Externe Suffix Arrays

Motivation, Funktionsweise und Analyse

Ausarbeitung zum Informatikseminar: Algorithm Engineering
Sommersemester 2009
Leiterin: Prof. Dr. Mutzel
Betreuer: Karsten Klein

Verfasser der Ausarbeitung:
Tobias Brinkjost
t.brinkjost@web.de
TU-Dortmund

April 2009

Inhaltsverzeichnis

1. Einleitung
- 1.1 Einstieg
- 1.2 Die Datenstruktur Suffix Arrays
- 1.3 Motivation zur Betrachtung externer Algorithmen und Datenstrukturen
- 1.4 Messen der Komplexität - Das I/O Modell
- 1.5 Variablendefinitionen und etwas Vorarbeit

2. Algorithmen zum Aufbau Externer Suffix Arrays
- 2.1 Einleitung Algorithmen
- 2.2 Doubling Algorithmus
- 2.3 Pipelining
- 2.4 Doubling mit Verwerfen
- 2.5 α-Tupling
- 2.6 DC3 Algorithmus

3. Analyse
- 3.1 Basis der Analyse
- 3.2 Analyseergebnisse
- 3.3 Fazit der Analyse

4. Fazit

5. Quellenangaben und Verweise

1.1 Einstieg

Diese Ausarbeitung wird sich mit externen Suffix Arrays (External Suffix Array) beschäftigen. Dabei wird ein Fokus auf die Konstruktion einer solchen Datenstruktur gesetzt.
Um die Arbeitsweise der folgenden Algorithmen zum Aufbau eines Suffix Arrays in einer externen Speicherumgebung zu verstehen, wird zunächst die Arbeitsweise interner Suffix Arrays und deren Konstruktion kurz angerissen. Im anschließenden Teil wird das externe Speichermodell oder I/O Modell erläutert, das unseren Messungen der Komplexität (Laufzeit, Speicherverbrauch) zugrunde liegt. Der Hauptteil beschäftigt sich mit der Konstruktion eines Suffix Array im externen Speicher und erläutert und vergleicht die Algorithmen Doubling, Doubling mit Verwerfen, α-Tupling und DC3. Einige Techniken, wie das Pipelining, werden einführend anhand des Doubling Algorithmus behandelt.

Abschließend werden die einzelnen Algorithmen in einer experimentellen Umgebung gegenübergestellt. Dabei werden verschiedene Eingaben verwendet (Source Code, englische Texte, etc.), um eine möglichst allgemeine Aussage über die Effizienz der Agorithmen geben zu können.

Das Fazit resumiert die gewonnenen Ergebnisse und gibt einen kleinen Ausblick auf die derzeitige Entwicklung beziehungsweise auch Zukunft.

Diese Ausarbeitung bezieht sich hauptsächlich auf die Werke “On Constructing Suffix Arrays in External Memory” von Andreas Crauser und Paolo Ferragina [2], sowie auf “Better External Memory Suffix Array Construction” von Roman Dementiev, Juha Kärkkäinen, Jens Mehnert und Peter Sanders [3]. Die folgende Einführung in die Datenstruktur Suffix Array bezieht sich auf die Vorlesungsfolien “Algorithmen und Datenstrukturen” von Professorin Petra Mutzel von der TU-Dortmund [5].

1.2 Die Datenstruktur Suffix Array

Suffix Arrays werden für das so genannte String Matching verwendet. Dabei gilt es ein gegebenes Suchwort (Sample) in einem Text T zu finden. Es gibt viele verschiedene Herangehensweisen an diese Problemstellung und die Suffix Arrays stechen mit ihrer Art der Repräsentation der Daten von allen anderen heraus.
Mit anderen sei hier beispielsweise der Boyer-Moore- oder der Knuth-Morris-Pratt-Algorithmus zu nennen. Diese Algorithmen verwenden ein Preprocessing, welches auf dem Sample operiert und so die Suche im Originaltext beschleunigt. So wird für jede einzelne Suchanfrage (bei verschiedenen Samples) eine Zeit von $O(m+N)$ (bei Boyer-Moore) im Worst Case, bei Textlänge N und Samplelänge m, benötigt.
Die Suffix Arrays verwenden zwar ebenfalls ein Preprocessing, allerdings führen sie dieses nicht auf dem Sample durch, sondern speichern die Indizes der jeweiligen Suffixposition des Originaltextes in lexikografischer Ordnung. Dies ermöglicht die mehrfache Suche nach unterschiedlichen Samples im selben Originaltext mit nur einem Preprocessing.
Abbildung 1.1.1 verdeutlicht ein Suffix Array bei Eingabe des Textes “SuffixArray”.
Diese Art der Eingabespeicherung ermöglicht einen sehr effizienten Speicherplatzbedarf. Insbesondere spielt hier das Eingabealphabet bei der benötigten Größe keine Rolle.
Der Aufbau eines Suffix Array im internen Speicher kann in Zeit $\Theta(N)$ erfolgen, wohingegen beispielsweise der Boyer-Moore Algorithmus mit einer Preprocessing-Zeit von $O(m + |\Sigma|)$ auskommt.

Die Eingabe mit passenden Indizes der Suffixe

i	1	2	3	4	5	6	7	8	9	10	11
T	S	u	f	f	i	x	A	r	r	a	y

Das resultierende Suffix Array mit den Indizes in lexikographischer Ordnung

SA	7	10	3	4	5	9	8	1	2	6	11

Abbildung 1.1.1 - Suffix Array des Textes “SuffixArray”

Diese Zeit ist zwar geringer im Vergleich zu den Suffix Arrays, allerdings kann der Aufwand der Proprocessingsphase der Suffix Arrays über mehrere Suchen amortisiert werden. Entscheidend hierbei ist, dass das Suffix Array bei einer erneuten Suche über den selben Text kein weiteres Preprocessing benötigt, wohingegen bei dem Boyer-Moore oder auch Knuth-Morris-Pratt Algorithmus für jede Suche ein Preprocessing erfolgen muss.

Die Suche im Suffix Array funktioniert im Prinzip wie die binäre Suche. Allerdings reicht diese Technik noch nicht aus, um die erreichte Worst Case Laufzeit von $O(m + logN)$ zu gewährleisten. Daher werden hier verschiedene Zeiger und zusätzliche Informationen verwendet (LCP-Werte, Longest Common Prefix, R- und L-Zeiger, die zudem die Tiefe vom linken, sowie vom rechten Suchbereichsrand markieren), um den Suchbereich schneller eingrenzen zu können. Dabei werden einerseits die LCP-Werte für eine horizontale Eingrenzung des Suchbereiches verwendet, indem bekannte Präfixe bei der Suche ausgelassen werden, andererseits werden die L-LCP- und R-LCP-Werte für das Überspringen bekannter Präfixe oder Zeichen in die Vertikale (Tiefe) verwendet.

Ein wesentlicher Bestandteil der Konstruktion eines Suffix Array im externen Speichermodell ist das Sortieren von Strings. Da, wie später unter Punkt 1.4 erläutert wird, bei der Messung der Komplexität der Austausch von Datenblöcken zwischen internem und externem Speicher ausschlaggebend ist, hängt diese Komplexität von dem Verhältnis zwischen Stringlänge und Blockgröße entscheidend ab. In diesem Zusammenhang ist das Kriterium der Zerteilbarkeit oder Unzerteilbarkeit von Strings ein weiterer wesentlicher Faktor.

Dazu seien folgende Variablen definiert:

- K_1 = *Kurze Strings, Stringlänge* $< B$
- K_2 = *Lange Strings, Stringlänge* $\geq B$
- N_1 = *Anzahl Zeichen in* K_1
- N_2 = *Anzahl Zeichen in* K_2

mit $N_1 + N_2 = N$ und $K_1 + K_2 = K$.

Es seien folgende Modelle definiert:

- Modell A: Einzelne Strings unzerteilbar
- Modell B: Zerteilbarkeit der Strings nur im internen Speicher
- Modell C: Zerteilbarkeit der Strings im externen und internen Speicher

Es lässt sich zeigen [2], dass unter Modell A eine Laufzeit von $\Theta(K_2 \log_{M/B} K_2 + N_2/B)$ zur Sortierung der langen Strings benötigt wird, wohingegen unter Modell B eine Laufzeit von $\Theta(K_2 \log_M K_2 + N_2/B)$ ausreichend ist. Zu beachten ist, dass der Logarithmus nun eine größere Basis hat.

1.3 Motivation zur Betrachtung externer Algorithmen und Datenstrukturen

Trotz der großen zur Verfügung stehenden Kapazitäten an internem Speicher in modernen Rechnern, sind diese oft nicht ausreichend für manche Einsatzgebiete. Dazu sei zu nennen, dass allein im Jahr 2002 über 4,999,230 Terabyte [4] erzeugt und auf magnetischen Datenträgern gespeichert wurden. Das entspricht in etwa der Summe der Wörter, die je von menschlichen Wesen gesprochen wurde.
Durch den Trend der Digitalisierung erreichen digitale Bibliotheken, textbasierte oder relationale Datenbanken oft Größen, die nicht in dem internen Speicher von Rechnern verarbeitet werden können.
Ein weiterer allgemeiner Grund für die Betrachtung von externen Algorithmen und Datenstrukturen ist, dass die Diskrepanz zwischen der Geschwindigkeit von internem und externem Speicher stetig steigt. Daher dominiert bei der Anwendung von internen Algorithmen, in einer externen Speicherumgebung, oftmals der Datenaustausch zwischen diesen Speichern die Laufzeit. Denn diese versuchen nicht in einem gelesenen oder geschriebenen Block aus mehreren Daten möglichst viele nützliche Informationen zu integrieren, sondern fordern für jedes Datum einen einzelnen Block an.
Insgesamt sind die Suffix Arrays eine platzsparende Datenstruktur, allerdings sind die Anforderungen von beispielsweise der Bioinformatik höher, als die Möglichkeiten diese im internen Speicher effizient zu behandeln.
Neben den Compressed Suffix Arrays, die versuchen die Daten so stark zu komprimieren, so dass eine Bearbeitung innerhalb des internen Speichers effizient möglich ist, besteht ein alternativer Ansatz in der effizienten Auslagerung der Daten in den externen Speicher (External Suffix Arrays).

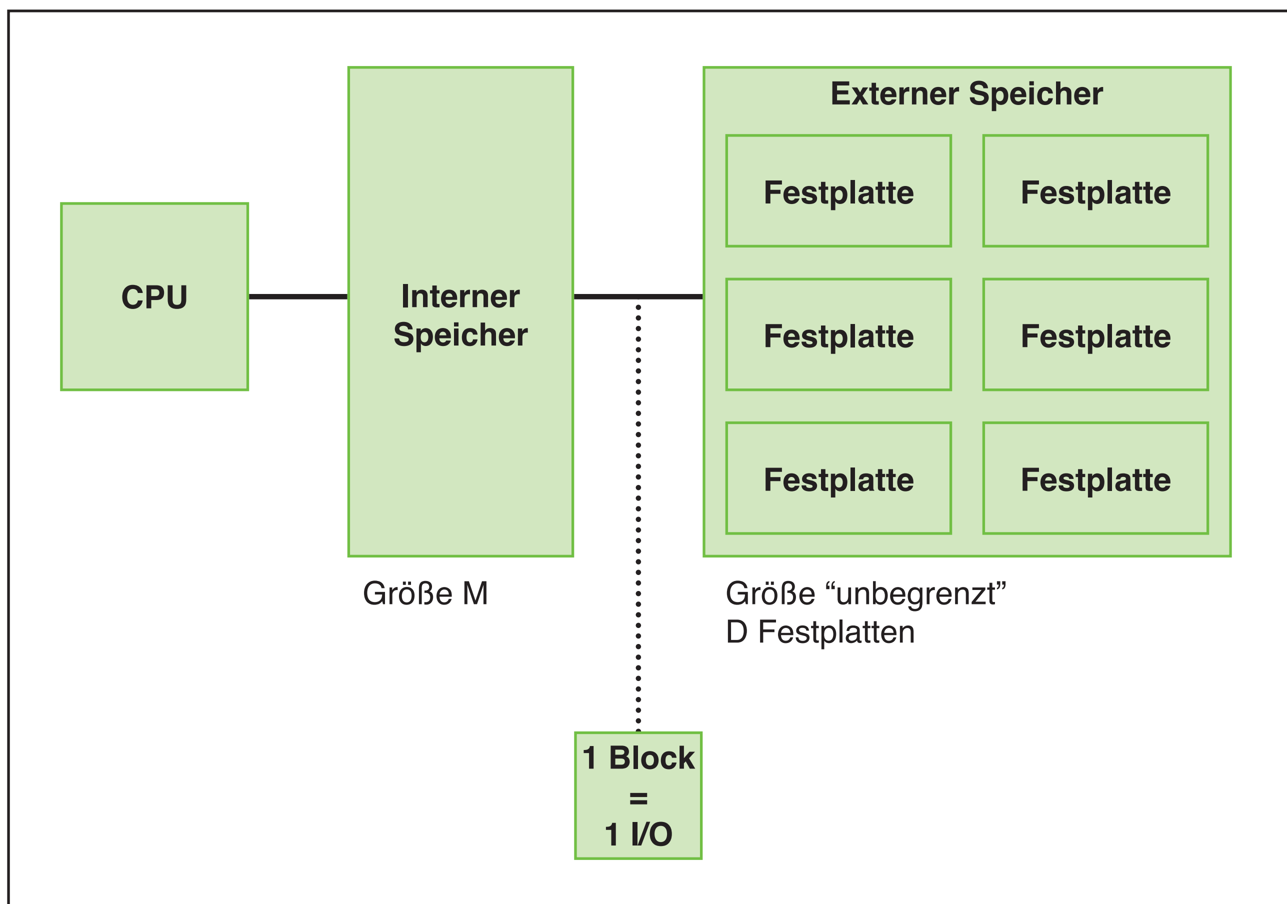


Abbildung 1.4.1 - Grafische Darstellung des I/O Modells

1.4 Messen der Komplexität - Das I/O Modell

Für die Bewertung einzelner Algorithmen in Umgebungen, in denen die Daten extern vorliegen und nur ein geringer interner Speicher zu Verfügung steht, benötigt man ein weitaus umfangreicheres Computermodell, um einzelne Algorithmen miteinander verlgeichen zu können. Als Standardmodell hat sich das I/O-Model von Shriver und Vitter (1994) [6] etabliert, welches nun vorgestellt wird.

Das I/O Modell besteht aus 3 Basisbausteinen (vergleiche Abbildung 1.4.1):

- CPU
- Interner Speicher (schnell), Größe M
- Externer Speicher (langsam), Größe “unbegrenzt”

Unbegrenzt bedeutet hier theoretisch unbegrenzt. In der Praxis gelten natürlich die Grenzen der Technik.

Der Datenaustausch zwischen CPU und internem Speicher ist “kostenfrei”. Zwischen internem und externem Speicher werden Blöcke der Größe B ausgetauscht, die für die Analyse gezählt werden.

Sämtliche Rechenoperationen müssen im Hauptspeicher stattfinden. Der externe Speicher dient in diesem Modell als reiner Speicher, in dem Daten abgelegt und auf diese wieder zugegriffen werden können.

Zur Berücksichtigung der Features moderner Festplatte, beispielsweise Caching oder Prefetching) wird die Konstante c, $c < 1$, eingeführt, die beim Lesen des gesamten Speichers wie folgt in die Formel einfließt: *Anzahl Blöcke* $= c \cdot M / B$.

In einigen Algorithmen taucht ebenfalls eine Variable D, $D =$ *Anzahl Festplatten*, auf, die einen Mehrfestplattenbetrieb berücksichtigt. Dadurch können D Blöcke gleichzeitig gelesen beziehungsweise geschrieben werden. *Anzahl Blöcke* $= c \cdot M / (DB)$.

Bei den abschließenden Analysen und Gegenüberstellungen werden häufig die folgenden drei Größen verwendet und anhand dieser die Algorithmen verglichen:

- Verwendeter Speicher zum Aufbau
- CPU Zeit
- Anzahl der gelesenen Blöcke = Anzahl I/Os

1.5 Variablendefinitionen und etwas Vorarbeit

Wir nehmen an, dass alle extern zur Verfügung stehenden Strings in einem Gesamtstring $T[1,N]$ enthalten sind, wobei N der Gesamtgröße der Strings entspricht.
Weiter gilt, dass die Blockgröße B kleiner als die Größe des internen Speichers M, also $B < M$, ist.
Die Funktion $sort(x)$ gibt die Anzahl I/Os an, die zum Sortieren von x Zeichen aus $T[1,N]$ benötigt werden mit $sort(x) \approx 2x/DB \lceil \log_{M/B} x/M \rceil$.
Die Funktion $scan(x)$ gibt die Anzahl I/O’s an, die zum sequentiellen Lesen oder Schreiben von x Worten benötigt werden mit $scan(x) = \lceil x/DB \rceil$.

2.1 Einleitung Algorithmen

Im folgenden Abschnitt werden einzelne Algorithmen zum Aufbau von Suffix Arrays im externen Speichermodell vorgestellt. Insbesondere wird hier ein Augenmerk auf den Doubling Algorithmus gelegt, der die Grundlage vieler Variationen ist und an dem viele allgemeine Techniken einfach verdeutlicht werden können. Zum Schluss wird der DC3 Algorithmus genauer betrachtet, der, wie sich herausstellen wird, ein I/O optimaler Algorithmus ist.

2.2 Doubling Algorithmus

Die Basisidee des Doubling Algorithmus ist das Ersetzen der Zeichen $T[i]$ der Eingabe durch lexikografische Namen, die die lexikografische Ordnung des Strings der Länge 2^k, $T[i, i + 2^k]$, in der k-ten Iteration repräsentieren.
Im Gegensatz zur ursprünglichen Version aus [2] wird hier eine Sequenz aus Paaren (c,i) manipuliert, in der jeder Name c mit seiner Position i in der Eingabe kombiniert wird.
Zur Erzeugung der Namen für die nächste Iteration $k + 1$ werden die Namen für $T[i, i + 2^k)$ und $T[i, i + 2^{k+1})$ mit der Position i in einer Sequenz S gespeichert und sortiert.
Die neuen Namen können durch Scannen der Sequenz und Vergleichen adjazenter Tupel vergeben werden. Die Sequenz wiederum kann durch die Verwendung von fortlaufenden Elementen aus P, sofern wir P mittels dem Paar $(i \bmod 2^k, i \operatorname{div} 2^k)$ sortieren, erzeugt werden. Die Zahl der Iterationen des Doublingalgorithmus hängt von dem Logarithmus des LPC (longest common prefix) ab.

> Insgesamt benötigt der Doublingalgorithmus $O(sort(N)\lceil \log maxlcp \rceil)$ I/Os zur Konstruktion eines Suffix Arrays bei Eingabelänge N.

Theorem 1

Die Abbildung 2.2.1 veranschaulicht die Arbeitsweise des Doubling Algorithmus mit und ohne Verwerfen. In beiden Fällen werden für die Beispieleingabe insgesamt 4 Iterationen benötigt.
In der ersten Spalte stehen die Indizes des Suffix Arrays und in der zweiten die Suffixpositionen in bis dahin sortierter Reihenfolge. In der letzten Spalte stehen zum Verständnis die entsprechenden Suffixe.
In der ersten Iteration werden alle Suffixe nach ihrem ersten Zeichen verglichen und entsprechend sortiert. Die zweite Iteration berücksichtigt die ersten beiden Zeichen. Also vergleicht die k-te Iteration insgesamt Präfixe Länge $\leq 2^k$ der einzelnen Suffixe.
Man sieht schnell, dass bereits in der ersten Iteration die Positionen 0, 10 und 11 im Suffix Array eindeutig sortiert sind. Der bisher vorgestellte Doubling Algorithmus berücksichtigt dieses allerdings nicht, sondern vergleicht auch diese Positionen in jeder weiteren Iteration erneut. Der im Abschnitt 2.4 vorgestellte Doubling Algorithmus mit Verwerfen erkennt diese eindeutige Sortierung dieser Positionen und berücksichtigt in den folgenden Iterationen nur noch die schwarzen Suffixindizes und “verwirft” die grünen.

Eingabe S: ATAATACGATAATAA$

Iteration 1			Iteration 2			Iteration 3			Iteration 4		
	SA_1	$S[SA_1[i]]$		SA_2	$S[SA_2[i],SA_2[i]+1]$		SA_4	$S[SA_4[i],SA_4[i]+3]$		SA_8	$S[SA_8[i],SA_8[i]+7]$
0	15	$	0	15	$	0	15	$	0	15	$
1	0	A	1	14	A$	1	14	A$	1	14	A$
2	2	A	2	2	AA	2	13	AA$	2	13	AA$
3	3	A	3	10	AA	3	2	AATA	3	10	AATAA$
4	5	A	4	13	AA	4	10	AATA	4	2	AATACGAT
5	8	A	5	5	AC	5	5	ACGA	5	5	ACGATAAT
6	10	A	6	0	AT	6	0	ATAA	6	11	ATAA$
7	11	A	7	3	AT	7	8	ATAA	7	8	ATAATAA$
8	13	A	8	8	AT	8	11	ATAA	8	0	ATAATACG
9	14	A	9	11	AT	9	3	ATAC	9	3	ATACGATA
10	6	C	10	6	CG	10	6	CGAT	10	6	CGATAATA
11	7	G	11	7	GA	11	7	GATA	11	7	GATAATAA
12	1	T	12	1	TA	12	12	TAA$	12	12	TAA$
13	4	T	13	4	TA	13	1	TAAT	13	9	TAATAA$
14	9	T	14	9	TA	14	9	TAAT	14	1	TAATACGA
15	12	T	15	12	TA	15	4	TACG	15	4	TACGATAA

Abbildung 2.2.1 - Beispiel Doubling Algorithmus mit und ohne Verwerfen [7]

```
        Function doubling(T)
(0)     S := [((T[i], T[i + 1]), i) : i ∈ [0, N)]
        for k := 1 to ⌈logN⌉ do
(1)             sort S
(2)             P := name(S)
                invariant ∀(c,i) ∈ P :
                        c is a lexicographic name for T[i, i + 2^k)
                if the names in P are unique then
(3)                     return [i : (c,i) ∈ P]
(4)             sort P by (i mod 2^k, i div 2^k))
(5)             s := <((c,c'), i) : j ∈ [0, N),  (c,i) = P[j] , (c', i + 2^k) = P[j + 1]>

        Function name(S : Sequence of Pair)
                q := r:= 0; (l, l') := ($,$)
                result := <>
                foreach ((c,c'), i) ∈ S do
                        q++
                        if (c,c') ≠ (l, l') then
                                r := q; (l, l') := (c,c')
                        append (r,i) to result
                return result
```

Abbildung 2.2.2 - Pseudocode Doubling Algorithmus

2.3 Pipelining

Die Technik des Pipelining ist schon lange bekannt und ist bei der Architektur heutiger Mikroprozessoren “state of the art”. Da aber auch hier eine Folge von Befehlsausführungen abgearbeitet werden ist die Frage, ob das Pipelining im Bereich des Algorithm Engineering effizient eingesetzt werden kann. Diese Frage lässt sich mit einem klaren “Ja” beantworten. Das Pipelining ermöglicht bei vielen Externspeicheralgorithmen eine Verbesserung der I/O Komplexität um einen Fakor 2, der merklich nachweisbar ist.

Um das Pipelining genauer zu verstehen und um diese Technik bei den Algorithmen anwenden zu können, werden diese Agorithmen als Datenflussgraph interpretiert. Ein Datenflussgraph ist ein direkter, azyklischer Graph $G = (V = F \cup S \cup R, E)$.

Datenknoten $f \in F$:	bekommt Daten und benötigt einen Buffer $b(f) = \Omega(BD)$
Steamingknoten $s \in S$:	lesen von $\{0,1\}^*$ Sequenzen und geben $\{0,1\}^*$ Sequenzen aus verwenden internen Buffer $b(s)$
Sortierungsknoten $r \in R$:	lesen einer Sequenz S und sortierte Ausgabe von S verwenden internen Buffer $b(r) = \theta(M)$
Kanten $e \in E$:	werden mit der Zahl w(e) an Worten beschriftet, die zwischen den adjazenten Knoten ausgetauscht werden

> Die Anweisungen eines Datenflussgraphen $G = (V = F \cup S \cup R, E)$ mit Kantenflüssen $w : E \rightarrow \mathbf{R}_+$ und Bufferanforderungen $b : V \rightarrow \mathbf{R}_+$ können mit folgender I/O Komplexität ausgeführt werden :
>
> $$\sum_{e \in E \cap (F \times V \cup V \times F)} scan(w(e)) \quad + \sum_{e \in E \cap (V \times R)} sort(w(e))$$

Theorem 2 - Pipeliningtheorem

Der Beweis ist in [3] aufgeführt.

Bei dem vorgestellten Doubling Algorithmus führt die Einbeziehung des Pipelining zu folgenden Änderungen:

- Die sortierten Tupel müssen nicht erst gespeichert werden, sondern können direkt in die Namensvergabefunktion (*name(S)*) geleitet werden. (Abb. 2.2.2 Zeile 2)
- Am Ende einer jeden Iteration können die sortierten Tupel direkt in die nächste Iteration geleitet werden. (Abb. 2.2.2 Zeile 5)

Der folgende Datenflussgraph in Abbildung 2.3.1 zeigt diese Veränderungen am Doubling Algorithmus. Die Nummern in den einzelnen Knoten verweisen auf Codezeilen im Pseudocode aus Abbildung 2.2.2.

> Der Doubling Algorithmus mit Pipelining kann in einer I/O Komplexität von $O(sort(5N) \lceil \log maxlcp \rceil + O(scan(N))$ implementiert werden.

Theorem 3 - I/O Komplexität Doubling Algorithmus mit Pipelining

Beweis:
Der Graph in Abbildung 2.3.1 zeigt, dass jede Iteration in $sort(2N) + sort(3N) \leq sort(5N)$ I/Os benötigt. Nach $\lceil log\ maxlcp \rceil$ Iterationen terminiert der Algorithmus. Der $O(sort(N))$ Term wird berechnet für die I/Os in Zeile 0 und für das Erstellen des finalen Resultates.
Zu beachten ist, dass das Benennen kostenfrei erfolgt, aber das Resultat ergibt sich erst, wenn die Namensvergabe beendet ist.

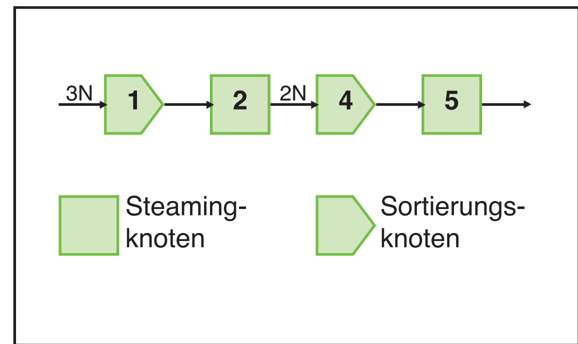


Abbildung 2.3.1 - Datenflussdiagramm Doubling Algorithmus

Zu diesem Zeitpunkt ist die erste Phase des Sortierens in Zeile 4 ebenfalls beendet und hat ebenfalls I/Os verursacht. Die passende Anordnung der Paare *(c,i)* aus *P* ist nun zerstört. Jetzt kann der Sortierungsprozess beendet, die falsche Anordnung rückgängig gemacht und die korrekte Ausgabe berechnet werden. ■

2.4 Doubling mit Verwerfen

Die Grundidee des Verwerfens ist, dass in jeder Iteration im Originalalgorithmus [2] die komplette Eingabe bearbeitet wird. Dabei wird außer Acht gelassen, dass sich die eindeutigen Paare schon an ihrer richtigen Position befinden und somit nicht weiter sortiert oder bearbeitet werden müssen. Daher werden diese aus den restlich zu sortierenden Paaren verworfen.
Sei c_i^k der lexikographische Name des Suffix *T[i, i + 2]*, also der mit *i* kombinierte Wert in Iteration *k*. Da c_i^k die Nummer der strikt kleineren Substrings der Länge 2^k ist, stellt dieses eine nicht-absteigende Funktion von *k* dar. Genauer ist $c_i^{k+1} - c_i^k$ die Anzahl an Positionen *j* so, dass $c_i^k = c_j^k$ aber $c_{j+2^k}^k < c_{i+2^k}^k$ ist. Das offenbart und motiviert die alternative Betrachtung der Namensvergabe und resultiert in der Funktion *name2(S)* in Abbildung 2.4.2.

Das Verwerfen versucht aus diesem Zusammenhang Vorteile zu ziehen.
Genauer wird das Paar *(c,i)* von weiteren Iteration entfernt, sofern *c* eindeutig ist. Die neue Funktion arbeitet korrekt, auch wenn wir alle Paare *((c,c'), i)* von *S* entfernen. Allerdings ist es nicht möglich *((c,c'), i)* zu entfernen, wenn *c'* eindeutig, aber *c* nicht eindeutig ist. In diesem Fall ist es möglich *(c,i)* teilweise (partiell) zu verwerfen, sofern *c* eindeutig ist. Wir können $(c,i) = (c_i^k,i)$ vollständig verwerfen, wenn $c_{j-2^k}^k$ oder $c_{i-2^{k+1}}^k$ eindeutig sind. Denn in diesem Fall ist in jeder Iteration $h > k$ die erste Komponente des Tupels $((c_{i-2^h}^h, c_i^h), i - 2^h)$ eindeutig.
Der Doublingalgorithmus mit Verwerfen wird in Abbildung 2.4.2 vorgestellt.

Die I/O Komplexität des Doublingalgorithmus mit Verwerfen beträgt: *sort(5N log dps) + O(sort(N))* I/Os.

Theorem 4 - I/O Komplexität Doubling Algorithmus mit Verwerfen

Beweis:
Zu zeigen ist, dass der komplette Datenaufwand innerhalb der verschiedenen Schritte des Algorithmus über die komplete Ausführungszeit in dem Datenflussgraphen (Abbildung 2.4.1) abzulesen ist.
Der nicht triviale Punkt ist, dass einerseits *n = N log dps* Tupel in jedem Sortierungsschritt bearbeitet werden und die meisten dieser *N* Tupel in *P* geschrieben werden. Es folgt aus der Tatsache, dass ein Suffix *i* in die Sortierungsschritte so lange mit einfließt, bis es einen eindeutigen Rang besitzt, welches exakt innerhab von $\lceil \log(1+dps(i)) \rceil$ Iterationen geschieht. Um das zu zeigen stellen wir fest, dass ein Tupel (c,i) nur dann in Iteration *k* zu *P* hinzugefügt wird, wenn das vorherige Tupel $(c', i - 2^k)$ nicht eindeutig war. Dieses Tupel wird eindeutig in der nächsten Iteration, denn es ist durch $((c',c), i - 2^k)$ in S repräsentiert. Da jedes Tupel nur einmal eindeutig wird, ist die vollständige Anzahl an Tupeln, die in *P* hinzugefügt werden, höchstens *N*.

■

Ergänzungen:
$dps(i) := 1 + \max\{lcp(i,j), lpc(i-1, i), lpc(i, i+1)\}$

$\log dps(i) := \frac{1}{N} \sum_{0 \le i < N} \log(dps(i))$

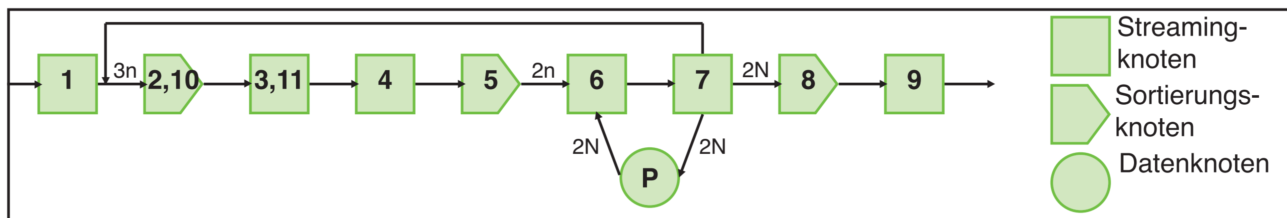


Abbildung 2.4.1 - Datenflussdiagramm Doublingalgorithmus mit Verwerfen

```
        Function doubling+discarding(T)
(1)         S := [((T[i], T[i + 1]), i) : i ∈ [0,N)]
(2)         sort S
(3)         U := name(S)                //unverworfen
            P := <>                     //partiell verworfen
            F := <>                     //vollständig verworfen
            for k := 1 to ⌈log N⌉ do
(4)             mark unique names in U
(5)             sort U by (i mod 2^k, i div 2^k)
(6)             merge P into U; P := <>
                S := <>
                count := <>
(7)             for each (c,i) ∈ U do
                    if c is unique then
                        if count < 2 then
                            append (c,i) to F
                        else append (c,i) to P
                    else
                        let (c',i') be the next pair in U
                        append ((c,c'), i) to S
                        count++
                if S = ø then
(8)                 sort F by first component
(9)                 return [i : (c,i) ∈ F]
(10)            sort S
(11)            U := name2(S)

        Function name2(S : Sequence of Pair)
            q := q' := 0; (l, l') := ($,$)
            result := <>
            for each ((c,c'), i) ∈ S do
                if c ≠ l then q := q':= 0; (l, l') := (c,c')
                elseif c' ≠ l' then q':= q; l':= c'
                append (c + q', i) to result
                q++
            return result
```

Abbildung 2.4.2 - Pseudocode Doublingalgorithmus mit Verwerfen und neue name2(S) Funktion

2.5 α-Tupling

Es liegt auf der Hand den Doublingalgorithmus aus den Kapiteln 2.2 und 2.4 in so fern zu verallgemeinern, dass aus der Basis 2 eine Basis α wird. Dadurch repräsentieren in der Iteration k die lexikographischen Namen die Strings der Länge α^k.
Das Sortieren und die Namensvergabe erfolgen wie in den Algorithmen zuvor. Der Pseudocode dieses Algorithmus ist in Abbildung 2.5.1 dargestellt. Abbildung 2.5.2 gibt eine kurze Übersicht über das Verhältnis zwischen dem α und der resultierenden I/O Komplexität.

```
Function atupling(T)
    S :=[((T[i], T[i+1], ... , T[i + α - 1]), i) : i ∈ [0, N)]
    for k:= 1 to ⌈log_α N⌉
        sort S
        P := name(S)
        invariant ∀(c,i) ∈ P : c is a lexicographic name for T[i, i + α^k]
        if the names in P are unique then return (i, : (c,i) ∈ P)
        sort P by (i mod α^k, i div α^k)
        S := [((c_0, ... , c_q, ... , c_α-1), i) : j ∈ [0, N), (cq, i + q•α^k) = P[j + q], q ∈ [0, α)]
```

Abbildung 2.5.1 - Pseudocode α-Tuplingalgorithmus

Interessant bei dem α-Tuplingalgorithmus ist der Tradeoff aus höheren Kosten pro Iteration und einer geringeren Anzahl an Iterationen. Dieses Verhältnis wird durch den Term

$$\frac{\alpha+3}{\log \alpha} N$$

zum Ausdruck gebracht.

Insgesamt lässt sich der α-Tuplingalgorithmus in folgender I/O Komplexität implementieren:
$sort(\frac{\alpha+3}{\log \alpha} N) \log maxlcp + O(sort(N))$ oder $sort(\frac{\alpha+3}{\log \alpha} N) \log dps + O(sort(N))$

Theorem 5 - I/O Komplexität α-Tuplingalgorithmus

α	2	3	4	5	6	7
(α + 3)/ log α	5,00	3,78	3,50	3,45	3,48	3,56

Abbildung 2.5.2 - Gegenüberstellung α Wert zu I/O Komplexität

2.6 DC3 Algorithmus

Wie eingangs erwähnt ist der DC3 ein I/O optimaler Algorithmus. Die hier vorgestellte Variante verwendet das zuvor angesprochene Pipelining.

Der Ablauf des Algorithmus lässt sich in 3 Schritten aufteilen:

- **Schritt 1**
  Konstruiere ein Suffix Array *A* aus den Suffixen beginnend an den Positionen *i mod 3 ≠ 0*. Dies geschrieht durch Problemreduktion auf die Suffix Array Konstruktion eines Strings der Länge *2/3*, welches rekursiv gelöst wird.
- **Schritt 2**
  Konstruiere ein Suffix Array *B* aus den restlich verbliebenen Suffixen durch Verwenden der Ergebnisse des ersten Schrittes.
- **Schritt 3**
  Verschmelze die beiden Suffix Arrays *A* und *B*.

Die folgende Implementierung, Abbildung 2.6.1, zeigt den Preudocode einer DC3 Implementierung. An dieser Abbildung werden die drei Schritte deutlich.

	Function DC3(T)	
(1)	$S := [((T[i, i+2]), i : i \in [0,N), i \bmod 3 \neq 0]$	Schritt 1
(2)	sort S by the first component	
(3)	P := name(S)	
	if the names in P are not unique then	
(4)	sort the $(i,r) \in P$ by (i mod 3, i div 3)	
(5)	$SA^{12} := DC3([c : (c,i) \in P])$	
(6)	$P := [(j+1, SA^{12}[j]) : j \in [0, 2N/3]$	
(7)	sort P by the second component	
(8)	$S_0 := \langle(T[i], T[i+1], c', c'', i) : i \bmod 3 = 0, (c', i+1), (c'', i+2) \in P\rangle$	
(9)	$S_1 := \langle(c, T[i], c', i) : i \bmod 3 = 1, (c, i), (c', i+1) \in P\rangle$	
(10)	$S_2 := \langle(c, T[i], T[i+1], c'', i) : i \bmod 3 = 2, (c, i), (c'', i+2) \in P\rangle$	
(11)	sort S_0 by components 1, 3	Schritt 2
(12)	sort S_1 and S_2 by component 1	
(13)	S := merge (S_0, S_1, S_2) comparison function:	Schritt 3
	$(t, t', c', c'', i) \in S_0 \leq (d, u, d', j) \in S_1 \Leftrightarrow (t, c') \leq (u, d')$	
	$(t, t', c', c'', i) \in S_0 \leq (d, u, u', d'', j) \in S_2 \Leftrightarrow (t, t', c'') \leq (u, u', d'')$	
	$(c, t, c', i) \in S_1 \leq (d, u, u', d'', j) \in S_2 \Leftrightarrow c \leq d$	
(14)	return [last component of s: $s \in S$]	

Abbildung 2.6.1 - Pseudocode DC3 Algorithmus

In den Zeilen 1-6 erfolgt die rekursive Teilung und Konstruktion der Arrays, um die lexikografischen Namen zu finden, die die relative Sortierung der Suffixe exakt beschreiben. Danach geschieht in den Zeilen 7-10 die Vorbereitung für die Schritte zwei und drei. Die einzelnen Samplenamen werden verwendet, um für jede einzelne Suffixposition i den globalen Rang zu bestimmen. Hier reichen meist die ersten zwei Zeichen für eine eindeutige Zuweisung.

Ab Zeile 11 folgt der oben genannte Schritt zwei des Algorithmus. Es werden nun die Suffixe T_i mit *i mod 3 = 0* nach dem ersten Zeichen und dem Namen für T_{i+1} in diesem Sample sortiert.
In Zeile 11 wird die Reihenfolge der *mod-2* und *mod-3* Suffixe rekonstruiert. Nun sind beide Suffix Arrays konstruiert und es kann die Verschmelzung, also Schritt drei, in Zeile 13 erfolgen. Dabei unterscheidet die Vergleichsfunktion 3 Fälle:

- **Fall 1**
  Ein *mod-0* Suffix T_i kann mit einem *mod-1* Suffix T_j durch vergleichen der ersten Zeichen und den lexikografischen Namen innerhalb des Samples für T_{i+1} und T_{j+1} verglichen werden.
- **Fall 2**
  Der Vergleich zwischen einem *mod-0* Suffix T_i und einem *mod-2* Suffix T_j muss sich einer anderen Technik bedienen, da T_{j+1} nicht das passende Sample ist. Daher werden T_{i+2} und T_{j+2} verwendet und der Vergleich geschieht wieder auf Basis der ersten Zeichen und der Namen T_{i+2} und T_{j+2}.
- **Fall 3**
  Mod-1 Suffixe und *mod-2* Suffixe werden durch die Verwendung der Namen in deren Sample verglichen.

Das Pipelining in diesem Algorithmus kann nicht so einfach wie beim Doubling Algorithmus angewendet werden, da die Eingabe *T* an zwei Stellen (Zeile 2 - Sample generieren und Zeilen 8-10 - Scannen nach Namen der Sample Suffixe) benötigt wird. Allerdings lässt sich das Pipelining an einigen Stellen umsetzen. Die folgende Abbildung des Flussgraphen verdeutlicht die Pipeliningstruktur des hier vorgestellten DC3. Die einzelnen Knoten sind mit den entsprechenden Zeilennummern des Pseudocodes beschriftet, um einen direkten Vergleich zu ermöglichen.

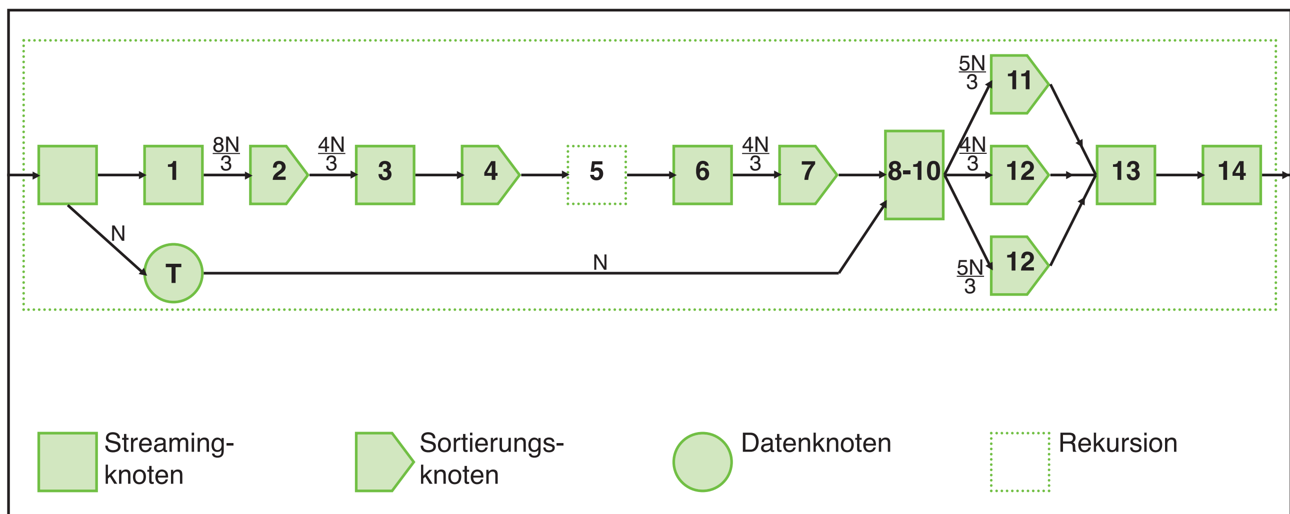


Abbildung 2.6.1 - Datenflussdiagramm DC3 Algorithmus

Der Datenfluss in der Abbildung 2.6.1 erfolgt von links nach rechts. Die Abbildung zeigt deutlich, dass an die Eingabe an zwei Stellen benötigt wird, zu Anfang und im Knoten 8-10. Die Knoten 4,5 und 6 werden nur ausgeführt, wenn die lexikgrafischen Namen nicht eindeutig sind. Die Rekursion in Schritt 5 wird somit solange ausgeführt, bis eindeutige Namen vorliegen. Die Kantenbeschriftungen entsprechenden den zwischen den Knoten ausgetauschten Wörtern.

Anhand des Datenflussgraphen kann nun die Zahl der benötigten I/Os abgelesen werden.

Der DC3 Algorithmus benötigt *sort(30N) + scan(6N) I/O's*.

Theorem 6 - I/O Komplexität DC3 Algorithmus

Beweis:
Sei V(N) die Anzahl der I/Os des externen DC3 Algorithmus. Anhand des Pipeliningtheorems und der Kantenbeschriftungen aus dem Graphen können wir auf folgende Formel schließen:

$$V(N) \leq sort((8/3 + 4/3 + 4/3 + 5/3 + 4/3 + 5/3)N) + scan(2N) + V(2N/3)$$
$$= sort(10N) + scan(2N) + V(2N/3)$$

Die Rekursion hat eine Tiefe von $V(N) \leq 3(sort(10N) + scan(2N)) \leq sort(30N) + scan(6N)$.
Zu beachten ist die Annahme, dass die Eingabe als Datenstrom in die Funktion eingegeben wird. Dennoch erhalten wir die gleiche Komplexität, wenn die Eingabe als Datei vorliegt, jedoch müssen wir dann die Eingabe nur einmal lesen und sparen das Schreiben in den Datenknoten *T*.

■

3.1 Basis der Analyse

Die folgende, hier in Auszügen dargestellte, Analyse ist entnommen aus “Better External Memory Suffix Array Construction” [3].

Als **Basisrechner** wird ein 2-Prozessorsystem mit folgenden Eigenschaften verwendet:
- 2 CPUs, jeweils 2,0 GHz Intel Xeon
- 1 GB Arbeitsspeicher
- 4 80GB Festplatten vom Typ 120GXP der Firma IBM

Die einzelnen **Algorithmen** sind in C++ (g++ 3.2.3 compiler, optimzation level -02--omit-framepointer) implementiert. Hier werden folgende berücksichtigt:
- **Doubling**
- **Doubling mit Verwerfen**
- **Quadrupling mit Verwerfen**
- **DC3**

Die aufgezählten Algorithmen sind als Pipeline-Variante implementiert.

Als Eingabe werden folgende fünf **Instanzen** verwendet:
- **Random2**
  Zwei konkatenierte Kopien eines durch Zufall generierten Strings der Länge *N/2*.
- **Gutenberg**
  Frei verfügbarer englischer Text von http://promo.net/pg/list.htm.
- **Genom**
  Die bekannten Teile des menschlichen Genoms von http://genome.ucsc.edu/downloads.html (Stand Mai 2004).
  Diese Instanz wurde normalisiert verwendet, um Groß- und Kleinschreibung ignorieren zu können.
- **HTML**
  Webseiten von einer Crawl-Suchmaschine, die nur .gov Seiten berücksichtigt hat. Diese Seiten wurden so gefiltert, dass keine Binärdaten oder Bilder im Quelltext vorhanden sind.
- **Source**
  Source Code, vorangig C++ der unter anderem gcc, core-utils, gimp, kde, xfree, emacs, gdb, Linux Kernel und Open Office beinhaltet.

Abbildung 3.1.1 zeigt tabellarisch die Eigenschaften der gewählten Instanzen.

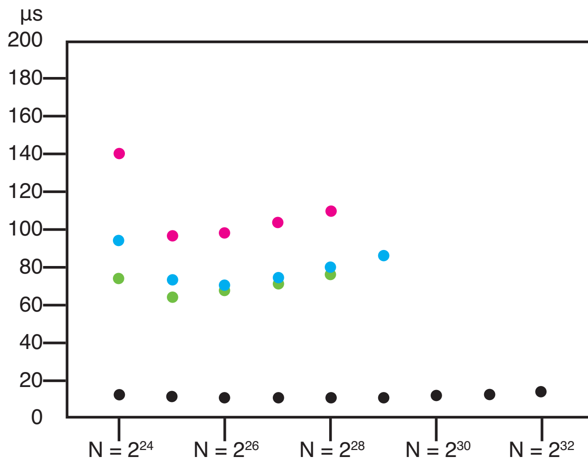
Instanz	N = ITI	IΣI	maxlcp
Random2	2^{32}	128	2^{31}
Gutenberg	3.277.099.756	128	4.819.356
Genom	3.070.128.194	5	21.999.999
HTML	4.214.295.245	128	102.356
Source	547.505.710	128	173.317

Abbildung 3.1.1 - Eigenschaften der Instanzen

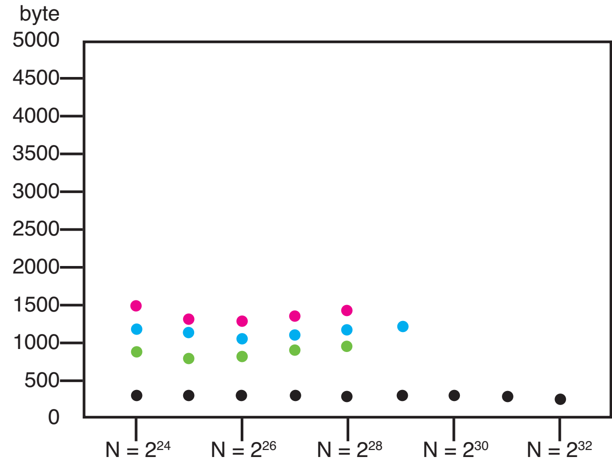
3.2 Analyseergebnisse

- Doubling
- Doubling mit Verwerfen
- Quadrupling mit Verwerfen
- DC3

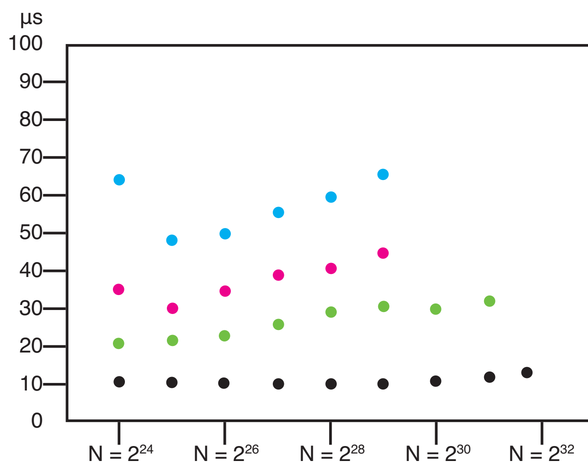
Random2 - Zeit / N



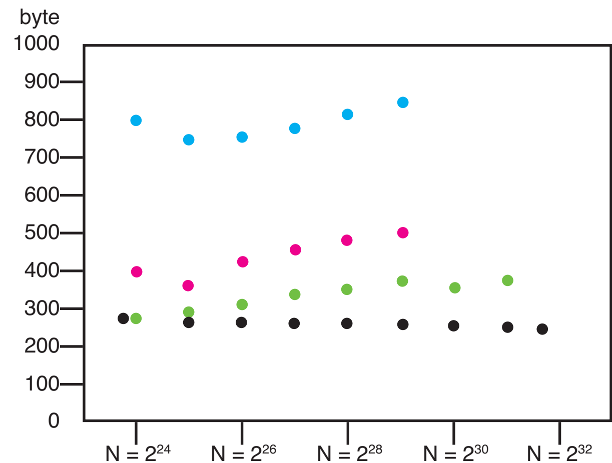
Random2 - I/O byte / N



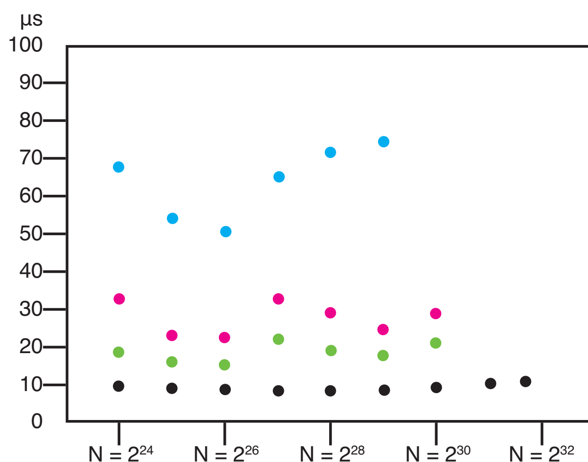
Gutenberg - Zeit / N



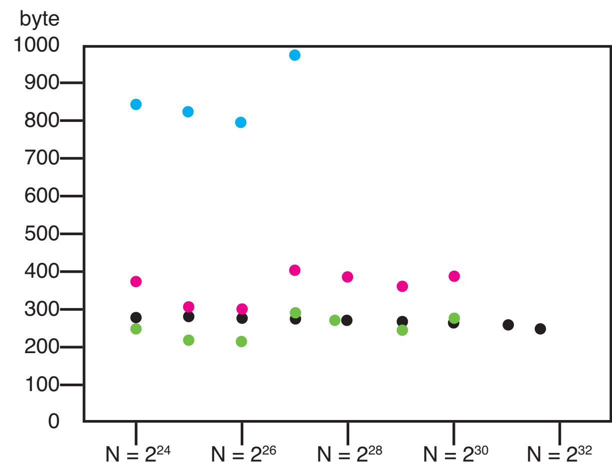
Gutenberg - I/O byte / N



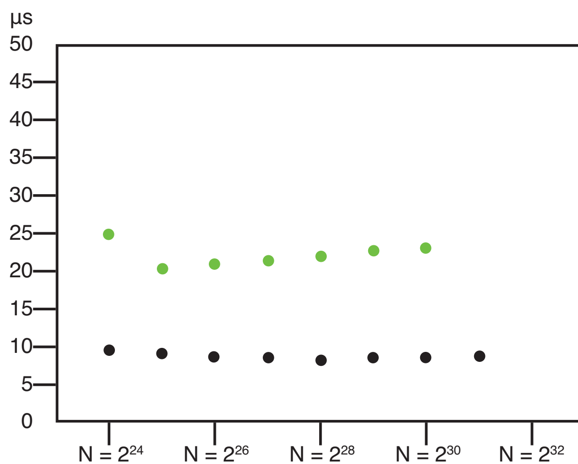
Genom - Zeit / N



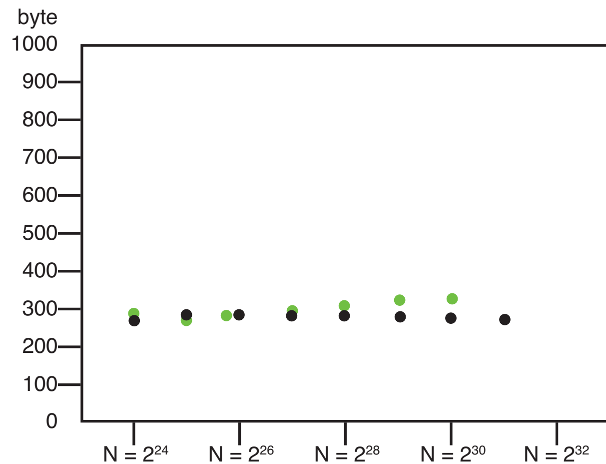
Genom - I/O byte / N



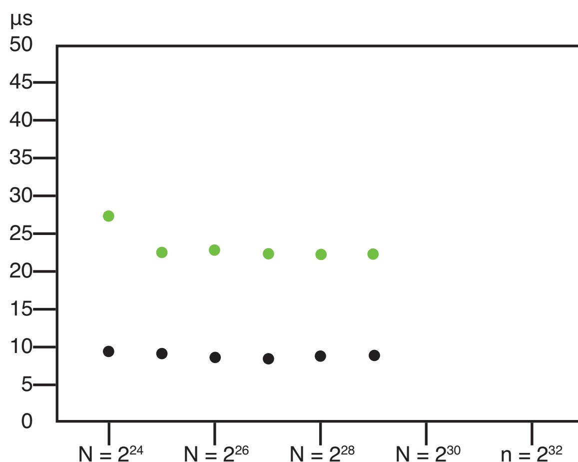
HTML - Zeit / N



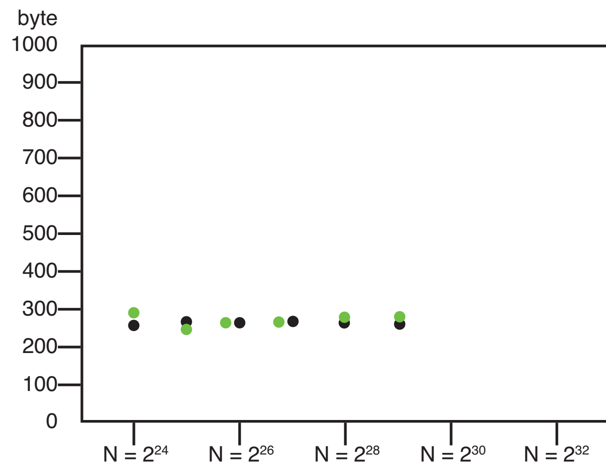
HTML - I/O byte / N



Source - Zeit / N



Source - I/O byte / N



3.3 Fazit der Analyse

Die Diagramme sprechen klar für den DC3 Algorithmus unter den ausgewählten Algorithmen. Die Laufzeit pro Eingabezeichen ist in allen Fällen die geringste und häufig um einen Faktor von mindestens 2 schneller gegenüber dem Zweitbesten.

Ein weiteres Merkmal der Ergebnisse ist, dass die Option des Verwerfens bei den Doubling Algorithmen dazu führt, dass die Laufzeit und die Anzahl der I/Os doch deutlich gesenkt werden kann. Wenn man zwischen den Punkten eine Kurve zeichnen würde, so wäre die Kurve des Doubling Algorithmus mit Verwerfen wie eine nach unten verschobene Kopie des normalen Doubling Algorithmus.

Insgesamt zeigt sich, dass der auch im internen Speicher effizient arbeitende DC3 Algorithmus für die Verwendung in externen Speichermodellen prädestiniert ist.

Bei der Betrachtung der Ergebnisse sollte man aber nicht vergessen, dass es sich hierbei um Algorithmen handelt, die alle das Pipelining verwenden.

Die hier vorgestellten Algorithmen bilden nur einen kleinen Ausschnitt der Möglichkeiten Suffix Arrays in einer Externspeicherumgebung zu berechnen. Dennoch ist mit dem DC3 ein I/O-optimaler Algorithmus in dieser Ausarbeitung enthalten.
Wir haben gesehen, dass die Technik des Pipelining eine nicht zu unterschätzende Möglichkeit bietet, die I/O Komplexität, also die Anzahl der I/Os, eines Algorithmus zu verbessern. In der vorgestellten Analyse wurde dieser Effekt nur durch Abwesenheit des Non-Pipelinging Algorithmus deutlich. Der Doubling Algorithmus ohne Pipelining ist in dem Paper von Dementiev, Kärkäinnen, Mehnert und Sanders [3] zwar enthalten, aber dessen Analysewerte steigen von der ersten zu den folgenden Instanzen so stark, dass eine sinnvolle Darstellung aller Analyseergebnisse der Algorithmen nicht möglich ist.

Insgesamt lässt sich sagen, dass die Datenstruktur Suffix Array noch relativ jung ist. Speziell der Bereich der externen Suffix Array (seit 1999 [2]) wird gerade erst seit 10 Jahren durch eine vergleichsweise kleine Gruppe, weltweit verteilter Wissenschaftler, erforscht. Dennoch sind die Resultate dieser Forschung bemerkenswert.
Aber auch wenn es lineare und I/O-optimale Konstruktionsalgorithmen gibt, werden sicher noch weitere Techniken und Erfolge ausstehen, bis Suffix Array für externe Speichermedien letztendlich einen höheren Stellenwert in der Praxis einnehmen werden.

Auch ingesamt werden die externen Speicheralgorithmen und Datenstrukturen in der Zukunft eine höhere Bedeutung erhalten, denn eine Entspannung der Diskrepanz zwischen der Geschwindigkeit eines Prozessors, den verfügbaren Speichermedien und unter den Speichermedien ist nicht zu erwarten. Im Gegenteil, die Leistung der Prozessoren wird weiterhin um einen höheren Faktor steigen und die Parallelisierung wird ihr Übriges dazu beitragen, diesen Faktor zu erhalten oder sogar zu erhöhen. Für externe Speichermedien werden daher Techniken erforderlich sein, die nicht auf den bisherigen Festplattenprinzipien beruhen, um die nötige Geschwindigkeit zu erreichen, die das Vernachlässigen des I/O Modells zur Folge hätte.

Eine aktuelle Weiterentwicklung der Suffix Array bezieht sich auf die Veränderung der rein statischen Datenstruktur zu einer dynamischen. In diesem Kontext sind die “Dynamic Extended Suffix Arrays” von Mikaël Salson, Martine Léonard, Thierry Lecroq und Laurent Mouchard aus dem Jahre 2009 zu nennen [1], die eine Aktualisierung des Suffix Arrays in einer Zeit von $O(N \log N(1 + \log \sigma / \log \log N))$ durchführen können.

Quellenangaben und Verweise

[1] Dynamic Extended Suffix Arrays, February 5th 2009
Mikaël Salson, Thierry Lecroq, Martine Léonard, Laurent Mouchard
LITIS, University of Rouen
Erschienen im: Journal of Discrete Algorithms

[2] On Constructing Suffix Arrays in External Memory, 1999
Andreas Crauser, Paolo Ferragina
Max-Planck-Institut für Informatik, Saarbrücken, Deutschland
Erschienen im: Proceedings of the 7th Annual European Symposium on Algorithms (ESA-99)

[3] Better External Memory Suffix Array Construction
Roman Dementiev[1], Juha Kärkkäinen[2], Jens Mehnert[3], Peter Sanders[1]
1) Fakultät für Informatik, Universität Karlsruhe, Deutschland
2) Departement of Computer Science, University of Helsinki, Finnland
3) Max-Planck-Insitut für Informatik, Saarbrücken, Deutschland
Erschienen im: ACM Journal of Experimental Algorithmics 12 (2008)

[4] Studie: How Much Information? 2003
UC Berkeley's School of Information Management and Systems, University of California
http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/

[5] Vorlesungsfolien Algorithmen und Datenstrukturen von Professorin Petra Mutzel
Wintersemester 2008/2009, Technische Universität Dortmund
http://ls11-www.cs.uni-dortmund.de/people/gutweng/AD08/

[6] Algorithms for parallel memory I: two level memories 1994
E.A. Shriver, J.S. Vitter
Algorithmica 12(2-3), Seiten 110-147

[7] Large-scale genome sequence processing 2006
Masahiro Kasahara, Shinichi Morishita
Imperial College Press, ISBN 1860946356, 9781860946356