6 Suffixbäume und *lca*–Anfragen

Definition 19 *Sei $\mathcal{T}$ ein Baum, r seine Wurzel und v, u Knoten. Dann definieren wir den* lowest common ancestor*:*

$lca(u,v)$ = tiefster Knoten, der auf dem Weg von v nach r und auf dem Weg von u nach r liegt.

Nehmen wir nun an, wir könnten nach einer Vorverarbeitung von $\mathcal{T}$ in Linearzeit, die *lca*–Anfragen in konstanter Zeit beantworten. Dann könnten wir folgende Probleme effizient lösen.

Beispiel 1 Longest–Common–Extension:

Seien S_1, S_2 Strings. Zu einem gegebenen Paar (i,j) suchen wir nun das maximale k, so dass $S_1[i,i+k] = S_2[j,j+k]$. Dazu betrachten wir den gemeinsamen Suffixbaum von S_1 und S_2 und bestimmen den $lca(\text{Blatt}(1,i), \text{Blatt}(2,j))$. Dabei bezeichnet $\text{Blatt}(l,i)$, $l = 1,2,\ldots$, das Blatt, das im gemeinsamen Suffixbaum von S_1 und S_2 zum Suffix $S_l[i,n_l]$ gehört.

Das Konstruieren des gemeinsamen Suffixbaumes geht in linearer Zeit, und nach Annahme können wir nach ebenfalls linearer Vorverarbeitung die *lca*–Anfragen in konstanter Zeit beantworten.

Beispiel 2 Maximale Palindrome

Wir suchen zu einem vorgegebenen q in einem String S das maximale gerade Palindrom mit Mitte zwischen Position q und $q+1$. Palindrome sind Worte $\alpha \in \Sigma^*$, mit $\alpha = \alpha^{rev}$, und gerade soll gerade Anzahl von Zeichen in α bedeuten. Wir betrachten den gemeinsamen Suffixbaum von S und S^{rev} und bestimmen $lca(q+1, n-q+1)$ für das maximale gerade Palindrom mit Mitte zwischen q und $q+1$.

Beispiel 3 Matching mit $\leq k$ Mismatches

Gegeben ein Pattern P der Länge n und ein Text T der Länge m, so sind alle Auftreten von P in T mit $\leq k$ Mismatches gesucht. Eine Möglichkeit, das Problem mit Hilfe von *lca*–Anfragen in Zeit $O(k \cdot m)$ zu lösen, ist die folgende:

Für jede Stelle i in T wird in Zeit $O(k)$ bestimmt, ob hier ein Match mit $\leq k$ Fehlern zwischen P und T anfängt. Dazu wird die Longest–Common–Extension (kurz *lce*) zwischen P und $T[i,m]$ bestimmt. Hat diese die Länge j, so gibt es also ein Mismatch zwischen $P(j+1)$ und $T(i+j)$. Weiter geht es mit *lce* von $P[j+2,n]$ und $T[i+j+1,m]$, ... Jede *lce*–Anfrage braucht laut Beispiel 1 nur konstante Zeit. Ist nach $\leq k$ Wiederholungen das Ende von P erreicht, so ist Position i von T der Anfang eines der gesuchten Matches.

6.1 *lca*–Anfragen in konstanter Zeit

Dieses Kapitel präsentiert ein rein graphentheorethisches Resultat, das jedoch äußerst wichtige Anwendungen im String–Matching Bereich hat, siehe vorhergehendes Kapitel. Die Aufgabenstellung ist wie folgt:

Wir wollen nach *linearer* Vorverarbeitung eines Baumes $\mathcal{T}$, für dessen Knotenmenge gilt: $|V(\mathcal{T})| = n$, eine *lca*–Anfrage für beliebige $x,\ y \in V(\mathcal{T})$ in *konstanter* Zeit (unabhängig von n!) beantworten. Wir betrachten zuerst vollständige binäre Bäume.

6.1.1 *lca*–Anfragen in vollständigen binären Bäumen

Sei $\mathcal{B}$ ein vollständiger binärer Baum mit p Blättern. Damit hat $\mathcal{B}$ $(2p-1)$ Knoten und die Tiefe $d = \log_2 p$.

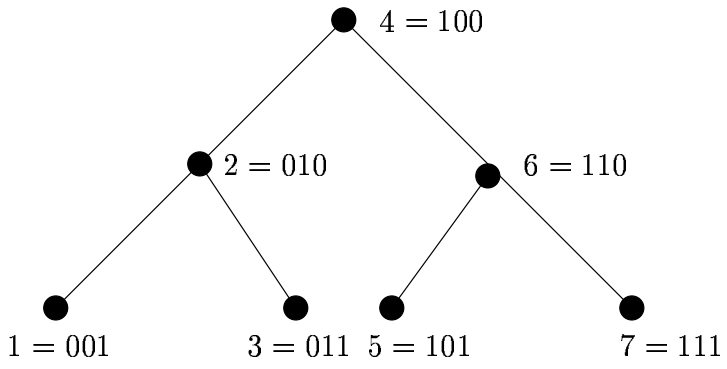
Nun führen wir darauf folgende Abbildung von $V(\mathcal{B})$ in die Menge der $(0,1)$–Vektoren der Länge $(d+1)$ ein.

$$v \mapsto \text{ Wegzahl}(v) = \text{Position in der InOrderTreeWalk–Nummerierung}$$

Zur Erinnerung: Die InOrder–Traversierung besucht zuerst rekursiv das linke Kind, dann den Elternknoten, dann rekursiv das rechte Kind.

Für einen (0,1)–Vektor a nennen wir die Position der letzten Eins – von hinten gezählt – Höhe $h(a)$. Die so definierte Höhe entspricht dann in einem vollständigen, binären Baum der „natürlichen“ Höhe, d.h. die Blätter haben die Höhe 1, die darüberliegenden Knoten die Höhe 2 ....

Beispiel.



Beobachtung. Eine andere Möglichkeit, die Wegzahl zu beschreiben, ist die folgende: Die Wegzahl eines Knotens v kodiert den Weg von der Wurzel zu

v. Geht der i–te Schritt auf dem Weg nach links, so wird die i–te Stelle der Binärdarstellung der Wegzahl (jetzt gezählt von vorne) auf Null gesetzt. Geht der Schritt nach rechts, wird sie auf 1 gesetzt. Um auf $d+1$ Bits zu kommen, wird zum Schluss mit einer 1 und so vielen Nullen wie nötig aufgefüllt. Beweis zur Korrektheit dieser Charakterisierung in der Übung.

Der $lca(v_1, v_2)$ von zwei Knoten v_1, v_2 mit Wegzahl$(v_1) <$ Wegzahl(v_2) ist nun genau der Knoten, dessen Wegkodierung das längste Anfangsstück der Wegkodierungen von v_1 und v_2 ist, das übereinstimmt. Zuerst wird der Fall $lca(v_1, v_2) \in \{v_1, v_2\}$ behandelt. Dies kann man leicht mit Hilfe der dfs–Nummerierung der Knoten tun, siehe Übung. Ansonsten findet man $lca(v_1, v_2)$ mit folgendem Algorithmus:

1. Bilde die XOR–Summe der beiden Wegzahlen.
2. Finde darin die linkeste 1, und nenne diese Position k.
3. $lca(v_1, v_2)$ hat dann die Wegzahl $v_1(1)v_1(2)\ldots v_1(k-1)$, aufgefüllt mit einer 1 und $(d+1)-k$ Nullen.

Bemerkung. Die Länge der Auffüllung entspricht genau der Höhe des Knotens im Baum.

Beispiel.
1. $lca(5,7):\ 101\ XOR\ 111 = 010$. Dann ist $k = 2$ und Wegzahl$(lca(5,7)) =$ 110 bzw. $lca(5,7) = 6$.

2. $lca(2,5):\ 010\ XOR\ 101 = 111$. Dann ist $k = 1$ und Wegzahl$(lca(2,5)) =$ 100 bzw. $lca(2,5) = 4$.

6.1.2 lca–Anfragen in allgemeinen Bäumen – Vorverarbeitung

Für allgemeine Bäume werden wir die lca–Anfrage an einen Baum $\mathcal{T}$ mit n Knoten in einen passenden Binärbaum $\mathcal{B}$ abbilden. Dazu führen wir als erstes eine Tiefensuche bzw. DepthFirstSearch in $\mathcal{T}$ durch. Deren Nummerierung übernimmt man dann als Knotennamen in $\mathcal{T}$.

Zur Erinnerung: Die Tiefensuche besucht zuerst den Elternknoten, dann nacheinander rekursiv alle Kinder.

Zusätzlich definieren wir folgende Abbildung I von der Menge der Knotennamen $\{1, \ldots, n\}$ in sich selbst. Sei v ein Knoten, dann suchen wir im Unterbaum mit Wurzel v den Knoten w, dessen $h(w)$–Wert maximal ist. Hierbei

sei h wie im vorigen Abschnitt die Länge der Auffüllung, also die Anzahl von Zeichen ab der rechtesten 1 nach rechts.

$$v \mapsto I(v) = w$$

Beispiel.

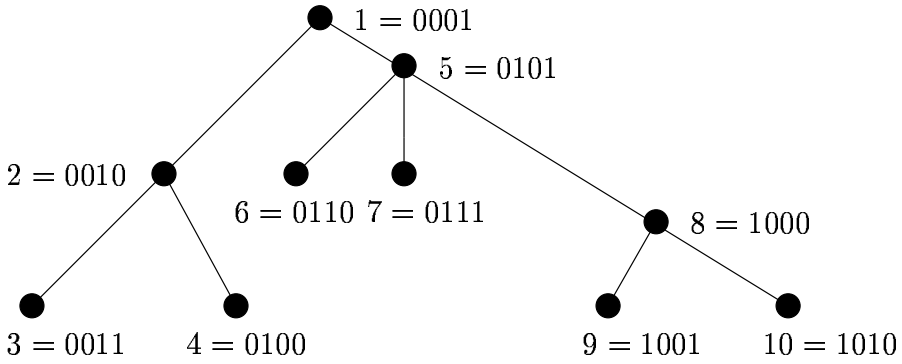


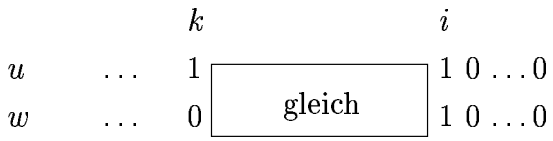
Abbildung 19: Hier ist $I(1) = I(5) = I(8) = 8$; $I(2) = 4$.

Beobachtung. Die Abbildung I zerlegt den Baum $\mathcal{T}$ in Teilbäume mit demselben I–Wert. Diese werden *Runs* genannt. Man kann sich überlegen, dass jeder Run ein Pfad in $\mathcal{T}$ ist. Der Knoten eines Runs, der am nächsten an der Wurzel ist, wird *Kopf* genannt.

Es ist jedoch noch zu klären, ob I überhaupt wohldefiniert ist. Das ist nur dann der Fall, wenn in jedem Teilbaum der Knoten mit dem maximalen h–Wert *eindeutig* ist. Dazu ein Lemma.

Lemma 4 *Der Knoten $I(v) = w$ aus der Abbildung I ist eindeutig.*

Beweis. Wir nehmen an, es gäbe zwei verschiedene Knoten $u \neq w$ mit $i := h(u) = h(w) \geq h(q)$ für alle Knoten q im Unterbaum von v. D.h. u und w unterscheiden sich in einem Bit *links* von i. Sei k das kleinste dieser unterschiedlichen Bits und o.E. $u > w$.



Konstruiere $x = \underbrace{\ldots \overset{k}{1}}_{\text{wie } u} 0 \ldots 0$.

Dann gilt $u > x > w$, und damit ist x die dfs–Nummer eines Knotens unter v, denn diese Nummern bilden ein Intervall. Aber $h(x) > i = h(u) \rightarrow$ Widerspruch zur Annahme, dass i der größte h–Wert ist, der in diesem Unterbaum auftaucht. □

Aus dem Lemma folgt, dass jeder Run in $\mathcal{T}$ auf einen Knoten in $\mathcal{B}$ abgebildet wird. Ein Run ist immer ein *Aufwärtsweg*, d.h., dass ein Run Teil eines Pfades von der Wurzel zu einem Knoten ist. Damit ist es möglich, die $I(v)$ Werte bottom–up zu berechnen.

Die lineare Vorverarbeitung von $\mathcal{T}$:

1. Tiefensuche auf $\mathcal{T}$, jeder Knoten bekommt einen Pointer zum Elternknoten.
2. Bottom–up Berechnung der Werte $I(v)$, jeder dieser Werte erhält einen Pointer zum Kopf des entsprechenden Runs.
3. $\mathcal{T}$ auf $\mathcal{B}$ abbilden, $v \mapsto I(v)$.
4. Für jedes v berechnen von A_v. Dabei ist A_v eine Bitzahl der Länge $\log n$, und das i–te Bit ist 1 $\iff$ es existiert ein Vorfahre u von v mit $h(I(u)) = i$.

6.1.3 Beantwortung der *lca*–Anfragen in konstanter Zeit

Lemma 5 *Sei z echter Vorfahre von x in $\mathcal{T}$, dann ist $I(z)$ Vorfahre von $I(x)$ in $\mathcal{B}$. Das bedeutet, unter der Abbildung I bleiben die Verhältnisse zwischen den Runs erhalten.*

Beweis. Falls $I(z) = I(x)$ ist, sind wir fertig. Andernfalls gilt wegen der Definition von I und h

$$i := h(I(z)) > h(I(x)).$$

Wir wollen zeigen, dass $I(z)$ und $I(x)$ in allen Bits links von i (die Bits werden von rechts gezählt) identisch sind. Angenommen, das sei nicht der Fall. Dann sei $k > i$ die linkeste Position mit einem Unterschied in $I(z)$ und $I(x)$ in Position k, und zwar sei diese Stelle o.B.d.A. 1 in $I(z)$ und 0 in $I(x)$, folglich $I(z) > I(x)$.

Analog der Argumentation im Beweis von Lemma 4 folgt nun die Existenz eines Knoten $v \in \mathcal{T}$ mit $I(x) < v < I(z)$ UND $h(v) > h(I(z))$. Das widerspricht der Definition von I, da v wegen der ersten Bedingung im Unterbaum von z liegt, und $h(I(z))$ den maximalen h–Wert bezeichnet, der in diesem Unterbaum auftaucht.

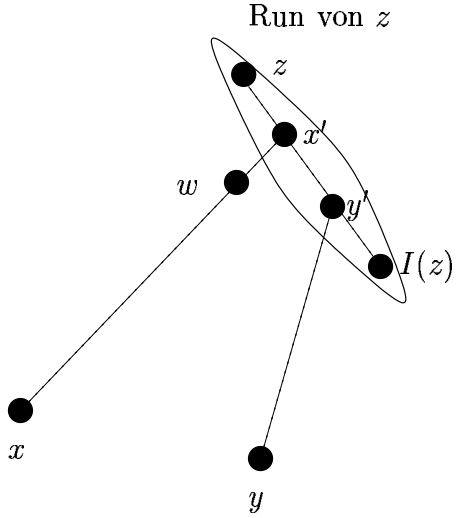
Nach Definition von i ist das Bit an Position i die rechteste 1 in $I(z)$, und links von diesem Bit ist der Weg von der Wurzel nach $I(z)$ in $\mathcal{B}$ kodiert. Da die Bits links von i in $I(x)$ und $I(z)$ gleich sind folgt, dass der Weg von $I(x)$ durch $I(z)$ geht. Damit ist $I(z)$ ein Vorfahre von $I(x)$ in $\mathcal{B}$. □

Satz 7 *Seien x, y Knoten und $z = lca(x, y)$. Sei $h(I(z))$ gegeben, dann läßt sich z in konstanter Zeit bestimmen.*

Beweis. Wir werden im folgenden die Knoten oft mit ihrer dfs–Nummer identifizieren. Zuerst eine kleine Beobachtung: Wir betrachten den Run, der z enthält. Die Wege von x bzw. y zur Wurzel müssen diesen Run treffen. Wir nennen x' bzw. y' den Knoten, in dem diese Wege auf den Run von z treffen. Dann erhalten wir aus der Definition von lca, daß z gleich x' oder y' ist, und $z = x' \iff x' < y'$.

Nun berechnen wir x' bzw. y' aus $h(I(z))$. Sei dazu $h(I(z)) =: j$.

1. Fall: Sei $h(I(x)) = j$, dann ist x im Run von z und $x' = x$.



2. Fall: Sei $h(I(x)) < j$ bzw. $x \neq x'$. Sei w der letzte Knoten auf dem Weg von x nach x', der außerhalb des Runs von z liegt. Aus $h(I(z))$ und dem Vektor A_x berechnen wir nun $h(I(w))$, $I(w)$, w und x'. Es gilt:

$h(I(w)) < j$ und $h(I(w))$ ist die größte Position $k < j$, so dass A_x eine 1 im Bit k hat. Man hat somit $w = x$ oder w ist ein echter Vorfahre von x, und im zugeordneten Binärbaum $\mathcal{B}$ ist $I(w) = I(x)$ oder $I(w)$ ist ein echter Vorfahre von $I(x)$, siehe Lemma 5. $I(w)$ hat eine 1 an der Stelle k und – nach der Wahl von k – Nullen rechts davon. w ist ein Vorfahre von x, und damit ist $I(w)$ in den Bits links von k identisch zu $I(x)$. Wir kennen also $I(w)$, fehlt noch w. w ist – nach Wahl – Kopf eines Runs, und alle Knoten eines Runs zeigen auf dem Kopf, insbesondere $I(w)$. x' ist nun der Vater von w.

Analog kann y' bestimmt werden, und mit unserer Beobachtung erhalten wir damit z. □

Nachdem wir nun z aus $h(I(z))$ bestimmen können, fehlt uns noch $h(I(z))$. Dazu der folgende Satz. Wir vergeben vorher noch ein paar Bezeichnungen: Sei $b = lca(I(x), I(y))$ im Binärbaum $\mathcal{B}$, und $h(b) =: i$.

Satz 8 *Sei $j \geq i$ die kleinste Position, so daß A_x und A_y in diesem Bit eine 1 haben. Dann ist $h(I(z)) = j$.*

Beweis. Sei $h(I(z)) =: k$. Wir werden nun zeigen $k \geq j$ und $k \leq j$.

$k \geq j$: z ist Vorfahre von x und y, also haben A_x und A_y in Position k eine 1. Damit ist $I(z)$ Vorfahre von $I(x)$ und $I(y)$, also $k \geq i$, und nach Wahl von j gilt $k \geq j \geq i$.

$k \leq j$: A_x hat ein 1–Bit in Position $j \geq i$. Also gibt es einen Vorfahren x' von x in $\mathcal{T}$, so dass gilt $I(x')$ ist Vorfahre von $I(x)$ in $\mathcal{B}$ und $h(I(x')) = j$. Analog erhalten wir einen Vorfahren y' von y mit $h(I(y')) = j$. Nun sind $I(x')$ und $I(y')$ Vorfahren von b, und somit ist $I(y') = I(x')$. Damit sind beide im gleichen Run, und ohne Einschränkung sei x' Vorfahre von y'. D.h. x' wird mindestens auf die Höhe von z in $\mathcal{B}$ abgebildet, also $j \geq k$. □

Nachdem wir die Sätze beisammen haben, folgt nun die Zeit der Ernte. Wir können den Algorithmus angeben und uns im Anschluß seinen Anwendungen zuwenden.

Algorithmus

1. Finde $b = lca(I(x), I(y))$ in $\mathcal{B}$.
2. Finde kleinstes $j \geq h(b)$ mit: A_x, A_y haben eine 1 in Bit j.
3. Setze $h(I(z)) = j$.
4. Finde x', y'.
5. $x' < y' \Rightarrow z = x'$, sonst $z = y'$.

6.2 Anwendung

In Abschnitt 5.1.3 haben wir uns mit dem Longest–Common–Substring von K Strings $S_1,\ S_2,\ \ldots, S_K$, $\sum_i |S_i| = n$, beschäftigt und einen Algorithmus mit Laufzeit $O(K \cdot n)$ angegeben. Dieses Problem wollen wir jetzt in Zeit $O(n)$ lösen, dabei verwenden wir natürlich unseren neuen Algorithmus.

Erinnerung: Wir suchen für $2 \leq k \leq K$ den längsten Teilstring, der in $\geq k$ Strings vorkommt, dessen Länge sei l_k.

In dem gemeinsamen Suffixbaum von $S_1,\ S_2,\ \ldots, S_K$ hat jeder String sein eigenes Endsymbol. Wir wollen wie in 5.1.3 die Abbildung

$$v \mapsto C(v) = \#\text{ Blätter verschiedener Strings unter } v$$

einführen, um danach v mit maximaler Tiefe und $C(v) = k$ zu bestimmen.

Wie bestimmen wir $C(v)$? Dazu führen wir folgende Bezeichnungen ein:

$$S(v) = \#\text{ Blätter unter } v,$$

$$u_i(v) = \#\text{ Blätter mit Stringnummer } i \text{ unter } v.$$

Dann ist sicher $C(v) \leq S(v)$. Nun setzen wir

$$U(v) = \sum_{i:u_i(v)>1} (u_i(v) - 1),$$

das heißt U ist ein Korrekturfaktor der angibt, wie viele „Duplikate“ von Suffixen unter v vorkommen. Dann gilt $C(v) = S(v) - U(v)$, und S läßt sich in linearer Zeit bestimmen.

Wir berechnen nun noch $U(v)$ bzw. die Werte $u_i(v) - 1$. Dazu erstellen wir mit Hilfe einer Tiefensuche die Listen $L_1,\ L_2,\ \ldots, L_K$, wobei in L_i die durch die Tiefensuche geordneten Blätter mit Label i stehen.

Lemma 6 *Wenn man für alle benachbarten Paare in L_i die lca–Anfragen berechnet, so landet man für jedes v genau $(u_i(v) - 1)$ Mal im Unterbaum von v.*

Nun halten wir für jedes $w \in \mathcal{T}$ einen Zähler $h(w)$, der zählt, wie oft w als Ergebnis einer solchen *lca*-Anfrage auftritt. Dann bestimmen wir U bottom–up in linearer Zeit durch:

$$U(v) = \sum_{w \in \text{ Unterbaum von } v} h(w).$$

7 Approximatives Matching

Das im Folgenden vorgestellte Verfahren zum approximativen Matching beruht im wesentlichen auf dem Prinzip des dynamischen Programmierens.

7.1 Editierabstand, globales & lokales Alignment

Wir berücksichtigen folgende Operationen, um einen String S_1 in einen anderen String S_2 zu transformieren:

- i Insert
- d Delete
- r Replace
- m Match (keine Operation)

Beispiel. Sei S_1=VINTNER, S_2=WRITERS

		V	_	I	N	T	N	E	R	_
		W	R	I	_	T	_	E	R	S
$S_1 \rightarrow S_2$	:	r	i	m	d	m	d	m	m	i

Definition 20 *Wir nennen die minimale Anzahl von Operationen i, d, r, um einen String S_1 in einen anderen String S_2 zu transformieren, den* Editierabstand *(auch* Lewenshtein–Abstand*), und schreiben dafür $D(S_1, S_2)$.*

Definition 21 *Ein globales Alignment von S_1 und S_2 entsteht durch Einfügen von Lücken (auch Spaces) in S_1 und S_2, so dass danach die Wörter gleichlang sind und kein Space einem Space gegenüber liegt.*

Beobachtung. Ein globales Alignment ist somit das Ergebnis einer Editiertransformationsprozesses.

Wir wollen nun $D(S_1, S_2)$ bestimmen und verwenden dabei dynamisches Programmieren. Sei $|S_1| = n$ und $|S_2| = m$. Bezeichnen wir $D(i, j) := D(S_1[1, i], S_2[1, j])$, dann suchen wir $D(n, m)$. $S[1, 0]$ bezeichnet dabei den leeren String. Dazu folgender Satz:

Satz 9 *Es gilt:*

$$\forall i : D(i,0) = i$$
$$\forall j : D(0,j) = j$$
$$D(i,j) = \min\{D(i-1,j)+1,\ D(i,j-1)+1,\ D(i-1,j-1)+t(i,j)\},$$

$$\textit{wobei } t(i,j) = \begin{cases} 1 & : \quad S_1(i) \neq S_2(j) \\ 0 & : \quad S_1(i) = S_2(j) \end{cases}.$$

Beweis. ($\rightarrow$ Übung.)

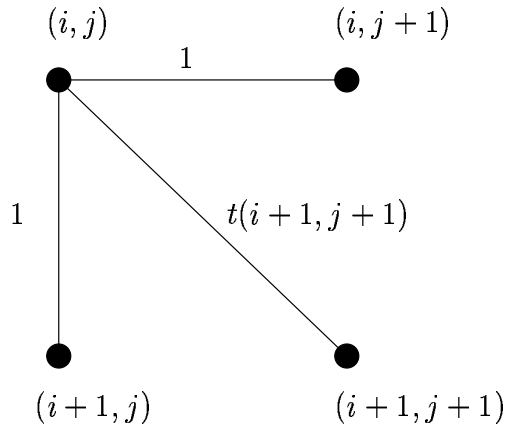
Korollar 5 *Der Editierabstand zwischen zwei Strings S_1 und S_2 kann bottom–up in Zeit $O(n \cdot m)$ berechnet werden.*

Diese Rechnung führt zu einer $n \times m$–Tabelle. Daraus kann man durch Backtracing eine Editierfolge gewinnen. Hierfür gilt:

Satz 10 *Jeder Weg von (n,m) nach $(0,0)$ bestimmt eine optimale Transformation (und ein Alignment) mit $D(S_1,S_2)$ Operationen.*

Man kann den Editierabstand verallgemeinern, indem man Gewichte einführt. Einerseits durch Gewichte auf den Operationen, andererseits durch zeichenabhängige Gewichte. Bei den Gewichten für verschiedene Operationen sollte gelten, dass ein Mismatch weniger wiegt als eine Folge von Delete und Insert.

Man kann den Editierabstand auch als graphentheoretisches Problem auffassen. Wir haben einen Editiergraphen mit $(n+1)\cdot(m+1)$ Knoten. Der Knoten (i,j) entspricht dabei dem Paar $(S_1[1,i], S_2[1,j])$, die Kanten entsprechen den Editieroperationen. Um den Editierabstand $D(S_1,S_2)$ zu bestimmen, suchen wir nach der Länge eines kürzesten Weges von $(0,0)$ nach (n,m).



7.2 Ähnlichkeit von Strings

Sei Σ ein Alphabet von Zeichen, $|\Sigma| = n$, und $\Sigma^{(-)} := \Sigma \cup \{\_\}$.

Nun wollen für dieses Alphabet den Begriff Ähnlichkeit einführen, dazu geben wir Matches positives Gewicht und Mismatches negatives Gewicht. Konkret: wir führen eine Score–Abbildung auf dem kartesischen Produkt ein.

Definition 22 Score *sei eine Abbildung*

$$s : \Sigma^{(-)} \times \Sigma^{(-)} \to \mathbb{R}.$$

Beobachtung. Ein Score kann man durch eine $(n+1) \times (n+1)$–Matrix festlegen.

Definition 23 *Sei eine Scoringmatrix gegeben, S_1, S_2 Strings. Dann nennen wir die* Ähnlichkeit *v von S_1 und S_2:*

$$v(S_1, S_2) = \textit{max. Wert eines Alignments.}$$

$v(i,j)$ bezeichne die Ähnlichkeit zwischen $S_1[1,i]$ und und $S_2[1,j]$. Falls $i = 0$ oder $j = 0$, so ist damit der leere String gemeint. Zur Berechnung können wir nun wie im Fall des Editierabstandes vorgehen:

$$v(0,j) = \sum_{k=1}^{j} s(\_, S_2(k))$$

$$v(i,0) = \sum_{k=1}^{i} s(S_1(k), \_)$$

$$\begin{aligned} v(i,j) = \max\{&v(i-1,j) + s(S_1(i), \_),\ v(i,j-1) + s(\_, S_2(j)),\\ &v(i-1,j-1) + s(S_1(i), S_2(j))\}. \end{aligned}$$

Man benötigt $O(n \cdot m)$ Zeit für die Erstellung der Tabelle und $O(n+m)$ Zeit für die Ausgabe eines optimalen globalen Alignments. Ähnlichkeit läßt sich auch graphentheoretisch modellieren. Wir betrachten den selben Graphen wie oben, wobei jetzt die Kantengewichte durch s bestimmt werden. Gesucht ist nun ein *maximaler* Weg von $(0,0)$ nach (n,m).

Beispiel. Wir wollen $lcs(S_1, S_2)$, die längste gemeinsame Teil*sequenz* von S_1 und S_2 , in unserer neu gelernten Sprache ausdrücken. Eine gemeinsame Teilsequenz wird durch je eine Teilfolge aus $1, \ldots, n$ bzw. $1, \ldots, m$ bestimmt.

WRITERS und VINTNER enthalten folgende gemeinsame Teilsequenzen: R, I, E, T, IT, IE, IR, TE, TR, ER, ITE, IER, TER, ITR, ITER.

Die längste gemeinsame Teilsequenz $lcs(S_1, S_2)$ lässt sich als optimales globales Alignment bzgl. der Scoring–Matrix s in Zeit $O(m \cdot n)$ finden. Dabei sei

$$s(Match) = 1 \quad \text{und} \quad s(Mismatch) = s(Space) = 0$$

Zum Vergleich: Das Auffinden des längsten gemeinsamen Teil*strings* ist in Zeit $O(m + n)$ möglich.

Die *end–space–free* Variante berechnet nichts für den Vergleich Buchstabe gegen Space, wenn der Space vor oder nach dem String kommt. Zur Modellierung werden einfach die Anfangsbedingungen beim Berechnen der Ähnlichkeit angepasst. Sie lauten nun:

$$v(i, 0) = 0 = v(0, j) \text{ für alle } i, j.$$

Gesucht ist das Alignment, das den Wert in der letzten Zeile oder Spalte maximiert.

7.3 Das local–Alignment–Problem

Seien 2 Strings S_1, S_2, mit $|S_1| = n$, $|S_2| = m$, gegeben. Gesucht sind Teilstrings α, β, die $v(\alpha, \beta) =: v^*$ maximieren.

Das geht mit folgendem Verfahren: Wir definieren für $1 \leq i \leq n$, $1 \leq j \leq m$

$$v(i, j) = \max\{v(\alpha, \beta) \mid \alpha \text{ Suffix von } S_1[1, i] \text{ und } \beta \text{ Suffix von } S_2[1, j]\}.$$

α, β können dabei auch leer sein. Um v^* zu bestimmen, haben wir den folgenden Satz.

Satz 11 *Mit der Notation von oben gilt:*

$$v^* = \max\{v(i, j) \mid 1 \leq i \leq n,\ 1 \leq j \leq m\}.$$

Die $v(i, j)$ lassen sich wie folgt bestimmen:

$$v(i, 0) = 0 = v(0, j) \text{ für alle } i, j.$$

$$\begin{aligned} v(i, j) = \max\{0,\ & v(i, j-1) + s(\_, S_2(j)),\ v(i-1, j) + s(S_1(i), \_), \\ & v(i-1, j-1) + s(S_1(i), S_2(j))\}. \end{aligned}$$

7.4 Gaps und Gap Weights

Unser Ziel ist es, die Verteilung an Spaces in einem Alignment beeinflussen zu können. Dazu werden wir diese mit verschiedenen Gewichten ausstatten.

Definition 24 *Gegeben sei ein Alignment S' eines Strings S. Wir nennen einen maximalen Teilstring, der nur aus Spaces besteht, Gap.*

Je nach Zweckmäßigkeit gibt es nun verschiedene Möglichkeiten, Gaps zu gewichten. Eine Auswahl:

- **Konstantes** Gap–Gewicht: Man wähle eine Konstante $W_g > 0$ als Gap–Strafe und maximiere den Ausdruck

$$\sum_i s(S_1'(i), S_2'(i)) - W_g \cdot (\#\text{gaps}),$$

  wobei S_i' der String zu S_i im Alignment ist, und $\forall x : s(\_, x) = 0 = s(x, \_)$.

- **Affines** Gap–Gewicht ist wohl das am häufigsten benutzte. Man bestraft sowohl das Auftreten eines Gaps, als auch dessen Länge, d.h. ein Gap wird mit $W_g + W_s \cdot |Gap|$ bestraft. Zu maximieren ist jetzt also

$$\sum_i s(S_1'(i), S_2'(i)) - W_g \cdot (\#\text{gaps}) - W_s(\#Spaces).$$

- **Konvexes** Gap–Gewicht: Wieder betraft man das Auftreten eines Gaps mit W_g und dessen Länge. Dabei wird im Unterschied zu vorher die Länge jetzt mit einer konvexen Funktion f (d.h. die zweite Ableitung von f ist < 0) bestraft. Dadurch werden lange Gaps anteilig weniger als kurze Gaps betraft. Insgesamt wird für ein Gap also $W_g + f(|Gap|)$ berechnet.

Laufzeiten: Für beliebige Gewichte $O(nm^2+n^2m)$, für affine bzw. konstante Gewichte $O(nm)$ und für konvexe Gewichte $O(nm \log m)$.

Beweis. Wir betrachten ein Alignment von $S_1[1,i]$ zu $S_2[1,j]$ mit dem Wert $V(i,j)$. Dann können 3 Typen von Alignments auftreten:

1. $S_1[1,i]$ endet im Alignment mit Gap, diesen Wert nennen wir $E(i,j)$.
2. $S_2[1,j]$ endet im Alignment mit Gap, diesen Wert nennen wir $F(i,j)$.

3. beide enden im Alignment ohne Gap, diesen Wert nennen wir $G(i,j)$.

Es gilt nun: $V(i,j) = \max\{E(i,j), F(i,j), G(i,j)\}$

$G(i,j) = V(i-1,j-1) + s(S_1(i), S_2(j))$,

$E(i,j) = \max_{0 \le k \le j-1}\{V(i,k) - w(j-k)\}$,

$F(i,j) = \max_{0 \le l \le i-1}\{V(l,j) - w(i-l)\}$,

wobei w eine beliebige positive Gap–Gewichtsfunktion ist. Dazu kommt die folgende Initialisierung:

$V(i,0) = F(i,0) = -w(i)$, $V(0,j) = E(0,j) = -w(j)$, $V(0,0) = G(0,0) = w(0) = 0$, $G(i,j)$ ist nicht definiert für $i = 0$ und $j \neq 0$ oder $i \neq 0$ und $j = 0$.

Für allgemeine Gap–Gewichte erhält man damit Laufzeit $O(nm^2 + n^2m)$. Das sieht man wie folgt: Betrachte $V(i,j)$, man benötigt in einer Zeile $\sim m^2$ mal Einträge für die E–Werte und in einer Spalte $\sim n^2$ mal Einträge für die F–Werte.

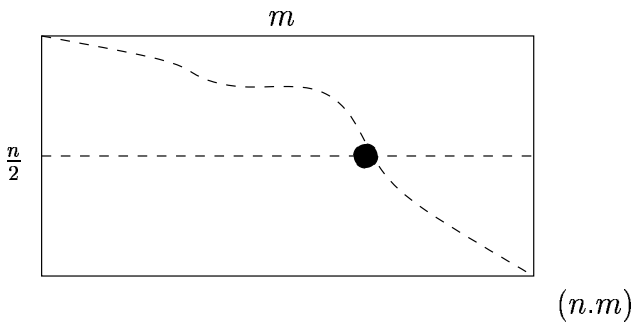
Zeitschranken für die anderen Gewichte zur Übung. □

7.5 Verbesserung der dynamischen Programmierung

7.5.1 Linearer Speicherbedarf

Sei $S_1 = n$, $S_2 = m$. Dann bekommen wir mit dem Ansatz aus Kapitel 7.1 in $O(n \cdot m)$ Zeit und mit $O(n \cdot m)$ Speicher ein optimales Alignment von S_1 und S_2. In diesem Abschnitt wollen wir den Speicherbedarf auf $O(n+m)$ beschränken, um ein optimales Alignment zu finden. Bisher können wir den *Wert* des Editierabstandes mit diesem Speicherplatz bestimmen, nicht jedoch den Alignment–*Weg* — für diesen haben wir die ganze Matrix gebraucht.

Idee: Teile und herrsche & Rekursion.



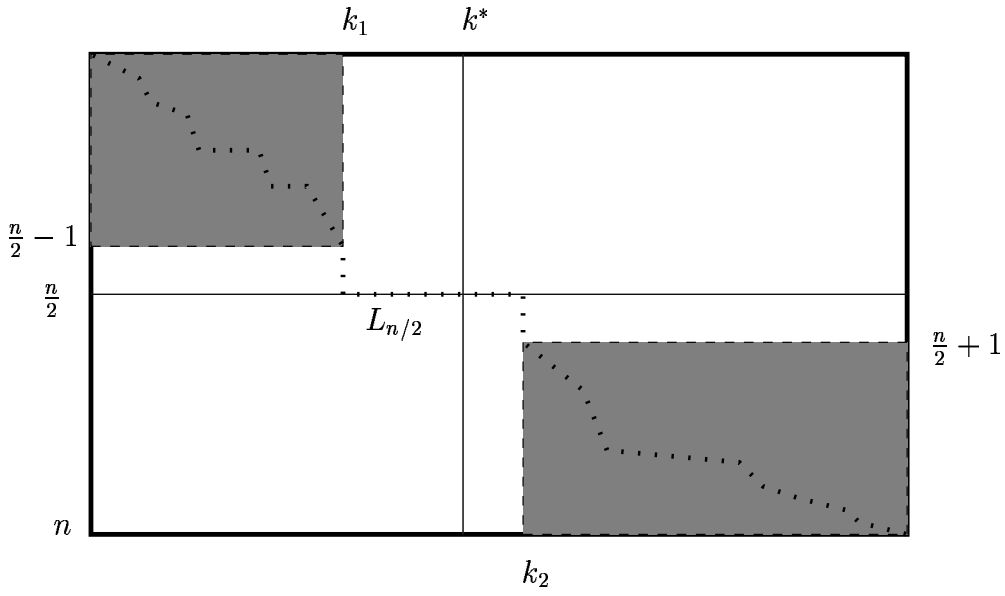
Der Ansatz ist folgendermaßen: Wir berechnen, wo der Alignment–Weg die $n/2$–te Zeile kreuzt. Außerdem benutzen wir, dass wir mit einem optimalen Alignment für S_1, S_2 auch ein optimales Alignment für die Strings S_1^{rev}, S_2^{rev} kennen. Wir lesen dazu das optimale Alignment von hinten.

Lemma 7 *Sei $V^{rev}(n/2, m-k)$ die Ähnlichkeit von $S_1^{rev}[1, n/2]$ und $S_2^{rev}[1, k]$ bzw. die Ähnlichkeit von $S_1[n/2+1, n]$ und $S_2[m-k+1, m]$. Mit dieser Notation gilt nun:*

$$V(n,m) = \max_{0 \le k \le m} \{V(n/2, k) + V^{rev}(n/2, m-k)\}$$

Es sei k^* ein Spaltenindex, der den maximalen Wert für $V(n,m)$ liefert. Dieses Lemma erlaubt es dann, ein optimales Alignment in zwei Schritten zu bestimmen: Zuerst ein optimales Alignment von $S_1[1, n/2]$ mit $S_2[1, k^*]$, dann ein optimales Alignment von $S_1[n/2+1, n]$ und $S_2[k^*+1, m]$. Dabei wird letzteres über die „umgedrehten“ Strings berechnet.

Es gibt nun einen optimalen Alignment–Weg L, der durch den Knoten $(n/2, k^*)$ im Alignment–Graphen geht. $L_{n/2}$ sei der Teil des Weges L, der durch Knoten in Zeile $n/2$ verläuft, und der letzte Knoten vor sowie der erste Knoten nach diesem Stück. Die wichtige Beobachtung hier ist: $L_{n/2}$ ist Teil eines optimalen Alignments.



Fakt: Wir finden k^* und $L_{n/2}$ in $O(n \cdot m)$ Zeit und $O(m)$ Speicher.

Damit kommen wir zu folgender Analyse:

Für Eingabe–Größen p und q wird zum füllen der Tabelle mit dynamischem Programmieren $c \cdot p \cdot q$ Zeit benötigt, c eine gewisse Konstante. Mit der Unterteilung in zwei Mal $n/2$ Zeilen ergibt sich eine Zeit von $c \cdot n/2 \cdot m + c \cdot n/2 \cdot m = c \cdot n \cdot m$ für die ganze Tabelle, insbesondere also die Zeilen, die gebraucht werden. Jedoch wird dabei mehr gelernt als nur die Tabelle: Die Werte k^*, k_1, k_2 sowie das Pfadstück $L_{n/2}$ sind nach dieser Zeit bekannt. In diesem Schritt wird also nicht nur der Wert $v(n, m)$ berechnet, sondern auch ein Teilstück des optimalen Alignments, nämlich $L_{n/2}$. Die Aufgabe ist jetzt darauf reduziert, optimale Alignments für die in der Abbildung grau gekennzeichneten Stücke zu finden, d.h. optimale Alignments zwischen $S_1[1, n/2 - 1]$ und $S_2[1, k_1]$ sowie $S_1[n/2 + 1]$ und $S_2[k_2, m]$. Damit wird das Problem in zwei disjunkte Teilprobleme der Größen $n/2 \times k^*$ und $n/2 \times (m - k^*)$ zerlegt, welche mit Rekursion gelöst werden können. Die Laufzeit dafür ist $c \cdot n/2 \cdot k^* + c \cdot n/2 \cdot (m - k^*) = c \cdot n \cdot m/2$.

Insgesamt ergibt sich mit diesem Verfahren eine Laufzeit von

$$\sum_{i=1}^{\log n} \frac{cnm}{2^{i-1}} \leq 2cnm.$$

Bemerkung. Auf analoge Weise läßt sich ein optimales lokales Alignment mit linearem Speicher bestimmen, Details in der Übung.

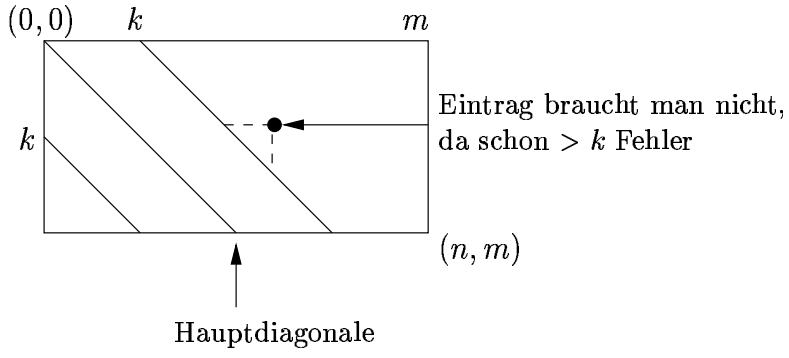
7.5.2 k–difference Alignment/Matching

Wir wollen nun die Anzahl der Unterschiede im Alignment begrenzen.

Variante 1: k–difference global Alignment

Dazu sind zwei Strings S_1, S_2 der Länge n bzw. m gegeben, gesucht ist das beste globale Alignment mit $\leq k$ Mismatches und Spaces.

Idee: Gehen wir im Alignment–Graph nach rechts oder nach unten (nicht diagonal), so entspricht dies einem Space in S_1 bzw. S_2. Um nun ein k–difference global Alignment zu finden, müssen wir somit nur den Streifen der Breite $2k$ um die Hauptdiagonale (i, i) betrachten. Für ein k–difference global Alignment muß insbesondere gelten: $|m - n| \leq k$. Es müssen also nur $O(km)$ Matrixeinträge berechnet werden. Angenommen ein maximales globales Alignment macht $k^* \leq k$ Fehler, dann müssten nur $O(k^* \cdot m)$ Matrixeinträge für die Bestimmung von k^* berechnet werden. Dies erreicht man, indem man die Streifenbreite von 2 aus beginnend immer verdoppelt.



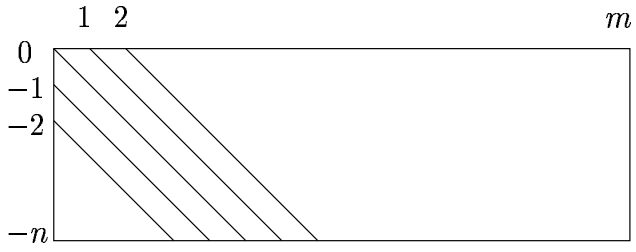
Nach dem Ausfüllen eines Streifens wird jeweils getestet, ob der Eintrag in (n, m) höchstens die aktuelle Streifenbreite ist. Sei dies bei Breite k' das erste Mal der Fall. Dann ist $k' \leq 2k^*$, und k^* steht in (n, m). Insgesamt wird $O(m + 2m + 4m + \ldots + k'm) = O(k^* \cdot m)$ Zeit benötigt.

Variante 2: k–difference inexact Matching (Landau,Vishkin)

Sei P ein Pattern und T ein Text mit Länge n bzw. m. Nun suchen wir alle Matches von P in T mit $\leq k$ Fehlern.

Mit einem Hybridalgorithmus bestehend aus dynamischen Programmieren und Suffixbäumen läßt sich eine Laufzeit von $O(km)$ erreichen. Zur Erinnerung: Insertions, Deletions und Mismatches werden mit 1 gewichtet, zur Initialisierung werden alle Einträge in Zeile 0 auf Null gesetzt.

Definition 25 *Wir nennen einen Weg α, der in Zeile O des Alignmentgraphen beginnt und genau d Fehler macht, einen* d–Weg. *So einen d–Weg nennen wir* i–tiefsten Weg, *wenn er auf der Diagonalen i endet und unter allen d–Wegen, die auf dieser Diagonalen enden, den tiefsten Endpunkt hat.*



Nummerierung der Diagonalen

Beobachtung. Jeder d–Weg, der die Zeile n erreicht, entspricht einem Matching mit d Fehlern.

Unser Ziel ist es somit, alle d–Wege mit $d \leq k$ zu finden, die die n–te Zeile erreichen.

Algorithmus:

1. Der Suffixbaum für T und P und die Vorverarbeitung für *lce*–Anfragen werden in Zeit $O(n+m)$ berechnet.

2. Iteriere für $0 \leq d \leq k$.

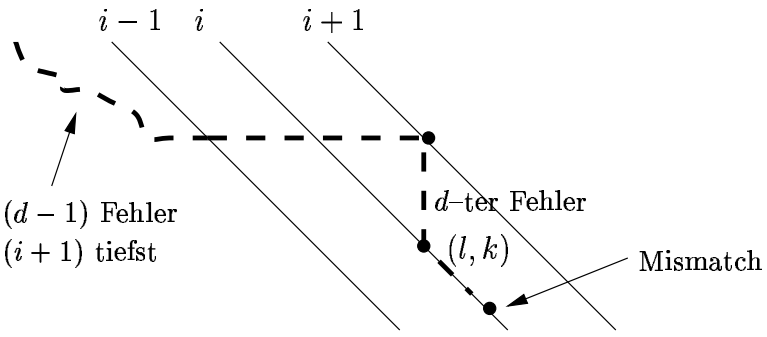
   Für $-n \leq i \leq m$ konstruiere den i–tiefsten d–Weg aus gewissen schon bekannten $(d-1)$–Wegen in $O(n+m)$ Zeit für jedes d. Genauer:

$d = 0$: Man findet die i–tiefsten 0–Wege durch *lce*–Anfragen von einem Blatt mit Label P und Blättern mit Label $T[i,m]$, $i = 0, \ldots, m$.

Jede *lce*–Anfrage kann in konstanter Zeit beantwortet werden, d.h. die i–tiefsten 0–Wege werden in Zeit $O(m)$ gefunden.

$d > 0$: Für den i–tiefsten Weg, $-n \leq i \leq m$, tritt einer der folgenden drei Fälle auf:

(a) Der Weg besteht aus einem $(i+1)$–tiefsten $(d-1)$–Weg, gefolgt von einer vertikalen Kante zur Diagonalen i und dem längstmöglichen Stück entlang der Diagonalen i. In anderen Worten: Nach $d-1$ Fehlern befinden wir uns auf der Diagonalen $(i+1)$ und sind dort maximal tief. Dann geschieht der d–te Fehler, und damit landen wir auf der Diagonalen i. Durch eine *lce*–Anfrage für $P[k,n]$ und $T[l,m]$ im Suffixbaum erhalten wir den Knoten vor dem nächsten Mismatch.



(b) Der Weg besteht aus einem $(i-1)$–tiefsten $(d-1)$–Weg, einer horizontalen Kante zur Diagonalen i und dem längstmöglichen Match entlang der Diagonalen i.

(c) Der Weg besteht aus einem i–tiefsten $(d-1)$–Weg, einer diagonalen Kante und dem längstmöglichen Match entlang der Diagonalen i.

7.6 Suffixbäume und gewichtete Alignments

Sei wie immer P ein Pattern und T ein Text der Länge n bzw. m, außerdem sei eine Scoring–Matrix gegeben. Dann interessieren wir uns für folgende zwei Probleme.

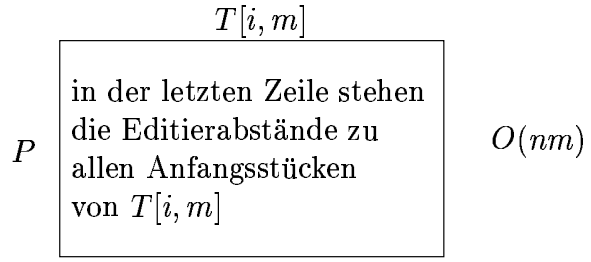
1. P–against–all: Berechne die gewichteten Editierabstände zu allen Teilstrings T' von T.
2. All-against–all: Finde zu einem Schwellwert d alle Teilstrings $P' \subset P$ und $T' \subset T$ mit Editierabstand $\leq d$.

In den zwei folgenden Abschnitten wird die Lösung dieser beiden Probleme erläutert.

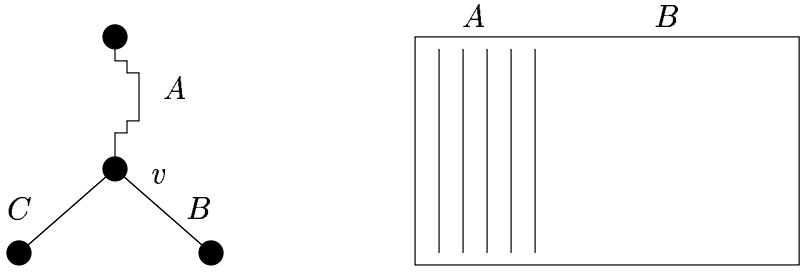
7.6.1 P–against–all

Der naive Ansatz führt zu folgender Überlegung: Es gibt $\sim m^2$ viele Substrings von T, d. h. wir müssen $\sim m^2$ Editierabstände der Form $D(P, T')$ berechnen. Wir haben somit eine Gesamtlaufzeit $O(nm^3)$.

Glücklicher weise geht es besser: Wir betrachten die m Suffizes von T und berechnen dann $D(T[i,m], P)$, für $1 \leq i \leq m$. Dabei berechnen wir alle anderen Abstände mit, der Abstand von P und $T[i,j]$ steht in Zelle $(n, j - i + 1)$. So wird eine Laufzeit von $O(nm^2)$ erreicht.



Das ist schon sehr schön, jedoch: Es geht sogar noch besser! Bisher berechnen wir Abstände immer noch mehrfach für den Fall, daß in T Wiederholungen vorkommen. Der neue Ansatz ist: Wir sortieren die Suffizes im Suffixbaum von T mit einer dfs–Suche, und füllen die Tabelle spaltenweise mit dynamischem Programmieren bezüglich der neuen Suffix–Sortierung. Dadurch wird Mehrfachberechnung vermieden. Seien also T' und T'' Suffizes von T, und außerdem $T' = AB$ und $T'' = AC$. Dann gibt es, wie im Bild, einen Knoten v mit Label A. In den Tabellen von P gegen T' und P gegen T'' sind die ersten $|A|$ Spalten identisch. Nun speichern wir im Knoten v einfach die entsprechende letzte Zeile und Spalte der Tabelle, so dass wir für die Berechnung der Editierabstände zu weiteren Suffizes, die mit A beginnen, an dieser Stelle in der Tabelle ansetzen können.



Dabei wird nun das Label jeder Kante von $\mathcal{T}$ nur ein Mal mit P verglichen, um alle Einträge zu bestimmen, wird also $O(n \cdot (\text{Länge von } \mathcal{T}))$ Zeit gebraucht. Außerdem muss in jeden inneren Knoten von $\mathcal{T}$ als Zwischenergebniss eine Spalte der Länge n und eine Zeile der Länge $\leq m$ geschrieben werden. Das führt zu einer Gesamtlaufzeit von $O(n \cdot (\text{Länge von } \mathcal{T}) + m(n+m))$, der Speicherbedarf beträgt $O(m^2)$.

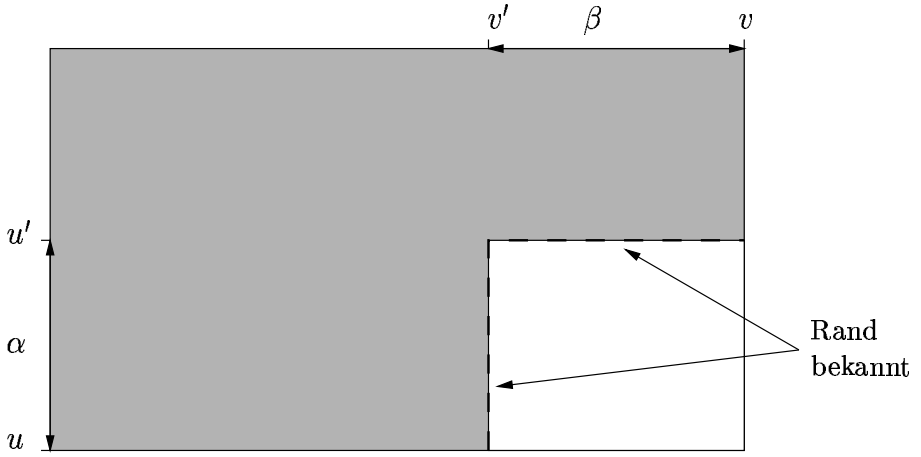
Bemerkung. Durch verdoppeln der Rechenzeit kann der Speicherplatzbedarf auf $O(m+n)$ reduziert werden, Details in der Übung.

7.6.2 All-against–all

Zuerst ein Mal ist zu bemerken, dass bei dieser Problemstellung allein die Ausgabe eine Größe von $O(n^2m^2)$ hat. Um diese Zeitschranke einzuhalten, kann man es sich sehr leicht machen: Für jedes der mn Paare von Startpositionen (i, j) in P bzw. T berechne die Tabelle für den Abstand von $P[i, n]$ und $T[j, m]$ in $O(mn)$ Zeit. In der Zelle (k, l) einer solchen Tabelle steht wieder der Abstand von $T[i, l]$ und $P[j, k]$.

Oft braucht man jedoch gar nicht wirklich *alle* Abstände, sondern nur einen gewissen Teil davon. Damit wird möglicherweise die Ausgabegröße entscheidend kleiner. In diesem Fall helfen wieder Suffixbäume, um die Rechenzeit verglichen mit der naiven Methode deutlich zu senken.

Es sei nun $\mathcal{T}_P$ der Suffixbaum von P und $\mathcal{T}_T$ der Suffixbaum von T. Weiter sei u ein Knoten aus $\mathcal{T}_P$ und v ein Knoten aus $\mathcal{T}_T$. Dem Paar (u, v) wird die (dynamische–Programmier–) Tabelle vom Vergleich zwischen $Label(u)$ und $Label(v)$ zugeordnet. Mit u', v' werden die Elternknoten von u, v bezeichnet, und es seien $\alpha = Label(u', u)$, $\beta = Label(v', v)$.



Wenn die Tabellen für (u', v'), (u, v'), (u', v) bekannt sind, müssen für die Tabelle (u, v) nur noch $|\alpha| \cdot |\beta|$ Einträge berechnet werden.

Nun sortieren wir die Paare (u, v) lexikographisch nach aufsteigenden Stringlängen, und berechnen in dieser Reihenfolge die Tabellen unter Ausnutzung des eben beschriebenen Tricks. Damit kommen wir zu einem Gesamtaufwand von $O((\text{ Länge von } \mathcal{T}_P) \cdot (\text{ Länge von } \mathcal{T}_T) + \text{Ausgabegröße })$.

7.7 Ausschlußverfahren

Wir wollen nun mit einem anderen Zugang die bisherigen Laufzeit von $O(km)$ für die Probleme k–Mismatch und k–Difference verbessern. Wir gehen dabei nach folgenden Ausschlußprinzip vor: Das Pattern wird zerkleinert. Wenn die kleinen Stücke im Text nicht vorkommen, kann auch das ganze Pattern nicht erscheinen.

Das Ziel ist es dabei, eine erwartete sublineare Laufzeit zu erreichen. Der hier vorgestellte Ansatz folgt einer Idee von Baeza–Yates und Perleberg, 1992.

Gegeben sind P und T wie immer, gesucht sind Matches zwischen P und T mit maximal k Fehlern. Dazu zerlegen wir P in Stücke der Länge $\lfloor \frac{n}{k+1} \rfloor =: r$.

Beobachtung. Falls P den Teilstring $T' \subset T$ mit $\leq k$ Fehlern matcht, dann enthält P ein Stück der Länge r, das ein Stück der Länge r in T *ohne Fehler* matcht.

Algorithmus:

1. Sei $\mathcal{P}$ die Menge der ersten $k+1$ Stücke von P (i.A. $(k+1) \not| m$).

2. Finde alle exakten Matches in T von allen $p \in \mathcal{P}$, z.B. mit dem Algorithmus von Aho–Corasick.

3. Für alle diese Matches: Sei i die Position des Matches in T, dann suche approximatives Match von P in $T[i-k-n, i+k+n]$.

Analyse für Schritt 3: Sei Σ ein Alphabet mit $|\Sigma| = \sigma$, und seien T und P beliebige Strings über Σ. Sei p ein beliebiges, festes Stück von P der Länge $|p| = r$. Dann gilt für den Erwartungswert:

$E(\#\text{ exaktes Match von } p) \sim \frac{m}{\sigma^r}$,

$E(\#\text{ exaktes Match irgendeines Stückes } p) \sim (k+1) \cdot \frac{m}{\sigma^r}$,

$E(\text{Zeit für Schritt 3}) \sim m \cdot n^2 \cdot \frac{k+1}{\sigma^r}$.

Das Ziel ist es, Schritt 3 in weniger als $c \cdot m$ Zeit durchzuführen. Die Frage ist nun also, wie die Konstante k gewählt werden muß, damit

$$m \cdot n^2 \cdot \frac{k+1}{\sigma^r} < cm$$

ist für ein gewisses c. Zur Vereinfachung der Rechnung setze $k = n-1$ und bestimme r, so dass gilt

$$\frac{mn^3}{\sigma^r} = cm\,.$$

Das gilt für $r = \log_\sigma n^3 - \log_\sigma c$, gleichzeitig war jedoch auch $r = \lfloor \frac{n}{k+1} \rfloor$. Damit ergibt sich für $k \in O(n/(\log n))$ eine Laufzeit von $O(m)$ für den obigen Algorithmus.

In anderen Worten: Für eine Fehlerrate kleiner als $1/(\log_\sigma n)$ läuft der Algorithmus mit in m linearer Laufzeit.

7.7.1 Verbesserung: Mehrfachfilter für k–Mismatches

Die Idee zu folgendem Verfahren mit erwarteter sublinearer Laufzeit stammt von G. Myers, 1994. Die Verbesserung gegenüber der vorherigen Methode beruht darauf, dass größere Stücke von T, in denen kein k–Mismatch mit P auftreten kann, ausgeschlossen werden.

Definition 26 *Sei die Fehlerrate $0 \leq \varepsilon < 1$ vorgegeben. Dann nennen wir S einen ε–Match von T, wenn höchstens $\varepsilon|S|$ Fehler vorkommen.*

Algorithmus

1. Partitioniere P in Stücke der Länge $\log_\sigma m$.
2. Finde alle ε–Matches der Stücke in T.
3. Iterativ: Versuche, die Länge der ε–Matches bis zum ε–Match von P zu verdoppeln. Stücke, bei denen das nicht gelingt, scheiden aus. Es fallen $O(\log \frac{n}{\log_\sigma m})$ Iterationen an.

Den Schlüssel für die Laufzeitabschätzung des Algorithmus liefert das folgende Lemma.

Lemma 8 *Es sei α ein String mit der Zerlegung $\alpha = \alpha_0\alpha_1$, $|\alpha_0| = |\alpha_1|$, und β ein String. Dann gilt: Falls α ε–matches β, dann kann β zerlegt werden in $\beta = \beta_0\beta_1$ mit α_0 ε–matches β_0 oder α_1 ε–matches β_1.*

Damit kann man zeigen, dass die erwartete Gesamtlaufzeit des Algorithmus $O(km^{p(\varepsilon)} \log m)$ ist.

7.8 Ein kombinatorischer Algorithmus für *lcs*

Nachdem wir die longest common Subsequence zweier Strings S_1, S_2 schon einmal in Kapitel 7.2 mit Hilfe eines gewichteten Alignment–Problems in $O(|S_1| \cdot |S_2|)$ Zeit gelöst haben, werden wir nun kombinatorische Methoden verwenden. Wir reduzieren das Problem auf ein anderes, nämlich longest increasing Subsequence (*lis*). Dabei haben wir eine Folge π von n Zahlen gegeben, und suchen eine längste monoton steigende Teilfolge bzw. deren Länge.

Bemerkung. Wir nennen eine Teilfolge aufsteigend bzw. fallend, wenn sie streng monoton wachsend bzw. schwach monoton fallend ist. Eine Menge

von fallenden Teilfolgen, in der jede Zahl von π vorkommt, nennen wir eine *Überdeckung*. Eine Überdeckung nennen wir *minimal*, wenn sie aus einer minimalen Anzahl von Teilfolgen besteht.

Es gilt: Ist U eine minimale Überdeckung, dann folgt $|U| = lis$.

Nun wollen wir einen Greedy–Algorithmus mit Laufzeit $O(n \log n)$ für das lis–Problem angeben:

Fange beim Einfügen der Zahlen links an. Füge die Zahl am unteren Ende des ersten Stapels ein, dessen unterste Zahl *größer* als die Zahl selber ist. Falls es keinen solchen Stapel gibt, beginne mit einem neuen Stapel.

Beispiel. Sei $\pi = \{5, 3, 4, 9, 6, 2, 1, 8, 7, 10\}$

5	4	9	8	10	Ein neues Element müsste in
3		6	7		die Folge 1, 4, 6, 7, 10
2					eingefügt werden.
1					

Da die kleinsten Elemente der fallenden Folgen aufsteigend angeordnet sind, geht das Einfügen mittels binärer Suche in $O(n \log n)$ Zeit.

Beobachtung. Es existiert eine lis mit je einem Element aus jeder fallenden Folge der Überdeckung, diese finden wir in $O(n)$ Zeit.

Wenn man alle Stapel „nebeneinanderstellt“ sieht man, dass es eine aufsteigende oder fallende Folge der Länge mindestens $\sqrt{n}$ in π gibt.

Die Reduktion geht nun folgendermaßen: Sei $|S_1| = m$, $|S_2| = n$, $n \leq m$, und Σ ein Alphabet. Dann definieren wir:

$$r(i) = \#\text{ Vorkommen des } i\text{–ten Zeichens von } S_1 \text{ in } S_2,$$

und $r = \sum_i r(i)$, $0 \leq r \leq mn$.

Damit bekommen wir lcs in $O(r \log n)$ Zeit wie folgt:

1. Für jedes x in S_1 bestimme die fallende Liste der Positionen von x in S_2.

2. Es sei $\pi(S_1, S_2) := S_1$, wobei jedes Zeichen durch die entsprechende Liste aus dem ersten Schritt ersetzt ist.

Beobachtung. Jede aufsteigende Folge in $\pi(S_1, S_2)$ definiert eine gemeinsame Teilsequenz und umgekehrt.

Beispiel. $S_1 = abacx$, $S_2 = baabca$.

Schritt 1: $a \mapsto 6, 3, 2$, $b \mapsto 4, 1$, $c \mapsto 5$
Schritt 2: $\pi(S_1, S_2) = 6, 3, 2, 4, 1, 6, 3, 2, 5$

Die Folge 2,3,5 in $\pi(S_1, S_2)$ entspricht der gemeinsamen Teilsequenz aac, 3,4,6 gehört zu abc.