Diplomarbeit

Untersuchungen von Suffixbäumen auf ihre Anwendbarkeit für lange Sequenzen in Genom-Datenbanken und im Information Retrieval

Klaus-Bernd Schürmann[1]

Freie Universität Berlin,
Fachbereich für Mathematik und Informatik

Betreuer: Prof. Dr. Heinz Schweppe, Zara Kanaeva,
Prof. Dr. Jens Stoye (Universität Bielefeld)

15. Juli 2002

[1]e-mail: shuerman@inf.fu-berlin.de

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Motivation

Seit der Erfindung des Buchdrucks durch Johannes Gutenberg (um 1450) bis zum heutigen Internet hat die Verbreitung und Speicherung von Informationen eine revolutionäre Veränderung erfahren. Stetige technologische Verbesserungen haben die Verbreitung des Wissens beschleunigt, wodurch die Informationsverarbeitung an Bedeutung gewonnen hat. Besonders mit der Entwicklung der Computertechnologie und der damit verbundenen Digitalisierung von Informationen, die heute über das Internet von jedem beliebigen Ort innerhalb von Sekunden abgerufen werden können, setzte ein starkes Wachstum der zu verarbeitenden Datenmengen ein. Beschränkte sich die digitale Datenhaltung bis Mitte der 80er Jahre noch weitgehend auf die Datenverwaltung in großen Unternehmen, so hat sie mittlerweile Einzug in alle Bereiche unseres Lebens gehalten. Das bringt, abgesehen von den Anforderungen in neueren Bereichen der Informatik (Bioinformatik, Geoinformatik, usw.), viele neue Herausforderungen an klassische Gebiete, wie das Information Retrieval, mit sich.
Die Genom-Forschung ist ein aktuelles Gebiet in der Molekularbiologie, das durch die Einführung effizienter Maschinen zur Sequenzierung von DNA ein exponentielles Wachstum an Genomdaten zu verzeichnen hat. In ähnlicher Weise ist im Information Retrieval das Internet für eine wachsende Zahl von Informationen (Zeitungsarchive, Wörterbücher, Enzyklopädien, html-Seiten, usw.) verantwortlich.
Trotz des ständigen Fortschritts in der Computertechnologie, mit der Entwicklung von schnelleren und größeren Speichereinheiten, gewinnt ein effizientes Speichermanagement immer mehr an Bedeutung, da das Wachstum des Datenumfangs den Anstieg der Speichergröße und -geschwindigkeit übersteigt.
Um sehr lange Strings wie DNA-Sequenzen effizient zu durchsuchen, ist es notwendig eine Indexstruktur auf dem Text zu haben. Es gibt einige wortbasierte Indexstrukturen, zum Beispiel invertierte Dateien [12] oder den String-B-Baum [20, 21, 47].

Da DNA-Sequenzen aber nicht in Worte aufgebrochen werden können, sind solche Indexstrukturen für dieses Anwendungsgebiet nicht geeignet. Ein den Suffixbäumen ähnlicher Ansatz, das Suffixarray [43], ist zwar nicht wortbasiert, benötigt aber für das Finden eines Musters w von t $O(|w| \times log\, n)$ Suchzeit. Der Suffixbaum ist für dieses Gebiet die geeignete Indexstruktur. Die Konstruktion und Speicherung eines Suffixbaumes von einem Text t benötigt $O(|t|)$ Zeit und Speicherplatz. Nachdem der Suffixbaum konstruiert wurde, ist es möglich das Vorkommen eines Teilstrings w von t in $O(|w|)$ Schritten (unabhängig von t) zu bestimmen.
Obwohl Suffixbäume ein in der Algorithmik viel behandeltes Thema mit vielen Anwendungen sind, finden sie weniger Gebrauch in der Praxis, als man erwarten würde. Die linearen Konstruktionsalgorithmen von Weiner, McCreight und Ukkonen zeigen zwar optimales asymptotisches Verhalten, aber ihr schlechtes Lokalitätsverhalten [10] führt auf heutigen Computerarchitekturen mit mehreren *Cacheebenen* zu einem Effizienzverlust. Wächst der Suffixbaum über die Grenzen des Hauptspeichers, so ist keine Konstruktion in angemessener Zeit mehr möglich. Zusammen mit den hohen Speicherplatzanforderungen der Linearzeitalgorithmen an die Suffixbaum-Repräsentation ist es nicht möglich, Suffixbäume für sehr lange Strings (das menschliche Genom enthält ungefähr $3,2 \times 10^9$ Basenpaare) aufzubauen.
Die Konstruktion von Suffixbäumen auf langen Strings kann mehrere Stunden in Anspruch nehmen. Daher ist es nicht angemessen, sie nur für einige wenige Suchoperationen zu konstruieren. Allerdings verändern sich die Strings der von mir betrachteten Anwendungsbereiche sehr selten, so dass durch eine persistente Speicherung der Suffixbäume keine wiederholte Konstruktion notwendig ist. Die Konstruktionskosten können sich deshalb über mehrere Suchen amortisieren, was somit in erster Linie vom Konstruktionsalgorithmus und einer effektiven, persistenten Speicherung abhängt.
Ideal sind effiziente Konstruktionsalgorithmen mit einer speichereffizienten Repräsentation, so dass die Suffixbäume nicht durch die Hauptspeichergröße limitiert sind. Ebenso sollten leistungsfähige persistene Speichermechanismen einen schnellen Zugriff auf die Suffixbäume erlauben.

1.2 Zielsetzung der Arbeit

Ziel der Diplomarbeit ist die Untersuchung von Suffixbäumen im Hinblick auf die Gebiete Genom-Datenbanken und Information Retrieval. Wie schon im vorherigen Abschnitt erwähnt, beschäftigt man sich in diesen Bereichen mit sehr großen Textmengen. Damit sich die Konstruktion von Suffixbäumen amortisiert, ist eine schnelle Konstruktion und eine effiziente, persistente Speicherung der Suffixbäume notwendig.
In Bezug auf die Konstruktion der Suffixbäume bilden zwei Algorithmen die Grund-

lage der Arbeit, die explizit das Lokalitätsverhalten bei der Konstruktion einbeziehen, was auf heutigen Rechnerarchitekturen eine entscheidende Rolle spielt. Dies sind ein auf einer Partitionierung des Suffixbaumes basierender Algorithmus von Hunt *et al.* [29] und der *wotd*(write-only-top-down)-Algorithmus in einer Implementierung von Kurtz *et al.* [23]. Diese Algorithmen sollen vorgestellt und bezüglich ihrer Anwendbarkeit für lange Sequenzen analysiert werden. Im Anschluss an die Analyse erfolgen weiterführende Überlegungen zur Verbesserung dieser Algorithmen.
Zur persistenten Speicherung der Suffixbäume sollen verschiedene Verfahren diskutiert und implementiert werden. Die Grundlage hierfür können einerseits spezielle Formate zur Speicherung von Suffixbäumen bilden. Andererseits ist es aber auch möglich, etablierte Hintergrundspeichermechanismen, wie relationale Datenbanken, zu verwenden. Insbesondere ist der Vergleich geeigneter Formate zur persistenten Speicherung von Suffixbäumen Ziel dieser Arbeit. Diese werden mit der Abbildung von Suffixbäumen auf relationale Datenbanken verglichen.
Nach den theoretischen Überlegungen bezüglich der Konstruktion und Speicherung von Suffixbäumen sollen anhand eigener Implementierungen experimentelle Untersuchungen durchgeführt werden. Diese experimentellen Untersuchungen umfassen sowohl die Konstruktionsalgorithmen, als auch die persistent gespeicherten Suffixbäume, auf denen die Laufzeit der Mustersuche überprüft werden soll.
Basierend auf den Ergebnissen der experimentellen Untersuchungen sollen abschließend die Eigenschaften der Konstruktionsalgorithmen und der persistenten Speichermechanismen diskutiert und Aussagen bezüglich ihrer praktischen Anwendung in den Bereichen Information Retrieval und Genom-Datenbanken getroffen werden.

Kapitel 2

Grundlagen

Bei der Indizierung von Sequenzen ist zwischen wortbasierten Indexstrukturen und sequenzbasierten Indexstrukturen zu unterscheiden. Wortbasierte Indexstrukturen, wie zum Beispiel invertierte Dateien, beruhen auf der Zerlegung des Textes in einzelne Worte. Sequenzbasierte Indexstrukturen sind dagegen unabhängig von dieser Wortstruktur, denn sie können auch Texte wie DNA-Sequenzen indizieren, die nicht in Worte zerlegbar sind. Solche sequenzbasierten Indexstrukturen sind beispielsweise Suffixarrays [14, 26, 39, 43], *Suffix-Binary-Search-Trees* [28, 31] oder die in der Arbeit behandelten Suffixbäume [2, 27], die eine in der Sequenzverarbeitung etablierte Datenstruktur sind.
In diesem Kapitel wird die in dieser Arbeit verwendete Notation definiert, die Suffixbaum-Datenstruktur eingeführt und Operationen auf Suffixbäumen beschrieben.

2.1 Notation

Um die Notation über die gesamte Arbeit konsistent zu halten, sind hier einige grundlegende Definitionen gegeben.

- Σ sei ein endliches Alphabet.
- \$ sei ein ausgezeichnetes Zeichen, $\$ \notin \Sigma$.
- $t \in \Sigma^n$ sei ein Text der Länge $n = |t|$, $n \in \mathbb{N}$.
- $t^+ := t\$$ sei der Text t, erweitert um das ausgezeichnete Zeichen \$ ($\notin \Sigma$).
- $t_q := t[q]$ sei das Zeichen an der Stelle q von t, $1 \leq q \leq |t|$.
- $t[a, b] = t[(a, b)] := t_a t_{a+1} \ldots t_b$ sei der Teilstring von t, der von der Stelle a zur Stelle b reicht, $1 \leq a \leq b \leq |t|$.

- $t_a \oplus t_b := t_a t_b = t_{a_1} \dots t_{a_{|t_a|}} t_{b_1} \dots t_{b_{|t_b|}}$ sei die Konkatenation der Texte t_a und t_b.
- p ist nicht-leeres Präfix von t

$$\begin{aligned} &:\Longleftrightarrow \quad p \sqsubseteq t \\ &:\Longleftrightarrow \quad \exists q,\, 1 \leq q \leq |t| :\; p = t[1,q] \end{aligned}$$

- s ist nicht-leeres Suffix von t

$$\begin{aligned} &:\Longleftrightarrow \quad p \sqsupseteq t \\ &:\Longleftrightarrow \quad \exists q,\, 1 \leq q \leq |t| :\; s = t[q,|t|] \end{aligned}$$

- $s_i(t) \sqsupseteq t$, $s_i(t) := t[i,|t|]$ heißt i-tes Suffix von t
- Ein String w heißt Muster oder Teilstring von t, genau dann wenn:

$$\exists\, j,\, 1 \leq j \leq |t| - |w| + 1 \;\; : t\,[i\,,\; i + |w| - 1] = w$$

- Das längste gemeinsame Präfix lcp von zwei Texten t_1 und t_2 ist definiert als:

$$lcp(t_1, t_2) := t_1[1,j]\,;\; mit\; j = max\{k \in \mathbb{N} |\, t_1[1,k] = t_2[1,k]\}$$

2.2 Suffixbäume

Ein Suffixbaum ist eine speichereffiziente Datenstruktur zur Indizierung aller Suffixe eines gegebenen Textes t. Er entsteht durch die Durchführung einiger Komprimierungsschritte aus einem Suffix-Trie. Diese Komprimierungen und die Eigenschaften des Suffixbaumes werden im Folgenden erläutert.

Suffix-Trie

Tries sind eine auf Bäumen basierende Datenstruktur, um Worte (Strings) zu speichern. Der Name kommt aus einem ihrer Hauptanwendungsgebiete, dem Re**trie**val von Informationen. Sie erlauben das schnelle Finden von Informationen durch Präfixsuche. Ein Trie ist ein Wurzelbaum, der eine Menge von Texten enthält. Jede Kante des Baumes ist mit genau einem Zeichen beschriftet. Jedes Wort aus der enthaltenen Menge von Texten entspricht dem Ablaufen eines Weges von der Wurzel zu einem Blatt bei gleichzeitigem Konkatenieren aller Zeichen auf den Kanten. Damit diese Eigenschaft nicht verletzt wird, ist jedem Text t ein Zeichen `$`, das nicht im Alphabet Σ vorkommt, anzuhängen. Die Textmenge ergibt sich dabei durch das

Ablaufen aller Wege von der Wurzel zu den Blättern.
Ein Suffix-Trie enthält keine unabhängigen Wörter, sondern alle Suffixe zu einem gegeben Text t. Ein Beispiel für einen Suffix-Trie über dem Text `CATATACTA$` ist in Abbildung 2.1 dargestellt. In den Blättern wird die Nummer des entsprechenden Suffixes gehalten. Um den Suffix `TA` mitsamt seiner Erweiterung `$` zu bekommen, ist dem Weg von der Wurzel zu dem Blatt mit der Nummer 8 zu folgen. Durch Verzicht auf die Erweiterung `$` würde der 8-te Suffix nicht in einem Blatt enden, was die bijektive Beziehung zwischen Blättern und Suffixen verletzen würde.
Der Suffix-Trie hat schon die Funktionalität eines Suffixbaumes. Das Problem ist seine hohe Speicherplatzkomplexität von $O(|t|^2)$. Für sehr lange Sequenzen sind somit die Speicherplatzanforderungen von Suffix-Tries zu groß.

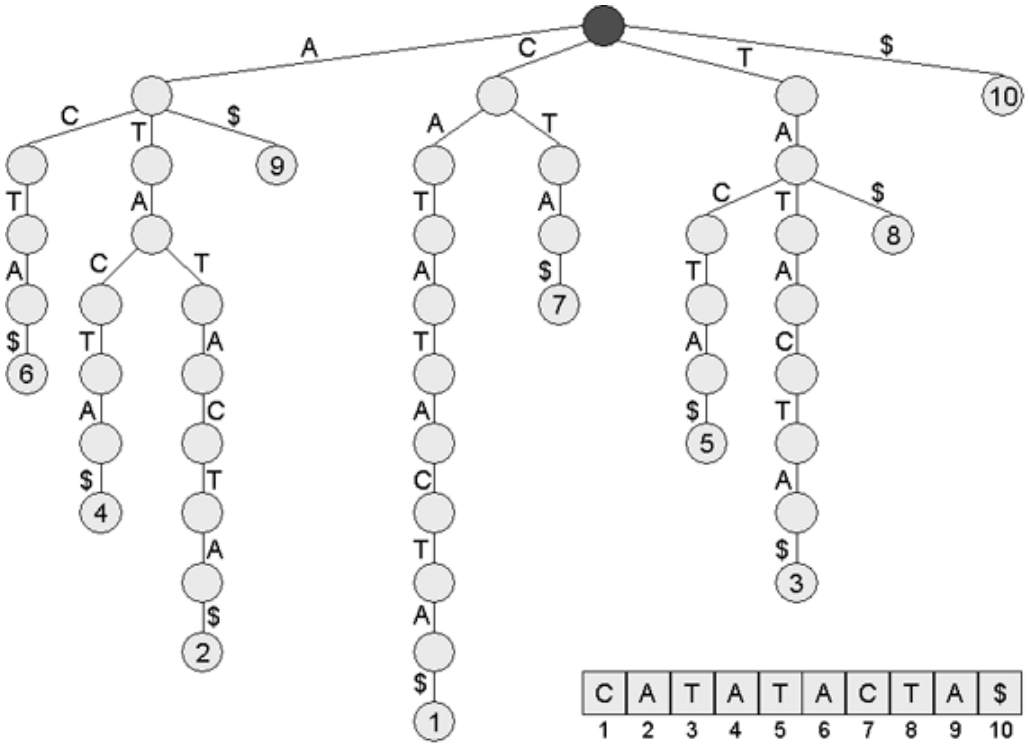


Abbildung 2.1: Suffix-Trie

Compacted-Suffix-Trie

Um die Speicherplatzanforderungen zu reduzieren, wird nun versucht überflüssige Knoten zu eliminieren. Dies geschieht, indem nicht verzweigende Knoten zusammengefasst werden. Die Kantenbeschriftung ergibt sich jetzt aus der Konkatenation der Kantenbeschriftungen, der zusammengefassten Kanten. Beispielsweise werden

die beiden Buchstaben T und A, aus dem gezeigten Beispiel, zum String `TA` zusammengefasst. Die Anzahl der inneren Knoten ist nun durch die Anzahl der Blätter beschränkt, da jeder innere Knoten des resultierenden Compacted-Suffix-Tries jetzt mindestens 2 Kinder hat. Die Anzahl der Blätter im Suffixbaum entspricht genau der Länge des Textes t^+. Damit enthält der Compacted-Suffix-Trie genau $n + 1$ Blätter und maximal n innere Knoten. Die Speicherplatzanforderungen für die Knoten des Compacted-Suffix-Tries liegen somit in $O(n)$. Der Gesamtspeicherplatzbedarf liegt aber immer noch in $O(n^2)$, da die Anzahl der Zeichen auf den Kanten die gleiche geblieben ist wie beim Suffix-Trie.

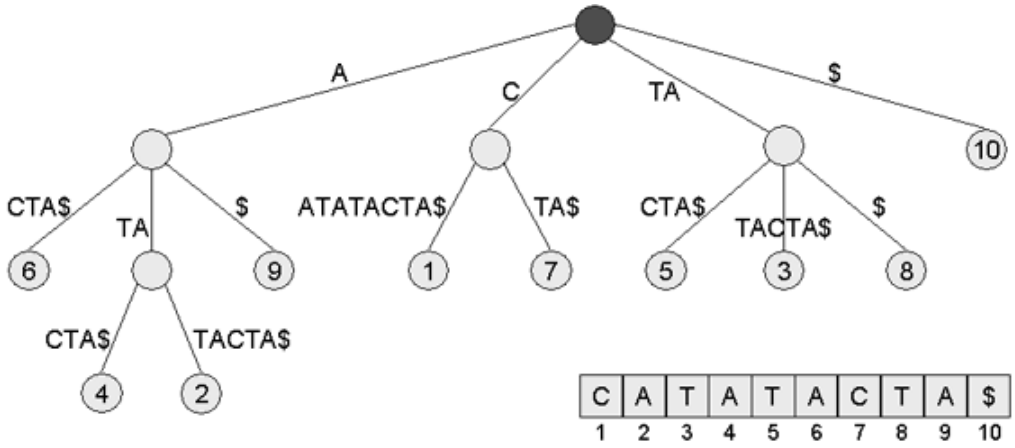


Abbildung 2.2: Compacted-Suffix-Trie

Suffixbaum

Der letzte Schritt zu einem Suffixbaum ist die Komprimierung der Informationen auf den Kanten. Eine Möglichkeit hierfür wird im Patricia-Trie gewählt, der nur die minimalen Verzweigungsinformationen, in Form des ersten Zeichens der Kantenbeschriftung, enthält. Diese Komprimierung ist allerdings verlustbehaftet. Die Worte sind zwar unterscheidbar, um aber an die kompletten Informationen über ein Wort zu kommen, muss man den Patricia-Trie bis zum entsprechenden Blatt durchlaufen. Die Komprimierung beim Suffixbaum ist im Gegensatz zum Patricia-Trie nicht verlustbehaftet. Hier werden die Kantenbeschriftungen, die immer einem Teilwort vom Text t^+ entsprechen, durch den Verweis auf den Anfangs- und Endindex dieses Teilwortes ersetzt. In meinem Beispiel entspricht das Wort `TA` dem Teilstring $t^+[3, 4]$. `TA` wird also durch den Anfangsindex 3 und den Endindex 4 substituiert. Die Informationen auf jeder Kante in einem Suffixbaum belaufen sich jetzt nur noch auf den konstanten Speicherplatzaufwand für den Anfangs- und Endindex. Es ergibt sich jetzt auch für den gesamten Suffixbaum ein linearer Speicherplatzaufwand in

n, weil die Anzahl der Kanten in einem Baum durch die Anzahl der $O(n)$ Knoten beschränkt ist.
Ein Suffixbaum ist daher als ein komprimierter Trie anzusehen, der alle Suffixe eines gegebenen Textes t indiziert, wobei jeder innere Knoten eines Suffixbaumes mindestens zwei Kinder hat und die Kanten einen Teilstring von s referenzieren. Dabei enthält jedes Blatt den Schlüssel eines bestimmten Suffixes und die Konkatenation der über die Kanten referenzierten Teilstrings auf dem Weg von der Wurzel ρ zu einem Blatt l ergibt den durch die Beschriftung im Blatt gegebenen Suffix.

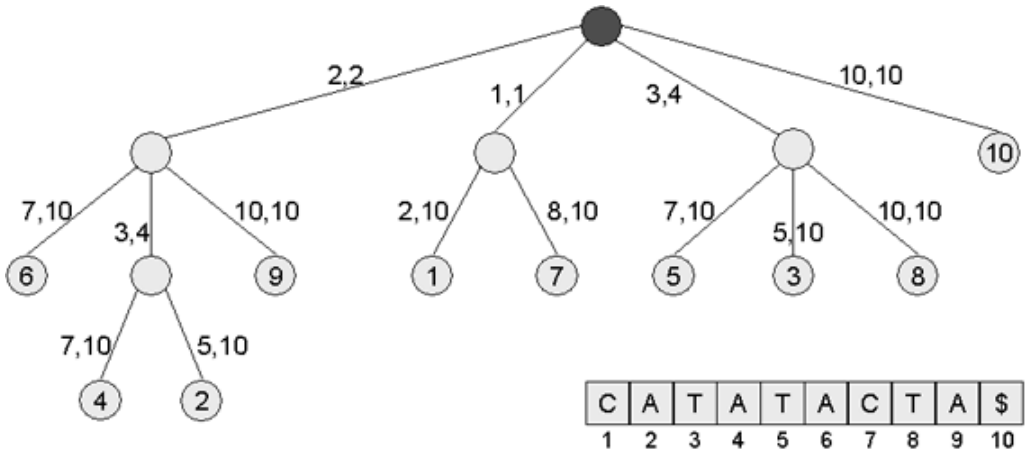


Abbildung 2.3: Suffixbaum

Leider gibt es in den bekannten Arbeiten nur informelle Definitionen von Suffixbäumen. Um sich hier auf ein mathematisch vollständiges Modell eines Suffixbaumes stützen zu können, sind von mir einige Definitionen gegeben.

Definition 1 (Suffixbaum) *Ein Suffixbaum* $\mathcal{T}(t) = (t^+, I, L, \overrightarrow{E}, \rho, \lambda, \omega)$ *über dem String* t *ist ein Baum mit folgenden Eigenschaften:*

- I *ist Menge von inneren Knoten.*
- L *ist Menge von Blättern mit* $|L| = n + 1$.
- $\overrightarrow{E} \subset (I \times (I \cup L))$ *ist Menge von gerichteten Kanten.*
- $\rho \in I$ *ist ausgezeichnete Wurzel.*
- $\lambda : L \to \{x \in \mathbb{N} \mid 1 \leq x \leq |t^+|\}$ *bijektiv, ist Nummerierung der Suffixe in den Blättern.*
- $\omega : \overrightarrow{E} \to \{(x, y) \in \mathbb{N} \times \mathbb{N} \mid 1 \leq x \leq y \leq n + 1\}$ *ist Beschriftung der Kanten mit dem linken und rechten Index des Intervalls auf* t^+
- $\forall i \in I : |\{(x, y) \in \overrightarrow{E} \mid x = i\}| \geq 2$ *(jeder Knoten verzweigt).*

- *Jede ausgehenden Kante eines Knotens beginnt mit einem anderen Zeichen*

$$\forall i \in I \, \forall e, e' \in \{(a,b) \in \overrightarrow{E} \mid a = i\} : \\ ((x,y) = \omega(e) \wedge (x',y') = \omega(e') \wedge t[x] = t[x']) \Longrightarrow e = e'.$$

- $\forall \mathbf{p} = p_1 \ldots p_j;\ p_1, \ldots p_{j-1} \in I,\ p_j \in L;\ p_1 = \rho$, $\mathbf{p}$ *ist der eindeutige Weg von der Wurzel zu einem Blatt:*

$$\bigoplus_{i=1}^{j-1} t^+[\omega(p_i, p_{i+1})] = t^+[\lambda(l_k), n+1] = s_i(t^+).$$

Weiterhin soll gelten, dass $V = I \cup L$ die Menge aller Knoten im Suffixbaum ist. Die Tiefe eines Knotens k im Suffixbaum sollte, anders als in anderen Bäumen, nicht über die Anzahl der abgelaufenen Kanten von der Wurzel zu k definiert sein, sondern über die Längen der Kantenintervalle von der Wurzel ρ zum Knoten k.

Definition 2 (Tiefe eines Knotens) *Die Tiefe $depth(k)$ eines Knotens $k \in V(\mathcal{T}(t))$ im Suffixbaum sei definiert als:*

$$\begin{aligned} depth(k) \quad := \quad & \sum_{i=1}^{j-1} ((x_{i+1} - x_i) + 1), \\ & \text{wobei } \mathbf{p} = p_1 \ldots p_j;\ p_1, \ldots p_j \in I(\mathcal{T}(t)),\ p_j \in V(\mathcal{T}(t));\ p_1 = \rho, \\ & \text{Weg von } \rho \text{ zu kund } (x_i, x_{i+1}) = \omega(p_i, p_{i+1}). \end{aligned}$$

Entsprechend der Tiefe des Suffixbaumes definiert sich auch die Höhe über die maximale Tiefe der inneren Knoten.

Definition 3 (Höhe eines Suffixbaumes) *Sei $\mathcal{T}(t)$ ein Suffixbaum, dann ist die Höhe $(\mathcal{T}(t))$ dieses Baumes height definiert als:*

$$height(\mathcal{T}(t)) \quad := \quad Max\{depth(k) \mid k \in I(\mathcal{T}(t))\}.$$

Der Weg von der Wurzel ρ zu einem Knoten k entspricht einem Teilwort w von t^+. Jeder Knoten $k \in \mathcal{T}(t)$ repräsentiert ein Teilwort von t.

Definition 4 *Sei $k \in V(\mathcal{T}(t))$ und $\mathbf{p} = p_1 \ldots p_j;\ \rho = p_1,\ k = p_j,$ $p_1, \ldots, p_j \in V(\mathcal{T}(t))$, der Weg von der Wurzel zu k, dann bezeichne ich k als $\overline{w}$, wenn das Wort w dem Weg von der Wurzel zum Knoten k entspricht*

$$\overline{w} := k \quad \Longleftarrow \quad w = \bigoplus_{i=1}^{j-1} t^+[\omega(p_i, p_{i+1})].$$

Endet das Wort w beim Durchlaufen des Suffixbaumes nicht in einem Knoten, sondern auf einer Kante, so ist $\overline{w}$ der Knoten in dem die Kante endet

$$\overline{w} = k \Longleftarrow \quad w = uv \quad mit\ u = \bigoplus_{i=1}^{j-2} t^+[\omega(p_i, p_{i+1})] \quad und\ vz = t^+[\omega(p_{j-1}, p_j)].$$

Die Nachkommen eines Knotens $\overline{w}$ repräsentieren Teilworte von t, die mit Präfix w beginnen. Entsprechend repräsentieren die Blätter in dieser Menge alle Suffixe mit Präfix w.

Definition 5 (Blattmenge eines Knotens) *Die Blattmenge $leafSet(k)$ eines Knotens $k \in (I(\mathcal{T}(t)) \cup L(\mathcal{T}(t)))$ ist wie folgt definiert:*

$$\begin{aligned} leafSet(\overline{s}) &= \lambda(\overline{s}) && ;\ \overline{s} \in L(\mathcal{T}(t)) \\ leafSet(\overline{w}) &= \bigcup_{(u,v) \in \{(x,y) \in \vec{E} \mid x = \overline{w}\}} leafSet(v) && ;\ \overline{w} \in I(\mathcal{T}(t)). \end{aligned}$$

2.3 Operationen

Suffixbäume bilden die Grundlage von vielen Anwendungen auf Sequenzen. Eine ausführliche Einführung über den Einsatz von Suffixbäumen, insbesondere für biologische Anwendungen, findet man in einem Buch von Gusfield [27].

2.3.1 Mustersuche

Es gibt verschiedene Algorithmen für die Suche eines exakten Teilwortes w in einem Text t. Die Algorithmen von Boyer-Moore [13] bzw. Knuth, Morris und Pratt [35] finden Anwendung, wenn das Suchmuster w vorzeitig bekannt ist. Sie finden das Vorkommen von w im Text t in $O(|t|)$ Laufzeit. Für den Fall, dass t bekannt ist oder viele Muster auf t gesucht werden, bietet die Konstruktion eines Suffixbaumes die beste Möglichkeit der Vorverarbeitung.
Die Mustersuche ist die klassische Operation auf Suffixbäumen. Nach der Konstruktion des Suffixbaumes ist es möglich in $O(|w|)$ Schritten, unabhängig vom Text t, das Vorkommen des Musters w in t zu bestimmen. Der Algorithmus beruht dabei auf der Tatsache, dass jedes Muster w in t immer auch ein Präfix eines Suffixes s von t ist ($s \sqsupseteq t$, $w \sqsubseteq s$). Die Zeichen von w entlang des eindeutigen Weges im Suffixbaum $\mathcal{T}(t)$ sind so lange zu vergleichen, bis das ganze Muster w durchlaufen oder bis keine weitere Übereinstimmung mehr möglich ist. Im ersten Fall werden die Anfangspositionen aller Vorkommen von w durch $leafSet(\overline{w})$ nummeriert, anderenfalls existiert das Muster w nirgendwo in t.

Die Mustersuche auf Suffixbäumen findet vielfach Anwendung in der Genom-Analyse, wo häufig das Vorkommen eines Musters w in t überprüft wird.

2.3.2 Andere Anwendungen

Abgesehen von der exakten Mustersuche zeigen Suffixbäume ihren großen Wert auch in vielen anderen Anwendungen in der Stringverarbeitung.
Um weniger exakte Beziehungen zwischen zwei Strings w und t zu formalisieren, gibt es die Edit-Distanz oder allgemein die Ähnlichkeit von Strings. Die Edit-Distanz von zwei Strings ist die minimale Anzahl von Edit-Operationen, um die Strings ineinander zu überführen. Der Ähnlichkeitsbegriff kann in verschiedenen Anwendungen viele unterschiedliche Bedeutungen haben. In natürlichsprachigen Texten können Worte beispielsweise als ähnlich angesehen werden, wenn sie synonym sind. Dagegen wird Ähnlichkeit in der Biologie, bezüglich Proteinen, üblicherweise mit Hilfe von sogenannten Score-Matrizen definiert, welche die Ähnlichkeiten von Aminosäuren beschreiben. Suffixbäume sind ein häufig diskutiertes Hilfsmittel in diesem Bereich [27]. Sie werden zum Beispiel in Hybriddatenstrukturen [7] zur Suche von Ähnlichkeiten eingesetzt, um als Teilproblem die exakte Suche zu realisieren. Eine andere Möglichkeit ist, zuerst ähnliche Strings zu einem Muster w zu bestimmen und dann eine Mustersuche durchzuführen.
BLAST (basic local alignment search tool) [1], ein für Biologen wichtiges Werkzeug zur Erforschung von Proteinen und DNA-Sequenzen, arbeitet noch ohne eine Indizierung des Textes. Ähnlichkeiten werden durch serielles Scannen der Sequenz gefunden. Bei der in den letzten Jahren stark zunehmenden Datenmenge ist es mittlerweile nur durch Aufbau von Rechnerfarmen mit starker CPU-Leistung möglich, diese Aufgabe zu meistern. Der Einsatz von Suffixbäumen könnte hier zu einer erheblichen Effizienzsteigerung führen.
Weitere Anwendungen von Suffixbäumen sind das Finden von sich wiederholdenden Teilstrings eines Textes [38], das Finden eines längsten Teilstrings zweier Sequenzen t_1 und t_2 [16], die Datenkomprimierung [9] und vieles mehr [17, 19, 25, 27, 36].

Kapitel 3

Konstruktionsalgorithmen für Suffixbäume

Der erste Linearzeitalgorithmus zur Konstruktion von Suffixbäumen wurde 1973 von Weiner [53] vorgestellt. McCreight [44] veröffentlichte ein paar Jahre später (1976) einen speichereffizienteren Algorithmus zur Konstruktion von Suffixbäumen in linearer Zeit. Danach vergingen fast 20 Jahre bis Ukkonen [51] 1995 einen Online-Algorithmus zum Aufbau von Suffixbäumen entwarf. Dieser Algorithmus hat eine sehr viel einfacher zu verstehende Struktur und dabei alle Vorteile des McCreight-Algorithmus. Auf geeignete Weise betrachtet, kann er sogar als Variante des Algorithmus von McCreight angesehen werden [22]. Als letzte Entwicklung stellte Farach 1997 [18] einen konzeptionell anderen Konstruktionsalgorithmus zum optimalen Aufbau von Suffixbäumen mit großen Alphabeten vor.
Obwohl mehr als 25 Jahre vergangen sind seit Weiner den ersten Linearzeitalgorithmus zur Konstruktion von Suffixbäumen entwickelt hat und Suffixbäume sowie verwandte Strukturen, wie Affix-Bäume [49, 42], ein in der Wissenschaft viel behandeltes Thema mit sehr vielen Anwendungen sind, finden sie weniger Gebrauch in der Praxis, als man nach den umfangreichen theoretischen Erkenntnissen erwarten würde. Skiena [48] hat festgestellt, dass der Suffixbaum die Datenstruktur mit der höchsten Anforderung für bessere Implementierungen ist.
Das hat mehrere Gründe. So haben die Linearzeitalgorithmen den Ruf äußerst kompliziert zu sein, obwohl sie sicher nicht schwieriger zu verstehen sind als einige andere, vielfach gelehrte Algorithmen und bei geeigneter Implementierung praktische Lösungen zu vielen String-Problemen bieten.
Andere Gründe sind die hohen Speicherplatzanforderungen der Linearzeitalgorithmen. Die speichereffizienteste Repräsentation von McCreights Algorithmus nach Kurtz [37] benötigt im mittleren Fall 10,1 Bytes pro indizierten Buchstaben. Im schlechtesten Fall liegt der Speicherbedarf allerdings immer noch bei 20 Bytes pro Zeichen. Diese hohen Speicherplatzanforderungen sind unter anderem zurückzufüh-

ren auf die Benutzung von Suffixlinks, die notwendig für die Konstruktion der Suffixbäume in linearer Zeit sind. Die Suffixlinks sind Verweise zwischen den Knoten des Suffixbaumes, die Abhängigkeiten zwischen bestimmten Suffixen im Suffixbaum auf geeignete Weise darstellen, um bei der Konstruktion genutzt zu werden.
Außerdem hat die Benutzung von Suffixlinks wahlfreie Speicherzugriffe auf den Suffixbaum während der Konstruktion zur Folge. Das Lokalitätsverhalten von Suffixbäumen wurde von Behrens [10] in einer Diplomarbeit untersucht. Dabei weisen die Algorithmen von McCreight und Ukkonen eine hohe Anzahl von *Cache-Misses* auf. Bei längeren Sequenzen mit mittlerer Alphabetgröße beträgt der Anteil der *Cache-Misses*, bezogen auf die Datenzugriffe, zwischen 30 und 35%.
Dieses schlechte Lokalitätsverhalten beschränkt die Größe der Suffixbäume leider auf die Größe des verfügbaren Hauptspeichers. Zusammen mit den hohen Speicherplatzanforderungen ist es daher nicht möglich, Suffixbäume für sehr lange Strings aufzubauen.
Wünschenswert wären Konstruktionsalgorithmen zum effizienten Aufbau von Suffixbäumen ohne Hauptspeicherbeschränkungen, um Suffixbäume auch für lange Texte anwendbar zu machen. Algorithmen, die diesen Anforderungen genügen, sollten ein gutes Lokalitätsverhalten aufweisen, speichereffiziente Repräsentationen der Suffixbaum-Struktur erlauben und den Hintergrundspeicher mit in die Konstruktion einbeziehen.
Die folgenden Abschnitte basieren auf Algorithmen, die in diese Richtung gehen. Sie vermeiden Suffixlinks und opfern damit optimale lineare Laufzeit zugunsten eines besseren Lokalitätsverhaltens.
Der Partitionierungsalgorithmus von Hunt *et al.* [29] wird in Kapitel 3.1 beschrieben und analysiert. Zusätzlich wird eine ähnliche Struktur, der Hashed-Position-Tree von Heumann *et al.*, besprochen. In Kapitel 3.2 werden von mir entwickelte Verbesserungen des Partitionierungsalgorithmus erläutert und ausgewertet.
Das Kapitel 3.3 beschäftigt sich mit dem *wotd*(write-only-top-down)-Algorithmus, dessen Implementierung von Kurtz *et al.* als eine der effizientesten zur Konstruktion von Suffixbäumen gilt. Die Effizienz des *wotd*-Algorithmus hängt von der schnellen Sortierung der Suffixe ab. In Kapitel 3.4 werden unterschiedliche Sortieralgorithmen beschrieben, die innerhalb des *wotd*-Algorithmus verwendet werden können. Auf speicherplatzeffiziente Repräsentationen der Suffixbäume wird dann in Kapitel 4 genauer eingegangen.

3.1 Der Partitionierungsalgorithmus

Der Partitionierungsalgorithmus wurde 2001 von Hunt *et al.* [29] vorgestellt. Er basiert auf einer Einteilung der Suffixe in verschiedene Partitionen, bei der Suffixe mit gleichem Präfix in die selbe Partition fallen.

Bei den Linearzeitalgorithmen ist es nicht möglich, Teile des Suffixbaumes unabhängig aufzubauen, da diese verschiedene Zweige innerhalb des Baumes durch Suffixlinks verbinden. Damit existiert kein eindeutiger Weg mehr von der Wurzel zu jedem Blatt, womit die Baumstruktur aufgehoben wird. Gibt man die Suffixlinks aber auf, so zerlegt das Zerschneiden einer Kante den Suffixbaum in zwei disjunkte Zweige. Diese Eigenschaft macht sich der Partitionierungsalgorithmus zunutze, um Teile des Suffixbaumes $\mathcal{T}(t)$ unabhängig voneinander aufzubauen, so dass nicht mehr der gesamte Baum $\mathcal{T}(t)$ im Hauptspeicher gehalten werden muss. Schneidet man ihn auf einer bestimmten Tiefe d ab, so repräsentiert ein abgetrennter Zweig $\widehat{p_w}$ alle Suffixe $\{s_i(t^+) \mid wx = s_i(t^+), 0 \leq i \leq |t^+|, x \in (\Sigma \cup \{\$\})^*\}$ mit Präfix w.
Abbildung 3.1 beschreibt einen Suffixbaum $\mathcal{T}(CATATACTA\$)$, der auf Ebene $d = 1$ abgeschnitten wurde. Man sieht vier Zweige, deren enthaltene Suffixe mit gleichem Präfix A, C, T bzw. \$ beginnen. Um Zweige des Suffixbaumes $\mathcal{T}(t)$ nun un-

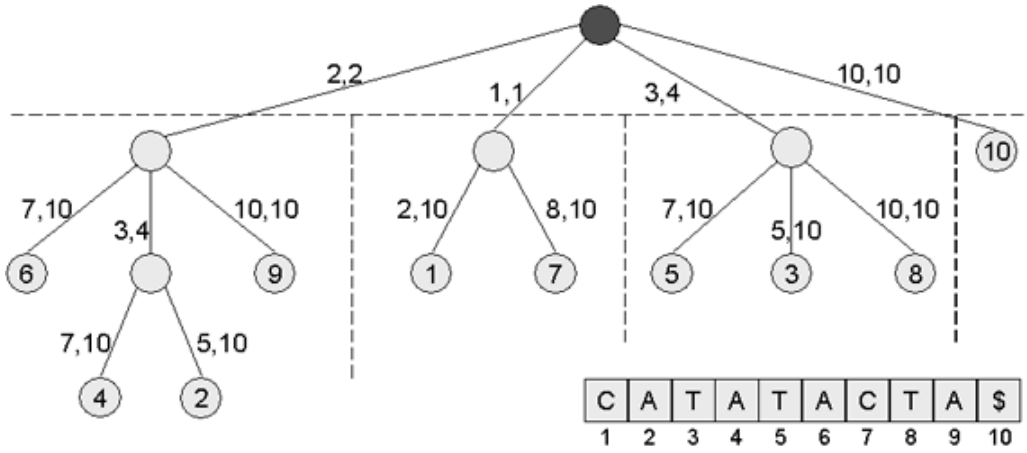


Abbildung 3.1: Partitionierter Suffixbaum

abhängig voneinander zu konstruieren, werden die Suffixe entsprechend ihrem Präfix w auf disjunkte Partitionen p_w abgebildet. Jeder Präfix w der Länge d entspricht einem eindeutigen Hashwert. Dazu wird das zugrundeliegende Alphabet Σ des Strings t mittels einer Funktion *code* ($code : \Sigma \to \{n \in \mathbb{N} \mid 0 \leq n < |\Sigma|\}$) bijektiv auf die Zahlen 0 bis $|\Sigma| - 1$ abgebildet . Der Hashwert $h_d(i)$, des i-ten Suffixes $s_i(t^+)$ ausreichender Länge, ergibt sich entsprechend seinem Präfix w folgendermaßen:

$$\begin{aligned} h_d: \quad & \{i \in \mathbb{N} \mid 1 \leq i \leq (|t^+| - d)\} \longrightarrow \{j \in \mathbb{N}_0 \mid 0 \leq j < \Sigma^d\} \\ h_d(i) \quad & = \sum_{j=0}^{d-1} \left(code(t^+[i+j]) \times |\Sigma|^{d-j-1}\right) \end{aligned}$$

Bei einer Partitionstiefe d induziert diese Hash- bzw. Präfixfunktion $|\Sigma|^d$ disjunkte Partitionen

$$\begin{aligned} P &= \{p_{w_j} \subset \mathbb{N} \mid p_{w_j} = h_d^{-1}(j),\ 0 \leq j < |\Sigma|^d\} \\ &= \{p_{w_j} \subset \mathbb{N} \mid p_{w_j} = leafSet(\overline{w_j}),\ 0 \leq j < |\Sigma|^d\} \end{aligned}$$

auf der Menge aller Suffixe. Ich bezeichne ein Wort w der Länge d als w_j, wenn die Suffixe, deren Präfix w ist, durch h_d auf j abgebildet werden.
Die Partitionstiefe d muss ausreichend groß gewählt werden, um den zu jeder Partition $p_w \in P$ entsprechenden Suffixbaum-Zweig $\widehat{p_w}$ komplett im Hauptspeicher konstruieren zu können. Wird dabei von einer zufälligen Verteilung des Textes ausgegangen, werden mindestens $\lceil \frac{\mathcal{S}_n}{M} \rceil$ Partitionen benötigt, wobei M die zur Verfügung stehende Hauptspeichergröße und $\mathcal{S}_n$ die Größe einer minimalen Hauptspeicherinstanziierung eines Suffixbaumes über einem Text der Länge n ist.

3.1.1 Algorithmus

Der Partitionierungsalgorithmus ist in Abbildung 3.2 dargestellt. Die Schleifen durchlaufen für jede Partitionen (Abb. 3.2, Zeile 1) $p_{w_j} \in P$ alle Suffixe (Abb. 3.2, Zeile 2) und fügen den i-ten Suffix $s_i(t^+)$ in $\widehat{p_{w_j}}$ ein (Abb. 3.2, Zeile 4), falls er einen entsprechenden Präfix w_j hat (Abb. 3.2, Zeile 3). Nach jedem Durchlauf aller Suffixe und dem damit verbundenen Aufbau eines Teilbaumes $\widehat{p_w}$ übernimmt das Speichermanagement der verwendeten Plattform die persistente Speicherung von $\widehat{p_w}$ (Abb. 3.2, Zeile 7). Hunt *et al.* [29] benutzen zu diesem Zweck PJama [46] eine Java basierte Plattform, die den Persistenzmechanismus in die Programmiersprache integriert.

BuildPartitionSuffixTree(t, d)

```
1: for all p_{w_j} ∈ P do
2:    for all i ∈ {i ∈ ℕ| 1 ≤ i ≤ n + 1} do
3:       if i ∈ p_{w_j} ; ( h_d(i) = j ) then
4:          insertSuffix(s_i(t^+))
5:       end if
6:    end for
7:    checkpoint
8: end for
```

Abbildung 3.2: Partitionierungsalgorithmus von Hunt *et al.*

Um den i-ten Suffix $s_i(t^+)$ einzufügen, wird, ähnlich zur Mustersuche, die Einfügestelle gesucht. Beginnend an der Wurzel ρ wird der Suffix entlang des eindeutigen

Weges im schon aufgebauten Teil des Suffixbaumes verglichen, bis keine weitere Übereinstimmung mehr möglich ist. Ausgehend von diesem Punkt wird eine neue Kante k ($k = (k_1, k_2)$ *mit* $k_1 \in I(\mathcal{T}(t))$, $k_2 \in L(\mathcal{T}(t))$) in den Suffixbaum eingefügt. Die Kantenbeschriftung $\omega(k) = (i + m - 1, n + 1)$ von k ist ein Intervall, was auf den Suffix $s_{i+m-1}(t^+)$ verweist. Dabei ist m die Stelle von $s_i(t^+)$ für die keine Übereinstimmung mehr im Baum gefunden wurde. Falls sich die Einfügestelle auf einer Kante befindet, so wird diese gesplittet und das durch sie referenzierte Intervall von t^+ entsprechend geteilt.

3.1.2 Eigenschaften

Die entsprechenden Zweige des Suffixbaumes können unabhängig konstruiert und gespeichert werden, da Partitionen disjunkt sind. Im Gegensatz zu den Linearzeitalgorithmen, bei denen die Suffixbaum-Größe durch die Hauptspeichergröße beschränkt ist, kann diese Vorgehensweise auch auf lange Texte angewandt werden. Hunt *et al.* [29] haben gezeigt, dass der Algorithmus selbst für DNA-Sequenzen mittlerer Länge eine vergleichbare Laufzeit aufweist, was auf das gute Lokalitätsverhalten zurückzuführen ist.
Bezüglich des Partitionierungsalgorithmus ist die pseudo-zufällige Verteilung der Präfixe der Grund für die Unabhängigkeit von der Hauptspeichergröße. Leider ist er nicht stabil bezüglich Entartungen, denn falls das häufige Vorkommen eines Präfixes zu einem Ungleichgewicht in der Partitionsfüllung führt, so dass ein Zweig die Hauptspeichergröße übersteigt, scheitert der Algorithmus. Auch die Analyse der erwarteten Laufzeit des Algorithmus zeigt einen weiteren Nachteil.

Analyse 1 *Sei $n = |t|$ die Länge des Eingabetextes t und $T_{PartSft}$ die Laufzeit des Partitionierungsalgorithmus, dann gilt:*

$$T_{PartSft}(n) \quad \in \quad \Theta(n^2) \tag{3.1}$$

Beweis 1 Sei M der zur Verfügung stehende Hauptspeicherplatz, $\mathcal{S}_n$ die minimale Größe einer Hauptspeicherinstanziierung eines Suffixbaumes über einem Text der Länge n und P die Menge aller Partitionen.
Zur Analyse des Partitionierungsalgorithmus werden zunächst die Laufzeiten der einzelnen Teile des Algorithmus innerhalb der äußeren Schleife bestimmt. Die Gesamtlaufzeit setzt sich aus der Addition der Laufzeiten über alle Partitionen zusammen. Für jede Partition $p \in P$ wird die innere Schleife durchlaufen, enthaltene Suffixe eingefügt und der Suffixbaum-Zweig $\widehat{p_w}$ persistent gespeichert.

$$T_{PartSft}(n) = \sum_{p_w \in P} (T_{innerloop}(n, p_w) + T_{insertions}(n, p_w) + T_{checkpoint}(n, p_w)) \tag{3.2}$$

Da ein innerer Schleifendurchlauf unabhänging von der aktuellen Partition p ist und alle n Suffixe von t^+ durchlaufen werden gilt:

$$T_{innerloop}(n, p_w) = \Theta(n) \tag{3.3}$$

Genügt der Text t einer Bernoulli-Verteilung (alle Zeichen von t sind unabhängig voneinander verteilt), so ist die erwartete Tiefe $d(s)$ der Einfügestelle eines beliebigen Suffixes s in den Suffixbaum $\mathcal{T}(t)$ logarithmisch in n (eine einfache Folgerung aus den Ergebnissen von Apostolico und Szpankowski [4]). Szpankowski [50] hat sogar später gezeigt, dass die Höhe des Suffixbaumes im Bernoulli-Modell (alle Zeichen des Textes sind unabhängig voneinander verteilt) sowie im Markov-Modell (das Zeichen $t[i]$ ist abhängig vom Zeichen $t[i-1]$) fast sicher gegen $\Theta(log\, n)$ konvergiert. Damit ergibt sich bei einer erwarteten logarithmischen Einfügetiefe die erwartete Laufzeit für den Aufbau einer Partition p_w mit:

$$T_{insertions}(n, p_w) = |p_w| \times O(log\, n) \tag{3.4}$$

Bei einem Checkpoint übernimmt das Speichermanagement der verwendeten Plattform die persistente Speicherung der Daten. Die benötigte Laufzeit ist hier linear in der Größe des Suffixbaum-Zweiges $\widehat{p_w}$. Somit ergibt sich für diese persistente Speicherung:

$$T_{checkpoint}(n, p_w) = \Theta(|\widehat{p_w}|) = \Theta(|p_w|) \tag{3.5}$$

Setzt man nun die Abschnittslaufzeiten in 3.2 ein, so erhält man die Gesamtlaufzeit des Algorithmus:

$$T_{PartSft}(n) \tag{3.6}$$

$$= \sum_{p_w \in P} (\Theta(n) + |p_w| \times O(log\, n) + \Theta(|p_w|)) \tag{3.7}$$

$$= \Theta\left(\sum_{p_w \in P} n\right) + \sum_{p_w \in P} (|p_w| \times O(log\, n)) + \sum_{p_w \in P} \Theta(|p_w|) \tag{3.8}$$

$$= \Theta\left(|P| \times n\right) + \left(\sum_{p_w \in P} |p_w|\right) \times O(log\, n) + \Theta\left(\sum_{p_w \in P} |p_w|\right) \tag{3.9}$$

Da $|P|$ eine Partitionierung alles Suffixe von t ist, gilt: $n = \sum_{p_w \in P} |p_w|$. Eingesetzt in 3.9 ergibt das:

$$T_{PartSft}(n) = \Theta\left(|P| \times n\right) + n \times O(log\, n) + \Theta(n) \tag{3.10}$$

$\mathcal{S}_n \in \Theta(n)$ und alle Partitionen $p \in P$ sind durch die Hauptspeichergröße beschränkt. Die minimale Anzahl von Partitionen ist somit

$$|P| = \left\lceil \frac{\mathcal{S}_n}{M} \right\rceil \in \Theta(n). \tag{3.11}$$

Zusammen mit 3.10 ergibt das eine erwartete Gesamtlaufzeit von

$$\begin{aligned} T_{PartSft}(n) &= \Theta(n \times n) + n \times O(log\, n) + \Theta(n) && (3.12)\\ &= \Theta(n^2). && (3.13) \end{aligned}$$

Die erwartete Laufzeit des Partitionierungsalgorithmus ist unabhängig von der Verteilung des Textes t immer quadratisch. □

Obwohl die erwartete Laufzeit über alle Einfügeoperationen in $O(n\ log\, n)$ liegt, ergibt sich durch das wiederholte durchlaufen des Textes t insgesamt eine quadratische Komplexität. Der Algorithmus sollte insbesondere für längere Texte schneller arbeiten, da durch die separaten Zugriffe auf einzelne Suffixbaum-Zweige das globale Lokalitätsverhalten verbessert wird (die Zugriffe beschränken sich innerhalb eines kurzen Zeitintervalls auf nur eine Partition) und der Algorithmus außerdem eine einfache Struktur aufweist. Dies gilt allerdings nur, solange der Suffixbaum ausbalanciert ist, da er keinen Mechanismus zur Behandlung von Entartungen enthält.

3.1.3 Hashed-Position-Trees

Eine Datenstruktur von Mewes und Heumann [45] geht in die gleiche Richtung wie der Partitionierungsalgorithmus, ist aber noch durch explizite seitenbasierte Komponenten erweitert. Allerdings entsteht nicht mehr ein reiner Suffixbaum, sondern eine hybride Datenstruktur, genannt Hashed-Position-Tree, welche die Konzepte des Hashings, der Suffixbaum-Struktur und des seitenbasierten Zugriffs vereinigt.
Er besteht aus vier verschiedenen Komponenten:

- Die Hashtabelle ersetzt den Teil des Baumes von der Wurzel bis zu einer bestimmten Tiefe, die abhängig von der Länge des Präfixes ist, auf den sich die Hashfunktion bezieht.
- Ein Kollisionsklassenbaum verfeinert die Kollisionsklassen der Hashtabelle bis jede verfeinerte Kollisionsklasse in einem Segment fester Größe gespeichert werden kann.
- Eine Seite entspricht der gleichnamigen Einheit der Festplatte und enthält mehrere Segmente.

- Ein Segment speichert eine Menge von Suffixnummern ($leafSet(\overline{w})$), die einen gleichen Präfix w aufweisen. Ein Segment kann somit als Vereinigung von Suffixbaum-Blättern, die am gleichen Zweig hängen, aufgefasst werden.

Der Hashed-Position-Tree wird, wie der Suffixbaum beim Partitionierungsalgorithmus, via Einfügen der einzelnen Suffixe konstruiert. Durch die blockorientierten Komponenten scheint er eine große Ähnlichkeit mit dem String-B-Baum zu haben. Die Unterschiede ergeben sich durch die Art der Einfügeoperationen, die nicht gewährleisten, dass die Baumstruktur ausbalanciert bleibt.
Beim String-B-Baum wird ein B-Baum-Knoten geteilt, wenn er maximal gefüllt ist und man in ihn einfügen möchte. Dabei ist es durchaus möglich, dass nachfolgende B-Baum-Knoten noch Daten aufnehmen können.
Beim Hashed-Position-Tree wird die Teilung der blockorientierten Komponenten durch den Überlauf der jeweils darunterliegenden Komponenten ausgelöst. Entsteht beispielsweise durch die Teilung eines Segments ein Weiteres, so dass eine Seite diese höhere Anzahl von Segmenten nicht mehr fassen kann, wird auch sie geteilt. Das gleiche gilt für Seiten und Kollisionsklassen. Letztere werden zu kleineren geteilt, da sie unterhalb der Hashtabelle die umgebende Struktur für die Seiten und Segmente bilden.

Der Hashed-Position-Tree zeigt die gleiche erwartete $O(n\ log\, n)$-Komplexität wie der Partitionierungsalgorithmus. Zusätzlich besitzt er eine blockorientierte Speicherverwaltung, die zusammen mit der Kollisionsbehandlung eine geeignete Strategie zur Konstruktion von entarteten Bäumen darstellt.
Es wäre wünschenswert, den Partitionierungsalgorithmus um solche Features erweitern zu können.

3.2 Verbesserter Partitionierungsalgorithmus

Die wesentlichen Schwachstellen des Partitionierungsalgorithmus sind sein schlechtes Verhalten bei ungleichmäßiger Verteilung der Suffixe und seine nicht blockorientierten Zugriffe auf den Hintergrundspeicher. Diese fallen allerdings nicht sehr ins Gewicht, da die Daten nur einmalig geschrieben und zur weiteren Konstruktion nicht erneut geladen werden müssen.
Der Hashed-Position-Tree, der im Gegensatz zum Partitionierungsalgorithmus eine explizite Kollisionsbehandlung mit blockorientiertem Zugriff enthält, muss aber ähnlich wie der String-B-Tree auf schon geschriebene Daten wieder zugreifen. Das erfordert zwar nur wenige Hintergrundspeicherzugriffe, aber wie man am Beispiel des String-B-Baumes sieht, leidet die Konstruktionszeit doch sehr stark darunter. Untersuchungen von Manuel Scholz innerhalb seiner Diplomarbeit haben ergeben, dass der String-B-Baum zwar ein asymptotisch sehr gutes Verhalten aufweist, aber

in der Praxis durch die langsamen Festplattenzugriffe nicht konkurrenzfähig ist.
Meine verbesserte Version des Partitionierungsalgorithmus integriert die Kollisionsbehandlung und den blockorientierten Zugriff. Damit wird der Algorithmus stabil bezüglich Entartungen und berücksichtigt die Datenzugriffsmechanismen heutiger Rechnerarchitekturen.

3.2.1 Algorithmus

Der Pseudocode meines verbesserten Algorithmus ist in Abbildung 3.3 dargestellt. In der ersten Phase werden alle Partitionen erzeugt, deren Suffixbauminstanziierung eine feste Größe *blocksize* nicht übersteigt (Abb.3.4).
Der Pseudocode in Abbildung 3.4 beschreibt die Berechnung dieser Partitionen. Zunächst erzeugt die Funktion *ComputePartitions* die Menge aller Σ^d Partitionen, die Anfangs leer sind (Abb. 3.4, oben, Zeilen 1-5). Danach werden alle Suffixe gemäß der Funktion h, durch einmaliges *Scannen* des Textes t, auf ihre Partitionen abgebildet (Abb. 3.4, oben, Zeilen 6-9).
Nachdem nun alle Partitionen $p_w \in P$ einschließlich ihrer Kardinalitäten $|p_w|$ bekannt sind, werden im nächsten Schritt (Abb. 3.4, oben, Zeilen 10-15) die überlaufenden Partitionen $\{p_w \in P \,|\, |p_w| > \frac{blocksize}{c}\}$, deren maximale Suffix-Zweig-Instanziierung im Hauptspeicher größer als die Blockgröße *blocksize* ist, verfeinert. (Da eine Suffixbaum-Instanziierung $\mathcal{S}_{\mathcal{T}(t)}$ von der Länge des Eingabetextes t abhängt, wird sie durch einen konstanten Faktor c nach oben abgeschätzt. Wenn also $\mathcal{S}_{\mathcal{T}(t)}$ die Größe einer maximalen Hauptspeicherinstanziierung von $\mathcal{T}(t)$ und $\mathcal{S}_t$ die Größe der Instanziierung von t ist, so ist $\mathcal{S}_{\mathcal{T}(t)}$ durch $c \times \mathcal{S}_t$ nach oben beschränkt.)
Die Blockgröße *blocksize* sollte dabei ein Vielfaches der verwendeten Seitengröße des Betriebssystems sein und den verfügbaren Hauptspeicherplatz nicht überschreiten.
Die Verfeinerung der Partition wird durch *Refine* beschrieben (Abb. 3.4, unten). Der Präfix w, dessen entsprechende Suffixe in die Partition p_w fallen, wird um ein Zeichen b verlängert. Dadurch entstehen $|\Sigma|$ neue Partitionen (Abb. 3.4, unten, Zeilen 2-5). Die Suffixe aus p_w werden, entsprechend ihrem Zeichen an der Stelle $|w| + 1$, auf die neuen Partitionen aufgeteilt (Abb. 3.4, unten, Zeilen 6-9). Falls es immer noch Partitionen gibt, welche die zulässige Größe überschreiten, werden diese weiter verfeinert (Abb. 3.4, unten, Zeilen 10-15).
Nachdem nun alle Partitionen $p_w \in P$ eine Größe $|p_w|$ haben, die es erlaubt jeden Zweig $\widehat{p_w}$ komplett im Hauptspeicher zu konstruieren, werden diese Zweige jeweils nacheinander aufgebaut und persistent gespeichert (Abb. 3.3, Zeilen 3-14). Zuerst wird zu einer Partition p_w der anfangs leere Zweig $\widehat{p_w}$ kreiert (Abb. 3.3, Zeile 4). Danach wird jeder Suffix $s \in p_w$ in den Zweig $\widehat{p_w}$ eingefügt (Abb. 3.3, Zeilen 5-7). Der Unterschied zum grundlegenden Partitionierungsalgorithmus besteht darin, dass die Einfügeoperationen nicht an der Wurzel des Suffixbaumes $\mathcal{T}(t)$ beginnen, sondern direkt im Knoten $\overline{w}$, der sich auf einer Tiefe $|w|$ im Baum $\mathcal{T}(t)$ befindet.

Die Zweige $\widehat{p_w}$ liegen nach vollständiger Konstruktion in einem Hauptspeicherformat vor und müssen in eine geeignete Hintergrundspeicherrepräsentation konvertiert werden. Passt der Zweig $\widehat{p_w}$ nicht vollständig in den bereits mit den vorherigen Zweigen gefüllten Block B, so wird der vorherige Block persistent gespeichert (Abb. 3.3, Zeile 9) und ein neuer Block B kreiert (Abb. 3.3, Zeile 10). Im aktuellen Block B ist nun ausreichend Speicherplatz für den zu speichernden Zweig $\widehat{p_w}$ vorhanden, so dass $\widehat{p_w}$ in B geschrieben werden kann (Abb. 3.3, Zeile 12). Sind alle Partitionen abgearbeitet, wird auch der noch verbliebene Block B gespeichert (Abb. 3.3, Zeile 14).
Nachdem jetzt der untere Teil des Suffixbaumes $\mathcal{T}(t)$ schon fertig konstruiert ist, muss zum Schluss noch der *"Stumpf"* $\widehat{tr}$ oberhalb der Zweige kreiert und persistent gespeichert werden (Abb. 3.3, Zeilen 15+16). Unter dem Stumpf versteht man den Teil des Suffixbaumes bis zu einer bestimmten Ebene, die durch die Partitionierungstiefe bestimmt ist (vergleiche Abb. 3.1).

```
ImprovedPartitionSuffixTree(t, d)
 1: P ← ComputePartitions(d)
 2: B ← createEmptyBlock()
 3: for all partitions p_w ∈ P do
 4:    p_w^ ← createEmptyTreePartition(p_w)
 5:    for all suffixes s ∈ p_w do
 6:       insertIntoTreePartition(s, p_w^)
 7:    end for
 8:    if p_w^ does not fit in previous block B then
 9:       writeBlockToDisk(B)
10:       B ← createEmptyBlock()
11:    end if
12:    writeToBlock(p_w^, B)
13: end for
14: writeBlockToDisk(B)
15: tr^ ← buildSuffixTreeTrunk()
16: writeTrunkToDisk(tr^)
```

Abbildung 3.3: Verbesserter Partitionierungsalgorithmus

3.2.2 Eigenschaften

Die vorher erwähnten wiederholten Durchläufe des Strings t im Basisalgorithmus resultieren in einer $\Theta(n^2)$ Komplexität. Obwohl mein verbesserter Ansatz im schlech-

ComputePartitions(t, d)

1: $P \leftarrow$ createEmptyPartitionSet()
2: **for all** $j \in \{j \in \mathbb{N} \mid 0 \leq j < \Sigma^d\}$ **do**
3: $p_{w_j} \leftarrow$ createEmptyPartition(w_j)
4: $P \leftarrow P \cup \{p_{w_j}\}$
5: **end for**
6: **for all** $i \in \{i \in \mathbb{N} \mid 1 \leq i \leq (n+1) - d\}$ **do**
7: $j \leftarrow h_d(i)$
8: $p_{w_j} \leftarrow p_{w_j} \cup \{i\}$
9: **end for**
10: **for all** $p_w \in P$ **do**
11: **if** $|p_w| > \frac{blocksize}{c}$ **then**
12: $P_{ref} \leftarrow$ Refine(p_w)
13: $P \leftarrow (P \setminus \{p_w\}) \cup P_{ref}$
14: **end if**
15: **end for**
16: return(P)

Refine(t, p_w)

1: $P_{ref} \leftarrow$ createEmptyPartitionSet()
2: **for all** $b \in \Sigma$ **do**
3: $p_{wb} \leftarrow$ createEmptyPartition(wb)
4: $P_{ref} \leftarrow P_{ref} \cup \{p_{wb}\}$
5: **end for**
6: **for all** $i \in p_w$ **do**
7: b $\leftarrow t[i + |w|]$
8: $p_{wb} \leftarrow p_{wb} \cup \{i\}$
9: **end for**
10: **for all** $p \in P_{ref}$ **do**
11: **if** $|p| > \frac{blocksize}{c}$ **then**
12: $P_{ref2} \leftarrow$ Refine(p)
13: $P_{ref} \leftarrow (P_{ref} \setminus \{p\}) \cup P_{ref2}$
14: **end if**
15: **end for**
16: return(P_{ref})

Abbildung 3.4: Verbesserte Partitionierung

testen Fall das gleiche asymptotische Verhalten von $O(n^2)$ hat, ist seine erwartete Laufzeit $O(n \log n)$.

Analyse 2 *Sei $n = |t|$ die Länge des Eingabetextes t und $T_{ImprPart}(n)$ die erwartete Laufzeit des verbesserten Partitionierungsalgorithmus, dann gilt im erwarteten Fall:*

$$T_{ImprPart}(n) \quad \in \quad O(n \log n) \tag{3.14}$$

Beweis 2 Da die verbesserte Version des Partitionierungsalgorithmus sich nicht grundlegend von dem Basisansatz unterscheidet, sondern diesen nur erweitert, ist auch der Beweis ähnlich.
Sei M der zur Verfügung stehende Hauptspeicherplatz, $\mathcal{S}_n$ die minimale Größe einer Hauptspeicherinstanziierung eines Suffixbaumes über einem Text t der Länge $n = |t|$ und P die Menge aller Partitionen.
Zur Analyse wird der Algorithmus wieder in seine Teillaufzeiten aufgebrochen. Die Gesamtlaufzeit addiert sich dann aus den einzelnen Teillaufzeiten für die Partitionierung, den Aufbau und die Speicherung der Zweige sowie der Konstruktion und Speicherung des Suffixbaum-Stumpfs.

$$T_{ImprPart}(n) = \tag{3.15}$$

$$T_{computePartitions}(n) + T_{refinePartitions}(n, P) \tag{3.16}$$

$$+ \sum_{p_w \in P} (T_{create}(n, p_w) + T_{insertions}(n, p_w) + T_{writeD}(n, p_w)) \tag{3.17}$$

$$+T_{buildTrunk}(n) + T_{writeTrunk}(n) \tag{3.18}$$

In der ersten Phase des Algorithmus werden durch einmaliges Scannen der Sequenz alle Partitionen $p_w \in P$ berechnet (Abb. 3.3, Zeile 1), was linearen Zeitaufwand erfordert:

$$T_{computePartitions}(n) \quad = \quad \Theta(n) \tag{3.19}$$

Da die Verteilung des Textes als zufällig angenommen wird und die initiale Partitionierungstiefe d ausreichend gewählt werden kann, ist es im erwarteten Fall nicht nötig die Partitionen zu verfeinern. Der Algorithmus überprüft alle Partitionen ohne weitere Berechnungen durchzuführen. Mit 3.11 ist die Teillaufzeit also:

$$T_{refinePartitions}(n, P) \quad = \quad O(|P|) = O(n) \tag{3.20}$$

Das kreieren eines noch leeren Zweiges ist unabhängig von n und p_w und ist in konstanter Zeit zu realisieren:

$$T_{create}(n, p_w) \quad = \quad O(1) \tag{3.21}$$

Da der Suffixbaum nach Apostolico und Szpankowski [4] bzw. nach Szpankowski [50] erwartete logarithmische Höhe hat, ist auch die Laufzeit der Einfügeoperationen logarithmisch:

$$T_{insertions}(n, p_w) = |p_w| \times O(log\, n) \tag{3.22}$$

Die Umwandlung ins Festplattenformat und die persistente Speicherung ist abhängig von der Größe der entsprechenden Partition p_w:

$$T_{writeD}(n, p_w) = \Theta(|p_w|) \tag{3.23}$$

Zum Schluss wird noch der Suffixbaum-Stumpf konstruiert. Die Höhe des Stumpfes entspricht der Partitionierungstiefe d. Diese sollte maximal logarithmisch in n sein, da es sonst mehr Partitionen gibt als Suffixe im Text t. Für jede Partition p_w wird nun der entsprechende Präfix w in den Stumpf $\widehat{tr}$ eingefügt

$$\begin{aligned} T_{buildTrunk}(n, P) &= |P| \times O(d) && (3.24)\\ &= O(n) \times O(log\, n) && (3.25)\\ &= O(n\, log\, n) && (3.26) \end{aligned}$$

und dann der Stumpf $\widehat{tr}$ persistent gespeichert:

$$T_{writeTrunk}(n, P) = O(n) \tag{3.27}$$

Nach dem Einsetzen aller Teillaufzeiten in 3.15 bis 3.18 ist die Gesamtlaufzeit:

$$\begin{aligned} T_{ImprPart}(n) &= \Theta(n) + O(n) && (3.28)\\ &\quad + \sum_{p_w \in P} (O(1) + |p_w| \times O(log\, n) + \Theta(|p_w|)) && (3.29)\\ &\quad + O(n\, log\, n) + O(n) && (3.30) \end{aligned}$$

Jetzt werden einige Teillaufzeiten zusammengefasst und die Summe in kleinere Terme zerlegt.

$$\begin{aligned} T_{ImprPart}(n) &= \sum_{p_w \in P} O(1) && (3.31)\\ &\quad + \sum_{p_w \in P} (|p_w| \times O(log\, n)) && (3.32)\\ &\quad + \sum_{p_w \in P} \Theta(|p_w|) && (3.33)\\ &\quad + O(n\, log\, n) && (3.34) \end{aligned}$$

Nach etwas Arithmetik ergibt sich:

$$\begin{aligned} T_{ImprPart}(n) &= O(|P| && (3.35)\\ &\quad + O(log\, n) \times \sum_{p_w \in P} |p_w| && (3.36)\\ &\quad + \Theta\left(\sum_{p_w \in P} |p_w|\right) && (3.37)\\ &\quad + O(n\, log\, n) && (3.38) \end{aligned}$$

Da P eine Partitionierung aller Suffixe von t^+ ist und damit $n = \sum_{p_w \in P} |p_w|$, folgt zusammen mit 3.11:

$$\begin{aligned} T_{ImprPart}(n) &= O(n) && (3.39)\\ &\quad + O(log\, n) \times n && (3.40)\\ &\quad + \Theta(n) && (3.41)\\ &\quad + O(n\, log\, n) + O(n) && (3.42)\\ &= O(n\, log\, n) && (3.43) \end{aligned}$$

Somit hat der verbesserte Partitionierungsalgorithmus eine erwartete Laufzeit von $O(n\, log\, n)$. □

Richtet man einen genaueren Blick auf die Zerlegung des Suffixbaumes durch meinen Algorithmus, so ist zu erkennen, dass beim Aufbau der einzelnen Zweige die Einfügeoperationen nicht in der Wurzel beginnen, sondern in dem Knoten, von dem ein Zweig ausgeht. Bei einer Partitionierung nach Wortlänge d liegt dieser Knoten auf Ebene d des Suffixbaumes. Die Einfügeoperationen benötigen somit bei einer erwarteten Einfügehöhe von $c \times log\, n$ nur $c \times log\, n - d$ Schritte. Dies führt mich zu folgender Analyse:

Analyse 3 *Sei $n = |t|$ die Länge des Eingabetextes, $T_{ImprPart}$ die erwartete Laufzeit des verbesserten Partitionierungsalgorithmus, d die Partitionierungstiefe, dann gilt:*

$$\begin{aligned} &\exists c \in \mathbb{R}: && (3.44)\\ &\qquad d = c\, log\, log\, n && (3.45)\\ &\implies T_{ImprPart}(n) \in O\left(n\, log\left(\frac{n}{log\, n}\right)\right) && (3.46) \end{aligned}$$

Beweis 3 Logarithmen können bis auf Multiplikation mit einer Konstanten ineinander überführt werden. Im folgenden nehme ich zur Vereinfachung an, dass die Logarithmen zur Basis $a = |\Sigma|$ gegeben sind.

Nach Apostolico und Szpankowski [4]) bzw. Szpankowski [50] ist die erwartete Einfügehöhe eines Suffixes logarithmisch in n. Das heißt, es gibt ein $c \in \mathbb{R}$, so dass die erwartete Einfügehöhe $c\,log\,n$ ist. Wählt man nun abhängig von diesem c die Partitionstiefe mit $d = c\,log\,log\,n$, dann ergibt sich für eine Einfügeoperation die Laufzeit:

$$\begin{aligned} T_{ins}(n) &= c\,log\,n - c\,log\,log\,n && (3.47)\\ &= c \times (log\,n - log\,log\,n) && (3.48)\\ &= c\,log\left(\frac{n}{log\,n}\right) && (3.49)\\ &= O\left(log\left(\frac{n}{log\,n}\right)\right) && (3.50) \end{aligned}$$

Die Laufzeit für die Konstruktion eines Suffixbaum-Zweiges ist somit

$$T_{insertions}(n, p_w) = |p_w| \times O\left(log\left(\frac{n}{log\,n}\right)\right) \quad (3.51)$$

Die dynamische Partitionierungstiefe hat nicht nur Einfluss auf die Konstruktion der Zweige, sondern auch auf die Einfügeoperationen in den Suffixbaum-Stumpf. Bei $|P| = a^{c\,log\,log\,n}$ Partitionen und einer Länge der einzufügenden Präfixe mit $d = c\,log\,log\,n$, ist in diesem Fall die Laufzeit zur Konstruktion des Stumpfes:

$$\begin{aligned} T_{buildTrunk}(n, P) &= a^{c\,log\,log\,n} \times c\,log\,log\,n && (3.52)\\ &= (log\,n)^c \times c\,log\,log\,n && (3.53)\\ &= O\left((log\,n)^c\right) \times O(log\,log\,n) && (3.54) \end{aligned}$$

Da $(log\,n)^c$ asymptotisch kleiner ist als n und $log\,n$ asymptotisch kleiner ist als $\frac{n}{log\,n}$, erhält man nach entsprechendem Einsetzen:

$$\begin{aligned} T_{buildTrunk}(n, P) &= O(n) \times O\left(log\left(\frac{n}{log\,n}\right)\right) && (3.55)\\ &= O\left(n\,log\left(\frac{n}{log\,n}\right)\right) && (3.56) \end{aligned}$$

Setzt man nun 3.51 und 3.56 zusammen mit 3.19-3.21, 3.23, 3.27 in 3.15-3.18 ein, so ergibt sich eine Gesamtlaufzeit von:

$$T_{ImprPart}(n) = \Theta(n) + O(n) \quad (3.57)$$

$$+ \sum_{p_w \in P} \left(O(1) + |p_w| \times O\left(log\left(\frac{n}{log\,n}\right)\right) + \Theta(|p_w|)\right) \quad (3.58)$$

$$+O\left(n\,log\left(\frac{n}{log\,n}\right)\right) + O(n) \quad (3.59)$$

Nach Zusammenfassung der schon bestimmten asymptotischen Werte und Aufsplitten der Summe reduziert sich die Gleichung zu:

$$T_{ImprPart}(n) = \sum_{p_w \in P} O(1) \tag{3.60}$$

$$+ \sum_{p_w \in P} \left(|p_w| \times O\left(log\left(\frac{n}{log\, n} \right) \right) \right) \tag{3.61}$$

$$+ \sum_{p_w \in P} \Theta(|p_w|) \tag{3.62}$$

$$+ O\left(n\ log\left(\frac{n}{log\, n} \right) \right) \tag{3.63}$$

Mit etwas Arithmetik werden die Summen vereinfacht.

$$T_{ImprPart}(n) = O(n) \tag{3.64}$$

$$+ O\left(log\left(\frac{n}{log\, n} \right) \right) \times \sum_{p_w \in P} |p_w| \tag{3.65}$$

$$+ \Theta\left(\sum_{p_w \in P} |p_w| \right) \tag{3.66}$$

$$+ O\left(n\ log\left(\frac{n}{log\, n} \right) \right) \tag{3.67}$$

Da $n = \sum_{p_w \in P} |p_w|$ folgt zum Schluss:

$$T_{ImprPart}(n) = O(n) \tag{3.68}$$

$$+ O\left(log\left(\frac{n}{log\, n} \right) \right) \times n \tag{3.69}$$

$$+ \Theta\left(n \right) \tag{3.70}$$

$$+ O\left(n\ log\left(\frac{n}{log\, n} \right) \right) \tag{3.71}$$

$$= O\left(n\ log\left(\frac{n}{log\, n} \right) \right) \tag{3.72}$$

Es gibt daher bei einer zufälligen Struktur des Textes t immer eine Partitionierungstiefe d, so dass $T_{ImprPart}(n) = O\left(n\ log\left(\frac{n}{log\, n} \right) \right)$ ist. □

Eine weitere wichtige Eigenschaft meines verbesserten Algorithmus ist seine Stabilität bezüglich entarteter Suffixbäume. Wenn der Eingabetext t nicht zufälliger Natur

ist, produziert die Partitionierung möglicherweise Partitionen, deren Suffixbaum-Zweige nicht in den Hauptspeicher passen. Die Verfeinerung der überlaufenden Partitionen stellt sicher, dass zum Schluss alle Suffixbaum-Zweige im Hauptspeicher aufgebaut werden können. Im Gegensatz zum Hashed-Position-Tree werden Entartungen so schon frühzeitig erkannt und behandelt, so dass während der Konstruktion keine lesenden Zugriffe auf schon geschriebene Suffixbaum-Teile erforderlich sind.
Die mittlere Füllung der Blöcke ist mindestens $\frac{blocksize}{2}$, da mein Algorithmus jeden Block b mit Suffixbaum-Zweigen auffüllt, bis der nächste Zweig nicht mehr in den aktuellen Block passt. Schaut man sich zwei benachbarte Blöcke B_x und B_{x+1} an, so entspricht die Addition der beiden Füllungen mindestens einer Blockgröße $blocksize$, anderenfalls wären sie durch den Algorithmus zusammengelegt worden.

3.3 Der *wotd*-Algorithmus

Eine weitere Möglichkeit zur effizienten Konstruktion von Suffixbäumen bietet der *wotd*(write-only-top-down)-Algorithmus. Er ist ein an der Universität Bielefeld oft verwendeter und in mehreren Varianten untersuchter Ansatz [23], dessen Effizienz sich auf sein gutes Lokalitätsverhalten zurückführen lässt. Auf Grund dieser guten Lokalität, scheint er auch für den Aufbau von Suffixbäumen auf langen Texten gut geeignet zu sein.

3.3.1 Algorithmus

Der *wotd*-Algorithmus macht sich zunutze, dass für jeden inneren Knoten $\overline{u}$, der ein Teilwort u vom Text t repräsentiert, der entsprechende Zweig unterhalb dieses Knotens durch alle Suffixe bestimmt wird, die u als Präfix haben. Verfügt man also über die Menge aller Suffixe $S(\overline{u}) := \{s \mid us \sqsupseteq t^+\} = \{s_{i-|u|}(t^+) | i \in leafSet(\overline{u})\}$, die den Suffixen des Teilbaumes unterhalb von $\overline{u}$, ohne ihren Präfix u, entsprechen, so ist es möglich die Kinder von $\overline{u}$ zu bestimmen. Dieser Prozess wird das Evaluieren eines Knotens $\overline{u}$ genannt. Zur Evaluierung werden zuerst die Suffixe aus $S(\overline{u})$ entsprechend ihrem ersten Zeichen in Gruppen geteilt. Sei nun $group(\overline{u}, b) := \{v \in (\Sigma^+)^* | bv \in S(\overline{u})\}$ die Gruppe mit einem Anfangsbuchstaben $b \in B$. Man unterscheidet zwei Fälle:

- Falls die Gruppe $group(\overline{u}, b)$ aus nur einem Element v besteht, so gibt es keine weiteren Suffixe in $S(\overline{u})$ die ub als Präfix enthalten. Es gibt demnach ein Blatt dessen Kantenbeschriftung bv entspricht.

- Wenn die Gruppe $group(\overline{u}, b)$ aus mehreren Elementen besteht, so erhält man einen inneren Knoten. Zur Bestimmung der Kantenbeschriftung muss dann noch der längste gemeinsame Präfix r aller Suffixe in der Gruppe $group(\overline{u}, b)$

errechnet werden. Die Kantenbeschriftung entspricht dann dem Teilstring br von t. Der neue Knoten kann später über die Menge $S(\overline{ubr})$ evaluiert werden.

Um den kompletten Baum zu konstruieren, startet der *wotd*-Algorithmus an der Wurzel mit der Menge aller Suffixe ($S(\overline{p})$) und evaluiert dann alle Knoten des Baumes von oben nach unten.

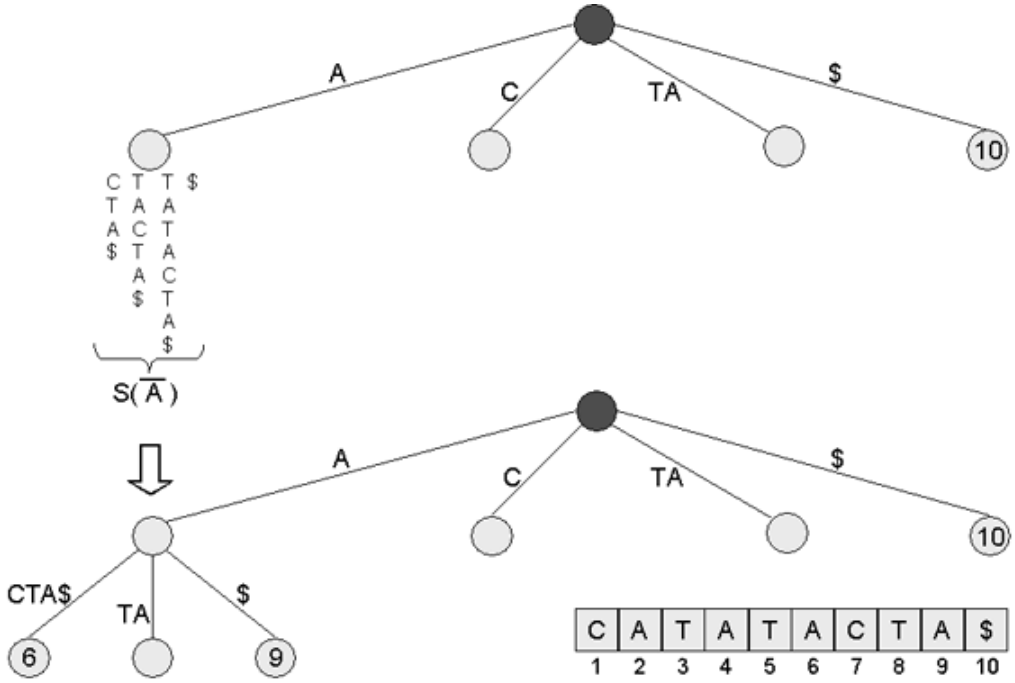


Abbildung 3.5: Evaluation of Node $\overline{A}$

Abbildung 3.5 beschreibt die Evaluierung des Knotens $\overline{A}$ für den Suffixbaum aus den vorhergehenden Beispielen. Die Suffixmenge $S(\overline{A}) = \{CTA\$,\ TATACTA\$,\ TACTA\$,\ \$\}$ entspricht der Blattmenge $leafSet(\overline{A})$. Diese Menge wird dann in drei Gruppen entsprechend den Anfangsbuchstaben C, T, \$ geteilt. Die C-Gruppe und die \$-Gruppe enthalten je nur ein Element. Über die C-Gruppe $group(\overline{A}, C) = \{TA\$\}$ erhält man das Blatt $\overline{ACTA\$}$, was dem Suffix mit der Nummer 6 entspricht. Für die zugehörige Kante $(\overline{A}, \overline{ACTA\$})$ ergibt sich die Beschriftung durch den Suffix in der C-Gruppe, einschließlich des Zeichens C ($\omega(\overline{A}, \overline{ACTA\$}) = (7, 10)$). Auf die gleiche Art und Weise wird aus der \$-Gruppe das Blatt mit der Nummer 9 evaluiert. Die T-Gruppe ($group(\overline{A}) = \{ATACTA\$, ACTA\$\}$) enthält zwei Elemente und wird somit zum inneren Knoten evaluiert. Dazu muss noch der längste gemeinsame Präfix $lcp(ATACTA\$, ACTA\$) = A$ berechnet werden. Die Kantenbeschriftung entspricht dann dem Teilstring $TA = t[\omega(\overline{A}, \overline{ATA})]$. Nachdem der innere Knoten $\overline{A}$ evaluiert ist, fährt der Algorithmus fort, bis zu jedem inneren Knoten alle Kinder generiert sind und damit der Suffixbaum komplett ist.

3.3.2 Eigenschaften des *wotd*-Algorithmus

Die *top-down*-Konstruktion ist in der Literatur vielfach erwähnt. Durch seine *worst-case*-Laufzeit von $O(n^2)$ scheint der *wotd*-Algorithmus, wie der Partitionierungsalgorithmus, auf den ersten Blick nicht konkurrenzfähig zu sein. Seine erwartete Laufzeit liegt dennoch in $O(n \log n)$ und experimentelle Ergebnisse von Giegerich *et al.* [23] deuten an, dass er für Strings mittlerer Länge nahezu Linearzeitverhalten zeigt. Dies kann insbesondere durch sein gutes Lokalitätsverhalten erklärt werden. Die herausstechende Eigenschaft des *wotd*-Algorithmus ist, dass die Konstruktion von oben nach unten vorgeht. Sind zu einem inneren Knoten alle Kinder erzeugt, so ist es nicht mehr nötig diesen bei der Konstruktion von anderen Teilen des Suffixbaumes wieder zu besuchen. Der *wotd*-Algorithmus zeigt deshalb optimale Lokalität innerhalb der Baumstruktur. Auch bezüglich der Zugriffe auf den Text t verhält sich der *wotd*-Algorithmus sehr gut. Je tiefer sich die zu evaluierenden Knoten im Suffixbaum befinden, desto kleiner wird auch die Menge der verbleibenden Suffixe im darunterliegenden Teilbaum. Die Suffixe werden immer weiter gekürzt und befinden sich am Ende des Strings, so dass ab einer bestimmten Ebene im Baum dieser Teil des Textes t in eine *Cacheseite* passt und keine weiteren *Cache-Misses* mehr für den Textzugriff auftreten. Durch dieses sehr gute Lokalitätsverhalten bezüglich der Suffixbaum- als auch der Textzugriffe scheint der *wotd*-Algorithmus sich auch gut für Texte zu eignen, deren Länge es nicht mehr erlaubt, den Suffixbaum komplett im Hauptspeicher zu konstruieren.

3.4 Sortierung innerhalb des *wotd*-Algorithmus

Der *wotd*-Algorithmus verlagert das Problem der nicht lokalen Datenzugriffe aus dem Suffixbaum in die Gruppierung der Suffixe. Dazu wird vor Beginn des Evaluierungsprozesses ein Feld suf angelegt, das die Nummern aller Suffixe von t speichert. Für jeden nicht evaluierten Knoten k gibt es ein Intervall $[left(k), right(k)]$ auf suf, das alle Nummern der Suffixe aus $S(k)$ enthält. Die Gruppierung erfolgt nun durch Sortierung des Intervalls $[left(k), right(k)]$ von suf bezüglich des ersten Zeichens der referenzierten Suffixe. Durch die Sortierung erhält man mehrere Teilintervalle von $[left(k), right(k)]$. Die referenzierten Suffixe eines Teilintervalls besitzen alle den gleichen Anfangsbuchstaben. Aus diesen Gruppen können dann die entsprechenden Knoten evaluiert werden.
Die Speicherung von Daten in einem Feld ist wesentlich näher an dem Aufbau heutiger Speichermedien orientiert als eine dynamische Baumstruktur, bei der keine Voraussagen über die Verteilung der Daten auf dem Speichermedium gemacht werden können. In einem Feld hat der implementierte Algorithmus die Kontrolle über die *Clusterung* der Daten. Bei guter *Clusterung* gibt es weniger *Cache-Misses* und das *virtuelle Speichermanagement* des Betriebssystems arbeitet effizienter bei der

Auslagerung der Daten.
Sortieralgorithmen sind in der Theorie und Praxis schon sehr gut erforscht. Zur Sortierung bietet sich in diesem Fall das *Countingsort* an, da das zugrundeliegende Alphabet Σ des Textes t im allgemeinen endlich ist. Das *Countingsort* hat eine für Sortieralgorithmen optimale Laufzeit von O(n). Das Problem beim *Countingsort* sind die zusätzlichen Speicherplatzanforderungen durch ein benötigtes Hilfsarray. Es wird also 2-mal mehr Speicherplatz benötigt als beim *Quicksort*. Das Quicksort kann die Sortierung *"in-place"* realisieren. Der Nachteil des *Quicksorts* ist aber seine erwartete $O(n \log n)$ Laufzeit. Bei einer Benutzung dieses Sortierverfahrens im *wotd*-Algorithmus würde sich für diesen eine erwartete Laufzeit von $O(n(\log n)^2)$ ergeben. Der Vorteil ist natürlich, dass auch für lange Texte das Array suf komplett in den Hauptspeicher passt.
Über die geeignete Sortierung von Suffixen gibt es viele andere Arbeiten [11, 39]. In [11] wird beispielsweise eine Variante des *Quicksort* besprochen, die den $\log n$-Faktor durch eine geschickte Partitionierung auf $|\Sigma|$ reduziert. Die genaue Untersuchung von mehreren Sortierverfahren ist leider im Rahmen dieser Arbeit nicht möglich. Ich gehe deshalb im folgenden nur auf ein weiteres von mir entwickeltes Sortierverfahren, das *Scansort* ein.

3.4.1 Scansort

Optimal für die Gruppierung im *wotd*-Algorithmus wäre eine *"in-place"* Sortierung, die in linearer Zeit zu berechnen ist.
Zu beachten sind die Art der Texte auf die ich mich in dieser Arbeit beziehe. Im biologischen Bereich sind sie dadurch gekennzeichnet, dass die zugrundeliegenden Alphabete klein sind. (DNA-Sequenzen haben eine Alphabetgröße von 4.) Ich habe auf dieser Grundlage einen Algorithmus entwickelt, der keinen zusätzlichen Speicherplatz erfordert (*"in-place"*-Sortierung), eine einfache Struktur hat und besonders gut auf Strings mit kleinen Alphabeten funktioniert.
Der Algorithmus erhält seinen Namen durch sein Verhalten. Er sortiert das Array durch mehrmaliges *Scannen* der gesamten Sequenz.

Scansort-Algorithmus

Der Algorithmus ist in Abbildung 3.6 angegeben. Die Variablen $left$ und $right$ beschreiben das Intervall $[left, right]$ von $array$, was noch nicht sortiert ist. Für ein Zeichen $b \in \Sigma$ werden dann durch *Separate(b, array, left, right)* alle Vorkommen von b in $array$ an den Anfang des Bereichs $[left, right]$ sortiert. Diese Prozedur wiederholt sich für alle Zeichen $b \in \Sigma$ auf dem noch nicht sortierten Bereich $[left, right]$, bis alle Zeichen zwischen $left$ und $right$ durchlaufen sind, oder nichts mehr zu sortieren ist. Die Trennung eines Buchstabens $b \in \Sigma$ von den anderen Zeichen in $array$

orientiert sich an der Partitionierung beim *Quicksort* [15]. Der Unterschied besteht darin, dass eine Partition aus nur gleichen Zeichen b besteht und die andere aus allen von b verschiedenen Zeichen. Die Funktion *Separate(b, array, left, right)* enthält zwei Zeiger i und j. Der Zeiger j sucht vom Ende des Feldes $array$ nach Vorkommen von b. Von links sucht i beginnend mit $left$ Zeichen, die ungleich b sind. Ist jeweils ein Paar gefunden worden, so werden die Zeichen ausgetauscht. Treffen sich i und j, befinden sich alle $b's$ links von i, am Anfang des Arrays.

```
Scansort(array, Σ)
 1: left ← 1
 2: right ← length(array)
 3: for all b ∈ Σ (in order) do
 4:    left ← Separate(b, array, left, right)
 5:    if left >= right then
 6:       return
 7:    end if
 8: end for
```

```
Separate(b, array, left, right)
 1: i ← left − 1
 2: j ← right + 1
 3: while TRUE do
 4:    repeat
 5:       j ← j − 1
 6:    until  j = i ∨ array[j] = b
 7:    if j = i then
 8:       return i+1
 9:    end if
10:    repeat
11:       i ← i + 1
12:    until  j = i ∨ array[i] ≠ b
13:    if j = i then
14:       return i
15:    end if
16:    swap(array[i], array[j])
17: end while
```

Abbildung 3.6: *Scansort*-Algorithmus *et al.*

Scansort-Eigenschaften

Der *Scansort*-Algorithmus sortiert alle Vorkommen eines Zeichens b an den Anfang des noch nicht sortierten Bereichs von $array$. Das Separieren des Zeichens von den anderen wird durch einmaliges Durchlaufen aller Elemente realisiert ($T_{seperate}(n) = O(n)$). Dieser Vorgang wiederholt sich für jedes Zeichen, so dass sich eine Gesamtlaufzeit von $O(n \times \Sigma)$ ergibt. Für die Sortierung von Strings mit kleinen Alphabeten, wie DNA-Sequenzen, sollte diese Sortierung besonders schnell funktionieren.
Das *Quicksort* ist in der Praxis als schnelles Sortierverfahren etabliert. In den von mir betrachteten Anwendungsgebieten treten sehr viele Wiederholungen eines Zeichens im String auf. Der Rekursionsanker des *Quicksort* hängt aber von der Länge des noch nicht sortierten Abschnitts ab. Dabei wird nicht erkannt, ob ein Bereich schon sortiert ist. Mit der Sortierung wird fortgefahren, obwohl ein breiter Bereich aus nur gleichen Zeichen besteht.
Das *Scansort* zeigt bezüglich der Separierung der Zeichen eine große Ähnlichkeit mit der Partitionierung innerhalb von *Quicksort*. Es übernimmt die in der Praxis bewährte Partitionierung von *Quicksort* in die Separierung eines Zeichens. Die Terminierung der Sortierung ist dabei unabhängig von der Breite des als noch nicht sortiert erkannten Bereichs. Die Separierung wird nur einmal für jedes Zeichen durchgeführt und sollte dadurch auf DNA-Sequenzen mit einem Alphabet der Größe 4 sehr effizient sein.
Der Vorteil gegenüber *Countingsort* liegt in der *"in place"* Sortierung. Es wird, im Gegensatz zum *Countingsort*, kein zusätzlicher Speicherplatz für die Sortierung benötigt. Dieser Vorteil macht sich besonders bei der Sortierung von langen Sequenzen bemerkbar, wenn der benötigte Speicherplatz, für die Sortierung durch das *Countingsort*, die Hauptspeichergröße übersteigt, was bei den von mir betrachteten Anwendungsgebieten sehr oft der Fall ist.

Kapitel 4

Speicherung von Suffixbäumen

Die Suffixbaum-Repräsentationen erfordern erheblichen Speicherplatz, obwohl die Größe eines Suffixbaumes linear in der Länge des zu indizierenden Textes ist. Im Gegensatz zu klassischen Indexstrukturen, bei denen der Speicherplatz für den Index üblicherweise sehr viel kleiner ist als die zu indizierenden Daten, benötigt der Suffixbaum sehr viel mehr Speicherplatz als der zugrundeliegende Text. Die Repräsentation von McCreight [44] benötigt beispielsweise 28 Bytes pro indizierten Buchstaben.
Die Menge der zu indizierenden DNA-Strings wuchs in den letzten Jahren exponentiell. Es ist daher notwendig, speichereffiziente Repräsentationen der Suffixbäume zu entwickeln, um diese ständig wachsende Masse an String-Daten noch effizient indizieren zu können.
In Kapitel 4.1 werden speichereffiziente Hauptspeicherrepräsentationen für den Partitionungsalgorithmus (Abschnitte 3.2+3.1) und den *wotd*-Algorithmus (Abschnitt 3.3) beschrieben, die sich an einem Format von Giegerich *et al.* [23] orientieren. Bei diesem beschränken sich die Speicherplatzanforderungen auf, im schlechtesten Fall, 12 Bytes pro indiziertem Zeichen. In Kapitel 4.2 wird, bezogen auf das Format aus [23], auf die geeignete persistente Speicherung der Suffixbäume eingegangen.
Bei dem Wechsel in die Datenbankwelt ergeben sich weitere Herausforderungen. Heute wird die meiste Zahl der strukturierten Daten in Datenbanken gespeichert. Es wäre wünschenswert, auch die Suffixbaum-Struktur in bestehende Datenbanktechnologien zu integrieren, um auch dort lange Strings indizieren zu können.
Momentan werden im wesentlichen zwei Datenbanktechnologien verwendet, die unterschiedlichen Paradigmen entsprechen – die objektorientierten Datenbanken [5] und die relationalen Datenbanken [52].
Objektorientierte Datenbanken [5] scheinen zur Speicherung einer Baumstruktur auf den ersten Blick gut geeignet zu sein. Hunt *et al.* [29] verwenden zur Speicherung ihres Suffixbaumes beispielsweise eine objektorientierte Technologie, die den Zugriffsmechanismus auf die Datenbank schon in die Programmiersprache (PJa-

ma [46]) integriert. Diese Technologie spart zwar Entwicklungszeit, resultiert aber in einem zusätzlichen Speicherplatzaufwand, den die objektorientierte Technologie mit sich bringt. Pro Knoten (maximal $2n - 1$) werden im Hauptspeicher 28 Bytes (!) benötigt und PJama [46] fügt bei der persistenten Speicherung sogar noch einige hinzu. Auch wenn andere objektorientierte Datenbanken mit weniger Speicherplatz auskommen, ergibt sich doch in jedem Fall ein *Overhead* für die Speicherung der Objekte. In einem Suffixbaum multipliziert sich jedes Byte, das für die Repräsentation eines Knotens aufgewendet wird, mit der Anzahl der Knoten ($> n$). Zusammen mit der großen Masse der zu indizierenden String-Daten, scheint die objektorientierte Technologie damit leider nicht geeignet zur Speicherung von Suffixbäumen zu sein. Die relationale Technologie wurde, durch die Unterstützung der Wirtschaft, intensiver verfolgt als die objektorientierte Datenbanktechnologie. Aus diesem Grund haben relationale Datenbanken eine große Verbreitung erfahren. Auch ein großer Anteil der Daten, die in Relation zu den von uns betrachteten Strings stehen, wird in relationalen Datenbanken gespeichert (Biologische Daten, ...). Die Integration sequenzbasierter Indexstrukturen, wie die der Suffixbäume, ist hier ein weiteres Ziel. Kapitel 4.3 diskutiert die Möglichkeit der Speicherung von Suffixbäumen in einer relationalen Datenbank und beschreibt eine Abbildung der Datenstruktur auf Tabellen.

4.1 Hauptspeicherrepräsentationen

Die hohen Speicherplatzanforderungen von Suffixbäumen sind auch mit ein Grund dafür, dass sie sich in der Praxis auf Bereiche beschränken, denen die entsprechenden Mittel zur Verfügung stehen, Computer mit großen Speicherkapazitäten zu unterhalten. Es ist notwendig speicherplatzsparende Repräsentationen der Suffixbäume zu entwerfen, damit diese vielseitige Datenstruktur anwendbar für viele verschiedene Bereiche wird.
Die Linearzeitalgorithmen von Weiner [53], McCreight [44] oder Ukkonen [51] lassen nur eine Hauptspeicherkonstruktion der Suffixbäume zu. Sobald die Suffixbaum-Größe den verfügbaren Hauptspeicherplatz überschreitet, steigen die Laufzeiten dieser Algorithmen stark an, so dass keine Konstruktion in angemessener Zeit mehr möglich ist. Damit nun möglichst große Teile des Suffixbaumes im Hauptspeicher konstruiert werden können, werde ich, orientiert an einem Suffixbaum-Format von Giegerich *et al.* [23], einige Komprimierungsschritte durchführen, die zu einer Hauptspeicherrepräsentation des Partitionierungsalgorithmus führen, bei der die Speicherplatzanforderungen nur 12 Bytes pro Knoten betragen, was maximal 24 Bytes pro Eingabezeichen sind. Die *top-down*-Konstruktion des wotd-Algorithmus lässt sogar eine Repräsentation zu, die nur 8 Bytes für jeden inneren Knoten und 4 Bytes für jedes Blatt benötigt. Andere in letzter Zeit veröffentlichte Arbeiten zur Repräsenta-

tion von Suffixbäumen benötigen 17 Bytes [17] bzw. 21 Bytes [29] pro indiziertem Zeichen.
Es gibt aber auch andere Datenstrukturen zur Mustersuche in Texten mit sehr geringen Speicherplatzanforderungen. Das Suffixarray von Manber und Myers [43] kommt, einschließlich Konstruktion, mit 9 Bytes pro indiziertem Zeichen aus. Der *Suffix-Kaktus* von Kärkkäinen [33] benötigt, wie auch der *Suffix-Binary-Search-Tree* von Irving und Love [32], $10n$ Bytes. Der *LC-Trie* von Andersson und Nilsson [3] erfordert ebenso wie das hier betrachtete Format [23] des Suffixbaumes $12n$ Bytes. Einige dieser Datenstrukturen sind in ihren Speicherplatzanforderungen dem Suffixbaum überlegen, realisieren die Mustersuche aber nicht in O(Musterlänge) oder setzen eine vorherige Konstruktion eines Suffixbaumes voraus. Damit ist der Suffixbaum, mit seiner Vielzahl von Anwendungsmöglichkeiten, diesen Datenstrukturen trotz der höheren Speicherplatzanforderungen überlegen.

4.1.1 Redundanzen im Suffixbaum

Bei einer naiven Suffixbaum-Repräsentation (Abb. 2.3) wird Speicherplatz für maximal $2|t|+1$ Knoten und maximal $2|t|$ Kanten benötigt. Ein Knoten referenziert alle seine Kinder und enthält zusätzlich eine Suffixnummer, falls er ein Blatt ist. Die Kanten beschreiben durch zwei Indexe ein Teilwort von t^+.
In Abbildung 4.1 ist der Suffixbaum aus Abschnitt 2.2 abgebildet, bei dem die Kin-

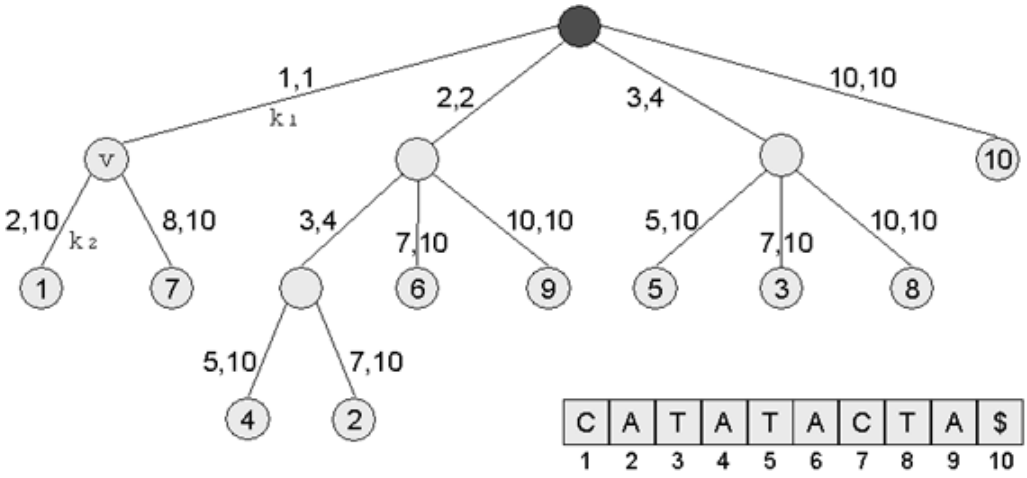


Abbildung 4.1: *Left-Child-Right-Sibling*-Repräsentation des Suffixbaumes aus Abbildung 2.3

der eines Knotens nach dem linken Index der zugehörigen Kantenbeschriftung von links nach rechts sortiert sind. Die Kante $k_1 = (x, v)$ endet im Knoten v, dessen ausgehende Kante $k_2 = (v, z)$ nach der entsprechenden Sortierung am weitesten links

steht. Für die Kante k_1 mit der Beschriftung $(1,1)$ und die Kante k_2 mit der Beschriftung $(2,10)$ gilt, dass der rechte Index ($rightIndex(k_1) = 1$) der Kantenbeschriftung von k_1 durch den linken Index ($leftIndex(k_2) = 2$) der Kantenbeschriftung von k_2 zu berechnen ist ($rightIndex(k_1) = leftIndex(k_2) - 1$). Diese Eigenschaft gilt für alle inneren Knoten des Baumes. Die Kanten, die auf die Blätter verweisen, haben $n+1 = |t^+|$ als Wert für den rechten Index. Bei einer geeigneten Wahl der Intervalle auf den Kanten und einer entsprechen Sortierung der Kinder kann somit auf die Speicherung des rechten Indexes für eine Kante verzichtet werden.
Die Blätter enthalten die Nummer i des Suffixes $s_i(t^+)$, den man durch Konkatenation der Teilstrings, entsprechend der Kantenbeschriftungen, auf dem Weg von der Wurzel zu einem Blatt erhält. Damit lässt sich auch die Suffixnummer i durch die Länge des Suffixes $|s_i(t^+)|$ über $i = n - |s_i(t^+)| + 2$ mit

$$|s_i(t^+)| \quad = \quad depth(l), \quad wobei\ l \in L(\mathcal{T}(t))\ und\ \lambda(l) = i$$

berechnen.
Algorithmen, welche die Suffixnummer benötigen, summieren beim Durchlaufen des Suffixbaumes die Länge der Kantenintervalle auf und bestimmen, entsprechend der obigen Formel, die Suffixnummer eines Blattes. Deshalb kann auch auf die Speicherung der Suffixnummer in den Blättern verzichtet werden.

4.1.2 Repräsentation Partitionierunsalgorithmus

Die festgestellten Redundanzen führen zu einer Repräsentation des Suffixbaumes im Partitionierungsalgorithmus, die ohne den rechten Index auf den Kanten und ohne die Suffixnummer in den Blättern auskommt.
In einem Wurzelbaum ist es möglich, die Kantenbeschriftung einer Kante (u, v) im Knoten v, in dem die Kante endet, zu speichern, da jede Kante in einem über alle Kanten eindeutigen Knoten endet. Die *Left-Child-Right-Sibling*-Repräsentation des Suffixbaumes wird gewählt, um sicherzustellen, dass alle Knoten höchstens zwei Verweise speichern müssen.
In Abbildung 4.2 ist der Suffixbaum aus Abbildung 4.1 in einer *Left-Child-Right-Sibling*-Repräsentation dargestellt. Die Kinder eines Knotens sind in einer Liste enthalten, die nach der im vorherigen Abschnitt beschriebenen Sortierung aufgebaut ist. Damit benötigt ein Knoten nur noch zwei Zeiger – einen auf den ersten Knoten seiner Kindliste und einen zum rechten Nachbarn. Die Kantenbeschriftungen werden in die inzidenten Kinder verlagert, womit, abgesehen von den in den Knoten gespeicherten Referenzen, keine explizite Struktur zur Darstellung der Kanten mehr notwendig.
In dieser knotenorientierten Repräsentation benötigt ein Knoten Speicherplatz für

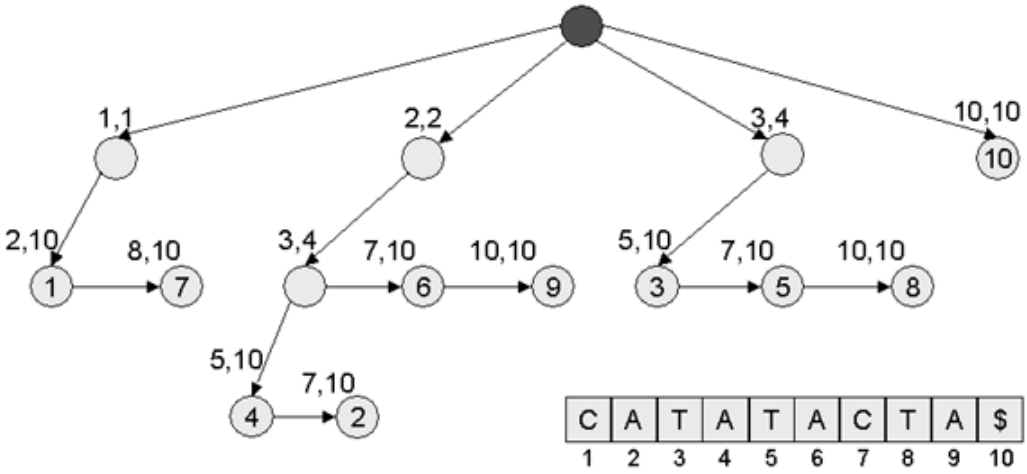


Abbildung 4.2: *Left-Child-Right-Sibling*-Repräsentation des Suffixbaumes

den linken Index (*long*-Wert) der entsprechenden Kantenbeschriftung sowie Verweise auf seinen rechten Nachbarn und sein linkes Kind (Zeiger). Übernimmt man die Größen der Datentypen aus der Programmiersprache C [34], in der *long*-Werte, genau wie Zeiger, 4 Bytes benötigen, so ergibt sich ein Bedarf von $4 + (2 \times 4) = 12$ Bytes pro Knoten im Suffixbaum.
Dabei müssen die Einfügeoperationen im Partitionierungsalgorithmus sicherstellen, dass der rechte Index der Kanten immer aus dem linken Index der linkesten folgenden Kante errechnet werden kann, bzw. die Kante zu einem Blatt führt. Es gibt bei den Einfügeoperationen zwei Fälle:

- Befindet sich die Einfügestelle genau in einem Knoten k_{parent}, so wird diesem k_{parent} ein neues Kind $k_{newchild}$ als Blatt hinzugefügt. Die Beziehung von k zu seinem linken Kind $k_{leftChild}$ verändert sich nicht, wenn das neue Kind in der Kindliste nicht an die erste Stelle tritt. Der rechte Index einer Kante, die zu einem Blatt führt, ist in jedem Fall $n + 1$, die Länge von t^+. Damit bleibt die Invariante bei diesem Vorgehen erhalten.

- Falls sich die Einfügestelle auf einer Kante $k = (u, v)$ befindet, so wird diese in $k_1 = (u, x)$ und $k_2 = (x, v)$ aufgesplittet. Dadurch entsteht ein neuer innerer Knoten x, dem dann, wie im ersten Fall, ein neues Kind (als Blatt) angehängt wird. Wenn $(a, b) = \omega(k)$ die Kantenbeschriftung von k ist, so ergeben sich für die neuen Kantenbeschriftungen die Werte $\omega(k_1) = (a, f)$ bzw. $\omega(k_2) = (f + 1, b)$, wobei f die Länge des Abschnitts der Kantenbeschriftung von k bis zur Einfügestelle ist. Die Nebenbedingung wird damit auch für diesen Fall nicht verletzt.

Die beschriebene Suffixbaum-Repräsentation eignet sich für den Partitionierungsalgorithmus, da dieser es bei geeigneter Implementierung zulässt, dass die redundanten Informationen nicht gespeichert werden müssen. Die maximalen Speicherplatzanforderungen dieser Suffixbaum-Repräsentation sind $12 \times 2n = 24n$ Bytes.

4.1.3 Repräsentation *wotd*-Algorithmus

Der *wotd*-Algorithmus hat die Eigenschaft, dass jeder Knoten während der Konstruktion nach seiner Evaluierung nicht noch einmal berührt wird. Die Baumstruktur wird von der Wurzel bis zu den Blättern, Ebene für Ebene, konstruiert. Durch die Evaluierung eines Knotens wird die vollständige Liste aller seiner Kinder erzeugt. Diese Kinder können alle physisch zusammen gespeichert werden, da keine Einfügeoperationen wie beim Partitionierungsalgorithmus stattfinden. Es bietet sich deshalb an, alle Knoten entsprechend ihrer Konstruktionsreihenfolge hintereinander in einem Array zu speichern (ähnlich wie in einem *Heap*). Dabei wird ein Zeiger auf einen Knoten im Baum durch einen Index im Array beschrieben.
Die Kinder eines Knotens werden in aufeinanderfolgenden Stellen im Array gespeichert, so dass der Zeiger zum rechten Kind entfallen kann. Jedem Knoten wird ein zusätzliches Bit hinzugefügt, welches bestimmt, ob ein Knoten der Letzte einer Kindliste ist. Dieses Bit wird noch zusätzlich im *long*-Wert des linken Indexes gespeichert.
Die meisten Knoten im Suffixbaum sind Blätter, die keine Kinder haben. Anstatt die leere Referenz zu speichern, wird jedem Knoten noch ein zweites Bit hinzugefügt, welches bestimmt, ob der Knoten ein Blatt ist. In Abbildung 4.3 ist die Array-Repräsentation des Suffixbaumes aus Abbildung 4.2 dargestellt. Die inneren Knoten bestehen aus je zwei Werten. Der erste Wert entspricht dem linken Index der Kantenbeschriftung und der zweite dem Index, an dessen Stelle sich das linke Kind befindet. Die Wurzel enthält nur die Referenz auf ihr linkes Kind und muss deshalb nicht gespeichert werden. Der erste Knoten $\overline{C}$ im Beispiel hat die Kantenbeschriftung $(1, 1)$, wobei sich der rechte Index dieser Kantenbeschriftung aus dem Wert 2 an der Stelle 8, der Adresse seines Kindes im Array, durch $2 - 1 = 1$ errechnen lässt. Eine Geschwisterliste in dieser Repräsentation endet, wenn das Bit (*last child bit*) für das letzte Kind dieser Liste gesetzt ist. Das Blatt an der Stelle 7 benötigt, wie alle Blätter, außer dem *Last-Child-Bit* und dem Bit, was es als Blatt kennzeichnet, nur noch den linken Index seiner zugehörigen Kantenbeschriftung, in diesem Falle 10. Der rechte Index ergibt sich mit $n + 1$ aus der Länge des Textes t^+.
Die Speicherplatzanforderungen für jeden inneren Knoten belaufen sich in dieser Repräsentation auf 4 Bytes für einen linken Index (*long*-Wert) der Kantenbeschriftung und 4 Bytes für den Verweis auf das linke Kind (Zeiger), der bei einem Blatt sogar entfällt. Damit ergibt sich bei 8 Bytes für jeden inneren Knoten und 4 Bytes für jedes Blatt ein gesamter Speicherbedarf von maximal $(8 + 4)n = 12n$ Bytes für den

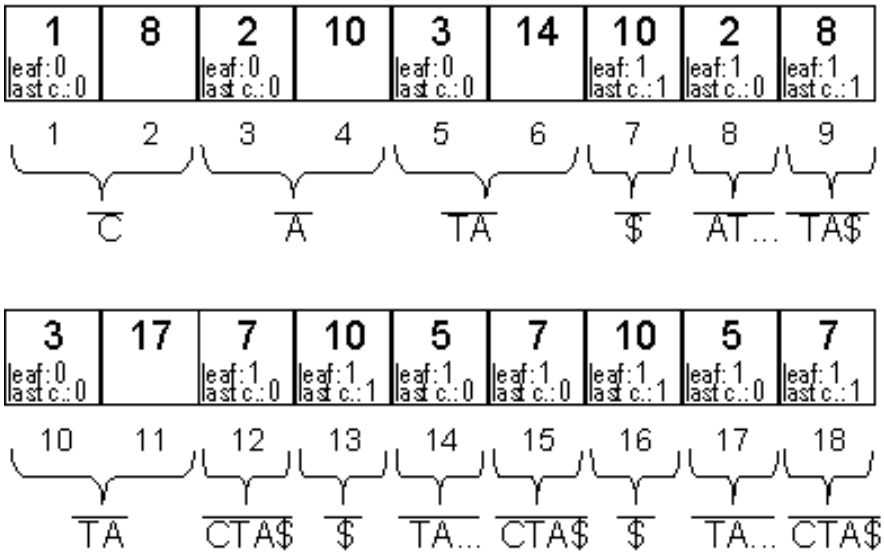


Abbildung 4.3: Repräsentation des Suffixbaumes in einem Array

gesamten Suffixbaum.
Der *wotd*-Algorithmus muss wie auch der Partitionierungsalgorithmus sicherstellen, dass der rechte Index einer Kantenbeschriftung durch den linken Index des linken Kindes errechnet werden kann. Zu diesem Zweck definieren Giegerich *et al.* [23] eine totale Ordnung $\prec$ auf den Kindern eines inneren Knotens.

Definition 6 (Suffix Baum) *Seien* $\overline{uv} \in V(\mathcal{T}(t))$ *und* $\overline{uw} \in V(\mathcal{T}(t))$ *zwei Kinder des gleichen inneren Knotens* $\overline{u} \in I(\mathcal{T}(t))$*, dann gilt:*

$$\overline{uv} \prec \overline{uw} \iff min(leafSet(\overline{uv})) < min(leafSet(\overline{uw})).$$

Bei der Gruppierung, die durch einen Sortieralgorithmus realisiert wird, befindet sich der Suffix mit der kleinsten Nummer stets in der ersten Gruppe. Die Nummer dieses Suffixes bestimmt den linken Index der entsprechenden Kantenbeschriftung im evaluierten Knoten. Entscheidend ist dabei nur die erste Gruppe, aus der ein linkes Kind generiert wird. Bei der Evaluierung eines Knotens und der damit verbundenen Sortierung der Suffixe wird die Ordnung auf den Buchstaben so angepasst, dass der kleinste Buchstabe dieser Ordnung immer dem führenden Zeichen des Suffixes mit der kleinsten Suffixnummer in der zu evaluierenden Gruppe entspricht. Wendet man dieses Verfahren innerhalb des *wotd*-Algorithmus an, so wird die Invariante nicht verletzt (siehe [23] für einen Korrektheitsbeweis).

4.2 Persistente Speicherung von Suffixbäumen

Die persistente Speicherung der Suffixbäume sollte einen schnellen Zugriff auf die Datenstruktur erlauben, um eine Vielzahl von Suchanfragen innerhalb eines kurzen Zeitraums bearbeiten zu können. Zusätzlich müssen die Speicherplatzanforderungen klein gehalten werden, da die zu indizierenden Textmengen sehr groß sind und die Suffixbäume mehr Speicherplatz benötigen als die ohnehin schon sehr großen Textmengen.
Als Format zur persistenten Speicherung der Suffixbäume dient die Repräsentation in einem Array (Abschnitt 4.1.3), die schon der *wotd*-Algorithmus zur Konstruktionszeit verwendet hat. Der Suffixbaum liegt nach der Konstruktion durch den *wotd*-Algorithmus schon in dieser serialisierten Form vor (Abschnitt 4.1.3) und muss nur noch auf die Festplatte geschrieben werden. Dagegen erzeugt der Partitionierungsalgorithmus nur eine Hauptspeicherrepräsentation des Suffixbaumes. Somit muss jeder konstruierte Zweig vor der Speicherung noch in das serielle Format (Abschnitt 4.1.3) umgewandelt werden.
Bei der Speicherung des Suffixbaumes sollten Knoten, auf die während der Suche aufeinanderfolgend zugegriffen wird, räumlich nah gespeichert werden. Befinden sich Knoten, die im Suchvorgang hintereinander benötigt werden, in verschiedenen Festplattenblöcken, so sind mehrere Festplattenzugriffe nötig, um diese Knoten zu bearbeiten.
Der Suchalgorithmus startet beginnend mit der Wurzel des Baumes und fährt dann mit dem Kind aus der Kindliste, welches auf die Suchanfrage passt, fort. Dieser Vorgang wiederholt sich, bis das Suchmuster durchlaufen ist bzw. kein passender Weg mehr im Suffixbaum gefunden wird. Dabei solltendie Kinder eines Knotens räumlich nah gespeichert werden, da der Suchalgorithmus zur Bestimmung eines passenden Kindes die Kindliste von vorne bis zum passenden Kind durchläuft. Die Knoten eines Suffixbaumes werden bei der Suche von oben, der Wurzel, nach unten, Richtung der Blätter, durchlaufen.
Die Positionierungszeit des Lesekopfes hat den größten Anteil an der Zugriffszeit auf einen Festplattenblock. Damit der Lesekopf auf der Festplatte während einer Suchoperation eine möglichst geringe Strecke zurücklegt, sollten die Ebenen des Suffixbaumes von der Wurzel zu den Blättern hintereinander gespeichert werden. Dadurch wird sichergestellt, dass sich der Lesekopf während einer Suche zumindest nur in eine Richtung bewegt.
Die Array-Repräsentation (Abschnitt 4.1.3) des Suffixbaumes weist diese Eigenschaft auf. Die Geschwisterknoten werden nebeneinander gespeichert und Knoten, die sich in den oberen Ebenen des Suffixbaumes befinden, stehen im Array vor den unteren Ebenen. In Abbildung 4.4 ist die Speicherung eines Suffixbaumes dargestellt, in der die Ebenen hintereinander von der Wurzel zu den Blättern gespeichert sind.
Es werden aber üblicherweise nicht nur einzelne Suchanfragen gestellt, sondern sehr

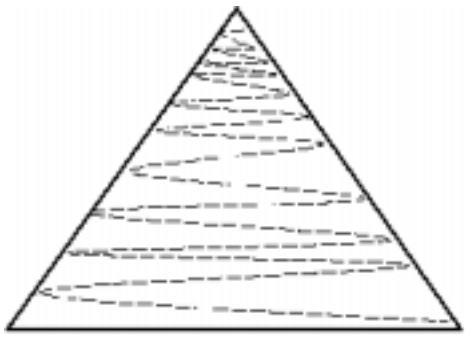


Abbildung 4.4: Die Ebenen eines Suffixbaumes hintereinander gespeichert

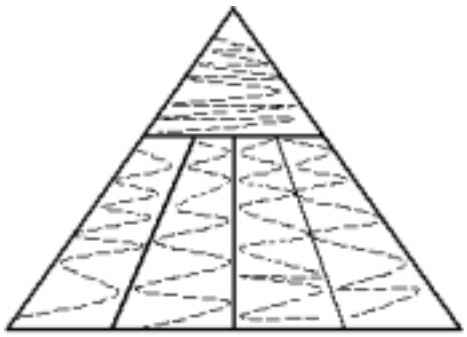


Abbildung 4.5: Geclusterte Speicherung eines Suffixbaumes

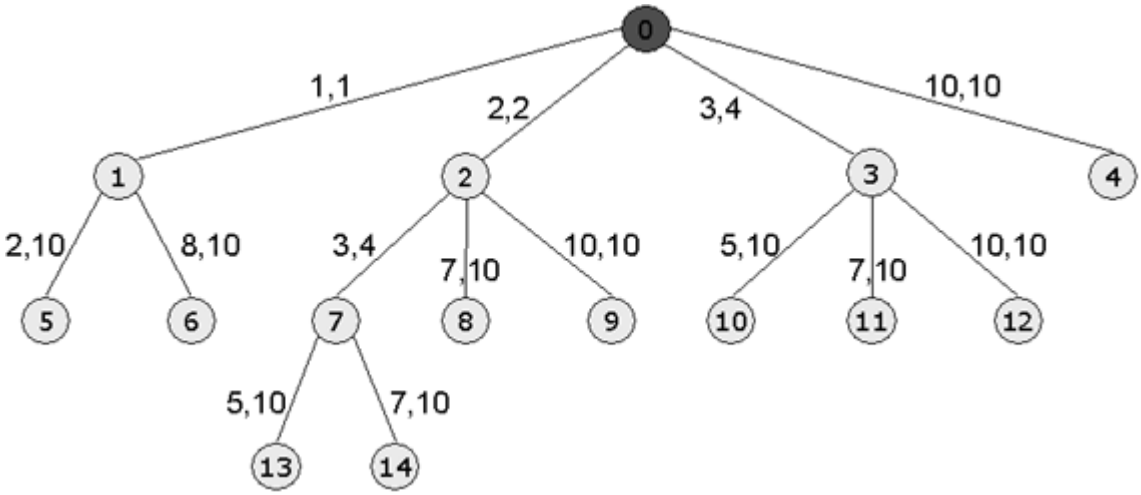
viele in einem kurzen Zeitraum. Jede Suchanfrage beginnt an der Wurzel. Über mehrere Anfragen wird häufig auf die Knoten des Suffixbaumes zugegriffen, die sich in den oberen Ebenen in der Nähe dieser Wurzel befinden. Abbildung 4.5 beschreibt eine *Clusterung* des Suffixbaumes, die das Zugriffsverhalten über viele Suchanfragen berücksichtigt. Die Knoten des "Stumpfes" werden im oberen *Cluster* zusammengefasst. Dieser sollte während der Anfragebearbeitungen möglichst dauerhaft im Hauptspeicher gehalten werden, da über viele Suchanfragen die meisten Zugriffe auf Knoten aus diesem "Stumpf" erfolgen. Die anderen *Cluster* entsprechen Zweigen des Suffixbaumes, die durch Abschneiden des "Stumpfes" entstehen. In ihnen sind die Knoten wiederum ebenenweise hintereinander gespeichert. Unter der Annahme, dass jeder Zweig in einen Festplattenblock passt und der "Stumpf" ständig komplett im Hauptspeicher gehalten werden kann, ergibt sich für jede Suchanfrage maximal ein Festplattenzugriff. Diese *Clusterung* sollte damit zu einer deutlichen Effizienzsteigerung der Suchoperationen führen.

4.3 Die Speicherung in einer relationalen Datenbank

Anfang der siebziger Jahre wurde das relationale Datenmodell konzipiert. Es beruht auf der *mengenorientierten* Verarbeitung von Daten und ist sehr einfach strukturiert. Im wesentlichen gibt es nur flache Tabellen (Relationen), in denen die Zeilen (Tupel) den Datenobjekten entsprechen. In dieser sehr einfachen – fast schon spartanischen – Struktur liegt aber anscheinend der Erfolg der relationalen Datenbanktechnologie begründet, die heute eine marktdominierende Stellung besitzt.
Auf den ersten Blick scheint es sehr gewagt, einen Suffixbaum in einer Tabelle speichern zu wollen, weil das relationale Paradigma keine dynamischen Baumstrukturen berücksichtigt. Die Nutzung des guten Hintergrundspeichermanagements der hoch-

entwickelten relationalen Datenbanksysteme könnte jedoch zu einer Effizienzsteigerung führen.
Da der Suffixbaum eine komplexe Datenstruktur ist, muss er zuerst in Elemente aufgebrochen werden, die dann zu einer Menge zusammengefasst werden können. Der Baum besteht im wesentlichen aus einer Knoten- bzw. Kantenmenge mit zusätzlichen Informationen auf den Knoten bzw. Kanten. Diese Menge muss auf eine Tabelle abgebildet werden. Abbildung 4.6 stellt zwei Tabellen dar, die den über ihnen abgebildeten Suffixbaum repräsentieren. Die Knoten dieses Baumes sind, beginnend mit der Wurzel, durchnummeriert.
Die Array-Repräsentation eines Suffixbaumes (Abb. 4.3) beschreibt die Speicherung von Knoten in einem Array. Diese knotenorientierte Speicherung kann auch in ein relationales Schema übernommen werden. Die linke Tabelle in Abbildung 4.6 orientiert sich an dieser *Left-Child-Right-Sibling*-Repräsentation ($LR-Schema$: $\{\underline{Node : int}, LeftIndex : int, LeftmostChild : int, LastSiblingBit : bit\}$). In diesem Schema werden aber keine inneren Knoten und Blätter unterschieden, da diese Differenzierung zwei Tabellen benötigen würde, so dass die Kindlisten über diese Tabellen verteilt wären und damit bei der Mustersuche ein Performanzverlust zu erwarten wäre. Ein Tupel dieses *LR-Schemas* benötigt pro Knoten vier Attribute. Da das relationale Modell auf der Anwendungsebene keine physische Adressierung erlaubt wird ein Schlüssel zur eindeutigen Identifizierung eines Knotens benötigt. Der *LeftIndex* und das *LastSiblingBit* entsprechen der Array-Repräsentation und das *LeftmostChild* ist die Referenz auf das linke Kind. Dieser Wert entspricht aber nur der logischen Adresse.
Ein anderes Schema wird durch die zweite Tabelle (Abb. 4.6, rechts) dargestellt. Dieses Schema ($Edge-Schema$: $\{\underline{Parent : int, Child : int}, LeftIndex : int\}$) realisiert eine Kantenrepräsentation des Suffixbaumes. Jedes Tupel enthält den Anfangs- und Endknoten einer Kante sowie den linken Index der Kantenbeschriftung. Die Kanten sind entsprechend ihrem Endknoten sortiert.
Beim Vergleich dieser beiden Schemata ist zu bemerken, dass das *Edge-Schema* mit einer Spalte weniger auskommmt. Die Kinder eines Knotens entsprechen keiner Liste (*LR-Schema*), sondern sind implizit durch die Kanten (Parent, Child) gegeben, so dass auf das terminierende Bit verzichtet werden kann. Jedes Tupel im *LR-Schema*, das einem Blatt entspricht, enthält in der Spalte *LeftChild* einen *Null*-Wert. Insgesamt sind das mehr als $\frac{Anzahl\ Tupel}{2}$ *Null*-Werte, da es mehr Blätter als innere Knoten gibt. Dagegen kommen beim *Edge-Schema* in keiner der Spalten *Null*-Werte vor, weil jede Kante immer einen linken Index und einen Anfangs- bzw. Endknoten besitzt. Andere Schemata, die zu jedem Knoten die Menge aller Kinder speichern, enthalten zu viele *Null*-Werte oder sind nicht in Normalform und damit ungeeignet.

Nachteile Der Nachteil einer Abbildung von Suffixbäumen auf Tabellen ist, dass das relationale Paradigma keine rekursiven Strukturen unterstützt. Die Operatio-



Node	Left Index	Left Child	Last Sib. Bit
1	1	5	0
2	2	7	0
3	3	10	0
4	10	Null	1
5	2	Null	0
6	8	Null	1
7	3	13	0
8	7	Null	0
9	10	Null	1
10	5	Null	0
11	7	Null	0
12	10	Null	1
13	5	Null	0
14	7	Null	1

Parent	Child	Left Index
0	1	1
0	2	2
0	3	3
0	4	10
1	5	2
1	6	8
2	7	3
2	8	7
2	9	10
3	10	5
3	11	7
3	12	10
7	13	5
7	14	7

Abbildung 4.6: Abbildung des obigen Suffixbaumes auf Tabellen: knotenorientierte Abbildung (links), kantenorientierte Abbildung (rechts)

nen auf Suffixbäumen (insbesondere die Mustersuche) starten an der Wurzel und gehen rekursiv in die Tiefe des Baumes. Der Zugriff auf einen Knoten ist dabei in der Regel abhängig von seinem Vater, so dass die Knoten der nächsten Ebene (die Kinder) erst bestimmt werden können, wenn der entsprechende Vater feststeht. Die Anzahl der Kinder eines Knotens ist sehr klein, so dass keine Effizienzsteigerung durch die ausgereiften Mengen- bzw. Joinoperationen der relationalen Datenbanken zu erreichen ist.
Neben den konzeptionellen Nachteilen ergeben sich zusätzlich Probleme bei der Speicherung der sehr großen Datenmengen. Vor allem ist die Adressierung in einer relationalen Datenbank nur logisch, über zusätzliche Schlüssel (*LR-Schema*) zu realisieren, so dass der Tabelle eine weitere Spalte hinzugefügt werden muss. Damit werden die ohnehin schon beträchtlichen Speicherplatzanforderungen der Suffixbäume noch erhöht.
Außerdem benötigt das Verfolgen einer logischen Referenz zusätzliche Laufzeit. Angenommen die Indexierung ist durch einen B-Baum realisiert, dann benötigt das Auffinden eines Knotens $log\, n$ Schritte. Damit ist die Komplexität für die Mustersuche auf einer Tabelle insgesamt in $O(|w| log\, n)$, also nicht mehr unabhänging von der Länge des Textes.
Ein weiterer Schwachpunkt ist, dass die Speicherung der zugrundeliegenden (langen) Strings in einer relationalen Datenbank nur über Erweiterungen der relationalen Standarddatentypen erreicht werden kann. Solche Erweiterungen sind zum Beispiel *BLOBs* (**b**inary **l**arge **ob**jects), die es erlauben, Objekte beliebiger Größe und Struktur zu speichern. Die persistente Ausprägung der Suffixbäume und der zugehörigen Strings bleibt dem Anwendungsentwickler aber in jedem Fall verborgen. Er kann nur Einfluss auf die logische, aber nicht auf die physische Struktur der Daten nehmen.

Vorteile Die Vorteile der Benutzung von relationalen Datenbanken sind insbesondere in der schnellen Anwendungsentwicklung für Applikationen, die auf Suffixbäumen basieren, zu sehen. Viele Entwickler sind mit den relationalen Datenbanken vertraut, so dass sie über schon bekannte Schnittstellen leicht auf den Suffixbaum zugreifen können. Sie müssen sich nicht erst mit einem für sie fremden Format und der physischen Speicherung auseinandersetzen.

Andere Möglichkeiten für die Speicherung in einer Datenbank können Erweiterungen wie benutzerdefinierte Typen, bzw. Funktionen, bieten. Diese in neueren Datenbanksystemen (PostgreSQL [40], Oracle 8i [41], MySQL [6], ...) unterstützten benutzerdefinierten Typen erlauben neben der Speicherung der Datenstruktur auch die Speicherung der auf dieser Struktur arbeitenden Programme. Das Suffixbaum-Format und die Operationen können in diesem Fall unabhängig von der Tabellenstruktur entwickelt und dann in die Datenbank integriert werden. Solche Ansätze wurden im Rahmen dieser Arbeit allerdings nicht weiter verfolgt.

Kapitel 5

Vorarbeiten zu den Experimentellen Untersuchungen

Nachdem die Aspekte der Konstruktionsalgorithmen (Kapitel 3) und der persistenten Speicherung (Kapitel 4) von Suffixbäumen intensiv erörtert wurden, sollen nun die experimentellen Untersuchungen, ein weiterer zentraler Aspekt dieser Arbeit, behandelt werden. Um diese durchführen zu können, wurden Prototypen der beschriebenen Algorithmen und persistenten Speichermechanismen entwickelt, die als Basis für die experimentellen Untersuchungen dienten. In Abschnitt 5.1 wird ein Überblick über diese Implementierungen gegeben. Im darauffolgenden Abschnitt 5.2 wird dann auf die Planung der Experimente eingegangen.

5.1 Implementierung

Die Implementierungen wurden in der Programmiersprache C auf Basis des gcc-Compilers angefertigt. C ist als effiziente Möglichkeit zur Entwicklung von zeitkritischen Anwendungen etabliert. Die heute in der Softwareentwicklung vielfach verwendeten objektorientierten Sprachen wie Java bzw. C++ sind keine Alternativen, da sie C im Hinblick auf den Speicherplatzverbrauch und die Laufzeiteffizienz unterlegen sind.
Die Prototypen für die Konstruktionsalgorithmen umfassen den verbesserten Partitionierungsalgorithmus und den *wotd*-Algorithmus. Zur Untersuchung der persistent gespeicherten Suffixbäume wurde die Mustersuche, die klassische Operation auf Suffixbäumen, für die verschiedenen persistenten Repräsentationen implementiert.

Partitionierungsalgorithmus

Zum Partitionierungsalgorithmus existiert nur eine, mir bekannte, Implementierung von Hunt *et al.* [29]. Diese Implementierung wurde in PJama einer Java basierten

Umgebung angefertigt, welche die Aspekte der persistenten Speicherung in die Programmiersprache integriert. Durch diesen proprietären Speichermechanismus ist es nicht möglich, auf die Struktur der gespeicherten Daten Einfluss zu nehmen. Ein weiterer eklatanter Nachteil ist der Speicherplatzverbrauch. Die objektorientierte Speicherung von PJama benötigt zusätzlichen Speicherplatz für jedes persistente Objekt, was bei ohnehin schon speicherkritischen Anwendungen in der Stringverarbeitung nicht tolerierbar ist.
Die hier implementierte Variante basiert auf dem verbesserten Partitionierungsalgorithmus (Abb. 3.3). Allerdings wurde auf die Partitionsverfeinerung und die blockorientierte Speicherung verzichtet, weil bei den Strings aus dem betrachteten Anwendungsgebiet damit zu rechnen ist, dass alle Partitionen gleichmäßig gefüllt sind. Zuerst wird für einen String aus einer Datei *sequence* die Abbildung der Suffixe auf die einzelnen Partitionen berechnet. Diese Informationen werden in einem Partitionsarray, das genau so lang ist wie der String, gespeichert. Jede Stelle in diesem Array enthält einen Verweis auf den jeweils folgenden Suffix einer Partition. Ein zweites Array speichert für alle Partitionen das erste Suffix im Partitionierungsarray. Danach wird für jede Partition der Suffixbaum-Zweig aufgebaut und die einzelnen Zweige in der Konstruktionsreihenfolge hintereinander in eine Datei geschrieben. Das zweite Array enthält nach der Konstruktion aller Zweige die relativen Anfangsadressen aller Partitionen in dieser Datei. Diese Informationen werden dann für die Konstruktion des Baumstumpfes benötigt, der in einer separaten Datei gespeichert wird.
Durch die Konstruktion des Suffixbaumes entstehen zwei neue Dateien. Die Datei *sequence.stt* enthält den Suffixbaum-Stumpf und die Datei *sequence.sft* enthält alle Partitionen unterhalb des Stumpfes. Die Blätter des Stumpfes verweisen auf die Zweige in der *.sft*-Datei. Durch diese Struktur realisiert der Partitionierungsalgorithmus die *Clusterung* des Suffixbaumes (Abb. 4.5).

wotd-Algorithmus

Eine Implementierung des *wotd*-Algorithmus von Stefan Kurtz gilt als eine der schnellsten zur Konstruktion von Suffixbäumen. Für die Vergleichbarkeit der implementierten Algorithmen ist es allerdings von großer Bedeutung, dass diese gleiche Speicherzugriffsmechanismen verwenden und eine ähnliche Codestruktur aufweisen, so dass keiner der Algorithmen nur durch eine geschickte Codeoptimierung eine schnellere Laufzeit erreicht. Aus diesem Grund wurde auch der *wotd*-Algorithmus von mir selber implementiert.
Die *wotd*-Implementierung beginnt mit der Evaluierung der Wurzel. Alle Kinder der Wurzel werden generiert und in einem Array gespeichert, was den späteren Suffixbaum enthält. Das Array wird dann von vorn nach hinten durchlaufen. Dabei werden zu jedem noch nicht evaluierten Knoten alle Kinder generiert und an die nächsten

freien Stellen im Array geschrieben, so dass die einzelnen Ebenen des Suffixbaumes nach der Konstruktion hintereinander im Array liegen. Dieses Array wird dann nur noch auf die Festplatte geschrieben. Dadurch wird die *nicht-geclusterte* Speicherung des Suffixbaumes realisiert.

Ukkonen-Algorithmus

Zum Vergleich mit den Linearzeitalgorithmen war auch eine Untersuchung des Ukkonen-Algorithmus notwendig. Dafür stand mir eine Implementierung von Jens Stoye und Stefan Kurtz zur Verfügung, die einen Suffixbaum in einer Hauptspeicherrepräsentation zurückliefert. Diese Implementierung wurde von mir durch Mechanismen zur persistenten Speicherung erweitert.

Suche auf persistenten Suffixbäumen

Die Mustersuche wurde von mir sowohl für *geclusterte* als auch für *nicht-geclusterte* persistente Suffixbäume implementiert. Zusätzlich habe ich eine Abbildung der Suffixbäume auf eine relationale Datenbank gemäß dem *LR-Schema* realisiert. Ich habe mich für dieses Schema entschieden, weil es dem Array-Format entspricht und eine gute Vergleichbarkeit der einzelnen Implementierungen eine wichtige Voraussetzung für die experimentellen Untersuchungen ist.
Alternativen für eine relationale Datenbank waren Oracle [41], PostgreSQL [40] und MySQL [6]. Oracle ist die einzige kommerzielle Datenbank, die am Institut verfügbar ist und die anderen sind die am meisten genutzten freien Datenbanken. Ich möchte, dass meine Implementierung frei einsetzbar ist und habe mich deshalb dazu entschieden, auf Oracle zu verzichten. Von den verbleibenden Datenbanken gilt MySQL als die schlankere und damit schnellere Datenbank. Sie wird zwar von vielen Fachleuten nicht als vollständige Datenbank angesehen, da sie keine Transaktionen unterstützt, dies spielt für die hier betrachteten Anwendungen aber keine Rolle, weil ein Suffixbaum nur einmal konstruiert und danach nicht mehr verändert wird.
Bevor der Suffixbaum in der Datenbank gespeichert werden kann, wird er zunächst mit dem *wotd*-Algorithmus konstruiert und dann in eine mit Tabulatoren getrennte Datei exportiert. Diese wird später in die entsprechende Datenbanktabelle geladen. Für alle Suchvarianten wurde eine Suchschnittstelle implementiert, die vor den eigentlichen Suchoperationen zuerst geladen werden muss. Sie besteht aus dem indizierten String und wird, um für die verschiedenen Varianten spezifische Daten, ergänzt.
Bei der Suche auf dem *nicht-geclusterten* Suffixbaum wird die Schnittstelle zum Suffixbaum durch eine in C bewährte Struktur namens *mmap* realisiert. Dieses *mmap* überträgt dem virtuellen Speichermanagement des Betriebssystems den Zugriff auf die persistenten Daten . Zur Implementierung stand mir eine Bibliothek von Stefan

Kurtz zur Verfügung, die dieses *mmap* kapselt.
Die gleiche Zugriffsvariante wird für die Zweige bei der Suche auf dem *geclusterten* Suffixbaum genutzt. Der Stumpf wird allerdings beim Laden der Schnittstelle vollständig in den Hauptspeicher gelesen.
Die Schnittstelle zu dem in der Datenbank gespeicherten Suffixbaum besteht neben dem String nur aus der Datenbankverbindung, die beim Laden der Schnittstelle hergestellt wird.

5.2 Planung der Experimente

Nachdem nun die wichtigsten Aspekte der implementierten Prototypen erläutert wurden, sollen jetzt die Grundlagen für die Durchführung der Experimente geschaffen werden. Die Experimente sollen sich besonders auf die Bereiche Genom-Datenbanken und Information Retrieval beziehen und umfassen die Konstruktion von Suffixbäumen sowie, als wichtigste Operation, die Mustersuche. Diese beiden Bereiche sollen hier noch genauer betrachtet werden:

- **Untersuchung der Konstruktionsalgorithmen**
  Die zu überprüfenden Konstruktionsalgorithmen wurden in Kapitel 3 intensiv behandelt und die Implementierung der Prototypen des verbesserten Partitionierungsalgorithmus und des *wotd*-Algorithmus in Abschnitt 5.1 kurz beschrieben. Zum Vergleich soll auch ein Linearzeitalgorithmus in die Untersuchungen einbezogen werden.
  Weiterhin werden zum einen die Einstellungen der Algorithmen wie Partitionierungstiefe bzw. Sortieralgorithmus untersucht und zum anderen sollen sie bezüglich der Struktur der Eingabetexte betrachtet werden. Ein sehr wichtiger Aspekt dabei ist die Untersuchung von langen Eingabetexten, wenn die Suffixbaum-Instanziierungen die Hauptspeichergröße übersteigt. Basierend auf den Ergebnissen sollen Aussagen bezüglich der Anwendbarkeit der verschiedenen Konstruktionsalgorithmen für die hier betrachteten Gebiete gemacht werden.

- **Untersuchung der Speicherplatzanforderungen** Die Suffixbaum-Formate liegen nach der Konstruktion in drei verschiedenen Varianten vor. Die Erste ist die Repräsentation, die nach der Konstruktion durch den *wotd*-Algorithmus vorliegt (*nicht-geclustert*). Die Zweite ist der *geclusterte* Suffixbaum der durch den Partitionierungsalgorithmus erzeugt wird und die dritte Variante bezieht sich auf den in einer relationalen Datenbank gespeicherten Suffixbaum (MySQL). Für diese drei Formate werden die Speicherplatzanforderungen überprüft.

- **Untersuchung der Mustersuche auf den persistenten gespeicherten Suffixbäumen**

Die Mustersuche wird ebenso wie die Speicherplatzanforderungen auf den drei verschiedenen Repräsentationen untersucht.

Für die beschriebenen Bereiche gibt es verschiedene Einflussgrößen, die Auswirkungen auf die gemessenen Werte haben. Diese werden im folgenden Abschnitt näher betrachtet.

5.2.1 Einflussgrößen

Die Einflussparameter können vier Bereichen zugeordnet werden – den Umgebungs-, Programm-, Eingabe- oder Datenbankparametern.

- **Umgebungsparameter**
  Diese Parameter umfassen die das Programm "umgebende" Systemplattform, bestehend aus Hardware und Betriebssystem. Diese wird für die Untersuchungen als unveränderlich angenommen.

  - *Hardware*
    Die zugrundeliegende Hardware bezeichnet die Gesamtheit sämtlicher technischer Bauteile des Computers, auf dem die Experimente durchgeführt werden. Die wichtigsten Parameter dieser Hardware sind die CPU-Leistung bzw. die Größe und Zugriffszeiten der Speicherbausteine.
  - *Betriebssystem*
    Das Betriebssystem fungiert als die Schnittstelle zur Hardware. Es stellt dem Programm Resourcen in Form von Speicherplatz oder CPU-Zeit zur Verfügung. Übersteigt der Speicherplatzbedarf der Suffixbäume die Hauptspeichergröße, so ist das virtuelle Speichermanagement des Betriebssystem für die Auslagerung der Daten verantwortlich.

- **Programmparameter**
  Diese Parameter umfassen die veränderlichen Einstellungen, die für die Programme gewählt werden können. Sie sind für die unterschiedlichen Algorithmen verschieden.

  - *Partitionierungstiefe*
    Die Tiefe der Ebene, auf der beim Partitionierungsalgorithmus die in separaten Blöcken gespeicherten Zweige des Suffixbaumes beginnen, spielt nicht nur bei der Konstruktion eine Rolle, sondern hat direkten Einfluss auf die persistente Speicherung. Der Stumpf des Suffixbaumes wird separat gespeichert und bei der Suche immer im Hauptspeicher gehalten. Die Partitionierungstiefe darf nicht zu klein gewählt werden, damit die abgeschnittenen Zweige des Suffixbaumes nicht zu groß werden. Sie sollte

aber ebenso nicht zu groß gewählt werden, um zu gewährleisten, dass der Stumpf vollständig in den Hauptspeicher passt.

- *Sortierung innerhalb des wotd-Algorithmus*
  Der verwendete Sortieralgorithmus zur Gruppierung der Suffixe im *wotd*-Algorithmus wirkt sich nur auf die Schnelligkeit der Konstruktion aus, nicht aber auf die persistente Speicherung der Daten. In den Experimenten werden das *Countingsort*, das *Quicksort* und das von mir entwickelte *Scansort* untersucht.

- **Eingabeparameter**
  Die Eingabeparameter umfassen sämtliche veränderlichen Größen der zu indizierenden Texte.

  - *Länge des Textes*
    Die Längen von DNA- bzw. Protein-Sequenzen, die durch Suffixbäume indiziert werden sollen, nehmen sehr große Werte an und auch im Information Retrieval hat die Länge der zu indizierenden Texte zugenommen. Ein wichtiges Kriterium für die Algorithmen ist dabei ihr Verhalten bei Suffixbäumen, die nicht mehr komplett im Hauptspeicher konstruiert werden können.
  - *Größe des Alphabetes*
    Je nach Anwendungsgebiet können die Alphabetgrößen der Eingabetexte schwanken. In der Genetik, wo viele DNA-Sequenzen untersucht werden, beträgt deren Alphabetgröße 4, während natürlichsprachige Texte (deutsch) aus ca. 90 Zeichen bestehen.
  - *Struktur des Textes*
    Die Struktur des Textes hängt im wesentlichen vom Anwendungsgebiet ab. DNA-Sequenzen haben beispielsweise keine wortbasierte Struktur wie der Quellcode von Programmen. Dieser kann beispielsweise durch Programmierumgebungen generiert werden, so dass sehr viele sich wiederholende Muster auftreten, was zu einer Ungleichgewichtung der Baumstruktur führen kann.

- **Datenbankparameter**
  Diese Parameter umfassen das Datenbanksystem und seine Konfiguration und sind nur für die Suche in der relationalen Datenbank von Bedeutung.

  - *Datenbanksystem*
    Das Datenbanksystem (MySQL) wird für meine Untersuchungen als unveränderlich angenommen.

– *Datenbankkonfiguration*
In einer relationalen Datenbank können sehr viele Parameter zur Optimierung der Anfragen und der Speicherung eingestellt werden. Ich werde für meine Experimente die Standardkonfiguration des benutzten Datenbanksystems verwenden.

5.2.2 Messgröße

Nachdem nun die Einflussparameter beschrieben wurden, soll jetzt erörtert werden, was im Hinblick auf die praktische Relevanz die geeigneten Messgrößen für die Experimente sind. Diese betreffen die Laufzeiten der Algorithmen und die Speicherplatzanforderungen der persistenten Formate.

Zeit

In der Praxis interessiert den Benutzer nur, wie lange er warten muss bis das Programm eine Aufgabe verrichtet hat. Deshalb ist auch die gesamte Ausführungszeit, vom Starten bis zur Terminierung des Programmes, die Messgröße, auf die ich mich in den Experimenten beziehe. Die Gesamtlaufzeit setzt sich aus der reinen Rechenzeit (CPU-Zeit), der Zeit für die Speicherzugriffe und der Zeit zusammen, die das System für andere Aufgaben benötigt (z.B. Betriebssystemprozesse).
Die CPU-Zeit des Programmes ist die Dauer, die der Prozessor zur Ausführung benötigt. Diese Zeit ist unabhängig von anderen gleichzeitig ausgeführten Prozessen und wird deshalb vielfach zur Laufzeitmessung verwendet. Sie spielt allerdings für diese Arbeit nur als Teil der Gesamtlaufzeit eine Rolle, weil die CPU-Zeit die Zeit für die Festplattenzugriffe nicht beinhaltet, diese aber einen großen Teil der Gesamtlaufzeit ausmacht.
Die Hintergrundspeicherzugriffe haben, insbesondere in diesen Experimenten, einen sehr großen Anteil an der Gesamtlaufzeit, weil sie sehr langsam sind und dieses bei den Algorithmen, die ihre Daten nicht nur im Hauptspeicher halten, starke Auswirkungen auf die Gesamtlaufzeit hat. Beim Zugriff auf einen Festplattenblock sind zwei Größen zu beachten – die Positionierungszeit des Lesekopfes und die Transferrate. Die meiste Zeit für das Lesen bzw. Schreiben eines Blockes nimmt auf heutigen Festplattentechnologien die Positionierung des Lesekopfes ein. Sie ist abhängig von der Lokalität der gespeicherten Daten. Die Transferrate ist von dieser Lokalität unabhängig und spezifisch für jede Festplatte.
Weitere Einflüsse auf die Gesamtlaufzeit ergeben sich aus dem Dateisystem und der Seitengröße des Betriebssystems oder aus der Block- und Puffergröße der Festplatte. An dieser Stelle soll jedoch nicht näher darauf eingegangen werden.
Bei den Untersuchungen sollen externe Einflüsse, die eine Verzerrung der Testergebnisse hervorrufen, weitgehend vermieden werden. Trotzdem müssen sie bei der

Analyse der Gesamtlaufzeiten immer miteinbezogen werden.
Die Messung der Gesamtlaufzeit erfolgt in den relevanten Abschnitten der Implementierungen mit Hilfe der C-Funktion *time()*. Auf Grund des hohen Zeitaufwandes für die Experimente konnte aber jeweils nur ein Durchlauf der Programme gemessen werden.

Speicherplatz

Neben der Gesamtlaufzeit sind die Speicherplatzanforderungen der Suffixbäume ein wichtiger Gesichtspunkt. Diese wurden schon in Kapitel 4 genauer besprochen. Allerdings wurde dort nur der maximale Speicherplatzaufwand erörtert. Die Suffixbäume, der für die Praxis relevanten Texte, benötigen weniger Speicherplatz. Deshalb werden in Abschnitt 6.3 die Speicherplatzanforderungen der Suffixbäume, der in der Praxis verwendeten Strings, dargestellt. Diese Werte werden üblicherweise in Bytes pro Eingabezeichen angegeben.

Kapitel 6

Experimentelle Untersuchungen

Dieser Abschnitt beschreibt einleitend die Methodik der experimentellen Untersuchungen. Im Anschluss daran werden die Untersuchungen der Konstruktionsalgorithmen und der Mustersuchen erläutert und die Ergebnisse vorgestellt. Dabei erfolgt zu jedem Experiment eine Analyse und Auswertung der Ergebnisse.

6.1 Methodik der Untersuchungen

Die Untersuchungen wurden auf einem Computer mit folgenden Umgebungsparametern durchgeführt:

CPU:	266 MHz, Intel Pentium II (Klamath); 512 KB L1-Cache
Hauptspeicher:	128 MB RAM
Festplatte:	9,5 GB
Betriebssystem:	Linux (Suse Distribution 7.2), Kernelversion 2.4.4

Die Programme wurden mit dem gcc-Compiler (Version 2.95.3) und der Optimierungsoption "-O3" übersetzt. Parallel zum Testbetrieb liefen nur niedrig prioritäre Systemprozesse.
Für die Konstruktionsalgorithmen wurde der komplette Zeitabschnitt beginnend mit dem Laden des Strings aus der Datei bis zur persistenten Speicherung aller Daten gemessen. Für die Suchoperationen wurde zuerst die Suchschnittstelle mit den relevanten Suchdaten geladen, bevor dann die Zeit für die wirklichen Suchen gemessen wurde.
Der Quellcode der implementierten Programme ist unter `http://www.inf.fu-berlin.de/~shuerman/Diplom/SuffixTree/` zu finden.

6.2 Untersuchungen der Konstruktionsalgorithmen

Die Konstruktionsalgorithmen sollen auf ihre Anwendbarkeit in den Gebieten Genom-Datenbanken und Information-Retrieval überprüft werden. Die Strings, die in diesen Gebieten indiziert werden, unterscheiden sich besonders durch die Alphabetgröße und ihre Struktur, die eine mathematisch schwer zu fassende Größe ist. Das Hauptproblem, was die in dieser Arbeit relevanten Gebiete eint, ist die große Menge an langen Textdaten.
Um eine Basis für die praktisch relevanten Untersuchungen zu finden, sollen die Konstruktionsalgorithmen im Hinblick auf die Stringlänge und die Alphabetgröße betrachtet und darauf aufbauend, Strings unterschiedlicher Struktur untersucht werden.
Die verschiedenen Algorithmen besitzen außer den Eingabeparametern noch spezifische Programmparameter wie die Partitionierungstiefe oder den Sortieralgorithmus. Zunächst werden der Partitionierungsalgorithmus und der *wotd*-Algorithmus bezüglich der optimalen Einstellungen der Programmparameter untersucht und danach miteinander verglichen.
Die Basisuntersuchungen wurden auf zufällig erzeugten Strings mit unterschiedlicher Alphabetgröße durchgeführt. Die Buchstaben der Strings wurden dabei unabhängig voneinander, gleichverteilt erzeugt (Bernoulli-Modell).

6.2.1 Untersuchungen des verbesserten Partitionierungsalgorithmus

Der Programmparameter des Partitionierungsalgorithmus ist die Tiefe auf der die Zweige des Suffixbaumes abgeschnitten werden. In Abschnitt 3.2 wurden die theoretischen Analysen des verbesserten Partitionierungsalgorithmus vorgestellt. Nun soll die praktische Relevanz dieses Algorithmus bezüglich der Eingabeparameter Stringlänge und Alphabetgröße untersucht werden.

Vergleich verschiedener Partitionierungstiefen

Wie schon vielfach erwähnt wurde, ist die Länge eines Strings der kritische Parameter bei den hier untersuchten Gebieten.
Der Partitionierungsalgorithmus wird mit Partitionierungstiefen zwischen 2 und 10 bei einem Alphabet der Größe 4 und Partitionierungstiefen zwischen 1 und 3 bei einem Alphabet der Größe 90 untersucht. Die Längen der zufällig erzeugten Strings bewegen sich zwischen 1 Million und 30 Millionen Zeichen.
Mit zunehmender Partitionierungstiefe und damit mehr Partitionen ist zu erwarten, dass der Algorithmus schneller arbeitet, bis dann eine Sättigung und danach sogar eine Verschlechterung eintritt, weil der Suffixbaum-Stumpf zu groß wird. Die Laufzeit

wird stark ansteigen, sobald die Partitionierungsdaten und der größte Suffixbaum-Zweig nicht mehr in den Hauptspeicher passen.

Ergebnis In Abbildung 6.1 sind die Laufzeiten des Partitionierungsalgorithmus bei einer Alphabetgröße 4 graphisch dargestellt. Die obere Graphik veranschaulicht das Laufzeitverhalten in den unteren Laufzeitbereichen, während die untere Graphik das Verhalten an den Rändern der Hauptspeichergröße visualisiert, wenn nämlich die Daten zur Konstruktion des Suffixbaumes nicht mehr komplett im Hauptspeicher gehalten werden können. Bei Partitionierungstiefen von 2 bis 8 und einer Stringlänge von 1 Million Zeichen liegen die Laufzeiten im Bereich von 4 (Tiefe 6) bis 12 (Tiefe 2) Sekunden. Nur die Partitionierungstiefe 10 sticht mit der Laufzeit von 25 Sekunden gegenüber den anderen heraus. Als beste Werte für die Partitionierungstiefen bei Stringlängen bis zu 23 Millionen Zeichen erweisen sich die Tiefen 6, 7 und 8. Die Laufzeiten bei der Partitionierungstiefe 7 sind nicht in Abb. 6.1 dargestellt. Sie weichen allerdings nur um 1 bis 2% von denen der Partitionierungstiefen 6 und 8 ab.
Alle Laufzeiten wachsen etwas stärker als proportional zur Stringlänge an. Dabei ist das Wachstum bei größeren Partitionierungstiefen etwas schwächer als bei kleineren. Bei den Partitionierungstiefen zwischen 4 und 10 ändert sich dieses Verhalten ab einer Stringlänge von 22 Millionen Zeichen. Dabei steigen die Laufzeiten scheinbar exponentiell an. Bei der Partitionierungstiefe 2 unterscheidet sich das Laufzeitverhalten von den übrigen Partitionierungstiefen deutlicher. Anfangs steigt die Laufzeit stärker an als bei den größeren Partitionierungstiefen, was sich sogar ab einer Stringlänge von 17 Millionen Zeichen noch beschleunigt. Im Gegensatz zu den größeren Partitionierungstiefen setzt aber nicht das scheinbar exponentielle Wachstum ab einer bestimmten Stringlänge ein, so dass der Suffixbaum auch noch bei Stringlängen von bis zu 30 Millionen Zeichen in weniger als 40 Minuten zu konstruieren ist. Bei den übrigen Partitionierungstiefen mussten die Experimente schon bei 26 Millionen Zeichen nach mehr als 6 Stunden pro Konstruktion aus Zeitgründen abgebrochen werden.
Wie in Abbildung 6.1 für die Alphabetgröße 4 sind in Abbildung 6.2 auch die Laufzeiten des Partitionierungsalgorithmus mit einer Alphabetgröße 90 dargestellt. Die obere Graphik zeigt wieder das Verhalten in den unteren Laufzeitbereichen, während die untere Graphik das Verhalten an den Rändern des Hauptspeichers veranschaulicht. Mit der Partitionierungstiefe 3 liegt bei einer Stringlänge von 1 Million Zeichen die anfängliche Laufzeit bei 26 Sekunden und deutlich über denen, bei den Partitionierungstiefen 1 (10 Sek.) und 2 (4 Sek.). Der weitere Anstieg der Laufzeitkurve ist bei der Partitionierungstiefe 3 etwas schwächer als bei der Partitionierungstiefe 2, so dass sich die Laufzeitkurven bei einer Stringlänge von 19 Millionen Zeichen schneiden. Die Laufzeiten mit der Partitionierungstiefe 1 steigen anfangs (bis 23 Mill. Zeichen) sehr viel stärker an als die mit den Partitionierungstiefen 2 und 3, bis bei

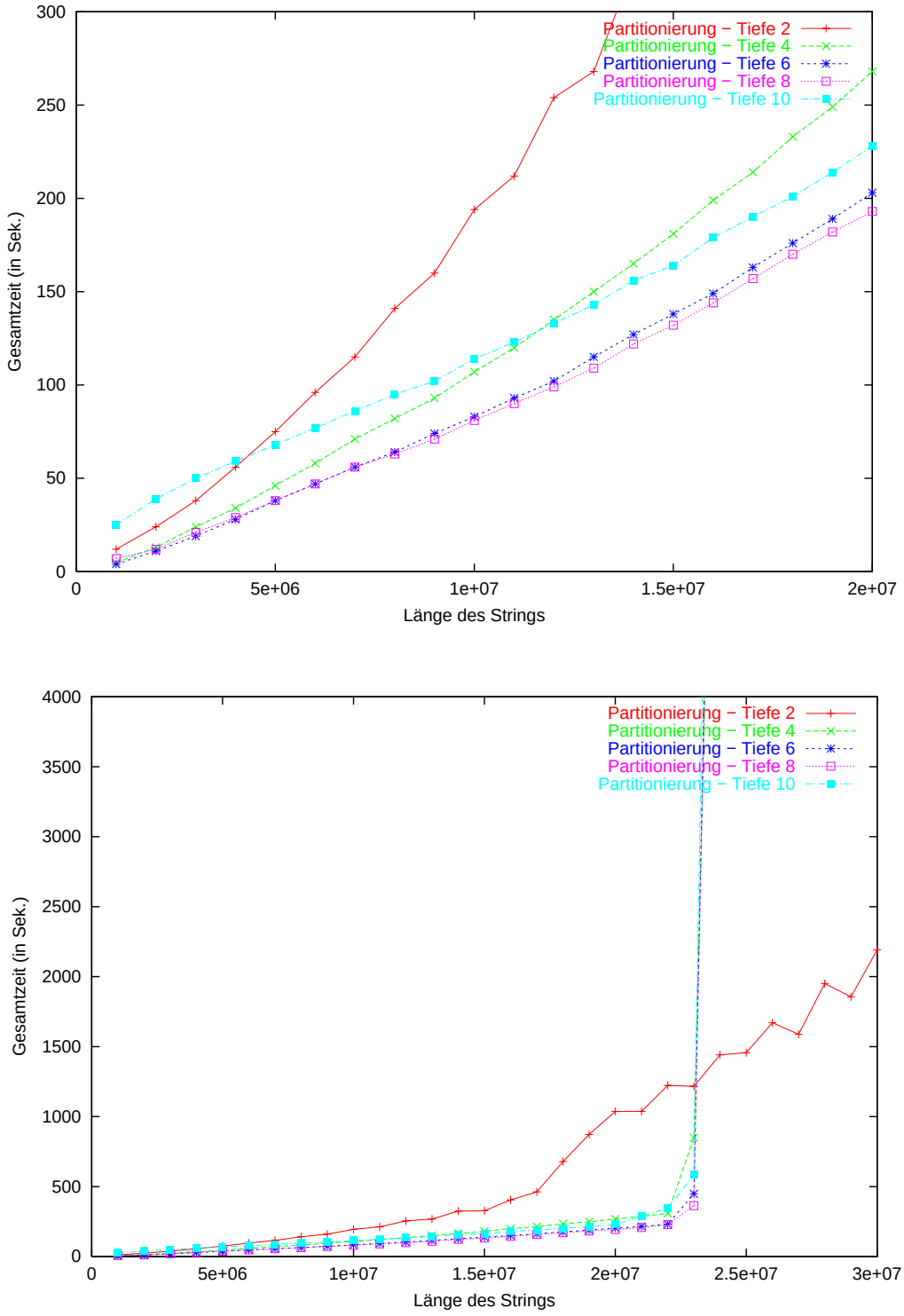


Abbildung 6.1: Konstruktionszeiten des Partitionierungsalgorithmus bei einer Alphabetgröße 4 für verschiedene Partitionierungstiefen bei linear wachsender Stringlänge

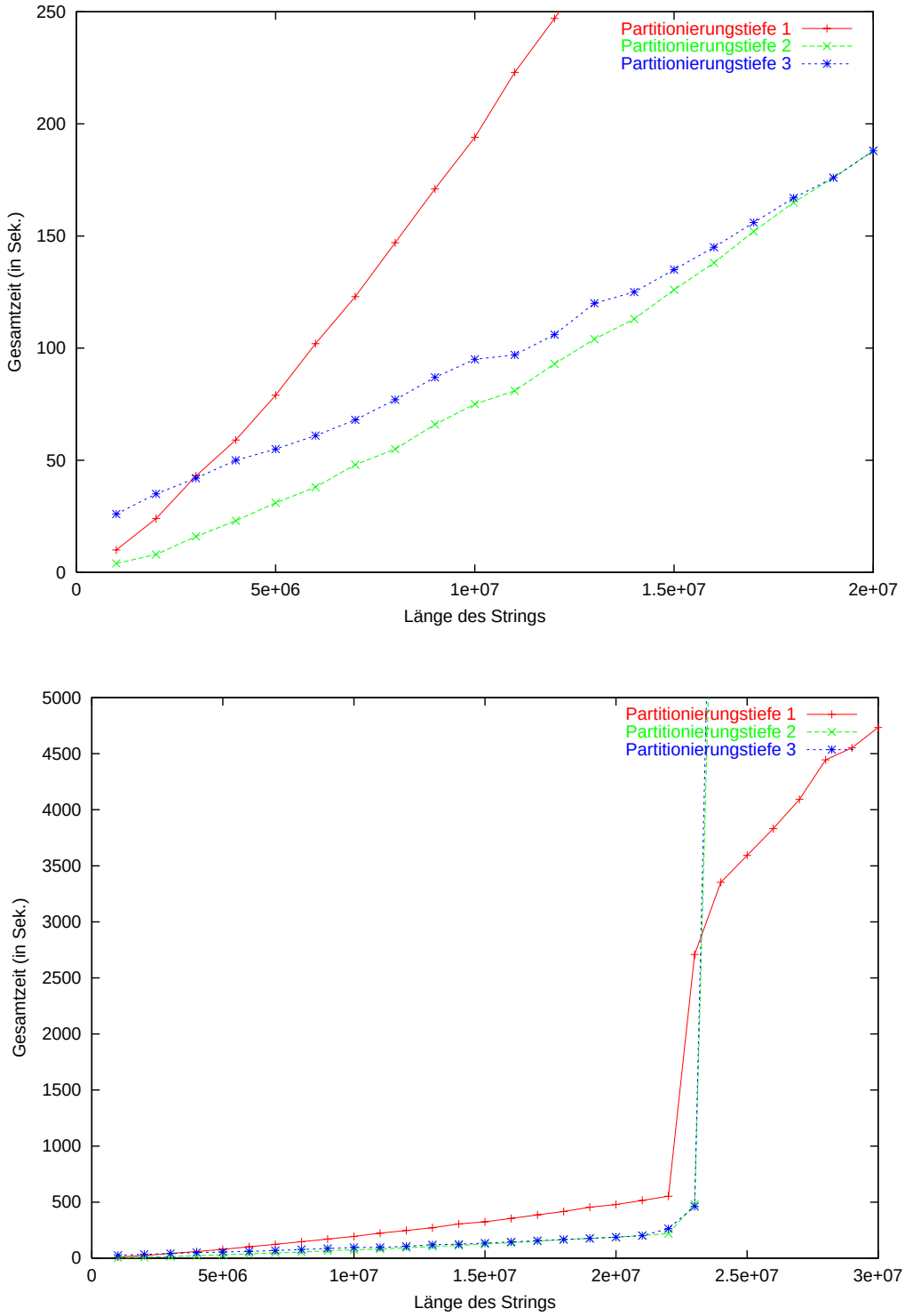


Abbildung 6.2: Konstruktionszeiten des Partitionierungsalgorithmus bei einer Alphabetgröße 90 für verschiedene Partitionierungstiefen bei linear wachsender Stringlänge

diesen größeren Partitionierungstiefen, wie schon bei der Alphabetgröße 4 zu sehen war, ein starker Laufzeitanstieg einsetzt.

Analyse Die anfänglich längere Laufzeit bei größeren Partitionierungstiefen lässt sich durch den höheren Aufwand für die Verwaltung der Partitionen erklären. Die Anzahl der Partitionen steigt exponentiell zur Partitionierungstiefe an, so dass bei der Tiefe 10 und der Alphabetgröße 4 insgesamt $4^{10} = 1048576$ Partitionen verarbeitet werden müssen. Bei einer großen Partitionierungstiefe wird also ein entsprechend hoher Anteil der Gesamtlaufzeit nur zur Verwaltung der Partitionen aufgewendet.
Die Laufzeitkurve bei den kleineren Partitionierungstiefen wächst schneller an als bei größeren, weil die Füllung der einzelnen Partitionen bei linear wachsender Stringlänge und logarithmisch kleinerer Partitionsanzahl viel schneller zunimmt. Wenn die Partitionierungstiefe jedoch zu groß gewählt wird, ist die Laufzeit für die Verwaltung der vielen Partitionen und die Konstruktion des größeren Suffixbaum-Stumpfes zu hoch, um durch die schnellere Konstruktionszeit für die Suffixbaum-Zweige kompensiert zu werden.
Das sehr starke Wachstum der Konstruktionszeiten für die größeren Partitionierungstiefen (≥ 4 bei Alphabetgröße 4 bzw. ≥ 2 bei Alphabetgröße 90) ab einer Stringlänge von ca. 22 Millionen Zeichen, ist durch die Hauptspeicherbegrenzung zu erklären. Ab dieser Stringlänge passen die Partitionierungsdaten und der größte zu konstruierende Suffixbaum-Zweig nicht mehr in den Hauptspeicher, so dass das Betriebssystem anfängt Daten auszulagern. Dabei entfällt der größte Speicherplatzbedarf (4n Bytes) auf das Array, was die Abbildung der Suffixe auf die Partitionen speichert. Der restliche Hauptspeicher wird vom String selber (n Bytes), dem Array, welches alle Partitionen beinhaltet ($8 \times |\Sigma|^{P.Tiefe}$ Bytes) und dem aktuell zu konstruierenden Suffixbaum-Zweig belegt. Der Laufzeitanstieg an den Grenzen des Hauptspeichers tritt für die kleinen Partitionierungstiefen (2 bzw. 1) schon sehr viel früher ein, da die Suffixbaum-Zweige größer werden.
Grund für diesen starken Anstieg sind nicht etwa die Zugriffe auf die Suffixbaum-Zweige, sondern die Art der Speicherung der einzelnen Partitionen. Bei der Konstruktion eines Zweiges muss das Array, was die Abbildung der Suffixe auf die Partitionen speichert, von vorne nach hinten durchlaufen werden, um alle zu einer Partition gehörigen Suffixe ausfindig zu machen. Wenn dieses Array nun vom Betriebssystem ausgelagert werden muss, so finden bei kleineren Partitionierungstiefen, wegen der weitaus kleineren Anzahl an Partitionen, sehr viel weniger Lesezugriffe auf dieses Array und damit auf den Hintergrundspeicher statt. Anhand der unteren Graphik von Abbildung 6.2 lässt sich dieses Verhalten deutlich nachvollziehen. Bei der Partitionierungstiefe 1 macht die Laufzeitkurve bei einer Stringlänge von 22 Millionen Zeichen einen Sprung von 552 auf 2709 Sekunden und flacht dann wieder ab. Dieser Sprung tritt genau dort auf, wo das Betriebssystem die Partitionierungsdaten auslagern muss und damit jeder Block mehrmals von der Festplatte gelesen wird.

Zusammenfassend lässt sich somit feststellen, dass für die Partitionierungstiefen auf Sequenzen mittlerer Länge ein Gleichgewicht zwischen der Größe des Suffixbaum-Stumpfes und der Größe der einzelnen Suffixbaum-Zweige zu finden ist, um eine für diesen Algorithmus optimale Laufzeit zu erhalten. Wachsen dagegen die zur Konstruktion notwendigen Speicherplatzanforderungen über die Hauptspeichergröße, so sind kleine Partitionstiefen effizienter.

Unterschiedliche Alphabetgrößen

Die Alphabetgröße des Eingabestrings hat einen direkten Einfluss auf die Struktur des resultierenden Suffixbaumes. Von ihr hängt der erwartete Verzweigungsgrad (Anzahl der Kinder) der Knoten ab.
Nachdem schon unterschiedliche Einstellungen der Partitionierungstiefe auf den Alphabetgrößen 4 und 90 untersucht wurden, soll jetzt die Auswirkung der Alphabetgröße bei gleichbleibender Anzahl von Partitionen überprüft werden.
Es wurden zufällig erzeugte Strings verschiedener Länge über Alphabeten verschiedener Größe untersucht. Diese Strings haben eine Länge von 5, 10, 15 bzw. 20 Millionen Zeichen und die Alphabetgrößen sind 4, 8, 16 und 64. Andere Alphabetgrößen konnten nicht untersucht werden, da die Partitionierungstiefe für jedes Alphabet immer einer bestimmten Anzahl von Partitionen entsprechen muss, um eine Vergleichbarkeit zu gewährleisten. Ich bin dabei von 4096 Partitionen ausgegangen, die beispielsweise bei einer Alphabetgröße 4 durch eine Partitionierungstiefe von 6 erreicht werden. Für die anderen Alphabete wurden die Partitionierungstiefen auch entsprechend dieser Anzahl der Partitionen gewählt.
Für dieses Experiment ist zu erwarten, dass der Partitionierungsalgorithmus für mittelgroße Alphabete schneller läuft als für kleine und große. Dieses ist einerseits abhängig von dem Verzeigungsgrad der Knoten, weil bei den Einfügeoperationen die Kindlisten von vielen Knoten nach dem passenden Element durchsucht werden müssen und die Laufzeit für diesen Vorgang mit dem erwarteten Knotengrad steigt, der wiederum von der Alphabetgröße abhängt. Andererseits ist die Wahrscheinlichkeit für lange sich wiederholende Teilstrings bei größeren Alphabeten kleiner als bei denen geringerer Größe. Die optimale Alphabetgröße sollte deshalb im mittleren Bereich liegen.

Ergebnis Die Graphik in Abbildung 6.3 visualisiert die Ergebnisse aus diesem Experiment. Bei einer Stringlänge von 5 Millionen Zeichen ist die Laufzeit bei einer Alphabetgröße von 4 mit 37 Sekunden deutlich langsamer als bei der Alphabetgröße 8 mit 32 Sekunden oder bei den Alphabetgrößen 16 bzw. 64 mit 29 Sekunden. Mit zunehmender Stringlänge verschlechtern sich die Laufzeiten der Alphabetgröße 64 relativ zu den kleineren Alphabeten. Bei einer Stringlänge von 20 Millionen Zeichen und der Alphabetgröße 64 weist der Partitionierungsalgorithmus eine Laufzeit von

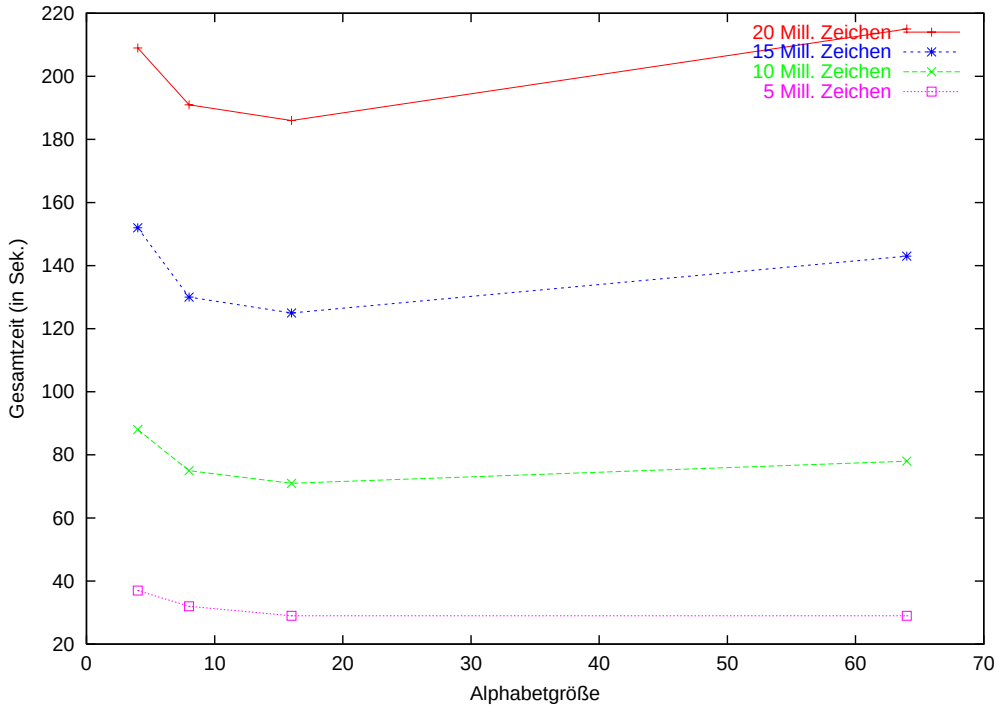


Abbildung 6.3: Konstruktionszeiten des Partitionierungsalgorithmus bei Alphabetgrößen zwischen 4 und 64 bei Stringlängen zwischen 5 und 20 Millionen Zeichen

215 Sekunden auf, während die Laufzeiten mit Alphabetgrößen von 4, 8 und 16 nur 209, 191 bzw. 186 Sekunden betragen. Die schnellsten Laufzeiten ergeben sich unabhängig von der Stringlänge für die Alphabetgröße 16. Mit kleiner bzw. größer werdenden Alphabetgrößen wird auch die Laufzeit schlechter.

Analyse Dieses Verhalten entspricht der Hypothese, dass die Alphabetgröße sich, wie beschrieben, auf den Verzweigungsgrad und damit auf die Laufzeiten auswirkt. Die Verschlechterung bei der Alphabetgröße 64 mit zunehmender Stringlänge hängt wohl damit zusammen, dass damit auch die Füllung der einzelnen Suffixbaum-Zweige steigt und die Konstruktion der Zweige den meisten Teil der Konstruktionszeit innnerhalb des Partitionierungsalgorithmus in Anspruch nimmt. Je mehr Suffixe die einzelnen Zweige enthalten, desto größer wird der erwartete Verzweigungsgrad.
Nach diesen Ergebnissen ist der Algorithmus leider für Alphabetgrößen, der von mir betrachteten Strings, wie DNA-Sequenzen oder Programmcode, langsamer als für mittelgroße Alphabete. Bei der Alphabetgröße 4 sind die Laufzeiten zwischen 13 und 28% langsamer als bei einem Alphabet mit 16 Zeichen. Gegenüber der Alphabet-

größe 64 ist der Vorteil der Alphabetgröße 16 ab einer Stringlänge von 10 Millionen Zeichen mit zwischen 10 und 16% schnelleren Laufzeiten etwas geringer.

6.2.2 Untersuchungen der Sortieralgorithmen innerhalb des *wotd*-Algorithmus

Das im Rahmen dieser Untersuchungen betrachtete zweite Konstruktionsverfahren ist der *wotd*-Algorithmus, welcher im Abschnitt 3.3 erläutert wurde. Während sich die Umgebungs- und Eingabeparameter nicht von denen des Partitionierungsalgorithmus unterscheiden, sind die programmbezogenen Parameter nicht miteinander zu vergleichen. Die *top-down* Konstruktion des *wotd*-Algorithmus ist unabhängig von Einstellungen, die sich auf die Struktur der Suffixbaum-Konstruktion auswirken. Dagegen ist der Sortieralgorithmus zur Realisierung der Gruppierungen von Suffixen der wichtigste hier untersuchte veränderbare Parameter. Verschiedene Sortieralgorithmen wie das *Countingsort*, das *Quicksort* oder das von mir entwickelte *Scansort* wurden in Abschnitt 3.4 vorgestellt. Diese sollen nun bezüglich ihrer Eignung innerhalb des *wotd*-Algorithmus untersucht werden.
Genau wie im Partitionierungsalgorithmus werden die Auswirkungen der Stringlänge und der Alphabetgröße untersucht.

Auswirkungen der Textlänge

Wie schon häufig erwähnt ist das Verhalten der Konstruktionsalgorithmen auf langen Strings ein wesentlicher Gesichtspunkt dieser Untersuchungen. Der *wotd*-Algorithmus bezieht die Festplatte mit in die Konstruktion des Suffixbaumes ein, indem er bei überlaufendem Hauptspeicher dem virtuellen Speichermanagement des Betriebssystems die Auslagerung der Daten überlässt und dabei auf das gute Lokalitätsverhalten der Zugriffe auf den Suffixbaum vertraut. Die unterschiedlichen Speicherplatzanforderungen der verschiedenen Sortieralgorithmen wirken sich natürlich auf die Stringlänge aus, ab der das Betriebssystem anfängt, Konstruktionsdaten auszulagern.
Der Versuchsaufbau ist dem des Partitionierungsalgorithmus sehr ähnlich, nur dass nicht mehr unterschiedliche Partitionierungstiefen, sondern verschiedene Sortieralgorithmen untersucht werden. Die Längen der zufällig erzeugten Strings über den Alphabeten mit den Größen 4 bzw. 90 bewegen sich zwischen 1 Million und 30 Millionen Zeichen.
Das *Countingsort* ist mit seiner linearen Laufzeit der theoretisch beste Algorithmus bei fester Alphabetgröße. Das *Quicksort* hat ein schlechteres asymptotisches Verhalten, benötigt aber weniger Speicherplatz. Das *Scansort* hat die gleichen niedrigeren Speicherplatzanforderungen wie das *Quicksort*, unterscheidet sich von der Komplexität des *Countingsorts* allerdings nur durch den Faktor Alphabetgröße und

weist eine ähnlich einfache Struktur wie das *Quicksort* auf. Damit ist zu erwarten, dass die Sortierung durch das *Countingsort* anfänglich sehr schnell ist und dann gegenüber den anderen Sortierverfahren langsamer wird, wenn die Speicherplatzanforderungen des *Countingsorts*, zusammen mit den anderen Konstruktiondaten, die Hauptspeichergrenzen überschreiten. Das *Scansort* sollte besonders schnell bei der Alphabetgröße 4 sein, während das *Quicksort* unabhängig von der Größe des Alphabets ist.

Ergebnis In den Abbildungen 6.4 und 6.5 sind die Ergebnisse dieses Experiments für die Alphabetgrößen 4 bzw. 90 graphisch dargestellt. Für beide Alphabetgrößen ist zu erkennen, dass der *wotd*-Algorithmus mit dem *Quicksort* deutlich langsamer ist als die anderen beiden Sortierverfahren. Für alle ist auffällig, dass es zwei Stufen gibt, bei denen der Anstieg der Laufzeit sich vergrößert. Die erste Stufe liegt bei einer Stringlänge von ca. 11 bis 12 Millionen Zeichen und die zweite befindet sich zwischen 21 und 24 Millionen Zeichen.
Bei der Alphabetgröße 4 ist auffällig, dass die Laufzeiten vom *Scansort* und *Countingsort* bis zu einer Stringlänge von 11 Millionen Zeichen nahezu identisch sind. Bei dieser Stringlänge zeigt das *Scansort* eine Laufzeit von 212 und das *Countingsort* eine Laufzeit von 218 Sekunden. Ab dieser ersten Stufe ist die Laufzeit des *wotd*-Algorithmus unter Benutzung des *Scansorts* bis zu einer Stringlänge von 15 Millionen Zeichen immer zwischen 25 und 30 Sekunden schneller als die unter der Benutzung des *Countingsorts*. Diese bessere Laufzeit verstärkt sich weiter, bis sich die Laufzeiten unter Benutzung des *Scansorts* und des *Countingsorts* mit der zweiten Stufe wieder annähern. In den darauf folgenden Laufzeiten bewegen sich die Laufzeitvorteile des *Scansorts* dauerhaft im Bereich zwischen 95 und 120 Sekunden. Die eine Ausnahme bei 27 Millionen Zeichen kann als Anomalie des Strings aufgefasst werden, was allerdings keinen Einfluss auf die weitere Auswertung hat.
Die Relationen zwischen dem *Countingsort* und dem *Quicksort* bleiben auch bei einer Alphabetgröße 90 erhalten, während das *Scansort* sich deutlich verschlechtert und bis zur zweiten Stufe nur wenig schnellere Konstruktionszeiten aufweist als das *Quicksort*. Für Strings ab einer Länge von 24 Millionen Zeichen wächst die Laufzeit des *Scansorts* noch einmal sehr stark an und ist bei 30 Millionen Zeichen mehr als doppelt so langsam wie die des *Countingsorts*.

Analyse Zur Analyse der Laufzeiten ist es hilfreich, sich die Speicherplatzanforderungen des Suffixbaumes und der Sortieralgorithmen wieder in Erinnerung zu rufen. Die maximalen Speicherplatzanforderungen des Suffixbaumes belaufen sich bei der hier untersuchten Repräsentation, wie in Abschnitt 4.1.3 beschrieben, auf 12 Bytes pro indiziertes Zeichen, dürften aber im erwarteten Fall etwas niedriger liegen. Damit ist zu erklären, dass die Laufzeiten ab einer Stringlänge von 11 Millionen Zeichen stärker zu wachsen beginnen, weil nicht mehr der gesamte Suffixbaum in den Haupt-

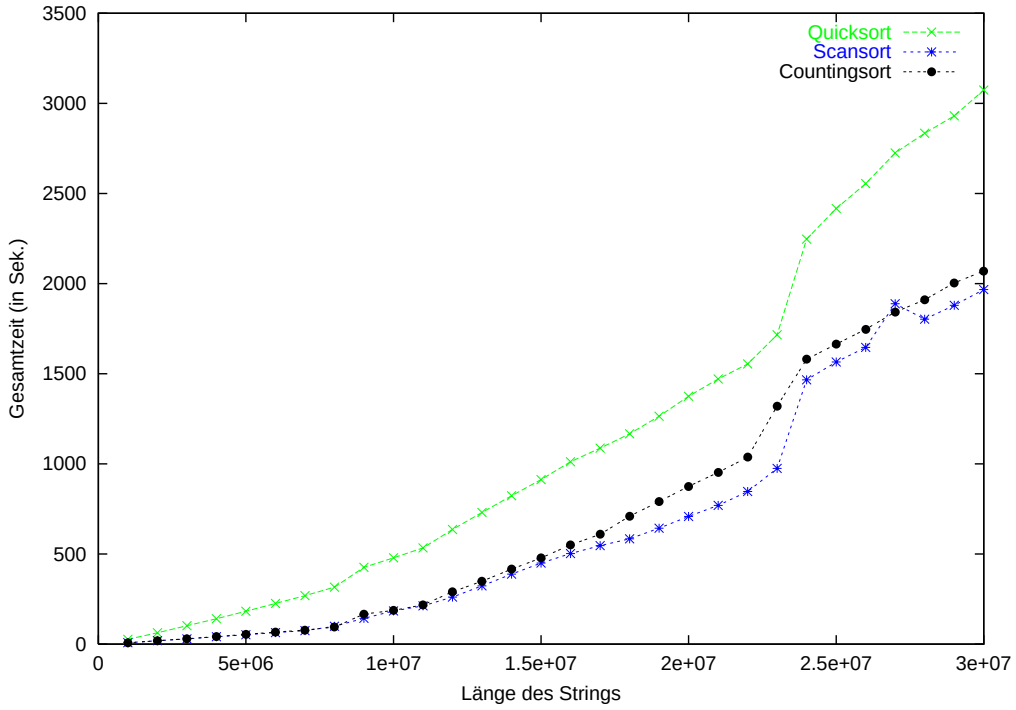


Abbildung 6.4: Konstruktionszeiten des *wotd*-Algorithmus bei einer Alphabetgröße 4 für verschiedene Sortieralgorithmen bei linear wachsender Stringlänge

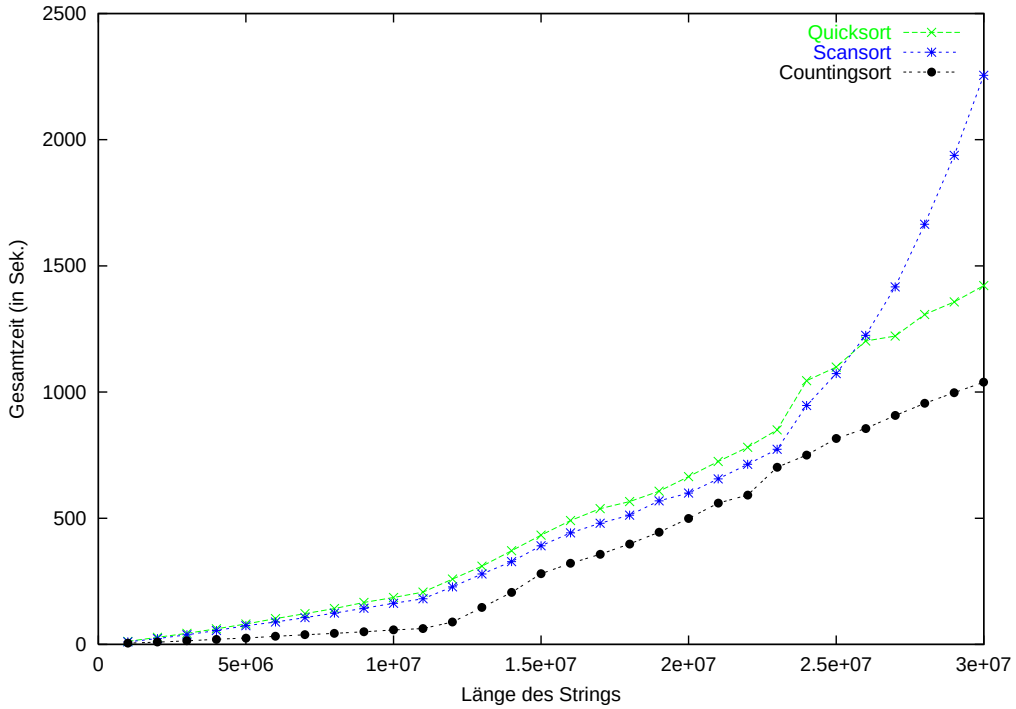


Abbildung 6.5: Konstruktionszeiten des *wotd*-Algorithmus bei einer Alphabetgröße 90 für verschiedene Sortieralgorithmen bei linear wachsender Stringlänge

speicher passt.
Die hohen Speicherplatzanforderungen des *Countingsorts*, das außer dem zu sortierenden Array noch zusätzlichen Speicherbedarf für ein Hilfsarray erfordert, begründen die schnelleren Laufzeiten durch die Benutzung des *Scansorts* ab einer Stringlänge von 15 Millionen Zeichen. Bei dieser Stringlänge betragen die anfänglichen Speicherplatzanforderungen zur Sortierung des Arrays ca. 120 MB ($2 \times 4n$ Bytes für Array und Hilfsarray), was fast der Hauptspeichergröße entspricht. Zusammen mit den Speicherplatzanforderung für den String übersteigt dieser Wert die Hauptspeicherbeschränkung und das Betriebssystem fängt an auszulagern. Der relative Laufzeitanstieg durch die Benutzung des *Countingsorts* gegenüber der des *Scansorts* fällt nicht stärker aus, weil das Hilfsarray nur zu Beginn die Länge des gesamten Suffixarrays erreicht und danach nur der Größe der immer kleiner werdenden zu bearbeitenden Gruppen entspricht. Das *Quicksort* ist im Vergleich zu den anderen beiden Sortieralgorithmenauf Grund seines logarithmischen Faktors bei der Komplexität ($O(n \log n)$) nicht konkurrenzfähig.
Wächst das Suffixarray bei 23 Millionen Zeichen nun zusammen mit dem String über die Hauptspeichergröße, so ergibt sich ein deutlicher Laufzeitanstieg, der sich in den Abbildungen 6.4 und 6.5 in der zweiten Stufe zeigt. Das Betriebssystem muss dabei auch Teile des Suffixarrays auslagern, was bei der Benutzung des *Scansorts*, welches das Suffixarray für jedes Zeichen einmal durchläuft, auf großen Alphabeten einen sehr starken Laufzeitanstieg zur Folge hat. Dies begründet den im Verhältnis zu den anderen Sortieralgorithmen stärkeren Anstieg der Kurve in Abbildung 6.5.
Die Größe des Suffixbaumes, von dem Teile ausgelagert werden müssen, hat auf die Laufzeiten einen geringen Einfluss, da jeder Knoten nach seiner Evaluation nicht mehr berührt wird.

Verschiedene Alphabetgrößen

Beim Partitionierungsalgorithmus hat sich gezeigt, dass die Alphabetgröße und damit der Verzweigungsgrad der Knoten einen starken Einfluss auf das Laufzeitverhalten hat. Nun soll auch der *wotd*-Algorithmus mit seinen verschiedenen Sortierverfahren auf den Parameter Alphabetgröße untersucht werden.
Die Untersuchungen wurden dabei auf Alphabeten mit den Größen aller 2er-Potenzen zwischen 4 und 128 durchgeführt. Die untersuchten zufällig generierten Strings haben für dieses Experiment Längen von 10 bzw. 20 Millionen Zeichen.
Der *wotd*-Algorithmus ist im wesentlichen unabhängig von der Alphabetgröße, da die Kindlisten der Knoten bei der *top-down*-Konstruktion nicht durchsucht werden wie beim Partitionierungsalgorithmus. Als einziger hier überprüfter Sortieralgorithmus ist allerdings die Laufzeit des *Scansorts* abhängig vom Alphabet. Bei steigender Alphabetgröße sollten demnach die Laufzeiten bei der Benutzung des *Scansorts* im *wotd*-Algorithmus schlechter werden. Wie sich aber schon bei den Untersuchungen

des Partitionierungsalgorithmus herausgestellt hat, führen kleinere Alphabetgrößen zu einem geringeren Verzweigungsgrad und damit zu einem größeren Suffixbaum, was höhere Konstruktionszeiten zur Folge haben sollte.

Ergebnis In der Abbildung 6.6 sind die Ergebnisse dieses Experiments visualisiert. Es ist deutlich zu erkennen, dass der *wotd*-Algorithmus unter Benutzung des *Counting*- bzw. *Quicksorts* mit zunehmender Alphabetgröße deutlich schneller wird. Bei der Benutzung des *Scansorts* nimmt die Laufzeit bei einer Stringlänge von 10 Millionen Zeichen, beginnend mit der Alphabetgröße 4 bis zur Alphabetgröße 8, zunächst ab, steigt dann allerdings wieder deutlich an. Bei der größeren Stringlänge von 20 Millionen nimmt die Laufzeit erst bei der Alphabetgröße 32 wieder deutlich gegenüber den kleineren Alphabeten zu.
Vergleicht man die Laufzeiten bezüglich der verschiedenen Sortierverfahren, so wird deutlich, dass die Benutzung des *Scansorts* bei kleineren Alphabeten mindestens so effektiv wie die des *Countingsorts* ist. Bei einer Stringlänge von 10 Millionen Zeichen und Alphabetgrößen zwischen 4 und 16 weisen das *Scan*- und *Countingsort* nahezu identische Laufzeiten auf, und bei einer Stringlänge von 20 Millionen Zeichen ist die Benutzung des *Scansorts* sogar deutlich effektiver. Mit zunehmender Alphabetgröße wird der *wotd*-Algorithmus mit dem integrierten *Countingsort* allerdings deutlich schneller als bei der Nutzung des *Scansorts*.

Analyse Schon im vorherigen Abschnitt wurde erläutert, warum das *Scansort* bei Alphabetgröße 4 und steigender Stringlänge dem *Countingsort* überlegen ist. Da die Komplexität des *Scansorts* abhängig von der Alphabetgröße ist, sollten die Laufzeiten bei steigender Anzahl von Zeichen im Alphabet ansteigen. Dieser Effekt tritt bei der Benutzung im *wotd*-Algorithmus aber erst ab einer mittleren Alphabetgröße ein, da durch den kleinen Verzweigungsgrad auch die Größe des Suffixbaumes zunimmt und der *wotd*-Algorithmus unabhängig vom *Scansort* mehr Zeit benötigt, was sich auch durch die Laufzeiten bei der Benutzung der alphabetunabhängigen Sortieralgorithmen belegen lässt.
Es bleibt festzuhalten, dass der *wotd*-Algorithmus mit dem *Scansort* auf kleinen Alphabeten sehr effizient ist, aber bei großen dem *Countingsort* unterliegt.

6.2.3 Vergleich der verschiedenen Konstruktionsverfahren

Ein wichtiges Ziel dieser Arbeit ist nicht allein die Untersuchung der einzelnen Algorithmen, sondern, im Hinblick auf die praktische Nutzbarkeit, auch der Vergleich der verschiedenen Konstruktionsverfahren. Basierend auf den bisherigen Untersuchungen werden zunächst die Konstruktionszeiten auf den zufällig erzeugten Strings verglichen.
Die Struktur der Strings aus den untersuchten Bereichen entspricht aber nur selten

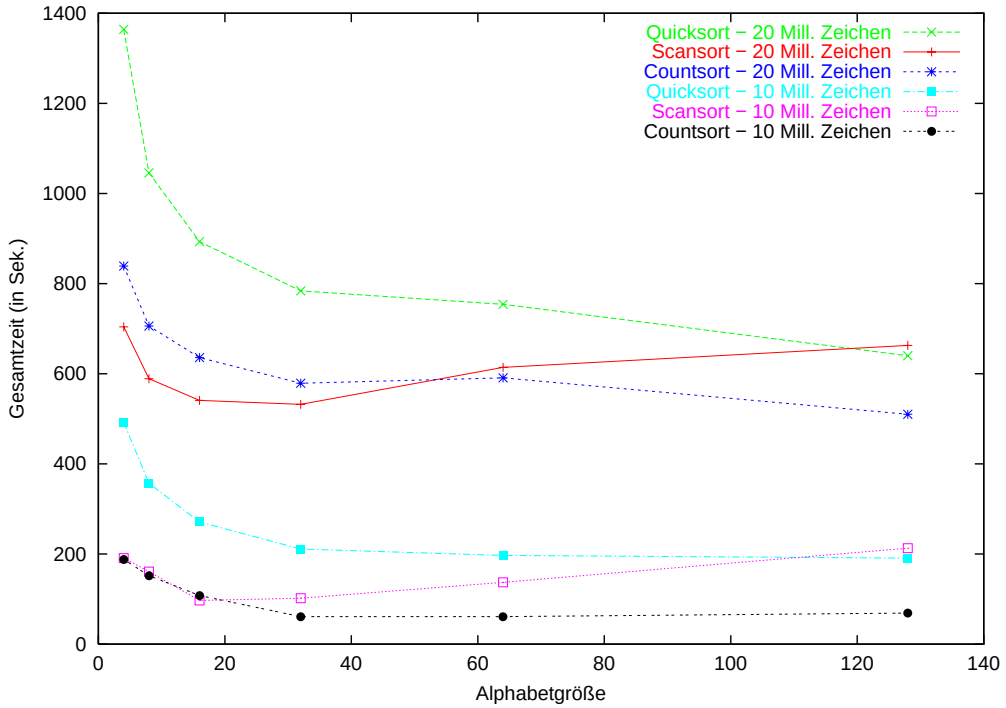


Abbildung 6.6: Konstruktionszeiten des Partitionierungsalgorithmus für Alphabetgrößen zwischen 4 und 64 bei Stringslängen zwischen 10 und 20 Millionen Zeichen

einer zufälligen Verteilung. In Genom-Datenbanken gibt es beispielsweise Anwendungen, in denen viele Proteine oder DNA-Abschnitte gemeinsam, d.h. konkateniert in einem Suffixbaum indiziert werden, so dass viele lange wiederholende Teilstrings auftreten. Deshalb werden neben den zufällig erzeugten Strings auch Sequenzen, die lange Wiederholungen aufweisen, untersucht, bis dann abschließend die Konstruktionszeiten auf realen Texten verglichen werden.

Auswirkungen der Textlänge

Die Untersuchungen auf den zufällig erzeugten Strings mit linear zunehmender Länge, die schon bezüglich der einzelnen Konstruktionsalgorithmen durchgeführt wurden, werden hier um den Vergleich der verschiedenen Konstruktionsverfahren erweitert.
Zusätzlich zum Partitionierungs- und *wotd*-Algorithmus wird noch der Ukkonen-Algorithmus als Vertreter der Linearzeitalgorithmen mit einbezogen. Die Laufzeiten der Algorithmen entsprechen den Ergebnissen aus den vorherigen Experimenten.
In den Abbildungen 6.7 und 6.8 sind die Laufzeiten des Ukkonen-, Partitionierungs- und *wotd*-Algorithmus, für die Alphabetgrößen 4 und 90 bei zunehmender String-

länge visualisiert. Die Auswahl der Programmparameter für die einzelnen Algorithmen entspricht den optimalen Einstellungen, die sich aus den Basisexperimenten hervorgetan haben. Bezüglich des Partitionierungsalgorithmus sind das bei der Alphabetgröße 4 die Partitionierungstiefen 2 (bei sehr langen Strings) bzw. 8 (bei mittlerer Stringlänge) und bei der Alphabetgröße 90 die Partitionierungstiefen 1 bzw. 3. Der *wotd*-Algorithmus arbeitet am schnellsten mit dem zur Gruppierung verwendeten *Scan*- bzw. *Countingsort*. Der Ukkonen-Algorithmus hat wie alle anderen Linearzeitalgorithmen keine veränderlichen Programmparameter.

Ergebnis In der Graphik zur Visualisierung der Laufzeiten bei der Alphabetgröße 4 (Abb. 6.7) ist zu erkennen, dass der Linearzeitalgorithmus von Ukkonen mit 4 Sekunden Laufzeit bei einer Stringlänge von 1 Million Zeichen noch schneller ist als der Partitionierungs- (12 bzw. 7 Sek.) oder *wotd*-Algorithmus (8 bzw. 7 Sek.). Mit zunehmender Stringlänge steigt die Laufzeit allerdings sehr stark an und bei 15 Millionen Zeichen benötigt der Ukkonen-Algorithmus schon mehr als 15 Stunden (!) Konstruktionszeit.
Der Partitionierungsalgorithmus mit der Partitionierungstiefe 2 ist bis zur Stringlänge von 10 Millionen Zeichen langsamer als der *wotd*-Algorithmus mit integriertem *Scan*- bzw. *Countingsort*. Bei Stringlängen zwischen 11 und 17 Millionen Zeichen ist er dann aber schneller als der *wotd*-Algorithmus, bis sich danach, bei größeren Stringlängen ab 22 Millionen Zeichen, der Partitionierungs- und der *wotd*-Algorithmus ähnlich verhalten.
Bis zu einer Stringlänge von 23 Millionen Zeichen ist der Partitionierungsalgorithmus mit Tiefe 8 deutlich schneller als der *wotd*-Algorithmus. In dem Bereich zwischen 13 und 22 Millionen Zeichen ist letzterer sogar mehr als dreimal langsamer. Erst ab Werten von 24 Millionen Zeichen nehmen die Laufzeiten des Partitionierungsalgorithmus nahezu exponentiell zu.
Mit der höheren Alphabetgröße 90 kommen die Vorteile des *wotd*-Algorithmus mehr zum tragen. Bis zu einer Stringlänge von 12 Millionen Zeichen ist dieser unter der Benutzung des *Countingsorts* um bis zu drei mal schneller als der Partitionierungsalgorithmus mit der Partitionierungstiefe 3 und damit deutlich das schnellste Konstruktionsverfahren. Für Stringlängen zwischen 13 und 23 Millionen Zeichen ist der Partitionierungsalgorithmus dann wieder deutlich effizienter, bis die Laufzeiten danach wieder stark ansteigen. Die kleinere Partitionierungstiefe 1 zeigt im Bereich zwischen 17 und 22 Millionen Zeichen ähnliche Laufzeiten wie der *wotd*-Algorithmus mit dem *Countingsort*. Diese steigen dann aber, wie bei der Partitionierungstiefe 3, sehr stark an, so dass der *wotd*-Algorithmus für Strings ab 24 Millionen Zeichen konkurrenzlos ist.

Analyse Die einzelnen Laufzeitkurven wurden schon in vorhergehenden Abschnitten analysiert. Um das unterschiedlichen Laufzeitverhalten der Konstruktionsalgo-

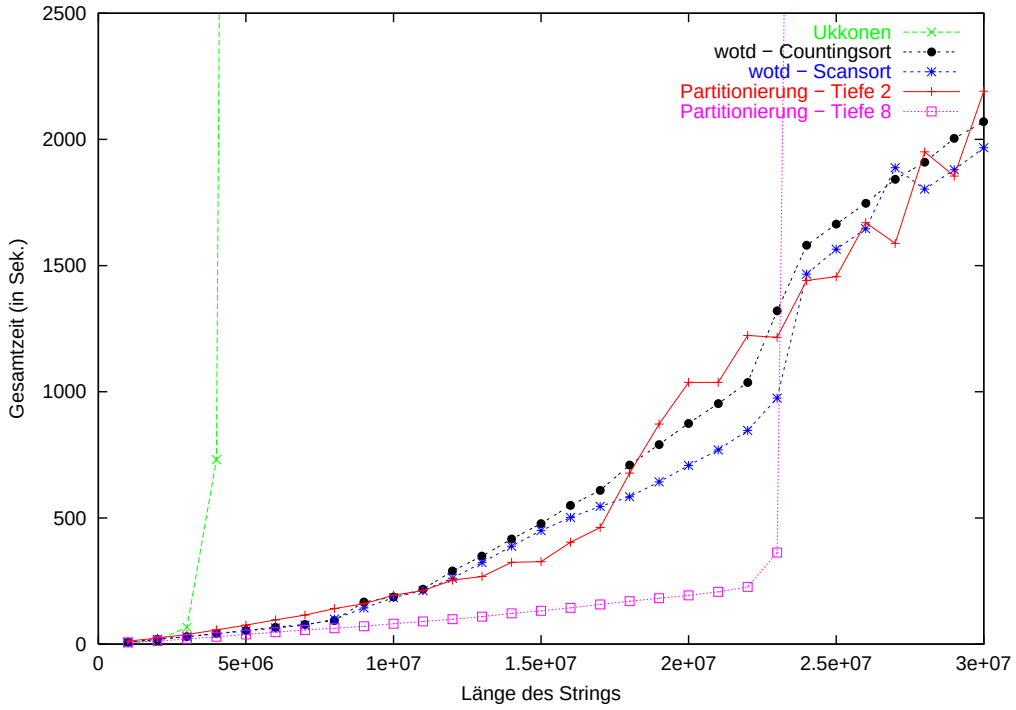


Abbildung 6.7: Konstruktionszeiten der verschiedenen Konstruktionsalgorithmen bei der Alphabetgröße 4 für verschiedene Sortieralgorithmen bei linear wachsender Stringlänge

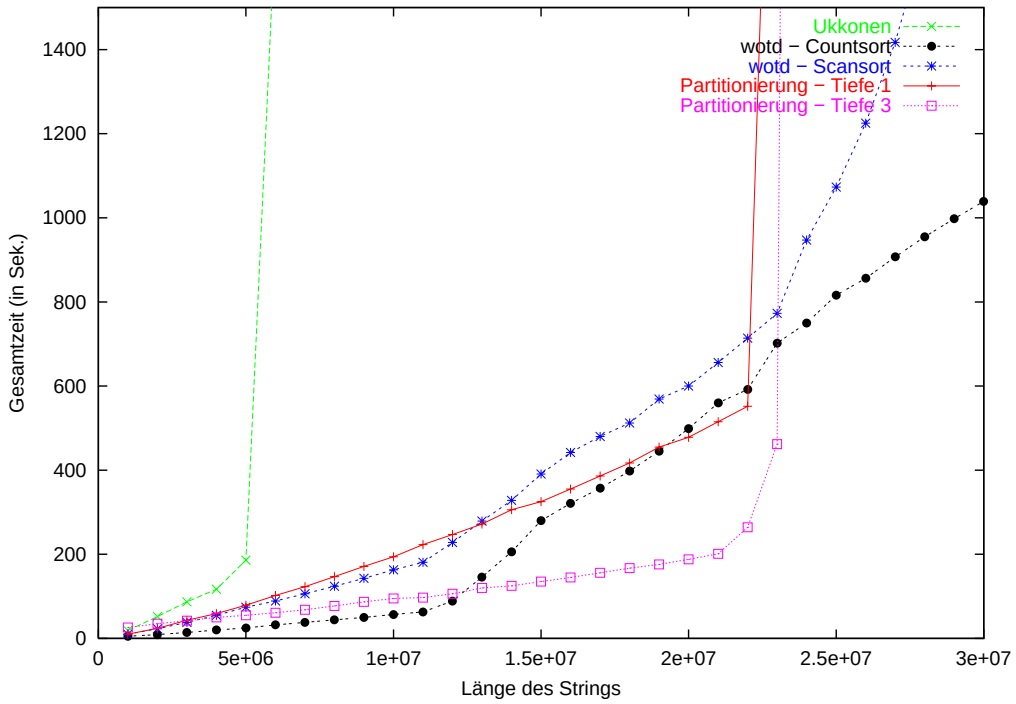


Abbildung 6.8: Konstruktionszeiten der verschiedenen Konstruktionsalgorithmen bei der Alphabetgröße 90 für verschiedene Sortieralgorithmen bei linear wachsender Stringlänge

rithmen zu ergründen, ist es notwendig, auf die Struktur der einzelnen Algorithmen einzugehen.
Die Suffixbaum-Repräsentation im Ukkonen-Algorithmus hat mit 20 Bytes pro Knoten sehr hohe Speicherplatzanforderungen, so dass er schnell über die Hauptspeichergröße wächst. Die Laufzeiten steigen dann exponentiell an, da der Algorithmus durch die Benutzung der Suffixlinks ein schlechtes Lokalitätsverhalten aufweist und damit die Anzahl der Festplattenzugriffe durch die Datenauslagerung sehr stark wächst.
Der Partitionierungsalgorithmus basiert auf einer Konstruktion des Suffixbaumes in einzelnen Teilen, so dass nicht der gesamte Suffixbaum in den Hauptspeicher passen muss. Dem gegenüber steht der *wotd*-Algorithmus, der den Suffixbaum im Ganzen aufbaut, dabei aber die geringen Speicherplatzanforderungen und das sehr gute Lokalitätsverhalten bezüglich der Zugriffe auf den Suffixbaum ausnutzt. Der Partitionierungsalgorithmus ist bei der Alphabetgröße 4 und Partitionierungstiefe 8 bis zu dem Punkt, an dem die zur Konstruktion verwendeten Daten nicht mehr komplett im Hauptspeicher gehalten werden können, deutlich schneller als der *wotd*-Algorithmus, da er die Suffixbaum-Teile komplett im Hauptspeicher konstruieren kann und weil die von mir verbesserte Variante mit ihren Einfügeoperationen in die Suffixbaum-Zweige nicht an der Wurzel beginnt, sondern direkt in den Zweigen.
Dieses Verhalten ist wesentlich für das Erreichen der schnellen Laufzeiten. Für alle Suffixe einer Partition wird nur eine Einfügeoperation in den Suffixbaum-Stumpf durchgeführt, so dass die Laufzeiten, wie schon durch meine theoretische Analyse in Abschnitt 3.2 belegt wurde, gegenüber der Variante mit den Einfügeoperationen ab der Wurzel schneller sind. Zusätzlich haben die oberen Ebenen des Suffixbaumes eine erwartet dichtere Besetzung an Knoten und diese Knoten einen höheren Verzweigungsgrad als die unteren Ebenen, so dass sich der Effizienzgewinn in der Praxis noch deutlicher niederschlägt.
Bei der Alphabetgröße 90 ist der *wotd*-Algorithmus anfangs das effizienteste Konstruktionsverfahren, da dieser im Gegensatz zum Partitionierungsalgorithmus unabhängig von der Alphabetgröße ist und nicht in das serialisierte Format umgewandelt werden muss. Bis dann allerdings, bei steigender Stringlänge, die beschriebenen Vorteile des Partitionierungsalgorithmus zum Tragen kommen.
Steigen nun aber auch die Speicherplatzanforderungen des Partitionierungsalgorithmus über die Hauptspeichergröße, so ist das, gegenüber dem *wotd*-Algorithmus, schlechtere Lokalitätsverhalten der Hauptgrund für die plötzlich sehr stark ansteigenden Laufzeiten. Die Zugriffe auf den Suffixbaum im *wotd*-Algorithmus sind bezüglich der Lokalität optimal und auch die Sortieralgorithmen zeigen in dieser Hinsicht ein gutes Verhalten, so dass auch für sehr lange Strings, bei denen die zur Konstruktion notwendigen Daten vom Betriebssystem ausgelagert werden, eine Konstruktion in angemessener Zeit erfolgen kann. Das virtuelle Speichermanagement arbeitet bei der Speicherung der Daten in einem Array (wie im *wotd*-Algorithmus) wesentlich besser als bei Baumrepräsentationen, wie im Partitionierungsalgorithmus,

bei denen die Knoten beliebig über den Hauptspeicher verteilt sind.
Der verbesserte Partitionierungsalgorithmus profitiert von der Konstruktion des Suffixbaumes in einzelnen Teilen, die jeweils komplett im Hauptspeicher aufgebaut werden können. Der *wotd*-Algorithmus zieht seinen Nutzen dagegen aus seinem exzellenten Lokalitätsverhalten, was auch bei sehr großen Suffixbäumen, bei denen der Hintergrundspeicher zur Konstruktion mit einbezogen werden muss, einen Aufbau in angemessener Zeit erlaubt.

6.2.4 Untersuchungen an Fibonacci-Strings

Die bisherigen Experimente wurden auf zufällig erzeugten Strings durchgeführt. Dabei sind die resultierenden Suffixbäume weitgehend ausgeglichen, was aber in der Praxis nicht immer der Fall ist. Deshalb sollen nun entartete Suffixbäume untersucht werden, die durch Texte mit sich langen wiederholenden Teilstrings entstehen. Zu diesem Zweck wurden Fibonacci-Strings [30] der Längen zwischen 50000 und 1 Million Zeichen erzeugt und für die verschiedenen Algorithmen untersucht. Sie haben ein Alphabet der Größe 2 und weisen sehr lange Wiederholungen auf, die zu Entartungen des Suffixbaumes führen. Auf Grund der kleinen Alphabetgröße wurden die Partitionierungstiefen mit 10 und 16 größer gewählt als in den vorherigen Experimenten.
Bei diesem Experiment war zu erwarten, dass der Ukkonen-Algorithmus das schnellste Konstruktionsverfahren ist, weil die untersuchten Stringlängen noch eine Konstruktion im Hauptspeicher zulassen und dieser Algorithmus unabhängig von Entartungen des Suffixbaumes ist.

Ergebnis In Abbildung 6.9 sind die Ergebnisse dieser Untersuchung dargestellt. Der Ukkonen-Algorithmus benötigt bis zu einer Stringlänge von 1 Million Zeichen weniger als 8 Sekunden zur Konstruktion, während die Experimente für den Partitionierungs- bzw. *wotd*-Algorithmus schon vorher nach einigen Stunden abgebrochen werden mussten. Beim Partitionierungsalgorithmus ist zu erkennen, dass die Konstruktionszeiten nahezu unabhängig von der Partitionierungstiefe sind. Für eine Stringlänge von 500000 Zeichen und der Partitionierungstiefe 10 bzw. 16 liegt die maximale Füllung einer Partition bei 72948 bzw. 45084 Suffixen, obwohl der erwartete Füllungsgrad sich nur auf 488,28 bzw. 7,63 Suffixe beläuft.
Auch der *wotd*-Algorithmus zeigt bei den verschiedenen Sortierverfahren fast identische Laufzeiten. Insgesamt ist er aber deutlich langsamer als der Partitionierungsalgorithmus.

Analyse Es ist eindeutig zu sehen, dass weder der Partitionierungs- noch der *wotd*-Algorithmus konkurrenzfähig für die Konstruktion entarteter Suffixbäume ist, da die Laufzeiten beider Algorithmen von der Suffixbaum-Struktur abhängig sind.

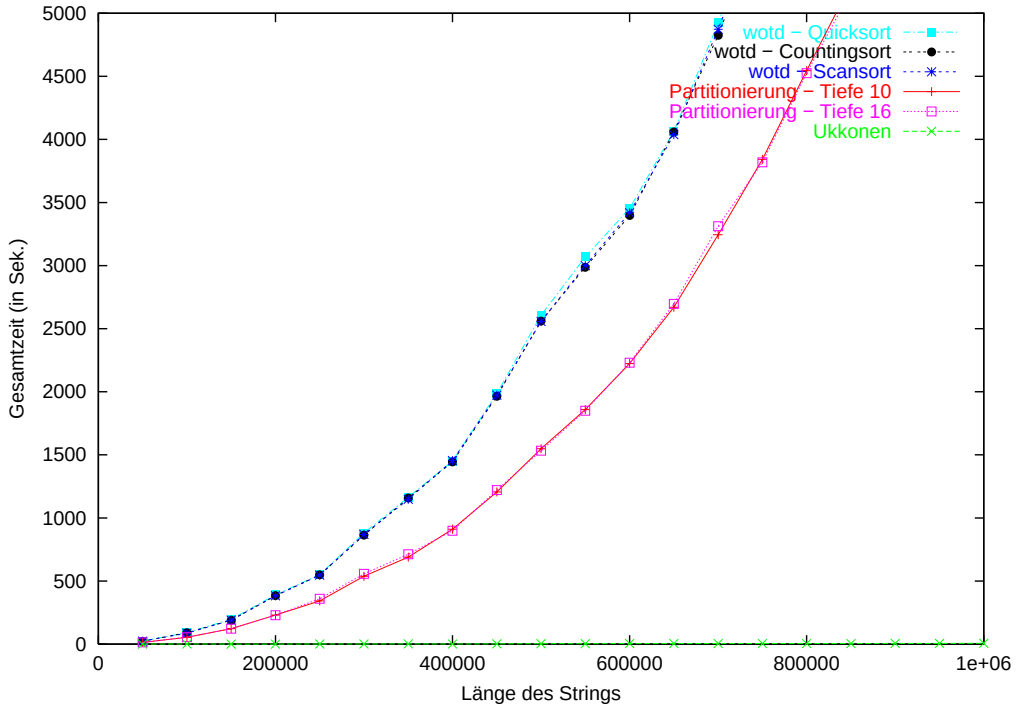


Abbildung 6.9: Konstruktionszeiten der verschiedenen Algorithmen auf Fibonacci-Strings

Der Ukkonen-Algorithmus dagegen ist unabhängig von diesen Entartungen und stellt sich bei den kürzeren Stringlängen, deren Suffixbaum vollständig im Hauptspeicher aufgebaut werden kann, als mit Abstand effizienteste Variante heraus.
Die Sortieralgorithmen haben hier einen sehr geringen Einfluss auf die Laufzeit des *wotd*-Algorithmus, weil die meiste Zeit für das Finden der längsten gemeinsamen Präfixe aufgewendet wird und die Gruppierung nur einen kleinen Teil der gesamten Konstruktionszeit ausmacht. Aus ähnlichen Gründen ergibt sich die Unabhängigkeit des Partitionierungsalgorithmus von der Partitionierungstiefe.
Der Partitionierungs- und der *wotd*-Algorithmus in dieser Form sind nicht geeignet für die Konstruktion von Suffixbäumen auf Strings, die solche Entartungen des Suffixbaumes hervorrufen, während der Ukkonen-Algorithmus unabhängig von solchen Textstrukturen ist, aber leider durch die Hauptspeichergröße beschränkt bleibt. Allerdings schlagen Giegerich *et al.* [24] eine Variante des *wotd*-Algorithmus vor, die es erlaubt Berechnungen der längsten gemeinsamen Präfixe zu speichern und diese Werte wiederzuverwenden. Dies wirkt sich auf die Konstruktionszeiten der Strings mit langen Wiederholungen positiv aus.

6.2.5 Verschiedenartige Texte aus der Praxis

Die bis hier durchgeführten Experimente waren geeignet, um einzelne Aspekte der Konstruktionsalgorithmen, wie Programm- oder Eingabeparameter, unabhängig voneinander zu untersuchen. Die wichtigsten Untersuchungen sind allerdings die auf den realen Texten, da durch sie die praktische Anwendbarkeit der Konstruktionsalgorithmen bestimmt werden kann.
Die hier untersuchten Textmengen ergeben sich aus den Anwendungsgebieten Genom-Datenbanken und Information Retrieval, die relevant für diese Arbeit sind.
In den Genom-Datenbanken sind DNA-Sequenzen die am meisten durch Suffixbäume indizierten Strings. Hinzu kommen noch Proteine, auf denen allerdings weniger nach exakten Mustern gesucht wird, sondern Ähnlichkeiten bezüglich sogenannter Score-Matrizen bestimmt werden. Die Mustersuche kann allerdings auch hier als Teilproblem von komplexeren Anwendungen dienen.
Die Texte aus dem Information Retrieval beschränken sich üblicherweise nicht auf spezielle Bereiche. Sie umfassen sowohl natürlichsprachige Texte verschiedener Sprachen, wie auch Quellcode von Programmen oder automatisch generierte Texte. Die ausgewählten Textdaten sollen nun genauer betrachtet werden:

- Die **DNA-Sequenzen** bilden die Grundlage der Genom-Forschung und ein Hauptanwendungsgebiet für die Indizierung durch Suffixbäume. Die hier untersuchten DNA-Sequenzen sind das komplette Genom des Bakteriums Escherichia coli (*E. coli*), sowie das dritte und fünfte Chromosom des Fadenwurms Caenorhabditis elegans (*C. elegans*).

- Anhand von **Bibeltexten** der Sprachen dänisch, englisch, deutsch und französisch sollen die Auswirkungen der unterschiedlichen Sprachen auf die Textstruktur und damit auf die Suffixbaum-Konstruktion untersucht werden.

- Der **Quellcode** von Programmen ist eine der häufigsten Textdaten, mit denen ein Informatiker zu tun hat. In großen Programmen können parametrisierte Suffixbäume bei der Suche von Strukturen eingesetzt werden, um beispielsweise festzustellen, ob Funktionen schon programmiert und wiederverwendet werden können [8].
  Die Untersuchten Strings sind der gesamte Quellcode vom Robocup-Team unseres Fachbereichs, der gesamte Quellcode des gcc-Compilers und ein Teil des Linuxkernels. Für die Implementierung der Fussballroboter wurde die Programmiersprache C++ benutzt. Der hier untersuchte String ist die Konkatenation aller Dateien mit den Endungen *.cpp*, *.h* und *.ui*, wobei Dateien mit der Endung *.ui* die von der Programmierumgebung generierten Informationen zu den Benutzerschnittstellen enthalten. Die anderen beiden Programme sind in C geschrieben und haben keine solchen Benutzerschnittstellen. Für sie sind nur die Dateien mit den Endungen *.c* und *.h* enthalten.

- Die verwendeten **Proteindaten** sind eine Sammlung von vielen einzelnen Proteinen im FASTA-Format [1]. Der String besteht hier nicht nur aus den Aminosäuren, sondern noch aus zusätzlichen Textinformationen zu den Proteinen.
- **Sonstige** hier untersuchte Dateien sind ein Index des FTP-Servers der TU-Berlin und das *CIA World Fact Book*, welches die Eckdaten sämtlicher Länder der Welt beinhaltet.

Die Längen und Alphabetgrößen dieser Strings befinden sich für die DNA-Sequenzen, zusammen mit den Laufzeitdaten, in der Tabelle 6.1 und für die anderen Texte in der Tabelle 6.2.

DNA-Sequenzen

Die untersuchten DNA-Sequenzen haben verschiedene Größen. Das Genom des *E. coli* Bakteriums hat eine Länge von ca. 4,6 Mbp (Mega Basenpaaren) und das dritte bzw. fünfte Chromosom des Wurmes haben eine Länge von ungefähr 12,8 bzw. 20,5 Mbp.

Ergebnis Die Ergebnisse der Untersuchungen sind in der Tabelle 6.1 dargestellt. Sie enthält zu den verschiedenen DNA-Sequenzen die Stringlänge und die Laufzeiten der Algorithmen mit den optimalen Einstellungen.
Bei den Wurmchromosomen war die Konstruktion durch den Ukkonen-Algorithmus

String	String-länge	Alph.-größe	Laufzeiten				
			Part.-Tiefe 6	Part.-Tiefe 8	wotd-Scan.	wotd-Count.	Ukk.
E. coli	4638690	4	34	38	48	48	10237
Wurm Chr. 3	12836730	4	132	122	289	283	-
Wurm Chr. 5	20551922	4	282	265	910	1125	-

Tabelle 6.1: Konstruktionszeiten (in Sek.) der verschiedenen Algorithmen auf DNA-Sequenzen

nicht möglich und die Tabelle enthält deshalb an den entsprechenden Stellen keine Werte. Für den Partitionierungsalgorithmus haben sich die Partitionierungstiefen 6 und 8 als effizienteste Einstellungen erwiesen und der *wotd*-Algorithmus arbeitet mit dem *Countingsort* und dem *Scansort* am besten. Die Laufzeiten mit den anderen Einstellungen der Programmparameter sind deshalb hier nicht aufgeführt.
In Abbildung 6.10 sind die Laufzeiten normiert auf eine Stringlänge von 1 Million Zeichen dargestellt. Es ist deutlich zu erkennen, dass der Partitionierungsalgorith-

[1] Das FASTA-Format ist ein in der Genforschung häufig verwendetes Textformat für DNA- und Protein-Daten

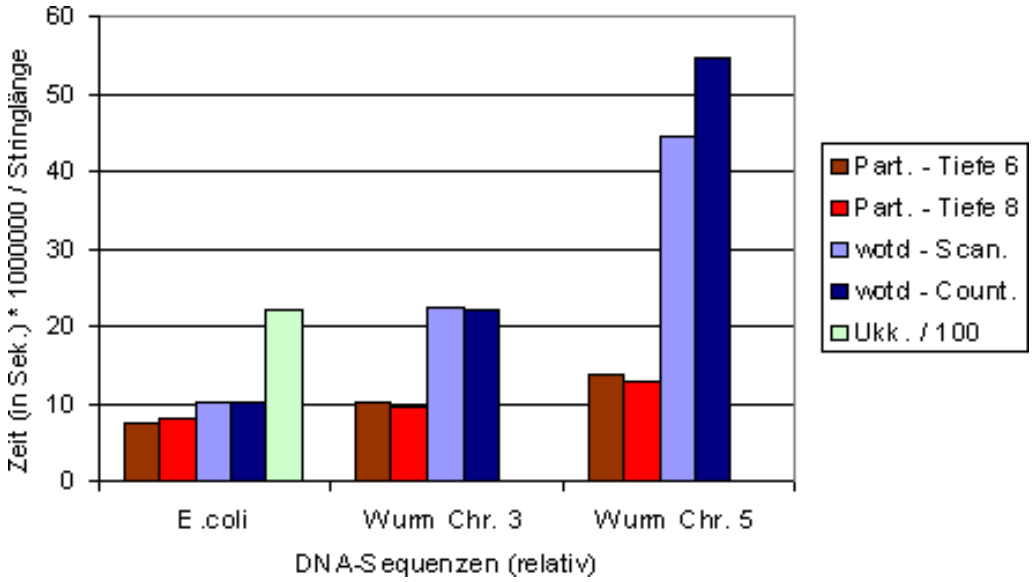


Abbildung 6.10: Konstruktionszeiten der verschiedenen Algorithmen auf DNA-Sequenzen in Sekunden normiert auf 1000000 Zeichen

mus für alle DNA-Sequenzen die schnellsten Konstruktionszeiten aufweist. Dabei besteht zwischen den Partitionierungstiefen 6 und 8, mit Laufzeiten von 34 bzw. 38 Sekunden bei *E. Coli* und Laufzeiten von 282 bzw. 265 Sekunden beim fünften Chromosom von *C. elegans*, nur ein geringer Unterschied.
Die relativen Laufzeiten wachsen für den Partitionierungsalgorithmus bei steigender Stringlänge nicht sehr stark an. Der *wotd*-Algorithmus ist dagegen, bei der kürzesten hier untersuchten DNA-Sequenz des *E. coli* Bakteriums, mit 48 Sekunden Laufzeit nur wenig langsamer als der Partitionierungsalgorithmus. Bei den längeren Wurmchromosomen nehmen die Laufzeiten jedoch stark zu. Das *Scansort* stellt sich beim Chromosom 5 des Wurmes, der längsten Sequenz, mit einer Laufzeit von 910 Sekunden als schnellstes Gruppierungsverfahren innerhalb des *wotd*-Algorithmus heraus. Bei den beiden kürzeren Sequenzen unterscheidet sich die Benutzung des *Scansorts* von der des *Countingsorts* dagegen kaum.
Der Ukkonen-Algorithmus ist mit einer Laufzeit von über 2 Stunden, wie nach den Basisuntersuchungen zu erwarten war, die langsamste Konstruktionsvariante. Um eine geeignete Visualisierung zu gewährleisten musste die Laufzeit des Ukkonen-Algorithmus beim *E. Coli* Bakterium durch 100 dividiert werden. Bei den Wurmchromosomen war wie schon erwähnt keine Konstruktion mehr möglich.

Analyse Die Ergebnisse dieser Experimente entsprechen weitgehend den Erwartungen nach den Basisuntersuchungen, was darauf hindeutet, dass auch die hier konstruierten Suffixbäume weitgehend ausgeglichen sind.

Der geringe Laufzeitanstieg für den Partitionierungsalgorithmus bei steigender Sequenzlänge ist auf die dichtere Füllung der Zweige und des Suffixbaum-Stumpfes zurückzuführen, wodurch beim Einfügen mehr Knoten durchlaufen werden müssen. Der stärkere Anstieg beim *wotd*-Algorithmus ist durch die Auslagerung der Konstruktionsdaten auf den Hintergrundspeicher zu erklären. Es ist um so erstaunlicher, dass der Laufzeitanstieg durch die langsamen Festplattenzugriffe nicht stärker ausfällt. Dies spricht für das exzellente Lokalitätsverhalten des *wotd*-Algorithmus.
Meine verbesserte Version des Partitionierungsalgorithmus ist allerdings die effizienteste Möglichkeit zur Konstruktion von Suffixbäumen auf DNA-Sequenzen, solange die Konstruktionsdaten noch komplett im Hauptspeicher gehalten werden können und eine reine Hauptspeicherkonstruktion durch den Ukkonen-Algorithmus nicht mehr möglich ist. Das von mir entwickelte *Scansort* ist innerhalb des *wotd*-Algorithmus eine effiziente Alternative zum *Countingsort*. Bei DNA-Sequenzen mit Längen bis zu ca. 10% der Hauptspeichergröße sind die Laufzeiten durch die Benutzung der beiden Sortierverfahren sehr ähnlich. Bei länger werdenden DNA-Sequenzen (Wurmchromosom 5) ist die Benutzung des *Scansorts* dem *Countingsort* durch seine geringeren Speicherplatzanforderungen, überlegen.

Andere Texte (Bibeln, Quellcode, Proteine, ...)

Die anderen hier untersuchten Texte haben wesentlich größere Alphabete als die DNA-Sequenzen. Die optimalen Einstellungen des Partitionierungsalgorithmus sind

String	String-länge	Alph.-größe	Laufzeiten				
			Part.-Tiefe 6	Part.-Tiefe 8	wotd-Scan.	wotd-Count.	Ukk.
Bibel (dän.)	3838509	83	32	32	50	42	71
Bibel (engl.)	4047392	63	32	35	51	46	151
Bibel (deut.)	4638707	91	40	42	63	56	4785
Bibel (franz.)	5122590	78	48	48	72	64	15143
Robocup	7243042	108	2620	2638	5504	5417	-
Gcc	14893334	99	187	185	396	427	-
Linux	20775894	105	322	316	1082	1219	-
Protein DB	20000000	83	221	232	738	824	-
FTP Index	4862809	88	51	51	86	88	-
CIA World	2473400	94	16	15	32	26	18

Tabelle 6.2: Konstruktionszeiten der verschiedenen Algorithmen auf unterschiedlichen Texten

für die hier betrachteten Alphabetgrößen zwischen 63 und 108 Zeichen die Partitionierungstiefen 2 oder 3. Für den *wotd*-Algorithmus sind das *Scan-* bzw. *Countingsort* dem *Quicksort* wieder überlegen. Darum findet das *Quicksort* hier keine

weitere Berücksichtigung. In der Tabelle 6.2 sind die Textlängen zusammen mit deren Alphabetgröße und den Laufzeiten für die besten Einstellungen der Algorithmen dargestellt.

Bibeln verschiedener Sprache

Für die Konstruktionszeiten der Bibeln ist nicht zu erwarten, dass die Sprache einen besonderen Einfluss hat. Wiederholen sich viele Textpassagen in einer Bibel, so werden die gleichen Wiederholungen sich auch in einer anderen Übersetzung wiederfinden.
Die normierten Konstruktionszeiten der Bibeln verschiedener Sprachen sind in Abbildung 6.11 nach aufsteigender Textlänge von links nach rechts dargestellt. Der Partitionierungsalgorithmus ist für jede Bibel die schnellste Konstruktionsvariante und der *wotd*-Algorithmus arbeitet bei den Alphabeten dieser Größe, wie nicht anders zu erwarten, am effektivsten mit dem *Countingsort*. Die normierten Konstruktionszeiten wachsen mit zunehmender Textlänge leicht an. Dabei ist eine Abhängigkeit von der Sprache nicht festzustellen.
Die Laufzeit des Ukkonen-Algorithmus ist nur bei der dänischen Bibel noch ver-

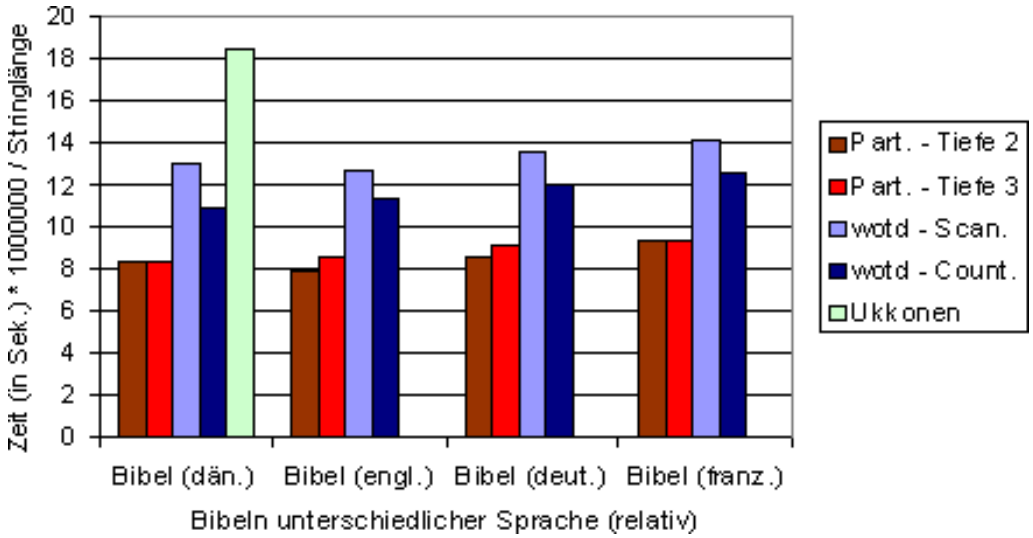


Abbildung 6.11: Konstruktionszeiten der verschiedenen Algorithmen auf unterschiedlichen Texten in Sekunden normiert auf 1000000 Zeichen

gleichbar mit den anderen Konstruktionsalgorithmen. Dagegen wird dieser Algorithmus für Bibeln mit mehr Zeichen sehr schnell langsamer.
Der Partitionierungsalgorithmus ist somit auch auf natürlichsprachigen Texten mit Textgrößen von 3 bis 5 % der Hauptspeichergröße das schnellste Konstruktionsverfahren.

Quellcode und sonstige Texte

In diesem Abschnitt werden die anderen verschiedenen Texte aus dem Information Retrieval, wie der Quellcode von Programmen, die Proteindatenbank, der FTP-Index und das *CIA World Fact Book* behandelt.
Für die Experimente war zu erwarten, dass die Laufzeiten der Algorithmen im wesentlichen von der Länge des Textes abhängen und die Textstruktur keinen großen Einfluss nimmt, da nicht zu erwarten ist, dass einer der Texte viele lange Wiederholungen aufweist.

Ergebnis Die absoluten Laufzeiten dieses Experiments sind in der Tabelle 6.2 enthalten. Zusätzlich sind in Abbildung 6.12, zur einfacheren Analyse, wieder die Laufzeiten relativ zur Textlänge visualisiert.
Der Partitionierungsalgorithmus ist für alle hier betrachteten Texte das effizienteste

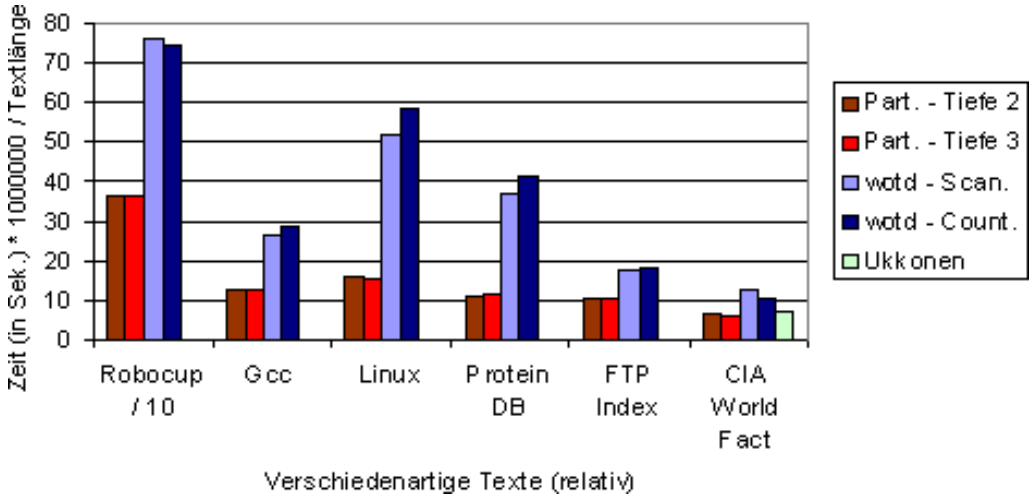


Abbildung 6.12: Konstruktionszeiten der verschiedenen Algorithmen auf unterschiedlichen Texten in Sekunden normiert auf 1000000 Zeichen

Konstruktionsverfahren. Für den Quellcode der verschiedenen Programme ist zu sehen, dass der Robocup-Quellcode die langsamsten Laufzeiten aufweist (Partitionierungsalg.: 2620 bzw. 2638 Sek., *wotd*: 5504 bzw. 5417 Sek.). Der Linux-Quellcode, der mit einer Länge von über 20 Millionen Zeichen fast 3-mal so lang ist wie der Robcup-Quellcode, zeigt dagegen nur Laufzeiten von 322 bzw. 316 Sekunden beim Partitionierungsalgithmus und 1082 bzw. 1219 Sekunden für den *wotd*-Algorithmus. Die relativen Laufzeiten des Robocup-Quellcodes mussten zur Visualisierung (Abb. 6.12) sogar noch durch 10 geteilt werden, um geeignet dargestellt werden zu können.
Für alle hier untersuchten Texte ist zu sehen, dass der *wotd*-Algorithmus mit dem *Scansort* ähnliche Laufzeiten wie mit dem *Countingsort* aufweist. Für den Linux-Quellcode ist die Gruppierung mit dem *Scansort* sogar um 11,2 % schneller als die

Benutzung des *Countingsorts*. Auch auf der Proteindatenbank ist das *Scansort* mit 10,4 % weniger Laufzeit effizienter.
Beim FTP-Index hat der Partitionierungsalgorithmus eine Laufzeit von 51 Sekunden und der *wotd*-Algorithmus benötigt 86 bzw. 88 Sekunden. Der Ukkonen-Algorithmus ist wie erwartet nur bei sehr kurzen Texten wie dem *CIA World Fact Book* konkurrenzfähig, jedoch bei steigender Textlänge nicht mehr geeignet.

Analyse Herausstechendes Ergebnis sind die langsamen Konstruktionszeiten von Suffixbäumen auf dem Robocup-Quellcode. Auffallend bei diesem Text ist, dass dieser sich nicht nur aus den üblichen Programmdateien mit den Endungen *.cpp* bzw. *.h* zusammensetzt, sondern dass er zusätzlich noch die Dateien zur Beschreibung der Benutzerschnittstelle enthält. Diese Dateien werden von der Programmierumgebung generiert und weisen deshalb sehr viele sich wiederholende Abschnitte auf. Ruft man sich die Ergebnisse der Experimente an den Fibonacci-Strings in Erinnerung, bei denen festgestellt wurde, dass weder der Partitionierungs- noch der *wotd*-Algorithmus effizient auf Texten mit vielen sich wiederholenden Teilstrings arbeitet, so wird klar, dass die sich wiederholenden Abschnitte der Benutzerschnittstellendefinitionen verantwortlich für die langsamen Laufzeiten sind.
In den Konstruktionszeiten des Suffixbaumes für die anderen Dateien spiegeln sich weitgehend die Ergebnisse der Basisuntersuchungen auf den zufällig erzeugten Texten wieder. Diese Texte führen zu einer weitgehenden Gleichgewichtung der Suffixbäume. Diese können damit sehr schnell durch den Partitionierungs- bzw. *wotd*-Algorithmus aufgebaut werden.
Außerdem ist erstaunlich, dass der *wotd*-Algorithmus mit dem integrierten *Scansort* für den Gcc- und Linux-Quellcode sowie auf der Proteindatenbank effizienter ist als mit dem *Countingsort*, obwohl diese Texte aus mehr als 80 Zeichen bestehen. Allerdings tritt dieses Verhalten nur bei den längeren Texten auf, so dass es im wesentlichen auf die geringen Speicherplatzanforderungen des *Scansorts* zurückzuführen ist.
Das *Scansort* innerhalb des *wotd*-Algorithmus eignet sich in der Praxis somit nicht nur für die Konstruktion von Suffixbäumen auf Strings mit kleinem zugrundeliegenden Alphabet, sondern auch für andere Texte, wie den Quellcode von Programmen oder die Proteindaten. Schnellster Konstruktionsalgorithmus auf Texten bis zu einer Länge von einem Sechstel der Hauptspeichergröße bleibt aber auch hier die von mir verbesserte Version des Partitionierungsalgorithmus. Natürlich wieder unter der Voraussetzung, dass eine reine Hauptspeicherkonstruktion durch den Ukkonen-Algorithmus nicht mehr möglich ist.

6.3 Speicherplatzanforderungen

Nachdem die Suffixbäume konstruiert wurden, liegen sie, je nach Konstruktionsalgorithmus, in verschiedenen Formaten vor. Die Speicherplatzanforderungen des persistenten Suffixbaum-Formats wie es durch den *wotd*-Algorithmus erzeugt wird, liegen bei 8 Bytes für jeden inneren Knoten und bei 4 Bytes für jedes Blatt (Abschnitt 4.1.3). Die gesamte Größe eines Suffixbaumes ist damit durch $12\,n$ Bytes beschränkt. In der Praxis liegen die Größen allerdings noch deutlich unter diesem Wert.

Ergebnis Die Speicherplatzanforderungen der Suffixbäume über den Texten, die im vorherigen Abschnitt untersucht wurden, sind in Tabelle 6.3 normiert auf Bytes pro Zeichen dargestellt. Die Spalte für die Suffixbäume aus dem Ukkonen-Algorithmus ist nicht vollständig, da nicht immer eine Konstruktion im gegebenen Zeitrahmen möglich war.
Die Speicherung in der relationalen Datenbank hängt nicht vom Konstruktionsalgorithmus ab. Die Knoten werden aus einer Textdatei, die vorher aus dem *wotd*-Format exportiert wurde, in die Datenbank geladen. Allerdings konnten auf Grund der hohen Speicherplatzanforderungen auch in diesem Fall nicht alle Strings untersucht werden. Der *nicht-geclusterte* Suffixbaum benötigt zwischen 8,08 und 9,39 Bytes für

Text	Text-länge	Suffixbaum			
		Gecl.	Nicht gecl.	Ukk.	Datenbank
E. coli	4638690	9,15	9,13	11,70	22,99 + 40,36
Wurm Chr. III	12836730	9,31	9,31	-	23,30 + 40,91
Wurm Chr. V	20551922	9,32	9,32	-	-
Protein DB	20000000	8,08	8,08	-	21,14 + 37,12
Bibel (dän.)	3838509	8,14	8,13	10,20	21,24 + 37,28
Bibel (engl.)	4047392	8,43	8,42	10,64	21,75 + 38,18
Bibel (deut..)	4638707	8,32	8,31	10,46	21,54 + 37,82
Bibel (franz.)	5122590	8,53	8,52	10,78	21,92 + 38,49
Robocup	7243042	9,39	9,36	-	23,38 + 41,05
Gcc	14893334	9,07	9,05	-	22,85 + 40,12
Linux	20775894	8,99	8,98	-	-
CIA World	2473400	8,36	8,32	10,48	21,57 + 37,87
FTP Index	4862809	8,60	8,58	10,87	22,02 + 38,67
Mittelwerte	9686386	8,75	8,73	10,74	22,16 + 38,90

Tabelle 6.3: Speicherplatzanforderungen der Suffixbäume pro indiziertes Zeichen (Datenbank: Daten + Index)

jedes indizierte Zeichen und das *geclusterte* Format erfordert nur einige Bytes mehr. Die Speicherplatzanforderungen der hier verwendeten Suffixbaum-Repräsentation

des Ukkonen-Algorithmus sind deutlich höher. Sie benötigen auf den hier indizierten Texten, für die eine Konstruktion durch den Ukkonen-Algorithmus möglich war, zwischen $10,2\,n$ und $11,7\,n$ Bytes.
Am meisten Speicherplatz erfordert allerdings die Speicherung in der relationalen Datenbank. Diese belegt dabei nicht nur sehr viel Speicher für die Datentabellen, sondern benötigt darüberhinaus noch einen Index zur effizienteren Verfolgung der logischen Adressen. Die Suffixbäume, deren Abbildung auf die Datenbank untersucht wurde (siehe Tabelle 6.3), benötigen allein für die Datentabellen einen Speicherplatz von über $21\,n$ Bytes. Die Ausmaße des hier verwendeten B-Baum-Indexes erhöhen diesen Wert um weitere $37,12\,n$ bis $41,05\,n$ Bytes.

Analyse Der *nicht-geclusterte* Suffixbaum liegt mit weniger als 10 Bytes pro Zeichen des indizierten Strings deutlich unter dem Maximum von 12 Bytes pro Eingabezeichen, da ein innerer Knoten im Mittel mehr als 2 Kinder besitzt. Das *geclusterte* Format erfordert für die zusätzlichen Knoten, die durch das Abtrennen der Zweige und das daraus resultierende Aufsplitten der Kanten entstehen, nur einige Bytes mehr.
Die hier verwendete Implementierung des Ukkonen-Algorithmus benötigt zur Repräsentation der Kantenbeschriftungen abgesehen vom linken auch den rechten Index und damit 4 zusätzliche Bytes. Damit sind die Speicherplatzanforderungen der Suffixbäume mit im Mittel $10,74\,n$ Bytes deutlich höher. Allerdings existieren auch bessere Repräsentationen für die Linearzeitalgorithmen. Beispielsweise schlägt Kurtz [37] ein Hauptspeicherformat für den McCreight-Algorithmus [44] vor, welches im Mittel nur $10,1\,n$ Bytes benötigt und dabei noch die Suffixlinks beinhaltet. Die Knoten werden dabei in einem Array gespeichert, so dass sich dieses Format auch leicht auf den Hintergrundspeicher übertragen lässt.
Die Speicherung in der relationalen Datenbank erfordert am meisten Speicherplatz, weil in den Tabellen Schlüssel zur Verfolgung der logischen Referenzen gespeichert werden müssen. Dabei wird nicht nur sehr viel Speicher von den Datentabellen belegt, sondern darüberhinaus noch für einen Index zur effizienteren Verfolgung dieser logischen Adressen. Würde aus Platzgründen auf die Indizierung verzichtet werden, so hätte das einen starken Laufzeitanstieg bei der Mustersuche zur Folge, da zum Auffinden der logischen Adressen im schlechtesten Fall alle Tupel durchlaufen werden müssten. Die Suffixbaum-Struktur würde dadurch ihren Wert verlieren.
Die Speicherplatzanforderungen des *geclusterten* bzw. *nicht-geclusterten* Suffixbaum-Formats liegen für alle hier indizierten Strings deutlich unter 10 Bytes pro Zeichen und sind damit, bezogen auf die Speicherplatzanforderungen, sehr gut für die Speicherung der Suffixbäume geeignet. Dagegen ist der hohe Platzbedarf bei der Speicherung in einer relationalen Datenbank zu hoch ist, um Suffixbäume auf langen Texten auf diese Art zu speichern. Aus diesem Grund scheint diese Möglichkeit zur persistenten Datenhaltung der Suffixbäume ungeeignet zu sein.

Welche Art der persistenten Speicherung die Geeignete ist, hängt aber außer von den Speicherplatzanforderungen noch von der Laufzeiteffizienz der Operationen auf dem persistenten Suffixbaum-Format ab.

6.4 Untersuchungen der Mustersuche

Nachdem die Suffixbäume konstruiert und gespeichert wurden, dienen sie als Index für viele Operationen, wie die Mustersuche. Entscheidend für die Effizienz dieser Operationen ist der schnelle Zugriff auf die persistent gespeicherten Suffixbäume. In diesem Abschnitt soll die Laufzeiteffizienz der Mustersuche auf den verschiedenen Suffixbaum-Formaten experimentell untersucht werden. Diese Formate sind die *geclusterte* und *nicht-geclusterte* Suffixbaum-Repräsentation sowie die Speicherung in einer relationalen Datenbanktabelle.
Die theoretische Laufzeit der Suche eines Musters w auf einem Suffixbaum liegt in $O(|w|)$. Damit ist die Suchmusterlänge der entscheidende Parameter dieser Operation.
Bei der Betrachtung, der für diese Arbeit relevanten Bereiche, ergeben sich Anwendungen wie die Suche auf DNA-Sequenzen oder das Auffinden von Textabschnitten in großen Textcorpora. Erfahrungen aus der Praxis besagen, dass die Suchmusterlängen bei der Suche auf DNA-Sequenzen Werte von 200 Zeichen selten übersteigen und oft kürzer sind. Zu kurze Muster differenzieren allerdings die Teilworte nicht ausreichend, so dass Musterlängen unter 10 Zeichen in der Praxis nicht relevant sind. Auch im Information Retrieval werden häufig kürzere Suchmuster verwendet, aber auch hier gibt es Anwendungsfälle, wie beispielsweise die Plagiatsuche, in denen die Musterlängen bis zu 200 Zeichen erreichen können.
Neben der Suchmusterlänge sind auch der Verzweigungsgrad der Knoten und die Größe des Suffixbaumes wichtige Einflussgrößen bei der Mustersuche. Ein großes Alphabet führt zu einer langsameren Laufzeit, da im Mittel mehr Kinder eines Knotens betrachtet werden müssen, um das passende zu finden und dann mit der Suche fortzufahren. Auch die Größe eines Suffixbaumes wirkt sich auf den Verzweigungsgrad aus, weil bei einer größeren Anzahl von Suffixen die erwartete Anzahl der Knoten in den oberen Ebenen des Suffixbaumes ansteigt. Dieses hat zusätzlich den Effekt, dass die Speicherplätze der Knoten aus aufeinander folgenden Ebenen weit voneinander entfernt sind und somit im erwarteten Fall mehr Festplattenblöcke gelesen werden müssen.
In den hier durchgeführten Experimenten soll die Mustersuche auf einem Teil der in Abschnitt 6.2.5 untersuchten Suffixbäume betrachtet werden. Die kleineren Suffixbäume (Bibeln, FTP-Index, *CIA World Fact Book*) werden nicht mit in die Untersuchungen einbezogen, weil sie vollständig in den Hauptspeicher passen und die Art der Speicherung damit nur einen geringen Einfluss auf die Suchzeiten hat.

In den Experimenten wurden für jeden String die Gesamtlaufzeit für 10000 Suchoperationen auf den persistent gespeicherten Suffixbäumen bei Musterlängen zwischen 10 und 200 gemessen. Als Suchmuster wurden zufällig ausgewählte Teilstrings aus dem Eingabetext verwendet. Diese Vorgehensweise ist besser geeignet als die Suche nach beliebig erzeugten Mustern, weil die Tiefe, bis zu der im Baum gesucht wird, konstant ist. Bei beliebig erzeugten Mustern ist das nicht der Fall und die Ergebnisse wären dadurch weniger aussagekräftig.
Beim *geclusterten* Format wurde entsprechend den Partitionierungstiefen die Höhe des Suffixbaum-Stumpfes bei DNA-Sequenzen auf 9 festgelegt und bei anderen Texten mit größerem Alphabet auf 3. Die Suffixbaum-Tabelle in der relationalen Datenbank wurde vor den Messungen mit dem MySQL-Befehl "OPTIMIZE TABLE" am Index ausgerichtet, um eine optimale Laufzeit gewährleisten zu können.
Es war zu erwarten, dass die Suchzeiten auf dem *nicht-geclusterten* Suffixbaum bei kurzer Musterlänge schneller sind als die auf dem *Geclusterten*, da die oberen Ebenen des Baumes in diesem Fall noch komplett in den Hauptspeicher passen. Sobald dies bei steigender Musterlänge jedoch nicht mehr zutrifft, sollte die Laufzeit auf dem *geclusterten* Suffixbaum schneller werden als auf dem *nicht-geclusterten* Format.
Die Zugriffe auf die relationale Datenbank sollten sehr viel langsamer sein als die anderen beiden Techniken, weil durch die logische Adressierung das zusätzlich Durchlaufen des B-Baum-Indexes sehr viel Zeit kostet.

Ergebnis In der Abbildung 6.13 sind die Laufzeiten von 10000 Suchoperationen auf dem dritten Chromosom des Wurmes *C. elegans* und dem Gcc-Quellcode dargestellt. Die Mustersuchen auf der Datenbank sind nicht enthalten, weil die Laufzeiten mehr als zwanzig mal schlechter sind als die hier visualisierten und damit nicht mehr geeignet dargestellt werden können. Auffällig ist, dass bei einer Musterlänge von 10 Zeichen die Suchzeit auf dem *nicht-geclusterten* Suffixbaum des Wurmchromosoms mit 13 Sekunden viel kleiner ist als die entsprechende Laufzeit beim *geclusterten* Suffixbaum mit 91 Sekunden. Für den Gcc-Quellcode sind die Laufzeiten mit 113 bzw. 84 Sekunden dagegen nicht soweit voneinander entfernt. Mit zunehmender Musterlänge werden die Suchen auf den *nicht-geclusterten* Suffixbäumen schnell langsamer und zwischen 30 und 40 Zeichen tritt eine Sättigung mit Laufzeiten von ca. 135 bzw. 250 Sekunden ein. Dagegen nehmen die Laufzeiten auf dem *geclusterten* Suffixbaum ab der Musterlänge von 20 Zeichen schon nicht mehr merklich zu. Die 10000 Suchoperationen benötigen für die untersuchten Musterlängen über 20 Zeichen ca. 75 Sekunden auf dem Suffixbaum des Wurmchromosoms bzw. ca. 105 Sekunden auf dem des Gcc-Quellcodes.
Dieses Sättigungsverhalten ab einer Musterlänge von ca. 30 Zeichen war auch bei den Laufzeitmessungen auf den anderen hier untersuchten Suffixbäumen zu beobachten. Aus diesem Grund sind in den Abbildungen 6.14 und 6.15 nur die Laufzeiten bei ei-

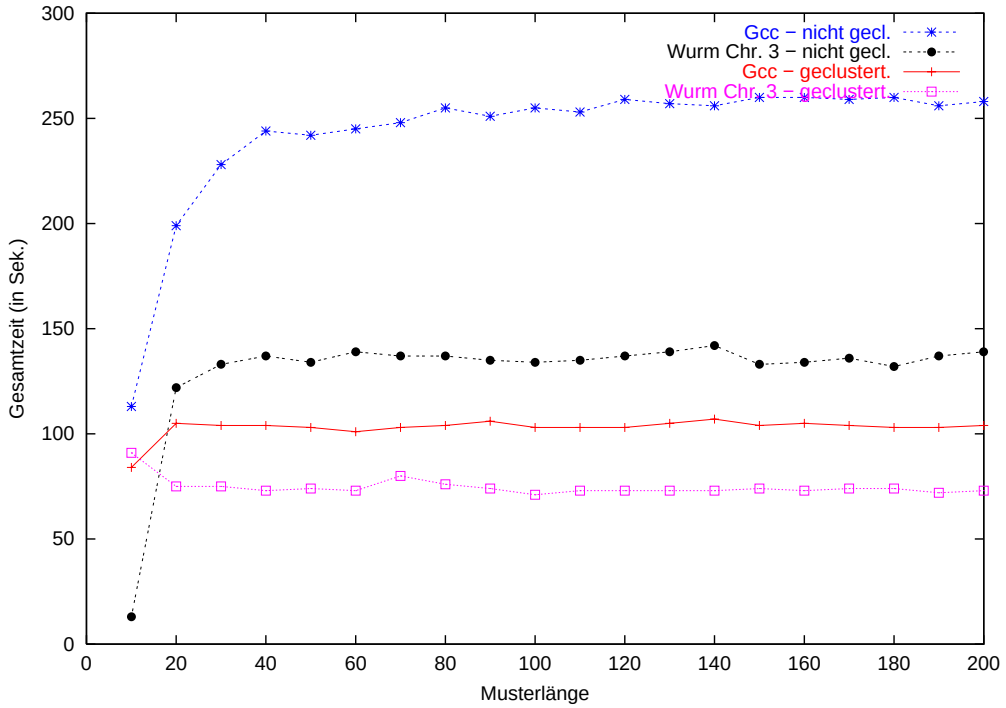


Abbildung 6.13: Zeit für 10000 Suchen auf dem Gcc-Quellcode und auf dem dritten Chromosom des Wurmes auf dem geclusterten und nicht geclusterten Suffixbaum

ner Musterlänge von 10 und 30 Zeichen dargestellt. Die Zeiten der Datenbanksuche sind in den Abbildungen durch 10 geteilt dargestellt, um sie bei den hohen Werten noch geeignet visualisieren zu können. Die Geschwindigkeiten dieser Suchen sind für die hier untersuchten Texte bei den Musterlängen zwischen 10 und 200 Zeichen zwischen zehn und hundertmal so langsam wie die Mustersuche auf den anderen beiden Formaten.

Bei den kleineren Suffixbäumen ist zu beobachten, dass die Laufzeiten der Suche auf der *geclusterten* und der *nicht-geclusterten* Repräsentation sehr ähnlich sind. Die Suchen auf dem *geclusterten* Suffixbaum des Robocup-Quellcodes benötigen 20 Sekunden bei einer Musterlänge von 10 Zeichen und 24 Sekunden bei 30 Zeichen. Die Werte für den *nicht-geclusterten* Baum weichen mit 16 bzw. 25 Sekunden nur ein wenig ab.

Bei den größeren Suffixbäumen, deren Speicherplatzanforderungen die Hauptspeichergrenzen überschreiten, ergeben sich allerdings größere Unterschiede, die von der Art der Sequenz und der Länge des Suchmusters abhängen. Bei einer Musterlänge von 10 Zeichen ist die Suche auf den *nicht-geclusterten* Suffixbäumen der Wurmchromosomen deutlich schneller als die auf dem *geclusterten* Format. Die Suchoperatio-

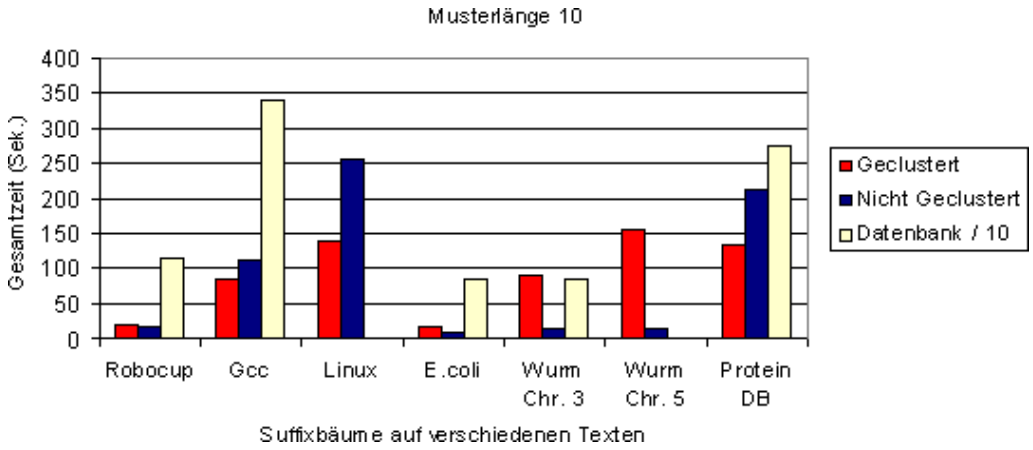


Abbildung 6.14: Gesamtzeiten für 10000 Suchen mit Mustern der Länge 10 auf den persistenten Suffixbaum-Repräsentationen verschiedener Texte

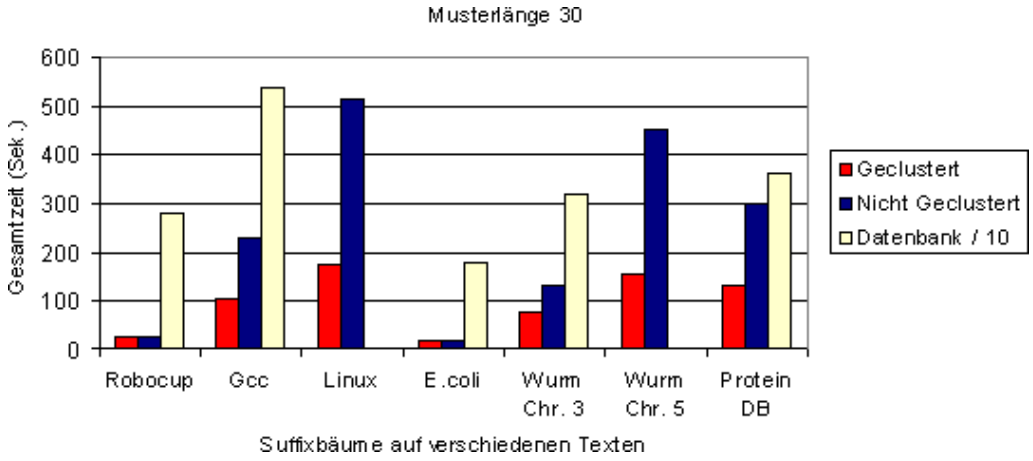


Abbildung 6.15: Gesamtzeiten für 10000 Suchen mit Mustern der Länge 30 auf den persistenten Suffixbaum-Repräsentationen verschiedener Texte

nen benötigen für das Chromosom 5 mit 157 Sekunden für die *geclusterte* Variante und 15 Sekunden für die *nicht-geclusterte* Repräsentation mehr als zehnmal soviel Zeit. Nimmt nun die Musterlänge zu, verändern sich die Verhältnisse zu Gunsten des *geclusterten* Formats, für das die Suchoperationen dann beim Chromosom 5 des Fadenwurmes dreimal schneller sind. Für die anderen Textstrukturen, wie den Quellcode und die Proteindaten ist die *geclusterte* Repräsentation schon ab 10 Zeichen Musterlänge effizienter als das *nicht-geclusterte* Format. Die Suchen für den Linux-Quellcode benötigen hier 140 bzw. 257 Sekunden. Dieser Vorteil erhöht sich bei 30 Zeichen Musterlänge noch. Die Suchzeiten betragen dabei auf dem *geclusterten* Format 173 Sekunden und auf dem *nicht-geclusterten* Suffixbaum 513 Sekunden. Neben dem Vergleich zwischen den verschiedenen persistenten Speichermechanismen ist festzustellen, dass die Laufzeiten bei steigender Größe der Suffixbäume zunehmen, obwohl in der Theorie die Komplexität des Mustersuche auf Suffixbäumen unabhängig von der Größe des zugrundeliegenden Textes ist.

Analyse Zur Analyse der Laufzeiten ist die Feststellung der unterschiedlichen Einflussparameter sehr wichtig. In diesen Experimenten sind dies die Suchmusterlänge, die Art des Textes und die Größe des Suffixbaumes.
Die Sättigung des Laufzeitanstiegs ab einer Suchmusterlänge von 30 Zeichen ist durch die, mit weniger Knoten besetzten, unteren Ebenen des Suffixbaumes zu erklären, da durch diese geringere Dichte an Knoten auch die Kantenbeschriftungen längere Teilabschnitte des zugrundeliegenden Strings beschreiben. Der Vergleich dieser langen Teilabschnitte entlang des Musters, ist deshalb sehr schnell zu berechnen. Außerdem ist durch die dicht besetzten oberen Ebenen der erwartete Verzweigungsgrad in der Nähe der Wurzel höher, so dass in einem Knoten, zur Bestimmung eines dem Muster entsprechenden Kindes, im Mittel mehr Elemente aus der Kindliste durchlaufen werden müssen.
Die Laufzeiten bei kleinen Suffixbäumen (Robocup, *E. coli*) des *geclusterten* und *nicht-geclusterten* Formats gleichen sich, weil diese Suffixbäume komplett im Hauptspeicher gehalten werden können. Damit wird ein Speicherblock nur einmal gelesen und verbleibt dann für alle weiteren Suchoperationen im Hauptspeicher.
Die Art des Textes hat besonderen Einfluss auf die Alphabetgröße und damit den erwarteten Verzweigungsgrad der Knoten. Der Vorteil des *nicht-geclusterten* Formats bei der Suche mit kleiner Musterlänge 10 auf den Wurmchromosomen begründet sich durch die Speicherplatzanforderungen der oberen Ebenen bis zur Tiefe 10 des Suffixbaumes. Diese liegen unterhalb der Hauptspeichergröße und können deshalb über alle Suchoperationen dauerhaft im Hauptspeicher gehalten werden, während bei den Suchoperationen auf dem *geclusterten* Format verschiedene Blöcke mehrmals von der Festplatte gelesen werden müssen. Bei größeren Alphabeten allerdings, wie sie beim Quellcode oder bei den Proteinen auftreten, geht der Suffixbaum mehr in die Breite und die oberen Ebenen sind mit mehr Knoten besetzt, so dass diese

nicht vollständig im Hauptspeicher gehalten werden können. In diesem Fall ist das *geclusterte* Format effizienter, da die am meisten benutzten Knoten dauerhaft im Hauptspeicher gehalten werden.
Das *geclusterte* Format ist immer dann schneller als der *nicht-geclusterte* Suffixbaum, wenn die Suchmusterlänge die Anzahl der Ebenen, die im Hauptspeicher Platz finden, übersteigt. Dies hängt im wesentlichen vom Verzweigungsgrad und der Suchmusterlänge ab.
Die längeren Laufzeiten auf den größer werdenden Suffixbäumen entstehen durch die häufigeren Festplattenzugriffe. Bei kleineren Suffixbäumen sind die Knoten benachbarter Ebenen recht nah zusammen gespeichert, so dass sich die Kinder eines Knotens oftmals mit dem Vater auf dem selben Festplattenblock befinden. Je größer der Suffixbaum wird, um so weiter rücken diese Knoten auseinander, so dass mehr Festplattenzugriffe notwendig sind.
Die Suchoperationen auf der relationalen Datenbank sind im Vergleich zu den anderen beiden Formaten sehr langsam. Dies lässt sich vor allem durch die logische Adressierung der Knoten und durch die höheren Speicherplatzanforderungen für einzelne Knoten erklären. Die logische Adressierung hat zur Folge, dass der physische Speicherplatz eines Knotens erst durch das zusätzliche Durchlaufen eines B-Baum-Indexes bestimmt werden kann, was sehr viel zusätzliche Zeit kostet. Außerdem sind die Speicherplatzanforderungen pro Knoten sehr hoch, was dazu führt, dass weniger Knoten in einem Festplattenblock enthalten sind und somit mehr Blöcke gelesen werden müssen. Dieses wirkt sich negativ auf die Laufzeiten aus.
Zur Mustersuche bleibt abschließend zu sagen, dass die von mir entwickelte *geclusterte* Speicherung eines Suffixbaumes bei längeren Suchwörtern effizienter ist als die *nicht-geclusterte* Speicherung, die allerdings mit kleinerer Musterlänge, bei der die gesamten zur Suche benötigten Knoten der oberen Ebenen in den Hauptspeicher passen, besser ist. Eine relationale Datenbank ist in der hier beschriebenen Form nicht geeignet für die Speicherung eines Suffixbaumes, weil die Zugriffe auf die Tabellen zu langsam sind.

Kapitel 7

Zusammenfassung

Ziel dieser Arbeit war die Untersuchung von Suffixbäumen auf ihre Anwendbarkeit in Genom-Datenbanken und im Information Retrieval. Die zu verarbeitenden Strings aus diesen Gebieten nehmen sehr große Längen an, so dass sich die dadurch ergebenden langen Konstruktionszeiten der Suffixbäume nur über viele Suchoperationen amortisieren können. Dafür sind einerseits schnelle Konstruktionalgorithmen erforderlich und andererseits effiziente persistente Speichermechanismen, um die Suffixbäume nicht mehrmals aufbauen zu müssen.

Zunächst wurden grundlegende Definitionen bezüglich der Suffixbaum-Struktur gegeben und wichtige Eigenschaften, wie die linearen Speicherplatzanforderungen, beschrieben, die eine Indizierung von langen Texten erst ermöglichen. Danach wurde die Mustersuche, die klassische Operation auf Suffixbäumen und weitere Anwendungen dieser vielseitigen Datenstruktur beschrieben.
Die wichtigste Eigenschaft der Konstruktionsalgorithmen ist die Anwendbarkeit für lange Texte, wie sie in den hier betrachteten Bereichen häufig auftreten. Es wurde festgestellt, dass die theoretisch optimalen Linearzeitalgorithmen auf Grund ihrer hohen Anforderung an die Hauptspeichergröße in der Praxis nicht geeignet sind. Deshalb wurden zwei Algorithmen, die auch den Hintergrundspeicher mit in die Konstruktion einbeziehen, als Basis für die weiteren Untersuchungen ausgewählt. Dies sind der Partitionierungsalgorithmus von Hunt *et al.* und der *wotd*-Algorithmus, dessen Implementierung von Stefan Kurtz als eine der schnellsten zur Konstruktion von Suffixbäumen gilt.
Die Analyse des Partitionierungsalgorithmus ergab, dass dieser eine Komplexität von $\Theta(n^2)$ aufweist. Außerdem ist er auf Grund einer fehlenden Kollisionsbehandlung bei der Partitionierung nicht stabil bezüglich Entartungen des Suffixbaumes und berücksichtigt aktuelle Rechnerarchitekturen mit blockorientierten Speichermedien nicht. Daraufhin wurden von mir Modifikationen vorgenommen, so dass die erwartete Komplexität des veränderten Algorithmus nun im erwarteten Fall in $O(n \log n)$

und bei geeigneter Wahl der Partitionierungstiefe sogar in $O\left(n\ log\left(\frac{n}{log\,n}\right)\right)$ liegt. Desweiteren integriert mein verbesserter Algorithmus eine Kollisionsbehandlung sowie den blockorientierten Zugriff. Damit ist er stabil bezüglich Entartungen und berücksichtigt die blockorientierten Zugriffsmechanismen heutiger Rechnerarchitekturen.
Für den *wotd*-Algorithmus wurde festgestellt, dass er ein sehr gutes Lokalitätsverhalten aufweist und seine Effizienz im wesentlichen vom integrierten Sortieralgorithmus abhängt. Diesbezüglich wurden das *Countingsort* und das *Quicksort* mit einem von mir entwickelten Sortierverfahren – dem *Scansort* – verglichen. Dieses *Scansort* erlaubt eine *"in place"* Sortierung und ist mit einer Komplexität von $O(n|\Sigma|)$ besonders effizient bei kleinen Alphabeten.

Einen anderen wichtigen Teil dieser Arbeit stellten die Untersuchungen zur geeigneten Speicherung der Suffixbäume dar. Diese Speicherung sollte speicherplatzeffizient sein sowie den Operationen einen schnellen Zugriff auf den persistent gespeicherten Suffixbaum erlauben und gegebenenfalls in schon bestehende Datenbanktechnologien eingebunden werden können.
Als Grundlage für diese persistente Speicherung wurde ein speicherplatzeffizientes Suffixbaum-Format von Giegerich *et al.* [23] genutzt, was mit maximal 12 Bytes pro Eingabezeichen auskommt. Darauf aufbauend habe ich eine *Clusterung* des Suffixbaumes entworfen, die das Zugriffsverhalten über viele Suchoperationen berücksichtigt.
Neben dem speziellen Suffixbaum-Format wurde zusätzlich die Speicherung der Suffixbaum-Struktur in einer relationalen Datenbank diskutiert. Zu diesem Zweck wurde der Suffixbaum auf Tabellen abgebildet. Dabei ist die logische Adressierung der größte Nachteil dieses Verfahrens, weil sie zu erhöhten Speicherplatzanforderungen und einem Effizienzverlust führt. in den relationalen Datenbank, die zu erhöhten Speicherplatzanforderungen und einem Effizienzverlust führt.

Die Konstruktionsverfahren und die persistenten Speichermechanismen wurden dann in Experimenten, anhand eigener Implementierungen, untersucht.
Die untersuchten Konstruktionsalgorithmen waren der verbesserte Partitionierungsalgorithmus ohne den blockorientierten Zugriff und ohne die Kollisionsbehandlung, der *wotd*-Algorithmus mit den integrierten Sortieralgorithmen und der Ukkonen-Algorithmus als Vertreter der Linearzeitalgorithmen. Die Laufzeiteffizienz dieser Algorithmen wurden an zufällig erzeugten Strings, an Fibonacci-Strings und an realen Texten überprüft.
Für den Partitionierungsalgorithmus haben sich bei einer Alphabetgröße 4 (DNA-Sequenzen) bzw. 90 (allgemeinsprachliche Texte) die Partitionierungstiefen zwischen 6 und 8 bzw. die Tiefen 2 und 3 als effizienteste Einstellungen erwiesen. Innerhalb des *wotd*-Algorithmus ist die Benutzung des *Scansorts* bei kleinen Alphabeten eine

effiziente Alternative zum *Countingsort*, wogegen sich das *Quicksort* als nicht konkurrenzfähige gezeigt hat.
Der Vergleich der verschiedenen Konstruktionsverfahren an zufällig erzeugten Strings hat erwiesen, dass die Konstruktion durch den Ukkonen-Algorithmus ab einer Stringlänge von einem Fünfundzwanzigstel der Hauptspeichergröße nicht mehr in angemessener Zeit möglich ist. Der verbesserte Partitionierungsalgorithmus bzw. der *wotd*-Algorithmus ermöglichen dagegen auch den effizienten Aufbau von größeren Suffixbäumen. Sobald die Größe des Suffixbaumes die Hauptspeichergröße überschreitet ist der Partitionierungsalgorithmus bei Texten bis zu einem Sechstel der Hauptspeichergröße das deutlich schnellste Konstruktionsverfahren. Sofern sich diese Ergebnisse auf größere Speicher übertragen ließen würde er für die Indizierung des menschlichen Genoms, das eine Länge von ca. $3,2 \times 10^9$ Zeichen aufweist, einen Hauptspeicher von 20 MB Größe erfordern. Steigt die Länge des Textes jedoch weiter an, so ist der *wotd*-Algorithmus das effizientestes Konstruktionsverfahren.
Diese Aussagen gelten allerdings nur solange der Suffixbaum nicht entartet. Die Untersuchungen an Fibonacci-Strings haben ergeben, dass weder der Partitionierungs- noch der *wotd*-Algorithmus für Texte, die lange sich wiederholende Teilstrings aufweisen, geeignet ist.
Die hier untersuchten verschiedenartigen Texte aus den Bereichen Genom-Daten und Information Retrieval resultierten aber nicht in einer Entartung des Suffixbaumes. Die Laufzeiten spiegelten deshalb die Ergebnisse der Untersuchungen an den zufällig erzeugten Strings wieder. Auch in diesem Fall hat sich der Partitionierungsalgorithmus für die untersuchten Texte, die Längen zwischen einem Vierzigstel und einem Sechstel der Hauptspeichergröße aufwiesen, als schnellstes Konstruktionsverfahren herausgestellt.

Nachdem die persistenten Suffixbäume der in der Praxis verwendeten Texte nun konstruiert waren, wurden sie bezüglich der Speicherplatzanforderungen und der auf ihnen arbeitenden Mustersuche untersucht.
Die Suffixbäume der *geclusterten* und der *nicht-geclusterten* Suffixbaum-Repräsentation benötigen im Mittel $8,75$ bzw. $8,73$ Bytes pro Eingabezeichen, während die Speicherung in der relationalen Datenbank allein für die Datentabellen im Mittel $22,16\,n$ Bytes benötigt und durch einen zusätzlichen Index noch weitere $38,9\,n$ Bytes hinzukommen.
Die Untersuchungen der Mustersuche über viele Suchanfragen haben gezeigt, dass die *nicht-geclusterten* Suffixbaum-Repräsentation die schnellsten Zugriffszeiten aufweist, solange der zu durchsuchende Teil des Baumes komplett im Hauptspeicher gehalten werden kann. Ist dies durch größere Bäume und größere Musterlängen nicht mehr der Fall, so zeigt die Mustersuche auf dem *geclusterten* Format die schnellsten Laufzeiten.
Die Suchen auf den in Tabellen der relationalen Datenbank gespeicherten Suffix-

bäume weisen mehr als zwanzig mal langsamere Laufzeiten auf als die auf den anderen beiden Repräsentationen. Damit hat sich diese Art der Speicherung in einer relationalen Datenbank als ungeeignet erwiesen hat.

7.1 Schlussfolgerungen

Der von mir verbesserte Partitionierungsalgorithmus hat bei geeigneter Partitionierungstiefe eine Komplexität von $O\left(n\ log\left(\frac{n}{log\ n}\right)\right)$ und weist auch in der Praxis bei langen Strings bis zu Größen von einem Sechstel der Hauptspeichergröße die besten Konstruktionszeiten für Suffixbäume auf.
Der *wotd*-Algorithmus ist bei noch längeren Texten (> ein Sechstel der Hauptspeichergröße) das schnellste Konstruktionsverfahren und profitiert bei kleinen Alphabeten von der Benutzung des *Scansorts*.

Für die persistente Speicherung weisen sowohl die *nicht-geclusterten* Suffixbaum-Repräsentation wie auch das *geclusterte* Format geringe Speicherplatzanforderungen als auch geringe Laufzeiten bei der Mustersuche auf. Bei kleineren Suffixbäumen bzw. kurzen Mustern, bei denen der zu durchsuchender Teil des Baumes komplett im Hauptspeicher gehalten werden kann, sind die Laufzeiten der Mustersuchen auf dem ersten Format schneller. Bei größeren Suffixbäumen und langen Mustern ist letzteres effizienter.
Die Abbildung der Suffixbäume auf die Tabellen einer relationalen Datenbank ist zur persistenten Speicherung nicht geeignet.

Kapitel 8

Ausblick

Die zukünftige Arbeit kann in vier verschiedene Bereiche geteilt werden.

- Verbesserung und Optimierung der Konstruktionsalgorithmen
- Weiterführende experimentelle Untersuchungen auf größeren Systemen mit längeren Texten
- Integration von Datenbanktechnologien (*Recovery*-Mechanismen)
- Integration der Algorithmen in Anwendungsprogramme

Beim Prototypen des Partitionierungsalgorithmus wird die Zuordnung der einzelnen Suffixe zu den Partitionen in einem langen Array gespeichert, dass bei der Konstruktion für jeden Zweig komplett durchlaufen werden muss. Hier könnte eine geeignete *Clusterung* dieser Partitionen einen weiteren Effizienzgewinn zur Folge haben. Außerdem ist eine Implementierung des blockorientierten Zugriffs und der Verfeinerung bei Kollisionen vorgesehen, was sich positiv auf die Konstruktionszeiten bei Texten mit langen sich wiederholenden Teilstrings auswirken sollte. Für den *wotd*-Algorithmus wäre die Entwicklung von Sortierverfahren interessant, die auch den Hintergrundspeicher einbeziehen, so dass auch für Strings, deren Suffixarray nicht komplett im Hauptspeicher gehalten werden kann, eine effiziente Suffixbaum-Konstruktion möglich wäre.
Die in dieser Arbeit durchgeführten experimentellen Untersuchungen mussten aus technischen Gründen auf einem Computer mit relativ kleinem Hauptspeicher durchgeführt werden. Allerdings wären weiterführende Untersuchungen notwendig, um festzustellen, ob sich die Ergebnisse wirklich auf größere Systeme mit längeren Strings übertragen lassen.
Bei sehr langen Texten können auch die Laufzeiten des Partitionierungsalgorithmus mehrere Stunden betragen. *Recovery*-Mechanismen, wie sie in Datenbanktechnologien verwendet werden, können hier verwendet werden, um nach einem Systemfehler

schon konstruierte Zweige nicht erneut aufbauen zu müssen.
Die wichtigste zukünftige Aufgabe wird jedoch die Integration der persistenten Suffixbäume in biologische Anwendungssoftware zur Analyse von Genom-Daten sein. Solche Werkzeuge speichern ihre Daten zumeist in Datenbanken, so dass die Integration der Suffixbäume in diese Technologien ein weiteres wichtiges Ziel bleibt.

Literaturverzeichnis

[1] S. F. Altschul, W. Gisch, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215(3):403–410, 1990.

[2] A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. *Algorithmica*, 23(3):246–260, 1999.

[3] A. Andersson and S. Nilsson. Efficient implementation of suffix trees. *Software: Practice and Experience*, 25(2):129–141, 1995.

[4] A. Apostolico and W. Szpankowski. Self-alignments in words and their applications. *J. Algorithms*, 13(3):446–467, 1992.

[5] M. P. Atkinson, F. Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. In *Proceedings of the First International Conference on Deductive and Object-Oriented Databases*, pages 223–240, Kyoto, Japan, 1989.

[6] D. Axmark, M. M. Widenius, J. Cole, A. Lentz, and P. DuBois. *MySQL Reference Manual*. MySQL AB, 2001.

[7] R. A. Baeza-Yates and Navarro G. A hybrid indexing method for approximate string matching. *Journal of Discrete Algorithms*, 1(1):205–239, 2000.

[8] B. S. Baker. Parameterized duplication in strings: Algorithms and an application to software maintenance. *SIAM J. Computing*, 26(5):1343–1362, 1997.

[9] B. Balkenhol, S. Kurtz, and M. Yuri. Modifications of the burrows and wheeler data compression algorithm). In *Proceedings of the IEEE Data Compression Conference*, pages 188–197. IEEE Computer Society Press, 1999.

[10] H. Behrens. Lokalitätsverhalten von Algorithmen zur Sequenzanalyse. Master's thesis, Technische Fakultät der Universität Bielefeld, AG Praktische Informatik, 1995.

[11] J. L. Bentley and R Sedgewick. Fast algorithms for sorting and searching strings. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, 1997.

[12] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht. Complete inverted files for efficient text retrieval and analysis. *Journal of the ACM (JACM)*, 34(3):578–595, 1987.

[13] R. S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communications of the ACM*, 20(10):762–772, 1977.

[14] S. Burkhardt, A. Crauser, P. Ferragina, H.-P. Lenhof, E. Rivals, and M. Vingron. q-gram based database searching using a suffix array (quasar). In *Proceedings of the third annual international conference on Computational molecular biology*, pages 77–83. ACM Press, 1999.

[15] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press, 1990.

[16] A. L. Delcher, S. Kasif, R. D. Fleischmann, J. Peterson, O. White, and S. L. Salzberg. Alignment of whole genomes. *Nucleic Acids Research*, 27(11):2369–2376, 1999.

[17] B. Dorohonceanu and C. G. Neville-Manning. Accelerating protein classification using suffix trees. In *Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology (ISMB)*, pages 128–133. AAAI Press, 2000.

[18] M. Farach. Optimal suffix tree construction with large alphabets. In *Proc. of the 38th Annual Symposium on the Foundations of Computer Science (FOCS 97)*, pages 137–143, 1997.

[19] M. Farach and S. Muthukrishnan. Perfect hashing for strings: Formalization and algorithms. In D. S. Hirschberg and E. W. Myers, editors, *Proceedings of the 7th Annual Symposium on Combinatorial Pattern Matching*, pages 130–140, Laguna Beach, CA, 1996. Springer-Verlag, Berlin.

[20] P. Ferragina and R. Grossi. An experimental study of SB-trees.

[21] P. Ferragina and 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.

[22] R. Giegerich and S. Kurtz. From Ukkonen to McCreigth and Weiner: A unifying view of linear-time suffix tree constructions. *Algorithmica*, 19(3):331–353, 1997.

[23] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Proc. of the 3rd Workshop on Algorithm Engineering (WAE 99)*, pages 30–42, 1999.

[24] R. Giegerich, S. Kurtz, and J. Stoye. Efficient implementation of lazy suffix trees. In *Submitted to: Software: Practice and Experience*, 2002.

[25] R. Grossi and G. Italiano. Suffix trees and their applications in string algorithms. In *Proc. of the 1st South American Workshop on String Processing*, pages 57–76, 1993.

[26] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. In *ACM Symposium on Theory of Computing*, pages 397–406, 2000.

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

[28] E. Hunt, R. W. Irving, and M. Atkinson. Persistent suffix trees and suffix binary search trees as DNA sequence indexes. Technical Report TR-2000-63, Univ. of Glasgow, Dept. of Computing Science, 2000.

[29] E. Hunt, R. W. Irving, and M. P. Atkinson. A database index to large biological sequences. In *Proc. of the 27th International Conference on Very Large Databases (VLDB 2001)*, pages 139–148, 2001.

[30] C. S. Iliopoulos, D. Moore, and W. F. Smyth. A characterization of the squares in a Fibonacci string. *Theoretical Computer Science*, 172(1–2):281–291, 1997.

[31] R. W. Irving and L. Love. The suffix binary search tree and suffix AVL tree. Technical Report TR-2000-54, Univ. of Glasgow, Dept. of Computing Science, 2000.

[32] R. W. Irving and L. Love. Suffix array and suffix binary search trees. Technical Report TR-2001-82, Univ. of Glasgow, Dept. of Computing Science, 2001.

[33] J. Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In *Proc. of the 6th Annual Symposium on Combinatorial Pattern Matching CPM*, Lecture notes in Computer Science 937, pages 191–204. Springer Verlag, 1995.

[34] B. W. Kernighan and D. M. Ritchie. *Programmieren in C*. Carl Hanser Verlag, München, 2. edition, 1990.

[35] D. E. Knuth, J. H. Morris, and V. R. Pratt. A fast string-searching algorithm. *SIAM Journal on Computing*, 6(2):323–350, 1977.

[36] S. R. Kosaraju and A. L. Delcher. Large-scale assembly of DNA strings and space-efficient construction of suffix trees. In *Proceedings of the 27th annual ACM symposium on Theory of computing*, pages 169–177, Las Vegas, Nevada, United States, 1996. ACM Press.

[37] S. Kurtz. Reducing the space requirements of suffix trees. *Software: Practice and Experience*, 29(13):1149–1171, 1999.

[38] S. Kurtz, J. M. Choudhuri, E. Ohlebusch, C. Schleiermacher, J. Stoye, and R. Giegerich. REPuter: the manifold applications of repeat analysis on a genomic scale. *Nucleic Acids Research*, 29(22):4633–4642, 2001.

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

[40] T. Lockhart. *The PostgreSQL Programmer's Guide*. The PostgreSQL Global Development Group, 2001.

[41] K. Loney and M. Theriault. *Oracle8i DBA-Handbuch*. Carl Hanser Verlag, München, 2001.

[42] M. Maaß. Linear bidirectional on-line construction of affix trees. In *Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching (CPM 2000)*, volume 1848 of *LNCS*, pages 320–334. IEEE Computer Society Press, June 2000.

[43] U. Manber and E. W. Myers. Suffix arrays: A new method for on-line string searches. *SIAM Journal on Computing*, 22(5):935–948, 1993.

[44] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23:262–272, 1976.

[45] H. W. Mewes and K. Heumann. Genome analysis: Pattern search in biological macromolecules. In *Proc. of the 6th Annual Symposium on Combinatorial Pattern Matching (CPM 95)*, pages 261–285, 1995.

[46] T. Printezis, M. P. Atkinson, Daynès, S. Spence, and P. Bailey. The design of a new persistent object store for PJama. In *Proceedings of the Second International Workshop on Persistence and Java (PJW2)*, Half Moon Bay, CA, USA, 1997.

[47] M. Scholz. Experimentelle Untersuchungen von String B-Trees bezüglich ihrer Anwendbarkeit in Genom-Datenbanken und im Information Retrieval. Master's thesis, Freie Universität Berlin, Institut für Informatik, 2002.

[48] S.S. Skiena. Who is interested in algorithms and why? Lessons from the stony brook algorithms repository. In *Proc. of the 1st Workshop on Algorithm Engineering (WAE 98)*, pages 204–212, 1998.

[49] J. Stoye. Affixbäume. Master's thesis, Technische Fakultät der Universität Bielefeld, AG Praktische Informatik, 1995.

[50] W. Szpankowski. Asymptotic properties of data compression and suffix trees. *IEEETIT: IEEE Transactions on Information Theory*, 39, 1993.

[51] E. Ukkonen. On-line construction of suffix-trees. *Algorithmica*, 14:249–260, 1995.

[52] J. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, 2. edition, 2002.

[53] P. Weiner. Linear pattern matching algorithms. In *Proc. of the 14th IEEE Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig angefertigt habe und alle verwendeten Quellen und Hilfsmittel im Literaturverzeichnis angeführt habe.

Berlin, den 15. Juli 2002

(Klaus-Bernd Schürmann)