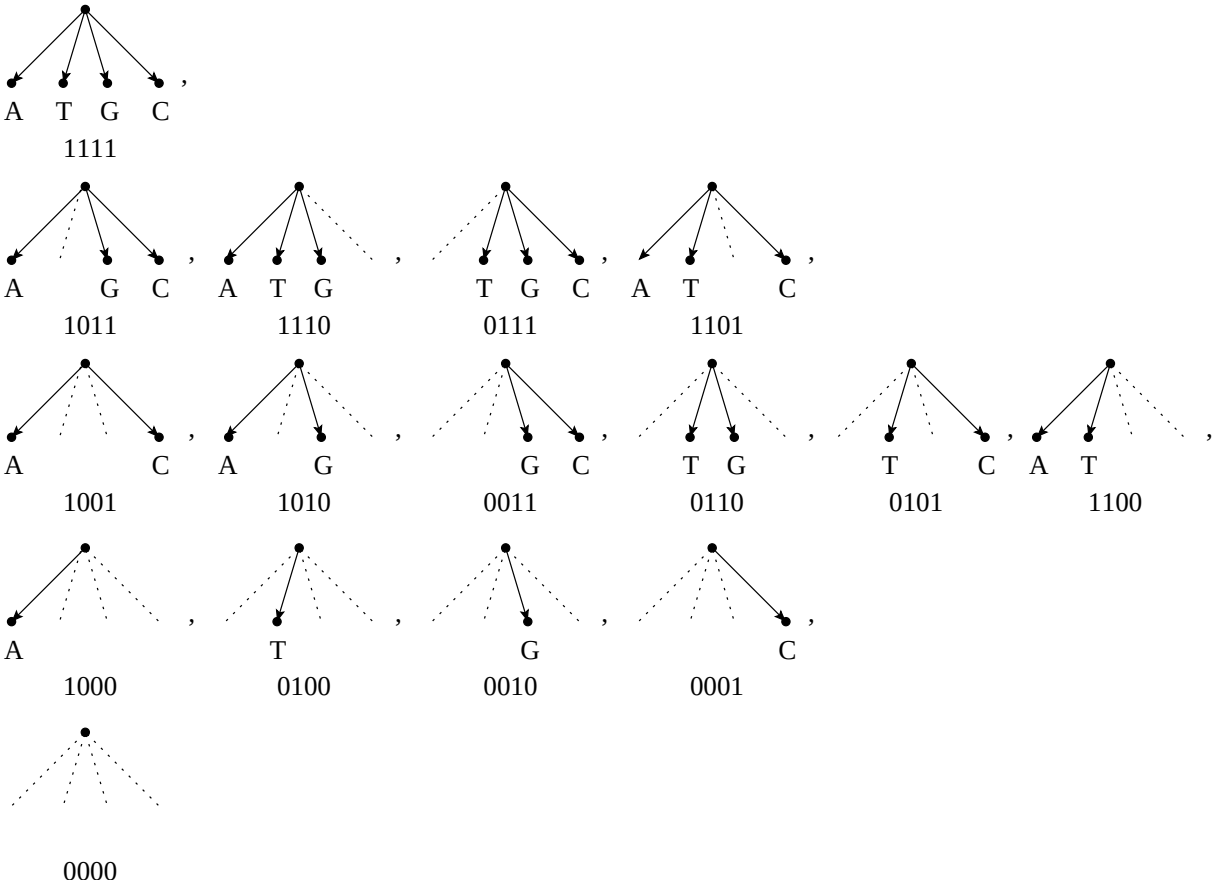
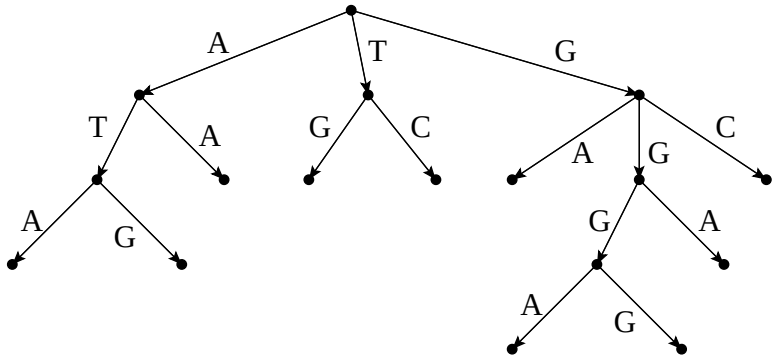


Abbildung 3.4: Darstellung der lokalen Tree Topologie und ihrer jeweiligen Kodierung

Es wird als erstes eine feste Reihenfolge der Buchstaben eines Alphabetes definiert. Dieses kann, zum Beispiel im einfachsten Fall, die lexikographische Ordnung sein. Für das DNA Alphabet, wird hier A, T, G, C festgelegt.

Die Verzweigungsstruktur der Kanten unterhalb eines Knotens, die lokale Tree Topologie, kann nun eindeutig durch eine Bitmap ausgedrückt werden. Die Angabe eines Bitmap Schema ist für jedes beliebige Alphabet möglich. Im Fall von DNA Sequenzen illustriert Abbildung 3.4 die jeweiligen Darstellungen. Da ein Suffix Tree die Vereinigung solcher Baumteilelemente ist, kann man den Baum auf diesem Wege linearisieren. Man wählt einen Startknoten und definiert den Weg des Durchlaufes, der jeden Knotenpunkt genau einmal besucht. Der kanonische Startpunkt in einem Wurzelbaum ist die Wurzel. Es läßt sich aber auch jeder andere Knoten als Startpunkt im Graphen bestimmen. Die Durchlaufstrecke muß nicht entlang der Kanten liegen. Sie muß nicht zusammenhängend sein. Abbildung 3.5 demonstriert eine solche Kodierung eines Baumes. Hierbei wurden die Knoten des Baumes an den Kanten von links nach rechts mittels der Tiefensuche durchlaufen. Ein anderer trivialer Weg wäre die Breitensuche. Würden wir nur Bäume betrachten, deren Verzweigungskanten mit einem Buchstaben beschriftet sind (wie zum Beispiel in Abbildung 3.5), so wäre die angestrebte Linearisierung

erreicht.



1110 1100 1010 0000 0000 0000 0011 0000 0000
1011 0000 1010 0000 1010 0000 0000 0000

Abbildung 3.5: Beispiel einer linearisierten Baumkodierung

Gibt es Kanten mit mehr als einem Buchstaben, betrachte hier xt mit $x \in \Sigma, t \in \Sigma^*$, so werden diese bei der Kodierung nach dem ersten Buchstaben auseinander gebrochen. Es entstehen zwei hintereinanderliegende Kanten, die erste beschriftet mit x, die zweite mit t. Die erste Kante wird mit der Treekodierung dargestellt. Die zweite Kante als Sequenz mit dem Huffmann Code verschlüsselt. Das gleiche Prinzip gilt für Kanten mit Referenzierungspositionen (*suffixpos*).

Bei der oben betrachteten Treekodierung wurde die Behandlung des terminalen Symbols \$ völlig vernachlässigt. Wie im vorherigen Kapitel 3.1 ausgeführt, sind die Pfade an den Kanten nur so lang wie der Suffix eindeutig dadurch festgelegt wird. Bei den hier betrachteten sehr großen DNA Sequenzen, tritt das in der Praxis vor dem Erreichen des Endsymbols ein. Das heißt, bis auf wenige Ausnahmen gibt es keinen Pfadstring, welcher das \$-Symbol enhält. Diese Ausnahmen werden in der Implementierung extra behandelt.

Mit der Bitmap Kodierung erübrigt sich bei der Suche des richtigen Kindknotens das Durchlaufen der Geschwisterknoten, wie es bei der Implementierungsvariante des Kantenbaumes üblich ist. Bei kleiner Alphabetgröße ist dies für Hauptspeicherimplementierungen völlig unerheblich. Aber auf der Disk ist ein solcher Nachteil auch bei geringem Verzweigungsfaktor nicht vernachlässigbar.

Die Kodierung eines Baumes als lineare Bitmap ist um so effizienter je kleiner das Alphabet. Mit steigender Alphabetgröße nimmt die Wahrscheinlichkeit von Leerbelegungen der Bitstellen zu. Die Kosten steigen exponentiell. Das heißt, für den Fall von DNA Sequenzen befähigt die geringe Größe des Alphabetes einem zu beidem – erstens – einer effizienten Darstellung der Tree Topologie – und zweitens – einer effizienten Kantenbeschriftung.

In den Standardsuchoperationen, dem Exact String Matching, ist es notwendig, den so kodierten Baum selektiv zu durchlaufen. Das heißt, die gesuchten Teilbäume des gesamten Baumes, die aufgrund des Durchlaufweges physisch getrennt wurden, zu extrahieren. Da die Kanten des Baumes in einem einzigen Array angeordnet sind, benötigt man die Anzahl der Knoten (und Kanten), die im Array zu überspringen sind. Dieses Problem läßt sich durch den Gebrauch zusätzlicher *Skipp Pointer* lösen. Die Anzahl und die Ausprägung dieser Skipp Pointer hängt vom definierten Durchlaufweg ab.

In unserem Fall, wird für jede existierende Kante ein Pointer gesetzt. Würde man hierfür absolute Pointerwerte nehmen, vorgegeben vom jeweiligen Arrayindex, so wäre dies sehr Speicherplatz intensiv. Die Pointer würden das Vielfache des Platzes, welchen die Daten aquirieren, einnehmen. Daher ist es sinnvoller, nur die Differenz zwischen dem Knoten im Array und dem nachfolgenden Knoten zu bestimmen, das heißt die tatsächliche Anzahl an Kanten, die übergangen werden muß. Diese, sogenannten *Relativen Skipp Pointer* sind für Knoten mit ausgedehnten unterliegenden Teilbäumen sehr groß. Die obersten Knoten können in Textdatenbanken Werte annehmen, die nur mit 16 Byte speicherbar sind. Für Knoten nahe der Wurzel wird hingegen sehr wenig Speicher benötigt. Auf dem feinsten Level treten Werte bis $4 \cdot K_1$ auf (wobei K_1 die Bitrepräsentation des Treekodes und der Textpointer (*suffixpos*) beinhaltet, siehe Tabelle 3.1), auf dem zweitfeinstem Level bis $16 \cdot K_2$ (in K_2 fließt kumulativ K_1 ein).

Daher ist es zur Darstellung relativer Skipp Pointer nützlich, jeweils Speicher von variabler Größe zu allozieren. Desweiteren auch die Anzahl der existierenden Pointer zu kodieren. Da maximal 4 Geschwisterkanten vorhanden sein können, benötigt man zur Kodierung der Anzahl nur 2 Bit. Für die Kodierung der Pointergröße kann man ebenfalls nur 2 Bit nutzen. 00 bedeutet eine Pointergröße von 1 Byte, 01 für 2 Byte, 10 für 4 Byte und 11 für 8 Byte. Je nach Skalierung der Datenbank $\mathcal{D}$ läßt sich das weiter erhöhen, indem für die Pointergröße 3 Bit und mehr gewählt wird. Durch die Benutzung variabler Größen sind die Skipp Pointer nicht mehr direkt die Anzahl der Knoten im Teilbaum, sondern die Gesamtheit an Bytes, welches das Bitmuster des ganzen Teilbaumes einnimmt. Mit dieser Methode ist der durchschnittliche Speicherbedarf für einen Skipp Pointer weniger als 2 Byte, unabhängig von der Tiefe des Baumes.

Die oben beschriebene Methode führt auf einen Datentyp, dargestellt wie in Tabelle 3.1. Der durchschnittliche Speicherplatzbedarf für einen Knoten beträgt ungefähr 7 Byte. Betrachtet man dies bezüglich der Länge des Textes, so erhält man durchschnittlich $11n$ Bytes.

Tabelle 3.1: Datentyp eines Rekords variabler Größe

Knoten Art	Treekode	Pointer Anzahl	Pointer Größe	....	Pointer Größe	Skipp Pointer	....	Skipp Pointer
2 Bit	4 Bit	2 Bit	2 Bit	...	2 Bit	variabel	...	variabel

Tabelle 3.2: Rekords aus dem Beispiel von Abbildung 3.5

Knoten Art	Treekode	Pointer Anzahl	Pointer Größe .. .. Pointer Größe	Skipp Pointer	Skipp Pointer	Skipp Pointer
10	1110	10	00 00 00 -	3	36	58
10	1100	01	00 00 - -	2	23	
10	1010	01	00 00 - -	2	10	
11	0000	00	Textpointer			
11	0000	00	Textpointer			
11	0000	00	Textpointer			
10	0011	01	00 00 - -	2	10	
11	0000	00	Textpointer			
11	0000	00	Textpointer			
10	1011	10	00 00 00 -	3	11	45
11	0000	00	Textpointer			
10	1010	01	00 00 - -	2	10	
11	0000	00	Textpointer			
10	1010	01	00 00 - -	2	10	
11	0000	00	Textpointer			
11	0000	00	Textpointer			
11	0000	00	Textpointer			

Um die unterschiedliche Struktur der Knoten zu kennzeichnen, wurden spezielle Knotenarten eingeführt. 10 bedeutet ein Verzweigungsknoten, kodiert wie oben beschrieben nach Abbildung 3.4. Dagegen werden kompakte Pfade mit nur einem Kindknoten mit 00 spezifiziert. Im Feld Treekode wird die Länge des Strings und in den Restabschnitten der String im Huffmann Schlüssel gespeichert. Für eine verbesserte Suche aller Blätter eines Teilbaumes, werden Blätter mit 11 markiert. Dadurch wird die aufwendige rekursive Blattsuche, wie in einem normalen transienten Suffix Tree üblich, umgangen. Anstelle relativer Skipp Pointer werden $suffixpos(s_i)$ in den zu indexierenden Text (Textpointer) gehalten. Eingestreute Hash-Tabellen unterhalb der Wurzel, als Beispiel wie in Abbildung 3.2 demonstriert, werden durch 01 unterschieden.

Tabelle 3.2 zeigt das Schema des Baumes aus Abbildung 3.5. Die Pfeile unter den ersten Skipp Pointer zeigen auf die Rekords, auf welche sie referenzieren.

Gerade bei mehreren hintereinander liegenden Suchoperationen werden Bereiche des Baumes nahe der Wurzel oft frequentiert. Diese sollten im Hauptspeicher gehalten werden. Da, wie im obigen Kapitel 3.1 ausgeführt, die obersten Schichten des Baumes kompressiert sind, wird in erster Linie die Hash-Tabelle transient aufgebaut.

Vergleicht man die hier eingeführte Repräsentation mit dem 'Enhanced Suffix Array' (ESA) von [Abo02], so ergeben sich folgende Unterschiede. Das ESA kodiert einen Tree wie einen Kantenbaum in einem Array. Um beim Exact String Matching den richtigen Kindknoten zu finden, müssen im schlechtesten Fall alle Geschwisterkanten durchlaufen werden. Daher ist für große Alphabete die ESA Repräsentation nicht performant. Die Referenzierung bewirkt einen mehrfachen unerwünschten Wechsel zwischen Datenstruktur und Text. Weiterhin liefert die in dieser Arbeit vorgestellte Darstellung die Möglichkeit von Updates. Besonders die Variante der inkrementellen Erweiterung wird hier unterstützt. Gerade auf dem Gebiet der Indexierung über sehr langen Texten ist die Fähigkeit der Erweiterung des Index eine wesentliche Eigenschaft.

Im nächsten Kapitel werden Konstruktionsvarianten einer solchen Datenstruktur vorgestellt und ihre Nichtanwendbarkeiten für lange Texte erläutert. Es wird sich in der weiteren Arbeit zeigen, daß die hier vorgestellte und neu eingeführte Repräsentation einer Suffix Tree Struktur die Grundlage bildet, um einen optimalen persistenten Konstruktionsalgorithmus zu erhalten, und eine inkrementelle Erweiterbarkeit des Indexes, wie sie für große Datenmenge unabdingbar ist, sicher zu stellen.

Kapitel 4

Konstruktionsalgorithmen für persistente Indexstrukturen

4.1 Konstruktionsalgorithmen für persistente Suffix Trees

Über Konstruktionsalgorithmen für Suffix Trees wurde viel geforscht. Ein naïver Weg zur Konstruktion ist das inkrementelle Einfügen jedes Suffixes des Textes t beginnend bei $s_1 \in \mathcal{S}(t)$ bis $s_{n+1} \in \mathcal{S}(t)$, jeweils ab der Wurzel des aufzubauenden Baumes. Der Baum ist komplett, falls alle Suffixe eingefügt wurden. Dieser Algorithmus ist von der Ordnung $O(n^2)$ in der Zeit.

Dagegen, Weiner[Wei73], McCreight [MCr76], Ukkonen [Ukk95], und Farach [Far97] haben 'linear time' Algorithmen eingeführt. Klassische Algorithmen davon sind [Wei73], [MCr76], [Ukk95]. Sie haben eine Zeitkomplexität von $O(n \log |\Sigma|)$ und einen Platzbedarf von $O(n)$. Der, im Gegensatz dazu, erst kürzlich entwickelte Algorithmus von Farach ([Far97]) löst sich in der Zeitkomplexität von der Abhängigkeit vom Alphabet.

All diese Algorithmen benötigen 'random access' auf dem verwendeten Speicher. Ist die Zugriffszeit auf diesen Speicher gering, so sind diese Methoden auch in der Praxis anwendbar. Zum Beispiel, wenn der gesamte zu konstruierende Baum in den Hauptspeicher paßt, ist dies gewährleistet. Ist dem aber nicht so, so ziehen sie nicht explizit die verwendete Computer Architektur bzw. das Speichermodell in Betracht.

Überschreitet die Größe des Suffix Trees die Hauptspeichergröße, so werden die einzelnen Teile automatisch – durch das Betriebssystem – erst in den Cache gelagert und dann auf die Festplatte geschrieben. Dieser Vorgang ist Betriebssystem abhängig und wird meist durch die Methode 'last used, first saved' geregelt. Da nicht bekannt ist, in welchem Teilbaum der nächste Suffix String einzufügen ist, müssen immer andere Seiten aus dem Cache oder von der Festplatte geholt werden. Das heißt, praktisch gesehen, diese Konstruktionsalgorithmen sind in ihrem Einsatz durch die Hauptspeichergröße begrenzt.

Suffix Trees haben theoretisch einen linearen Speicherplatzbedarf $O(n)$. Bei klassischen

Implementierungen werden bei einem Text t der Länge n ungefähr $28n$ Bytes benötigt. Dieser Bedarf konnnte durch Kurtz [Kur99] reduziert werden, auf im schlechtesten Fall $20n$ Bytes, aber durchschnittlich $10n$ Bytes. Trotz dieser Verbesserung von ungefähr 10 Bytes pro Zeichen – es ist nicht möglich, einen Suffix Tree für einen sehr langen Text zu konstruieren. Auf einem üblichen Computer mit einem Hauptspeicher von 512 MB ist es in angemessener Zeit nicht realisierbar, einen Suffix Tree von einem Text, welcher größer als $55 \cdot 10^6$ Zeichen ist, zu konstruieren.

Das grundlegende Element aller 'linear time' Konstruktionsalgorithmen ist die Verwendung von Suffix Links. Ein Suffix Link verbindet einen Knoten mit dem Pfad ax ($a \in \Sigma, x \in \Sigma^*$) mit dem Knoten mit dem Pfad x. Die oben erwähnten 'linear time' Konstruktionsalgorithmen benutzen Suffix Links als wesentlichen Baustein um eine lineare Zeitkomplexität zu erreichen. Entgegen dem naïven Algorithmus, muß beim Übergang des Einfügens von $s_i \in \mathcal{S}(t)$ zu $s_{i+1} \in \mathcal{S}(t)$, der Knoten ab dem s_{i+1} einzufügen ist, nicht ab der Wurzel gesucht werden, sondern wird über den Suffix Link von $s_i = xs_{i+1}$ aus sofort gefunden. Dieses erhöhte Querspringen im Baum kann in der Praxis nur sinnvoll sein, wenn ausschließlich 'random memory access' zur Verfügung steht. Wenn nicht, führt dies zu einer hohen Anzahl von 'cache misses', addiert zu denen vom naïven Algorithmus. Weiterhin benötigen Suffix Links noch einmal zusätzlichen Speicherplatz, welcher einen erheblichen Anteil am Platzbedarf einnimmt. Da aber viele Anwendungen über Suffix Trees diese dann nicht benötigen, werden sie nur während des Konstruktionsprozeßes aufgebaut.

Wünschenswert wäre es, einen Algorithmus zu finden, welcher ohne Suffix Links in linearer Zeit einen Suffix Baum konstruieren kann.

Reduktion der Aufgabenstellung durch die Betrachtung der Konstruktion von Suffix Arrays

Eine neuere Richtung der Forschung beschäftigt sich mit einer reduzierten Aufgabenstellung – der Konstruktion von Suffix Arrays. Ein Suffix Array läßt sich als eine rudimentäre Form eines Suffix Tree betrachten, nämlich als Tree ohne Tree Topologie. Seine Konstruktion ist daher eine Teilaufgabe der Konstruktion eines Trees. Da beim Array die Tree Topologie nicht mitzuberechnen ist, benötigt seine Repräsentation mit $O(4n)$ wesentlich weniger Platz. Und genauso wichtig, der zusätzliche Platzbedarf während der Konstruktion ist zwar wie beim Suffix Tree linear, aber mit einer kleineren oberen Konstanten. Je geringer dieser zusätzliche Platzbedarf, um so größer die zu indexierenden Texte im Hauptspeicher. Wählt man einen $O(n^2)$ Algorithmus zur Konstruktion, so ist der zusätzliche Platzbedarf vernachlässigbar gering. Die Zeitkosten sind aber in der praktischen Durchführung indiskutabel. Für einen $O(n)$ Konstruktionsalgorithmus benutzt man $O(10n)$ extra Platz. Das Ziel dieser Forschungsrichtung ist es, unter Benutzung einer $O(n \log n)$ Konstruktionszeit einen geringen linearen oder sogar sublinearen extra Platzbedarf zu erhalten. Siehe hierzu neue Ergebnisse von [Bur03] und [Hon03]. Das bedeutet, bei gleichbleibender Speichergröße wird die zu indexierende Textlänge nach hinten verschoben.

In dieser Arbeit folgen wir bewußt einer anderen Aufgabenstellung. Wir wollen die Textgrößen Barriere dadurch überwinden, indem wir von der Benutzung mehrerer Speicherhierarchien ausgehen.

Ein Konstruktionsalgorithmus, welcher bisher entwickelt wurde, und das Speichermodell in Betracht zieht, das heißt nicht nur mit dem Hauptspeicher arbeitet, ist der Partitionierungsalgorithmus, entwickelt von Hunt *et al* ([Hun02]).

Festlegung des Speichermodells

Das Standardmodell zur Berechnung der Laufzeitkosten, wonach ein Algorithmus evaluiert wird, nimmt an, daß jeder Speicherzugriff eine Zeiteinheit kostet. Dabei wird im allgemeinen nicht unterschieden, auf welche Speicherart dieser Zugriff erfolgt. Es wird weiterhin auch keine Rücksicht auf die Speicherhierachie genommen.

Im folgenden wollen wir das 'Two-Level Memory Model' *oder* eine 'Disk Access Maschine' (DAM) betrachten. Sie wurde eingeführt, unter anderem, von Vitter und Shriver in [Vitt93]. Ein 'Two-Level Memory' besteht aus einer Verarbeitungseinheit, dem Hauptspeicher der Größe M und einen großen externen Speicher der Größe N, welcher in Transfer Einheiten unterteilt ist, den sogenannten Diskseiten. Jede Diskseite kann B Rekords R speichern. Genauer: Wenn N die Anzahl der Rekords in einem File ist und M die Anzahl der Rekords, welche in den Hauptspeicher ladbar sind. So muß mindestens $M < N$ und $1 \leq B \leq M/2$ gelten.

Es ergibt sich in einem 'Two-Level Memory Model' folgende Möglichkeit der Analyse von Algorithmen: Da der Zugriff auf den Hauptspeicher wesentlich weniger Zeit kostet, als der auf die Festplatte, könnte man die Anzahl der jeweils unterschiedlichen Speicherzugriffe einfach zählen. Das würde aber noch nicht ganz der realen Situation entsprechen. Denn nicht jeder Diskzugriff ist gleich teuer. Hat man bereits eine Speicherseite geladen, so ist der Zugriff auf die genau folgende Speicherseite geringer in der Zeiteinheit. Das heißt, sequentielle I/O's (Input/Output's) sind weniger teuer als zufällige. Folgende Variante versucht den sequentiellen I/O vom zufälligen I/O zu unterscheiden. Sei $s \in \mathbb{N}$ eine Konstante. Wir definieren eine Sequenz als sR hintereinanderliegender Speicherorte im Hauptspeicher oder auf der Disk sR/B Seiten. Ein Sequenztransfer ist jeder I/O, welcher alle Seiten einer gesamten Sequenz zwischen Hauptspeicher und Festplatte bewegt. Jeder Seitentransfer, welcher nicht Teil einer Sequenz ist, wird ein zufälliger I/O genannt. Sei $S(.)$ die 'worst case' Komplexität des sequentiellen I/O und $Z(.)$ des zufälligen. Der totale I/O auf die Disk ist dann: $\Omega(.) = Z(.) + S(.)\frac{sR}{B}$. Für die unteren Schranken gelten jeweils: $\omega(.) = z(.) + s(.)\frac{sR}{B}$. [1]

Zur Bewertung eines Algorithmus läßt sich nun ein Tupel zweier Tripel

$$< (o(.), s(.), z(.)), (O(.), S(.), Z(.)) >$$

angeben, welches die Zeitkomplexität, unterteilt in drei unterschiedliche Zählungen, darstellt. Als obere Schranke $(O(.), S(.), Z(.))$, und als untere Schranke $(o(.), s(.), z(.))$.

[1] Unterscheide $s(.)$ hier Funktion und nicht obige Konstante.

Die Angabe von c oder C gibt den Vorfaktor der Komplexität an. Dieser ist im allgemeinen von der Implementierungsvariante abhängig.

Bei der Bewertung obiger in-memory Algorithmen wurde die Zeitkomplexität nur bezüglich des Hauptspeicher, das heißt die interne Abarbeitungszeit, angegeben. Die Algorithmen sind meist so implementiert, daß weder zufälliger noch sequentieller I/O auftritt.

Desweiteren sind auch andere Speichermodelle denkbar. Zum Beispiel: die Erweiterung des 'Two-Level Memory Models' zum allgemeinen 'Hierarchical Memory Model' (HMM), beschrieben in [Vitt94]. Und es lassen sich auch parallele Architekturen betrachten, wie Paralleles DAM (PDAM) oder Paralleles RAM (PRAM).

4.2 Evaluation linearer Algorithmen

Gute Beschreibungen dazu finden sich in [Wei73], [Ukk95] und [Gus97]. Bezüglich des oben definierten Speichermodells gilt:

Satz 1 :
Der Algorithmus von Ukkonen oder McCreight hat eine Zeitkomplexität von

$$< (o(n),\, 0\,,\, z(c(2)n))\,,\, (O(C(|\Sigma|)\cdot n),\, 0\,,\, Z(C(|\Sigma|+1)\cdot n)) > .$$

Beweis:
Siehe das bekannte Resultat aus obiger Literatur für die Hauptspeicherkomplexität. Da die Disk als Erweiterung des Hauptspeichers betrachtet wird und der Algorithmus in dieser klassischen Grundform keinen sequentiellen Speicherzugriff unterstützt, erhält man bei einer Auslagerung auf die Disk $Z(C(|\Sigma|)n)$ zufällige Speicherzugriffe. Da der Input Text auch auf der Festplatte liegt, addieren sich dazu noch einmal $Z(n)$ Speicherzugriffe. □

Weiterhin bekannt ist der 'naïve Algorithmus'. Er hat eine Zeitkomplexität von $O(n^2)$. Der naïve Algorithmus behandelt das Konstruktionsproblem als ein Problem des String Sortierens von Suffixen, welches schon voll evaluiert wurde. Die 'worst case' Zeitkomplexität ist $O(n^2)$. In der Praxis wird bei durchschnittlichen Textlängen und Textzusammensetzungen meist eine Zeitkomplexität von $O(n \log n)$ erreicht.

Ist der zu indexierende Text so klein, daß der Tree vollständig im Hauptspeicher Platz findet, so ist der zufällige I/O in der Praxis 0 und die lineare Hauptspeicherkomplexität ein akzeptables Resultat. Ist dies nicht der Fall, so sieht man leicht aus diesem Satz, daß sich in der Praxis erhebliche Berechnungszeiten ergeben, weil der teuerste I/O, der zufällige I/O, bei diesem Resultat sehr groß ist.

Meine weiteren Untersuchungen haben das Ziel – erstens – den zufälligen I/O zu reduzieren, – zweitens – den totalen Disk I/O zu reduzieren. Dabei wird versucht den zufälligen Disk I/O unter anderem in einen sequentiellen I/O umzuwandeln. Desweiteren wird eine Erhöhung des Hauptspeicher I/O in Kauf genommen, da diese Operationen im Gegensatz zum Disk I/O, um ein wesentliches geringer sind.

4.3 Wege der Partitionierung

Das grundlegende Problem ist, daß die Daten zu groß sind und der Hauptspeicher zu klein, um einen insgesamten Suffix Tree über dem Eingabetext im Speicher zu konstruieren. Es müssen zur Konstruktion externe Speicher verwendet werden. Daher sind Algorithmen deren Geschwindigkeit auf dem 'random access' beruhen für solche Datenvolumen nicht geeignet.

Problem : Gesucht ist ein *Externer Suffix Tree Konstruktionsalgorithmus*.

Charakteristisch für diese Situation ist:

- während der Konstruktion werden Daten extern gespeichert
- die Zugriffskosten auf die externen Daten sind signifikant größer, als die Kosten verwendeter Hauptspeicheralgorithmen
- es gibt Beschränkungen beim Zugriff auf externe Daten

Daher sind die **Kriterien für die Entwicklung eines externen Algorithmus:**

- Minimierung der Zugriffe auf externe Daten
- Unterstützung eines sequentiellen Zugriffes auf die externen Daten

Sei t ein gegebener Text. Die Menge aller Suffixe über t ist

$$\mathcal{S}(t) = \{s_i : s_i \text{ Suffix von } t, 1 \leq i \leq n+1\}.$$

Gesucht ist eine disjunkte Aufteilung $\{S_i\}_{i=1}^{k}$ der Menge $\mathcal{S}(t)$ mit $S_i \subseteq \mathcal{S}(t), \mathcal{S}(t) = \dot{\bigcup} S_i$. Die Elemente jeder Menge S_i sollen im Hauptspeicher verarbeitbar sein.

Teilung nach Präfixen einer festen Länge

Die Menge aller Suffixe $\{s_i(t^+) : 1 \leq i \leq n+1\}$ wird nach ihren Präfixen der Länge d sortiert. Betrachte alle $\omega \in \Sigma^d$. Durch die Teilmengen $S_\omega := \{s_i \in \mathcal{S}(t) : \omega \text{ Präfix von } s_i\}$ ist eine Aufteilung von $\mathcal{S}(t)$ gegeben.

Da aus der Definition eines Suffix Trees folgt, daß Suffixe mit gleichem Präfix im gleichen Teilbaum eines Gesamtbaumes liegen, baut man bei einer anschließenden Konstruktion, Partitionen von diesem Gesamtbaum auf. Diese Strategie nennt sich *Index Partitionierung*.

Die Suffixe jeder Teilmenge S_ω haben im Text t keine Indexpositionen, welche eine direkte Folge mit jeweils festen Zwischenabständen bildet. Um von solch einer Menge S_ω einen Suffix Tree aufzubauen, ist es nicht möglich Suffix Links so zu verwenden, daß man in der Zeit einen linearen Algorithmus erhält. Es läßt sich nur ein naïver Algorithmus verwenden. Auf diese Weise wird bei der Strategie der Index Partitionierung nur eine Zeitkomplexität von $O(n^2)$ erreicht.

Diese Methode wurde in Kapitel 4.5 genauer beschrieben.

Teilung nach einer vorgegebenen Schrittlänge

Sei $k \in \mathbb{N}$ als Schrittlänge gegeben. Dann ist $S_i := \{s_{lk+i} \in \mathcal{S}(t) : 0 \leq l < \frac{n+1}{k}, lk+i \leq n+1\}$ eine Aufteilung der Menge $\mathcal{S}(t)$ mit $\mathcal{S}(t) = \bigcup_{i=1}^{k} S_i$.

Zwischen den einzelnen Suffixen einer Teilmenge S_i besteht ein fester Abstand. Daher gibt es die Möglichkeit bei einem Konstruktionsalgorithmus über S_i Suffix Links zu benutzen. Man kann auf diese Weise, wie in Kapitel 4.4 beschrieben, einen linearen Algorithmus erhalten.

Teilung nach einer vorgegebenen Textteillänge

Ein völlig anderer Weg ist es, die Menge aller Suffixe $\mathcal{S}(t) = \{s_i(t^+) : 1 \leq i \leq n+1\}$ eines Textes t so zu unterteilen, daß in jeder Teilung S_j nur Suffixe $s_i(t^+)$ liegen, deren Indizes i eine direkt aufeinanderfolgende Reihenfolge besitzen. Das heißt, sie unterscheiden sich nur durch den Abstand 1. Das ist zu einem gewissen Grade äquivalent zu der Tatsache, daß die gesamten Input Daten in kürzere Teiltexte gebrochen werden. Dieses Vorgehen nennt man *Daten Partitionierung*.

Ist m die maximale Länge eines Textes, dessen Suffix Tree Repräsentation im Hauptspeicher darstellbar ist, so gibt es $\lceil (n+1)/m \rceil$ Aufteilungsmengen. Es ist

$$S_j(t) = S_j := \{s_i \in \mathcal{S}(t) : (j-1)\,m + 1 < i \leq j\,m\}, \qquad \text{wobei } 1 \leq j \leq \lceil (n+1)/m \rceil\,. \quad (4.1)$$

Diese Mengen sind gleich groß. Die Menge aller Suffixe ist auf diese Partitionen gleichverteilt. Dadurch kann eine unvollständige Auslastung des Hauptspeichers bei der Konstruktion von Suffix Teilbäumen ausgeschlossen werden. Desweiteren läßt sich zur Konstruktion, des zu einer Partition gehörenden Suffix Trees, ein linearer Konstruktionsalgorithmus verwenden.

Nun hat man bei diesem Verfahren noch nicht den gesamten Baum über den ganzen Eingabetext konstruiert. Der gesamte Baum ergibt sich, indem man die $\lceil (n+1)/m \rceil$ Teilbäume miteinander mischt - *Merging (Mischen)*. Es wird im weiteren, Kapitel 5, gezeigt, daß dieses Mischen in mehreren Stufen in quasi-linearer Zeit möglich ist, und wie es im Hauptspeicher effizient zu realisieren ist.

Beliebige Aufteilung

Es lassen sich die drei oben beschriebenen Wege, eine Aufteilung zu definieren, miteinander kombinieren, bis hin zur Wahl einer völlig willkürlichen Aufteilung S_i. Bei einer willkürlichen Aufteilung läßt sich nur der naïve Algorithmus zur Konstruktion, mit einer Komplexität von $O(n^2)$, über S_i einsetzen. Da keine Präfixclusterung gegeben war, müssen die entstanden Teilbäume gegeneinander sortiert werden.

4.4 Selektive Partitionierung – Ein tatsächlich linearer Konstruktionsalgorithmus und dessen Nichtverwendbarkeit

Die untere Schranke der Komplexität zur Berechnung eines Suffix Trees liegt bei $O(n)$. Hat man nur ein Speichermedium zur Verfügung, so wird diese erreicht von Weiner und Ukkonen ([Wei73], [Ukk95]).

Wenn man ein 'Two-Level Memory Model' benutzt, läßt sich ebenfalls ein optimaler Weg finden. Im weiteren wird ein linearer Konstruktionsalgorithmus für persistente Bäume vorgestellt.

Sei m die durchschnittliche Länge eines Textes, für welche ein Suffix Tree im Hauptspeicher der Größe M konstruierbar ist. Dann wähle $k := \lceil (n+1)/m \rceil$ als Schrittlänge. Damit erhält man die Aufteilung $S_1, \ldots, S_k$ mit

$$S_i := \{s_{lk+i} \in \Sigma^* : 0 \leq l < \frac{n+1}{k}, lk+i \leq n+1, s_{lk+1} \text{ Suffix von } t^+\}.$$

Satz 2 : Für jede Teilmenge $S_i \subseteq \mathcal{S}(t)$ läßt sich der dazugehörige Suffix Baum in *linearer* Zeit vollständig im Speicher aufbauen.

Beweis:
Betrachte Strings der Länge k, $x \in \Sigma^k$. Sei

$$\Sigma^k(t) := \{x \in \Sigma^k : x \text{ ist Präfix von } s_{lk+1}, s_{lk+1} \text{ Suffix von } t^+, 0 \leq l < \frac{n+1}{k}\}.$$

Die Menge $\Sigma^k(t)$ kann man nun lexikographisch ordnen und abzählen. Das heißt, es existiert eine eindeutige Abbildung:

$$F : x \in \Sigma^k(t) \longmapsto i \in \mathbb{N}.$$

$F(\Sigma^k(t)) \subset \mathbb{N}$ ist eine endliche Menge. Die Menge $F(\Sigma^k(t))$ läßt sich als ein neues Alphabet betrachten.

Man kann eine Abbildung zwischen einem beliebigen Text $x \in \Sigma^*$ auf einen Text aus $F(\Sigma^k(x))^*$ definieren. Betrachte hierfür als Texte die k ersten Suffixe des Textes t. Für jedes $1 \leq i \leq k : s_i$ Suffix von t gilt: s_i läßt sich darstellen durch $s_i = x_1^i x_2^i x_3^i \dots x_m^i x_{m+1}^i$, wobei $x_j^i \in \Sigma^k$, $\forall 1 \leq j \leq m$ und $|x_{m+1}^i| < k$. Dann

$$\widetilde{F} : s_i \in \Sigma^* \longmapsto y_i \in F(\Sigma^k(s_i))^* \quad \text{mit} \quad y_i = F(x_1^i)F(x_2^i)\dots F(x_m^i)\$^i.$$

Mit [Wei73] oder [Ukk95] kann man nun in linearer Zeit den Suffix Tree $\mathcal{T}_{S(y_i)}$ über den Text y_i aufbauen. Die Komplexität liegt bei $O(m \log(|F(\Sigma^k(s_i)|)$ für jedes i.

Aus dem Baum $\mathcal{T}_{S(y_i)}$ läßt sich in linearer Zeit der kompakte Baum $\mathcal{T}_{S_i}$ ableiten. Die jeweiligen Kanten, welche mit Strings $x \in F(\Sigma^k(s_i))$ beschriftet sind, lassen sich eindeutig zurücktransformieren mit F^{-1}. Die Länge der Kantenbeschriftung ist nun ein Vielfaches von k (ausgenommen bei der Transformation von $\$^i$). $F^{-1}(\mathcal{T}_{S(y_i)})$ bezeichnet den erhaltenen Baum.

Bei dieser Rücktransformation haben unterschiedliche Geschwisterpfade nicht mehr einen unterschiedlichen Buchstaben als Präfix der jeweiligen Kantenbeschriftung. Daher ist der Tree $F^{-1}(\mathcal{T}_{S(y_i)})$ einmal zu durchlaufen und die jeweiligen Pfade zu sortieren. Da die Anzahl der Knoten in $\mathcal{T}_{S(y_i)}$ linear, so auch die Anzahl der Knoten in $F^{-1}(\mathcal{T}_{S(y_i)})$. Bei einem Durchlauf ist jeder Knoten des Baumes $F^{-1}(\mathcal{T}_{S(y_i)})$ zu besuchen und die k-ersten Buchstaben aller ausgehenden Pfade, deren Anzahl maximal $|\Sigma^k|$ werden kann, zu vergleichen. Dies ist linear, das heißt in $O(C(k, |\Sigma^k|)\, m)$ Schritten möglich.

□

BEMERKUNG:

Wenn $\Sigma^k(t) = \Sigma^k$, dann ist $F(\Sigma^k(t)) = \{1, \dots, |\Sigma|^k\}$.

Betrachte einen gegebenen Text t mit fester Länge n. Sei k fest. Ist $|\Sigma|$ groß, so ist im allgemeinen $\Sigma^k(t) \subset \Sigma^k$. Je größer das Alphabet, um so größer ist die Differenz $F(\Sigma^k) - F(\Sigma^k(t))$. Bei sehr großem $|\Sigma|$ kann es auftreten, daß gilt: $|\Sigma^k(t)| \ll |\Sigma|$. Sei $|\Sigma| = |t| = n$, dann $|\Sigma^k(t)| = n/k$. Je kleiner jedoch das Alphabet, um so wahrscheinlicher ist es, daß bei einem gegebenen sehr langen Text fast alle Permutationen der Ordnung k enthalten sind. In diesem Fall ist meist $|\Sigma| < |\Sigma^k(t)|$.

Andererseits, sei wieder t gegeben und Σ sei ein festes Alphabet. Je kleiner k, um so wahrscheinlicher ist die Existenz eines $x \in \Sigma^k$ im Text t. Das bedeutet, für kleine k ist meist auch die Differenz $F(\Sigma^k) - F(\Sigma^k(t))$ kleiner.

Gerade bei der DNA ist die Alphabetgröße im Gegensatz zu anderen Texten klein – betrachtet man hier zum Beispiel: englische, deutsche oder chinesische Texte. Die Textlänge kann als beliebig groß gesehen werden. Bei festem M gilt: $k \to \infty$ für $n \to \infty$. Und das bedeutet bei geringer Alphabetgröße, daß $|F(\Sigma^k(t))|$ nahe an $|\Sigma^k|$ liegt. Das bedeutet, für $n \to \infty$ gilt auch $|F(\Sigma^k(t))| \to \infty$.

Theoretisch läßt sich für jedes große n ein k so wählen, daß $|\Sigma^k(t)|$ sehr klein wird. Wir sind in dieser Arbeit aber an der praktischen Umsetzbarkeit einer Methode in einen Algorithmus interessiert. Dies wäre bei einem solchen Ansatz nicht gegeben.

Obiger Satz 2 besagt, daß der kompakte Tree $\mathcal{T}_{S_i}$ in linearer Zeit konstruierbar ist. Wie im Beweis gezeigt, hat man bei der Konstruktion einen konstanten Vorfaktor, welcher von $|\Sigma^k|$ abhängt. Dieser Vorfaktor wird in der Praxis sehr groß und beeinflußt die Rechenzeit wesentlich. Obwohl die Berechnung des Baumes über S_i vollständig im Hauptspeicher abläuft, braucht man trotzdem mehrere Stunden.

□

***Postprocessing* der Bäume $\mathcal{T}_{S_i}$**

Bisher wurden die kompakten Bäume zu jeder Aufteilungsmenge S_i konstruiert. Um daraus den insgesamten Baum $\mathcal{T}_{S(t)}$ abzuleiten, müssen die $\mathcal{T}_{S_i}$ vereinigt werden. Aufgrund der selektiven Wahl der S_i wird sich zeigen, daß auch dies in linearer Zeit möglich ist.

Satz 3 : Je zwei aufeinanderfolgende Teilbäume $\mathcal{T}_{S_i}$ und $\mathcal{T}_{S_{i+1}}$ lassen sich in linearer Zeit 'einfach' vereinigen.

ZUSATZ: 'Einfach' vereinigen bedeutet, daß für den resultierenden Tree $\mathcal{T} := \mathcal{T}_{S_i} \cup_e \mathcal{T}_{S_{i+1}} \neq \mathcal{T}_{S_i} \cup \mathcal{T}_{S_{i+1}}$ gilt.

Beweis:

Vorgehen: Partielle Vereinigung der Bäume durch den Vergleich von Präfixen einer festen Länge jeder Kantenbeschriftung

Jedes $\mathcal{T}_{S_j}$ hat durchschnittlich die Größe m. Da $\mathcal{T}_{S_j}$ ein kompakter Baum ist, so ist die Anzahl der Knoten von der Ordnung $O(m)$ (siehe [Gus97]). Sei $\ell \in \mathbb{N}$ fest. Durchlaufe beide Bäume simultan in DFS-weise ab der Wurzel und vergleiche an jedem Knoten jede ausgehende Kante des Baumes $\mathcal{T}_{S_i}$ mit jeder ausgehenden Kante des Baumes $\mathcal{T}_{S_{i+1}}$ auf die Präfixe der Kantenbeschriftung der Länge ℓ. Dazu benötigt man $O(C(\ell)m)$ Vergleichsschritte.

Dabei werden niemals gleiche Kanten als verschieden identifiziert, aber manchmal verschiedene Kanten als gleich. Man erhält einen kompakten Baum $\mathcal{T}$, welcher nicht identisch mit dem gesuchten Baum $\mathcal{T}_{S_i} \cup \mathcal{T}_{S_{i+1}}$ ist.

□

BEMERKUNG: Je größer $\ell \in \mathbb{N}$, um so näher liegt $\mathcal{T}$ an $\mathcal{T}_{S_i} \cup \mathcal{T}_{S_{i+1}}$.

Sei

$$\mathcal{T}_V := \mathcal{T}_{S_1} \cup_e \mathcal{T}_{S_2} \cup_e \ldots \cup_e \mathcal{T}_{S_k}.$$

Satz 4 : Der Baum $\mathcal{T}_V$ läßt sich in linearer Zeit in den gesuchten Baum $\bigcup_{i=1}^{k} \mathcal{T}_{S_i}$ transformieren.

Beweis:

Das Ziel ist die Trennung falsch vereinigter Kanten.

Da der Baum $\mathcal{T}_V$ kompakt, so ist die Anzahl der Knoten von der Ordnung $O(n)$. Ein Scan des gesamten Baumes ist daher in linearer Zeit möglich. Durch diesen Scan sollen diejenigen Teile des Baumes identifiziert werden, deren Vereinigung aufgelöst werden muß. Dabei sind besonders die Regionen in $\mathcal{T}_V$ wichtig, die aus Knoten bestehen, deren Blätter aus S_i und S_j mit $i \neq j$ stammen.

Sei

$$\mathcal{N} := \{u \in \mathcal{V}_{\mathcal{T}_V} : \exists\, l_1k+i \in S_i, l_2k+j \in S_j,\, i \neq j \text{ mit } u = lca(l_1k+i, l_2k+j)\}.$$

Schritt 1: Merkmal zur Identifizierung falsch vereinigter Knoten in $\mathcal{N}$

In linearer Zeit läßt sich zu jedem Knoten $u \in \mathcal{V}_{\mathcal{T}_V}$ die Länge des Pfades $|path_{\mathcal{T}_V}(u)|$ in $\mathcal{T}_V$ bestimmen. Somit auch für die relevanten Knoten $u \in \mathcal{N}$. Sei $u \in \mathcal{N}$ der Knoten, welcher der letzte gemeinsame Vorfahre ('least common ancestor' $= lca$) in $\mathcal{T}_V$ zu den Blättern l_1k+i und l_2k+j ist. Das heißt, $u = lca(l_1k+i, l_2k+j)$. Dann wurde der Knoten u richtig vereinigt, falls

$$|path_{\mathcal{T}_V}(u)| \;=\; |lcp(l_1k+i, l_2k+j)|$$

gilt, wobei $lcp(x,y)$ der 'längste gemeinsame Präfix' von x und y bedeutet. Bei falscher Vereinigung gilt:

$$|path_{\mathcal{T}_V}(u)| \;>\; |lcp(l_1k+i, l_2k+j)|.$$

BEMERKUNG: Der lca-Knoten zweier Blätter bezieht sich immer auf einen Baum, daß heißt er kann in verschiedenen Bäumen auch unterschiedliche Werte annehmen. Dagegen ist der lcp Wert zweier Blätter ein String und wird eindeutig durch den Text t bestimmt.

□

Schritt 2: Vielfachheit von $u \in \mathcal{N}$

Unterschiedliche Blätter können zwar die gleiche lca-Position in $\mathcal{T}_V$ haben, aber verschiedene lcp-Längen. Das heißt, es können $u \in \mathcal{N}$ existieren, mit $\exists\, l_1k+i_1, l_2k+i_2, l_3k+i_3$, wobei i_1, i_2, i_3 voneinander verschieden, und es gilt

$$u = lca(l_1k+i_1, l_2k+i_2) \quad \text{und} \quad u = lca(l_1k+i_1, l_3k+i_3).$$

Aber es gilt andererseits

$$|lcp(l_1k+i_1, l_2k+i_2)| \neq |lcp(l_1k+i_1, l_3k+i_3)|.$$

Betrachte für $u \in \mathcal{N}$ Paare aus $S_i \times S_j$, $(i \neq j)$, wobei $u = lca(l_1k+i, l_2k+j)$. Numeriere die Vielfachheit dieser Paare für u, und betrachte solche u als verschieden. Dies bezeichne die Menge $\mathcal{N}_u$. Für $u_1 \in \mathcal{N}_u$ und $u_2 \in \mathcal{N}_u$ mit $u_1 \neq u_2$ gilt: $pos_{\mathcal{T}_V}(u_1) = pos_{\mathcal{T}_V}(u_2)$ [2].

[2] Die Funktion *pos* legt eine Ortsangabe in einem Suffix Baum fest. Eine exakte Definition findet sich in Kapitel 5.1.3.

Sei

$$\widetilde{\mathcal{N}} := \bigcup_{u \in \mathcal{N}} \mathcal{N}_u.$$

Da $\mathcal{N} \subseteq \mathcal{V}_{\mathcal{T}_V}$, so ist $|\mathcal{N}|$ von der Ordnung $O(n)$. Jeder Knoten $u \in \mathcal{N}$ kann maximal die Vielfachheit $\binom{k}{2}$ haben. Dann ist die Größe der Menge $\widetilde{\mathcal{N}}$ von der Ordnung $O(C(\binom{k}{2})n)$. Für jedes Element aus $(l_1k+i, l_2k+j) \in S_i \times S_j$, $(i \neq j)$ ist $u \in \widetilde{\mathcal{N}}$ eindeutig bestimmt. Daher sei $u := lca_{\widetilde{\mathcal{N}}}(l_1k+i, l_2k+j)$.

Wie oben erwähnt, ist die Berechnung $|path_{\mathcal{T}_V}(u)|$ linear. Es folgt, wie sich auch $|lcp(l_1k+i, l_2k+j)|$ in linearer Zeit berechnen läßt.

□

Schritt 3: Berechnung von $|lcp(l_1k+i, l_2k+j)|$

Für die lcp-Werte zweier Blätter gilt:

$$lcp(l_1k+i, l_2k+j) = \begin{cases} 1 + lcp(l_1k+i+1, l_2k+j+1) & \text{,falls } t(l_1k+i+1) = t(l_2k+j+1); \\ 0 & \text{,sonst.} \end{cases} \tag{4.2}$$

Stufenweiser Algorithmus:

Es läßt sich eine Abbildung $f : \widetilde{\mathcal{N}} \to \widetilde{\mathcal{N}}$ definieren durch:

$$f(u) = v, \text{ falls } u = lca_{\widetilde{\mathcal{N}}}(l_1k+i, l_2k+j) \text{ und } v = lca_{\widetilde{\mathcal{N}}}(l_1k+i+1, l_2k+j+1)$$

gilt.

Diese Abbildung f definiert einen Baum $\mathcal{T}_f$ über den Knoten aus $\widetilde{\mathcal{N}}$. Der Baum ist vollständig, wenn jedes Blatt $l_1k+i \in S_i$ und $l_2k+j \in S_j$ mindestens einmal durch die lcp-Funktion in Gleichung (4.2) aufgerufen wurden. Beziehungsweise, wenn jedes Element aus $\widetilde{\mathcal{N}}$ einmal Argument von f war.

Hilfssatz: *Die Berechnung dieses Baumes $\mathcal{T}_f$ erfolgt in linearer Zeit.*

Beweis: Erstens – $|\widetilde{\mathcal{N}}|$ ist von der Ordnung $O(n)$. Zweitens – die Abfrage des 'least common ancestors' aus $\mathcal{T}_V$, des $lca_{\widetilde{\mathcal{N}}}$ ist in konstanter Zeit möglich. Dafür wird ein lineares Preprocessing benötigt.

Denn: Es reicht zu zeigen, daß die Berechnung des lca-Knotens in $\mathcal{T}_V$ linear ist und die jeweiligen Abfragen in konstanter Zeit möglich sind. Dies wurde gezeigt bei [Har84] und verbessert von [SVi88]. □

□

Schritt 4: Die gesuchten Werte $|lcp(l_1k+i, l_2k+j)|$ sind gleich der Tiefe, des jeweiligen Knotens $u = lca_{\widetilde{\mathcal{N}}}(l_1k+i, l_2k+j) \in \widetilde{\mathcal{N}}$ im neuen Baum $\mathcal{T}_f$. Die Anzahl der Knoten von $\mathcal{T}_f$ ist von der Ordnung $O(n)$. Daher ist auch die Bestimmung der Tiefe linear.

□

Schritt 5: Spalten der falsch vereinigten Knoten und Kanten

Dieser Schritt wurde im Artikel von [Far97] gezeigt und kann an dieser Stelle genauso angewendet werden.

□

Aus obiger Bemerkung folgt, daß dieses Verfahren in der Praxis jedoch nicht anwendbar ist.

4.5 Präfix Clusterung

Partitionierungs-Algorithmus von *Hunt*

Der folgende Partitionierungs-Algorithmus wurde eingeführt von Hunt *et al*. Es ist im wesentlichen eine Anpassung des naïven Konstruktionsalgorithmus an die Speicherhierachien. Hierbei ist die Verwendung von Suffix Links nicht möglich. Damit erhält man eine 'worst case' Zeitkomplexität von $O(n^2)$. Es wird in Kauf genommen, daß mehrere *Preprocessing* Scans über den gesamten Text t mit $|t| = n$ durchgeführt werden müssen.

Hierbei muss bedacht werden, daß eine große endliche Anzahl von Scans über einen sehr langen Text mehr Rechenzeit in Anspruch nehmen kann als ein $O(n^2)$ Algorithmus im Hauptspeicher. Das allein verursacht in der Zeitkomplexität einen nicht vernachlässigbaren konstanten Vorfaktor.

Der Partitionierungsalgorithmus basiert auf der Tatsache, daß Suffixe mit gleichen Präfixen $\omega \in \Sigma^*$ im gleichen Teilbaum $\mathcal{T}_\omega(t)$ [3] des Suffix Trees $\mathcal{T}_{S(t)}$ geclustert liegen. Abhängig von der Länge des Textes wird eine feste Präfixlänge d vorgegeben. Sei $\{\omega \in \Sigma^d : |\omega| = d\}$ die Menge aller Präfixe der Länge d.

Die Suffixe werden abgebildet auf die Menge S_ω, definiert in Kapitel 4.3. Aus jeder Menge S_ω wird eine Treepartition $\mathcal{T}_{S_\omega} = \mathcal{T}_\omega(t) \subseteq \mathcal{T}_{S(t)}$ aufgebaut. Hierbei mußte d so gewählt werden, daß jeder ω-Teilbaum $\mathcal{T}_\omega(t)$ nicht die Hauptspeichergröße überschreitet. Bei nicht sachgemäßer Justierung arbeitet der Konstruktionsalgorithmus automatisch mit dem Auslagerungsmechanismus des Betriebssystems, auch beschrieben in Kapitel 4.1. Der Vorteil der Partitionierung geht damit verloren.

Der Algorithmus arbeitet nach folgendem Pseudocode:

Algorithmus – BauePartitionBaum(t, d)

1: **for all** Partitionen $\mathcal{T}_\omega(t) \subset \mathcal{T}_{S(t)}$ **do**
2: **for all** $i = 1 \cdots n + 1$ **do**

[3] Mit $\mathcal{T}_\omega(t)$ wird der Teilbaum von $\mathcal{T}_{S(t)}$ bezeichnet, welcher unterhalb des Knotens $\overrightarrow{w}$ in $\mathcal{T}_{S(t)}$ beginnt.

3: **if** $s_i \in S_\omega(t)$ **then**
4: SuffixEinfügen($s_i(t^+)$)
5: **end if**
6: **end for**
7: checkpoint
8: **end for**

Partitionierungsalgorithmus von Hunt *et al, beschrieben in [Hun01]*

Für einen festen, beliebigen Präfix $\omega \in \Sigma^d$, werden alle Teilstrings aus t, welche mit ω beginnen, registriert (Zeile 1+2). Wenn an Position i der Suffix den Präfix ω besitzt, so wird der Suffix $s_i(t^+)$ in den inkrementell wachsenden Baum $\mathcal{T}_\omega(t)$ eingefügt (Zeile 3-5). Nachdem der gesamte Text t durchlaufen wurde, ist der Teilbaum $\mathcal{T}_\omega(t)$ vollständig konstruiert worden. Die Speicherverwaltung der benutzten Plattform, übernimmt die persistente Speicherung, in diesem Fall durch einen Datenkank *checkpoint*. In ihrer Implementierung nutzten [Hun02] die persistente, auf Java basierende Plattform, PJama.

Analyse:
Da die Teilbäume $\mathcal{T}_\omega(t)$ und $\mathcal{T}_{\omega'}(t)$ $(\omega, \omega' \in \Sigma^d, \omega \neq \omega')$ disjunkt sind, können sie unabhängig voneinander aufgebaut und gespeichert werden. Im Gegensatz zu den 'linear time' Konstruktionsalgorithmen ist die Größe des Suffix Tree nicht beschränkt durch den Hauptspeicher. Dieser Ansatz kann auch auf sehr lange Texte angewendet werden, da die Partitionen so klein wie gewünscht reduziert werden können. [Hun02] *et al* zeigten, das gerade für eine moderate Textgröße ihr Algorithmus eine angemessene Laufzeit besitzt. Diese wird erreicht durch ein gutes Lokalitätsverhalten.

Die Laufzeit dieses Algorithmus birgt jedoch einen Nachteil. Das wiederholte Durchlaufen des Baumes von der Wurzel bis zur Einfügestelle bei jedem Einfügeprozeß ist dafür verantwortlich, daß die 'worst case' Laufzeit von der Ordnung $O(n^2)$ ist. Ein anderer Nachteil ist, daß durch obigen Pseudocode, beim Aufbau von jeder Partition $\mathcal{T}_\omega$ der Text t in voller Länge durchlaufen wird.

Da die Größe der ω-Teilbäume nicht die Größe des Hauptspeichers überschreiten darf, läßt sich eine untere Grenze für die Anzahl der Partitionierungen angeben. Wenn M die Hauptspeichergröße und $\sigma_n = |\mathcal{T}_{S(t)}|$ die Größe einer minimalen Suffix Tree Hauptspeicherinstanz eines Textes mit Länge n, so ist σ_n / M die minimale Anzahl von Partitionierungsschritten. Da $\sigma_n \in O(n)$ und der Speicher eine feste Größe M hat, wächst die Anzahl der Partitionen, welche den gesamten Baum bilden, $|\mathcal{T}_{S(t)}|$ linear mit der Textlänge n. Folglich ist die insgesamte Zeit, um alle Partitionen zu durchlaufen $O(n^2)$. Da die n Einfügungen in $O(n \log n)$ Zeit durchgeführt werden können, so hat der insgesamte Algorithmus eine Zeitkomplexität von $O(n^2)$ unabhängig vom Eingabetext t.

Satz 5:

Die Komplexität des obigen Algorithmus ist

$$< (o(n), s(n), z(|\Sigma|^d + n)), (O(n^2), S(C(2\frac{1}{B})n), Z(C(1/|\omega|)n)) > .$$

BEMERKUNG:
Für eine realistische Komplexitätsanalyse mit der Einschätzung von Laufzeiten ist meist die Wahl des Wurzelbaumes entscheidend. Ist ein Vorfaktor, wie $|\Sigma|^d$ vorhanden, so führt dies zu erhöhten Laufzeiten.

Ein weiterer Nachteil dieser Methode ist, daß durch die einmal festgelegte Länge d Suffixteilmengen S_ω gebildet werden, welche völlig unterschiedliche Größen haben. Die Länge d wird nur an der größten dieser Teilmengen festgelegt. Es gibt daher viele Mengen, die so wenige Suffixe beinhalten, daß die Größe der aufzubauenden Partition $|\mathcal{T}_\omega| \ll M$ ist, also wesentlich kleiner als der Hauptspeicher. Die Methode nutzt auf weiten Strecken die vorhandene Kapazität nicht aus.

Der Vorteil dieser Methode ist, daß es kein *Postprocessing* gibt.

Der geclusterte Konstruktionsalgorithmus

In diesem Abschnitt wird ein Algorithmus von [Sch03] vorgestellt, welcher obigen Indexpartitionierungsalgorithmus verbessert. Die 'worst case' Komplexität verbessert sich jedoch nicht, bleibt bei $O(n^2)$. Der gesamte Baum wird, wie beim vorhergehenden Algorithmus, durch die Konstruktion unabhängiger Teilbäume aufgebaut. Diese werden in Clustern gespeichert. Der Algorithmus heißt *Clusterungs-Algorithmus*.

Wie beim originalen Algorithmus zuvor, startet die Methode mit einer festen Präfixlänge $d = |\omega|$. Dieses d wird Clusterungstiefe genannt. Anders als bei obiger Methode, wird hier an dieser Stelle der gesamte Text t einmal seriell durchlaufen und jede Suffix Position auf ihre dazugehörige Suffixteilmenge abgebildet. Erst danach werden die unabhängigen Teilbäume aufgebaut.

Für jede Menge $S_\omega(t) \in \mathcal{S}(t)$ ($\omega \in \Sigma^d$) wird durch einen geeigneten Konstruktionsalgorithmus der Suffix Tree Teilbaum $\mathcal{T}_\omega(t)$ aufgebaut. Dieses wird realisiert durch das inkrementelle Einfügen der jeweiligen Suffixe $s_i(t^+) \in S_\omega(t)$ in einer zu Anfang leeren Kante. Die Verbesserung zum obigen Algorithmus besteht darin, daß die Einfügeprozedur nicht an der Wurzel des gesamten Suffix Trees startet, sondern direkt im Knoten u, welcher den Pfad $\omega = path(u)$ besitzt. Nach der Konstruktion wird die Hauptspeicher Repräsentation des Teilbaumes $\mathcal{T}_\omega(t)$ in ihre Repräsentation der Festplatte umgeformt und dort gespeichert.

Sollte eine Teilmenge S_ω so klein sein, daß ihre Instanz den Hauptspeicher kaum ausfüllt, so wird versucht vom Pfad $\omega = \text{path}(u)$ des Knotens u den letzten Buchstaben abzutrennen, und somit den Pfad ω_1 zu betrachten. Damit ist ω_1 der größtmögliche

echte Präfix von ω. Es wird geprüft, ob die Tree Instanz von S_{ω_1} in den Speicher paßt. Sollte dem so sein, wird $\mathcal{T}_{\omega_1}$ aufgebaut anstatt $\mathcal{T}_{\omega}$.

Auf diese Weise ist bei dem Verfahren, die einmal anfänglich festgelegte Präfixlänge d nicht über die gesamte Verlaufsdauer konstant. Die Suffix Teilmengen $S_{\omega_1}, \ldots, S_{\omega_k}$ haben ungefähr die gleiche Kardinalität. Und die Teilbäume $\mathcal{T}_{\omega_1}, \ldots, \mathcal{T}_{\omega_k}$ damit fast die gleiche Größe. Diese Methode behebt das Manko von [Hun02].

Sind alle Teilbäume entwickelt, wird am Schluß der obere Restbaum $\mathcal{R}_{d(\omega_i)}(t)$ bis zur variablen Tiefe $d(\omega_i)$ aufgebaut. Es werden alle verbliebenen ω_i eingefügt, für welche $S_{\omega_i} \neq \emptyset$ ist. Dieser Restbaum wird separat auf der Festplatte gespeichert oder im Hauptspeicher belassen.

Im Prinzip ist es möglich verschiedene Konstruktionsalgorithmen jeweils für den Restbaum oder für die Teilbäume $\mathcal{T}_{\omega_i}(t)$ anzuwenden. Man kann hier entweder den wotd-Algorithmus ([Gie03]) oder einen inkrementellen Algorithmus einsetzen. Allen jedoch gleich, es können nur Algorithmen von der Ordnung $O(n^2)$ sein.

Analyse:

1. Teil

Die Suffix Teilmengen S_ω werden, anders als bei [Hun01], tatsächlich berechnet bevor der Konstruktionsalgorithmus startet. Diese Berechnung wird realisiert durch das Verschieben eines Fensters der Länge d – dieses ist zu Beginn noch fest – über den gesamten Text t. Es können alle Partitionen in einer Zeit von $O(n)$ ermittelt werden.

Es wird davon ausgegangen, daß t auf der Festplatte liegt. Beim 1. Teil des Algorithmus wird der Text sequentiell in Blöcken der Größe B gelesen, daher $S(C(\frac{1}{B})n)$.

Die einzelnen Mengen S_ω sind ebenfalls zu groß, als das sie in ihrer Vereinigung im Hauptspeicher gehalten werden könnten. Da man bei jedem Suffix $s_i(t^+)$ nicht weiß, zu welcher Teilmenge S_ω dieser einzuordnen ist, verursacht dies einen zufälligen Diskzugriff. Das heißt, es entsteht die Komplexität $Z(n)$. Durch eine Cachestrategie im Hauptspeicher, bei welcher gewisse *Block*'s zu jedem S_ω angesammelt werden, läßt sich diese zu $Z(C(\frac{1}{Block})\ n)$ reduzieren.

2. Teil

Zu Beginn wird d so gewählt, daß der Suffix Tree der größten Partition im Speicher konstruierbar ist. Wenn man von einer gleichmäßigen Verteilung der Präfixe über t ausgeht, dann sind die n Suffixe gleichmäßig über alle Partitionen verteilt. Dies ist aber für reale DNA Sequenzen nicht der Fall. Es existieren viele kleine Partitionen, S_ω mit $|\omega| = d$, welche einen gemeinsamen Präfixpfad ω_1 haben. Daher ist die Strategie der Clusterung gerade bei DNA Sequenzen erfolgreich anwendbar.

Der Teilbaum $\mathcal{T}_{\omega_i}(t)$ wird durch das Einfügen der $|S_{\omega_i}|$ Suffixe konstruiert. Da n Suffixe über alle Partitionen einzufügen sind, ist die Einfügezeit $O(n^2)$.

In den einzelnen Mengen S_ω sind nur die Indizes i der Suffixe $s_i(t^+)$ gespeichert. Zum Aufbau von $\mathcal{T}_\omega$ werden die S_ω sequentiell von der Festplatte gelesen. Die Vereinigung

aller S_ω ist von der Ordnung $S(C(\frac{1}{B})n)$. Um jedoch einer Textposition i auch die jeweilige Buchstabenfolge des Suffixes zuordnen zu können, muß die entsprechende Position in t gesucht werden. Da die Indizes in S_ω nicht direkt hintereinander liegen, macht das insgesamt $Z(n)$ Diskzugriffe.

Weiterhin werden $|\{\omega_1, \ldots, \omega_k\}|$ Teilbäume auf der Festplatte gespeichert. Damit erhält man als untere Schranke $z(1)$ und als obere Schranke $Z(|\{\omega_1, \ldots, \omega_k\}|)$. Durch die Speicherung der Teilbäume ergeben sich noch $S(C(\frac{1}{C})\, n)$ Zugriffe.

3. Teil

Die gleiche Zeitgrenze, wie bei der Konstruktion der Teilbäume, ist auch gültig für den Restbaum $\mathcal{R}_{d(\omega_i)}(t)$, da $|\mathcal{S}(t)| \in O(n)$ Präfixe einzufügen sind. Durch die Addition, der zu erwartenden Zeiten aller 3 Teile des Algorithmus, erhält man eine insgesamte 'worst case' Laufzeit von $O(n) + O(n^2) + O(n^2) = O(n^2)$. Durch die Addition der anderen Zugriffsarten erhält man eine Komplexität wie folgende:

Satz 6 :
Die Komplexität des Algorithmus ist

$$< (o(n), s(C(2)n), z(n)), (O(n^2), S(C(2\frac{1}{B})\, n), Z(C(\frac{1}{Block})\, n + |\{\omega_1, \ldots, \omega_k\}|)) > .$$

Zusammenfassung beider Algorithmen

Beide Konstruktionsalgorithmen verfolgen mit ihrem Management über die Festplatte die gleiche Strategie und haben ähnliche Laufzeiten.

Jedoch beleuchten beide vorgestellte Algorithmen die praktische Effizienz der Suffix Tree Konstruktion mit zwei Speicherarten.

Analyse der bisherigen Konstruktionsalgorithmen

Die fundamentale wissenschaftliche Fragestellung lautet: Ist es möglich einen Suffix Tree in fast linearer Zeit zu konstruieren, ohne Suffix Links zu verwenden. Diese Frage ist Gegenstand der Forschung und wurde auch in [Col03] angesprochen.

Kapitel 5

Datenpartitionierung – Ein neuer Konstruktionsalgorithmus

Die Grundidee des neuen Algorithmus besteht in der Teilung der Datensequenz und dem anschließenden Mischen der resultierenden Bäume. Der Algorithmus wird durch den untenstehenden Pseudocode beschrieben. In den folgenden Kapiteln der Arbeit werden die einzelnen Teilschritte genauer erklärt. Desweiteren wird zu den Teilschritten eine explizite Analyse der jeweiligen Komplexität angegeben.

Bezeichnungen:

Sei m die Länge eines Strings, für welchen ein Suffix Tree bei durchschnittlicher Implementierung [1] in den Hauptspeicher der Kapazität M ladbar ist.

Die Partitionen $S_j(t)$ (aus Zeile 2) sind in Kapitel 4.3 Gleichung (4.1) definiert.

Der Suffix Baum $\mathcal{T}_{\omega,j}(t)$ ist der Teilbaum aus $\mathcal{T}_{S_j}(t)$, welcher unterhalb des Knotens $\overrightarrow{w}$ beginnt (siehe Zeile 8).

Neuer Daten Partitionierungsalgorithmus – *BauePersistentenBaum* (t, n, m)

Schritt 1: **Partitionierung der Daten**

1: $k := \lceil (n+1)/m \rceil$;
2: **for all** Partitionen $S_j(t)$, $j = 1 \cdots k$ **do**
3: $\quad \mathcal{T}_{S_j}(t) = \mathit{BaueSuffixTree}\ (S_j(t))$;
4: $\quad \mathit{PersistenteSpeicherung}\ (\mathcal{T}_{S_j}(t))$;
5: **end for**;

Schritt 2: **Mischen der Bäume**

[1] Siehe hierzu den Anfang von Kapitel 4.1.

6: $d = 0$;
7: $l = \lceil k/2 \rceil$;
8: **while** ($\sum_{j=1}^{l} |\mathcal{T}_{\omega,j}(t)| > M$)
9: **for all** $(\omega, z) \in \Sigma^d \times \Sigma$ **do**
10: **for all** $j = 1 \cdots l$ **step** $j = j + 2$ **do**
11: *Einlesen* ($\mathcal{T}_{\omega,j}(t)$), *Einlesen* ($\mathcal{T}_{\omega,j+1}(t)$);
12: $\mathcal{T}_{\omega \cup z,j}(t) = \mathit{MergeTree}\ (z, \mathcal{T}_{\omega,j}(t), \mathcal{T}_{\omega,j+1}(t))$;
13: *PersistenteSpeicherung* ($\mathcal{T}_{\omega \cup z,j}(t)$);
14: **end for**;
15: **end for**;
16: $d = d + 1$;
17: $l = \lceil l/2 \rceil$;
18: **end while**;

Schritt 3 : **Endmischen**

19: **for all** $\omega \in \Sigma^d$ **do**
20: $\mathcal{T}^*_\omega(t) = \mathcal{T}_{\omega,1}(t)$;
21: **for all** $j = 2 \cdots l$ **do**
22: *Einlesen* ($\mathcal{T}_{\omega,j}(t)$);
23: $\mathcal{T}^*_\omega(t) = \mathit{MergeTree}\ (\epsilon, \mathcal{T}^*_\omega(t), \mathcal{T}_{\omega,j}(t)$; $\#\epsilon =$ leeres Wort
24: **end for**;
25: *PersistenteSpeicherung* ($\mathcal{T}^*_\omega(t)$);
26: **end for**;

5.1 Schritt 1 : Partitionierung der Daten

5.1.1 Beschreibung

In Schritt 1 wird die Sequenz in so kleine Teilstücke gebrochen, daß für jedes Teilstück eine Konstruktion im Hauptspeicher möglich ist. Diese Partitionierung hängt auch vom Texttyp, in diesem Fall DNA Sequenzen, und der dadurch zu erwartenden Größe eines Baumes ab. Ist die Hauptspeichergröße eines Rechners M und die Länge der Input Sequenz n, so lassen sich k Partitionen der Menge aller Suffixe bilden (Zeile 1). Für diesen ersten Schritt kann man lineare Suffix Tree Konstruktionsalgorithmen verwenden, wie zum Beispiel von McCreight oder Ukkonen (Zeile 3). Ein wesentliches Element von ihnen allen ist, daß sie ihre Linearität nur durch den Gebrauch von Suffix Links erreichen, welche die Größe einer Input Sequenz weiter dezimiert.
Generell ist eine der wichtigsten Funktionen von Suffix Links, ihr Einsatz bei der Konstruktion. Aber auch einige Suchalgorithmen verwenden diese Datenstruktur – auch Algorithmen des Approximativen String Matchings. Mein Fokus bei der Anwendung von Algorithmen, welche über einem Suffix Tree operieren, liegt in der Verwendung von Prozeduren, welche Suffix Links nicht benötigen.

5.1.2 Lineare in-memory Algorithmen

Ein wesentlicher Teil des neuen persistenten Konstruktionsalgorithmus ist die Anwendung eines geeigneten in-memory Algorithmus. Wie in Kapitel 4 beschrieben, existieren derzeit drei bekannte Hauptspeicheralgorithmen:

- der naïve Algorithmus,
- linearer Algorithmus von Weiner / McCreight, dokumentiert in [Wei73],
- und der lineare Algorithmus von Ukkonen.

Der naïve Algorithmus hat eine Laufzeitkomplexität von $O(n^2)$ und es bestehen keine Beschränkungen, welche seine Anwendung unterbinden. Die beiden letzten Algorithmen unterscheiden sich von der Idee her, aufgrund derer sie eine lineare Zeitkomplexität erreichen, nicht wirklich (siehe dazu Kapitel 4.1).

Der Algorithmus von McCreight benötigt für seine Implementierung etwas mehr Speicherplatz und ist ein wenig schneller in der praktischen Laufzeit.

Für den in dieser Arbeit beschriebenen Algorithmus, wurde für Schritt 1 Ukkonens Algorithmus verwendet. Es ist somit ein Teilalgorithmus, der in dieser Arbeit vorgestellten Methode.

Daher werden im weiteren die Grundzüge von Ukkonens Algorithmus beschrieben und es wird ausführt, warum dieser $O(n)$ ist. Um den ursprünglichen Algorithmus in

unseren integrieren zu können, mußte dieser in einigen Punkten modifiziert werden. Es wird daher eine modifizierte Version dargestellt. Eine detaillierte Abhandlung des originalen Algorithmus findet sich auch in [Gus97].

5.1.3 Der modifizierte Algorithmus von Ukkonen – *mukk*

Nach der existierenden Hauptspeichergröße M, wird eine feste Datenpartitionsgröße m gewählt. Das Ziel ist die Konstruktion der kompakten Tries $\mathcal{T}_{S_j}(t)$ aus der Familie $\{S_j(t)\}$, wobei $S_j(t) = S_j := \{s_i \in \mathcal{S}(t) : (j-1)\,m + 1 < i \leq j\,m\}$ mit $1 \leq j \leq \lceil (n+1)/m \rceil$.

Wähle zur Betrachtung ein beliebiges $S \in \{S_j\}_{1 \leq j \leq \lceil (n+1)/m \rceil}$, o.B.d.A. $S = S_1$. Es stellt sich damit folgende Aufgabe:

Aufgabe: Konstruktion des kompakten Tries über der Menge der Wörter $S := \{s_i(t^+) : 1 \leq i \leq m\}$.

BEMERKUNG: Der gesuchte Trie $\mathcal{T}_S$ ist nicht identisch mit dem Suffix Tree über dem Text $t' = t(1, m) = t_1 t_2 \ldots t_m$.

Der Algorithmus von Ukkonen ist ein "on-line"-Algorithmus, was soviel bedeutet, daß der zu prozessierende Text von vorne nach hinten eingelesen werden kann. Dies ist aus zwei Gründen nützlich:

1. Der zu indexierende Text ist so groß, daß er auf der Festplatte gehalten wird. Von dort kann er in einer Sequenz Diskseiten-weise von links nach rechts geladen werden. Dies reduziert die I/O Komplexität.

2. Da die Menge der Wörter S indexiert wird – und gerade nicht der Text $t(1, m) = t_1 t_2 \ldots t_m$ – ist nicht von Anfang an vorgegeben, bis zu welcher Stelle $m + c$ der Text eingelesen werden muß, um von allen Strings den Präsuffix enthalten zu haben.

Im allgemeinen wird beim originalen Algorithmus von Ukkonen weniger Speicherplatz benötigt, als beim Algorithmus von McCreight, aber auf Kosten der Zeit. Doch diese Zeiteinheiten sind gegenüber der Zeit der Diskzugriffe zu vernachlässigen.

Ein wesentliches Element des Algorithmus zum Erreichen der Linearität sind Suffix Links. Suffix Links sind eine zusätzliche Datenstruktur zum Suffix Tree und erhöhen den Speicherplatzbedarf weiter.

Definition: (*Suffix Link*)
Der Zeiger vom Knoten $\overrightarrow{a\omega}$ ($a \in \Sigma, \omega \in \Sigma^*$) auf den Knoten $\overrightarrow{\omega}$ (geschrieben $\overrightarrow{a\omega} \dashrightarrow \overrightarrow{\omega}$) wird, falls er existiert, Suffix Link genannt.

Suffix Links von Blattknoten sind für die Konstruktion nicht von Interesse. Von algorithmischer Seite werden besonders Suffix Links von inneren Knoten genutzt. Abbildung 5.1 zeigt einen Suffix Tree mit allen Suffix Links.

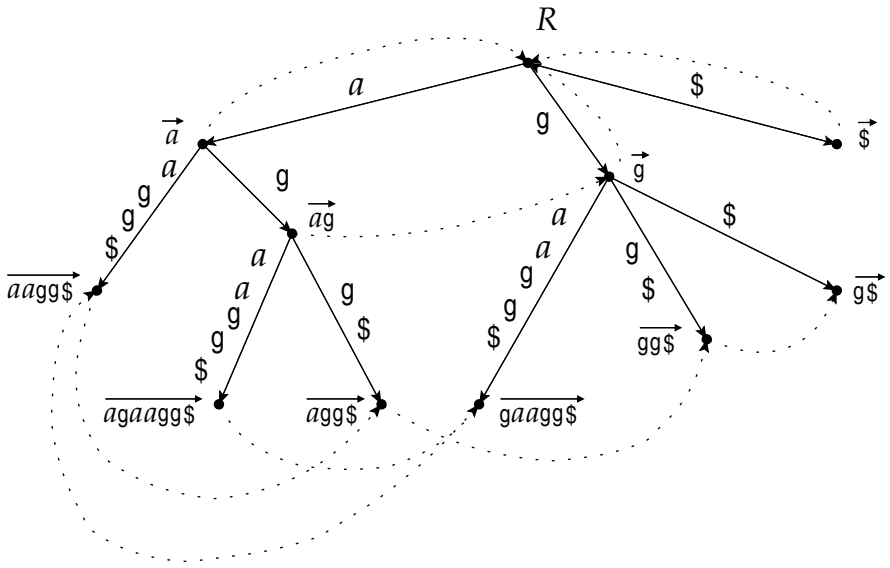


Abbildung 5.1: Suffix Tree mit Suffix Links für den Text $t^+ = agaagg\$$

Vorstellung des Algorithmus

Definition : (*Impliziter Suffix Tree*)
Ein *impliziter Suffix Tree* für einen String t ist der Tree, welcher durch das Entfernen aller \$-Zeichen aus den Kantenbeschriftungen des Suffix Trees zu t^+ entsteht. Daraus resultierende leere Kanten werden gelöscht. Hieraus resultierende Knoten mit nur einem Kind werden zusammengefaßt.

Normalerweise tritt beim Übergang zum impliziten Suffix Tree ein Informationsverlust auf. Nicht für jeden Suffix des Textes ist ein Blattknoten vorhanden. Da wir ohnehin nur Präsuffixe betrachten, das heißt Wörter, welche eindeutige echte Präfixe von Suffixen sind, kann das hier nicht eintreten.

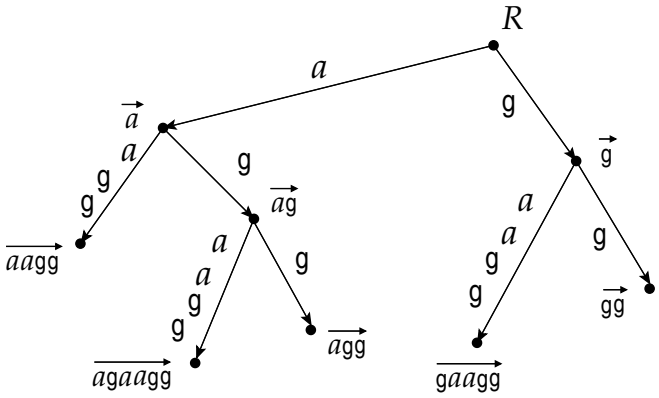


Abbildung 5.2: Impliziter Suffix Tree über den Text $t = agaagg$ aus dem Beispiel von Abbildung 3.1

In *mukk* wird für jeden Präfix $t(1,i)$ der zugehörige implizite Suffix Tree sukzessiv aus dem impliziten Suffix Tree des vorhergehenden Präfixes $t(1,i-1)$ konstruiert. Der letzte implizite Tree über $t(1,m+c)$ ist der gesuchte Trie.

$$i\mathcal{T}_S(\epsilon),\ i\mathcal{T}_S(t(1,1)),\ i\mathcal{T}_S(t(1,2)),\ \ldots,\ i\mathcal{T}_S(t(1,m+c))$$

Da der erste implizite Tree, bestehend nur aus der Wurzel R, bekannt ist, braucht man sich nur auf den Schritt

$$i\mathcal{T}_S(t(1,i)) \Longrightarrow i\mathcal{T}_S(t(1,i+1)) \tag{5.1}$$

zu konzentrieren.

Sei $x \in \Sigma$ mit $t(1,i+1) = t(1,i)x$. Die Menge der neu einzufügenden Strings ist $INPUT_{i+1} = \{sx : s \text{ ist Suffix von } t(1,i)\}$. Es gilt: Alle $\overrightarrow{sx}$ mit $sx \in INPUT_{i+1}$ sind ein Blattknoten in $i\mathcal{T}_S(t(1,i+1))$. Die Menge $INPUT_{i+1}$ läßt sich in folgende 3 Mengen zerlegen, welche unterschiedlich behandelt werden müssen.

$$\begin{aligned}
INPUT_{i+1,\text{irrelevant}_1} &= \{sx : \vec{s} \text{ ist Blattknoten in } i\mathcal{T}_S(t(1,i))\} \\
INPUT_{i+1,\text{irrelevant}_2} &= \{sx : \vec{s} \text{ ist ein innerer Knoten in } i\mathcal{T}_S(t(1,i)) \text{ und} \\
&\qquad sx \text{ ist ein Teilstring von } t(1,i)\} \\
INPUT_{i+1,\text{relevant}} &= \{sx : \vec{s} \text{ ist ein innerer Knoten in } i\mathcal{T}_S(t(1,i)) \text{ und} \\
&\qquad sx \text{ ist } \textit{kein} \text{ Teilstring von } t(1,i)\}
\end{aligned}$$

Es gilt

$$INPUT_{i+1} = INPUT_{i+1,\text{irrelevant}_1} \ \dot{\cup}\ INPUT_{i+1,\text{irrelevant}_2} \ \dot{\cup}\ INPUT_{i+1,\text{relevant}}.$$

<u>1. Fall:</u> $sx \in INPUT_{i+1,\text{irrelevant}_1}$

Dann ist $\vec{s}$ ein Blattknoten in $i\mathcal{T}_S(t(1,i))$. Somit kann $\overrightarrow{sx}$ nur ein Blattknoten in $i\mathcal{T}_S(t(1,i+1))$ sein. Die Blattkante von $\vec{s}$ ist dabei nur durch den Buchstaben x zu erweitern. Das läßt sich mit dem Konzept des impliziten Suffix Tree automatisch realisieren.

Schritt (5.1) reduziert sich dadurch auf das Einfügen aller Wörter von $INPUT_{i+1,\text{relevant}}$.

<u>2. Fall:</u> $sx \in INPUT_{i+1,\text{irrelevant}_2}$

Ist jedoch auch sx selbst ein Teilstring von $t(1,i)$, so ist in dieser Phase i des Algorithmus nichts zu tun.

<u>3. Fall:</u> $sx \in INPUT_{i+1,\text{relevant}}$

Satz 7:
Wenn $sx \in INPUT_{i+1,\text{relevant}}$, dann ist s ein Teilstück des Strings $t(1,i)$, welches mindestens zweimal auftritt.

Bezeichnung:
LCS(s) = der längste Suffix von s, welcher mindestens zweimal in s auftritt.
(LCS = "Longest Common Suffix")

Satz 8 :
Es gilt: $sx \in INPUT_{i+1,\text{relevant}} \iff |\text{LCS}(t(1,i))x| \geq |sx| > |\text{LCS}(t(1,i)x)|$.

Für das Beispiel aus Abbildung 5.1 mit $t = agaagg$ ergibt sich:

$$
\begin{array}{llll}
\epsilon & \Longrightarrow & \text{LCS}(\epsilon) = \epsilon & \Longrightarrow \quad INPUT_{0,\text{relevant}} = \{\epsilon\} \\
a & \Longrightarrow & \text{LCS}(a) = \epsilon & \Longrightarrow \quad INPUT_{1,\text{relevant}} = \{a\} \\
g & \Longrightarrow & \text{LCS}(ag) = \epsilon & \Longrightarrow \quad INPUT_{2,\text{relevant}} = \{g\} \\
a & \Longrightarrow & \text{LCS}(aga) = a & \Longrightarrow \quad INPUT_{3,\text{relevant}} = \emptyset \\
a & \Longrightarrow & \text{LCS}(agaa) = a & \Longrightarrow \quad INPUT_{4,\text{relevant}} = \{aa\} \\
g & \Longrightarrow & \text{LCS}(agaag) = ag & \Longrightarrow \quad INPUT_{5,\text{relevant}} = \emptyset \\
g & \Longrightarrow & \text{LCS}(agaagg) = g & \Longrightarrow \quad INPUT_{6,\text{relevant}} = \{agg, ag\}
\end{array}
$$

Beginnend bei dem leeren String ϵ ist LCS(ϵ) = ϵ. Es läßt sich folgender Algorithmus schreiben :

Algorithmus – $mukk = BaueSuffixTree(\mathcal{S}_j(t))$

1: $set = 0$;
2: **for all** $i = 0 \ldots \lceil \frac{n}{m} \rceil + c$ **do**
3: $u :=$ LCS$(t(1,i))t(i,i)$; # mit $\epsilon := t(1,0)$
4: **while** $\neg$ *TrittAuf*$(u, \mathcal{T}_S(t))$; **do**
5: *Einfügen*$(u, \mathcal{T}_S)$;
6: $u :=$ *StreicheErstenBuchstaben*(u);
7: *GeheZuPosition*(u);
8: **if** $i > \lceil \frac{n}{m} \rceil$ **then** $set = 1$;
9: **end while**
10: **if** $set = 1$ **then** BREAK;
11: LCS$(t(1,i)t(i,i)) := u$;
12: **end for**

Satz 9 :
Der Algorithmus *mukk* ist von der Ordnung $O(n)$ bezüglich der Hauptspeicher Komplexität.

Um tatsächlich die Linearität zu beweisen, benötigt man eine Vorstellung von Ortsangaben in einem Suffix Tree.

Definition : (*Position*)
Sei ein String $s \in \Sigma^*$ gegeben. Dann ist die *Position* von s in einem gegebenen Suffix Tree $\mathcal{T}_S$ definiert durch: $pos(s) =< Knoten, Rest\_der\_Kantenbeschriftung >$.

Beispiel : Gegeben sei der Suffix Tree über dem Text $t^+ = agaagg\$$

$$s = aga \Rightarrow pos(s) =< \overrightarrow{ag}, a >$$
$$s = aag \Rightarrow pos(s) =< \overrightarrow{a}, ag >$$
$$s = g \quad \Rightarrow pos(s) =< \overrightarrow{g}, \epsilon >$$

Beweis des Satzes 9 :

Zu zeigen ist: Die Schritte in Zeile 4 und 7 sind in konstanter Zeit zu realisieren.

$Pos(s)$ definiert die Kante oder den Knoten, den man nach $path(s)$ ab der Wurzel erreicht. Die Funktion *TrittAuf* ist damit von der Größenordnung $O(1)$, denn: Wenn $TrittAuf(u, \mathcal{T}_S(t))$ mit $u = asx$ aufgerufen wird, dann befindet man sich an dem Ort $pos(as)$ im Suffix Baum $\mathcal{T}_S$. Ab dieser Position ist nur die Existenz eines Kindpfades beginnend mit dem Präfix x zu suchen. Und das ist genau eine Zeiteinheit.

Durch die Existenz von *Suffix Links* ist der Schritt in Zeile 7, von der Ordnung $O(1)$. Die Benutzung von Suffix Links benötigt genau eine Zeiteinheit. Bevor die Funktion $GeheZuPosition(u)$ aufgerufen wird, befindet man sich in $\mathcal{T}_S$ am Ort $pos(as)$. Durch den Aufruf der Funktion wechselt man von dort mit Hilfe der Suffix Links zur Position $pos(s)$. Damit ist auch Zeile 7 in konstanter Zeit abzuarbeiten. □

BEMERKUNG zum Beweis :

1. Würde diese zusätzliche Struktur nicht existieren, so müßte nach dem Streichen des ersten Buchstaben, die Position von $path(u)$ neu im Baum gesucht werden. Das wird normalerweise duch eine Suche ab der Wurzel des Baumes realisiert. Man hätte damit eine Komplexität der Ordnung $O(|path(u)|)$, und würde insgesamt keinen linearen Algorithmus erreichen.

ALLGEMEINE BEMERKUNG :

1. Im Prinzip wäre auch jeder andere Hauptspeicheralgorithmus denkbar. Doch bei keinem wäre eine solch nahtlose Präsuffix-Berechnung möglich.

2. Die oben eingeführte Konstante c ist von der Art des Textes abhängig. Da gerade bei DNA Strings sehr lange sich wiederholende Teilstücke auftreten können, bis über 200 Basen, muß in diesem Fall auch $c > 200$ gelten. Wird die Konstante nicht richtig gewählt, läuft der Algorithmus fehl. Eine exakte Berechnung des Suffix Trees, ist nicht gegeben. Allerdings ist der entstehende Fehler sehr gering.

5.1.4 Persistente Speicherung

Nach der Verarbeitung der ersten Datenpartition ist der Hauptspeicher voll belegt. Um auch von der nachfolgenden Datenpartition den Tree aufbauen zu können, muß dieser erste Tree aus dem Hauptspeicher auf die Festplatte abgebildet werden. Eine effiziente Darstellung wurde im vorherigen Kapitel 3 beschrieben. Bei der Konvertierung der in-memory Datenstruktur in die persistente Struktur, muß der Baum einmal auf minimalem Wege durchlaufen werden.

Definition : (*Weg*)
Ein Weg w im Baum $\mathcal{T}$ ist eine endliche Folge von Kanten $\{n_{1,1} \rightarrow n_{1,2}, n_{2,1} \rightarrow n_{2,2}, \ldots, n_{\ell,1} \rightarrow n_{\ell,2}\}$, $\ell \in \mathbb{N}$, für welche gilt:

1. $(n_{i,1}, n_{i,2}) \in \mathcal{V}_\mathcal{T} \times \mathcal{V}_\mathcal{T}, \quad n_{i,1} \rightarrow n_{i,2} \in \mathcal{E}_\mathcal{T}$,
2. $n_{i,2} = n_{i+1,1}$.

Ein *gesamter Weg* über dem Baum $\mathcal{T}$ ist ein Weg, welcher jede Kante des Baumes mindestens einmal durchläuft.

Folgerung :
Das Minimum aller gesamten Wege über einem Baum ist der "Depth-First" - Weg. Der "Depth-First" - Weg durchläuft jede Kante des Baumes genau zweimal.

Der Algorithmus der Depth-First Suche spart sich bei einer einmal durchlaufenen Kante den Rückweg (zweiter Durchlauf) durch das Merken des ausgehenden Knotens in einem Stack.

Algorithmus : DFS ($\mathcal{T}$) – *Depth First Suche*

Input: Ein allgemeiner Trie $\mathcal{T}$
0: *DFS-Teilbaum* (R, nil); #[2]

DFS-Teilbaum $(u, left)$:
1: **for all** $v \in \text{Kind}(u) \wedge left \prec v$ **do** #[3]

[2]Der leere Knoten wird mit *nil* bezeichnet.
[3]Die Operation $left \prec v$ bedeutet, daß $left \in \mathcal{V}_\mathcal{T}$ ein links liegender Geschwisterknoten von $v \in \mathcal{V}_\mathcal{T}$ ist.

2: **if** $v \neq$ Blattknoten **then**
3: *DFS-Teilbaum* (v, nil);
5: **end if**
4: $left = v$;
6: **end for**

Die Laufzeit des DFS Algorithmus ist $O(|\mathcal{V}_T| + |\mathcal{E}_T|)$.

Der Tree wird von links nach rechts und von oben nach unten in linearer Zeit durchlaufen (Zeile 4). Bei diesem sequentiellen Durchlauf können die einzelnen Knoten und Unterbäume der transienten Datenstruktur aus dem Hauptspeicher gelöscht und in einer Sequenz auf die Festplatte geschrieben werden.
An diesem Punkt werden die Suffix Links nicht mehr benötigt und somit bei der Abbildung nicht übertragen. Es wäre aber auch möglich diese zu belassen.

Danach ist der Hauptspeicher frei, um die nächste Datenpartition verarbeiten zu können. Insgesamt wird dieser Schritt k-mal iteriert (Zeile 2,5).

5.1.5 Analyse von Schritt 1

Da der Eingabetext einmal sequentiell durchlaufen wird und dieser wegen seiner Größe nicht im Hauptspeicher gehalten werden kann, ist die Zeitkomplexität für die Funktion *BaueSuffixTree* linear – $O(n)$, genauer $O(\frac{n}{m} \cdot (m + c))$ für den Hauptspeicher und $S(\frac{n}{m} \cdot \frac{m}{B})$ für den sequentiellen Diskzugriff. Zufällige Diskzugriffe treten nicht auf. Das ist die erste Stufe des neuen Algorithmus.

Betrachtung zur Funktion *PersistenteSpeicherung* :

Beobachtung : Der im Kapitel 2 vorgestellte Suffix Tree hat eine Knotenanzahl der Ordnung $O(C(2)n)$ und eine Höhe der Ordnung $O(\log_2 n)$.

Beweis :
Gegeben sei ein Suffix Tree $\mathcal{T}_S(t^+)$ über einem Text der Länge $|t^+| = n + 1$.

Dann hat dieser Tree genau $n + 1$ Blattknoten. Da jeder innere Knoten nach Definition mindestens 2 Kindknoten besitzt, kann es auf dem Level über den Blattknoten höchstens $\frac{n+1}{2}$ Knoten geben. Ein weiteres Level darüber höchstens $\frac{n+1}{4}$, usw. Dann gilt als obere Grenze aller Level k, daß für k gelten muß: $2^k \leq n + 1 < 2^{k+1}$. Das heißt, es ist: $\log_2(n + 1) - 1 < k \leq \log_2(n + 1)$. Damit ist die maximale Anzahl der Level von der Ordnung $O(\log_2 n)$.

Die Anzahl der Knoten läßt sich nun folgendermaßen abschätzen:

$$\begin{aligned}
|\mathcal{V}_{\mathcal{T}_S}| &\leq \frac{n+1}{2^0} + \frac{n+1}{2^1} + \frac{n+1}{2^2} + \ldots + \frac{n+1}{2^k} \\
&= (n+1) \cdot \sum_{i=0}^{k} \left(\frac{1}{2}\right)^i = (n+1)\frac{1-\left(\frac{1}{2}\right)^{k+1}}{1-\frac{1}{2}} = (n+1) \cdot 2\left(1-\left(\frac{1}{2}\right)^{k+1}\right) \\
&\leq (n+1) \cdot 2\left(1-\left(\frac{1}{2}\right)^{\log_2(n+1)-1+1}\right) \\
&= (n+1) \cdot 2\left(1-\frac{1}{n+1}\right) = 2n+2-2 = 2n \approx O(n)
\end{aligned}$$

□

BEMERKUNG:

1. Der Platzbedarf ist ebenfalls linear, falls sich jede Kante als konstante Speichergröße darstellen läßt. Das läßt sich durch eine Referenzierung in den Text erreichen. Für die in Kapitel 3 besprochene Variante läßt sich dies gerade nicht erreichen.

2. Dieses Resultat kann nicht für allgemeine Tries gelten, da diese nicht kompakt sind.

Da es zu jedem Knoten, außer dem Wurzelknoten, eine eingehende Kante gibt, ist auch die Anzahl der Kanten von der Ordnung $O(C(2)n)$. Damit gilt für die Laufzeit $(O(\,C(4)\,n),$ $S(C \cdot n/B)\,,\,0)$.

Satz 10:
Die obere Schranke der Laufzeit des ersten Schrittes beträgt $(O(n), S(n/m), 0)$.

5.2 Schritt 2: Mischen der Bäume

5.2.1 Beschreibung

Als Resultat von Schritt 1 erhält man k persistente Suffix Trees. Die Kanten dieser Trees sind nicht mit einer Referenzierung in den Text gelabelt, sondern mit den tatsächlichen Buchstaben des Alphabetes. Diese Kantenbeschriftung ist wichtig, um Schritt 2 durchführen zu können.
Würde man an dieser Stelle aufhören, so würde der Index aus diesen k Bäumen bestehen und man müßte bei der Suche (z.B. Exact String Matching) auch die k Bäume gleichzeitig durchsuchen. In Schritt 2 müssen nun die k resultierenden Suffix Trees gemischt werden, um gleiche Wurzelkanten miteinander zu vereinigen und damit den endgültig gesuchten Tree zu erhalten .

Gegeben: $\mathcal{T}_i(t^+)$, für $1 \leq i \leq k$

Gesucht: $\mathcal{T}_S(t^+) := \mathcal{T}_1(t^+) \dot{\cup} \ldots \dot{\cup} \mathcal{T}_k(t^+)$

[4]Da jeder der k Suffix Trees so groß ist, daß er beim Einladen in den Hauptspeicher diesen vollständig ausfüllen würde, so wird man auf gar keinen Fall zwei gesamte Bäume gleichzeitig in den Hauptspeicher laden können, um sie dort zu vereinigen (mischen). Daher muß jeder der k Bäume aufgeteilen *bzw.* partitioniert werden, und zwar auf die gleiche Weise. Es wird damit aus zum Beispiel zwei Bäumen, zwei Mengen von Teilbäumen gebildet.

5.2.2 Partitionierung eines Suffix Tree

Gegeben sei ein Suffix Tree $\mathcal{T}_S(t)$ über einem beliebigen Text t^+. Die Menge $U_{\mathcal{T}_S}$, die Menge aller Knoten und (beschrifteten) Kanten, ist eine Darstellung eines Suffix Trees.

Definition : (*Segmentierung/Partition eines Suffix Trees*)
Eine endliche Folge von Teilmengen, $U_1, U_2, \ldots, U_n$ aus $U_{\mathcal{T}_S}$ mit U_i paarweise disjunkt, nennt sich Segmentierung *bzw.* Partition des gegebenen Suffix Trees $\mathcal{T}_S$, falls

$$U_{\mathcal{T}_S} = \bigcup_{i=1}^{n} U_i$$

gilt. Jedes U_i wird Segment genannt.

Definition : (*zusammenhängendes Segment*)
Ein Segment U_i ist zusammenhängend, falls ein gesamter Weg über dem Segment existiert.

BEMERKUNG: Ob ein Graph zusammenhängend ist oder nicht kann mit der Tiefen- oder Breitensuche verifiziert werden.

Darstellung eines Segmentes :

Anhand des Beispieles aus Abbildung 3.1 :

Es ist $U_{\mathcal{T}_S} = \{\mathcal{V}_{\mathcal{T}_S}, \mathcal{E}_{\mathcal{T}_S}\}$ mit

$$\begin{aligned}
\mathcal{V}_{\mathcal{T}_S} &= \{R, \overrightarrow{a}, \overrightarrow{g}, \overrightarrow{\$}, \overrightarrow{ag}, \overrightarrow{agaagg\$}, \overrightarrow{gaagg\$}, \overrightarrow{aagg\$}, \overrightarrow{agg\$}, \overrightarrow{gg\$}, \overrightarrow{g\$}\} \\
\mathcal{E}_{\mathcal{T}_S} &= \{R \xrightarrow{a} \overrightarrow{a}, R \xrightarrow{g} \overrightarrow{g}, R \xrightarrow{\$} \overrightarrow{\$}, \overrightarrow{a} \xrightarrow{agg\$} \overrightarrow{aagg\$}, \overrightarrow{a} \xrightarrow{g} \overrightarrow{ag}, \overrightarrow{ag} \xrightarrow{aagg\$} \overrightarrow{agaagg\$}, \overrightarrow{ag} \xrightarrow{g\$} \overrightarrow{agg\$}, \\
&\quad \overrightarrow{g} \xrightarrow{aagg\$} \overrightarrow{gaagg\$}, \overrightarrow{g} \xrightarrow{g\$} \overrightarrow{gg\$}, \overrightarrow{ag} \xrightarrow{\$} \overrightarrow{ag\$}\}.
\end{aligned}$$

BEMERKUNG: Normalerweise wird ein allgemeiner Graph durch seine Menge an Knoten und Kanten dargestellt. Beim speziellem Graphen, dem allgemeinen Trie, lassen sich

[4]Die disjunkte Vereinigung zweier Suffix Trees ist als Vereinigung gleicher Pfade definiert.

innere Knoten und Blattknoten unterscheiden. Dabei besitzt jeder innere Knoten mindestens eine ausgehende Kante. Daher ist ein allgemeiner Trie durch die Menge seiner Kanten $\mathcal{E}_{\mathcal{T}_S}$ schon eindeutig bestimmt, $U_{\mathcal{T}_S} \approx \mathcal{E}_{\mathcal{T}_S}$.

Dann läßt sich ein Segment darstellen durch $U_i = \{\mathcal{V}_{\mathcal{T}_S,i}, \mathcal{E}_{\mathcal{T}_S,i}\}$, wobei $\mathcal{V}_{\mathcal{T}_S,i} \subseteq \mathcal{V}_{\mathcal{T}_S}$, $\mathcal{E}_{\mathcal{T}_S,i} \subseteq \mathcal{E}_{\mathcal{T}_S}$.

Im betrachtetem Beispiel würden folgende Teilmengen eine Segmentierung bilden:

$$
\begin{aligned}
U_1 &= \{\{R, \overrightarrow{a}, \overrightarrow{g}, \overrightarrow{\$}, \overrightarrow{ag}, \overrightarrow{agaagg\$}\}, \{R \xrightarrow{a} \overrightarrow{a}, R \xrightarrow{g} \overrightarrow{g}, R \xrightarrow{\$} \overrightarrow{\$}, \overrightarrow{g} \xrightarrow{aagg\$} \overrightarrow{gaagg\$}\}\} \\
U_2 &= \{\{\overrightarrow{gaagg\$}, \overrightarrow{aagg\$}, \overrightarrow{agg\$}, \overrightarrow{gg\$}, \overrightarrow{g\$}\}, \\
&\quad \{\overrightarrow{a} \xrightarrow{agg\$} \overrightarrow{aagg\$}, \overrightarrow{a} \xrightarrow{g} \overrightarrow{ag}, \overrightarrow{ag} \xrightarrow{aagg\$} \overrightarrow{agaagg\$}, \overrightarrow{ag} \xrightarrow{g\$} \overrightarrow{agg\$}\}\}
\end{aligned}
$$

Für die weiteren Überlegungen ist nicht jede Segmentierung sinnvoll, zum Beispiel insbesondere diejenigen, welche Knoten und Kanten auseinanderreißen. Daher definieren wir aus der Klasse aller Segmentierungen, die für den Algorithmus zulässigen Segmentierungen.

Definition: (*zulässige Segmentierung*)
Eine gegebene Segmentierung eines Suffix Trees heißt *zulässig*, falls der Tree nur in zusammenhängende Segmente zerlegt wird.

BEMERKUNG: Gerade zulässige Segmentierungen bewirken einen sequentiellen Diskzugriff und reduzieren damit die Laufzeitkomplexität.

Beobachtung 1:
Ein zusammenhängendes Segment ist wieder ein allgemeiner Trie. Insbesondere: Ein zusammenhängendes Segment eines Suffix Trees ist ein kompakter Trie.

Beweis:
Ist U ein zusammenhängendes Segment, so gibt es einen Knoten v, für welchen die Länge des Pfades ab der Wurzel zum Knoten v minimal ist. Unterhalb dieses eindeutigen Knotens liegt nach Voraussetzung ein zusammenhängendes Segment, welches nach Definition ein gewurzelter Baum ist. □

ZUSATZ: Innere Knoten des Ursprungsbaumes werden nicht zu Blättern eines Segmentes. Denn sonst müßte ein Segment existieren, welchem der Wurzelknoten fehlt. Dieses Segment wäre aber kein zusammenhängendes Segment. Die Endstellen eines Segmentes bilden offene Kanten oder gegebene Blattknoten.

Beobachtung 2:
Sei $P = \{U_1, U_2, \ldots, U_n\}$ eine zulässige Segmentierung eines gegebenen Suffix Tree $\mathcal{T}_S(t^+)$. Dann kann jede Segmentierung U_i eindeutig mit einem String $\omega_i \in \Sigma^*$ identifiziert werden. Also

$$F : P \longrightarrow \Sigma^*, \qquad U_i \mapsto \omega_i$$

existiert.

Beweis :
Aus Beobachtung 1 – jedes Segment ist ein Wurzelbaum und besitzt damit eine eindeutig bestimmte Wurzel. Diese Wurzel u ist ein beliebiger Knoten aus dem gegebenen Suffix Tree $\mathcal{T}_S$. Der String ω, welcher der Pfad von der Wurzel zum Knoten u ist, $\omega = path(u)$, ist eindeutig bestimmt und charakterisiert das Segment U. □

5.2.3 Optimale Segmentierung

Definition : (*optimale Segmentierung für einen gegebenen Algorithmus*)
Eine zulässige Segmentierung ist optimal für einen gegebenen Algorithmus, falls eine durch den Algorithmus gegebene Anzahl von p Segmenten gleichzeitig in den Hauptspeicher ladbar ist und zu diesen p Segmenten durch $\{G_1, G_2, \ldots, G_p\}$ p Größenangaben gegeben sind.

BEMERKUNG: Der kanonische Fall für die meisten Algorithmen ist die Gleichheit in den Größen.

Größe eines Segmentes : Zu einem gegebenen Segment U sei dessen Größe durch $|U|$ als Menge aller Knoten und Kanten, die zum Segment gehören, definiert.

Die tatsächliche Größe wird ebenfalls noch durch die Größe der Kantenbeschriftung festgelegt. Diese wollen wir hier durch eine feste Konstante abschätzen. Sie wäre jedoch leicht in die nachfolgenden Betrachtungen einzuführen.

Für oben eingeführte Konstruktionsmethode ist eine optimale Segmentierung, eine Segmentierung in $p = 2$ Segmente von gleicher Größe.
Sei M die Hauptspeichergröße und C die Anzahl von Bytes für einen Knoten und dessen ausgehende Kante benötigt wird. Es muß also gelten: $|U| \cdot C \leq \frac{M}{2}$. Im optimalen Fall jedoch: $|U| \cdot C \approx \frac{M}{2}$.

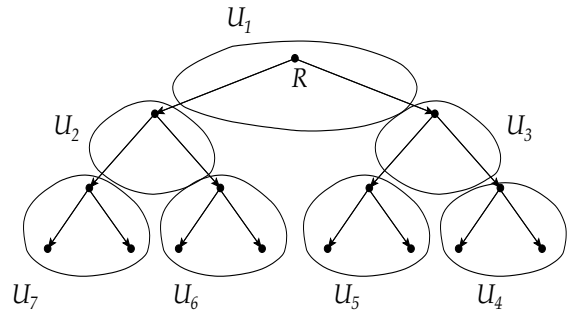
Problem : Finde eine Methode, welche eine optimale Segmentierung eines gegebenen Baumes liefert.

Das heißt, man wünscht eine Segmentierung $\{U_1, \ldots, U_n\}$ zu finden, für welche $|U_i| \leq G_i$ mit möglichst $|U_i| \approx G_i$ gilt. Bei dieser Berechnung der optimalen Segmentierung wird es immer ein Segment oder zwei geben, welche nicht den geforderten Größenangaben entsprechen, das heißt das $|U_i| < G_i$ gilt.

Spezialfall 1 : Balancierter Baum

Definition : Ein Baum heißt balanciert, falls er eine feste Höhe und eine feste Anzahl von Kindknoten besitzt.

Methode – sei K die Anzahl von Kindknoten und k die Anzahl der Schichten eines balancierten Baumes. Gesucht sei eine optimale Segmentierung $\{G_1, G_2, \ldots, G_p\}$ (im Bei-



$$|U_1| = |U_2| = |U_3| = |U_4| = 5$$
$$|U_5| = |U_6| = |U_7| = 3$$

Abbildung 5.3: Beispiel eines balancierten Baumes der Höhe 3 mit jeweils 2 Kindknoten

spiel von Abbildung 5.3 ist es $\{5,5,5,5,5,5,5\}$). Die Größe eines Segmentes unterhalb einem gegebenen Knoten ergibt sich durch:

$$2 \cdot K^k + 2 \cdot K^{k-1} + \ldots + 2 \cdot K^1 + 1 = 2\left(\frac{1-K^{k+1}}{1-K}\right) - 1$$

Für jedes Segment muß also $2\left(\frac{1-K^{k_i+1}}{1-K}\right) - 1 \leq G_i$. Damit berechnet sich die Anzahl von Schichten, welche jeweils zu einem Segment zusammengefaßt werden müssen, aus:

$$k_i \leq \log_K[(G_i+1) \cdot \frac{1}{2}(K-1)+1] - 1.$$

Spezialfall 2 : Unbalancierter Baum

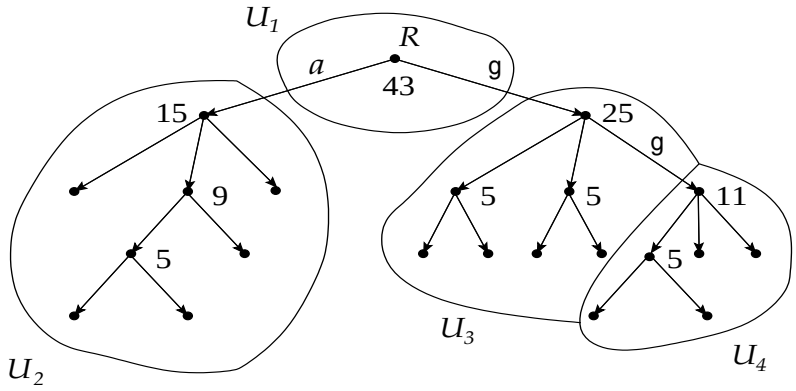
Diese Bäume sind charakterisiert durch eine völlig unausgewogene Treetopologie.

Methode zum Finden einer optimalen Segmentierung für unbalancierte Bäume

1. Durchlauf Tiefensuche (TS)
   Bei diesem Durchlauf des Graphen wird für jeden Knoten die Größe des darunterliegenden Baumes berechnet. Die gesamte Komplexität ist $O(n)$, denn:
   Die Anzahl aller Knoten ist $|\mathcal{V}_{\mathcal{T}_S}| \leq 2n$ (siehe Kapitel 5.1.5). Der Algorithmus der Tiefensuche besucht jeden Knoten $(1+|\Sigma|)$-mal. Dann läßt sich die Gesamtanzahl aller Schritte abschätzen:

   $$|\text{Schritte}_{TS}| \leq (1+|\Sigma|) \cdot 2n$$

2. Durchlauf Breitensuche (BS)
   In diesem Schritt werden die jeweils schon geeigneten Unterbäume abgetrennt. Mit einer analogen Argumentation wie in Schritt 1, ergibt sich $|\text{Schritte}_{BS}| \leq (1+|\Sigma|) \cdot 2n$. Das heißt, die Komplexität ist $O(n)$.



$|U_1| = 3, \quad U_1 \mathrel{\widehat{=}} R$
$|U_2| = 15, \quad U_2 \mathrel{\widehat{=}} a$
$|U_3| = 14, \quad U_3 \mathrel{\widehat{=}} g$
$|U_4| = 11, \quad U_4 \mathrel{\widehat{=}} gg$

Abbildung 5.4: Beispiel unbalancierter Baum

Der Baum soll in gleichgroße Teile mit $G_i = 15$ zerlegt werden. Die Knoten des Baumes sind mit der Größe ihres jeweiligen Unterbaumes beschriftet, außer die Blätter, welche die Größe 1 haben. Es läßt sich zur Berechnung der optimalen Segmentierung nicht einfache Division mit Rest anwenden, wegen der Unausgewogenheit der Treetopologie.

Die beschriebene Methode liefert die optimale Segmentierung in linearer Zeit. Der maximale Vorfaktor bei jedem Schritt liegt bei $4 \cdot (1 + |\Sigma|)$.

Die Bedeutung der optimalen Segmentierung liegt in der effizienten Ausnutzung des Hauptspeichers.

5.2.4 Festlegung eines Muster geeignet bei DNA Sequenzen

Aufgrund Beobachtung 2 läßt sich jedes Segment eindeutig durch einen String repräsentieren, siehe Abbildung 5.4. Die gesamte Segmentierung kann man nun als eine Menge von Strings darstellen. Beim Beispiel aus Abbildung 5.4 ist g ein Teilwort des Wortes gg. Das bedeutet, daß gg ein Kindsegment des Segmentes g ist.

Definition : (*Muster*)
Eine Menge von Strings, welche jeweils Teilwörter des Textes t^+ sind, nennt sich Muster über dem Text t^+.

BEMERKUNG: Durch die Angabe eines Musters ist eindeutig eine Segmentierung definiert.

Bei bekannten Texten sind auch die Häufigkeitsverteilungen bekannt, damit auch die Ausprägungen der Unbalanciertheit des Baumes. Es ist daher meist in der Praxis völlig ausreichend ein heuristisches Verfahren, durch Angabe eines Musters anzuwenden, um auf diese Weise eine "*fast*" optimale Segmentierung zu erreichen.

Analyse von DNA Sequenzen und deren geeignete Muster

Verteilung der Pattern bei Eukaryonten und Prokaryonten

Hierfür wurden die Genome folgender Eukaryonten[5] gewählt: 202,7 Mbp menschlichen Genoms (aus Chromosom 2 und 22), 120 Mbp *Arabidopsis Thaliana*, 224,8 Mbp Teilstück von Chromosom 2 des Genoms der Maus. Es wurde die Anzahl der jeweiligen Pattern in der Menge der Eukaryonten insgesamt genommen, und diese in Bezug zur Gesamtlänge aller Genome gestellt. In den Tabellen 5.1, 5.2 und 5.3 ist das prozentuale Auftreten eines Pattern in den gewählten eukaryontischen Genomen angegeben.

Die Menge der Prokaryonten[6] ergibt sich aus allen derzeit verfügbaren Bakteriengenomen (zum Beispiel unter: *http://www.ncbi.nlm.nih.gov/genomes/*) mit einer Gesamtlänge von ungefähr 450 Mbp. Die Werte des Autretens der Pattern sind in Prozent angegeben.

Aus den Tabellen 5.1 läßt sich erkennen, daß die einzelnen Buchstaben *A*, *T*, *G* und *C* bei Prokaryonten nahezu gleichverteilt sind, während bei Eukaryonten die *A*, *T* Werte vor den *C*, *G* Werten überwiegen. Dies entspricht der Beobachtung, daß die kodierenden Bereiche, die Gene, in der Sequenz mehr *C*, *G* Zeichen enthalten, als die nicht kodierenden Teilstücke. Das prokaryontische Genom besteht aus einer dichten Reihenfolge von Genen. Im eukaryontischem Genom gibt es lange Teilstücke, welche nicht kodierend sind, daß heißt nicht in Eiweiße exprimiert werden. Jedoch unabhängig von dieser Gleichverteilung in Prokaryonten, sind die Pattern AA, TT, AAA und TTT in allen Genomen wesentlich öfter vertreten. An der Verteilung der Pattern der Länge 4 sieht man deutich, daß in Eukaryonten Pattern, welche mit *A* und *T* beginnen, gehäuft auftreten, in Prokaryonten Pattern mit *C* und *G*. Desweiteren zeigt die Verteilungskurve bei Eukaryonten starke Schwankungen. Das Maximum bei $AAAA$ mit $1,43\%$ und das Minimum bei $CGCG$ mit $0,03\%$. Auch Teilstrings der Länge 4 sind bei Prokaryonten gleichverteilter (siehe Tabelle 5.3).

Tabelle 5.1: Verteilung der Pattern der Länge 1 und 2 in eukaryontischen und prokaryontischen Genomen

Pattern	Euk.	Pro.
A	29,87	24,91
T	29,85	24,95
G	20,15	25,07
C	20,13	25,07

Pattern	Euk.	Pro.
AA	9,88	7,62
AT	7,88	6,74
AG	6,92	5,46
AC	5,20	5,10
TA	6,70	4,86
TT	9,85	7,64
TG	7,16	6,37
TC	6,14	6,08

Pattern	Euk.	Pro.
GA	6,15	6,08
GT	5,20	5,11
GG	4,84	6,12
GC	3,95	7,76
CA	7,15	6,36
CT	6,92	5,46
CG	1,22	7,12
CC	4,84	6,12

[5] In den Tabellen 5.1, 5.2 und 5.3 mit Euk. abgekürzt.
[6] In den Tabellen 5.1, 5.2 und 5.3 mit Pro. abgekürzt.

Tabelle 5.2: Verteilung der Pattern der Länge 3 in eukaryontischen und prokaryontischen Genomen

Pattern	Euk.	Pro.	Pattern	Euk.	Pro.	Pattern	Euk.	Pro.	Pattern	Euk.	Pro.
AAA	3,74	2,53	TAA	2,09	1,50	GAA	2,06	1,82	CAA	1,98	1,76
AAT	2,46	1,92	TAT	2,12	1,49	GAT	1,44	1,76	CAT	1,86	1,56
AAG	2,09	1,66	TAG	1,32	0,85	GAG	1,66	1,22	CAG	1,85	1,72
AAC	1,58	1,50	TAC	1,16	1,01	GAC	0,99	1,26	CAC	1,46	1,32
ATA	2,13	1,48	TTA	2,09	1,51	GTA	1,16	1,01	CTA	1,32	0,85
ATT	2,44	1,93	TTT	3,73	2,54	GTT	1,59	1,50	CTT	2,09	1,66
ATG	1,86	1,56	TTG	1,98	1,76	GTG	1,47	1,32	CTG	1,85	1,72
ATC	1,44	1,76	TTC	2,05	1,82	GTC	0,99	1,26	CTC	1,66	1,23
AGA	2,33	1,38	TGA	1,96	1,66	GGA	1,51	1,36	CGA	0,34	1,66
AGT	1,60	1,08	TGT	2,11	1,25	GGT	1,15	1,45	CGT	0,34	1,32
AGG	1,63	1,32	TGG	1,73	1,69	GGG	1,17	1,20	CGG	0,30	1,91
AGC	1,35	1,67	TGC	1,34	1,76	GGC	1,01	2,11	CGC	0,24	2,21
ACA	2,11	1,24	TCA	1,96	1,67	GCA	1,34	1,76	CCA	1,73	1,69
ACT	1,60	1,08	TCT	2,33	1,39	GCT	1,36	1,67	CCT	1,63	1,32
ACG	0,33	1,32	TCG	0,34	1,66	GCG	0,24	2,21	CCG	0,30	1,92
ACC	1,15	1,45	TCC	1,51	1,36	GCC	1,01	2,11	CCC	1,17	1,20

Tabelle 5.3: Alle Pattern der Länge 4, deren Anteil $\geq 0,6\%$ ist (in eukaryontischen und prokaryontischen Genomen)

Pat.	Euk.	Pat.	Euk.	Pat.	Euk.	Pat.	Euk.
AAAA	1,43	ATAT	0,73	TTAT	0,67	TGTT	0,65
AAAT	0,97	ATTT	0,96	TTTA	0,80	TGTG	0,60
AAAG	0,73	AGAA	0,82	TTTT	1,43	TCTT	0,74
AAAC	0,61	AGAG	0,61	TTTG	0,75	GAAA	0,75
AATA	0,72	ACAA	0,61	TTTC	0,75	GTTT	0,61
AATT	0,73	TAAA	0,81	TTGT	0,61	CAAA	0,75
AAGA	0,74	TATA	0,63	TTCA	0,63	CTTT	0,73
AACA	0,65	TATT	0,71	TTCT	0,82	CTCT	0,61
ATAA	0,67	TTAA	0,66	TGAA	0,63		

Pat.	Pro.	Pat.	Pro.
AAAA	0,90	CAGC	0,61
AAAT	0,66	CGGC	0,77
ATTT	0,66	CGCG	0,60
TTTT	0,90	CGCC	0,73
GGCG	0,73	CCGC	0,62
GCTG	0,61		
GCGG	0,62		
GCGC	0,73		
GCCG	0,78		

Das bedeutet aber auch, das Teilungsmuster ist von der Art des zu indexierenden Strings abhängig. Würde die Verteilung der Buchstaben des Input Textes gleichmäßig sein, dann würde A, T, G, C zum Beispiel ein geeignetes Muster sein. Die Teilbäume unter den Pfaden ab der Wurzel wären gleich groß. Oder AA, AT, AG, AC, TA, TT, TG, TC, GA, GT, GG, GC, CA, CT, CG, CC. Oder $\{AA, AT\}, \{AG, AC\}$, und so weiter. Doch wie oben evaluiert liegt eine Gleichverteilung nur in Ausnahmen vor. Man erkennt leicht das Suffixe mit den Präfixen A, AA, AAA usw. öfter auftreten, als Suffixe mit dem Präfixen C, CT, und so fort. Daher sind, die aus Schritt 1 resultierenden Suffix Trees, stark unbalanciert. Das Segmentierungsmuster muß nach der Beschaffenheit des Textes so gewählt werden, daß es die Unbalanciertheit der Bäume ausgleicht.

Der Parameter für den hier vorgestellten Algorithmus zum Finden einer optimalen Segmentierung, am Anfang von Kapitel 5, ist $p = 2$ und $\{M/2, M/2\}$ und die Wahl von

möglichst gleich großen Segmenten.

Beispiel eines Musters: ergibt eine *"fast"* optimale Segmentierung für eukaryontische DNA Sequenzen

$$M := \{AA, AT, AG, AC, TA, T, G, CA, CG, CT, CC, \epsilon\}.$$

Bei der Beschreibung von Schritt 2 des Pseudocodes wurde das simpelste Muster gewählt, und zwar die Buchstaben des Alphabetes: $Muster = \{A, T, G, C\}$. Das heißt, in diesem Fall erhält man für jeder Baum 4 Segmente.

5.3 Externes Mischen

Aus Schritt 1 erhalten wir k Bäume.

Aussehen dieser k Bäume

Sei $\forall \ell : 1 \leq \ell \leq k$ und $\forall i : (\ell - 1)m + 1 \leq i \leq \ell m$:

$$C_i^\ell := \max\{C_{ij}^\ell \in \Sigma^* : C_{ij}^\ell := LCP(s_i(t^+), s_j(t^+)),\ (\ell - 1)m + 1 \leq j \leq \ell m, j \neq i\}$$

die Menge der längsten gemeinsamen Präfixe (LCP) aus den Mengen $\mathcal{S}_\ell$ mit $1 \leq \ell \leq k$. Sei

$$C_i := \max\{C_{ij} \in \Sigma^* : C_{ij} := LCP(s_i(t^+), s_j(t^+)),\ 1 \leq j \leq n, i \neq j\}$$

die Menge der längsten gemeinsamen Präfixe von t^+. Hierbei bedeutet $LCP(s_i, s_j)$ – "Longest Common Prefix" der Suffixe s_i und s_j. Dann gilt

$$\forall\, \ell,\ 1 \leq \ell \leq k : \quad |C_i^\ell| \leq |C_i| \qquad \forall\, i \in \mathbb{N}.$$

In den k Bäumen sind jeweils die Mengen $\{C_i^1\}, \{C_i^2\}, \ldots, \{C_i^k\}$ gespeichert. Bei der Vereinigung dieser k Bäume muß aus den Mengen $\{C_i^1\}, \{C_i^2\}, \ldots, \{C_i^k\}$ die Menge $\{C_i\}$ berechnet werden. Das heißt, für jedes i mit $1 \leq i \leq n$ muß $Rest_i \in \Sigma^*$ ermittelt werden, so daß gilt:

$$C_i \;=\; C_i^\ell \cdot Rest_i.$$

Der fehlende Teil $Rest_i$ wird beim Mischen der k Bäume nachberechnet.

Verschiedene Möglichkeiten des Mischens unter der Annahme $|Rest_i| = 0$

<u>1. Fall:</u> Festes Segmentierungsschema gegeben

Sei mit $p \geq 2$ eine beliebige optimale Segmentierung $\{M/p, \ldots, M/p\}$ über jeden der k Bäume gegeben. Man erhält k Segmentierungsmengen:

$$\{U_{\omega_1}^1, \ldots, U_{\omega_p}^1\}, \ldots, \{U_{\omega_1}^k, \ldots, U_{\omega_p}^k\}.$$

Hierbei gibt es genau k zueinander äquivalente Segmente:

$$\begin{aligned} [\omega_1] &= \{U^1_{\omega_1}, \ldots, U^k_{\omega_1}\}, \\ \vdots & \qquad \vdots \\ [\omega_p] &= \{U^1_{\omega_p}, \ldots, U^k_{\omega_p}\}. \end{aligned}$$

Es sei $b \in [2, p]$ die Zahl der Bäume, dessen äquivalente Segmente gleichzeitig im Hauptspeicher gemischt werden.

Wenn man $|Rest_i| = 0$ für alle i annimmt, dann lassen sich "*bestmöglich*" $b = p$ Segmente nacheinander im Hauptspeicher mischen, da

$$\underbrace{M/p + \ldots + M/p}_{b\text{–mal}} = M.$$

Da jeder Baum aus p Segmenten besteht, benötigt man für b Bäume $p \cdot (b-1)$ *"innere"* Mischoperationen.

Bezeichnung: Es sei definiert

$$\begin{aligned} \lceil r \rceil &:= \min\{n \in \mathbb{N} : r \leq n\} \\ \lfloor r \rfloor &:= \max\{n \in \mathbb{N} : n \leq r\} \end{aligned}$$

In den Anwendungen werden große k betrachtet. Es gilt: $b \ll k$. Das heißt, es müssen in dieser ersten Stufe höchstens $\lceil \frac{k}{b} \rceil \cdot p(b-1)$ *"innere"* Mischoperationen durchgeführt werden. Man erhält also aus diesen Operationen $\lceil \frac{k}{b} \rceil \cdot p$ neue Bäume:

$$\begin{aligned} \mathcal{T}^1_{S,\omega_i} &= U^1_{\omega_i} \dot{\cup} \ldots \dot{\cup} U^b_{\omega_i} \\ \mathcal{T}^2_{S,\omega_i} &= U^{b+1}_{\omega_i} \dot{\cup} \ldots \dot{\cup} U^{2b}_{\omega_i} \\ \vdots & \qquad \vdots \\ \mathcal{T}^{\lceil \frac{k}{b} \rceil}_{S,\omega_i} &= U^{\left(\lceil \frac{k}{b} \rceil - 1\right)b+1}_{\omega_i} \dot{\cup} \ldots \dot{\cup} U^k_{\omega_i} \end{aligned}$$

mit

$$|\mathcal{T}^\nu_{S,\omega_i}| \leq M, \qquad \forall 1 \leq i \leq p, 1 \leq \nu \leq \left\lceil \frac{k}{b} \right\rceil.$$

Je kleiner b um so größer der Abstand $|M - |\mathcal{T}^\nu_{S,\omega_i}|| > 0$ und um so mehr Bäume existieren zu einem ω_i.

Es sei $k_1 := \lceil \frac{k}{b} \rceil$.

Für jedes feste ω_i läßt sich wiederholt die gegebene optimale Segmentierung $\{M/p, \ldots, M/p\}$ auf die dazugehörige Menge von Bäumen $\{\mathcal{T}^\nu_{S,\omega_i} : 1 \leq \nu \leq k_1\}$ anwenden. Und es sei wieder b die Anzahl der gleichzeitig im Hauptspeicher zu mischenden äquivalenten Segmente.

Es sind in Stufe 2 insgesamt $p \left\lceil \frac{k_1}{b} \right\rceil \cdot p(b-1)$ Operationen auszuführen.

Man erhält aus dieser Stufe $\left\lceil \frac{k_1}{b} \right\rceil \cdot p^2 = \left\lceil \frac{\lceil \frac{k}{b} \rceil}{b} \right\rceil \cdot p^2 = \lceil \frac{k}{b^2} \rceil \cdot p^2$ neue Bäume:

$$\begin{aligned}
\mathcal{T}^1_{S,\omega_i\omega_j} &= U^1_{\omega_j} \overset{\bullet}{\cup} \ldots \overset{\bullet}{\cup} U^b_{\omega_j} \\
\mathcal{T}^2_{S,\omega_i\omega_j} &= U^{b+1}_{\omega_j} \overset{\bullet}{\cup} \ldots \overset{\bullet}{\cup} U^{2b}_{\omega_j} \\
\vdots & \quad \vdots \\
\mathcal{T}^{\lceil \frac{k_1}{b} \rceil}_{S,\omega_i\omega_j} &= U^{\left(\lceil \frac{k_1}{b} \rceil - 1\right)b+1}_{\omega_j} \overset{\bullet}{\cup} \ldots \overset{\bullet}{\cup} U^{\lceil \frac{k}{b} \rceil}_{\omega_j}
\end{aligned}$$

Für jedes feste $\omega_i\omega_j$ gibt es genau $k_2 := \left\lceil \frac{k_1}{b} \right\rceil$ neue äquivalente Bäume $\left\{ \mathcal{T}^{\mu}_{S,\omega_i\omega_j} : 1 \leq \mu \leq k_2 \right\}$ mit $\left| \mathcal{T}^{\mu}_{S,\omega_i\omega_j} \right| \leq M$. In der nächsten Stufe gibt es $k_3 := \left\lceil \frac{k_2}{b} \right\rceil$ äquivalente Bäume.
Der Vorgang ist beendet, falls

$$k_n = 1$$

gilt. Das gilt für n, falls gilt

$$n = \lceil \log_b k \rceil .$$

Berechnung der Anzahl $A_{Mischen}$

Für die Anzahl der *"inneren"* Mischoperationen über alle Stufen gilt dann:

$$\begin{aligned}
|A_{Mischen}| &\leq p^0 \cdot \left\lceil \frac{k}{b} \right\rceil \cdot p\,(b-1) + p^1 \cdot \left\lceil \frac{k_1}{b} \right\rceil \cdot p\,(b-1) + \ldots + p^{n-1} \cdot \left\lceil \frac{k_{n-1}}{b} \right\rceil \cdot p\,(b-1) \\
&= p^0 \cdot \left\lceil \frac{k}{b} \right\rceil \cdot p\,(b-1) + p^1 \cdot \left\lceil \frac{k}{b^2} \right\rceil \cdot p\,(b-1) + \ldots + p^{n-1} \cdot \left\lceil \frac{k}{b^n} \right\rceil \cdot p\,(b-1) \\
&= p\,(b-1) \cdot \left(\sum_{i=0}^{\lceil \log_b k \rceil} p^i \left\lceil \frac{k}{b^{i+1}} \right\rceil \right) \qquad (5.2)
\end{aligned}$$

<u>2. Fall:</u> Variables Segmentierungsschema gegeben

Aus Schritt 1 des Algorithmus sind $k_0 := k$ Bäume gegeben.

<u>1. Stufe:</u> Sei für $p_1 \geq 2$ eine optimale Segmentierung $\{M/p_1, \ldots, M/p_1\}$ gegeben. Und sei $b_1 \in [2, \min(k_0, p_1)]$ die Anzahl der äquivalenten Segmente, welche gleichzeitig im Hauptspeicher gemischt werden. Dann werden höchstens $\left\lceil \frac{k_0}{b_1} \right\rceil \cdot p_1(b_1 - 1)$ Operationen durchgeführt. Und man erhält $p_1 \left\lceil \frac{k_0}{b_1} \right\rceil$ Bäume.

Es sei $k_1 := \left\lceil \frac{k_0}{b_1} \right\rceil$.

<u>2. Stufe:</u> Gegeben sei für $p_2 \geq 2$ eine Segmentierung $\{M/p_2, \ldots, M/p_2\}$ und $b_2 \in [2, p_2]$ mit der Funktion wie b_1 in Stufe 1. Dann sind maximal $p_1 \left\lceil \frac{k_0}{b_1} \right\rceil \cdot p_2(b_2 - 1)$ Operationen auszuführen.

$\vdots$

n. Stufe: Die Methode bricht ab, falls $k_n = 1$ eintritt.

Bei diesem Verfahren läßt sich die Größe der b_i an die Größe der k_i anpassen.

Berechnung der Anzahl $A_{Mischen}$

Die insgesamte Anzahl aller *"inneren"* Mischoperationen läßt sich folgendermaßen nach oben abschätzen:

$$\begin{aligned}
|A_{Mischen}| &\leq \left\lceil \frac{k_0}{b_1} \right\rceil \cdot p_1 (b_1 - 1) + p_1 \left\lceil \frac{k_1}{b_2} \right\rceil \cdot p_2 (b_2 - 1) + p_2 p_1 \left\lceil \frac{k_2}{b_3} \right\rceil \cdot p_3 (b_3 - 1) \\
&\quad + \ldots + p_{n-1} \cdots p_1 p_2 \cdot \left\lceil \frac{k_{n-1}}{b_n} \right\rceil \cdot p_n (b_n - 1) \\
&= \left\lceil \frac{k}{b_1} \right\rceil \cdot p_1 (b_1 - 1) + p_1 \left\lceil \frac{k}{b_1 b_2} \right\rceil \cdot p_2 (b_2 - 1) + p_2 p_1 \left\lceil \frac{k}{b_1 b_2 b_3} \right\rceil \cdot p_3 (b_3 - 1) \\
&\quad + \ldots + p_{n-1} \cdots p_1 p_2 \cdot \left\lceil \frac{k}{b_1 \cdots b_n} \right\rceil \cdot p_n (b_n - 1)
\end{aligned}$$

Als untere Schranke gilt:

$$\begin{aligned}
|A_{Mischen}| &\geq \left\lfloor \frac{k_0}{b_1} \right\rfloor \cdot p_1 (b_1 - 1) + p_1 \left\lfloor \frac{k_1}{b_2} \right\rfloor \cdot p_2 (b_2 - 1) + p_2 p_1 \left\lfloor \frac{k_2}{b_3} \right\rfloor \cdot p_3 (b_3 - 1) \\
&\quad + \ldots + p_{n-1} \cdots p_1 p_2 \cdot p_n (\min(k_{n-1}, b_n) - 1)
\end{aligned} \tag{5.3}$$

Spezialfall $p = b$:

Dieser Spezialfall ist in der Praxis nur dann relevant, wenn $|Rest_i| = 0$ gilt. In den in dieser Arbeit behandelten Anwendungen ist dieser Fall sehr unwahrscheinlich und stellt die "bestmögliche" Annahme dar, um die Laufzeitkomplexität zu verringern.

Vorausgesetzt wird eine sich wiederholende gleiche Segmentierung $\{M/p, \ldots, M/p\}$. Dann gilt mit (5.2)

$$|A_{Mischen}| \leq (p-1) \cdot \left(\sum_{i=0}^{\lceil \log_p k \rceil} p^{i+1} \left\lceil \frac{k}{p^{i+1}} \right\rceil \right).$$

Mit

$$k \leq p^{i+1} \left\lceil \frac{k}{p^{i+1}} \right\rceil \leq k + p^{i+1} \leq k + p^{\lceil \log_p k \rceil + 1} \leq k + (kp) \cdot p$$

folgt für die obere Schranke

$$\begin{aligned}
|A_{Mischen}| &\leq (p-1) \cdot \sum_{i=0}^{\lceil \log_p k \rceil} k(1+p^2) = k(p-1)(1+p^2) \frac{\left\lceil \log_p k \right\rceil}{2} \left(\left\lceil \log_p k \right\rceil + 1 \right) \\
&\leq \frac{1}{2} k (p-1)(1+p^2) \left(\frac{\ln k}{\ln p} + 2 \right)^2 =: OS(p,k).
\end{aligned}$$

Aus (5.3) gilt für die untere Schranke:

$$\begin{aligned}
|\mathrm{A}_{Mischen}| &\geq p(p-1)\cdot\sum_{i=0}^{\lceil\log_p k\rceil-1} p^i\left\lfloor\frac{k}{p^{i+1}}\right\rfloor + p^n\left(\min\left(k_{n-1},p\right)-1\right)\\
&= (p-1)\cdot\sum_{i=0}^{\lceil\log_p k\rceil-1} p^{i+1}\left\lfloor\frac{k}{p^{i+1}}\right\rfloor + p^n\left(\min\left(\left\lceil\frac{k}{p^{n-1}}\right\rceil,p\right)-1\right).
\end{aligned}$$

Mit

$$k \geq p^{i+1}\left\lfloor\frac{k}{p^{i+1}}\right\rfloor \geq k-p^{i+1}$$

folgt

$$\begin{aligned}
|\mathrm{A}_{Mischen}| &\geq (p-1)\cdot\left((k-p^{\lceil\log_p k\rceil})+(k-p^{\lceil\log_p k\rceil-1})+\ldots+(k-p)\right)\\
&\quad + p^n\left(\min\left(\left\lceil\frac{k}{p^{n-1}}\right\rceil,p\right)-1\right)\\
&= (p-1)\cdot\left(\left\lceil\log_p k\right\rceil k+1-\frac{p^{\lceil\log_p k\rceil+1}-1}{p-1}\right) + p^{\lceil\log_p k\rceil}\left(\min\left(\left\lceil\frac{k}{p^{n-1}}\right\rceil,p\right)-1\right)\\
&\geq (p-1)\cdot\left(\left\lceil\log_p k\right\rceil k+1-\frac{kp-1}{p-1}\right) + p^{\log_p k}(\min(\underbrace{\left\lceil\frac{k}{p^{n-1}}\right\rceil}_{\geq 2},p)-1)\\
&\geq (p-1)\cdot\left(\frac{\ln k}{\ln p}k+1-\frac{kp-1}{p-1}\right)+k =: US(p,k).
\end{aligned}$$

Für jedes feste, beliebige k gilt

$$\frac{\partial}{\partial p}OS(p,k)>0 \qquad \text{und} \qquad \frac{\partial}{\partial p}US(p,k)>0 \qquad \text{für } p\in[2,\infty).$$

Die Monotonie der oberen und unteren Schranke sagt noch nichts endgültig über die Monotonie der Funktion $|\mathrm{A}_{Mischen}|$ aus. Dazu muß man sich diese Funktion genauer ansehen.

Im Spezialfall $p=b$ gilt:

$$\begin{aligned}
F(p,k,n) := |\mathrm{A}_{Mischen}| &= \left\{\left\lfloor\frac{k}{p}\right\rfloor p(p-1)+p\max\left(0,gR(k,p)-1\right)\right\} +\\
&\quad \left\{\left\lfloor\frac{k_1}{p}\right\rfloor p^2(p-1)+p^2\max\left(0,gR(k_1,p)-1\right)\right\} + \ldots\\
&\quad + \left\{\left\lfloor\frac{k_{n-1}}{p}\right\rfloor p^n(p-1)+p^n\max\left(0,gR(k_{n-1},p)-1\right)\right\},
\end{aligned}$$

wobei wie oben $k_0 := k$, $k_1 := \left\lceil\frac{k}{p}\right\rceil$, $k_{i+1} := \left\lceil\frac{k_i}{p}\right\rceil$, $n = \left\lceil\log_p k\right\rceil$ bedeutet und gR der ganzzahlige Rest der Division zweier natürlicher Zahlen $gR(a,b) := a - b\lfloor a/b\rfloor$ ist.

In obiger Definition ist n aufgrund der Darstellung immer ≥ 1. Für $n = 0$ definieren wir $F(p, k, 0) = 0$.

Das Ziel ist es für ein beliebiges, festes k dasjenige p zu wählen, für welches die Anzahl der *"inneren"* Mischoperationen minimiert wird. Das heißt, gesucht ist $p^*(k)$ mit

$$p^*(k) = \min\{F(p, k) : p \in [2, \infty)\}.$$

Wir definieren eine Funktion $\widetilde{F}$, welche sehr nahe an F liegt:

$$\widetilde{F}(p, k) := k(p-1)\frac{\ln k}{\ln p}.$$

Es gilt, $\widetilde{F}$ ist für jedes feste $k > 1$ auf dem Intervall $p \in [2, \infty)$ monoton wachsend. Denn

$$\frac{\partial \widetilde{F}}{\partial p}(p, k) \;=\; k\frac{\ln k}{\ln p} \;-\; k(p-1)\frac{\ln k}{\ln^2 p}\frac{1}{p} \;=\; k\frac{\ln k}{\ln p}\underbrace{\left(1 \;-\; \frac{p-1}{p \ln p}\right)}_{> 0 \text{ für } p \in [2,\infty)}.$$

Das heißt, $\widetilde{F}$ nimmt das Minimum in $p = 2$ an.

Und es gilt: F liegt nahe an $\widetilde{F}$:
Sei $k, p \in \mathbb{N}$ beliebig, dann existieren eindeutig $a_\ell \in \mathbb{N}$, $0 \leq a_\ell < p$ und $0 \leq \ell \leq \lfloor \log_p k \rfloor$ mit

$$k \;=\; \sum_{\ell=0}^{\lfloor \log_p k \rfloor} a_\ell \, p^i.$$

Für k_i gilt dann

$$k_i \;=\; \left\lceil \frac{k_{i-1}}{p} \right\rceil \;=\; \left\lceil \frac{k}{p^i} \right\rceil \;=\; \sum_{\ell=0}^{n-i} a_{\ell+i} p^\ell \;+\; \begin{cases} 1 & , \text{ falls } \exists_{0 \leq j \leq i-1} : a_j \neq 0 \\ 0 & , \text{ falls } a_0 = a_1 = \ldots = a_{i-1} = 0 \end{cases}.$$

Dann gilt für jedes $i = 1, \ldots, n$:

$$
\begin{aligned}
&\left| \left\{ \left\lfloor \frac{k_{i-1}}{p} \right\rfloor (p-1) + \max(0, gR\left(k_{i-1}, p\right) - 1) \right\} \cdot p^i - k(p-1) \right| \\
&= \left| \left\{ \left(\sum_{\ell=0}^{n-i} a_{\ell+i} p^\ell + \left\{ \begin{array}{ll} 1 & , \text{ falls } \exists_{0 \le j \le i-2} : a_j \neq 0 \ \wedge \ p = a_{i-1} + 1 \\ 0 & , \text{ sonst} \end{array} \right\} \right) (p-1) \right. \\
&\quad \left. + \left\{ \begin{array}{cl} (a_{i-1}+1) - 1 & , \text{ falls } \exists_{0 \le j \le i-2} : a_j \neq 0 \ \wedge \ p \neq a_{i-1} + 1 \\ a_{i-1} - 1 & , \text{ falls } a_0 = a_1 = \ldots = a_{i-2} = 0,\ a_{i-1} \neq 0 \\ 0 & , \text{ sonst} \end{array} \right\} \right\} \cdot p^i - k(p-1) \Bigg| \\
&= \left| \left(\sum_{\ell=0}^{n-i} a_{\ell+i} p^{\ell+i} + \left\{ \begin{array}{ll} p^i & , \text{ falls } \exists_{0 \le j \le i-2} : a_j \neq 0 \ \wedge \ p = a_{i-1} + 1 \\ 0 & , \text{ sonst} \end{array} \right\} \right) (p-1) \right. \\
&\quad + \left\{ \begin{array}{cl} a_{i-1}\, p^i & , \text{ falls } \exists_{0 \le j \le i-2} : a_j \neq 0 \ \wedge \ p \neq a_{i-1} + 1 \\ (a_{i-1} - 1)\, p^i & , \text{ falls } a_0 = a_1 = \ldots = a_{i-2} = 0,\ a_{i-1} \neq 0 \\ 0 & , \text{ sonst} \end{array} \right\} \\
&\quad \left. - \left(\sum_{\ell=0}^{n-i} a_{\ell+i} p^{\ell+(i+1)} + \sum_{\ell=0}^{i-1} a_\ell p^{\ell+1} - \sum_{\ell=0}^{n-i} a_{\ell+i} p^{\ell+i} - \sum_{\ell=0}^{i-1} a_\ell p^\ell \right) \right| \\
&= \left\{ \begin{array}{cl} 0 & , \text{ falls } \ a_0 = a_1 = \ldots = a_{i-2} = 0,\ a_{i-1} = 0 \\ \left| a_{i-1} p^{i-1} - p^i \right| & , \text{ falls } \ a_0 = a_1 = \ldots = a_{i-2} = 0,\ a_{i-1} \neq 0 \\ \left| \left(p^{i+1} - \sum_{\ell=0}^{i-1} a_\ell p^{\ell+1} \right) - \left(p^i - \sum_{\ell=0}^{i-1} a_\ell p^\ell \right) \right| & , \text{ falls } \ \exists_{0 \le j \le i-2} : a_j \neq 0 \ \wedge \ p = a_{i-1} + 1 \\ \left| \sum_{\ell=0}^{i-2} a_\ell p^{\ell+1} - \sum_{\ell=0}^{i-1} a_\ell p^\ell \right| & , \text{ falls } \ \exists_{0 \le j \le i-2} : a_j \neq 0 \ \wedge \ p \neq a_{i-1} + 1 \end{array} \right.
\end{aligned}
$$

und

$$
\sum_{i=1}^{\lceil \log_p k \rceil} \left(\frac{k}{p_i} (p-1) \right) p_i = k(p-1) \left\lceil \log_p k \right\rceil \approx k(p-1) \log_p k \,.
$$

Hiermit läßt sich folgender Satz zeigen:

Satz 11 :
Sei $k \in I\!N \setminus \{9\}$ beliebig. Dann ist $F(p,k)$ eine in $p \in [2,k]$ monoton wachsende Funktion. Das heißt, es gilt: $p^*(k) = 2$.

Beweis :
Für die Betrachtung zum Minimum siehe oben. Für $k = 9$ sieht man leicht, daß $p^*(9) = 3 > 2$ ist. Denn es gilt $F(3,9) = 36 < 40 = F(2,9)$. Der Rest des Beweises folgt aus obiger Betrachtung.
Desweiteren nimmt $F(p,k)$ für $p = k$ sein Maximum an. □

BEMERKUNG 1 :

1. Die Größe von k wird hauptsächlich durch das Verhältnis von Hauptspeichergröße zur Textlänge bestimmt.

2. Der obige Satz 11 gilt für $p \in [2, \infty)$. Es läßt sich jeder Baum in beliebig viele Segmente zerlegen und $F(p,k)$ wächst in diesem Fall weiter an.

3. **Zerteilung in $p = k$ Segmente**
   Aus obigem Satz 11 wird klar, daß dies eine der ungünstigsten Strategien zur Segmentierung ist, auch wenn man nur einen Segmentierungslevel $n = \log_p k = 1$ durchlaufen muß. Der gesamte Konstruktionsalgorithmus besteht dann aus genau zwei Schritten:

   Erster Schritt: Finden einer optimalen Segmentierung $\{M/k, \ldots, M/k\}$.
   Zweiter Schritt: Durchführen von $k \cdot (k-1)$ Mischvorgänge

4. Eine sich gleichmäßig wiederholende Zerteilung in p Segmente ist nicht immer die optimale Strategie, da p besonders in den letzten Stufen der Methode ein Vielfaches der Anzahl äquivalenter Bäume sein müßte, um die auftretenden großen Potenzen von p so klein wie möglich zu halten.

Der nächste Satz 12 zeigt, daß sich die letzten Stufen, die "Endstufen" der Methode vermeiden lassen. Es lassen sich nun für jedes $\nu < n$, $F(p,k,\nu)$ Schritte ausführen. Dann erhält man p^ν Mengen äquivalenter Bäume, jede mit einer maximalen Mächtigkeit von k_ν. Für jede dieser Mengen kann durch $p_1 = k_\nu$ ein neues Segmentierungsschema gewählt werden. Es sind für jede Menge $F(p_1, k_\nu, \left\lceil \log_{p_1} k_\nu \right\rceil)$ weitere Operationen auszuführen. Wie in Bermerkung 1.2 ergeben sich $k_\nu(k_\nu - 1)$ Schritte. Und man erhält insgesamt – für p^ν Mengen: $k_\nu(k_\nu - 1) \cdot p^\nu$ Operationen. Diesen letzten Schritt nennen wir "*Endmischen*".

Der folgende Satz gibt an, daß ein solches ν in jedem Fall existiert, auch wenn es nicht in jedem Fall zu einer Minimierung der Gesamtzahl aller Operationen führt.

Satz 12 :
Es sei k mit $k \geq 2$ und p mit $2 \leq p \leq k$ fest gegeben. Dann existiert ein ν mit $0 \leq \nu < \left\lceil \log_p k \right\rceil$, so daß

$$F(p,k,\nu) \; + \; k_\nu(k_\nu - 1) \cdot p^\nu \; \leq \; F(p,k, \left\lceil \log_p k \right\rceil) \tag{5.4}$$

gilt.

ZUSATZ: Für $p = 2$ gilt für jedes beliebige k Gleichheit.

Beweis :
Für $k = p$ ist die Aussage für $\nu = 0$ erfüllt. Sei $k > p$. Wähle ν, so daß gilt:

$$\left\lfloor \frac{k_{\nu-1}}{p} \right\rfloor \; \leq \; p+1 \; < \; \left\lfloor \frac{k_{\nu-2}}{p} \right\rfloor .$$

Das heißt, es muß gelten: $\left\lceil \frac{k}{p^\nu} \right\rceil \leq p+1 < \left\lceil \frac{k}{p^{\nu-1}} \right\rceil$. Das wird erfüllt von:

$$\nu := \left\lceil \log_p \frac{k}{p+1} \right\rceil .$$

Wegen der Monotonie des Logarithmus ist $0 \leq \nu < \left\lceil \log_p k \right\rceil$ erfüllt, und k_ν kann nur drei verschiedene Werte annehmen: entweder $k_\nu = p+1$ oder $k_\nu = p$ oder $k_\nu < p$.

Es ist nach Definition

$F(p,k,\left\lceil \log_p k \right\rceil) - F(p,k,\nu)$

$$= \left\{ \left\lfloor \frac{k_\nu}{p} \right\rfloor p^{\nu+1}(p-1) + p^{\nu+1} \max\left(0, gR(k_\nu,p)-1\right) \right\} + \ldots$$
$$\ldots + \left\{ \left\lfloor \frac{k_{n-1}}{p} \right\rfloor p^{n}(p-1) + p^{n} \max\left(0, gR(k_{n-1},p)-1\right) \right\}$$

$$= \begin{cases} 1\cdot p^{\nu+1}(p-1) + p^{\nu+2} & = & (2p-1)\,p^{\nu+1} & \text{für} & k_\nu = p+1 \\ 1\cdot p^{\nu+1}(p-1) + 0 & = & (p-1)\,p^{\nu+1} & \text{für} & k_\nu = p \\ 0\cdot p^{\nu+1}(p-1) + p^{\nu+1}(k_\nu-1) & = & p^{\nu+1}\,(k_\nu-1) & \text{für} & k_\nu < p \end{cases}$$

$$\begin{cases} \geq & (p+1)\,p^{\nu+1} & \text{für} & k_\nu = p+1 \\ = & (p-1)\,p^{\nu+1} & \text{für} & k_\nu = p \\ > & k_\nu(k_\nu-1)\,p^{\nu} & \text{für} & k_\nu < p \end{cases}$$

$$= k_\nu(k_\nu-1)\,p^\nu .$$

□

Ein solches ν existiert in jedem Fall, ist aber nicht für jedes gewählte k und p gleich bedeutend. Aus obigem Beweis geht hervor, daß nur in zwei Fällen auch tatsächlich Ungleichheit im Sinne von "$<$" in Gleichung (5.4) besteht:

1. $\left\lceil \log_p \frac{k}{p+1} \right\rceil + 2 = \left\lceil \log_p k \right\rceil$ und $p \neq 2$
2. $\left\lceil \log_p \frac{k}{p+1} \right\rceil + 1 = \left\lceil \log_p k \right\rceil$ und $\left\lceil \frac{k}{p^\nu} \right\rceil \neq p$ mit $\nu := \left\lceil \log_p \frac{k}{p+1} \right\rceil$.

Nur in diesen Fällen wird eine Verringerung der Laufzeitkomplexität erreicht. Bei festem k ist die Möglichkeit einer Verringerung umso effizienter je größer p.

Beispiele:
für welche das zutrifft mit relevanten Größenordnungen für k in der Bioinformatik

Bedingung 2 ist erfüllt für	$k=1227, p=3,$	$k=763, p=5$
	$k=470, p=4,$	$k=3476, p=9$;
	$\ldots$	
Bedingung 1 ist erfüllt für	$k=731, p=3,$	$k=631, p=5$
	$k=17, p=4,$	$k=6572, p=9$
	$k=2433, p=4,$	$\ldots$.

BEMERKUNG 2:
Die kleinste realisierbare Segmentierung, wie in dieser Arbeit definiert, ist $p = 4$. Diese anzuwenden ist auch deshalb am sinnvollsten, da in der Praxis $|Rest_i| \neq 0$ gilt.

Mit Satz 12 ist gezeigt, daß Schritt 3 des Algorithmus aus Kapitel 5, auch für den betrachteten Spezialfall $|Rest_i| = 0$, sinnvoll ist. In diesem Fall ist ν vorhanden, aber mit entweder $\nu = n - 2$ oder $\nu = n - 1$ noch nicht weit von n entfernt.

Spezialfall $p > b$:

Wie oben erläutert ist dieser Fall für die Anwendungen relevant. Es liegt folgende Entwicklung vor:

$$|Rest_i^0| > |Rest_i^1| > \cdots > |Rest_i^{n_1}| = 0\,,$$

wobei ein $n_1 \leq \lceil \log_b k \rceil$ existiert.

Es ist jedoch a-priori nicht bekannt, wie groß dieses n_1 ist. Für die Größe von $|Rest_i|$ gilt: $|Rest_i| = O(|p - b|)$. Die Anzahl aller Mischvorgänge ergibt sich, mit $n = \lceil \log_b k \rceil$:

$$F(p, b, k, n) \;=\; \sum_{i=1}^{n} \left\{ \left\lfloor \frac{k_{i-1}}{b} \right\rfloor (b-1) + \max(0, gR(k_{i-1}, b) - 1) \right\} p^i\,,$$

wobei $k_i = \left\lceil \frac{k}{b^i} \right\rceil$ gilt. Wie oben gezeigt, gilt $F(p, k, \lceil \log_p k \rceil) < F(p, b, k, \lceil \log_b k \rceil)$ für alle $p > b, k$. Für eine realistische Betrachtung $p \geq 4$ wird das Minimum in $b = 2, p = 4$ angenommen.

Ebenfalls gilt:

$$F(p, b, k, \nu) \;+\; k_\nu(k_\nu - 1) \cdot p^\nu \;\leq\; F(p, b, k, \lceil \log_b k \rceil)$$

für ein $\nu \in [0, \lceil \log_b k \rceil)$ und $k_\nu = \left\lceil \frac{k}{b^\nu} \right\rceil$.

Die Existenz dieses k_ν begründet, warum Schritt 3 im Algorithmus sinnvoll ist, trotzdem die beschriebene Methode eigentlich ein natürliches Ende hat. Ist insbesondere $b \ll p$, dann kann theoretisch dieses $\nu = 0$ werden. Dann würde jedoch gelten, daß $Rest_i^\nu \neq 0$ ist. Eine Aufteilung, der zu verarbeitenden Bäume in k_ν gleichgroße Teile würde zu einem Überlauf des Hauptspeichers führen. Es müßte ein $k^* > k_\nu$ in Abhängigkeit der Größe von $Rest_i^\nu$ gewählt werden, so daß $\left| \bigcup_{i=1}^{k^*} U_i \right| \leq M$ gilt. Ist man sich über dieses k^* nicht sicher, so ist es sinnvoll $\nu = n_1$ zu wählen, da ab diesem Stadium $Rest_i^\nu = 0$ gilt. Das "Endmischen" läßt sich dann ohne Probleme durchführen. Der Fall $b = p$ wurde im vorigen Abschnitt gezeigt. Das heißt: Je größer $|p - b|$, um so kleiner kann der Wert ν gewählt werden.

Serielle Komplexität

Die Anzahl der externen Mischvorgänge ist ein Maß für die Komplexität der seriellen Festplattenzugriffe. Die Anzahl der seriellen Diskzugriffe im schlechtestem Fall ergibt sich mit:

$$S\left(\sum_{i=1}^{n} \left\{ \left\lfloor \frac{k_{i-1}}{b} \right\rfloor (b+1) + \max(0, gR(k_{i-1}, b) + 1) \right\} p^i \right)\,.$$

Diese lassen sich, wie oben beschrieben, zu

$$S\left(\sum_{i=1}^{\nu}\left\{\left\lfloor\frac{k_{i-1}}{b}\right\rfloor(b+1)+\max(0,gR(k_{i-1},b)+1)\right\}p^i\ +\ k_\nu(k_\nu-1)\cdot p^\nu\right) \tag{5.5}$$

reduzieren. Die wesentliche Reduktion der Laufzeit, wie sie die Experimente gegenüber anderen Methoden verzeichnen, beruht auf der Ausnutzung dieser Tatsache.

5.3.1 Beschreibung des tatsächlichen Algorithmus

In Schritt 2 betrachten wir immer Paare von Suffix Trees. Das heißt, b wurde 2 gesetzt. Wir haben $\lceil k/2\rceil$ Paare. Dann wird jeder Baum dieses Paares mit $k=4$ segmentiert und anschließend die zueinander äquivalenten Teilsegmente gemischt. Dazu laden wir diese von der Festplatte in den Hauptspeicher und mischen sie dort. Hierbei werden immer ab der Wurzel gleiche Kanten miteinander vereinigt. Als Ergebnis erhält man einen Tree, welcher die Gesamtheit der Blätter der beiden Eingabe Bäume enthält. Beim Benutzen des einfachsten Segmentierungsmusters, wie im Pseudocode beschrieben, erhält man als Ergebnis dieser ersten Iteration von jedem der $\lceil k/2\rceil$ Paare (oder einzelner Restbäume) 4 Output Trees, $\mathcal{T}_{A,j}, \mathcal{T}_{T,j}, \mathcal{T}_{G,j}, \mathcal{T}_{C,j}$, entstanden aus dem Muster $\{A,T,G,C\}$. (Bezeichnung: $\mathcal{T}_{A,j}$ bezeichnet den j-ten Baum abgeleitet nach dem Musterstring A.) Das heißt, nach der ersten Iterationsstufe erhält man $4\cdot\lceil k/2\rceil$ Indexfiles. Nach dem 2. Loop sind es $4^2\cdot\lceil k/2^2\rceil$ resultierende Bäume, nämlich $\mathcal{T}_{AA,j}, \mathcal{T}_{AT,j}, \mathcal{T}_{AG,j}$, $\mathcal{T}_{AC,j}, \mathcal{T}_{TA,j}, \mathcal{T}_{TT,j}, \mathcal{T}_{TG,j}, \ldots$. Nach der i-ten Iteration sind es $4^i\cdot\lceil k/2^i\rceil$. Es wurde gleichbleibend das einfachste Muster zur Segmentierung gewählt. Daher wächst bei diesem iterativen Prozeß die Anzahl der Indexfiles, während deren Größe abnimmt. In der Anwendung stoppt der Zyklus, falls die totale Summe der Teilbäume mit gleichem Präfix auf einmal in den Hauptspeicher geladen werden kann.

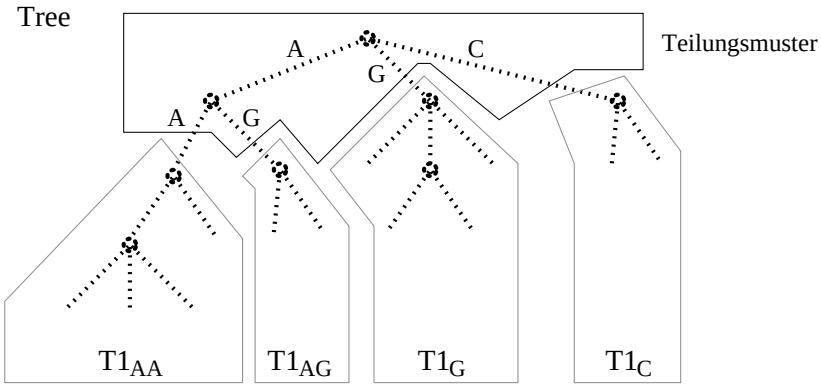


Abbildung 5.5: Segmentierung eines Suffix Trees

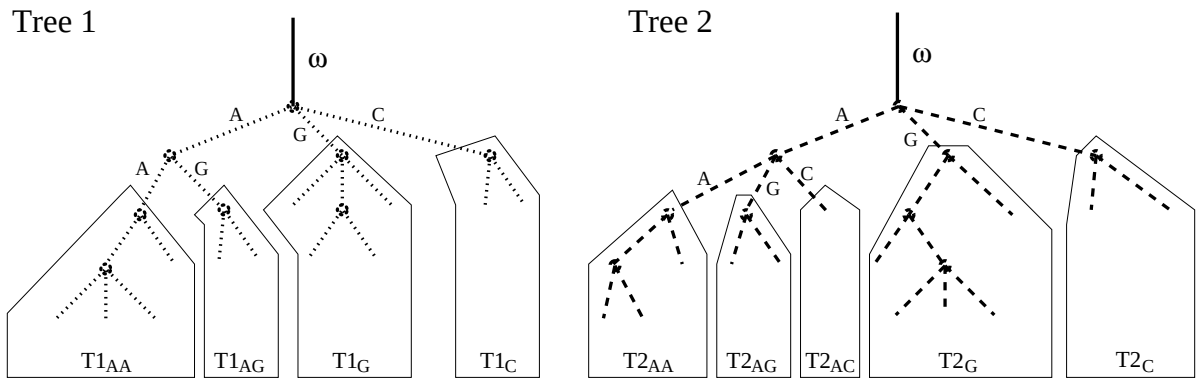


Abbildung 5.6: Segmentierung zweier Suffix Trees in Paare zueinander äquivalenter Teilbäume

Vorteile des Algorithmus: Es ist möglich, die zu mischenden Teilbäume so zu wählen, daß der Ungleichverteilung der Buchstaben bei DNA Sequenzen, und damit der Unbalanciertheit des gesamtem Suffix Trees, besser entgegengekommen wird. Das Muster ist nach der Beschaffenheit der Daten Sequenz auszuwählen.

5.4 Inneres Mischen

Beschreibung der inneren Prozedur $MergeTree(z, \mathcal{T}_{\omega,j}(t), \mathcal{T}_{\omega,j+1}(t))$:

Wenn zwei Teilsegmente in den Hauptspeicher geladen wurden, werden sie dort gemischt. Das heißt, beide Bäume werden simultan von links nach rechts in Depth-First Weise durchlaufen, und die nicht im Baum 1 vorhandenen Knoten und Pfade des Baumes 2 werden in den ersten Baum hineingemischt. Beim Durchlaufen des zweiten Baumes werden sukzessiv alle nicht mehr benötigten Knoten, welche nicht in Baum 1 verschoben wurden, gelöscht. Das bedeutet, daß während des gesamten Vorgangs die Belegung des Hauptspeichers ungefähr konstant ist.

Ein Suffix Tree ist eigentlich ein Tree mit komprimierten Pfaden, das heißt ein Pfad ist nicht nur mit einem Buchstaben beschriftet, sondern mit einem String von variabler Länge. Als erstes wird ein solcher Tree in einen Trie übersetzt, und zwar wird aus jeder Kante deren Beschriftung die Länge d hat, $d-1$ Knoten und $d-1$ Kanten erzeugt, die jeweils immer nur mit einem Buchstaben gelabelt sind.

In dieser Darstellung werden beide Bäume in einem Durchlauf miteinander verglichen. Es können folgende Fälle auftreten:

1. Es existiert zu einem Knoten ein Pfad in Baum 2, aber nicht in Baum 1. Dann wird der Pfad mit dem gesamten darunterliegenen Teilbaum aus Baum 2 in Baum 1 verschoben.

2. Äquivalent dazu der umgekehrte Fall: Es existiert zu einem Knoten ein Pfad in Baum 1, aber nicht in Baum 2. In diesem Fall braucht man nichts zu tun, da Baum

1 am Ende auch der resultierende “Merged Tree” ist.

3. Man erreicht in Baum 2 ein Blatt und befindet sich in Baum 1 aber immer noch an einem inneren Knoten. Mit dem Label des Blattes springt man in die zu indexierende Sequenz und traversiert mit dem erhalten String solange den Baum 1 hinab, bis man zu einem Missmatch Knoten kommt. Es wird die entsprechend fehlende Kante an diesen Knoten angefügt und das Blatt, beschriftet mit dem betrachteten Suffix, angehängt.

4. Der umgekehrte Fall zu 3: Man erreicht in Baum 1 ein Blatt und befindet sich in Baum 2 aber immer noch an einem inneren Knoten. Dann verschiebt man den Knoten aus Baum 2 mitsamt darunterliegenden Teilbaum an die Stelle des Blattes in Baum 1. Der zum Blatt gehörige Suffix wird aus dem Eingabe Text geholt. Damit traversiert man Baum 1 solange hinab, bis man einen Missmatch Knoten erreicht. Dort wird die Suffixposition als Blatt eingetragen.

5. Man erreicht in Baum 1 und Baum 2 jeweils einen Blattknoten. Dann vergleicht man die dazugehörigen Suffixe bis zum Missmatch und fügt diese Kante in Baum 1 ein.

Analyse der inneren Merge Prozedur:

Deterministische Analyse der inneren Merge Prozedur:

Am Start des Schrittes 2 hat jeder der k Bäume n/k Suffixe. Das heißt, n/k Blätter. Wie im Kapitel 5.1.3 ausgeführt, kann die maximale Pfadlänge $n/k + c$ sein.

Die Aufgabe der Segmentierung in den folgenden Schritten ist es, die jeweils zu mischenden Segmente etwa gleich groß zu halten. Das bedeutet nicht, daß auch die Anzahl der darin vertretenden Suffixe jeweils ungefähr die gleiche Größe haben. Sondern nur, daß bei einer Übersetzung des Segmentes in einen Trie, die vorhandenen Tries ungefähr die gleiche Knotenanzahl besitzen.

Die innere Merge Prozedur hat im ’worst case’ eine Zeitkomplexität von $O((n/k + c)^2)$. Denn jeder der Tries hat höchstens $O((n/k + c)^2)$ Knoten. Da an jedem Knoten genau eine Vergleichsoperation durchgeführt wird, ergibt sich die Komplexität.

Zur Angabe von durchschnittlichen Werten beim ’inneren Mischen’ führen wir im nächsten Kapitel ein stochastisches Modell für DNA Sequenzen ein.

5.5 Stochastische Analyse

Stochastische Analyse der inneren Merge Prozedur

Für ein allgemeines stochastisches Modell über Tries sollte man verschiedende Kriterien für eine gute Annahme einbeziehen:

1. besondere Merkmale des Alphabetes, das bedeutet die Verteilung der einzelnen Buchstaben des Alphabetes in den betrachteten Wörtern,
2. die statistische Abhängigkeit der Wörter untereinander,
3. die Anzahl der betrachteten Wörter.

Der zweite Punkt ist für die hier eingeführte Datenstruktur einfach zu beantworten. Wie in Kapitel 2 beschrieben ist ein Suffix Tree ein Pfad-komprimierter Trie, welcher aus den Teilwörtern – den Suffixen – eines gegebenen Strings gebildet wird. Würde man einen allgemeinen Trie, ob kompakt oder nicht, über einer Menge von völlig willkürlich gewählten Wörtern betrachten, so wären diese statistisch unabhängig. Die hier betrachtete Menge an Wörtern $\mathcal{S}(t)$ stammt von einem gegebenen Text ab und kann deshalb dagegen nicht statistisch unabhängig voneinander sein. Das bedeutet, daß das Wissen eines Suffix einem Informationen über die mögliche Beschaffenheit eines anderen Suffixes gibt.

Wir betrachten, wie in der Einleitung Kapitel 1 ausgeführt, große Datenmengen. Der dritte Punkt gilt für großes, aber festes n.

Für eine stochastische Analyse sind spezielle Annahmen gerade für DNA Sequenzen zu wählen. Damit die Aussagen der Analyse anwendbar sind, ist man einerseits bestrebt die Annahmen so realistisch wie möglich zu wählen, und andererseits eine Analyse noch unter diesen Annahmen möglich zu lassen. In unterschiedlicher Literatur werden Modelle beschrieben bei denen davon ausgegangen wird, daß eine Gleichverteilung der Aminosäuren (der Basen) in der DNA Kette vorliegt. Wie statistische Untersuchungen an realen Genomen belegen, ist dies in keiner Weise der Fall – siehe hierzu die Tabellen 5.1, 5.2 und 5.3 aus Kapitel 5.2.4.

Die konkreten Modellannahmen nach den Kriterien 1 bis 3 lauten:

Sei C_{ij} die Länge des längsten Präfixes der Suffixe s_i und s_j mit $i \neq j$. Und $C_d : \{1, \ldots, n\} \rightarrow \mathbb{N}_0$ sei eine Folge von Zufallsvariablen mit $C_d := C_{i,i+d}$.

Modellannahmen:

1. $x_i \in \Sigma$ tritt mit der Wahrscheinlichkeit p_i auf;
2. die Wahrscheinlichkeit des Auftretens des Buchstaben x_i an der k-ten Stelle der

Sequenz ist bedingt durch die $(k-1)$-te Stelle, das heißt, es existieren Übergangswahrscheinlichkeiten $p_{ji} = Pr(T_k = x_i | T_{k-1} = x_j)$.

Hierbei ist T_k der stochastische Prozeß, welcher den gegebenen Text t modelliert.

Gerade die letztere Annahme ist für DNA Sequenzen wesentlich wahrscheinlicher, da jeweils Triplets von Aminosäuren ein Eiweiß kodieren. Das bedeutet, daß bestimmte Triplet-Konstellationen wahrscheinlicher sind als andere Triplet-Zusammensetzungen. Desweiteren gibt es markante Sequenzabschnitte in DNA Sequenzen, genannt Motife, die gewisse Funktionen erfüllen, wie das Startcodon von Genen : AGT, oder der PolyA-Teil : $AAA \ldots A$. Das heißt, daß zum Beispiel einem A wieder ein A folgt ist wesentlich wahrscheinlicher als das ein T folgt.

Die Bernoulli Annahme besagt, daß die Buchstaben einer Sequenz voneinander unabhängig sind. Das zum Beispiel die Buchstaben der Sequenz AA voneinander unabhängig sind, wird hier nicht vorausgesetzt. Es wird ein Markov Modell vorausgesetzt.

1. Nicht das Bernoulli-Modell, auch nicht das unabhängige Modell beim Suffix Tree, da die Menge der Suffixe eine Menge von Strings ist, welche voneinander abhängt;

2. Bedingung stationäres Problem: T_k stationär $\Rightarrow Pr(C_{ij})$ hängt nur von d, $d = |j - i|$ ab;

3. Markov Modell vorausgesetzt.

Es gilt folgende bekannte Tatsache aus der Kombinatorik:

Hilfssatz : *Seien s_i, s_j zwei beliebige Suffixe des Textes t^+, mit $i < j$ und $j - i = d$. Sei weiter Z der längste gemeinsame Präfix von s_i und s_j mit $|Z| = C_{ij} = k \geq 0$. Dann kann Z dargestellt werden durch $Z = V^\ell V'$, wobei $|V| = d, |V'| = r < d$, V' ist Präfix von V, und V^ℓ ist definiert als ℓ-fache Konkatenation von V.*

Der Beweis findet sich in [Lot82].

Aus obigem Hilfssatz gewinnt man eine einfache Folgerung:

Folgerung : Sei $k \leq d$, so gilt:

$$Pr(C_d \geq k) = \sum_{\mathcal{T}_k} Pr(V U' V). \qquad (*)$$

Sei $d \leq k$, so gilt:

$$Pr(C_d \geq k) = \sum_{\mathcal{T}_d} Pr(V^{\lfloor \frac{k}{d} \rfloor + 1} V'). \qquad (**)$$

Hierbei wurde $\mathcal{T}_k := \{t \in \Sigma^* : |t| = k\}$ definiert. Und mit U', V' wurden die jeweiligen Zwischen- und Endstücke notiert.

Betrachte H_n die Höhe eines Suffix Baumes:

$$H_n := \max_{1 \le i < j \le n} \{C_{ij}\} + 1$$

Satz 13 : *Es gilt:*

$$E\,H_n \le a_n + C,$$

wobei $a_n \approx 2/(\log P^{-1}) \cdot \log n$ *mit einer Konstante* C *und* $P = \sum_{i=1}^{|\Sigma|} p_i$.

Beweis: Aus der Monotonie des Erwartungswertes folgt:

$$\begin{aligned} E\,H_n = E\left(\max_{1\le d\le n} C_d\right) &\le E\left(a + \sum_{1\le d\le n} \max(C_d - a, 0)\right) \qquad \text{wobei } a \in \mathbb{N}_0, a > 0 \text{ beliebig.} \\ &= aE(1) + \sum_{1\le d\le n} \sum_{x\in\mathbb{N}} \max(x-a,0)\cdot Pr\,(C_d = x) \\ &= a + \sum_{x=a+1}^{n} (x-a) \sum_{1\le d\le n} Pr\,(C_d = x) \\ &= a + \sum_{x=0}^{n-a} x \sum_{1\le d\le n} Pr\,(C_d = a+x), \end{aligned}$$

weil die Ereignismengen $\{C_d = a + x\}$ disjunkt für $k = 0, \ldots, n-1$ sind, gilt:

$$\le a + \sum_{1\le d\le n} n \cdot Pr\,(C_d \ge a).$$

Für jedes beliebige $n \in \mathbb{N}$ ist nun ein a_n gesucht, so daß der zweite Summand endlich wird. Das heißt, es muss gelten:

$$\sum_{1\le d\le n} n \cdot Pr\,(C_d \ge a_n) \;=\; C \qquad \forall n \in \mathbb{N}.$$

Und das heißt, es muss gelten:

$$\underbrace{\sum_{d=1}^{a_n} n \cdot Pr\,(C_d \ge a_n)}_{(1)} + \underbrace{\sum_{d=a_n+1}^{n} n \cdot Pr\,(C_d \ge a_n)}_{(2)} \;=\; C \qquad \forall n \in \mathbb{N}.$$

Es gilt für (2): Aus obiger Folgerung mit Gleichung $(*)$ ist

$$Pr\,(C_d \ge a_n) = \sum_{\mathcal{T}_{a_n}} Pr\,(V\,U'\,V), \qquad \text{wobei} \quad \begin{matrix} V = V_1 V_2 \ldots V_{a_n} \\ U' = U'_1 \ldots U'_{d-a_n} \end{matrix}$$

$$\begin{aligned} \Longrightarrow (2) &= n \sum_{d=a_n+1}^{n} \sum_{\mathcal{T}_{a_n}} Pr\,(V\,U'\,V) \\ &= n \sum_{d=a_n+1}^{n} \sum_{\mathcal{T}_{a_n}} Pr\,(V_1 V_2 \ldots V_{a_n} \cdot U'_1 \ldots U'_{d-a_n} \cdot V_1 V_2 \ldots V_{a_n}) \end{aligned}$$

Aus dem Markov Modell folgt:

$$
\begin{aligned}
&= n \sum_{d=a_n+1}^{n} \sum_{\mathcal{T}_{a_n}} Pr(V_1)Pr(V_2|V_1)\dots Pr(V_{a_n}|V_{a_n-1}) \cdot Pr(U'_1|V_{a_n})\dots Pr(V_1|U'_{d-a_n}) \cdot \\
&\qquad \cdot Pr(V_2|V_1)\dots Pr(V_{a_n}|V_{a_n-1}) \\
&\leq n \sum_{d=a_n+1}^{n} \sum_{\mathcal{T}_{a_n}} Pr(V_1) \cdot \underbrace{Pr(V_1|U'_{d-a_n})}_{\leq Pr(V_1)} \cdot Pr^2(V_2|V_1)\cdots Pr^2(V_{a_n}|V_{a_n-1}) \\
&\leq n \sum_{d=a_n+1}^{n} \sum_{\mathcal{T}_{a_n}} \underbrace{Pr^2(V_1) \cdot Pr^2(V_2|V_1)\cdots Pr^2(V_{a_n}|V_{a_n-1})}_{= Pr^2(V)} \\
&\leq n \cdot n \sum_{\mathcal{T}_{a_n}} Pr^2(V) = n^2 \cdot \sum_{\substack{\{A\cdots A, A\cdots T, \dots\} \\ |\mathcal{T}_{a_n}| = |\Sigma|^{a_n} = 4^{a_n}}} Pr^2(V)
\end{aligned}
$$

$$
\begin{aligned}
= n^2 \cdot (& Pr^2(V_1 = A) \{ \underbrace{p_{AA}^2 p_{AA}^2 \cdots p_{AA}^2}_{(a_n-1)\text{- mal}} + \underbrace{p_{AA}^2 p_{AA}^2 \cdots p_{AT}^2}_{(a_n-1)\text{- mal}} + \dots + \underbrace{p_{AC}^2 p_{CC}^2 \cdots p_{CC}^2}_{(a_n-1)\text{- mal}} \} \\
& + Pr^2(V_1 = T) \{ \underbrace{p_{TA}^2 p_{AA}^2 \cdots p_{AA}^2}_{(a_n-1)\text{- mal}} + \dots + p_{TC}^2 p_{CC}^2 \cdots p_{CC}^2 \} \\
& + Pr^2(V_1 = G) \{ p_{GA}^2 p_{AA}^2 \cdots p_{AA}^2 + \dots + p_{GC}^2 p_{CC}^2 \cdots p_{CC}^2 \} \\
& + Pr^2(V_1 = C) \{ p_{CA}^2 p_{AA}^2 \cdots p_{AA}^2 + \dots + p_{CC}^2 p_{CC}^2 \cdots p_{CC}^2 \})
\end{aligned}
$$

$$
= n^2 \cdot \begin{pmatrix} Pr(V_1 = A) \\ Pr(V_1 = T) \\ Pr(V_1 = G) \\ Pr(V_1 = C) \end{pmatrix}^T \cdot \begin{pmatrix} p_{AA}^2 & p_{AT}^2 & p_{AG}^2 & p_{AC}^2 \\ p_{TA}^2 & p_{TT}^2 & p_{TG}^2 & p_{TC}^2 \\ p_{GA}^2 & p_{GT}^2 & p_{GG}^2 & p_{GC}^2 \\ p_{CA}^2 & p_{CT}^2 & p_{CG}^2 & p_{CC}^2 \end{pmatrix}^{a_n-1} \cdot \begin{pmatrix} Pr(V_1 = A) \\ Pr(V_1 = T) \\ Pr(V_1 = G) \\ Pr(V_1 = C) \end{pmatrix}
$$

$$
\begin{aligned}
&\leq n^2 \cdot \underbrace{\left\| \begin{pmatrix} Pr(V_1 = A) \\ Pr(V_1 = T) \\ Pr(V_1 = G) \\ Pr(V_1 = C) \end{pmatrix}^T \right\|_{\max}}_{\leq 1} \cdot \left\| \begin{pmatrix} p_{AA}^2 & p_{AT}^2 & p_{AG}^2 & p_{AC}^2 \\ p_{TA}^2 & p_{TT}^2 & p_{TG}^2 & p_{TC}^2 \\ p_{GA}^2 & p_{GT}^2 & p_{GG}^2 & p_{GC}^2 \\ p_{CA}^2 & p_{CT}^2 & p_{CG}^2 & p_{CC}^2 \end{pmatrix}^{a_n-1} \right\| \cdot \underbrace{\left\| \begin{pmatrix} Pr(V_1 = A) \\ Pr(V_1 = T) \\ Pr(V_1 = G) \\ Pr(V_1 = C) \end{pmatrix} \right\|_{\max}}_{\leq 1} \\
&\leq n^2 \cdot \left\| \begin{pmatrix} p_{AA}^2 & p_{AT}^2 & p_{AG}^2 & p_{AC}^2 \\ p_{TA}^2 & p_{TT}^2 & p_{TG}^2 & p_{TC}^2 \\ p_{GA}^2 & p_{GT}^2 & p_{GG}^2 & p_{GC}^2 \\ p_{CA}^2 & p_{CT}^2 & p_{CG}^2 & p_{CC}^2 \end{pmatrix} \right\|^{a_n-1}
\end{aligned}
$$

Die Matrix A läßt sich auf Jordan'sche Normalform bringen. Es existiert eine unitäre Transformationsmatrix M und Eigenwerte $\lambda_1, \lambda_2, \lambda_3, \lambda_4$ von A mit $A = MDM^T$, wobei

D die Diagonalmatrix der Eigenwerte ist. Da $||M|| \leq 1$ gilt, erhält man:

$$\begin{aligned} n \cdot \sum_{d=a_n+1}^{n} \cdot Pr(C_d \geq a_n) &\leq n^2 \cdot \max_{i=1}^{4} \lambda_i^{a_n-1} \\ &= n^2 \cdot \lambda^{a_n-1}. \end{aligned}$$

Dieser Term ist beschränkt $\leq C$, falls gilt:

$$\begin{aligned} \frac{1}{C} \cdot n^2 \cdot \lambda^{a_n-1} \leq 1 &\Leftrightarrow \log C^{-1} + 2\log n + (a_n - 1)\log \lambda \leq 0 \\ &\Leftrightarrow (a_n - 1)\log \lambda \leq -2\log n - \log C^{-1} \\ &\Leftrightarrow a_n \leq \frac{2\log n + \log C^{-1}}{\log \lambda^{-1}} + 1 \end{aligned}$$

Das heißt, es gilt:

$$a_n = O(\log n).$$

Betrachtung von (1):
Mit der Darstellung von $(**)$ gilt:

$$\begin{aligned} & n \cdot \sum_{d=1}^{a_n} Pr(C_d \geq a_n) \\ &= n \cdot \sum_{d=1}^{a_n} \sum_{\mathcal{T}_d} Pr(V^{\lfloor \frac{a_n}{d} \rfloor + 1} V') \qquad |V| = d, |V'| = r < d, V' \text{ Präfix von } V \\ &= n \cdot \sum_{d=1}^{a_n} \sum_{\mathcal{T}_d} Pr(V_1) Pr(V_2|V_1) \ldots Pr(V_d|V_{d-1}) \cdot \underbrace{Pr(V_1|V_d)}_{\leq Pr(V_1)} \ldots Pr(V_d|V_{d-1}) \\ &\qquad \cdot Pr(V_1|V_d) \ldots Pr(V_r|V_{r-1}) \\ &\leq n \cdot \sum_{d=1}^{a_n} \sum_{\mathcal{T}_d} Pr(V^{\lfloor \frac{a_n}{d} \rfloor} V') \cdot Pr(V) \\ &\leq n \cdot \sum_{d=1}^{a_n} \sqrt{\sum_{\mathcal{T}_d} Pr^2(V^{\lfloor \frac{a_n}{d} \rfloor} V')} \cdot \sqrt{\sum_{\mathcal{T}_d} Pr^2(V)} \text{ mit der Cauchy-Schwarz'schen Ungleichung.} \end{aligned}$$

Es ist $|V^{[\frac{a_n}{d}]} V'| = a_n$ und $\mathcal{T}_d \subset \mathcal{T}_{a_n}$. Damit folgt:

$$\begin{aligned} &\leq n \cdot \sum_{d=1}^{a_n} \sqrt{\sum_{\mathcal{T}_{a_n}} Pr^2(W)} \cdot \sqrt{\sum_{\mathcal{T}_d} Pr^2(V)} \\ &\leq n \cdot \sqrt{\sum_{\mathcal{T}_{a_n}} Pr^2(W)} \cdot \sum_{d=1}^{a_n} \sqrt{\sum_{\mathcal{T}_d} 1} \\ &= n \cdot \sqrt{\sum_{\mathcal{T}_{a_n}} Pr^2(W)} \cdot \sum_{d=1}^{a_n} \sqrt{|\Sigma|}^d \\ &= n \cdot \sqrt{\sum_{\mathcal{T}_{a_n}} Pr^2(W)} \cdot \frac{1 - \sqrt{|\Sigma|}^{a_n - 1}}{1 - \sqrt{|\Sigma|}} \end{aligned}$$

In obiger Rechnung wurde der Term $\sum_{\mathcal{T}_{a_n}} Pr^2(W)$ abgeschätzt. Es gilt (mit obiger Notation):

$$\sum_{\mathcal{T}_{a_n}} Pr^2(W) \;\leq\; \lambda^{a_n-1} \qquad \text{wobei } \lambda := \max_{i=1}^{4} \lambda_i \text{ gilt.}$$

Damit folgt insgesamt für den Summanden (1):

$$n \cdot \sum_{d=1}^{a_n} Pr(C_d \geq a_n) \;\leq\; n \cdot \sqrt{\lambda}^{a_n-1} \cdot \frac{1-\sqrt{|\Sigma|}^{a_n-1}}{1-\sqrt{|\Sigma|}}.$$

Gesucht sind jetzt diejenigen a_n, für welche die rechte Seite beschränkt ist. Beschränktheit gilt, falls:

$$\begin{aligned}
&\frac{1}{C} \cdot n \cdot \sqrt{\lambda}^{a_n-1} \cdot \frac{1-\sqrt{|\Sigma|}^{a_n-1}}{1-\sqrt{|\Sigma|}} \leq 1 \\
&\quad\Leftrightarrow\quad \log C^{-1} + \log n + (a_n-1)\log\sqrt{\lambda} + (a_n+1)\log(1-\sqrt{|\Sigma|}) - \log(1-\sqrt{|\Sigma|}) \leq 0 \\
&\quad\Leftrightarrow\qquad a_n \;\leq\; \frac{\log n + \log(C\cdot\sqrt{\lambda})^{-1}}{\log(\sqrt{\lambda}(1-\sqrt{|\Sigma|}))^{-1}}.
\end{aligned}$$

Das bedeutet:

$$a_n = O(\log n).$$

Damit ist die Behauptung gezeigt. □

Das heißt, die durchschnittliche Länge eines Präsuffix ist $O(\log n)$. Dann ist die durchschnittliche Länge aller n Präsuffixe von der Ordnung $O(n \log n)$.

Damit ist klar, die obere Schranke für die Hauptspeicherkomplexität der inneren Merge Prozedur ist $O(n \log n)$. Zufällige und sequentielle Diskzugriffe gibt es keine.

Bemerkungen zur Analyse des Konstruktionsalgorithmus

Analyse des Konstruktionsalgorithmus:

BEMERKUNG 1 :

- Die schnelle Laufzeit ergibt sich im wesentlichen aus der Tatsache, daß bei jedem Merging herunterberechnet wird. Das heißt, es ist sinnvoll so kurz wie möglich unter der Wurzel abzuschneiden. Siehe hierzu den Spezialfall aus $p > b$ Kapitel 5.3.
- Weiterhin erreicht man mit dem Traversing des Baumes von links nach rechts in 'Depth-First' Weise eine gewisse Sequentialität, die beim Bewältigen eines 'Memory Bottelnecks' wichtig ist.

BEMERKUNG 2 :

Im allgemeinen beschreibt der Algorithmus eine Version, wie sich ein Suffix Tree unter Benutzung zweier oder mehrerer Speicherarten aufbauen läßt. Ohne Beschränkung der Allgemeinheit lassen sich fürs erste zwei Speicherarten annehmen. Erstens, ein langsamer Speicher, welcher von der Größe her unbeschränkt zur Verfügung steht. Und ein schneller Speicher, welcher in der Regel teuer ist, und daher von der Größe nicht unbeschränkt. Das Schreiben und Lesen von der ersten Speicherart in die Zweite ist meist sehr Zeit-intensiv, und wird als 'Memory Bottleneck' bezeichnet.

Benötigt man für die Abarbeitung eines Programmes beide Speichermedien gleichzeitig, so zeigt sich, daß es uneffizient ist, Algorithmen so zu designen, als wenn beide Medien zusammen eine homogene Einheit bilden. Der 'Memory Bottleneck' zwischen beiden Medien muss beachtet werden.

In tatsächlichen Computerarchitekturen tritt dieser Bottleneck, zum Beispiel zwischen Hauptspeicher und Sekundärspeicher (Festplatte) auf – wenn man den Hauptspeicher als homogenes Medium betrachtet. Dieser ist signifikant und kann bei der tatsächlichen Praxis nicht außer acht gelassen werden. Bei Nichtbeachtung können Laufzeitverluste von mehreren Stunden eintreten, wenn nicht gar Tagen. Betrachtet wird dies besonders bei Anwendungen im Datenbankbereich, wo traditionell die bearbeiteten Datenmengen so groß sind, daß sie auf der Festplatte gespeichert werden, man aber trotzdem einen effizienten Zugriff auf verschiedenen ausgewählten Teilen benötigt.

Aber auch der tatsächliche Hauptspeicher ist an sich in weitere kleinere Einheiten unterteilt. In eine sehr schnelle und kleine CPU-Einheit und mehrere langsamere Caches. Auch zwischen diesen Einheiten existieren 'Memory Bottlenecks'. Diese sind aber, wesentlich geringer und ihre Nichtbeachtung bewirken nur Performance Verluste, welche in den Anwendungen im Sekundenbereich liegen.

Vorausichtliche Entwicklung der Hauptspeichergrößen: Da man die nächsten 10 Jahre von einem starken Wachstum der realen Hauptspeichergrößen ausgeht, können sich bei gößeren Dimensionen die heutigen Sekunden zu Minuten aufaddieren. Daher gibt es schon heute Bestrebungen Algorithmen zu finden, die auch diese Form von 'Bottelneck' berücksichtigen.

Das abstrakte Speichermodell, welches die Grundlage der Analyse dieses Algorithmus darstellt, definiert nur einen Bottleneck. Der vorgestellte Algorithmus beschreibt *nur* die Überwindung eines solchen Bottlenecks. Aufgrund der Inhomogenität selbst heutiger Hauptspeicher erhöht die Methode die Performance, sowohl in heutigen Hauptspeichern als auch beim Management zwischen Hauptspeicher und Festplatte.

Im weiteren Verlauf werden Laufzeitenvergleiche in erster Linie nur im Hinblick auf letzteres unternommen.

Desweiteren ist es für die Bedeutung des Algorithmus wichtig, ob auch in fernerer Zukunft beim rasanten Anstieg der Hauptspeichergröße immer noch Texte existieren, deren Suffix Trees nicht in den Hauptspeicher passen. Bei der in dieser Arbeit vornehmli-

chen Anwendung in der Bioinformatik, ist dies auch in Zukunft zu erwarten. Die Menge an Sequenzdaten wächst mehr als exponentiell. Die Methoden der Sequenzierung sind immer besser geworden – 'Throughput' Verfahren haben sich durchgesetzt. Und es wurden derzeit nur kleinere Genome – Mensch und Maus Genome – vollständig sequenziert. Die Entwicklung der Genom Sequenzierungsprojekte ist dahingehend, daß begonnen wurde weitere Artengenome, sowohl der Tier- als auch Pflanzenwelt, wie zum Beispiel Kulturpflanzen, zu entschlüsseln. Diese sind meist weit größer als die bekannten 3 Giga 'Base Pairs' von Maus oder Mensch.

Einführung eines Memory – Sequenz Koeffizienten und Bewertung des Algorithmus nach diesem Koeffizienten (theoretisch)

Es wird die Nutzbarkeit von Konstruktionsalgorithmen bezüglich des Memory-Sequenz Koeffizienten (M-S Koeffizient) betrachtet:
Der M-S Koeffizient ist das Verhältnis der Hauptspeichergröße zur zu indexierenden Sequenzgröße. Reine Hauptspeicheralgorithmen haben ein Verhältnis je nach Größe des Indexes. Das heißt, bei Ukkonens Variante, naive implementiert, beträgt das Verhältnis ungefähr $1,4\%$. Dies kann sich durch eine geschickte Implementierung von Ukkonens Algorithmus bis zu $10,7\%$ verbessern, gezeigt durch [Kur99]. Ist ein sehr langer Text gegeben, das heißt es liegt real ein größeres Verhältnis vor, so wächst die Laufzeit ab diesem Verhältnis exponentiell an. [Sch03] stellte die erste Variante für eine persistente Implementierung vor. Hier liegt das Verhältnis bei $15,6\%$ $(80/512)$. Der Algorithmus verhält sich ab diesem Koeffizienten in der Laufzeit exponentiell, läuft aber nicht in einen Error. Ein weiterer Algorithmus ist wotd ([Gie03]). Dieser hat einen Koeffizienten von 26% $(120/512)$ und verhält sich ab dort exponentiell. Für sehr große Werte ist eine Abarbeitung nicht möglich. Der in dieser Arbeit vorgestellte Algorithmus ist experimentell bis zu 87% getestet worden. Aber ab dort liegen keine Grenzen für eine weitere Indexierung. Der Algorithmus terminiert auch dann in angemessener Zeit, falls die Sequenz größer als der Hauptspeicher ist. Diese Eigenschaft konnte bei anderen Algorithmen nicht nachgewiesen werden.

Degenerierung des Algorithmus

Aber aus theoretischer Überlegung ist klar, daß auch der vorgestellte Algorithmus degeneriert. Dies kann für einen zu großen M-S Koeffizienten eintreten. Im schlechtesten Fall, falls sich nach einer anfänglichen Datenteilung zu jedem Segment nur der Wurzelknoten berechnen ließe. In dieser Arbeit sind wir in den Experimenten nicht bis zu dieser Grenze gegangen.

Suffix Tree Checkers

Als Suffix Tree Checkers werden Algorithmen bezeichnet, welche zu einem gegebenen Suffix Tree und einem Text prüfen, ob der Tree tatsächlich der zugehörige Suffix Tree

zum Text ist. Bei sehr großen DNA Texten ist dies eine offene Fragestellung.

5.6 Schritt 3 : Endmischen

Aus der Analyse von Schritt 2 ist ersichtlich, warum Schritt 3 sinnvoll ist. Obwohl die Methode aus Schritt 2 eigentlich ein natürliches Ende hat, wird im Kapitel 5.3 unter Spezialfall $p > b$ die Existenz eines k_ν gezeigt, für welches der Wert $k_\nu(k_\nu - 1) \cdot p^\nu$ geringer ist als die Restoperationen von Schritt 2 ab k_ν.

Damit hat Schritt 3 eine durchschnittliche Komplexität der Ordnung $O(n \log n)$. Die obere Schranke für die seriellen Diskzugriffe ist $S(k_\nu(k_\nu - 1) \cdot p^\nu)$. Zufällige Diskzugriffe werden komplett vermieden.

Das heißt, als obere Schranke der Komplexität ergibt sich

$$(O(n \log n),\ S\left(\lceil \lfloor (n+1)/m \rfloor / b^\nu \rceil \left(\lceil \lfloor (n+1)/m \rfloor / b^\nu \rceil - 1\right) p^\nu,\ 0\right).$$

Zusammenfassung:

Die Analyse der einzelnen Schritte zeigt, daß auf diese Weise ein quasilinearer Konstruktionsalgorithmus mit wesentlich reduziertem Disk I/O erreicht wurde. Im nächsten Kapitel wird sich dieses theoretische Resultat auch in den Laufzeiten wiederspiegeln.

Kapitel 6

Experimentelle Ergebnisse

In den Experimenten haben wir uns auf die zwei Hauptmeßwerte konzentriert. Erstens die Konstruktionszeit, zweitens die Anwortzeit bezüglich einer Anfrage nach einem exaktem Pattern. Wie in Kapitel 2 vorgestellt, basieren viele Methoden der inexakten Suche auf der Suche nach exakten Mustern. Je komplizierter die Suchart, um so mehr exakte Pattern sind für diese auszuwerten. Daher wurden Analysen bezüglich einer hohen Suchzahl durchgeführt.

6.1 Konstruktionszeiten

Tabelle 6.1: Konstruktionszeiten für verschiedene Sequenzen

Sequenz	Länge	Konstruktionszeit
E. Coli Genom	4,6 Mbp	21 sec
S. Cerevisiae Genom	12,1 Mbp	4,6 min
H. Sapiens Teil vom Chr. 22	33 Mbp	20 min
H. Sapiens Teil vom Chr. 22	87 Mbp	1,5 h
A. Thaliana Genom	120 Mbp	2 h
H. Sapiens Chr. 2	220 Mbp	7 h
Bakterien Genom (Teil)	336 Mbp	10,2 h
Bakterien Genom	450 Mbp	18 h

Die Laufzeitexperimente wurden auf einem Rechner mit $2,4$ GHz CPU und 512 MB RAM unter einer Suse Linux Distribution 8.2 durchgeführt. Der Datenpartitionierungsalgorithmus aus Kapitel 5, abgekürzt mit TP für Textpartitionierung [1], wurde prototy-

[1]Die Abkürzung TP richtet sich danach, daß die betrachteten Daten in unserem Fall nur Texte sind

pisch implementiert.

Laufzeit bei unterschiedlichen Genomen von unterschiedlicher Länge

Der Konstruktionsalgorithmus wurde auf einer Vielzahl von Genomen angewendet. Unterschiedliche Genome haben unterschiedliche Merkmale in der Verteilung. Grob lassen sich Prokaryonten und Eukaryonten unterteilen (siehe Kapitel 1).

Als Prokaryonten wurde eine Aneinanderreihung aller Bakterien Genome erzeugt. Für diesen Test wurden alle derzeit frei verfügbaren bakteriellen Genome, downloadbar im FASTA Format, unter anderem unter *http://www.ncbi.com/....*, gewählt. 155 Bakterien Genome mit jeweils Längen zwischen $0,4$ MB bis $9,3$ MB wurden in willkürlich festgelegter Reihenfolge aneinander gekettet.

Man erhält einen String von der Länge 450 MB. Bakterielle Genome sind meist sehr klein. Daher ist eine Vielzahl von ihnen schon vollständig sequenziert. Sie wurden als Text über dem Alphabet $\{A, T, G, C, N\}$ veröffentlicht. Die Buchstaben N drücken Ungenauigkeiten in der experimentellen Bestimmung der tatsächlichen Basen aus. In den letzten Jahren wurden diese Buchstaben gegen die tatsächlichen Zeichen (A,T,G oder C) ausgetauscht und somit bisherige Sequenzierungen weiter korrigiert. Da bakterielle Genome schon zum Beginn des Genom Sequenzierungsprojektes [2] 1993 erforscht wurden, sind derartige Korrekturen meistens erledigt. Daher ändert sich die Struktur und Verteilung der Basen durch das Löschen der N Zeichen wenig. Das Herausstreichen der N Symbole hat auf die Konstruktion kaum Auswirkungen.

Als Eukaryonten wurden betrachtet: *Saccharomyces Cerevisiae* (Bäcker Hefe), *Arabidopsis Thaliana*, Chromosom 2 vom Maus Genom und das Genom des Menschen, ebenfalls Chromosom 2. In Tabelle 6.1 befinden sich die Konstruktionszeiten für die ausgewählten Sequenzen. Der längste zusammenhängende String ist die Konkatenation aller Bakterien Genome – zur Zeit des Schreibens 450 MB.

Graphik 6.1 illustriert das Verhalten der Kurve der Performance gemessen in der Laufzeit abhängig von der Länge des Strings. Dies ist eine Performancebewertung im Großen. Daher sind die zwei kleinsten Strings weggelassen. Es zeigt sich auch in den Laufzeiten, daß sich der Algorithmus $O(n \log n)$ verhält und zwar für große n. Das heißt, daß das asymptotische Verhalten des Algorithmus gilt, wie in der stochastischen Analyse nachgewiesen.

Man sieht in der Kurve deutliche Knicke an den Punkten 119 MB und 337 MB. Diese ergeben sich aus den jeweiligen Durchläufen bzw. Stufen der Partitionierungen auf der Festplatte. Das heißt, sie ergeben sich aus den Durchläufen des externen Mischens, wie in Kapitel 5.3 beschrieben.

Für 450 Mbp benötigt man 4 Durchläufe für das externe Mischen. Für die 220 Mbp menschlichen Genoms wurden 3 externe Mischvorgänge durchgeführt. Ebenfalls 3 wer-

und wir daher gerade Textpartitionierung durchführen.

[2] Das Genom Sequenzierungsprojekt ist derzeit abgeschlossen.

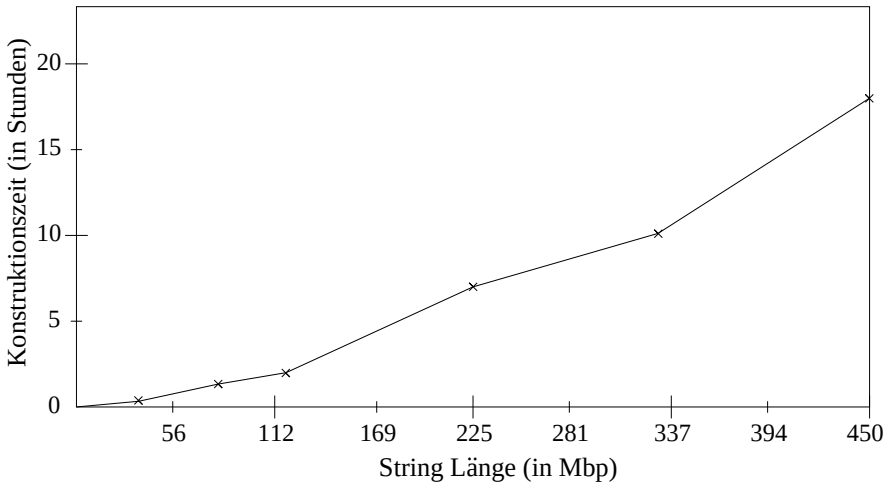


Abbildung 6.1: Entwicklung der Laufzeiten abhängig von der Länge des indexierten Textes

den bei 336 Mbp durchgeführt. Es gibt für jede Hauptspeichergröße, hier $M = 512$ MB, gewisse Schwellenwerte, für welche die Kurve der Konstruktionszeit einen Sprung macht. Bis zu diesem Schwellenwert steigt die Zeit 'fast' linear an. Der Schwellenwert ist dasjenige Argument, für welches sich die Anzahl des externen Mischens um 1 erhöht. In unserem Fall existert ein Schwellenwert $s_{3-4} \in (336\text{ Mbp}, 450\text{ Mbp})$.

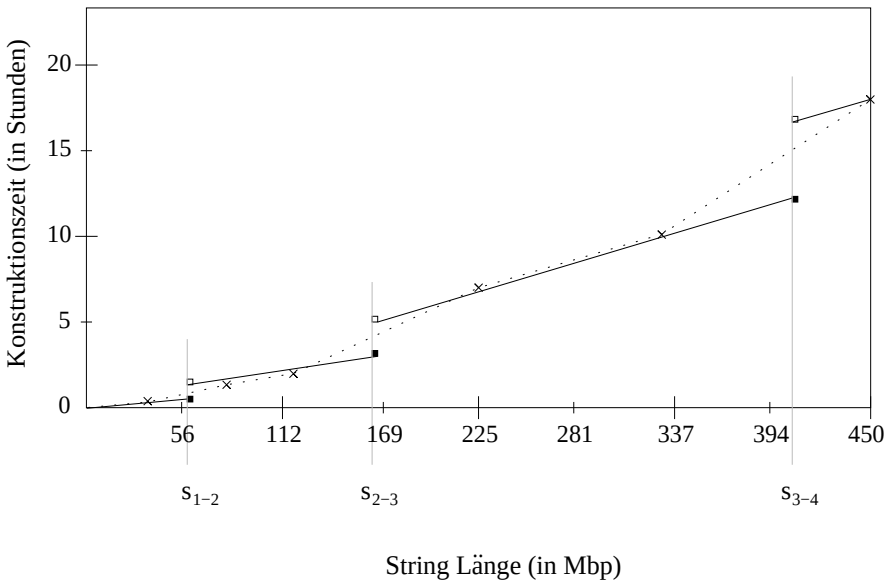


Abbildung 6.2: Voraussichtliches Verhalten der tatsächlichen Entwicklung der Laufzeiten abhängig von der Länge des indexierten Textes

Für 87 Mbp wurden 2 Durchläufe benötigt. Ein weiterer Schwellenwert liegt daher zwischen 87 Mbp und 220 Mbp, $s_{2-3} \in (87\text{ Mbp}, 220\text{ Mbp})$. Und es ist $s_{1-2} \in (12,1\text{ Mbp}, 87\text{ Mbp})$.

Beim *E.Coli* Genom wurden nur Schritt 1 des Algorithmus aus Kapitel 5 durchgeführt.

Schritt 2 und Schritt 3 waren nicht notwendig, da die Sequenz klein genug war. Daher beinhaltet die gemessene Zeit nur die Konstruktion eines Suffix Trees in-memory und den Vorgang des Schreibens auf die Disk, wie in Kapitel 3 beschrieben. Das heißt, der Schwellenwert ist $s_{0-1} \in (0\text{ Mbp}, 12,1\text{ Mbp})$.

Diese Schwellenwerte wurden nicht exakt berechnet. In Abbildung 6.2 sind nur fiktive Näherungswerte eingetragen. Die durchgehende Kurve stellt das vermutete tatsächliche Laufzeitverhalten dar, mit den entsprechenden diskreten Unstetigkeitsstellen.

Genaue Analyse der Laufzeiten einzelner Teilschritte

Tabelle 6.2: Konstruktionszeiten für verschiedene Sequenzen

Sequenz	Länge	Muster	Rechenzeit
E. Coli	4,6 Mbp	$\emptyset$	21 sec
S. Cerevisiae	12,1 Mbp	$\emptyset$	72 sec
		$\{A, T, G, C\}$	3, 4 min
H. Sapiens	220 Mbp	$\emptyset$	1 h 20 min
		$\{A, T, G, C\}$	1 h
		$\{A, T, G, C\}$	1 h
		$\{AA, AT, AG, AC, \ldots, CC\}$	3 h 40 min
Bakterien G.	450 Mbp	$\emptyset$	1 h
		$\{A, T, G, C\}$	1 h 20 min
		$\{A, T, G, C\}$	1 h 50 min
		$\{AA, AT, AG, AC, \ldots, CC\}$	3 h 20 min
		$\{AA, AT, AG, AC, \ldots, CC\}$	10 h 50 min

In Tabelle 6.2 sind durch die Muster, Beispiele für Segmentierungen angegeben, wie sie in den Experimenten genutzt wurden.

Laufzeiten im Vergleich mit anderen Referenzalgorithmen

Die oben vorgestellten Konstruktionszeiten werden hier mit Referenzalgorithmen zur Konstruktion verglichen.

Es wurde ein Algorithmus von Ukkonen gewählt mit einer klassischen Implementierungstechnik als Kantenbaum. Diese Implementierungsvariante ist mit durchschnittlich 53 Byte pro indexiertem Zeichen nicht besonders Speicherplatz effizient, im Gegensatz zur Variante von [Kur99]. Es handelt sich um eine Open-Source Implementierung von Tsadok und Yona aus dem Jahr 2002, downloadbar im Internet unter *http://cs.haifa.ac.il/~shlomo/suffix_tree/.* . Bei einer Betriebssystem Instanz von Suse Linux 8.2 und einer Hauptspeichergröße von 512 MB terminierte der Algorithmus für Texte bis zu einer Größe von 7, 9 MB. Für darüber hinausgehende Texte mußte der Algorithmus nach 24 Stunden abgebrochen werden.

Diese 7, 9 MB stellen eine Barriere der Anwendbarkeit des Algorithmus dar. Für Platz-

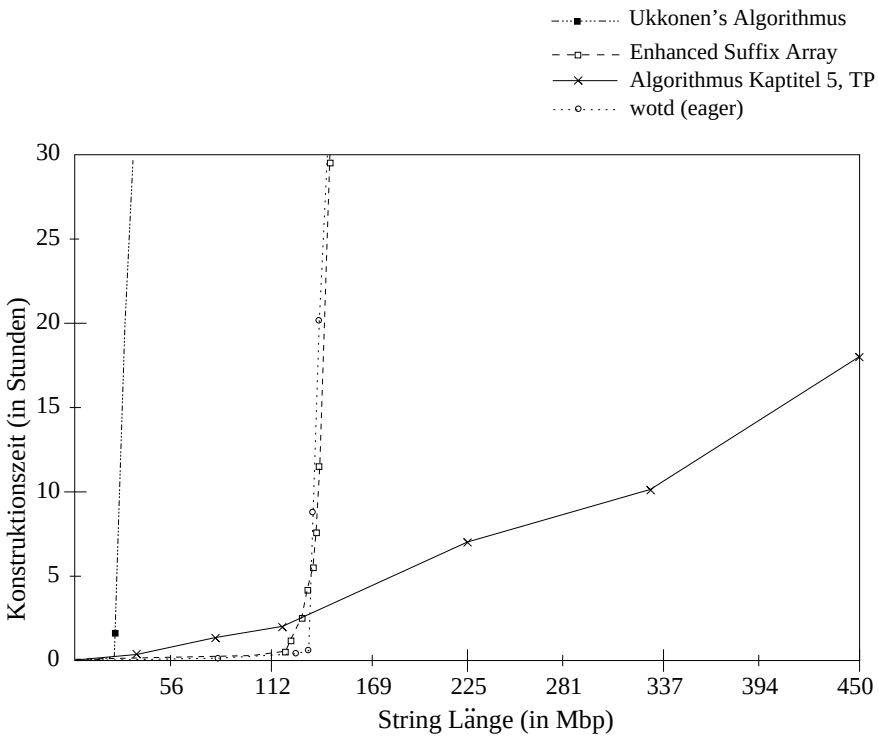


Abbildung 6.3: Entwicklung der Laufzeiten im Vergleich zu anderen Algorithmen

effizientere Implementierungen wird diese Textgrenze zwar weiter nach der Größe des Textes hinausgeschoben, aber die Existenz einer solchen Barriere ist damit nicht aufgehoben.

Möchte man längere Texte indexieren, hat man nur die Wahl, entweder die Hauptspeichergröße zu erhöhen oder eine noch effizientere Implementierung zu finden.

Eine effizientere Implementierung ist das Enhanced Suffix Array (ESA), dokumentiert in [Abo02]. Ein Suffix Array ist dem Namen nach keine Implementierung eines ganzen Baumes, sondern nur eine lexikographische Anordnung der Blätter eines solchen Baumes. Diese Blätter werden in einem Array gespeichert. Der Speicherplatzbedarf ist bei dieser einfachen Variante $4n$ Bytes für $|t| = n$. Die Suchzeit für exakte Pattern hat eine Komplexität von $O(|p| \log n)$ [3]. Dieses einfache, eindimensionale Array kann durch die Angabe eines Arrays mit lcp-Werten[4] zu einem zweidimensionalen Array erhöht werden. Der Speicherplatzbedarf erhöht sich und die Komplexität der Suche reduziert sich. Beschrieben wurde dies in [Man93].

Beim Enhanced Suffix Array wurde die Dimension des Array's soweit erhöht, daß die Komplexität der Suche nach exakten Pattern von der Ordnung $O(|p|)$ ist. Das bedeutet, daß die Tree Topologie des Baumes, welche beim einfachen Suffix Array gestrichen wurde, hier vollständig kodiert ist. Das Enhanced Suffix Array ist die Abbildung aller Suffix Tree Elemente auf ein Array. Der Speicherplatzbedarf bei DNA Sequenzen liegt

[3] Von einigen Autoren auf $O(|p| + \log n)$ verbessert.
[4] lcp = 'longest common prefix'

ungefähr bei 6 Byte pro Textzeichen und konnte somit weiter reduziert werden. Das ESA wird persistent auf der Disk gehalten und stellt damit einen guten Vergleich dar.

Die Implementierung von ESA ist als Teil der 'vmatch' Distribution verfügbar. Als DNA Datenstring wurden die 450 MB bakterieller Genome gewählt und auf verschiedene Testlängen, siehe Tabelle 6.3, gekürzt. Der Programmaufruf lautet *./mkvtree -db <dna_file> -dna -pl -allout -v*.

Tabelle 6.3: Laufzeitverhalten von ESA

Länge der Sequenz in Mbp	Konstruktionszeit	Länge der Sequenz in Mbp	Konstruktionszeit
2	1,81 sec	124	2 h 17 min
40	1 min 12,5 sec	126	4 h 31 min
60	2 min 0,86 sec	128	5 h
120	24 min	130	5 h 26 min
122	46 min	132	7 h 29 min
		134	11 h 31 min
		136	29 h 49 min

Die gesamten 450 Mbp konnten wegen Fehlermeldung nicht verarbeitet werden. Ein Versuch mit 159 Mbp wurde nach drei Tagen abgebrochen. Der Vergleich der Laufzeiten zeigt, daß bis zu einer Größe von 122 Mbp die Konstruktionszeiten des ESA wesentlich geringer sind, als die Konstruktionszeiten aus Tabelle 6.1. Ab 124 Mbp steigt die Konstruktionszeit drastisch an. Sie liegt ab dieser Textlänge höher als bei dem in dieser Arbeit vorgestellten TP Algorithmus. Eine Barriere der Anwendbarkeit für diese Methode liegt bei höchstens 140 MB.

Zieht man die durchschnittlich 6 Byte pro Zeichen Platzbedarf in Betracht, so sind dies bei 136 Mbp ungefähr 816 MB an Platzbedarf.[5] Dies geht doppelt über die nutzbare Hauptspeichergröße hinaus. Der Algorithmus arbeitet anders als Ukkonens Methode, bei der Indexierung auch mit der Disk. Bei der Angabe der Laufzeiten ist die persistente Speicherung enhalten.

Wotd.x ([Gie03]) ist eine Implementierung des naïven Suffix Tree Konstruktionsalgorithmus. Daher ist die theoretische Komplexität von der Ordnung $O(n^2)$ (siehe Kapitel 4). Wie in Kapitel 4 erläutert benötigt man keinen zusätzlichen Platz für Suffix Links. Daher konnte man einen Speicherbedarf von ≈ 11 Byte pro Buchstabe bei DNA Sequenzen realisieren. Die Barriere vom Ukkonen's Algorithmus ist aus diesem Grund von $7,9$ Mbp schon einmal auf ungefähr 38 Mbp verschoben.

Wie in Kapitel 4 beschrieben, hat der naïve Algorithmus eine 'bad locality behaviour'.

[5] Bei einer Hauptspeichergröße von 512 MB kann man bei laufender Linux Instanz nicht diese kompletten 512 MB nutzen, sondern je nach Betriebssystem nur ungefähr 400 MB.

Dem versucht der *wotd*-Algorithmus in Ansätzen entgegen zu steuern. Beim *wotd*-Algorithmus wurde eine Implementierung realisiert, welche durch geschichtete Sortierungen der Suffixe eine gewisse Sequentialität in die Abarbeitung der Konstruktion einführt. Aufgrund dieser Sequentialität ist es möglich, Zwischenergebnisse teilweise Zeit effizient auf der Disk zu speichern. Der Algorithmus kann daher mit zwei Speicherarten arbeiten. Es wird sich zeigen, daß sich die Anwendbarkeitsgrenze weit über die 38 Mbp hinausschieben läßt.

Tabelle 6.4: Laufzeitverhalten von *wotd* -eager

Länge der Sequenz in Mbp	Konstruktionszeit	Länge der Sequenz in Mbp	Konstruktionszeit
2	2,91 sec	131	42 min 4 sec (Fehler)
40	1 min 55 sec	132	40 min 13 sec (Fehler)
60	3 min 30 sec	134	36 min 15 sec (Fehler)
80	4 min 53 sec	136	3 h 19 min (Fehler)
120	11 min 30 sec	138	10 h 42 min (Fehler)
130	12 min 40 sec	140	11 h 31 min (Fehler)

Wotd ist als Implementierung frei verfügbar. Es werden folgende Varianten von wotd unterschieden: wotdeager (*wotd -eager*) und wotdlazy (*wotd -lazy*). Wotdlazy berechnet den Baum nicht vollständig. Die Berechnung der Baumteile wird erst durch die Suche der generierten Pattern ausgelöst. Wieviele Suchen gestartet werden, wird durch einen Faktor $\rho \in [0,1]$ angegeben. Bei $\rho = 1$ erhält man einen inakzeptablen Performance-Verlust. Desweiteren ist festzustellen, daß sowohl wotdeager als auch wotdlazy Vorteile für Strings mit größeren Alphabet bringen, aber wie von den Autoren Giegerich *et al* [Gie03] gezeigt, gerade bei DNA Sequenzen keine gute Laufzeiten in der Praxis für sehr kurze Strings zeigen. Sie sind auf jedem Fall langsamer als der lineare Algorithmus von McCreigth. Dafür hat dieser Algorithmus den Vorteil, daß er für wesentlich größere Sequenzen skalierbar ist. Wotdeager ist bei den meisten Anwendungen langsamer als wotdlazy.

Für die Meßwerte aus Tabelle 6.4 wurden, wie oben die 450 Mbp bakteriellen Genoms verwendet. Dieser Text wurde auf die entsprechenden Längen aus der Tabelle gekürzt. Der Funktionsaufruf lautet: *./wotd.x -eager 0 0 1 <dna_file>*.

Der wotd Algorithmus macht den entstandenen Tree nicht persistent. Daher ist die genommene Zeit nicht voll mit dem ESA Algorithmus und dem Datenpartitionierungsalgorithmus zu vergleichen. Für Werte bis zu 130 Mbp ist die Laufzeit wesentlich geringer als beim Datenpartitionierungsalgorithmus (Tabelle 6.1). Für kleine Filelängen von 0 bis 60 Mbp liegen die Laufzeiten von *wotd.x* sogar über denen von ESA, obwohl die Methode rein im Hauptspeicher arbeitet. Für 120 bis 130 Mbp sind die Laufzeiten für *wotd.x* wesentlich geringer. Ab 131 Mbp terminierte das Programm nicht mehr. Die Meßwerte

liefern die Zeit bis zur Fehlermeldung. Wenn das Programm mehr Platz als RAM und SWAP zusammen braucht, dann tritt der Fehler: 'not enough memory' auf. Die Versuche aus Tabelle 6.4 zeigen jedoch, daß der Abbruch im Bereich von 134 bis 140 Mbp immer später erfolgt, das heißt, daß in diesem Bereich die Sortierungskomplexität exponentiell ansteigt. Versuche mit 160 Mbp zeigten, ab dieser Menge endete das Programm nicht vor 3 Tagen und 4 Stunden.

Die Barriere der Anwendbarkeit liegt bei höchstens 140 Mbp.

Unterschiedliche Skalen bzgl. der Anwendbarkeit mit Referenzalgorithmen

Es lassen sich damit unterschiedliche Skalen der Anwendbarkeit der Algorithmen unterteilen.

Einteilung in drei Bereiche: $0 - 10$ Mbp, $10 - 140$ Mbp, $140 - \infty$ Mbp

Diese Einteilung bezieht sich auf, die in diesen Untersuchungen durchgängig gewählte, Hauptspeichergröße von 512 MB.

- Strings der Länge 0 bis 10 Mbp

  Bei dieser Größe des Eingabetextes ist es möglich, sowohl die Sequenz als auch den gesamten Index voll im Hauptspeicher während der Konstruktion zu halten. Es kann daher ein freier und beliebiger Zugriff auf jede Memory Referenz erfolgen. Algorithmen mit einem "bad locality behaviour", wie gerade lineare von Ukkonen, können angewendet werden. Diese sind am performantesten.

  Der getestete Algorithmus von Ukkonen hat in diesem Bereich geringere Laufzeiten als *wotd.x*, ESA, und der Datenpartitionierungsalgorithmus. Der Grenzwert 10 wurde hier abgeschätzt für eine effizientere Implementierungsvariante (oben $7,9$ Mbp).

- Strings der Länge 10 bis 140 Mbp

  Es sind hier jene Algorithmen am performantesten, welche die Sequenz im Hauptspeicher halten können *und* einen großen Teil des Indexes auch. In unserem Test sind das der *wotd*- und der ESA Algorithmus.

  Bei diesen Algorithmen wurden Strategien entwickelt, um ausgewählte Teilabschnitte des Indexes aus dem Speicher ein- und auszuladen. Es wird damit eine gewisse Sequentialität erreicht. Dies markiert den Übergang vom "bad locality behaviour" zum "improved locality behaviour".

- Strings der Länge 140 bis ∞ Mbp

  Der Datenpartitionierungsalgorithmus stellt eine Variante vor, bei welcher sowohl

die gesamte Sequenz als auch der Index auf der Festplatte während der Konstruktion belassen werden können. Daher skaliert er auch über 140 Mbp hinaus. Die Lauffähigkeit ist auch für Texte größer als der Hauptspeicher gegeben.

Die Laufzeiten erhält man durch die Sequentialisierung und durch die Nutzung der totalen Dereferenzierung.

6.1.1 Zusammenfassung und Interpretation

Die Implementierung bestätigt, daß der vorgestellte Algorithmus für die angestrebten Textlängen anwendbar ist und damit weit hinter den bisher genutzten Varianten skaliert. Auf diese Weise ist ein tatsächlicher Aufbau eines persistenten Indexes für relevante, anwendungsnahe Größenordnungen erst ermöglicht. Die umfangreichen Experimente mit Referenzalgorithmen dokumentieren dies.

Vorgelegte Experimente zeigen das Verhalten des Algorithmus für eine Maschine mit $0,5$ GB RAM. Mit einem Computer von 2 GB RAM läßt sich in ganz andere Bereiche skalieren.

6.2 Meßzeiten der exakten Suche

Alle Experimente wurden durchgeführt auf einem 1200 MHz AMD Athlon PC mit 256 MB RAM, unter der Linux Version: Red Hat 7.0. Für diese Experimente wurden $33,4$ MB DNA Sequenz indexiert. Diese bestehen aus aneinander gehängten Stücken menschlicher DNA von Chromosom 2.

Man unterscheidet, wie bei Anwendungen in Datenbanken typisch, 'cold store' und 'warm store' Anfrage Verhalten. 'Cold store'Anfrage Verhalten bedeutet, die tatsächliche Zeit, welche zum Finden und Laden der Positionen in den Hauptspeicher benötigt wird. Es ist die Zeit im 'worst case'. In Datenbanken werden Anfragen mehrmals hintereinander gestellt, und damit vorherige Anfragen im Cache zwischengespeichert. Die Messung dieser Zeit nennt sich 'warm store' Anfrage Verhalten.

Abbildung 6.4 zeigt einen Vergleich der wesentlich verringerten Zugriffszeiten auf den Index, infolge der verbesserten Repräsentation, beschrieben in Kapitel 3.

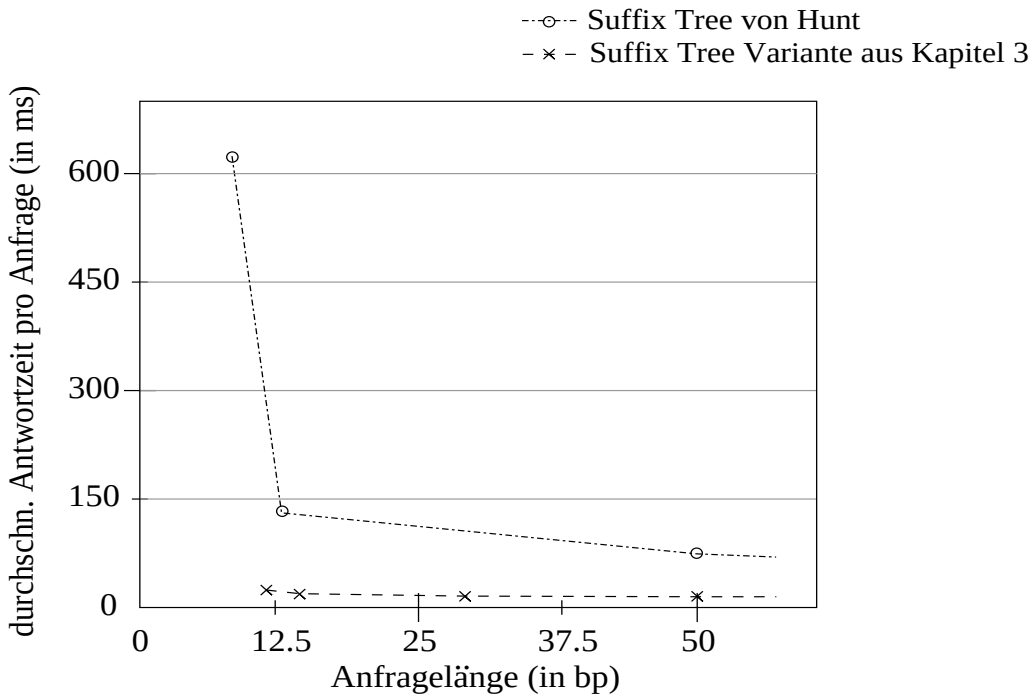


Abbildung 6.4: Vergleich der Zugriffszeiten mit den Zugriffszeiten vom Tree, entwickelt von *Hunt* (siehe Kapitel 4.5)

Tabelle 6.5: 'Cold Store' Anfrage Verhalten

Größe der Batches	Anfrage Länge in Byte	durchschn. Antwortzeit pro Anfrage in ms	alle Treffer
100	12	40	86.690
1.000	12	19	195.840
10.000	12	18,8	989.802
50.000	12	16,42	4.103.959
100	17	30	26.113
1.000	17	16	46.636
10.000	17	15,3	244.059
50.000	17	15,58	1.112.030
100	30	20	885
1.000	30	15	2.765
10.000	30	14,9	17.897
50.000	30	15,24	100.094
100	50	20	118
1.000	50	15	771
10.000	50	14,3	6.874
50.000	50	15,3	34.607
100	70	10	64
1.000	70	15	657
10.000	70	15,5	6.494
50.000	70	15,3	32.315

Tabelle 6.6: 'Warm Store' Anfrage Verhalten

Größe der Batches	Anfrage Länge in Byte	durchschn. Antwortzeit pro Anfrage in ms	alle Treffer
100	12	0	86.690
1.000	12	0,5	195.840
5.000	12	0,4	543.521
7.000	12	0,29	671.647
100	17	0	26.113
1.000	17	0	46.636
5.000	17	0,1	146.656
7.000	17	0,14	1.112.030
100	30	0	885
1.000	30	0	2.765
5.000	30	0,1	11.800
7.000	30	0,14	14.247

Dank

Ich danke in erster Linie Herrn Prof. Hoffmann, der mich durch seine Unterstützung angespornt hat, das Thema der Arbeit zielgerichtet zu verfolgen, und mir jederzeit mit seinem fachlichen Rat zur Verfügung stand. Ich danke Herrn Prof. Kramer, der mit viel Anregung und Kritik aus der Bioinformatik zum Gelingen der Arbeit beigetragen hat.

Literaturverzeichnis

[Abo02] M.I. Abouelhoda, E. Ohlebusch, S. Kurtz. *Optimal exact string matching based on suffix arrays.* Proceedings of the Ninth International Symposium on String Processing and Information Retrieval, Springer Verlag, Berlin, 2002.

[Alt90] S.F. Altschul et al. *Basic local alignment search tool.* Journal of Molecular Biology, 215: 403-10, 1990.

[Alt97] S.F. Altschul, T.L. Madden, A.A. Schaeffer, J. Zhang, W. Miller, D.J. Lipman. *Gapped BLAST and PSI-BLAST: a new generation of protein search programs.* Nucleic Acids Research, 25: 3389-3402, 1997.

[And95] A. Andersson, S. Nilsson. *Efficient Implementation of Suffix Trees.* Software Praktice and Experience, 25(2): 129-141, 1995.

[AG86] A. Apostolico, R. Giancarlo. *The Boyer-Moore-Galil string searching strategies revisited.* SIAM Journal on Computing, vol 15, no 1, pp 89-105, 1986.

[Bae99] R. Baeza-Yates, B. Ribeiro-Neto. *Modern Information Retrieval.* ACM Press, New York, 1999.

[Bed04] S.J. Bedathur, J.R. Haritsa. *Engineering a fast online persistent suffix tree construction.* Proceedings of the International Conference on Data Engineering, 2004.

[Ben99] G. Benson. *Tandem repeats finder: a program to analyze DNA sequences.* Nucleic Acids Research, vol 27, no 2, pp 573-580, 1999.

[Bes01] J. Besemer, A. Lomsadze, M. Borodovsky. *GeneMarkS: a self-training method for predicition of gene starts in microbial genomes. Implications for finding sequence motifs in regulatory regions.* Nucleic Acids Research, vol 29, no 12, pp 2607-2618, 2001.

[Bie95] P.]. Bieganski. *Genetic Sequences Data Retrieval and Manipulation based on Generalized Suffix Trees.* PhD Thesis, University of Minnesota, USA, 1995.

[BM77] R.S. Boyer, J.S. Moore. *A Fast String Searching Algorithm.* Communications of the ACM 20, 10, pp 762-772, 1977.

[Bur99] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, M. Vingron. *Q-gram based database searching using suffix array.* Proceedings of the 3rd Annual International Conference on Computational Molecular Biology (RECOMB '99), ACM Press, pp 77-83, 1999.

[Bur03] S. Burkhardt, J. Kärkkäinen. *Fast Lightweight Suffix Array Construction and Checking.* Combinatorial Pattern Matching (CPM), pp 55-69, 2003.

[Cra02] A. Crauser, P. Ferragina. *A theoretical and experimental study on the construction of suffix arrays in external memory.* Algorihtmica, 32(1): 1-35, 2002.

[Coo01] B.F. Cooper, N. Sample, M.J. Franklin, G.R. Hjaltason, M. Shadmon. *A Fast Index for Semistructured Data.* Proceedings of the 27th VLDB Conference, Roma, Italy, 2001.

[Col03] R. Cole, R. Hariharan. *Faster Suffix Tree Construction with Missing Suffix Links.* SIAM Journal on Computing, vol 33, no 1, pp 26-42, 2003.

[EStat] EMBL Statistik: *http://www3.ebi.ac.uk/Services/DBStats*.

[Fal84] C. Faloutsos, S. Christodoulakis. *Signature Files: An Access Method for Documents and Its Analytical Performance Evaluation.* ACM TOIS 2(4), pp 276-288, 1984.

[Far97] M. Farach. *Optimal Suffix Tree Construction with Large Alphabets.* Foundations of Computer Science (FOCS'97), pp 137-143, 1997.

[Far98] M. Farach, P. Ferragina, S. Muthukrishnan. *Overcoming the memory bottleneck in suffix tree construction.* Proceedings of the 39th Annual Symposium on Foundations of Computer Sciences, pp 174-185. IEEE press, 1998.

[Fer99] P. Ferragina, R. Grossi. *The string B-tree: a new data structure for string search in external memory and its applications.* Journal of the ACM, 46(2): 236-280, 1999.

[Fer00] P. Ferragina, G. Manzini. *Opportunistic data structures with applications.* FOCS'00, pp 390-398, 2000.

[Gie03] R. Giegerich, S. Kurtz, J. Stoye. *Efficient Implementation of Lazy Suffix Trees.* Journal: Software – Practice and Experience, vol 33(11), pp 1035-1049, 2003.

[Gra02] L. Gravano, P.G. Ipeirotis, H.V. Jagadish. *Using q-grams in a DBMS for Approximate String Processing.* IEEE Data Engineering Bulletin, vol 24, no 4, December 2002.

[Gro99] R. Grossi, J.S. Vitter. *Compressed suffix array and suffix tree with applications to text indexing and string matching.* Annual ACM Symposium on Theory of Computing, pp 397-406, 1999.

[Gus97] D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge University Press, 1997.

[Har84] D. Harel, R.E.Tarjan. *Fast algorithms for finding nearest common ancestors.* SIAM Journal on Computing, vol 13, pp 338-355, 1984.

[Hon03] W.-K. Hon, K. Sadakane, W.-K. Sung. *Breaking a Time-and-Space Barrier in Constructing Full-Text Indices.* FOCS, pp 251-260, 2003.

[Hui92] L.C.K. Hui. *Color set size problem with application to string matching.* Proceedings of Combinatorial Pattern Matching, pp 230-243, Springer Verlag, 1992.

[Hun01] E. Hunt, R.W. Irving, M.P. Atkinson. *A database index to large biological sequences.* Proceedings of the 27th International Conference on Very Large Databases, pp 139-148, 2001.

[Hun02] E. Hunt, M.P. Atkinson, R.W. Irving. *Database indexing for large DNA and protein sequence collections.* VLDB Journal, 11: 256-271, 2002.

[Irv03] R.W. Irving, L. Love. *The suffix binary search tree and the AVL tree.* Journal of Discrete Algorithms, vol 1, issue 5-6, pp 387-408, 2003.

[Kär95] J. Kärkkäinen. *Suffix cactus: A cross between suffix tree and suffix array.* Proceedings of the Sixth Symposium on Combinatorial Pattern Matching (CPM '95), (eds. Z. Galil and E. Ukkonen), LNCS 937, Springer, pp 191-204, 1995.

[Kah01] T. Kahveci, A.K. Singh. *An Efficient Index Structure for String Databases.* VLDB, Roma, Italy, Sept 2001.

[Kah03] T. Kahveci, A.K. Singh. *MAP: Searching Large Genome Databases.* PSB, Kaua'i Hawaii, 2003.

[Ken03] G.C. Kennedy, H. Matsuzaki, et al. *Large-scale genotyping of complex DNA.* Nature Biotechnology, vol 21, no 10, pp 1233-1237, October 2003.

[Ken02] J. Kent. *BLAT - The BLAST-Like Alignment Tool.* Genome Research 12: 656-664, 2002.

[KMP77] D.E. Knuth, J.H. Morris, V.R. Pratt. *Fast Pattern Matching in Strings.* SIAM Journal of Computing, vol 6, no 2, pp 656-664, 1977.

[Kur99] S. Kurtz. *Reducing the space requirement of suffix trees.* Software – Practice and Experience, vol 29, pp 1149-1171, 1999.

[Lot82] M. Lothaire. *Combinatorics on Words.* Addison-Wesley, 1982.

[Man93] U. Manber, G. Myers. *Suffix Arrays: A new method for on-line string searches.* SIAM J. Computation, 22(5): 935-948, October 1993.

[MCr76] E. McCreight. *A space-economic suffix tree construction algorithm.* Journal of the ACM, 23(2): 262-272, 1976.

[Mon02] K. Monostori, A.B. Zaslavsky, H. Schmidt. *Suffix vector: space- and time-efficient alternative to suffix trees.* Conference in Research and Practice in Information Technology Series, Melbourne, pp 157-165, 2002.

[Nar00] G. Navarrov, R. Baeza-Yates. *A Hybrid Indexing Method for Approximate String Matching.* Journal of Discrete Algorithms, 1: 21-49, 2000.

[Nar01] G. Navarrov. *A Guided Tour to Approximate String Matching.* ACM Computing Surveys 33(1): 31-88, 2001.

[Nar02] G. Navarrov. *Indexing Text using the Ziv-Lempel Trie.* Proceedings SPIRE'02, pp 325-336, LNCS 2476, 2002.

[Nav04] G. Navarrov. *Text Databases.* In: L. Rivero, J. Doorn, V. Ferraggine (editors), Encyclopedia of Database Technologies and Applications, Idea Group Inc., Pennsylvania, USA, 2004.

[Pei03] J. Pei, R. Sadreyev, N.V. Grishin. *PCMA: fast and accurate multiple sequence alignment based on profile consistency.* Bioinformatics 19: 427-428, 2003. (*ftp://iole.swmed.edu/pub/PCMA/*)

[Sag00] L. Marsan, M.-F. Sagot. *Extracting structured motifs using a suffix tree - algorithms and application to promoter consensus identification.* RECOMB 2000, pp 210-219, 2000.

[Sad00] K. Sadakane. *Compressed text databases with efficient query algorithms based on the compressed suffix array.* ISAAC, LNCS 1969, pp 410-421, 2000.

[Sad02] K. Sadakane. *Succint representations of lcp information and improvements in the compressed suffix arrays.* In Proc. SODA, pp 225-232, 2002.

[SVi88] B. Schieber, U. Vishkin. *On finding lowest common ancestors: Simplification and parallelization.* SIAM Journal of Computing, 17: 1253-1262, 1988.

[Sch02] K.-B. Schürmann, B. Stoye. *Suffix Tree Construction for Large Strings.* Proceedings 14. Workshop of Fundamentals of Databases, Rostock, Germany, May 2002.

[Sch03] K.-B. Schürmann, J. Stoye. *Suffix tree construction and storage with limited main memory.* Technical Report 2003-06, Universität Bielefeld, Technische Fakultät, Abteilung Informationstechnik, 2003.

[Sil01] A. Silberschatz, H.F. Korth, S. Sudarshan. *Database System Concepts.* Fourth Edition, McGraw-Hill, 2001.

[Sta03] M. Stanke, S. Waack. *Gene prediction with a hidden Markov model and new intron submodel.* Bioinformatics, 19(Suppl. 2), pp 215-225, 2003

[Ukk95] E. Ukkonen. *On-line construction of suffix trees.* Algorithmica, 14(3): 249-260, 1995.

[Vitt93] J.S. Vitter, E.A.M. Shriver. *Algorithms for Parallel Memory: Two-Level Memories.* Technical Report DUKE–TR–1993–01.

[Vitt94] J.S. Vitter, E.A.M. Shriver. *Algorithms for Parallel Memory II: Hierarchical Multi-level Memories.* Algorithmica, vol 12(2-3), pp 148-169, 1994.

[Wei73] P. Weiner. *Linear pattern matching algorithm.* Proceedings of the 14th Annual Symposium on Foundations of Computer Science, pp 1-11, IEEE press, 1973.