1 Suchen in Texten

1.1 Überblick

In Anwendungen, bei denen Texte verarbeitet werden, ist es oft wichtig, in einem langen Text effizient nach den Vorkommen eines kürzeren Such-Textes (genannt Muster) suchen zu können. Wir betrachten einen Algorithmus, der sehr gut geeignet ist, wenn man in demselben Text nacheinander nach vielen verschiedenen Mustern suchen will. In diesem Fall lohnt es sich, eine etwas aufwändige Vorverarbeitung des Textes zu machen, weil dadurch das anschließende Suchen nach Mustern beschleunigt werden kann.

Ein Wort w ist eine Folge von Zeichen aus einem festen Alphabet Σ (mit konstanter Alphabet-Größe $|\Sigma|$, z.B. ASCII-Zeichen mit $|\Sigma| = 256$). Mit $|w|$ bezeichnen wir die Länge (Anzahl Zeichen) des Wortes w und mit $w[i]$ (für $0 \leq i < |w|$) das i-te Zeichen von w. $w[i..j]$ bezeichnet das Teilwort von w, das mit dem i-ten Zeichen beginnt und mit dem j-ten Zeichen endet. Ein Teilwort der Form $w[i..|w|-1]$ heißt auch *Suffix* von w. Ein Suffix ist *echt* (engl. *proper*), wenn $i \neq 0$.

Der gegebene Text ist ein (langes) Wort, das wir mit X bezeichnen. Sei $|X| = n$. Wir nehmen an, dass der Text mit dem speziellen Zeichen '$' endet, das sonst weder im Text noch in den Suchmustern vorkommt. Man nennt '$' einen *Sentinel*. Diese Annahme stellt sicher, dass kein Suffix der Anfang (Präfix) eines anderen Suffix von X sein kann. (Ein Suffix, welches das echte Präfix eines anderen Suffixes ist, bezeichnen wir als *eingebettet* (engl. *nested*).) Mit X_i bezeichnen wir das Suffix $X[i..n-1]$ von X. Ein Suchmuster Y der Länge m kommt in X vor, wenn es ein i gibt mit $X[i..i+m-1] = Y$. Wir wollen in Zeit $O(n \log n)$ eine Datenstruktur aufbauen, die es uns erlaubt, in X nach einem beliebigen Muster Y der Länge m in Zeit $O(m \log n)$ zu suchen.

Eine dafür geeignete Datenstruktur sind *Suffix-Arrays*. Wir wollen hier zeigen, wie man Suffix-Arrays direkt und relativ effizient in linear-logarithmischer Zeit $(n \log n)$ konstruieren kann. Wir halten uns dabei an den Original-Artikel von Udi Manber und Gene Myers [1]. Die Sortier-Methode könnte man als Recursive-Forward-Bucket-Sort bezeichnen. Wir werden zu einem gegebenen Text der Länge n ein Integer-Array **Pos** der Länge n konstruieren, der die lexikographisch sortierten Indizes der Suffixe enthält.

Text	b	c	c	a	a	b	a	b	a	$
Index	0	1	2	3	4	5	6	7	8	9
Sort. Index	9	8	3	6	4	7	5	0	2	1
Suffix	$	a$	aaba$	aba$	abaa$	ba$	baa$	bccaaba$	caaba$	ccaaba$

Abbildung 1: Suffix Array für den Text "bccaababa$"

1.2 Suchen

Die Suche nach dem Muster Y vollzieht sich in zwei Schritten. Zuerst wird der linkeste Index L_Y in **Pos** gesucht, so dass Y ein Präfix vom **Pos**$[L_Y]$-ten Suffix ist, anschließend wird die entsprechende rechteste Stelle R_Y gesucht. Dabei verwenden wir binäre Suche.

Zur Suche des linkesten Index L_Y in **Pos** mit Y als Präfix vom **Pos**$[L_Y]$-ten Suffix beginnen wir mit den Indizes $L = 0$ und $R = n - 1$. Als Invariante wollen wir erhalten, dass das **Pos**$[L]$-te Suffix lexikographisch kleiner als Y ist und das **Pos**$[R]$-te Suffix lexikographisch größer oder gleich Y ist.

Falls $R - L = 1$ ist brechen wir ab. In jedem Schritt betrachten wir **Pos**$[M]$. Wenn das **Pos**$[M]$-te Suffix lexikographisch größer oder gleich Y ist, so setzen wir $R := M$, andernfalls $L := M$. Natürlich wird M entsprechend der binären Suche gewählt. Siehe Abb. 2.

Input: Text X, Suchmuster Y, Suffix-Array **Pos**.
Output: L_Y.
Algorithmus:

```
if Y ist lexikographisch kleiner als X_Pos[0] then
    return 0;
else if Y ist lexikographisch größer als oder gleich groß wie X_Pos[|X|-1] then
    return |X|;
else
    L := 0;
    R := |X| - 1;
    while R - L > 1 do
        M := ⌈(L + R)/2⌉;
        if Y ist lexikographisch kleiner als X_Pos[M] then
            R := M;
        else
            L := M;
    return R;
```

Abbildung 2: Binäre Suche nach dem linkesten Index L_Y in **Pos**, so dass Y ein Präfix des **Pos**$[L_Y]$-ten Suffix ist.

Nach Terminierung des Algorithmus ist die Anzahl der Vorkommen von Y in X gegeben durch $R_Y - L_Y + 1$. Die linken Endpunkte der Suffixe (ihre Startpositionen) sind gegeben durch $\mathbf{Pos}[L_Y], \mathbf{Pos}[L_Y + 1], \ldots, \mathbf{Pos}[R_Y]$.
Insgesamt können wir mit dem Suffix-Array **Pos** alle Vorkommen von Y in X in einer Laufzeit von $O(|Y| \log |X|)$ bestimmen.

1.3 Index-Erstellung: Sortieren der Suffixe

Das Sortieren der Suffixe erfolgt in $\log n$ Phasen. In der h-ten Phase (wir beginnen bei 0) werden die Suffixe so sortiert, dass die lexikographische Ordnung bezüglich der ersten 2^h Buchstaben richtig ist. Dazu nutzen wir die Informationen aus der vorherigen Phase und den Fakt, dass wir Suffixe sortieren.
Wir teilen die Suffixe in Bereiche (*Buckets*) auf, so dass alle Suffixe eines Buckets in den ersten 2^h Buchstaben gleich sind. Wir sortieren dann jeweils die Suffixe eines Buckets.
Angenommen wir haben zwei Suffixe X_j und X_k aus einem Bucket, die bei den ersten 2^h Buchstaben gleich sind, d.h. $X[j..j+2^h] = X[k..k+2^h]$. Wir müssen also ab dem 2^h-ten Buchstaben 2^h weitere Buchstaben vergleichen. Das Ergebnis ist das selbe, wie wenn wir von den Suffixen X_{j+2^h} und X_{k+2^h} die ersten 2^h Buchstaben vergleichen. Wir kennen allerdings schon die Relation zwischen diesen beiden Suffixen aus der vorherigen Phase und müssen nur in der Lage sein, die Position der Indizes $j+2^h$ und $k+2^h$ in **Pos** zu finden.
Da wir in jeder Phase nur $O(n)$ Aufwand treiben wollen, können wir die Elemente eines Buckets nicht einfach mit einem $n \log n$-Algorithmus sortieren. Statt dessen gehen wir so vor, dass wir das Suffix Array von vorne bis hinten in seiner groben Sortierung bucketweise durchlaufen und dabei die bestehende Sortierung von Suffixen auch in die jeweils anderen Buckets übertragen. D.h. wenn X_k ein Element des ersten Buckets ist, so muss X_{k-2^h} in seinem Bucket vor allen Elementen kommen, die zu Suffixen in höheren Buckets korrespondieren.

Je nachdem wie geschickt man die Buckets verwaltet, braucht man mehr oder weniger Platz dafür. Wir benutzen noch zusätzlich zwei boolsche Arrays **BucketStart** und **BucketStart2** und zwei ganzzahlige Arrays **BucketSize** und **Suf**.(alle der Länge n). Außerdem noch ein Array **Bucket** der Länge $|\Sigma|$, das die bis zu $|\Sigma|$ verschiedenen Buckets kennzeichnet.
BucketStart$[i]$ ist TRUE, wenn an der Stelle i ein neuer Bucket in **Pos** beginnt und sonst FALSE. **Suf** ist invers zu **Pos**, d.h. es gilt **Suf**[**Pos**$[i]] = i$. Die anderen beiden Arrays sind für temporäre Werte.

In einer initialen Phase werden die Suffixe mit einer Art Radix-Sort nach ihrem ersten Buchstaben sortiert. Das gibt uns die Arrays **Pos**, **Suf** und **BucketStart** in Zeit $O(n)$.
Nach der zweiten Schleife gilt für einen beliebigen Buchstaben $d \in \Sigma$, dass in $i :=$ **Bucket**$[d]$ das letzte Vorkommen von d in X steht (oder -1). In **Pos**$[i]$ steht ein Zeiger auf die nächst-letzte Position, wo d in X vorkommt (oder -1).
Diese Kette wird dann in der dritten Schleife abgelaufen und man merkt sich in **Suf**$[i]$ dann stattdessen die Position an der das i-te Suffix in der lexikographischen Ordnung bezüglich des ersten Buchstabens vorkommt (in Abb. 3 ist dies c und wird für jedes Suffix um eins inkrementiert). Abb. 3 zeigt die initiale Phase.

Dann beginnt das eigentliche Sortieren. Es gibt insgesamt $\lfloor \log n \rfloor$ Phasen, die jeweils $O(n)$ Zeit benötigen. Nehmen wir nun an, dass die Arrays **Pos**, **Suf** und **BucketStart** nach der Phase h die korrekten Werte beinhalten. Wir betrachten Phase $h+1$.
Die linken und rechten Grenzen des aktuell betrachteten Buckets werden mit l bzw. r gekennzeichnet.

Wir setzen zunächst **BucketSize**$[l]$ auf 0 für jede linke Bucketgrenze l. Außerdem setzen wir **Suf**$[i]$ auf das erste Element des Buckets, in dem sich das i-te Suffix befindet (im Gegensatz zur exakten Position des Suffixes innerhalb des Buckets). Danach durchlaufen wir das Array **Pos** Bucket für Bucket und betrachten jeweils das um 2^h verlängerte Suffix im bei $k :=$ **Suf**$[d]$ beginnenden Bucket mit $d :=$ **Pos**$[i] - 2^h$. Die ersten **BucketSize**$[k]$ Elemente sind schon belegt und das j-te Suffix wird an die nächste freie Stelle $k +$ **BucketSize**$[k]$ gesetzt und **BucketSize**$[k]$ wird inkrementiert. Für jedes Suffix merken wir uns in **BucketStart2**, ob es bewegt wurde.
Bevor wir dann mit dem nächsten Bucket weitermachen, schauen wir noch einmal über alle Suffixe im aktuellen Bucket und setzen **BucketStart2** so dass nur der Eintrag, der mit dem linkesten Suffix im neuen Bucket korrespondiert, auf TRUE gesetzt wird. Damit markiert dann **BucketStart2** die Grenzen der neuen Buckets.
Anschließend wird auch das Array **Pos** aktualisiert und wir setzen die neuen Bucket-Grenzen im Array **BucketStart** mit den Informationen in **BucketStart2**. Abb. 4 zeigt das iterative Sortieren.
Man kann so alle Buckets in linearer Zeit durchlaufen, insgesamt benötigt eine Phase also $O(n)$ Schritte.

Input: Text X der Länge n.
Output: Suffixsortierung **Suf** für den ersten Buchstaben,
Initialisierung von **Suf**, **Pos**,
BucketStart, **BucketStart2** und **BucketSize**.
Algorithmus:

```
Erstelle die Arrays Suf, Pos,
BucketStart, BucketStart2 und BucketSize;
//Radix Sort auf dem ersten Buchstaben
for each c ∈ Σ do
    Bucket[c] := −1;
for i = 0 . . . n − 1 do
    b := Bucket[X[i]];
    Bucket[X[i]] := i;
    Pos[i] := b;
c := 1;
for each d ∈ Σ do
    i := Bucket[d];
    while i ≠ −1 do
        j := Pos[i];
        Suf[i] := c;
        if i = Bucket[d] then
            BucketStart[c] := TRUE;
        else
            BucketStart[c] := FALSE;
        c + +;
        i := j;
//Arrays initialisieren
BucketStart[n] := TRUE;
for i = 0 . . . n − 1 do
    Pos[Suf[i]] := i;
```

Abbildung 3: Initiale Sortier-Phase

Input: Text X der Länge n, initialisierte Arrays **Suf**, **Pos**,
BucketStart, **BucketStart2** und **BucketSize**.
Output: Suffixsortierung in **Pos**.
Algorithmus:

```
for h = 0 ... ⌊log n⌋ do
    for each Bucket [l, r) do
        BucketSize[l] := 0;
        for i = l ... r − 1 do
            Suf[Pos[i]] := l;
    for each Bucket [l, r) do
        for i = l ... r − 1 do
            d := Pos[i] − 2^h;
            if d < 0 or d ≥ n then continue;
            k := Suf[d];
            Suf[d] := k + BucketSize[k];
            BucketSize[k] + +;
            BucketStart2[Suf[d]] := TRUE;
        for i = l ... r − 1 do
            d := Pos[i] − 2^h;
            if d < 0 OR d ≥ n OR NOT BucketStart2[Suf[d]] then
                continue;
            k := min{j : j > Suf[d]
                AND (BucketStart[j] OR NOT BucketStart2[j])};
            for j = Suf[d] + 1 ... k − 1 do
                BucketStart2[j] := FALSE;
    for i = 0 ... n − 1 do
        Pos[Suf[i]] := i;
        BucketStart[i] := BucketStart[i] OR BucketStart2[i];
```

Abbildung 4: Recursive-Bucket-Sort-Phase

Literatur

[1] Udi Manber and Gene Myers. Suffix arrays: A new method for on-line string searches. *SIAM J. COMPUT.*, 22(5):935–948, oct 1993.

[2] Mohamed Ibrahim Abouelhoda, Enno Ohlebusch, and Stefan Kurtz. The enhanced suffix array and its applications to genome analysis. In *Proceedings of the 2nd Workshop on Algorithms in Bioinformatics*, volume 2452 of LNCS, pages 449–463. Springer, 2002.