Kapitel 3

Inexakte Suche und Ähnlichkeit

Aufgabenstellungen und Anwendungen

- Globale Ähnlichkeit/Finden von Unterschieden

  Gesucht sind der Abstand bzw. ein Maß für die Ähnlichkeit zweier Zeichenketten sowie eine Darstellung der Unterschiede.
  Anwendung: Unix-Werkzeug `diff`

- Inexakte Suche (Approximate Pattern Matching)

  Gegeben sind ein Suchwort P und ein Text T.
  Gesucht sind alle Vorkommen von zu P hinreichend ähnlichen Wörtern in T.
  Anwendung: Rechtschreibprüfung, Suchmaschinen

- Lokale Ähnlichkeit

  In zwei (im Ganzen stark unterschiedlichen) Zeichenketten sind Regionen großer Ähnlichkeit gesucht.
  Anwendung: Vergleich von biomolekularen Sequenzen

Übersicht

- Alignments als Darstellungsmöglichkeit von Ähnlichkeiten
- Konstruktion von Alignments durch Dynamische Programmierung
- Varianten und Verbesserungen des Basis-Algorithmus
- Shift-And-Algorithmus für die inexakte Suche
- (Ähnlichkeit in mehr als zwei Zeichenketten: Multiple Alignments)

Abstände von Wörtern

Abstand: Abbildung $d : \Sigma^* \times \Sigma^* \to \mathbb{N}$ mit

1. $d(v, w) = d(w, v)$ für alle $v, w \in \Sigma^*$
2. $d(v, w) = 0 \iff v = w$ für alle $v, w \in \Sigma^*$
3. $d(u, w) \leq d(u, v) + d(v, w)$ für alle $u, v, w \in \Sigma^*$

Beispiel: Levenshtein-Abstand.
Es seien $S_1, S_2 \in \Sigma^*$. Der Levenshtein-Abstand von S_1 und S_2 ist die minimal notwendige Anzahl von Operationen

Ersetzen eines Zeichens $x \in \Sigma$ durch ein anderes Zeichen $y \in \Sigma$,
Löschen eines Zeichens $x \in \Sigma$,
Einfügen eines Zeichens $y \in \Sigma$,

die man zur Umwandlung von S_1 in S_2 benötigt.

Ähnlichkeit von Zeichenketten

Ähnlichkeit: symmetrische Abbildung $d : \Sigma^* \times \Sigma^* \to \mathbb{Z}$
große Ähnlichkeit entspricht hohem Wert von d

Beispiel: Ähnlichkeitsmatrizen für Aminosäuren.
Für jedes Paar (x, y) von Aminosäuren (Buchstaben) wird ein Wert der Ähnlichkeit definiert. Auf Moleküle (Wörter) wird der Begriff der Ähnlichkeit in passender Weise erweitert.

Abstände sind Spezialfälle der Ähnlichkeit.
Für die Suche nach lokaler Ähnlichkeit ist die Verwendung von Abständen nicht angebracht.

Alignments – Idee

Gegenüberstellung von zwei Zeichenketten, wobei man Lücken einfügen darf.

Beispiel. Es seien $S_1 =$ `tempel` und $S_2 =$ `treppe`. Es gibt u.a. folgende Alignments:

```
t     e m p e l          t     e m p     e l
t r e p p e              t r e     p p e
```

Bewertung eines Alignments:
Bewerte gegenübergestellte Zeichen und summiere die Bewertungen.

Abstand bzw. Ähnlichkeit definiert man als niedrigste bzw. höchste Bewertung der möglichen Alignments.

Alignments – Formalisierung

Alphabet Σ, Lückenzeichen $\_ \notin \Sigma$.

Definition. Es seien $S_1, S_2 \in \Sigma^*$. Ein Alignment von (S_1, S_2) ist ein Paar (S_1', S_2') mit $S_1', S_2' \in (\Sigma \cup \{\_\})^*$, wobei

- S_1' bzw. S_2' aus S_1 bzw. S_2 durch Einfügen von Lückenzeichen $\_$ entsteht,
- $|S_1'| = |S_2'|$ gilt.

Definition. Es sei $d : (\Sigma \cup \{\_\})^2 \to \mathbb{Z}$ eine Bewertungsfunktion. Die Bewertung $d(S_1', S_2')$ eines Alignments (S_1', S_2') mit $|S_1'| = |S_2'| = t$ ist

$$d(S_1', S_2') := \sum_{i=1}^{t} d(S_1'[i], S_2'[i]).$$

Alignments und Levenshtein-Abstand

(x, y) im Alignment $(x \neq y)$	entspricht	Ersetzung von x durch y
$(x, \_)$ im Alignment	entspricht	Löschen von x
$(\_, y)$ im Alignment	entspricht	Einfügen von y

Mit $d(x, x) = 0$ und $d(x, y) = 1$ für $x \neq y$ ist die Bewertung des minimalen Alignments gleich dem Levenshtein-Abstand.

Beispiel.

```
t     e  m  p  e  l          t     e  m  p     e  l
t  r  e  p  p  e             t  r  e     p  p  e
```

Das erste (optimale) Alignment hat die Bewertung 3, das zweite die Bewertung 4.

3.1 Konstruktion optimaler Alignments

Gegeben: Zeichenketten S_1, S_2; $|S_1| = m, |S_2| = n$; Bewertung d.
Gesucht: optimales Alignment
d.h. minimale Bewertung für Abstand, maximale Bewertung für Ähnlichkeit

Definition der Bewertung von Alignments erlaubt induktive Berechnung
(Dynamische Programmierung)

Abwandlung für inexakte Suche und lokale Alignments möglich

Die grundlegende Rekursionsbeziehung

Notation: $D_{i,j} = D_{i,j}(S_1, S_2)$ sei die Bewertung des optimalen Alignments von $(S_1[1 \ldots i], S_2[1 \ldots j])$ für $0 \leq i \leq m = |S_1|$, $0 \leq j \leq n = |S_2|$.

Satz. Es sei $d : (\Sigma \cup \{\_\})^2 \to \mathbb{Z}$ eine Bewertungsfunktion mit $d(\_, \_) = 0$; und das Alignment mit minimaler Bewertung sei gesucht. Dann gilt:

1. $D_{0,0} = 0$,
2. $D_{i,0} = D_{i-1,0} + d(S_1[i], \_)$ für $0 < i \leq m$,
3. $D_{0,j} = D_{0,j-1} + d(\_, S_2[j])$ für $0 < j \leq n$,
4. $D_{i,j} = \min\{D_{i-1,j-1} + d(S_1[i], S_2[j]), D_{i-1,j} + d(S_1[i], \_), D_{i,j-1} + d(\_, S_2[j])\}$ für $0 < i \leq m, 0 < j \leq n$.

Beweisidee

Mögliche Alignments von αx und βy

1. Alignment von (α, β) und (x, y):

α	x
β	y

2. Alignment von $(\alpha, \beta y)$ und $(x, \_)$:

α	x
βy	$\_$

3. Alignment von $(\alpha x, \beta)$ und $(\_, y)$:

αx	$\_$
β	y

Das optimale Alignment von $(\alpha x, \beta y)$ entsteht durch Erweiterung eines optimalen Alignments von (α, β) bzw. $(\alpha, \beta y)$ bzw. $(\alpha x, \beta)$.

Algorithmus 3.1 Ermittlung des minimalen Alignment-Wertes

Eingabe: Wörter S_1, S_2, $|S_1| = m, |S_2| = n$

Ausgabe: Bewertung des minimalen Alignments von S_1 und S_2

(1) $D_{0,0} \leftarrow 0$;
(2) **for** $i \leftarrow 1$ **to** m
(3) $\quad D_{i,0} \leftarrow D_{i-1,0} + d(S_1[i], \_)$;
(4) **for** $j \leftarrow 1$ **to** n
(5) $\quad D_{0,j} \leftarrow D_{0,j-1} + d(\_, S_2[j])$;
(6) **for** $i \leftarrow 1$ **to** m
(7) $\quad$ **for** $j \leftarrow 1$ **to** n
(8) $\quad\quad D_{i,j} \leftarrow \min\{D_{i-1,j-1} + d(S_1[i], S_2[j]), D_{i-1,j} + d(S_1[i], \_),$
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad D_{i,j-1} + d(\_, S_2[j])\}$;
(9) **return** $D_{m,n}$;

Beispiel

Für $S_1 =$ `tempel` und $S_2 =$ `treppe` ergibt sich bei Verwendung des Levenshtein-Abstandes die folgende Alignment-Tabelle

		t	r	e	p	p	e
	0	1	2	3	4	5	6
t	1	0	1	2	3	4	5
e	2	1	1	1	2	3	4
m	3	2	2	2	2	3	4
p	4	3	3	3	2	2	3
e	5	4	4	3	3	3	2
l	6	5	5	4	4	4	3

und damit die optimale Bewertung 3.

Algorithmus 3.2 Ermittlung eines optimalen Alignments

Eingabe: Wörter S_1, S_2, $|S_1| = m, |S_2| = n$, Alignment-Tabelle $(D_{i,j})$
Ausgabe: ein optimales Alignment A von S_1 und S_2

(1) $A \leftarrow \varepsilon$; $(i,j) \leftarrow (m,n)$;
(2) **while** $(i,j) \neq (0,0)$
(3) $\quad V \leftarrow \emptyset$;
(4) $\quad$ **if** $D_{i,j} = D_{i-1,j-1} + d(S_1[i], S_2[j])$ **then** $V \leftarrow V \cup \{\nwarrow\}$;
(5) $\quad$ **if** $D_{i,j} = D_{i,j-1} + d(\_, S_2[j])$ **then** $V \leftarrow V \cup \{\leftarrow\}$;
(6) $\quad$ **if** $D_{i,j} = D_{i-1,j} + d(S_1[i], \_)$ **then** $V \leftarrow V \cup \{\uparrow\}$;
(7) $\quad p \leftarrow$ ein Element aus V;
(8) $\quad$ **if** $p = \nwarrow$ **then** $A \leftarrow (S_1[i], S_2[j]) \cdot A$; $i \leftarrow i-1$; $j \leftarrow j-1$;
(9) $\quad$ **if** $p = \leftarrow$ **then** $A \leftarrow (\_, S_2[j]) \cdot A$; $j \leftarrow j-1$;
(10) $\quad$ **if** $p = \uparrow$ **then** $A \leftarrow (S_1[i], \_) \cdot A$; $i \leftarrow i-1$;
(11) **return** A;

Satz. Es seien S_1 und S_2 Wörter mit $|S_1| = m$, $|S_2| = n$. Die Algorithmen 3.1, 3.2 bestimmen den Levenshtein-Abstand und ein optimales Alignment von (S_1, S_2) mit einem Aufwand von $O(mn)$.

Beispiel

Für $S_1 =$ `tempel` und $S_2 =$ `treppe` erhält man bei Verwendung des Levenshtein-Abstandes folgenden Alignment-Pfad

		t	r	e	p	p	e
	0	1	2	3	4	5	6
t	1	0	1	2	3	4	5
e	2	1	1	1	2	3	4
m	3	2	2	2	2	3	4
p	4	3	3	3	2	2	3
e	5	4	4	3	3	3	2
l	6	5	5	4	4	4	3

und das (in diesem Fall einzige) optimale Alignment

t	_	e	m	p	e	l
t	r	e	p	p	e	_

Graphentheoretische Interpretation

Zu (S_1, S_2) mit $|S_1| = m, |S_2| = n$ konstruiere den gerichteten Graphen $G = (U, E)$ mit

$$\begin{aligned} U &= \{(i,j) : 0 \leq i \leq m, 0 \leq j \leq n\}, \\ E &= \{((i-1,j-1),(i,j)) : 1 \leq i \leq m, 1 \leq j \leq n\} \cup \\ &\quad \{((i-1,j),(i,j)) : 1 \leq i \leq m, 0 \leq j \leq n\} \cup \\ &\quad \{((i,j-1),(i,j)) : 0 \leq i \leq m, 1 \leq j \leq n\} \end{aligned}$$

und der Bewertungsfunktion $d : E \to \mathbb{Z}$ mit

$$d(e) = \begin{cases} d(S_1[i], S_2[j]), & \text{falls } e = ((i-1,j-1),(i,j)) \\ d(S_1[i], \_), & \text{falls } e = ((i-1,j),(i,j)) \\ d(\_, S_2[j]), & \text{falls } e = ((i,j-1),(i,j)) \end{cases}$$

Graphentheoretische Interpretation – Fortsetzung

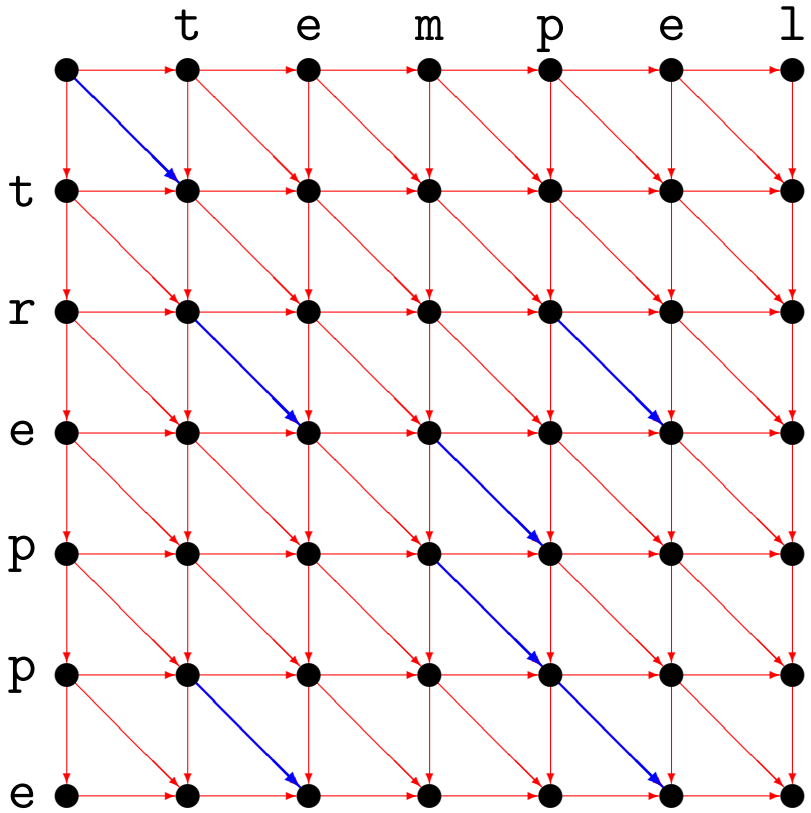
Knoten des Graphen sind die Felder unserer Tabelle.

Kanten gehen zu den Nachbar rechts, unten sowie rechts unten.
Bewertung einer Kante entspricht Bewertung des zugehörigen Paares.

Ein Alignment entspricht einem gerichteten Pfad von $s = (0, 0)$ nach $t = (m, n)$.

Damit ist das Alignment-Problem auf ein Kürzeste-Wege-Problem zurückgeführt das wir z.B. auch mit dem DIJKSTRA-Algorithmus in $O(mn)$ Schritten lösen könnten.
Unser Algorithmus nutzt die besondere Struktur des Graphen aus.

Graphentheoretische Interpretation – Beispiel



rote Kante: Gewicht 1

blaue Kante: Gewicht 0

Variante: Ähnlichkeit statt Abstand

gesucht: Alignment mit maximaler Bewertung

Modifikation der Rekursionsbeziehung:
$D_{i,j} = \max\{D_{i-1,j-1} + d(S_1[i], S_2[j]), D_{i-1,j} + d(S_1[i], \_), D_{i,j-1} + d(\_, S_2[j])\}$
für $0 < i \leq m, 0 < j \leq n$.

Spezialfall: Längste gemeinsamen Teilfolge (longest common subsequence).

Definition. Ein Wort α der Länge t heißt Teilfolge des Wortes S, wenn es eine Indexfolge $1 \leq j_1 < j_2 < \ldots < j_t \leq |S|$ mit $\alpha[i] = S[j_i]$ für alle $1 \leq i \leq t$ gibt.

Längste gemeinsame Teilfolge von S_1 und $S_2 \rightarrow$ Bestimme maximales Alignment von (S_1, S_2) für $d(x, y) = \begin{cases} 1, & \text{falls } x = y \text{ und } x, y \in \Sigma \\ 0, & \text{sonst} \end{cases}$

Beispiel

Für $S_1 =$ `tempel` und $S_2 =$ `treppe` ergibt sich für die längste gemeinsame Teilfolge die folgende Alignment-Tabelle einschließlich den möglichen Alignment-Pfaden:

		t	r	e	p	p	e
	0	0	0	0	0	0	0
t	0	1	1	1	1	1	1
e	0	1	1	2	2	2	2
m	0	1	1	2	2	2	2
p	0	1	1	2	3	3	3
e	0	1	1	2	3	3	4
l	0	1	1	2	3	3	4

Ein optimales Alignment ist z.B.

```
t _ e m p _ e l
t r e _ p p e _
```

mit der längsten Teilfolge `tepe`.

Inexakte Suche

Gesucht: optimales Alignment von S_1 mit einem Teilwort von S_2.

Lösung: Bestimme für jedes Paar (i, j) das optimale Alignment von $S_1[1 \ldots i]$ mit einem Suffix von $S_2[1 \ldots j]$.

Es sei (für ein Minimierungsproblem) $D'_{i,j} = \min_{0 \le t \le j} \{D(S_1[1 \ldots i], S_2[t \ldots j])\}$.

Lemma.

1. $D'_{0,0} = 0$, $D'_{0,j} = 0$, $D'_{i,0} = D'_{i-1,0} + d(S_1[i], \_)$;
2. $D'_{i,j} = \min\{D'_{i-1,j-1} + d(S_1[i], S_2[j]), D'_{i-1,j} + d(S_1[i], \_), D'_{i,j-1} + d(\_, S_2[j])\}$ für $1 \le j \le n$, $1 \le i \le m$.

Inexakte Suche – Beispiel

Suche $S_1 =$ `fische` in $S_2 =$ `fritzefischtefrische` bezüglich Levenshtein-Abstand.

		f	r	i	t	z	e	f	i	s	c	h	t	e	f	r	i	s	c	h	e
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f	1	0	1	1	1	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1
i	2	1	1	1	2	2	2	1	0	1	2	2	2	2	1	1	1	2	2	2	2
s	3	2	2	2	2	3	3	2	1	0	1	2	3	3	2	2	2	1	2	3	3
c	4	3	3	3	3	3	4	3	2	1	0	1	2	3	3	3	3	2	1	2	3
h	5	4	4	4	4	4	4	4	3	2	1	0	1	2	3	4	4	3	2	1	2
e	6	5	5	5	5	5	4	5	4	3	2	1	1	1	2	3	4	4	3	2	1

Die besten Treffer mit jeweils der Bewertung 1 enden an den Textpositionen 11, 12, 13 und 20 und liefern die Alignments

$$\begin{array}{c} \texttt{fische} \\ \texttt{fisch\_} \end{array}, \quad \begin{array}{c} \texttt{fische} \\ \texttt{fischt} \end{array}, \quad \begin{array}{c} \texttt{fisch\_e} \\ \texttt{fischte} \end{array} \quad \text{und} \quad \begin{array}{c} \texttt{f\_ische} \\ \texttt{frische} \end{array}.$$

Lokale Alignments

Definition. Ein lokales Alignment von (S_1, S_2) ist ein Alignment von (α, β), wobei α bzw. β Infixe von S_1 bzw. S_2 sind.

Bewertung durch Ähnlichkeitsfunktionen
(mit positiven Bewertungen als Zeichen der Ähnlichkeit)

$d : (\Sigma \cup \{\_\})^2 \to \mathbb{Z}$

Aufgabenstellungen

1. Bestimme global bestes lokales Alignment.
2. Bestimme lokal optimale Alignments, also solche, die weder durch Verkürzen noch Verlängern verbessert werden können.

Bestimmung lokal optimaler Alignments

Für $0 \leq i \leq m$, $0 \leq j \leq n$ sei $D''_{i,j}$ die Bewertung eines optimalen Alignments eines Suffixes von $S_1[1 \ldots i]$ mit einem Suffix von $S_2[1 \ldots j]$.

Lemma.

1. $D''_{0,0} = D''_{i,0} = D''_{0,j} = 0$;
2. $D''_{i,j} = \max\{D''_{i-1,j-1} + d(S_1[i], S_2[j]), D''_{i-1,j} + d(S_1[i], \_), D''_{i,j-1} + d(\_, S_2[j]), 0\}$, für $1 \leq i \leq m$, $1 \leq j \leq n$.

Ende eines globalen Optimums: (i, j) mit $D''_{i,j} \to \max$
Ende eines lokalen Optimums: (i, j) mit

$$D''_{i,j} \geq \max\{D''_{i-1,j-1}, D''_{i,j-1}, D''_{i-1,j}, D''_{i+1,j+1}, D''_{i,j+1}, D''_{i+1,j}\}.$$

Ein optimaler lokaler Alignment-Pfad führt von einer Endposition zu einem Feld mit dem Wert 0.

Lokale Alignments – Beispiel

Ähnlichkeitsmaß: $d(x,x) = 2$ für $x \in \{a,b,c,d\}$, $d(x,y) = -1$, sonst;
$S_1 = caabcacb$, $S_2 = dddadbddddadabdd$;
Pfade lokal optimaler Alignments mit Bewertung von mindestens 3 sind markiert.

		d	d	d	a	d	b	d	d	d	d	a	d	a	b	d	d
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
c	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
a	0	0	0	0	2	1	0	0	0	0	0	2	1	0	0	0	0
a	0	0	0	0	1	1	0	0	0	0	0	1	1	3	2	1	0
b	0	0	0	0	0	0	3	2	1	0	0	0	0	2	5	4	3
c	0	0	0	0	0	0	2	2	1	0	0	0	0	1	4	4	3
a	0	0	0	0	2	1	1	1	1	0	0	2	1	2	3	3	3
c	0	0	0	0	1	1	0	0	0	0	0	1	1	1	2	2	2
b	0	0	0	0	0	0	3	2	1	0	0	0	0	0	3	2	1

Die optimalen lokalen Alignments sind $\begin{matrix}\texttt{aab}\\ \texttt{adb}\end{matrix}$, $\begin{matrix}\texttt{acb}\\ \texttt{adb}\end{matrix}$, $\begin{matrix}\texttt{a\_ab}\\ \texttt{adab}\end{matrix}$ und $\begin{matrix}\texttt{acb}\\ \texttt{a\_b}\end{matrix}$.

Verkettung lokaler Alignments

Aufgabe: Bestimme globales Alignment, das lokale Alignments mit optimaler Gesamtbewertung enthält

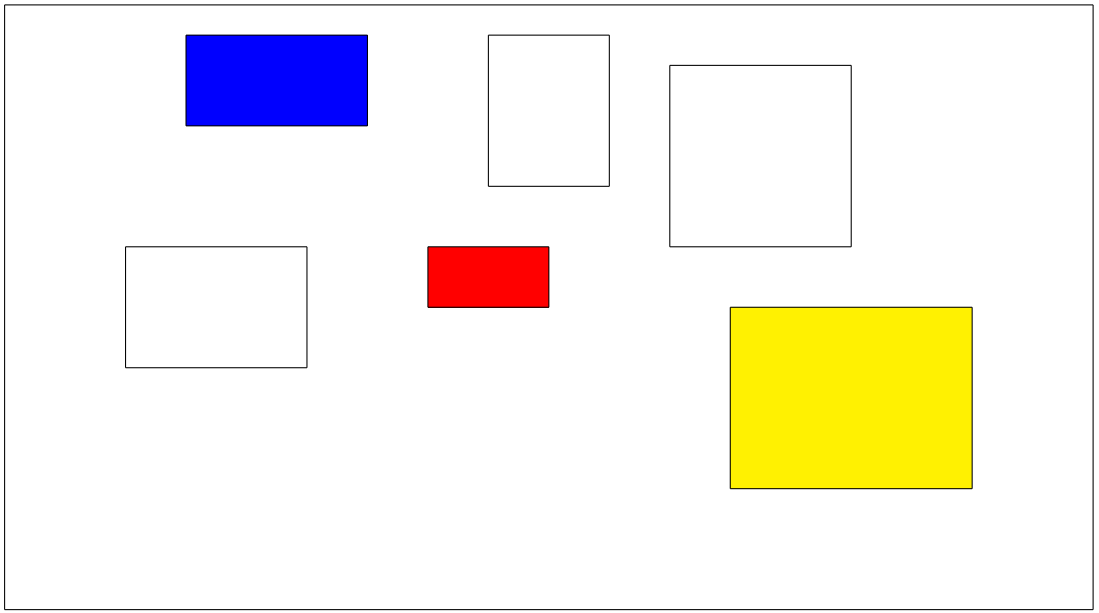
Gewertet werden nur die markierten lokalen Alignments.

Gewertete Intervalle müssen disjunkt sein.

Ordnung der gepaarten Intervalle muss in beiden Zeichenketten gleich sein.

Geometrische Interpretation

Lokales Alignment von $(S_1[i_1 \ldots i_2], S_2[j_1 \ldots j_2])$ entspricht Rechteck $[i_1, i_2] \times [j_1, j_2]$.



Das heißt: Bestimme in einer Menge von Rechtecken eine Kette von Rechtecken mit optimaler Bewertung.

Die linke obere Ecke des nächsten Rechtecks in einer Kette muß rechts unten von der rechten unteren Ecke des aktuellen Rechtecks liegen.

Halbordnungen und Ketten

Definition. Auf der Menge der Intervalle definieren wir die Halbordnung $<_I$ mit $[l_1, r_1] <_I [l_2, r_2] \iff r_1 < l_2$.

Definition. Auf der Menge der Rechtecke definieren wir die Halbordnung $<_R$ mit $[l_1, r_1] \times [o_1, u_1] <_R [l_2, r_2] \times [o_2, u_2] \iff r_1 < l_2 \wedge u_1 < o_2$.

Definition. Es sei $(M, <)$ eine Halbordnung. Eine (endliche) Kette in M ist eine Folge $C = (c_1, c_2, \ldots, c_r)$ mit $c_1 < c_2 < \ldots < c_r$.

Aufgabe:
Gegeben: endliche Menge $\mathcal{R}$ von bewerteten Intervallen (bzw. Rechtecken).
Gesucht: Kette in $\mathcal{R}$ mit maximaler Bewertung

Verkettung von Intervallen – Idee

Gegeben: Menge von Intervallen $\mathcal{R} = \{[l_1, r_1], [l_2, r_2], \ldots, [l_s, r_s]\}$, $d : \mathcal{R} \to \mathbb{N}$.

Konstruiere Array X der Länge $2s$ zur Speicherung der Endpunkte.

Elemente von X: Paare der Form (j, e) mit $j \in \{1, \ldots, s\}$, $e \in \{l, r\}$;
j gibt das Intervall an und e die Art des Endpunktes (links oder rechts).

Elemente von X sind geordnet nach der Zahl e_j.
Falls $l_j = r_k$, so steht (j, l) vor (r, k).

Durch Dynamische Programmierung bestimmen wir für jedes Intervall aus $\mathcal{R}$ die maximale Kette, die mit diesem Intervall endet.

Algorithmus 3.3 Verkettung von Intervallen

Eingabe: Menge von Intervallen $\mathcal{R} = \{[l_1, r_1], [l_2, r_2], \ldots, [l_s, r_s]\}$,
Bewertung $d : \{1, \ldots, s\} \to \mathbb{N}$, $d(i) = d([l_i, r_i])$
Ausgabe: Bewertung der optimalen Kette in $\mathcal{R}$

(1) Bestimme das Array X der Länge $2s$;
(2) $max \leftarrow 0$;
(3) **for** $i \leftarrow 1$ **to** $2s$
(4) $\quad (j, x) \leftarrow X[i]$;
(5) $\quad$ **if** $x = l$ **then** $D[j] \leftarrow d(j) + max$;
(6) $\quad$ **if** $x = r$ **then** $max \leftarrow \max\{max, D[j]\}$;
(7) **return** max;

Satz. Algorithmus 3.3 bestimmt die maximale Bewertung einer Kette in der Menge von Intervallen $\mathcal{R}$ mit einem Aufwand von $O(s \log s)$.

Verkettung von Rechtecken – Idee

Gegeben: Menge von Rechtecken
$\mathcal{R} = \{[l_1, r_1] \times [o_1, u_1], [l_2, r_2] \times [o_2, u_2], \ldots, [l_s, r_s] \times [o_s, u_s]\}$, Bewertung $d : \mathcal{R} \to \mathbb{N}$.

Array X enthält linke und rechte Enden wie im eindimensionalen Fall.

Statt Zahl max benutze Menge Max von Paaren (D, j) mit $D \in \mathbb{N}$, $1 \leq j \leq s$.

Ist in der Menge Max ein Paar (D, j) enthalten und ist in der Menge X ein Endpunkt x abgeschlossen, so hat die beste Kette im Bereich $[1, u_j] \times [1, x]$ den Wert D und endet mit dem j-ten Rechteck.

Die Elemente (D, j) von Max sind bezüglich u_j geordnet.

Satz. Algorithmus 3.4 bestimmt die maximale Bewertung einer Kette in der Menge von s Rechtecken $\mathcal{R}$ mit einem Aufwand von $O(s \log s)$.

Algorithmus 3.4 Verkettung von Rechtecken

Eingabe: Rechtecke $\mathcal{R} = \{[l_1, r_1] \times [o_1, u_1], [l_2, r_2] \times [o_2, u_2], \ldots, [l_s, r_s] \times [o_s, u_s]\}$,
Bewertung $d : \{1, \ldots, s\} \to \mathbb{N}$, $d(j) = d([l_j, r_j] \times [o_j, u_j])$
Ausgabe: Bewertung der optimalen Kette in $\mathcal{R}$

(1) Bestimme das Array X der Länge $2s$;
(2) $Max \leftarrow \{(0, 0)\}$;
(3) **for** $i \leftarrow 1$ **to** $2s$
(4) $(j, x) \leftarrow X[i]$;
(5) **if** $x = l$
(6) $(D, k) \leftarrow$ Element aus Max mit $u_k < o_j$ und $u_k \to \max$;
(7) $D[j] \leftarrow d(j) + D$;
(8) **if** $x = r$
(9) $(D, k) \leftarrow$ Element aus Max mit $u_k \leq u_j$ und $u_k \to \max$;
(10) **if** $D < D[j]$
(11) Ordne $(D[j], j)$ in Max ein;
(12) $(D, k) \leftarrow$ Element aus Max mit $u_k \geq u_j$ und $u_k \to \min$;
(13) **while** $D \leq D[j]$
(14) Lösche (D, k);
(15) $(D, k) \leftarrow$ Element aus Max mit $u_k \geq u_j$ und $u_k \to \min$;
(16) $(D, k) \leftarrow$ letztes Element von Max;
(17) **return** D;

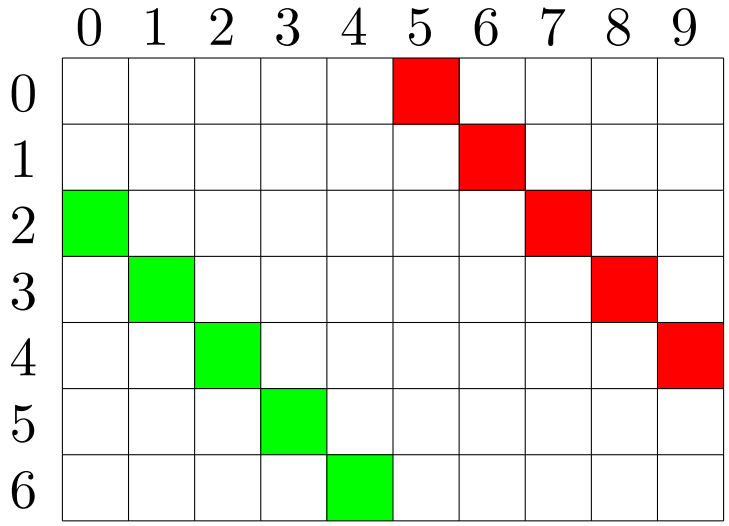
3.2 Alignments mit beschränkter Fehlerzahl

- Bewertung: Levenshtein-Abstand
- Von Interesse sind nur Alignments mit höchstens k Fehlern
- globale Alignments: Modifikation des DP-Algorithmus, Laufzeit $O(kn)$
- inexakte Suche: Modifikation des DP-Algorithmus, mittlere Laufzeit $O(kn)$
  weitere Algorithmen später

Diagonalen einer Matrix

Definition. Für eine Matrix mit den Zeilen $0, 1, \ldots, m$ und den Spalten $0, 1, \ldots, n$ ist die k-te Diagonale $(-m \leq k \leq n)$ definiert als
$Diag_k = \{(i, j) : 0 \leq i \leq m \wedge 0 \leq j \leq n \wedge j - i = k\}$.

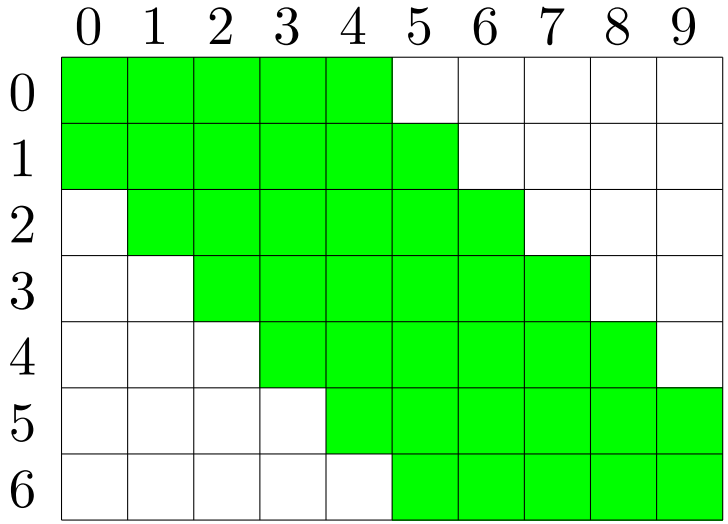
Beispiel. In der folgenden Matrix sind die Diagonalen $Diag_{-2}$ und $Diag_5$ dargestellt.



Globales Alignment mit k Fehlern

Beobachtung: Es sei $|S_1| = m \leq |S_2| = n$, $n - m = \delta \leq k$.
Existiert ein Alignment mit der Bewertung $\leq k$, so verläuft sein Pfad zwischen den Diagonalen $-\lfloor (k-\delta)/2 \rfloor$ und $\lfloor (k+\delta)/2 \rfloor$.

Beispiel. Für $m = 6$, $n = 9$ kann ein Alignmentpfad mit höchstens 6 Fehlern nur zwischen den Diagonalen -1 und 4 verlaufen.



Modifikation des DP-Algorithmus

Berechne Werte $D^k_{i,j}$ des optimalen Alignments im für k zulässigen Bereich.

Für Zeile i zwischen $A_i = \max\{0, i - \lfloor (k-\delta)/2 \rfloor\}$ und $E_i = \min\{n, i + \lfloor (k+\delta)/2 \rfloor\}$

$$D^k_{i,j} = \begin{cases} \min\{D^k_{i-1,j-1} + d(S_1[i], S_2[j]), D^k_{i-1,j} + 1, D^k_{i,j-1} + 1\} & \text{falls } A_i < j < E_i \\ \min\{D^k_{i-1,j-1} + d(S_1[i], S_2[j]), D^k_{i-1,j} + 1\} & \text{falls } j = A_i \\ \min\{D^k_{i-1,j-1} + d(S_1[i], S_2[j]), D^k_{i,j-1} + 1\} & \text{falls } j = E_i \end{cases}$$

Pro Zeile sind höchstens $(k+1)$ Werte zu bestimmen.

Lemma. $D^k_{i,j} = D_{i,j}$, falls $D_{i,j} \leq k$.

Beschränkte Fehlerzahl: Beispiel

Für $S_1 =$ `tempel` und $S_2 =$ `treppe` gilt $\delta = 0$.

$k = 1$

		t	r	e	p	p	e
	0						
t		0					
e			1				
m				2			
p					2		
e						3	
l							4

$k = 3$

		t	r	e	p	p	e
	0	1					
t	1	0	1				
e		1	1	1			
m			2	2	2		
p				3	2	2	
e					3	3	2
l						4	3

Verbesserung des allgemeinen Alignment-Algorithmus

Allgemeines Alignment-Problem kann mit Aufwand $O(k^*m)$ gelöst werden, wobei der Levenshtein-Abstand k^* nicht im Voraus bekannt ist.

(1) $k \leftarrow 1$;
(2) **while** optimales Alignment nicht gefunden
(3) Versuche, Alignment mit höchstens k Fehlern zu bestimmen;
(4) $k \leftarrow 2k$;

Satz. Der Levenshtein-Abstand und ein zugehöriges optimales Alignment von (S_1, S_2) können mit einem Aufwand von $O(k^*m)$ bestimmt werden, wobei k^* der Levenshtein-Abstand von S_1 und S_2 ist.

Inexakte Suche mit k Fehlern (Verbesserung des DP-Algorithmus)

Lemma. Es seien S_1 und S_2 Wörter mit $|S_1| = m$, $|S_2| = n$. Mit dem Levenshtein-Abstand als Abstandsmaß gilt

1. $D'_{i,j} \geq D'_{i,j-1} - 1$ für $0 \leq i \leq m$, $1 \leq j \leq n$,
2. $D'_{i,j} \geq D'_{i-1,j} - 1$ für $1 \leq i \leq m$, $0 \leq j \leq n$,
3. $D'_{i,j} \geq D'_{i-1,j-1}$ für $1 \leq i \leq m$, $1 \leq j \leq n$.

Folgerung. Wenn $D'_{i,j} \leq k$, dann $i = 0$ oder $D'_{i-1,j-1} \leq k$.

Für die Suche mit k Fehlern braucht man also $D'_{i,j}$ nur zu bestimmen, wenn $D'_{i-1,j-1} \leq k$.
Mittlerer Aufwand: $O(kn)$.

Beispiel

Für die Suche nach $S_1 =$ `fische` in $S_2 =$ `fritzefischtefrische` mit einem Levenshtein-Abstand von höchstens 1 werden folgende Werte der Alignment-Tabelle bestimmt.

		f	r	i	t	z	e	f	i	s	c	h	t	e	f	r	i	s	c	h	e
	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f	1	0	1	1	1	1	1	0	1	1	1	1	1	1	0	1	1	1	1	1	1
i		1	1	1	2	2	2	1	0	1	2	2	2	2	1	1	1	2	2	2	2
s			2	2	2				1	0	1					2	2	1			
c										1	0	1							1		
h											1	0	1							1	
e												1	1	1							1

3.3 Alignments mit Lücken (Gaps)

- Löschen bzw. Einfügen eines Teilwortes (Blocks) ist eine Operation.
  (Löschen eines Blocks der Länge k kostet weniger als Löschen von k einzelnen Zeichen.)

- Formal: Bewertung einer Lücke (gap) durch Gap-Funktion $g : \mathbb{N}_+ \to \mathbb{N}$.
  (bei Abständen)

- Lösung durch Anpassung des Grundalgorithmus.

  allgemeiner Fall: Aufwand $O(mn(m+n))$
  affine Gap-Funktionen: Aufwand $O(mn)$.

Gaps und Alignments

Definition. Es sei (α, β) ein Alignment. Eine Lücke (gap) ist eine maximale Folge von Lückenzeichen in α oder β.

Satz. Jedes Alignment (α, β) ist eindeutig zerlegbar als

$$(\alpha, \beta) = (\alpha_1, \beta_1) \cdots (\alpha_k, \beta_k),$$

wobei, für $1 \leq i \leq k$, α_i und β_i entweder kein Lückenzeichen enthalten oder eins von beiden eine Lücke ist.

Modifizierte Bewertung von Alignments

Abstandsfunktion $d : \Sigma \times \Sigma \to \mathbb{R}_+$
Gap-Funktion $g : \mathbb{N}_+ \to \mathbb{R}_+$

Bewertung eines Alignments (α, β) mit $|\alpha| = |\beta| = t$ bei Gap-Funktion g:

$$
\begin{aligned}
d(\alpha,\beta) &:= \sum_{i=1}^{t} d(\alpha[i],\beta[i]), \text{ falls } \alpha,\beta \in \Sigma^* \\
d(\alpha,\beta) &:= g(t), \text{ falls } \alpha \text{ oder } \beta \text{ Lücke} \\
d(\alpha,\beta) &:= \sum_{j=1}^{k} d(\alpha_j,\beta_j), \text{ bei Zerlegung wie im Satz}
\end{aligned}
$$

Beispiel

Für die Abstandsfunktion d mit $d(x,x) = 0$ und $d(x,y) = 1$ für $x \neq y$ und die Gap-Funktion g mit $g(n) = 1 + n/2$ ergeben sich folgende Bewertungen für die folgenden Alignments von $abaaaaaabb$ und $abaaba$:

a	b	a	a	a	a	a	a	b	b	Bewertung: 6
a	b		a			a		b	a	

a	b	a	a	a	a	a	a	b	b	Bewertung: 4
a	b					a	a	b	a	

Die Bewertung nach dem Levenshtein-Abstand ist für beide Alignments 5.

Lösung für allgemeine Gap-Funktionen

Für $0 \leq i \leq m$, $0 \leq j \leq n$ und $-j \leq h \leq i$ sei

$D_{i,j,h}$ der Wert eines optimalen Alignments von $(S_1[1 \ldots i], S_2[1 \ldots j])$,

das mit einem Gap der Länge $|h|$ endet,

wobei sich das Gap in S_2 bzw. in S_1 befindet, falls $h > 0$ bzw. $h < 0$ gilt.

Weiterhin seien:

$$\begin{aligned} D_{i,j,+} &:= \min\{D_{i,j,h} : 1 \leq h \leq i\}, \\ D_{i,j,-} &:= \min\{D_{i,j,h} : -j \leq h \leq -1\}. \end{aligned}$$

Satz. Im Falle einer beliebigen Gap-Funktion kann das optimale Alignment mit einem Aufwand von $O(mn(m+n))$ gefunden werden.

Rekursionsformeln

Offenbar: $D_{i,j} = \min\{D_{i,j,+}, D_{i,j,-}, D_{i,j,0}\}$.

Für $0 \leq i \leq m$ und $0 \leq j \leq n$ gilt:

$$D_{0,j,h} = \begin{cases} g(j), & \text{falls } h = -j \\ \infty, & \text{sonst} \end{cases} \qquad D_{i,0,h} = \begin{cases} g(i), & \text{falls } h = i \\ \infty, & \text{sonst} \end{cases}$$

Für $1 \leq i \leq m$, $1 \leq j \leq n$ und mit $\Delta(t) = g(t) - g(t-1)$ für $t \geq 2$ ergibt sich:

$$\begin{aligned} D_{i,j,0} &= D_{i-1,j-1} + d(S_1[i], S_2[j]) \\ D_{i,j,1} &= \min\{D_{i-1,j,-}, D_{i-1,j,0}\} + g(1) \\ D_{i,j,-1} &= \min\{D_{i-1,j,+}, D_{i-1,j,0}\} + g(1) \\ D_{i,j,h} &= D_{i-1,j,h-1} + \Delta(h), 2 \leq h \leq i \\ D_{i,j,h} &= D_{i,j-1,h+1} + \Delta(|h|), -j \leq h \leq -2 \end{aligned}$$

Affine Gap-Funktionen

Gap-Funktionen der Form $g(n) = a(n-1) + b$ mit $a, b \geq 0$.

Vereinfachte Berechnung der $D_{i,j,+}$, $D_{i,j,0}$, $D_{i,j,-}$, da $\Delta(t) = a$ für alle $t \geq 2$.
(Berechnung der $D_{i,j,h}$ ist nicht mehr nötig.)

Satz. Im Falle einer affinen Gap-Funktion kann das optimale Alignment mit einem Aufwand von $O(mn)$ gefunden werden.

Affine Gaps: Rekursionsbeziehungen

Für $1 \leq i \leq m, 1 \leq j \leq n$ gilt:

$$
\begin{aligned}
D_{0,0,0} &= 0, \quad D_{0,0,-} = D_{0,0,+} = \infty \\
D_{0,j,-} &= a(j-1) + b, \quad D_{0,j,0} = D_{0,j,+} = \infty \\
D_{i,0,+} &= a(i-1) + b, \quad D_{i,0,0} = D_{i,0,-} = \infty
\end{aligned}
$$

$$
\begin{aligned}
D_{i,j,0} &= D_{i-1,j-1} + d(S_1[i], S_2[j]) \\
D_{i,j,-} &= \min\{D_{i,j-1,0} + b, D_{i,j-1,+} + b, D_{i,j-1,-} + a\} \\
D_{i,j,+} &= \min\{D_{i-1,j,0} + b, D_{i-1,j,-} + b, D_{i-1,j,+} + a\}
\end{aligned}
$$

Affine Gaps: Alignment-Tabelle

Die Zelle (i, j) der Tabelle enthält die Elemente $\begin{pmatrix} D_{i,j,+} \\ D_{i,j,0} \\ D_{i,j,-} \end{pmatrix}$

Bestimmung eines optimalen Alignment-Pfades:
Starte mit einem Element (m, n, x), $x \in \{+, 0, -\}$, so dass $D_{m,n,x} = D_{m,n}$.

Es sei das Element (i, j, x) erreicht.
Das nächste Element im Pfad bestimmt sich aus der Berechnung von $D_{i,j,x}$:

$(i-1, j, +)$,	falls $x = +$ und $D_{i,j,+} = D_{i-1,j,+} + 1$,
$(i-1, j, 0)$,	falls $x = +$ und $D_{i,j,+} = D_{i-1,j,0} + 4$,
$(i-1, j, -)$,	falls $x = +$ und $D_{i,j,+} = D_{i-1,j,-} + 4$,
$(i-1, j-1, +)$,	falls $x = 0$ und $D_{i,j,0} = D_{i-1,j-1,+} + d(S_1[i], S_2[j])$,
$(i-1, j-1, 0)$,	falls $x = 0$ und $D_{i,j,0} = D_{i-1,j-1,0} + d(S_1[i], S_2[j])$,
$(i-1, j-1, -)$,	falls $x = 0$ und $D_{i,j,0} = D_{i-1,j-1,-} + d(S_1[i], S_2[j])$,
$(i, j-1, +)$,	falls $x = -$ und $D_{i,j,-} = D_{i,j-1,+} + 4$,
$(i, j-1, 0)$,	falls $x = -$ und $D_{i,j,-} = D_{i,j-1,0} + 4$,
$(i, j-1, -)$,	falls $x = -$ und $D_{i,j,-} = D_{i,j-1,+} + 1$.

Beispiel: $d(x,x) = 0$, $d(x,y) = 2$ für $x \neq y$, $g(n) = (n-1) + 4$.

		a	b	a	a	a	a	a	a	b	b
	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
	0	∞	∞	∞	∞	∞	∞	∞	∞	∞	∞
	∞	4	5	6	7	8	9	10	11	12	13
	4	8	9	10	11	12	13	14	15	16	17
a	∞	0	6	5	6	7	8	9	10	13	14
	∞	8	4	5	6	7	8	9	10	11	12
	5	4	8	9	10	11	12	13	14	15	16
b	∞	6	0	6	7	8	9	10	11	10	11
	∞	9	8	4	5	6	7	11	12	13	14
	6	5	4	8	9	10	11	14	15	14	15
a	∞	5	6	0	4	5	6	7	10	13	12
	∞	10	9	8	4	5	6	7	8	9	10
	7	6	5	4	8	9	10	11	12	13	14
a	∞	6	7	4	0	4	5	6	7	10	11
	∞	11	10	9	8	4	5	6	7	8	9
	8	7	6	5	4	8	9	10	11	12	13
b	∞	9	6	7	6	2	6	7	8	7	8
	∞	12	11	10	9	8	6	7	8	9	10
	9	8	7	6	5	6	10	11	12	11	12
a	∞	8	9	6	5	4	2	6	7	10	9
	∞	13	12	11	10	9	8	6	7	8	9

3.4 Lösung mit linearem Platzbedarf

- Grundalgorithmus hat
  Laufzeit von $\Theta(mn)$ – kein Problem
  Platzbedarf von $\Theta(mn)$ – problematisch für große m, n

- Rekursiver Algorithmus löst Problem mit Platz $\Theta(n)$ in $\Theta(mn)$ Schritten.

- ähnliches Vorgehen möglich bei anderen Algorithmen mit Dynamischer Programmierung

Rückblick auf den Standard-Algorithmus

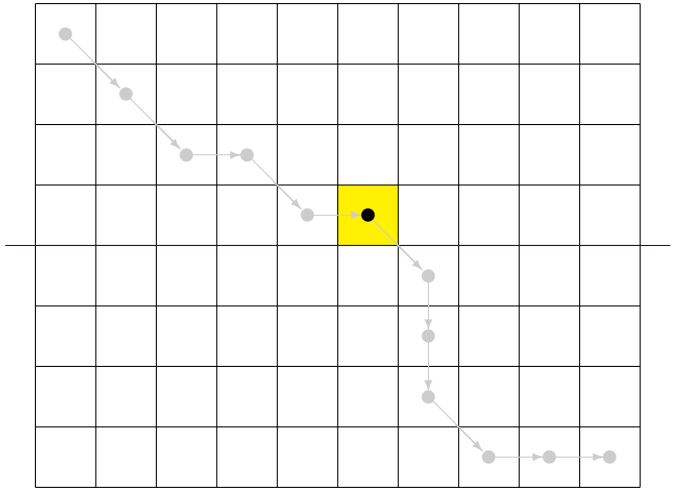
- Die Bestimmung des optimalen Alignment-Wertes $D_{m,n}$ (Algorithmus 3.1) ist mit linearem Platzbedarf möglich:
  Zur Bestimmung der i-ten Zeile braucht man nur die $(i-1)$-te Zeile.

- Die Bestimmung des optimalen Alignments (Algorithmus 3.2) benötigt die vollständige Tabelle (Platz $\Theta(mn)$).
  Mit linearem Platzaufwand ist lediglich das letzte Alignment-Paar zu bestimmen.
  Rekursive Bestimmung des jeweils letzten Paares: linearer Platz, kubische Zeit.

- Effizienterer rekursiver Algorithmus: Bestimme Feld "in der Mitte" des Alignment-Pfades. Bestimme rekursiv zwei Alignments.
  Teile-und-herrsche-Strategie

Idee des rekursiven Algorithmus

Bestimme im optimalen Alignmentpfad ein Feld $(\lfloor \frac{m}{2} \rfloor, p)$ (auf der "Mittellinie")
Zeit: $O(mn)$, Platz: $O(n)$



Bestimme rekursiv die optimalen Pfade von $(0, 0)$ nach $(\lfloor \frac{m}{2} \rfloor, p)$ und von $(\lfloor \frac{m}{2} \rfloor, p)$ nach (m, n).

Bestimmung des Feldes auf der Mittellinie

$P_{i,j}$ sei die letzte Spalte in der Zeile $\lfloor m/2 \rfloor$ auf einem optimalen Alignment-Pfad von $(0,0)$ nach (i,j).

Initialisierung: $P_{\lfloor m/2 \rfloor, j} = j$ für $0 \leq j \leq n$.

Rekursion für $i > \lfloor m/2 \rfloor$: $P_{i,j} = \begin{cases} P_{i-1,j-1}, & \text{falls} D_{i,j} = D_{i-1,j-1} + d(S_1[i], S_2[j]) \\ P_{i-1,j}, & \text{falls} D_{i,j} = D_{i-1,j} + d(S_1[i], \_) \\ P_{i,j-1}, & \text{falls} D_{i,j} = D_{i,j-1} + d(\_, S_2[j]) \end{cases}$

Das gesuchte Feld auf der Mittellinie ist $[\lfloor m/2 \rfloor, P_{m,n}]$.

Bestimmung der $P_{i,j}$ gleichzeitig mit Berechnung der $D_{i,j}$ in Zeit $O(mn)$ mit Platzbedarf $O(n)$.

Algorithmus 3.5 Alignment-Rekursiv (Hirschberg)

Eingabe: Wörter S_1, S_2, $|S_1| = m$, $|S_2| = n$
Ausgabe: Optimales Alignment von S_1 und S_2
Aufruf: $\mathrm{AR}(S_1, S_2)$

(1) **if** $m = 1$ **then return** Alignment nach Algorithmus 3.2;
(2) Bestimme Feld $(\lfloor m/2 \rfloor, p)$ im optimalen Alignment-Pfad;
(3) **return**
$\mathrm{AR}(S_1[1 \ldots \lfloor m/2 \rfloor], S_2[1 \ldots p]) \cdot \mathrm{AR}(S_1[\lfloor m/2 \rfloor + 1 \ldots m], S_2[p+1 \ldots n])$;

Satz. Algorithmus 3.5 bestimmt ein optimales Alignment mit einem Platzbedarf von $O(m + n)$ in einer Zeit von $O(mn)$.

Beispiel

$S_1 =$ `tempel`, $S_2 =$ `treppe`; 1. Wert: $D_{i,j}$, 2. Wert: $P_{i,j}$

		t	r	e	p	p	e
	0	1	2	3	4	5	6
t	1	0	1	2	3	4	5
e	2	1	1	1	2	3	4
m	$3,0$	$2,1$	$2,2$	$2,3$	$2,4$	$3,5$	$4,6$
p	$4,0$	$3,1$	$3,1$	$3,2$	$2,3$	$2,4$	$3,4$
e	$5,0$	$4,1$	$4,1$	$3,1$	$3,3$	$3,3$	$2,4$
l	$6,0$	$5,1$	$5,1$	$4,1$	$4,1$	$4,3$	$3,4$

Feld des Alignment-Pfades: $(3,4)$ (in der Tabelle markiert).
Bestimme rekursiv die optimalen Alignments von (`tem`,`trep`) und (`pel`,`pe`).

3.5 Levenshtein-Abstand in subquadratischer Zeit

- Ideen:
  - Statt $D_{i,j}$ benutze Differenzen (Offsets) zwischen $D_{i,j}$-Werten.
    $H_{i,j} = D_{i,j} - D_{i-1,j}$, $V_{i,j} = D_{i,j} - D_{i,j-1}$.
  - Berechne H-Werte nur in den Zeilen $p \cdot i$, V-Werte nur in den Spalten $q \cdot j$.
  - Levenshtein-Abstand ist aus diesen V- und H-Werten berechenbar.
  - Zur effizienten Berechnung der Offset-Werte benötigt man eine Tabelle (Block-funktion); in einem Präprozessing zu ermitteln.

- Laufzeit bei optimaler Wahl von p, q: $O(mn/\log(mn))$.

- Idee ist auf andere Algorithmen anwendbar;
  erstmals angewendet für die Multiplikation Boolescher Matrizen durch Arlazarov, Dinic, Kronrod und Faradžev (1970) ("Four Russians")

Offset-Werte

Statt der Werte $D_{i,j}$ bestimmt man Differenzen (Offset-Werte) benachbarter $D_{i,j}$-Werte.

vertikale Offsets: $V_{i,j} = D_{i,j} - D_{i-1,j}$
horizontale Offsets: $H_{i,j} = D_{i,j} - D_{i,j-1}$

Lemma. $D_{m,n}$ kann in $O(m+n)$ Schritten aus den Werten $V_{i,0}(1 \leq i \leq m)$ und $H_{m,j}(1 \leq j \leq n)$ berechnet werden.

Vorteil der Offsets: mögliche Werte nur $\{-1, 0, 1\}$.

Offset-Werte: Rekursionsbeziehungen

Satz. Es seien S_1 und S_2 Wörter mit $|S_1| = m$, $|S_2| = n$. Es gilt

- $H_{0,j} = 1$ für $1 \leq j \leq n$.
  $H_{i,j} = \min\{d(S_1[i], S_2[j]) - V_{i,j-1}, H_{i-1,j} - V_{i,j-1} + 1, 1\}$ für $1 \leq i \leq m$, $1 \leq j \leq n$.
- $V_{i,0} = 1$ für $1 \leq i \leq m$.
  $V_{i,j} = \min\{d(S_1[i], S_2[j]) - H_{i-1,j}, V_{i,j-1} - H_{i-1,j} + 1, 1\}$ für $1 \leq i \leq m$, $1 \leq j \leq n$.

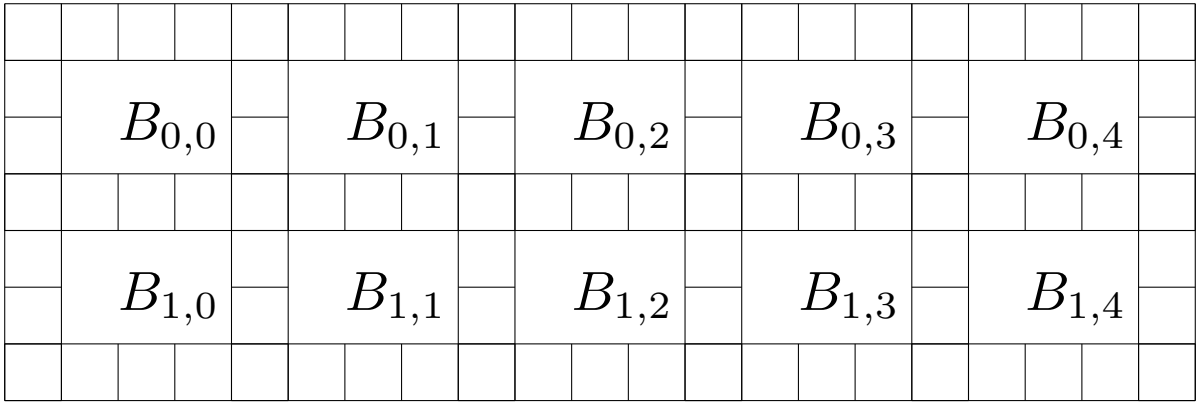
Folgerung. Es seien S_1 und S_2 Wörter mit $|S_1| = m$, $|S_2| = n$.

1. $H_{i,j} \in \{-1, 0, 1\}$ für $0 \leq i \leq m, 1 \leq j \leq n$,
   $V_{i,j} \in \{-1, 0, 1\}$ für $1 \leq i \leq m, 0 \leq j \leq n$.
2. Für $0 \leq i \leq m - p$ und $0 \leq j \leq n - q$ sind die Vektoren
   $(V_{i+1,j+q}, V_{i+2,j+q}, \ldots, V_{i+p,j+q})$ und $(H_{i+p,j+1}, H_{i+p,j+2}, \ldots, H_{i+p,j+q})$
   eindeutig durch $(V_{i+1,j}, V_{i+2,j}, \ldots, V_{i+p,j})$, $(H_{i,j+1}, H_{i,j+2}, \ldots, H_{i,j+q})$,
   $S_1[i+1 \ldots i+p]$ und $S_2[j+1 \ldots j+q]$ bestimmt.

Zerlegung der Tabelle in (p, q)-Blöcke

Gegeben: S_1, S_2 mit $|S_1| = m$, $|S_2| = n$.
Annahme (der Einfachheit halber): $m = pm'$ und $n = qn'$.

Block $\mathbf{B}_{i',j'}$ mit $0 \leq i' \leq m' - 1, 0 \leq j' \leq n' - 1$
enthält die Felder (i, j) mit $i'p \leq i \leq (i' + 1)p$, $j'q \leq j \leq (j' + 1)q$.
Das folgende Bild zeigt eine Überdeckung für $m = 6, n = 20, p = 3, q = 4$.



Ideen des Algorithmus

Unterer und rechter Rand eines Blockes sind eindeutig durch oberen und linken Rand sowie die zum Block gehörenden Teilwörter bestimmt. (Blockfunktion)

Anzahl der Argumente der Blockfunktion: $(3\sigma)^{p+q}$

1. Phase: Bestimme die Blockfunktion durch dynamische Programmierung.
Zeit: $O((3\sigma)^{p+q} \cdot p \cdot q)$.
Blockfunktion wird in Tabelle abgelegt.

2. Phase: Bestimme die Offset-Werte auf den Blockrändern durch Dynamische Programmierung unter Verwendung der Blockfunktion.
Wert der Blockfunktion wird der Tabelle entnommen, Aufwand je Block $O(p+q)$
Gesamte Zeit: $O(mn(p+q)/(pq))$.

Wahl von p und q

Theoretisch. Der Einfachheit halber sei $m = n$.
Wähle $p = q = (\log_{3\sigma} n)/2$.

Phase 1: $O(n \log^2 n)$ Schritte.
Phase 2: $O(n^2/\log n)$ Schritte.

Satz. Für S_1, S_2 mit $|S_1| = |S_2| = n$ kann der Levenshtein-Abstand mit einem Aufwand von $O(n^2/\log n)$ bestimmt werden.

Praktisch. $p = 1$ und Verwendung von Bit-Arithmetik.
Der Wert der Blockfunktion wird mit Aufwand $O(1)$ je Block bestimmt.
Beschleunigung um den Faktor q. (Realistisch z.B. $q = 6$).

3.6 Weitere Algorithmen für die inexakte Suche

Gegeben: Suchwort P mit $|P| = m$, Text T mit $|T| = n$, Schranke k
Gesucht: approximative Vorkommen von P in T mit höchstens k Fehlern

- Anwendung von Bit-Parallelismus (Byte-Länge w)

  Erweiterung des Shift-And-Algorithmus [Wu, Manber 1992]
  Laufzeit: $O(kn)$ für kurze Suchwörter, $O(k\lceil m/w \rceil n)$ im allgemeinen

  Verbesserte Variante [Baeza-Yates, Navarro 1999]
  Laufzeit: $O(n)$ für sehr kurze Suchwörter, $O(\lceil km/w \rceil n)$ im allgemeinen

- Filtermethoden zur Konstruktion einer (in der Regel kleinen) Kandidatenmenge; dabei exakte Suche nach Menge von Teilwörtern von P

NEA für die inexakte Suche

$P = x_1 x_2 \cdots x_m$, $x_i \in \Sigma$;

NEA zur Erkennung von P mit höchstens k Fehlern hat

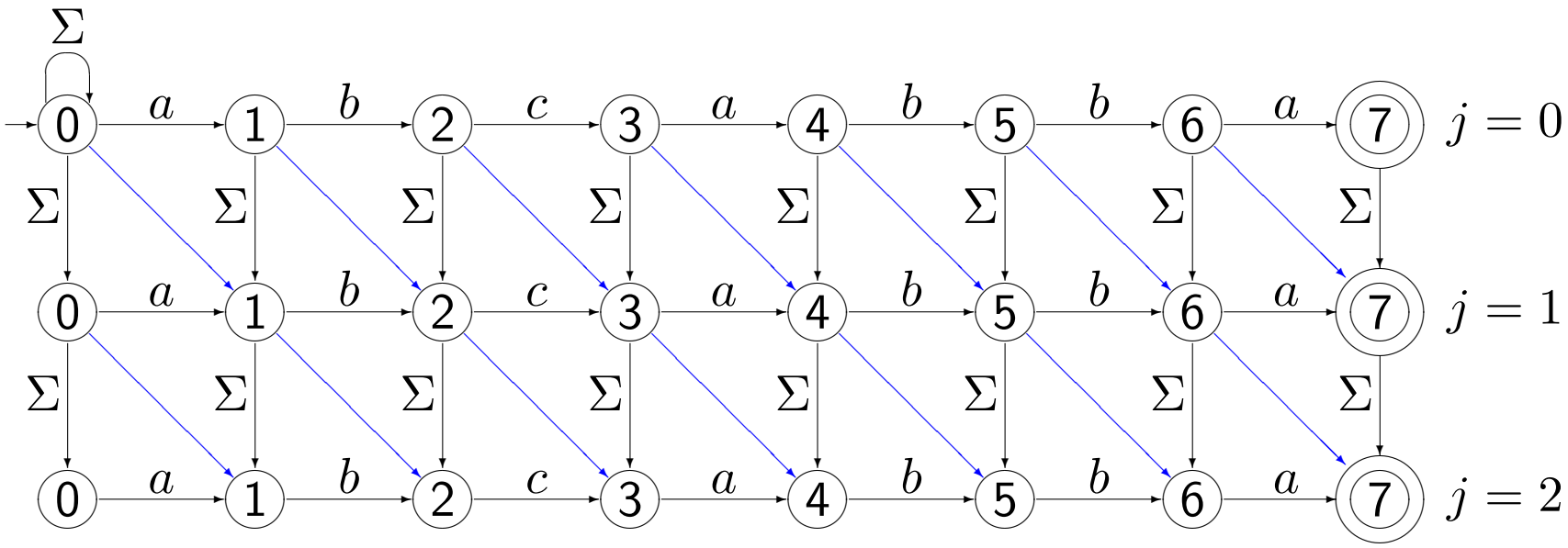
die Zustandsmenge $\{0, 1, \ldots, m\} \times \{0, 1, \ldots, k\}$,
die Transitionenmenge

$$\begin{aligned}
\delta \quad = \quad & \{((0,0), x, (0,0)) : x \in \Sigma\} \cup \{((i-1, j), x_i, (i, j)) : 1 \leq i \leq m, 0 \leq j \leq k\} \cup \\
& \{((i, j-1), x, (i, j)) : 0 \leq i \leq m, 1 \leq j \leq k, x \in \Sigma\} \cup \\
& \{((i-1, j-1), \varepsilon, (i, j)) : 0 \leq i \leq m, 1 \leq j \leq k\} \cup \\
& \{((i-1, j-1), x, (i, j)) : 1 \leq i \leq m, 1 \leq j \leq k, x \in \Sigma\},
\end{aligned}$$

den Startzustand $(0, 0)$
und die Menge der Endzustände $\{(m, j) : 0 \leq j \leq k\}$.

Beispiel

Für $P = abcabba$ und $k = 2$ ergibt sich der folgende NEA (Kanten ohne Beschriftung entsprechen Transitionen mit Symbolen aus $\Sigma \cup \{\varepsilon\}$):



Implementierung mit Bit-Vektoren

$B[x]$ wie gehabt für die Vorkommen der Symbole in P,
Z_j, $0 \leq j \leq k$, für die alten Zustände des NEA und
Z'_j, $0 \leq j \leq k$, für die neuen Zustände des NEA.

Vorkommen mit höchstens j Fehlern $\iff$ m-tes Bit in Z_j ist 1.

Initialisierung: $Z_j \leftarrow (1 << (j-1))$, für $1 \leq j \leq k$.
Aktualisierung für Textsymbol x:
$Z'_0 \leftarrow ((Z_0 << 1 \,|\, 1) \,\&\, B[x]$;
$Z'_j \leftarrow (((Z_j << 1 \,|\, 1) \,\&\, B[x]) \,|\, Z_{j-1} \,|\, (Z_{j-1} << 1) \,|\, (Z'_{j-1} << 1)$, für $1 \leq j \leq k$;
$Z_j \leftarrow Z'_j$, für $0 \leq j \leq k$.

Zeit für Aktualisierung beträgt $(k+1)$ Schritte, denn:
zur Aktualisierung von Z_j braucht man den alten und den neuen Wert von Z_{j-1}.

Verbesserung von Baeza-Yates und Navarro

Speichere den Automaten diagonalenweise
Bitvektor D_d für Zustände $\{(j+d, j) : 0 \leq j \leq k\}$ auf Diagonalen d

Vorteil der Diagonalendarstellung:

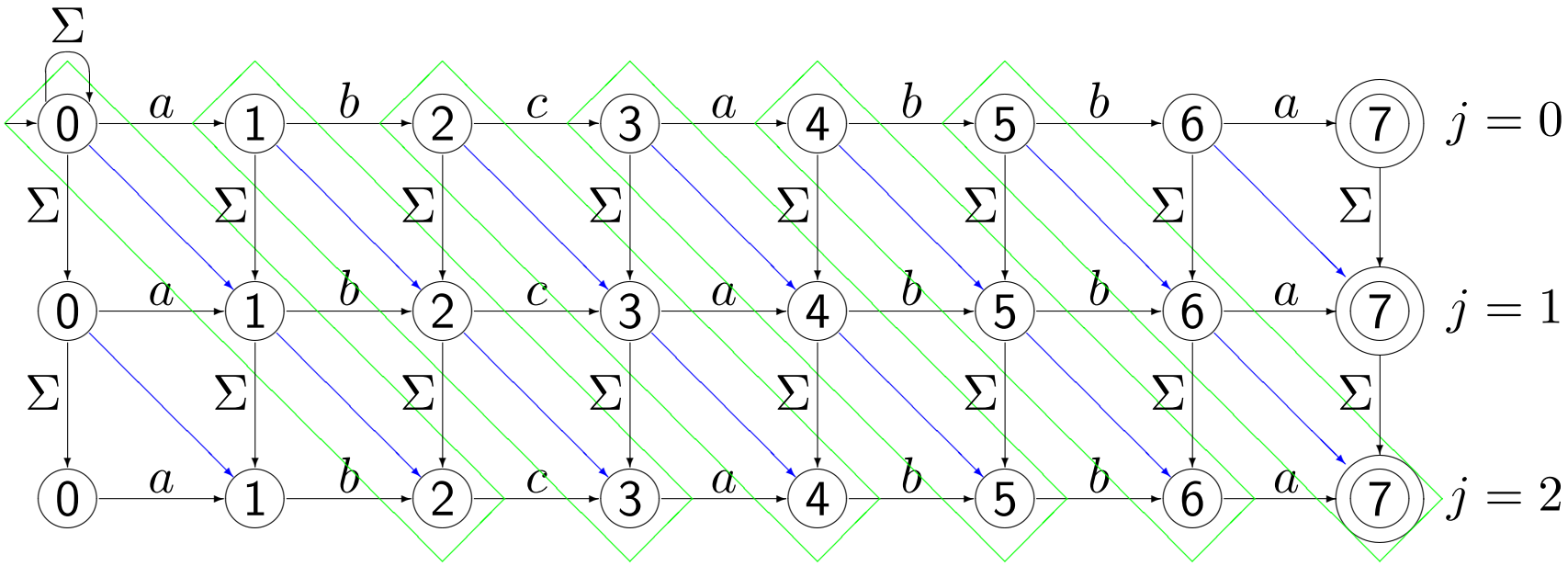
Neuer Wert von D_d hängt nur von den alten Werten von D_{d-1}, D_d, D_{d+1} ab.
Das heißt: Werte D_d können parallel in einem Schritt berechnet werden.

Falls $km \leq w$: Speicherung aller D_d in einem Byte;
Aktualisierung in Zeit $O(1)$

Allgemein: Speicherung der D_d in $\lceil km/w \rceil$ Bytes;
Aktualisierung in Zeit $O(\lceil km/w \rceil)$

Beispiel

Für $P = abcabba$ und $k = 2$ ergeben sich folgende Diagonalen:



Filtermethoden

Idee: Zerlege P in $(k+1)$ Teilwörter $P = P_1 P_2 \cdots P_{k+1}$.
Kommt P mit höchstens k Fehlern vor, so kommt mindestens eins der Wörter P_i exakt vor.

Filterung: Führe eine Suche nach der Menge $\{P_1, P_2, \ldots, P_{k+1}\}$ durch.
(geeignete Algorithmen: Horspool bzw. BNDM für Mengen, Wu-Manber)

Verifikation: Versuche, die exakten Treffer zu inexakten Vorkommen auszudehnen.

Praktikabel für kleine Werte von k. (mittlere Laufzeit: $O(kn \log_\sigma m/m)$)

Implementiert in `agrep` (Wu, Manber)

Filterung – Beispiel

$P = \texttt{textalgorithmen}$, $k = 3$, Zerlegung: $\texttt{text} \cdot \texttt{algo} \cdot \texttt{rith} \cdot \texttt{men}$

Das heißt: exakte Suche nach $\{\texttt{text}, \texttt{algo}, \texttt{rith}, \texttt{men}\}$

Hierarchische Verifikation:
Kommt das Wort `textalgorithmen` mit höchstens 3 Fehlern vor,
so kommt eines der Wörter `textalgo`, `rithmen` mit höchstens 1 Fehler vor.

Beispiel: Nach Finden eines exakten Vorkommens von `men`
zuerst Verifikation von `rithmen` mit höchstens 1 Fehler,
bei positivem Ausgang Verifikation des gesamten Wortes mit höchstens 3 Fehlern.

Inexakte Suche nach mehreren Wörtern

- **Gegeben:** Text T, Menge von Wörtern $\mathcal{P}$,
  Gesucht: inexakte Vorkommen von Wörtern aus $\mathcal{P}$ in T.

- Anwendung von Filteralgorithmen zur effizienten Lösung

- bei Fehlerschranke k Zerlegung jedes Wortes aus $\mathcal{P}$ in $(k+1)$ Teilwörter und exakte Suche (analog zur inexakten Suche für ein Wort

- bei Fehlerschranke 1 effizienter Algorithmus unter Benutzung von Hash-Funktionen (Erweiterung des Karp-Rabin-Algorithmus)
  Mittlere Laufzeit: $O(rm + mn)$ für die Suche mit 1 Fehler nach r Wörtern der Länge m

Suche nach mehreren Wörtern mit Hashing

Definition. Für ein Wort S der Länge m und $1 \leq j \leq m$ sei $S^{(j)} = S[1 \ldots j-1]S[j+1 \ldots m]$, d.h. S ohne sein j-tes Symbol.

Satz. Haben zwei Wörter S_1 und S_2 der Länge m einen Levenshtein-Abstand von höchstens 1, dann gibt es $1 \leq i, j \leq m$ mit $S_1^{(i)} = S_2^{(j)}$.

Idee des Algorithmus

Präprozessing: Bestimme für jedes Wort $P \in \mathcal{P}$ und jedes $1 \leq j \leq m$ Hash-Werte für $P^{(j)}$.

Suche: Bestimme für ein Textfenster $T_i = T[i \ldots i+m-1]$ und jedes $1 \leq j \leq m$ Hash-Werte für $T_i^{(j)}$.
Bei Übereinstimmung der Hash-Werte Vergleich von Text und Suchwort.

Wahl der Hash-Funktion: Wie beim Karp-Rabin-Algorithmus

Alignment an einem Trie

Gegeben: Menge von Wörtern $\mathcal{P} = \{P_1, P_2, \ldots, P_r\}$ gegeben durch ihren Trie $\mathcal{T}$, Wort S

Gesucht: Wort aus $\mathcal{P}$ mit geringstem Abstand zu S

Anwendung: Rechtschreibprüfung mit Korrekturvorschlag

- Bestimme für jeden Knoten v von $\mathcal{T}$ und alle $1 \leq i \leq |S|$ den Abstand $D_{v,i} = D(\mathcal{L}(v), S[1 \ldots i])$, wobei $\mathcal{L}(v)$ die Beschriftung des Weges nach v ist.

- Induktionsanfang: Abstände für die Wurzel $root$
  $D_{root,0} = 0$, $D_{root,i} = D_{root,i-1} + d(\_, S[i])$.

- Induktionsschritt: Knoten v habe Vorgänger u mit Kantenbeschriftung a
  $D_{v,i} = \min\{D_{u,i-1} + d(a, S[i]), D_{u,i} + d(a, \_), D_{v,i-1} + d(\_, S[i])\}$

- Noch besser ist die Verwendung des minimalen DEA für P, der ein gerichteter azyklischer Graph (DAG) ist.

3.7 Multiple Alignments

Gegeben: Menge von Wörtern $\{S_1, S_2, \ldots, S_k\}$
Gesucht: gemeinsames optimales Alignment dieser Wörter
Anwendung: Bioinformatik
Spezialfall: längste gemeinsame Teilfolge (LCS)

Satz. LCS ist **NP**-vollständig.

Dynamische Programmierung liefert einen Algorithmus der Komplexität $O(n^k)$; nicht praktikabel für $n = 100$, $k \geq 8$.

Zahlreiche Techniken aus der kombinatorischen Optimierung wurden und werden zur Konstruktion von Näherungslösungen eingesetzt.

Multiple Alignments – Definition und Bewertung

Definition (Multiples Alignment)
Es seien $S_1, S_2, \ldots, S_k$ Wörter über Σ. Ein multiples Alignment von $(S_1, S_2, \ldots, S_k)$ ist ein k-Tupel $(S'_1, S'_2, \ldots, S'_k)$ von Wörtern gleicher Länge über $\Sigma \cup \{\_\}$, wobei S'_t aus S_t durch Einfügen von _ entsteht.

Definition (Sums-of-Pairs-Bewertung)
Es sei $d : (\Sigma \cup \{\_\})^2 \to \mathbb{Z}$ eine Bewertungsfunktion mit $d(\_, \_) = 0$.
Die Sums-of-Pairs-Bewertung (SP-Bewertung) eines Alignments $(S'_1, S'_2, \ldots, S'_k)$ mit $|S'_t| = n$ ergibt sich als

$$d(S'_1, S'_2, \ldots, S'_k) = \sum_{t=1}^{n} d(S'_1[t], S'_2[t], \ldots, S'_k[t]) \text{ mit}$$

$$d(a_1, a_2, \ldots, a_k) = \sum_{1 \le i < j \le k} d(a_i, a_j) \text{ für } a_1, a_2, \ldots, a_k \in \Sigma \cup \{\_\}.$$

Näherungslösung – Mittelpunkt-Alignment

Gegeben: $\{S_1, S_2, \ldots, S_k\}$, Abstandsfunktion d (z.B. Levenshtein-Abstand)

Bestimme i^*, so dass $\sum_{j \neq i^*} D(S_{i^*}, S_j) \to \min$ (S_{i^*} ist der "Mittelpunkt".)

Konstruiere die optimalen paarweisen Alignments für (S_{i^*}, S_j).

Bilde ein multiples Alignment, so dass die optimalen paarweisen Alignments von (S_{i^*}, S_j) bis auf Einfügung von $(\_, \_)$ bestehen bleiben.

Satz Es sei d^* die Bewertung des Mittelpunkt-Alignments und D die Bewertung des optimalen multiplen Alignments von $\{S_1, S_2, \ldots, S_k\}$. Dann gilt $d^* \leq 2D$.

Mittelpunkt-Alignment – Beispiel

$S_1 = aaba, S_2 = baaba, S_3 = aaaa, S_4 = aaabb \rightarrow i^* = 1.$
Optimale paarweise Alignments mit S_1:

$$\begin{array}{ccccc} \_ & a & a & b & a \\ b & a & a & b & a \end{array} \qquad \begin{array}{cccc} a & a & b & a \\ a & a & a & a \end{array} \qquad \begin{array}{ccccc} a & a & b & \_ & a \\ a & a & b & b & a \end{array} .$$

Mittelpunkt-Alignment:

$$\begin{array}{ccccc} \_ & a & a & b & \_ & a \\ b & a & a & b & \_ & a \\ \_ & a & a & a & \_ & a \\ \_ & a & a & b & b & a \end{array}$$