technische universität
dortmund

Bachelorarbeit

Effiziente Erstellung von Waveletmatrizen

Benedikt Oesing
157019
8. September 2016

Gutachter:
Prof. Dr. Johannes Fischer
M.Sc. Florian Kurpicz

Technische Universität Dortmund
Fakultät für Informatik
Lehrstuhl Algorithm Engineering
http://ls11-www.cs.tu-dortmund.de

Inhaltsverzeichnis

Kapitel 1

Motivation und Hintergrund

In dieser Arbeit wird die effiziente Erstellung von *Waveletmatrizen* behandelt. Die Waveletmatrix ist eine kompakte Datenstruktur zum Speichern von Zeichenfolgen. Sie erlaubt die Erweiterung der binären Operationen *access*, *rank* und *select* auf beliebige Alphabete. Die access Operation gibt das Bit oder Zeichen an einer bestimmten Stelle zurück.
Mit der rank Operation erhält man die Anzahl eines gesuchten Bits oder Zeichens bis zu einem gegebenen Index.
Und mit der dritten Operation select, kann man die Position bestimmen an der ein Bit oder Zeichen zum gesuchten Mal auftritt.
Neben der Waveletmatrix gibt es noch den *Wavelettree* [12] von dem die Waveletmatrix abgeleitet ist. Auch dieser erlaubt die Durchführung der genannten Operationen auf beliebigen Alphabeten. Beide Datenstrukturen haben die selbe Konstruktionsidee, das betrachtete Alphabet halbieren und die Wörter in die Hälften einordnen. Der Unterschied liegt bei der Waveletmatrix, sie verzichtet auf die Baumstruktur. Damit gilt nicht, anders als bei dem Wavelettree, dass die Kinder eines Knotens, das selbe Intervall des Textes betrachten. Da die Waveletmatrix gegenüber dem Wavelettree die genannten Operationen beschleunigt und es bereits zahlreiche gute Arbeiten zu dem Wavelettree gibt, beschäftigt sich diese Arbeit ausschließlich mit der Konstruktion der Waveletmatrix [9]. Die Waveletmatrix ist auf Grund der genannten Punkte eine relevante Datenstruktur für Algorithmen auf Zeichenketten, weshalb es interessant ist, verschiedene Konstruktionsalgorithmen auf ihre unterschiedlichen Eigenschaften zu untersuchen. Besonderer Schwerpunkt liegt dabei auf dem Speicherverbrauch und der Laufzeit. Im Laufe dieser Arbeit werden deshalb vier unterschiedliche Algorithmen zur Konstruktion einer Waveletmatrix vorgestellt und diese mit Testeingaben aus natürlichen Texten auf Platzverbrauch, sowie Zeitbedarf untersucht. Zwei der vier Konstruktionsalgorithmen sollen dabei mit einem geringen Speicherbedarf auskommen, wogegen die beiden übrigen Algorithmen auf zusätzlichen Speicher zugreifen können. Damit wird der Zusammenhang zwischen der Laufzeit und dem Speicherbedarf untersucht.

Die Ideen der Konstruktion beschränken sich auf ein intuitives Erstellen mit Hilfe von sortieren, der Erstellung der Waveletmatrix durch Anfragen auf den bereits erstellten Teilen der Waveletmatrix und einem Verfahren, welches durch Partitionierung die Waveletmatrix erstellt.
Angestrebt ist eine Implementierung zu finden, welche sich als besonders geeignet herausstellt die genannte Datenstruktur zu erzeugen. Zur Realisierung der Algorithmen wird sich der C++ Bibliothek: *Succinct Data Structure Librabry (SDSL)* [10] bedient.[1] Diese stellt dynamische Bit- und Integervektoren, rank und select Strukturen und eine Implementierung der Waveletmatrix bereit. Die Implementierung wird genutzt um die entwickelten Algorithmen mit ihr zu vergleichen.

[1] https://github.com/simongog/sdsl-lite, besucht 8. September 2016

Kapitel 2

Theoretische Grundlagen

Zur Einführung werden zentrale Begriffe dieser Arbeit geklärt und die Funktionsweise und Methoden einer Waveletmatrix beschrieben.

2.1 Allgemeine Begriffe

Ein Text T der Länge n, besitzt ein Alphabet Σ der Größe σ, definiert über die Menge aller verschiedenen Zeichen a des Textes.
Ein Bitvektor B, besitzt n Einträge aus dem binären Alphabet $\{0, 1\}$.
Eine der Operationen auf Bitvektoren und Waveletmatrizen wird *access(j)* genannt. Die Operation berechnet für einen beliebigen Index $0 \leq j < n$ das Bit an der Stelle j des Bitvektors oder das Zeichen an der Stelle j des Textes T über den die Waveletmatrix definiert ist.
Die *rank(j,a)* Operation auf Bitvektoren und Waveletmatrizen gibt für einen beliebigen Index $0 \leq j < n$ und ein Zeichen $a \in \Sigma$ die Anzahl der Vorkommnisse des Zeichens a im Intervall $B[0, j]$ oder $T[0, j]$ zurück.
Die dritte wichtige Operation auf Bitvektoren und Waveletmatrizen ist *select(r,a)*. Für eine beliebige Zahl $0 < r \leq n$ und ein Zeichen $a \in \Sigma$ berechnet diese Operation das r-te Vorkommen des Zeichens a im Bitvektor B oder dem Text T.
Einen Sortieralgorithmus nennt man *stabil*, wenn Elemente mit gleichem Inhalt ihre relative Position beibehalten.

2.2 Die Waveletmatrix

5	6	4	5	1	6	1	3	2	4	0	7	5
1	1	1	1	0	1	0	0	0	1	0	1	1
1	1	3	2	0	5	6	4	5	6	4	7	5
0	0	1	1	0	0	1	0	0	1	0	1	0
1	1	0	5	4	5	4	5	3	2	6	6	7
1	1	0	1	0	1	0	1	1	0	0	0	1
0	4	4	2	6	6	1	1	5	5	5	3	7

$z_0 = 8$

$z_1 = 5$

$z_2 = 7$

Abbildung 2.1: Waveletmatrix für den Text 5645161324075 über dem Alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$. Grau hinterlegt die Waveletmatrix. Die Zeichen über den Bits, sowie die letzte Zeile dienen nur dem Verständnis und werden nicht gespeichert.

Sei T ein Text aus n Wörtern der Größe w über dem Alphabet Σ der Größe σ, dann gilt für die Einträge $wm_{i,j}$ der Waveletmatrix $\mathrm{WM}_{i,j}$: $wm_{i,j} \in \{0, 1\}$. Für die i Zeilen und j Spalten der Waveletmatrix gelten die Dimensionen $\lceil \lg \sigma \rceil \times n$. Jedes Zeichen aus T wird damit durch $\lceil \lg \sigma \rceil$ Bits in der Matrix dargestellt, wobei in jeder Zeile der Matrix ein Bit pro Zeichen abgespeichert wird.
In der ersten Zeile wird das Bit mit dem höchsten Stellenwert gespeichert. Jede folgende Zeile i, speichert das um i Stellen vom höchsten Stellenwert entfernte Bit. Bits des selben Wortes stehen dabei i.d.R nicht in der selben Spalte, da für jede Zeile die Wörter, welche von der momentan betrachtenden Zeile bis zur letzten Zeile reichen, anhand ihrer Bits stabil sortiert werden.
Eine Beispielmatrix kann man in der Abbildung 2.1 betrachten.
Jede Zeile i besitzt einen Zähler z_i, welcher die Nullbits für die Zeile i speichert. Da die Anzahl der Nullbits für jede Zeile maximal n entspricht, ergibt sich ein Platzbedarf von $O(\lg n \lg \sigma)$ Bits und ist deshalb im Vergleich zum Gesamtbedarf von $O(n \lg \sigma)$ Bits vernachlässigbar.

2.3 Operationen auf der Waveletmatrix

Die Waveletmatrix ermöglicht die effiziente Ausführung der access, rank und select Operationen [9].

2.3.1 Access

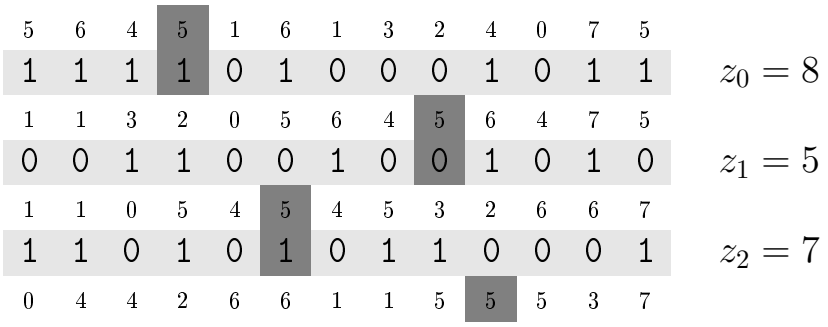


Abbildung 2.2: Ausführung der access Operation für den Index 3 auf der Waveletmatrix $\mathrm{WM}_{i,j}$.

Die Access Operation berechnet für einen Index das dazugehörige Zeichen des Textes. Zum berechnen der Operation, iteriert man über jede Zeile der Matrix. Für jede Zeile wird das Bit an dem aktuellen Index gespeichert und danach für den Index die Nachfolgerposition berechnet. Formal wird der Nachfolger für einen Index durch die Funktion *nachfolgerBit* bestimmt, siehe Gleichung 2.1.
Sei die Waveletmatrix gegeben durch $\mathrm{WM}_{i,j}$, wobei i die momentane Zeile und j die momentane Spalte darstellt und z_i entspricht der Anzahl an Nullbits in der Zeile i, so gilt formal für die Nachfolgeposition:

$$nachfolgerBit(i,j) = \begin{cases} Bitrank(\mathrm{WM}_{i,j}, j, 0) - 1 & \text{if } \mathrm{WM}_{i,j} = 0 \\ Bitrank(\mathrm{WM}_{i,j}, j, 1) - 1 + z_i & \text{if } \mathrm{WM}_{i,j} = 1 \end{cases} \tag{2.1}$$

Auf diese Weise liest der Algorithmus alle Bits des gesuchten Wortes ein. Diese Bits werden dann genutzt um das ursprüngliche Wort zu berechnen, bevor es zurückgegeben wird.
In der Abbildung 2.2 ist die Ausführung von $access(\mathrm{WM}, 3)$ auf der Beispielmatrix, siehe Abbildung 2.1, gezeigt. Die dunkel markierten Stellen, repräsentieren die Positionen die der Algorithmus abläuft.
In der ersten Zeile der Waveletmatrix an der Position 3 wird eine 1 gelesen, dementsprechend wird in der Fallunterscheidung der Funktion nachfolgerBit die untere Berechnungsvorschrift, $Bitrank(\mathrm{WM}_{i,j}, j, 1) - 1 + z_i$, ausgeführt, mit dem Ergebnis 8. Für die darauf folgende Zeile wird an der Position 8 eine 0 gelesen, man verwendet die obere Berechnungsvorschrift, $Bitrank(\mathrm{WM}_{i,j}, j, 0) - 1$ und erhält 5. An dieser Position liest man eine 1, so erhält man für das Beispiel die Bits $1, 0, 1$, welche dem Zeichen 5 entsprechen.

Eingabe: Waveletmatrix WM, $0 \leq j \leq n$
Ausgabe: $T[j]$
  $H \leftarrow Bitvektor(\lceil \lg \sigma \rceil)$
  for $i = 0$ to $\lceil \lg \sigma \rceil - 1$ **do**
    $H[i] = \mathrm{WM}_{i,j}$
    if $\mathrm{WM}_{i,j} = 0$ **then**
      $j_{i+1} \leftarrow Bitrank(\mathrm{WM}_{i,j}, j_i, 0) - 1$
    else
      $j_{i+1} \leftarrow Bitrank(\mathrm{WM}_{i,j}, j_i, 1) - 1 + z_i$
    end if
  end for
  return Char(H)

Algorithmus 2.1: Access($\mathrm{WM}_{i,j}$,j) Operation auf Waveletmatrix

2.3.2 Rank

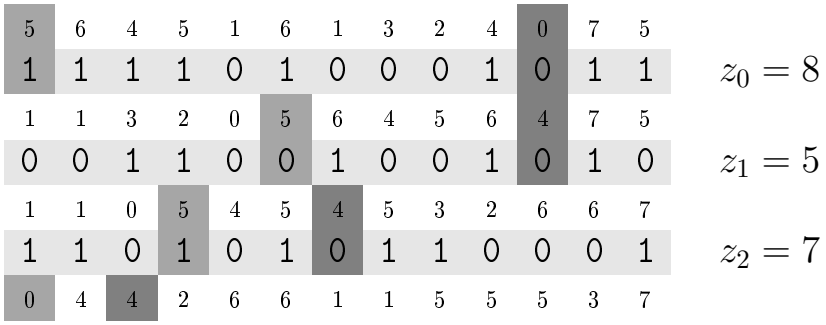


Abbildung 2.3: Ausführung der rank Operation für den Index 10 und dem Zeichen 4 auf der Waveletmatrix $\mathrm{WM}_{i,j}$

Eine weitere Operation auf der Waveletmatrix ist die rank Funktion. Sie zählt auf dem Text T die Anzahl der Vorkommnisse eines Zeichens a bis zu einem Index von j.
Zu Anfang erzeugt man ein Intervall $[0, j]$, wobei jederzeit alle Vorkommen des gesuchten Zeichens a, bis zum gegebenen Index j, in diesem Intervall liegen. Für jede Zeile werden nun die Intervallgrenzen, abhängig davon ob das betrachtete Zeichen a im unteren oder oberen Teil des Alphabets liegt, neu berechnet. Dies hat zur Folge, dass das Betrachtungsintervall in jedem Schritt schrumpft, bis nur noch das gesuchte Zeichen im Intervall vorhanden ist. Formal ergeben sich die neuen Intervallgrenzen durch die Funktion *nachfolgerZeichen*, siehe Gleichung 2.2.
Sei die Matrix gegeben durch $\mathrm{WM}_{i,j}$, wobei i die momentane Zeile und j die momentane

Spalte darstellt, z_i entspricht der Anzahl an Nullen in der Zeile i und das betrachtete Alphabet wird definiert als Intervall $[\alpha, \omega)$ so gilt formal für die Nachfolgeposition:

$$nachfolgerZeichen(i,j) = \begin{cases} Bitrank(\mathrm{WM}_{i,j}, j, 0) - 1 & \text{if } a < 2^{\lceil \lg(\omega-\alpha) \rceil - 1} \\ Bitrank(\mathrm{WM}_{i,j}, j, 0) - 1 + z_i & sonst \end{cases} \tag{2.2}$$

Sieht man sich Abbildung 2.3 an, so erkennt man die untere Intervallgrenze im hellen Grau und die obere Intervallgrenze im dunklen Grau. Für die erste Zeile liegt das betrachtete Zeichen 4 in der oberen Hälfte des Alphabets, weshalb die untere Berechnungsvorschrift, $Bitrank(\mathrm{WM}_{i,j}, j, 0) - 1 + z_i$, genutzt wird. Das Intervall wird von $[0, 10]$ auf $[5, 10]$ verbessert. In der zweiten Zeile dagegen liegt die 4 im unteren Teil des Alphabets und durch die obere Berechnungsvorschrift, $Bitrank(\mathrm{WM}_{i,j}, j, 0) - 1$, ergibt sich $[3, 6]$ für das betrachtete Intervall. Für die letzte Zeile muss wieder die obere Berechnungsvorschrift, $Bitrank(\mathrm{WM}_{i,j}, j, 0) - 1$, angewandt werden, da die 4 im unteren Teil des betrachteten Alphabets liegt und damit erhält man das endgültige Intervall von $[0, 2]$. Die Differenz von 2 und 0 wird als Ergebnis zurückgegeben.
Zur Verbesserung der Laufzeit, kann man für jedes Zeichen $\sigma \in \Sigma$ der Waveletmatrix einen Zeiger speichern, welcher auf die erste Position des Zeichens σ in der Waveletmatrix verweist. Hat man diese Art der Implementierung gewählt, kann auf die Berechnung der unteren Intervallgrenze verzichtet werden, da diese bereits durch die Zeiger bekannt ist. Diese Art der Implementierung führt zu einem erhöhten Speicherbedarf von bis zu $O(n \lg n)$ zusätzlichen Bits.

Eingabe: Waveletmatrix $\mathrm{WM}_{i,j}, a \in \Sigma, 0 \leq j \leq n$
Ausgabe: $\#a \in T[0, j]$

```
p_0 ← 0
for i = 0 to ⌈lg σ⌉ − 1 do
  if a < 2^⌈lg(ω−α)⌉−1 then
    j_{i+1} ← Bitrank(WM_{i,j}, j_i, 0) − 1
    p_{i+1} ← Bitrank(WM_{i,j}, p_i, 0) − 1
  else
    j_{i+1} ← Bitrank(WM_{i,j}, j_i, 1) − 1 + z_i
    p_{i+1} ← Bitrank(WM_{i,j}, p_i, 1) − 1 + z_i
  end if
end for
return j − p
```

Algorithmus 2.2: Rank($\mathrm{WM}_{i,j}$,j,a) Operation auf Waveletmatrix

2.3.3 Select

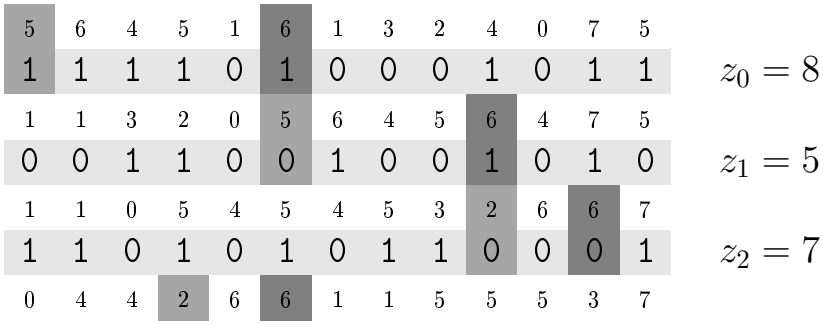


Abbildung 2.4: Ausführung der select Operation für die Anzahl 2 und dem Zeichen 6 auf der Waveletmatrix $\mathrm{WM}_{i,j}$

Die dritte Operation auf der Waveletmatrix ist die select Operation. Diese gibt die Ursprungsposition, des r-ten Vorkommens eines beliebigen Zeichen a, des Textes T zurück. Initial setzt man einen Index auf die Position 0 und wendet auf diesen für jede Zeile der Waveletmatrix die Funktion nachfolgerZeichen, aus Definition 2.2 an. Wie bereits für die rank Operation erhält man damit, die untere Intervallgrenze für das gesuchte Zeichen. Nun wird der Index um r Stellen nach hinten verschoben. Das hat zum Ergebnis, dass der Index auf das r-te Vorkommen des Zeichens a verweist. Um die Ursprungsposition wieder zu finden, wird für jede Zeile der Matrix die Vorgängerposition des Indexes berechnet. Die abschließende Position des Indexes wird als Ergebnis zurück gegeben.
Formal kann der Vorgänger eines Indexes durch die Funktion *vorgaenger*, siehe die Definition der Gleichung 2.3, bestimmt werden.
Sei die Matrix gegeben durch $\mathrm{WM}_{i,j}$, wobei i die momentane Zeile und j die momentane Spalte darstellt, z_i entspricht der Anzahl an Nullen in der Zeile i so gilt formal für die Vorgängerposition:

$$vorgaenger(i,j) = \begin{cases} Bitselect(\mathrm{WM}_{i,j}, j+1, 0) & \text{if } j < z_i \\ Bitselect(\mathrm{WM}_{i,j}, j - z_i + 1, 1) & sonst \end{cases} \tag{2.3}$$

In der Abbildung 2.4 kann man die Ausführung der Operation auf der Beispielmatrix beobachten. Das helle Grau steht für die erste Phase und das dunklere Grau für die zweite und dritte Phase. Für die erste Zeile liegt das gesuchte Zeichen, die 6, in der oberen Hälfte des Alphabets. Damit verwendet man die untere Berechnungsvorschrift, $Bitrank(\mathrm{WM}_{i,j}, j, 0) - 1 + z_i$ und verschiebt den Index von 0 auf 5. Die Nachfolgeposition die sich in der zweiten Zeile ergibt, wird wieder durch die Berechnung von $Bitrank(\mathrm{WM}_{i,j}, j, 0) - 1 + z_i$ hergeleitet und man erhält einen Index auf der Position 9. In der dritten Zeile dagegen liegt das gesuchte Zeichen in der unteren Hälfte des Alphabets, weshalb man die obere Berechnungsvorschrift, $Bitrank(\mathrm{WM}_{i,j}, j, 0) - 1$, verwendet und erhält die Position 3 als untere

Intervallsgrenze. Da das 2-te Vorkommen gesucht ist, summiert man 2 zu der Position 3 und erhält 5. Dies wird durch die dunklere Markierung angedeutet. Die Anzahl der Nullen der letzten Zeile der Waveletmatrix beträgt 5 und damit wird die obere Berechnungsvorschrift, $Bitselect(\mathrm{WM}_{i,j}, j+1, 0)$, angewandt und der Index wechselt von der Position 5 auf 11. In der zweiten Zeile übersteigt die Position des Indexes die Anzahl der Nullbits der Zeile. Damit verwendet man die untere Berechnungsvorschrift, $Bitselect(\mathrm{WM}_{i,j}, j - z_i + 1, 1)$, und erhält 9 als neue Position. Auch in der ersten Zeile muss die untere Berechnungsvorschrift, $Bitselect(\mathrm{WM}_{i,j}, j - z_i + 1, 1)$, angewendet werden und die endgültige Position lautet 5. Die Laufzeit kann, wie bereits bei der rank Operation beschrieben, verbessert werden, indem man die im vorherigen Abschnitt beschriebenen Zeiger erzeugt. Durch die Verwendung der Zeiger kann man den ersten Schritt, das finden der unteren Intervallgrenze, einsparen, da die untere Grenze für jedes Zeichen bekannt ist.

Eingabe: Waveletmatrix $\mathrm{WM}_{i,j}, a \in \Sigma, 0 < r \leq n$
Ausgabe: Die Position des r-ten Vorkommen von a in T

$j_0 \leftarrow 0$
for $i = 0$ to $\lceil \lg \sigma \rceil - 1$ **do**
  if $a < 2^{\lceil \lg(\omega-\alpha) \rceil - 1}$ **then**
    $j_{i+1} \leftarrow Bitrank(\mathrm{WM}_{i,j}, j_i, 0)$
  else
    $j_{i+1} \leftarrow Bitrank(\mathrm{WM}_{i,j}, j_i, 1) + z_i$
  end if
end for
$j \leftarrow j + r$
for $i = i - 1$ to 1 **do**
  if $a < 2^{\lceil \lg(\omega-\alpha) \rceil - 1}$ **then**
    $j_{i-1} \leftarrow Bitselect(\mathrm{WM}_{i,j}, j_i, 0)$
  else
    $j_{i-1} \leftarrow Bitselect(\mathrm{WM}_{i,j}, j_i - z_i, 1)$
  end if
end for
return j

Algorithmus 2.3: Select($\mathrm{WM}_{i,j}$,r,a) Operation auf Waveletmatrix

2.4 Der Wavelettree

5	6	4	5	1	6	1	3	2	4	0	7	5	
1	1	1	1	0	1	0	0	0	1	0	1	1	$z_0 = 8$
1	1	3	2	0	5	6	4	5	6	4	7	5	
0	0	1	1	0	0	1	0	0	1	0	1	0	$z_1 = 5$
1	1	0	3	2	5	4	5	4	5	6	6	7	
1	1	0	1	0	1	0	1	0	1	0	0	1	$z_2 = 7$
0	1	1	2	3	4	4	5	5	5	6	6	7	

Abbildung 2.5: Wavelettree für den Text 5645161324075 über dem Alphabet $\Sigma = \{0, 1, 2, 3, 4, 5, 6, 7\}$. Grau hinterlegt der Wavelettree. Die Zeichen über den Bits, sowie die letzte Zeile dienen nur dem Verständnis und werden nicht gespeichert.

Sei T ein Text der Länge n über dem Alphabet Σ der Größe σ, dann kann man über diesen einen Wavelettree TR mit $2^{\lceil \lg \sigma \rceil}$ vielen Knoten v erzeugen. Jeder Knoten besitzt einem zugehörigen Bitvektor B_v.
Der Wavelettree ist rekursiv definiert für ein Teilalphabet $[\alpha, \omega] \in \Sigma$. Die Abbruchbedingung der Rekursion tritt ein für $\alpha = \omega$, denn dann besteht das Subproblem aus einem Baum mit nur einem Blatt mit dem Inhalt α/ω. Sonst existiert ein Wurzelknoten v, der das Alphabet $[\alpha_v, \omega_v]$ behandelt für den Text $T[0, n]$. Zu diesem Knoten v gehört ein Bitvektor B_v welcher wie folgt definiert ist:

$$B_v[i] = \begin{cases} 0, & \text{if } T[i] \leq (\alpha + \omega)/2 \\ 1, & sonst \end{cases} \tag{2.4}$$

Dieser Wurzelknoten v hat zwei Wavelettrees als Kinder. Das linke Kind behandelt die Teilsequenz $T[0, n_l]$ für alle Zeichen $a \in T, a \leq (\alpha + \omega)/2$ über dem Alphabet $[\alpha \ldots \lfloor(\alpha+\omega)/2\rfloor]$. Und das rechte Kind behandelt die Teilsequenz $T[0, n_r]$ für alle Zeichen $a \in T, a > (\alpha + \omega)/2$ über dem Alphabet $[\lfloor(\alpha + \omega)/2\rfloor + 1 \ldots \omega]$. Einfach gesprochen wird an der Wurzel begonnen und das betrachtete Alphabet halbiert. Das linke Kind kümmert sich dann um die untere Hälfte des Alphabets und das rechte Kind um die obere Hälfte des Alphabets $[\alpha, \omega]$. Es wird ein Bitvektor erstellt und für jedes Zeichen aus T geprüft liegt es in der linken oder rechten Hälfte. Zeichen die im Alphabet des linken Kindes liegen werden mit einer 0 repräsentiert und die restlichen mit einer 1. Die Kinderknoten arbeiten nun auf den Zeichen ihres Alphabets weiter. Der gesamte Text T ergibt sich über alle Knoten einer Ebene. Die Blätter des Wavelettrees erhalten keinen Bitvektor.
Der Wavelettree ist ein balancierter Binärbaum der Höhe $\lceil \lg \sigma \rceil$. Pro Ebene des Baumes werden n Bits gespeichert, was zu einem Platzbedarf von $O(n \lg \sigma)$ führt. Man unterscheidet zwischen dem einfachen Wavelettree und dem zeigerlosen Wavelettree. Für die einfache Variante werden zusätzlich $O(\sigma \lg n)$ Bits benötigt für die Zeiger der Baumstruktur.

Der zeigerlose Wavelettree verzichtet auf die Baumstruktur und verwaltet stattdessen pro Ebene nur einen Bitvektor.
Der Wavelettree ermöglicht es ebenfalls wie die Waveletmatrix die Operationen access, select und rank auf ein allgemeines Alphabet zu erweitern und damit die Operationen auf beliebigen Texten auszuführen.
Die access Operation wird im einfachen Wavelettree sehr ähnlich zu der Waveletmatrix berechnet. Der Unterschied liegt einzig in der Baumstruktur. Statt wie bei der Waveletmatrix den Aufruf über die gesamte nächste Zeile durchzuführen, wird der Aufruf auf den nächsten Kindsknoten reduziert. Die Anzahl der binären rank Anfragen ändert sich nicht.

Eingabe: Wavelettree TR, $0 \leq j \leq n$
Ausgabe: $T[j]$
  $H \leftarrow Bitvector(\lceil \lg \sigma \rceil)$
  $v \leftarrow root(TR)$
  for $i = 0$ to $\lceil \lg \sigma \rceil - 1$ **do**
    $H[i] \leftarrow B_v[j]$
    if $B_v[j] = 0$ **then**
      $j_{i+1} \leftarrow Bitrank(\mathrm{TR}, j_i, 0) - 1$
      $v \leftarrow leftChild(v)$
    else
      $j_{i+1} \leftarrow Bitrank(\mathrm{TR}, j_i, 1) - 1 + z_i$
      $v \leftarrow rightChild(v)$
    end if
  end for

Algorithmus 2.4: Access(TR,j) Operation auf einfachen Waveletree

Die Durchführung der rank Operation auf dem einfachen Wavelettree ist wieder sehr ähnlich zu der Waveletmatrix. Dank der Baumstruktur ist es nicht nötig die Anzahl der Nullen einer Zeile zu kennen, es genügt zu wissen an welcher Position in einem Knoten der Index sich befindet. Dadurch ergibt sich auch, dass die untere Intervallsgrenze für das gesuchte Zeichen a nicht extra berechnet werden muss. Dies spart pro rank Operation einen zusätzlichen binären rank Aufruf.

Eingabe: Wavelettree TR, $a \in \Sigma, 0 \leq j \leq n$
Ausgabe: $\#a \in T[0, j]$
  $v \leftarrow root(TR)$
  for $i = 0$ to $\lceil \lg \sigma \rceil - 1$ **do**
    if $a < 2^{\lceil \lg(\omega - \alpha) \rceil - 1}$ **then**
      $j_{i+1} \leftarrow Bitrank(\mathrm{TR}, j_i, 0) - 1$
      $v \leftarrow leftChild(v)$
    else
      $j_{i+1} \leftarrow Bitrank(\mathrm{TR}, j_i, 1) - 1 + z_i$
      $v \leftarrow rightChild(v)$
    end if
  end for
  return j

Algorithmus 2.5: Rank(TR,j,a) Operation auf einfachen Waveletree

Die Ausführung der letzten Operation select auf dem einfachen Wavelettree profitiert ebenfalls von der Baumstruktur. Der erste Schritt, der bei der Waveletmatrix dafür gesorgt hat, dass die untere Intervallgrenze des Zeichens a berechnet wurde entfällt. Auch ist es nicht nötig die Anzahl der Nullen in einer Zeile zu berücksichtigen, da nur der Bitvektor des momentanen Knotens betrachtet wird. Im Vergleich zur Waveletmatrix spart diese Operation für jeden Aufruf des binären select eine binäre rank Operation.

Eingabe: Wavelettree TR, $a \in \Sigma, 0 < r \leq n$
Ausgabe: Die Position des r-ten Vorkommen von a in T
  $v \leftarrow node(\mathrm{TR}, a)$
  $j \leftarrow r$
  for $i = \lceil \lg \sigma \rceil - 1$ to 1 **do**
    if $a < 2^{\lceil \lg(\omega - \alpha) \rceil - 1}$ **then**
      $j_{i-1} \leftarrow Bitselect(\mathrm{TR}, j_i, 0)$
      $v \leftarrow parent(v)$
    else
      $j_{i-1} \leftarrow Bitselect(\mathrm{TR}, j_i - z_i, 1)$
      $v \leftarrow parent(v)$
    end if
  end for
  return j

Algorithmus 2.6: Select(TR,r,a) Operation auf einfachen Waveletree

Die zeigerlose Variante berechnet die Operationen mit der selben Idee wie der einfache Wavelettree. Da aber nicht eindeutig klar ist, an welcher Stelle die Knoten beginnen und enden, ist es nötig die Intervallgrenzen zu berechen und zu speichern, die ein Knoten einschließen würde. Sei $[b, e]$ das betrachtete Intervall auf T. Inital wäre es $[0, n-1]$. Das Intervall wird für die nächste Ebene berechnet, indem durch eine rank Anfrage die Nullen bis zur Position e bestimmt werden und davon, durch eine zweite rank Operation bestimmt, die Anzahl der Nullen bis b abgezogen werden. Nennen wir diese Differenz m so ist das Intervall für die nächste Zeile entweder $[b, b+m-1]$ für eine gelesene 0 oder $[b+m, e]$ für eine gelesene 1.
Dies hat zur Folge, dass pro Iterationsschritt des einfachen Wavelettrees zwei zusätzliche binäre rank Anfragen berechnet werden müssen für die zeigerlose Variante. Damit erhöht sich die Laufzeit bis auf das Dreifache.

2.5 Countingsort

Das bevorzugte Sortierverfahren dieser Arbeit ist der *Countingsort*. Er findet seine Anwendung in drei der vier Konstruktionsalgorithmen. Countingsort berechnet für eine Eingabe der Größe n über dem Alphabet $\{0..k-1\}$ stabil die Sortierung.
Zuerst wird über die Eingabe iteriert und dabei die Anzahl der Vorkommnisse r_a eines jeden Zeichens $a \in \{0..k-1\}$ gezählt. Aus diesem Schritt errechnet man sich dann für jedes a, die vorderste Position p_a an der das Zeichen in die Ausgabe eingefügt werden kann. Die Position p_0 ist trivial 0. Jede weitere Position ergibt sich aus der Position summiert mit der Anzahl der Vorkommnisse des vorherigen Elementes. So ergibt sich die Prefixsumme $p_a = p_{a-1} + r_{a-1}$. Wobei $a-1$ das vorherige Zeichen darstellt. Hat der Algorithmus die Anfangspositionen alle berechnet, kann jedem Element der Eingabe die korrekte Position in der Ausgabe zugewiesen werden. Der Algorithmus endet damit, dass über die Eingabe iteriert wird und jedes Zeichen a abhängig von der gespeicherten Position p_a in die Ausgabe eingefügt und danach die Position zum nächsten Einfügen auf die nachfolgende Adresse $p_a + 1$ verschoben wird.
Die Zeitkomplexität von Countingsort liegt bei $O(n+k)$. Da der Algorithmus für das Zählen, sowie für das Erstellen der Ausgabe nur je einmalig über die Eingabe der Größe n iterieren muss, ergibt sich für diesen Teil eine Laufzeit in $O(n)$. Zugriffe auf die Datenstruktur, die die Zählung enthält, werden durch eine Bijektion zwischen dem Wert a und der Position in der Datenstruktur in $O(1)$ realisiert. Die Bestimmung der Positionen erfordert $k-1$ Summationen, die in konstanter Zeit ablaufen, es ergibt sich $O(k)$. Die beschriebene Laufzeit wird also erfüllt.
Der Speicherbedarf liegt bei $O(n + k \lg n)$ Bits. Genauer wird ein zusätzlicher Speicher von n Bits als Ausgabe und ein Zähler r_a für die k unterschiedlichen Elemente gebraucht. Die

k Zähler benötigen zusammen bis zu $k \lg n$ Bits.
Da in den vorgestellten Algorithmen ausschließlich Bitvektoren sortiert werden, ergibt sich ein sehr kleines k von 2. Dies macht Countingsort zu einem geeigneten Algorithmus für Bitvektoren.

Kapitel 3

Konstruktionsalgorithmen für die Waveletmatrix

In diesem Kapitel werden die vier Algorithmen vorgestellt. Die ersten beiden Varianten versuchen einen möglichst geringen Speicherbedarf zu erhalten und arbeiten deshalb *inplace*. Die letzteren beiden Varianten versuchen dagegen eine schnelle Konstruktion zu ermöglichen und erhalten dafür zusätzlichen Speicher.
Variante eins geht dabei intuitiv vor und sortiert die Waveletmatrix nach jedem Extraktionsschritt, um die relevanten Bits für den folgenden Schritt in die korrekte Reihenfolge zu bringen. Diese Variante wird gekürzt *Konstruktion mit n extra Bits und Sortierung* genannt.
Der zweite Konstruktionsalgorithmus, verzichtet auf eine Sortierung und stattdessen bestimmt er für jedes Bit, durch select Anfragen auf den bereits erstellten Zeilen der Waveletmatrix, die korrekte Position in der aktuellen Zeile. Die Variante trägt den Namen *Konstruktion mit n extra Bits und Select Anfragen*.
Die dritte Variante hat den selben Ansatz wie Methode eins, mit dem Unterschied, dass sie zusätzlichen Platz verwendet, um die Sortierung und die Konstruktion zu vereinfachen. Deshalb nennt man sie *Konstruktion mit* $n \lg \sigma$ *extra Bits und Sortierung*.
Die letzte Variante nutzt ihren zusätzlichen Speicher, um die Positionen vor zu berechnen, an denen die einzelnen Wörter liegen werden. Im folgenden wird sie *Konstruktion mit* $n \lg \sigma$ *extra Bits und Partitionierung* genannt.
Zusätzlich zu dem angegebenen Speicherbedarf wird extra Platz für den rank und select Support auf der fertigen Waveletmatrix benötigt. Es werden zwei Support Strukturen für die select Operation benötigt und zusätzlich ein Support für die rank Operation. Der Speicherbedarf eines select Supports liegt praktisch bei $\leq 0,2n$ Bits und $0,0625n$ Bits für den rank Support.[1] Beide Strukturen werden in linearer Zeit erzeugt.

[1] `http://simongog.github.io/assets/data/sdsl-cheatsheet.pdf`, besucht 8. September 2016

3.1 Konstruktion mit n extra Bits und Sortierung

a_0	a_1	a_2	b_0	b_1	b_2	c_0	c_1	c_2	d_0	d_1	d_2	e_0	e_1	e_2
a_2	b_2	c_2	d_2	e_2	a_0	a_1	b_0	b_1	c_0	c_1	d_0	d_1	e_0	e_1

Abbildung 3.1: Verschieben der Wörter für die Konstruktion. Die Eingabe ist im *little endian* gegeben. Es wird davon ausgegangen, dass die höchstelligen Bits a_2, b_2, c_2, d_2, e_2 in sortierter Reihenfolge vorliegen.

Der Konstruktionsalgorithmus berechnet pro Iterationsschritt eine Zeile der Waveletmatrix. Das Erstellen einer Zeile beginnt damit, die höchstelligen Bits der Wörter in den Hilfsvektor zu kopieren, während die um das höchstellige Bit gekürzten Wörter nach hinten verschoben werden. Dies hat den Vorteil, dass die kopierten Bits überschrieben werden können, ohne diese zu verlieren. Auf diese Weise entsteht in der Waveletmatrix ein freier Bereich von n Bits, zwischen den bereits berechneten Zeilen der Waveletmatrix und den noch zu bearbeitenden Wörtern. In diese Lücke schreibt der Algorithmus die Bits aus dem Hilfsvektor. Die so eingefügten Bits entsprechen der letztendlichen Zeile der Waveletmatrix. Den Kopiervorgang kann man beispielhaft in der Abbildung 3.1 sehen. Abgeschlossen wird ein Iterationsschritt, indem die eben eingefügten Bits genutzt werden um die Wörter der Eingabe stabil zu sortieren.
Der Sortierschritt baut auf den *Bitmergesort* von *Tischler* [18] auf. Dieser Algorithmus verfolgt einen Mergesort Ansatz, welcher eine stabile Sortierung ohne zusätzlichen Speicherbedarf in der Zeit $O(n \lg n)$ ermöglicht. Dies erlaubt der Algorithmus durch die Beobachtung, dass zwei sortierte Bitvektoren X und Y stabil zu dem ebenfalls sortierten Bitvektor $|Z| = |X| + |Y|$ zusammengefasst werden können in Zeit $O(|X| + |Y|) = O(|Z|)$ [1]. Ein sortierter Bitvektor, erfüllt die Regel, dass alle Nullbits links von den Einsbits stehen.
Zur Verbesserung der Laufzeit ist die Abbruchbedingung des Mergesort so gewählt, dass sobald die Anzahl der Bits der betrachteten Wörter n entsprechen oder n unterschreiten, ein Countingsort auf diese Wörter angewendet wird. Siehe Kapitel 2.5 für die ausführliche Erklärung des Countingsort. Die Bedingung ist so gewählt, da man dann den Hilfsvektor als Ausgabespeicher für die Sortierung verwenden kann.

3.1.1 Lemma. *Der Algorithmus „Konstruktion mit n extra Bits und Sortierung“ berechnet die Waveletmatrix mit einer Laufzeit von* $O(\lg \sigma(n + n \lg \sigma + n \lg \sigma \lg \lg \sigma))$ *und einem Speicherbedarf von* $O(n \lg \sigma + n + \lg(n \lg \sigma))$ *Bits.*

Die Laufzeit der Konstruktion ergibt sich aus der $\lceil \lg \sigma \rceil$-fachen Durchführung der Schleife, die pro Iterationsschritt eine Zeile der Waveletmatrix erzeugt. Jeder Schleifendurchlauf, setzt sich aus einem Verschiebevorgang und einem Sortiervorgang zusammen. Für das Verschieben der Bits in den Hilfsvektor braucht man n Lesezugriffe und genau soviele Schreib-

zugriffe. Diese geschehen in konstanter Zeit. Das zurückschreiben vom Hilfsvektor in die Waveletmatrix erfordert exakt den selben Aufwand, nämlich $2n$ Zugriffe. Zusammen ergibt sich für die Kopiervorgänge von der Waveletmatrix in den Hilfsvektor und zurück ein Zeitaufwand von $4n$. Hinzu kommt das Verschieben der Wörter um Platz in der Waveletmatrix zu erhalten. Es ergibt sich ein Zeitbedarf von $2n\lceil \lg \sigma \rceil$. Jedes der n Wörter, der Größe von $\lceil \lg \sigma \rceil$ Bits, muss eingelesen werden und an eine neue Position geschrieben werden. Die Laufzeit für den gesammten Verschiebevorgang beträgt dementsprechend $4n + 2n\lceil \lg \sigma \rceil$. Für die Laufzeit des Sortierens, wird der Mergesort Ansatz betrachtet. Die theoretische Laufzeit des verwendeten Mergesort ist $O(\lg \sigma(n + k) + n \lg \sigma \lg \lg \sigma)$ [18]. So ergibt sich eine gesamte Laufzeit für die Schleife von $O((n+n \lg \sigma+n \lg \sigma \lg \lg \sigma))$. Da diese nun $\lceil \lg \sigma \rceil$ oft ausgeführt wird erhält man die obere Laufzeit.
Der Speicherbedarf des Algorithmus setzt sich zusammen aus $n\lceil \lg \sigma \rceil$ Bits für die Eingabe und n weiteren Bits für den Hilfsvektor. Zusätzlich verwendet man bei der Ausführung des Countingsort einen Zähler für die Anzahl der Nullbits, welcher bis zu $\lceil \lg(n \lg \sigma) \rceil$ Bits groß ist.

Eingabe: Bitvektor B, $|B| = n\lceil \lg \sigma \rceil$
Ausgabe: Waveletmatrix WM

1: $H \leftarrow Bitvektor(n)$
2: $w_0 \leftarrow \lceil \lg \sigma \rceil$
3: $w \leftarrow w_0$
4: $p_B \leftarrow w - 1$
5: $off \leftarrow 0$
6: **for** $i = 0$ to $w_0 - 2$ **do**
7: **for** $j = 0$ to $n - 1$ **do**
8: $H[j] \leftarrow B[p_B]$
9: $p_B \leftarrow p_B + w$
10: **end for**
11: $p_B \leftarrow off + ((n-1)w)$
12: $p_{Bnew} \leftarrow p_B + 1$
13: **while** $p_B \geq off$ **do**
14: $B[p_{Bnew}..p_{Bnew} + w - 2] \leftarrow B[p_B..p_B + w - 1]$
15: $p_B \leftarrow p_B - w$
16: $p_{Bnew} \leftarrow p_{Bnew} - (w - 1)$
17: **end while**
18: $B[off..off + n - 1] \leftarrow H[0..n - 1]$
19: $w \leftarrow w - 1$
20: $off \leftarrow off + n$
21: $Sort(B[off..off + n - 1])$
22: **end for**
23: **return** B

Algorithmus 3.1: Konstruktion mit n extra Bits und Sortierung

3.2 Konstruktion mit n extra Bits und Select Anfragen

Die zweite Variante eines Konstruktionsalgorithmus arbeitet ebenfalls mit einem Hilfsvektor von n zusätzlichen Bits. Auch dieser Algorithmus erzeugt in jedem Iterationsschritt eine Zeile der Waveletmatrix. Identisch zu dem ersten Algorithmus werden die höchstelligen Bits der Wörter in korrekter Reihenfolge in den Hilfsvektor kopiert und die gekürzten Wörter wie in Abbildung 3.1 zu sehen, nach hinten geschoben. Auch das füllen der so entstandenen Lücke, zwischen den erfolgreich erstellten Zeilen der Waveletmatrix und den noch zu bearbeitenden Wörtern, funktioniert identisch zum ersten Algorithmus. Der signifikante Unterschied liegt nun darin, wie die höchstelligen Bits in korrekter Reihenfolge in den Hilfsvektor eingefügt werden. Da anders als in der ersten Variante auf das Sortieren

der Teilwörter verzichtet wird, muss jede Zeile mit Hilfe von select Anfragen auf den vorhergehenden Zeilen erzeugt werden. Um den Hilfsvektor für eine Zeile zu füllen, wird für alle Einträge des Hilfsvektors $j \in \{0..n-1\}$, solange die Vorgängerposition berechnet, bis die Berechnung in der ersten Zeile stoppt. Hierfür wird wieder die Funktion $vorgnger(i, j)$, siehe Funktionsbeschreibung 2.3 verwendet. Sei x die resultierende Position eines solchen Aufrufes, so wird in den Hilfsvektor an der Stelle j das höchstellige Bit des x-ten Wortes eingefügt. Um das Berechnen der Vorgängerfunktion zu ermöglichen, wird nach jedem erfolgreichen Einfügen einer Zeile in die Waveletmatrix, die Anzahl der Nullbits in der gerade eingefügten Zeile berechnet und gespeichert und außerdem die Hilfsstruktur für die select Anfragen erzeugt.
Die SDSL bietet eine Implementierung für einen select Support für Bitvektoren an, welcher in konstanter Zeit select Anfragen auswertet. Dieser besitzt einen zusätzlichen Speicherplatzbedarf von $\frac{3n}{\lceil \lg \lg n \rceil} + O(n^{\frac{1}{2}} \lg n \lg \lg n)$ Bits [3]. Die Konstruktion findet in linearer Zeit statt.

3.2.1 Lemma. *Der Algorithmus „Konstruktion mit n extra Bits und Select Anfragen“ berechnet die Waveletmatrix mit einer Laufzeit von $O(n \lg^2 \sigma + n \lg \sigma)$ und einem Speicherbedarf von $O(n \lg \sigma + n)$ Bits.*

Die Laufzeit ergibt sich wieder durch die $\lceil \lg \sigma \rceil$-fache Wiederholung einer Schleife. In jedem Schritt der Schleife wird jeweils eine Zeile der Waveletmatrix erzeugt. Die Laufzeit für das Verschieben der Wörter ist bereits durch die Analyse des vorherigen Algorithmus bekannt. Der gesamte Verschiebevorgang hat eine Laufzeit von $4n + 2n\lceil \lg \sigma \rceil$. Zusätzlich muss in jedem Schleifendurchlauf die korrekte Reihenfolge der höchstelligen Bits berechnet werden. Diese Berechnung setzt sich aus n Anfragen zusammen, welche aus $\lceil \lg \sigma \rceil$ vielen select Anfragen besteht. Eine select Anfrage kann in konstanter Zeit abgewickelt werden, damit ergibt sich für diesen Teil der Operation eine Laufzeit von $O(n \lg \sigma)$. Zusammen erhält man eine Laufzeit pro Schleifendurchlauf von $O(n + n \lg \sigma)$, diese wird $\lceil \lg \sigma \rceil$ oft ausgeführt und damit erhält man die zu Anfang genannte Laufzeit.
Der Speicherbedarf des Algorithmus setzt sich zusammen aus der Eingabegröße von $n\lceil \lg \sigma \rceil$ und dem Speicherbedarf des Hilfsvektors von zusätzlich n Bits.
Besonders bei der beschriebenen Implementierung ist, die SDSL Klassen für die select Anfragen wurden erweitert. Vor der Anpassung wurde ein *Select Support* der SDSL immer über die gesamte Länge eines Bitvektors aufgebaut. Dies ist nicht nur sehr unflexibel, sondern führt zu Situationen, in denen unnötiger Speicher oder Laufzeit für die Konstruktion verschwendet wird. Angenommen man möchte einen select Support über die letzten n Bits eines BItvektors erzeugen. Vor der Anpassung wäre dies nicht möglich gewesen und wir hätten einen Support über den gesamten Bitvektor erhalten. Die erweiterte Version der Select Support Klassen, ermöglichen es für einen Bitvektor einen Select Support zu erzeugen,

der an einem beliebigen Offset beginnt und eine beliebige Länge besitzt. Auf diese Weise wird garantiert, dass nur relevante Bits einbezogen werden, was die Laufzeit verbessert und den Speicherbedarf schont. Selbiges wurde auch für den *Rank Support* implementiert, findet aber hier keine Anwendung.

Eingabe: Bitvektor B, $|B| = n \cdot \lceil \lg \sigma \rceil$
Ausgabe: Waveletmatrix WM

1: $H \leftarrow Bitvektor(n)$
2: $w_0 \leftarrow \lceil \lg \sigma \rceil$
3: $Zero \leftarrow Array(w_0)$
4: $Select_0 \leftarrow Array(w_0)$
5: $Select_1 \leftarrow Array(w_0)$
6: $w \leftarrow w_0$
7: $p_B \leftarrow off + w - 1$
8: $off \leftarrow 0$
9: **for** $i = 0$ to $w_0 - 1$ **do**
10: **for** $j = 0$ to $n - 1$ **do**
11: $level \leftarrow i$
12: **for** $level = i$ to 1 **do**
13: **if** $vp < Zero[level - 1]$ **then**
14: $vp \leftarrow Select_0[level - 1].select(vp + 1)$
15: **else**
16: $vp \leftarrow Select_1[level - 1].select(vp + 1 - Zero[level - 1])$
17: **end if**
18: $level \leftarrow level - 1$
19: **end for**
20: $H[j] \leftarrow B[p_B + vp \cdot w]$
21: **end for**
22: $p_B \leftarrow off + ((n - 1) \cdot w)$
23: $p_{Bnew} \leftarrow p_B + 1$
24: **while** $p_B \neq off$ **do**
25: $B[p_{Bnew}..p_{Bnew} + w - 2] \leftarrow B[p_B..p_B + w - 1]$
26: $p_B \leftarrow p_B - w$
27: $p_{Bnew} \leftarrow p_{Bnew} - (w - 1)$
28: **end while**
29: $B[off..off + n - 1] \leftarrow H[0..n - 1]$
30: $w \leftarrow w - 1$
31: $off \leftarrow off + n$
32: $Zero \leftarrow Count_0(H)$
33: $Select_0[i] \leftarrow SelectSup_0(H, j * n, n)$
34: $Select_1[i] \leftarrow SelectSup_1(H, j * n, n)$
35: **end for**
36: **return** B

Algorithmus 3.2: Konstruktion mit n extra Bits und Select Anfragen

3.3 Konstruktion mit $O(n \lg \sigma)$ extra Bits und Sortierung

Dieser Konstruktionsalgorithmus arbeitet mit der selben Idee wie die Variante Konstruktion mit n extra Bits und Sortierung. Der Unterschied zwischen den beiden Algorithmen liegt dabei in der Größe des zugewiesenen Speichers. Dieser Algorithmus erhält einen Hilfsvektor von $n\lceil \lg \sigma \rceil$ Bits, also der Größe der Eingabe. Genutzt wird der zusätzliche Speicher um auf das Verschieben der höchstelligen Bits und das kürzen der Teilwörter zu verzichten. Konkret bedeutet es, dass die Eingabe genutzt wird um die momentane Sortierung der Wörter darzustellen und der Hilfsvektor wird verwendet um die Wavletmatrix zu generieren.
Pro Iterationsschritt konstruiert der Algorithmus eine Zeile der Waveletmatrix. Die Zeile i der Matrix wird nun nach dem $i-1$-ten Sortiervorgang erstellt, indem im i-ten Iterationsschritt die i-ten Bits aus den Wörtern gelesen werden und in den Hilfsvektor eingefügt werden. Danach werden die Teilwörter nach den Bits der gerade eingefügen Zeile stabil sortiert. Zur Sortierung wird der aus dem Kapitel 2.5 bekannte Countingsort verwendet.

3.3.1 Lemma. *Der Algorithmus „Konstruktion mit $O(n \lg \sigma)$ extra Bits und Sortierung “ berechnet die Waveletmatrix mit einer Laufzeit von $O(n \lg \sigma + \lg \sigma)$ und einem Speicherbedarf von $O(n \lg \sigma + \lg(n \lg \sigma))$.*

Da man auf das Verschieben, wie in den ersten beiden Algorithmen, verzichten kann, ist die Laufzeit nur noch abhängig vom Sortierschritt und dem kopieren von Bits aus der sortierten Waveletmatrix in den Hilfsvektor. Countingsort hat eine Laufzeit von $O(n+k)$. Für den Algorithmus ist es ausreichend die Wörter nach einzelnen Bits zu sortieren, deshalb gilt $k = 2$. Zusätzlich kommen jeweils n Lesezugriffe und n Schreibzugriffe hinzu. Für das erstellen einer Zeile der Waveletmatrix ergibt sich damit eine Laufzeit von $O(3n + 2)$. Da insgesamt $\lceil \lg \sigma \rceil$ Zeilen in der Matrix erstellt werden müssen ergibt sich die am Anfang genannte Laufzeit.
Der Speicherbedarf ergibt sich aus $n\lceil \lg \sigma \rceil$ Bits für die Eingabe, einem Hilfsvektor und einem Ausgabevektor, beide in der Größe der Eingabe. Der Ausgabevektor wird für den Countingsort benötigt. Zuletzt wird ein Zähler benötigt um die Anzahl der Nullbits zu speichern. Dieser Zähler hat einen maximalen Speicherbedarf von $\lceil \lg(n \lg \sigma) \rceil$ Bits.

Eingabe: Bitvektor B, $|B| = n \cdot \lceil \lg \sigma \rceil$
Ausgabe: Waveletmatrix WM

```
1: H ← Bitvektor(n⌈lg σ⌉)
2: w_0 ← ⌈lg σ⌉
3: off ← 0
4: for i = 0 to w_0 − 2 do
5:    p_B ← w_0 − i − 1
6:    for j = 0 to n − 1 do
7:       H[off + j] ← S[p_B]
8:       p_B ← p_B + w_0
9:    end for
10:   off ← off + n
11:   Sort(B[off..off + n − 1]
12: end for
13: return H
```

Algorithmus 3.3: Konstruktion mit $\operatorname{nlg} \sigma$ extra Bits und Sortierung

3.4 Konstruktion mit $O(n \lg \sigma)$ extra Bits und Partitionierung

Der letzte Konstruktionsalgorithmus, basiert auf dem Countingsort für allgemeine Alphabete. Der Algorithmus verwendet einen zusätzlichen Speicher in der Größe der Eingabe, in dem die Waveletmatrix konstruiert wird. Dieser Hilfsvektor hat damit eine Größe von $n\lceil \lg \sigma \rceil$ Bits. Anders als im dritten Konstruktionsalgorithmus bleibt die Eingabe komplett unverändert, sie wird nur für Lesezugriffe benötigt.
Pro Iterationsschritt wird wieder eine Zeile der Waveletmatrix konstruiert. Für den Countingsort werden die Wörter als Dezimalzahlen interpretiert. Dabei werden im i-ten Schritt der Konstruktion, die i höchsten Bits eines Wortes aus der Eingabe gelesen und als Dezimalzahl interpretiert. Durch die Struktur der Waveletmatrix ergibts sich dann zum Beispiel für die zweite Zeile einer Matrix die Folge $\{0, 2, 1, 3\}$ oder binär $\{00, 10, 01, 11\}$. Um nun den Countingsort effektiv nutzen zu können muss eine Bijektion berechnet werden, zwischen den dezimalen Schlüsseln und ihrer Position in der Zeile der Waveletmatrix. Für das Beispiel würde dies bedeuten, 0 wird auf die 0 abgebildet, 2 wird auf die 1 abgebildet, 1 wird auf 2 abgebildet und schließlich 3 auf 3. Initial sieht die Bijektion wie folgt aus: $\{0, 1\}$, dabei bilden beide Dezimalzahlen auf sich selbst ab. Eine Zeile der Waveletmatrix wird erstellt, indem unter Zuhilfenahme der Bijektion der Countingsort angewendet wird auf die als Dezimalzahl interpretierten Wörter. Die resultierende Sortierung wird dann genutzt um das i-te Bit eines jeden Wortes in den Hilfsvektor zu kopieren. Abgeschlossen wird ein Iterationsschritt, indem die Bijektion für die kommende Zeile neu berechnet wird.

Sei Bi die Bijektion und b_x das x-te Element dieser, so wird Bi für die kommende Zeile berechnet, indem für alle b_x die Menge $\{b_0 \ll 1, b_1 \ll 1, \dots b_{2|Bi|-1} \ll 1\}$ vereint wird mit der Menge $\{1 + b_0 \ll 1, 1 + b_1 \ll 1, \dots 1 + b_{2|Bi|-1} \ll 1\}$.

3.4.1 Lemma. *Der Algorithmus „Konstruktion mit n extra Bits und Partitionierung“ berechnet die Waveletmatrix mit einer Laufzeit von* $O(n + \sigma + \frac{2^{1+\lceil \lg \sigma \rceil}}{\lg \sigma}$ *und einem Speicherbedarf von* $O(n \lg \sigma + \sigma \lg \sigma + \sigma \lg n)$ *Bits.*

Die Laufzeit ergibt sich durch die $\lceil \lg \sigma \rceil$-fache Ausführung der Schleife, welche die Zeilen der Matrix konstruiert. In dieser Schleife muss die Bijektion berechnet werden, der eigentliche Countingsort ausgeführt werden und zuletzt die Bits in den Hilfsvektor kopiert werden. Die Berechnung der Bijektion hat eine Laufzeit abhängig vom aktuellen Schritt i. Für den i-ten Schritt müssen 2^i Elemente berechnet werden, wobei durch die Initialisierung der erste Schritt bereits gegeben ist. Im Maximum werden aber höchstens $2^{\lceil \lg \sigma \rceil}$ viele Elemente berechnet. Dieser Teil hat eine Laufzeit von $O(\frac{2^{1+\lceil \lg \sigma \rceil}}{\lg \sigma})$. Der Countingsort hat wie bekannt eine Laufzeit von $O(n + \sigma)$. Zuletzt benötigt das einlesen und kopieren der Bits in den Hilfsvektor $2n$ Operationen.
Der angegebene Speicherbedarf ergibt sich durch die Größe der Eingabe von $n\lceil \lg \sigma \rceil$ Bits, dem Hilfsvektor und einem Ausgabevektor beide in der Größe der Eingabe, der Bijektion der Größe $\sigma\lceil \lg \sigma \rceil$ und der Datenstruktur für das Zählen der Wörter mit $\sigma\lceil \lg n \rceil$ Bits.
Theoretisch ist es möglich die Bijektion auch direkt zu berechnen. Eine Formel für beliebige n lautet: "Write n in binary, reverse bits, subtract 1, divide by 2"(Schreibe die Zahl n im Binärsystem, kehre die Reihenfolge der Bits um, subtrahiere mit 1 und halbiere die Zahl), wie in der On-Line Encyclopedia of Integer Sequences beschrieben. [2] Da der Algorithmus aber von der vierten Zahl der Folge an alle Elemente ununterbrochen verwendet, wird keine Berechnung unnötig durchgeführt. Speichert man die Elemente der Folge zwischen, so genügt eine Bitshift Operation und eine Addition für die Berechnung. Die gerade genannte Formel verwendet zusätzlich einen Schritt zum umkehren der Bits. Die implementierte Variante ist deshalb zu bevorzugen.

[2] `https://oeis.org/A030109`, besucht 8. September 2016

Eingabe: Bitvektor B, $|B| = n \cdot \lceil \lg \sigma \rceil$
Ausgabe: Waveletmatrix WM

$H \leftarrow Bitvektor(n\lceil \lg \sigma \rceil)$
$Prefixes \leftarrow List(0, 1)$
$w \leftarrow \lceil \lg \sigma \rceil$
for $j = 0$ to $n - 1$ **do**
  $H[j] \leftarrow B[(j + 1) * w - 1]$
end for
for $i = 0$ to $w - 2$ **do**
  if $i \neq 0$ **then**
    $Prefixes \leftarrow NextPrefixes$
  end if
  $Translator \leftarrow Array(|Prefixes|)$
  for $j = 0$ to $|Prefixes| - 1$ **do**
    $Translator[Prefixes[j]] \leftarrow i$
  end for
  $Positions \leftarrow Array(|Prefixes|)$
  for $j = 0$ to $n - 1$ **do**
    $word \leftarrow B[(j + 1) * w - 1 - i \ldots (j + 1) * w - 1]$
    $Positions[Translator[word] + 1] \leftarrow Positions[Translator[word] + 1] + 1$
  end for
  for $j = 1$ to $|Prefixes| - 1$ **do**
    $Positions[j] \leftarrow Positions[j] + Positions[j - 1]$
  end for
  for $j = 0$ to $n - 1$ **do**
    $word \leftarrow B[(j + 1) * w - 1 - i \ldots (j + 1) * w - 1$
    $wordH \leftarrow B[(j + 1) * w - 1 - (i + 1)]$
    $H[(i + 1) * n + Positions[Translator[word]]] \leftarrow wordH$
  end for
end for
return H

Algorithmus 3.4: Konstruktion mit $\text{nlg}\,\sigma$ extra Bits und Partitionierung

3.5 Konstruktionsalgorithmus der SDSL

Da die Algorithmen mit der SDSL verglichen werden, wird der Algorithmus der Bibliothek einmal vorgestellt.

Die SDSL verwendet für ihren Konstruktionsalgorithmus [3] einen zusätzlichen Hilfsvektor mit einem Platzbedarf von $n\lceil \lg \sigma \rceil$ Bits. Außerdem noch eine dynamische Datenstruktur von bis zu $n\lceil \lg \sigma \rceil$ Bits zum Zwischenspeichern von Wörtern und einen Integer, in dem die Waveletmatrix konstruiert wird.
Pro Schritt der Konstruktion wird eine Zeile der Waveletmatrix erzeugt. Der Schritt i beginnt damit, dass eine Maske erzeugt wird mit genau einem Einsbit an der Position des i-t höchsten Bits, mit dessen Hilfe das zu betrachtende Bit isoliert werden kann. Nun wird durch Verundung mit der Maske für jedes Wort entschieden, ob das isolierte Bit eine 1 oder eine 0 darstellt. Handelt es sich um ein Einsbit, so wird das dazugehörige Wort zwischengespeichert und der Integer durch Veroderung so manipuliert, dass an der korrekten Position eine 1 auftaucht. Wird stattdessen ein Nullbit gelesen, so wird das entsprechende Wort hinter alle bislang gelesenen Wörter geschrieben, die ebenfalls ein Nullbit enthielten. Dafür muss der Algorithmus die Anzahl der gelesenen Nullbits abspeichern. Ein Iterationsschritt wird beendet, indem die zwischengespeicherten Wörter, in der Reihenfolge in der sie gelesen wurden, hinter die bereits korrekt einsortieren Wörter, zurück in den Hilfsvektor geschrieben werden.
Auffällig bei der Implementierung, jedes 64 Bit Wort der Waveletmatrix wird aus dem RAM auf die Festplatte geschrieben und erst bei Vervollständigung der Waveletmatrix wird die gesamte Matrix von der Festplatte geladen und initialisiert.
Die Laufzeit der Konstruktion ist $O(n \lg \sigma)$.[4] Dies lässt sich nachvollziehen, wenn man überdenkt, dass der Algorithmus für jede der $\lceil \lg \sigma \rceil$ Zeilen, alle n Wörter betrachtet und jedes Wort maximal zweimal kopiert wird.
Der theoretische Speicherbedarf ist $n \lg \sigma + O(1)$ Bits. In der Praxis siehe Kapitel 4 hat sich gezeigt, dass dieser Speicherbedarf realistisch ist.

[3] `http://algo2.iti.kit.edu/gog/docs/html/wm__int_8hpp_source.html` Zeile 147f.,besucht 8. September 2016

[4] `http://algo2.iti.kit.edu/gog/docs/html/classsdsl_1_1wm__int.html#abb6318d3eb320199b7616c8b1e57f46c`, besucht 8. September 2016

Kapitel 4

Experimente

Zum testen der realen Laufzeit und dem Speicherbedarf der vier vorgestellten Algorithmen, konstruiert man für jeden Algorithmus die Waveletmatrix über Texte mit verschiedenen Alphabetgrößen und misst die Laufzeit sowie den Speicherbedarf. Für die Messung der Laufzeit wurde jede Konstruktion fünf Mal wiederholt und aus allen Messergebnissen das arithmetische Mittel gebildet.
Die Durchführung der Experimente wurde auf einem Computer mit dem Betriebssystem GNU/Linux, Debian 3.16.36 durchgeführt. Der Prozessor ist ein Intel Core i5-4590 CPU mit 3700MHz. Der Computer verfügt über 8 GB RAM. Die Implementierung der Algorithmen wurde in *C++* durchgeführt mit dem Compiler *g++ 4.9.2* und -03 Optimierung. Alle Experimente verwenden einen einzigen Prozess.

4.1 Testeingaben

Diese Arbeit bedient sich der Texte die vom *Pizza&Chili Corpus* bereitgestellt werden.[1]
Die Texte gliedern sich in sechs Kategorien. Konkret sind das: *DNA, Englisch, Programmcode, Proteine, Tonhöhen* und zuletzt *XML*.
Wie in Tabelle 4.1 zu sehen ist, variieren die Alphabetgrößen von 16 Zeichen für den Text der DNA bis zu einer Alphabetgröße von 239 Zeichen für den englischen Text. Getestet werden die Textgrößen von 20 MB über 50 MB, 100 MB, 200 MB, bis 400 MB. Nicht bereit gestellte Texte werden erzeugt, indem die Texte des Corpus gekürzt werden oder mehrere Texte zusammengefasst werden. Diese Vielfalt erlaubt es, die Effizienz der Algorithmen abhängig von der Art und Größe der Eingabe zu prüfen.

[1] `http://pizzachili.dcc.uchile.cl/texts.html`, besucht 8. September 2016

Texte	DNA	Englisch	Programmcode	Proteine	Tonhöhen	XML
Alphabetgröße	16	239	230	27	130	97

Tabelle 4.1: Übersicht der Alphabetgrößen

4.2 Konstruktionsdauer

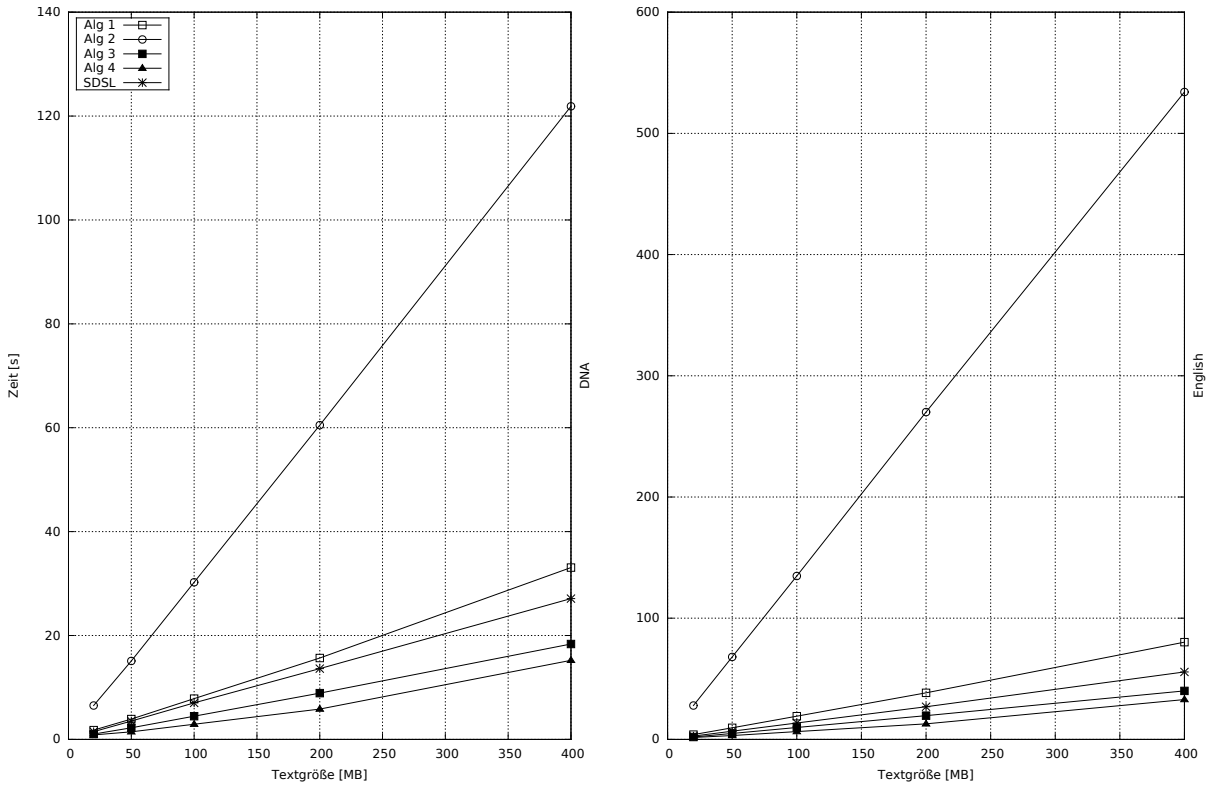


Abbildung 4.1: Laufzeit der vorgestellten Algorithmen für Eingaben der Art DNA und englischem Text. Die Algorithmen sind für die Legende gekürzt. Alg 1 steht für die „Konstruktion mit n Bits und Sortierung“. Alg 2 ist der Algorithmus„Konstruktion mit n Bits und Select Anfragen“. Der Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Sortierung“ wird mit Alg 3 abgekürzt. Der letzte entworfene Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Partitionierung“ wird Alg 4 genannt. Und zulezt der Konstruktionsalgorithmus der SDSL, welcher mit dem Namen der Bibliothek abgekürzt wird.

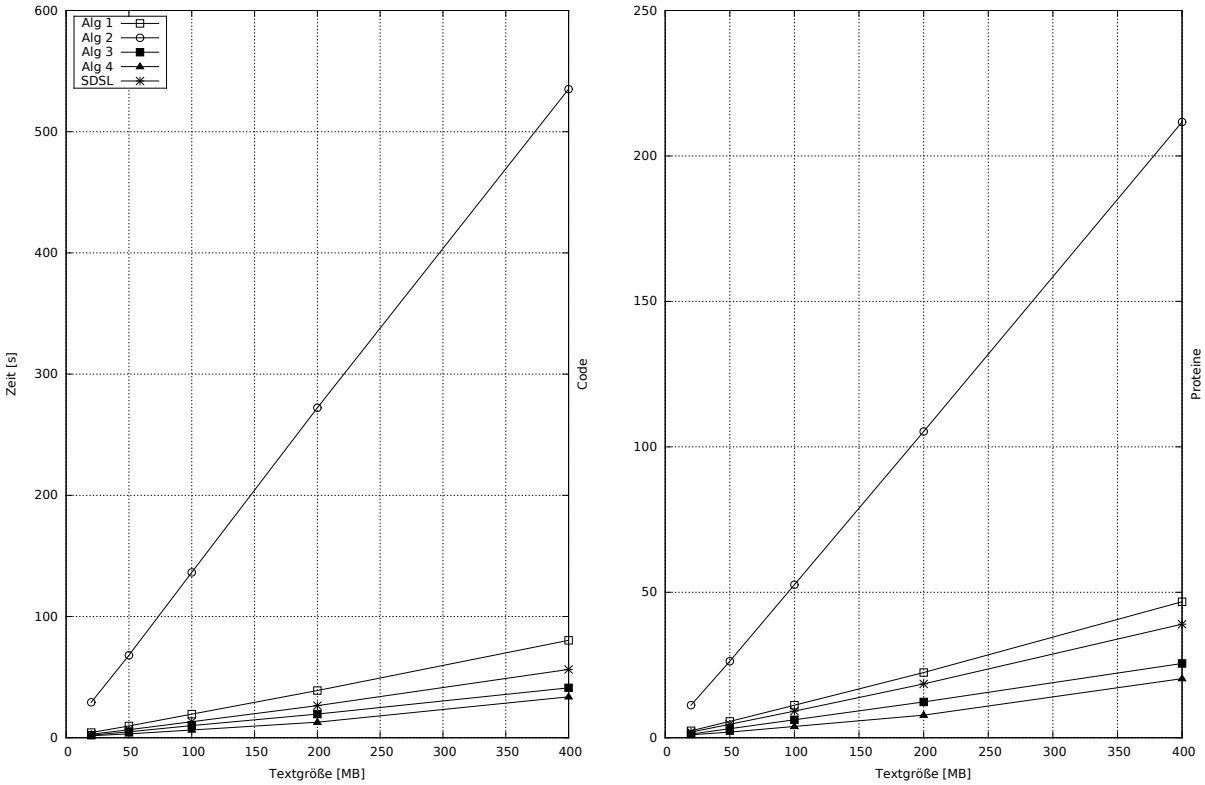


Abbildung 4.2: Laufzeit der vorgestellten Algorithmen für Eingaben der Art Programmcode und Protein. Die Algorithmen sind für die Legende gekürzt. Alg 1 steht für die „Konstruktion mit n Bits und Sortierung“. Alg 2 ist der Algorithmus„Konstruktion mit n Bits und Select Anfragen“. Der Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Sortierung“ wird mit Alg 3 abgekürzt. Der letzte entworfene Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Partitionierung“ wird Alg 4 genannt. Und zulezt der Konstruktionsalgorithmus der SDSL, welcher mit dem Namen der Bibliothek abgekürzt wird.

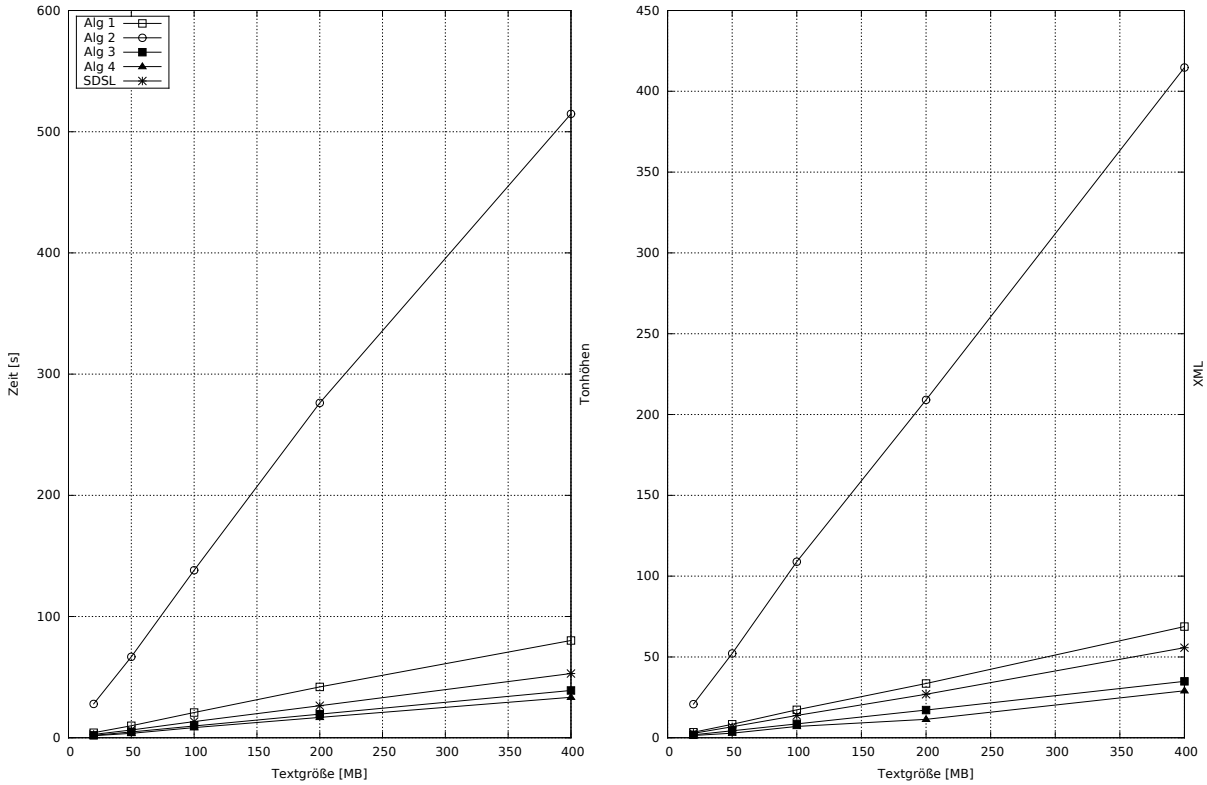


Abbildung 4.3: Laufzeit der vorgestellten Algorithmen für Eingaben der Art Tonhöhen und XML. Die Algorithmen sind für die Legende gekürzt. Alg 1 steht für die „Konstruktion mit n Bits und Sortierung“. Alg 2 ist der Algorithmus„Konstruktion mit n Bits und Select Anfragen“. Der Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Sortierung“ wird mit Alg 3 abgekürzt. Der letzte entworfene Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Partitionierung“ wird Alg 4 genannt. Und zulezt der Konstruktionsalgorithmus der SDSL, welcher mit dem Namen der Bibliothek abgekürzt wird.

Die Laufzeit der Konstruktion der vorgestellten Algorithmen wird mit der Laufzeit der Implementierung der SDSL verglichen.
Die Abbildungen 4.1, 4.2 und 4.3 zeigen die Konstruktionsdauer für alle Testeingaben. Die Laufzeit ist in Sekunden angegeben.
Es ist gut zu erkennen, dass die Laufzeit aller Algorithmen linear mit der Textgröße ansteigt. Außerdem nimmt die benötigte Laufzeit für größere Alphabete zu. Dies ergibt sich durch die erhöhte Komplexität der Konstruktion.
Der Algorithmus „Konstruktion mit $O(n \lg \sigma)$ extra Bits und Partitionierung“ ist in allen Testfällen der schnellste Algorithmus. Unabhängig von der Alphabetgröße ist die Laufzeit des Algorithmus $1,9$ bis $3,4$ schneller als die Implementierung der SDSL.
Der zweit schnellste Algorithmus ist die „Konstruktion mit $O(n \lg \sigma)$ extra Bits und Sortierung“. Er berechnet die Waveletmatrix $1,3 - 1,5$ mal schneller als die SDSL.
Die inplace Varianten schneiden in der Laufzeit schlechter als die SDSL ab. Der Algorithmus „Konstruktion mit n extra Bits und Sortierung“ hat eine um $1,2 - 1,5$ mal längere

Laufzeit als die SDSL. Dabei fällt auf, dass mit größeren Alphabeten der Unterschied zunimmt. Der letzte Algorithmus ist klarer Verlierer dieses Testes „Konstruktion mit $O(n \lg \sigma)$ extra Bits und Partitionierung“ hat über alle Testfälle eine Laufzeit die fünf bis zehn Mal höher als die der SDSL ist.

4.3 Speicherbedarf

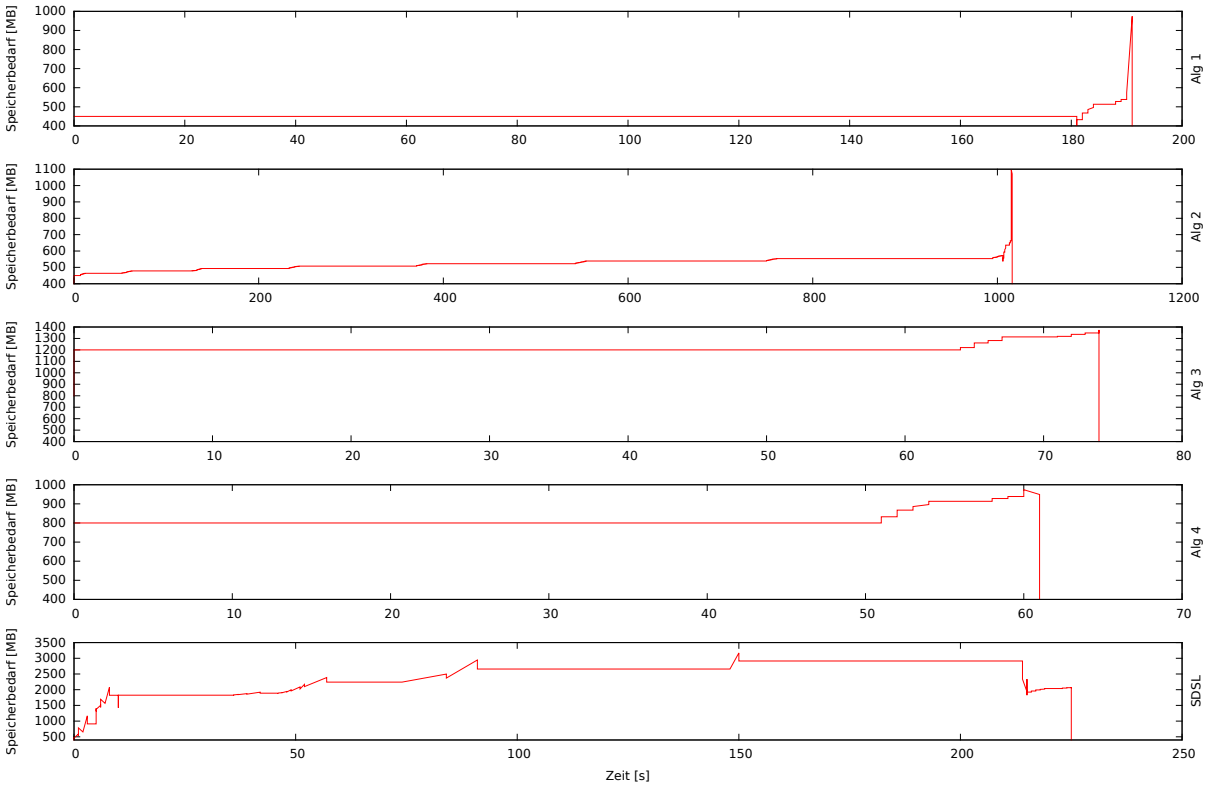


Abbildung 4.4: Speicherbedarf der vorgestellten Algorithmen für einen Programmcode der Größe 400 MB. Die Algorithmen sind für die Legende gekürzt. Alg 1 steht für die „Konstruktion mit n Bits und Sortierung“. Alg 2 ist der Algorithmus„Konstruktion mit n Bits und Select Anfragen“. Der Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Sortierung“ wird mit Alg 3 abgekürzt. Der letzte entworfene Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Partitionierung“ wird Alg 4 genannt. Und zulezt der Konstruktionsalgorithmus der SDSL, welcher mit dem Namen der Bibliothek abgekürzt wird.

Der Speicherbedarf der Algorithmen wird bestimmt mit dem Tool *malloc_ count.*[2] Das Tool, überschreibt die *C* Methoden *malloc, free, realloc* und *calloc*, die genutzt werden um Speicher zu allokieren oder Speicher frei zu geben. Die überschriebenen Methoden werden dann genutzt um den aktuellen Speicherbedarf zu messen. Nachteil dabei, Speicherveränderungen die nicht durch die überschriebenen Methoden geschehen, werden nicht erfasst.

[2] `https://github.com/bingmann/malloc_count`, besucht 8. September 2016

Die Abbildung 4.4, zeigt den beispielhaften Verlauf des Speicherbedarfs der Algorithmen, für einen Programmcode der Größe 400 MB. Die für die Abbildung angegebene Laufzeit, gemessen in Sekunden, ist durch die Speicheranalyse verfälscht. Der Speicherbedarf wird angegeben in Megabyte und die Achsenbeschriftung beginnt bei der Größe der Eingabe. Dadurch ist es einfacher nach zu vollziehen, wie viel zusätzlicher Speicherbedarf verwendet wird.
Der Verlauf der Kurven sieht für alle natürlichen Texte ähnlich aus. Auch die Textgröße spielt nur eine geringe Rolle. Dies erkennt man leicht an den zusätzlichen Speicherverläufen im Anhang, siehe Kapitel A. Für größere Texte sind die besonderen Merkmale der Kurven besser ausgeprägt. Für Algorithmus „Konstruktion mit n Bits und Sortierung“ erkennt man, den linearen Verlauf des Speichers. Der Algorithmus arbeitet also sichtbar inplace. Der spontane Anstieg um die 180 Sekunden Marke hängt mit der Konstruktion der Hilfsstrukturen zusammen.
Der zweite Algorithmus „Konstruktion mit n Bits und Select Anfragen“ erhöht seinen Speicherbedarf stetig, da die Hilfsstrukturen für die select Anfragen gespeichert werden müssen. Bei genauer Betrachtung erkennt man die Plateaus zwischen den Erzeugungen der einzelnen Select Strukturen. Der spontane Anstieg wie im ersten Algorithmus ist wieder durch die zusätzlichen Hilfsstrukturen zu erklären.
Bei dem Verlauf des dritten Algorithmus, „Konstruktion mit $O(n \lg \sigma)$ Bits und Sortierung“ kann man gut erkennen, dass zu Anfang der Hilfsvektor und ein Ausgabevektor, von jeweils der Größe der Eingabe, erzeugt werden und ausschließlich auf diesen und der Eingabe gearbeitet wird. Der Anstieg bei ungefähr 65 Sekunden ist wieder durch die Hilfsstrukturen zu erklären.
Auch beim vierten Algorithmus, „Konstruktion mit $O(n \lg \sigma)$ Bits und Partitionierung“ erkennt man den zusätzlichen Speicher, in der Größe der Eingabe, den der Hilfsvektor einnimmt. Leider wird durch das Tool malloc_count nicht aufgezeichnet, wie der Speicherbedarf während der Konstruktion schwankt. Der Anstieg des Speicherbedarfs nach etwa 50 Sekunden ist durch die Hilfsstrukturen zu erklären.
Der Konstruktionsalgorithmus der SDSL beginnt mit einer kurzen Initialisierungsphase, in der aus dem Buffer gelesen wird. Danach steigt der Speicherbedarf kontinuierlich an. Es ist auffällig, dass der Algorithmus trotz seiner externen Implementierung einen enormen Speicherbedarf von 5 bis 9 mal der Größe der Eingabe besitzt. Der Algorithmus schöpft seinen theoretischen Speicherbedarf von $n \lg \sigma + O(1)$ voll aus.

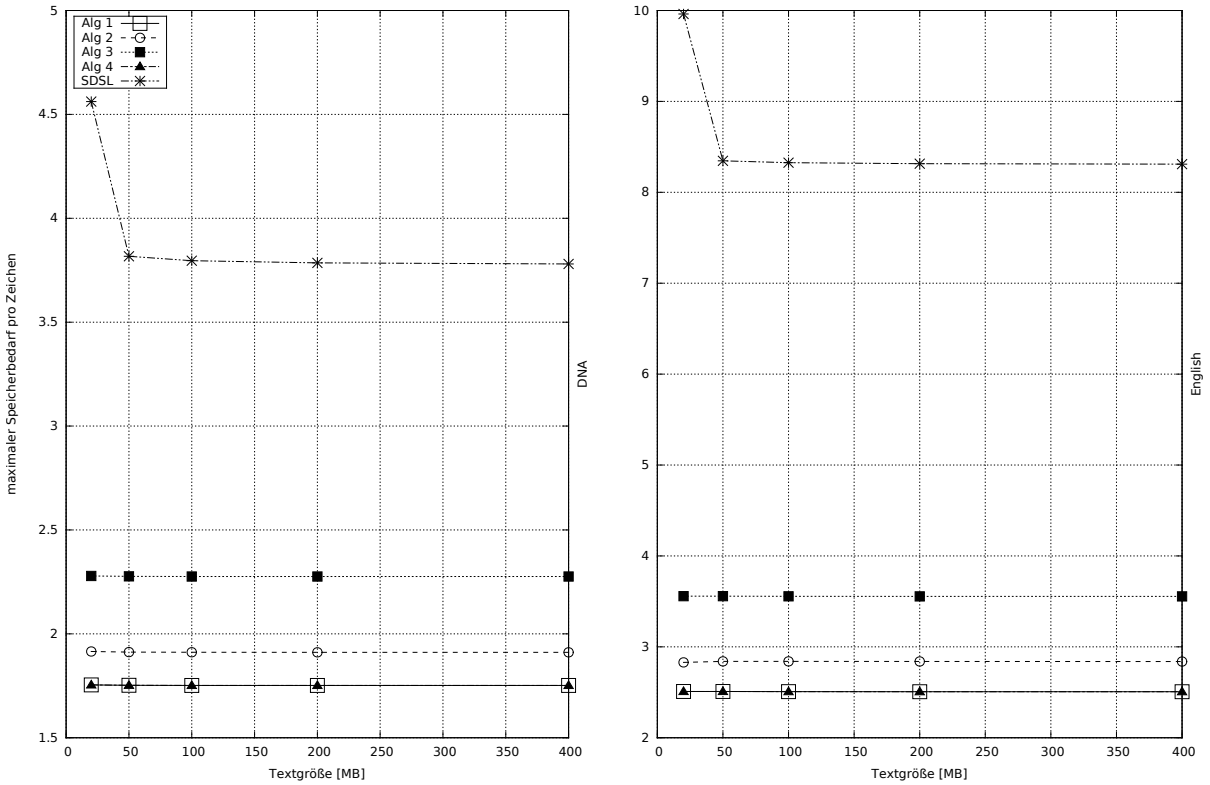


Abbildung 4.5: Laufzeit der vorgestellten Algorithmen für Eingaben der Art DNA und Englisch. Die Algorithmen sind für die Legende gekürzt. Alg 1 steht für die „Konstruktion mit n Bits und Sortierung“. Alg 2 ist der Algorithmus„Konstruktion mit n Bits und Select Anfragen“. Der Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Sortierung“ wird mit Alg 3 abgekürzt. Der letzte entworfene Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Partitionierung“ wird Alg 4 genannt. Und zulezt der Konstruktionsalgorithmus der SDSL, welcher mit dem Namen der Bibliothek abgekürzt wird.

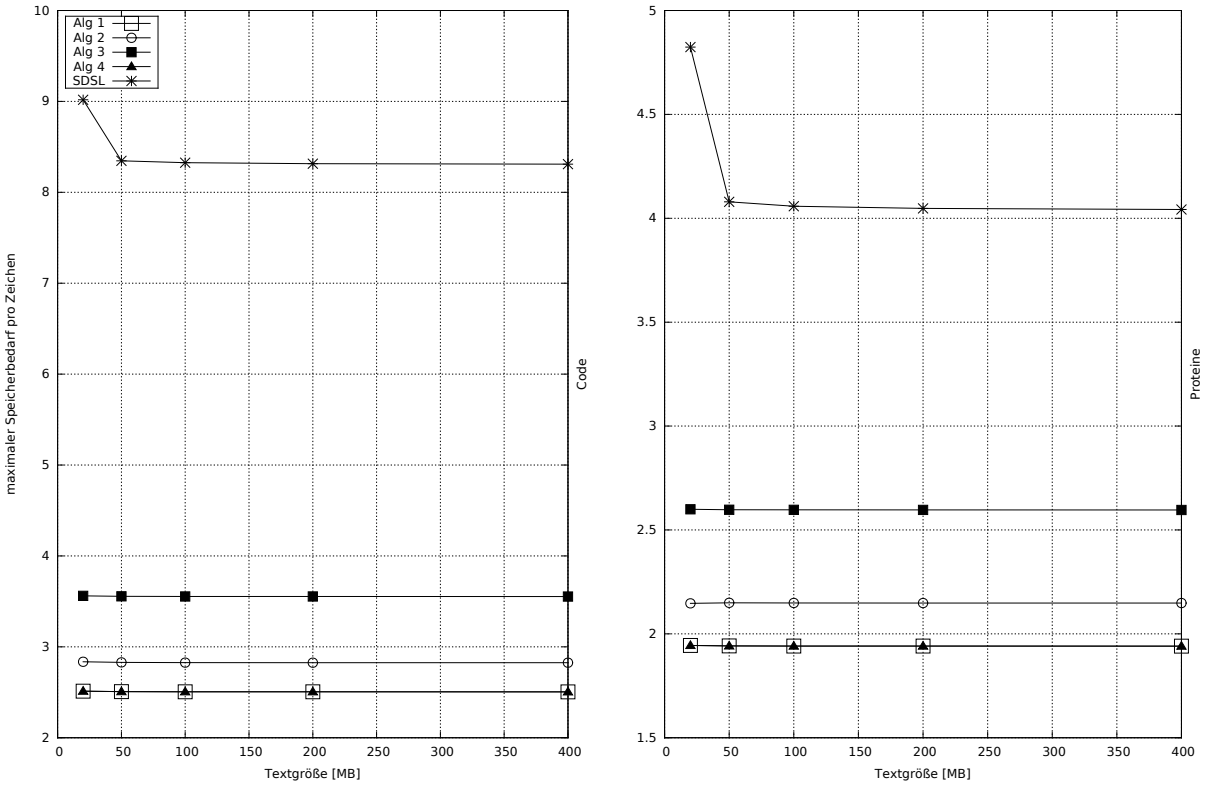


Abbildung 4.6: Laufzeit der vorgestellten Algorithmen für Eingaben der Art Programmcode und Protein. Die Algorithmen sind für die Legende gekürzt. Alg 1 steht für die „Konstruktion mit n Bits und Sortierung“. Alg 2 ist der Algorithmus„Konstruktion mit n Bits und Select Anfragen“. Der Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Sortierung“ wird mit Alg 3 abgekürzt. Der letzte entworfene Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Partitionierung“ wird Alg 4 genannt. Und zulezt der Konstruktionsalgorithmus der SDSL, welcher mit dem Namen der Bibliothek abgekürzt wird.

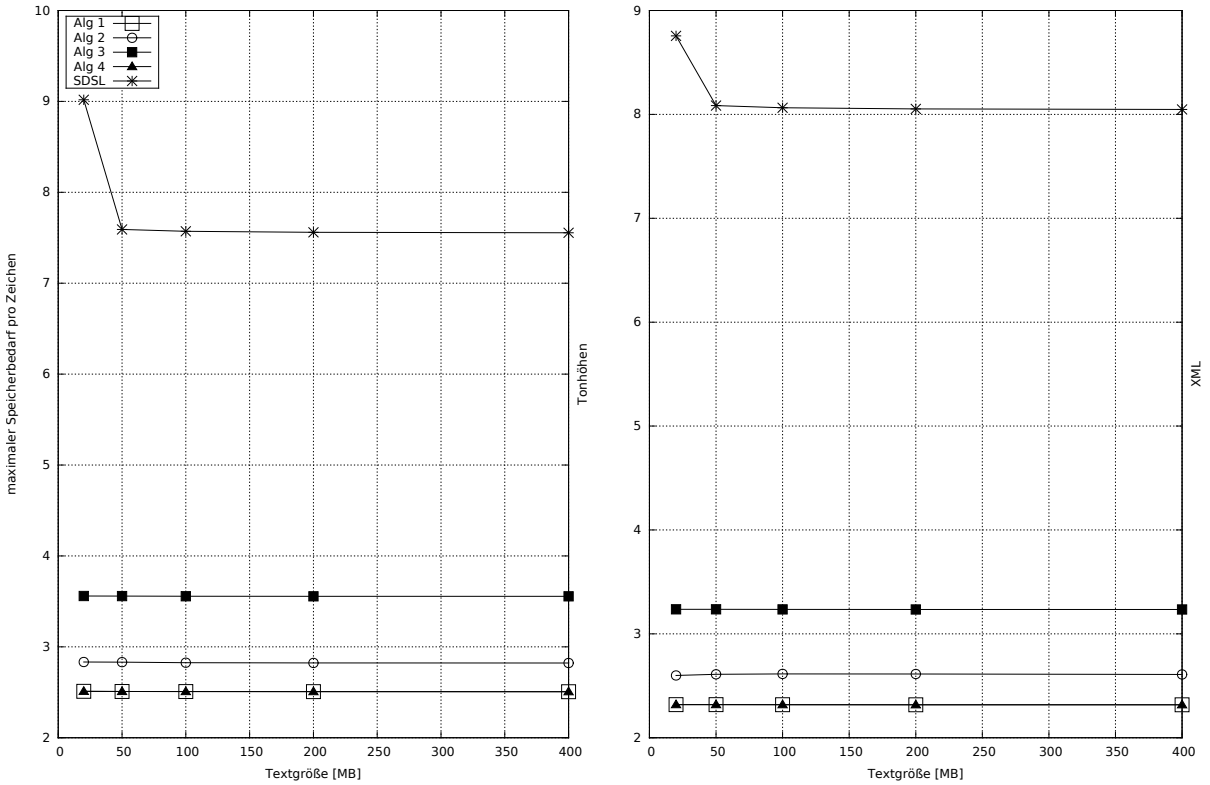


Abbildung 4.7: Laufzeit der vorgestellten Algorithmen für Eingaben der Art Tonhöhen und XML. Die Algorithmen sind für die Legende gekürzt. Alg 1 steht für die „Konstruktion mit n Bits und Sortierung“. Alg 2 ist der Algorithmus„Konstruktion mit n Bits und Select Anfragen“. Der Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Sortierung“ wird mit Alg 3 abgekürzt. Der letzte entworfene Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Partitionierung“ wird Alg 4 genannt. Und zulezt der Konstruktionsalgorithmus der SDSL, welcher mit dem Namen der Bibliothek abgekürzt wird.

Abbildungen 4.5 ,4.6 und 4.7 zeigen den maximalen Speicherbedarf pro Zeichen. Diesen Wert zu kennen ist nützlich um exakt vorauszusagen, wie groß der maximale Speicherbedarf für einen Text sein wird. Betrachtet man die Abbildungen erkennt man einen linearen Zusammenhang zwischen der Textgröße und dem maximalen Speicherbedarf pro Zeichen für alle Algorithmen und Texte. Dies ist ein wichtiges Ergebnis, denn dadurch ist es sehr wahrscheinlich den maximalen Speicherbedarf für Texte von über 400 MB präzise errechnen zu können. Ausnahme bildet in jeder Abbildung die Implementierung der SDSL. Sie besitzt für kleine Texte einen höheren maximalen Speicherbedarf pro Zeichen. Dies liegt am Overhead der Bibliothek. Dafür spricht, dass der Verlauf geringfügig sinkt aber sonst linear verläuft. Der Overhead macht also einen immer geringeren Teil des Speicherbedarfs aus.
Wie zu erwarten steigt auch der maximale Speicherbedarf pro Zeichen mit der Größe des Alphabets.
Erwähnenswert ist auch, dass der Algorithmus „Konstruktion mit n Bits und Sortierung“

einen sehr ähnlichen aber nicht identischen maximalen Speicherbedarf pro Zeichen besitzt wie der Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Partitionierung“. Dies lässt darauf schließen, dass beide Algorithmen ihren maximalen Speicherbedarf bei der Erstellung der Hilfsstrukturen erreichen.

Kapitel 5

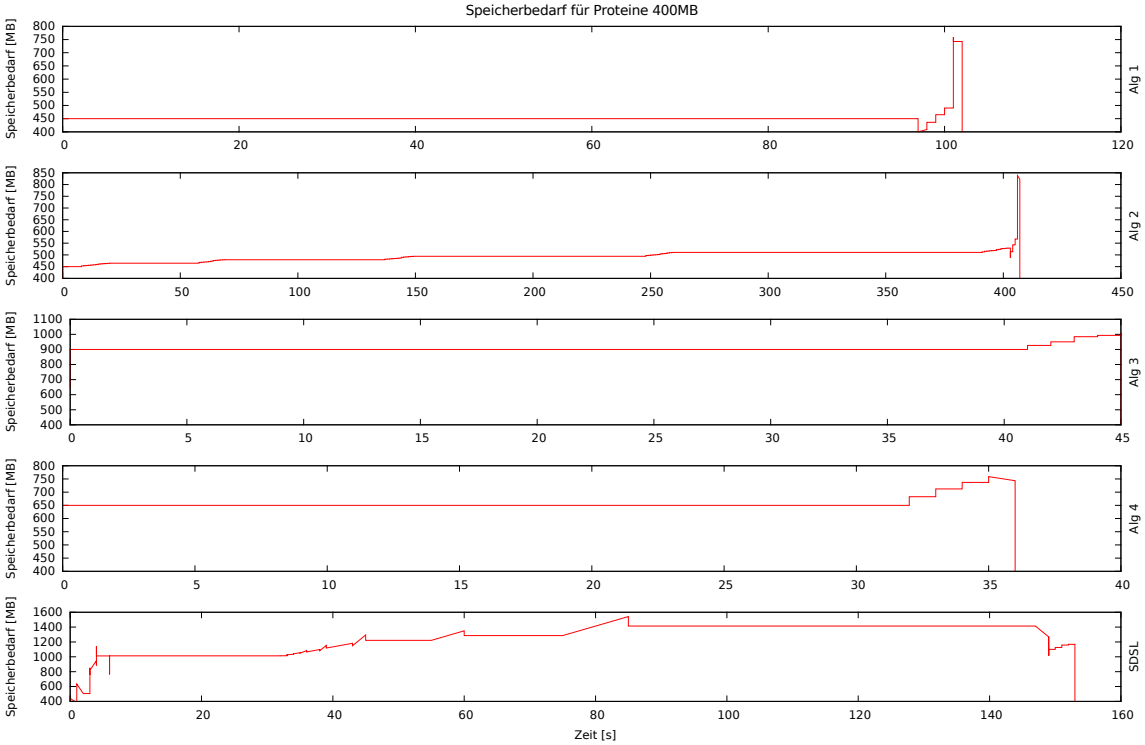
Fazit

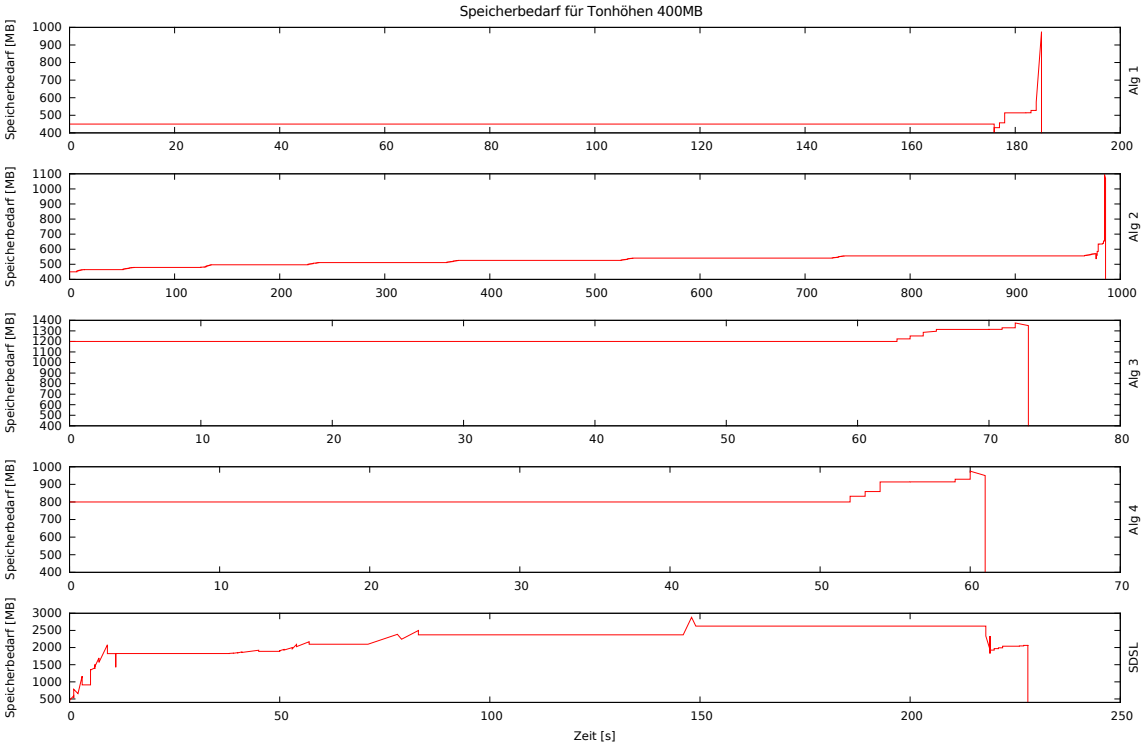
In dieser Arbeit wurden vier Konstruktionsalgorithmen detailliert beschrieben und implementiert. Die Implementierungen der Algorithmen wurden auf verschiedenartigen Texten untersucht. Dabei wurde Wert darauf gelegt unterschiedlich große Alphabete zu verwenden. Untersucht wurden die Algorithmen auf Konstruktionsdauer, Verlauf des Speicherbedarfes und dem maximalen Speicherbedarf pro Zeichen der Eingabe. Ziel sollte es sein einen möglichst geeigneten Algorithmus für die Konstruktion einer Waveletmatrix anzugeben.
Es hat sich herauskristallisiert, dass drei der vier Algorithmen eine gute Konstruktion ermöglichen, unter der Abwägung von Speicherbedarf und Laufzeit. Gemeint sind die beiden Algorithmen, die als Grundidee auf die Sortierung aufbauen und der Algorithmus welcher durch die Partitionierung der Eingabe die Waveletmatrix erzeugt.
Obwohl die inplace Variante nur einen zusätzlichen Speicher von n Bits erhält, ist sie nur $1,2-1,5$ mal langsamer als der Konstruktionsalgorithmus der SDSL. Der Algorithmus arbeitet nachgewiesen inplace mit einem konstanten Speicher. Existiert das Bedürfnis einen Konstruktionsalgorithmus zu wählen, welcher einen möglichst geringen Speicherbedarf hat, ohne horrende Laufzeiten, so könnte man diesen Algorithmus empfehlen.
Der ähnliche Algorithmus, welcher auf die selbe Idee aufbaut, aber zusätzlichen Speicher verwendet, bietet ebenfalls eine gute Konstruktion. Der Algorithmus berechnet die Waveletmatrix $1,3-1,5$ mal schneller als die SDSL und das mit akzeptablem zusätzlichen Speicher.
Als besten Algorithmus hat sich aber die „Konstruktion mit $O(n \lg \sigma)$ extra Bits und Partitionierung“ herausgestellt. Diese Variante berechnet die Waveletmatrix doppelt bis dreieinhalb Mal so schnell wie die SDSL, wobei sie weniger Speicherplatz benötigt als der zuvor erwähnte Algorithmus. Wenn der Speicherplatz nicht Priorität Nummer eins einnimmt, so sollte immer dieser Algorithmus gewählt werden.
Als absolut ungeeignet hat sich die Idee herausgestellt, die Waveletmatrix durch select Anfragen auf den bereits erstellten Teilen der Matrix zu konstruieren. Sie arbeitet zwar inplace, erzeugt die Waveletmatrix aber fünf bis zehn Mal langsamer als die SDSL. Es

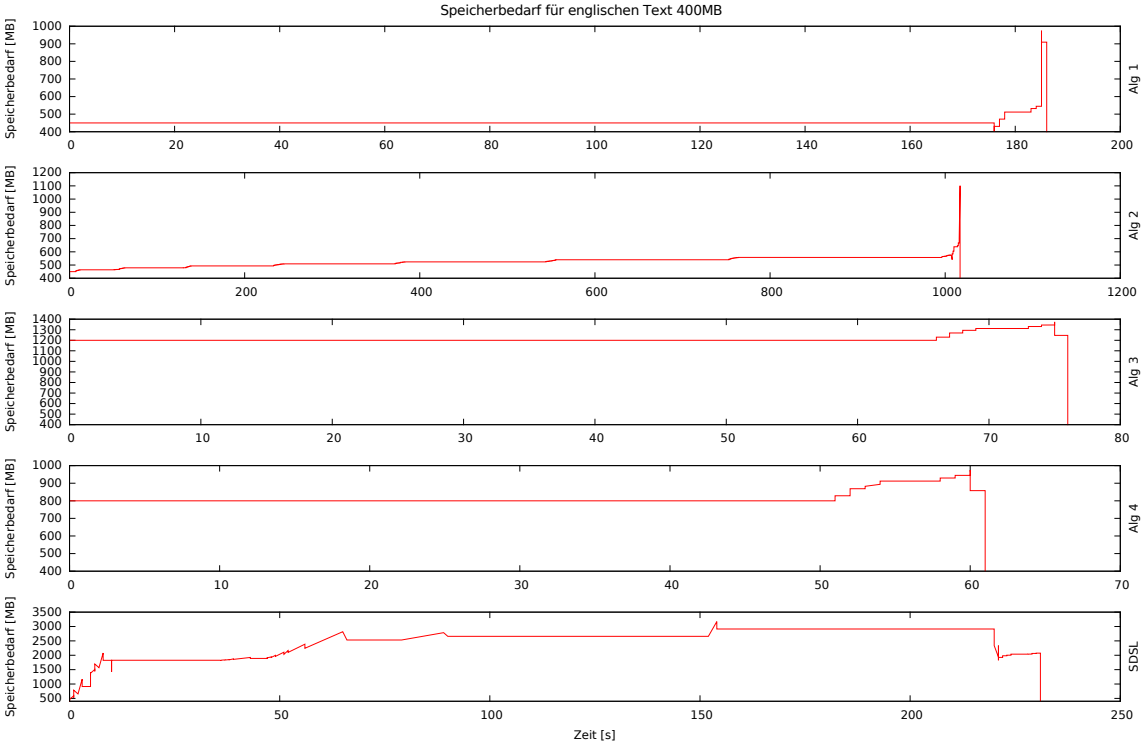
existiert auch kaum Verbesserungsspielraum, da die Anfragen bereits in konstanter Zeit berechnet werden können und die Hilfsstrukturen werden auch in linearer Zeit erstellt.
Bei der Konstruktion einer Waveletmatrix, muss immer abgewogen werden zwischen Laufzeit und Speicherbedarf. Die Tests haben gezeigt, dass die inplace Variante gegenüber der ähnlichen Variante mit zusätzlichem Speicher, mehr als doppelt soviel Zeit in Anspruch nimmt.
Die Waveletmatrix ist eine wichtige Datenstruktur und diese Arbeit hat gezeigt, dass noch Potential für effizientere Algorithmen besteht. Der Algorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Sortierung“ eignet sich hervorragend für *Multithreading*. Die Hauptarbeit liegt in der Sortierung und es bietet sich an, die zu sortierende Eingabe in beliebig viele Unterprobleme zu teilen und jeweils einen Prozess auf dieses anzuwenden um z.B. die Nullbits parallel zu zählen, oder die Wörter parallel zu verschieben. Gleiches gilt für den Konstruktionsalgorithmus „Konstruktion mit $O(n \lg \sigma)$ Bits und Partitionierung“. Auch dieser Algorithmus kann von Parallelität profitieren. Da für die Waveletmatrix, anders als für den Wavelettree, noch wenig dieser Ansätze verfolgt wurden, sollte in der Zukunft dort das Augenmerk liegen.

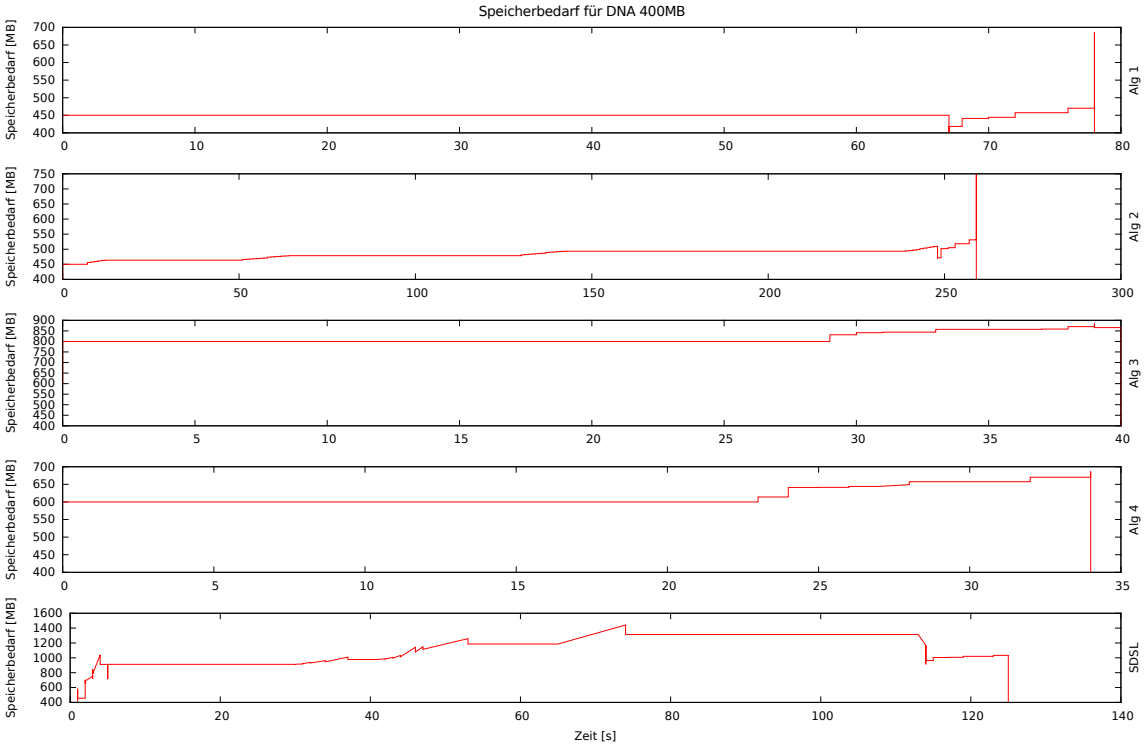
Anhang A

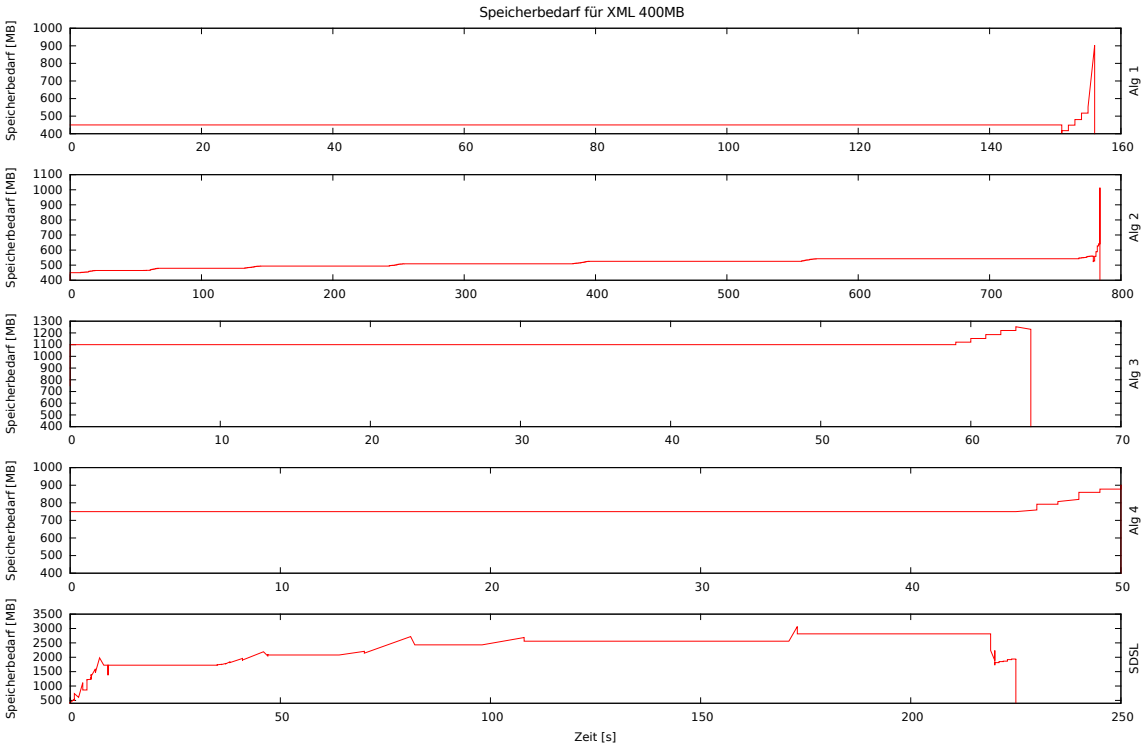
Weitere Informationen




Speicherbedarf für Tonhöhen 400MB
Speicherbedarf [MB]
Alg 1
Alg 2
Alg 3
Alg 4
SDSL
Zeit [s]


Speicherbedarf für englischen Text 400MB
Speicherbedarf [MB]
Alg 1
Alg 2
Alg 3
Alg 4
SDSL
Zeit [s]


Speicherbedarf für DNA 400MB
Speicherbedarf [MB]
Alg 1
Alg 2
Alg 3
Alg 4
SDSL
Zeit [s]


Speicherbedarf für XML 400MB
Speicherbedarf [MB]
Alg 1
Alg 2
Alg 3
Alg 4
SDSL
Zeit [s]

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[1] Aleksandr, Kronrod: *Optimal ordering algorithm without operational field.* DOKLADY AKADEMII NAUK SSSR, 186(6):1256, 1969.

[2] Alexander, Bowe: *Multiary wavelet trees in practice.* Doktorarbeit, Honours thesis, RMIT Univ., Australia, 2010.

[3] Clark, David: *Compact Pat Trees.* Doktorarbeit, PhD thesis, University of Waterloo, 1998.

[4] Claude, Francisco und Gonzalo Navarro: *Practical Rank/Select Queries over Arbitrary Sequences.* In: Amir, Amihood, Andrew Turpin und Alistair Moffat (Herausgeber): *String Processing and Information Retrieval, 15th International Symposium, SPIRE 2008, Melbourne, Australia, November 10-12, 2008. Proceedings*, Band 5280 der Reihe *Lecture Notes in Computer Science*, Seiten 176–187. Springer, 2008.

[5] Claude, Francisco, Patrick K. Nicholson und Diego Seco: *Space Efficient Wavelet Tree Construction.* In: Grossi, Roberto, Fabrizio Sebastiani und Fabrizio Silvestri (Herausgeber): *String Processing and Information Retrieval, 18th International Symposium, SPIRE 2011, Pisa, Italy, October 17-21, 2011. Proceedings*, Band 7024 der Reihe *Lecture Notes in Computer Science*, Seiten 185–196. Springer, 2011.

[6] Ferragina, P und G Navarro: *The Pizza & Chili Corpus*, 2007.

[7] Ferragina, Paolo, Raffaele Giancarlo und Giovanni Manzini: *The myriad virtues of Wavelet Trees.* Inf. Comput., 207(8):849–866, 2009.

[8] Ferragina, Paolo, Giovanni Manzini, Veli Mäkinen und Gonzalo Navarro: *Compressed representations of sequences and full-text indexes.* ACM Trans. Algorithms, 3(2), 2007.

[9] Francisco Claude, Gonzalo Navarro, Alberto Ordóñez Pereira: *The wavelet matrix: An efficient wavelet tree for large alphabets.* Inf. Syst., 47:15–32, 2015.

[10] Gog, Simon, Timo Beller, Alistair Moffat und Matthias Petri: *From Theory to Practice: Plug and Play with Succinct Data Structures.* In: Gudmundsson, Joachim und Jyrki Katajainen (Herausgeber): *Experimental Algorithms - 13th International Symposium, SEA 2014, Copenhagen, Denmark, June 29 - July 1, 2014. Proceedings*, Band 8504 der Reihe *Lecture Notes in Computer Science*, Seiten 326–337. Springer, 2014.

[11] Golynski, Alexander, J. Ian Munro und S. Srinivasa Rao: *Rank/select operations on large alphabets: a tool for text indexing.* In: *Proceedings of the Seventeenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2006, Miami, Florida, USA, January 22-26, 2006*, Seiten 368–373. ACM Press, 2006.

[12] Grossi, Roberto, Ankur Gupta und Jeffrey Scott Vitter: *High-order entropy-compressed text indexes.* In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.*, Seiten 841–850. ACM/SIAM, 2003.

[13] Jeffrey S. Salowe, William L. Steiger: *Simplified Stable Merging Tasks.* J. Algorithms, 8(4):557–571, 1987.

[14] Liskiewicz, Maciej und Rüdiger Reischuk: *The Complexity World below Logarithmic Space.* In: *Proceedings of the Ninth Annual Structure in Complexity Theory Conference, Amsterdam, The Netherlands, June 28 - July 1, 1994*, Seiten 64–78. IEEE Computer Society, 1994.

[15] Michael Burrows, David Wheeler: *A block-sorting lossless data compression algorithm.* In: *DIGITAL SRC RESEARCH REPORT*. Citeseer, 1994.

[16] Navarro, Gonzalo und Eliana Providel: *Fast, Small, Simple Rank/Select on Bitmaps.* In: Klasing, Ralf (Herausgeber): *Experimental Algorithms - 11th International Symposium, SEA 2012, Bordeaux, France, June 7-9, 2012. Proceedings*, Band 7276 der Reihe *Lecture Notes in Computer Science*, Seiten 295–306. Springer, 2012.

[17] Rajeev Raman, Venkatesh Raman, Srinivasa Rao Satti: *Succinct indexable dictionaries with applications to encoding* k*-ary trees, prefix sums and multisets.* ACM Trans. Algorithms, 3(4), 2007.

[18] Tischler, German: *On Wavelet Tree Construction.* In: Giancarlo, Raffaele und Giovanni Manzini (Herausgeber): *Combinatorial Pattern Matching - 22nd Annual Symposium, CPM 2011, Palermo, Italy, June 27-29, 2011. Proceedings*, Band 6661 der Reihe *Lecture Notes in Computer Science*, Seiten 208–218. Springer, 2011.

[19] Veli Mäkinen, Gonzalo Navarro: *Rank and select revisited and extended.* Theor. Comput. Sci., 387(3):332–347, 2007.