1 On-line Suchen in Texten

Diesmal betrachten wir ein kombinatorisches Problem von einer ganz anderen Art als die Graphenprobleme, die wie bisher gesehen haben. Gegeben ist ein Text (eng. “string”), den wir als eine Folge von Zeichen (eng. “characters”) aus einem festen Alphabet auffassen. Wir möchten gerne Fragen der folgenden Art beantworten können: Kommt ein bestimmtes Wort in dem Text vor? Wo kommt es gegebenenfalls vor? Wie oft kommt es vor? Wo kommt es zum ersten Mal vor? Das Suchwort nennen wir “Muster” (eng. “pattern”), und das Problem “Suchen in einem Text nach einem Muster(wort)” nennt sich deswegen “Pattern Matching”. Das Problem hat viele Varianten; hier betrachten wir nur eine, das sogenannte *On-line Pattern Matching Problem.* In dieser Variante erhält der Algorithmus zuerst den Text X, danach möchten wir gerne schnellstmöglich für verschiedene Muster Y Fragen wie die oben angegebenen beantworten können.

Sei ab jetzt X unser Text, bestehend aus einer Folge von Zeichen $x_1x_2\ldots x_n$ aus einem festen Alphabet Σ. Die Länge von Text X ist die Anzahl der Zeichen, geschrieben $|X| = n$. Das Muster Y ist ebenfalls eine Folge von Zeichen aus dem selben Alphabet mit Länge $|Y| = m$. Es ist klar, daß sich das Suchen von Y in X mit $O(mn)$ Operationen lösen läßt. Damit sind wir aber nicht zufrieden. Das *off-line* Problem, wo der Algorithmus zuerst Y und danach X erhält, läßt sich in Zeit $O(m+n)$ lösen (Algorithmen von Knuth-Morris-Pratt, Boyer-Moore). Mittels *Suffix-Bäumen* läßt sich das on-line Problem mit

- für das Preprocessing von Text X: $O(n)$ Operationen
- für jede Anfrage bezüglich einem Muster Y: $O(m)$ Operationen

lösen.

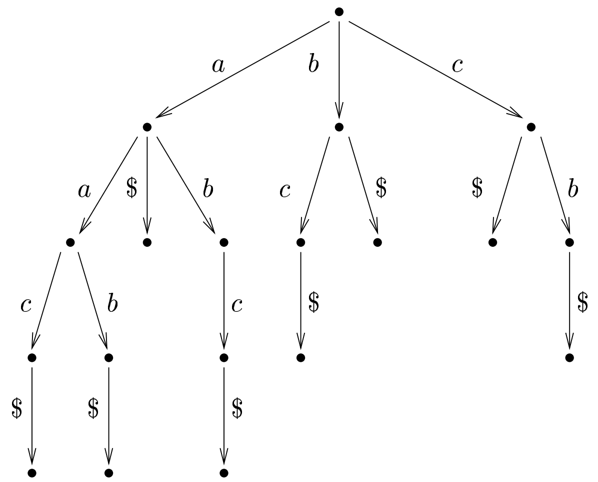
2 Tries und Suffix-Bäume

Sei ein Text X gegeben. Wir numerieren die Zeichen $X = x_1x_2\ldots x_n$ und bezeichnen das Zeichen auf Position i mit $X[i]$, so daß $x_i = X[i]$. Für ein Teilwort in X von Länge m, das in Position i anfängt, schreiben wir $X[i, i+m-1]$. Per Konvention ist $X[i,j]$ der *leere* Text, wenn $j < i$. Ein Muster Y kommt in X auf Position i vor, genau dann wenn $Y = X[i, i+m-1]$, d.h. $y_1y_2\ldots y_m = x_ix_{i+1}\ldots x_{i+m-1}$. Ein *Prefix* von einem Text $X = X[1,n]$ ist ein Wort $X[1,i], 1 \le i \le n$. Ein *Suffix* von X ist ein Wort $X[i,n], 1 \le i \le n$.

Sei jetzt unser (festes) Alphabet Σ der Größe $|\Sigma|$, und sei D eine Menge von Wörtern über Σ. Wir nehmen an, daß jedes Wort aus D mit einem speziellen Schlußzeichen $\$ \in \Sigma$ endet, das nur als Schlußzeichen vorkommt; damit ist klar, daß kein Wort aus D ein *echtes* Prefix von einem anderen Wort aus D ist. Ein *Trie* T (aus eng. “re*trie*val”, auch “digital search tree”) über D, Σ ist ein gewurzelter, gerichteter Baum, der es erlaubt, schnell zu entscheiden, ob ein Wort Y sich in der Menge D befindet. Jede Kante in T ist gelabelt mit einem Zeichen aus Σ. Jeder Knoten in T hat Verzweigungsgrad höchstens $|\Sigma|$. Alle Kanten, die von einem Knoten ausgehen, haben verschiedene Labels. Die Kanten auf einem Pfad von der Wurzel haben Orientierung weg von der Wurzel, und jedes

Wort aus D korrespondiert zu genau einem Pfad von der Wurzel zu einem Blatt: für jedes $Y \in D$ gibt es genau einen Pfad so, daß $Y[i]$ gleich dem Label auf der $i$ten Kante auf dem Pfad von der Wurzel zu dem Blatt ist.

Der Trie über $D = \{a\$, b\$, c\$, bc\$, aab\$, abc\$, aac\$, cb\$\}, \Sigma = \{a, b, c, \$\}$ ist unten abgebildet.



Suchen nach einem Wort Y in einem Trie T ist einfach. Nehme an, daß auch Y mit \$ endet. Man fängt an der Wurzel an und sucht die Kante, die mit $Y[1]$ gelabelt ist. Existiert eine solche Kante nicht, ist Y nicht in D. Sonst geht man zum Endknoten von der Kante und sucht rekursiv nach dem Wort $Y[2, m]$. Falls Y erschöpft ist, bevor ein Blatt erreicht wird, ist Y nicht in D. Wenn Y erschöpft ist und man auf einem Blatt in T steht, so ist Y in D. Der Zeitbedarf dieses Suchverfahrens ist proportional zu der Länge des Musters Y (unter der Annahme, daß das Alphabet fest, d.h. von konstanter Größe ist). Bei diesem Suchverfahren ist in dem $i$ten Schritt (rekursiven Aufruf) ein Prefix $Y[1, i]$ von Y gefunden. Wir werden oft den Knoten v_i in T, wo wir in diesem Schritt stehen, mit dem Prefix $Y[1, i]$ identifizieren, und schreiben $\texttt{word}(v_i) = Y[1, i]$.

Man könnte das on-line pattern matching Problem lösen, indem man einen Trie konstruiert, der alle (verschiedenen) Teilwörter $X[i, j], 1 \leq i, j \leq n$ von Text X enthält. Damit könnten wir zumindest Fragen wie "Ist Y in X?" in Zeit $O(m)$ beantworten. Nachteil ist natürlich, daß wir keine Chance haben, diesen Trie in linearer Zeit zu konstruieren. Stattdessen konstruieren wir einen Trie, der alle Suffixe von X enthält, also einen Trie über den Wörtern $\{X[1, n], X[2, n], X[3, n], \ldots, X[n, n]\}$ (Wir nehmen an daß $X[n] = \$$, so daß kein Suffix ein echtes Prefix von einem anderen Suffix ist). Unser Suchverfahren muß in diesem Fall etwas modifiziert werden: Y ist gefunden, wenn wir in dem obigen Verfahren Y erschöpft haben (Y soll in diesem Fall nicht mit \$ enden). In diesem Fall ist Y ein Prefix von einem Suffix von X, $Y = (X[i, n])[1, m]$. Da X nur n Suffixe hat, besteht hier die Chance, daß wir einen Trie von linearer Größe in linearer Zeit konstruieren können. Leider geht auch das nicht so einfach. Beispielsweise hat der Trie für $X = a^n b^n \$, n \geq 1$, Größe $\Omega(n^2)$.

Es ist relativ leicht zu erreichen, daß der Trie nur lineare Größe hat. Man kompaktiere alle Pfade von Knoten mit Ausgangsgrad 1 zu je einer Kante; diese Kante bekommt als

Label das Wort bestehend aus den Zeichen an den ursprünglichen Kanten. Sei z.B.

$$\cdot \xrightarrow{a} \cdot \xrightarrow{b} \cdot \xrightarrow{c} \cdot \xrightarrow{a} \ldots \cdot \xrightarrow{a} \cdot$$

ein solcher Pfad (nur Anfangs- und Endknoten haben Ausgangsgrad größer als 1), dann wird dieser Pfad durch eine Kante mit Label $abca \ldots a$ ersetzt. Jeder innere Knoten in dem kompaktierten Trie T' hat jetzt Ausgangsgrad mindestens 2, und deswegen hat T' höchstens $2n-1$ Knoten. Der kompaktierte Trie der Suffixe von X ist der *Suffix-Baum* über X.

Aus Effizienzgründen repräsentieren wir die Labels in T' nicht explizit, sondern mittels zwei Zeigern zu Anfangs- und Schlußposition des Labels in Text X (dies ist nicht eindeutig festgelegt, aber jede Wahl ist gültig). Mit $\texttt{label}(v, w)$ bezeichnen wir das Label von Kante (v, w), also Anfangs- und Schlußposition des Wortes, das (v, w) repräsentiert.

3 Konstruktion von Suffix-Bäumen in linearer Zeit

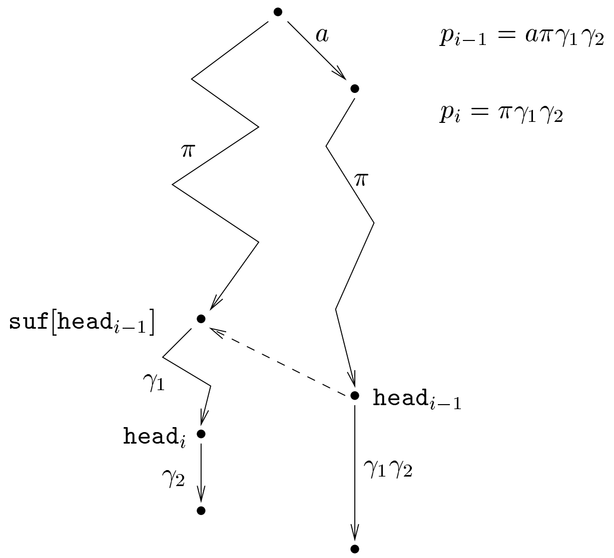
Wir beschreiben jetzt McCreight's Algorithmus [McC76] zur Konstruktion von Suffix-Bäumen in linearer Zeit. Mit $p_i = X[i, n]$ bezeichnen wir das $i$te Suffix von X; p_i hat Länge $n+1-i$. Konstruieren wir zuerst den Suffix-Baum T für den ersten Suffix p_1. Dieser Baum hat Wurzel r und nur eine Kante (r, u) mit $\texttt{label}(r, u) = (1, n)$. Wir betrachten die übrigen Suffixe in der Reihenfolge $p_2, p_3, \ldots p_n$ und fügen in jedem Schritt einen neuen Suffix in den aktuellen Trie T ein. Um p_i einzufügen suchen wir in T, bis wir einem Knoten v finden, so daß $\texttt{word}(v) = p_i[1, j-1]$ und v keine ausgehende Kante mit Anfangszeichen $p_i[j]$ hat. Diesen Knoten bezeichnen wir mit $\texttt{head}_i$; das ist der Knoten, wo der Rest von p_i einzufügen ist (es gilt $p_i = p_i[1, j-1]p_i[j, n+1-i]$). Einfügen des Rests von p_i als Kante aus $\texttt{head}_i$ bezeichnen wir mit $\texttt{insert}(p_i, \texttt{head}_i)$. Dieser Aufruf erzeugt ein Blatt $\texttt{leaf}_i$, das p_i repräsentiert, d.h. $\texttt{word}(\texttt{leaf}_i) = p_i$, und eine neue Kante $(\texttt{head}_i, \texttt{leaf}_i)$ mit $\texttt{label}(\texttt{head}_i, \texttt{leaf}_i) = (i+j-1, n)$. Der Knoten $\texttt{head}_i$ hat die Eigenschaft, daß $\texttt{word}(\texttt{head}_i)$ der längste Prefix von p_i ist, der gleichzeitig ein Prefix von einem früheren Suffix $p_j, j < i$ ist. Es läßt sich beweisen, daß $|\texttt{word}(\texttt{head}_i)| \geq |\texttt{word}(\texttt{head}_{i-1})|$.

Beim Suchen in T nach $\texttt{head}_i$ müssen wir aufpassen: T ist ja ein kompaktierter Trie, und weil die Kanten längere Wörter repräsentieren existiert der Knoten $\texttt{head}_i$ nicht immer explizit in T. Wenn die Suche entlang einer kompaktierten Kante nicht fortgesetzt werden kann, muß die Kante in zwei Kanten aufgeteilt und ein neuer Knoten erzeugt werden. Wir brauchen also ein Suchverfahren, das in dem kompaktierten Trie sucht und falls notwendig einen neuen Knoten $\texttt{head}_i$ erzeugt. Dies wird von der Funktion $\texttt{slowfind}(\alpha, v)$ getan. Die Funktion sucht von v aus eine Kante (v, w) mit Label β, so daß $\alpha[1] = \beta[1]$. Existiert eine solche Kante nicht, liefert die Funktion v zurück. Sonst wird entlang dieser Kante gesucht, solange $\alpha[j] = \beta[j]$. Wird ein $j \leq |\beta|$ erreicht, so daß $\alpha[j] \neq \beta[j]$ (oder α erschöpft ist), muß ein neuer Knoten w' erzeugt werden. Die Kante (v, w) wird ersetzt durch zwei neue Kanten (v, w') und (w', w) mit $\texttt{label}(v, w') = \beta[1, j-1]$ und $\texttt{label}(w', w) = \beta[j, |\beta|]$, und $\texttt{slowfind}(\alpha, v)$ liefert den neuen Knoten w' zurück. Falls $\alpha[j] = \beta[j]$ für alle $j \leq |\beta|$, so wird die Suche bei w fortgesetzt.

Leider braucht auch dieses Verfahren im schlimmsten Fall quadratische Zeit, weil das Suchen nach $\texttt{head}_i$ zu lange dauern kann. McCreight's Algorithmus beseitigt dieses Problem mittels *Suffixzeigern*, die es erlauben, $\texttt{head}_i$ schnell genug zu identifizieren, damit die gesamte Konstruktionszeit linear bleibt.

Wir definieren jetzt den *Suffixzeiger* für ein Wort $a\pi$ (Zeichen $a \in \Sigma$ gefolgt von einem Wort π) als $\texttt{suf}[a\pi] = \pi$, so daß $\texttt{suf}[a\pi]$ auf den Knoten, der π repräsentiert, zeigt. Der Algorithmus konstruiert, während er läuft, diejenigen Zeiger, die wir brauchen. Suffixzeiger haben zwei entscheidende Eigenschaften (die wir hier unbewiesen stehen lassen):

1. $\texttt{head}_i$ ist ein Nachkomme von $\texttt{suf}[\texttt{head}_{i-1}]$; siehe Bild.



2. Für alle v ist $\texttt{suf}[v]$ Nachkomme von $\texttt{suf}[\texttt{parent}(v)]$.

Diese Eigenschaften ermöglichen es, viel schneller als mit `slowfind()` im Baum zu suchen. Ist $\texttt{suf}[\texttt{head}_{i-1}]$ gefunden, setzen wir ab diesem Punkt `slowfind()` ein; und $\texttt{suf}[\texttt{head}_{i-1}]$ können wir schnell finden, wenn $\texttt{suf}[\texttt{parent}(\texttt{head}_{i-1})]$ bekannt ist. Nach 2. muß der Pfad von $\texttt{suf}[\texttt{parent}(\texttt{head}_{i-1})]$ nach $\texttt{suf}[\texttt{head}_{i-1}]$ völlig im Baum enthalten sein, und dies nützen wir aus, um schneller $\texttt{suf}[\texttt{head}_{i-1}]$ zu finden. Statt wie mit $\texttt{slowfind}(\alpha, v)$ jedes einzelne Zeichen in α mit den Zeichen auf dem Pfad $\beta_1\beta_2\cdots\beta_\ell$ zu vergleichen, brauchen wir nur den Pfad zu identifizieren, und dazu reicht es, das Anfangszeichen auf jeder Kante β_i mit α zu vergleichen. Die Funktion $\texttt{fastfind}(\alpha, v)$ soll dieses Pfadsuchen implementieren. Die Funktion sucht aus v eine Kante (v, w) mit Label β, so daß $\beta[1] = \alpha[1]$. Existiert eine solche Kante nicht, sind wir fertig und liefern v zurück. Ist $|\alpha| < |\beta|$, so muß die Kante (v, w) geteilt werden wie bei `slowfind()`, d.h. ein Knoten w' wird erzeugt und die Kante (v, w) ersetzt durch Kanten (v, w') und (w', w) mit $\texttt{label}(v, w') = \beta[1, |\alpha|]$ und $\texttt{label}(w', w) = \beta[|\alpha| + 1, |\beta|]$, und wir liefern wieder w' zurück. Sonst wissen wir, daß β ein Prefix von α ist, und müssen nichts weiter überprüfen und setzen die Suche bei w fort.

McCreight's Algorithmus erfüllt die Invariante, daß $\texttt{suf}[\texttt{head}_j]$ für $j < i - 1$ vor Beginn des $i$ten Schrittes berechnet ist, und sieht so aus:

1. Konstruiere einen Baum T, der nur eine Kante mit Label p_1 hat. $\texttt{head}_1$ ist die Wurzel von T, ebenfalls ist $\texttt{suf}[\texttt{head}_1]$ die Wurzel von T. `parent` von der Wurzel zeigt auf die Wurzel selbst.

2. Für alle Suffixe p_i in der Reihenfolge $i = 2, 3, \ldots, n$:

3. (a) $\beta = \texttt{label}(\texttt{parent}(\texttt{head}_{i-1}), \texttt{head}_{i-1})$
   (b) $\gamma = \texttt{label}(\texttt{head}_{i-1}, \texttt{leaf}_{i-1})$
   (c) Falls $\texttt{parent}(\texttt{head}_{i-1})$ gleich der Wurzel von T ist, entferne das erste Zeichen von dem Wort $\beta\gamma$
   (d) $u = \texttt{suf}[\texttt{parent}(\texttt{head}_{i-1})]$ /* ist per Invariante bereits berechnet */
   (e) $v = \texttt{fastfind}(\beta, u)$ /* um $\texttt{suf}[\texttt{head}_{i-1}]$ zu finden */
   (f) $\texttt{suf}[\texttt{head}_{i-1}] = v$ /* um Invariante zu erhalten */
   (g) Falls v ein neuer Knoten ist, dann $\texttt{head}_i = v$, sonst $\texttt{head}_i = \texttt{slowfind}(\gamma, v)$
   (h) $\texttt{insert}(p_i, \texttt{head}_i)$

Nach der $i$ten Iteration ist auch $\texttt{suf}[\texttt{head}_{i-1}]$ berechnet. In dem Fall, wo $\texttt{parent}(\texttt{head}_{i-1})$ und $\texttt{head}_{i-1}$ der gleiche Knoten ist, ist β das leere Wort und $\texttt{fastfind}(\beta, u)$ liefert den Knoten u zurück. Die Korrektheit des Algorithmus sollte klar sein.

Die Entfernung von dem ersten Zeichen von $\beta\gamma$ im Schritt (3c) ist so zu verstehen: Falls β das leere Wort ist, wird das erste Zeichen aus γ entfernt und die Suche nach $\texttt{head}_i$ passiert mittels $\texttt{slowfind}()$ ab der Wurzel. In diesem Fall wird $\texttt{suf}[\texttt{head}_{i-1}]$ zur Wurzel gesetzt. Falls β nicht das leere Wort ist, wird das erste Zeichen aus β entfernt, und die Suche nach v passiert mittels $\texttt{fastfind}()$ ab $\texttt{suf}[\texttt{parent}(\texttt{head}_{i-1})]$. Danach wird $\texttt{head}_i$ mittels $\texttt{slowfind}()$ gefunden. In diesem Fall wird $\texttt{suf}[\texttt{head}_{i-1}]$ zu dem von $\texttt{fastfind}()$ returnierten Knoten gesetzt. Bemerke, daß $\texttt{fastfind}(\beta, v)$, falls mit einem leeren Wort β aufgerufen, den Knoten v selber returnieren soll. Die Bedingung "$\texttt{parent}(\texttt{head}_{i-1})$ ist die Wurzel von T" ist übrigens erfüllt, wenn entweder $\texttt{head}_{i-1}$ die Wurzel von T oder ein Kind der Wurzel von T ist.

Falls $\texttt{suf}[\texttt{head}_{i-1}]$ vor Schritt (3e) schon berechnet ist, kann man den Aufruf von $\texttt{fastfind}()$ sparen. Dies mag den Algorithmus beschleunigen.

Suchen in einem nach dem obigen Algorithmus konstruierten Suffix-Baum T nach einem Muster Y kann mittels einer $\texttt{slowfind}()$-ähnlichen Funktion implementiert werden. Um Fragen nach dem ersten Vorkommen von Y, dem letzten Vorkommen von Y und der Anzahl von Vorkommen von Y beantworten zu können, muß der Suffix-Baum um zusätzliche Informationen erweitert werden. Das Blatt $\texttt{leaf}_i$ korrespondiert zum Suffix p_i, das in Position i in X zum ersten und letzten Mal (und damit genau einmal) vorkommt. Mittels einer Bottom-up-Berechnung berechnet man für jeden inneren Knoten v das erste und letzte Vorkommen des Wortes $\texttt{word}(v)$; diese sind ja die ersten und letzten Vorkommen der Wörter $\texttt{word}(w)$, die Kinderknoten w von v entsprechen. Ebenso kann für jeden inneren Knoten leicht die Anzahl der Vorkommen des Wortes $\texttt{word}(v)$ berechnet werden; das ist ja genau die Summe der Anzahlen der Vorkommen seiner Kinder. Mit diesen Informationen kann das Suchverfahren leicht erweitert werden, so daß wir die gewünschten Fragen in Zeit $O(m)$ beantworten können.

Bezeichnen wir die neue Suchfunktion zum Suchen in einem nach dem obigen Algorithmus zu einem Text X konstruierten Suffix-Baum T mit `find`(γ). Die Funktion fängt an der Wurzel von T an und sucht wie `slowfind`() zeichenweise entlang einem kompaktierten Pfad in T. Wird ein Punkt erreicht, wo die Suche nicht weitergehen kann, weil entweder ein Knoten w erreicht ist, wo keine ausgehende Kante mit dem nächsten Zeichen in γ übereinstimmt, oder das nächste Zeichen an einer kompaktierten Kante nicht mit dem nächsten Zeichen in γ übereinstimmt, ist γ nicht in X. Wird γ erschöpft, ist das Wort in X gefunden. Die Funktion `find`(γ) soll in diesem Fall einen Knoten in T returnieren, wo Erst-, Letzt-, und Anzahl der Vorkommen von γ berechnet sind. Terminiert die Suche in einem existierenden Knoten w von T, ist w dieser Knoten. Terminiert die Suche auf einer Kante (v, w), ist der Endknoten w der gesuchte Knoten, der returniert werden muß.

Literatur

[McC76] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM 23*, 2 (1976), 262–272.