

Universität Stuttgart - Institut für formale Methoden der Informatik

**Hauptseminar im Wintersemester 06/07
Inside Google - Algorithmen für Suchmaschinen**

Suffixbäume

Sergey Telejnikov

18. Dezember 2006

Betreuer: Stefan Schwoon

Zusammenfassung

In dieser Seminararbeit werden Suffixbäume vorgestellt. Suffixbäume finden oft in Bioinformatik und Information Retrieval Anwendung, sie haben gewünschte lineare Komplexität und sind deswegen für mehrere Probleme einsetzbar. Es wird auch Ukkonens Algorithmus zur Konstruktion von Suffixbäumen detailliert, mit allen Implementierungstricks erläutert. Der Vorteil von Suffixbäumen besteht darin, dass man nur einmal n Schritte für die Konstruktion eines Suffixbaumes für den Text der Länge n benötigt, und dann ist es möglich, nur proportional der Länge des Suchmusters m und unabhängig von der Textgröße in solchem Suffixbaum zu suchen. Andere Datenstrukturen erlauben dagegen nur die Komplexität $O(n + m)$.

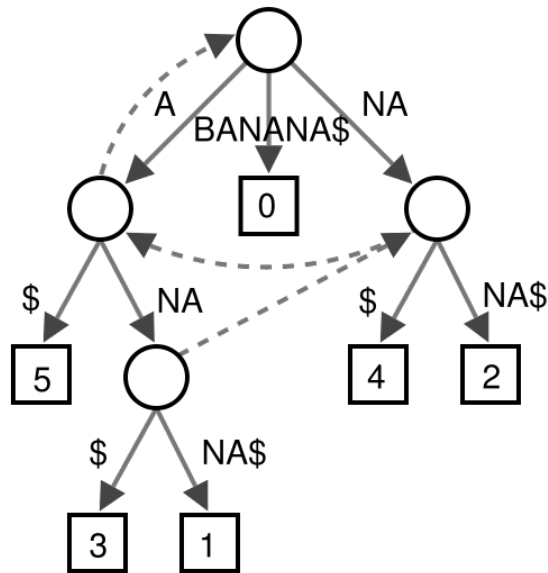


Abbildung 1: Suffixbaum für BANANA (Quelle[2])

Inhaltsverzeichnis

Abbildungsverzeichnis	3
1 Motivation	4
2 Historie	4
3 Grundlagen von Suffixbäumen	5
3.1 Definition	5
3.2 Suche in Suffixbäumen	6
4 Algorithmen für Konstruktion von Suffixbäumen	8
4.1 Naiver Ansatz	8
4.2 Ukkonens Algorithmus	9
4.2.1 Impliziter Suffixbaum	10
4.2.2 Algorithmus in $O(m^3)$	10
4.2.3 Implementierungstricks	11
4.3 Weiners und McCreights Algorithmen	15
5 Zusammenfassung und Fazit	16
Literatur	17

Abbildungsverzeichnis

1 Suffixbaum für BANANA (Quelle[2])	2
2 Suffixbaum für COCOS	6
3 Suffixbaum für COCO	6
4 Suffixbaum für COCO\$	7
5 Suche im Suffixbaum für COCOS	8
6 Naive Konstruktion vom Suffixbaum für COCOS	9
7 Impliziter Suffixbaum für COCO\$	10
8 Veranschaulichung von Ukkonens Algorithmus für COCO	12
9 Suffix-Link	12
10 Veranschaulichung von Tricks	16

1 Motivation

In dieser Arbeit wird eine interessante und sehr nützliche Datenstruktur vorgestellt, nämlich Suffixbaum. Es wird an Beispielen erläutert, wie man in Suffixbäumen effizient suchen kann und wie ein solcher Baum zu konstruieren ist. Dazu wird Ukkonens Algorithmus, der die Konstruktion von Suffixbäumen in linearer Zeit löst, schrittweise beschrieben.

Ein Suffixbaum ist eine Datenstruktur, die einen internen Aufbau eines Strings darstellt. Als String muss man sich hier nicht unbedingt einen gewöhnlichen String wie z.B. *abbaxa* vorstellen, sondern auch eine große Sequenz von Zeichen, also den Text.

Warum ist diese Datenstruktur so wichtig? Suffixbäume kann man benutzen um das *exact matching problem* (also exakte Textsuche) in linearer Zeit zu lösen. Und zwar linear zu der Länge des Suchmusters. Diese Struktur erlaubt auch die Lösung von weiteren komplexeren Stringverarbeitungsproblemen. Dieses Problem taucht überall auf, wo man suchen muss, nämlich in den Gebieten von Information Retrieval - Internet, datenbankbasierten Anwendungen, Texteditoren, Bibliothekssystemen, WWW-Verzeichnissen usw. Vor allem heutzutage, wo die Information exponentiell wächst, ist es wichtig, Information im Internet *schnell* zu finden. Suffixbäume werden in internen Algorithmen der Suchmaschinen verwendet. Als noch ein wichtiges Einsatzgebiet für Suffixbäume ist Bioinformatik zu nennen. Da DNA und RNA als String darstellbar sind, wird diese Datenstruktur in diesem Gebiet oft benutzt.

Der Vorteil von Suffixbäumen besteht darin, dass man für die Konstruktion eines Suffixbaumes für einen Text der Länge n nur einmal die Komplexität $O(n)$ braucht, und dann kann man unabhängig von der Textgröße im Text nach einem Suchmuster der Länge m in $O(m)$ suchen. Deswegen werden im Rahmen des Hauptseminars *Inside Google* Suffixbäume vorgestellt, die exakte Textsuche in $O(m)$ ermöglichen und selbst in $O(n)$ konstruiert werden können.

2 Historie

In diesem Abschnitt möchte ich einen kurzen Blick auf die Geschichte des Suffixbäumen werfen.

Ein *Suffixbaum* (engl. Suffix Trees oder auch PAT¹ Trees genannt) für einen String S ist eine Datenstruktur, die alle Teilstrings von S enthält. Suffixbäume sind sogenannte Indizes von Teilstrings eines Strings.

Ein erster Algorithmus zur Konstruktion von Suffixbäumen, der lineare Komplexität hat, wurde im Jahr 1973 von Weiner vorgestellt [5]. Suffixbäume wurden da aber Positionsbäume genannt. Ein paar Jahre später stellte McCreight einen anderen effizienteren (bezüglich Platzkomplexität) Algorithmus vor [6]. Einen konzeptuell anderen Algorithmus, der alle beträchtlichen Vorteile von McCreights Algorithmus hat, entwickelte im Jahr 1995 Ukkonen [4]. Dieser Algorithmus ist im Vergleich zu McCreights Algorithmus

¹PAT steht für Patricia Tree - *Practical Algorithms to Retrieve Information Coded in Alphanumeric* [3].

einfach zu erklären. Es ist zu sagen, dass Weiners Algorithmus damals nicht so viel Aufmerksamkeit erregt hat. Er war nicht populär, weil zwei originale Artikel von Weiner extrem schwer zu verstehen schienen. In dieser Arbeit versuche ich mit einfachen Worten und mit Hilfe von Beispielen zu erklären, wie Suffixbäume aufgebaut sind und wie man die mittels den oben genannten Algorithmen aufbaut.

3 Grundlagen von Suffixbäumen

In diesem Kapitel werden zuerst Suffixbäume formal definiert. Danach wird es betrachtet, wie man in einem Suffixbaum sucht.

Die klassische Anwendungsmöglichkeit von Suffixbäumen ist das *substring problem*:

Gegeben ist ein Text T der Länge n und ein String S der Länge m . Es sind alle Vorkommen von S in T zu finden oder zu sagen, dass S nicht in T enthalten ist. Dabei muss die Vorverarbeitung des Textes in $O(n)$ liegen - also proportional der Länge der Textes und die eigentliche Suche in $O(m)$ - proportional der Länge des Strings S und unabhängig von der Textgröße. Das alles kann man genau mit Hilfe von Suffixbäumen erreichen. In diesem Kapitel wird also gezeigt, dass Suche in Suffixbäumen in $O(m)$ liegt, und in dem nächsten Kapitel wird die Konstruktion von Suffixbäumen in $O(n)$ bewiesen.

3.1 Definition

Umgangsprachlich kann man einen Suffixbaum folgendermaßen definieren:

Ein Suffixbaum ist eine Datenstruktur, die dazu dient, schnelle Suche in großen Textmengen zu ermöglichen. Ein Suffixbaum ist, wie schon der Begriff aussagt, ein Baum von Suffixen eines Strings. Ein String der Länge n hat genau n nichtleere Suffixe. Und wir verwenden natürlich ein endliches Alphabet.

Definition: Ein Suffixbaum B für einen String S der Länge m ist ein gerichteter Baum mit genau m Blättern (nummeriert von 1 bis m) mit folgenden Eigenschaften:

- Jede Kante ist mit einem nichtleeren Teilstring von S beschriftet. (1)
- Jeder innere Knoten besitzt mindestens zwei Kinder, (2)
- deren Kantenbeschriftungen nie mit dem gleichen Symbol beginnen. (3)
- Für jedes Blatt i stimmt die Konkatenation von Kantenbeschriftungen auf dem Pfad von der Wurzel zu diesem Blatt genau mit dem Suffix mit der Anfangsposition i überein. (4)

Somit enthält B alle Suffixe von S . Die letzte Eigenschaft ist die wichtigste in der Definition.

Auf der Abbildung 2 ist ein Suffixbaum für einen String *cocos* zu sehen. Der Pfad von der Wurzel zum Blatt 3 ist genau der dritte Suffix *cos*. Die Frage „Existiert ein Suffixbaum für jeden String?“ ist aber mit „Nein“ zu beantworten. Das Problem tritt nur

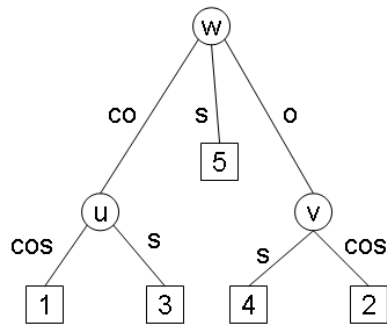


Abbildung 2: Suffixbaum für COCOS

dann auf, wenn ein Suffix ein Präfix eines anderen Suffixes ist. Zum Beispiel für einen String *coco*, wie im Abbildung 3 angezeigt ist, kann man keinen Suffixbaum konstruieren, weil der Suffix *co* ein Präfix des Suffixes *coco* ist.

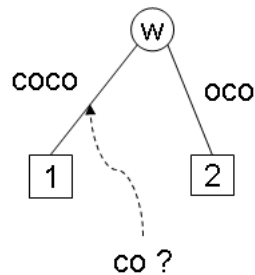


Abbildung 3: Suffixbaum für COCO

Dieses Problem ist aber leicht zu lösen: Man muss nur noch am Ende des Strings ein Terminierungssymbol hinzufügen, das nicht im Alphabet enthalten ist, z.B \$, sodass kein Suffix ein Präfix eines anderen Suffixes ist. Dann sieht der Suffixbaum für *coco* mit dem Terminierungssymbol so aus (Abbildung 4).

Für weitere Beschreibung brauchen wir eine Definition:

Definition: Die Pfadbeschriftung eines Knotens im Suffixbaum ist die Konkatenation aller Kantenbeschriftungen von der Wurzel bis zu diesem Knoten.

3.2 Suche in Suffixbäumen

Bevor wir uns mit der Konstruktion von Suffixbäumen beschäftigen, ist zuerst zu klären, wie man in einem Suffixbaum *B* für einen String *S* nach einem Teilstring *s'* sucht. Es ist

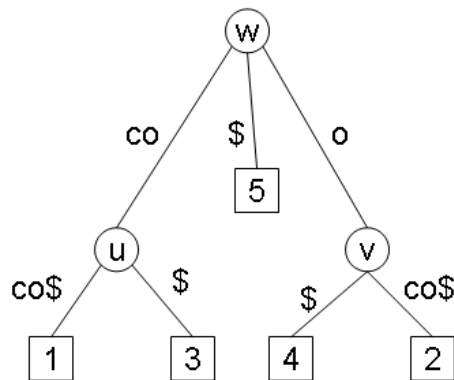


Abbildung 4: Suffixbaum für COCO\$

aber sehr einfach zu beschreiben und mittels eines Beispiels zu sehen:

Vergleiche alle Zeichen des Teilstrings mit den Zeichen im Suffixbaum von der Wurzel ausgehend, entlang des eindeutigen Pfades. Wenn der ganze Teilstring abgearbeitet ist, dann ist jedes Blatt des Teilbaumes unter dem Punkt des letzten Vergleichs mit einer Anfangsposition von s' in S nummeriert. Wenn es aber keine Übereinstimmung beim Vergleich gibt, dann ist s' nicht in S enthalten. Die Eindeutigkeit des Pfades wird wegen der Eigenschaft (3) gewährleistet. Formal kann man obige Beschreibung so im Pseudo-Code erfassen²:

Algorithmus der Suche in einem Suffixbaum:

```

Position := Wurzel;
for i in 1..m do
  if (Position = Kante) then
    if Zeichen auf dem Pfad im Baum = s'(i) then
      laufe weiter;
    else
      Put(s' wurde in S nicht gefunden);
    fi;
  else
    if(existiert eine Kante
       mit der Kantenbeschriftung l mit l(1) = s'(i) ) then
      Position := Diese Kante;
    else
      Put(s' wurde in S nicht gefunden);
    fi;
  fi;
fi;

```

² S habe die Länge n und s' die Länge m

od;

Der Algorithmus hat offensichtlich Zeitkomplexität $O(m)$, also proportional zur Länge der Eingabe.

Als Beispiel betrachten wir die Suche im String *cocos* nach Teilstring *co* (Abbildung 5). Beim Vergleich sind wir im Knoten *u* gelandet, und das heißt, dass alle Blätter im Teilbaum von *u* die Anfangspositionen vom gesuchten Teilstring sind. In unserem Fall sind das 1 und 3.

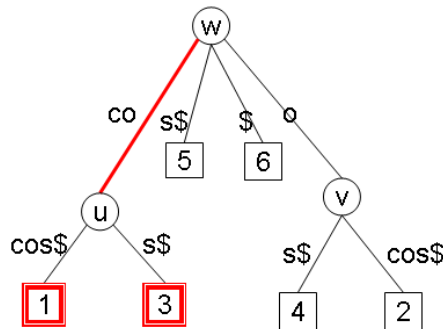


Abbildung 5: Suche im Suffixbaum für COCOS

4 Algorithmen für Konstruktion von Suffixbäumen

In diesem Kapitel wird zuerst ein naiver Ansatz von Suffixbaumkonstruktion vorgestellt. Demnächst wird Ukkonens Algorithmus mit allen Implementierungstricks detailliert erläutert.

4.1 Naiver Ansatz

Nachdem wir gesehen haben, wie einfach man in Suffixbäumen suchen kann, schauen wir jetzt, wie man solche Bäume naiv konstruiert. Die Aufgabe lautet also: „Konstruiere einen Suffixbaum für einen String S der Länge m “.

Ein naives Verfahren besagt Folgendes: Es werden nacheinander alle Suffixe $S[i \dots m]$ in einen anfangs leeren Baum hinzugefügt. Sei N_i ein Baum, der schon alle Suffixe von $S[1 \dots m]$ bis $S[i \dots m]$ enthält. Erstens wird der Baum N_1 , also für den gesamten String konstruiert. Dann wird der Baum N_{i+1} aus N_i beim Hinzufügen von Suffix $S[i + 1 \dots m]$ entstehen. Die obige Beschreibung kann man ein wenig detailliert wiederum im Pseudo-Code umsetzen:

Naiver Algorithmus für Konstruktion eines Suffixbaumes:

```

Konstruiere die Kante (Wurzel,1) mit der Kantenbeschriftung S[1..m]\$;
for i in 2..m do
  finde der längste Pfad von der Wurzel ausgehend entlang
    des eindeutigen Pfades, der mit dem Suffix S[i..m]\$ übereinstimmt;
  if (Pfad endet an dem Knoten u) then
    füge eine neue Kante (u,i) mit der Kantenbeschriftung S[i..m]\$ hinzu,
      wenn S[1..i] - Pfadbeschriftung von u ist;
  elseif (Pfad endet in der Mitte der Kante (w,u)) then
    füge an dieser Stelle einen anderen v Knoten hinzu;
    füge eine neue Kante (v,i) mit der Kantenbeschriftung S[i..m]\$ hinzu,
      wenn S[1..i] - Pfadbeschriftung von u ist;
  fi;
od;

```

Man kann aus dem Algorithmus sofort sagen, dass er quadratischen Aufwand hat. Der Verlauf der Konstruktion ist mit Hilfe von Abbildung 6 leicht nachvollziehen.

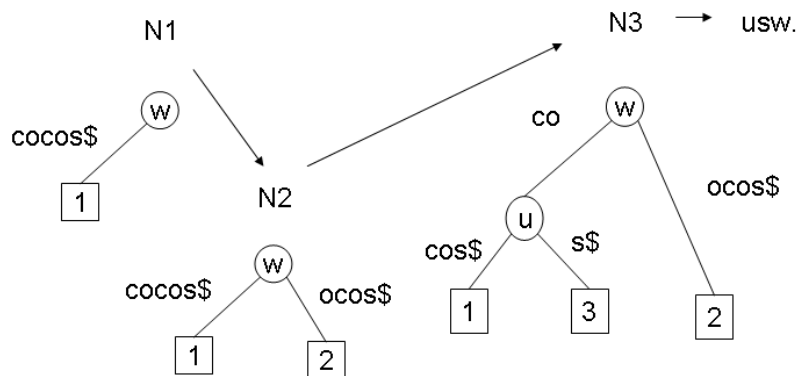


Abbildung 6: Naive Konstruktion vom Suffixbaum für COCOS

4.2 Ukkonens Algorithmus

Esko Ukkonen hat im Jahr 1995 einen interessanten Algorithmus zur Konstruktion von Suffixbäumen erfunden. Ukkonens Algorithmus ist ein *Online-Algorithmus*, d.h. für die Konstruktion im bestimmten Schritt braucht man nicht den gesamten Text zu betrachten, sondern nur die Zeichen, an denen gearbeitet wird.³

Einfachheit des Algorithmus ist dadurch erreicht, dass am Anfang ein einfacher und ineffizienter Algorithmus entworfen wird, der danach mit Hilfe von Implementierungstricks bezüglich der Komplexität verbessert wird.

³Ein Online-Algorithmus ist ein Lösungsverfahren für Probleme, bei denen zu Beginn des Berechnungsvorgangs nicht alle Eingabedaten verfügbar oder benötigt sind. *nach Quelle[2]*

4.2.1 Impliziter Suffixbaum

Für den Algorithmus benötigen wir eine vereinfachende Variante von Suffixbäumen:

Definition: Ein impliziter Suffixbaum ist ein Suffixbaum, in dem alle Terminierungssymbole \$ und folglich alle leeren Knoten und Knoten mit nur einer ausgehenden Kante entfernt wurden.

Anmerkung: Ein Suffixbaum für einen String, für den kein Suffix ein Präfix für einen anderen Suffix ist, ist gleich ein impliziter Suffixbaum (vergleiche Abbildung 2 und 5).

Für einen Suffixbaum für *coco*\$ ist ein impliziter Suffixbaum auf der Abbildung 7 zu sehen. Wie man sieht, sind implizite Suffixbäume weniger informativ als echte Suffixbäume. Sie werden nur als Hilfsmittel für Ukkonens Algorithmus benutzt.

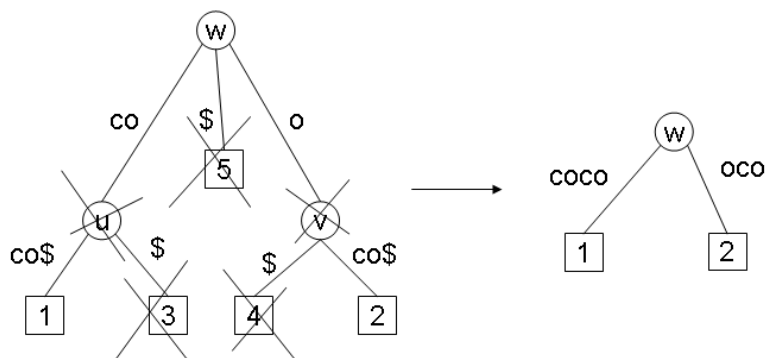


Abbildung 7: Impliziter Suffixbaum für COCO\$

4.2.2 Algorithmus in $O(m^3)$

Ukkonens Algorithmus konstruiert zuerst eine Sequenz von impliziten Suffixbäumen. Sei I_i mit $i = 1 \dots m$ ein impliziter Suffixbaum für einen Präfix $S[1 \dots i]$. Am Ende der Konstruktion wird der letzte implizite Suffixbaum I_m in einen echten Suffixbaum umgewandelt, indem man den Algorithmus letztendlich auf den $S[1 \dots i]$ anwendet. Ukkonens Algorithmus besteht aus m Phasen. In Phase $i + 1$ wird ein Baum I_{i+1} aus I_i konstruiert. Jede Phase ist in $i + 1$ Erweiterungen unterteilt. In der Erweiterung j wird zunächst das Ende des Pfades mit der Kantenbeschriftung $S[j \dots i]$, von der Wurzel ausgehend, gefunden, und danach wird ein Zeichen $S(i + 1)$ am Ende des Pfades hinzugefügt, falls es das Zeichen noch nicht im Baum an dieser Stelle gibt. Beim Hinzufügen unterscheidet man 3 Fälle:$

Fall 1: Wenn $S[j \dots i]$ an einem **Blatt** endet, dann wird am Ende das Zeichen $S(i + 1)$ hinzugefügt.

Fall 2: Wenn $S[j \dots i]$ an einem Punkt endet, von welchem **kein** Pfad mit dem Zeichen $S(i+1)$ beginnt, dann wird eine neue Kante mit der Kantenbeschriftung $S[i+1]$ konstruiert.

Fall 3: Wenn $S[j \dots i]$ an einem Punkt endet, von welchem **ein** Pfad mit dem Zeichen $S(i+1)$ beginnt, dann ist es nichts zu machen, weil der Pfad $S[j \dots i+1]$ schon im Baum enthalten ist.

Ukkonens Algorithmus⁴ in $O(m^3)$

```

Konstruiere I(1);           -- die Kante (Wurzel,1)
    mit der Kantenbeschriftung S[1];
for i in 1..m-1 do         -- Phase i+1
    for j in 1..i+1 do     -- Erweiterung j
        finde das Ende des Pfades mit der Kantenbeschriftung
            S[j..i], von der Wurzel ausgehend;
        -- wenn nötig, füge das Zeichen S(i+1) am Ende des Pfades ein;
        if (S[j..i] endet an einem Blatt) then
            füge das Zeichen S(i+1) am Ende des Pfades ein;
        elseif (S[j..i] an einem Punkt endet,
            von welchem kein Pfad mit der S(i+1) beginnt) then
            konstruiere eine Kante mit der Kantenbeschriftung S[i+1];
        elseif (S[j..i] an einem Punkt endet,
            von welchem ein Pfad mit der S(i+1) beginnt) then
            skip;
        fi;
    od;
od;
Konstruiere B(m) aus I(m);

```

Auf der Abbildung 8 kann man die Konstruktion des Suffixbaumes für *coco* mit Hilfe von Ukkonens Algorithmus zu verfolgen.

Bezüglich der Komplexität lassen sich folgende Aussagen erwähnen: In jeder Phase wird ein Baum I_{i+1} aus I_i in $O(i^2)$ Schritten konstruiert, und da wir genau m Phasen haben, benötigen wir $O(m^3)$ Schritte, um den Baum I_m zu konstruieren.

4.2.3 Implementierungstricks

Wie früher erwähnt wurde, werden wir jetzt Aufwand $O(m^3)$, der absolut inakzeptabel ist (sogar im Vergleich zu dem naiven Ansatz), zu $O(m)$ reduzieren. Zuerst verbessern wir die Komplexität des gesamten Algorithmus zu $O(m^2)$, indem wir den Aufwand $O(m)$ für eine einzelne Phase schaffen. Dafür wird einen *skip/count trick* verwendet, der jedoch eine Definition und ein paar darauf aufbauende Aussagen erfordert.

⁴nach Quelle[1]

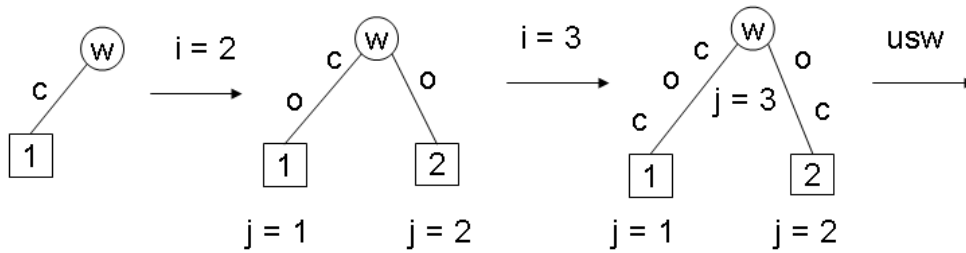


Abbildung 8: Veranschaulichung von Ukkonens Algorithmus für COCO

Definition: Ein Suffix-Link ist ein Verweis von einem Knoten u auf einen anderen Knoten $s(u)$, wobei u die Pfadbeschriftung $\chi\alpha$ und $s(u)$ die Pfadbeschriftung α haben muss, mit $\chi \in \Sigma$ und $\alpha \in \Sigma^*$.

Ein Suffix-Link (u, v) ist auf der Abbildung 9 zu sehen⁵.

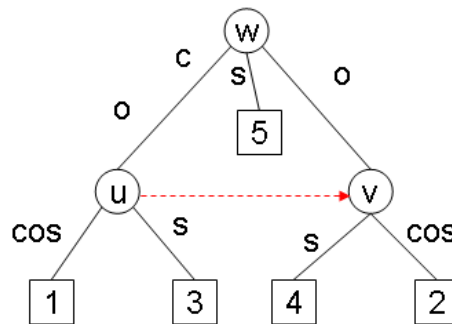


Abbildung 9: Suffix-Link

Anmerkung: Wenn α leer ist, dann geht der Verweis zur Wurzel, die selbst keine Suffix-Links hat.

Lemma: Wenn ein innerer Knoten u mit der Pfadbeschriftung $\chi\alpha$ in der Erweiterung j von der Phase $i + 1$ erzeugt wird, dann existiert entweder bereits ein innerer Knoten mit der Pfadbeschriftung α , oder er wird in der Erweiterung $j + 1$ von der Phase $i + 1$ erzeugt.

⁵ $\chi\alpha = co$ und $\alpha = o$

Daraus folgen einige Korollare⁶:

Korollar: *In Ukkonens Algorithmus hat jeder neu erzeugte Knoten spätestens in der nächsten Erweiterung einen ausgehenden Suffix-Link.*

Korollar: *Wenn es im Baum I_i einen Knoten u mit der Pfadbeschriftung $\chi\alpha$ gibt, dann gibt es auch den Knoten $s(u)$ mit der Pfadbeschriftung α .*

Mit Hilfe von Suffix-Links und unter Verwendung des letzten Korollars läßt sich das Auffinden vom Ende des Pfades mit der Kantenbeschriftung $S[j \dots i]$ verifizieren. Es muss nicht der ganze Pfad von der Wurzel durchgelaufen werden. Das Ende des Pfades mit der Kantenbeschriftung $S[1 \dots i]$ ist ein Blatt, und wenn wir einen Zeiger auf dieser Stelle setzen, können wir in jeder Phase den Durchlauf der ersten Erweiterung sparen. Für weitere Erweiterungen ist das generelle Vorgehen wie folgt zu beschreiben: Erstens, starte am Ende des Strings $S[j - 1 \dots i]$ und gehe nach oben zu der Wurzel, bis ein Knoten u erreicht wird. Sei die Kantenbeschriftung von dieser Kante γ . Angenommen u ist keine Wurzel, dann folge dem Suffix-Link, die dieser Knoten nach Korollar besitzt. Von $s(u)$ gehe nach unten entlang γ . Schließlich erweitere die Kantenbeschriftung mit $S(i+1)$.

SEA⁷ Algorithmus:

```
finde den ersten Knoten u vom Ende S[j-1..i];
y := Kantenbeschriftung von (u,-);
if (u = Wurzel) then
  folge den Pfad S[j..i] wie im naiven Algorithmus;
else
  traversiere Suffix-Link von u nach s(u);
  gehe nach unten und folge y;
fi;
erweitere die Kantenbeschriftung mit S(i+1);
if (einen Knoten w in der Erweiterung j-1 erzeugt wurde) then
  erzeuge ein Suffix-Link (w,s(w));
fi;
```

Bei der Ausnutzung des Suffix-Links reduziert man viele Schritte, aber im *worst-case* wird die Verbesserung der Zeitkomplexität nicht garantiert. Jetzt wird ein Trick vorgestellt, der Komplexität $O(m^2)$ auch im schlechtesten Fall gewährleistet.

Der Durchlauf von γ unten von $s(u)$ erfordert $|\gamma|$ Schritte, der so genannte **skip/count trick** reduziert dagegen diesen Aufwand auf $|\text{Anzahl von Knoten in } \gamma|$, sodass der Aufwand für alle Durchläufe in einer Phase $O(m)$ bleibt.

⁶Die Beweise zum Lemma und Korollare ist in [1] zu finden.

⁷Single extension algorithm

Skip/count trick: Sei g am Anfang die Länge von γ und g' die Länge der aktuellen Kante. Dann dank der Eigenschaft (3) in der Definition von Suffixbäumen, brauchen wir nicht jedes Zeichen auf dem Pfad zu vergleichen, sondern wenn $g > g'$ ist, können wir sofort zu dem nächsten Knoten, wenn nicht, dann ist man fertig, weil der Pfad in dieser Kante genau nach $g' - g$ Zeichen nach dem Knoten (die man auch nicht durchlaufen muss) endet. g muss man dabei in jedem Schritt aktualisieren.

Definition: Die Tiefe eines Knotens ist die Anzahl von Knoten auf dem Pfad von der Wurzel bis zu diesem Knoten.

Lemma: Sei $(u, s(u))$ ein Suffix-Link, dann ist die Tiefe von u höchstens um eins größer als die Tiefe von $s(u)$.

Mit diesem Lemma lässt sich folgender Satz beweisen⁸.

Satz: Bei der Verwendung von Suffix-Links und dem skip/count trick benötigt eine Phase von Ukkonens Algorithmus die Komplexität $O(m)$.

Und schließlich, da wir genau m Phasen haben, ergibt sich mit der Multiplikation folgender Satz:

Satz: Mit Hilfe von Suffix-Links und dem skip/count trick ist Ukkonens Algorithmus in $O(m^2)$ zu lösen.

Man kann überrascht sein, dass wir jetzt bei der Verbesserung der Komplexität genau an der Stelle sind, wo wir angefangen haben, weil auch der naive Algorithmus den Aufwand $O(m^2)$ hatte. Aber mit Hilfe von kleinen Tricks sind wir jetzt bei $O(m)$ für den gesamten Algorithmus.

Es ist aber noch ein kleines Detail zu erwähnen. Wir haben davor nur von Zeitkomplexität geredet, die Platzkomplexität von Suffixbäumen kann aber im schlechtesten Fall $\Theta(m^2)$ sein. Das ist aber unerwünscht. Um das zu vermeiden, benutzt man wiederum einen Trick: anstatt die Zeichen explizit zu speichern, speichert man nur die Indices, die Anfang und Ende eines Teilstrings bezeichnen. Da es maximal $2m - 1$ Kanten geben kann, und man für jede Kante nur zwei Zahlen speichert, wird dadurch erwünschte $O(m)$ -Komplexität erreicht.

Die zwei nächsten Tricks sind der letzte Punkt auf dem Weg zur linearen Komplexität.

Show stopper Dieser Trick besagt Folgendes: Wenn in der Phase i in der Erweiterung j beim Hinzufügen von $S[i+1]$ der Fall 3 auftritt, dann tritt der Fall 3 (Sieh 4.2.2) in jeder nächsten Erweiterung dieser Phase auf. Das ist leicht zu erklären, weil wenn ein String $S[j \dots i+1]$ im Baum enthalten ist, dann sind auch alle Strings $S[k \dots i+1]$, mit $k > j$, im Baum enthalten. Deswegen braucht man in diesem Fall keine weiteren Erweiterungen dieser Phase explizit zu machen und kann sofort zur nächsten Phase springen. Es wird auch kein neuer Knoten erzeugt und daher auch kein Suffix-Link.

⁸Die Beweise zum Lemma und zum Satz ist in [1] zu finden.

Once a leaf, always a leaf Dieser Trick besagt Folgendes: Wenn in der Phase i in der Erweiterung j beim Hinzufügen von $S[i+1]$ der *Fall 1* oder der *Fall 2* auftritt, dann tritt in der Erweiterung j in jeder nächsten Phase nur der *Fall 1* auf. Deswegen kann man einen globalen Index (man erinnert sich an die Speicherung mittels Indizes) e benutzen, der das Ende des Strings $S[k \dots i+1]$ bezeichnet und wird in jeder Phase auf $i+1$ gesetzt. Man muss sich den Index der letzten Erweiterung, bei deren *Fall 1* oder der *Fall 2* auftritt, merken.

Aufbauend auf diesen Tricks ist folgender Algorithmus für eine Phase zu konstruieren: Sei j_i die letzte Erweiterung in Phase i in *once a leaf, always a leaf* Trick und j_i^* die erste Erweiterung in Phase i im *show stopper* Trick.

SPA⁹ Algorithmus:

```
e := i+1;
for j in j(i)+1..j(i)* do
SEA(j);
od;
j(i+1) := j*-1;
```

In der ersten Zeile des Algorithmus wird ein globaler Index, der das Ende des Strings, der erweitert werden muss, bezeichnet, auf $i+1$ gesetzt. Damit werden, dank des *once a leaf, always a leaf* Tricks, Erweiterungen $1 \dots j_i$ in $O(1)$ implizit realisiert. Dann werden weitere Erweiterungen $j_i + 1 \dots j_i^*$ mit Hilfe von SEA-Algorithmus explizit berechnet, bis in Erweiterung j_i^* der *Fall 3* auftritt. Danach werden, dank des *show stopper* Tricks, alle nachfolgenden Erweiterungen $j_i^* + 1 \dots i+1$ auch implizit in $O(1)$ realisiert. In der letzten Zeile des Algorithmus findet eine Vorbereitung zur nächsten Phase statt. Dabei wird j_{i+1} auf $j_i^* - 1$ gesetzt. Wenn man diesen Übergang von Phase i nach Phase $i+1$ betrachtet, den auch auf der Abbildung 10 veranschaulicht ist, wird es klar, dass der Algorithmus gewünschte Komplexität besitzt:

Satz: *Mit Hilfe von Suffix-Links und den Tricks „skip/count trick“, „show stopper trick“ und „once a leaf, always a leaf trick“ ist Ukkonens Algorithmus in $O(m)$ zu lösen.*

4.3 Weiners und McCreights Algorithmen

Hier wird kurz über weitere Suffixbaumkonstruktionsalgorithmen im Überblick und im Vergleich zu Ukkonens gesprochen.

Weiners Algorithmus [5] ist der erste Suffixbaumkonstruktionsalgorithmus, deshalb ist er in historischen Aspekten interessant. Er ist aber sehr speicherintensiv, und bezüglich der Vorgehensweise verhält er sich anders als Ukkonens Algorithmus, deswegen ist er relativ schwer zu verstehen.

⁹Single phase algorithm

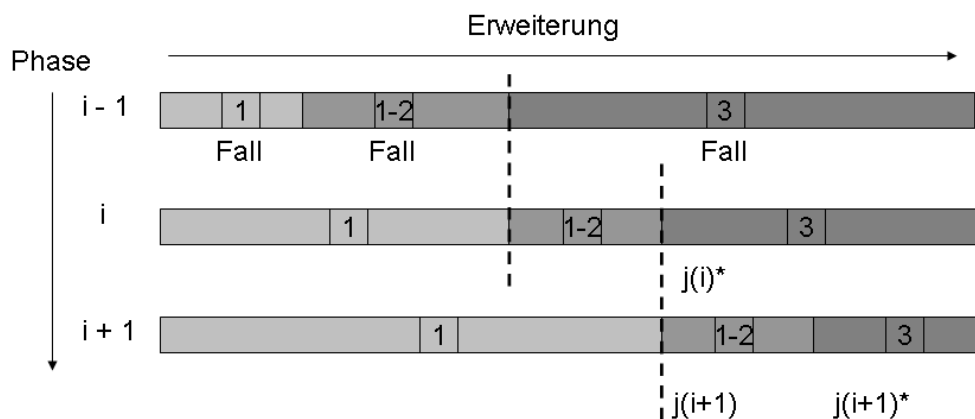


Abbildung 10: Veranschaulichung von Tricks

McCreights Algorithmus [6] wurde früher als Ukkonens Algorithmus entworfen, und hat alle ursprünglichen Ideen, die in Ukkonens Algorithmus auch benutzt wurden, die Vorgehensweise und Implementierungstricks sind sehr ähnlich, deswegen kann man Ukkonens Algorithmus als eine Variante von McCreights Algorithmus betrachten. Er ist auch wie Ukkonens Algorithmus speichersparend. McCreights Algorithmus hat aber im Vergleich mit Ukkonens Algorithmus einen Nachteil. Er ist kein *Online-Algorithmus*. Und aus diesem Grund ist er bei bestimmten Problemen nicht einsetzbar. Aber er kann relativ einfach an veränderten Text angepasst werden.

5 Zusammenfassung und Fazit

In dieser Arbeit wurden Suffixbäume vorgestellt. Wir haben gesehen, wie man eine solche Datenstruktur naiv konstruieren kann und wie man in ihr suchen kann. Es wurde auch Ukkonens Algorithmus zur Konstruktion von Suffixbäumen detailliert vorgestellt. Dabei wurden alle Implementierungstricks, mit Hilfe von denen lineare Komplexität erreicht wird, erläutert.

Suffixbäume entsprechen allen Anforderungen für exakte Textsuche, deswegen finden sie in vielen Suchmaschinen Anwendung. Diese Struktur erlaubt auch die Lösung von weiteren komplexeren Stringverarbeitungsproblemen, so wie: längste gemeinsame Zeichenkette, Palindrome, Suche mit Wildcards usw. Das war die erste Datenstruktur für die Lösung des *longest common substring problem* (also längste gemeinsame Zeichenkette) in linearer Zeit.

Suffixbäume, wie schon früher erwähnt wurde, sind außer *Information Retrieval* auch in *Bioinformatik* von besonderer Bedeutung.

Literatur

- [1] Gusfield, Dan: Algorithms on Strings, Trees, and Sequences. Cambridge University Press, 1997.
- [2] <http://de.wikipedia.org/wiki/Suffixbaum>
- [3] G. H. Gonnet, R. A. Baeza-Yates, T. Snider: New Indices for Text: PAT Trees and PAT arrays. In: Frakes, William; Baeza-Yates, Ricardo (eds.): Information Retrieval. Data Structures and Algorithms. Englewood Cliffs, N.J.: Prentice Hall, 1992 (Kap. 5)
- [4] Ukkonen, Esko: On-Line Construction of Suffix Trees. In: Algorithmica 14 (1995), Nr. 3, S. 249-260, Department of Computer Science, University of Helsinki, Finland. <http://www.cs.helsinki.fi/u/ukkonen/SuffixT1withFigs.pdf>
- [5] Weiner, Peter: Linear pattern matching algorithms. 1973, S. 1-11
- [6] McCreight, Edward M.: A Space-Economical Suffix Tree Construction Algorithm. In: Journal of the ACM 23, 1976, Nr. 2, S. 262-272