1 Suchen in Texten II

1.1 Überblick

Falls ein sehr großer Text X ($n := |X|$) für eine effiziente Suche indiziert werden muß (z.B. das menschliche Genom), so sind Suffixbäume manchmal zu groß, um komplett in den Speicher zu passen. In diesem Fall ist man bereit einen asymptotisch schlechteren Algorithmus zu benutzen, wenn der Index dafür kleiner ist. Eine dafür geeignete Datenstruktur sind **Suffix Arrays**.

Suffixbäume benötigen selbst bei einer sehr effizienten Implementierung im Schnitt zusätzlich zum Text ca. 12-20 Byte pro Zeichen (bei 4 Byte Zeigern). Im Gegensatz dazu kann man Suffix Arrays sehr sparsam implementieren. Die Basis ist ein einfaches Array von Suffix-Nummern, welches nach den Suffixen in lexikographischer Ordnung sortiert ist. Ein solches Array (der Größe $4n$ Bytes) kann mit Hilfe eines weiteren Indexarrays (der Größe $4n$ Bytes) in der Zeit $O(n \log n)$ sortiert werden [1]. Durch binäres Suchen kann man ein Muster Y damit in der Zeit $O(|Y| \log n)$ finden. Durch zwei weitere Indexarrays kann die Suchzeit auf $O(|Y| + \log n)$ gedrückt werden. Nimmt man noch mehr Informationen in den Index auf kann man mit einem weiteren Array die Zeit sogar derer von Suffixbäumen angleichen und in $O(|Y|)$ suchen. Das lohnt sich aber nur, wenn man durch geschicktes Optimieren keine vollen 4 Byte Integers für die Arrays benutzt. In diesem Bereich gibt es nicht nur für Suffix Arrays, sondern auch für Suffixbäume eine Fülle von Möglichkeiten. Man sollte im Hinterkopf behalten, dass hierbei immer nur das uniforme Kostenmodell zugrunde gelegt wird.

Natürlich kann man auch erst einen Suffixbaum aufbauen und daraus den Suffix Array erzeugen. Damit hat man aber bezüglich des Speicherplatzes nichts gewonnen. 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 der Länge n konstruieren, der die lexikographisch sortierten Indices der Suffixe enthält.

1.2 Suchen

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

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

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

```
Input: Text X, pos, Muster Y.
Output: L_Y.
Algorithmus:
    L := 0;
    R := |X| - 1;
    if X[pos[L]] ist lex. kleiner als Y then
        return 0;
    else if X[pos[R]] ist lex. kleiner als Y then
        return |X|;
    else
        while R - L > 1 do
            M := ⌈(L + R)/2⌉;
            if X[pos[M]] ist lex. kleiner als Y then
                R := M;
            else
                L := M;
        return R;
```

Abbildung 1: Binäre Suche nach dem linkesten Index i in **pos**, so dass Y ein Präfix des **pos**$[i]$-ten Suffix ist.

1.3 Indexerstellung: 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 dasselbe, 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 **pos** zu finden. Dazu bauen wir am Ende jeder Phase ein zu **pos** reverses Array **prm** auf (dann ist $\mathbf{pos}[\mathbf{prm}[i]] = i$). 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 in die 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. 2 dargestellt.

Jenachdem wie geschickt man die Buckets verwaltet, braucht man mehr oder weniger

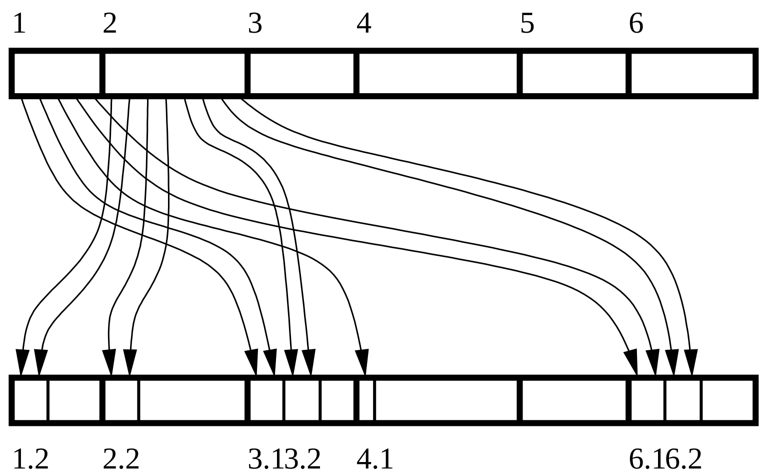

Abbildung 2: Schematische Darstellung der Forward-Sortierung beim Aufbau eines Suffix Arrays

Platz dafür. Wir benutzen noch zusätzlich zwei Arrays **BH** und **B2H** von Bool'schen Werten und einen temporären Array **count** (alle der Länge n). **B2H** und **count** sind für temporäre Werte. **BH**$[i]$ ist 'wahr', wenn an der Stelle i ein neuer Bucket in **pos** beginnt und sonst 'falsch'.

In einer initialen Phase werden die Suffixe mit einem Radix-Sort nach ihrem ersten Buchstaben sortiert. Nach der ersten Schleife gilt für einen beliebigen Buchstaben d aus Σ, dass in $i :=$ **pos**$[d]$ das letzte (von hinten) Vorkommen von d in X steht (oder -1). In **prm**$[i]$ steht ein Zeiger auf die nächst-letzte Position, wo d in X vorkommt (oder -1). Diese Kette wird dann in der zweiten Schleife abgelaufen und man merkt sich in **prm**$[i]$ dann stattdessen die Position an der das i-te Suffix in der lexikographischen Ordnung bezüglich des ersten Buchstabens vorkommt (dies ist c – c wird für jedes Suffix um eins inkrementiert).

Dann beginnt das „rekursive“ Sortieren. **prm**$[i]$ zeigt normalerweise auf die Position des i-ten Suffix in **pos**. Wir iterieren mittels **BH** über die Buckets und setzen **prm**$[i]$ auf das erste Element des Buckets, in dem sich das i-te Suffix befindet. Ausserdem setzen wir **count**$[l] = 0$ (l ist die linke Bucket-Grenze). Danach durchlaufen wir das Array **pos** Bucket für Bucket und betrachten jeweils das um 2^h verlängerte Suffix $d :=$ **pos**$[i] - 2^h$, welches sich im Bucket, der bei $k :=$ **prm**$[d]$ beginnt, befindet. Die ersten **count**$[k]$ Elemente sind schon belegt und das j-te Suffix wird an Stelle $k +$ **count**$[k]$ gesetzt und **count**$[k]$ wird inkrementiert. Für jedes Suffix merken wir uns in **B2H**, ob es bewegt wurde. Anschliessend setzen wir mit diesen Informationen in **B2H** die neuen Bucket-Grenzen. In einem letzten Schritt müssen wir **BH** und **pos** wieder aus **B2H** und **prm** richtig setzen. Insgesamt benötigt eine Phase also $O(n)$ Schritte.

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 **BH**$[r] =$ **true**. Der nächste Bucket beginnt bei r. Man kann so alle Buckets in linearer Zeit durchlaufen. Abb. 3 zeigt die initiale Phase, das iterative Sortieren die Abb. 4.

Input: Text X der Länge n.
Output: Suffix Sortierung **pos** für den ersten Buchstaben,
Initialisierung von **pos**, **prm**, **BH**, **B2H** und **count**.
Algorithmus:

```
lege die Arrays pos, prm, BH, B2H und count an;
//initiale Phase: Radix Sort auf den ersten Buchstaben
for c from 1 to |Σ| do
    pos[c] := −1;
for i from 0 to n − 1 do
    b := pos[X[i]];
    pos[X[i]] := i;
    prm[i] := b;
c := 1;
for d from 1 to |Σ| do
    i := pos[d];
    while i ≠ −1 do
        j := prm[i];
        prm[i] := c;
        if i = pos[d] then
            BH[c] := true;
        else
            BH[c] := false;
        c := c + 1;
        i := j;
//Arrays initialisieren
BH[n] := true;
for i from 0 to n − 1 do
    pos[prm[i]] := i;
```

Abbildung 3: Initiale Sortierphase (Radix Sort)

Input: Text X der Länge n, Initialisierte Arrays **pos**, **prm**, **BH**, **B2H** und **count**.
Output: Suffix Sortierung in **pos**.
Algorithmus:

```
for h from 0 to ⌊log n⌋ do
    for each Bucket [l, r) do
        count[l] := 0;
        for i from l to r − 1 do
            prm[pos[i]] := l;
    for each Bucket [l, r) do
        for i from l to r − 1 do
            d := pos[i] − 2^h;
            if d < 0 or d ≥ n then continue;
            k := prm[d];
            prm[d] := k + count[k];
            count[k] := count[k] + 1;
            B2H[prm[d]] := true;
        for i from l to r − 1 do
            d := pos[i] − 2^h;
            if d < 0 or
              d ≥ n or
              not B2H[prm[d]] then
                continue;
            k := min{j : j > prm[d] and (BH[j] oder nicht B2H[j])};
            for j from prm[d] + 1 to k − 1 do B2H[j] := false;
    for i from 0 to n − 1 do
        pos[prm[i]] := i;
    for i from 0 to n − 1 do
        if B2H[i] and not BH[i] then
            BH[i] := B2H[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.