

# Algorithmen auf Sequenzen

## Fehlertolerante Mustersuche: Algorithmen

Sven Rahmann

Genominformatik  
Universitätsklinikum Essen  
Universität Duisburg-Essen  
Universitätsallianz Ruhr

- Als Fehlermaß bei der fehlertoleranten Mustersuche wird oft die Edit-Distanz  $d$  verwendet (Metrik auf  $\Sigma^*$ )

- Als Fehlermaß bei der fehlertoleranten Mustersuche wird oft die Edit-Distanz  $d$  verwendet (Metrik auf  $\Sigma^*$ )
- Gegeben: Text  $T \in \Sigma^n$ , Muster  $P \in \Sigma^m$ , Fehlerschranke  $k \in \{0, 1, 2, \dots\}$
- Gesucht:
  - 1 Intervalle  $[i..j]$ , so dass  $d(T[i..j], P) \leq k$
  - 2 Endpunkte  $j$ , so dass es  $i$  gibt mit  $d(T[i..j], P) \leq k$

- Als Fehlermaß bei der fehlertoleranten Mustersuche wird oft die Edit-Distanz  $d$  verwendet (Metrik auf  $\Sigma^*$ )
- Gegeben: Text  $T \in \Sigma^n$ , Muster  $P \in \Sigma^m$ , Fehlerschranke  $k \in \{0, 1, 2, \dots\}$
- Gesucht:
  - 1 Intervalle  $[i..j]$ , so dass  $d(T[i..j], P) \leq k$
  - 2 Endpunkte  $j$ , so dass es  $i$  gibt mit  $d(T[i..j], P) \leq k$
- Achtung:

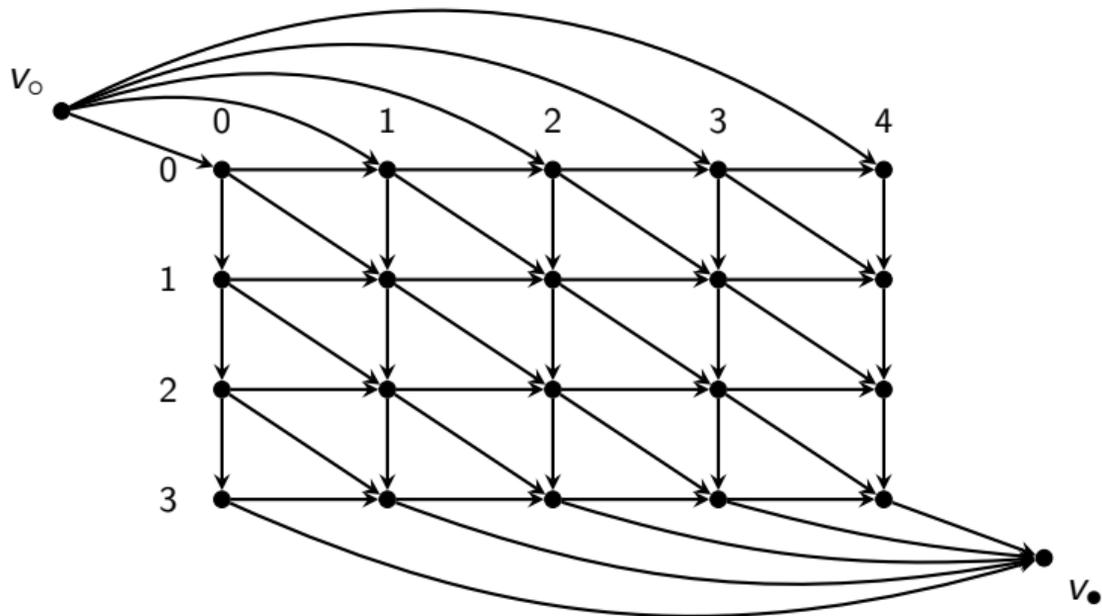
Es geht nicht um  $d(T, P)$ , sondern um  $d$  zwischen  $P$  und Teilstrings von  $T$  !

- Sei  $D[i, j]$  die minimale Edit-Distanz des Präfixes  $P[: i]$  zu einem Teilstring  $T[j' : j]$  mit  $j' \leq j$ .
- $D[i, 0] = i$  für  $0 \leq i \leq m$ ,  
 $D[0, j] = 0$  für  $0 \leq j \leq n$ , Rekurrenz ändert sich nicht!

- Sei  $D[i, j]$  die minimale Edit-Distanz des Präfixes  $P[: i]$  zu einem Teilstring  $T[j' : j]$  mit  $j' \leq j$ .
- $D[i, 0] = i$  für  $0 \leq i \leq m$ ,  
 $D[0, j] = 0$  für  $0 \leq j \leq n$ , Rekurrenz ändert sich nicht!
- Gesucht (Variante 2):  $j$  mit  $D[m, j] \leq k$ :  
 $P$  endet an Position  $j - 1$  des Textes mit höchstens  $k$  Edit-Operationen (und beginnt bei irgendeinem  $j' \leq j$ )
- Das Problem ist auch als **semiglobales Alignment** bekannt:  
Ganz  $P$  wird zu einem Teil von  $T$  aligniert.

# Edit-Graph-Algorithmus

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



# Optimierung von Ukkonen

- Esko Ukkonen (geb. 1950, Professor in Helsinki, hat auch die Linearzeit-Suffixbaum-Konstruktion erfunden) hat 1985 einen optimierten Algorithmus veröffentlicht.
- Wenn eine Fehlerschranke  $k$  gegeben ist, muss nicht die komplette DP-Matrix berechnet werden.

# Optimierung von Ukkonen

- Esko Ukkonen (geb. 1950, Professor in Helsinki, hat auch die Linearzeit-Suffixbaum-Konstruktion erfunden) hat 1985 einen optimierten Algorithmus veröffentlicht.
- Wenn eine Fehlerschranke  $k$  gegeben ist, muss nicht die komplette DP-Matrix berechnet werden.
- Sei  $last_k(j)$  die letzte Zeile in Spalte  $j$ , die einen Wert  $\leq k$  hat:

$$last_k(j) := \max\{i \mid D[i, j] \leq k\}$$

- Felder  $D[i, j]$  mit  $i > last_k(j)$  müssen nicht exakt berechnet werden:

$$D[i, j] > k \text{ für alle } i > last_k(j).$$

# Beispiel

$T = \text{AMOAMAMAOM}$ ,  $P = \text{MAOAM}$ ,  $k = 1$ .

Blaue Linie zeigt Verlauf von  $last_1(j)$ .

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

# Beispiel

$T = \text{AMOAMAMAOM}$ ,  $P = \text{MAOAM}$ ,  $k = 1$ .

Blaue Linie zeigt Verlauf von  $\text{last}_1(j)$ .

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

- Woher weiß man, dass unter  $\text{last}_k(j)$  nur höhere Werte als  $k$  stehen, ohne die ganze Spalte zu berechnen, d.h. wie kann man  $\text{last}_k$  effizient aktualisieren?

# Beispiel

$T = \text{AMOAMAMAOM}$ ,  $P = \text{MAOAM}$ ,  $k = 1$ .

Blaue Linie zeigt Verlauf von  $\text{last}_1(j)$ .

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

- Woher weiß man, dass unter  $\text{last}_k(j)$  nur höhere Werte als  $k$  stehen, ohne die ganze Spalte zu berechnen, d.h. wie kann man  $\text{last}_k$  effizient aktualisieren?
- Es gilt  $\text{last}_k(j+1) \leq \text{last}_k(j) + 1$ .

## Lemma

*Unter Einheitskosten (Edit-Distanz) gilt*

$$D[i - 1, j - 1] \leq D[i, j] \leq D[i - 1, j - 1] + 1.$$

Inbesondere kann nicht  $D[i, j] < D[i - 1, j - 1]$  sein.

## Lemma

*Unter Einheitskosten (Edit-Distanz) gilt*

$$D[i - 1, j - 1] \leq D[i, j] \leq D[i - 1, j - 1] + 1.$$

Insebesondere kann nicht  $D[i, j] < D[i - 1, j - 1]$  sein.

## Beweis:

**1**  $D[i, j] \leq D[i - 1, j - 1] + 1$  gilt wegen der Rekurrenz.

## Lemma

*Unter Einheitskosten (Edit-Distanz) gilt*

$$D[i-1, j-1] \leq D[i, j] \leq D[i-1, j-1] + 1.$$

Insebesondere kann nicht  $D[i, j] < D[i-1, j-1]$  sein.

## Beweis:

- 1**  $D[i, j] \leq D[i-1, j-1] + 1$  gilt wegen der Rekurrenz.
- 2** Betrachte optimales Alignment von  $P[: i]$  und  $T[j' : j]$  mit Kosten  $D[i, j]$ . Durch Streichen von Alignmentsspalten, bis die jeweils letzten Zeichen  $P[i-1]$  und  $T[j-1]$  verschwunden sind, werden die Kosten nicht größer. Also gilt  $D[i-1, j-1] \leq D[i, j]$ .

- $last_k(0) = \min(k, m)$
- $last_k(j + 1) \leq last_k(j) + 1$
- Berechne Spalte  $j + 1$  bis zur Obergrenze, dann dekrementiere  $last_k(j)$  an Hand der berechneten Werte
- Ist  $D[m, j] \leq k$  (damit auch  $last_k(j) = m$ , wurde  $P$  mit  $D[m, j]$  Fehlern gefunden.
- Erwartete Laufzeit (ohne Beweis):  $\mathcal{O}(kn + zm)$  statt  $\mathcal{O}(mn)$  mit  $z$ : Anzahl Treffer

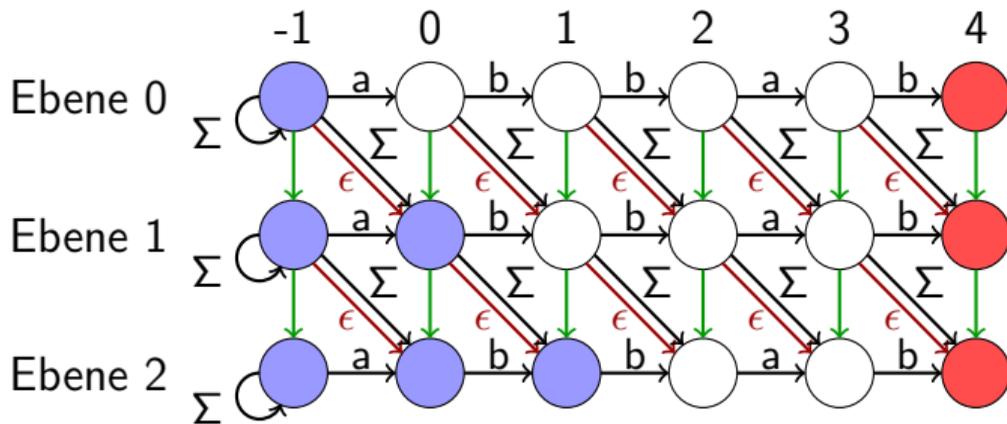
```
1 def ukkonen(P, T, k):
2     m, n = len(P), len(T)
3     Dj = list(range(m + 1)) # Spalte j (j=0)
4     Dp = [k + 1] * (m + 1) # vorige Spalte
5     lastk = min(k, m)
6     for j, tj in zip(count(1), T):
7         Dj, Dp, lastk = Dp, Dj, min(lastk+1, m)
8         # Dj[0] = 0 # gilt bereits
9   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

# Fehlertoleranter Shift-And-Algorithmus

- Als Alternative zum Algorithmus von Ukkonen betrachten wir eine Variante des Shift-And-Algorithmus zur fehlertoleranten Mustersuche mit  $k$  Fehlern.
- Hierbei sind  $k + 1$  Kopien des NFAs miteinander verbunden; jeder Kopie steht für eine Fehlerzahl  $0, 1, \dots, k$ .

# Fehlertoleranter Shift-And-Algorithmus

- Als Alternative zum Algorithmus von Ukkonen betrachten wir eine Variante des Shift-And-Algorithmus zur fehlertoleranten Mustersuche mit  $k$  Fehlern.
- Hierbei sind  $k + 1$  Kopien des NFAs miteinander verbunden; jeder Kopie steht für eine Fehlerzahl  $0, 1, \dots, k$ .
- Beispiel:  $P = \text{abbab}$  und  $k = 2$



# Formale Konstruktion

- Zustandsraum  $Q := \{0, \dots, k\} \times \{-1, 0, \dots, |P| - 1\}$
- Jede der  $k + 1$  Ebenen entspricht einem Shift-And-Automaten mit schwarzen horizontalen Kanten, die  $P$  buchstabieren.

# Formale Konstruktion

- Zustandsraum  $Q := \{0, \dots, k\} \times \{-1, 0, \dots, |P| - 1\}$
- Jede der  $k + 1$  Ebenen entspricht einem Shift-And-Automaten mit schwarzen horizontalen Kanten, die  $P$  buchstabieren.
- Zustand  $(i, j)$  und  $(i + 1, j)$  mit  $0 \leq i < k$  sind durch eine grüne vertikale  $\Sigma$ -Kante verbunden (Insertion)

- Zustandsraum  $Q := \{0, \dots, k\} \times \{-1, 0, \dots, |P| - 1\}$
- Jede der  $k + 1$  Ebenen entspricht einem Shift-And-Automaten mit schwarzen horizontalen Kanten, die  $P$  buchstabieren.
- Zustand  $(i, j)$  und  $(i + 1, j)$  mit  $0 \leq i < k$  sind durch eine grüne vertikale  $\Sigma$ -Kante verbunden (Insertion)
- Zustand  $(i, j)$  und  $(i + 1, j + 1)$  sind durch eine schwarze diagonale  $\Sigma$ -Kante verbunden (Substitution)

- Zustandsraum  $Q := \{0, \dots, k\} \times \{-1, 0, \dots, |P| - 1\}$
- Jede der  $k + 1$  Ebenen entspricht einem Shift-And-Automaten mit schwarzen horizontalen Kanten, die  $P$  buchstabieren.
- Zustand  $(i, j)$  und  $(i + 1, j)$  mit  $0 \leq i < k$  sind durch eine grüne vertikale  $\Sigma$ -Kante verbunden (Insertion)
- Zustand  $(i, j)$  und  $(i + 1, j + 1)$  sind durch eine schwarze diagonale  $\Sigma$ -Kante verbunden (Substitution)
- Zustand  $(i, j)$  und  $(i + 1, j + 1)$  sind zusätzlich durch eine rote diagonale  $\epsilon$ -Kante verbunden (Deletion)

# Formale Konstruktion

- Zustandsraum  $Q := \{0, \dots, k\} \times \{-1, 0, \dots, |P| - 1\}$
- Jede der  $k + 1$  Ebenen entspricht einem Shift-And-Automaten mit schwarzen horizontalen Kanten, die  $P$  buchstabieren.
- Zustand  $(i, j)$  und  $(i + 1, j)$  mit  $0 \leq i < k$  sind durch eine grüne vertikale  $\Sigma$ -Kante verbunden (Insertion)
- Zustand  $(i, j)$  und  $(i + 1, j + 1)$  sind durch eine schwarze diagonale  $\Sigma$ -Kante verbunden (Substitution)
- Zustand  $(i, j)$  und  $(i + 1, j + 1)$  sind zusätzlich durch eine rote diagonale  $\epsilon$ -Kante verbunden (Deletion)
- Startzustände sind alle  $(i, j)$  mit  $-1 \leq j \leq i \leq k$  („Dreieck“ links unten im Automaten).

- 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.

# 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  $c$ .
- $A_0 \leftarrow (A_0^{(alt)} \ll 1 | 1) \& \text{mask}[c]$
- Für  $1 \leq i \leq k$  mit  $S_i := (1 \ll (i + 1)) - 1$  (Startzustände in Zeile  $i$ ):

$$A_i \leftarrow ((A_i^{(alt)} \ll 1) \& \text{mask}[c]) | S_i | \underbrace{(A_{i-1}^{(alt)})}_{\text{Insertion}} | \underbrace{(A_{i-1}^{(alt)} \ll 1)}_{\text{Substitution}} | \underbrace{(A_{i-1} \ll 1)}_{\text{Deletion}}$$

# 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  $c$ .
- $A_0 \leftarrow (A_0^{(alt)} \ll 1 | 1) \& \text{mask}[c]$
- Für  $1 \leq i \leq k$  mit  $S_i := (1 \ll (i + 1)) - 1$  (Startzustände in Zeile  $i$ ):

$$A_i \leftarrow ((A_i^{(alt)} \ll 1) \& \text{mask}[c]) | S_i | \underbrace{(A_{i-1}^{(alt)})}_{\text{Insertion}} | \underbrace{(A_{i-1}^{(alt)} \ll 1)}_{\text{Substitution}} | \underbrace{(A_{i-1} \ll 1)}_{\text{Deletion}}$$

- Laufzeit  $\mathcal{O}(knm/W)$  bei Registerbreite  $W$ . Sinnvoll, wenn  $m < W$ .

- Die bisherigen Algorithmen zur fehlertoleranten Mustersuche erfordern keine Indizierung.
- Jetzt: Anpassen des **Backward-Search-Algorithmus** (mit FM-Index)
- Laufzeit hängt nicht mehr von der Textlänge ab.
- Die hier vorgestellte Variante ist ein Hybrid aus dem fehlertoleranten NFA und der exakten Backward-Search.

- Es wird eine  $(k + 1) \times (m + 1)$  Matrix  $M = (M[0..k, 0..m])$  aufgestellt.
- $M[i, j]$  speichert die Menge von Intervallgrenzen  $[L, R]$ , so dass das Länge- $j$ -Suffix von  $P$  mit höchstens  $i$  Fehlern an  $pos[r]$  für jedes  $r \in [L..R]$  beginnt.
- Es ist  $M[0, 0] = \{[0..n - 1]\}$

- Es wird eine  $(k + 1) \times (m + 1)$  Matrix  $M = (M[0..k, 0..m])$  aufgestellt.
- $M[i, j]$  speichert die Menge von Intervallgrenzen  $[L, R]$ , so dass das Länge- $j$ -Suffix von  $P$  mit höchstens  $i$  Fehlern an  $pos[r]$  für jedes  $r \in [L..R]$  beginnt.
- Es ist  $M[0, 0] = \{[0..n - 1]\}$
- Pro Feld  $M[i, j]$  und pro Intervall  $[L..R]$  darin werden vier Schritte durchgeführt:
  - 1 Übereinstimmungen:  $[L..R]$  wird mit dem entsprechenden Buchstaben von  $P$  aktualisiert zu  $[L^+..R^+]$  (Backward Search) und in  $M[i][j + 1]$  eingetragen (sofern nicht leer, d.h.  $L^+ \leq R^+$ ).

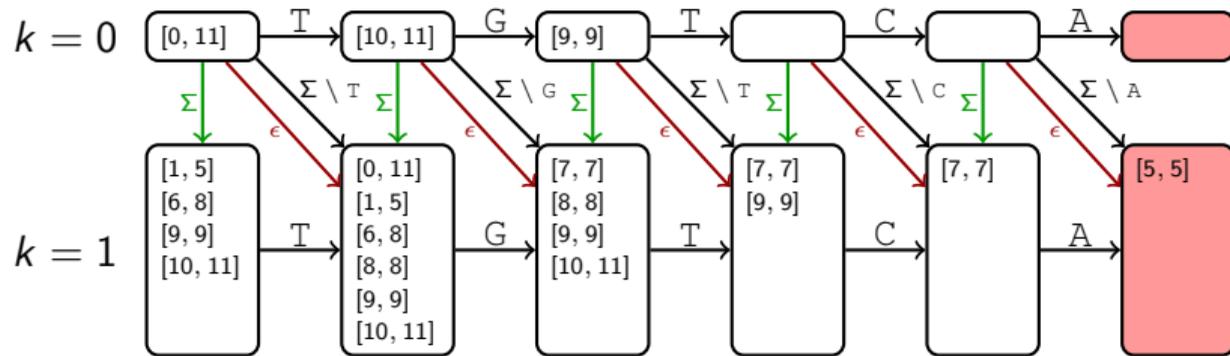
- Es wird eine  $(k + 1) \times (m + 1)$  Matrix  $M = (M[0..k, 0..m])$  aufgestellt.
- $M[i, j]$  speichert die Menge von Intervallgrenzen  $[L, R]$ , so dass das Länge- $j$ -Suffix von  $P$  mit höchstens  $i$  Fehlern an  $pos[r]$  für jedes  $r \in [L..R]$  beginnt.
- Es ist  $M[0, 0] = \{[0..n - 1]\}$
- Pro Feld  $M[i, j]$  und pro Intervall  $[L..R]$  darin werden vier Schritte durchgeführt:
  - 1 Übereinstimmungen:  $[L..R]$  wird mit dem entsprechenden Buchstaben von  $P$  aktualisiert zu  $[L^+..R^+]$  (Backward Search) und in  $M[i][j + 1]$  eingetragen (sofern nicht leer, d.h.  $L^+ \leq R^+$ ).
  - 2 Löschungen: das alte Intervall  $[L..R]$  wird in  $M[i + 1][j + 1]$  kopiert

- Es wird eine  $(k + 1) \times (m + 1)$  Matrix  $M = (M[0..k, 0..m])$  aufgestellt.
- $M[i, j]$  speichert die Menge von Intervallgrenzen  $[L, R]$ , so dass das Länge- $j$ -Suffix von  $P$  mit höchstens  $i$  Fehlern an  $pos[r]$  für jedes  $r \in [L..R]$  beginnt.
- Es ist  $M[0, 0] = \{[0..n - 1]\}$
- Pro Feld  $M[i, j]$  und pro Intervall  $[L..R]$  darin werden vier Schritte durchgeführt:
  - 1 Übereinstimmungen:  $[L..R]$  wird mit dem entsprechenden Buchstaben von  $P$  aktualisiert zu  $[L^+..R^+]$  (Backward Search) und in  $M[i][j + 1]$  eingetragen (sofern nicht leer, d.h.  $L^+ \leq R^+$ ).
  - 2 Löschungen: das alte Intervall  $[L..R]$  wird in  $M[i + 1][j + 1]$  kopiert
  - 3 Einfügungen: für alle  $s \in \Sigma$  wird  $[L..R]$  aktualisiert und das Ergebnis  $[L_s^+..R_s^+]$  (wenn nicht leer) in  $M[i + 1][j]$  eingetragen

- Es wird eine  $(k + 1) \times (m + 1)$  Matrix  $M = (M[0..k, 0..m])$  aufgestellt.
- $M[i, j]$  speichert die Menge von Intervallgrenzen  $[L, R]$ , so dass das Länge- $j$ -Suffix von  $P$  mit höchstens  $i$  Fehlern an  $pos[r]$  für jedes  $r \in [L..R]$  beginnt.
- Es ist  $M[0, 0] = \{[0..n - 1]\}$
- Pro Feld  $M[i, j]$  und pro Intervall  $[L..R]$  darin werden vier Schritte durchgeführt:
  - 1 Übereinstimmungen:  $[L..R]$  wird mit dem entsprechenden Buchstaben von  $P$  aktualisiert zu  $[L^+..R^+]$  (Backward Search) und in  $M[i][j + 1]$  eingetragen (sofern nicht leer, d.h.  $L^+ \leq R^+$ ).
  - 2 Löschungen: das alte Intervall  $[L..R]$  wird in  $M[i + 1][j + 1]$  kopiert
  - 3 Einfügungen: für alle  $s \in \Sigma$  wird  $[L..R]$  aktualisiert und das Ergebnis  $[L_s^+..R_s^+]$  (wenn nicht leer) in  $M[i + 1][j]$  eingetragen
  - 4 Ersetzungen:  $[L_s^+..R_s^+]$  wird zusätzlich in  $M[i + 1][j + 1]$  eingetragen

# Fehlertoleranter Backward-Search-Algorithmus

Beispiel für  $T = \text{AAAACGTACCT}\$, P = \text{ACTGT}, \Sigma = \{A, C, G, T\}$ :



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

- Ukkonen's Algorithmus:  
 $\mathcal{O}(nk)$  statt  $\mathcal{O}(nm)$  erwartete Laufzeit
- Fehlertoleranter NFA ( $k + 1$  Ebenen für  $k$  Fehler):  
 $\mathcal{O}(nkm/W)$  Laufzeit
- Fehlertoleranter Backward Search Algorithmus,  
Hybrid aus Backward Search und NFA.  
Laufzeit hängt exponentiell von  $|\Sigma|$ ,  $m$  und  $k$  ab:  
 $\mathcal{O}(\sqrt{\min(k, m)}(1 + \sqrt{2})^{2 \min(k, m)} 3^{|k-m|} |\Sigma|^k)$  (in der Praxis besser)