1 Suchen in Texten

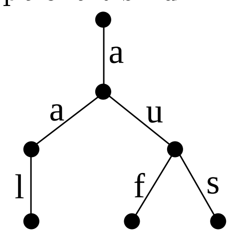
In Anwendungen, bei denen Texte verarbeitet werden, ist es oft wichtig, in einem langen Text effizient nach den Vorkommen eines kürzeren Suchtextes (genannt Muster) suchen zu können. Wir betrachten einen Algorithmus, der sehr gut geeignet ist, wenn man in demselben Text nacheinander nach vielen verschiedenen 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.

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. Diese Annahme stellt sicher, daß kein Suffix der Anfang (Präfix) eines anderen Suffix von X sein kann. 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 uns erlaubt, in X nach einem beliebigen Muster Y der Länge m in Zeit $O(m)$ zu suchen.

1.1 Tries

Ein *Trie* ist ein gewurzelter Baum, 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 ist ein Beispiel für einen Trie, in dem die Wörter "aal", "auf" und "aus" gespeichert sind.

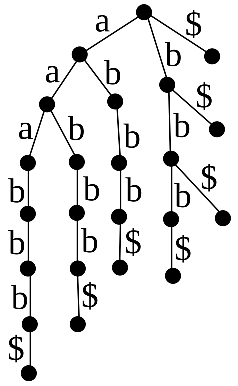


In einem Trie kann man sehr effizient nach einem Muster Y suchen. Man beginnt an der Wurzel und prüft, ob es eine Kante mit Beschriftung $Y[1]$ gibt. Falls nein, so ist Y nicht im Trie enthalten; falls ja, so sucht man rekursiv bei dem entsprechenden Kind nach $Y[2..|Y|]$ weiter. Falls alle Zeichen von 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

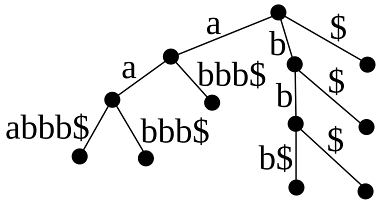
Knotens beispielsweise in einer linearen Liste, in einem Feld, in einer Hash-Tabelle, oder in einem balancierten Baum speichern; je nach den Anforderungen bzgl. Laufzeit und Speicherplatzbedarf kann die eine oder die andere Lösung besser sein.

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 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.

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. Hier als Beispiel ein Trie, der alle Suffixe des Textes "aaabbb\$" enthält.



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 für "aaabbb\$" folgender „pfadkomprimierter" Trie, der *Suffixbaum* genannt wird.

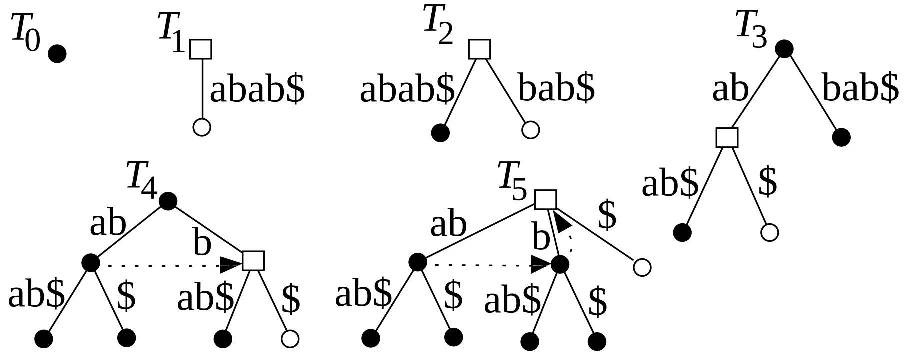


Da in einem Suffixbaum jeder interne Knoten (außer evtl. der Wurzel) mindestens zwei Kinder hat, kann man leicht zeigen, daß er insgesamt höchstens $2n$ Knoten haben kann (bei n Blättern). Die Beschriftungen der Kanten sind Teilwörter des Textes X. Wenn eine Kante die Beschriftung $X[i..j]$ hat, reicht 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 McCreight's Algorithmus

Nun betrachten wir einen Algorithmus, der zu einem gegebenen Text X der Länge n in Zeit $O(n)$ den Suffixbaum konstruiert. Der Algorithmus beginnt mit einem leeren Baum T_0 und fügt im i-ten Schritt, $1 \leq i \leq n$, den Suffix $\mathbf{suf}_i$ in den Baum T_{i-1} ein, um T_i zu erhalten. Der Baum T_n ist dann der gesuchte Suffixbaum. Hier als Beispiel die Bäume T_0 bis T_5, die der Algorithmus für den Text "abab\$" konstruiert (die Bedeutung der gepunkteten Pfeile wird später erklärt):



Führen wir einige Bezeichnungen ein. Für einen Knoten v in T_i bezeichnen wir mit $word(v)$ das Wort, das sich als Konkatenation der Kantenbeschriftungen auf dem Pfad von der Wurzel zu v ergibt. Ferner sei $\mathbf{leaf}_i$ das im i-ten Schritt (d.h. bei der Erzeugung von T_i) neu eingefügte Blatt und $\mathbf{head}_i$ der Vaterknoten von $\mathbf{leaf}_i$ in T_i. In der obigen Zeichnung sind die $\mathbf{leaf}_i$ als nicht ausgefüllte Knoten gezeichnet, die $\mathbf{head}_i$ als quadratische Knoten. Man beachte, daß der Knoten $\mathbf{head}_i$ manchmal in T_{i-1} noch gar nicht existiert; z.B. ist oben der Knoten $\mathbf{head}_3$ in T_2 noch nicht vorhanden und entsteht bei der Berechnung von T_3 erst durch Aufspalten der Kante mit Beschriftung "abab\$" in zwei Teilstücke mit Beschriftungen "ab" und "ab\$".

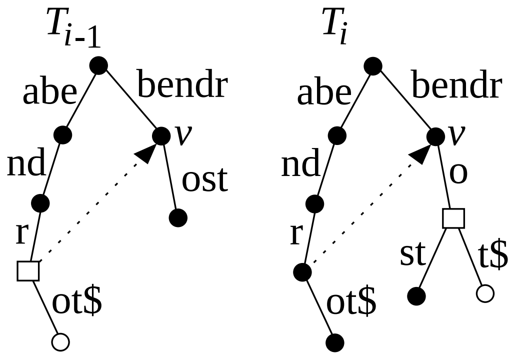
Um aus T_{i-1} den Baum T_i zu konstruieren, könnte man einfach folgendermaßen vorgehen. Man startet bei der Wurzel und läuft im Baum abwärts, als würde man nach $\mathbf{suf}_i$ suchen. Irgendwann bleibt diese Suche stecken, da $\mathbf{suf}_i$ ja noch nicht in T_{i-1} enthalten ist. Falls die Suche in einem Knoten v steckenbleibt (von dem es keine passende ausgehende Kante gibt), so ist v der gesuchte $\mathbf{head}_i$ und man erhält T_i, indem man an v ein neues Kind $\mathbf{leaf}_i$ anhängt und die Kante zu diesem Kind mit dem noch nicht abgearbeiteten Teil von $\mathbf{suf}_i$ beschriftet. Falls die Suche auf einer Kante steckenbleibt (d.h. der Anfang der Kantenbeschriftung paßte noch zu $\mathbf{suf}_i$, aber das nächste Zeichen ist verkehrt), so wird ein neuer Zwischenknoten auf der Kante vor dem ersten nicht passenden Zeichen eingefügt und dieser Knoten wie oben als $\mathbf{head}_i$ verwendet (d.h. an ihn wird das neue Kind angehängt und die Kante entsprechend beschriftet).

Dieser einfache Algorithmus ist zwar korrekt, würde aber für den Aufbau des Suffixbaums quadratische Laufzeit benötigen. Wir wollen aber mit $O(n)$ Zeit auskommen. Daher brauchen wir eine Methode, $\mathbf{head}_i$ schneller als durch eine Suche ab der Wurzel zu finden. Folgende Beobachtung hilft uns weiter:

> Wenn $\mathit{word}(\mathbf{head}_{i-1}) = x\alpha$ für ein $x \in \Sigma$ und ein $\alpha \in \Sigma^*$, so ist α ein Präfix von $\mathit{word}(\mathbf{head}_i)$.
>
> **Beweis:** Wenn $\mathit{word}(\mathbf{head}_{i-1}) = x\alpha$, war $x\alpha$ bereits in T_{i-2} enthalten. Es muß also ein $j < i-1$ gegeben haben, so daß $x\alpha$ ein Präfix von $\mathbf{suf}_j$ war. Dann ist α ein Präfix von $\mathbf{suf}_{j+1}$. Da $j+1 < i$, ist $\mathbf{suf}_{j+1}$ in T_{i-1} bereits enthalten; $\mathit{word}(\mathbf{head}_i)$ muß also mit α beginnen.

Um diese Beobachtung auszunutzen, führen wir sogenannte *Suffixzeiger* ein. Der Suffixzeiger eines Knotens v mit $\mathit{word}(v) = x\alpha$ soll dabei zu einem Knoten w mit $\mathit{word}(w) = \alpha$ zeigen. Wir werden erreichen, daß nach Erzeugung von T_i alle internen Knoten außer der Wurzel und möglicherweise $\mathbf{head}_i$ einen korrekten Suffixzeiger haben. Die gepunkteten Pfeile in der obigen Abbildung der Bäume T_0 bis T_5 stellen genau diese Suffixzeiger dar.

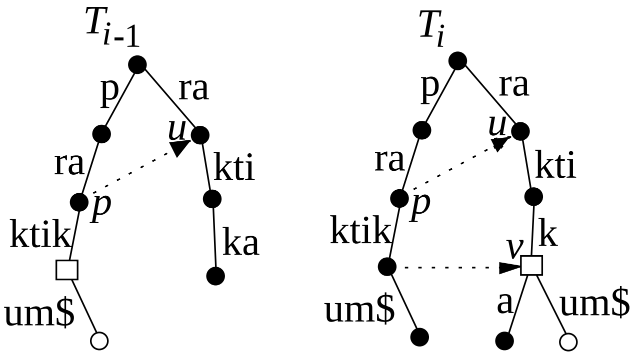
Was können wir nun mit diesen Suffixzeigern anfangen? Nehmen wir an, daß T_{i-1} berechnet ist und wir T_i erzeugen wollen. Wenn $\mathit{word}(\mathbf{head}_{i-1}) = x\alpha$ und der Suffixzeiger von $\mathbf{head}_{i-1}$ definiert ist, dann können wir mittels des Suffixzeigers zu einem Knoten v mit $\mathit{word}(v) = \alpha$ gelangen. Sei $\mathtt{suf}_i = \alpha\gamma$, d.h. γ ist genau die Beschriftung der Kante von $\mathbf{head}_{i-1}$ zu $\mathbf{leaf}_{i-1}$. Nun suchen wir ab Knoten v nach γ. Sobald die Suche steckenbleibt, haben wir $\mathbf{head}_i$ gefunden (falls die Suche auf einer Kante steckenbleibt, muß $\mathbf{head}_i$ als Zwischenknoten neu erzeugt werden) und können wie oben beschrieben $\mathbf{leaf}_i$ einfügen. Dadurch, daß wir uns die Suche nach α ab der Wurzel von T_{i-1} sparen, verbessert sich die Laufzeit erheblich. Die folgende Skizze verdeutlicht das Vorgehen (mit $x =$ "a", $\alpha =$ "bendr" und $\gamma =$ "ot\$"):



Über den Suffixzeiger gelangt der Algorithmus zu v und sucht von dort aus nach "ot\$". Die Suche bleibt auf der mit "ost" beschrifteten Kante zwischen "o" und "st" stecken; daher wird ein neuer Zwischenknoten eingefügt und als $\mathbf{head}_i$ verwendet.

Wenn zwar $\mathit{word}(\mathbf{head}_{i-1}) = x\delta$ für geeignete $x \in \Sigma$ und $\delta \in \Sigma^*$ gilt, aber der Suffixzeiger von $\mathbf{head}_{i-1}$ noch nicht berechnet ist, gehen wir folgendermaßen vor. Sei p der Vater von $\mathbf{head}_{i-1}$ und sei β die Beschriftung der Kante von p zu $\mathbf{head}_{i-1}$. Wir nehmen an, daß p nicht die Wurzel von T_{i-1} ist. Sei $\mathit{word}(p) = x\alpha$ und folglich $\mathit{word}(\mathbf{head}_{i-1}) = x\alpha\beta$.

Sei γ die Beschriftung der Kante von $\mathbf{head}_{i-1}$ zu $\mathbf{leaf}_{i-1}$. Über den Suffixzeiger von p gelangen wir zu einem Knoten u mit $word(u) = \alpha$. Von u aus verfolgen wir nun einen Pfad im Baum abwärts, dessen Kantebeschriftungen β entsprechen. Dabei müssen wir nicht jedes einzelne Zeichen auf jeder Kante betrachten, sondern nur jeweils das erste Zeichen der Kantenbeschriftung, weil wir schon sicher wissen, daß die ab u ausgeführte Suche nach β nicht steckenbleibt. Ist β erschöpft, wenn wir gerade bei einem Knoten v sind, so setzen wir den Suffixzeiger von $\mathbf{head}_{i-1}$ auf v und suchen ab v ganz normal (Zeichen für Zeichen) nach γ, um $\mathbf{head}_i$ zu finden und dort wie üblich $\mathbf{leaf}_i$ einzufügen. Wenn β auf einer Kante erschöpft ist, so erzeugen wir an der entsprechenden Stelle einen Zwischenknoten v, setzen den Suffixzeiger von $\mathbf{head}_{i-1}$ auf diesen Knoten v und verwenden v als $\mathbf{head}_i$, d.h. wir fügen an v eine mit γ beschriftete Kante zu einem neuen Blatt $\mathbf{leaf}_i$ ein. Auch hierzu wollen wir ein Beispiel betrachten (mit $x =$ p, $\alpha =$ "ra", $\beta =$ "ktik" und $\gamma =$ "um\$"):



Über den Suffixzeiger gelangt der Algorithmus von p zu u und sucht von dort aus zuerst nach "ktik". Da sicher ist, daß "ktik" im Baum enthalten ist, muß der Algorithmus nur das erste Zeichen der Kantenbeschriftung "kti" ansehen und kann dann sofort zum Knoten am anderen Ende der Kante gehen und von da aus den Rest "k" von "ktik" suchen. Da dieser Rest auf der Kante "ka" zwischen dem "k" und dem "a" erschöpft ist, wird ein neuer Zwischenknoten erzeugt und als $\mathbf{head}_i$ verwendet.

Zwei Fälle sind bisher noch nicht behandelt. Der erste Fall ist der, daß $\mathbf{head}_{i-1}$ gleich der Wurzel von T_{i-1} ist. In diesem Fall müssen wir ab der Wurzel Zeichen für Zeichen nach $\mathbf{suf}_i$ suchen, um $\mathbf{head}_i$ zu finden und $\mathbf{leaf}_i$ anhängen zu können. Der zweite Fall ist der, daß $\mathbf{head}_{i-1}$ ein Kind der Wurzel ist und noch keinen gültigen Suffixzeiger hat. Sei $word(\mathbf{head}_{i-1}) = x\beta$ für ein $x \in \Sigma$ und ein $\beta \in \Sigma^*$. Wir suchen dann ab der Wurzel mit der schnellen Methode (nur die ersten Zeichen auf jeder Kante betrachten) nach β. Sei β bei Knoten v erschöpft, wobei v gegebenenfalls ein neu erzeugter Zwischenknoten ist. Dann setzen wir den Suffixzeiger von $\mathbf{head}_{i-1}$ auf v. Falls v ein neu erzeugter Zwischenknoten ist, verwenden wir v als $\mathbf{head}_i$, ansonsten suchen wir ab v mit der langsamen Methode (Zeichen für Zeichen vergleichen) nach γ (der Beschriftung der Kante von $\mathbf{head}_{i-1}$ zu $\mathbf{leaf}_{i-1}$), bis die Suche bei einem (evtl. neu erzeugten) Knoten steckenbleibt, der dann $\mathbf{head}_i$ ist.

Der Algorithmus von McCreight zum Aufbau eines Suffixbaums in linearer Zeit ist in folgendem Pseudo-Code noch einmal zusammengefaßt:

```
Input: Text X
Output: Suffixbaum T
Algorithmus:
   T := Baum, der aus einer Wurzel r mit einem Kind w besteht,
         wobei die Kante von r nach w mit X beschriftet ist;
   head_1 := r; leaf_1 := v;
   for i = 2 to n do
         // Suffix suf_i in T einfügen
         γ := Label der Kante von head_{i-1} zu leaf_{i-1};
         if head_{i-1} = r then
               v := r;
               γ := γ[2..|γ|];
         else if Suffixzeiger von head_{i-1} ist definiert then
               v := Knoten, auf den der Suffixzeiger von head_{i-1} zeigt;
         else // head_{i-1} ist ungleich r und hat keinen Suffixzeiger
               p := Vaterknoten von head_{i-1};
               β := Label der Kante von p zu head_{i-1};
               if p = r then
                     β := β[2..|β|];
                     u := r;
               else
                     u := Knoten, auf den der Suffixzeiger von p zeigt;
               fi
               v := fastfind(u, β);
               Suffixzeiger von head_{i-1} auf v setzen;
         fi
         head_i := find(v, γ);
         leaf_i := insert(suf_i, head_i);
   done
```

In dem Algorithmus werden folgende Funktionen bzw. Prozeduren verwendet:

- *find*(v, γ) sucht ab Knoten v in T nach dem Wort γ, das nicht in T enthalten ist. Während der Suche wird zeichenweise verglichen. Wenn die Suche bei einem Knoten w steckenbleibt, weil es keine passende ausgehende Kante gibt, so wird w zurückgeliefert. Wenn die Suche auf einer Kante steckenbleibt, so wird an der entsprechenden Stelle auf der Kante ein neuer Zwischenknoten w (mit undefiniertem Suffixzeiger) erzeugt und zurückgeliefert. Der Zeitbedarf für *find*(v, γ) ist $O(|\gamma|)$.

- *fastfind*(u, β) sucht ab Knoten u in T nach dem Wort β, das bereits vollständig in T enthalten ist. Während der Suche wird immer nur das erste Zeichen jeder Kantenbeschriftung berücksichtigt. Ist β genau dann erschöpft, wenn die Suche einen Knoten w erreicht, so wird w zurückgeliefert. Ist β auf einer Kante erschöpft, so wird die Kante an der entsprechenden Stelle aufgeteilt und ein neuer Zwischenknoten (mit undefiniertem Suffixzeiger) eingefügt und zurückgeliefert. Der Zeitbedarf für *fastfind*(u, β) ist proportional zur Anzahl von Knoten auf dem Pfad von u bis zum zurückgelieferten Knoten.

- *insert*$(\mathbf{suf}_i, \mathbf{head}_i)$ erzeugt ein neues Blatt w von T und macht es zu einem Kind von $\mathbf{head}_i$. Die Beschriftung der Kante von $\mathbf{head}_i$ zu w wird so gewählt, daß $\mathit{word}(w) = \mathbf{suf}_i$ gilt. (Bei der Implementierung sollte man ausnützen, daß die Beschriftung genau der Restteil von γ ist, der beim zuvor ausgeführten *find*-Aufruf nicht mehr gefunden wurde.) Das neue Blatt w wird zurückgeliefert.

Der Zeitbedarf für das Einfügen von $\mathbf{suf}_i$ in T ist bei effizienter Implementierung abgesehen vom Zeitbedarf für die Aufrufe von *fastfind* und *find* konstant. Weiter kann man beweisen, daß der Gesamtaufwand für die Aufrufe von *fastfind* und *find* in allen Iterationen zusammen ebenfalls linear ist. Daher hat der Algorithmus von McCreight Laufzeit $O(n)$.

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 gleich $\mathbf{leaf}_i$ ist, dann kommt Y genau einmal in X vor, nämlich an Stelle i. Falls y ein innerer Knoten von T ist, kommt Y mehrmals in X vor, und zwar an allen Positionen i, so daß $\mathbf{leaf}_i$ ein Blatt im Unterbaum von y ist.

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 lediglich 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.