ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Theoretische Informatik

Berechnung der k-Burrows-Wheeler Transformation und Datenstrukturen zur Suche darauf: Ein Vergleich verschiedener Algorithmen

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
David Mödinger
david.moedinger@uni-ulm.de

Gutachter:
Prof. Dr. Enno Ohlebusch

Betreuer:
Simon Gog

2012

Fassung 23. April 2012

Inhaltsverzeichnis

Abstract. Es werden drei Algorithmen zur Konstruktion der k-Burrows-Wheeler Transformation basierend auf der Laufzeit ihrer Implementierung in C++ untersucht und mit einer effizienten Berechnung eines Suffixarrays verglichen, die außer in sehr speziellen Fällen schnellere Ergebnisse liefert.

1 Einleitung

Sind auf einem Text, der sich nur selten ändert, viele Suchanfragen nach Textausschnitten zu erwarten, lohnt es sich häufig im Voraus eine Struktur zu erzeugen, auf der effizienter gesucht werden kann. Die Entwicklung solcher Strukturen führte zur Entwicklung von selbstreferenzierenden Indizes die nur wenig größer als der ursprüngliche Text sind. Ein Beispiel für einen solchen Index ist die Burrows-Wheeler Transformation, die zusätzlich zu einer Permutation des Textes noch einen Index der im Text verwendeten Zeichen benötigt.
Die Burrows-Wheeler Transformation kann zwar in linearer Zeit berechnet werden, wie [KA03] und [KS03] zeigten, um die Ausfürungszeit weiter zu verringern, wurde die k-Burrows-Wheeler Transformation (kurz k-BWT) untersucht. Diese benötigt eine lexikographische Sortierung nur bis zur Tiefe $1 < k \ll n-1$. Um die Erzeugung dieser k-BWT geht es in dieser Arbeit. Neben den hervorragenden Eigenschaften zur Suche bietet die Burrows-Wheeler Transformation auch gute Eigenschaften zur Kompression, um diese für die k-BWT zu nutzen, wäre auch die Rücktransformation der k-BWT nötig, um die soll es in dieser Arbeit jedoch nicht gehen, genaueres ist in [CPP11] zu finden.
In Kapitel 2 werden zunächst die nötigen Konzepte und Strukturen, die BWT und k-BWT, Bitvektoren, Wavelet-Bäume und der Radixsort, und weshalb sie für eine erfolgreiche Rückwärtssuche nötig sind, erläutert werden. Die Verwendung und Erzeugung der Konzepte und Strukturen wird in Kapitel 3 in einzelne Schritte unterteilt, diese Schritte untersucht und verglichen und eine Kombination aus den effizientesten erzeugt. Das Fazit dieser Untersuchung wird dann in Kapitel 4 besprochen.

2 Die verwendeten Konzepte

In diesem Abschnitt sollen, ausgehend von der Problemstellung, die nötigen Begriffe und Komponenten geklärt und beschrieben werden.

2.1 Die Problemstellung

Das zugrunde liegende Problem ist das Problem der exakten Suche:

Definition 2.1 (Exakte Suche im Text) *Exakte Suche ist das Problem in einem Text T, bestehend aus den Zeichen $t_0, t_1, \ldots, t_{n-1}$, einen kürzeren Text P, bestehend aus den Zeichen $p_0, \ldots, p_{m-1}$, zu finden. Hierbei die Länge des Textes T als $n := |T|$ und die Länge des Textes P als $m := |P|$ gegeben. Die Zeichen t_i, p_j entstammen demselben Alphabet Σ mit $|\Sigma|$ unterschiedlichen Symbolen.*

Hier wird davon ausgegangen, dass das letzte Symbol des zugrunde liegenden Textes stets ein Abschlusszeichen, hier \$, ist, das sonst nicht im Text vorkommt. Der zu suchende Text P ist nicht automatisch mit diesem Zeichen abgeschlossen. In der Praxis wird das Nullbyte verwendet.
Zur Lösung dieses Problems gibt es verschiedene Ansätze: Vom naiven Ansatz mit direkten Vergleichen in $O(n*m)$, über den Rabin-Karp-Algorithmus, mit einer durchschnittlichen Laufzeit von $O(n+m)$, hin zum Boyer-Moore-Algorithmus, der im Mittel $O(\frac{n}{m})$ Zeit benötigt. Diese Algorithmen benötigen keine zusätzliche Verarbeitung des Textes T. Der Knuth-Morris-Pratt Algorithmus verwendet eine Verarbeitung des zu suchenden Textes P und erreicht eine Komplexität im Worst-Case von $O(n+m)$. Um die Laufzeit der Suche zu verbessern, wurde eine Verarbeitung des Textes T untersucht: Das Suffixarray.

2.2 Das Suffixarray

Das in [MM90] vorgestellte Suffixarray basiert auf den Konzepten der lexikographischen Ordnung und des Suffixes, wobei die lexikographische Ordnung eine Ordnung auf dem Alphabet Σ, mit $\$ < c \; \forall c \in \Sigma$ und $c \neq \$$, benötigt.

Definition 2.2 (Lexikographische Ordnung) *Ein Text M=$m_0, \ldots, m_{|M|-1}$, heißt lexikographisch kleiner als ein Text N=$n_0, \ldots, n_{|N|-1}$, geschrieben M<N, falls gilt:*

Basisfall $m_0 < n_0$ *oder*
Induktiv $m_k < n_k$ *falls* $m_j = n_j$ *für* $j = 0, \ldots, k-1$

Da jeder Text mit dem Abschlusszeichen $ versehen ist, führt dies immer zu einem Ergebnis. Da ein Text im Vergleich mit einem kürzerer Text zuletzt immer auf eine Überprüfung $<c mit $c \neq$ $ stoßen wird, denn $ kommt nach Voraussetzung nicht im Text vor.

Definition 2.3 (Suffix) *Ein Suffix über einem Text T sind die Zeichen* $t_j, t_{j+1}, \ldots, t_{n-1}$ *aus T ab Position* $j \in \{0, \ldots, n\}$ *bis zum letzten Zeichen aus T.*

Beispiel 2.1 (Lexikographische Ordnung und Suffixe) *Der Text „abracadabra“ hat zum Beispiel die Suffixe „abra“ (abracadabra), „dabra“ (abracadabra) und „abracadabra“. Es gilt: „abra“<„dabra“, da* $a < d$*, sowie „abra“<„abracadabra“, da zwar die ersten vier Zeichen übereinstimmen, der letzte Vergleich mit* $k = 4$ *jedoch* $ $<c$ *ist und $ ist kleiner als jedes von $ verschiedene Zeichen.*

Ausgehend von der Überlegung, dass das Finden aller Vorkommen des Textes P im Text T äquivalent zum Finden aller Suffixe in T ist, die mit P beginnen, wird das Suffixarray betrachtet.

Definition 2.4 (Suffixarray) *Das Suffixarray besteht aus den n lexikographisch geordneten Suffixen des Textes T und kann durch Zeiger auf die Position des Textes, an dem das Suffix beginnt, effizient gespeichert werden.*

Beispiel 2.2 (Suffixarray) *Das Suffixarray für „abracadabra“ hat 12 Einträge, da abracadabra 12 Suffixe besitzt:*

Auf dem Suffixarray kann nun mit binärer Suche nach einem Text effizient gesucht werden. Im Rahmen dieser Arbeit wird mit der divsufsort Funktion der Bibliothek divsufsort[1] das Suffixarray als Zeiger auf den Text berechnet und mit den Ergebnissen der eigenen Konstruktion verglichen. Jedoch benötigt das Suffixarray auf einem System mit 32 Bit Integern $4n$ Bytes Platz, ansonsten $4n \log n$ Bits. Um diesem Problem entgegen zu wirken, wurde eine Transformation verwendet, die ähnliche Sucheigenschaften hat wie das Suffixarray: Die Burrows-Wheeler Transformation.

[1] Siehe `http://code.google.com/p/libdivsufsort/`

Nr.	SA	Suffix
1	11	$
2	10	a$
3	7	abra$
4	0	abracadabra$
5	3	acadabra$
6	5	adabra$
7	8	bra$
8	1	bracadabra$
9	4	cadabra$
10	6	dabra$
11	9	ra$
12	2	racadabra$

Tabelle 1: Beispiel Suffixarray für den Text „abracadabra“, Suffixe sind nur zur Veranschaulichung mit angegeben.

2.3 Die Burrows-Wheeler Transformation

Die Burrows-Wheeler Transformation (BWT) wurde in [BW94] als Möglichkeit zur Komprimierung vorgestellt, bietet jedoch auch die Möglichkeit sie als Suchindex zu verwenden.

Definition 2.5 (Burrows-Wheeler Transformation) *Die Burrows-Wheeler Transformation ist eine Permutation eines Textes. Sie wird durch die Formel*

$$BWT[i] = T_{SA[i]-1 \mod n}$$

aus dem Suffixarray gewonnen.

Einfacher ausgedrückt ist das i-te Zeichen der BWT dasjenige Zeichen, das vor dem i-ten Suffix des Suffixarrays steht. Ist das i-te Suffix der gesamte Text, wird stattdessen $ verwendet. Um die BWT zur effizienten Suche verwenden zu können benötigt man jedoch ein weiteres Konstrukt der Größe $O(|\Sigma| * log(n))$: Einen Index C.

Definition 2.6 (Index C) *Der Index C enthält für jedes Zeichen $\sigma \in \Sigma$ die Position des ersten Vorkommens im Suffixarray als erstes Zeichen im Suffix.*

Ein Beispiel zur Veranschaulichung der Konzepte:

Beispiel 2.3 (BWT und Index C) *Der Index C für „abracadabra“ hat folgende Gestalt:*

$$C = [\$ = 0, a = 1, b = 6, c = 8, d = 9, r = 10]$$

Und die BWT ergibt sich aus dem Zeichen vor dem ersten Zeichen des Suffixes:

$$BWT = [ard\$rcaaaabb]$$

Zur Erzeugung der BWT ist jedoch ein vollständig lexikographisch sortiertes Suffixarray nötig, was Vergleiche von Strings bis zur Tiefe von n-1 nötig machen kann. Um den Erzeugungsaufwand zu verringern, wurden die Suchmöglichkeiten auf einer Struktur untersucht, die nur bis zu einer festen Tiefe $k \ll n-1$ sortiert wurde. Man erhofft sich eine weitere Verbesserung der Ausführungszeit, obwohl bereits lineare Algorithmen zur Konstruktion des Suffixarrays bekannt sind, wie unter anderem [KA03] zeigte.

2.4 Die k-BWT

Für die k-BWT wird eine stabile Sortierung benötigt.

Definition 2.7 (Stabile Sortierung) *Eine Sortierung der Folge $k_1, \ldots, k_n$ gilt dann als stabil, wenn für $i, j \in \{1, \ldots, n\}$ und $k_i = k_j$ mit $i < j$ gilt, dass k_i in der sortierten Folge vor k_j kommt.*

Anschaulich bedeutet dies, dass gleichwertige Elemente ihre ursprüngliche Reihenfolge behalten.

Definition 2.8 (k-BWT) *Die k-BWT wird ähnlich der BWT erzeugt:*

$$kBWT[i] = T_{kSA[i]-1 \mod n}$$

Wobei kSA ein Suffixarray ist, das nur bis zur Tiefe k stabil sortiert wurde.

Eine stabile Sortierung des Suffixarrays bis zur Tiefe k bedeutet, das in Bereichen, die bis zur Tiefe k identisch sind, die Suffixe mit zunahme der Position kürzer werden. Die zu sortierende Folge von Texten besteht aus der Menge aller Suffixe, wobei das erste Suffix der vollständige Text ist und das letzte Suffix nur noch das Abschlusszeichen

enthält. Die Länge eines Suffixes in der unsortierten Folge, ist also Invers zu seiner Position.

Beispiel 2.4 (k-BWT) *Würde „abracadabra“ nur mit $k = 1$ stabil sortiert, sähe das resultierende kSA wie in Tabelle 2 aus. Und die zugehörige k-BWT hätte die Form:*

Nr.	kSA	Suffix
1	11	$
2	0	abracadabra$
3	3	acadabra$
4	5	adabra$
5	7	abra$
6	10	a$
7	1	bracadabra$
8	8	bra$
9	4	cadabra$
10	6	dabra$
11	2	racadabra$
12	9	ra$

Tabelle 2: Beispiel kSA mit k=1 für den Text „abracadabra“, Suffixe sind nur zur Veranschaulichung mit angegeben.

$$k\text{-}BWT = [a\$rcdraaaabb]$$

Für die Rückwärtssuche wird jedoch mehr benötigt.

2.5 Der Bitvektor D

Die zusätzliche Datenstruktur, die für die k-BWT sowie die (k+1)-BWT benötigt wird, ist ein Bitvektor, der Rank- und Selectanfragen in konstanter Zeit unterstützen muss.

Definition 2.9 (Bitvektor) *Ein Bitvektor ist eine Datenstruktur mit n Einträgen über dem Alphabet {0,1}, auf dem, nach einer Vorverarbeitung mit linearer Zeit, Rank- und Selectanfragen gestellt werden können.*

Definition 2.10 (Rank- und Selectanfragen) *Sei $b \in \{0,1\}$ und $0 \leq i < n$. Eine Rankanfrage $rank_B(i,b)$, auf einem Bitvektor $B = b_0, \ldots, b_n$ gibt an, wie oft das Bit b in*

den Bits b_0 bis b_{i-1} vorkommt. Eine Selectanfrage $select_B(i,b)$ auf einem Bitvektor B gibt die Position des i-ten Vorkommens des Bits b in B zurück.

Wie [Jac88] für Rankanfragen, und später [Cla98] für Selectanfragen zeigte, sind diese in konstanter Zeit beantwortbar. Um Rank- und Selectanfragen bearbeiten zu können werden zusätzliche Informationen benötigt. Zum besseren Verständnis ein Beispiel.

Beispiel 2.5 (Bitvektor, Rank- und Selectanfrage) *Ein Bitvektor B mit folgender Gestalt*

$$B = [0110001011]$$

würde auf die Anfrage $rank_B(4,0)$ eine 2 zurück liefern, da $b_0 \dots b_{i-1} = b_0b_1b_2b_3 =$ 0110 genau zweimal die Null enthält. Die Anfrage $select_B(4,1)$ würde 8 zurück liefern, da die vierte Eins in B das Zeichen b_8 ist.

Die in dieser Arbeit verwendeten Bitvektoren, sowie die Strukturen zur Umsetzung der Rank- und Selectanfragen werden von der Succinct Data Structure Library (SDSL)[2] zur Verfügung gestellt.
Im Kontext der k-BWT hat der Bitvektor D_k (bzw. D_{k+1} für die (k+1)-BWT) die Aufgabe, die bis zur Tiefe k identischen Bereiche zu markieren. Es gilt:

$$D_k[i] = \begin{cases} 0, & \text{falls } kSA[i]_j = kSA[i-1]_j \text{ für alle } j \in 0,\dots,k \\ 1, & \text{sonst} \end{cases}$$

Beispiel 2.6 (D_k für die k-BWT) *Für die oben genannte 1-BWT von „abracadabra“ lautet der Bitvektor*

$$D_k = 110000101110$$

da alle Suffixe, die mit demselben Buchstaben beginnen, bis zur Tiefe k identisch sind. So wird für jede Position an der ein neuer Buchstabe erreicht wird eine Eins gesetzt.

2.6 Der Wavelet-Baum

Die in [GGV03] vorgestellte Datenstruktur des Wavelet-Baums ist eine Struktur, welche die erwähnten Bitvektoren verwendet, um Rank- und Selectanfragen für größere Alphabete anzubieten. Hierfür wird eine Bitcodierung des Alphabets benötigt.

[2] Siehe `http://www.uni-ulm.de/in/theo/research/sdsl.html`

Definition 2.11 (Wavelet-Baum) *Ein Wavelet-Baum WT wird für einen Text T über einem Alphabet Σ gebildet, wobei ein Zeichen $t \in \Sigma$ in $O(\log|\Sigma|)$ Bits codiert ist. Eine Rank- und Selectanfrage über dem Alphabet Σ wird in $O(\log|\Sigma|)$ Rank- und Selectanfragen über dem Alphabet 0/1 umgewandelt. Dies sind genau so viele wie Bits für die Repräsentation dieses Zeichens nötig sind.*
Ein Wavelet-Baum bildet eine Binärbaumstruktur, wobei jeder Knoten einen Bitvektor enthält, welcher auf der i-ten Ebene dem i-ten Bit der Bitrepräsentation der Buchstaben entspricht. Die Zuordnung der Zeichen zum rechten oder linken Kind des Knotens geschieht über das zugeordnete Bit dieser Tiefe. Alle Zeichen, die auf dieser Ebene durch eine 0 (bzw. 1) repräsentiert werden, werden im selben Kindknoten weitergeführt.

Die Rank- und Selectanfragen über einem Wavelet-Baum WT werden für das Zeichen c bis zur Stelle i, bzw. das i-te Vorkommen von c, als $rank_{WT}(i,c)$, bzw. $select_{WT}(i,c)$, bezeichnet.
Wavelet-Baums können sowohl eine Standard ASCII Codierung verwenden oder auch eine unbalancierte Codierung wie den Huffman Code. Zur Veranschaulichung der Möglichkeiten ein Beispiel:

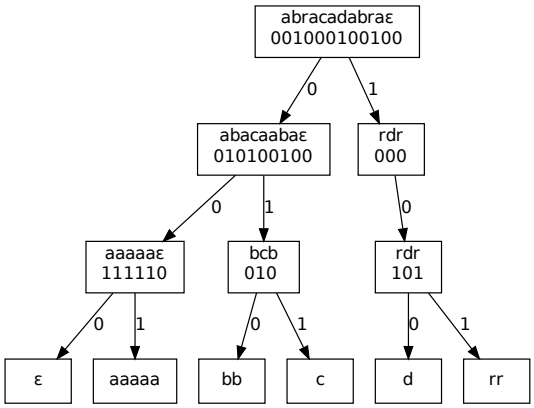
Beispiel 2.7 (Wavelet-Baum) *Für dieses Beispiel werden die Zeichen des Wortes „abracadabra“ auf zwei Möglichkeiten codiert, die in Tabelle 3 dargestellt sind.*

Zeichen	Codierung A	Codierung B
$	000	00000
a	001	1
b	010	01
c	011	0001
d	100	00001
r	101	001

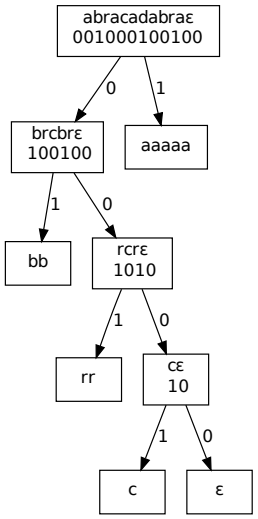
Tabelle 3: Zwei einfache Codierungen des Wortes „abracadabra“.

Die beiden Codierungen sind auch anschaulich an der Form des Wavelet-Baums gut zu unterscheiden, wie man in Abbildung 1 sieht.

Durch das Wesen des Wavelet-Baums ist, für eine balancierte Repräsentation, die Zeit für eine Rank- oder Selectanfrage mit $O(\log|\Sigma|)$ gegeben. In der Praxis werden Wavelet-Bäume über eine Huffmancodierung verwendet, welche diese Grenze nicht



(a) Wavelet-Baum zu Codierung A.



(b) Wavelet-Baum zu Codierung B.

Abbildung 1: Beispiele von Wavelet-Baums mit unterschiedlicher Codierung. Texte nur zur Übersicht eingefügt, gespeichert werden nur die Bitvektoren.

garantieren können, da ihre Tiefe nicht auf $O(\log|\Sigma|)$ beschränkt ist. Wavelet-Bäume werden verwendet, um die k- und (k+1)-BWT abzuspeichern und werden wie die Bitvektoren durch die SDSL[3] Biblitohek bereitgestellt.

2.7 Die Rückwärtssuche

Wie [Pet+11] dargelegt hat, wird für die Rückwärtssuche auf der k-BWT mehr als nur der Index C benötigt. Das hier dargestellte Vorgehen und die weiteren Strukturen sowie die benötigen Bitvektoren für die k- und (k+1)-BWT entstammen [Pet+11].

Definition 2.12 (Rückwärtssuche) *Rückwärtssuche bedeutet, das der zu suchende Text P bestehend aus den Zeichen $p_1, p_2, \ldots, p_m \in \Sigma$ beginnend mit dem letzten Zeichen p_m gesucht wird. Dann wird iterativ mit dem Zeichen $p_i : i = m-1, m-2, \ldots, 1$ fortgefahren.*

Auf der BWT bedeutet dies, einen Bereich $\langle start, ende \rangle$ zu finden, so dass im Suffixarray die Suffixe von $SA[start]$ bis $SA[ende]$ mit dem Text P beginnen.

[3] Siehe `http://www.uni-ulm.de/in/theo/research/sdsl.html`

$$start_j = LF(start_{j+1} - 1, p_j) + 1$$

$$ende_j = LF(ende_{j+1}, p_j)$$

Wobei mit $j = m = |P|$ begonnen wird und hierfür eine einfache Anfrage mittels C ausreicht. Die verwendete Funktion LF(i,c) heißt auch LF-Mapping.

Definition 2.13 (LF-Mapping) *Das LF-Mapping ist eine Funktion LF(i,c) mit $c \in \Sigma$ und $i \in \{0,\ldots,n-1\}$ für die BWT. Sie ist definiert als*

$$LF(i,c) = C[c] + rank_{BWT}(i,c)$$

.

Auf der k-BWT ist dies nicht ohne weiteres möglich, falls $m > k$ gilt. Gilt $m \leq k$, bilden die Suffixe, die mit P beginnen, einen geschlossenen Bereich. Eine Illustration dieses Sachverhalts bietet das folgende Beispiel.

Beispiel 2.8 (Rückwärtssuche auf der k-BWT) *Das Beispiel zur kSA (siehe Tabelle 2) findet noch einmal Verwendung. Wird der Text „a“ gesucht, liegen alle Ergebnisse im Bereich $\langle 2,6\rangle$. Wird jedoch der Text „ab“ gesucht, liegen zwar alle Ergebnisse im Bereich $\langle 2,5\rangle$, darunter sind jedoch nur Zwei echte Ergebnisse.*

Bei einer Suche bis zur Länge (k+1) funktioniert dies, da die Grenzen mit der untersuchten Länge übereinstimmt. Für darüber hinausgehende Suchen muss das LF-Mapping anders bestimmt werden. Hierfür werden zwei weitere Strukturen benötigt: Ein Text S und ein Vorhersagearray A.

Definition 2.14 (Der Text S) *Der Text S ähnelt im Aufbau der k-BWT. Die Formel zur Berechnung lautet nur wenig anders:*

$$S[i] = T_{kSA[i]+k+1 \mod n}$$

Es wird also das k-te Zeichen verwendet statt das vorausgehende. S wird wie die k-BWT und (k+1)-BWT als Wavelet-Baum gespeichert.

Definition 2.15 (Das Vorhersagearray A) *Das Vorhersagearray A enthält jeweils das (k+1)-te Zeichen der Bereiche im (k+1)-Suffixarray, die bis zur Tiefe (k+1) identisch sind.*

$$A[i] = T_{(k+1)SA[rank_{D_{k+1}}(i,1)]}$$

A enthält für jede Eins im Bitvektor D_{k+1} einen Eintrag, dies sind höchstens $|\Sigma|^k$ Einträge, da es höchstens so viele Unterschiedliche Präfixe geben kann unter den Suffixen.
Mit diesen Informationen kann nun das LF-Mapping auch für die Suche auf längeren Suchtexten genutzt werden, hierfür sind jedoch einige weitere Berechnungen nötig. Um die Position i des vorausgehenden Zeichens des aktuellen Kandidaten, der an der Position j liegt, zu finden, also LF(j,k-BWT_j) zu bestimmen, wird zunächst ein Wert p und ein Wert p* sowie ein Zeichen a bestimmt:

$$p = select_{D_{k+1}}(rank_{D_{k+1}}(j,1),1)$$

$$p* = select_{D_k}(rank_{D_k}(j,1),1)$$

$$a = A[rank_{D_{k+1}}(j,1)].$$

Das Zeichen a ist jenes Zeichen, das dem aktuellen Suffix an der Stelle k+1 folgen würde. Hierbei ist p der Beginn der Gruppe, der bis zur Tiefe k+1 identischen Suffixe in der (k+1)SA, und p* der Beginn der Gruppe, der bis zur Tiefe k identischen Suffix der kSA. Es wird noch ein weiterer Zwischenwert r benötigt:

$$r = select_S(rank_S(p*-1,a) + j - p + 1, a).$$

Mit diesen Zwischenwerten und dem gesuchten Zeichen c, können nun die beiden Werte bestimmt werden, aus denen sich danach das LF-Mapping zusammensetzen lässt:

$$p' = C[c] + rank_{(k+1)\text{-}BWT}(p*-1,c) + 1$$

$$q = rank_{k\text{-}BWT}(r,c) - rank_{k\text{-}BWT}(p*-1,c)$$

$$i = LF(j,k\text{-}BWT_j) = p' + q - 1.$$

Für diese Berechnung wird eine konstante Anzahl an Rank- und Selectanfragen auf den verschiedenen Wavelet-Bäumen benötigt. Die Berechnung von LF ist somit durch

die Berechnung dieser Anfragen eingeschränkt und abhängig von der gewählten Form des Wavelet-Baums. Für die Korrektheit des Vorgehens sei auf [Pet+11] verwiesen.

Die Suchanfrage nach einem Text P der Länge m benötigt damit $2(m-1)$ LF-Anfragen, die sich bei der k-BWT unterteilen: In bis zu $(k-1)$ Anfragen mit nur einer Rankanfrage und in $(m-k)$ Anfragen der oben beschriebenen Art mit einer konstanten Anzahl Rank- und Selectanfragen auf Wavelet-Bäumen. Damit ergibt sich eine Gesamtkomplexität von

$$2(k-1)c_{Rank_{WT}} + 2(m-k)(ac_{Rank_{WT}+bc_{Select_{WT}}}) = O(m(c_{Rank_{WT}} + c_{Select_{WT}})).$$

Hierbei stellt $c_{Rank_{WT}}$ die Komplexität einer Rank- und $c_{Select_{WT}}$ die einer Selectanfrage auf einem Wavelet-Baum dar. Es sind a und b jeweils Konstanten. Für nicht balancierte Wavelet-Bäume ergibt dies eine Komplexität von $O(m|\Sigma|)$. Damit liegt die Suche auf der BWT und der k-BWT in der selben Komplexitätsklasse, in der Praxis sind jedoch mehr Zugriffe für die k-BWT nötig, was die Suche verlangsamt.

Alternativ zu diesem Weg lassen sich die Unterschiede zwischen dem LF-Mapping der BWT und dem LF-Mapping der k-BWT auch direkt speichern und abfragen. [Pet+11] argumentiert hierbei, das für ausreichend kleine k der Weg über die Wavelet-Bäume weniger Speicherplatz benötigt als die direkte Korrekturinformation.

Zur Verdeutlichung dieses Vorgehens ein Beispiel.

Beispiel 2.9 (Rückwärtssuche auf der k-BWT 2) *Gegeben ist die k- und (k+1)-BWT, die Bitvektoren D_k und D_{k+1}, S und A. Die bis zur Tiefe (k+1) sortierten Suffixe sind zur Verständlichkeit mit angegeben.*

Die Strukturen die Benötigt werden, sind, bis auf C, in Tabelle 4 angegeben. C hat, wie in Beispiel 2.3 die Form $C = [\$ = 0, a = 1, b = 6, c = 8, d = 9, r = 10]$.

Gesucht werden soll der Text „bra“. Hierfür wird zunächst mit C der Bereich bestimmt, in welchem die Suffixe mit a beginnen: $\langle C[a], C[b]-1\rangle = \langle 1,5\rangle$. *Mittels einer Rankanfrage auf der (k+1)-BWT erfährt man, dass das erste r in diesem Bereich an der Position 1 und das letzte r an Position 4 zu finden ist. Nun möchte man den Bereich* $\langle 1,4\rangle$ *mittels des LF-Mappings zurückverfolgen.*

Nr	k			k+1			Suffixe
	D_k	k-BWT	S	D_{k+1}	(k+1)-BWT	A	
0	1	a	b	1	a	b	$
1	1	r	a	1	r	a	a$
2	1	$	r	1	$	r	abracadabra$
3	0	d	r	0	d	a	abra$
4	1	r	a	1	r	a	acadabra$
5	1	c	a	1	c	a	adabra$
6	1	a	a	1	a	d	bracadabra$
7	0	a	a	0	a	b	bra$
8	1	a	d	1	a	$	cadabra$
9	1	a	b	1	a	c	dabra$
10	1	b	c	1	b		ra$
11	0	b	$	1	b		racadabra$

Tabelle 4: Die, bis auf C, nötigen Strukturen zur Suche am Beispiel des Textes „abracadabra“. Bis zur Tiefe k+1 sortierte Suffixe zur Verständlichkeit mit angegeben.

Da der Wert k+1 noch nicht erreicht ist, kann das LF-Mapping für die BWT verwendet werden:

$$LF(1) = LF(1,r) = C[r] + rank_{(k+1)\text{-}BWT}(1,r) = 10+0 = 10$$
$$LF(4) = LF(4,r) = C[r] + rank_{(k+1)\text{-}BWT}(4,r) = 10+1 = 11$$

Damit wird der Bereich $\langle 10,11\rangle$ erhalten. Im (k+1)-Suffixarray erkennt man, dass dies die Suffixe sind, die mir „ra“ beginnen. Um fortfahren zu können, muss auf die zweite Variante zur Bestimmung von LF gewechselt werden.

Bez.	LF(10,b)	LF(11,b)
p	10	11
p*	10	10
a	$	c
r	11	10
p’	7	7
q	1	0
LF	7	6

Tabelle 5: Bestimmung von LF(10) und LF(11) mittels der erweiterten Methode.

In Tabelle 5 ist die Berechnung der einzelnen Werte dargestellt. Auf diese Weise erhält man den Bereich $\langle 6,7\rangle$, der genau dem Bereich des (k+1)-Suffixarrays entspricht, wobei die Suffixe mit „bra“ beginnen. Da jede Position, in diesem Bereich der (k+1)-BWT, das Zeichen a enthält, kann auch geschlossen werden, das hier der Text „abra“ vorliegt.

Hiermit sind alle Strukturen und Methoden für die Suche auf der k-BWT gesammelt, zu ihrer Erzeugung wird noch der Radixsort benötigt.

2.8 Der Radixsort

Der Radixsort ist ein Sortierverfahren nach dem Schema eines Bucketsort und bietet je nach Variante ein stabiles oder nicht stabiles Sortierergebnis. In [MBM93] werden gleich mehrere Implementierungen des Radixsort in C angegeben und untersucht. [KR08] bietet noch Verbesserungen dieser Implementierung und zeigt experimentell, dass diese auf einigen Eingangsdaten die effizienteste bekannte Möglichkeit darstellt, eine Menge von Texten lexikographisch zu sortieren. Der verwendete Radixsort entspricht einer Variante des Most Significant Digit (MSD) Radixsort, der vom ersten zum letzten Element gehend arbeitet.
Der Radixsort benötigt ein Alphabet das einer totalen Ordnung unterliegt, das also gilt: Für $t_i, t_j \in \Sigma$

$$t_i \neq t_j \Longrightarrow t_i < t_j \text{ oder } t_i > t_j$$

Diese Ordnung wird in der Praxis durch die Bitrepräsentation eines Zeichens und die Kleiner-als Relation $<$ zwischen diesen Repräsentationen gegeben.
Der Grund, weshalb solch eine totale Ordnung benötigt wird, ist anhand der Funktionsweise schnell ersichtlich:

Definition 2.16 (Radixsort) *Der Radixsort sortiert die Zeiger auf Texte $T^1, T^2, \ldots, T^n$. Der Text, auf den der Zeiger T^i zeigt, besteht aus den Zeichen $t^i_1, t^i_2, \ldots, t^i_{|\Sigma|} \in \Sigma$. Die aktuelle Tiefe des Sortierschrittes ist d, beim Start des Algorithmus gilt $d = 0$.*

- *Bilde $|\Sigma|$ Buckets $B_{t_1}, \ldots, B_{t_{|\Sigma|}}$ in der gewünschten Ordnung.*
- *Für i = 1 bis n:*
  - *Weise T^i dem Bucket $B_{t^i_d}$ zu.*
- *Für $c = t_1, \ldots, t_{|\Sigma|}$*

  - *Sortiere das Bucket B_c mit $d_{neu} = (d+1)$.*
  - *Entnehme den ersten Text aus B_c und wähle diesen als T^p, wobei T^p der erste noch nicht neu vergebene Zeiger ist.*
- *Gib die neu gewählten Zeiger $T^1, T^2, \ldots, T^n$ zurück.*

Der hier angegebene Radixsort ist ein Most-Significant-Digit (kurz MSD) Radixsort und beginnt mit dem ersten Zeichen, es existieren auch Varianten, welche mit dem letzten Zeichen beginnen.Um das Verhalten des Radixsort besser nachvollziehen zu können hier wieder ein kurzes Beispiel.

Beispiel 2.10 (Radixsort) *Die Texte T^1 =„abra“,T^2 =„adabra“ und T^3 =„dabra“ werden folgendermaßen sortiert:*
Tiefe d=0; benötigte Buckets: B_a, B_d

Bucket	Zugewiesene Texte
B_a	*a*bra, *a*dabra
B_d	*d*abra

Der Text „dabra“ landet in Bucket B_d, da sein Zeichen mit der Nummer Null t_0^3 =d ist, während die Texte „abra“ und „adabra“ jeweils als untersuchtes Zeichen $t_0^1 = t_0^2$ =a haben.
Nun müssen die einzelnen Buckets sortiert werden. Für B_a wird also d=1 gewählt. Wieder werden nun die neuen Buckets B_b und B_d benötigt, da sie als Zeichen an der Stelle Eins ein b und ein d haben.

Bucket	Zugewiesene Texte
B_b	a*b*ra
B_d	a*d*abra

Die sortierten Buckets B_b und B_d werden nun geleert. Zunächst wird B_b vollständig geleert, da b<d gilt. T^1 ist demnach „abra“. Hiernach wird T^2 als „adabra“ gewählt. Im Schritt d=0 entspricht das Bucket B_a den eben gewählten Texten T^1 und T^2. Das endgültige T^1 wird also gleich dem aus B_a gewählt: T^1 =„abra“. Ebenso T^2 =„adabra“. Das Bucket B_d ist ebenfalls bereits sortiert und der darin enthaltene Text wird als T^3 gewählt. So entsteht die sortierte Reihenfolge der Texte als „abra“,„adabra“,„dabra“.

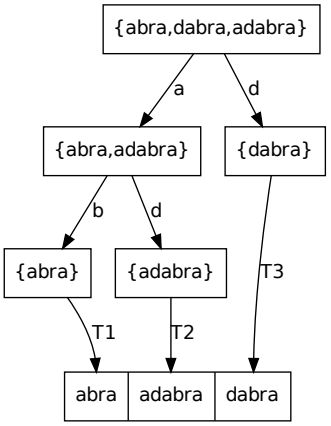


Abbildung 2: Das Radixsort Beispiel aufgelöst als Verlaufsgraph.

In einer Implementierung des Radixsort müssen jedoch einige Feinheiten beachtet werden, da sonst viel Geschwindigkeit durch das Cachingverhalten des ausführenden Rechners verloren geht. Diese Punkte werden in [KR08] genauer betrachtet und Möglichkeiten angegeben, das Cachingverhalten eines Rechners zur Steigerung der praktischen Ausführungsgeschwindigkeit zu nutzen.
Der dort angegebene American-Flag-Sort bietet eine nicht stabile Sortierung, die alle Verbesserungen eingebaut hat. Für diese Arbeit wird die Implementierung dieses Algorithmus verwendet.

2.9 Systematik

Die Implementierung aus Kapitel 3 wurde mit der GNU Compiler Collection (GCC) Version 4.6.1, mit dem Compilerflag O3 zur Optimierung, übersetzt. Als zugrunde liegendes Betriebssystem wurde für den Test Ubuntu 11.10 mit dem Linux Kernel 3.0.0 auf einem AMD Phenom^{TM} II X4 965 mit einem Betriebstakt von 3.2 GHz und 8 GB DDR3 Ram eingesetzt.
Die verwendete Bibliothek SDSL[4] wurde in der Version 0.9.8 vom Oktober 2011 und die nötige Bibliothek divsufsort[5] wurde in der Version 2.0.1 vom November 2010 verwendet. Beide wurden mit der oben angegebenen GCC Version übersetzt.

[4]Siehe `http://www.uni-ulm.de/in/theo/research/sdsl.html`
[5]Siehe `http://code.google.com/p/libdivsufsort/`

Als Entwicklungssprache wurde C++ gewählt. Die Messung der benötigten Zeit geschieht über mehrere Durchläufe mit der von der SDSL bereitgestellten stop_watch Klasse. Für den Vergleich wird der Durchschnitt der gemessenen Werte der Usertime verwendet. Die Verwendeten Testeingabedaten entstammen dem Pizza&Chili Corpus[6].

[6]Siehe `http://pizzachili.dcc.uchile.cl/texts.html`

3 Die Algorithmen im Vergleich

In diesem Kapitel werden die in Kapitel 2 vorgestellten Konzepte und Datenstrukturen tatsächlichen verwendet und erzeugt. Die Implementierung der hier beschriebenen Algorithmen liegt als C++ Klassenstruktur vor, die der Abbildung 3 entspricht.

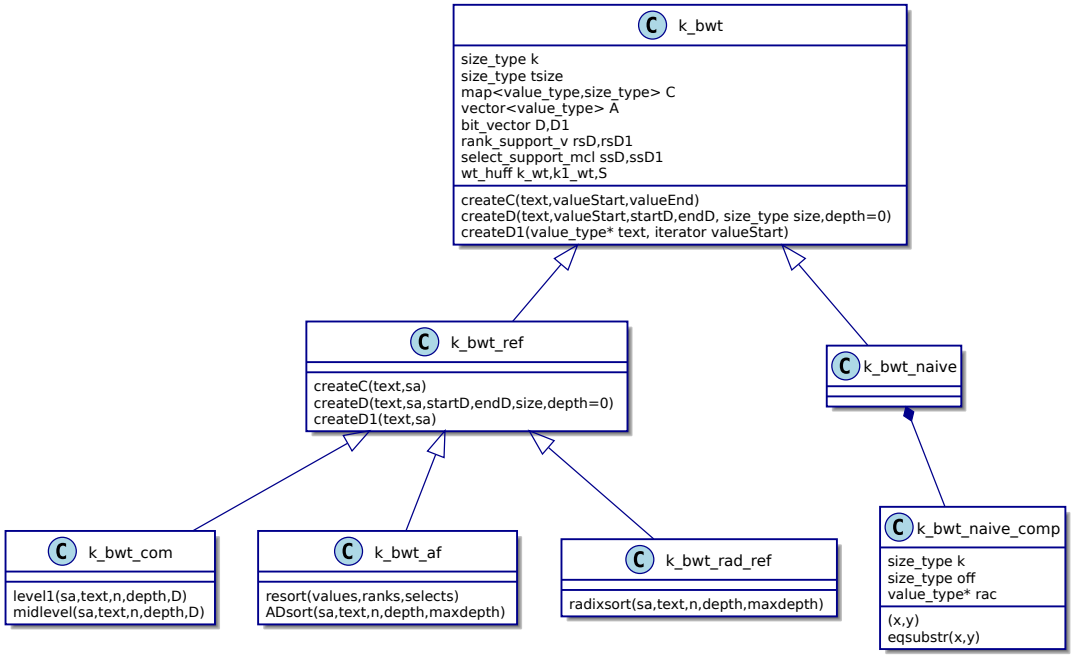


Abbildung 3: Die Struktur der Klassen im Namespace bc_m.

3.1 Die Basiskonstruktion

Um die verschiedenen Methoden zur Erzeugung des k-SA möglichst vergleichbar zu halten, ist die Basiskonstruktion in verschiedene Schritte unterteilt, die sich nur wenig voneinander unterscheiden und wenn möglich die gleichen Methoden verwenden. Allgemein lässt sich die Konstruktion in folgende Schritte aufspalten:

1. Erzeuge den Container des zukünftigen k-SA.
2. Berechne das k-SA.
3. Berechne den Bitvektor D_k.
4. Berechne die k-BWT und S aus dem k-SA.
5. Berechne den Index C.
6. Erzeuge den Wavelet-Baum WT_k der k-BWT.

7. Berechne das (k+1)-SA.
8. Berechne den Bitvektor D_{k+1}.
9. Berechne die (k+1)-BWT aus dem (k+1)-SA.
10. Erzeuge den Wavelet-Baum WT_{k+1} der (k+1)-BWT.
11. Erzeuge des Vorhersagearray A.

Hier werden zunächst die allgemein verwendeten Schritte untersucht, die sich unterscheidende Berechnung des k-SA wird weiter unten besprochen.

3.1.1 Die k-SA Initialisierung

Die Erzeugung des Containers des k-SA enthält auch das Füllen des Containers mit den Werten $0, 1, \ldots, n$, die als Zeiger auf den Text verstanden werden. Hierbei ist Text[0] das Suffix, das an Stelle 0 beginnt, also der gesamte Text.
Die Erzeugung, auf einem System mit 32 Bit Integern, benötigt mindestens 5n Bytes, da der Text als Zeichen-Array und das k-SA als Integer-Array vorliegt.

3.1.2 Der Bitvektor D_k

Der Bitvektor D_k ist mit 0 initialisiert, die Stellen an denen eine 1 benötigt wird, werden rekursiv ermittelt. Es wird abgebrochen, sobald die vorgegebene Tiefe von k erreicht wurde oder der zu untersuchende Bereich nur noch ein Zeichen lang ist. In beiden Fällen wird die erste Stelle des Bereichs mit 1 markiert.
Zeichen treten stets in Gruppen auf. Die vordere Grenze dieser Gruppe bildet der erste Buchstabe, die hintere Grenze wird mittels binärer Suche ermittelt. Innerhalb dieser Gruppe wird dann eine Ebene tiefer gegangen und der Algorithmus wiederholt. Hiernach wird die nächste Gruppe bearbeitet, indem der Startpunkt auf die erste Stelle nach der bisherigen Gruppe gesetzt wird.
Die Komplexität der Funktion ist $O(n^k \log n)$, da die Rekursion schlimmstenfalls bis zur Tiefe k durchgeführt wird und hierbei jedes mal eine binäre Suche nötig ist. Diese Komplexität übersteigt diejenige der Sortierung, zudem ähnelt das Vorgehen dem der Sortierung. Deshalb ist die Erzeugung des Bitvektors D_k ein guter Kandidat für eine Zusammenlegung mit der Erzeugung der k-SA. Zum besseren Vergleich der Sortierstrategien, auch mit der naivem Implementierung die einen weniger ähnlichen Sortiervorgang verwendet, wird diese Zusammenlegung noch nicht vorgenommen.

3.1.3 Die k-BWT, (k+1)-BWT und S

Die Erzeugung der k-BWT aus dem k-SA wird gleichzeitig genutzt, um einen Container zu erstellen, der einfach weiter verarbeitet werden kann. Die Konstruktion für die k- und (k+1)-BWT ist einfach, es ist eine direkte Übernahme der in Definition 2.8 angegebenen Funktion - für S eine direkte übernahme der Funktion in Definition 2.14. Der Nachteil ist jedoch eine an jeder Stelle durchgeführte Überprüfung ob das k-SA hier auf den vollen Text verweist, da in diesem Fall das letzte Zeichen des Textes verwendet werden muss.

Dieses Problem kann umgangen werden, wenn dies beim Einlesen des Textes berücksichtigt wird und das Feld vor dem ersten Zeichen mit zur Verfügung steht und identisch mit dem letzten Zeichen ist. Selbiges gilt für das Hilfskonstrukt S, hierbei müssen jedoch k weitere Zeichen am Ende des Textes existieren.

Der Algorithmus benötigt $O(n)$ Zeit und erzeugt eine Datenstruktur die ungefähr n Bytes Platz verbraucht. Auch hier wird jedoch zusätzlicher Speicher für die Verwaltung nötig.

3.1.4 Der Index C

Die Erstellung des Index C erfolgt mittels eines einzelnen Durchlaufs des k-SA. Dies reicht aus, da die erste Stelle der Texte auf jeden Fall sortiert vorliegt. Der Index C wird mittels einer Map realisiert, das heißt, dass jedem, im Text vorkommenden, Symbol aus dem Alphabet Σ mit seinem ersten Vorkommen im k-SA verknüpft wird.

Die Laufzeit des Algorithmus liegt damit in $O(n)$ während der zusätzliche Speicher der Struktur ungefähr 5*|(verwendete Zeichen in T)|+$O(|\Sigma|)$ beträgt, wobei $O(|\Sigma|)$ dem Overhead entspricht, jener Speicher der von der Map Struktur zur Verwaltung benötigt wird. Dieser Overhead ist implementierungsabhängig. Nach [Das+08] wird ein Rot-Schwarz-Baum verwendet, der damit eine Platzkomplexität von $O(|\Sigma|)$ hat. Auch die Erstellung des Index C ist ein guter Kandidat für Verbesserung, da das Zählen der Zeichen zu verschiedenen Zeitpunkten durchgeführt werden kann: Sowohl beim einlesen des Textes, als auch beim ersten Durchlauf des Radixsort, der die Größen seiner Behälter bestimmen muss.

3.1.5 Die Wavelet-Bäume WT_k, WT_{k+1} und WT_S

Das Erzeugen des Wavelet-Baums wirkt zunächst nicht intuitiv, hat allerdings Gründe. Für die (k+1)-BWT wird der Wavelet-Baum analog erzeugt.
Die erzeugte k-BWT wird in eine Datei ausgelagert, um dann von einem speziellen Konstruktor geladen zu werden. Dieser Umweg wird gegangen, da in Version 0.9.8 der Bibliothek SDSL die Konstruktoren unterschiedlich implementiert sind. Die Verwendung der construct Methode, die eine Datei als Eingabe erfordert, führt zu einer starken Verkürzung der Ausführungszeit auf den getesteten Eingabedaten, wie man in Tabelle 6 sehen kann.

Eingabedaten	Größe	Standardkonstruktor	construct Methode
XML	100MB	22,5	2,3
DNA	100MB	14,8	2,4
Code	100MB	26,2	2,6
XML	50MB	11,1	1,1
DNA	50MB	7,3	1,4
Code	50MB	13,0	1,3
Englisch	50MB	13,3	1,4

Tabelle 6: Erzeugungsdauer in 1000ms des Wavelet-Baums WT_k für unterschiedliche, mit k=5 verarbeitete Eingabedaten, jeweils mit Standard- und spezialisiertem Konstruktor.

Da bei der Konstruktion Zeit für die Auslagerung verloren geht, ist es wünschenswert, in späteren Versionen der SDSL auf einen verbesserten Konstruktor zu wechseln.

3.1.6 Der Bitvektor D_{k+1}

Der Bitvektor D_{k+1} bietet den Vorteil, dass er auf Basis des Bitvektors D_k berechnet werden kann. Es gilt nur die 0-Bereiche auf der Tiefe k zu untersuchen. Zu diesem Zweck wird als erstes eine Kopie des Bitvektors D_k angelegt, sowie die Gesamtzahl der 1-Einträge in D_k bestimmt.
In einer Schleife wird jeder 0-Bereich im Bitvektor D_k untersucht, ob es auf Tiefe k+1 Unterschiede gibt, da bekannt ist, das er bis zur Tiefe k identisch ist.
Bei einer ersten Auswertung mit k=5 fiel jedoch auf, dass der Algorithmus zur Bestimmung von D_k um einiges schneller war als der zur Bestimmung von D_{k+1}.

Deshalb wurde der Algorithmus für D_k modifiziert um auch D_{k+1} bestimmen zu können, wobei temporär k=k+1 gesetzt wird. Dieser wurde dann mit dem, oben angegebenen, Algorithmus für D_{k+1} verglichen. Das Ergebnis ist in Abbildung 4 zu sehen.
Wie man leicht sieht, wird der Algorithmus um D_{k+1} aus D_k zu bestimmen mit zunehmendem k schneller, wohingegen der modifizierte Algorithmus für die gesamte Bestimmung nur in niedrigen Bereichen für k, ungefähr $k \leq 11$, schneller ist. Hierbei scheint das entsprechende k jedoch mit der Eingabegröße anzusteigen. Als Lösung kann nun, je nach gewähltem k, der wahrscheinlich effizientere Algorithmus verwendet werden.

3.2 Berechnung des k-SA

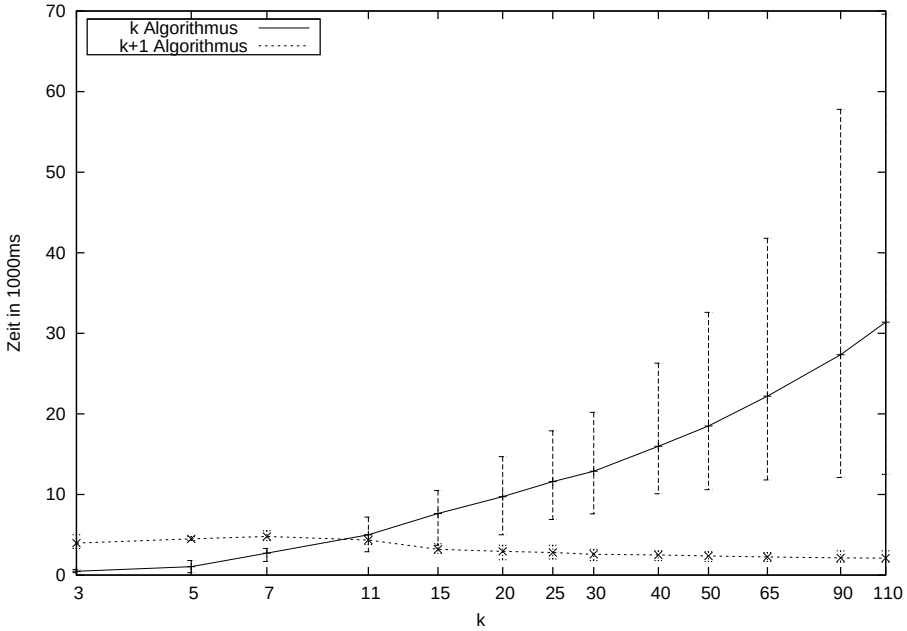
Zur Berechnung des k-SA wird der Radixsort verwendet, außer in der naiven Implementierung bei welcher der stable_sort der STL zum Einsatz kommt. Der Radixsort kommt in zwei Versionen zum Einsatz: Als

- American-Flag-Sort, aus [MBM93],
- dem stabilen Radixsort CE2 aus [KR08].

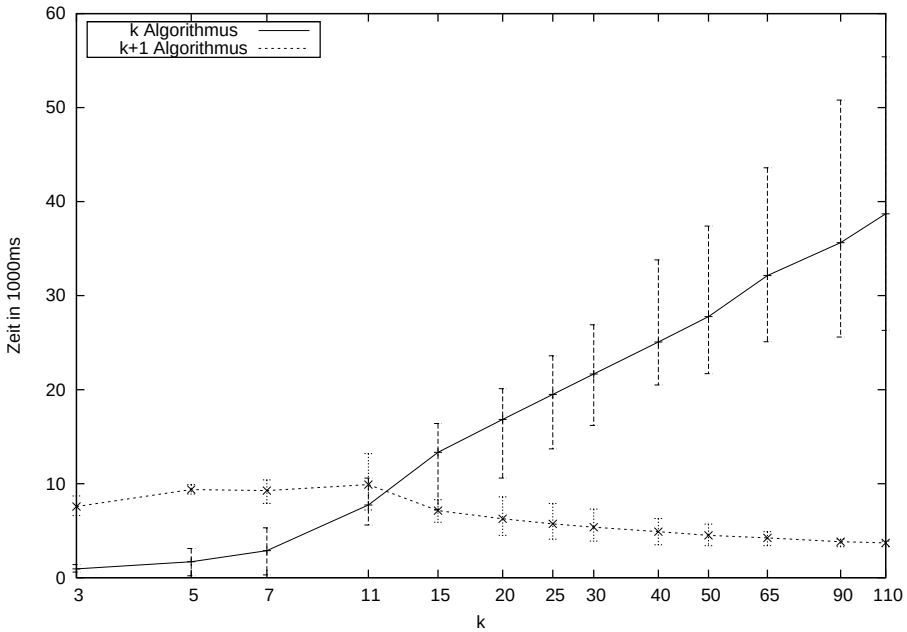
Der American-Flag-Sort ist jedoch nicht stabil und benötigt daher eine Rücksortierung um die Stabilität zu erreichen. Die theoretische Komplexität des Radixsort beträgt $O(kn)$ für die Tiefe k und n als die Länge des Eingabetextes.

3.2.1 Die Rücksortierung

Um mit dem American-Flag-Sort zu einem stabilen Ergebnis zu kommen, wird jeder Bereich in des k-SA, der einem 0-Bereich im zugehörigen Bitvektor entspricht, mittels des sort Algorithmus der STL sortiert. Da die Werte des k-SA eindeutig sind und innerhalb des Bereichs aufsteigend sein müssen, führt dies zum gewünschten Ergebnis. Obwohl die Minimalwerte der American-Flag-Sortierung die Maximalwerte der stabilen Sortierung oft überschreiten, ist in Abbildung 5 gut zu erkennen, dass die durchschnittliche Bearbeitungszeit der stabilen Sortierung niedriger ist. Mit größerem k wird das Verhältnis der Laufzeiten zwar besser, die stabile Sortierung behält jedoch ihren Vorteil.

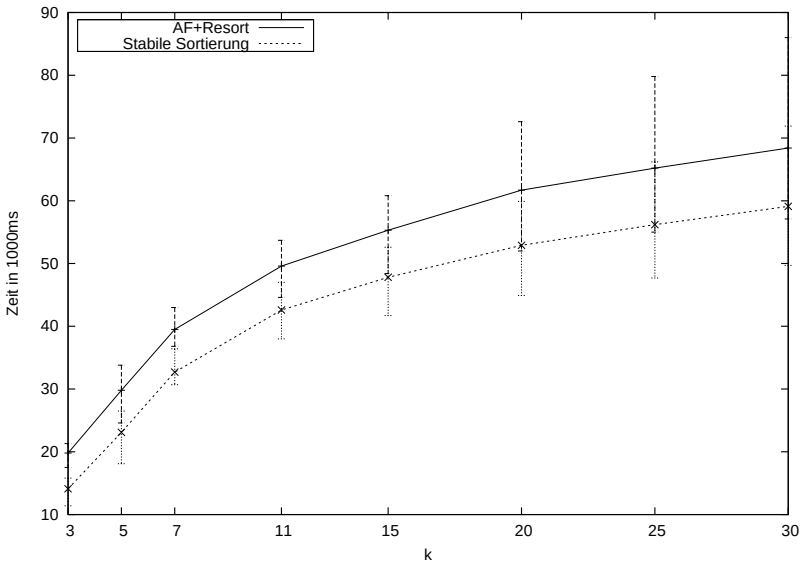


(a) Eingabedateien der Größe 50 MB.

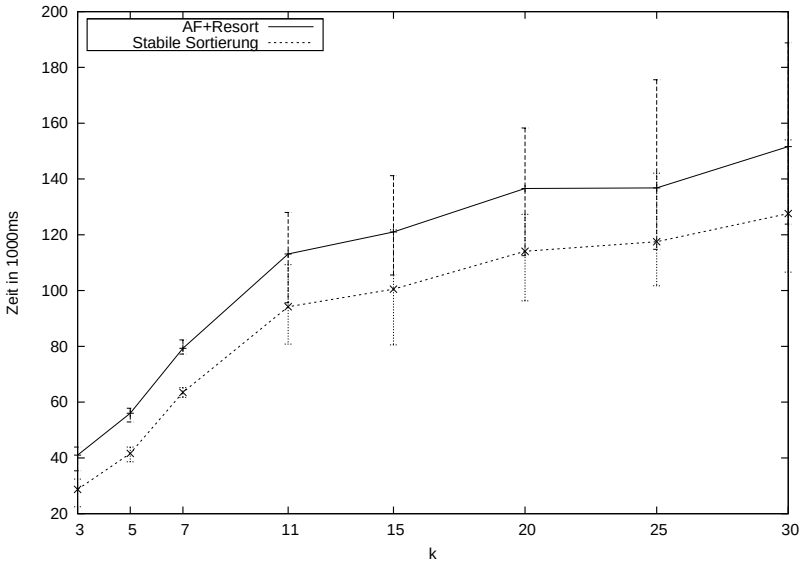


(b) Eingabedateien der Größe 100 MB.

Abbildung 4: Vergleich der Algorithmen zur Bestimmung von D_{k+1} anhand ihrer Laufzeit in 1000ms. Abgebildet sind die Durchschnittswerte mit den Maximal und Minimalabweichungen. Die X-Achse ist mit logarithmischen Abständen gewählt.



(a) Eingabedateien der Größe 50 MB.



(b) Eingabedateien der Größe 100 MB.

Abbildung 5: Vergleich der Möglichkeit zur Sortierung mittels des American-Flag-Sorts mit einer Rücksortierung der Werte und der stabilen Sortierung. Messzeiten in 1000ms.

3.2.2 Weiterverarbeitung zum (k+1)-SA

Der Sortieralgorithmus kommt ein weiteres mal zum Einsatz, um das (k+1)-SA zu erzeugen. Hierbei muss nicht die gesamte Sortierung neu vorgenommen werden, sondern nur die bis zur Tiefe k identischen Bereiche, die im Bitvektor D_k bestimmt wurden, um eine weitere Ebene sortiert werden.

Um dies zu erreichen wird der Algorithmus zur Bestimmung der 0-Bereiche auf D_k verwendet und diese dann ausgehend von der Tiefe (k-1) bis einschließlich der Tiefe k sortiert.

3.2.3 Das Vorhersagearray A

Für die Erzeugung des Vorhersagearrays A ist es nur nötig, für jeden Eintrag des Bitvektors D_{k+1} der eine Eins enhält ein Element zu speichern. Da es höchstens $|\Sigma|^k$ solcher Einträge gibt und die Laufzeit der Rank und Selectanfragen, die zum Finden des nächsten Eintrags mit einer Eins, konstant ist, beträgt die Laufzeit $O(|\Sigma|^k)$.

3.3 Optimierungen

In diesem Teil sollen kurz Optimierungsmöglichkeiten besprochen werden, die zu weiteren Verbesserungen beitragen können.

3.3.1 Berechnung der (k+1)-BWT aus der k-BWT

Die (k+1)-BWT müsste nicht vollständig neu aus dem (k+1)-SA gewonnen werden, da sich einige Bereiche gar nicht ändern. Es wäre also möglich zunächst eine Kopie der k-BWT zu erstellen und diese dann zu modifizieren um die (k+1)-BWT zu erhalten.

Da hierbei immer zusammenhängende Speicherbereiche kopiert würden und meist nur wenige Bereiche neu durch das (k+1)-SA bestimmt werden müssten, könnte das Verfahren im Allgemeinen vom Cachingverhalten eines Rechners profitieren.

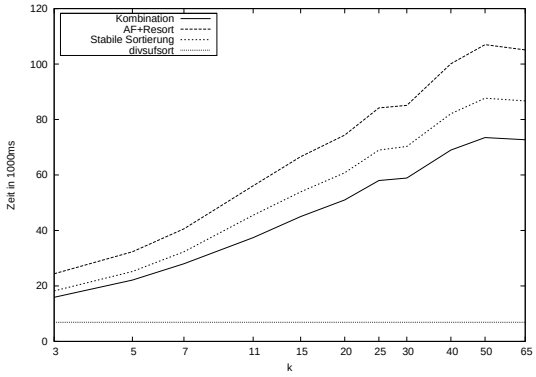
Hierfür müsste jedoch genauer untersucht werden, welche Bereiche sich tatsächlich ändern, um den Gewinn an Ausführungszeit nicht zunichtezumachen, durch das Überschreiben großer Bereiche und zusätzlichen Suchaufwand für die Bereiche.

3.3.2 Eliminierung von Abfragen

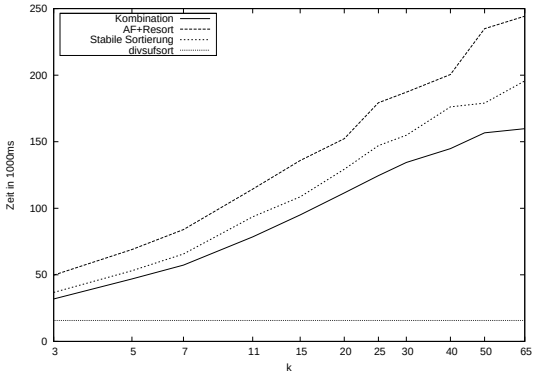
Durch ein weiteres Datenelement vor dem Array *rac*, dem Container des Textes, das mit dem Wert des letzten Elements identisch ist, können Abfragen vermieden werden, die Bereichsüberschreitungen beim Bestimmen der k-BWT verhindern.
Durch die Existenz des Feldes $rac[-1]$ und dessen Gleichheit mit $rac[size-1]$ ist eine Unterscheidung zwischen den beiden Ereignissen, das k-SA zeigt auf die Stelle Null oder sie tut es nicht, nicht mehr nötig. Da der Text möglicherweise von Außen gegeben ist, würde dies in diesem Fall eine Reallokierung des gesamten Textes nötig machen, was den Nutzen aufwiegen könnte. Wird das Einlesen des Textes jedoch auch von der Konstruktion übernommen, ist eine effiziente Möglichkeit einfach umzusetzen.

3.4 Kombination

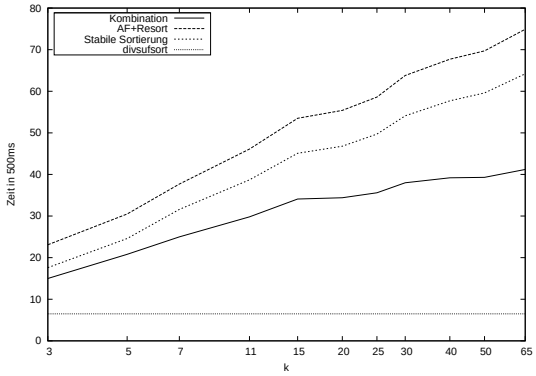
Da der CE2-Radixsort eine erste Zählphase hat, in der die Größe der verwendeten Buckets bestimmt wird, und zudem die Buckets bei Tiefe k genau den Bereichen entsprechen, die im Bitvektor D markiert werden müssen, kann weitere Zeit gespart werden, wenn diese Prozesse verschmolzen werden.
Zudem wird die in Eliminierung von Abfragen (3.3.2) vorgestellte Optimierung schon beim Einlesen des Textes eingebaut, es wird also von vornherein ein zusätzliches Feld alloziert, das ebenfalls das Nullbyte enthält.
Hierfür ist eine Modifikation der in [KR08] vorgestellten Algorithmen nötig. Es muss zwischen erster Ebene, welche die Erstellung des Index C übernimmt, und anderen Ebenen unterschieden werden. Zudem muss der jeweils nötige Abschnitt des Bitvektors D mit übergeben und in den Abbruchbedingungen und im verwendeten Insertion-Sort richtig gesetzt werden.
Die Auswertung dieses Algorithmus über verschiedenen Eingabedaten zeigt Abbildung 6. Die Kombination der verschiedenen Algorithmen bringt noch einmal eine signifikante Verbesserung der Laufzeit, so dass für sogar die Laufzeit des divsufsorts für einen einzigen Fall unterschritten werden kann, siehe Abbildung 6f. Die ebenfalls untersuchte naive Implementierung bleibt in allen Fällen weit hinter den angegebenen Algorithmen zurück.
Wird zudem eine einfache Variante der Berechnung der (k+1)-BWT aus der k-BWT (3.3.1) verwendet, verbessert dies die Ausführungszeit nicht in allen Fällen, sondern



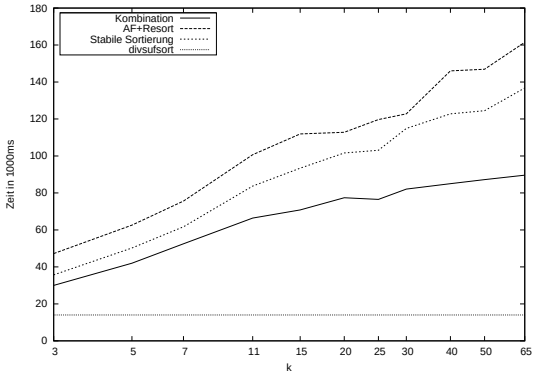
(a) XML Daten der Größe 50 MB.



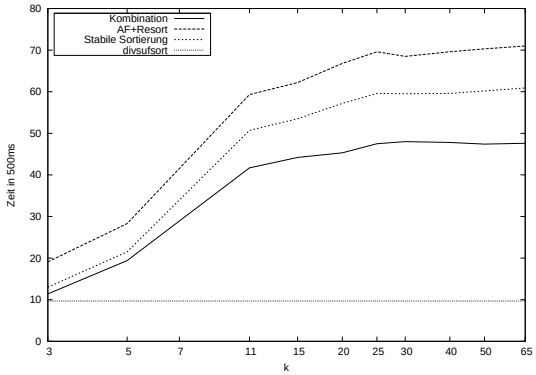
(b) XML Daten der Größe 100 MB.



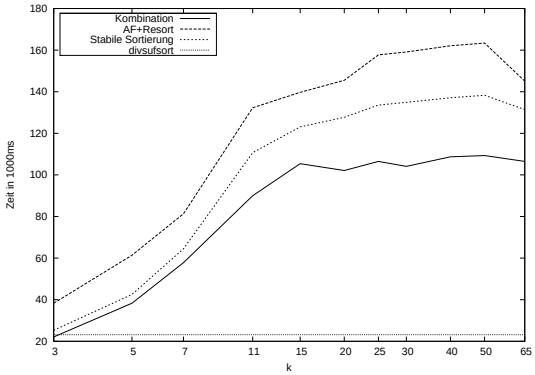
(c) Quellcodes der Größe 50 MB.



(d) Quellcodes der Größe 100 MB.



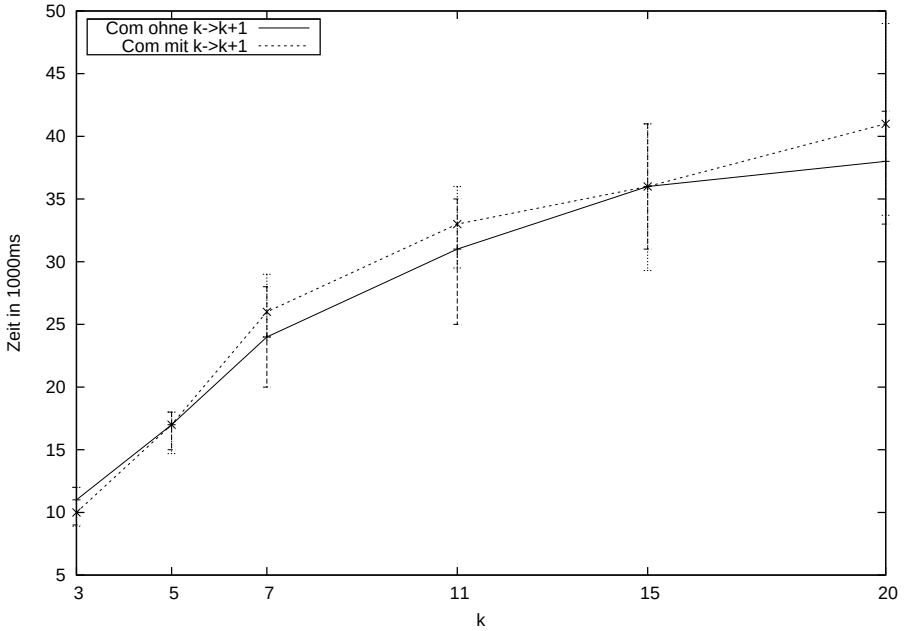
(e) DNA Eingabedaten der Größe 50 MB.



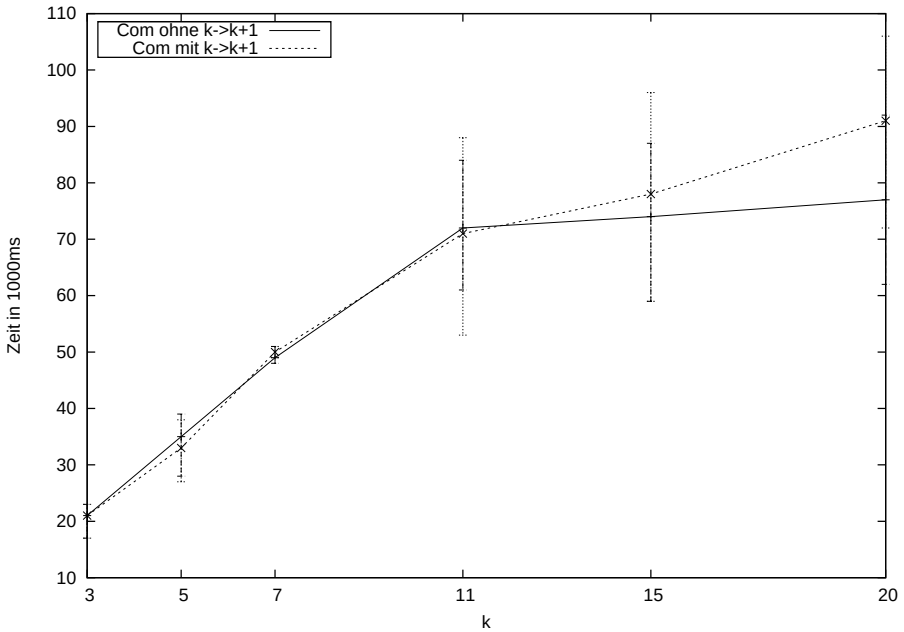
(f) DNA Eingabedaten der Größe 100 MB.

Abbildung 6: Vergleich der Algorithmen zur Berechnung aller Strukturen und des divsufsorts mit verschiedenen Eingabedaten. Die X-Achse ist mit logarithmischen Abständen gewählt.

verschlechtert sich in den meisten, siehe Abbildung 7. Die Optimierung muss also sorgfältiger gewählt werden.



(a) Auf Eingabedateien der Größe 50 MB.



(b) Auf Eingabedateien der Größe 100 MB.

Abbildung 7: Vergleich des Kombinierten Algorithmus mit einfacher Optimierung der (k+1) Berechnung und ohne Optimierung. Ohne die Berechnung von S und A.

4 Zusammenfassung

In Kapitel 3 wurden im wesentlichen drei Algorithmen, zur Erzeugung der k-Burrows-Wheeler Transformation und der vorgestellten Strukturen, untersucht. Jeweils unter Verwendung:

1. Des *stable_sort* Algorithmus der Standard-Template-Library.
2. Des stabilen Radixsort *CE2* aus [KR08].
3. Des nicht stabilen *American-Flag-Sort* aus [MBM93] mit einer Rücksortierung zur Erzeugung der Stabilität. Für die Rücksortierung wurde ein Quicksort verwendet.

Während sich die Verwendung des *stable_sort* als nicht konkurrenzfähig herausstellte, zeigten die beiden Varianten des Radixsort vergleichbare Ergebnisse, obwohl die stabile Variante, Punkt 2 der Liste, einen deutlichen Vorsprung innehatte.
Weitere Optimierungen am Algorithmus, welcher die stabile Variante des Radixsort einsetzte, um überflüssige Vorgänge und Berechnungen einzuschränken, konnten das Ergebnis weiter verbessern. In wenigen Fällen konnte er sogar die Laufzeit des *divsufsort*[7], zur Erzeugung des Suffixarrays, unterschreiten.
Die Möglichkeiten der Optimierung sind mit den hier eingearbeiteten nicht erschöpft. Das Cachingverhalten moderner Rechner benötigt besondere Aufmerksamkeit und die hier verwendete Erzeugung der Wavelet-Bäume ist, aufgrund der verwendeten Bibliothek, nicht optimal.
Bis diese und weitere Verbesserungen untersucht und eingearbeitet sind, sind die Algorithmen zur Erzeugung noch nicht zu empfehlen. Mit der Berechnung des Suffixarrays steht eine schnellere Berechnung auch für große Datenmengen zur Verfügung. Zudem kann aus dieser, mit der Burrows-Wheeler-Transformation, ein Index erzeugt werden, der geringere Platzansprüche hat als die hier vorgestellten Strukturen.

[7] Eine Bibliotheksfunktion, der gleichnamigen C-Bibliothek, die als sehr schnell gilt.

Literatur

[BW94] M. Burrows und D. J. Wheeler. *A block-sorting lossless data compression algorithm*. Techn. Ber. 1994.

[Cla98] D. R. Clark. „Compact pat trees“. Diss. Waterloo, Ont., Canada, 1998.

[CPP11] J. Shane Culpepper, Matthias Petri und Simon J. Puglisi. „Revisiting bounded context block-sorting transformations“. In: *Software: Practice and Experience* (2011), n/a–n/a.

[Das+08] D. Das u. a. „Speeding up STL Set/Map Usage in C++ Applications“. In: *Performance Evaluation: Metrics, Models and Benchmarks*. Hrsg. von Samuel Kounev, Ian Gorton und Kai Sachs. Bd. 5119. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2008, S. 314–321.

[GGV03] Roberto Grossi, Ankur Gupta und Jeffrey Scott Vitter. „High-order entropy-compressed text indexes“. In: *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*. SODA '03. Philadelphia, PA, USA: Society for Industrial und Applied Mathematics, 2003, S. 841–850.

[Jac88] G. J. Jacobson. „Succinct static data structures“. Diss. Pittsburgh, PA, USA, 1988.

[KA03] P. Ko und S. Aluru. „Space efficient linear time construction of suffix arrays“. In: *Journal of Discrete Algorithms*. Springer, 2003, S. 200–210.

[KR08] J. Kärkkäinen und T. Rantala. „Engineering Radix Sort for Strings“. In: *String Processing and Information Retrieval*. Hrsg. von Amihood Amir, Andrew Turpin und Alistair Moffat. Bd. 5280. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 3–14.

[KS03] Juha Kärkkäinen und Peter Sanders. „Simple linear work suffix array construction“. In: *Proceedings of the 30th international conference on Automata, languages and programming*. ICALP'03. Eindhoven, The Netherlands: Springer-Verlag, 2003, S. 943–955.

[MBM93] P. M. McIlroy, K. Bostic und M. D. McIlroy. „Engineering Radix Sort“. In: *COMPUTING SYSTEMS* 6 (1993), S. 5–27.

[MM90] U. Manber und G. Myers. „Suffix arrays: a new method for on-line string searches“. In: *Proceedings of the first Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’90. Philadelphia, PA, USA: Society for Industrial und Applied Mathematics, 1990, 319–327.

[Pet+11] M. Petri u. a. „Backwards Search in Context Bound Text Transformations“. In: *2011 First International Conference on Data Compression, Communications and Processing (CCP)*. IEEE, 2011, S. 82–91.

Name: David Mödinger Matrikelnummer: 691295

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

David Mödinger