Anfrage-Sprachen

Teil 2

Viele Folien in diesem Abschnitt sind eine deutsche Übersetzung der Folien von Raymond J. Mooney (http://www.cs.utexas.edu/users/mooney/ir-course/). Ein weiterer großer Anteil wurde mit freundlicher Genehmigung von Peter Becker übernommen (http://www2.inf.fh-rhein-sieg.de/~pbecke2m/retrieval/).

Forts. Bayer-Moore-Algorithmus

- Die entsprechenden Werte werden in einer Preprocessingphase ermittelt und in der *Shift-Tabelle D* abgelegt.
- $D[j] := \min_{s>0}\{s \mid$ (BM1) und (BM2) gilt für j und $s\}$
- Der Algorithmus von Boyer und Moore verwendet nun im Falle eines Mismatches an Position j den in $D[j]$ abgelegten Wert, um *pat* nach rechts zu verschieben.

Algorithmus 6.2 [Algorithmus von Boyer und Moore]

$i := 1$

while $i \leq n - m + 1$ **do**

  $j := m$

  while $j \geq 1$ **and** $pat[j] = \text{text}[i + j - 1]$ **do**

    $j := j - 1$ **end**

  if $j = 0$ **then return true**

  $i := i + D[j]$

end

return false

Beispiel

Definition 6.1 Der n-te *Fibonacci-String* Fn ($n \geq 0$) ist wie folgt definiert:

- $F_0 = \varepsilon$
- $F_1 = b$
- $F_2 = a$
- $F_n = F_{n-1}F_{n-2}$ für $n > 2$

- $D[j]$ lautet für F_7:

j	1	2	3	4	5	6	7	8	9	10	11	12	13
$pat[j]$	*a*	*b*	*a*	*a*	*b*	*a*	*b*	*a*	*a*	*b*	*a*	*a*	*b*
$D[j]$	8	8	8	8	8	8	8	3	11	11	6	13	1

Algorithmus 6.2 kann noch weiter verbessert werden: Tritt an Stelle *m* von *pat* (also bereits beim ersten Vergleich) ein Mismatch auf, so wird *pat* momentan nur um eine Stelle nach rechts verschoben.

Es sei

$$last[c] := \max_{1 \le j \le m} \{j \mid pat[j] = c\}$$

und $last[c] := 0$ falls c nicht in *pat* auftritt.

$last[c]$ gibt für ein $c \in \Sigma$ die jeweils letzte Position von c in *pat* an. Kommt es nun an Stelle j zu einem Mismatch, kann statt

$$i := i + D[j]$$

die Anweisung

$$i := i + \max(D[j];\, j - last[text[i + j - 1]])$$

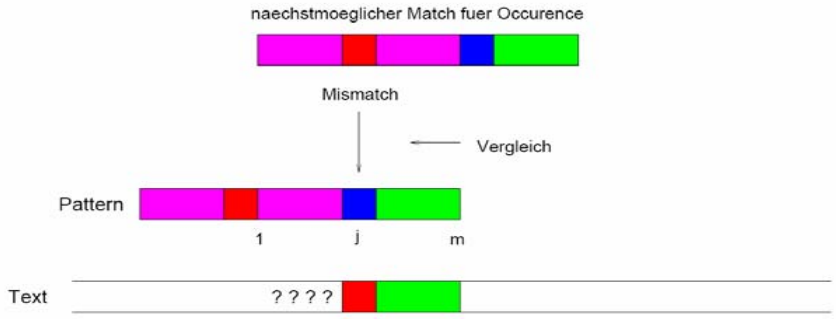
verwendet werden. Damit ergeben sich noch größere Verschiebungen.

Bemerkungen

- Verschiebungen der Länge $j - last[text[i+j-1]]$ heißen *Occurrence-Shift*.
- Wird nur der Occurence-Shift verwendet, d.h. die Verschiebeanweisung lautet
  $$i := i + \max(1, j - last[text[i + j - 1]])$$
  so spricht man von einem *vereinfachten Boyer-Moore-Algorithmus*.
- DieWorst-Case-Laufzeit des vereinfachten Boyer-Moore-Algorithmus beträgt O(*nm*).
- Auf gewöhnlichen Texten verhält sich die vereinfachte Version i.d.R. nur marginal schlechter als die ursprüngliche Version.

Bemerkungen

- Auf gewöhnlichen Texten verhält sich die vereinfachte Version i.d.R. nur marginal schlechter als die ursprüngliche Version.
- Bei kleinem $|\Sigma|$ ist die Occurence-Heuristik i.d.R. nutzlos.



Berechnung der Shift-Tabelle

Es bleibt das Problem, die Shift-Tabelle D zu berechnen. Dies geschieht in zwei Phasen. In der ersten Phase werden nur Verschiebungen der Form $D[j] < j$ betrachtet. Für solche Verschiebungen muss gelten:

(BM1) $pat[j+1 \ldots m] = pat[j+1-s \ldots m-s]$

(BM2) $pat[j-s] \neq pat[j]$

Veranschaulichung:

```
           pat:  *  *  *  *  b  a  b  a  a  b  a  a  b
pat:  *  *  *  *  b  a  b  a  a  b  a  a  b ← s →
                  ↑        ↑
                j − s      j
```

Berechnung der Shift-Tabelle

Ein Suffix von *pat* muss mit einem inneren Teil von *pat* übereinstimmen:

Vorgehensweise für die erste Phase: man berechnet (für $0 \le j \le m$) $rborder[j]$ mit

$$rborder[j] := \max_{1 \le k \le m-j} \{k \mid pat\,[j+1 \ldots j+k-1] = pat\,[m-k+2 \ldots m]\}$$

bzw.

$$rborder[j] := \max_{1 \le k \le m-j} \{k \mid pat\,[j+1 \ldots j+k-1] \text{ ist echter Suffix von } pat[j+1 \ldots m]\}$$

Weiterhin gelte $rborder[m] = 0$.

Beispiel

$rborder[j]$ lautet für F_7:

j	0	1	2	3	4	5	6	7	8	9	10	11	12	13
$pat[j]$		a	b	a	a	b	a	b	a	a	b	a	a	b
$rborder[j]$	6	5	4	3	2	6	5	4	3	2	1	1	1	0

Berechnung der Shift-Tabelle

```
i := m – 1
j := 1
while i >= 0 do
        while i – j + 1 >= 1 and pat[m – j + 1] = pat[i – j + 1] do
                j := j + 1
                rborder[i – j + 1] = j
        end
        i := i – j + rborder[m – j + 1]
        j := max(rborder[m – j + 1]; 1)
end
```

Wegen (BM2) treten relevante Situationen zur Berechnung von $D[j]$ nur im Falle eines Mismatchs in der inneren While-Schleife auf. Dementsprechend wird zur Berechnung von $D[j]$ der Algorithmus hinter der inneren While-Schleife um die folgenden Anweisungen erweitert:

Berechnung der Shift-Tabelle

```
if j > 1 then
s := m – i
t := i – j + 1
   if t + s > s then
        D[t + s] = min(s;D[t + s])
   endif
endif
```

Veranschaulichung:

```
          pat:  *   *   *   *   b   a   b   a   a   b   a   a   b
pat: *   *   *   *   b   a   b   a   a   b   a   a   b    ← s →
                     ↑          ↑     ↑
                t = i – j + 1   t + s  i
```

Damit steht der Algorithmus für die erste Phase.

Beispiel

D[j] nach der ersten Phase für den String F_7:

j	1	2	3	4	5	6	7	8	9	10	11	12	13
$pat[j]$	a	b	a	a	b	a	b	a	a	b	a	a	b
$D[j]$	13	13	13	13	13	13	13	3	13	13	6	13	1

Berechnung der Shift-Tabelle

In der zweiten Phase werden Verschiebungen mit $D[j] \geq j$ betrachtet. Für solche Verschiebungen muss die Bedingung

(BM1) $pat[1 \dots t-1] = pat[m-t+2 \dots m]$ mit $j \leq m-t+1$

gelten. (BM2) ist in dieser Phase stets wahr und braucht nicht weiter betrachtet zu werden.

Veranschaulichung:

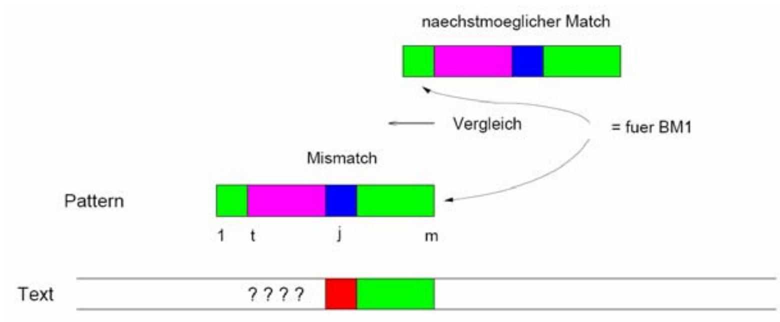
pat: a b a a b a b a a b a a b

pat: a b a a b a b a a b a a b $\leftarrow s = m - t + 1 \rightarrow$

↑ ↑ ↑

j $t-1$ $m-t+2$

Berechnung der Shift-Tabelle



- Es werden nun in absteigender Reihenfolge mögliche Werte für t betrachtet, und $D[j]$ wird entsprechend korrigiert.
- Der größte mögliche Wert für t ergibt sich durch *rborder*[0] (Länge des längsten Substrings, der sowohl echter Präfix als auch echter Suffix ist).

Berechnung der Shift-Tabelle

Die weiteren möglichenWerte für t ergeben sich durch $rborder[m-t+1]$.

```
t := rborder[0]
l := 1
while t > 0 do
    s = m – t + 1
    for j := l to s do
        D[j] := min(D[j], s)
    end
    t := rborder[s]
    l := s + 1
end
```

Algorithmus 6.3 (Berechnung der Shift-Tabelle für Boyer-Moore)

```
/* Initialisierung */
rborder[m] := 0; D[m] := 1
for j := m – 1 down to 0 do
    rborder[j] = 1; D[j] = m
end
/* Phase 1 */
i := m – 1
j := 1
while i >= 0 do
    while i – j + 1 >= 1 and pat[m – j + 1] = pat[i – j + 1] do
        j := j + 1
        rborder[i – j + 1] = j
    end
    if j > 1 then
        s := m – i
        t := i – j + 1
        if t + s > s then
            D[t + s] = min(s;D[t + s])
        endif
    endif
    i := i – j + rborder[m – j + 1]
    j := max(rborder[m – j + 1]; 1)
end
```


Algorithmus 6.3 (Berechnung der Shift-Tabelle für Boyer-Moore)

```
/* Phase 2 */
t := rborder[0]
l := 1
while t > 0 do
    s = m – t + 1
    for j := l to s do
        D[j] := min(D[j]; s)
    end
    t := rborder[s]
    l := s + 1
end
```


Beispiel: Algorithmus von Boyer und Moore

Der String F_7 wird in dem String abaababaabacabaababaabaab gesucht.

```
                        a b a a b a b a a b a a b
                a b a a b a b a a b a a b
            a b a a b a b a a b a a b
  a b a a b a b a a b a a b
a b a a b a b a a b a a b

a b a a b a b a a b a c a b a a b a b a a b a a b
```


Beispiel 4.8. [Shift-Tabellen für Boyer/Moore]

datenbank	retrieval	compiler
999999991	999999991	88888881

kuckuck	rokoko	papa	abrakadabra
3336661	662641	2241	77777771131
			00

Satz 4.5. *Algorithmus 6.2 löst Problem 6.1(a) in Zeit $O(n + m)$ und Platz $O(m)$.*

Bemerkungen

- Als scharfe obere Schranke für die Anzahl an Zeichenvergleichen ergibt sich $3n$.
- Würde man statt (BM1) und (BM2) nur (BM1) verwenden, so wäre keine lineare Laufzeit mehr gewährleistet ($O(mn)$).
- Sucht man mit dem Algorithmus von Boyer und Moore nach allen Matches für *pat* in *text*, so ist die Laufzeit ebenfalls $O(mn)$.

Reguläre Ausdrücke

- Sprache zur Bildung von komplexen Strukturen aus einfacheren.
  - Ein individuelles Zeichen ist ein regex (regulärer Ausdruck) .
  - Vereinigung: Wenn e_1 und e_2 regexes sind, dann ist (e_1 | e_2) ein regex, der zu allem passt, was zu e_1 oder zu e_2 passt.
  - Verknüpfung: Wenn e_1 und e_2 regexes sind, dann ist e_1 e_2 ein Regex, der zu einem String passt, der aus einem Substring besteht, der zu e_1 passt, sofort gefolgt von einem Substring, der zu e_2 passt.
  - Wiederholung (Kleene-Abschluss): Wenn e_1 ein regex ist, dann ist e_1* ein regex, der zu einer Folge von null oder mehr Strings passt, die zu e_1 passen.

Beispiele regulärer Ausdrücke

- (u|e)nabl(e|ing) passt zu
  - unable
  - unabling
  - enable
  - enabling
- (un|en)*able passt zu
  - able
  - unable
  - unenable
  - enununenable

Erweiterte Regex's (Perl)

- Perl enthält spezielle Terme für bestimmte Zeichentypen, wie alphabetische oder numerische oder allgemeine "Wildcards".
- Spezieller Wiederholungsoperator (+) für 1 oder mehrere Vorkommen.
- Spezieller optionaler Operator (?) für 0 oder 1 Vorkommen.
- Spezieller Wiederholungsoperator für spezifische Anzahl von Vorkommen : {min,max}.
  - A{1,5} - ein bis fünf A's.
  - A{5,} - fünf oder mehr A's
  - A{5} - genau fünf A's

Perl

- Zeichenklassen:
  - \w (word char) jedes alpha-numerische Zeichen (nicht: \W)
  - \d (digit char) jede Zahl (nicht: \D)
  - \s (space char) jeder Zwischenraum (nicht: \S)
  - . (wildcard) alles
- Ankerpunkte:
  - \b (boundary) Wortgrenze
  - ^ Beginn eines Strings
  - $ Ende eines Strings

Perl-Regex-Beispiele

- U.S. Tel.Nr. ohne optionalen Bereichscode:
  - /\b(\(\d{3}\)\s?)?\d{3}-\d{4}\b/
- (amer.) Email-Addresse:
  - /\b\S+@\S+(\.com|\.edu|\.gov|\.org|\.net)\b/

Hinweis: Pakete zur Unterstützung von Perl regex's sind in Java verfügbar.

Approximatives String-Matching

- Was ist, wenn ein Dokument Tippfehler oder falsche Buchstaben enthält?
- Ähnlichkeitsmaße zwischen Wörtern (beliebigen Strings):
  - Editier-Abstand (Levenstein distance)
  - Längste gemeinsame Teilfolge (longest common substring, LCS)
- Suche alle Strings, die näher sind als ein vorgebener Schwellwert.

Approximatives String-Matching

- Bisher haben wir String-Matching-Probleme betrachtet, bei denen das Muster *pat* exakt mit einem Substring von *text* übereinstimmen musste.
- Beim Matching von regulären Ausdrücken lässt man zwar Varianten zu, aber ebenfalls keine Fehler.
- In vielen praktischen Fällen ist es wünschenswert, die Stellen von *text* zu finden, die mit *pat* "nahezu" übereinstimmen, d.h. man erlaubt Abweichungen zwischen *pat* und *text*.

Anwendungsbeispiele

- Molekularbiologie (Erkennung von DNA-Sequenzen)
- Ausgleich verschiedener Schreibweisen (Grafik vs. Graphik)
- Ausgleich von Beugungen
- Toleranz gegenüber Tippfehlern
- Toleranz gegenüber OCR-Fehlern

String-Metriken

- Der Begriff “nahezu” wird durch eine Metrik auf Strings formalisiert.
- Zur Erinnerung: Sei M eine Menge. Eine Funktion $d: M \times M \rightarrow \mathrm{IR}$ heißt *Metrik*, wenn die folgenden Bedingungen erfüllt sind:
  - $d(x,y) \geq 0$ für alle $x,y \in M$
  - $d(x,y) = 0 \Leftrightarrow x = y$ für alle $x, y \in M$
  - $d(x,y) = d(y,x)$ für alle $x,y \in M$
  - $d(x,z) \leq d(x,y) + d(y,z)$ für alle $x.y,z \in M$.

  (M,d) ist dann ein *metrischer Raum*.

String-Metriken

Problem 6.2. Gegeben seien ein String *pat*, ein String *text*, eine Metrik d für Strings und ein ganze Zahl $k \geq 0$. Man finde alle Substrings y von text mit $d(pat, y) \leq k$.

Bemerkungen:

- Für $k = 0$ erhält man das exakte String-Matching Problem.
- Problem 4.2 ist zunächst ein “abstraktes” Problem, da nichts über die Metrik d ausgesagt wird.
- Zur Konkretisierung von Problem 6.2 und zur Entwicklung von entsprechenden Algorithmen müssen zunächst sinnvolle Metriken betrachtet werden.

Längste gemeinsame Teilfolge (LCS)

- Länge der längsten Teilfolge von Zeichen, die zwei Strings gemeinsam ist.
- Eine *Teilfolge* eines String wird durch das Löschen von null oder mehreren Zeichen erreicht.
- Beispiele:
  - “misspell” to “mispell” is 7
  - “misspelled” to “misinterpretted” is 7 “mis…p…e…ed”

Hamming-Distanz

Definition 6.2. Für zwei Strings x und y mit $|x| = |y| = m$ ergibt sich die *Hamming-Distanz (Hamming Distance)* durch:

$$d(x, y) = |\{1 \leq i \leq m | x[i] \neq y[i]\}|$$

Bemerkungen:
Die Hamming-Distanz ist die Anzahl der Positionen, an denen sich x und y unterscheiden. Sie ist nur für Strings gleicher Länge definiert. Wird in Problem 6.2 für d die Hamming-Distanz verwendet, so spricht man auch von "string matching with k mismatches".

Beispiel: *Die Hamming-Distanz der Strings* abcabb *und* cbacba *beträgt* 4.

Editier- (Levenstein-)Abstand

- Die minimale Anzahl von *Löschungen*, *Hinzufügungen* und *Änderungen* von Zeichen, um zwei Strings gleich zu machen.
  - "misspell" zu "mispell" hat Abstand 1
  - "misspell" zu "mistell" hat Abstand 2
  - "misspell" zu "misspelling" hat Abstand 3

Editier-/Levenstein-Distanz

Definition 6.3.
Für zwei Strings x und y ist die *Editierdistanz (Edit Distance) edit*(x, y) definiert als die kleinste Anzahl an Einfüge- und Löschoperationen, die notwendig sind, um x in y zu überführen.

Läßt man zusätzlich auch die Ersetzung eines Symbols zu, so spricht man von einer *Levenstein-Metrik (Levenshtein Distance) lev*(x, y).

Nimmt man als weitere Operation die Transposition (Vertauschung zweier benachbarter Symbole) hinzu, so erhält man die *Damerau-Levenstein-Metrik dlev*(x, y).

Editier-/Levenstein-Distanz

Bemerkungen:

- Offensichtlich gilt stets
  $$dlev(x, y) \leq lev(x, y) \leq edit(x, y).$$
- Die Damerau-Levenstein-Metrik wurde speziell zur Tippfehlerkorrektur entworfen.
- Wird in Problem 6.2 für d eine der Metriken aus Definition 6.3 verwendet, dann spricht man auch von "string matching with k differences" bzw. von "string matching with k errors".

Beispiel:

Für x = abcabba und y = cbabac gilt:

edit$(x, y) = 5$

- *abcabba → bcabba → cabba → cbba → cbaba → cbabac*

dlev$(x, y) =$ *lev*$(x, y) = 4$

- *abcabba → cbcabba → cbabba → cbaba → cbabac*
- *abcabba → bcabba → cbabba → cbabab → cbabac*

Berechnung der String-Distanz

Problem 6.3. Gegeben seien zwei Strings x und y. Man ermittle *edit*(x, y) bzw. *lev*(x, y) bzw. *dlev*(x, y) sowie die zugehörigen Operationen zur Überführung der Strings.

Bemerkungen:

- Wenn x und y Dateien repräsentieren, wobei $x[i]$ bzw. $y[j]$ die i-te Zeile bzw. j-te Zeile darstellt, dann spricht man auch vom *File Difference Problem*.
- Unter UNIX steht das Kommando diff zur Lösung des File Difference Problems zur Verfügung.

- Da die Metriken *edit*, *lev* und *dlev* sehr ähnlich sind, wird im folgenden nur die Levenstein-Metrik betrachtet.
- Algorithmen für die anderen Metriken erhält man durch einfache Modifikationen der folgenden Verfahren.

Berechnung der String-Distanz

- Im folgenden sei $m = |x|$ und $n = |y|$ und es gelte $m \leq n$.

- Lösungsansatz: dynamische Programmierung
- Genauer: berechne die Distanz der Teilstrings $x[1 \ldots i]$ und $y[1 \ldots j]$ auf Basis bereits berechneter Distanzen.

Berechnung der String-Distanz

Die Tabelle LEV sei definiert durch:
$LEV\,[i, j] := lev(x[1 \ldots i], y[1 \ldots j])$ mit $0 \leq i \leq m, 0 \leq j \leq n$

Die Werte für $LEV\,[i, j]$ können mit Hilfe der folgenden Rekursionsformeln berechnet werden:

- $LEV\,[0, j] = j$ für $0 \leq j \leq n$,
- $LEV\,[i, 0] = i$ für $0 \leq i \leq m$

- $LEV\,[i, j] = \min\{LEV\,[i-1, j] + 1,$
  $LEV\,[i, j-1] + 1,$
  $LEV\,[i-1, j-1] + (x[i], y[j])\}$

mit $\delta\,(a, b) = \begin{cases} 0, & \text{falls } a = b \\ 1, & \text{sonst} \end{cases}$

Berechnung der String-Distanz

Bemerkungen:

- Die Rekursionsformel spiegelt die drei Operation Löschen, Einfügen und Substitution wider.
- Die Stringdistanz ergibt sich als $LEV\,[m, n]$.
- Möchte man nur die Stringdistanz berechnen, so genügt es, sich auf Stufe i der Rekursion die Werte von LEV der Stufe $i - 1$ zu merken.
- Benötigt man die zugehörigen Operationen, speichert man LEV als Matrix und ermittelt die zugehörigen Operationen in einer “Rückwärts-rechnung”.

Berechnung der String-Distanz

Algorithmus 6.4. [Berechnung der Stringdistanz]

for $i := 0$ **to** m **do** $LEV\,[i, 0] := i$ **end**
for $j := 1$ **to** n **do** $LEV\,[0, j] := j$ **end**
for $i := 1$ **to** m **do**
  for $j := 1$ **to** n **do**
    $LEV\,[i, j] := \min\{LEV\,[i-1, j] + 1, LEV\,[i, j-1] + 1,$
    $LEV\,[i-1, j-1] + (x[i], y[j])\}$
  end
end
return $LEV\,[m, n]$

Berechnung der String-Distanz

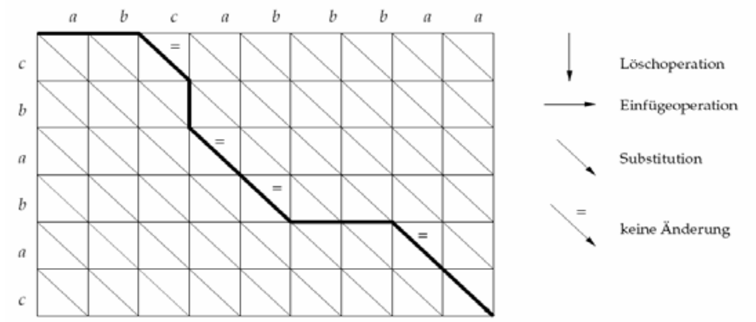
- **Beispiel:**
- Darstellung von LEV als Matrix für $x = cbabac$ und $y = abcabbbaa$:

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

- Die zugehörigen Umwandlungen lauten:
  cbabac → ababac → abcabac → abcabbac → abcabbbac → abcabbbaa

Berechnung der String-Distanz

- **Veranschaulichung:** Die Berechnung der Stringdistanz kann als Pfad in einem Graphen veranschaulicht werden.



Der dargestellte Pfad entspricht der folgenden (nicht optimalen) Umwandlung:
cbabac → acbabac → abcbabac → abcabac → abcabbac → abcabbbac → abcabbbaa

Berechnung der String-Distanz

Aus der Rekursionsformel und den Bemerkungen folgt:

Satz 6.2. *Die Stringdistanz (für edit, lev und dlev) kann in Zeit* $O(mn)$ *und Platz* $O(m)$ *berechnet werden. Problem 6.3 kann mit Platz* $O(mn)$ *gelöst werden.*

Berechnung der String-Distanz

Mit einer kleinen Änderung kann die angegebene Rekursionsformel auch zur Lösung von Problem 6.2 eingesetzt werden:

Es sei *MLEV* definiert durch:

$$\text{MLEV}\ [i, j] := \min_{1 \le l \le j} \{lev(pat[1 \ldots i], text[l \ldots j])\}$$

d.h., *MLEV* $[i, j]$ ist die kleinste Distanz zwischen $pat[1 \ldots i]$ und einem Suffix von text$[1, j]$.

Es gilt nun: *MLEV* $[0, j] = 0$ für $0 \le j \le n$, denn $pat[1 \ldots 0] = \varepsilon$ und ε ist stets in $text[1 \ldots j]$ ohne Fehler enthalten.

Berechnung der String-Distanz

Ansonsten berechnet sich *MLEV* $[i, j]$ wie *LEV* $[i,; j]$, d.h.:

$$MLEV\ [i, 0] = i \quad \text{für } 0 \le i \le \text{m}$$

$$\begin{aligned} MLEV\ [i, j] \ = \ \min \{ & MLEV\ [i-1, j] + 1, \\ & MLEV\ [i, j-1] + 1, \\ & MLEV\ [i-1, j-1] + \delta\ (x[i], y[j]) \} \end{aligned}$$

Gilt nun *MLEV* $[m, j] \le k$, so endet in Position j ein Substring y von *text* mit $lev(pat, y) \le k$ (wobei m die Patternlänge ist).

Berechnung der String-Distanz

Beispiel: Die Tabelle *MLEV* für *pat = ABCDE* und *text = ACEABPCQDEABCR*.

	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
i			A	C	E	A	B	P	C	Q	D	E	A	B	C	R
0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	A	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1
2	B	2	1	1	2	1	0	1	2	2	2	2	1	0	1	2
3	C	3	2	1	2	2	1	1	1	2	3	3	2	1	0	1
4	D	4	3	2	2	3	2	2	2	2	2	3	3	2	1	1
5	E	5	4	3	2	3	3	3	3	3	3	2	3	3	2	2

Für $k = 2$ ergeben sich die Positionen 3, 10, 13 und 14. Die zugehörigen Substrings von text sind *ACE*, *ABPCQDE*, *ABC* und *ABCR*.

Satz 6.3. *Problem 6.2 kann für die Metriken edit, lev und dlev in Zeit* O(mn) *gelöst werden.*