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 aufwendige 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 $\mathbf{suf}_i$ 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$	aababa$	aba$	ababa$	ba$	baba$	bccaababa$	caababa$	ccaababa$

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 i in **suffixAtPos** gesucht, so dass Y ein Präfix vom **suffixAtPos**$[i]$-ten Suffix ist, anschließend wird die entsprechende rechteste Stelle gesucht. Dabei verwenden wir binäre Suche.

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

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

Input: Text X, **suffixAtPos**, Muster Y.
Output: L_Y.
Algorithmus:

```
L := 0;
R := |X| − 1;
if X[suffixAtPos[L]] ist lex. größer als Y then
    return 0;
else if X[suffixAtPos[R]] ist lex. kleiner als Y then
    return |X|;
else
    while R − L > 1 do
        M := ⌈(L + R)/2⌉;
        if X[suffixAtPos[M]] ist lex. größer als Y then
            R := M;
        else
            L := M;
    return suffixAtPos[R];
```

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

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 $\mathbf{suf}_j$ und $\mathbf{suf}_k$ aus einem Bucket, die bei den ersten 2^h Buchstaben gleich sind. 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 $\mathbf{suf}_{j+2^h}$ und $\mathbf{suf}_{k+2^h}$ die ersten 2^h Buchstaben vergleichen. Wir kennen es schon aus der ersten Phase und wir müssen nur in der Lage sein, die Position der Indizes $j + 2^h$ und $k + 2^h$ in **suffixAtPos** zu finden. Dazu bauen wir am Ende jeder Phase ein zu **suffixAtPos** reverses Array **posOfSuffix** auf (dann ist $\mathbf{suffixAtPos}[\mathbf{posOfSuffix}[i]] = i$). 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 $\mathbf{suf}_k$ ein Element des ersten Buckets ist, so muss $\mathbf{suf}_{k-2^h}$ in seinem Bucket vor allen Elementen kommen, die zu Suffixen in höheren Buckets korrespondieren. Schematisch ist das ganze in Abb. 3 dargestellt.

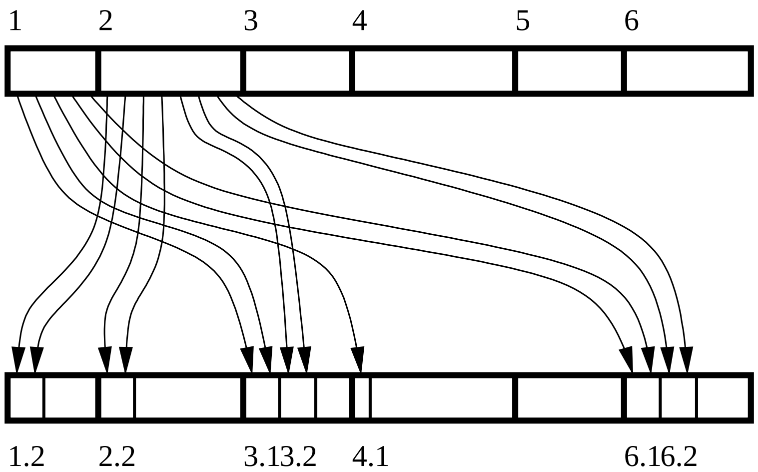

Abbildung 3: Schematische Darstellung der Forward-Sortierung beim Aufbau eines Suffix Arrays

Je nachdem wie geschickt man die Buckets verwaltet, braucht man mehr oder weniger Platz dafür. Wir benutzen noch zusätzlich zwei Arrays **bucketStart** und **bucketStart2** von Boolschen Werten und einen Array **bucketSize** (alle der Länge n). $\mathbf{bucketStart}[i]$ ist 'wahr', wenn an der Stelle i ein neuer Bucket in **suffixAtPos** beginnt und sonst 'falsch'. **bucketStart2** und **bucketSize** sind für temporäre Werte.

In einer initialen Phase werden die Suffixe mit einer Art Radix-Sort nach ihrem ersten Buchstaben sortiert. Nach der zweiten Schleife gilt für einen beliebigen Buchstaben d aus Σ, dass in $i := \mathbf{suffixAtPos}[d]$ das letzte Vorkommen von d in X steht (oder -1). In $\mathbf{posOfSuffix}[i]$ 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 $\mathbf{posOfSuffix}[i]$ dann stattdessen die Position an der das i-te Suffix in der lexikographischen Ordnung bezüglich des ersten Buchstabens vorkommt (dies ist c und c wird für jedes Suffix um eins inkrementiert). Abb. 4 zeigt die initiale Phase.

Dann beginnt das „rekursive“ Sortieren. Es gibt insgesamt $\lfloor \log n \rfloor$ Phasen, die jeweils $O(n)$ Zeit benötigen. Zum Iterieren über die Buckets kann man jeweils zwei Zeiger l und r benutzen. Man setzt l und r auf den Anfang des Buckets und inkrementiert r bis $\mathbf{bucketStart}[r] = \mathbf{true}$. Der nächste Bucket beginnt bei r.

Wir iterieren über die Buckets und setzen $\mathbf{posOfSuffix}[i]$ auf das erste Element des Buckets, in dem sich das i-te Suffix befindet. Außerdem setzen wir $\mathbf{bucketSize}[l] = 0$ (l ist die linke Bucket-Grenze). Danach durchlaufen wir das Array **suffixAtPos** Bucket für Bucket und betrachten jeweils das um 2^h verlängerte Suffix $d := \mathbf{suffixAtPos}[i] - 2^h$, das sich in dem Bucket befindet, der bei $k := \mathbf{posOfSuffix}[d]$ beginnt. Die ersten $\mathbf{bucketSize}[k]$ Elemente sind schon belegt und das j-te Suffix wird an Stelle $k + \mathbf{bucketSize}[k]$ gesetzt und $\mathbf{bucketSize}[k]$ wird inkrementiert. Für jedes Suffix merken wir uns in **bucketStart2**, ob es bewegt wurde.

Anschließend wird auch das Array **posOfSuffix** aktualisiert und wir setzen die neuen Bucket-Grenzen im Array **bucketStart** mit den Informationen in **bucketStart2**. Abb. 5 zeigt das iterative Sortieren

Man kann so alle Buckets in linearer Zeit durchlaufen, insgesamt benötigt eine Phase also $O(n)$ Schritte.

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.

Input: Text X der Länge n.
Output: Suffix Sortierung **suffixAtPos** für den ersten Buchstaben,
Initialisierung von **suffixAtPos**, **posOfSuffix**,
bucketStart, **bucketStart2** und **bucketSize**.
Algorithmus:

```
lege die Arrays suffixAtPos, posOfSuffix,
bucketStart, bucketStart2 und bucketSize an;
//initiale Phase: Radix Sort auf den ersten Buchstaben
for c from 1 to |Σ| do
    suffixAtPos[c] := −1;
for i from 0 to n − 1 do
    b := suffixAtPos[X[i]];
    suffixAtPos[X[i]] := i;
    posOfSuffix[i] := b;
c := 1;
for d from 1 to |Σ| do
    i := suffixAtPos[d];
    while i ≠ −1 do
        j := posOfSuffix[i];
        posOfSuffix[i] := c;
        if i = suffixAtPos[d] then
            bucketStart[c] := true;
        else
            bucketStart[c] := false;
        c + +;
        i := j;
//Arrays initialisieren
bucketStart[n] := true;
for i from 0 to n − 1 do
    suffixAtPos[posOfSuffix[i]] := i;
```

Abbildung 4: Initiale Sortier-Phase

Input: Text X der Länge n, Initialisierte Arrays **suffixAtPos**, **posOfSuffix**, **bucketStart**, **bucketStart2** und **bucketSize**.
Output: Suffix Sortierung in **suffixAtPos**.
Algorithmus:

```
for h from 0 to ⌊log n⌋ do
    for each Bucket [l, r) do
        bucketSize[l] := 0;
        for i from l to r − 1 do
            posOfSuffix[suffixAtPos[i]] := l;
    for each Bucket [l, r) do
        for i from l to r − 1 do
            d := suffixAtPos[i] − 2^h;
            if d < 0 or d ≥ n then continue;
            k := posOfSuffix[d];
            posOfSuffix[d] := k + bucketSize[k];
            bucketSize[k] + +;
            bucketStart2[posOfSuffix[d]] := true;
        for i from l to r − 1 do
            d := suffixAtPos[i] − 2^h;
            if d < 0 or d ≥ n or not bucketStart2[posOfSuffix[d]] then
                continue;
            k := min{j : j > posOfSuffix[d]
                and (bucketStart[j] or not bucketStart2[j])};
            for j from posOfSuffix[d] + 1 to k − 1 do
                bucketStart2[j] := false;
    for i from 0 to n − 1 do
        suffixAtPos[posOfSuffix[i]] := i;
        bucketStart[i] := bucketStart[i] or bucketStart2[i];
```

Abbildung 5: Recursive-Bucket-Sort-Phase