

String Matching Algorithmen

Skript von Stephan Hell zur Vorlesung von Frank Hoffmann

Sommersemester 2000

Inhaltsverzeichnis

1	Exaktes Pattern Matching : Klassische Methoden	4
1.1	Z-Algorithmus (nach Gusfield)	4
1.2	Boyer-Moore Algorithmus ('77)	7
1.3	Knuth-Morris-Pratt Algorithmus ('77)	8
2	Exaktes Matching mit mehreren Mustern	9
2.1	Aho-Corasick Algorithmus	10
2.1.1	Bestimmung der Fehler-Links in $O(n)$	11
2.1.2	Anwendung 1: Matching mit Wildcards	14
2.1.3	Anwendung 2: Zweidimensionales Matching	15
3	Matching mit regulären Ausdrücken	15
3.1	Wiederholung: Reguläre Ausdrücke	15
3.2	Der Algorithmus	17
4	Seminumerisches Stringmatching	18
4.1	Shift-AND-Methode (nach Baeza-Yates, Gonnet '92)	18
4.1.1	Exaktes Matching	18
4.1.2	Matching mit Fehlern (Wu, Manber '92)	19
4.2	Fingerprint-Methode nach Karp-Rabin '87	20
4.2.1	Algorithmus	20
4.2.2	Einiges über Primzahlen	21
4.2.3	Die Wahrscheinlichkeit für ein falsches Match	22
5	Suffix-Bäume und ihre Anwendungen	23
5.1	Anwendungen	24
5.1.1	Exaktes Matching	24
5.1.2	Eine Datenbank für Muster	24
5.1.3	Longest-Common-Substring von S_1 und S_2	25
5.1.4	All-Pairs-Suffix-Präfix-Matching	25
5.2	Konstruktion von Suffixbäumen in Linearzeit	26
5.2.1	Erster Trick: Suffix-Links	28
5.2.2	Zweite Verbesserung: Skip & Count	30
5.3	Suffix-Arrays und Anwendungen	33

6	Suffixbäume und <i>lca</i>-Anfragen	34
6.1	<i>lca</i> -Anfragen in konstanter Zeit	35
6.1.1	<i>lca</i> -Anfragen in vollständigen binären Bäumen	35
6.1.2	<i>lca</i> -Anfragen in allgemeinen Bäumen – Vorverarbeitung	36
6.1.3	Beantwortung der <i>lca</i> -Anfragen in konstanter Zeit . . .	38
6.2	Anwendung	41
7	Approximatives Matching	42
7.1	Editierabstand, globales & lokales Alignment	42
7.2	Ähnlichkeit von Strings	44
7.3	Das local-Alignment-Problem	45
7.4	Gaps und Gap Weights	46
7.5	Verbesserung der dynamischen Programmierung	47
7.5.1	Linearer Speicherbedarf	47
7.5.2	<i>k</i> -difference Alignment/Matching	49
7.6	Suffixbäume und gewichtete Alignments	52
7.6.1	<i>P</i> -against-all	52
7.6.2	All-against-all	53
7.7	Ausschlußverfahren	54
7.7.1	Verbesserung: Mehrfachfilter für <i>k</i> -Mismatches	56
7.8	Ein kombinatorischer Algorithmus für <i>lcs</i>	56

1 Exaktes Pattern Matching : Klassische Methoden

Definition 1 Sei Σ ein Alphabet aus Zeichen, $S \in \Sigma^*$ nennen wir Wort oder String. $S(i)$ bezeichnet den i -ten Buchstaben von S , $S[i, j]$ das Teilwort $S(i) \dots S(j)$.

Aufgabe:

- Gegeben: Text $T \in \Sigma^*$, $|T| = m$, Pattern $P \in \Sigma^*$, $|P| = n \leq m$.
- Gesucht: alle Vorkommen von P als Teilwort in T .

Naive Methode:

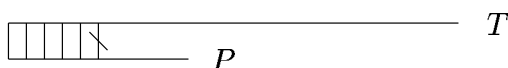


Abbildung 1: Naive Methode

Das Muster P (=Pattern) wird von links nach rechts über den Text T geschoben, und jede Stelle von T wird als möglicher Anfang des Musters getestet. Dabei werden $n \cdot (m - n + 1)$ Vergleiche gemacht, der Algorithmus benötigt also $O(n \cdot m)$ Zeit (1 Vergleich = 1 Zeiteinheit).

Eine offensichtliche untere Schranke für das Problem ist $\Omega(n + m)$, denn der Text und das Pattern müssen gelesen werden. Das Ziel ist es nun, einfache Linearzeitalgorithmen zu entwickeln (deren *erwartete* Laufzeit sogar sublinear ist).

Idee: Beim Testen von P lernen wir T lokal, dadurch sind größere Sprünge möglich \rightarrow Vorverarbeitung von Text (Suffixtrees) und/oder Pattern.

1.1 Z-Algorithmus (nach Gusfield)

Definition 2 Sei S ein String, $i > 1$ eine Position. Wir definieren:

$Z_i = Z_i(S) =$ Länge des längsten Präfix von S , der auch Präfix von $S[i, |S|]$ ist.

Beispiel. $S = aabcaabxaaaz$, $Z_5(S) = 3$.

Definition 3 Es sei $S[i, i + Z_i - 1]$ Z-Box für die Position i , wobei $Z_i > 0$. Für $i \geq 1$ sei r_i das größte rechte Ende einer Z-Box, die vor oder bei i beginnt, und l_i das zu diesem r_i gehörige linke Ende.

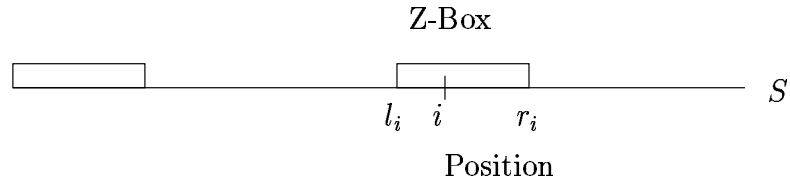


Abbildung 2: Z – Box

Satz 1 Die Z-Boxen können in $O(|S|)$ berechnet werden.

Korollar 1 Pattern matching geht in linearer Zeit.

Beweisskizze zum Korollar: Findet man in diesem S eine Z-Box mit Länge $|P|$, so hat man das Pattern gefunden und ein Matching liegt vor.

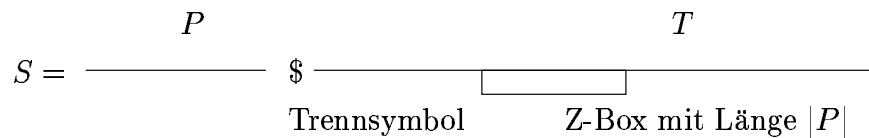


Abbildung 3: Beweisskizze

Beweis. (vom Satz)

- Z_2 wird explizit berechnet.
- Die erste Z-Box wird berechnet:

$$r = r_2 = \begin{cases} Z_2 + 1 & : Z_2 > 0 \\ 0 & : \text{sonst} \end{cases}$$

$$l = l_2 = \begin{cases} 2 & : Z_2 > 0 \\ 0 & : \text{sonst} \end{cases}$$

- Angenommen Z_i sei für $1 < i < k$ bekannt, setze $r = r_{k-1}$ und $l = l_{k-1}$. Wir berechnen nun Z_k, l_k, r_k .

1. Fall: $k > r$

Dann setze $r_k := k + Z_k - 1$ und $l_k = k$, falls $Z_k > 0$.

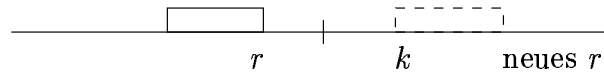


Abbildung 4: 1.Fall: $k > r$

2. Fall: $k \leq r$

Die Position k liegt somit in einer schon berechneten Z-Box α , welche wie der Anfang aussieht. Nennen wir den Teilstring $T[k, r]$ β . Nun gibt es am Anfang die Position $k' = k - l + 1$, die genau wie die Position k aussieht und von der wir die Z-Box γ - nach Annahme - schon berechnet haben.

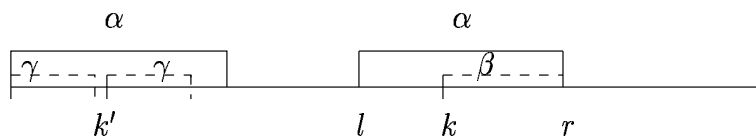


Abbildung 5: 2.Fall: $k \leq r$

- $Z_{k'} = |\gamma| < |\beta|$, dann setze $Z_k = Z_{k'}$, r und l wie vorher.
- $Z_{k'} = |\gamma| \geq |\beta|$, dann vergleiche $S(r + 1)$ mit $S(|\beta| + 1) \dots$, bis ein Mismatch bei Position $q \geq r + 1$ auftritt. Dann setze $Z_k = q - k$, $r = q - 1$ und $l = k$. \square

Analyse:

- Zeit = #Iterationen + #Vergleiche, wobei gilt:
Man hat $|S| = \#Iterationen$ und die Anzahl der Vergleiche hängt von den Matches und Mismatches ab. Dabei hat man $\#Mismatches \leq |S|$, $r_k \geq r_{k-1}$ und $\#Matches \leq |S|$.
- $O(|S|)$ zusätzlicher Speicher wird benötigt.
- Die Laufzeitbetrachtung ist alphabetunabhängig.

1.2 Boyer–Moore Algorithmus ('77)

Idee:

1. Das Pattern P wird von links nach rechts über den Text T geschoben, aber von rechts nach links verglichen. Ziel: Sprünge beim Vergleichen.

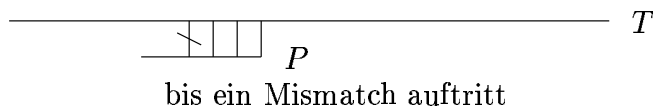


Abbildung 6: Idee

2. (a) Bad-Character-Regel

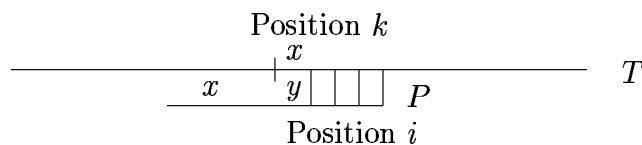


Abbildung 7: 2. Fall: $k \leq r$

Definition 4

$$R(x) = \begin{cases} \text{rechtste Position von } x \text{ in } P \\ 0, \text{ wenn kein } x \text{ in } P \end{cases}$$

Wenn es nun zu einem Mismatch $T(k) \neq P(i)$ kommt, besagt die Regel: Verschiebe P um $\max\{1, i - R(T(k))\}$ Positionen.

Beispiel.

Sei $i = 3, k = 5$.

T : xpbctbxabpqxctb

P : tpaxab

Somit ist $T(k) = t$ und $R(T(k)) = 1$, und daraus erhält man nach der Regel einen shift um $3 - 1 = 2$ Positionen.

- (b) Strong-good-Suffix-Regel

Nach einem Mismatch verschieben wir das Pattern P bis zur rechtensten Kopie t' von t , die in P links von t steht und bei der der Buchstabe links von t' ein anderer ist als links von t . Wenn so

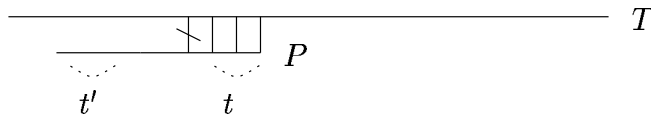


Abbildung 8: Strong-good-Suffix-Regel

eine Kopie t' nicht existiert, verschieben wir das linke Ende von P soweit in t , bis ein Präfix von P einen Suffix von t matcht. Falls dieser Präfix nicht existiert, verschieben wir um n Positionen. Analog verfahren wir nach einem Matching.

Beispiel.

p	r	s	t	a	b	s	t	u	b	a	b	v	\dots	Text T
	q	c	a	b	d	a	b	d	a	b				Pattern P
								q	c	a	b	d	\dots	

Abbildung 9: Beispiel

$t = ab$ startet bei $P(3)$ und Mismatch bei Position 10, damit shift um 6 Positionen.

Der Boyer-Moore Algorithmus verschiebt um das Maximum der beiden shift-Werte, die von der Bad-Character-Regel und der Strong-good-Suffix-Regel geliefert werden. Zum Auffinden der richtigen Kopien t' an jeder Position des Patterns ist eine Vorverarbeitung des umgedrehten Patterns P^{rev} mit dem Z-Algorithmus notwendig.

Laufzeitanalyse (mit Modifikation): Im worst-case $O(n + m)$, die beste bekannte Schranke ist $\leq 2m$ Vergleiche (\rightarrow Apostolico, Giancarlo '86).

1.3 Knuth-Morris-Pratt Algorithmus ('77)

Idee:

- Verschieben des Patterns P von links nach rechts.
- Vergleichen von links nach rechts.
- Wie weit kann man das Pattern nach einem Mismatch verschieben?

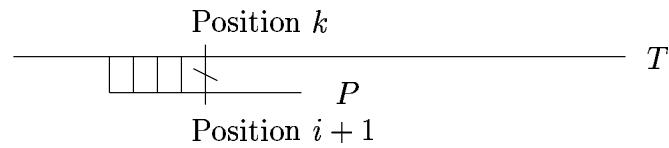


Abbildung 10: Idee

Definition 5 $sp'_i =$ Länge des längsten Suffix von $P[1, i]$, der Präfix von P matcht und $P(i + 1) \neq P(sp'_i + 1)$.

Wir verschieben um $(i - sp'_i)$ Stellen, und testen als nächstes $T(k)$ gegen $P(sp'_i)$. Falls kein Mismatch auftritt, verschieben wir um $n - sp'_n$ Stellen. Dabei wird tatsächlich kein Auftreten des Patterns in T übersehen, Beweis in der Übung.

Beispiel.

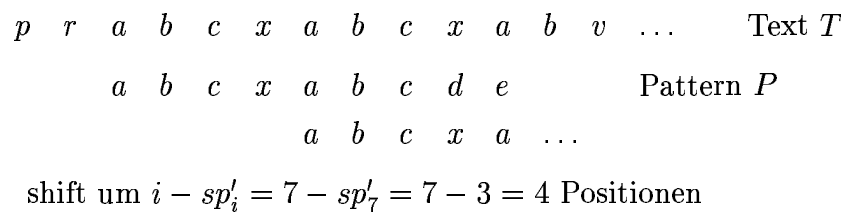


Abbildung 11: Beispiel

Um die sp'_i zu bestimmen, ist eine Vorverarbeitung des Patterns P mit dem Z-Algorithmus notwendig.

2 Exaktes Matching mit mehreren Mustern

Sei T ein Text, $|T| = m$ und eine Menge von Pattern

$$\mathcal{P} = \{P_1, P_2, \dots, P_z\}, \text{ mit } |\mathcal{P}| = \sum_{i=1}^z |P_i| = n,$$

gegeben.

Dann läßt sich auf triviale Art und Weise durch z -fache Anwendung der kennengelernten Algorithmen eine Laufzeit von $O(n + z \cdot m)$ erreichen.

Das Ziel ist ein Algorithmus mit Laufzeit $O(n + m + k)$, wobei k der Anzahl der Vorkommen der Pattern P_1, P_2, \dots, P_z entspricht. Beachte: die Anzahl k der Pattern kann echt größer sein als die Summe der Text- und Patternlänge. Dazu ein

Beispiel. Sei $T = aa \dots a$ und

$$\mathcal{P} = \left\{ \underbrace{aa \dots a}_{\frac{m}{2}}, \underbrace{aa \dots a}_{\frac{m}{4}}, \dots \right\}$$

Dann hat man: $|\mathcal{P}| = O(m)$ und

$$\# \text{Vorkommen: } \underbrace{\frac{m}{2} + \frac{3m}{4} + \dots}_{\log_2 m \text{ Summanden}} \rightarrow O(m \log m).$$

2.1 Aho–Corasick Algorithmus

Annahme: Die Menge \mathcal{P} der Muster sei teilstringfrei (d.h. kein P_i ist Teilstring eines anderen P_j).

Idee: Wir führen als neue Datenstruktur einen Keyword–Baum \mathcal{K} ein, der folgende Eigenschaften hat:

- \mathcal{K} ist ein gerichteter Baum mit Wurzel r .
- Die Kanten des Baumes tragen Zeichen aus dem Alphabet.
- Kanten, die von einem Knoten ausgehen, tragen verschiedene Zeichen.
- Ein Weg von der Wurzel zu einem Blatt entspricht einem Muster P_i .

Analog zum KMP Algorithmus wird der Baum gegen den Text getestet. Dabei ist wieder die Frage, wie weit die Wurzel nach einem Mismatch verschoben werden kann. Um das entscheiden zu können, ist eine Vorverarbeitung des Keyword–Baumes notwendig. Dabei werden sogenannte *Fehlerlinks* gesetzt.

Definition 6 Sei $v \in \mathcal{K}$ ein Knoten. Wir nennen

$$L(v) := \text{Wort von der Wurzel } r \text{ bis zu } v$$

das Label von v . Außerdem bezeichne

$$lp(v) = \begin{cases} \text{Länge des längsten echten Suffix } \alpha \text{ von} \\ L(v), \text{ der Präfix eines Muster aus } \mathcal{P} \text{ ist.} \end{cases}$$

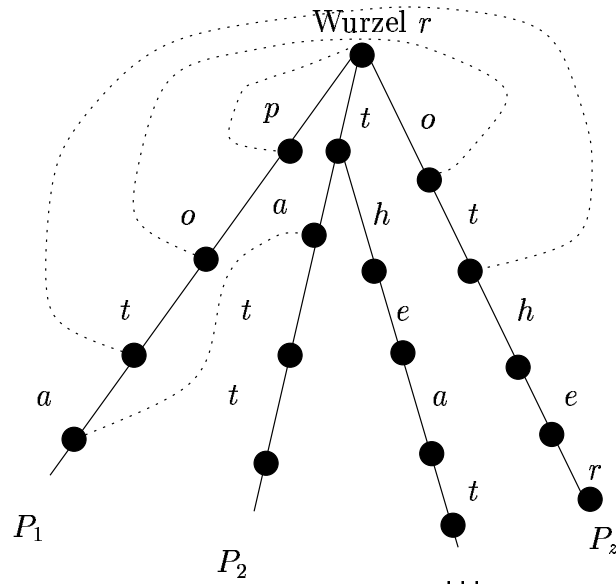


Abbildung 12: Keyword-Baum mit Fehler-Links am linken Ast

Wir setzen Fehler-Links n_v für jedes $v \in \mathcal{K}$, wobei

$$n_v = \text{Knoten in } \mathcal{K} \text{ mit } L(n_v) = \alpha.$$

Algorithmus:

Wir vergleichen den Text T mit dem Keyword-Baum \mathcal{K} . Nun tritt ein Mismatch zwischen $T(c)$ und der Kante (w, w') auf. Dann setzen wir:

$$l = c - lp(w), w = n_w, \text{ falls } lp(w) > 0 \text{ und } l = c, w = r, \text{ falls } lp(w) = 0.$$

2.1.1 Bestimmung der Fehler-Links in $O(n)$

Wir bestimmen die Fehler-Links induktiv über den Abstand zur Wurzel r .

- Induktionsanfang: Sei v ein Knoten mit Abstand 1. Dann setzen wir $n_v = r$.
- Induktionsannahme: Seien die Fehlerlinks für Knoten mit Abstand $\leq k$ zur Wurzel bekannt.
- Induktionsschritt: Sei v ein Knoten mit Abstand $k+1$ zur Wurzel. Wir suchen nun n_v .

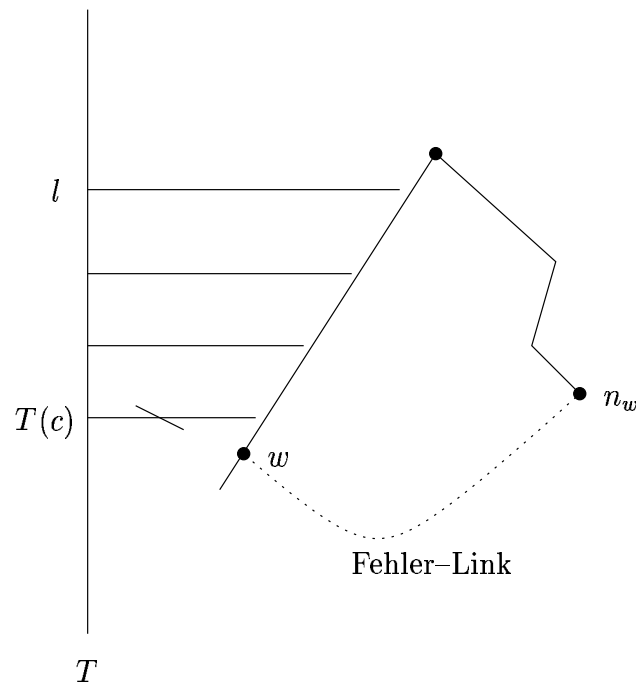


Abbildung 13: Aho-Corasick Alg.

Es existiert ein Vorgänger v' von v und eine Kante (v', v) mit einem Zeichen x . Der Abstand von v' zur Wurzel ist k , und damit kennen wir seinen Fehler-Link $n_{v'}$ nach Annahme. $L(n_v)$ muß ein Suffix von $L(n_{v'}) \cdot x$ sein. Geht nun von diesem Knoten $n_{v'}$ eine Kante mit dem Zeichen x ab, so setzen wir den Fehler-Link n_v von v auf die Endecke dieser Kante.

Existiert diese Kante nicht, so kennen wir nach Voraussetzung den Fehler-Link $n_{n_{v'}}$, denn der Abstand zur Wurzel wird in jedem Schritt echt kleiner. Geht nun von diesem Knoten $n_{n_{v'}}$ eine Kante mit dem Zeichen x ab, so setzen wir den Fehler-Link n_v von v auf die Endecke dieser Kante.

Existiert diese Kante nicht, so kennen wir den Fehler-Link ... usw.

Die Fehlerlinks werden damit tatsächlich in Zeit $O(n)$ bestimmt, das sieht man wie folgt:

Sei P_i , $|P_i| = t$, ein Muster aus \mathcal{P} . Wieviel Zeit wird gebraucht, um die Fehler-Links der Knoten auf dem Pfad von P in \mathcal{K} zu bestimmen? Wir wissen:

- $lp(v) \leq lp(v') + 1$.
- Wenn die Schleife zur Berechnung der n_v k -mal durchlaufen wurde, dann ist $lp(v) \leq lp(v') - k$
- $lp(v) < t$ und $lp(v) \geq 0$

Das heißt also, für alle Knoten auf dem Pfad eines Musters der Länge t wird die Schleife insgesamt höchstens t Mal durchlaufen, innerhalb der Schleife wird konstante Zeit benötigt. Die Gesamtlänge aller Muster ist n , also kann es zu höchstens n Schleifendurchläufen kommen, und damit werden *alle* Fehler-Links in $O(n)$ Zeit bestimmt.

Für den Fall, daß man P_i als Teilstring von P_j zuläßt, muß man zusätzliche Output-Links hinzufügen, die einem sagen, daß man gerade über ein Muster P_i gelaufen ist. Dabei hilft die folgende

Beobachtung. Angenommen, bei der Suche wird der Knoten v in \mathcal{K} erreicht, c ist die aktuelle Position in T . Dann gilt :

P_i ist in T und endet in c
 \iff
 v ist mit i markiert oder es gibt einen gerichteten Pfad von Fehler-Links zu einem mit i markierten Knoten (\rightarrow Output-Link).

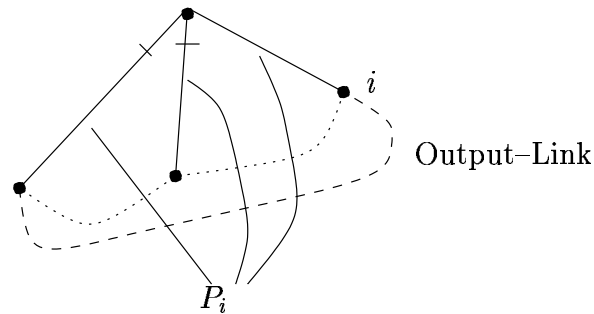


Abbildung 14: Output-Links

Das heißt, es reicht im Algorithmus, bei jeder Kante (w, w') in \mathcal{K} zusätzlich ein Auftreten von P_i mit Ende in Position c im Text zu melden, wenn w' mit i markiert ist *oder* es einen Pfad von Fehler-Links von w' zu einem mit i markierten Knoten gibt.

2.1.2 Anwendung 1: Matching mit Wildcards

Definition 7 Wir nennen ein Zeichen $\phi \notin \Sigma$, das jedes Zeichen $a \in \Sigma$ matcht, eine Wildcard.

Aufgabe: Exaktes Matching von einem Muster P , $|P| = n$, mit Wildcards gegen einen Text T , $|T| = m$, ohne Wildcards.

Ansatz: Wir haben ein Muster P der Form

$$P = \phi\phi \dots P_1\phi \dots \phi P_2\phi \dots P_k\phi \dots ,$$

wobei die P_i Teilmuster *ohne* Wildcards sind. Wir definieren die Multimenge $\mathcal{P} = \{P_1, \dots, P_k\}$ und bestimmen zusätzlich den Vektor l der Anfangspositionen der P_i in P

$$l = (l_1, l_2, \dots, l_k).$$

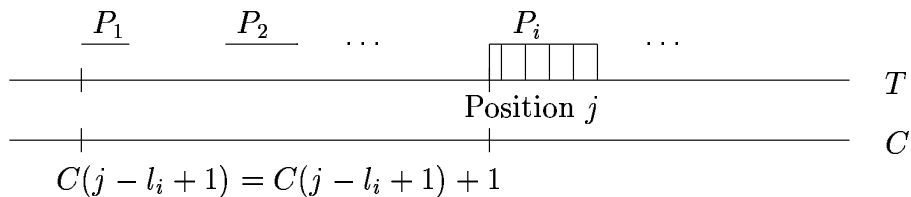


Abbildung 15: wild cards

Nun wenden wir hierfür den Aho–Corasick Algorithmus an. In einem Kontrollfeld C der Länge $|T|$ addieren wir bei einem Matching eines P_i an der Position j im Text in C an der Stelle $C(j - l_i + 1)$ eins. D.h. jedes Auftreten eines Musters P_i an einer Stelle j dient als *Zeuge* einer eventuellen Anfangsposition von ganz P bei $j - l_i + 1$. Liegen am Ende k Zeugen an einer Stelle l in C vor, so ist dies eine Anfangsposition von P . Oder kurz :

$$C(l) = k \iff P \text{ startet in } T(l)$$

Es gilt: $C(l) \leq k$, denn jedes Pattern P_i kann an einer festen Stelle l nur ein Mal ein Heraufsetzen des Zählers verursachen.

Analyse: Die Konstruktion des Keyword–Baumes geht in Zeit $O(n)$. Bei der Suche entspricht eine Zeiteinheit einem Vergleich oder einem Hochzählen in C .

Man hat $C(i) \leq k \Rightarrow$ Der A.C. Algorithmus läuft in Zeit $O(n + r \cdot m) = O(rm)$, wobei r die Anzahl der Vorkommen von Mustern aus \mathcal{P} bezeichnet. Ist diese nach oben beschränkt, so erhält man als Laufzeit $O(m)$.

2.1.3 Anwendung 2: Zweidimensionales Matching

Seien ein rechteckiger Text T mit $|T| = m$ und ein rechteckiges Muster P mit $|P| = n$ gegeben. Dabei seien zuerst alle Zeilen von P verschieden. Gesucht sind alle Auftreten von P in T .



Abbildung 16: 2-dim. Matching

Man wandelt T um in die Form $T' = \text{Zeile1 } \$ \text{ Zeile2 } \$ \dots$, wobei $\$$ als Sonderzeichen dient, um ein Matching über das Zeilenende hinaus zu verhindern. Es sei P_i die i -te Zeile des Musters P , und $\mathcal{P} = \{P_1, \dots, P_z\}$. Der Algorithmus von Aho–Corasick, angewendet auf den Text T' und die Patternmenge \mathcal{P} , liefert nun alle Auftreten von Zeilen von P in T .

In einem rechteckigen Kontrollfeld C , $|C| = m$, trägt man i an der Stelle (p, q) ein, wenn das Pattern P_i in Zeile p und Spalte q anfängt.

Um die Vorkommen von ganz P in T zu finden, muß man in C das Pattern $1\ 2\ \dots\ z$ in einer Spalte finden. Das geht mit einem beliebigen der nun schon vorgestellten Algorithmen zur Mustersuche in Zeit $O(m + z)$.

Die Restriktion, daß alle Zeilen von P verschieden sein müssen, läßt sich leicht aufheben: Identische Zeilen erhalten ein gemeinsames Label (z.B. das der kleinsten Zeile dieser Gestalt), und zum Schluß wird in C nach dem Pattern gesucht, das an der i -ten Stelle das *Label* der i -ten Zeile hat.

3 Matching mit regulären Ausdrücken

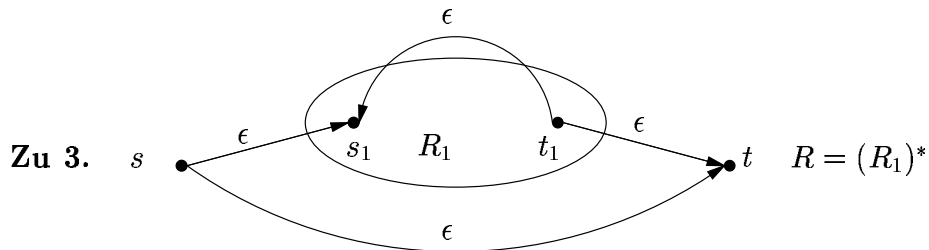
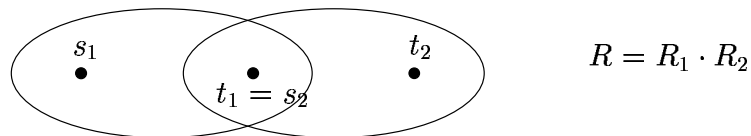
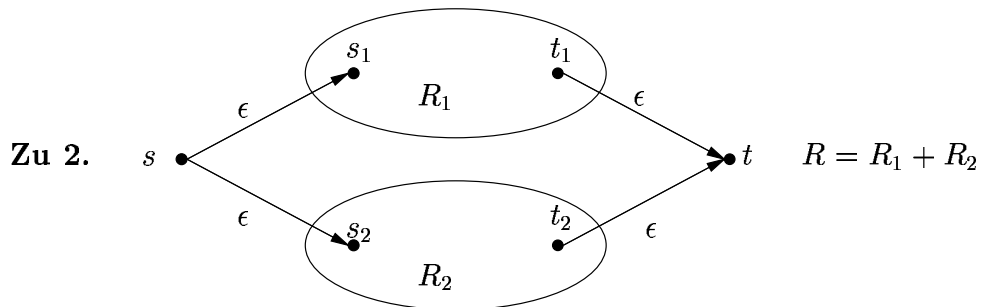
3.1 Wiederholung: Reguläre Ausdrücke

Definition 8 Sei Σ ein endliches Alphabet für das gilt: $(,) , + , \epsilon \notin \Sigma$. Wir definieren induktiv:

1. ϵ ist ein regulärer Ausdruck und $\forall a \in \Sigma$ gilt: a ist ein regulärer Ausdruck.
2. Seien R_1 und R_2 reguläre Ausdrücke, dann sind auch die Vereinigung $R_1 + R_2$, die Konkatenation $R_1 \cdot R_2$ und (R_1) reguläre Ausdrücke.
3. Sei R regulärer Ausdruck, dann auch der Kleene-Abschluß $(R)^*$.

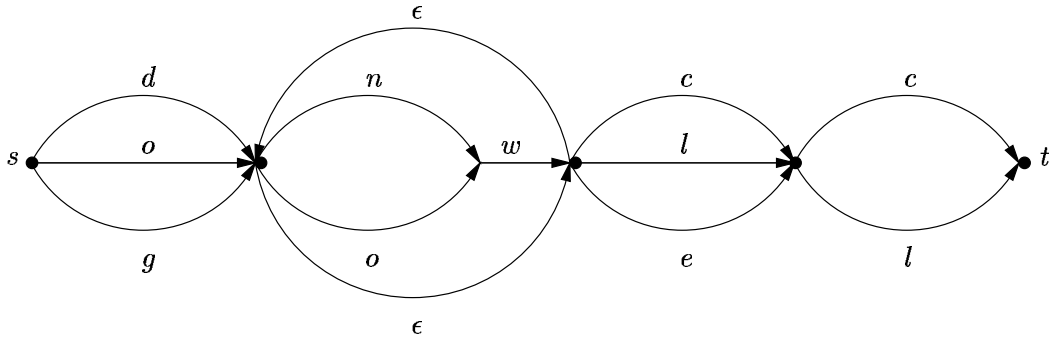
Zu jedem regulären Ausdruck läßt sich ein gerichteter Graph, der einem nicht-deterministischen, endlichen Automaten (nfa) entspricht, mit ausgezeichneten Knoten s und t sowie Kantenlabels aus $\Sigma \cup \epsilon$ konstruieren. Es entspricht dann jeder gerichtete Weg von s nach t einem Wort, das gemäß dem regulären Ausdruck gebildet wurde. Die Konstruktionsbeschreibung dieses Graphen ist ebenfalls induktiv:

Zu 1. $s \xrightarrow{a} t$ $R = a$



Beobachtung. $\# \text{ Zustände} \leq 2|R|$, denn in jedem Schritt werden höchstens zwei neue Knoten zum Graphen hinzugefügt.

Beispiel. $R = (d + o + g)((n + o)w)^*(c + l + \epsilon)(c + l)$



3.2 Der Algorithmus

Aufgabe: Sei T ein Text und R ein regulärer Ausdruck. Wir suchen die Vorkommen von R in T , anders ausgedrückt: Gibt es k, j , so dass $T[k, j]$ entsprechend R gebildet ist?

Dazu bilden wir den regulären Ausdruck Σ^*R und testen für alle i , ob Σ^*R den Präfix $T[1..i]$ matcht. Das wiederum geht wie folgt.

Definition 9 Sei G der gerichtete Graph zu Σ^*R , dann definieren wir die $N(i)$ -Nachbarschaft von s in G induktiv:

- $N(0)$ = Alle Zustände, die von s auf ϵ -Wegen erreichbar sind.
- $N(i)$ = Alle Zustände, die von $N(i-1)$ mit $T(i)$ - oder ϵ -Übergängen zu erreichen sind.

Damit gilt: $T[1, i]$ matcht $R \iff t \in N(i)$.

Satz 2 (ohne Beweis) Sei T , $|T| = m$, ein Text und R ein regulärer Ausdruck mit n Zeichen. Dann kann man in Zeit $O(n \cdot m)$ testen, ob ein Teilstring von T den Ausdruck R matcht.

4 Seminumerisches Stringmatching

Bisher haben wir Zeichen für Zeichen Text und Pattern miteinander verglichen, nun wenden wir uns Algorithmen zu, in denen wir uns der Algebra, Zahlentheorie, ... bedienen.

4.1 Shift-AND-Methode (nach Baeza-Yates, Gonnet '92)

Diese Methode benutzt Bitoperationen und ist besonders effektiv für kurze Muster.

4.1.1 Exaktes Matching

Ein exaktes Matching zwischen einem Text T , $|T| = m$ und einem Muster P , $|P| = n$ geht wie folgt:

Definition 10 Wir definieren eine $n \times m$ -Matrix M , die nur Nullen und Einsen als Einträge enthält.

$$M_{ij} = \begin{cases} 1 & , \text{ wenn } P[1, \dots, i] \text{ } T[j - i + 1, \dots, j] \text{ matcht} \\ 0 & , \text{ sonst} \end{cases}$$

Wenn der Eintrag $M_{nj} = 1$ ist, dann haben wir ein Matching von P an der Position j endend. Diese Matrix läßt sich folgendermaßen aufbauen:

Definition 11 Sei $x \in \Sigma$. Zu x definiere den Vektor $U(x) \in \{0, 1\}^n$ durch $U(x)_i = 1 \iff P(i) = x$, also $U(x) = (0, \dots, 0, \underbrace{1}_{\text{Position } i}, 0, \dots, 0)$

und Vektor $\text{Bitshift}(M(j)) = \text{Spalte } j \text{ von } M \text{ um eine Position nach unten verschoben und eine } 1 \text{ an die oberste Position gesetzt. } \text{Bitshift}(M(j)) \in \{0, 1\}^n$.

Beobachtung. $M_{ij} = 1 \iff M_{i-1, j-1} = 1$ und $U(T(j))_i = 1$.

Im Algorithmus gehen wir bei der Berechnung von M induktiv vor.

1. Wir initialisieren die 1. Spalte $M(1)$:

$$M_{11} = \begin{cases} 1 & : \text{ wenn } T(1) = P(1), \\ 0 & : \text{ sonst.} \end{cases}$$

$$\forall i \in \{2..n\} \quad M_{i1} = 0.$$

2. Sei $M(j - 1)$ berechnet.
3. Setze $M(j) = \text{Bitshift}(M(j - 1)) \text{ AND } U(T(j))$.

Beispiel. Sei $P = abaac$, $T = xabxabaaxa$. Damit ist $T(7) = a$, und

$$M(7) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \text{ und } U(a) = \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}.$$

Nun bestimmen wir $M(8) \in \{0, 1\}^n$ mit Hilfe der Definition...

$$\left. \begin{array}{l} P[1] = T[8] \Rightarrow M_{18} = 1. \\ P[1, 2] \neq T[7, 8] \Rightarrow M_{28} = 0. \\ P[1, 3] = T[6, 8] \Rightarrow M_{38} = 0. \\ P[1] = T[5, 8] \Rightarrow M_{48} = 1. \\ P[1] \neq T[4, 8] \Rightarrow M_{58} = 0. \end{array} \right\} \Rightarrow M(8) = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

... und nach Induktionsvorschrift:

$$M(8) = \begin{pmatrix} 1 \\ 1 \\ 0 \\ 1 \\ 0 \end{pmatrix} \text{ AND } \begin{pmatrix} 1 \\ 0 \\ 1 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}$$

Analyse.

- Es fallen $O(n \cdot m)$ Bitoperationen an. Für kleine n ist der Algorithmus sehr effizient, da die Bitoperationen im Rechner sehr schnell ablaufen.
- Der Speicherverbrauch ist gering, da immer nur 2 Spalten aktiv sind.

4.1.2 Matching mit Fehlern (Wu, Manber '92)

Wir verallgemeinern nun den Algorithmus, indem wir bis zu k Fehler zulassen. Dabei können Fehler Mismatches, Insertions oder Deletions sein. Der Unix-Befehl `agrep` arbeitet mit diesem Algorithmus.

Wir wenden uns nur den Mismatches zu. Dazu definieren wir uns $k + 1$ $(n \times m)$ -Matrizen M^0, M^1, \dots, M^k .

Definition 12

$$M_{ij}^l = \begin{cases} 1 & , \text{ wenn } P[1, \dots, i] \text{ matcht } T[j - i + 1, \dots, j] \text{ mit } \leq l \text{ Fehlern.} \\ 0 & , \text{ sonst.} \end{cases}$$

Beobachtung. Bei der induktiven Berechnung von $M^l(j)$ können 3 Fälle auftreten:

1. Es liegen maximal $l - 1$ Fehler bis zur Position j vor $\rightarrow M^{l-1}(j)$.
2. Es liegen bis zur Position $j - 1$ schon l Fehler vor, und $P(i) = T(j)$.
3. Es liegen bis zur Position $j - 1$ erst $l - 1$ Fehler vor, daher ist es egal, ob das letzte Zeichen richtig ist.

Daraus erhalten wir die Formel

$$M^l(j) = M^{l-1}(j) \text{ OR } (\text{Bitshift}(M^l(j - 1)) \text{ AND } U(T(j))) \text{ OR } M^{l-1}(j-1) .$$

Es ergibt sich eine Laufzeit von $O(k \cdot m \cdot n)$.

4.2 Fingerprint-Methode nach Karp-Rabin '87

Wir betrachten das exakte Matching-Problem. Dabei werden arithmetische Operationen an Stelle von Vergleichen durchgeführt. Das Verfahren ist randomisiert, und liefert alle Matches sowie möglicherweise einige falsche Matches. Wir werden sehen, daß es möglich ist, die Wahrscheinlichkeit für ein falsches Match sehr klein zu halten.

In diesem Abschnitt schränken wir uns auf das Alphabet $\Sigma = \{0, 1\}$ ein, die Methode kann aber auch so modifiziert werden, daß sie für größere Alphabete arbeitet.

4.2.1 Algorithmus

Definition 13 Sei T , $|T| = m$, ein Text und P , $|P| = n$, ein Pattern. Dann führen wir folgende Abbildung in die natürlichen Zahlen ein.

$$P \mapsto H(P) := \sum_{i=1}^n 2^{n-i} \cdot P(i)$$

$$T_r \mapsto H(T_r) = \sum_{i=1}^n 2^{n-i} \cdot T_r(i),$$

wobei T_r das bei r beginnende Teilwort von T bezeichnet (Also gilt $T_r(i) = T(r+i-1)$).

Eine Verallgemeinerung des Algorithmus auf Alphabete mit s Zeichen erhält man, indem man in der Abbildung Potenzen von s betrachtet.

Beobachtung. P startet in T an der Position $r \iff H(P) = H(T_r)$, denn jede natürliche Zahl hat eine eindeutige Darstellung als Summe von 2-er Potenzen.

Die Zahlen, die dabei auftreten, können sehr groß sein. Um dies zu vermeiden, rechnen wir modulo einer zufällig gewählten Primzahl p . Dabei können allerdings falsche Matches entstehen, wenn p ein Teiler von $|H(P) - H(T_r)|$ ist.

Auf der einen Seite wollen wir p möglichst klein wählen, um die Ergebnisse klein zu halten, andererseits werden dadurch falsche Matches wahrscheinlicher. Wir verwenden eine zufällig gewählte Primzahl, und es wird gezeigt, daß die Wahrscheinlichkeit für ein dadurch entstehendes falsches Match klein ist.

$$H_p(P) := H(P) \bmod p$$

Bei der Berechnung von $H_p(P)$ gehen wir nach dem Hornerschema vor, so daß $O(n)$ Rechenoperationen anfallen.

$$H_p(P) = [[\dots [P(1) \cdot 2 \bmod p + P(2)] \cdot 2 \bmod p + \dots + P(n-1)] \cdot 2 \bmod p + P(n)] \bmod p$$

$H_p(T_1)$ läßt sich ebenso mit $O(n)$ Rechenoperationen berechnen. Wie wir in der Übung gesehen haben, läßt sich $H_p(T_r)$ aus $H_p(T_{r-1})$ mit konstantem Aufwand berechnen.

Bevor wir eine Aussage über die Wahrscheinlichkeit von falschen Matches machen, zuerst ein kleiner Ausflug in die Zahlentheorie.

4.2.2 Einiges über Primzahlen

Definition 14 Sei n eine beliebige natürliche Zahl, dann sei

$$\pi(n) := \text{Anzahl der Primzahlen } \leq n.$$

Dazu nun ein Satz, ein Lemma und ein Korollar.

Satz 3 (ohne Beweis) Sei $n \in \mathbb{N}$. Dann gilt folgende Ungleichung:

$$\frac{n}{\ln n} \leq \pi(n) \leq 1.26 \frac{n}{\ln n}$$

Lemma 1 Für $n \geq 29$ gilt:

$$\prod_{p \leq n, p \text{ prim}} p > 2^n$$

Korollar 2 Sei $n \geq 29$ und $x \leq 2^n$. Dann hat x weniger als $\pi(n)$ verschiedene Primteiler.

Beweis. (indirekt)

Wir nehmen an, x habe mehr als $\pi(n)$ verschiedene Primteiler q_1, q_2, \dots, q_k , und gleichzeitig $2^n \geq x$. Dann gilt:

$$x \geq \prod_{i=1}^k q_i \geq \prod(\text{erste } k \text{ Primzahlen}) > \prod(\text{erste } \pi(n) \text{ Primzahlen}) > 2^n.$$

Widerspruch: $2^n > 2^n$. □

4.2.3 Die Wahrscheinlichkeit für ein falsches Match

Satz 4 Sei P , $|P| = n$, ein Pattern, T , $|T| = m$, ein Text und $n \cdot m \geq 29$. Für $i \in \mathbb{N}$ gilt nun: Falls p eine zufällige Primzahl $\leq i$ ist, so ist die Wahrscheinlichkeit eines falschen Matches zwischen P und $T \leq \frac{\pi(n \cdot m)}{\pi(i)}$.

Beweis. Sei R die Menge der Positionen von T , an denen P nicht beginnt:

$$s \in R \iff H(P) \neq H(T_s).$$

Dann ist $|R| \leq |T| = m$, und es gilt $\prod_{s \in R} \underbrace{|H(P) - H(T_s)|}_{\leq 2^n} \leq 2^{n \cdot m}$.

Mit dem Korollar 2 erhalten wir:

$$\prod_{s \in R} |H(P) - H(T_s)| \text{ hat } \leq \pi(n \cdot m) \text{ verschiedene Primteiler.}$$

Zu einem falschen Match an der Stelle s im Text kommt es, wenn $H(P) = H(T_s) \pmod p$ ist, also wenn p die Zahl $|H(P) - H(T_s)|$ teilt. In diesem Fall

teilt p natürlich das obige Produkt, also gibt es ein falsches Match im Text dann, wenn p einer der $\leq \pi(n \cdot m)$ Primteiler des Produktes ist, und die Wahrscheinlichkeit Pr dafür beträgt

$$Pr \leq \frac{\pi(n \cdot m)}{\pi(i)}. \quad \square$$

Beispiel. Sei $n = 250$, $m = 4000$ und $i = n \cdot m^2$. Dann erhalten wir: $Pr(\text{falsches Match}) < 10^{-3}$ und $i < 2^{32}$. Die Zahlen bleiben damit für einen Computer klein genug, und die Fehlerwahrscheinlichkeit ist auch relativ klein.

5 Suffix-Bäume und ihre Anwendungen

Die Idee der Suffixbäume wurde erstmals von Weiner '73 ausgeführt. Sie hat allerdings erst '95 wirklich Beachtung gewonnen, als Ukkonen einen einfachen Linearzeitalgorithmus zu ihrer Konstruktion gefunden hat. Dazu später mehr.

Der herausragende Vorteil bei der Benutzung von Suffix-Bäumen gegenüber den nun schon bekannten Algorithmen zum String-Matching ist, daß der *Text* in linearer Zeit vorverarbeitet wird, und nachfolgend jede Anfrage für beliebige Pattern der Länge m in Zeit $O(m)$ beantwortet werden kann.

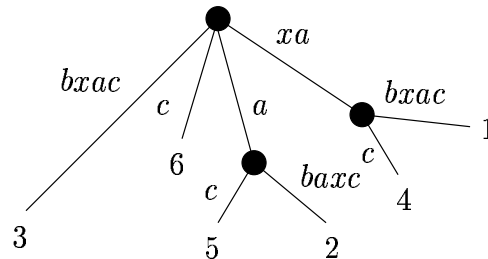
Definition 15 Sei S , $|S| = m$, ein String. Dann nennen wir einen gerichteten Baum T mit Wurzel und Blättern $1, \dots, m$ den Suffixbaum zu S , wenn folgende Eigenschaften erfüllt sind:

- Die Kanten sind mit Teilstrings markiert.
- Innere Knoten haben Ausgrad ≥ 2 .
- Keine 2 Kanten aus einem Knoten starten mit dem selben Buchstaben.
- Die Kantenlabel auf dem Weg von der Wurzel zu Blatt i entsprechen dem Suffix $S[i, m]$.

Bemerkung. Um zu vermeiden, dass ein Suffix von S ein echter Teilstring ist, fügen wir am Ende von S das Sonderzeichen $\$$ an. Damit wird erreicht, dass wirklich jeder Suffix von S einem Blatt von \mathcal{T} entspricht. Konstruiert man den Suffixbaum, indem von hinten startend immer wieder losläuft, so erhält man für seine Konstruktion eine Laufzeit von $O(m^2)$.

Man kann auch einen verallgemeinerten Suffix-Baum für mehrere Strings konstruieren. Dabei muß in den Blättern zusätzlich die Information, aus welchem String der Suffix, der hier endet, kommt, gespeichert werden.

Beispiel. $S = xabxac$.



5.1 Anwendungen

Zur Motivation nun ein paar Anwendungsbeispiele von Suffixbäumen.

5.1.1 Exaktes Matching

Sei P , $|P| = n$, ein Pattern und T , $|T| = m$ ein Text. Wir erstellen den Suffixbaum \mathcal{T} für T in $O(m)$ Zeit. Dann wird P mit \mathcal{T} verglichen, d.h., das Muster wird im Suffixbaum bei der Wurzel beginnend „abgelaufen“. Dabei fallen $O(n)$ Vergleiche an. Die Anfangsstellen von Matches von P in T entsprechen den Blättern unter einer Position im Baum mit Label P . Diese Blätter finden wir in $O(k)$ Zeit, wobei k der Anzahl der Matches entspricht. Alles in allem läuft exaktes Matching in $O(n + m + k)$. Der Baum ist hierbei unabhängig von P , und muß damit nur ein Mal erstellt werden. Bei erneutem Matching mit einem anderen Pattern ist die Laufzeit somit nur $O(n + k)$.

5.1.2 Eine Datenbank für Muster

Sei eine Datenbank von Mustern P_i , $\sum |P_i| = m$, und ein String S der Länge n gegeben. Wir suchen alle Muster, die S als Teilstring enthalten. Nachdem der verallgemeinerte, gemeinsame Suffixbaum für die Muster *ein Mal* in Zeit $O(m)$ konstruiert wurde, kann eine Anfrage für ein beliebiges S in Zeit $O(n + k)$ bearbeitet werden: Der String S wird im Suffixbaum bei der Wurzel beginnend abgelaufen, und alle Blätter unter S gehören zu Mustern, die S als Teilstring enthalten.

5.1.3 Longest-Common-Substring von S_1 und S_2

Wir konstruieren den gemeinsamen Suffixbaum von S_1 und S_2 und markieren die inneren Knoten mit $i \in \{1, 2\}$, falls der Unterbaum ein mit i markiertes Blatt enthält. Um den LCS zu finden, suchen wir den tiefsten Knoten mit *beiden* Marken.

Bei der Verallgemeinerung auf S_1, S_2, \dots, S_K mit Gesamtlänge n , wollen wir für alle $2 \leq k \leq K$ die Länge des längsten Teilstring, der in $\geq k$ Strings vorkommt, berechnen. Dazu definieren wir für alle Knoten v des Baumes die Abbildung

$$v \mapsto C(v) := \text{Anzahl verschiedener Blattlabels unter } v.$$

Gesucht sind dann die Knoten v mit $C(v) \geq k$. Die Abbildung bestimmen wir induktiv:

1. Wir fangen in den Blättern an. Deren Eltern wird ein (0-1)-Vektor w der Länge K zugeordnet mit $w(i) = 1 \iff$ Ein Blatt hat Marke i , und sonst $w(i) = 0$.
2. Sei v ein Knoten, dann ist

$$w(v) = \bigvee_{c \text{ Kinder}} w(c) \quad \text{und} \quad C(v) = \sum_{i=1}^K w(i).$$

Nun muß noch für $2 \leq k \leq K$ ein tiefster Knoten mit k verschiedenen Blattlabels berechnet werden, und die Ergebnisse von K auf $K-1, K-2, \dots$ akkumuliert werden (es muß ja nicht für jedes k ein Knoten mit $C(v) = k$ existieren).

Insgesamt fallen dabei $O(K \cdot n)$ Operationen an: $O(n)$ für die Konstruktion des Suffixbaumes, und $O(K \cdot n)$, um für jeden der $O(n)$ Knoten den Vektor der Länge K zu bestimmen. Das Suchen der tiefsten Knoten und akkumulieren der Werte geht ebenfalls in $O(K \cdot n)$ Zeit.

Später werden wir eine Lösung kennenlernen, die nur lineare Zeit benötigt.

5.1.4 All-Pairs-Suffix-Präfix-Matching

Gegeben sind Strings S_1, \dots, S_k mit Gesamtlänge n , gesucht ist für alle Paare (i, j) der längste Suffix von S_i , der einen Präfix von S_j matcht. Die Anzahl der Ergebnisse ist also k^2 . Dieses Problem tritt beim Suchen eines möglichst

kurzen Strings S auf, der alle S_i als Teilstrings enthält. Insbesondere ist das für biologische Anwendungen relevant.

Für die Strings wird ein gemeinsamer Suffix-Baum konstruiert. Dann speichern wir zu jedem inneren Knoten v eine Liste der Strings, für die bei v eine Terminalkante (Kante nur mit dem Terminalzeichen $\$$) abgeht. Wenn bei v eine Terminalkante von S_i abgeht heißt das, dass in v ein Suffix von S_i endet.

Um nun die längsten Suffix-Präfix-Matches zu bestimmen, wird jeder String S_j im Baum abgelaufen. Dabei wird für jedes i der tiefste Knoten v auf dem Pfad ausgegeben, bei dem i in der Liste der Terminalkanten steht.

Die Laufzeit beträgt $O(n)$ für die Konstruktion des Baumes, und $O(k^2)$ für das Finden der Matches, insgesamt also $O(n + k^2)$.

5.2 Konstruktion von Suffixbäumen in Linearzeit

Sei im folgenden S ein String mit m Zeichen und Σ ein endliches Alphabet. Bei der Konstruktion folgen wir dem Artikel von Ukkonen von '95.

Grundidee des Algorithmus:

- Wir konstruieren Suffixbäume für immer länger werdende Präfixe $S[1, i]$, $i = 1, \dots, m$.
- Wir konstruieren zunächst *implizite* Suffixbäume.

Definition 16 *Der implizite Suffixbaum (SB) von S entsteht aus dem SB von $S\$$ durch folgende Regeln:*

- *Streiche $\$$ aus den Kantenlabels.*
- *Entferne Kanten ohne Label.*
- *Entferne innere Knoten mit genau einem Kind.*

Wir bezeichnen mit \mathcal{T}_i den impliziten Suffixbaum zu $S[1, i]$.

Ukkonen's Algorithmus(high-level)

Erzeuge \mathcal{T}_1 .

for $i = 1, \dots, m - 1$

 for $j = 1, \dots, i + 1$

 Finde Ende des Weges $S[j, i]$ in \mathcal{T}_i und
 falls notwendig, füge $S(i + 1)$ an.

 end for

end for

} Phase $(i + 1)$

Beispiel. $S\$ = xabxa\$$.

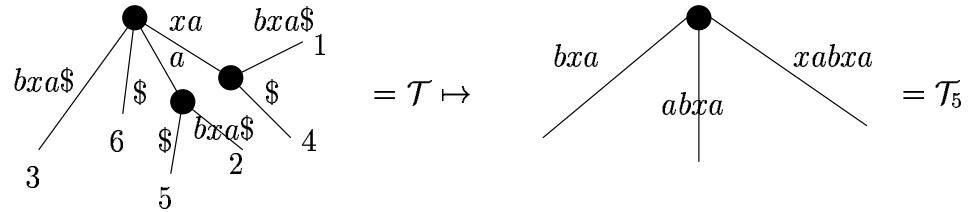


Abbildung 17: Suffixbaum und impliziter Suffixbaum von S .

In der Phase $(i + 1)$ werden die Suffixe $S[1, i + 1], S[2, i + 1], \dots, S[i, i + 1], S[i + 1, i + 1]$ des Präfixes $S[1, i + 1]$ in \mathcal{T}_i eingefügt. Dabei werden die folgenden Erweiterungsregeln benutzt:

Es sei $S(i + 1) = b$, und wir sind aktuell in der j -ten Erweiterung.

Regel 1: Der Weg endet in einem Blatt, dann erweitere das letzte Kantenlabel um b .

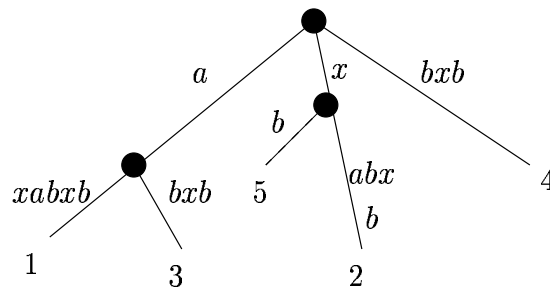
Regel 2: Der Weg endet „im Baum“.

1.Fall: In innerem Knoten: Kante mit Label b zu Blatt j einfügen.

2.Fall: Auf Kante: Neuen inneren Knoten und Kante mit Label b erzeugen.

Regel 3: $S[j, i] \cdot b$ gibt es schon, dann tue nichts.

Beispiel. Sei der implizite Suffixbaum \mathcal{T}_5 für den Teilstring $axabx$ schon konstruiert, und das 6. Zeichen sei b . Wir betrachten nun die Erweiterung von \mathcal{T}_5 nach \mathcal{T}_6 . An den Kanten zu den Blättern 1, 3, 2 und 4 wird Regel 1 angewendet und ein b angehängt. An der Kante zu Blatt 5 wird Regel 2 angewendet und ein neuer innerer Knoten geschaffen. An der Kante zu Blatt 4 wird für den Suffix b Regel 3 angewendet: b ist schon da, tue nichts.



Implementiert man die high-level-Version, dann fallen in der Phase $(i + 1)$ $O(i^2)$ Operationen an, und insgesamt erhält man $O(m^3)$ als Laufzeitverhalten. Nun folgen einige Verbesserungen, die schließlich zu einem Linearzeitalgorithmus führen.

5.2.1 Erster Trick: Suffix-Links

Als erste Verbesserung führen wir, ähnlich wie die Fehler-Links im Aho-Corasick Algorithmus, Suffix-Links ein, um Sprünge in der Phase $(i + 1)$ zu ermöglichen. Diese Sprünge werden es erlauben, die Enden der $S[j, i]$ schneller zu erreichen.

Definition 17 Sei $x\alpha$ ein String, $x \in \Sigma$. Wenn es für einen inneren Knoten v mit Label $x\alpha$ einen anderen inneren Knoten $s(v)$ mit Label α gibt, so heißt $(v, s(v))$ Suffix-Link. Wenn α der leere String ist, setzen wir einen Suffix-Link zur Wurzel.

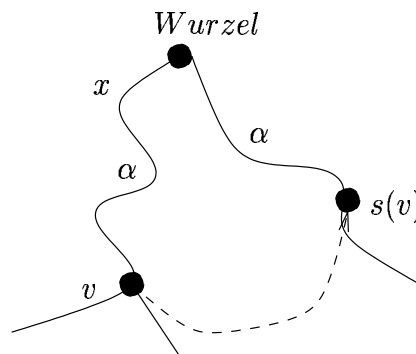
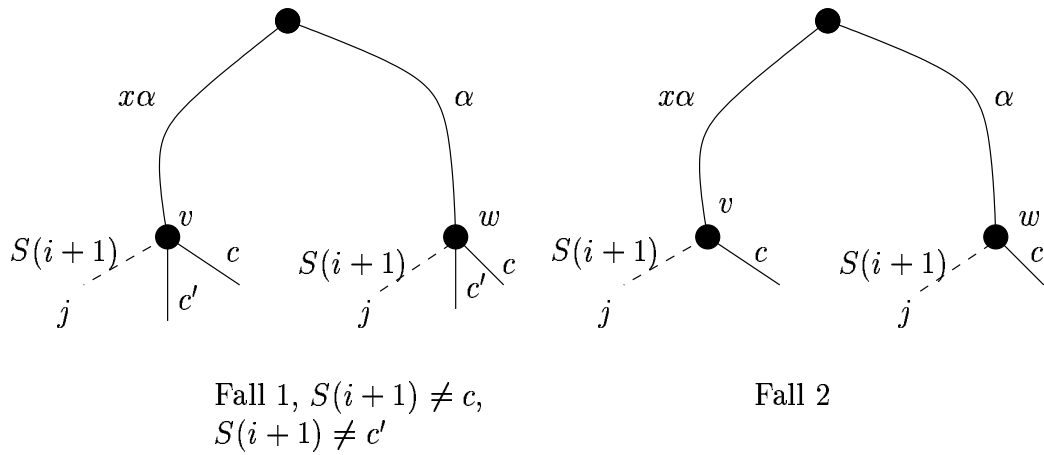


Abbildung 18: Suffix-Link

Lemma 2 Falls eine neue Kante mit Label b am inneren Knoten v mit Label $x\alpha$ in der j -ten Erweiterung der Phase $(i + 1)$ entsteht, dann gibt es bereits einen inneren Knoten w mit Label α , oder w entsteht in der $(j + 1)$ -ten Erweiterung, nämlich dann, wenn v selbst erst in der j -ten Erweiterung entstanden ist.

Beweis. Eine neue Kante entsteht, d.h. wir haben Regel 2 angewendet.

Fall 1: v ist schon vor der j -ten Erweiterung innerer Knoten, und es gehen somit mindestens zwei Kanten mit Zeichen c bzw. c' beginnend bei v los,



wobei $c \neq c'$. Da α das Suffix $S[2, l]$ eines Präfixes $S[1, l]$, wobei $l < i$, ist, gibt es von der Wurzel einen Weg zu einem Knoten w mit Label α und ≥ 2 ausgehenden Kanten mit Label c bzw. c' .

Fall 2: v entsteht in der j -ten Erweiterung der Phase $(i+1)$, und es gehen zwei Kanten mit c bzw. $S(i+1)$ beginnend aus. In der $(j+1)$ -ten Erweiterung wird nun α um $S(i+1)$ erweitert, und der Knoten w entsteht analog zu v . □

Korollar 3 *In einem impliziten Suffixbaum gibt es am Ende jeder Phase für alle inneren Knoten Suffix-Links.*

Was hilft uns das nun weiter? Es spart jede Menge Wege von der Wurzel abwärts! Dazu die folgende

Beobachtung. Die erste Erweiterung in jeder Phase $(i+1)$ ist trivial: $S[1, i]$ endet in \mathcal{T}_i immer in einem Blatt. Auf dieses Blatt halten wir einen Pointer fest. Von Erweiterung j kommen wir nun leicht zur $(j+1)$ -ten Erweiterung: Sei $S[j, i] = x\alpha$, dabei $x \in \Sigma$, und sei (v, j) die Kante zu Blatt j . Gesucht ist für die $(j+1)$ -te Erweiterung ein Knoten mit Label α . Es können zwei Fälle auftreten:

1. v ist die Wurzel. Dann laufe α von der Wurzel aus.
2. v ist ein innerer Knoten. Dann gibt es einen Suffix-Link $(v, s(v))$, und $s(v)$ hat ein Label, das ein Präfix von α ist. Nun müssen wir von $s(v)$ aus nur noch den „Rest“ laufen.

Das alleine verbessert die Laufzeit im worst-case noch nicht, dafür werden weitere Überlegungen benötigt.

5.2.2 Zweite Verbesserung: Skip & Count

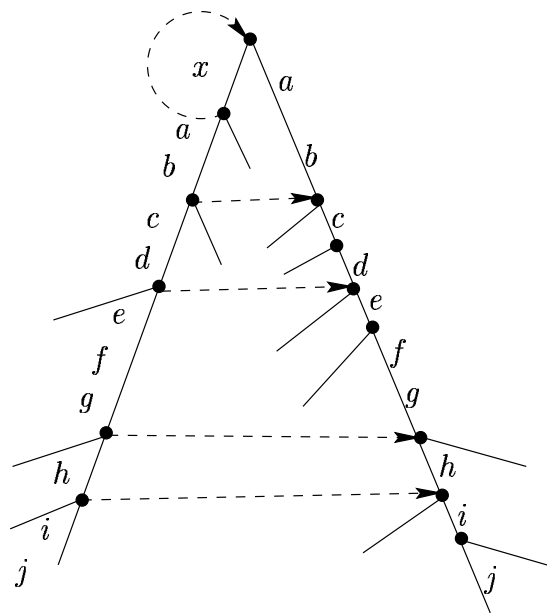
Hinter dem Skip & Count Trick verbirgt sich folgendes: Die Kantenlabels werden nicht explizit an den Kanten gespeichert, sondern nur ihre Anfangs- und Endpositionen im String. Beim Herunterlaufen von $s(v)$ aus wird nun nicht jeder Buchstabe verglichen, sondern nur der erste Buchstabe einer Kante. Da alle Kanten von einem Knoten aus mit unterschiedlichen Buchstaben beginnen, reicht das aus. Speichert man zu jeder Kante ihre Länge, so kann man nach einem Vergleich die restlichen Buchstaben dieser Kante überspringen.

Definition 18 Sei v Knoten in einem Suffixbaum. Dann nennen wir die Anzahl der Kanten auf dem Weg von v zur Wurzel Knotentiefe von v .

Lemma 3 Sei $(v, s(v))$ Suffix-Link. Dann gilt:

$$\text{Knotentiefe}(v) \leq \text{Knotentiefe}(s(v)) + 1$$

Beweis. Zuerst ein Bild:



Sei $(v, s(v))$ ein Suffix-Link. Dann ist nach Definition das Label $L(v) = x\alpha$, wobei $x \in \Sigma$ Zeichen und $\alpha \in \Sigma^*$ Teilstring, und $L(s(v)) = \alpha$. Sei w der Knoten mit $\text{Knotentiefe}(w) = 1$ auf dem Weg von der Wurzel r nach v ,

und die Kante (w, r) nur mit einem Zeichen markiert. Dann ist (w, r) ein Suffix-Link. Sei nun u ein beliebiger Knoten, $u \neq w$, auf dem Weg von r nach v . Dann ist u ein innerer Knoten, und es existiert somit ein Suffix-Link $(u, s(u))$. Der Knoten $s(u)$ ist dann ein Knoten auf dem Weg von r nach $s(v)$. Sei nun u' ein weiterer Knoten auf dem Weg von r nach v mit $u \neq u'$, dann gilt weiter $s(u) \neq s(u')$. Nach diesen Vorüberlegungen nun zum Beweis. Mit Ausnahme des ersten Knotens auf dem Weg von r nach v existiert *immer* ein Knoten auf dem Weg von r nach $s(v)$, und diese sind paarweise verschieden. Für den ersten Knoten auf dem Weg von r nach v gibt es *keinen* Suffix-Link (und damit keinen korrespondierenden Knoten auf dem Weg von r nach $s(v)$) genau dann, wenn die Kante von r dorthin nur ein Zeichen trägt. \square

Satz 5 *Mit Skip & Count kann jede Phase $(i+1)$ in $O(m)$ realisiert werden.*

Beweis. Zur Erinnerung noch einmal die Vorgehensweise bei Skip & Count: Wir befinden uns während einer Phase $(i+1)$ nach der j -ten Erweiterung irgendwo im Baum, und wollen nun an $S[j+1, i]$ das Zeichen $S(i+1)$ anhängen. Wir suchen also von $S[j, i+1]$ aus das Ende von $S[j+1, i]$. Dazu gehen wir im Baum nach oben zum nächsten Knoten v (auch up-Schritt genannt), und folgen dem Suffix-Link $(v, s(v))$. Diese beiden Schritte laufen in konstanter Zeit ab und sind daher nicht für die Laufzeitanalyse relevant. Bei $s(v)$ vergleichen wir das erste Zeichen mit den ausgehenden Kanten und springen dann bis zum nächsten Knoten (down-skip).

Nun analysieren wir, wie viele down-skips benötigt werden. Dazu beobachten wir den Verlauf der aktuellen Knotentiefe während einer Erweiterung und schließlich einer Phase. Wir wissen, dass die aktuelle Knotentiefe bei jeder Erweiterung maximal m ist, da der Baum maximale Tiefe m hat.

In jeder Erweiterung passiert folgendes: Wir gehen im up-Schritt im Baum ≤ 1 Schritte Richtung Wurzel, und beim Folgen des Suffix-Links wird der Abstand zur Wurzel nach dem vorherigen Lemma um ≤ 1 geringer. Insgesamt gehen wir also maximal 2 Schritte Richtung Wurzel, d.h. die aktuelle Knotentiefe verringert sich um höchstens 2. Über alle Erweiterungen 1 bis $i+1$ verringert sich damit die aktuellen Knotentiefe um $\leq 2 \cdot m$, da $(i+1) \leq m$.

Ohne die up-Schritte und Suffixlinks könnten wir maximal bis zur Tiefe m hinunterspringen. Nun springen wir zusätzlich um $\leq 2 \cdot m$ nach oben, d.h. wir können insgesamt nur $\leq 3 \cdot m$ down-skips machen. Damit ergibt sich für jede Phase eine Laufzeit von $O(m)$. \square

Die resultierende Gesamtlaufzeit ist an dieser Stelle immer noch quadratisch, doch mit der nächsten Beobachtung und einem weiteren kleinen Trick läßt sie sich zur versprochenen Linearzeit reduzieren.

Beobachtung.

1. Wenn Regel 3 das erste Mal angewendet wird, dann ist die Phase beendet, da danach nur noch Regel 3 vorkommt.
2. „Einmal Blatt, immer Blatt“. Wenn ein Blatt einmal erzeugt worden ist, so bleibt es immer ein Blatt. Alle folgenden Erweiterungen an dieser Stelle gehen somit gemäß Regel 1.

In der Phase $(i + 1)$ merken wir uns den größten Index j_{i+1} , für den gilt:

$\forall 1 \leq k \leq j_{i+1}$: Die k -te Erweiterung erfolgte nach Regel 1 oder 2, d.h. alle Suffixe $S[1, i + 1] \dots S[j_{i+1}, i + 1]$ gehören zu Blättern.

Bei der Implementierung gehen wir nach Beobachtung 2 beim Einfügen einer neuen Kante bzw. Blattes, wie folgt vor:

Für jede Kante, die zu einem Blatt führt, speichern wir das Paar (p, e) . Dabei ist p die Anfangsposition des entsprechenden Suffix in S und e ein *globaler* Index, der nur ein Mal pro Phase von i auf $i + 1$ erhöht wird. Damit wird mit *konstantem* Aufwand an *allen* Blättern gleichzeitig das neue Zeichen $S(i + 1)$ nach Regel 1 angefügt.

Diese Überlegungen verwenden wir nun, um den Algorithmus in der Phase $(i + 1)$ zu verbessern.

Phase $(i + 1)$:

1. Setze e auf $(i + 1)$. Damit werden alle Blätter nach Regel 1 erledigt.
2. Berechne die Erweiterungen $j_i + 1$ bis j^* explizit, wobei j^* das erste Anwenden von Regel 3 markiert.
3. Setze j_{i+1} auf $j^* - 1$.

Bemerkung. In der Phase $(i + 2)$ fangen wir mit den expliziten Vergleichen also bei der Erweiterung j^* an.

Satz 6 $\mathcal{T}_1, \mathcal{T}_2, \dots, \mathcal{T}_m$ können in $O(m)$ konstruiert werden.

Beweis. In jeder Phase wird für die impliziten Erweiterungen nach Regel 1 nur konstante Zeit benötigt, siehe oben. Über alle m Phasen summiert sich das also zu Zeit $O(m)$.

Zwei aufeinander folgenden Phasen haben ≤ 1 explizite Vergleiche gemeinsam, nämlich bei j^* . Die aktuelle Knotentiefe am Ende der expliziten Vergleiche einer Phase ist gleich der aktuellen Knotentiefe bei Beginn der expliziten Vergleiche der darauffolgenden Phase. Durch Betrachten der aktuellen Knotentiefe (die durch m beschränkt ist) im Verlauf des Algorithmus, und Argumente ähnlich denen im Beweis von Satz 5, kommt man auf insgesamt $O(m)$ explizite Erweiterungen während des gesamten Algorithmus. \square

Korollar 4 *Der Suffixbaum für $S\$$ läßt sich in $O(m)$ konstruieren.*

Beweis. Der implizite Suffixbaum wird in linearer Zeit konstruiert, und das Einfügen von $\$$ als letztem Zeichen geht ebenfalls in Linearzeit. \square

5.3 Suffix-Arrays und Anwendungen

Suffix-Arrays sind eine speichereffiziente Variante von Suffixbäumen, die besonders bei großen Alphabeten Verwendung findet.

Wir speichern dazu in einem Array Pos der Länge m die Anfangspositionen der Suffixes von S in lexikographischer Ordnung $<$, wobei gilt $\forall a \in \Sigma, \$ < a$.

Beispiel. $S = mississippi, Pos = 11, 8, 5, 2, 1, 10, 9, 7, 4, 6, 3$

Fakt. Das Suffix-Array erhält man aus dem Suffix-Baum durch lexikographische Tiefensuche in linearer Zeit.

Durch binäre Suche kann man exaktes Matching in Suffix-Arrays für ein Pattern P , $|P| = n$ und einen Text der Länge m in Zeit $O(n \cdot \log m)$ durchführen. Die Idee dazu ist:

Es bezeichnen L und R den linken und rechten Rand des aktuellen Suchintervalls in Pos , zu Anfang also $L = 1, R = m$. In jeder Iteration der binären Suche wird P mit dem Suffix $Pos(M)$, wobei $M = \lceil (L + R)/2 \rceil$ ist, verglichen. Je nach Ergebnis wird dann L oder R auf M gesetzt. Es reicht, in Pos den ersten und den letzten Suffix zu finden, der mit P anfängt. Wegen der lexikographischen Ordnung in Pos sind alle anderen Vorkommen von P in T genau die, die in Pos zwischen diesen beiden stehen.

In jeder Iteration gibt es $O(n)$ Vergleiche, und insgesamt $O(\log m)$ Iterationen, was zu der oben angegebenen Laufzeit führt. Durch einige Tricks kann man das verbessern und eine Laufzeit von $O(n + \log m)$ erreichen (\rightarrow Übung).