DIPLOMARBEIT

Bidirektionale indexbasierte Suche in Texten

Institut für Theoretische Informatik
Fakultät für Ingenieurwissenschaften und Informatik
Universität Ulm

vorgelegt von
Thomas Schnattinger

Gutachter: Prof. Dr. Enno Ohlebusch
Prof. Dr. Uwe Schöning

Betreuer: Prof. Dr. Enno Ohlebusch
Dipl.-Inf. Simon Gog

Ulm, den 12. Januar 2010

Zusammenfassung

Eine Suche ist die Bestimmung der Positionen aller Vorkommen eines Musters in einem Text. Bei der Untersuchung von sehr großen Texten ist es oft unerlässlich, diesen vorzuverarbeiten und einen Index zu erstellen, der diese Suche effizienter unterstützt. Mit der Einführung des Suffix Arrays im Jahr 1990 wurde der Grundstein gelegt für eine Vielzahl von Indexen, die sich zur Mustersuche eignen. Sie unterscheiden sich nicht zuletzt im Bedarf an Speicherplatz und in der Zeit, die eine Suche nach einem Muster benötigt. Erweitert man das Konzept der Mustersuche auf bidirektionale Suche, also um die Möglichkeit die Menge der gefundenen Vorkommen eines Musters durch die links- oder rechtsseitige Verlängerung des Musters weiter einzuschränken, so existieren bislang nur wenige Ansätze, die wirklich für praktische Zwecke geeignet sind. In dieser Arbeit werden verschiedene Verfahren der Mustersuche untersucht und kombiniert. Darauf aufbauend wird ein neuer Index vorgestellt, der bidirektionale Suche unterstützt und dabei sowohl sehr speicherplatz- als auch laufzeiteffizient ist. Die theoretischen Ergebnisse werden schließlich in einem umfassenden Experiment praktisch bestätigt.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Heutzutage ist es leicht, mit der Hilfe von Computern viele Gigabytes große Texte abzuspeichern. Möchte man aber eine sehr große Menge von Daten analysieren, so ist es wichtig sich Gedanken zu machen, wie man das algorithmisch am besten löst. Ein wichtiges Teilgebiet der Informatik beschäftigt sich mit Verfahren zur Mustersuche, also der Bestimmung aller Vorkommen eines Suchmusters in großen Texten. Um dieses Problem zu lösen, könnte man den zu suchenden String an den Anfang des Textes legen und Buchstabe für Buchstabe vergleichen. Tritt das gesamte Muster an dieser Stelle im Text auf, ist ein Vorkommen gefunden. Ansonsten wird das Muster um einen Buchstaben nach rechts verschoben und weiter verglichen. Dieses naive Verfahren findet wohl alle Vorkommen im Text, eignet sich jedoch wegen seines schlechten Laufzeitverhaltens nicht für größere Datenmengen. Für die effizientere Bestimmung der Vorkommen eines Musters existieren sogenannte Online-Suchverfahren, die in einem ersten Schritt das Suchmuster vorverarbeiten. Die Suche erfolgt zwar auch durch den sequentiellen Vergleich mit dem Text, durch die Vorverarbeitung des Musters ergibt sich aber eine verbesserte Laufzeit, die linear in der Länge des Textes ist. Auch wenn sie deutlich schneller sind als der naive Ansatz, sie durchsuchen immer noch den ganzen Text und sind deshalb bei der Verarbeitung von sehr großen Daten nicht einsetzbar.

Diese Arbeit beschäftigt sich mit der indexbasierten Suche in Texten. Ist der gesamte Text vor dem Suchen bekannt, kann für diesen ein Volltextindex erstellt werden, auf dem die Suchalgorithmen arbeiten. Dies erfordert zwar einmalig eine hohe Investition von Rechenzeit, erspart jedoch viel Zeit, wenn viele Suchanfragen auf ein und dem selben Text bearbeitet werden sollen.

Denn der Aufwand, um ein Muster im Text mit Hilfe eines Index zu finden, ist in der Regel um ein Vielfaches geringer als ohne. Die optimale Laufzeit einer Mustersuche ist sogar unabhängig von der Länge des Textes. Werden sehr große Mengen an Daten verarbeitet, so ist außerdem die Größe dieses Volltextindexes häufig von entscheidender Wichtigkeit.

Die klassische Form der Mustersuche ist, mit Hilfe eines Indexes alle Vorkommen eines bekannten Musters in einem Text zu finden. Ziel dieser Arbeit ist es die Frage zu untersuchen, ob sich bekannte Volltextindexe so erweitern oder kombinieren lassen, dass damit bidirektionale Suche möglich ist. Dabei handelt es sich um ein Verfahren, das in der Lage ist, die Menge der gefundenen Vorkommen eines Musters weiter einzuschränken, indem dieses auf der linken oder auf der rechten Seite um einen Buchstaben verlängert wird, ohne dabei die bisherige Suche wiederholen zu müssen. Damit ist die Suche nach palindromischen Mustern möglich, die zu Beginn der Suche noch nicht feststehen. Zum Beispiel ist bidirektionale Suche in der Bioinformatik einsetzbar bei der Analyse der Sekundärstruktur der RNA.

Diese Arbeit gliedert sich wie folgt. In Kapitel 2 werden grundlegende Notationen und Techniken erklärt und Datenstrukturen eingeführt, welche für die weitere Arbeit wichtig sind. Zunächst wird dann in Kapitel 3 das Konzept der Vorwärtssuche beschrieben und es werden verschiedene solche Verfahren vorgestellt. Das Kapitel 4 erläutert zwei Verfahren zur Rückwärtssuche. In Kapitel 5 werden verschiedene Möglichkeiten vorgestellt und bewertet, wie sich die Konzepte Vorwärtssuche und Rückwärtssuche zur bidirektionalen Suche kombinieren lassen. Außerdem werden die Datenstrukturen *Affixbaum* und *Affix Array* beschrieben, welche speziell für die bidirektionale Suche konzipiert wurden. Es wird schließlich im Kapitel 6 eine weitere speicherplatz- und laufzeiteffiziente Datenstruktur zur bidirektionalen Suche entwickelt, der *Bidirektionale Wavelet-Index*. Kapitel 7 gibt einen Einblick in eine praktische Anwendung der bidirektionalen Suche bei der Mustersuche in der RNA. In Kapitel 8 wird ein umfangreiches Experiment beschrieben, welches einige der vorgestellten Indexe anhand verschiedener komplexer Testfälle vergleicht.

Kapitel 2

Grundlagen

In dieser Arbeit werden Algorithmen und Datenstrukturen eingeführt, welche die Vorkommen eines (vergleichsweise kurzen) Musters in einem Text suchen. Dabei werden stets die folgenden Notationen verwendet.

Definition 1 (Grundlegende Notationen)**.**

- *Alle Strings sind definiert über dem vollständig geordneten Alphabet Σ. Der besondere Buchstabe $\$ \in \Sigma$ steht dabei ausschließlich an letzter Position im Text und ist kleiner als alle anderen Buchstaben in Σ.*
- *Ein Text T der Länge n wird $T[1..n]$ geschrieben, das Muster P der Länge m $P[1..m]$.*
- *$T[i]$ bezeichnet den i-ten Buchstaben im String T. Das i-te Suffix T_i bezeichnet den Teil des Strings T ab dem i-ten Buchstaben, also $T[i..n]$.*

Definition 2. *Das Alphabet Σ ist über die Kleiner-Relation $<$ vollständig geordnet. Die lexikographische Ordnung von Strings über Σ ist folgendermaßen definiert. Seien $c, d \in \Sigma$ Buchstaben aus Σ und $x, y \in \Sigma^*$ Wörter über Σ. Dann gilt:*

$$cx <_{lex} dy \quad \Leftrightarrow \quad c < d \quad \text{oder} \quad c = d \text{ und } x <_{lex} y$$

Falls x ein Präfix von y ist, dann gilt $x \leq_{lex} y$.

Durch die Konvention, dass der Text T immer mit dem besonderen Symbol „$“ endet und dieses auch nur an dieser Stelle vorkommt, sind die Suffixe T_i über die Relation $<_{\text{lex}}$ streng geordnet. Es existiert also eine Permutation π, für die gilt $T_{\pi(1)} <_{\text{lex}} T_{\pi(2)} <_{\text{lex}} \ldots <_{\text{lex}} T_{\pi(n)}$.

2.1 Suffix Arrays

Definition 3 (Suffix Array und inverses Suffix Array)**.** *Das Suffix Array SA eines Textes T ist ein Array, welches eine Permutation der Zahlen im Intervall* $[1..n]$ *enthält, so dass gilt:* $T_{SA[1]} <_{lex} T_{SA[2]} <_{lex} \ldots <_{lex} T_{SA[n]}$.

Das Inverse Suffix Array SA^{-1} *ist ein Array, welches die lexikographische Ordnung der Suffixe* $T_{SA[i]}$ *angibt, und es gilt:* $SA\left[SA^{-1}[i]\right] = i = SA^{-1}\left[SA[i]\right]$.

Beispiel 2.1. *Das Suffix Array von* $T =$ *„*`mississippi$`*“ lautet*

$$SA[1..12] = \{12, 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3\}$$

Dies bedeutet also dass $T_{SA[1]} = T_{12} =$ *„*`$`*“ das lexikographisch kleinste Suffix von T ist. Dagegen ist* $T_{SA[12]} = T_3 =$ *„*`ssissippi$`*“ das lexikographisch größte.*

Aus der lexikographischen Ordnung der Suffixe $T_{SA[i]}$ folgt eine entscheidende Tatsache. Kommt ein Wort ω im Text T k-mal vor, so gibt es genau k viele Suffixe, die mit ω beginnen. Werden die Suffixe lexikographisch sortiert, so bilden genau diejenigen, die ω als Präfix haben, ein Intervall der Größe k. Die Einträge im Suffix Array in diesem Intervall sind genau die Positionen im Text, an denen das Muster ω beginnt.

Definition 4 (ω-Intervall)**.** *Sei* ω *ein Wort über* Σ*. Dann ist das* ω*-Intervall* $[i..j]$ *genau das Intervall, für das gilt:*

- *Alle Suffixe* $T_{SA[k]}$ *mit* $i \leq k \leq j$ *haben* ω *als Präfix*
- *Kein anderes Suffix hat* ω *als Präfix*

Das Suchen nach allen Vorkommen eines Musters P in einem Text ist also gleichbedeutend mit der Bestimmung des entsprechenden P-Intervalls im Suffix Array.

Beispiel 2.2. *Das „*`is`*“-Intervall im Suffix Array zu* $T =$ *„*`mississippi$`*“ ist* $[4..5]$*, denn nur* $T_{SA[4]}$ *und* $T_{SA[5]}$ *haben „*`is`*“ als Präfix, und sonst keines (vgl. Abbildung 2.1). Übrigens ist das „*`iss`*“-Intervall ebenfalls* $[4..5]$.

2.2 Konstruktion

Die Herstellung des Suffix Arrays eines Textes geschieht durch eine lexikographische Sortierung aller seiner Suffixe.

i	T_i
1	mississippi$
2	ississippi$
3	ssissippi$
4	sissippi$
5	issippi$
6	ssippi$
7	sippi$
8	ippi$
9	ppi$
10	pi$
11	i$
12	$

$\xrightarrow[\text{Sortierung}]{\text{Lexikographische}}$

i	$SA[i]$	$T_{SA[i]}$
1	12	$
2	11	i$
3	8	ippi$
4	5	issippi$
5	2	ississippi$
6	1	mississippi$
7	10	pi$
8	9	ppi$
9	7	sippi$
10	4	sissippi$
11	6	ssippi$
12	3	ssissippi$

Abbildung 2.1: Konstruktion des Suffix Arrays zu $T =$ „mississippi$“

Es ist bereits seit 1973 bekannt, dass man den Suffixbaum eines Textes in linearer Zeit konstruieren kann ([Wei73]). Dieser enthält ebenfalls die lexikographische Ordnung aller Suffixe, und kann deshalb zur Herstellung des Suffix Arrays in linearer Zeit verwendet werden. Aufgrund des hohen Speicherplatzverbrauches des Suffixbaumes ist ein effizienter Algorithmus zur direkten Herstellung des Suffix Arrays jedoch wünschenswert.

Man könnte hier jeden gewöhnlichen Sortieralgorithmus einsetzen, der stets zwei Strings Buchstabe für Buchstabe vergleicht. Diese naive Lösung ist jedoch besonders dann sehr langsam, wenn der Text Abschnitte enthält, die häufig vorkommen. Die Tatsache, dass es sich um die Suffixe eines einzigen Textes handelt erlaubt wesentlich bessere Algorithmen. Mit der Vorstellung der Datenstruktur in [MM90] wurde zum Beispiel ein modifizierter Bucket-Sort-Algorithmus beschrieben, welcher für die Herstellung des Suffix Arrays eines Textes der Länge n eine Gesamtkomplexität von $O(n \log n)$ besitzt.

Im Jahr 2003 wurden schließlich unabhängig voneinander drei Algorithmen veröffentlicht, welche in der Lage sind, das Suffix Array direkt in linearer Zeit herzustellen. Häufig ist bei den linearen Algorithmen jedoch der konstante Faktor in der O-Notation so hoch, dass in der Praxis eher einfachere,

nichtlineare Algorithmen verwendet werden. Eine umfassende Übersicht über die verschiedenen Verfahren gibt [PST07].

Kapitel 3

Vorwärtssuche

3.1 Vorwärtssuche auf Suffix Arrays

Der erste vorgeschlagene Möglichkeit einer Suche im Suffix Array stammt von [MM90]. Durch buchstabenweisen Vergleich des Musters P mit dem Text ab der Stelle $SA[i]$ kann festgestellt werden, ob P lexikographisch kleiner, gleich oder größer als das Suffix $T_{SA[i]}$ ist. Damit führt eine binäre Suche auf dem Suffix Array zu den gesuchten Intervallgrenzen (siehe Algorithmus 3.1). Die binäre Suche erfordert dabei $O(\log n)$ Vergleiche von zwei Suffixen. Da jeder dieser Vergleiche bis zu m Schritte erfordert, ergibt das eine Gesamtkomplexität von $O(m \log n)$.

Algorithmus 3.1 Der originale Algorithmus nach [MM90] bestimmt durch binäre Suche die linke Intervallgrenze des P-Intervalls. Ersetzt man in der Schleife den Vergleich $\leq_{\text{lex}}$ durch $<_{\text{lex}}$, so wird die rechte Grenze berechnet.

```
if P ≤lex T_SA[1] then
  return 1
else if P >lex T_SA[n] then
  return n + 1
else
  (l, r) ← (1, n)
  while r − l > 1 do
    m ← (l + r)/2
    if P ≤lex T_SA[m] then
      r ← m
    else
      l ← m
  return r
```

Durch eine zusätzliche Optimierung kann dieser Algorithmus auf $O(m + \log n)$ verbessert werden. Dazu verwendet der Vergleich $\leq_{\text{lex}}$ zusätzliche Informationen, die teilweise aus vorherigen Vergleichen stammen, und teilweise bei der Konstruktion des Suffix Arrays vorausberechnet werden, um überflüssige Zeichenvergleiche zu vermeiden.

3.2 Vorwärtssuche mit der Ψ-Funktion

Die ursprüngliche Version der Vorwärtssuche von [MM90] vergleicht in jedem Schritt der binären Suche das Muster P mit dem Suffix $T_{SA[i]}$ und erfordert somit den Zugriff auf den Text. Ist der Text sehr groß, so möchte man möglicherweise nicht gleichzeitig das Suffix Array und den Text im Speicher halten. Eine andere Variante der Vorwärtssuche basiert auf der Ψ-Funktion.

3.2.1 Die Ψ-Funktion

In einem Suffix Array steht an der Stelle i das Suffix, das unter allen Suffixen die lexikographische Ordnung i besitzt. Es gibt jedoch auch noch weitere Möglichkeiten, diese Ordnung abzuspeichern. Der Wert der Ψ-Funktion an dieser Stelle i ist die lexikographische Ordnung des nächsten, links um einen Buchstaben verkürzten Suffixes.

Definition 5. *Gegeben sei ein Suffix Array SA zu einem Text T. Dann ist die Ψ-Funktion definiert durch*

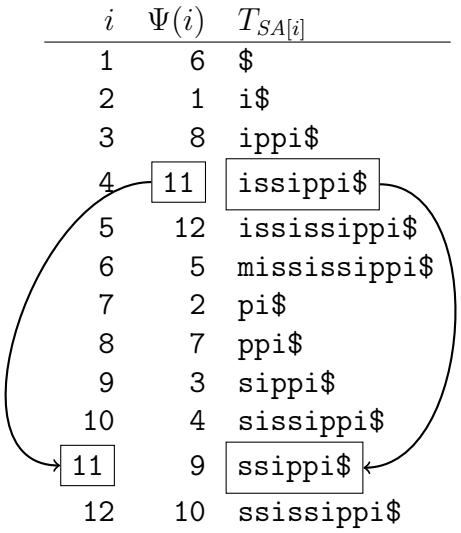
$$\Psi(i) = j \quad \Leftrightarrow \quad SA[j] = SA[i] + 1$$

Umgeformt kann man also schreiben $SA[\Psi(i)] = SA[i] + 1$.

Es wird sich zeigen, dass die Ψ-Funktion einige Eigenschaften besitzt, die sie in verschiedenen Teilen dieser Arbeit nützlich macht.

3.2.2 Die Funktion lookup_C

Zunächst suchen wir nach einer Möglichkeit, wie man von einer Position i im Suffix Array auf den Anfangsbuchstaben von $T_{SA[i]}$ kommt. Da alle Suffixe $T_{SA[i]}$ lexikographisch sortiert sind, gibt es genau $|\Sigma|$ viele Intervalle, in denen jeweils alle Suffixe mit dem selben Buchstaben beginnen. Es genügt also



i	$\Psi(i)$	$T_{SA[i]}$
1	6	$
2	1	i$
3	8	ippi$
4	11	issippi$
5	12	ississippi$
6	5	mississippi$
7	2	pi$
8	7	ppi$
9	3	sippi$
10	4	sissippi$
11	9	ssippi$
12	10	ssissippi$

Abbildung 3.1: Die Ψ-Funktion. Die dritte Spalte dient der Illustration und beinhaltet die entsprechenden Suffixe $T_{SA[i]}$.
In der vierten Zeile dieser Tabelle steht das Suffix „`issippi$`“. Der Wert $\Psi(4) = 11$ bedeutet, dass in der elften Zeile das um eins kürzere Suffix „`ssippi$`“ steht.

für jeden Buchstaben c die Information, an welcher Position das lexikographisch kleinste Suffix steht, welches mit c beginnt. Diese Positionen werden im sogenannten *C-Array* gespeichert.

Beispiel 3.1. *Gegeben sei der String $T =$ „`mississippi$`“ (vgl. Abbildung 3.1). Dann ist das C-Array*

i	*1*	*2*	*3*	*4*	*5*	*6*
$\Sigma[i]$	$	i	m	p	s	
$C[i]$	*1*	*2*	*6*	*7*	*9*	*13*

Um die Notation zu vereinfachen wird dieses Array häufig mit Buchstaben aus Σ adressiert. Dabei gelte für den i-größten Buchstaben $c \in \Sigma$ $C[c] := C[i]$. Außerdem gelte $C[c+1] := C[i+1]$. Damit kann die Funktion lookup_C definiert werden, die für jede Position i den gesuchten Anfangsbuchstaben von $T_{SA[i]}$ liefert.

Definition 6. *Gegeben sei ein C-Array wie oben beschrieben. Dann gilt*

$$\text{lookup}_C(i) = j \quad \Leftrightarrow \quad C[j] \leq i < C[j+1]$$

Da $SA[i]$ die Position im Text angibt, an der das Suffix mit der lexikographischen Ordnung i beginnt, gilt

$$T[SA[i]] = \text{lookup}_C(i)$$

Um den Funktionswert $\text{lookup}_C(i)$ zu berechnen muss also der Index $j \in [1..|\Sigma|]$ gefunden werden, für den die Aussage $C[j] \leq i < C[j+1]$ zutrifft. Dies kann durch eine binäre Suche über das C-Array geschehen, welche in $O(\log|\Sigma|)$ abläuft und ohne zusätzlichen Speicherplatz auskommt. Will man den Wert von lookup_C in konstanter Zeit wissen, so eignet sich hierfür ein Bit-Vektor $B[1..n]$, der genau an den Stellen den Wert 1 hat, an denen der erste Buchstabe von $T_{SA[i]}$ ungleich dem ersten Buchstaben von $T_{SA[i-1]}$ ist. Der Wert $\text{lookup}_C(i)$ ist dann gleich der Anzahl der Einsen in $B[1..i]$, und für die Lösung dieses Problems existieren effiziente Algorithmen ([Jac89]).

3.2.3 Vorwärtsdekodierung

Um bei der Vorwärtssuche das Muster der Länge m mit dem Suffix $T_{SA[i]}$ ohne direkten Zugriff auf den Text T zu vergleichen, benötigt man ein Verfahren um die Buchstaben $T_{SA[i]}[1], T_{SA[i]}[2], \ldots, T_{SA[i]}[m]$ zu berechnen.

Der erste Buchstabe lässt sich direkt mit der Funktion lookup_C ermitteln. Für den zweiten Buchstaben wenden wir einmal die Ψ-Funktion an, um die Position i auf diejenige Position im Suffix Array abzubilden, an der das um eins kürzere Suffix $T_{SA[i]+1}$ steht. Damit ist $T_{SA[i]}[2] = T_{SA[\Psi(i)]}[1]$, womit auch dieser Buchstabe einfach ermittelt werden kann. Allgemein lassen sich die gesuchten Buchstaben berechnen durch

$$T_{SA[i]}[k] \overset{\text{Def. 5}}{=} T_{SA[\Psi^{(k-1)}(i)]}[1] \overset{\text{Def. 6}}{=} \text{lookup}_C(\Psi^{(k-1)}(i))$$

Damit erfolgt der Vergleich $\leq_{\text{lex}}$ des Musters $P[1..m]$ mit dem Suffix $T_{SA[i]}$ wie in Algorithmus 3.2.

3.3 Vorwärtssuche Buchstabe für Buchstabe

Sowohl die Vorwärtssuche nach [MM90] als auch die Vorwärtssuche mit der Ψ-Funktion aus Abschnitt 3.2 bestimmen das Intervall des kompletten Suchmusters P in einem atomaren Schritt. Dieses Vorgehen ist zufriedenstellend,

Algorithmus 3.2 `compare(`$P[1..m]$`,i)`: Vorwärtssuche mit der Ψ-Funktion

```
for k = 1 to m do
  c ← lookup_C(i)
  if P[k] < c then
    return „<“
  else if P[k] > c then
    return „>“
  i ← Ψ(i)
return „=“
```

wenn das Muster vor der Suche bekannt ist, denn dann interessiert nur das Ergebnis, das P-Intervall. Eine komplexere Aufgabe wäre dagegen, zunächst ein Wort ω zu suchen, und als Zwischenergebnis das ω-Intervall $[i..j]$ zu bekommen, um anschließend das Muster um einen Buchstaben c nach rechts zu verlängern, und mit dem Zwischenergebnis das ωc-Intervall zu suchen.

Beispiel 3.2. *Seien ein (längeres) Wort $\omega \in \Sigma^*$ und ein Buchstabe $d \in \Sigma$ fest gegeben. Eine komplexe Aufgabe wäre die Bestimmung aller Vorkommen des Musters $\omega X d$ mit Platzhaltersymbol X, welches für alle Buchstaben aus Σ stehen kann. Dann ist dasjenige Suchverfahren, welches das ω-Intervall als Zwischenergebnis weiterverwenden kann, viel schneller als eines, das für jedes $c \in \Sigma$ eine komplette Suche nach ωcd starten muss.*

Da die Suffixe $T_{SA[i]}$ lexikographisch sortiert sind ist klar, dass das ωc-Intervall ein Teilintervall vom ω-Intervall sein muss. Die Suffixe im ω-Intervall haben nach Definition alle ω als Präfix. Entscheidend für die Berechnung der neuen Intervallgrenzen ist also lediglich die Information „Welcher Buchstabe steht im Suffix $T_{SA[i]}$ an $(|\omega| + 1)$-ter Stelle?“ Da diese Buchstaben alle sortiert sind, eignet sich die binäre Suche. Je nachdem, welche Datenstrukturen zur Vorwärtssuche verwendet werden, lässt sich dieser Buchstabe auf folgende Weisen berechnen.

- Im einfachsten Fall, wenn nämlich das Suffix Array und der Text gegeben sind, ist der k-te Buchstabe im Suffix $T_{SA[i]}$ gleich

$$T_{SA[i]}[k] = T[SA[i] + k - 1]$$

- Auch ohne direkten Zugriff auf den Text kann diese Information gefunden werden. Denn wie oben ist der k-te Buchstabe im Suffix $T_{SA[i]}$

gleich dem ersten Buchstaben von $T_{SA[i]+k-1}$. Das inverse Suffix Array gibt Auskunft über die lexikographische Ordnung dieses Suffixes. Und weiß man die lexikographische Ordnung eines Suffixes, so gibt die Funktion lookup_C genau dessen ersten Buchstaben zurück. Damit ist

$$T_{SA[i]}[k] = \text{lookup}_C(SA^{-1}[SA[i] + k - 1])$$

- Der gesuchte Buchstabe kann wie im letzten Abschnitt auch über die $(k-1)$-malige Anwendung der Funktion Ψ berechnet werden:

$$T_{SA[i]}[k] = \text{lookup}_C(\Psi^{k-1}(i))$$

Möchte man ausgehend vom ω-Intervall $[i..j]$ das ωc-Intervall suchen, so erfordert diese Berechnung in jedem Schritt der binären Suche $|\omega|$ viele Anwendungen der Ψ-Funktion. Die asymptotische Laufzeit gleicht damit der Suche aus dem vorigen Abschnitt, die ohne das Zwischenergebnis $[i..j]$ nach dem Muster ωc sucht. Die Ψ-Funktion eignet sich also nicht zur Vorwärtssuche Buchstabe für Buchstabe.

Vorausgesetzt, dass der k-te Buchstabe eines Suffixes in konstanter Zeit berechnet werden kann, hat diese Art der Vorwärtssuche nach einem Muster der Länge m, die für jeden Buchstaben des Musters durch binäre Suche die beiden Intervallgrenzen berechnet, also die Laufzeit $O(m \log n)$. Algorithmus 3.3 zeigt diese Idee.

Algorithmus 3.3 `search(`$P[1..m]$`)`: Vorwärtssuche Buchstaben für Buchstabe. Die Operationen *min* und *max* stehen für die binäre Suche, die jeweils auf dem Intervall $[i..j]$ beginnt.

$(i, j) \leftarrow (1, n)$
for $k = 1$ **to** m **do**
  $c \leftarrow P[k]$
  $i_c = \min\{l \mid l \geq i \ \wedge \ T_{SA[l]}[k] = c\}$
  $j_c = \max\{l \mid l \leq j \ \wedge \ T_{SA[l]}[k] = c\}$
  if $j_c < i_c$ **then return** $\perp$
  $(i, j) \leftarrow (i_c, j_c)$
return (i, j)

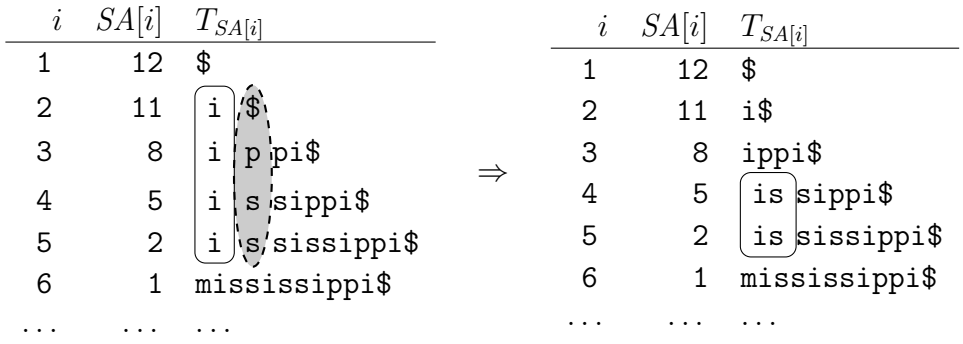


Abbildung 3.2: Vorwärtssuche vom „i“-Intervall zum „is“-Intervall

Beispiel 3.3

In Abbildung 3.2 sieht man einen Ausschnitt des Suffix Arrays von $T =$ „mississippi$“. Es sei das „i“-Intervall [2..5] gegeben und das „is“-Intervall gesucht. Es sind genau die Buchstaben $T_{SA[k]}[2]$ grau hervorgehoben, die bei einer Vorwärtssuche betrachtet werden müssen. Zwei binäre Suchen auf diesen Buchstaben ergeben die neuen Intervallgrenzen [4..5].

3.4 Enhanced Suffix Arrays

Der vorgestellte Algorithmus 3.3, der das Muster Buchstabe für Buchstabe sucht, verwendet zur Bestimmung der Intervalle die binäre Suche, wodurch sich in der Laufzeit stets der $O(\log n)$-Anteil ergibt. In [AKO04] wurde eine Möglichkeit vorgestellt, wie Intervalle durch zusätzliche Verzeichnisinformationen nicht binär gesucht werden müssen, sondern in konstanter Zeit vorliegen. Damit ist die Suche nach einem Muster der Länge m in der Zeit $O(m)$ möglich.

Diese Verbesserung der Vorwärtssuche basiert auf dem Array *LCP*, welches in linearer Zeit aus dem Suffix Array erstellt werden kann. Es enthält an der Stelle i die Länge des längsten gemeinsamen Präfixes (engl. *longest common prefix*) der beiden Suffixe $T_{SA[i-1]}$ und $T_{SA[i]}$ bzw. den Wert -1 an den Stellen 1 und $n+1$. Erweitert man das Suffix Array *SA* um dieses Array *LCP* und geeignete Verzeichnisinformationen, so spricht man vom *Enhanced Suffix Array*. Mit Hilfe der Werte definiert man das *LCP*-Intervall.

Definition 7. *Ein LCP-Intervall l-$[i..j]$ erfüllt folgende Bedingungen:*

- $LCP[i] < l$
- $LCP[k] \geq l \quad$ *für alle* k *mit* $\quad i+1 \leq k \leq j$
- $LCP[k] = l \quad$ *für mindestens ein* k *mit* $\quad i+1 \leq k \leq j$
- $LCP[j+1] < l$

Ein Index k in einem LCP-Intervall l-$[i..j]$ mit $LCP[k] = l$ heißt l-Index.

Beispiel 3.4. *Gegeben sei das Enhanced Suffix Array zum String „`el_anele_lepanelen$`“ (Abbildung 3.3). Das Intervall* [6..11] *ist ein LCP-Intervall mit $l = 1$, denn es gilt $LCP[6] = 0 < 1$, $LCP[k] \geq 1$ für $7 \leq k \leq 11$, $LCP[10] = 1$ und $LCP[12] = 0 < 1$. Alle Suffixe $T_{SA[k]}$ mit $i \leq k \leq j$ haben also ein längstes gemeinsames Präfix der Länge 1. Da $T_{SA[i]} =$ „`e`“ ist das* 1-[6..11]*-Intervall also das „`e`“-Intervall.*

Die Zentrale Operation auf Enhanced Suffix Arrays ist die effiziente Berechnung der Kindintervalle eines *LCP*-Intervalls. Dies geschieht in [AKO04] durch ein Array *childtab*, in welchem die vorberechneten Kindintervalle gespeichert sind. Die in dieser Arbeit vorgestellte Methode basiert auf der Tatsache, dass die Kindintervalle von l-$[i..j]$ entweder an der Stelle i oder an einem l-Index beginnen. Da die l-Indizes Minima des Intervalls $[i+1..j]$ sind, lässt sich das Problem reduzieren auf das Bestimmen dieser Minima. Dazu wird das Array *LCP* in linearer Zeit so vorverarbeitet, dass in konstanter Zeit die Frage nach dem kleinsten Element in einem Intervall beantwortet werden kann (engl. *range minimum query*).

Definition 8 (Range Minimum Query)**.** *Gegeben sei ein LCP-Array. Dann ist*

$$\mathrm{rmq}_{LCP}(i,j) := k \quad \textit{so dass} \quad LCP[k] = \min_{i \leq l \leq j} LCP[l]$$

Existieren mehrere Minima im Intervall $[i..j]$, so sei k das erste davon.

Da sich *LCP*-Intervalle nicht überschneiden können, entsteht über die Beziehung Elternintervall zu Kindintervall eine Baumstruktur. Dieser *LCP*-Intervallbaum (siehe Abbildung 3.4) kann allein durch Operationen auf dem Enhanced Suffix Array in einer Top-Down-Reihenfolge durchlaufen werden. Das Prinzip der Top-Down-Traversierung entspricht dabei der Idee der Vorwärtssuche Buchstabe für Buchstabe.

i	$SA[i]$	$LCP[i]$	$T_{SA[i]}$	LCP-Intervalle
1	19	-1	$	
2	3	0	_anele_lepanelen$	
3	9	1	_lepanelen$	
4	4	0	anele_lepanelen$	
5	13	5	anelen$	
6	8	0	e_lepanelen$	
7	1	1	el_anele_lepanelen$	
8	6	2	ele_lepanelen$	
9	15	3	elen$	
10	17	1	en$	
11	11	1	epanelen$	
12	2	0	l_anele_lepanelen$	
13	7	1	le_lepanelen$	
14	16	2	len$	
15	10	2	lepanelen$	
16	18	0	n$	
17	5	1	nele_lepanelen$	
18	14	4	nelen$	
19	12	0	panelen$	

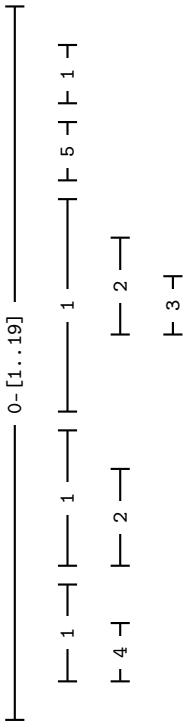


Abbildung 3.3: Das Enhanced Suffix Array zum String „el_anele_lepanelen$“ mit den Spalten SA und LCP. Die letzte Spalte veranschaulicht graphisch alle LCP-Intervalle.

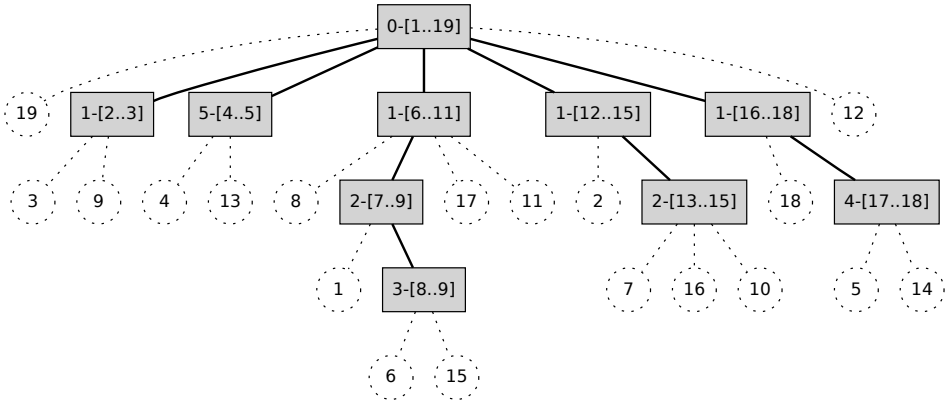


Abbildung 3.4: LCP-Intervallbaum zu $T =$ „`el_anele_lepanelen$`“. Gestrichelt angedeutet sind die sog. Singleton-Intervalle. Dies sind nur 1 breit und haben somit keinen l-Wert.

Beispiel 3.5

Gegeben sei der Text $T =$ „`el_anele_lepanelen$`“ und das entsprechende Enhanced Suffix Array aus Abbildung 3.3. Außerdem sei das Array LCP so vorverarbeitet, dass die *range minimum query* effizient beantwortet werden kann. Es soll nach dem Muster „`ele`“ gesucht werden.

1. Gestartet wird mit dem Wurzelintervall 0-[1..19], welches sieben l-Indizes besitzt.

   - $\text{rmq}_{LCP}(2, 19) = 2 \quad \Rightarrow$ Das erste Kindintervall ist [1..1] und steht für den Buchstaben $T_{SA[1]}[1] =$ „`$`“.
   - $\text{rmq}_{LCP}(3, 19) = 4 \quad \Rightarrow$ Das zweite Kindintervall ist [2..3] und steht für den Buchstaben $T_{SA[2]}[1] =$ „`_`“.
   - $\text{rmq}_{LCP}(5, 19) = 6 \quad \Rightarrow$ Das dritte Kindintervall ist [4..5] und steht für den Buchstaben $T_{SA[4]}[1] =$ „`a`“.
   - $\text{rmq}_{LCP}(7, 19) = 12 \quad \Rightarrow$ Das vierte Kindintervall ist [6..11] und steht für den Buchstaben $T_{SA[6]}[1] =$ „`e`“. Da $\min_{(6<k\leq 11)} LCP[k] = 1$, handelt es sich um ein LCP-Intervall mit $l = 1$. Es ist das gesuchte „`e`“-Intervall.

2. Ausgehend vom „e“-Intervall 1-[6..11] wird nach dem „el“-Intervall gesucht.

   - $\text{rmq}_{LCP}(7, 11) = 7 \quad \Rightarrow$ Das erste Kindintervall ist [6..6] und steht für den Buchstaben $T_{SA[6]}[2] =$ „␣“.
   - $\text{rmq}_{LCP}(8, 11) = 10 \quad \Rightarrow$ Das zweite Kindintervall ist [7..9] und steht für den Buchstaben $T_{SA[7]}[2] =$ „l“. Da $\min_{(7<k\leq 9)} LCP[k] = 2$, handelt es sich um ein LCP-Intervall mit $l = 2$. Es ist das gesuchte „el“-Intervall.

3. Im letzten Schritt wird das „ele“-Intervall bestimmt.

   - $\text{rmq}_{LCP}(8, 9) = 8 \quad \Rightarrow$ Das erste Kindintervall ist [7..7] und steht für den Buchstaben $T_{SA[7]}[3] =$ „␣“.
   - Da 8 der einzige l-Index im LCP-Intervall 2-[7..9] ist, ist das zweite und letzte Kindintervall [8..9] und steht für den Buchstaben $T_{SA[8]}[3] =$ „e“. Da $\min_{(8<k\leq 9)} LCP[k] = 3$, handelt es sich um ein LCP-Intervall mit $l = 3$. Es ist das gesuchte „ele“-Intervall.

Kapitel 4

Rückwärtssuche

Eine komplett andere Herangehensweise an das Suchproblem ist die Rückwärtssuche. Dabei wird das Muster vom letzten Buchstaben an bis zum ersten abgearbeitet. Zunächst wird also das $P[m]$-Intervall bestimmt. Ausgehend davon erfolgt in jedem Suchschritt vom $P[k..m]$-Intervall zum $P[k\text{-}1..m]$-Intervall. Die Rückwärtssuche arbeitet also bereits per Definition „Buchstabe für Buchstabe“.

4.1 Rückwärtssuche mit der Funktion *Occ*

Die Burrows-Wheeler-Transformation ([BW94]) wurde ursprünglich als erste Phase eines Verfahrens zur Datenkompression entwickelt. Es ist eine große Ähnlichkeit zum Suffix Array gegeben, denn diese umkehrbare Transformation erfolgt auch hier über eine Sortierung der Suffixe. Die Burrows-Wheeler-Transformation enthält an der Stelle i den Buchstaben, der im Text direkt vor dem Suffix mit der lexikographischen Ordnung i steht.

Definition 9 (Burrows-Wheeler-Transformation)**.** *Gegeben sei ein Text* $T[1..n]$ *mit* $T[n] =$ „$$“ *und sein Suffix Array SA. Das BWT-Array ist definiert durch* $BWT[i] =$ „$$“ *falls* $SA[i] = 1$ *und* $BWT[i] = T\,[SA[i] - 1]$ *sonst.*

Im Jahr 2000 wurde ein neuartiger Index unter dem Namen *opportunistic data structure* ([FM00]) vorgestellt, welcher in der Literatur unter dem Namen *FM-Index* bekannt wurde. Die Suche erfolgt in Rückwärtsrichtung, ohne dabei das Suffix Array selbst zu verwenden.

Zunächst benötigt man dazu wieder das Array C aus Abschnitt 3.2, welches für jeden Buchstaben $c \in \Sigma$ angibt, an welcher Position im Suffix Array

das lexikographisch kleinste Suffix steht, das mit c beginnt. Damit ist der erste Suchschritt nach dem Buchstaben $P[m]$ trivial. Ausgehend von einem ω-Intervall funktioniert ein Schritt der Rückwärtssuche nach $c\omega$ wie folgt.

Wenn man in jeder Zeile k den Eintrag $BWT[k]$ und $T_{SA[k]}$ konkateniert, erhält man das Suffix $T_{SA[k]-1}$. Wir betrachten zunächst die konkatenierten Suffixe $T_{SA[k]-1}$, die mit c beginnen, für die also gilt $BWT[k] = c$. Da sie im ersten Buchstaben übereinstimmen und die regulären Suffixe $T_{SA[k]}$ lexikographisch sortiert sind, sind auch diese konkatenierten Suffixe sortiert. Es handelt sich also um die selben Suffixe in der selben Reihenfolge wie die regulären Suffixe $T_{SA[k]}$ aus dem c-Intervall.

Wenn $[i..j]$ das ω-Intervall ist, dann weiß man auch, dass die Suffixe $T_{SA[k]}$ mit $i \leq k \leq j$ die einzigen sind, die ω als Präfix haben. Damit sind die konkatenierten Suffixe $T_{SA[k]-1}$ mit $i \leq k \leq j$ die einzigen Kandidaten, die mit $c\omega$ beginnen können. Es genügt also für die Rückwärtssuche nur die Spalte BWT zu betrachten. Die Anzahl der Vorkommen des Wertes c in $BWT[1..i\text{-}1]$ entspricht der Anzahl der Suffixe im c-Intervall, die lexikographisch echt kleiner sind als das gesuchte Muster $c\omega$. Und die Anzahl der Vorkommen des Wertes c in $BWT[i..j]$ entspricht der Anzahl der Suffixe im c-Intervall, die mit $c\omega$ beginnen. Das neue $c\omega$-Intervall $[p..q]$ berechnet sich nun also durch

$$p = C[c] + Occ(c, i-1)$$
$$q = C[c] + Occ(c, j) - 1$$

wobei die Funktion $Occ(c, i)$ angibt, wie oft der Wert von c im Array BWT bis einschließlich zur Position i vorkommt. Damit ergibt sich der Algorithmus 4.1.

Algorithmus 4.1 Rückwärtssuche nach $P[1..m]$ mit der Funktion Occ

```
(i, j) ← (1, n)
for k = m to 1 do
  c ← P[k]
  i ← C[c] + Occ(c, i − 1)
  j ← C[c] + Occ(c, j) − 1
  if j > i then return ⊥
return [i..j]
```

i	$BWT[i]$	$T_{SA[i]}$
1	n	$
2	l	_anele_lepanelen$
3	e	_lepanelen$
4	_	anele_lepanelen$
5	p	anelen$
6	l	e_lepanelen$
7	$	el_anele_lepanelen$
8	n	ele_lepanelen$
9	n	elen$
10	l	en$
11	l	epanelen$
12	e	l_anele_lepanelen$
13	e	le_lepanelen$
14	e	len$
15	_	lepanelen$
16	e	n$
17	a	nele_lepanelen$
18	a	nelen$
19	e	panelen$

⇒

i	$BWT[i]$	$T_{SA[i]}$
1	n	$
2	l	_anele_lepanelen$
3	e	_lepanelen$
4	_	anele_lepanelen$
5	p	anelen$
6	l	e_lepanelen$
7	$	el_anele_lepanelen$
8	n	ele_lepanelen$
9	n	elen$
10	l	en$
11	l	epanelen$
12	e	l_anele_lepanelen$
13	e	le_lepanelen$
14	e	len$
15	_	lepanelen$
16	e	n$
17	a	nele_lepanelen$
18	a	nelen$
19	e	panelen$

Abbildung 4.1: Ein Schritt der Rückwärtssuche vom „le“-Intervall [13..15] (links) nach dem Buchstaben „e“. Alle Kandidaten k mit $BWT[k] =$ „e“ sind markiert. Die beiden Kandidaten im Intervall [13..15] (grau) entsprechen genau den Suffixen, die mit „ele“ beginnen. Damit ist das „ele“-Intervall gleich [8..9] (rechts).

Beispiel 4.1

Gegeben sei der Text $T =$ „el_anele_lepanelen$“ und sein Suffix Array. Zunächst wird die Burrows-Wheeler-Transformation erstellt und so vorverarbeitet, dass die Anfrage $Occ(c, i)$ beantwortet werden kann (siehe hierzu Abbildung 4.1). Außerdem wird das C-Array erstellt (Tabelle 4.1).

Es soll nun das Muster $P[1..3] =$ „ele“ gesucht werden.

1. Da $P[3] =$ „e“ wird also zunächst das „e“-Intervall $[i_1..j_1]$ bestimmt durch

$$i_1 = C[e] = 6 \quad \text{und}$$
$$j_1 = C[l] - 1 = 12 - 1 = 11$$

i	1	2	3	4	5	6	7	8
$\Sigma[i]$	\$	_	a	e	l	n	p	
$C[i]$	1	2	4	6	12	16	19	20

Tabelle 4.1: C-Array zu $T =$ „el_anele_lepanelen$“

2. Zur Bestimmung des neuen „le“-Intervalls werden alle Vorkommen von „l“ in der Burrows-Wheeler-Transformation vor der linken (Diese gehören zwar auch zu Suffixen die im „l“-Intervall liegen, aber sie beginnen nicht mit „le“) und bis zur rechten Intervallgrenze von [6..11] gezählt. Das neue Intervall $[i_2..j_2]$ entspricht also

$$i_2 = C[l] + Occ(l, 5) = 12 + 1 = 13 \quad \text{und}$$
$$j_2 = C[l] + Occ(l, 11) - 1 = 12 + 4 - 1 = 15$$

3. Der letzte Schritt (in Abbildung 4.1 illustriert) erweitert das Muster nach links um den Buchstaben „e“ und es ergibt sich das Ergebnis $[i_3..j_3]$ durch

$$i_3 = C[e] + Occ(e, 12) = 6 + 2 = 8 \quad \text{und}$$
$$j_3 = C[e] + Occ(e, 15) - 1 = 6 + 4 - 1 = 9$$

Das Resultat der Rückwärtssuche ist das Intervall [8..9]. Das Muster „ele“ taucht im Text T also zweimal auf, und zwar an den Positionen $SA[8] = 6$ und $SA[9] = 15$. Die Bestimmung des gesuchten Intervalls auf dem FM-Index funktioniert ohne das Suffix Array. Man benötigt es jedoch, um aus dem gefundenen Intervall die Positionen im Text zu ermitteln, an denen das Muster vorkommt. Es wäre natürlich gut, wenn man dafür nicht das komplette Array speichern müsste. Wie man trotzdem an die Werte des Suffix Arrays gelangt, wird im Abschnitt 5.2.2 besprochen.

4.2 Die *Occ*-Funktion

Die *Occ*-Funktion lässt sich auf sehr verschiedene Weisen implementieren, welche sich besonders in Speicherplatzverbrauch und Laufzeit unterscheiden. Dabei gilt die Faustregel: je mehr Speicherplatz man investiert, desto geringer ist die Antwortzeit einer Datenstruktur (engl. *Time-Space-Tradeoff*).

4.2.1 Naive Verfahren

Das einfachste Verfahren zur Beantwortung der Anfrage $Occ(c, i)$ ist, sich die Burrows-Wheeler-Transformierte in einem Array zu speichern und jedes mal in einer Schleife von 1 bis i durchzuzählen, wie oft der Buchstabe „c“ vorkommt. Da ein Eintrag im Array $\log_2(|\Sigma|)$ breit, ist ist der Speicherplatzverbrauch für das BWT-Array $O(n \log |\Sigma|)$, die Antwortzeit beträgt $O(n)$. Das andere Extrem wäre das Abspeichern aller Antworten in einem Array. Zwar hat man hier eine Antwortzeit von $O(1)$ – allerdings auf Kosten der Größe von $O(|\Sigma| \cdot n)$ gespeicherten Antworten. Wie bereits der Name sagt, haben diese naiven Verfahren auf Grund ihrer Laufzeit bzw. Größe keine praktische Bedeutung. Im folgenden werden zwei Verfahren näher betrachtet, die zwischen diesen beiden Extremen liegen.

4.2.2 Indikator-Arrays

Möchte man konstante Antwortzeit gewährleisten, so kann man sich dafür $|\Sigma|$ Bit-Vektoren B_c definieren mit $B_c[i] = 1$ gdw. $BWT[i] = c$. Damit ist das Problem reduziert auf das Zählen von Bits, sogenannten rank_1-Anfragen.

i					5					10					15				
BWT	n	l	e	_	p	l	$	n	n	l	l	e	e	e	_	e	a	a	e
$B_\$$	0	0	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0
$B_\_$	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	**1**	0	0	0	0
B_a	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**1**	0
B_e	0	0	**1**	0	0	0	0	0	0	0	0	**1**	**1**	**1**	0	**1**	0	0	**1**
B_l	0	**1**	0	0	0	**1**	0	0	0	**1**	**1**	0	0	0	0	0	0	0	0
B_n	**1**	0	0	0	0	0	0	**1**	**1**	0	0	0	0	0	0	0	0	0	0
B_p	0	0	0	0	**1**	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Tabelle 4.2: Indikator-Arrays zu T^{BWT} = „nle_pl$nnlleee_eaae“

Beispiel 4.2. *Gegeben seien die Bit-Vektoren aus Tabelle 4.2. Dann ist*

$$Occ(e, 16) = \text{rank}_1(B_e, 16) = 5$$

Bis einschließlich zur Position 16 *kommt der Wert „e“ also* 5 *mal vor.*

Für die rank_1-Anfrage auf Bit-Vektoren der Länge n existieren Lösungen, die mit $o(n)$ Bits zusätzlichem Speicherplatz Antworten in konstanter Zeit

liefern ([Jac89]). Die Rückwärtssuche nach einem Muster der Länge m hat in dieser Variante damit eine optimale Gesamtlaufzeit von $O(m)$ bei einem Speicherverbrauch von $O(n \cdot |\Sigma|)$ Bits.

4.2.3 Wavelet Tree

Besonders bei großen Alphabeten ist die lineare Abhängigkeit des Speicherplatzes von $|\Sigma|$ nicht wünschenswert. Ein Ansatz ist die Benutzung eines *Wavelet Trees*, welcher erstmals in [GGV03] vorgestellt wurde.

Definition 10 (Wavelet Tree)**.** *Gegeben sei ein Text T über dem Alphabet Σ. Dann ist ein Wavelet Tree ein balancierter binärer Suchbaum mit folgenden Eigenschaften.*

- *Jedem Knoten eines Wavelet Trees ist ein Alphabetintervall $[l..r]$ in Σ zugeordnet und er repräsentiert den String $T^{[l..r]}$. Das ist die Teilfolge von T, die nur aus den Buchstaben l bis r besteht.*
- *Die Wurzel des Wavelet Tree steht für das Alphabetintervall $[1..|\Sigma|]$*
- *Ist einem Knoten das Alphabetintervall $[k..k]$ zugeordnet, so besitzt er keine Kinder. Ansonsten repräsentiert sein linkes Kind den String $T^{[l..m]}$, und sein rechtes Kind den String $T^{[m+1..r]}$, wobei $m = \lfloor \frac{l+r}{2} \rfloor$ ist.*

An Stelle des gesamten Strings $T^{[l..r]}$ wird in jedem Knoten nur in einem Bit-Vektor $B^{[l..r]}$ gespeichert, ob der Buchstabe an der entsprechenden Stelle zur ersten oder zur zweiten Hälfte des Alphabetintervalls $[l..r]$ gehört.

In Abbildung 4.2 sieht man den Wavelet Tree der Burrows-Wheeler-Transformation des Textes „`el_anele_lepanelen$`“. Da jeder innere Knoten verzweigend ist, ist die Höhe des Baumes $\lceil \log_2 |\Sigma| \rceil$. Auf jeder Ebene des Baumes werden höchstens n Bits gespeichert. Rechnet man den zusätzlichen Speicherplatz dazu, der für die Beantwortung der rank_1-Anfragen in konstanter Zeit nötig ist, so benötigt der gesamte Baum $n \log |\Sigma| + o(n \log |\Sigma|)$ Bits Speicher.

Die Berechnung des Funktionswerts $Occ(c, i)$ erfolgt durch einen Top-Down-Durchlauf des Wavelet Trees und beginnt an der Wurzel mit dem Alphabetintervall $[l..r] = [1..|\Sigma|]$. Befindet sich c nun in der ersten Hälfte des betrachteten Teilalphabets, so steht die Antwort weiter unten in dem

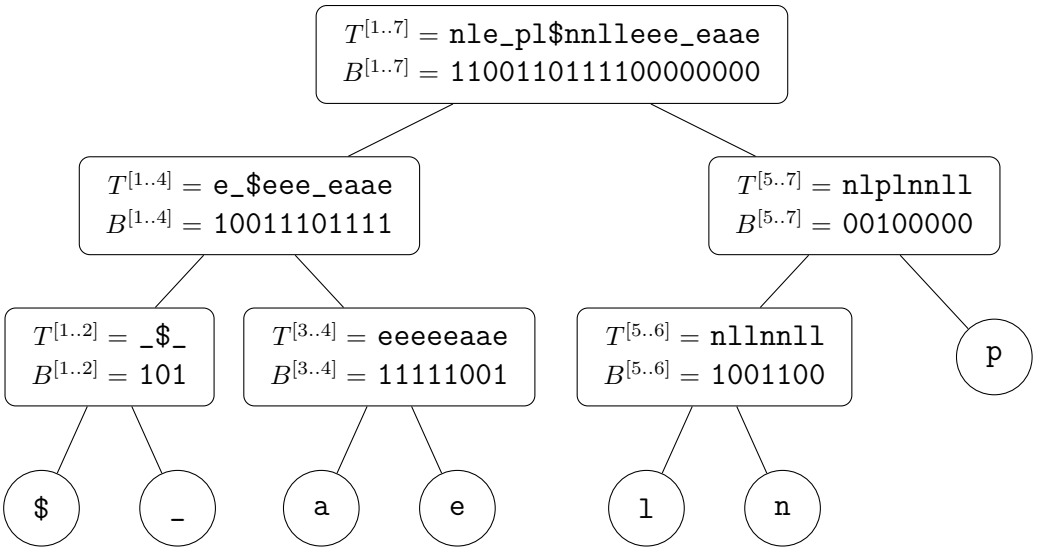


Abbildung 4.2: Wavelet Tree zu T^{BWT} = „nle_pl$nnlleee_eaae“

Teilbaum, dessen Wurzel das linke Kind vom Knoten $[l..r]$ ist. Da alle Buchstaben des linken Kindknotens in $B^{[l..r]}$ durch eine 0 repräsentiert sind, errechnet sich die Antwort rekursiv aus dem linken Teilbaum an der Stelle $\text{rank}_0(B^{[l..r]}, i)$. Falls sich der Buchstabe c zur zweiten Hälfte des Alphabetintervalls gehört, so steht die Antwort analog im rechten Teilbaum an der Stelle $\text{rank}_1(B^{[l..r]}, i)$. Diese Prozedur setzt sich rekursiv bis zum Blattknoten $[c..c]$ fort. Da der String in einem Blattknoten nur noch aus einem Buchstaben besteht, ist dort die Berechnung der *Occ*-Funktion trivial, denn in diesem Fall gilt $Occ'(c, i, [c..c]) = i$. Die gesamte Berechnung von $Occ(c, i) = Occ'(c, i, [1..|\Sigma|])$ wird beschrieben durch Algorithmus 4.2.

Algorithmus 4.2 Die Funktion $Occ'(c, i, [l..r])$ auf einem Wavelet Tree

/* $[l..r]$ ist das Alphabetintervall */

if $l = r$ **then**
  return i
else
  $m = \lfloor \frac{l+r}{2} \rfloor$
  if $c \leq m$ **then**
    return $Occ'(c, \text{rank}_0(B^{[l..r]}, i), [l..m])$
  else
    return $Occ'(c, \text{rank}_1(B^{[l..r]}, i), [m+1..r])$

Beispiel 4.3. *Gegeben sei der Wavelet Tree aus Abbildung 4.2. Dann ist*

$$Occ(e,16) = Occ'(e,16,[1..7]) = Occ'(e,\underbrace{\text{rank}_0(B^{[1..7]},16)}_{=8},[1..4]) =$$
$$Occ'(e,\underbrace{\text{rank}_1(B^{[1..4]},8)}_{=5},[3..4]) = Occ'(e,\underbrace{\text{rank}_1(B^{[3..4]},5)}_{=5},[4..4]) = 5$$

Da die Tiefe eines Blattknotens höchstens $\lceil \log_2 |\Sigma| \rceil$ ist, erreicht die Rückwärtssuche nach einem Muster der Länge m mit dem Wavelet Tree eine Gesamtlaufzeit von $O(m \log |\Sigma|)$.

4.3 Rückwärtssuche mit der Ψ-Funktion

Um mit der Ψ-Funktion Rückwärtssuche betreiben zu können muss man zunächst folgende Beobachtung festhalten. Für jedes $c \in \Sigma$ existiert ein Intervall $[i_c..j_c]$, so dass alle Suffixe $T_{SA[k]}$ mit $i_c \leq k \leq j_c$ mit dem Buchstaben c beginnen. Da die Ψ-Funktion an der Stelle k die Position des Suffixes $T_{SA[k]+1}$ angibt, und da die Suffixe in diesem Intervall lexikographisch sortiert sind, ist sie in diesem Intervall ansteigend. Es gilt also $\Psi(i_c) < \ldots < \Psi(j_c)$.

Sei nun $[i..j]$ das ω-Intervall. Ein Schritt der Rückwärtssuche nach einem Buchstaben c funktioniert dann folgendermaßen. Das gesuchte Intervall muss ein Teilintervall vom c-Intervall $[i_c..j_c]$ sein, da die Suffixe alle mit $c\omega$ beginnen sollen. Liegt nun der Wert der Ψ-Funktion an einer Stelle k zwischen i und j, so bedeutet das, dass sich das Suffix $T_{SA[k]+1}$ im ω-Intervall befindet. Genau die Stellen k im c-Intervall, für die gilt $i \leq \Psi(k) \leq j$, stehen also für diejenigen Suffixe, die mit $c\omega$ beginnen. Sie bilden also das gesuchte $c\omega$-Intervall. Die Grenzen dieses Intervalls können durch binäre Suche über die Werte der Ψ-Funktion, gestartet mit dem Intervall $[i_c..j_c]$, in $O(\log n)$ Schritten bestimmt werden.

Eine Suche nach dem Muster P der Länge m startet nun mit $[1..n]$, also dem Intervall des leeren Wortes. Beginnend mit $P[m]$ wird für dieses Muster Buchstabe für Buchstabe dieser Rückwärtsschritt durchgeführt bis das P-Intervall gefunden ist. Algorithmus 4.3 beschreibt dieses Vorgehen im Detail. Die Rückwärtssuche nach $P[1..m]$ benötigt also m Suchschritte mit jeweils zwei binären Suchen. Es ergibt sich die Gesamtkomplexität von $O(m \log n)$.

Auch bei dieser Methode wird zur Bestimmung des gesuchten Intervalls das Suffix Array nicht gebraucht. Will man jedoch anschließend die Posi-

Algorithmus 4.3 Rückwärtssuche mit der Ψ-Funktion. Die Operationen *min* und *max* stehen für die binäre Suche, die jeweils auf dem Intervall $[C[c]..C[c+1]-1]$ beginnt.

$(i,j) \leftarrow (1,n)$
for $i = m$ **down to** 1 **do**
  $c \leftarrow P[i]$
  $i_c = \min\{k \mid k \geq C[c] \ \wedge \ \Psi(k) \in [i..j]\}$
  $j_c = \max\{k \mid k < C[c+1] \ \wedge \ \Psi(k) \in [i..j]\}$
  if $j_c < i_c$ **then return** $\perp$
  $(i,j) \leftarrow (i_c, j_c)$
return $[i..j]$

tionen der gefundenen Vorkommen des Musters ermitteln, so benötigt man die Werte des Suffix Arrays in diesem Intervall. In Abschnitt 5.2.1 wird beschrieben, wie man diese *SA*-Werte mit Hilfe der Ψ-Funktion und wenigen abgespeicherten Einträgen von *SA* berechnen kann.

Beispiel 4.4

Abbildung 4.3 veranschaulicht an der Ψ-Funktion des Textes „`mississippi$`“ einen Schritt der Rückwärtssuche. Ausgehend vom „`i`“-Intervall [2..5] (in der linken Tabelle grau hervorgehoben) soll das „`si`“-Intervall berechnet werden. Dazu wird zunächst das „`s`“-Intervall [9..12] bestimmt (leicht hervorgehoben). Die Werte der Ψ-Funktion im „`s`“-Intervall sind $3, 4, 9$ und 10. Da die Werte der Ψ-Funktion an den ersten beiden Stellen im „`i`“-Intervall liegen, bilden diese zusammen das neue „`si`“-Intervall (rechte Tabelle).

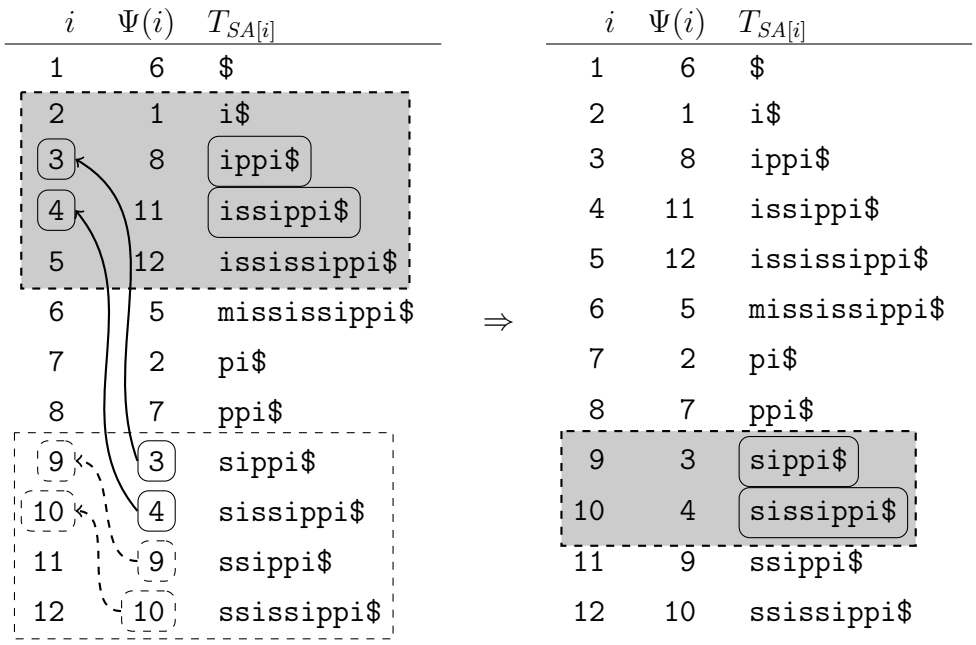


Abbildung 4.3: Rückwärtssuche mit der Ψ-Funktion

Kapitel 5

Bidirektionale Suche

Stellt man an einen Index nur die Anforderung, dass er ein Muster in einem gegebenen Text effizient suchen können muss, so ist es gleichgültig, ob er dazu Vorwärts- oder Rückwärtssuche verwendet, denn dann interessiert nur das Ergebnis. Es gibt aber auch komplexere Fragestellungen als diese. Wir haben Indexe kennen gelernt, die ausgehend von einem gefundenen Muster die Menge der Vorkommen weiter einschränken, indem sie dieses Muster entweder auf der rechten oder auf der linken Seite um einen Buchstaben verlängern. In diesem Kapitel soll versucht werden, einen Index zu konstruieren, der die Verlängerung des Musters auf beiden Seiten, und somit bidirektionale Suche unterstützt.

5.1 Kombination von Vorwärts- und Rückwärtssuche

In den Kapiteln 3 und 4 wurden Indexe vorgestellt, die jeweils nur eines dieser beiden Konzepte unterstützen. Sie haben aber alle gemeinsam, dass eine Suche nach einem Muster über die Bestimmung eines Intervalls im Suffix Array funktioniert. Man kann also für einen gegebenen Text einen Index für die Vorwärtssuche und einen Index für die Rückwärtssuche erstellen, und die Kombination davon ist ein Index, der bidirektionale Suche unterstützt. Es sind viele verschiedene Kombinationen für die bidirektionale Suche denkbar, z.B.

- Das Suffix Array SA und der Text T für die Vorwärtssuche, und die Ψ-Funktion für die Rückwärtssuche.

- Das Suffix Array SA und der Text T für die Vorwärtssuche, und die Funktion Occ für die Rückwärtssuche.

- Mit der Gleichung $\Psi(i) := SA^{-1}[SA[i]+1]$ genügt das Suffix Array und das inverse Suffix Array zur Vorwärts- und Rückwärtssuche.

Diese Kombinationen haben alle gemein, dass entweder die Vorwärts- oder die Rückwärtssuche nicht in optimaler Zeit funktioniert, also die Laufzeit der Suche nach einem Muster der Länge m langsamer ist als $O(m)$. Einzig die Kombination vom Enhanced Suffix Array und der Rückwärtssuche mit der Funktion Occ bietet lineare Laufzeit in beide Richtungen, sie hat aber einen sehr hohen Speicherplatzbedarf.

5.2 Compressed Suffix Arrays

Falls der Text sehr groß wird, steigt jedoch auch der Speicherplatzbedarf dieser Indexdatenstrukturen stark an. Im Jahr 2000 wurden die ersten Versuche unternommen ([GV00, Mä00]), gewisse „Regularitäten“ im Suffix Array auszunutzen und das Suffix Array in einer komprimierten Form abzuspeichern. Der Preis dafür ist der erhöhte Zeitaufwand beim Zugriff auf seine Werte. Das Ergebnis davon ist das *komprimierte Suffix Array* (engl. *Compressed Suffix Array*, CSA). Im folgenden Unterabschnitt wird zunächst ein CSA vorgestellt, welches seine SA-Werte mit Hilfe der Ψ-Funktion berechnet. Anschließend wird eine Datenstruktur entwickelt, die ähnlich funktioniert und auf der Funktion Occ basiert. Beide Varianten bieten zudem Zugriff auf die Werte vom inversen Suffix Array. Damit ist auf beiden Indexen bidirektionale Suche möglich.

5.2.1 CSA durch die Ψ-Funktion

Sadakane stellt in [Sad03] ein komprimiertes Suffix Array vor, welches im Wesentlichen auf der Ψ-Funktion basiert, und gleichzeitig den Zugriff auf die Werte SA und SA^{-1} gibt. Es gilt nach der Definition der Ψ-Funktion

$$SA[\Psi(i)] = SA[i] + 1 \tag{5.1}$$

Wendet man auf beiden Seiten der Gleichung das inverse Suffix Array

SA^{-1} an, und substituiert man i durch $SA^{-1}[i]$, so erhält man nach Vereinfachung die Gleichung

$$\Psi(SA^{-1}[i]) = SA^{-1}[i+1] \tag{5.2}$$

Damit dient die Ψ-Funktion nicht mehr nur als Abbildung von einem Suffix auf das um einen Buchstaben kürzere nächste Suffix (Gleichung 5.1), sondern auch als Beziehung zwischen zwei aufeinander folgenden Werten von SA^{-1}. Ist ein Wert $SA^{-1}[i]$ gegeben, so kann durch $(j-i)$-maliges Anwenden der Ψ-Funktion der Wert $SA^{-1}[j]$ für alle $j > i$ berechnet werden (Gleichung 5.2). Speichert man nur jeden k-ten Eintrag der Arrays SA und SA^{-1}, so verringert sich der Speicherplatzbedarf von jeweils $O(n \log n)$ Bits auf $O(\frac{n \log n}{k})$ Bits. Wählt man $k = \log n$, so benötigen die beiden Arrays SA und SA^{-1} nur noch linear viele Bits Speicher. Der Preis dafür ist, dass die Werte der beiden Arrays nicht mehr direkt zur Verfügung stehen, sondern erst mit $O(\log n)$ Schritten berechnet werden müssen. Die Funktion Ψ kann dabei komprimiert abgespeichert werden, so dass sie nur noch $O(n)$ Bits Speicher benötigt und mit Hilfe der „Vier-Russen-Methode“ trotzdem konstante Zugriffszeit bietet ([Sad03]).

Die Ψ-Funktion alleine ermöglicht ja bereits die Rückwärtssuche (Abschnitt 4.3). Kombiniert mit den Samples der Arrays SA und SA^{-1} lassen sich alle Werte dieser beiden Arrays berechnen, man kann also wie in Abschnitt 3.3 auch in Vorwärtsrichtung suchen. Damit hat man einen Index zur bidirektionalen Suche mit $O(n)$ Bits Speicherplatzbedarf. Dieser ist jedoch besonders bei der Vorwärtssuche nicht optimal, da in jedem Schritt der binären Suche ein SA- und ein SA^{-1}-Wert berechnet werden muss. Damit besitzt die Suche in Vorwärtsrichtung die Laufzeit $O(\log^2 n)$ pro Buchstabe.

5.2.2 CSA durch die *Occ*-Funktion

Genau wie die Ψ-Funktion kann auch die Funktion Occ dazu verwendet werden, die Einträge in den Arrays SA und SA^{-1} aus gespeicherten Samples dieser Arrays zu berechnen. Dazu definiert man die Funktion

$$LF(i) = C[BWT[i]] + Occ(BWT[i], i)$$

An der Stelle i im BWT-Array steht der Buchstabe, der dem Suffix $T_{SA[i]}$ im Text vorausgeht. Der Teilausdruck $Occ(BWT[i], i)$ gibt also an, wie viele

Suffixe, die lexikographisch kleiner oder gleich dem Suffix $T_{SA[i]}$ sind, ebenfalls im Text direkt hinter diesem Buchstaben beginnen. Somit bildet diese Funktion das Suffix $T_{SA[i]}$ auf $T_{SA[i]-1}$ ab. Es handelt sich also um die Umkehrfunktion von Ψ. Die Beziehungen

$$SA[LF(i)] = SA[i] - 1 \quad \text{und} \tag{5.3}$$

$$SA^{-1}[i-1] = LF(SA^{-1}[i]) \tag{5.4}$$

erlauben also die Berechnung aller Werte von SA und SA^{-1} aus wenigen abgespeicherten Einträgen wie in Abschnitt 5.2.1. Damit handelt es sich hierbei ebenfalls um einen Index zur bidirektionalen Suche, denn die Arrays SA und SA^{-1} genügen zur Vorwärtssuche, und die Funktion Occ zur Rückwärtssuche. Aber auch dieser Index ist noch nicht optimal, da der Zugriff auf die Werte von SA und SA^{-1} und somit die Vorwärtssuche in der Laufzeit stark von n abhängt.

5.3 Affixbaum und Affix Array

Lange Zeit war die grundlegende Datenstruktur für sehr viele Algorithmen, die sich mit der Verarbeitung von Strings beschäftigen, der *Suffixbaum* ([Wei73]). Dieser eignet sich zur effizienten Lösung vieler verschiedener Probleme, zum Beispiel auch zur Mustersuche. Als Verallgemeinerung des Suffixbaumes wurde in [Sto95] der *Affixbaum* vorgestellt, der sich zur bidirektionalen Suche eignet.

5.3.1 Suffixbaum

Ein *atomarer Suffixbaum* eines Textes T der Länge n ist ein gerichteter Suchbaum mit n Blättern. Jede Kante ist mit einem Buchstaben aus T beschriftet. Kein Knoten hat zwei ausgehende Kanten mit dem selben Buchstaben. Die Blätter sind mit $i \in [1..n]$ beschriftet. Konkateniert man die Buchstaben auf dem Pfad von der Wurzel bis zu einem Blatt i, so erhält man das Suffix T_i.

Im Worst Case hat der atomare Suffixbaum quadratisch viele Knoten. Deshalb verwendet man in der Praxis den *kompakten Suffixbaum*. Das besondere am kompakten Suffixbaum ist, dass Knoten, die nicht verzweigen, zusammengefasst werden. Die Kanten sind somit beschriftet mit Teilwörtern

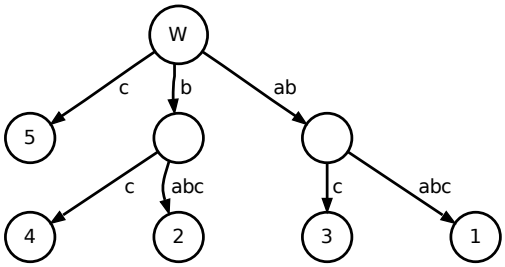


Abbildung 5.1: Kompakter Suffixbaum zu $T =$ „ababc“

von T. Da ein Baum mit n Blättern, in dem sich alle inneren Knoten verzweigen, höchstens $2n - 1$ Knoten besitzt, lassen sich Suffixbäume mit linearem Platzbedarf abspeichern. Es existieren außerdem Algorithmen, welche einen Suffixbaum in linearer Laufzeit erstellen.

Möchte man die Frage „Ist P ein Teilstring von T?“ mit Hilfe des Suffixbaumes untersuchen, so folgt man von der Wurzel aus genau dem Pfad, dessen Kanten die entsprechende zu P passende Beschriftung vorweisen (siehe Abbildung 5.1). Kommt man bei der Suche an einen Punkt, an dem keine passend beschriftete Kante existiert, so ist das Muster nicht in T enthalten.

Beispiel 5.1

Gegeben sei der kompakte Suffixbaum aus Abbildung 5.1.

- Möchte man das Muster „bc“ suchen, so startet man bei der Wurzel Ⓦ, folgt nacheinander den Kanten „b“ und „c“ und landet beim Knoten ④. Dies bedeutet, dass das Muster „bc“ an der Stelle 4 im Text vorkommt.
- Die Suche nach „ab“ resultiert in einem inneren Knoten. Da sich dieser verzweigt gibt es mehrere Suffixe, welche mit „ab“ beginnen. Die Blätter ③ und ① in diesem Teilbaum geben also genau die Positionen an, an denen das Muster „ab“ auftritt.

5.3.2 Affixbaum

Der Suffixbaum eignet sich also hervorragend als Index zur Suche nach einem Muster in Vorwärtsrichtung. Für die Suche in die Rückrichtung kann man

den den Suffixbaum des umgekehrten Textes T^{rev} erstellen. [Sto95] kombiniert diese beiden Datenstrukturen zum *Affixbaum*. Er besteht konzeptionell aus den beiden atomaren Suffixbäumen von T und T^{rev}. Jedem Knoten u aus dem einen entspricht ein Knoten v aus dem anderen Baum in dem Sinne, dass die Beschriftung des Pfades von der Wurzel nach u genau die umgekehrte Beschriftung von der anderen Wurzel nach v ist. Jeder Knoten des Affixbaums ist also gleichzeitig ein Koten in beiden ursprünglichen atomaren Suffixbäumen und besitzt somit ausgehende Kanten vom Suffixbaum von T (zum *Suffix-Nachfolger*) sowie vom Suffixbaum von T^{rev} (zum *Präfix-Nachfolger*). Entfernt man nun alle Nichtblattknoten, die in keiner dieser beiden Kantenmengen verzweigend sind, so erhält man den fertigen Affixbaum.

Jeder Knoten u wird durch die Beschriftung $B(u)$ seines Pfades von der Wurzel her über die Suffix-Nachfolger eindeutig identifiziert, und $B(u)$ ist ein Teilwort vom ursprünglichen Text. Ein Schritt zu einem Suffix-Nachfolger verlängert dieses Teilwort nach rechts und entspricht einer Vorwärtssuche, ein Schritt zu einem Präfix-Nachfolger verlängert $B(u)$ auf der linken Seite und entspricht einer Rückwärtssuche. Der Affixbaum unterstützt somit bidirektionale Suche.

Beispiel 5.2

Es sei der Affixbaum zum String „`ababc`“ wie in Abbildung 5.2 gegeben. Die durchgezogenen Kanten sind die Kanten des Suffixbaums, die gestrichelten die des umgekehrten Suffixbaums. Es soll nach dem Muster „`abc`“ gesucht werden.

- Dazu wird beim Wurzelknoten Ⓦ gestartet, der für das leere Wort steht. Zwei Vorwärts-Suchschritte nach „`b`“ und „`c`“ führen uns zunächst nach ② und dann zum Knoten ③, der für das Wort „`bc`“ steht. Eine Suche in Rückwärtsrichtung nach dem Buchstaben „`a`“ resultiert im Knoten ⑥.

- Zum gleichen Ziel führen zum Beispiel auch zwei Rückwärtssuchen nach „`b`“ und „`a`“ und eine Vorwärtssuche nach „`c`“.

- Der Knoten ⑥ steht also offensichtlich für den String „`abc`“. Da er keine Suffix-Nachfolger besitzt, kommt dieser String nur ein mal im Text vor.

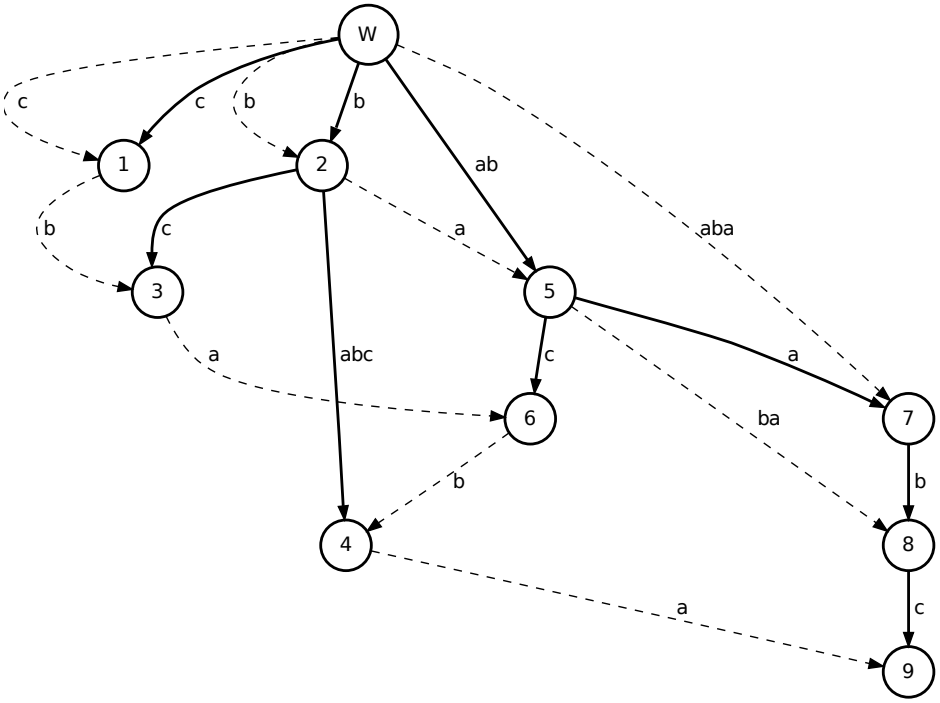


Abbildung 5.2: Affixbaum zu $T =$ „`ababc`“ (aus [Maa00])

Der Affixbaum kann in linearer Zeit erstellt werden. Der in [Maa00] vorgestellte Online-Algorithmus ist sogar in der Lage, den Text ohne Neuberechnung des kompletten Affixbaums in beide Richtungen zu verlängern. Beide Suffixbäume, und somit auch der Affixbaum, haben linearen Speicherplatzbedarf. In der Praxis benötigt er jedoch für jedes Zeichen des originalen Textes 45 Bytes Speicher.

5.3.3 Affix Array

In [Str07] wurde deshalb das *Affix Array* vorgestellt. Wie der Affixbaum basiert es auf der Tatsache, dass es Parallelen gibt zwischen dem Index eines Textes und dem des umgekehrten Textes. Es besteht daher aus einem Enhanced Suffix Array I vom Text T und einem Enhanced Suffix Array I^{rev} vom umgekehrten Text T^{rev}, dem *Reverse Prefix Array*.

Dass das Affix Array die Möglichkeit der Vorwärtssuche auf dem Text T in linearer Zeit bietet ist klar, denn dafür verwendet es den Index I. Die

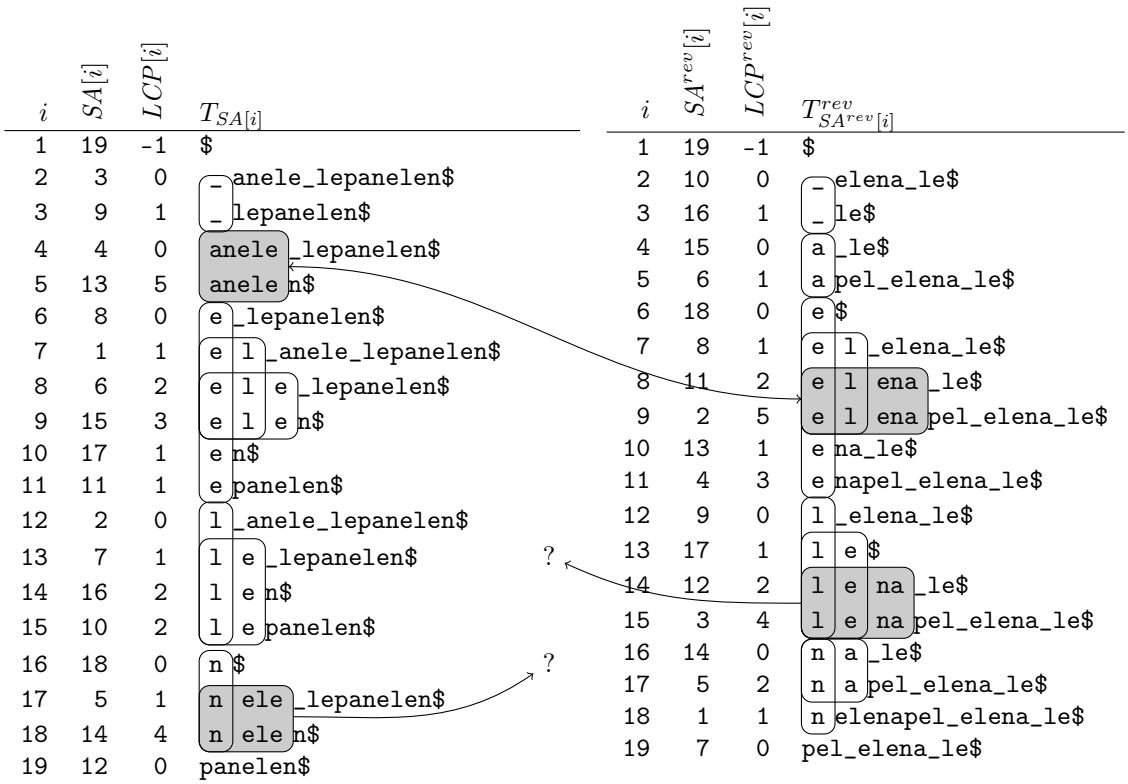


Abbildung 5.3: Affix Array von „`el_anele_lepanelen$`“ mit *LCP*-Intervallen

Rückwärtssuche eines Musters P im Text T entspricht der Vorwärtssuche vom umgekehrten Muster P^{rev} in T^{rev}. Durch den Index I^{rev} unterstützt das Affix Arrays somit auch die Rückwärtssuche in linearer Zeit. Der Zusammenhang zwischen den beiden Enhanced Suffix Arrays entsteht durch eine Abbildung zwischen den *LCP*-Intervallen dieser beiden Datenstrukturen. Falls diese Abbildung, und somit der Wechsel der Suchrichtung, in konstanter Zeit funktioniert, ist das Affix Array ein optimaler Index zur bidirektionalen Suche.

Abbildung 5.3 zeigt die beiden Enhanced Suffix Arrays I und I^{rev} mit ihren *LCP*-Intervallen. Man erkennt, dass einige *LCP*-Intervalle von I Entsprechungen in I^{rev} besitzen, wie z.B. das hervorgehobene *LCP*-Intervall 5-[4..5] in I und 5-[8..9] in I^{rev}, welche jeweils für die Wörter „`anele`“ bzw. „`elena`“ stehen. Andere *LCP*-Intervalle haben dagegen keine solchen direkten Entsprechungen. Es existiert also offensichtlich keine bijektive Abbildung zwischen den *LCP*-Intervallen der beiden Enhanced Suffix Arrays. Es lässt sich

jedoch zeigen, dass es zu jedem LCP-Intervall ein entsprechendes Intervall im anderen Index gibt, das gleich groß ist, möglicherweise jedoch einen größeren l-Wert besitzt. Für den effizienten Wechsel in den anderen Index wird jedes Enhanced Suffix Array um eine weitere Tabelle erweitert, welche zu jedem LCP-Intervall das entsprechende LCP-Intervall in der anderen Datenstruktur speichert. Damit ist der Wechsel der Suchrichtung in konstanter Zeit möglich.

Die beiden Enhanced Suffix Arrays bieten jeweils die Funktionalität eines Suffixbaumes, und somit unterstützt das Affix Array alle Algorithmen, die auch mit dem Affixbaum möglich sind. Der Vorteil ist, dass es mit 18 bis 20 Bytes pro Zeichen des ursprünglichen Textes wesentlich speicherplatzeffizienter ist.

Kapitel 6

Der bidirektionale Wavelet-Index

Motiviert vom Affix Array soll eine weitere, wesentlich einfachere Datenstruktur entwickelt werden, welche bidirektionale Suche ermöglicht. Beim Affix Array wird dies dadurch erreicht, dass ein Enhanced Suffix Array vom Text T für die Vorwärtssuche und ein Enhanced Suffix Array vom umgekehrten Text T^{rev} für die Rückwärtssuche erstellt wird. Die selbe Idee, also die Vorverarbeitung des Textes und des umgekehrten Textes führt uns zum *bidirektionalen Wavelet-Index.*

6.1 Grundlagen

Die bisher interessanteste vorgestellte Datenstruktur war der Wavelet Tree, denn er bietet einen sehr guten Kompromiss zwischen Größe und Laufzeit der Suche.

Definition 11. *Gegeben sei ein Text T. Dann besteht der bidirektionale Wavelet-Index von T aus*

- *dem Wavelet Tree R der Burrows-Wheeler-Transformation vom Text T (Rückwärtsindex) und*
- *dem Wavelet Tree V der Burrows-Wheeler-Transformation vom umgekehrten Text T^{rev} (Vorwärtsindex)* [1].

Außerdem wird das C-Array vom Text erstellt, und jeder k-te Wert von SA gespeichert, wobei $k \in O(\log n)$.

[1] Auch der umgekehrte Text T^{rev} besitzt das Zeichen `$` an der letzten Position.

Wie beim Affix Array gibt es zwischen diesen beiden Indexen folgende Parallele.

Satz 1. *Jedem ω-Intervall im Rückwärtsindex entspricht ein ω^{rev}-Intervall im Vorwärtsindex und umgekehrt, und beide sind gleich groß.*

Beweis. Es sei T ein Text der Länge n, ω ein Teilwort von T, SA das Suffix Array von T, R der Rückwärtsindex und V der Vorwärtsindex. Sei $[i..j]$ das ω-Intervall in R. Es gibt also genau $(j - i + 1)$ Suffixe $T_{SA[i]}, \ldots, T_{SA[j]}$ die mit ω beginnen. Folglich existieren $(j - i + 1)$ Positionen im umgekehrten Text T^{rev}, die mit ω^{rev} beginnen, und diese sind $p_k = n - (SA[k] + |\omega| - 1)$ für $i \leq k \leq j$. Da diese Positionen p_k die einzigen im umgekehrten Text T^{rev} sind, welche mit ω^{rev} beginnen, und alle Suffixe von T^{rev} in V lexikographisch sortiert sind, existiert ein Intervall $[i^{rev}..j^{rev}]$ in V, in dem alle Suffixe $T^{rev}_{SA^{rev}[k]}$ für $i^{rev} \leq k \leq j^{rev}$ mit ω^{rev} beginnen, und es gilt $j - i + 1 = j^{rev} - i^{rev} + 1$.

Diese Aussagen gelten analog für die Gegenrichtung. □

Da es keine direkte Abbildung von ω-Intervallen des einen Index auf ω^{rev}-Intervalle des anderen Index gibt, müssen während der Suche beide Intervalle gespeichert werden. Jeder Suchschritt in einem erfordert somit auch eine Anpassung des Intervalls im anderen Index.

Der bidirektionale Wavelet-Index ist eine symmetrische Datenstruktur. Die Rückwärtssuche erfolgt auf dem ersten Index, und erfordert eine Anpassung des Intervalls auf dem zweiten Index. Genau analog dazu erfolgt die Vorwärtssuche auf dem zweiten Index und erfordert die selbe Anpassung des Intervalls auf dem ersten Index. Deshalb wird exemplarisch nur die Rückwärtssuche erläutert.

Wie in Abschnitt 4.1 erfolgt die Rückwärtssuche auf dem Rückwärtsindex R mit der Funktion Occ. Dabei wird vom ω-Intervall $[i..j]$ ausgehend das $c\omega$-Intervall bestimmt. Im Vorwärtsindex V entspricht dieser Schritt einer Vorwärtssuche. Die Anpassung des zweiten Intervalls $[i^{rev}..j^{rev}]$ erfordert also die Information, welche Buchstaben die Suffixe des umgekehrten Textes in diesem Intervall an der Position $|\omega| + 1$ haben. Der Vorwärtsindex hat zwar keine direkte Information darüber, welche Buchstaben dort stehen, man weiß nur, dass diese lexikographisch sortiert sind. Die gesuchte Information steht aber in der Burrows-Wheeler-Transformierten BWT im Rückwärtsindex. Dort sind diese Buchstaben zwar nicht sortiert, falls man jedoch die Antwort auf die Frage weiß, wie viele Buchstaben in $BWT[i..j]$ kleiner sind

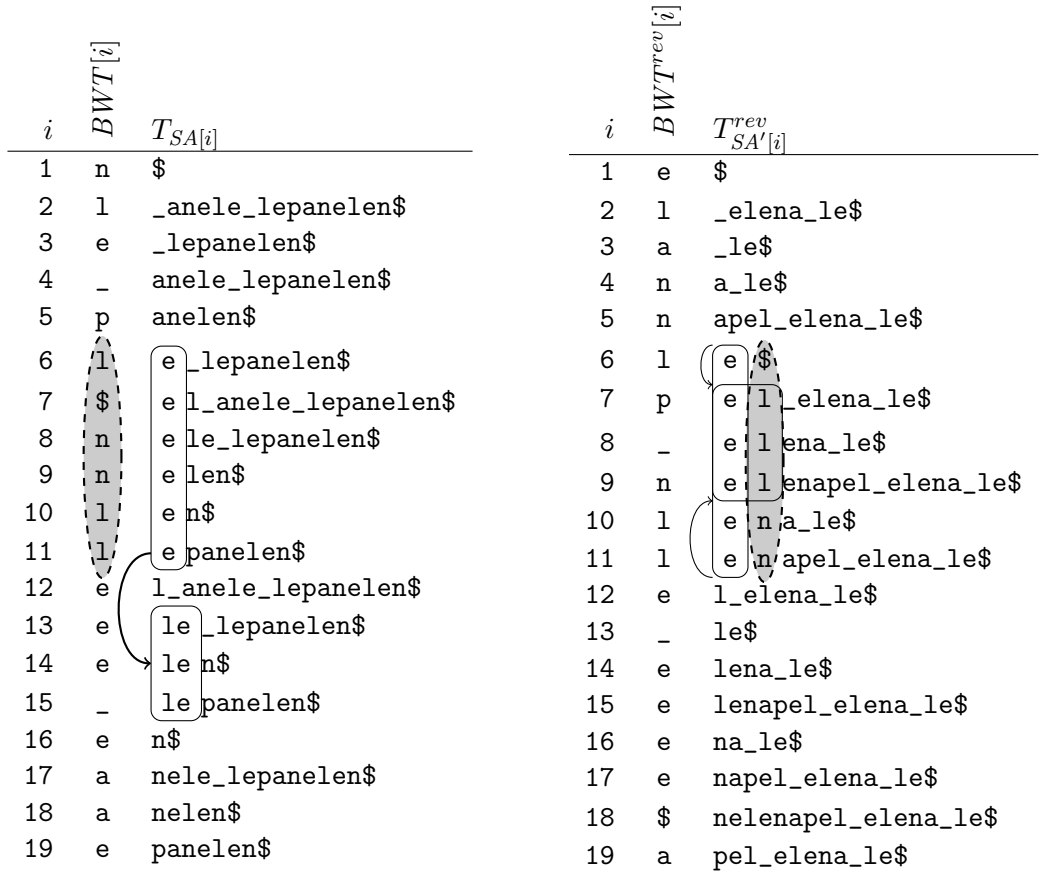


Abbildung 6.1: Ein Schritt der Rückwärtssuche auf dem bidirektionalen Wavelet-Index zu $T =$ „el_anele_lepanelen$“ von „e“ nach „le“.

als der gesuchte Buchstabe c, so weiß man auch, dass die linke Intervallgrenze i^{rev} um genau diese Anzahl nach rechts verschoben wird. Genauso verschiebt sich die rechte Intervallgrenze j^{rev} um die Anzahl von Buchstaben größer als c nach links.

Abbildung 6.1 veranschaulicht einen Suchschritt im bidirektionalen Wavelet-Index. Gegeben sind das „e“-Intervall [6..11] im Rückwärtsindex (links) und das „e“-Intervall [6..11] im Vorwärtsindex (rechts). Es soll ein Rückwärtssuchschritt nach dem Buchstaben „l“ durchgeführt werden. Dabei wird zunächst im Rückwärtsindex mit der *Occ*-Funktion das neue „le“-Intervall [13..15] bestimmt. Um das Intervall im Vorwärtsindex anzupassen, wird die Funktion $getBounds([6..11], [1..7], \text{„l“})$ auf dem Wavelet Tree des ersten Index aufgerufen. Diese bestimmt die beiden Anzahlen der Buchstaben in

$BWT[6..11]$ (in der Abbildung grau hervorgehoben), welche kleiner bzw. größer sind als „1“. Es handelt sich offensichtlich um die selben Buchstaben, die auch im zweiten Index hervorgehoben sind. Da ein Buchstabe kleiner ist und zwei größer wird nun die linke Intervallgrenze im Vorwärtsindex um 1 nach rechts verschoben, und die rechte Intervallgrenze um 2 nach links.

Vergleicht man diese neue Datenstruktur mit dem Affix Array, so erkennt man vor allem, dass der Speicherplatzbedarf wesentlich geringer ist. Ein Wavelet Tree benötigt $n \cdot \log|\Sigma| + o(n \cdot \log|\Sigma|)$ Bits Speicherplatz. Geht man davon aus, dass der zusätzliche Speicherplatz für die Rank-Anfragen $0{,}25 \cdot n$ Bits benötigen, dann ist der gesamte Index nur etwa 2,5 mal so groß wie der ursprüngliche Text. Zudem benötigt man zur Bestimmung des Ergebnisintervalls keinen Zugriff auf den Text.

Das Ergebnis der Suche nach dem Muster ω ist am Ende das ω-Intervall. Um herauszufinden, wo im Text das Muster vorkommt, benötigt man allerdings die Werte des Suffix Arrays in diesem Intervall. Wendet man die Technik mit der inversen Ψ-Funktion aus Abschnitt 5.2.2 an, so genügt es, nur einen Bruchteil der SA-Werte abzuspeichern.

6.2 Die Funktion *getBounds*

Es ist die Antwort gesucht auf die Frage „Wie viele Buchstaben in $BWT[i..j]$ sind kleiner bzw. größer als c?“ Auch diese Berechnung erfolgt über einen Top-Down-Durchlauf des Wavelet Trees, der mit dem gesamten Alphabet $[1..|\Sigma|]$ beginnt. Gehört der Buchstabe c zur linken Hälfte des betrachteten Alphabetintervalls, so sind auf jeden Fall alle Buchstaben, die zur rechten Hälfte gehören, größer als c. Die Anzahl dieser Buchstaben ist gleich der Anzahl der Einsen im Intervall $[i..j]$. Die weitere Berechnung erfolgt rekursiv im linken Kindknoten. Der String im linken Kind eines Knotens mit dem Alphabetintervall $[l..r]$ besteht nur noch aus denjenigen Buchstaben, die in $B^{[l..r]}$ durch eine 0 repräsentiert werden. Deshalb entspricht das Intervall $[i..j]$ im linken Kindknoten dem Intervall $[a..b]$, wobei a die Anzahl der Nullen im Intervall $[1..i-1]$ und b die Anzahl der Nullen im Intervall $[1..j]$ im Bit-Vektor $B^{[l..r]}$ ist. Analog gilt, dass alle Buchstaben der linken Hälfte des Alphabets kleiner sind als c, falls c zur rechten Hälfte gehört, und hier erfolgt die weitere rekursive Berechnung im rechten Kindknoten. Ist das betrachtete Alphabetintervall nur noch einen Buchstaben groß, so handelt es sich um den

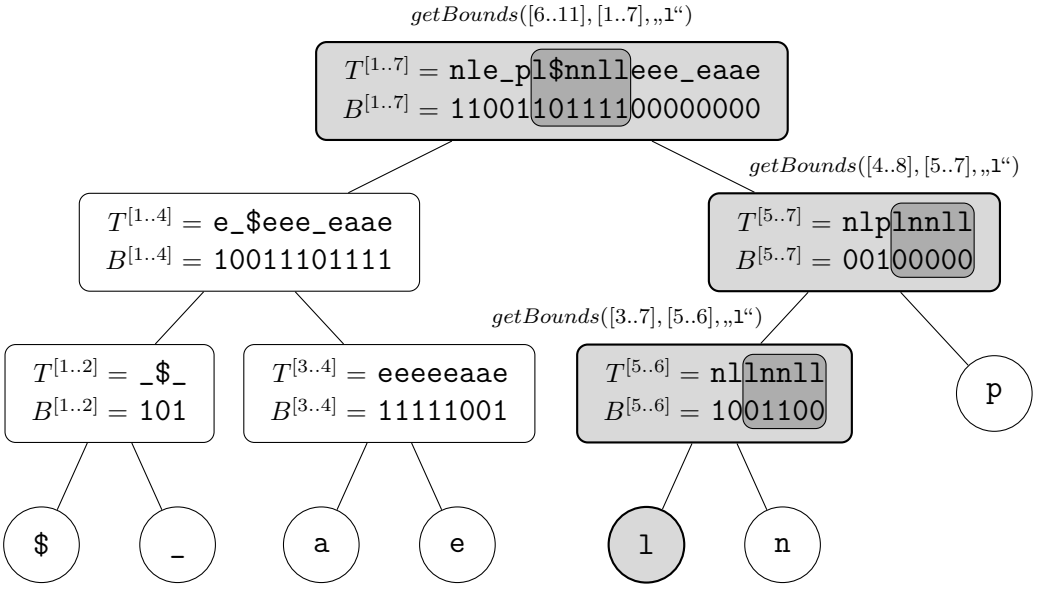


Abbildung 6.2: Die besuchten Knoten des Wavelet Trees für die Berechnung der Funktion $getBounds([6..11], [1..7], \text{„l“})$ sind leicht hervorgehoben. In $B^{[1..7]}[6..11]$ steht eine Null, also ist ein Buchstabe kleiner als „l“. In $B^{[5..6]}[3..7]$ stehen zwei Einsen, somit sind zwei Buchstaben größer als „l“.

Buchstaben c. In diesem Basisfall ist kein Buchstabe kleiner bzw. größer, der Rückgabewert ist also jeweils 0.

Algorithmus 6.1 berechnet somit genau die Anzahl der Buchstaben in $BWT[i..j]$, die kleiner bzw. größer als c sind. Er wird gestartet mit Alphabetintervall $[1..|\Sigma|]$. Da im Wavelet Tree nur die Knoten entlang des Pfades von der Wurzel bis zum Knoten $[c..c]$ besucht werden, ist die Komplexität $O(\log|\Sigma|)$. Es müssen in jeder Inkarnation der Funktion *getBounds* effektiv nur zwei rank-Anfragen berechnet werden, denn da Bit-Vektoren nur die Werte 0 und 1 beinhalten, ist der Wert von $\text{rank}_0(B, i)$ gleich $i - \text{rank}_1(B, i)$.

Satz 2. *Ein Suchschritt auf dem bidirektionalen Wavelet-Index in eine beliebige Richtung hat die Laufzeit* $O(\log|\Sigma|)$.

Beweis. Ein Schritt der Rückwärtssuche erfordert zwei Anwendungen der Funktion *Occ*. Da diese Funktion mit dem Wavelet Tree implementiert ist, dauert eine Anwendung die Zeit $O(\log|\Sigma|)$, genauso wie die Anpassung des zweiten Intervalls mittels der Funktion *getBounds*. Ein Schritt in Vorwärtsrichtung funktioniert analog durch einen Rückwärtsschritt auf dem zweiten Index und die Anpassung des ersten Intervalls. □

Algorithmus 6.1 *getBounds*$([i..j], [l..r], c)$ auf einem Wavelet Tree

```
/* [i..j] ist das Intervall, für das die Anfrage gestartet wird */
/* [l..r] ist das Alphabetintervall */
/* Rückgabe (smaller, greater), Zahl der Buchstaben kleiner bzw. größer als c */
if l = r then
  return (0, 0)
else
  (a_0, b_0) ← (rank_0(B^[l..r], i − 1), rank_0(B^[l..r], j))
  (a_1, b_1) ← (rank_1(B^[l..r], i − 1), rank_1(B^[l..r], j))
  m = ⌊(l+r)/2⌋
  if c ≤ m then
    (smaller, greater) ← getBounds([a_0 + 1..b_0], [l..m], c)
    return (smaller, greater + b_1 − a_1)
  else
    (smaller, greater) ← getBounds([a_1 + 1..b_1], [m + 1..r], c)
    return (smaller + b_0 − a_0, greater)
```

6.3 Beispiel

Es soll im Text T = „`el_anele_lepanelen$`“ mit dem bidirektionalen Wavelet-Index (Abbildung 6.1) nach dem Muster „`ele`“ gesucht werden. Zur Demonstration der bidirektionalen Suche wird diesmal zunächst das „`l`“-Intervall bestimmt. Danach erfolgt ein Schritt der Rückwärtssuche nach „`e`“ und ein Schritt der Vorwärtssuche nach „`e`“. Es bezeichne $[i_k^R..j_k^R]$ das Intervall im Rückwärtsindex im k-ten Suchschritt, analog $[i_k^V..j_k^V]$ im Vorwärtsindex. Des Weiteren beziehen sich Occ^R, Occ^V, $getBounds^R$ und $getBounds^V$ auf den entsprechenden Index.

1. Da auch der bidirektionale Wavelet-Index das C-Array speichert ist der erste Schritt trivial. Das „`l`“-Intervall ist im Rückwärts- und im Vorwärtsindex gleich

$$i_1^R = i_1^V = C[l] = 12 \quad \text{und}$$
$$j_1^R = j_1^V = C[l+1] - 1 = 15$$

2. Die Rückwärtssuche erfolgt zunächst auf dem Rückwärtsindex mit der Funktion Occ^R. Es gilt

$$i_2^R = C[e] + Occ^R(e, 11) = 6 + 1 = 7 \quad \text{und}$$
$$j_2^R = C[e] + Occ^R(e, 15) - 1 = 6 + 4 - 1 = 9$$

Es folgt die Anpassung des Intervalls im Vorwärtsindex

$$(smaller, greater) \leftarrow getBounds^R([i_1^R..j_1^R], [1..7], \text{„e“})$$
$$i_2^V = i_1^V + smaller = 12 + 1 = 13$$
$$j_2^V = j_1^V - greater = 15 - 0 = 15$$

3. Die Vorwärtssuche nach „e“ erfolgt als Rückwärtsschritt auf dem Vorwärtsindex. Es gilt

$$i_3^V = C[e] + Occ^V(e, 12) = 6 + 2 = 8 \quad \text{und}$$
$$j_3^V = C[e] + Occ^V(e, 15) - 1 = 6 + 4 - 1 = 9$$

Es folgt die Anpassung des Intervalls im Rückwärtsindex

$$(smaller, greater) \leftarrow getBounds^V([i_2^V..j_2^V], [1..7], \text{„e“})$$
$$i_3^R = i_2^R + smaller = 7 + 1 = 8$$
$$j_3^R = j_2^R - greater = 9 - 0 = 9$$

Das gesuchte „ele“-Intervall bezieht sich immer auf den Rückwärtsindex und ist somit [8..9].

6.4 Optimierung für große Alphabete

Häufig wird bei der Angabe der Laufzeitkomplexität von Indexen nur die Abhängigkeit von der Länge des Textes n betrachtet und die Größe des Alphabets $|\Sigma|$ als „konstanter Faktor“ vernachlässigt. Es sind aber durchaus Problemstellungen denkbar, in denen das zugrundeliegende Alphabet in der Größenordnung von einigen Tausend Buchstaben liegt, und dann lohnt sich eine Betrachtung der Abhängigkeit der Laufzeit von $|\Sigma|$. Man stelle sich einen Backtracking-Algorithmus vor, der durch Rückwärtssuche das Muster ω um alle möglichen Buchstaben nach links verlängern will. Dafür kann er für jedes $c \in \Sigma$ eine Rückwärtssuche starten, welche in der Zeit $O(\log |\Sigma|)$ entweder das nichtleere $c\omega$-Intervall oder das leere Intervall zurückliefert. Die gesamte Schleife läuft also in der Zeit $O(|\Sigma| \log |\Sigma|)$.

Wenn man die Antwort auf die Frage „Welche Buchstaben kommen ausgehend vom Intervall $[i..j]$ für die Rückwärtssuche nach dem Buchstaben c in Frage?“ effizient beantworten kann, muss man die Rückwärtssuche auch nur

für diese Buchstaben starten. In Frage kommen natürlich genau die Buchstaben, die in der Burrows-Wheeler-Transformation im Intervall $[i..j]$ stehen. Für den Wavelet Tree kann man hierfür den folgenden Algorithmus entwickeln.

Der Wavelet Tree wird in einer Top-Down-Reihenfolge durchlaufen. Falls in $B^{[l..r]}$ im Intervall $[i..j]$ Nullen stehen dann bedeutet das, dass an diesen Stellen in $T^{[l..r]}$ ein Buchstabe aus der ersten Hälfte des Alphabetintervalls steht. Dann muss in den linken Teilbaum abgestiegen werden. Analog muss anschließend auch der rechte Teilbaum untersucht werden, falls im Intervall $[i..j]$ Einsen stehen, denn diese repräsentieren Buchstaben aus der zweiten Hälfte des Alphabetintervalls. Ist der Tiefendurchlauf bei einem Blattknoten $[c..c]$ angekommen, so gehört c offensichtlich zu den gesuchten Buchstaben. Wie bei dem *getBounds*-Algorithmus 6.1 muss das zu untersuchende Intervall beim rekursiven Abstieg entsprechend verkleinert werden.

Algorithmus 6.2 $getCharacters([i..j], [l..r], list)$ auf einem Wavelet Tree

```
/* [i..j] ist das Intervall, für das die Anfrage gestartet wird */
/* [l..r] ist das Alphabetintervall */
/* list ist das Ergebnis, muss zu Beginn die leere Liste sein */
  if l = r then
    append(list, Σ[l])
  else
    (a_0, b_0) ← (rank_0(B^[l..r], i − 1), rank_0(B^[l..r], j))
    (a_1, b_1) ← (rank_1(B^[l..r], i − 1), rank_1(B^[l..r], j))
    m = ⌊(l+r)/2⌋
    if b_0 > a_0 then
      getCharacters([a_0 + 1..b_0], [l..m], list)
    if b_1 > a_1 then
      getCharacters([a_1 + 1..b_1], [m + 1..r], list)
```

Algorithmus 6.2 berechnet eine Liste der Buchstaben in $BWT[i..j]$. Die Komplexität ist dabei die Anzahl der besuchten Blätter (also die Länge des Ergebnisses) mal die Höhe des Baumes. Sei k die Anzahl der Buchstaben in BWT im Intervall $[i..j]$. Dann ist die Laufzeit des Algorithmus $O(k \log |\Sigma|)$.

Kapitel 7

Praxisanwendung: Sekundärstruktur der RNA

Die DNA (Desoxyribonukleinsäure) ist der Träger der Erbinformationen in allen Lebewesen. Es handelt sich dabei um ein Makromolekül, um eine Aneinanderkettung von sehr vielen einzelnen Bausteinen, den Nukleotiden. Jedes dieser Nukleotide besteht seinerseits aus einem Zuckermolekül, Phosphorsäure und einer von vier unterschiedlichen Basen, und diese heißen Adenin (A), Guanin (G), Cytosin (C) und Thymin (T). Abstrakt betrachtet handelt es sich bei DNA also um einen String über dem Alphabet $\Sigma = \{A, C, G, T\}$. Die Basenpaare A–T und C–G sind dabei zueinander komplementär und können untereinander Wasserstoffbrücken, sogenannte Basenpaarbindungen eingehen. Jedes DNA-Molekül besteht aus zwei identischen Einzelsträngen, die gegenläufig parallel angeordnet sind und somit einen Doppelstrang bilden. So kommt es, dass jedes Nukleotid des einen Stranges eine Basenpaarbindung mit dem komplementären Nukleotid des anderen Stranges eingeht.

Auf den DNA-Strängen gibt es Abschnitte, auf denen die Erbinformation codiert ist, und diese Abschnitte nennt man Gene. Die Transkription ist ein biologischer Prozess, bei dem RNA (Ribonukleinsäure) anhand eines Gens als Vorlage hergestellt wird. Dabei wird zunächst der Doppelstrang kurzzeitig aufgetrennt. Parallel zu einem dieser DNA-Stränge lagern sich Bausteine an, die sich zu einem RNA-Molekül verbinden. Dieses hat eine ähnliche Struktur wie die DNA. Ein Unterschied ist, dass statt Thymin die funktionell äquivalente Base Uracil (U) verwendet wird. Da sich an ein Nukleotid der DNA nur ein komplementäres Nukleotid der RNA anlagert, ist das Produkt der Transkription eine komplementäre Kopie dieses Abschnitts, bzw. eine exak-

te Kopie des entsprechenden Abschnitts des komplementären DNA-Stranges (bis eben auf die Tatsache dass Thymin durch Uracil ersetzt wurde).

Das entstehende Transkript lässt sich anhand seiner Funktion in weitere Gruppen einteilen. So wird zum Beispiel die Messenger-RNA (mRNA) direkt als Bauplan für die Proteinsynthese verwendet, ihre Nukleotidsequenz gibt dabei exakt vor wie das Protein auszusehen hat. Andere Arten der RNA werden dagegen zunächst in eine Sekundärstruktur umgewandelt, und haben dann andere Funktionen in der Zelle. Ein Beispiel, wo die Sekundärstruktur eine große Rolle spielt, ist bei der „Reifung“ der microRNA, welche eine wichtige Rolle bei weiteren genetischen Prozessen spielt. Hierbei wird das etwa 70 Nukleotide lange Transkript (die pre-microRNA) zunächst in eine Sekundärstruktur gefaltet, aus welcher sich anschließend die microRNA bildet, die nur noch 21–24 Nukleotide lang ist. Ihre Nukleotidabfolge hängt weniger von der genauen Sequenz der pre-microRNA ab, sondern hauptsächlich von dieser Sekundärstruktur. Möchte man nun für eine gegebene microRNA das kodierende Gen bestimmen, so muss man also die gesamte DNA nach einer Region durchsuchen, deren Transkript eine passende Sekundärstruktur ausbilden könnte.

Die Sekundärstruktur entsteht durch die Bindungskräfte der komplementären Nukletidpaare A–U, C–G und G–U. Zu den wichtigsten Elementen gehören sogenannte Hairpin-Loop-Strukturen. Sie bestehen aus einem Mittelteil, also einigen ungepaarten Nukleotiden, und aus dem Stem, welcher entsteht, wenn sich die Nukleotide links und rechts vom Mittelteil aneinander binden und so eine Schleife entstehen lassen (siehe Abbildung 7.1). Bei einer Suche nach einem Loop bestimmt man zuerst alle Vorkommen des Mittelteils ω. Diese Menge von Kandidaten wird anschließend genauer untersucht. Dazu verlängert man das Muster zunächst auf einer Seite um einen Buchstaben d und sucht nach dem neuen Muster ωd. Falls es Vorkommen gibt, wird das Muster jetzt auf der anderen Seite um einen komplementären Buchstaben c verlängert, welcher mit d eine Basenpaarbindung eingehen kann. Für diese Strategie ist also weder der ausschließliche Einsatz der Vorwärtssuche noch der Rückwärtssuche sinnvoll, sondern man benötigt einen Index, der bidirektionale Suche unterstützt.

In Algorithmus 7.1 ist das Vorgehen beschrieben, wie Vorwärts- und Rückwärtssuche kombiniert zur bidirektionalen Suche verwendet werden können (vgl. [Str07]). Die Funktion *bidirectionalSearch* realisiert rekursiv die

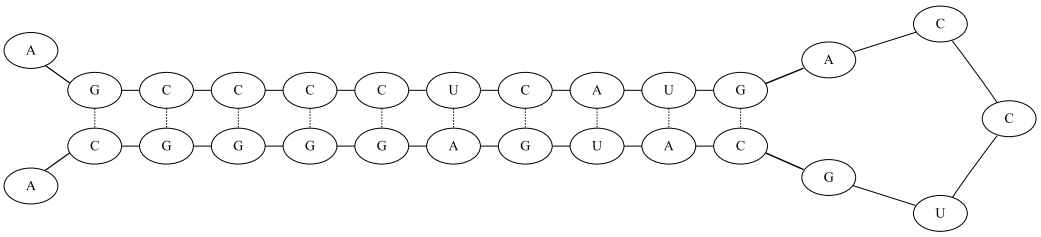


Abbildung 7.1: Beispiel für die Sekundärstruktur des Strings „AGCCCCCUCAUGACCUGCAUGAGGGGCA“. Es entsteht ein Hairpin Loop mit dem Mittelteil „ACCUG“ und einem Stem der Länge 10. Die gebundenen Basenpaare sind gestrichelt angedeutet.

Algorithmus 7.1 Pseudocode der bidirektionalen Suche im RNA-String

Sei $[i..j]$ das „ACCUG“-Intervall. Die Suche wird gestartet mit:

bidirectionalSearch($[i..j]$, „ACCUG“)

function **forwardSearch** ($[i..j]$, c, Musterlänge)
/ Gibt das ωc-Intervall zurück, oder $\perp$ falls dieses nicht existiert */*

function **backwardSearch** ($[i..j]$, c)
/ Gibt das $c\omega$-Intervall zurück, oder $\perp$ falls dieses nicht existiert */*

function **bidirectionalSearch** ($[i..j]$, ω)

```
1: /* Basisfall. Ziel erreicht, Ende */
2: if |ω| = MaxLength then
3:    print ω, [i..j]
4:    for each k ∈ [i..j] do
5:       print SA[k]
6:    return
7: end if
8: for each d ∈ Σ do
9:    [i'..j'] ← forwardSearch ([i..j], d, |ω|)
10:   if [i'..j'] ≠ ⊥ then
11:      for each c ∈ Complement(d) do
12:         [i''..j''] ← backwardSearch ([i'..j'], c)
13:         if [i''..j''] ≠ ⊥ then bidirectionalSearch([i''..j''], cωd)
14:      end for
15:   end if
16: end for
```

Backtracking-Suche[1]. In den Zeilen 2 bis 7 wird die Suche abgebrochen, da die geforderte Musterlänge erreicht ist, und es werden alle Ergebnisse ausgegeben. Ansonsten wird in Zeile 9 für jeden möglichen Buchstaben $d \in \Sigma$ die Funktion *forwardSearch* aufgerufen. Falls diese ein Intervall zurückliefert, dann existieren Vorkommen des Musters ωd. In diesem Fall wird in einer weiteren Schleife für jeden möglichen Buchstaben $c \in Complement(d)$, also für jeden Buchstaben der komplementär zu d ist, die Funktion *forwardSearch* gestartet. Liefert auch diese ein Intervall zurück, so ist $[i''..j'']$ das Intervall aller Vorkommen des Musters $c\omega d$, und die Funktion *bidirectionalSearch* ruft sich von neuem rekursiv auf.

[1]Der „innere Zustand“ der Backtracking-Suche ist hier nur die Musterlänge und das Intervall $[i..j]$. Beim *bidirektionalen Wavelet-Index* kommt noch das zweite Intervall aus dem Vorwärtsindex hinzu!

Kapitel 8

Implementierung

Zum Vergleich der verschiedenen vorgestellten Datenstrukturen und Algorithmen wurden insgesamt sechs unterschiedliche Indexe implementiert, die allesamt bidirektionale Suche unterstützen. Es wurden einige komplexe Testfälle ausgewählt, die im Wesentlichen durch eine Tiefensuche wie in Algorithmus 7.1 bearbeitet werden. Alle Tests wurden auf einem PC durchgeführt mit vier *Dual-Core AMD Opteron(tm) 8218* Prozessoren mit jeweils 2,6 GHz und mit 64 GB Arbeitsspeicher.

8.1 Vorstellung der Implementierung

Die gesamte Implementierung erfolgte in der Programmiersprache C++. Diese eignet sich für diesen Zweck besonders gut, denn sie erlaubt sowohl maschinennahe Programmierung als auch abstrakte Programmiertechniken wie Objektorientierung und Templates. Damit ist sowohl eine hohe Modularität der Implementierung und die Austauschbarkeit vieler Datenstrukturen als auch ein sehr schnelle Ausführung möglich.

Grundlage der implementierten Datenstrukturen ist die *succinct data structure library* ([Gog09]), welche von Simon Gog am Institut für theoretische Informatik der Universität Ulm entwickelt wird. Diese beinhaltet unter anderem Algorithmen zur Herstellung von Suffix Arrays, Vektoren für Ganzzahlen mit beliebiger aber fester Breite bis zu 64 Bit und Algorithmen zur Beantwortung von Rank- und Select-Anfragen für Bit-Vektoren in konstanter Zeit.

8.2 Die verschiedenen Indexe

Es wurden insgesamt sechs verschiedene Indexe implementiert, die in dieser Arbeit vorgestellt wurden.

① Der erste Index (SA, Text, Ψ) verwendet die binäre Suche auf dem Suffix Array und den Text zur Vorwärtssuche. Die Rückwärtssuche erfolgt mit der Ψ-Funktion.

② Ein wenig platzsparender ist der zweite Index (SA, SA^{-1}), welcher vorwärts mit dem Suffix Array und dem inversen Suffix Array sucht. Die Rückwärtssuche erfolgt mit der Ψ-Funktion, die berechnet wird durch $\Psi(i) = SA^{-1}[SA+1]$.

③ Der dritte Index (SA, Text, Occ) sucht vorwärts mit dem Suffix Array und dem Text, benutzt jedoch den Wavelet Tree, um mit der Funktion Occ rückwärts zu suchen.

④ Das komprimierte Suffix Array aus Abschnitt 5.2.1 (CSA/Ψ). Die Ψ-Funktion ist dabei zusätzlich komprimiert. Da für die Vorwärtssuche sehr häufig auf die Werte SA zugegriffen wird, ist ein Zwölftel der Werte fest abgespeichert. Experimente haben gezeigt, dass die Vorwärtssuche, die statt dem inversen Suffix Array den Text verwendet, wesentlich schneller ist, deshalb wird statt den Samples von SA^{-1} der Text gespeichert.

⑤ Den Wavelet Tree zur Rückwärtssuche und zur Berechnung der SA- und SA^{-1}-Werte für die Vorwärtssuche (CSA/Occ) aus Abschnitt 5.2.2. Auch hier ist ein Zwölftel der Werte von SA gespeichert, und statt den Samples von SA^{-1} der Text.

⑥ Der bidirektionale Wavelet-Index (BWI). Da die SA-Werte nicht für die eigentliche Suche, sondern nur zur Ausgabe des Ergebnisses benötigt werden, ist nur ein Hundertstel der SA-Werte gespeichert.

Leider standen für diesen Vergleich weder eine Implementierung des Affixbaums noch des Affix Arrays zur Verfügung.

8.3 Testumgebung

8.3.1 Texte

Die Texte, mit welchen die Laufzeitvergleiche durchgeführt wurden, stammen von der *UCSD Genome Browser Download Page*[1]. Für diese Arbeit wurden die Texte so angepasst, dass sie nur noch die vier Buchstaben `A`, `C`, `G`, `T` und `$` enthalten. Es werden insgesamt drei Texte verwendet:

1. „Saccharomyces“. Das Genom der Hefe (Saccharomyces cerevisiae) mit 12,2 Millionen Nukleotiden.[2]

2. „Drosophila“. Das Genom der schwarzbäuchigen Fruchtfliege (Drosophila melanogaster) mit 162 Millionen Nukleotiden.[3]

3. „Human“. Dieser Text besteht aus den Chromosomen 1 bis 5 des Menschen und umfasst eine Milliarde Nukleotide[4]. Dies entspricht etwa einem Drittel des gesamten menschlichen Genoms.

8.3.2 Muster

Zum Vergleich der Laufzeiten auf den verschiedenen Indexen wurden einige Testfälle aus [Str07] übernommen. Der Text und die Muster bestehen nur aus den Buchstaben `A`, `C`, `G` und `T`. Es gibt drei komplementäre Nukleotidpaare, und zwar `A-T`, `C-G` und `G-T`.

Die folgenden Muster sind in der intuitiven *Hybrid Pattern Language* (HyPaL, [GSKS01]) dargestellt. Der Buchstabe `N` im Muster ist ein Platzhaltersymbol, an dessen Stelle im Muster können also alle vier Buchstaben stehen. `(A|C)` bedeutet entweder `A` oder `C`. Eine Zahl `{k}` hinter einem Symbol gibt an, dass dieses Symbol genau k mal vorkommen soll. Zwei Zahlen `{k,l}` hinter einem Symbol bedeuten, dass dieses Symbol mindestens k und höchstens l mal vorkommen soll. Die Formulierung `^stem` steht für den umgekehrten komplementären String `stem`.

[1] `http://hgdownload.cse.ucsc.edu/downloads.html`

[2] Quelle: Saccharomyces Genome Database (S288C)
`http://hgdownload.cse.ucsc.edu/goldenPath/sacCer2/bigZips/chromFa.tar.gz`

[3] Quelle: Berkeley Drosophila Genome Project (Release 5)
`http://hgdownload.cse.ucsc.edu/goldenPath/dm3/bigZips/chromFa.tar.gz`

[4] Quelle: Genome Reference Consortium (GRCh37)
`http://hgdownload.cse.ucsc.edu/goldenPath/hg19/bigZips/chromFa.tar.gz`

- `hairpin1 = (stem:=N{20,50}) (loop:=NNN) ^stem`
- `hairpin2 = (stem:=N{10,50}) (loop:=GGAC) ^stem`
- `hairpin4 = (stem:=N{10,15}) (loop:=GGAC[1]) ^stem`

  `[1]` bedeutet, dass an einer beliebigen Stelle ein Buchstabe eingefügt werden darf. Der Mittelteil `loop` ist also `GGAC`, `NGGAC`, `GNGAC`, `GGNAC`, `GGANC` oder `GGACN`.
- `hloop(length) = (stem:=N{15,20}) (loop:=N{length}) ^stem`

  `length` gibt die Länge des Hairpin Loops an. Da der Mittelteil dieses Musters sehr unspezifisch ist wird nur `hloop(5)` getestet.
- `acloop(length) = (stem:=N{15,20}) (loop:=(A|C){length}) ^stem`

  Dieses Muster wird in drei Varianten mit den Parametern 5, 10 und 15 getestet.

Die Suche nach einem Muster erfolgt durch eine iterative Tiefensuche, die mit einem Stack implementiert ist. Zunächst werden alle möglichen Mittelteile gesucht und auf dem Stack gespeichert. Anschließend wird die bidirektionale Suche nach dem Stem durch eine iterative Variante von Algorithmus 7.1 gestartet.

8.4 Ergebnisse

In den folgenden drei Tabellen 8.1, 8.2 und 8.3 sind alle Messergebnisse zusammengefasst. Sie ergeben sich aus dem Durchschnitt aus vier unabhängigen Durchgängen jedes Testfalls.

Größe

Die Zahlen in den Klammern beziehen sich jeweils auf die drei Texte „Saccharomyces“, „Drosophila“ und „Human“.

- Ein Array der Größe n mit Zahlen im Bereich $[1..n]$ benötigt $n \cdot \log_2(n)$ Bits Speicher (Hier: 34 MB, 527 MB bzw. 3,48 GB).
- Der Text der Länge n bei einem Alphabet der Größe 5 belegt $n \cdot \lceil \log_2(5) \rceil$ Bits Speicher (Hier: 4,4 MB, 57,9 MB bzw. 358 MB).

- Ein Wavelet Tree der Größe n mit $|\Sigma| = 5$ benötigt etwa $n \cdot \log_2(5) \cdot 1{,}25$ Bits Speicher[5] (Hier: 4,2 MB, 56 MB bzw. 346 MB).

Ein gewöhnlicher Desktop-PC besitzt heutzutage selten mehr als vier Gigabyte Arbeitsspeicher. Für sehr große Texte scheiden damit die Indexe, die das Suffix Array, das inverse Suffix Array oder die Ψ-Funktion komplett speichern, praktisch aus. Dagegen ist der Wavelet Tree eines Textes T praktisch genau so groß wie der Text.

Laufzeiten

Wie erwartet schneidet der Index ② im Vergleich sehr schlecht ab. Da er nur unwesentlich größer, aber viel langsamer als Index ① ist hat er somit keine praktische Bedeutung. Die Suche auf dem Compressed Suffix Array ④ funktioniert genau so wie auf dem Index ①. Da die Ψ-Funktion bei ④ komprimiert ist, ist auch der Zugriff darauf langsamer. Und da die *SA*-Werte erst aufwändig berechnet werden müssen ist ④ um ein Vielfaches langsamer. Ähnlich verhält es sich auch bei Index ⑤. Der Speicherplatzverbrauch ist zwar wesentlich geringer als bei ③, der Preis den man dafür bei der Laufzeit zahlt ist jedoch sehr hoch.

Man erkennt, dass der Index ③ in jeder Kategorie am schnellsten ist, und somit bei Anwendungen, bei denen die Laufzeit sehr wichtig ist, durchaus seine Daseinsberechtigung hat. Der bidirektionale Wavelet-Index ⑥ verbindet jedoch den sehr geringen Speicherplatzbedarf mit einer guten Laufzeit. Einzig die Tatsache, dass die *SA*-Werte hier nicht direkt vorliegen, lässt ihn bei den Testfällen mit vielen Treffern eben etwas langsamer werden.

Affix Array

Der Speicherplatzbedarf des Affix Arrays wird in [Str07] je nach Implementierung fest mit 18 bis 20 Bytes pro Zeichen angegeben. Für die drei Texte wäre das entsprechende Affix Array damit 209 MB, 2,7 GB bzw. 16,8 GB groß und läge hier im Vergleich weit abgeschlagen auf dem letzten Platz. Zur Laufzeit der bidirektionalen Suche kann hier keine verbindliche Aussage getroffen werden. Da die Operationen auf dem Affix Array komplizierter als auf dem bidirektionalen Wavelet-Index sind kann man kann aber annehmen, dass es auch langsamer als dieser ist.

[5] Faktor 1,25 für die Rank-Unterstützung der Bit-Vektoren

Index	Größe	`hairpin1` 4 Treffer	`hairpin2` 2 Treffer	`hairpin4` 24 Treffer	`hloop(5)` 62 Treffer	`acloop(5)` 1 Treffer	`acloop(10)` 0 Treffer	`acloop(15)` 0 Treffer
① SA, Text, Ψ	74 MB	2174 ms	9 ms	86 ms	5568 ms	198 ms	61 ms	20 ms
② SA, SA^{-1}	70 MB	6123 ms	28 ms	239 ms	15771 ms	554 ms	198 ms	60 ms
③ SA, Text, Occ	44 MB	336 ms	0 ms	11 ms	820 ms	28 ms	12 ms	7 ms
④ CSA/Ψ + Text	23 MB	8386 ms	36 ms	336 ms	21751 ms	778 ms	310 ms	198 ms
⑤ CSA/Occ + Text	12 MB	5021 ms	22 ms	208 ms	13053 ms	462 ms	246 ms	266 ms
⑥ BWI	9 MB	456 ms	3 ms	17 ms	1141 ms	41 ms	14 ms	8 ms

Tabelle 8.1: Laufzeitvergleich auf dem Text „Saccharomyces“ (12,2 Mio Zeichen)

Index	Größe	`hairpin1` 107 Treffer	`hairpin2` 22 Treffer	`hairpin4` 402 Treffer	`hloop(5)` 1003 Treffer	`acloop(5)` 17 Treffer	`acloop(10)` 7 Treffer	`acloop(15)` 8 Treffer
① SA, Text, Ψ	1142 MB	25287 ms	114 ms	1037 ms	64896 ms	2170 ms	908 ms	443 ms
② SA, SA^{-1}	1084 MB	61057 ms	273 ms	2506 ms	156166 ms	5181 ms	2604 ms	1511 ms
③ SA, Text, Occ	661 MB	2039 ms	9 ms	83 ms	5328 ms	176 ms	80 ms	73 ms
④ CSA/Ψ + Text	307 MB	69359 ms	308 ms	2878 ms	178491 ms	5967 ms	3085 ms	2392 ms
⑤ CSA/Occ + Text	164 MB	31616 ms	145 ms	1349 ms	83396 ms	2809 ms	1596 ms	2225 ms
⑥ BWI	126 MB	2521 ms	13 ms	147 ms	6441 ms	207 ms	105 ms	85 ms

Tabelle 8.2: Laufzeitvergleich auf dem Text „Drosophila“ (162 Mio Zeichen)

Index	Größe	`hairpin1` 2343 Treffer	`hairpin2` 286 Treffer	`hairpin4` 3098 Treffer	`hloop(5)` 14870 Treffer	`acloop(5)` 294 Treffer	`acloop(10)` 224 Treffer	`acloop(15)` 75 Treffer
① SA, Text, Ψ	7688 MB	91640 ms	425 ms	3635 ms	226922 ms	7809 ms	3831 ms	1490 ms
② SA, SA^{-1}	7322 MB	272641 ms	1249 ms	10842 ms	676201 ms	23218 ms	11368 ms	6026 ms
③ SA, Text, Occ	4408 MB	8855 ms	41 ms	365 ms	22208 ms	781 ms	336 ms	249 ms
④ CSA/Ψ + Text	2045 MB	306472 ms	1423 ms	12392 ms	755802 ms	26291 ms	13058 ms	7822 ms
⑤ CSA/Occ + Text	1053 MB	137371 ms	651 ms	5642 ms	345860 ms	12174 ms	6381 ms	6528 ms
⑥ BWI	799 MB	11053 ms	79 ms	792 ms	28373 ms	958 ms	420 ms	274 ms

Tabelle 8.3: Laufzeitvergleich auf dem Text „Human“ (1 Mrd Zeichen)

Kapitel 9

Schluss

Die bidirektionale indexbasierte Suche in Texten ist ein Gebiet, in dem bisher noch nicht viele Ergebnisse veröffentlicht wurden. In dieser Arbeit wurde ein Überblick über Vorwärtssuche und Rückwärtssuche gegeben, und es wurden die beiden Konzepte *Affixbaum* von [Sto95] und *Affix Array* von [Str07] vorgestellt. Durch die Kombination von Vorwärts- und Rückwärtssuche wurden mehrere Indexe entwickelt, die bidirektionale Suche unterstützen. Alle diese Ansätze benötigen jedoch relativ viel Speicherplatz, und sind damit für die Analyse von sehr großen Daten nur begrenzt einsetzbar. Verwendet man komprimierte Versionen des Suffix Arrays, so bezahlt man den geringeren Speicherplatzverbrauch mit großen Einbußen in der Laufzeit der bidirektionalen Suche. Der neu vorgestellte *bidirektionale Wavelet-Index* vereint die Vorteile all dieser Ansätze, denn er benötigt nur etwa den 2,5-fachen Speicher des ursprünglichen Textes und unterstützt die bidirektionale Suche in linearer Zeit, was sich auch im experimentellen Vergleich entsprechend bestätigt hat.

An dieser Stelle soll noch ein Ausblick auf eine weitere Optimierung für den bidirektionalen Wavelet-Index gegeben werden. Falls sich die relative Häufigkeit der einzelnen Buchstaben stark unterscheidet, könnte man den Wavelet Tree auch anders definieren. In [BCF$^+$07] wird ein Wavelet Tree vorgestellt, der die Form eines Huffman-Baumes hat. Zunächst wird für den Text ein Huffman-Code erstellt. Die Verzweigung im Wavelet Tree erfolgt dann nicht mehr über die Position eines Buchstabens im Alphabet, sondern über seinen Präfixcode. Der Baum wäre dann nicht mehr balanciert, sondern die Tiefe eines Blattes ist umso größer, je seltener der Buchstabe im Text vorkommt, und damit werden die Bit-Vektoren im Wavelet Tree kürzer und

verbrauchen weniger Speicherplatz.

Die Funktion *getBounds* beruht auf der Tatsache, dass die Blätter im Wavelet Tree von links nach rechts alphabetisch sortiert sind. Da der Huffman-Baum diese Eigenschaft im Allgemeinen nicht besitzt, müsste man bereits bei der Konstruktion des Suffix Arrays die Ordnung des Alphabets Σ entsprechend so verändern, dass diese Eigenschaft wieder gegeben ist.

Bei DNA-Texten hat sich allerdings gezeigt, dass die vier Buchstaben $\{A, C, G, T\}$ alle ähnlich häufig vorkommen, weshalb sich eine solche Optimierung in diesem Fall nicht lohnt. Bei der Suche auf anderen Texten könnte sie jedoch interessant sein.

Anhang A

Implementierung

Die Implementierung steht unter der *GNU General Public License* und ist unter `http://www.uni-ulm.de/in/theo/research/seqana` verfügbar. Dort gibt es außerdem eine ausführliche Dokumentation. Die folgenden Abschnitte sollen nur einen groben Überblick verschaffen. Zum Übersetzen und zum Ausführen wird die Bibliothek *sdsl* benötigt.

A.1 Demonstrationsumgebung

In der Datei `text_indexes.cpp` befinden sich neben `main()` drei weitere Methoden. Diese sollen als Beispiel dienen wie Indexe erzeugt werden können und wie die Suche funktioniert.

- ```
  template <class Index>
  void test_index_forward();
  ```

  Diese Methode erstellt einen Index des Typs `Index` vom Text „`el_anele_lepanelen$`“ und sucht vorwärts nach dem Muster „`ele`“.

- ```
  template <class Index>
  void test_index_backward();
  ```

  Diese Methode erstellt einen Index des Typs `Index` vom Text „`el_anele_lepanelen$`“ und sucht rückwärts nach dem Muster „`ele`“.

A.2 Eigentliche Testumgebung

Das Tool „createindex“

Die Erstellung eines Volltextindex ist der „Flaschenhals“ der indexbasierten Suche, da er mit Abstand am meisten Zeit benötigt. Deshalb kann mit dem Tool *createindex* ein Index erstellt und in eine Datei gespeichert werden. Dann muss für eine Suche nur noch der fertige Index geladen werden, was weitaus weniger Rechenaufwand darstellt. Das Tool erwartet als Parameter die Eingabedatei, die Nummer des zu erstellenden Index und die Ausgabedatei. Um von der Datei `yeast.raw` den bidirektionalen Wavelet-Index zu erstellen und in die Datei `yeast.index` zu speichern verwendet man den folgenden Aufruf.

```
./createindex yeast.raw 6 yeast.index
```

Der zweite Parameter spezifiziert, welcher Index verwendet wird, wobei die folgende Auswahl möglich ist (Details siehe weiter unten). Die Nummerierung entspricht der aus Abschnitt 8.2.

```
 # | Index
---+-----------------------------------
 1 | index_sa_text_psi
 2 | index_sa_isa
 3 | index_sa_text_occ<>
 4 | index_csa_psi_text<>
 5 | index_csa_fmi_text<>
 6 | index_bidirectional_waveletindex<>
```

Das Tool „starttestcases“

Mit diesem Tool können die Ergebnisse aus dieser Arbeit reproduziert werden. Für die erstellte Index-Datei `yeast.index` startet man das Programm mit

```
./starttestcases yeast.index
```

Es wird automatisch erkannt, um welchen Index es sich bei der Datei handelt. Dieser wird aus der Datei geladen und dann werden die sieben Testfälle

nacheinander ausgeführt. Ein optionaler zweiter Parameter gibt außerdem die Anzahl der Wiederholungen der Testfälle an. Es ist in dieser Version noch nicht möglich, eigene Testfälle zu spezifizieren.

A.3 Die bidirektionale Suche

Die Datei `bidirectional.hpp` enthält hauptsächlich die eigentlichen Testfälle aus Kapitel 8. Diese sind alle „hart codiert“, es ist momentan noch nicht möglich eigene Testfälle zu spezifizieren. Alle Methoden besitzen einen Template-Parameter `Index`. Damit ist eine leichte Austauschbarkeit der verschiedenen Implementierungen gegeben. Der Parameter `out` erhält in der Regel den Stream zu einer Datei. Damit kann später geprüft werden, ob die jeweilige Implementierung das richtige Ergebnis gefunden hat.

- ```
  template<class Index>
  size_t locate_testpattern_hairpin1 (Index &index,
            std::ostream &out)
  ```

- ```
  template<class Index>
  size_t locate_testpattern_hairpin2 (Index &index,
            std::ostream &out)
  ```

- ```
  template<class Index>
  size_t locate_testpattern_hairpin4 (Index &index,
            std::ostream &out)
  ```

- ```
  template<class Index>
  size_t locate_testpattern_hloop (Index &index,
            size_t looplength, std::ostream &out)
  ```

- ```
  template<class Index>
  size_t locate_testpattern_acloop (Index &index,
            size_t looplength, std::ostream &out)
  ```
  ```

A.4 Indexe

Gemeinsame Oberklasse

Alle Indexe haben die gemeinsame Oberklasse *index*. Diese enthält alles, was die Indexe gemein haben. Dazu gehören die öffentlichen Attribute

- `const uint8_t sigma`

  Die Alphabetgröße des Textes.

- `const size_t C[257]`

  Das vorgestellte C-Array. Es enthält an der Stelle c die Position des lexikographisch kleinsten Suffixes, welches mit c beginnt.

- `const unsigned char char2comp[256]`

  Dies ist ein Array welches für die Umwandlung von Zeichen aus dem ASCII-Alphabet ins Arbeitsalphabet `[0..sigma-1]` benutzt wird.

- `const unsigned char comp2char[256]`

  Dies ist ein Array welches für die Umwandlung von Zeichen aus dem Arbeitsalphabet `[0..sigma-1]` ins ASCII-Alphabet benutzt wird.

Außerdem besitzt diese Klasse drei Methoden, welche für die erbenden Klassen sichtbar sind:

- `void setText (const unsigned char *str, size_t len)`

  Diese Methode initialisiert bei der Erstellung des Indexes die Attribute `sigma`, `C`, `char2comp` und `comp2char`.

- `size_t superserialize (std::ostream &out) const`

  Diese Methode serialisiert die vier Attribute `sigma`, `C`, `char2comp` und `comp2char` in den Output Stream `out`. Sie wird von allen erbenden Indexen beim Speichern in eine Datei aufgerufen.

- `void superload (std::istream &in)`

  Diese Methode lädt die Attribute `sigma`, `C`, `char2comp` und `comp2char` aus dem Input Stream `in`. Sie wird von allen erbenden Indexen beim Laden aus einer Datei aufgerufen.

Interface

Jeder Index muss von der Oberklasse *index* erben. Er muss einen leeren Standard-Konstruktor besitzen und einen Konstruktor, der als Parameter einen C-String erhält. Das abschließende Nullbyte zählt zum Eingabetext dazu und entspricht dem Zeichen „$“ in dieser Arbeit. Er muss außerdem folgende Daten und Methoden bereitstellen

- `struct searchstate`

  Ein Datentyp, der den „inneren Zustand“ der Backtracking-Suche speichert. In der Regel ist dies das Intervall $[i..j]$ und die Länge des bisher gefundenen Musters.

- `void init_search_state(searchstate &s)`

  Initialisiert ein searchstate-Objekt, so dass s das Intervall des leeren Wortes ist.

- `size_t forward_search(unsigned char c, searchstate &s)`

  Ausgehend vom Zustand s wird eine Vorwärtssuche nach dem Buchstaben s durchgeführt und das Ergebnis in s gespeichert. Der Rückgabewert ist die Größe des gefundenen Intervalls.

- `size_t backward_search(unsigned char c, searchstate &s)`

  Ausgehend vom Zustand s wird eine Rückwärtssuche nach dem Buchstaben s durchgeführt und das Ergebnis in s gespeichert. Der Rückgabewert ist die Größe des gefundenen Intervalls.

- `void extract_sa(size_t i, size_t j, int_vector<> &occ)`

  Kopiert die Werte des Suffix Arrays im Intervall von i (einschließlich) bis j (ausschließlich) in den Vektor *occ*.

- `size_t serialize(std::ostream &out) const`

  Diese Methode serialisiert die Datenstruktur in den Outputstream `out` und gibt ihren benötigten Speicherplatz in Bytes zurück.

- `void load(std::istream &in) const`

  Diese Methode lädt die Datenstruktur aus dem Inputstream `in`.

- `size_t used_bytes() const`

  Gibt den Speicherplatz in Bytes zurück, den der Index benötigt.

Verschiedene Implementierungen

Es wurden die sechs verschiedenen Indexe implementiert, welche auch in der Ausarbeitung erwähnt werden. Da sich alle Indexe an das beschriebene Interface halten ist eine leichte Austauschbarkeit dieser Implementierungen gegeben. Alle Methoden die einen Index verwenden besitzen hierfür einen Template-Parameter `Index`.

1. `index_sa_text_psi.hpp`

   Index ① (SA, Text, Ψ)

2. `index_sa_isa.hpp`

   Index ② (SA, SA^{-1})

3. `index_sa_text_occ.hpp`

   Index ③ (SA, Text, Occ)

4. `index_csa_sadakane.hpp`

   Index ④ (CSA/Ψ). Die Implementierung des verwendeten Compressed Suffix Arrays stammt aus der *sdsl*-Bibliothek (`csa_sada_prac.hpp`).

5. `index_csa_fmi.hpp`

   Index ⑤ (CSA/Occ)

6. `index_bidirectional_waveletindex.hpp`

   Index ⑥ (*Bidirektionaler Wavelet-Index*)

A.5 Weitere Datenstrukturen

- `wavelet_tree.hpp`

  Dies ist die Implementierung des Wavelet Trees. Dieser unterstützt sowohl die Funktionen Occ und *extract* als auch den vorgestellten neuen *getBounds*-Algorithmus.

Abbildungsverzeichnis

Tabellenverzeichnis

Algorithmenverzeichnis

Literaturverzeichnis

[AKO04] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing Suffix Trees with Enhanced Suffix Arrays. *J. of Discrete Algorithms*, 2(1):53–86, 2004.

[BCF+07] Nieves R. Brisaboa, Yolanda Cillero, Antonio Farina, Susana Ladra, and Oscar Pedreira. A New Approach for Document Indexing Using Wavelet Trees. In *DEXA '07: Proceedings of the 18th International Conference on Database and Expert Systems Applications*, pages 69–73, Washington, DC, USA, 2007. IEEE Computer Society.

[BW94] M. Burrows and D. J. Wheeler. A block-sorting lossless data compression algorithm. Technical Report 124, Digital SRC Research Report, 1994.

[FM00] P. Ferragina and G. Manzini. Opportunistic data structures with applications. In *FOCS '00: Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, page 390, Washington, DC, USA, 2000. IEEE Computer Society.

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

[Gog09] Simon Gog. Broadword Computing and Fibonacci Code Speed Up Compressed Suffix Arrays. In *SEA '09: Proceedings of the 8th International Symposium on Experimental Algorithms*, pages 161–172, Berlin, Heidelberg, 2009. Springer-Verlag.

[GSKS01] Stefan Gräf, Dirk Strothmann, Stefan Kurtz, and Gerhard Steger. HyPaLib: a Database of RNAs and RNA Structural Elements defined by Hybrid Patterns. *Nucleic Acids Res.*, 29(1):196–198, 2001.

[GV00] Roberto Grossi and Jeffrey Scott Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching. *STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing*, pages 397–406, 2000.

[Jac89] G. Jacobson. Space-efficient static trees and graphs. In *SFCS '89: Proceedings of the 30th Annual Symposium on Foundations of Computer Science*, pages 549–554, Washington, DC, USA, 1989. IEEE Computer Society.

[Maa00] Moritz Gerhard Maaß. Linear Bidirectional On-Line Construction of Affix Trees. *COM '00: Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pages 320–334, 2000.

[MM90] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 319–327, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[Mä00] Veli Mäkinen. Compact Suffix Array. In *COM '00: Proceedings of the 11th Annual Symposium on Combinatorial Pattern Matching*, pages 305–319, London, UK, 2000. Springer-Verlag.

[PST07] Simon J. Puglisi, W. F. Smyth, and Andrew H. Turpin. A taxonomy of suffix array construction algorithms. *ACM Comput. Surv.*, 39(2):4, 2007.

[Sad03] Kunihiko Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.

[Sto95] Jens Stoye. Affixbäume. Diplomarbeit, Universität Bielefeld, Technische Fakultät, 1995.

[Str07] Dirk Strothmann. The affix array data structure and its applications to RNA secondary structure analysis. *Theor. Comput. Sci.*, 389(1-2):278–294, 2007.

[Wei73] Peter Weiner. Linear pattern matching algorithms. In *SWAT '73: Proceedings of the 14th Annual Symposium on Switching and Automata Theory (swat 1973)*, pages 1–11, Washington, DC, USA, 1973. IEEE Computer Society.

Name: Thomas Schnattinger Matrikelnummer: 575619

Erklärung

Hiermit versichere ich, dass ich die vorliegende Diplomarbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ulm, den 12. Januar 2010 .....

Thomas Schnattinger