5. Das längste gemeinsame Teilsequenzenproblem

Literatur:

A. Apostolico, String Editing and Longest Common Subsequences, in G. Rozenberg, A. Salomaa (eds.), Handbook of Formal Languages Vol. 2, Springer 1997, 361 - 398.

Claus Rick, Zur Rekonstruktion optimaler Lösungen in Verfahren der Dynamischen Programmierung für die Sequenzanalyse, Dissertation Bonn, 2000.

A.V. Aho, D.S. Hirschberg, J.D. Ullman: Bounds on the Complexity of the Longest Common Subsequence Problem, JACM 23 (1976), 1-12.

C.K. Wong, A.K. Chandra: Bounds for the String Editing Problem, JACM 23 (1976), 13-16.

D.S. Hirschberg, An Information-theoretic Lower Bound for the Longest Common Subsequence Problem, IPL 7 (1978), 40-41.

Sei $A := a_1 a_2 \dots a_m$ ein String über ein Alphabet Σ. Ein String $S := s_1 s_2 \dots s_\ell$ ist eine Teilsequenz von A, wenn wir S aus A mittels Streichen von $m-\ell$ Symbolen erhalten. D.h.,

$$s_1 s_2 \dots s_\ell = a_{i_1} a_{i_2} \dots a_{i_\ell}, \quad \text{wobei } 1 \le i_1 < i_2 < \dots < i_\ell \le m.$$

Beispiel:

Sei $A = abbaacdabcdddda$. Dann ist z.B. $S := acddda$ eine Teilsequenz von A. Es gibt mehrere Möglichkeiten, S aus A zu konstruieren.

◊

Sei $B := b_1 b_2 \dots b_n$ ein String über Σ. S ist eine gemeinsame Teilsequenz von A und B, falls S eine Teilsequenz von A und auch eine Teilsequenz von B ist.

Beispiel:

$A = a\,b\,b\,a\,a\,c\,d\,a\,b\,c\,d\,d\,d\,a$

$B = b\,b\,c\,c\,d\,a\,c\,d\,a\,a\,b\,a\,b\,b\,c$

Einbettung von S in A und B.

$S := bbcdacda$ ist eine gemeinsame Teilsequenz von A und B.

◆

Längste gemeinsame Teilsequenzproblem (LCS)

gegeben: $A := a_1 a_2 \dots a_m$, $B := b_1 b_2 \dots b_n \in \Sigma^+$

gesucht: längste gemeinsame Teilsequenz S von A und B. D.h., $|S'| \leq |S|$ für alle gemeinsame Teilsequenzen S' von A und B.

Frage:
Besteht ein Zusammenhang zwischen dem LCS-Problem und der Berechnung der Editierdistanz (Stringeditierproblem) zweier Strings?

Zur Beantwortung dieser Frage betrachten wir das Stringeditierproblem bezüglich der Operationen Einfügen bzw. Streichen eines einzelnen Symbols, wobei jeder Operation die Kosten 1 zugeteilt wird.

Bezeichne $d(A,B)$ die Editierdistanz von A und B bezüglich obiger Operationenmenge und Kostenfunktion. Sei $LCS(A,B)$ eine längste gemeinsame Teilsequenz von A und B. Folgender Satz stellt den Zusammenhang zwischen $d(A,B)$ und $|LCS(A,B)|$ her:

Satz 5.1
Seien $A := a_1 a_2 \dots a_m$ und $B := b_1 b_2 \dots b_n$ zwei Strings über einem Alphabet Σ. Dann gilt

$$d(A,B) = m + n - 2|LCS(A,B)|.$$

Beweis:
Wir beweisen $d(A,B) \leq m+n-2|LCS(A,B)|$ und dann $d(A,B) \geq m+n-2|LCS(A,B)|$.

a) $d(A,B) \leq m+n-2|LCS(A,B)|$.

Folgende Operationen transformieren A nach B:

- Streiche alle m Zeichen aus A und füge dann alle n Zeichen von B in der richtigen Reihenfolge ein.

Kosten: $m+n$

Beobachtung:
Falls A und B gleiche Zeichen enthalten und diese in A und B in derselben Reihenfolge vorkommen, dann können diese vom Streichen und Einfügen ausgenommen werden.

Ersparnis pro Zeichen: 2

Diese übereinstimmenden und in derselben Reihenfolge vorkommenden Zeichen bilden eine gemeinsame Teilsequenz von A und B.

$\Rightarrow$

Es können $2\,|LCS(A,B)|$ viele Operationen eingespart werden. Also gilt

$$d(A,B) \leq m+n-2\,|LCS(A,B)|.$$

b) $d(A,B) \geq m+n-2\,|LCS(A,B)|$.

Annahme:
In einer optimalen Editierfolge werden p Streichungen und q Einfügungen durchgeführt.

$\Rightarrow$

$$d(A,B) = p+q.$$

Dann gilt

$$m - p + q = n$$
$$\Leftrightarrow \quad m - p = n - q$$

Beobachtung:
A und B enthalten eine gemeinsame Teilsequenz der Länge $m-p$

Also gilt:

$$\begin{aligned} d(A,B) &= p+q \\ &= n+m-(m-p)-(n-q) \\ &= n+m-2(m-p) \\ &\geq n+m-2|LCS(A,B)|. \end{aligned}$$

Wie das allgemeine Stringeditierproblem kann auch das LCS-Problem unter Verwendung von dynamischer Programmierung gelöst werden. Die hierzu benötigte Rekursionsgleichung ist etwas einfacher als beim allgemeinen Stringeditierproblem.

Bezeichnungen:

Sei $A := a_1 a_2 \dots a_m$ ein String über Σ. Dann definieren wir für $0 \leq i \leq m$:

$$A_i := \begin{cases} \varepsilon & \text{falls } i = 0 \\ a_1 a_2 \dots a_i & \text{falls } i > 0. \end{cases}$$

Für $1 \leq i \leq m+1$ definieren wir

$$A^i := \begin{cases} a_i a_{i+1} \dots a_m & \text{falls } i \leq m \\ \varepsilon & \text{falls } i = m+1 \end{cases}$$

Sei $B := b_1 b_2 \dots b_n$ ein String über Σ. Dann definieren wir

$$L_{i,j} := \max \{ |S| \mid S \text{ ist eine gemeinsame Teilsequenz von } A_i \text{ und } B_j \}.$$

Dann gilt

$$L_{m,n} = |LCS(A,B)|.$$

Beobachtung:

a) $L_{i,0} = 0$ für $0 \leq i \leq m$ und
$L_{0,j} = 0$ für $0 \leq j \leq n$.

b) Für $i > 0$ und $j > 0$ setzt sich eine LCS $S = s_1 s_2 \dots s_\ell$ der Präfixe A_i und B_j von A bzw. B wie folgt zusammen:

1. Fall: $a_i = b_j$

Dann gilt: $s_\ell = a_i = b_j$
(ansonsten könnte a_i als $(\ell+1)$-tes Zeichen an S angehängt werden)

und

$s_1 s_2 \dots s_{\ell-1}$ ist eine längste Teilsequenz von A_{i-1} und B_{j-1}.

(Gäbe es eine längere, dann könnte durch Anhängen von a_i an diese ebenfalls eine LCS der Länge $> l$ konstruiert werden.)

2. Fall: $a_i \neq b_j$

Falls $s_l \neq a_i$, dann ist S eine LCS von A_{i-1} und B_j. Falls $s_l \neq b_j$, dann ist S eine LCS von A_i und B_{j-1}.

Insgesamt ergibt sich folgende Rekursionsgleichung:

$$L_{i,j} := \begin{cases} 0 & \text{falls } i = 0 \text{ oder } j = 0 \\ L_{i-1,j-1} + 1 & \text{falls } a_i = b_j \\ \max\{L_{i-1,j}, L_{i,j-1}\} & \text{falls } a_i \neq b_j \end{cases}$$

Somit erhalten wir folgenden Algorithmus:

Algorithmus LAEBER
Eingabe: Strings $A := a_1 a_2 \dots a_m$, $B = b_1 b_2 \dots b_n$, wobei $m \leq n$
Ausgabe: $|LCS(A,B)|$
Methode:

```
(1) for i := 0 to m
        do
            L(i,0) := 0
        od;
(2) for j := 0 to n
        do
            L(0,j) := 0
        od;
```

(3) for $i := 1$ to m
do
for $j := 1$ to n
do
if $a_i = b_j$
then
$L(i,j) := L(i-1, j-1) + 1$
else
$L(i,j) := \max\{L(i-1,j), L(i,j-1)\}$
fi
od
od;

(4) Ausgabe := $L(m,n)$.

Obiger Algorithmus verwendet eine $(m+1) \times (n+1)$-Matrix zur Speicherung der Werte $L_{i,j}$, $0 \leq i \leq m$, $0 \leq j \leq n$ von Teillösungen und hat die Laufzeit $O(m \cdot n)$.

Unter Verwendung der gefüllten $(m+1) \times (n+1)$-Matrix kann leicht in einem sogenannten Trace-Back-Schritt eine LCS(A,B) rekonstruiert werden. Hierzu starten wir in der Position $[m,n]$ der Matrix.

Annahme: Position $[i,j]$ wird gerade besucht.

Dann wird festgestellt, aus welchem der drei möglichen Vorgängerwerte der Wert $L(i,j)$ entstanden ist.

Falls $a_i = b_j$, dann setzen wir in Position $[i-1, j-1]$ die Suche fort. Falls $a_i \neq b_j$, dann ist $L(i,j)$ aus dem größten der Werte $L(i-1,j)$ und $L(i,j-1)$ entstanden und wir setzen in der korrespondierenden Position die Suche fort.

Wenn wir folgende rekursive Prozedur mit print-lcs(m,n) aufrufen, dann rekonstruiert diese eine optimale Lösung.

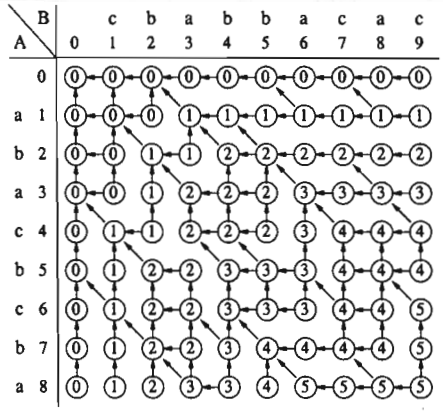
```
procedure print-lcs(i,j)
(1) if i=0 or j=0
      then
           exit
    fi;
(2) if a_i = b_j
      then
        print-lcs(i-1,j-1);
        print(a_i)
      else
        if L(i,j) = L(i-1,j)
          then
            print-lcs(i-1,j)
          else
            print-lcs(i,j-1)
        fi
    fi.
```

Alternativ könnte in einer weiteren $(m+1) \times (n+1)$-Matrix B schon beim Füllen der Matrix L durch Setzen entsprechender Zeiger in $B(i,j)$ gespeichert werden, wie $L(i,j)$ entstanden ist. Dann muss man zur Rekonstruktion einer LCS (A,B) nur noch ausgehend von $B(m,n)$ rückwärts den Zeigern folgen.

Beispiel:

A \ B		c	b	a	b	b	a	c	a	c
	0	1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0	0	0
a 1	0	0	0	1	1	1	1	1	1	1
b 2	0	0	1	1	2	2	2	2	2	2
a 3	0	0	1	2	2	2	3	3	3	3
c 4	0	1	1	2	2	2	3	4	4	4
b 5	0	1	2	2	3	3	3	4	4	4
c 6	0	1	2	2	3	3	3	4	4	5
b 7	0	1	2	2	3	4	4	4	4	5
a 8	0	1	2	3	3	4	5	5	5	5

(a)



(b)

Abbildung 2.4: Tabelle nach der Anwendung des Algorithmus (a) und korrespondierender Trace-Back-Graph (b).

◊

Übung:
Erweitern Sie die Prozedur print-lcs (i,j) bzw. den Algorithmus LAEBER, so dass alle LCS (A,B) rekonstruiert werden.

Wir können in der Matrix B die Positionen als Knoten und die Rückverweise als Kanten eines gerichteten, azyklischen Graphen interpretieren.

Der daraus resultierende Graph heißt Trace-Back-Graph.

Eigenschaften:

- Von jedem Knoten $[i,j]$ ist der Knoten $[0,0]$ über mindestens einen Pfad erreichbar.
- Jeder in $[m,n]$ beginnende und in $[0,0]$ endende Pfad spezifiziert die Einbettung einer LCS (A,B).
- Zu jeder Einbettung einer LCS (A,B) korrespondiert mindestens ein Pfad von $[m,n]$ nach $[0,0]$.

Nachteil des Trace-Back-Graphen:

- Der Speicherbedarf ist $\Theta(m \cdot n)$, obwohl viele Knoten des Graphen nicht von $[m,n]$ aus erreichbar sind.
- Darstellung ist unübersichtlich.

Folgender sogenannte LCS-Einbettungsgraph besitzt diese Nachteile nicht:

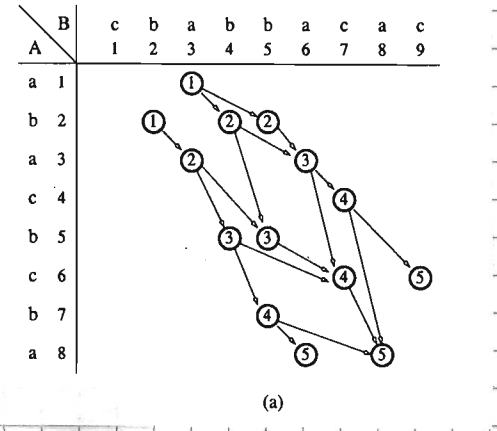
Seien $A := a_1 a_2 \dots a_m$ und $B := b_1 b_2 \dots b_n$ Strings über ein Alphabet Σ. Dann ist der LCS-Einbettungsgraph $G_E := (V, E)$ definiert durch:

$$V := \{ [i,j] \mid 0 \leq i \leq m,\ 0 \leq j \leq n,\ a_i = b_j \text{ und } \exists \text{ Pfad von } [m,n] \text{ nach } [i,j] \text{ im Trace-Back-Graphen } B \}$$

und

$$E := \{ ([i,j],[i',j']) \in V \times V \mid L(i,j) = L(i',j') - 1 \text{ und } \exists \text{ Pfad von } [i',j'] \text{ nach } [i,j] \text{ in } \mathcal{B} \}$$

Beispiel:



(a)

◊

Eigenschaften:

Wenn $p := |LCS(A,B)|$, dann lässt sich die Knotenmenge V in p disjunkte Teilmengen

$$V_k := \{ [i,j] \in V \mid L(i,j) = k \}, \quad 1 \leq k \leq p$$

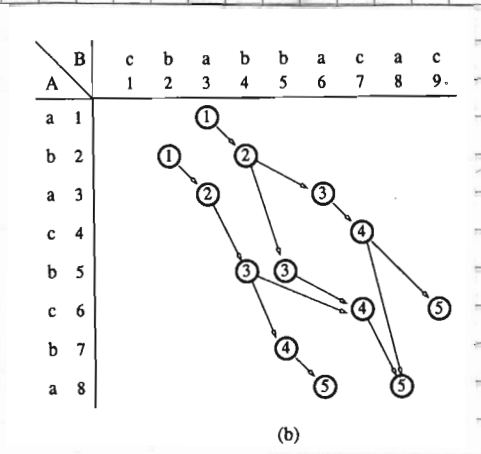
zerlegen.

Die Pfade von Knoten in V_1 zu Knoten in V_p korrespondieren in eindeutiger Weise zu Einbettungen längster gemeinsamer Teilsequenzen von A und B. Die zu den Knoten eines solchen Pfades korrespondierenden Zeichen in A und B zeigen an, welche Positionen in A und B zur Einbettung der korrespondierenden LCS(A,B) gehören.

Sei S eine gemeinsame Teilsequenz zweier Strings A und B. Falls wir bei der Konstruktion einer Einbettung von S in A und B stets von links nach rechts in A und B die erste Möglichkeit für das einzubettende Zeichen in S nehmen, dann erhalten wir die sogenannte kanonische Einbettung von S in A und B.

Mittels Entfernen von Knoten und Kanten aus G_E erhalten wir den kanonischen Einbettungsgraphen $G_{kE} := (V', E')$, der exakt die kanonischen Einbettungen der längsten gemeinsamen Teilsequenzen von A und B enthält.

Beispiel (Fortführung):



(b)

Alle bisher entwickelten Algorithmen benötigen gleichzeitig $\Theta(m \cdot n)$ Zeit und $\Theta(m \cdot n)$ Platz. Bei sehr langen Strings A und B verbietet sich mitunter ein Platzbedarf von $\Theta(m \cdot n)$. Somit drängt sich folgende Frage auf:

Welche der folgenden LCS-Probleme können wir auch bei Verwendung von lediglich linearem, d.h., $O(m+n)$ Platz in $O(m \cdot n)$ Zeit lösen?

1) Berechne $|LCS(A,B)|$.
2) Berechne $|LCS(A,B)|$ und konstruiere eine beliebige $LCS(A,B)$.
3) Berechne $|LCS(A,B)|$ und eine kompakte Repräsentation aller $LCS(A,B)$.

Problem 1 kann leicht gleichzeitig in linearem Platz und $O(m \cdot n)$ Zeit gelöst werden. Der Algorithmus LAEBER benötigt zur Berechnung von $L(i,j)$

- den Wert $L(i,j-1)$ aus derselben Zeile und
- die Werte $L(i-1,j-1)$ sowie $L(i-1,j)$ aus der vorangegangenen Zeile.

Die Werte der davor liegenden Zeilen werden auch für keinen anderen der noch zu berechnenden Werte benötigt.

$\Rightarrow$

Es muss stets nur die aktuelle Zeile, die gerade berechnet wird, und die Vorgängerzeile gespeichert werden. Also genügt $O(n)$ Platz.

Problem 2 ist wesentlich schwieriger zu lösen. Hierzu können wir nicht unter Verwendung obiger Lösung für Problem 1 die bisherigen Methoden zur

Rekonstruktion einer LCS(A,B) verwenden, da die hierfür benötigte Information nicht mehr vorhanden ist. Da der Trace-Back-Graph $\Theta(m \cdot n)$ Platz benötigt, dürfen wir diesen nicht aufbauen. Jedoch werden bei der Rekonstruktion einer LCS(A,B) unter Verwendung des Trace-Back-Graphen nur $O(n+m)$ Knoten dieses Graphen betrachtet. Diese Beobachtung wirft folgende Frage auf:

Kann man mit Hilfe von Neuberechnungen eine LCS(A,B) in linearem Platz rekonstruieren, wobei die benötigte Gesamtzeit $O(m \cdot n)$ bleibt?

Idee:
Teile die beiden Eingabestrings A und B an einer "geeigneten" Stelle (i,j) in Präfixe $A_i := a_1 a_2 \dots a_i$ und $B_j := b_1 b_2 \dots b_j$ sowie in Suffixe $A^{i+1} := a_{i+1} \dots a_m$ und $B^{j+1} := b_{j+1} b_{j+2} \dots b_n$ auf. Berechne rekursiv eine $LCS(A_i, B_j)$ und eine $LCS(A^{i+1}, B^{j+1})$ und konkateniere diese.

Problem:
Nicht jedes Paar (i,j) ist ein geeigneter Aufteilungspunkt. Falls jede LCS(A,B) einen Match (i',j') mit $(i' \leq i) \wedge (j' > j)$ oder $(i' > i) \wedge (j' \leq j)$ enthält, dann wäre $LCS(A_i, B_j) \cdot LCS(A^{i+1}, B^{j+1})$ keine LCS(A,B).

$\Rightarrow$

Es darf nur in einem Punkt (i,j), so dass $LCS(A_i, B_j) \cdot LCS(A^{i+1}, B^{j+1})$ eine $LCS(A,B)$ ist, aufgeteilt werden.

Solch ein Paar (i,j) heißt <u>zulässiger Schnitt</u>.

<u>Bemerkung</u>

- Ein zulässiger Schnitt muss nicht zu einem Match korrespondieren.
- Zu jedem gewählten i existiert ein j, so dass (i,j) ein zulässiger Schnitt ist.

<u>Beispiel</u>:

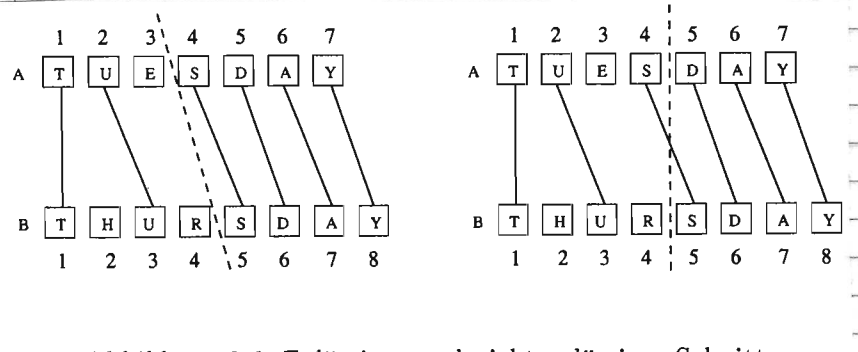


Abbildung 2.6: Zulässiger und nicht zulässiger Schnitt.

Bezeichne $T(A,B)$ die für die Konstruktion eines zulässigen Schnittes (i,j) der Strings A und B benötigte Zeit. Der Schnitt (i,j) heißt <u>balanciert</u>, falls

i) $T(A_i, B_j) + T(A^{i+1}, B^{j+1}) \leq \frac{1}{2} T(A,B)$ und

ii) die bzgl. A_i, B_j und A^{i+1}, B^{j+1} konstruierten Schnitte balanciert sind.

Beobachtung:

Für die Konstruktion aller Schnitte der 2^i Teilprobleme auf der i-ten Rekursionsstufe ergibt sich dann die Gesamtlaufzeit

$$\leq \left(\frac{1}{2}\right)^i \cdot T(A,B).$$

Demzufolge hat das gesamte Rekursionsschema folgende obere Schranke:

$$\sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^i \cdot T(A,B) = 2 \cdot T(A,B).$$

Die Laufzeit unseres Algorithmus zur Berechnung der Länge einer LCS ist proportional zum Produkt der Längen m und n der beiden Eingabestrings A und B.

Wie wir oben beobachtet haben, existiert für jedes i ein j, so dass der Schnitt (i,j) zulässig ist.

↝

Idee:
Teile das aktuelle Teilproblem bzgl. A stets in der Mitte und berechne bzgl. B eine Position, so dass sich ein zulässiger Schnitt ergibt.

Annahme:
A wird bei $\frac{m}{2}$ und B bei n' aufgeteilt.

Dann ergibt sich für die Lösung beider Teilpro-

benne eine Laufzeit von

$$\frac{m}{2} n' + \frac{m}{2}(n - n') = \frac{m}{2} n .$$

Frage:
Wie finden wir mit Hilfe unseres Basisalgorithmus in $O(n \cdot m)$ Zeit ein n', so dass $(i,j) := (\frac{m}{2}, n')$ ein zulässiger Schnitt ist?

Hierzu sind zwei Methoden bekannt.

1. Die Backpointer-Methode

Idee

Unter Verwendung von nur $O(n)$ zusätzlichen Speicherplatz berechnen wir für die ausgewählte Zeile i, in welcher Spalte j' ein in $[m,n]$ beginnender optimaler Pfad die Zeile i passieren würde.

Durchführung:

Hierzu erweitern wir den Basisalgorithmus, der in $O(n)$ Platz und $O(n \cdot m)$ Zeit $L_{m,n}$ berechnet derart, dass dieser zusätzlich ein n' berechnet, so dass $(\frac{m}{2}, n')$ ein zulässiger Schnitt ist.

Für $\frac{m}{2} \leq k \leq m$ und $0 \leq j \leq n$ bezeichne

- $P(k,j)$ eine Position j' in Zeile $\frac{m}{2}$, durch die ein in $[k,j]$ beginnender Pfad im Trace-Back-Graphen $\mathcal{B}$ laufen würde.

Offensichtlich gilt

$$P(\tfrac{m}{2}, j) = j \quad \text{für } 0 \le j \le n.$$

Aus der Rekursionsgleichung für $L_{i,j}$ ergibt sich für $k > \frac{m}{2}$ folgende Rekursionsgleichung:

$$P(k,j) := \begin{cases} P(k-1, j-1) & \text{falls } a_k = b_j \\ P(k-1, j) & \text{falls } a_k \neq b_j \text{ und } L_{k,j} = L_{k-1,j} \\ P(k, j-1) & \text{falls } a_k \neq b_j \text{ und } L_{k,j} > L_{k-1,j} \end{cases}$$

$P(m,n)$ enthält schließlich das gesuchte j'.

Rekursionsgleichung für $P(k,j)$ $\Rightarrow$

$P(k,j)$ ergibt sich nur aus den P-Werten der aktuellen oder der vorangegangenen Zeile und die Entscheidung hierüber hängt nur von L-Werten dieser Zeilen ab.

$\Rightarrow$

Ein zusätzliches Speicher der Größe $2(n+1)$ reicht für die Verwaltung der P-Werte aus.

Insgesamt haben wir gezeigt, dass für $i = \frac{m}{2}$ in $O(n \cdot m)$ Zeit und $O(n)$ Platz ein j' mit (i,j') ist ein zulässiger Schnitt, bestimmt werden kann.

2. Die Symmetriemethode

Bezeichne $\hat{A} := a_m a_{m-1} \dots a_1$ bzw. $\hat{B} := b_n b_{n-1} \dots b_1$ die reversen Strings von A bzw. B. Offensichtlich ist der reverse String einer LCS(A,B) eine LCS($\hat{A}, \hat{B}$).

Idee:
Wir berechnen gleichzeitig alle Werte L(i,j) und die Länge längster gemeinsamer Teilsequenzen zwischen A^{i+1} und jedem Suffix B^j, $1 \le j \le n+1$.
Wegen

$$|LCS(A,B)| = \max_{0 \le j \le n} \{ |LCS(A_i, B_j)| + |LCS(A^{i+1}, B^{j+1})| \}$$

gilt für j' mit

$$|LCS(A_i, B_{j'})| + |LCS(A^{i+1}, B^{j'+1})| = |LCS(A,B)|,$$

dass (i,j') ein zulässiges Schnitt ist.

Die Längen längster gemeinsamer Teilsequenzen von Suffixen können berechnet werden, indem wir unseren Algorithmus auf die reversen Strings

$\hat{A}_{m-i} := a_m a_{m-1} \dots a_{m-i}$ und $\hat{B} = b_n b_{n-1} \dots b_1$

ansetzen. Dann gilt offensichtlich stets $a_{m-i} = a_{i+1}$.

Bezeichne $\hat{L}_{k,\ell}$ die Länge einer längsten Teilsequenz zwischen einem Präfix der Länge k von $\hat{A}_{m-i}$ und einem Präfix der Länge ℓ von $\hat{B}$. Dann gilt für $0 \le k \le m-i$ und $0 \le \ell \le n$:

$$\hat{L}_{k,l} = |LCS(a_m a_{m-1} \dots a_{m-k+1}, b_n b_{n-1} \dots b_{n-l+1})|.$$

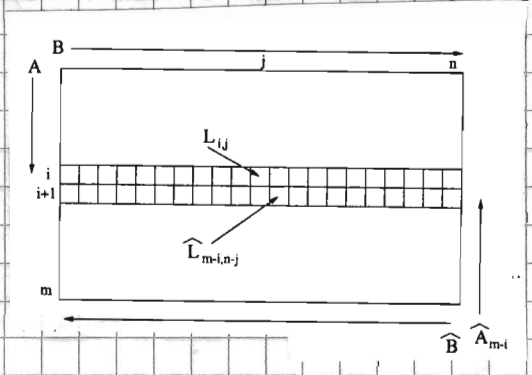
Dann ergeben sich die gesuchten Werte für die Aufgabe aus

$$|LCS(A^{i+1}, B^{j+1})| = \hat{L}_{m-i,n-j}.$$

Aufwandsanalyse:

Berechnung der Werte $L_{i,j}$, $0 \le j \le n$	$O(i \cdot n)$
Berechnung der Werte $\hat{L}_{m-i,n-j}$, $0 \le j \le n$	$O((m-i)n)$
Berechnung des Maximums	$O(n)$
Σ	$O(m \cdot n)$
Platzbedarf	$O(n)$.

Folgende Abbildung illustriert die Vorgehensweise:



Egal welche der beiden Methoden angewendet wird, die Rekursion sieht gleich aus.

Durchführung der Rekursion:

Annahme: Zulässiger Schnitt $(\frac{m}{2}, j)$ ist gefunden.

$\Rightarrow$

Aufwandsanalyse: (Platzbedarf)

Finden eines zulässigen Schnittes eines Teilproblems A_{i_1,i_2}, B_{j_1,j_2} $\quad O(j_2 - j_1) = O(n)$

Rekursionstiefe $\log m \Rightarrow$
zusätzlicher Platz zur Speicherung der Teilprobleme $\quad O(\log m)$

$\Sigma \quad O(n)$

Zwar verdoppelt sich die Laufzeit, bleibt jedoch weiterhin $\Theta(m \cdot n)$.

Effizientere Lösungsansätze:

Die bisher beschriebenen Methoden benötigen unabhängig von der Art der Eingabestrings $O(m \cdot n)$ Zeit. Falls die Eingabestrings gewisse strukturelle Eigenschaften besitzen, dann können diese zur Konstruktion von effizienteren Algorithmen genutzt werden.

Seien $A := a_1 a_2 \ldots a_m$ und $B := b_1 b_2 \ldots b_n$ die Eingabestrings. Die Menge M der Matches zwischen A und B ist dann definiert durch

$$M := \{ (i,j) \mid a_i = b_j, 1 \leq i \leq m, 1 \leq j \leq n \}$$

Bezeichne r die Gesamtanzahl der Matches zwischen A und B, d.h., $r = |M|$. Sei p

rekursive Anwendung der Strategie auf die Teilprobleme $A_{m/2}$, B_j sowie $A^{\frac{m}{2}+1}$, B^{j+1}.

Beobachtung:

- Ein Teilproblem ist spezifiziert durch ein Paar $A_{i_1,i_2} := a_{i_1} a_{i_1+1} \dots a_{i_2}$ und $B_{j_1,j_2} := b_{j_1} b_{j_1+1} \dots b_{j_2}$ von Teilstrings von A und B.
- Die Rekursion bricht ab, sobald ein Teilproblem trivial wird. D.h., sobald der zugehörige Teilstring von B leer ist ($j_2 < j_1$) oder der zugehörige Teilstring von A nur aus einem Zeichen besteht ($i_1 = i_2$). Im zweiten Fall wird im zugehörigen Teilstring B_{j_1,j_2} nach dem Vorkommen des Zeichens a_{i_1} gesucht und gegebenenfalls korrespondierende Position zur Lösung hinzugenommen.

Folgende Abbildung illustriert das Rekursionsschema:

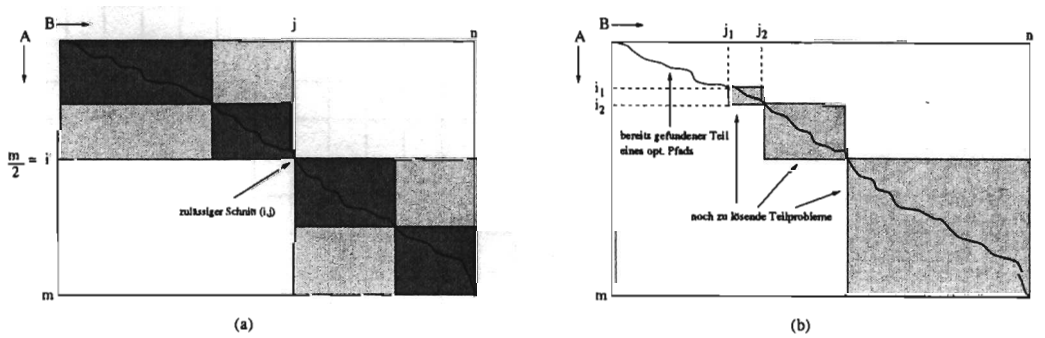


Abbildung 2.8: Rekursive Aufteilung mittels zulässiger Schnitte und Rekonstruktion einer LCS.

die Länge einer LCS(A,B). In der Vergangenheit wurden Algorithmen mit folgenden Laufzeiten konstruiert:

- $O(m + r \cdot \log p)$
- $O(n(m-p))$
- $O(p \cdot m)$
- $O(p(m-p))$ (Diplomarbeit Claus Rick)

Jeder dieser Algorithmen kann einem von folgenden zwei verschiedenen Lösungsansätze zugeordnet werden:

1. Kürzester Pfad im Editiergraphen

Ziel:
Im Editiergraphen will man einen Pfad vom Knoten [m,n] zum Knoten [0,0], der möglichst viele diagonale Kanten verwendet, finden.

2. Längste Folge von Matches

Die Menge M der Matches kann durch folgende sogenannte Match-Matrix veranschaulicht werden:

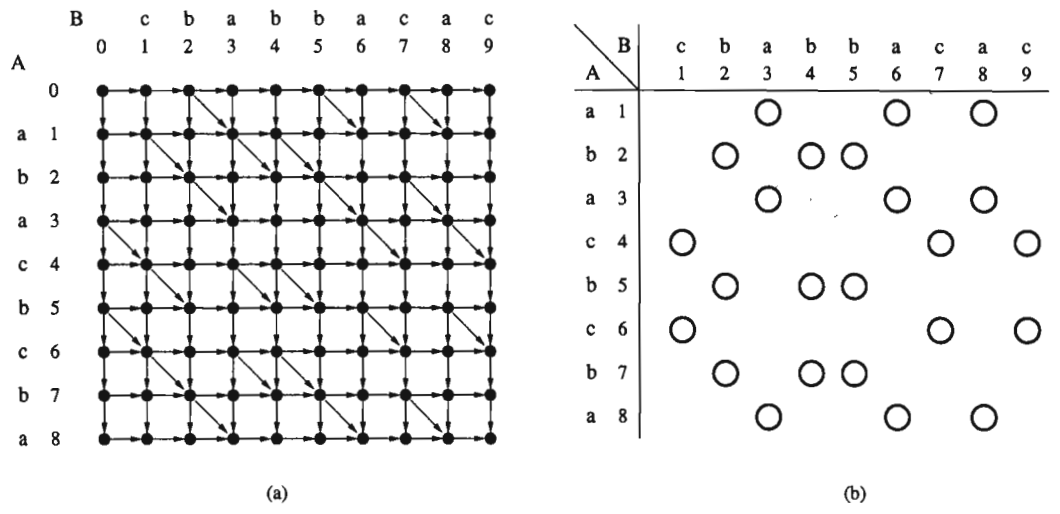


Abbildung 2.9: Editiergraph (a) und Match-Matrix (b).

Beobachtung:

- Jede Teilsequenz $S = s_1 s_2 \ldots s_e$ korrespondiert zu einer Folge $(i_1, j_1), (i_2, j_2), \ldots, (i_e, j_e)$ von Matches, die in beiden Komponenten streng monoton wachsend ist.
- Jede in beiden Komponenten streng monoton wachsende Folge von Matches korrespondiert zu einer Teilsequenz derselben Länge.

$\Rightarrow$

Die Konstruktion einer LCS(A,B) ist äquivalent zur Konstruktion einer längsten in beiden Komponenten streng monoton wachsenden Folge von Matches.

Ziel:

Ausarbeitung des 2. Lösungsansatzes und Verbesserung der Laufzeit für Eingaben, die spezielle

Eigenschaften besitzen.

Bezeichnungen:

Betrachten wir folgende Vorgängerrelation auf der Menge M von Matches:

$$(i,j) \prec (i',j') :\Leftrightarrow (i < i') \wedge (j < j').$$

Diese Relation definiert eine partielle Ordnung auf M.

Eine Teilmenge M' von M, so dass $\forall p, q \in M'$ mit $p \neq q$ entweder $p \prec q$ oder $q \prec p$ gilt, heißt Kette.

↝

Ziel:

Konstruktion einer längsten Kette.

Beobachtung:

- Der Wert $L_{i,j}$ gibt für einen Match (i,j) die maximale Länge einer in (i,j) endenden Kette an.
- Auch wenn wir uns nur für L-Werte bzgl. Matches interessieren, benötigt unsere bisherige Rekursionsgleichung zur Berechnung von $L_{i,j}$, dass alle Felder der L-Matrix ausgefüllt werden, was schließlich, unabhängig von der konkreten Eingabe, zu einer Laufzeit von $\Theta(n \cdot m)$ führt.

↝

Ziel:
Herleitung einer Rekursionsgleichung, die nicht das Ausfüllen der gesamten L-Matrix erfordert.

Bezeichnungen:

Ein Match (i,j) mit $L_{i,j} = k$ hat den Rang k.
Sei

$$C_k := \{ (i,j) \in M \mid L_{i,j} = k \}.$$

Wir vereinbaren, dass $C_0 := \{(0,0)\}$ und $(0,0) \lessdot (i,j)\ \forall (i,j) \in M$.
$(i,j) \in C_k$ heißt auch k-Match.

Idee:
Ausgehend von C_0 und mit Hilfe einiger Matches, deren Rang bereits bekannt ist, bestimme den Rang weiterer Matches.

Folgendes Lemma fasst einige einfache Eigenschaften zusammen:

Lemma 5.1

a) $(i,j) \in C_k \wedge (i,j) \lessdot (i',j') \Rightarrow (i',j') \in C_{k'}$ mit $k' \geq k+1$.

b) $\forall (i,j) \in C_k\ \exists (i',j') \in C_{k-1} : (i',j') \lessdot (i,j)$.

c) $(i,j), (i',j') \in C_k \Rightarrow (i,j) \not\lessdot (i',j')$ und $(i',j') \not\lessdot (i,j)$.

Bemerkung:

Aus Lemma 2.1 c) folgt direkt, dass sich die Matches $(i_1, j_1), (i_2, j_2), \dots, (i_\ell, j_\ell) \in C_\ell$ derart anordnen lassen, dass

$$i_1 \leq i_2 \leq \dots \leq i_\ell \text{ und } j_1 \geq j_2 \geq \dots \geq j_\ell.$$

Diese Anordnung heißt Standardsortierung von C_ℓ.

Anschaulich bedeutet dies, dass die Matches einer Klasse monoton von rechts oben nach links unten verlaufen. Dies veranschaulicht folgende Abbildung:

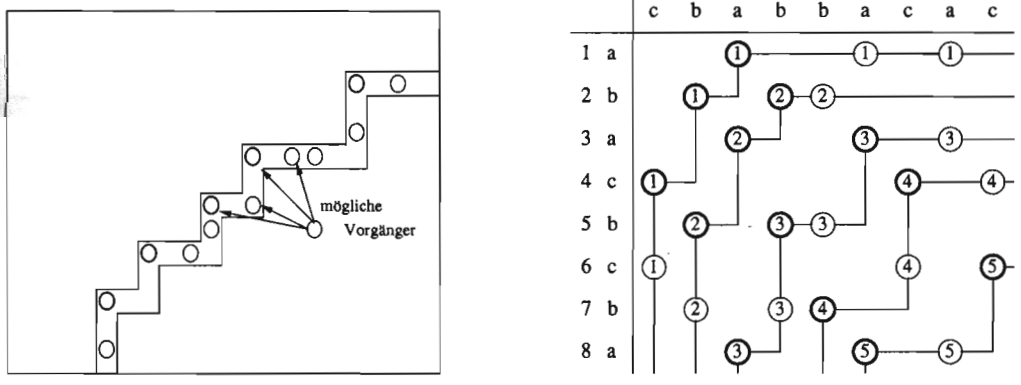


Abbildung 3.1: Die Matches einer Klasse legen eine Kontur fest.

Ist man in erster Linie an kanonischen Einbettungen interessiert, dann genügt es, in C_ℓ diejenigen Matches zu betrachten, die in einer Zeile am weitesten links und in einer Spalte am weitesten oben liegen. Dies legt folgende Definition nahe:

Seien $(i_1, j_1), (i_2, j_2) \in C_k$, $(i_1, j_1) \neq (i_2, j_2)$.
Der Match (i_1, j_1) <u>dominiert</u> genau dann den Match (i_2, j_2), wenn

$$(i_1 = i_2 \wedge j_1 < j_2) \text{ oder } (i_1 < i_2 \wedge j_1 = j_2).$$

Ein Match $(i,j) \in C_k$ heißt <u>dominant</u>, wenn er von keinem anderen Match in C_k dominiert wird. Seien

$$D_k := \{(i,j) \in C_k \mid (i,j) \text{ dominant}\}$$

und für $(i,j) \in D_k$

$$DOM(i,j) := \{(i',j') \in C_k \mid (i = i' \wedge j < j') \text{ oder } (i < i' \wedge j = j')\}$$

<u>Beobachtung:</u>

Konstruktion $\Rightarrow$ Die Matches in D_k reichen zur Generierung aller Matches in C_{k+1} aus. D.h.,

$$\forall (i,j) \in C_{k+1}\ \exists (i',j') \in D_k : (i',j') \preceq (i,j).$$

Demzufolge existiert immer eine LCS(A,B), die nur aus dominanten Matches besteht.

<u>Ziel:</u>
Effiziente Konstruktion einer obenartigen LCS(A,B).

Zum Erreichen dieses Zieles ist es nützlich, mit Hilfe der k-Matches diejenige Region abzugrenzen,

die sämtliche $(k+1)$-Matches enthält. Diese Region besteht aus allen Punkten (i,j), für die ein $(i',j') \in C_k$ mit $(i',j') \prec (i,j)$ existiert.

Die Matches in C_k liegen auf einer <u>Kontur</u> P_k, die diese Region nach oben und nach links abgrenzt. D.h.,

$$P_k := \{ (i,j) \mid L_{i,j} = k \wedge (L_{i-1,j} < k \vee L_{i,j-1} < k \vee L_{i-1,j-1} < k) \}$$

<u>Charakterisierung von P_k durch D_k:</u>

Betrachten wir zwei in der Standardsortierung benachbarte Matches (i,j) und (i',j'), $i < i'$ in D_k. Diese spezifizieren folgenden Teil der Kontur P_k:

(x,y) mit $(i \leq x \leq i') \wedge (y = j)$ und
(x,y) mit $(x = i') \wedge (j' \leq y \leq j)$.

$\Rightarrow$

i) Jeder dominante Match (i,j) trägt zur Spezifikation eines waagerechten und eines senkrechten Teils der Kontur bei.

ii) Die Region rechts/unterhalb der k-ten Kontur enthält alle Matches der Klassen C_ℓ, $\ell > k$. Die Region links/oberhalb der k-ten Kontur enthält alle Matches der Klassen C_ℓ, $\ell < k$.

Interpretation (Verlauf von P_k):

- Für einen Präfix A_i gibt der am weitesten links liegende Punkt (i,j) von P_k den kürzesten Präfix B_j an, so dass $|LCS(A_i, B_j)| = k$.

↝

Für jede Zeile $0 \leq i \leq m$ definieren wir:

$$T_{i,k} := \begin{cases} \min\{j \mid L_{i,j} = k\} & \text{falls } j \text{ existiert} \\ \infty & \text{sonst} \end{cases}$$

Sei

$$C_{i,k} := \{(s,t) \in C_k \mid 0 \leq s \leq i\}.$$

$T_{i,k}$ bezeichnet die am weitesten links stehende Spalte, die einen k-Match in einer Zeile $\leq i$ enthält.

⇒

Wir können alternativ $T_{i,k}$ wie folgt definieren:

$$T_{i,k} := \begin{cases} \min\{j \mid (i,j) \in C_{i,k}\} & \text{falls } C_{i,k} \neq \emptyset \\ \infty & \text{sonst} \end{cases}$$

Beobachtung:

Es gilt: $\infty = T_{0,k} \geqslant T_{1,k} \geqslant T_{2,k} \geqslant \cdots \geqslant T_{m,k}$.

Eigenschaften:

i) In den Zeilen $\leq i$ gibt es in den Spalten $< T_{i,k}$ keinen k-Match.

ii) Für einen $(k+1)$-Match (i,j) gilt notwendigerweise:

$$T_{i-1,k} < j.$$

iii) Es gilt stets: $T_{i+1,k} < T_{i,k+1}$.
Wegen $T_{i,k} \leq T_{i-1,k}$ folgt somit: $T_{i,k} < T_{i,k+1}$.

$\Rightarrow$

In einer festen Zeile i wachsen die linken Grenzen der Konturen verschiedener Klassen streng monoton.

$\Rightarrow$

Konturen zweier Klassen schneiden oder berühren sich niemals.

Für einen Match (i,j) gilt:

$$T_{i-1,k} < j \Rightarrow \text{Rang}(i,j) \geq k+1.$$

Frage: Wann gilt dann $\text{Rang}(i,j) = k+1$?

Damit (i,j) keinen Rang $> k+1$ besitzt, muss sichergestellt sein, dass er keinen Vorgänger mit Rang $> k$ besitzt. D.h., alle $(k+1)$-Matches in vorangehenden Zeilen liegen in Spalten $\geq j$.

$\Rightarrow$

$$T_{i-1,k+1} \geq j.$$

Insgesamt haben wir folgendes Lemma bewiesen:

Lemma 5.2

$$(i,j) \in C_k \iff T_{i-1,k-1} < j \leq T_{i-1,k}.$$

Somit erhalten wir folgende Rekursionsgleichung für $T_{i,k}$, $k \geq 1$, $1 \leq i \leq m$:

$$T_{i,k} := \begin{cases} \min\{ j \mid (a_i = b_j) \wedge (T_{i-1,k-1} < j \leq T_{i-1,k}) \} & \text{falls } j \text{ existiert} \\ T_{i-1,k} & \text{sonst} \end{cases}$$

$\Rightarrow$

Mittel dynamischer Programmierung können wir gleichzeitig die $T_{i,k}$-Werte und die Länge einer Matches berechnen.

Frage:
Gibt es eine Charakterisierung von dominanten Matches mit Hilfe der $T_{i,k}$-Werte?

Beobachtung:
Die dominanten k-Matches sind gerade diejenigen k-Matches, die bei einer zeilenweisen Betrachtung zu einer Verminderung der linken Konturengrenze führen.

Somit gilt folgendes Lemma:

Lemma 5.3

$(i,j) \in D_k \Leftrightarrow T_{i-1,k-1} < j \leq T_{i-1,k}$ und $j = T_{i,k}$.

Ziel:
Durchführung der dynamischen Programmierung.

Betrachten wir hierzu nochmals obige Rekursionsgleichung für $T_{i,k}$.

Eigenschaften:

i) Die Bedingung $T_{i-1,k-1} < j \leq T_{i-1,k}$ grenzt den Bereich, in dem das minimale j mit $a_i = b_j$ gesucht werden muss, ein.

ii) Es verbietet sich, jede Position daraufhin zu überprüfen

$\Rightarrow$

Wir benötigen eine Datenstruktur, die für eine gegebene Position j_0 und ein gegebenes $\sigma \in \Sigma$ das erste Vorkommen von σ in einer Position $\geq j_0$ in B liefert.

Falls hierfür konstante Zeit benötigt wird, dann lässt sich aus $T_{i-1,k-1}$ und $T_{i-1,k}$ der Wert $T_{i,k}$ in konstanter Zeit berechnen.

Bezeichne $\text{NEXT}_B : \Sigma \times \{1,2,\ldots,n\} \longrightarrow \{1,2,\ldots,n\}$ die korrespondierende Funktion. Wir berechnen dann

$$j_{min} := \text{NEXT}_B(a_i, T_{i-1,k-1}+1)$$

$$(*) \quad T_{i,k} := \begin{cases} j_{min} & \text{falls } j_{min} < T_{i-1,k} \\ T_{i-1,k} & \text{falls } j_{min} \geq T_{i-1,k} \end{cases}$$

Jenachdem, wie die $T_{i,k}$-Tabelle aufgefüllt wird, ergeben sich drei Berechnungsreihenfolgen. Folgende Abbildung illustriert diese:

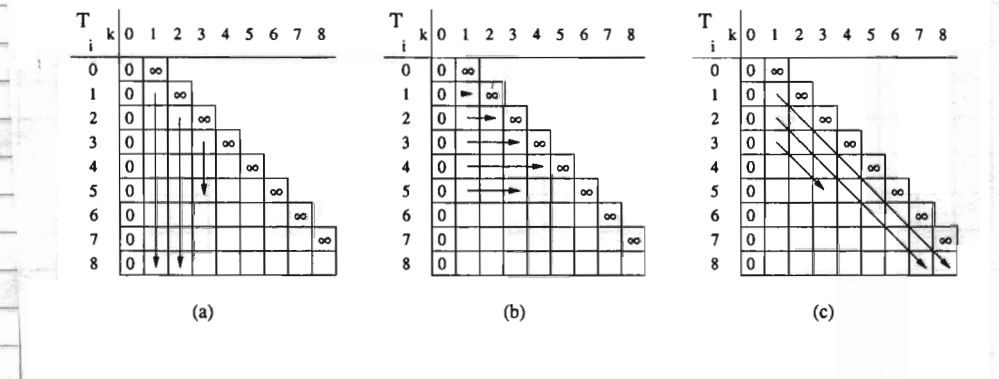


Abbildung 3.3: Drei mögliche Berechnungsreihenfolgen in der $T_{i,k}$-Tabelle.

- Da ein Präfix A_i von A keine LCS der Länge $> i$ mit B haben kann, bleibt die obere Dreiecksmatrix leer.
- Wird bei der Berechnung des Wert $T_{i-1,i}$ benötigt, dann verwenden wir hierfür den Wert ∞.
- Die linken Konkurrenzgrenzen der Klasse C_0 setzen wir in allen Zeilen auf 0. D.h.,

$$T_{i,0} := 0 \quad \text{für } 0 \leq i \leq m.$$

- Alle Berechnungen beginnen mit der Berechnung von $T_{1,1}$. Dies ist möglich, da die beiden benötigten

Vorgängerwerte ($T_{0,0} = 0$ und $T_{0,1} = \infty$) bekannt sind.

In Abhängigkeit, welche der drei möglichen Berechnungsreihenfolgen in der T-Tabelle wir nehmen, erhalten wir einen der folgenden drei Algorithmen:

a) Hirschberg: Konturenweises Vorgehen

Idee:
Beginne mit der Berechnung der k-ten Kontur erst, wenn die gesamte linke Grenze der $(k-1)$-ten Kontur bekannt ist.

Durchführung: ($k-1 \rightsquigarrow k$)

(1) Starte in derjenigen Zeile, die um eins tiefer liegt als die erste Zeile der $(k-1)$-ten Kontur.

(2) Unter Verwendung von (*) fülle die k-te Spalte der T-Matrix.

(3) Terminiere, sobald bei der Betrachtung einer Kontur kein einziger Match gefunden wurde.

Aufwandsanalyse

pro Phase $O(m)$

Anzahl der Phasen: $p+1$, wobei
$p := |LCS(A,B)|$.

Σ $O(p \cdot m)$.

b) Hunt/Szymanski: Zeilenweises Vorgehen

Idee:
Betrachte in der Match-Matrix vollständig eine Zeile nach der anderen. Führe dabei die linken Grenzen aller begonnenen Konturen fort. Überprüfe am Ende einer Zeile, ob in dieser Zeile eine neue Kontur beginnt

$\Rightarrow$
T-Matrix wird zeilenweise aufgefüllt.

Aufwandsanalyse:

pro Zeile	$O(p)$
Σ	$O(p \cdot m)$

c) Nakatsu et al.: Versetztzeilenweises Vorgehen

Idee:
Fülle die T-Matrix dergestalt auf, dass zunächst alle Positionen $T_{i,k}$ aufgefüllt werden, so dass eine gemeinsame Teilsequenz zwischen A_i und B der Länge k kein Zeichen von A_i nicht verwendet (d.h., $i-k=0$), ein Zeichen nicht verwendet ($d.h.$ $i-k=1$) u.s.w.

Obige Vorgehensweise entspricht einem diagonalweisen Auffüllen der T-Matrix.

Durchführung (Verbesserung durch Rick):

$i-k=0$:

Beginne mit dem ersten Zeichen von A und suche in B das erste passende Zeichen. Suche rechts von diesem Zeichen das zum zweiten Symbol in A passende Zeichen u.s.w., bis schließlich über das Ende von B hinausgegangen wird.

$\Rightarrow$

Es wird der längste Präfix von A, der eine Teilsequenz von B ist, bestimmt. Sei $\tilde{p}$ die Länge dieses Präfixes.

Interpretation:

Für jedes $1 \leq k \leq \tilde{p}$ wird von der k-Kontur der oberste dominante Match berechnet.

$i-k=1$:

Beginne mit dem 2. Zeichen von A und überprüfe, ob sich ein Match links der bisherigen linken Grenze der 1-Kontur, die sich in der ersten Zeile der Match-Matrix befindet, finden lässt.

allgemein:

Bestimme für jede begonnene k-Kontur die linke Grenze in Zeile k der Match-Matrix. Diese kann wieder mit Hilfe der

- linken Grenze der $(k-1)$-Kontur in Zeile k und
- der bisherigen Grenze der k-Kontur in Zeile k

berechnet werden.

In Zeilen $> \tilde{p}+1$ können sich noch weitere neue Konturen ergeben, wodurch sich gegebenenfalls die bisher bekannte untere Schranke für die Länge einer LCS(A,B) erhöht.

$\underline{i-k = g-1 \rightsquigarrow g}$:

Beginne in der Zeile $g+1$ mit der Berechnung von $T_{g+1,1}$ und fahre dann mit der Berechnung von $T_{g+2,2}$, $T_{g+3,3}$, ... fort.

Dieses Verfahren terminiert, sobald $g \geq m - \tilde{p}$. Dabei gilt $\tilde{p} = p := |LCS(A,B)|$, da beginnend bei $m - \tilde{p} + 1$ es nur noch $\tilde{p}$ viele Zeilen gibt, so dass sich der Wert von $\tilde{p}$ nicht mehr vergrößern kann.

Laufzeitanalyse:

Anzahl der Phasen	$\leq m-p+1$
Aufwand pro Phase	$O(p)$
$\sum$	$O(p(m-p))$

Bemerkung: In Originalarbeit: $O(n(m-p))$.

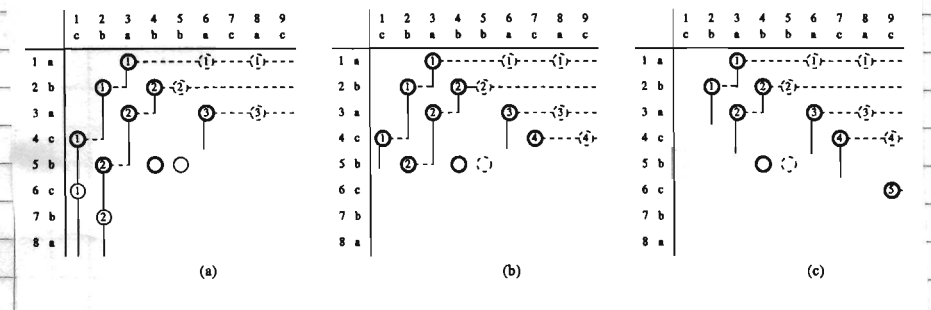


Abbildung 3.4: Die Berechnungsreihenfolgen in der Match-Matrix.
bzgl. der drei Lösungsansätzen.

Übung:
Arbeiten Sie die drei Lösungsansätze zu Algorithmen aus.

Beobachtung (Claus Rick):
Die Ineffizienz der beiden ersten Lösungsansätze rührt in erster Linie von der asymmetrischen Betrachtungsweise der Situation her.

- $T_{i,k}$-Werte spiegeln die linken Grenzen (d.h. senkrechten Teile) nicht jedoch die oberen Grenzen (d.h. waagerechten Teile) der Konturen wider. Demzufolge ermöglichen diese, das Betrachten nichtdominanter Matches in horizontalen Teilen der Konturen zu vermeiden.
- Nichtdominante Matches in vertikalen Teilen der Konturen werden weiterhin betrachtet.
- Verfolgt man eine Kontur von rechts/oben nach links/unten in der Match-Matrix, so stellt man fest, dass nichtdominante Matches am Anfang in erster Linie in horizontalen Teilen der Kontur vorkommen. Am Ende der Kontur häufen sich diese in den vertikalen Teilen.

Idee (Claus Rick):
- Analog zu den $T_{i,k}$-Werten könnte man mit Hilfe von $T'_{j,k}$-Werten obere Grenzen der

Konturen definieren, was zur Vermeidung explizi-
ter Betrachtung nichtdominanter Matches
in vertikalen Teilen der Konturen führen würde.

- Arbeite mit beiden Werten simultan und be-
rechne Konturen gleichzeitig von beiden En-
den her.

Durchführung:

Wir definieren für jede Spalte $0 \leq j \leq n$:

$$T'_{j,k} := \begin{cases} \min\{i \mid L_{i,j} = k\} & \text{falls } i \text{ existiert} \\ \infty & \text{sonst.} \end{cases}$$

$T'_{j,k}$ bezeichnet die oberste Zeile, die einen k-Match in einer Spalte $\leq j$ enthält, falls solche existiert.

Übung:
Stellen Sie bezüglich $T'_{j,k}$ analog diejenige Überle-
gungen an, die wir bezüglich $T_{i,k}$ durchgeführt haben.

Ziel:
Berechnung aller k-Matches (i,j) mit $i \leq j$ mittels
Verwendung der $T_{i,k}$-Werten und zeilenweisen
Vorgehens und aller k-Matches (i,j) mit $i > j$
unter Verwendung der $T'_{j,k}$-Werten und spalten-
weisen Vorgehens. Dabei soll zwischen der Be-
trachtung einer Zeile und einer Spalte abge-
wechselt werden.

Folgende Abbildung veranschaulicht diese Vorgehensweise:

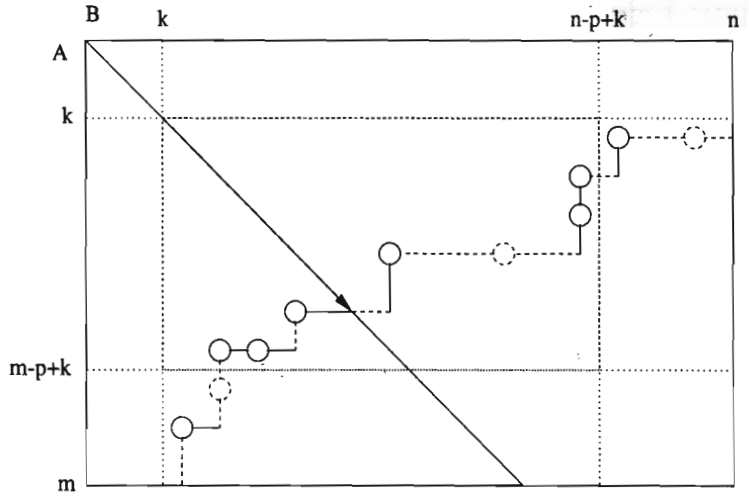


Abbildung 3.5: Berechnung einer Kontur durch abwechselnde Betrachtung von Zeilen und Spalten. Termination bei Schnitt der Diagonalen mit einem der beiden Konturteile.

durchgezogenen Linien ≙ tatsächlich betrachteten Grenzen der Kontur.

Der Algorithmus bewegt sich entlang der Hauptdiagonalen der Match-Matrix. Hierbei wird in jeder Position nach rechts eine Zeile und nach unten eine Spalte bearbeitet. Jenachdem, ob wir Hirschbergs oder Hunt/Szymanskis Methode verwenden, werden hierbei eine oder mehrere Konturen fortgeführt.

Frage:
Wann ist eine Kontur vollständig bestimmt?

Zur Beantwortung dieser Frage betrachten wir denjenigen Zeitpunkt, in dem die Diagonale einen der beiden bis zu diesem Zeitpunkt

konstruierten Konturteils schneidet.

Interpretation:
Die aktuelle linke Grenze des Konturteils rechts/oben ist auf oder links von der Diagonalen oder die aktuelle obere Grenze des Konturteils links/unten ist auf oder oberhalb der Diagonalen.

Beobachtung:
Der diese Grenze definierende dominante Match würde Vorgänger eines jeden anschließend ermittelten Matches sein.

$\Rightarrow$

Zum betrachteten Zeitpunkt ist die Kontur vollständig bestimmt.

$\rightsquigarrow$

Um eine Kontur P_k zu bestimmen, berechnen wir für $i = k, k+n, \ldots$ jeweils die Werte $T_{i,k}$ und $T'_{i,k}$, bis wir feststellen, dass $T_{i,k} = i$ oder $T'_{i,k} = i$.

Folgendes Lemma garantiert, dass dies für ein $i \leq n - p + k$ der Fall sein wird.

Lemma 5.4
Seien $|A| = m$, $|B| = n$, $m \leq n$ und $|LCS(A,B)| = p$. Dann existiert ein dominanter k-Match (x,y) mit $k \leq x \leq m - p + k$ und $k \leq y \leq n - p + k$.

Beweis:

Da stets eine mer aus dominanten Matches bestehende LCS existiert, gibt es einen dominanten k-Match. Jeder k-Match ist das k-te Element einer in beiden Komponenten streng monoton wachsenden Folge von Matches. Also gilt

$$k \leq x \quad \text{und} \quad k \leq y.$$

In einer LCS der Länge p muss dem k-Match (x,y) eine Kette von $p-k$ weiteren Matches folgen. Dies ist mer möglich, falls

$$x \leq m - (p-k) = m - p + k \quad \text{und}$$
$$y \leq n - (p-k) = n - p + k.$$

■

Lemma 5.4 $\Rightarrow$

Die k-te Kontur wird spätestens bei

$$i := \max\{x, y\} \leq n - p + k$$

von der Diagonalen geschnitten.

$\Rightarrow$

Für die k-te Kontur werden maximal

$$(n - p + k) - (k - 1) = n - p + 1$$

$T_{i,k}$- und $T'_{i,k}$-Werte bestimmt, was zu einer Gesamtlaufzeit von

$$O(p \cdot (n-p))$$

führt.

Da für jede der maximal m Positionen auf der Hauptdiagonalen pro Kontur höchstens ein $T_{i,k}$ - Wert und höchstens ein $T'_{i,k}$ - Wert berechnet wird, ist $O(p \cdot m)$ auch eine obere Schranke für die Gesamtlaufzeit.

$\Rightarrow$

Das Verfahren hat eine Gesamtlaufzeit von

$$O(n \cdot |\Sigma| + \min\{p \cdot m, p(n-p)\}).$$

Bemerkung:

In seiner Dissertation zeigt Claus Rick, wie unter Beibehaltung der obigen Zeitkomplexität eine LCS(A,B) in linearem Platz bestimmt werden kann.