technische universität
dortmund

Algorithmen auf Sequenzen

Fehlertolerantes Pattern-Matching

Dominik Kopczynski

Lehrstuhl für Algorithm Engineering (LS11)
Fakultät für Informatik
TU Dortmund

Beobachtung

- Bisher war die Mustersuche fast ausschließlich exakt.
- Fehlertolerantes Pattern-Matching wird an vielen Stellen verwendet: Rechtschreibkorrektur, Vorschläge, Read-Mapping in der Bioinformatik.
- Fehler konnten nur indirekt behandelt werden: mehrere ähnliche Pattern, erweiterte Patternklasse.
- Wie aber soll vorgegangen werden, wenn man z.B. einen Fehler (egal wo) erlauben möchte?

Abstands- und Ähnlichkeitsmaße

Zuerst muss formal definiert werden, was ähnlich überhaupt bedeutet. Idealerweise handelt es sich bei Abstandsmaßen um Metriken:

Definition (Metrik)

Sei X eine Menge. Eine Funktion $d : X \times X \to \mathbb{R}_{\geq 0}$ heißt Metrik, genau dann wenn:

1. $d(x, y) = 0$ genau dann, wenn $x = y$ (Definitheit),
2. $d(x, y) = d(y, x)$ für alle x, y (Symmetrie),
3. $d(x, y) \leq d(x, z) + d(z, y)$, für alle x, y, z (Dreiecksungleichung).

Hemming-Distanz

Für Strings gleicher Länge bietet sich die Hemming-Distanz als Abstandsmaß an.

Definition (Hemming-Distanz)

Für jedes Alphabet Σ und jedes $n \geq 0$ ist auf $X := \Sigma^n$ die *Hamming-Distanz* $d_{\mathrm{H}}(s, t)$ zwischen Strings s, t definiert als die Anzahl der Positionen, in denen sich s und t unterscheiden, also $d_{\mathrm{H}}(s, t) := \left|\{i \mid s_i \neq t_i\}\right|$.

q-gram-Distanz

Da für $|s| \neq |t|$ die Hamming-Distanz nicht definiert ist, kann die q-gram-Distanz eingesetzt werden. Seien dabei für einen String s die q-grams alle Substrings der Länge q, z.B. für $q = 3$, $s = \texttt{GATTACA} : \texttt{GAT}, \texttt{ATT}, \texttt{TTA}, \texttt{TAC}, \texttt{ACA}$.

Definition (q-gram-Distanz)

Für einen String $s \in \Sigma^*$ und ein q-gram $x \in \Sigma^q$ sei $N_x(s)$ die Anzahl der Vorkommen von x in s. Dann ist die *q-gram-Distanz* zwischen s und t definiert als

$$d_q(s,t) := \sum_{x \in \Sigma^q} |N_x(s) - N_x(t)|.$$

Edit/Levenshtein-Distanz

Leider handelt es sich bei der q-gram-Distanz um keine Metrik, so dass ein als am häufigsten Maß die Edit-Distanz (auch Levenshtein-Distanz genannt) verwendet wird.

Definition (Edit-Distanz, Levenshtein-Distanz)

Die *Edit-Distanz* zwischen zwei Strings s und t ist definiert als die Anzahl der Edit-Operationen, die man *mindestens* benötigt, um einen String in einen anderen zu überführen. *Edit-Operationen* sind jeweils Löschen, Einfügen und Verändern eines Zeichens.

Edit/Levenshtein-Distanz

Beispiele für die Edit-Distanz:

```
- a n a n a s
b a n a n a -
```

```
d u c k t a l e s
d u c t t a p e -
```

Edit/Levenshtein-Distanz

Beispiele für die Edit-Distanz:

```
- a n a n a s
b a n a n a -
```
Edit: 2

```
d u c k t a l e s
d u c t t a p e -
```
Edit: 3

Visualisierung des Edit-Prozesses

Es gibt viele Varianten einen String s in t zu überführen:

```
h a n d     h a n d - - - -     h a n d -
a n d i     - - - - a n d i     - a n d i
```

Visualisierung des Edit-Prozesses

Es gibt viele Varianten einen String s in t zu überführen:

```
h a n d     h a n d - - - -     h a n d -
a n d i     - - - - a n d i     - a n d i
```

Uns interessiert die minimale Anzahl von Edit-Operationen. Da die Edit-Operationen die Reihenfolge der Buchstaben nicht verändern, genügt es den Prozess von links nach rechts zu betrachten.

Visualisierung des Edit-Prozesses

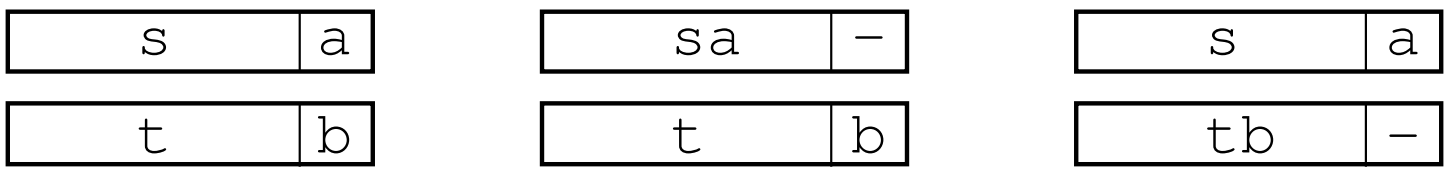
Der Edit-Prozess kann beispielsweise (wie in den vorherigen Beispielen gezeigt) durch ein Alignment visualisiert werden.

Definition (Globales Alignment)

Ein globales Alignment A von $s, t \in \Sigma^*$ ist ein String über $(\Sigma \cup \{-\})^2 \setminus \{(-,-)\}$ mit $\pi_1(A) = s$ und $\pi_2(A) = t$, wobei π_1 ein String-Homomorphismus mit $\pi_1(A_i) = \pi_1((a,b)_i) := a$ für $a \in \Sigma$ und $\pi_1((-,b)_i) := \epsilon$ ist. Analog ist π_2 der String-Homomorphismus mit $\pi_2(A_i) = \pi_2((a,b)_i) := b$ für $b \in \Sigma$ und $\pi_2((a,-)_i) := \epsilon$.

Berechnung der Edit-Distanz

Beobachtung: Ein Alignment der Strings sa und tb kann auf genau 3 verschieden Arten enden. Dabei sind $s, t \in \Sigma^*$ und $a, b \in \Sigma$.



Aus dieser Beobachtung ergibt sich das folgende Lemma zur Berechnung der Edit-Distanz.

Berechnung der Edit-Distanz

Lemma (Rekurrenz zur Edit-Distanz)

Seien $s, t \in \Sigma^*$, sei ϵ der leere String, $a, b \in \Sigma$ einzelne Zeichen. Ferner sei d die Edit-Distanz, dann gilt:

$$d(s, \epsilon) = |s|,$$
$$d(\epsilon, t) = |t|,$$
$$d(a, b) = \begin{cases} 1 & \text{falls } a \neq b, \\ 0 & \text{falls } a = b, \end{cases}$$
$$d(sa, tb) = \min \begin{cases} d(s, t) + d(a, b), \\ d(s, tb) + 1, \\ d(sa, t) + 1. \end{cases}$$

Berechnung der Edit-Distanz

Beweis (Rekurrenz zur Edit-Distanz)

Die elementaren Fälle $d(s, \epsilon), d(\epsilon, t), d(a, b)$ *sind trivial.*

Berechnung der Edit-Distanz

Beweis (Rekurrenz zur Edit-Distanz)

Die elementaren Fälle $d(s, \epsilon), d(\epsilon, t), d(a, b)$ sind trivial.
Für $d(sa, tb)$ gilt die Richtung "$\leq$", da die drei Möglichkeiten zulässige Edit-Operationen darstellen. Zu zeigen ist die Ungleichung "$\geq$".

Berechnung der Edit-Distanz

Beweis (Rekurrenz zur Edit-Distanz)

Die elementaren Fälle $d(s, \epsilon), d(\epsilon, t), d(a, b)$ *sind trivial.*
Für $d(sa, tb)$ *gilt die Richtung "*$\leq$*", da die drei Möglichkeiten zulässige Edit-Operationen darstellen. Zu zeigen ist die Ungleichung "*$\geq$*". Beweis durch Widerspruch: Angenommen es gilt* $d(sa, tb) < \min(\dots)$.

Berechnung der Edit-Distanz

Beweis (Rekurrenz zur Edit-Distanz)

Die elementaren Fälle $d(s, \epsilon), d(\epsilon, t), d(a, b)$ *sind trivial. Für* $d(sa, tb)$ *gilt die Richtung "*$\leq$*", da die drei Möglichkeiten zulässige Edit-Operationen darstellen. Zu zeigen ist die Ungleichung "*$\geq$*". Beweis durch Widerspruch: Angenommen es gilt* $d(sa, tb) < \min(\dots)$*. Ein Alignment von sa, tb muss jedoch auf eine der drei Arten enden: Entweder a steht über b, oder a steht über −, oder − steht über b.*

Berechnung der Edit-Distanz

Beweis (Rekurrenz zur Edit-Distanz)

Die elementaren Fälle $d(s, \epsilon), d(\epsilon, t), d(a, b)$ *sind trivial. Für* $d(sa, tb)$ *gilt die Richtung "*$\leq$*", da die drei Möglichkeiten zulässige Edit-Operationen darstellen. Zu zeigen ist die Ungleichung "*$\geq$*". Beweis durch Widerspruch: Angenommen es gilt* $d(sa, tb) < \min(\dots)$*. Ein Alignment von sa, tb muss jedoch auf eine der drei Arten enden: Entweder a steht über b, oder a steht über −, oder − steht über b. Je nachdem, welcher Fall im optimalen Alignment von sa und tb eintritt, müsste bereits* $d(s, t)$ *oder* $d(s, tb)$ *oder* $d(sa, t)$ *kleiner als optimal gewesen sein.* ↯ □

Berechnung der Edit-Distanz

- Die Edit-Distanz kann mit einem *Dynamic-Programming-Algorithmus* berechnet werden.
- Dynamic Programming (DP) ist eine algorithmische Technik, deren Anwendung sich immer dann anbietet, wenn Probleme im Prinzip rekursiv gelöst werden können, dabei aber (bei naiver Implementierung) dieselben Instanzen des Problems wiederholt gelöst werden müssten.
- Hierbei kann eine exponentielle Laufzeit auf eine polynomielle Laufzeit reduziert werden.

Berechnung der Edit-Distanz

- Die Edit-Distanz wird mit dem Algorithmus von Needleman und Wunsch (1970) wie folgt berechnet.
- Seien $m := |s|$ und $n := |t|$.
- Eine $(m+1) \times (n+1)$ Matrix $D = (D_{i,j})$ wird berechnet. Dabei ist $D[i,j]$ die Edit-Distanz zwischen dem s-Präfix der Länge i und dem t-Präfix der Länge j.

Berechnung der Edit-Distanz

Achtung: Hier rächt sich nun, dass die Indizierung von Sequenzen bei 0 beginnt; eine Zeile und Spalte ist nötig, um die leeren Präfixe zu behandeln; daher im Folgenden die Verschiebung um -1.

Initialisierung:

- Ein leeres Wort mit einem leeren Wort zu alignieren hat keine Kosten, also $D[0,0] = 0$.
- Die Präfixe von s in ein leeres Wort zu überführen benötigt $|s[:i]|$ Löschoperationen, also $D[i,0] = i$.
- Ein leeres Wort in die Präfixe von t zu überführen benötigt $|t[:j]|$ Einfügeoperationen, also $D[0,j] = j$.

Berechnung der Edit-Distanz

Alle weiteren Einträge können gemäß der rekursiven Vorschrift berechnet werden:

$$D[i,j] := \min \begin{cases} D[i-1,j-1] + d(s[i-1], t[j-1]), \\ D[i-1,j] + 1, \\ D[i,j-1] + 1. \end{cases}$$

Die Berechnung eines Feldes ist in konstanter Zeit möglich (drei Lookups), die Gesamtlaufzeit beträgt also $\mathcal{O}(mn)$.

Berechnung der Edit-Distanz

Alle weiteren Einträge können gemäß der rekursiven Vorschrift berechnet werden:

$$D[i,j] := \min \begin{cases} D[i-1,j-1] + d(s[i-1], t[j-1]), \\ D[i-1,j] + 1, \\ D[i,j-1] + 1. \end{cases}$$

Die Berechnung eines Feldes ist in konstanter Zeit möglich (drei Lookups), die Gesamtlaufzeit beträgt also $\mathcal{O}(mn)$. Die minimale Anzahl an Edit-Operationen steht dann im Feld $D[m,n]$.

Berechnung der Edit-Distanz

Beispiel einer Edit-Matrix mit $s = \texttt{andi}$ und $t = \texttt{handy}$.

	ϵ	h	a	n	d	y
ϵ	0	1	2	3	4	5
a	1	1	1	2	3	4
n	2	2	2	1	2	3
d	3	3	3	2	1	2
i	4	4	4	3	2	2

Berechnung der Edit-Distanz

Beispiel einer Edit-Matrix mit $s = \texttt{andi}$ und $t = \texttt{handy}$.

	ϵ	h	a	n	d	y
ϵ	0	1	2	3	4	5
a	1	1	1	2	3	4
n	2	2	2	1	2	3
d	3	3	3	2	1	2
i	4	4	4	3	2	2

Die Edit-Distanz zwischen s, t beträgt also 2.

Berechnung der Edit-Distanz

- Die Berechnung kann entweder Zeilenweise oder Spaltenweise erfolgen.
- Da für Feld $D[i, j]$ jeweils nur seine direkten "linken" und "oberen" Nachbarn notwendig sind, genügt es lediglich die aktuelle Zeile/Spalte und die Vorgängerzeile/spalte zu Speichern.
- Der Speicherverbrauch sinkt somit auf $\mathcal{O}(\min(m, n))$.
- Für die Rekonstruktion des Alignments ist aber die ganze Matrix erforderlich (wird später behandelt).

Berechnung der Edit-Distanz

```
1  def edit_distance(s, t):
2      m, n = len(s), len(t)
3      Dc = list(range(m + 1)) # current column
4      Dl = [0] * (m + 1) # last column
5
6      for j, tj in zip(range(1, n + 1), t):
7          Dl, Dc = Dc, Dl
8          Dc[0] = j
9          for i, si in zip(range(1, m + 1), s):
10             Dc[i] = min(Dl[i - 1] + (si != tj),
11                         Dl[i] + 1,
12                         Dc[i - 1] + 1)
13     return Dc[m]
```

Longest common Subsequence

- Mit ein paar Modifikationen kann der DP-Algorithmus genutzt werden, um die längste gemeinsame Teilsequenz (LCS) zu berechnen.
- Es muss lediglich die Logik umgekehrt werden.
- Insertionen, Deletionen und Substitutionen erzeugen keine Kosten.

Longest common Subsequence

- Mit ein paar Modifikationen kann der DP-Algorithmus genutzt werden, um die längste gemeinsame Teilsequenz (LCS) zu berechnen.
- Es muss lediglich die Logik umgekehrt werden.
- Insertionen, Deletionen und Substitutionen erzeugen keine Kosten.
- Lediglich ein Match bekommt einen Score von 1.
- Folglich wird die erste Zeile und Spalte in der Matrix mit 0 initialisiert.
- Beim Vergleich wird der größte Vorgängerwert genommen.

Longest common Subsequence

Sei $L = (L_{i,j})$ die Länge der längsten gemeinsamen Teilsequenz von $s[:i]$ und $t[:j]$:

$$d_{\mathrm{L}}(a,b) = \begin{cases} 0 & \text{falls } a \neq b, \\ 1 & \text{falls } a = b, \end{cases}$$

$$L[i,j] = \max \begin{cases} L[i-1,j-1] + d_{\mathrm{L}}(s[i-1], t[j-1]), \\ L[i-1,j], \\ L[i,j-1]. \end{cases}$$

Longest common Subsequence

Sei $L = (L_{i,j})$ die Länge der längsten gemeinsamen Teilsequenz von $s[:i]$ und $t[:j]$:

$$d_{\mathrm{L}}(a,b) = \begin{cases} 0 & \text{falls } a \neq b, \\ 1 & \text{falls } a = b, \end{cases}$$

$$L[i,j] = \max \begin{cases} L[i-1,j-1] + d_{\mathrm{L}}(s[i-1], t[j-1]), \\ L[i-1,j], \\ L[i,j-1]. \end{cases}$$

Auch hierbei benötigt man $\mathcal{O}(\min(m,n))$ Speicherplatz und $\mathcal{O}(mn)$ Zeit.

Longest common Factor

- Mit Hilfe des DP-Ansatzes lässt sich auch die Länge des längsten gemeinsamen Teilstrings (LCF) ermitteln.
- Tatsächlich hat diese Methode eine Laufzeit von $\mathcal{O}(mn)$ und somit eine asymptotisch schlechtere Laufzeit als der Suffix-Tree/Array-Ansatz mit Laufzeit $\mathcal{O}(m+n)$.
- Aufgrund der Einfachheit des Verfahrens ist dieser Ansatz einer Erwähnung Wert.

Longest common Factor

- Die Initialisierung der DP-Matrix ist die Gleiche wie beim LCS.
- Die Update-Formel sei:

$$L[i,j] = \begin{cases} L[i-1,j-1]+1 & \text{falls } s[i-1] = t[j-1], \\ 0 & \text{sonst.} \end{cases}$$

- Sei $lcf(s,t) = \max\{L[i,j] : 0 \leq i \leq m, 0 \leq j \leq n\}$.

Longest common Factor

- Die Initialisierung der DP-Matrix ist die Gleiche wie beim LCS.
- Die Update-Formel sei:

$$L[i,j] = \begin{cases} L[i-1,j-1]+1 & \text{falls } s[i-1] = t[j-1], \\ 0 & \text{sonst.} \end{cases}$$

- Sei $lcf(s,t) = \max\{L[i,j] : 0 \leq i \leq m, 0 \leq j \leq n\}$.

Beispiel:

	ϵ	a	b	a	b
ϵ	0	0	0	0	0
b	0	0	1	0	1
a	0	1	0	2	0
b	0	0	2	0	3
a	0	1	0	3	0

Edit-Graph

Im Folgenden soll das Problem der Berechnung des Sequenzalignments mit Hilfe eines Graphen dargestellt werden. Der *globale Alignment-Graph* oder auch *Edit-Graph* ist wie folgt definiert:

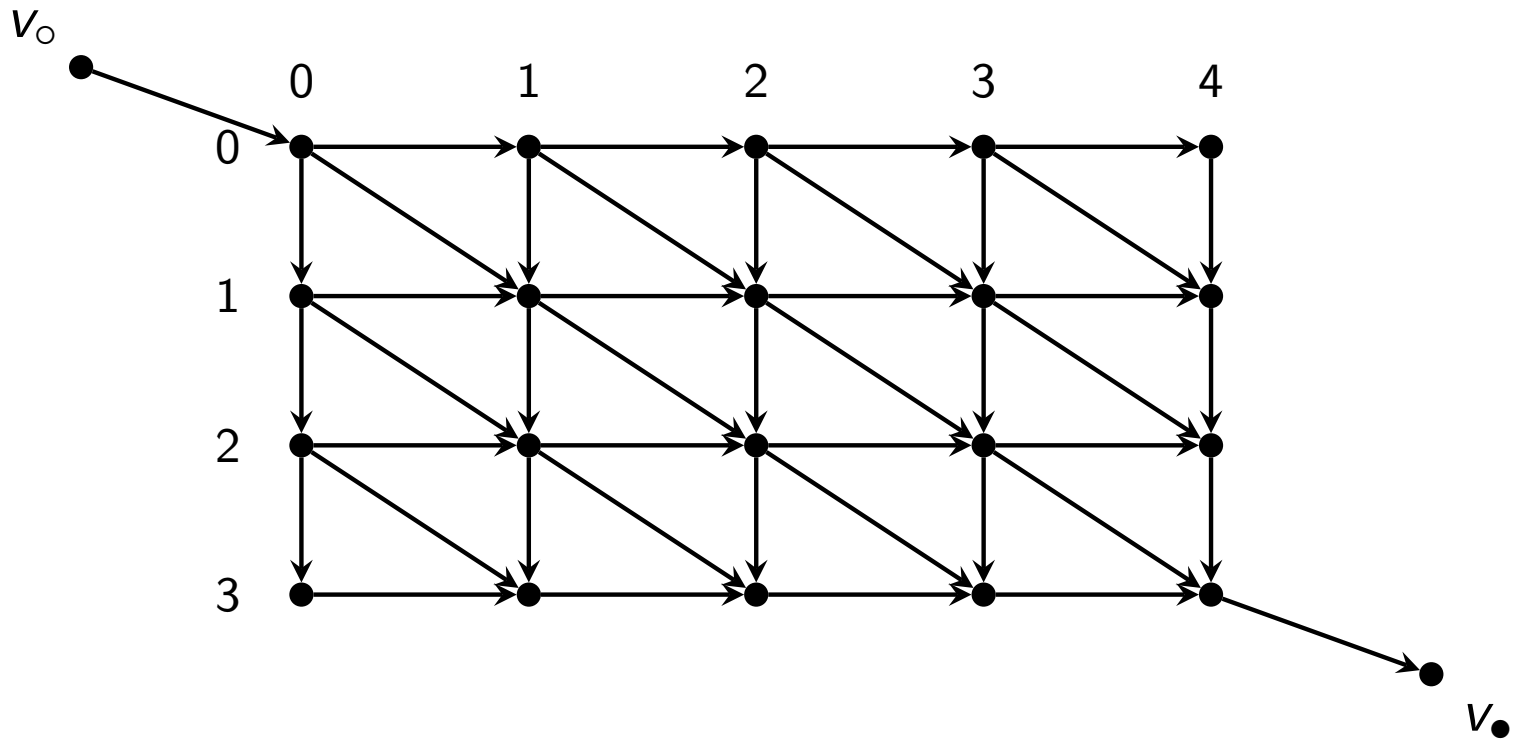
Definition (globaler Alignment-Graph, Edit-Graph)

- Knotenmenge $V := \{(i,j) : 0 \leq i \leq m, 0 \leq j \leq n\} \cup \{v_\circ, v_\bullet\}$
- Kanten:

	Kante	Label	Kosten
horizontal	$(i,j) \to (i,j+1)$	$\begin{bmatrix} - \\ t_j \end{bmatrix}$	1
vertikal	$(i,j) \to (i+1,j)$	$\begin{bmatrix} s_i \\ - \end{bmatrix}$	1
diagonal	$(i,j) \to (i+1,j+1)$	$\begin{bmatrix} s_i \\ t_j \end{bmatrix}$	$[s_i \neq t_j]$
Initialisierung	$v_\circ \to (0,0)$	ϵ	0
Finalisierung	$(m,n) \to v_\bullet$	ϵ	0

Edit-Graph

Beispiel für einen Edit- oder Alignment-Graph für ein globales Alignment:



Edit-Graph

- Jeder Pfad zwischen $v_\circ$ und $v_\bullet$ entspricht (durch die Konkatenation der Kantenlabel) genau einem Aligment von s und t.
- Mit der Definition des Edit-Graphen wird das Problem des Sequenzalignments als kürzestes-Wege-Problem in einem Edit-Graphen formuliert: Finde den Weg mit den geringsten Kosten von $v_\circ$ nach $v_\bullet$.
- Der Wert $D[i, j]$ lässt sich nun interpretieren als die minimalen Kosten eines Pfades vom Startknoten zum Knoten (i, j).

Anzahl globaler Alignments

- Die Anzahl $N(m, n)$ ist gleich der Anzahl der Möglichkeiten, eine Sequenz der Länge m in eine andere Sequenz der Länge n mit Hilfe von Edit-Operationen umzuschreiben.
- Dies ist ebenfalls gleich der Anzahl der Pfade im Edit-Graph vom Start- zum Zielknoten.
- Offensichtlich hängt $N(m, n)$ nur von der Länge der Sequenzen (und nicht von den Sequenzen selbst) ab.

Anzahl globaler Alignments

Lemma (Anzahl der Pfade im Edit-Graph)

Für die Anzahl $N(m,n)$ der Pfade im Edit-Graph vom Startknoten nach (m,n) gilt:

$$
\begin{aligned}
N(0,0) &= 1,\\
N(m,0) &= 1,\\
N(0,n) &= 1,\\
N(m,n) &= N(m-1,n-1) + N(m,n-1) + N(m-1,n).
\end{aligned}
$$

Anzahl globaler Alignments

Anzahl der Alignments $N(m, n)$ für $0 \leq m, n \leq 4$.

$m \backslash n$	0	1	2	3	4	...
0	1	1	1	1	1	...
1	1	3	5	7	9	...
2	1	5	13	25	41	...
3	1	7	25	63	129	...
4	1	9	41	129	321	...
⋮	⋮	⋮	⋮	⋮	⋮	⋱

Anzahl globaler Alignments

- Erkennbar, dass es schon für kleine m und n relativ viele Alignments gibt.
- Offensichtlich gilt $N(n,n) > 3N(n-1,n-1)$ und damit $N(n,n) > 3^n$.
- Man kann ausrechnen, dass sich asymptotisch $N(n,n) = \Theta\left(\sqrt{n} \cdot (1+\sqrt{2})^{2n+1}\right)$ ergibt; d.h., das Wachstum ist exponentiell mit Basis $(1+\sqrt{2})^2 \approx 5.8$.

Approximative Pattern-Suche

- Da ein Pattern P an jeder Stelle im Text T beginnen darf, ist es nicht sinnvoll ein globales Alignment, bzw. die Edit-Distanz von P und T zu berechnen.
- Zudem würde meistens die Edit-Distanz den Wert $d(P, T) \approx |T|$ annehmen, weil üblicherweise $P \ll T$ gilt.
- Es muss also die Definition der Tabelle D, bzw. des Edit-Graphen an die neue Situation angepasst werden.
- Der Einfachheit halber steht T immer oberhalb der Tabelle und P seitlich links.

Approximative Pattern-Suche

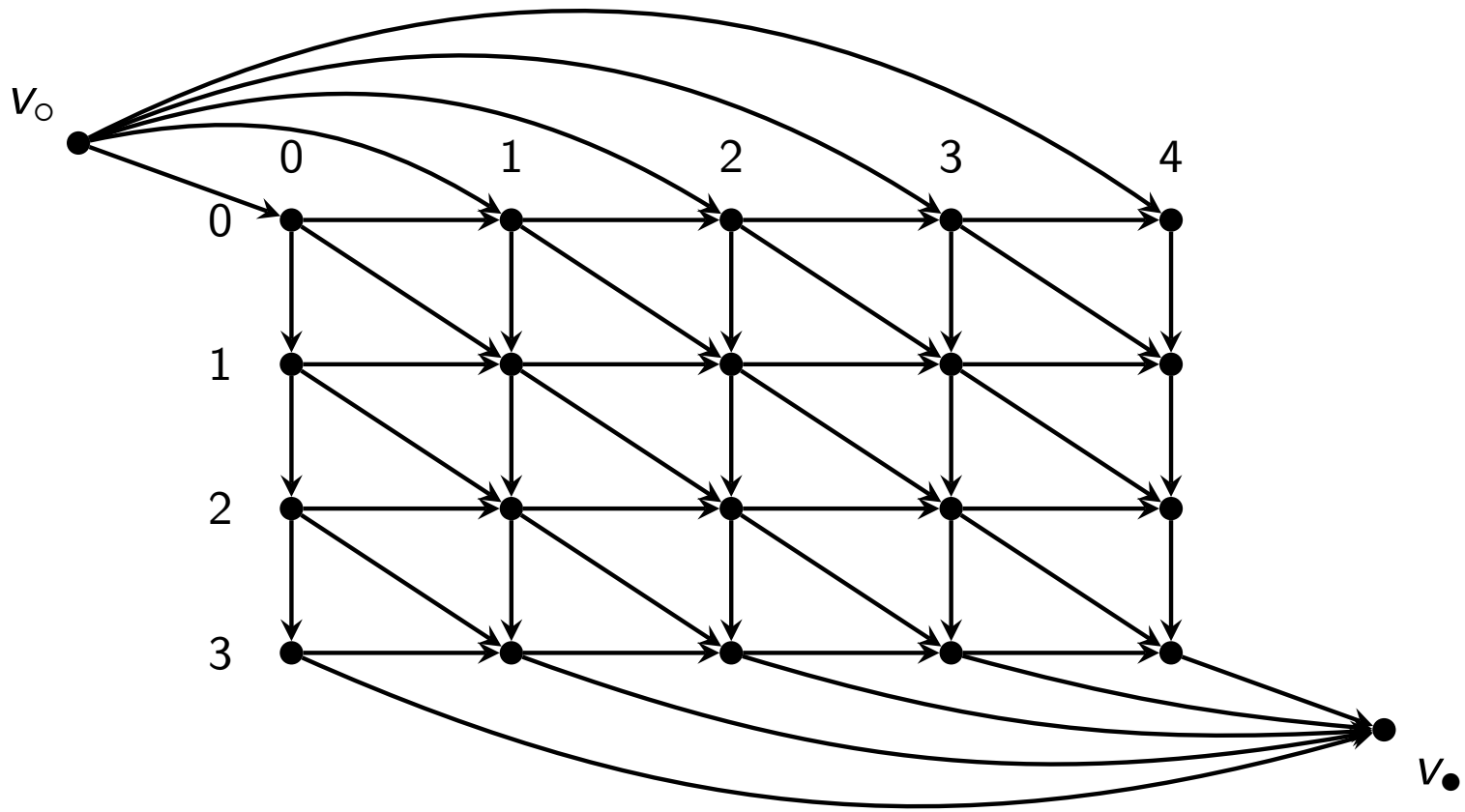
- Sei $D[i, j]$ die minimale Edit-Distanz des Präfixes $P[: i]$ mit einem Teilstring $T[j' : j]$ mit $j' \leq j$.
- Daraus ergibt sich die Initialisierung der nullten Zeile als $D[0, j] = 0$ für $0 \leq j \leq n$.

Approximative Pattern-Suche

- Sei $D[i, j]$ die minimale Edit-Distanz des Präfixes $P[: i]$ mit einem Teilstring $T[j' : j]$ mit $j' \leq j$.
- Daraus ergibt sich die Initialisierung der nullten Zeile als $D[0, j] = 0$ für $0 \leq j \leq n$.
- Ist die maximale Edit-Distanz k vorgegeben, wird nach Einträgen $D[m, j] \leq k$ gesucht; denn das bedeutet, dass das Pattern P an Position $j - 1$ des Textes mit höchstens k Fehlern (Edit-Operationen) endet (und bei irgendeinem $j' \leq j$ beginnt).
- Das Problem ist auch als *semiglobales Alignment* bekannt.

Edit-Graph

Beispiel für einen Edit- oder Alignment-Graph für ein semiglobales Alignment:



DP-Algorithmus von Ukkonen

- Ukkonen (ja, der selbe Ukkonen) hat 1985 eine modifizierte Version für das semiglobale Alignment veröffentlicht.
- Wenn eine Fehlerschranke k gegeben ist, muss nicht die komplette DP-Matrix berechnet werden.
- Sei $last_k(j)$ das tiefste Feld, das in Spalte j berechnet werden muss.

DP-Algorithmus von Ukkonen

- Ukkonen (ja, der selbe Ukkonen) hat 1985 eine modifizierte Version für das semiglobale Alignment veröffentlicht.
- Wenn eine Fehlerschranke k gegeben ist, muss nicht die komplette DP-Matrix berechnet werden.
- Sei $last_k(j)$ das tiefste Feld, das in Spalte j berechnet werden muss.
- Formal definiert sei

$$last_k(j) := \max\{i \mid D[i,j] \leq k,\ D[i',j] > k \text{ für alle } i' > i\}.$$

- Felder $D[i,j]$ mit $i > last_k(j)$ müssen nicht mehr betrachtet werden.

DP-Algorithmus von Ukkonen

Beispiel für die Felder, die beim Ukkonen-Algo' berechnet werden müssen für den Text $T = \texttt{AMOAMAMAOM}$, $P = \texttt{MAOAM}$, $k = 1$:

		A	M	O	A	M	A	M	A	O	M
	0	**0**	0	0	0	0	**0**	0	0	0	0
M	1	1	**0**	1	1	0	1	**0**	1	1	0
A	2	1	**1**	1	1	1	0	1	**0**	1	1
O	3	2	2	**1**	2	2	1	1	1	**0**	1
A	4	3	3	2	**1**	2	2	2	1	**1**	1
M	5	4	3	3	2	**1**	2	2	2	2	**1**

DP-Algorithmus von Ukkonen

- Woher weiß man, dass nach i' nur noch höhere Werte kommen ohne die ganze Spalte zu berechnen?

DP-Algorithmus von Ukkonen

- Woher weiß man, dass nach i' nur noch höhere Werte kommen ohne die ganze Spalte zu berechnen?
- Man macht sich die Einheitskosten der Edit-Operationen zu Nutze.
- Die Abstand zweier benachbarter Felder beträgt höchstens 1.
- Also gilt $last_k(j+1) \leq last_k(j) + 1$.

DP-Algorithmus von Ukkonen

Lemma

Seien die Kosten für alle Edit-Operationen $= 1$, *dann gilt* $d_{i,j} = d_{i-1,j-1}$ *oder* $d_{i,j} = d_{i-1,j-1} + 1$.

DP-Algorithmus von Ukkonen

Lemma

Seien die Kosten für alle Edit-Operationen $= 1$, *dann gilt* $d_{i,j} = d_{i-1,j-1}$ *oder* $d_{i,j} = d_{i-1,j-1} + 1$.

Beweis

- *Sei* $d_{0,0} = 0, d_{0,j} = 0, d_{i,0} = i$.

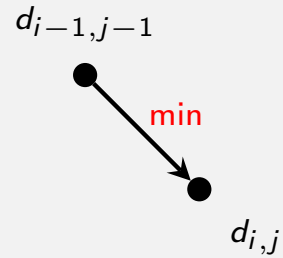
DP-Algorithmus von Ukkonen

Lemma

Seien die Kosten für alle Edit-Operationen $= 1$, *dann gilt* $d_{i,j} = d_{i-1,j-1}$ *oder* $d_{i,j} = d_{i-1,j-1} + 1$.

Beweis

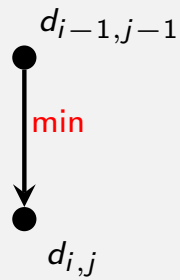
- *Sei* $d_{0,0} = 0, d_{0,j} = 0, d_{i,0} = i$.
- *Angenommen der minimale Pfad kommt von* $d_{i-1,j-1}$, *dann impliziert die Rekursionsgleichung dass* $d_{i,j} = d_{i-1,j-1}$ *oder* $d_{i,j} = d_{i-1,j-1} + 1$ *gilt.*



DP-Algorithmus von Ukkonen

Beweis

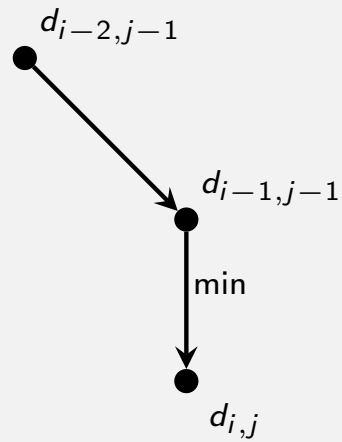
- *Angenommen der minimale Pfad kommt von $d_{i-1,j}$, dann gilt: $d_{i,j} = d_{i-1,j} + 1$.*



DP-Algorithmus von Ukkonen

Beweis

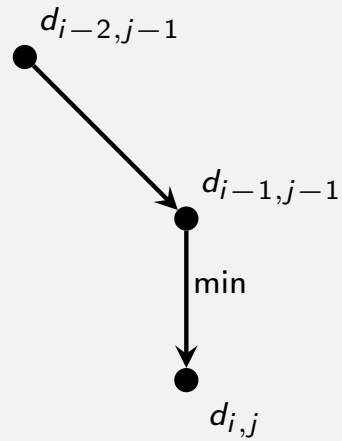
- *Angenommen der minimale Pfad kommt von $d_{i-1,j}$, dann gilt: $d_{i,j} = d_{i-1,j} + 1$.*
- *Demnach gilt nach Induktionsannahme: $d_{i-1,j} \geq d_{i-2,j-1}$.*



DP-Algorithmus von Ukkonen

Beweis

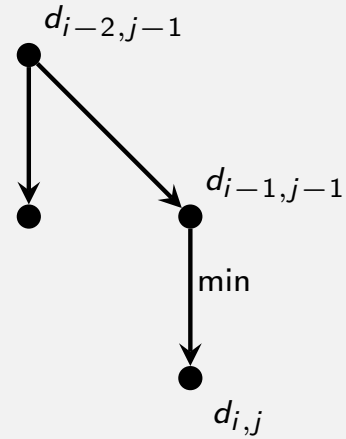
- *Angenommen der minimale Pfad kommt von $d_{i-1,j}$, dann gilt: $d_{i,j} = d_{i-1,j} + 1$.*
- *Demnach gilt nach Induktionsannahme: $d_{i-1,j} \geq d_{i-2,j-1}$.*
- *Daraus folgt: $d_{i,j} \geq d_{i-2,j-1} + 1$.*



DP-Algorithmus von Ukkonen

Beweis

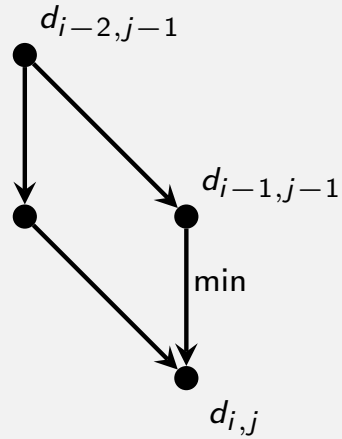
- *Angenommen der minimale Pfad kommt von $d_{i-1,j}$, dann gilt: $d_{i,j} = d_{i-1,j} + 1$.*
- *Demnach gilt nach Induktionsannahme: $d_{i-1,j} \geq d_{i-2,j-1}$.*
- *Daraus folgt: $d_{i,j} \geq d_{i-2,j-1} + 1$.*
- *Laut Rekursionsgleichung gilt: $d_{i-1,j-1} \leq d_{i-2,j-1} + 1$.*



DP-Algorithmus von Ukkonen

Beweis

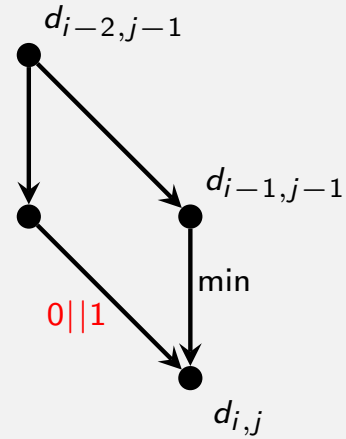
- *Angenommen der minimale Pfad kommt von $d_{i-1,j}$, dann gilt: $d_{i,j} = d_{i-1,j} + 1$.*
- *Demnach gilt nach Induktionsannahme: $d_{i-1,j} \geq d_{i-2,j-1}$.*
- *Daraus folgt: $d_{i,j} \geq d_{i-2,j-1} + 1$.*
- *Laut Rekursionsgleichung gilt: $d_{i-1,j-1} \leq d_{i-2,j-1} + 1$.*
- *Daraus folgt: $d_{i,j} \geq d_{i-1,j-1}$.*



DP-Algorithmus von Ukkonen

Beweis

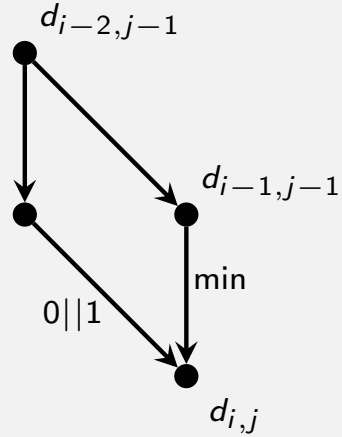
- *Angenommen der minimale Pfad kommt von* $d_{i-1,j}$*, dann gilt:* $d_{i,j} = d_{i-1,j} + 1$.
- *Demnach gilt nach Induktionsannahme:* $d_{i-1,j} \geq d_{i-2,j-1}$.
- *Daraus folgt:* $d_{i,j} \geq d_{i-2,j-1} + 1$.
- *Laut Rekursionsgleichung gilt:* $d_{i-1,j-1} \leq d_{i-2,j-1} + 1$.
- *Daraus folgt:* $d_{i,j} \geq d_{i-1,j-1}$.
- *Laut Rekursionsgleichung kann nur gelten:* $d_{i,j} = d_{i-1,j-1}$ *oder* $d_{i,j} = d_{i-1,j-1} + 1$.



DP-Algorithmus von Ukkonen

Beweis

- *Angenommen der minimale Pfad kommt von* $d_{i-1,j}$*, dann gilt:* $d_{i,j} = d_{i-1,j} + 1$.
- *Demnach gilt nach Induktionsannahme:* $d_{i-1,j} \geq d_{i-2,j-1}$.
- *Daraus folgt:* $d_{i,j} \geq d_{i-2,j-1} + 1$.
- *Laut Rekursionsgleichung gilt:* $d_{i-1,j-1} \leq d_{i-2,j-1} + 1$.
- *Daraus folgt:* $d_{i,j} \geq d_{i-1,j-1}$.
- *Laut Rekursionsgleichung kann nur gelten:* $d_{i,j} = d_{i-1,j-1}$ *oder* $d_{i,j} = d_{i-1,j-1} + 1$.



□

Beweis für minimalen Pfad aus $d_{i,j-1}$ analog.

DP-Algorithmus von Ukkonen

1. Zu Beginn sei $last_k(0) = \min(k, m)$.
2. Am Anfang jeder Spalte soll $last_k(j)$ um 1 inkrementiert werden.
3. Jede Spalte soll bis (einschließlich) $last_k(j)$ abgearbeitet werden.

DP-Algorithmus von Ukkonen

1. Zu Beginn sei $last_k(0) = \min(k, m)$.
2. Am Anfang jeder Spalte soll $last_k(j)$ um 1 inkrementiert werden.
3. Jede Spalte soll bis (einschließlich) $last_k(j)$ abgearbeitet werden.
4. Wenn $last_k(j) = m$ und $D[m, j] \leq k$ gilt, wurde ein Pattern gefunden.
5. Solange $D[last_k(j), j] > k$ gilt, soll $last_k(j)$ um 1 dekrementiert werden.

DP-Algorithmus von Ukkonen

```
1  def ukkonen(P, T, k):
2      m, n = len(P), len(T)
3      Dc = list(range(m + 1)) # current column
4      Dl = [k + 1] * (m + 1) # last column
5      lastk = min(k, m)
6      for j, tj in zip(range(1, n + 1), T):
7          Dc, Dl, lastk = Dl, Dc, lastk + 1
8          Dc[0] = 0
9          for i in range(1, lastk + 1):
10             pi = P[i - 1]
11             Dc[i] = min(Dl[i - 1] + (pi != tj),
12                         Dl[i] + 1,
13                         Dc[i - 1] + 1)
14         if lastk == m and Dc[m] <= k: yield j - 1, Dc[m]
15         while Dc[lastk] > k or lastk == m: lastk -= 1
```

Zusammenfassung

- Ähnlichkeiten / Distanzen zwischen zwei Strings lassen sich gut mit der Edit-Distanz ausdrücken.
- Durch Dynamic Programming lässt sich die Edit-Distanz, LCS und LCF in $\mathcal{O}(mn)$ berechnen.
- Der Ukkonen Algorithmus berechnet ein semiglobales Alignment (fehlertolerante Mustersuche).
- Dieser Algorithmus hat bei kleinem k und großem m eine erwartete Laufzeit von $\mathcal{O}(kn)$, aber eine worst-case Laufzeit von $\mathcal{O}(mn)$.

Fehlertoleranter Shift-And-Algorithmus

- Ein anderes Konzept zur fehlertoleranten Suche verfolgt eine weitere modifizierte Version des Shift-And-Algorithmus.
- Im Gegensatz zur exakten Suche werden hierbei aber $k+1$ viele NFAs bitparallel simuliert.
- Formaler ausgedrückt verwenden wir den Zustandsraum $Q = \{0, \ldots, k\} \times \{-1, 0, \ldots, |P| - 1\}$ für die Suche nach dem Pattern P mit maximal k Fehlern.

Konstruktion eines fehlertoleranten Shift-And-Algorithmus

- Jede der $k+1$ Ebene entspricht einem Shift-And-Automaten.
- Zustand (i,j) und $(i+1,j)$ mit $i \leq k$ sind durch eine Σ-Kante verbunden. Dies entspricht einer Insertion.

Konstruktion eines fehlertoleranten Shift-And-Algorithmus

- Jede der $k+1$ Ebene entspricht einem Shift-And-Automaten.
- Zustand (i,j) und $(i+1,j)$ mit $i \leq k$ sind durch eine Σ-Kante verbunden. Dies entspricht einer Insertion.
- Zustand (i,j) und $(i+1,j+1)$ sind durch eine Σ-Kante verbunden. Dies entspricht einer Substitution.

Konstruktion eines fehlertoleranten Shift-And-Algorithmus

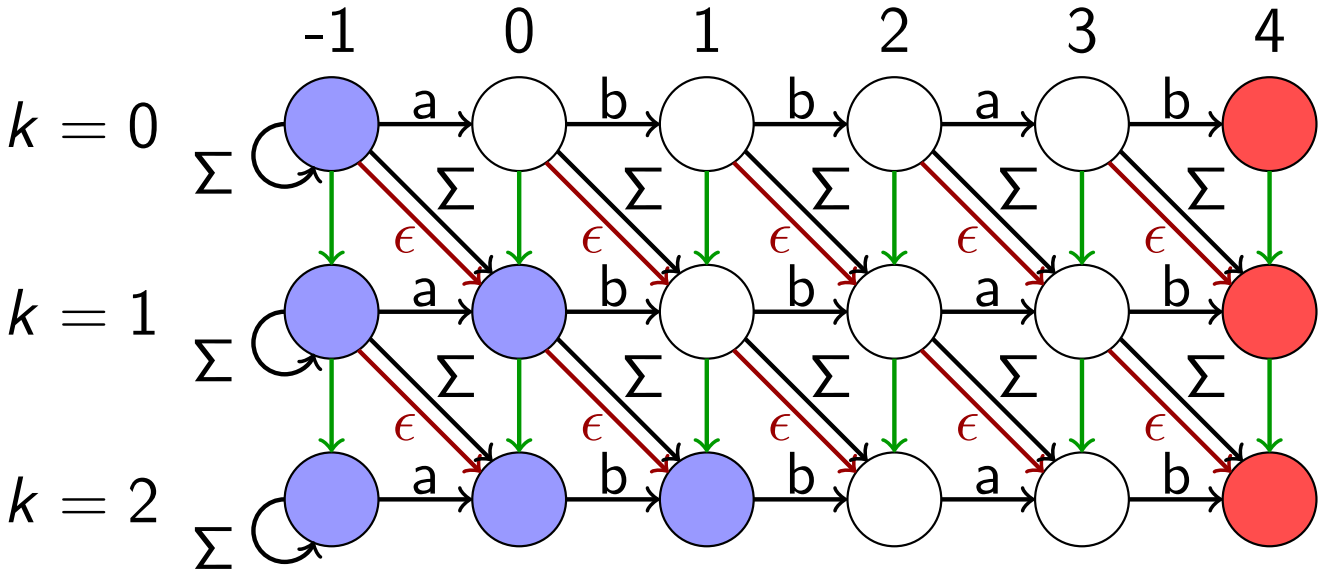
- Jede der $k+1$ Ebene entspricht einem Shift-And-Automaten.
- Zustand (i,j) und $(i+1,j)$ mit $i \leq k$ sind durch eine Σ-Kante verbunden. Dies entspricht einer Insertion.
- Zustand (i,j) und $(i+1,j+1)$ sind durch eine Σ-Kante verbunden. Dies entspricht einer Substitution.
- Zustand (i,j) und $(i+1,j+1)$ sind durch eine ϵ-Kante verbunden. Dies entspricht einer Deletion.

Konstruktion eines fehlertoleranten Shift-And-Algorithmus

- Jede der $k+1$ Ebene entspricht einem Shift-And-Automaten.
- Zustand (i,j) und $(i+1,j)$ mit $i \leq k$ sind durch eine Σ-Kante verbunden. Dies entspricht einer Insertion.
- Zustand (i,j) und $(i+1,j+1)$ sind durch eine Σ-Kante verbunden. Dies entspricht einer Substitution.
- Zustand (i,j) und $(i+1,j+1)$ sind durch eine ϵ-Kante verbunden. Dies entspricht einer Deletion.
- Startzustände sind alle (i,j) mit $0 \leq j \leq i \leq k$ (ein „Dreieck“ links unten im Automaten).

Fehlertoleranter Shift-And-Algorithmus

Beispiel für einen fehlertoleranten NFA für $P = \texttt{abbab}, k = 2$:



Grüne Kanten: Insertionen; Rote Kanten: Deletionen; Schwarze diagonale Kanten: Substitutionen.

Implementierung

- Für jede Zeile i gibt es einen Bitvektor A_i. Die Bitvektoren aus der vorherigen Iteration werden mit $A_i^{(alt)}$ bezeichnet.
- Die Erstellung der Masken bleibt unverändert.
- Es ergeben sich folgende Operationen beim Lesen eines Textzeichens c.
- $A_0 \leftarrow (A_0^{(alt)} \ll 1|1)\ \&\ mask[c]$
- $A_i \leftarrow ((A_i^{(alt)} \ll 1|\underbrace{((1 \ll (i+1)) - 1)}_{\text{Startzustände}})\ \&\ mask[c])\ |\ \underbrace{(A_{i-1}^{(alt)})}_{\text{Insertion}}\ |\ \underbrace{(A_{i-1}^{(alt)} \ll 1)}_{\text{Substitution}}\ |\ \underbrace{(A_{i-1} \ll 1)}_{\text{Deletion}}$
  für $0 < i \leq k$.

Implementierung

```
1  def shift_and(P, T, k):
2      S, m, n = set(P + T), len(P), len(T)
3      A = [0] * (k + 1)
4      ones = [(1 << (i + 1)) - 1 for i in range(k + 1)]
5      accept, mask = 1 << (m - 1), create_masks(P, S)
6
7      for j, c in enumerate(T):
8          for i in range(k, 0, -1): # ins and subst
9              A[i] = (((A[i] << 1) | ones[i]) & mask[c]) \
10                 | A[i - 1] | (A[i - 1] << 1)
11         A[0] = ((A[0] << 1) | 1) & mask[c]
12         if A[0] & accept: yield 0, j
13         for i in range(1, k + 1): # deletions
14             A[i] |= (A[i - 1] << 1)
15             if A[i] & accept:
16                 yield i, j  # error rate, end position
```

Zusammenfassung

- Der Shift-And-Algorithmus lässt sich sehr einfach auf eine fehlertolerante Variante erweitern.
- Die Laufzeit beträgt $\Theta(kn\lceil m/W \rceil)$.
- Die Methode ist weiterhin nur sinnvoll, wenn die Musterlänge nicht die Registergröße übersteigt.
- Somit ist eine worst-case Laufzeit von $\mathcal{O}(kn)$ (im Gegensatz zu $\mathcal{O}(mn)$ beim Ukkonen-Algorithmus) garantiert.

Fehlertoleranter Backward-Search-Algorithmus

- Alle bisher vorgestellten Mustersuchen haben “online” auf dem Text gesucht.
- Das passende Gegenstück zur indizierten fehlertoleranten Mustersuche liefert eine modifizierte Variante der *Backward-Search*.
- Hierdurch hängt die Laufzeit nicht mehr von der Textlänge ab.
- Die hier vorgestellte Variante ist ein Hybrid aus dem fehlertoleranten NFA und der exakten Backward-Seach.

Fehlertoleranter Backward-Search-Algorithmus

- Es wird eine $(k+1) \times (m+1)-$große Matrix M aufgestellt. Jedes Feld speichert eine Menge von *pos*-Intervallen.
- Ins Feld $M[0][0]$ wird das Intervall $(0, n-1)$ geschrieben.

Fehlertoleranter Backward-Search-Algorithmus

- Es wird eine $(k+1) \times (m+1)-$große Matrix M aufgestellt. Jedes Feld speichert eine Menge von *pos*-Intervallen.
- Ins Feld $M[0][0]$ wird das Intervall $(0, n-1)$ geschrieben.
- Pro Feld und pro Intervall werden vier Schritte durchgeführt:
  1. Match: (L, R) werden zu geupdated (L^+, R^+) und ins Feld $M[i][j+1]$ geschrieben, sofern $L <= R$ gilt.

Fehlertoleranter Backward-Search-Algorithmus

- Es wird eine $(k+1) \times (m+1)-$große Matrix M aufgestellt. Jedes Feld speichert eine Menge von *pos*-Intervallen.
- Ins Feld $M[0][0]$ wird das Intervall $(0, n-1)$ geschrieben.
- Pro Feld und pro Intervall werden vier Schritte durchgeführt:
  1. Match: (L, R) werden zu geupdated (L^+, R^+) und ins Feld $M[i][j+1]$ geschrieben, sofern $L <= R$ gilt.
  2. Delete: das alte Intervall (L, R) wird in Feld $M[i+1][j+1]$ geschrieben.

Fehlertoleranter Backward-Search-Algorithmus

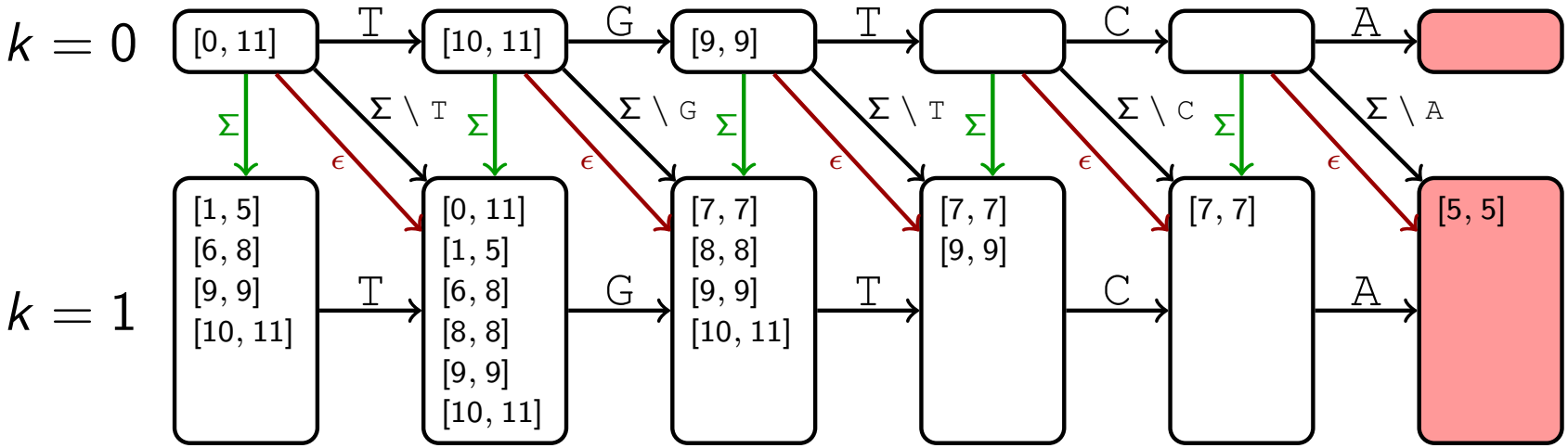
- Es wird eine $(k+1) \times (m+1)-$große Matrix M aufgestellt. Jedes Feld speichert eine Menge von *pos*-Intervallen.
- Ins Feld $M[0][0]$ wird das Intervall $(0, n-1)$ geschrieben.
- Pro Feld und pro Intervall werden vier Schritte durchgeführt:
  1. Match: (L, R) werden zu geupdated (L^+, R^+) und ins Feld $M[i][j+1]$ geschrieben, sofern $L <= R$ gilt.
  2. Delete: das alte Intervall (L, R) wird in Feld $M[i+1][j+1]$ geschrieben.
  3. Insertion: für alle $s \in \Sigma$ wird (L, R) geupdated und ins Feld $M[i+1][j]$ geschrieben.

Fehlertoleranter Backward-Search-Algorithmus

- Es wird eine $(k+1) \times (m+1)-$große Matrix M aufgestellt. Jedes Feld speichert eine Menge von *pos*-Intervallen.
- Ins Feld $M[0][0]$ wird das Intervall $(0, n-1)$ geschrieben.
- Pro Feld und pro Intervall werden vier Schritte durchgeführt:
  1. Match: (L, R) werden zu geupdated (L^+, R^+) und ins Feld $M[i][j+1]$ geschrieben, sofern $L <= R$ gilt.
  2. Delete: das alte Intervall (L, R) wird in Feld $M[i+1][j+1]$ geschrieben.
  3. Insertion: für alle $s \in \Sigma$ wird (L, R) geupdated und ins Feld $M[i+1][j]$ geschrieben.
  4. Substitution: für alle $s \in \Sigma$ wird (L, R) geupdated und ins Feld $M[i+1][j+1]$ geschrieben.

Fehlertoleranter Backward-Search-Algorithmus

Beispiel für einen Ablauf der fehlertoleranten Backward-Search für $T = \texttt{AAAACGTACCT\$}$, $P = \texttt{ACTGT}$, $\Sigma = \{\texttt{A}, \texttt{C}, \texttt{G}, \texttt{T}\}$:



Grüne Kanten: Insertionen; Rote Kanten: Deletionen; Schwarze diagonale Kanten: Substitutionen. Hier: Pattern mit einem Fehler an Position 5 im `pos`-Array gefunden.

Implementierung

```
1  def build_fm_index(T, S): # absolutly inefficiently implemented
2      less = {c: sum(1 for x in T if x < c) for c in S}
3      occ = {c: [sum(1 for x in T[:i + 1] if x == c) \
4            for i in range(len(T))] for c in S}
5      return less, occ
6
7  def update(L, R, c, less, occ):
8      L = less[c] + (occ[c][L - 1] if L > 0 else 0)
9      R = less[c] + (occ[c][R] if R >= 0 else 0) - 1
10     return L, R
11
12 def approximate_backward_search(T, P, k):
13     pos, bwt = SA_IS(T, 128), build_bwt(T)
14     m, n, S = len(P), len(T), set(P + T)
15     less, occ = build_fm_index(bwt, S)
16     M = [[set() for j in range(m + 1)] for i in range(k + 1)]
17     M[0][0].add((0, n - 1))
18     ...
```

Implementierung

```
18    ...
19    for i in range(k + 1):
20      for j, c in enumerate(P[::-1]): # flip because of BS
21        for L_old, R_old in M[i][j]:
22          L, R = update(L_old, R_old, c, less, occ)
23          if L <= R: M[i][j + 1].add((L, R))  # Match
24          if i < k:
25            M[i + 1][j + 1].add((L_old, R_old)) # Deletion
26            for s in S:
27              L, R = update(L_old, R_old, s, less, occ)
28              if L <= R:
29                M[i + 1][j].add((L, R)) # Insertion
30                # Substitution
31                if s != c: M[i + 1][j + 1].add((L, R))
32      if len(M[i][m]) > 0:
33        for v in M[i][m]:
34          for p in range(v[0], v[1] + 1): yield pos[p], i
```

Zusammenfassung

- Eine fehlertolerante Patternsuche auf einer Indexdatenstruktur ist mit der erweiterten Version einer Backward-Search möglich.
- Leider hängt die Laufzeit exponentiell von der Alphabetgröße $|\Sigma|$, der Patternlänge m und dem Fehler k ab.
- Die Laufzeit beträgt $\mathcal{O}(\sqrt{\min(k,m)}(1+\sqrt{2})^{2\min(k,m)}3^{|k-m|}|\Sigma|^k)$.
- Die worst-case Laufzeit wird so gut wie nie erreicht, da üblicherweise die Intervalle beim durchpropagieren "verhungern".