Seqs Zusammenfassung

February 9, 2024

Contents

1 Suffix Arrays

Sei $S = c_1c_2...c_n$ ein String. der i-te Suffix S_i ist definiert als der String $c_ic_{i+1}...c_n$. Ein Suffix Array nimmt die Menge aller n Suffixe und sortiert sie lexikographisch. Dafür ist eine Ordnung $<$ auf dem Alphabet Σ gegeben. Die naive Implementierung, die Substrings anhand ihrer ersten Character vergleicht, hat eine Laufzeit von $\mathcal{O}(n^2)$

1.1 Skew Algorithmus

Dieser ist ein Beispiel für einen linear-Zeit Algorithmus. Dieser wird im Folgenden am Beispiel $S = ctaataatg$ erklärt.
Schritt 1. Als erstes teilt der Algorithmus den String S in zwei Teile. Sei S_i der Substring beginnend an der Stelle i. Man definiert die Menge $S_L = \{S_i : \ i \mod 3 \neq 1\}$ und $S_R = \{S_i : \ i \mod 3 = 1\}$. Im Beispiel besteht S_L aus

- taataatg
- aataatg
- aatg
- tg
- g

und S_R aus

- ctaataatg
- ataatg
- atg

Die Suffixmenge von S_L wird mittels Radixsort rekursiv in linearer Zeit sortiert, dabei wird nur bezüglich der ersten drei Stellen aller Suffixe sortiert. Dadurch erhält man

- aat (S_3)
- aat (S_6)
- g\$\$ (S_9)
- taa (S_2)
- taa (S_5)

- tg$ (S_8)

Wären nun alle dieser Tripel echt verschieden, so wäre diese Liste bereits total sortiert, was in diesem Beispiel aber nicht der Fall ist. Die Tripel werden anschließend von oben nach unten nummeriert, wobei gleiche Tripel die gleiche Nummer erhalten. Diese werden als lexikografische Namen bezeichnet. Man definiert $\overline{S}$ als der String der lexikografischen Namen der Substrings S_i mit $i \mod 3 = 2$ konkateniert mit dem String der lexikografischen Namen der Substrings S_i mit $i \mod 3 = 0$. Im Beispiel ist $\overline{S} = 334112$. Auf $\overline{S}$ wird der Algorithmus rekursiv angewandt. Von $\overline{S}$ erhalten wir die Suffixe

1. 334112
2. 34112
3. 4112
4. 112
5. 12
6. 2

Diese Liste wird im Sinne eines Suffixarrays sortiert, wodurch man

- 112
- 12
- 2
- 334112
- 34112
- 4112

erhält. Für jeden dieser Suffixe erhält man einen dazugehörigen Suffix vom ursprünglichen String S und seine Position in i in S. In der Tat erhält man die Zuordnungen: $\overline{S}_1 \equiv S_2$, $\overline{S}_2 \equiv S_5$, $\overline{S}_3 \equiv S_8$, ...
Im Allgemeinen gilt $\overline{S}_{\frac{i+1}{3}} \equiv S_i$ wenn $i = 3 \mod 2$. Für $i = 0 \mod 3$ hingegen gilt $\overline{S}_{\frac{n+i}{3}} \equiv S_i$. Dafür wird die Notation

$$\tau(i) = \begin{cases} \frac{i+1}{3}, & i = 2 \mod 3 \\ \frac{n+i}{3}, & i = 0 \mod 3 \end{cases}$$

Da wir aber die Richtung von $\overline{S}$ zu S benötigen, brauchen wir die Umkehrabbildung von τ:

$$\tau^{-1}(j) = \begin{cases} 3j-1, \ 1 \leq j \leq \frac{n}{3} \\ 3j-n, \ \frac{n}{3} < j \leq \frac{2n}{3} \end{cases}$$

Mittels τ^{-1} können nun die Suffixe von $\overline{S}$ in jene von S umgerechnet werden, die dadurch sortiert sind.
Schritt 2: In diesem Teil müssen die Suffixe in S_R sortiert werden. In diesem Schritt wird jeder Suffix von seinem Anfangsbuchstaben getrennt. Man erhält

- $S_1 = cS_2$
- $S_4 = aS_5$
- $S_7 = aS_8$

Wendet man nun Radixsort auf die ersten Buchstaben an, so werden diese Strings automatisch sortiert.
Schritt 3: In diesem Schritt werden die beiden Sublisten wieder zu einem Suffixarray zusammengefügt. Wenn das in der naiven Merge-Sort Implementierung durchgeführt wird, so würde das in quadratischer Laufzeit resultieren. Deswegen wird wieder die Strategie von Schritt 2 verwendet; die Liste S_R wird wieder aufgespalten in erste Buchstaben und Suffixe von S_L. Dadurch sind die beiden Suffixe aber nicht zwingend in der selben Gruppe, weswegen auch noch der zweite Buchstabe abgespalten werden kann. Der Merge Teil kann nun zuerst auf Basis der ersten beiden Buchstaben entschieden werden, außer wenn die beiden Buchstaben der zu vergleichenden Suffixe gleich sind. Dieses Verfahren funktioniert für Substrings S_i aus S_L mit $i = 0 \mod 3$. Für Substrings S_i aus S_L mit $i = 2 \mod 3$ hingegen würde die abgespaltenen Suffixe wieder in verschiedenen Listen liegen. Deshalb wird in diesem Fall nur ein Buchstabe abgespalten.
Der Vergleich der beiden abgespaltenen Suffixe funktioniert aber nur in linearer Zeit. Deswegen wird hierfür das sogenannte inverse Suffixarray eingeführt.

Definition 1.1.1 (inverses Suffixarray (ISA))**.** Für ein Suffixarray SA mit $k \to SA[k]$ ist das inverse Suffixarray definiert durch $ISA[SA[k]] = k$.

Allgemein gilt dann $S_i < S_j \Leftrightarrow ISA[\tau(i)] < ISA[\tau(j)]$.

1.2 Longest common prefixes

Das LCP-Array ist definiert auf der Sortierung des Suffix-Arrays. $LCP[i]$ speichert die Länge des längsten gemeinsamen Präfixes von $S_{SA[i-1]}$ und $S_{SA[i]}$. Insbesondere ist daher $LCP[1]$ undefiniert und wird daher auf -1 gesetzt. Algorithmus:

1. setze $LCP[1] = 1$ und $LCP[n+1] = -1$
2. konstruiere ISA aus SA (linear)
3. man beginnt mit dem längsten Suffix (bekommt man konstant aus ISA) und vergleicht mit dem Suffix in der Zeile oben
4. dieser Wert l wird eingeschrieben und man geht zum zweitlängsten Suffix
5. Wieder vergleichen mit dem nächst-höheren Suffix im SA. Dort ist die Länge mindestens $l - 1$, daher erst ab dem l-ten Buchstaben vergleichen.
6. iterativ fortsetzen

1.3 LCP-Intervalle

Ein Intervall $[i, i]$ wird als Singleton bezeichnet. Ein Intervall $[i, j]$ in einem LCP Array mit $1 \leq i < j \leq n$ heißt LCP-Intervall mit LCP-Wert l, wenn

1. $LCP[i] \leq l$
2. $LCP[k] \geq l$ für alle k mit $i + 1 \leq k \leq j$
3. $LCP[k] = l$ für mindestens ein $i + 1 \leq k \leq j$
4. $LCP[j + 1] < l$

Wenn man jeden String mit einem Dummy-Wert (z.B. \$), dann ist das Intervall $[1, n]$ immer ein LCP-Intervall.

Lemma 1.3.1. *Zwei LCP-Intervalle können sich nicht überlappen, d.h. es muss einer der folgenden Fälle eingetreten sein:*

1. *Intervall $[i, j]$ ist echt in $[p, q]$ enthalten oder*
2. *Intervalle $[i, j]$ und $[p, q]$ sind disjunkt*

Definition 1.3.2. Ein m-Intervall ist in einem l-Intervall enthalten, falls höchstens einer der Ränder gleich ist und $m > l$ gilt.
Das l-Intervall $[i, j]$ ist das umschließende Intervall von $[p, q]$, wenn $[p, q]$ in $[i, j]$ enthalten ist und es kein anderes in $[i, j]$ enthaltenes Intervall gibt, das $[p, q]$ enthält. Diese Intervalle werden Eltern und Kinder-Intervalle genannt.
Ein Intervall der Form $[k, k]$ wird Singleton genannt. Das Elternintervall ist das kleinste LCP-Intervall, dass den Singleton enthält.

1.3.1 Range Minimum Queries

Gegeben ist ein Array A von n ganzen Zahlen. Ein Range Minimum Query gibt für das Array A und Indices i, j ein k zurück, sodass

$$A[k] = \min_{i \leq m \leq j} A[m]$$

So ein Query wird im Folgenden als $RMQ_A(i, j)$ bezeichnet. Man kann zeigen, dass A in linearer Zeit bearbeitet werden kann, sodass das RMQ in konstanter Zeit ausgeführt werden kann.

Definition 1.3.3. Aus der Definition eines LCP-Arrays folgt, dass es einen Index k gibt, an dessen Stelle der LCP-Wert l angenommen wird. All diese Indizes werden in Folge als l-Index bezeichnet.

1.3.2 Kind-Intervalle finden

Lemma 1.3.4. *Sei $[i, j]$ ein l-Intervall. Seien $i_1 < i_2 < ... < i_k$ die l-Indizes in aufsteigender Reihenfolge. Dann sind die Kind-Intervalle gerade*

$$[i, i_1 - 1], [i_1, i_2 - 1], \ldots [i_k, i_{k+1} - 1], \ldots [i_k, j]$$

Definition 1.3.5. Für $2 \leq i \leq n$ definiere

$$PSV[i] = \max\{j | 1 \leq j < i \text{ und } LCP[i] < LCP[j]\}$$

und

$$NSV[i] = \min\{j | i < j \leq n + 1 \text{ und } LCP[i] < LCP[j]\}$$

Lemma 1.3.6. *Sei $2 \leq k \leq n$ und $LCP[k] = l$. Dann ist $[PSV[k], NSV[k] - 1]$ ein LCP Intervall mit LCP-Wert l*

Lemma 1.3.7. *Sei $[i, j] \neq [1, n]$ ein LCP-Intervall mit $LCP[i] = p$ und $LCP[j + 1] = q$.*

- *Ist $p = q$, dann*
  - *ist das Elternintervall $[PSV[i], NSV[i] - 1]$*
  - *hat das Elternintervall den LCP-Wert $p = q$*
  - *sind i und $j + 1$ aufeinander folgende p-Indizes des Elternintervall*
- *Ist $p > q$, dann*

- *ist* $[PSV[i], j]$ *das Elternintervall*
- *hat das Elternintervall den LCP-Wert* p
- *ist* i *der letzte* p*-Index vom Elternintervall*

- *Ist* $p < q$*, dann*
  - *ist das Elternintervall* $[i, NSV[j+1]-1]$
  - *hat das Elternintervall den LCP-Wert* q
  - *ist* $j+1$ *der letzte* q*-Index des Elternintervalls*

Lemma 1.3.8. *Der LCP-Wert des Elternintervall von* $[i, j]$ *ist* $\max\{LCP[i], LCP[j+1]\}$.

1.4 Bottom-up Konstruktion des LCP-Baums

Man definiere $\langle p, a, b, A\rangle$ als das Viertupel mit LCP-Wert p, Intervallanfang a, -ende b und die Liste der Kindintervalle A. Algorithmus 5 aus dem Skript funktioniert in linearer Zeit.

Theorem 1.4.1. *Sei top das oberste Intervall auf dem Stack und* top_{-1} *das darunter für ein* k *der For-Schleife. Bemerke* $top.lcp > top_{-1}.lcp$*. Dann, bevor top gepoppt wird, gilt*

- *Ist* $LCP[k] \leq top_{-1}.lcp$*, dann ist top ein Kindintervall von* top_{-1}.
- *Ist* $LCP[k] > top_{-1}.lcp$*, dann ist top ein Kindintervall des ersten LCP-Intervall mit LCP-Wert* $LCP[k]$*, das* k *enthält. Genauer gesagt, ist top ein Kindintervall von* $[top.lb, NSV[k]-1]$.

Im Folgenden wird eine Anwendung der Bottom-Up Traversierung von LCP-Intervall-Bäumen dargestellt. Diese können sich zum Beispiel hinter der Funktion *process* im LCP-Intervall-Baum Algorithmus verstecken.

Example 1.4.2. *Finde alle Teilstrings eines Strings* S*, die mindestens* p *und höchstens* q *mal in* S $(2 \leq p \leq q)$ *vorkommen.*
Idee: Folgender Algorithmus

1. *process(lastIntervall)*
2. *for each* $\langle l, i, j+1, []\rangle in lastInterval.childlist$
3. *if* $p \leq (j-i+1)$ *and* $q \geq (j-i+1)$
4. *output* $(lastInterval.lcp + 1, l, [i, j])$

Die Interpretation der Ausgabe ist zu verstehen als:
Verwende das Intervall $[i, j]$ und betrachte die Präfixe der Länge von $lcp + 1$ bis l. Wenn am Ende die Kindliste auf leer gesetzt wird, dann kann man zusätzlich verhindern, dass der gesamte Baum konstruiert wird.

1.5 Suffix Tree

Sei S ein String der Länge n. Ein Suffix-Baum der Länge $S\$$ ist ein Wurzelbaum $ST(S\$)$ mit den folgenden Eigenschaften

1. ST hat $n + 1$ viele Blätter nummeriert mit 1 bis $n + 1$.
2. Jeder innere Knoten hat mindestens zwei innere Knoten
3. Jede Kante hat als Kantenlabel einen nicht leeren Substring von $S\$$
4. Für jeden Knoten α und jedes $a \in \Sigma$ gibt es nur eine Kante mit $a \Rightarrow \beta$ mit Label av für einen String v und einen Knoten
   beginbeta in ST
5. Für das Blatt i ist die Konkatenation der Kantenlabels des Pfades zu i genau der Suffix S_i.

1.6 Burrows-Wheeler Transformation

Die Transformation konvertiert einen String der Länge n in den String $BWT[1, n]$ definiert durch $BWT[i] = S[SA[i] - 1]$ für alle i mit $SA[i] \neq 1$ und $BWT[i] = \$$ sonst.

Definition 1.6.1. Ein Substring w von S ist ein Repeat, wenn er mindestens einmal in S vorkommt. Wir definieren die Mengen

$$lc(w) = \{BWT[k] | i \leq k \leq j\}$$

$$rc(w) = \{S[SA[k] + l] | i \leq k \leq j\}$$

Ein Repeat ist supermaximal wenn

$$|lc(w)| = |rc(w)| = occ(w)$$

Mithilfe der bottom-up Traversierung soll nun ein Algorithmus entwickelt werden, der alle supermaximalen Repeats bestimmt. Es fällt auf, dass der rc-Wert für den durch ein LCP-Intervall bestimmten Substring gleich occ ist, wenn das LCP-Intervall kein Kind hat. Der lc Wert kann ganz leicht mit einer Hashmap auf die Buchstaben mit occ verglichen werden.

Definition 1.6.2. Ein Repeat ist links-maximal, wenn $|lc(w)| \geq 2$ und rechts-maximal, wenn $|rc(w)| \geq 2$. Ein links- und rechts-maximaler Repeat ist maximal.

Lemma 1.6.3. *Jeder rechts-maximale Repeat korrespondiert zu einem LCP-Intervall.*

Wenn maximale Repeats gesucht werden, so ist die Rechts-Maximalität bereits gegeben. Die Links-Maximalität ist noch leichter. Um lineare Laufzeit zu gewinnen, muss die Links-Maximalität der Kindintervalle an die Elternintervalle weitergegeben werden.
Speichere die größte Position lastdiff mit $1 \leq lastdiff \leq i-1$ an der ein Buchstabe in $BWT[1...i-1]$ festgestellt wurde, d.h.

$$BWT[lastdiff - 1] \neq BWT[lastdiff]$$

Lemma 1.6.4 (Berechnung des längsten Repeats)**.** *Um den längsten Repeat zu bestimmen, genügt es, einmal über die LCP Tabelle zu iterieren und den maximalen LCP Wert zu bestimmen. Es wird der String ausgegeben, der an der Stelle des Maximums mit der Länge des LCP Werts steht. Es kann in konstanter Zeit darauf geachtet werden, dass jeder Repeat nur einmal ausgegeben wird.*

1.7 De Bruijn Sequenzen

Definition 1.7.1. Eine De Bruijn Sequenz der Ordnung k ist ein zyklischer String, der jedes String aus Σ^k genau einmal enthält. Zyklisch bedeutet hierbei, dass der Teilstring am Ende beginnen und am Anfang enden kann. Ein zyklischer De Bruijn String kann in einen nicht zyklischen String konvertiert werden, indem die ersten $k-1$ Zeichen wieder am Ende angehängt werden.

Definition 1.7.2 (De Bruijn Graph)**.** Der De Bruijn Graph für k über das Alphabet Σ enthält einen Knoten für jeden String aus Σ^{k-1}. Eine gerichtete Kante (u, v) existiert genau dann, wenn es einen String $w \in \Sigma^k$ sodass u ein Präfix und v ein Suffix von w ist. Von jedem Knoten gehen genau $|\Sigma|$ Kanten aus.

Eine lineare De Bruijn Sequenz hat Länge $n = |\Sigma|^k + k - 1$ und der maximale LCP Wert m wird an mehreren Stellen in der LCP Tabelle angenommen.

Lemma 1.7.3. *Die gesamte Ausgabelänge des Algorithmus zu Berechnung aller längsten Repeats für eine De Bruijn Sequenz S der Ordnung k ist $\Omega(n \log n)$, wobei $n = 2^k + k - 1$.*

2 komprimierte Ganztext Indizes

2.1 Burrows-Wheeler Transformation

Die BWT für einen String S entsteht aus einer Rotationmatrix des Strings. Dafür wird in Zeile 1 S geschrieben, in Zeile 2 der Suffix S_2 gefolgt von $S[1]$ und so weiter.
Anschließend werden die Zeilen lexikografisch sortiert. Die letzte Spalte L dieser Matrix ist die BWT. Das Problem dieser Konstruktion ist, dass die Laufzeit super-linear ist.

Definition 2.1.1. Sei L wie oben die letzte Spalte und F die erste. Die Funktion LF: $\{1, ..., n\} \rightarrow \{1, ..., n\}$ ist wie folgt definiert:
Ist $L[i] = c$ ist die k-te Wiederholung des Characters c, dann wird $LF(i)$ auf das j gesetzt, wo in Spalte F die k-te Wiederholung von c steht. Um die Stelligkeit des Buchstabens a priori zu wissen, und die Laufzeit linear zu halten, wird das C-Array erstellt. Das C-Array enthält an der Stelle des Buchstaben c die Anzahl aller Buchstaben in der F-Spalte, die kleiner sind, als c.

2.2 Datenkompression

Die BWT bietet sich für Datenkompression an, da direkt auf die Liste eine Lauflängen-Codierung angewandt werden kann. Als erstes müssen wir uns überlegen, wie die BWT zurück in den ursprünglichen String S transformiert werden kann. Das funktioniert mit der LF-Abbildung. Der String wird von hinten nach vorne konstruiert. Das letzte Zeichen (immer als \$ angenommen) wird also als erstes gesetzt. Der Algorithmus funktioniert wie folgt.

1. $S[n] \leftarrow \$$
2. $j \leftarrow 1$
3. for $i \leftarrow n - 1$ down to 1
4. $S[i] \leftarrow BWT[j]$
5. $j \leftarrow LF(i)$

Um die Korrektheit des Algorithmus zu zeigen, muss nur folgendes Lemma bewiesen werden.

Lemma 2.2.1. *Wenn Zeile i der sortierten Matrix M das Suffix j enthält, so enthält Zeile $LF[i]$ das Suffix S_{j-1}.*

Mit der Rückwärtskonstruktion kann nun die move-to-front-Codierung (MTF) eines Strings $L \in \Sigma^n$ eingeführt werden. Diese wird auf die BWT angewandt.

- initialisiere eine Liste ℓ der Zeichen aus Σ in aufsteigender Ordnung
- for $i \leftarrow 1$ to n
- $R[i] \leftarrow$ Anzahl der Zeichen vor dem Zeichen $L[i]$ in der Liste ℓ
- bewege Zeichen $L[i]$ an den Anfang der Liste ℓ

Auch die MTF kann wieder decodiert werden, in diesem Fall wieder zurück in die BWT, bzw. die Spalte L. Als Eingabe bekommen wir die Spalte R.

1. for $i \leftarrow 1$ to n
2. $L[i] \leftarrow \ell[R[i] + 1]$
3. bewege Zeichen $L[i]$ an den Anfang der Liste ℓ

2.3 Constant-time rank/select queries

Theorem 2.3.1. *Ein Bit-Vektor B kann in linearer Zeit so vorverarbeitet werden, dass die folgenden Abfragen in konstanter Zeit geschehen*

- $rank_b(B, i)$ *gibt die Anzahl der Vorkommen von b in $B[1, ..., i]$ zurück*
- $select_b(B, i)$ *gibt die Position des i-ten Vorkommens von b in $B[1, ..., n]$ zurück*

Für einen Text T über ein Alphabet Σ kann die rank und select Abfrage auf die Algorithmen für Bit-Arrays zurückgeführt werden.

2.4 Backward search

Wir führen eine Methode ein, um ein Muster effizient in einem String zu suchen. Das geschieht mit der sogenannten Rückwärts-Suche. Wenn wir einen String P der Länge m suchen, dann wird zunächst das $P[m]$-Intervall I bestimmt. Das funktioniert mithilfe des C-Arrays. Um anschließend innerhalb der Grenzen lb und rb nach $P[m-1, m]$ zu suchen, bestimmen wir das erste und letzte Vorkommen (p, q) von $P[m-1]$ in I. Das $P[m-1, m]$-Intervall liegt dann bei $LF(p)$ bis $LF(q)$. Diese Werte können in der Tat berechnet werden, ohne vorher explizit p und q bestimmen zu müssen. Falls p im Intervall liegt, so gilt $LF(p) = C[c] + rank_c(BWT, p) = C[c] + rank_c(BWT, lb - 1) + 1 := i$. Genauso gilt $LF(q) = C[c] + rank_c(BWT, rb) := j$.
Läge p nicht im Intervall, so würde $i > j$ gelten. Das wird in Folgendem Algorithmus

zusammengefasst:

1. $i \leftarrow 1$, $j \leftarrow n$, $k \leftarrow m$
2. while $i \leq j$ and $k \geq 1$
3. $c \leftarrow P[k]$
4. $i \leftarrow C[c] + rank_c(BWT, i-1) + 1$
5. $j \leftarrow C[c] + rank_c(BWT, j)$
6. $k \leftarrow k - 1$
7. endwhile
8. if $i \leq j$
9. return interval [i,j]
10. endif
11. return $\perp$

Dieser Suchalgorithmus hat Laufzeit $\mathcal{O}(m)$. Das ist die theoretische Untergrenze der Laufzeit.

2.5 Wavelet Bäume

Das Problem der Backward-search ist, dass eine vergleichsweise hohe Speicherplatz-Komplexität von $\mathcal{O}(\sigma \cdot n)$ benötigt wird. Wavelet-Bäume erlauben rank Abfragen in logarithmischer Zeit mit viel geringerem Speicherplatz. Ein Wavelet-Baum beginnt mit dem String S und dem sortierten Alphabet Σ. Es wird Σ in der Hälfte gespalten und ein Bitarray B erzeugt, das für jeden Buchstaben in S 0 speichert, wenn der Buchstaben in der linken Hälfte des gespaltenen Alphabet landet und 1 sonst. Alle mit 0 gekennzeichneten Buchstaben werden in der Reihenfolge von S in die nächste Stufe des Baums geschrieben und alle mit 1 gekennzeichneten in die rechte. Dieses Verfahren wird iteriert, bis nur noch ein Buchstabe in jeder Hälfte steht. Rank Abfragen funktionieren wie folgt.

$\underline{rank_c(BWT, i, [l, r])}$

1. if $l = r$, return i
2. else
3. $m = \lfloor \frac{l+r}{2} \rfloor$
4. if $c \leq \Sigma[m]$ then return $rank_c(BWT, rank_0(B^{[l,r]}, i), [l, m])$
5. else return $rank_c(BWT, rank_1(B^{[l,r]}, i), [m + 1, r])$
6. endif
7. endif

Die Speicherplatzkomplexität ist damit nur noch $\mathcal{O}(n \cdot \log \sigma)$. Die Zeit der Rückwärtssuche wächst damit aber auf $\mathcal{O}(m \cdot \log \sigma)$. Auf ähnliche Weise lassen sich Abfragen an die BWT (also zum Beispiel bestimme $BWT[i]$) in $\mathcal{O}(\log \sigma)$ beantworten, ohne die BWT selbst zu speichern. Select-Abfragen lassen sich auf Wavelet Bäumen ebenso in $\mathcal{O}(\log \sigma)$ umsetzen.
$\underline{rank_c(BWT, i) \text{ auf einem perfekten Wavelet Baum}}$

1. $l \leftarrow 1, r \leftarrow \sigma, j \leftarrow 1$
2. while $j \leq 2^h - 1$
3. $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$
4. if $c \leq \Sigma[m]$
5. $i \leftarrow rank_0(A[j], i), j \leftarrow 2j, r \leftarrow m$
6. else
7. $i \leftarrow rank_1(A[j], i), j \leftarrow 2j + 1, l \leftarrow m + 1$
8. endif
9. endwhile
10. return i

Dieser Algorithmus muss den gesamten Wavelet Baum absteigen und hat daher Laufzeit $\mathcal{O}(\log \sigma)$.
Um den Buchstaben der BWT an der Stelle i zu bestimmen, kann mit einem Wavelet Baum der folgende Algorithmus verwendet werden:
$\underline{BWT[i]}$

1. let $A[1, 2^h - 1]$ be the array of bit vectors
2. $j \leftarrow 1$
3. while $j \leq 2^h - 1$
4. if $A[j][i] = 0$
5. $i \leftarrow rank_0(A[j], i)$, $j \leftarrow 2j$
6. else
7. $i \leftarrow rank_1(A[j], i), j \leftarrow 2j + 1$
8. endif
9. endwhile
10. return $\Sigma[j - 2^h + 1]$

Es fehlt noch die Implementierung der select Abfrage auf Wavelet Bäumen.
$\underline{select_c(BWT, i)}$

1. $h \leftarrow \log \sigma, c \leftarrow \Sigma[k]$
2. $j \leftarrow 2^h - 1 + k$
3. while $j > 1$
4. if j is even
5. $b \leftarrow 0$
6. else
7. $b \leftarrow 1$
8. endif
9. $j \leftarrow \lfloor \frac{j}{2} \rfloor$
10. $i \leftarrow select_b(A[j], i)$
11. endwhile
12. return i

Definition 2.6.1. Der Wavelet Index eines Strings S besteht aus

- Dem Rückwärts-Index, der Rückwärtssuche auf Basis des Wavelet Baums erlaubt.
- Dem Vorwärts-Index, der Rückwärtssuche auf dem inversen String S^{rev} erlaubt. Hier benötigen wir den Wavelet Baum der BWT des umgekehrten Strings S^{rev}.

Wir benötigen für die Suche den folgenden Pseudo-Code.
$getBounds[i, j], c$

1. return $getBounds'([i, j], c, [1, \sigma], 0)$

wobei $getBounds'$ der folgende Pseudo-Code beschreibt
$getBounds'([i, j], c, [l, r], smaller)$

1. if $l = r$ return $(i, j, smaller)$
2. else
3. $(a_0, b_0) \leftarrow (rank_0(B^{[l,r]}, i - 1), rank_0(B^{[l,r]}, j))$
4. $(a_1, b_1) \leftarrow (i - 1 - a_0, j - b_0)$
5. $m = \lfloor \frac{l+r}{2} \rfloor$
6. if $c \leq \Sigma[m]$
7. return $getBounds'([a_0 + 1, b_0], c, [l, m], smaller)$
8. else
9. return $getBounds'([a_1 + 1, b_1], c, [m + 1, r], smaller + b_0 - a_0)$
10. endif
11. endif

Example 2.6.2. *Wir nehmen an, wir haben einen String S mit BWT und wissen, dass im Rückwärtsindex R das le-Intervall $[13, 15]$ und im Vorwärtsindex V das el-Intervall $[7, 9]$ gegeben ist. Nun wollen wir nach dem Pattern len suchen. Daher suchen wir in V nach nel und machen einen Rückwärtsschritt auf S^{rev}.*

2.7 Approximate String Matching

Im Folgenden werden Algorithmen zur approximativen String Suche eingeführt. Ziel dabei ist es, auch Matches zu finden, die sich kaum vom gesuchten Pattern unterscheiden. Die Ähnlichkeit zweier Strings kann anhand der Hamming-Distanz oder der Levenshtein Distanz gemessen werden.
Seien S und P Strings der Länge n bzw. m, wobei $m < n$. Sei $K < m$. Ein m-langer Substring $S[i, i+m]$ ist ein k-mismatch von P in S, wenn

$$hdist(P, S[i, i+m]) \leq k$$

wobei *hdist* die Hamming-Distanz sei. Wenn m und Σ klein sind, so kann dieses Problem mit einem naiven Ansatz gut gelöst werden. Wir definieren

$$\mathcal{P} = \{P : hdist(P, P') \leq k\}$$

als die Hamming-Kugel mit Radius k.
Für ein w-Intervall gibt $getIntervals([i, j])$ die Liste aller cw-Intervall zurück, wobei $c \in \Sigma$. Diese Funktion unterscheidet sich kaum von der $getBounds$-Funktion der Bidirektionalen Suche und wird daher nicht nochmal extra erklärt.
Wir beschäftigen uns damit, wie hoch die Laufzeit der Funktion ist. Da jeder Knoten im Wavelet-Baum im worst-case genau einmal besucht wird, haben wir eine worst-case-Laufzeit von $\mathcal{O}(\sigma)$. Sei nun aber q die Anzahl der Buchstaben c aus Σ, sodass cw in S vorkommt. Dann ist der Aufwand sogar beschränkt durch $O(q \log \sigma)$. Der Algorithmus für das k-mismatch-Problem ist dann wie folgt gegeben:
$k - mismatch(P, j, d, [lb, rb])$

1. if $d < 0$ return $\varnothing$
2. if $j = 0$ return $\{[lb, rb]\}$
3. $I \leftarrow \varnothing$, $list \leftarrow getIntervals([lb, rb])$
4. for $(c, [lb, rb])$ in list do
5. if $P[j] = c$ then $I \leftarrow I \cup k - msimatch(P, j-1, d, [lb, rb])$
6. else $I \leftarrow I \cup k - mismatch(P, j-1, d-1, [lb, rb])$
7. endfor
8. return I

Wenn man als Fehlermaß anstatt der Hamming Distanz die Levenshtein Distanz heranziehen möchte, so muss der Algorithmus an zwei Stellen erweitert werden.
In Zeile 4 wird

$$I \leftarrow I \cup k - differences(P, j-1, d-1, [lb, rb])$$

und in Zeile 5 wird

$$I \leftarrow I \cup k - differences(P, j-1, d, [lb, rb])$$

eingefügt. Zeile 4 entspricht dabei dem Fall, dass der Buchstabe $P[j]$ aus dem String an dieser Stelle gelöscht wurde und Zeile 5 dementsprechend das Einfügen eines Buchstabens c. Das Problem ist, das der Suchraum exponentiell größer ist, als mit der Hamming Distanz. Deswegen wird die Einschränkung an an d in Zeile 1 verändert auf eine untere Schranke. Im Folgenden werden wir uns Gedanken machen, wie diese zu wählen ist, damit kein approximativer Match verloren geht.

Definition 2.7.1. Für einen String S und ein Muster P gibt es eine eindeutige links-rechts-Zerlegung

$$P = w_1 x_1 w_2 c_2 ... w_k c_k w_{k+1}$$

sodass jedes w_i ein Substring von S ist, aber $w_i c_i$ nicht. Man definiert daraus

$$M_{lr}(P, S) = \{p_i | p_i = \sum_{j=1}^{i} |w_j c_j|\}$$

Man bezeichnet die c_i auch als markiere Buchstaben. $M_{lr}(P, S)$ ist dann die Menge der Positionen in P an denen die markierten Buchstaben vorkommen.

Lemma 2.7.2. *Ist*

$$|M_{lr}(P, s)| = k$$

dann gibt es keinen Substring von S der P mit weniger als k Fehlern matcht.

Definition 2.7.3. Wir definieren nun das M_{lr}-Array durch

$$M_{lr}[j] = |\{p_i \leq j | p_i \in M_{lr}(P, S)\}|$$

Corollary 2.7.4. *Ist $M_{lr}[j] = d$, dann gibt es keinen Substring von S der $P[1, j]$ mit weniger als d Fehlern matcht.*

2.8 Verwendung von bidirektionaler Suche

Das Vorgehen wird zunächst anhand der approximativen Suche mit nur einem Fehler erklärt. In diesem Fall sei $m = |P|$ und $s = \frac{m}{2}$. Dann ist dieser eine Fehler entweder links oder rechts von s. Ist er links, dann wird auf der linken Seite approximativ gesucht und rechts mit Rückwärtssuche exakt. Ist er rechts, wird auf der rechten Seite approximativ gesucht und auf der linken Seite wird Vorwärtssuche (dh Rückwärtssuche auf dem reversen String). In der Realität ist die Position des Fehlers nicht bekannt, weswegen beide Varianten getestet werden müssen.
Für eine allgemeine Zahl d an Fehlern, wird das Pattern in $d + 1$ Teile geteilt und das selber Verfahren verallgemeinert. Hierfür wird Bidirektionale Suche benötigt.

3 Weiterführendes

3.1 LCP-Tabelle

Wir wissen bereits, wie man die LCP-Tabelle berechnen kann, aber wir werden nun eine andere Version einführen, die die LCP-Tabelle anhand der BWT bestimmt.
LCP from BWT

1. $LCP[i] \leftarrow \perp, LCP[1] \leftarrow -1, LCP[n+1] \leftarrow -1, l \leftarrow 0$
2. initialize empty queue Q
3. for each $c \in \Sigma$
4. $enqueue(Q, [C[c] + 1, C[c + 1]])$
5. endfor
6. $size \leftarrow \sigma$
7. while $Q \neq \varnothing$
8. if $size = 0$
9. $l \leftarrow l + 1$, $size \leftarrow |Q|$
10. endif
11. $[lb, rb] \leftarrow dequeue(Q)$, $size \leftarrow size - 1$
12. if $LCP[rb + 1] = \perp$

13. $LCP[rb+1] \leftarrow l$
14. $list \leftarrow getInterval([lb, rb])$
15. foreach $[i, j] \in list$
16. $enqueue(Q, [i, j])$
17. endfor
18. endif
19. endwhile

Die worst-case Laufzeit des Algorithmus ist $O(n)$.