1 Suchen in Texten

In Anwendungen, bei denen Texte verarbeitet werden, ist es oft wichtig, in einem langen Text effizient nach den Vorkommen eines kürzeren Suchtextes (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 Alphabetgröße $|\Sigma|$, z.B. ASCII-Zeichen mit $|\Sigma| = 256$). Mit $|w|$ bezeichnen wir die Länge (Anzahl Zeichen) des Wortes w und mit $w[i]$ (für $1 \leq i \leq |w|$) 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 $w[i..|w|]$ heißt auch *Suffix* von w. Ein Suffix ist echt (engl. „proper"), wenn $i \neq 1$.

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 Suffixe ist, bezeichnen wir als „eingebettet" (engl. „nested").) Mit $\mathbf{suf}_i$ bezeichnen wir das Suffix $X[i..n]$ 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 $O(n)$ eine Datenstruktur aufbauen, die es uns erlaubt, in X nach einem beliebigen Muster Y der Länge m in Zeit $O(m)$ zu suchen.

1.1 Suffixtries

Ein **Trie** ist ein Baum mit Wurzel, in dem jede Kante mit einem Zeichen beschriftet ist. Man interpretiert den Trie so, daß jedes Blatt einem Wort entspricht, das aus den Zeichen besteht, mit denen die Kanten auf dem Pfad von der Wurzel zu diesem Blatt beschriftet sind. Weiter verlangt man, daß bei jedem Knoten die Kanten zu seinen Kindern mit verschiedenen Zeichen beschriftet sind. Hier (Abb. 1) ist ein Beispiel für einen Trie, in dem die Wörter "aal", "auf" und "aus" gespeichert sind.

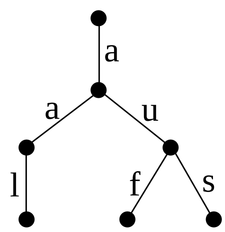
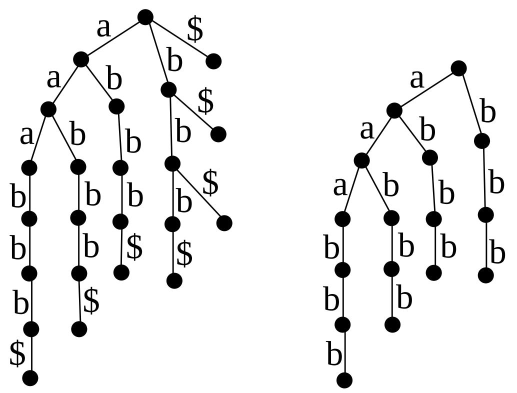


Abbildung 1: Ein Trie

In einem Trie kann man sehr effizient nach einem Muster Y suchen. Man beginnt an der Wurzel und prüft, ob es eine Kante mit Beschriftung $Y[1]$ gibt. Falls nein, so ist

Y nicht im Trie enthalten; falls ja, so sucht man rekursiv bei dem entsprechenden Kind nach $Y[2..|Y|]$ weiter. Falls alle Zeichen von Y bei dieser Suche erschöpft werden, so ist entweder Y im Trie enthalten (wenn die Suche bei einem Blatt geendet hat) oder Y ist ein Präfix eines Wortes, das im Trie gespeichert ist (wenn die Suche bei einem inneren Knoten geendet hat). Wenn man in jedem Knoten in konstanter Zeit die gesuchte Kante finden kann, benötigt die Suche nach Y insgesamt nur $O(|Y|)$ Zeit. Da wir konstante Alphabetgröße annehmen und jeder Knoten im Trie maximal $|\Sigma|$ Kinder hat, findet man die gesuchte Kante leicht in Zeit $O(1)$. Dazu kann man die Kanten zu den Kindern eines Knotens beispielsweise in einer linearen Liste, in einem Feld, in einer Hash-Tabelle, oder in einem balancierten Baum speichern; je nach den Anforderungen bzgl. Laufzeit und Speicherplatzbedarf kann die eine oder die andere Lösung besser sein.

Kommen wir zurück zu unserem ursprünglichen Problem: wir wollen zu einem Text X eine Datenstruktur erzeugen, mit deren Hilfe man effizient in X nach einem Suchmuster Y suchen kann. Es liegt nahe, einen Trie zu verwenden, in dem wir alle Suffixe $\mathbf{suf}_1$ bis $\mathbf{suf}_n$ des Textes speichern. Wenn das Suchmuster Y im Text an der Stelle i vorkommt, dann ist Y ein Präfix von $\mathbf{suf}_i$. Da $\mathbf{suf}_i$ im Trie gespeichert ist, kann man mit der oben beschriebenen effizienten Suchmethode feststellen, daß Y in X enthalten ist. Y kann das Präfix mehrerer Suffixe sein. Durch eine einfache Tiefensuche kann man in der Zeit $O(k)$ alle k Vorkommen von Y finden und ausgeben. Hier wird jetzt auch erkennbar, wozu der Sentinel '\$' dient. Durch den Sentinel ist kein Suffix Präfix eines anderen und somit ist jedes Suffix durch ein Blatt repräsentiert. Abb. 2 zeigt als Beispiel einen Trie, der alle Suffixe des Textes "aaabbb\$", und einen, der alle Suffixe des Textes "aaabbb" enthält.



(a) Mit Sentinel (b) Ohne Sentinel

Abbildung 2: Suffixtries für "aaabbb\$" und "aaabbb"

Man sieht, dass im zweiten Baum nicht alle Vorkommen von Y ='b' gefunden werden können, denn die beiden Suffixe $\mathbf{suf}_5$ ='bb' und $\mathbf{suf}_6$ ='b' sind eingebettet.

1.2 Suffixbäume

Es gibt aber noch ein Problem: der Trie, der alle Suffixe von X enthält, kann im schlimmsten Fall Speicherplatz quadratisch in n belegen, was bei langen Texten zu viel ist. Eine Familie von Wörtern, für die der Trie quadratisch groß ist, sind die Wörter '$\mathrm{a}^n\mathrm{b}^n\$$'.

Wir wollen eine Datenstruktur, die nur Platz $O(n)$ belegt. Deshalb modifizieren wir den Trie so, daß wir alle Knoten außer der Wurzel, die nur ein einziges Kind haben, löschen und dafür Kantenbeschriftungen zulassen, die aus beliebig vielen Zeichen bestehen. Damit ergibt sich aus dem obigen Trie (Abb. 2a) für "aaabbb$" folgender „pfadkomprimierter" Trie, der **Suffixbaum** genannt wird (Abb. 3).

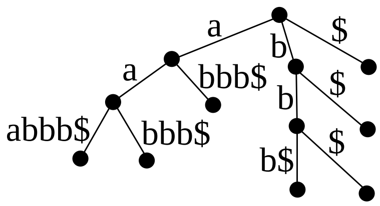


Abbildung 3: Ein Suffixbaum (engl. „Suffix Tree")

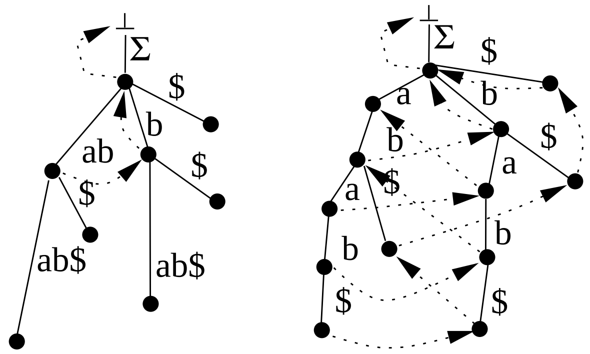
Da in einem Suffixbaum jeder interne Knoten (außer der Wurzel) mindestens zwei Kinder hat, kann man leicht zeigen, daß er insgesamt höchstens $2n$ Knoten haben kann (bei n Blättern). Die Beschriftungen der Kanten sind Teilwörter des Textes X. Wenn eine Kante die Beschriftung $X[i..j]$ hat, reicht es, lediglich die Indizes i und j bei der Kante zu speichern; damit benötigt man nur konstanten Speicherplatz für jede Kante, egal wie lang ihre Beschriftung ist. Folglich ist der Speicherplatz für den Suffixbaum $O(n)$, d.h. linear in der Textlänge.

Außerdem muß man die oben beschriebene Suchmethode für Tries nur leicht modifizieren, um genauso effizient in Suffixbäumen suchen zu können: bei jedem Knoten sucht man zuerst eine ausgehende Kante, so daß das erste Zeichen ihrer Beschriftung mit dem ersten Zeichen des zu suchenden Restwortes beginnt (falls keine solche Kante existiert, ist das Suchwort nicht im Suffixbaum enthalten). Dann vergleicht man die weiteren Zeichen des Restwortes mit den weiteren Zeichen der Kantenbeschriftung, bis entweder die Kantenbeschriftung erschöpft ist (dann sucht man rekursiv bei dem Knoten am anderen Ende der Kante weiter), das Suchwort erschöpft ist (dann ist das Suchwort im Suffixbaum enthalten) oder sich ein Zeichen des Restwortes von dem entsprechenden Zeichen der Kantenbeschriftung unterscheidet (dann ist das Suchwort nicht im Suffixbaum enthalten).

2 Ukkonen's Algorithmus

Nun betrachten wir einen Algorithmus, der zu einem gegebenen Text X der Länge n in Zeit $O(n)$ den Suffixbaum konstruiert. Dazu gibt es im wesentlichen drei bekannte Algorithmen von Weiner, McCreight und Ukkonen. Letzterer wurde zuletzt veröffentlicht, hat aber den Vorteil bestechend einfach zu sein und außerdem „on-line" zu arbeiten, d.h. man liest iterativ ein Zeichen des zugrunde liegenden Wortes und hat jedesmal den Suffixbaum für das schon gelesene Teilwort.

Zur effizienten Konstruktion müssen wir weitere Kanten im Suffixbaum einfügen, sogenannte „Suffixzeiger“ (engl. „suffix links“). Ein Suffixzeiger zeigt von einem Knoten, der ein Wort xY repräsentiert, zu dem Knoten, der das Wort Y repräsentiert. Beispiele sind die gestrichelten Pfeile in folgendem Suffixbaum und Suffixtrie für das Wort 'abab$' (im Suffixbaum werden keine Suffixzeiger für Blätter gebraucht). Der Einfachheit halber führen wir ausserdem einen zusätzlichen Knoten $\perp$ ein, der quasi den Fehlerzustand repräsentiert, wenn man die Bäume als Zustandsgraph einesAutomaten betrachtet.



(a) Suffixbaum (b) Suffixtrie

Abbildung 4: Ein Suffixbaum und ein Suffixtrie mit Suffixzeigern und $\perp$-Fehlerzustandsknoten

2.1 Suffixtries

Betrachten wir zunächst die **Konstruktion von Suffixtries**. Wir konstruieren online den Trie für ein immer größer werdendes Präfix von X. Sei T'_0 der leere Trie, T'_1 der Trie für $X[1..1]$ usw., dann konstruieren wir T'_{i+1} aus T'_i indem wir jedes Suffix in T'_i um $X[i+1]$ verlängern. Wir bezeichnen mit $\overline{\mathbf{suf}_j}$ den Knoten, der das j-te Suffix repräsentiert. Analog können wir mit $\overline{\text{aba}}$ einen Knoten bezeichnen, der das Wort „aba“ repräsentiert. Da für zwei aufeinanderfolgende Suffixe $\mathbf{suf}_j = X[i]\mathbf{suf}_{j+1}$ gilt, zeigt ein Suffixzeiger von $\overline{\mathbf{suf}_j}$ nach $\overline{\mathbf{suf}_{j+1}}$. Wenn wir uns das größte Suffix merken können wir bequem anhand der Suffixzeiger alle Suffixe ablaufen und verlängern. Dies erreichen wir mit dem Algorithmus von Abb. 5.

Die While-Schleife bricht ab, wenn entweder alle Suffixe abgelaufen worden sind (die Wurzel entspricht dem leeren Suffix und muß ebenso verlängert werden), oder wenn an einer Stelle kein neuer Knoten mehr erzeugt werden muß. In diesem Fall ist das verlängerte Suffix, o.B.d.A. $\mathbf{suf}_j$, schon als Präfix von einem anderen Suffix, o.B.d.A. $\mathbf{suf}_k$, vorhanden. Alle weiteren Suffixe $\mathbf{suf}_{j+r}$ $(r = 1 \ldots i-j+1)$ sind dann Präfix von $\mathbf{suf}_{k+r}$ und auch schon repräsentiert.

Input: Text X, Suffixtrie T'_i, ein Zeiger L auf das längste Suffix $\overline{\mathbf{suf}_1}$
Output: Suffixtrie T'_{i+1}, ein Zeiger L auf das längste Suffix $\mathbf{suf}_1$
Algorithmus:

```
l := L ;
p := ⊥ ; //Vorgänger
while l ≠ ⊥ do
    // Suffix l verlängern
    if es gibt keine Kante (l, v) mit Bezeichnung X[i + 1] then
        Erzeuge neuen Knoten v und Kante (l, v) mit Bezeichnung X[i + 1] ;
        if p ≠ ⊥ then
            suffixlink(p) := v;
        else
            L := v;
        fi
        p := v;
        l := suffixlink(l);
    else
        if p ≠ ⊥ then
            suffixlink(p) := v;
        fi
        l := ⊥;
    fi
od
```

Abbildung 5: Eine Iteration zum Aufbau eines Suffixtries

Der Algorithmus aus Abb. 5 wird mit dem leeren Baum das erste mal aufgerufen. Für jeden Buchstaben in X erfolgt ein Aufruf. Die While-Schleife wird pro Aufruf einmal betretet und jeweils einmal für jeden neu erzeugten Knoten iteriert. Daher ist die Laufzeit $O(\text{Größe von } T'_{|X|})$.

2.2 Suffixbäume

Wir gehen jetzt zur **Konstruktion von Suffixbäumen** über. Der Algorithmus funktioniert im Prinzip genauso, nur dass wir beim Verlängern der Suffixe drei Klassen einteilen und jetzt nur noch solche Knoten anlegen, die entweder Blätter sind, oder mehr als zwei Kinder haben. Alle Knoten, die nur ein Kind im Trie haben sind im Suffixbaum nicht mehr repräsentiert, sie sind „implizite“ Knoten. Um dennoch mit diesen impliziten Positionen arbeiten zu können benutzt man „Referenzpaare“, bestehend aus einem Knoten und einem Teilwort. Ein Referenzpaar ist quasi das Zwischenergebnis einer Suche nach einem Muster, wobei der Knoten den schon gefundenen Teil repräsentiert und das Teilwort den noch nicht gefundenen Teil. In Abbildung 4b ist das Muster „aba“ durch den

Knoten $\overline{\text{aba}}$ repräsentiert. In Abbildung 4a ist dieser Knoten nicht mehr vorhanden. Wir können das Muster „aba“ aber durch das Referenzpaar aus dem Knoten $\overline{\text{ab}}$ und dem Teilmuster „a“ eindeutig lokalisieren. Ebensogut könnten wir das Muster „aba“ auch durch das Referenzpaar aus der Wurzel und dem Teilmuster „aba“ eindeutig lokalisieren. Bei letzterem sind wir aber „weiter weg“ von unserem impliziten Knoten. Je näher dran wir sind, desto schneller können wir an der Stelle etwas machen. Wenn es für einen impliziten Knoten mehrere Referenzpaare gibt, so nennen wir das mit dem kürzesten Teilmuster das „kanonische“ Referenzpaar. Durch Ausführung des unten beschriebenen Suchalgorithmus (Abschnitt 3) kann man aus jedem Referenzpaar das kanonische Referenzpaar erzeugen. Wir benutzen nur Referenzpaare, die Teilworte unseres zugrunde liegenden Wortes (nämlich das sogenannte „aktive Suffix“) repräsentieren. Wir können ein Referenzpaar in konstant großem Platz speichern. Dazu speichern wir den Knoten, einen Index in das zugrunde liegende Wort X und die Länge des Teilmusters (bei obigem Beispiel also z.B. $(\overline{\text{ab}}, (3, 1))$).

In der i-ten Iteration teilen wir die Suffixe wie folgt in drei Klassen ein:

Klasse 1. Suffixe, die nicht Präfix eines anderen Suffixe sind (uneingebettete Suffixe).

Klasse 2. Suffixe, die in $X[1..i-1]$ Präfix eines anderen Suffixe sind, aber in $X[1..i]$ durch die Verlängerung mit $X[i]$ nicht mehr Präfix eines anderen Suffixe sind („relevante“ Suffixe).

Klasse 3. Suffixe, die in $X[1..i-1]$ und weiterhin in $X[1..i]$ Präfix eines anderen Suffixe sind.

Wie bei Suffixtries muss für die Suffixe der Klasse 3 nichts getan werden. Uneingebette Klasse 1 Suffixe können durch das erweitern des Wortes nicht eingebettet werden. Sobald also ein Suffix durch ein Blatt repräsentiert ist, bleibt dies auch in weiteren Iterationen so. Wir werden einen Trick verwenden: Um nicht jedes uneingebettete Suffix in jeder Iteration verlängern zu müssen, führen wird sogenannte „offene“ Kanten (engl. „Open Edge“) ein. Eine offene Kante führt zu einem Blatt und erstreckt sich immer bis zum Ende des zugrunde liegenden Wortes X. Bei der Kante merken wir uns nur den Anfangsindex und dass die Kante offen ist. Somit verlängern sich alle Blätter automatisch.

Damit ergibt sich ein Suffixlebenszyklus. In jeder Iteration wird sozusagen das leere Suffix als neues Suffix eingefügt. Das leere Suffix ist in jedes Suffix eingebettet und ist erstmal ein Suffix der Klasse 3. Irgendwann wird es so verlängert, dass es nicht mehr eingebettet ist. In dieser Iteration ist es Klasse 2. Danach ist es ein uneingebettetes Klasse 1 Suffix.

Letztlich müssen wir also nur für den Übergang von eingebettet zu uneingebettet etwas tun. Deshalb heißen Klasse 2 Suffixe „relevante“ Suffixe. Ist ein Suffix eingebettet, so sind alle kürzeren eingebettet, ist es nicht eingebettet, so sind auch alle längeren nicht eingebettet. Wir merken uns also das längste eingebettete Suffix (das „aktive Suffix“) mit einem Referenzpaar. In jeder Iteration überprüfen wir, ob das aktive Suffix auch nach der Verlängerung eingebettet ist. Wenn ja aktualisieren wir ggf. das Referenzpaar. Andernfalls fügen wir an der alten Position einen neuen Knoten ein und hängen an diesen ein Blatt für das nun nicht mehr eingebettete Suffix. Dann ersetzen wir den Knoten des Referenzpaars durch den Knoten auf den sein Suffixzeiger weist. Dadurch verkürzen wir unser Referenzpaar an der Vorderseite um einen Buchstaben und erhalten ein Referenzpaar für das nächst kürzere Suffix. Dieses machen wir kanonisch und fangen wieder mit dem Überprüfen an. Sobald das aktive Suffix kein relevantes Suffix mehr ist gehen wir zur nächsten Iteration. Abb. 6 zeigt eine Iteration dieses Algorithmus.

Input: Text X, Suffixbaum T_i, ein Referenzpaar $(n, (idx, len))$ für das aktive Suffix.
Output: Suffixbaum T_{i+1}, ein Referenzpaar $(n, (idx, len))$ für das neue aktive Suffix.
Algorithmus:

```
Verlängere den zugrunde liegenden String (die offenen Kanten);
v := testandsplit(n, (idx, len), X[i + 1]);
p := ⊥;
while v ≠ ⊥ do
    // Neues Blatt einfügen und zu nächstem aktiven Suffix
    w := neues Blatt;
    Verbinde v mit w durch eine offene Kante (i, ∞);
    if p ≠ ⊥ then
        suffixlink(p) := v;
    fi
    p := v;
    n := suffixlink(n);
    canonize(n, (idx, len));
    v := testandsplit(n, (idx, len), X[i + 1]);
od
if p ≠ ⊥ then
    // Neue Knoten wurden eingefügt
    suffixlink(p) := n;
fi
len := len + 1;
canonize(n, (idx, len));
```

Abbildung 6: Eine Iteration zum Aufbau eines Suffixbaums

Die Funktion $testandsplit(n, (idx, len), X[i + 1])$ prüft für das kanonische Referenzpaar $(n, (idx, len))$, ob es sich um $X[i + 1]$ verlängern läßt. Dazu muß lediglich der Buchstabe $X[i + 1]$ an $(len + 1)$-ter Stelle in der von n ausgehenden Kante liegen, die mit $X[idx]$ beginnt. Falls dies der Fall ist, gibt die Funktion $\perp$ zurück, andernfalls wird die Kante nach len aufgespalten und ein neuer Knoten eingefügt (falls dort noch kein Knoten ist — dann muss allerdings $len = 0$ sein) und zurückgeliefert.

Die Prozedur $canonize(n, (idx, len))$ kann effizienter als die Suche implementiert werden. Da das Referenzpaar auf ein eingebettetes Suffix weist, kommt der Text mit Sicherheit im Suffixbaum vor. Die Funktion prüft in einer While-Schleife daher nur, ob die

Kante, die mit $X[idx]$ beginnt, kürzer als len ist. Wenn ja, dann kann man n durch den Endknoten dieser Kante ersetzen, idx und die Kantenlänge erhöhen und len entsprechend verringern. Dies iteriert man solange, bis die Kante länger ist. Das Referenzpaar ist dann kanonisch. Siehe Abb. 7.

Input: Text X, Suffixbaum T_i, ein Referenzpaar $(n, (idx, len))$.
Output: kanonisches Referenzpaar $(n, (idx, len))$.
Algorithmus:

Sei $e := (n, v)$ die Kante, die als ersten Buchstaben $X[idx]$ hat;
$l := length(e)$;
while $len > 0$ und $l \leq len$ **do**
    $n := v$;
    $len := len - l$;
    $idx := idx + l$;
    Sei $e := (n, v)$ die Kante, die als ersten Buchstaben $X[idx]$ hat;
    $l := length(e)$;
od

Abbildung 7: Prozedur *canonize*

2.3 Komplexität

Lemma 1 (Amortisierte Komplexität). *Der Aufbau eines Suffixbaumes für einen Text der Länge n benötigt $O(n)$ viele Operationen.*

Beweis. Mit jeder Iteration wird ein weiterer Buchstabe angefügt. Es gibt daher n Iterationen. Außerhalb der While-Schleife wird pro Iteration einmal *canonize* aufgerufen. Wenn das Referenzpaar $(n, (idx, len))$ kanonisch ist, dann kostet der Aufruf der Funktion *testandsplit* nur konstant viel Zeit. Es werden nur konstant viele andere Operationen außerhalb der While-Schleife benötigt.

Für jede Iteration der While-Schleife wird ein neues Blatt eingefügt. Das es nur n Blätter gibt, wird die While-Schleife insgesamt n mal durchlaufen. Pro Durchlauf werden konstant viele Operationen ausgeführt und *canonize* einmal aufgerufen.

Insgesamt wird *canonize* höchstens $2n$ mal aufgerufen. Pro Aufruf werden außerhalb der While-Schleife konstant viele Operationen ausgeführt. Für jeden Durchlauf der While-Schleife wird len um mindestens 1 dekrementiert. len kann nicht negativ werden. Pro Iteration wird len um 1 inkrementiert, also insgesamt maximal um n. Daher kann die While-Schleife höchstens n mal durchlaufen werden.

Insgesamt werden daher nur $O(n)$ Operationen ausgeführt.

3 Suchen im Suffixbaum

Sei nun der Suffixbaum T für den Text X berechnet. Wenn ein Suchmuster Y in X gesucht werden soll, sucht man einfach ab der Wurzel von T Zeichen für Zeichen nach Y, bis die Suche entweder steckenbleibt (dann ist Y nicht in X enthalten) oder bis Y erschöpft ist (dann ist Y in X enthalten). Sei Y in X enthalten und sei y der Knoten in T, bei dem Y erschöpft ist. (Falls Y auf einer Kante erschöpft ist, wähle den *unteren* Endknoten der Kante als y.) Der Knoten y wird in Zeit $O(|Y|)$ gefunden. Falls y ein Blatt ist, dann kommt Y genau einmal in X vor (o.B.d.A. repräsentiere y das Suffix $\mathbf{suf}_i$, dann kommt Y an Stelle i vor). Falls y ein innerer Knoten von T ist, kommt Y mehrmals in X vor, und zwar an allen Positionen i, so daß $\mathbf{suf}_i$ durch ein Blatt im Unterbaum von y repräsentiert wird. Dies können durchaus deutlich mehr als $O(m)$ viele sein, weshalb eine Suche nach allen Vorkommen eines Patterns dann $O(m + k)$ kostet, falls k Vorkommen gefunden werden.

Um die Anzahl der Vorkommen von Y in X sowie die Position des ersten und letzten Vorkommens von Y in X in Zeit $O(|Y|)$ berechnen zu können, muß man den Suffixbaum um Zusatzinformationen erweitern, die an jedem internen Knoten die gewünschten Werte speichern. Diese Zusatzinformationen können z.B. direkt nach der Konstruktion des Suffixbaums mittels einer einfachen Tiefensuche in linearer Zeit $O(n)$ berechnet werden.

Literatur

[1] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM* 23(2):262–272, April 1976.

[2] E. Ukkonen. On-Line Construction of Suffix Trees. *Algorithmica*, 14:249–260, 1995.

[3] P. Weiner. Linear pattern matching. In *IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11. IEEE, 1973.