Informatik II - SS 2016

(Algorithmen & Datenstrukturen)

Vorlesung 21 (15.7.2016)

String Matching (Textsuche)

Approximate String Matching

Fabian Kuhn

Algorithmen und Komplexität

Textsuche / String Matching

Gegeben:

- Zwei Zeichenketten (Strings)
- Text T (typischerweise lang)
- Muster P (engl. pattern, typischerweise kurz)

Ziel:

- Finde alle Vorkommen von P in T

Notation:

- Länge Text T : $\boldsymbol{n}$,   Länge Muster P : $\boldsymbol{m}$

Was bisher geschah…

- Naiver Algorithmus
  - Gehe den Text von links nach rechts durch
  - Laufzeit O(nm)
- Rabin Karp Algorithmus
  - Hashe Muster und Fenster, so dass man in O(1) vergleichen kann
    (bei gleichem Hashwert → ziffernweise vergleichen)
  - Beim Verschieben des Fensters wird der neue Hashwert in Zeit O(1) berechnet
  - Laufzeit bei optimaler Hashfunktion O(n+km)

Knuth-Morris-Pratt Alg.: Kein Beispiel

Pattern: ABCAD

Text:

A	**B**	**C**	**A**	**B**	**C**	**A**	**D**	**E**	**E**	**E**	**E**
A	B	C	A	~~A~~							
				A	B	C	A	D			
			A	B	C	A	D				

3 geht nicht, da man ggf. das Muster verpasst

soweit dürfen wir schieben

Knuth-Morris-Pratt Alg.: Initialisierung

- Wir merken uns an jeder Stelle des Musters, wie weit wir das Suchfenster sicher bei einem "Mismatch" weiterschieben können.
- Äquivalent dazu: Stelle im Muster an der wir weiter suchen müssen
- Dies soll unabhängig vom Text sein (wichtig)

Vorberechnung: Array S der Länge $m+1$

- $S[i]$: Stelle in P, an welcher man die neue Suche beginnt, falls beim Testen der Stelle i im Pattern ein Mismatch auftritt
- $S[m]$: Stelle in P, an welcher man weitersucht, nachdem P erfolgreich gefunden wurde
- $S[0] = -1, \quad S[1] = 0$

Initialisierung

Vorberechnung von Array $\boldsymbol{S}$:

- $P = [\,\text{A, B, D, A, B, L, A, B, D, A, B, D}\,]$

  $S = [-1, 0, 0, 0, 1, 2, 0, 1, 2, \boxed{3}, 4, 5, 3]$

- An Position in $S[i]$ (für $i \in \{2, \dots, m\}$) steht

$$S[i] \coloneqq \underset{k<i}{\arg\max}\{\underbrace{P[i-k \dots i-1]}_{\text{Stück, welches vor der aktuellen Position endet}} = \overbrace{P[0 \dots k-1]}^{\text{Start des Musters}}\}$$

- $S[i]$: Länge des längsten **echten** Teilstückes von $P[0 \dots i-1]$, welches an Stelle $i-1$ endet, und welches auch Anfangsstück von P ist

- In gewissen Fällen könnte man sicher weiter schieben

  (aber obiges ist ausreichend für Laufzeit O(n))

Initialisierung

Vorberechnung von Array S:

- $S[i]$: Länge des längsten *passenden* Teilstückes von $P[1 \ldots i-1]$,
- $P = [$ `A, B, D, A, B, L, A, B, D, A, B, D` $]$

Nein

Ja → 3

- $S = [$ `-1, 0, 0, 0, 1, 2, 0, 1, 2, 3, 4, 5, 3` $]$

Berechnung von $S[i]$: (von links nach rechts)

- S[0]:=-1, S[1]:=0
- Für i>1 suchen wir Teilstücke, die wir verlängern können
  - Erster Kandidat: S[i-1] = 3
    Falls $P[S[i-1]] = P[i-1]$, dann ist $S[i] = S[i-1] + 1$
  - Zweiter Kandidat: S[S[i-1]]
    Falls $P[S[S[i-1]]] = P[i-1]$, dann ist $S[i] = S[S[i-1]] + 1$
  - Usw…

Berechnung von $S[i]$: Beispiel

$h \coloneqq S[i-1]$ // erste Kandidat zur Verlängerung

while $h \geq 0$ do

  if $P[i-1] = P[h]$ then

    $S[i] \coloneqq h+1$; $h \coloneqq -1$ } erfolgreiche Verlängerung

  else

    $h \coloneqq S[h]$ // nächste Kandidat zur Verlängerung

if $h = -1$ then $S[i] = 0$

h = # Zeilen des Kandidaten, den wir verlängern wollen.

Berechnung von $S[i]$: Laufzeit

$h \coloneqq S[i-1]$

`while` $h \geq 0$ `do`

    `if` $P[i-1] = P[h]$ `then`

        $S[i] \coloneqq h + 1$; $h \coloneqq -1$

    `else`

        $h \coloneqq S[h]$

`if` $h = -1$ `then` $S[i] = 0$

Beobachtung:

$$S[i] \leq S[i-1] + 1$$

Falls $S[i] = S[i-1] + 1$: 1 Schleifendurchlauf

Falls $S[i] < S[i-1]$:

- Wert von h nimmt in jedem Schleifendurchlauf ab
- Am Schluss ist $S[i] = h + 1$
- #Schleifendurchläufe $\leq \Delta h + 1 \leq \overbrace{S[i-1]}^{\text{Anfangs } h} - \overbrace{(S[i]-1)}^{h \text{ am Ende}} + 1$

$$\leq S[i-1] - (S[i]) + 2$$

Berechnung von $S[i]$: Laufzeit

$S[i] = S[i-1]+1$

Falls $\boldsymbol{S[i] = S[i-1]+1}$:

- Anzahl Schleifendurchläufe $= \underline{1} = S[i-1] - S[i] + 2$

Falls $\boldsymbol{S[i] < S[i-1]}$:

- Anzahl Schleifendurchläufe $\leq \Delta h + 1 = S[i-1] - S[i] + 2$

Gesamtlaufzeit:

$$\sum_{i=2}^{m} \left(S[i-1] - S[i] + 2\right) = 2\cdot(m-1) + \Big(S[1] - \cancel{S[2]} + \cancel{S[2]} - S[3] \ldots + S[m-1] - S[m]\Big)$$

$$= 2\cdot(m-1) + \underbrace{S[1]}_{=0} - \underbrace{S[m]}_{\geq 0} \leq 2(m-1)$$

$$\leadsto O(m)$$

Knuth-Morris-Pratt Alg.: Zusammenfassung

Knuth-Morris-Pratt Algorithmus:

- Berechnet zuerst in Zeit $O(m)$ das Array S der Länge m
  - hängt nur vom Pattern P ab und nicht vom Text T
  - beschreibt an jeder Position im Pattern, wo (im Pattern) man bei einem Mismatch sicher weitersuchen kann
- Mit Hilfe von S werden dann alle Vorkommen von P in T in Zeit $O(n)$ gefunden
  - In jedem Schritt kann man entweder die aktuelle Suchposition in T oder die Position des Suchfensters in T um mindestens 1 nach rechts verschieben

Gesamtlaufzeit: $\boldsymbol{O(m+n)=O(n)}$

Recap - Editierdistanz

Gegeben: Zwei Strings

$A = a_1 a_2 \dots a_m$ and $B = b_1 b_2 \dots b_n$

Ziel: Bestimme die minimale Anzahl ED(A,B) der Editieroperationen, die benötigt werden um A in B zu transformieren

Editieroperationen:

a) **Replace** a character from string A by a character from B

b) **Delete** a character from string A

c) **Insert** a character from string B into A

		insert	keine Änderung			Replace							delete(i)		
m	a	-	t	h	e	m	-	-	a	t	i	c	i	a	n
m	u	l	t	i	p	l	i	c	a	t	i	o	-	-	n

Editierdistanz - Allgemeines Kostenmodell

- Verschiedene Kosten für verschiedene Buchstaben:
  - Kosten für **replace(a,b)** = $\boldsymbol{c(a,b) \geq 0}$
  - Kosten für **delete(a)**= $\boldsymbol{c(a,\varepsilon)}$
  - Kosten für **insert(b)**= $\boldsymbol{c(\varepsilon,b)}$

  (kann c(a,b) darf anders sein als c(a',b'))

  ε=leere Wort

- **Wichtig dabei: Dreiecksungleichung**:

$$c(a,c) \leq c(a,b) + c(b,c)$$

(damit jeder Buchstabe maximal einmal geändert wird)

- **Unit cost model**: $c(a,b) = \begin{cases} 1, & \text{if } a \neq b \\ 0, & \text{if } a = b \end{cases}$

Editierdistanz – Rekursion

- $D_{k,l} = ED(a_1 a_2 \dots a_k, b_1 \mathrm{b}_2 \dots \mathrm{b_l})$

nach der letzten Operation unterscheiden

- **Rekursiongleichung** (für $k, \ell \geq 1$)

$$D_{k,\ell} = \min \begin{Bmatrix} D_{k-1,\ell-1} + c(a_k, b_\ell) \\ D_{k-1,\ell} \quad + c(a_k, \varepsilon) \\ D_{k,\ell-1} \quad + c(\varepsilon, b_\ell) \end{Bmatrix} = \underbrace{\min \begin{cases} D_{k-1,\ell-1} + 1/0 \\ D_{k-1,\ell} + 1 \\ D_{k,\ell-1} + 1 \end{cases}}_{=0 \text{ unit cost modell}}$$

- **Basisfälle**

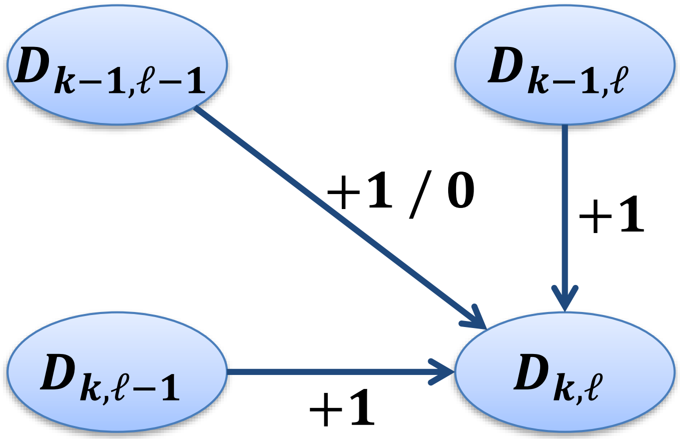
$$D_{0,0} = D(\varepsilon, \varepsilon) = 0$$

$$D_{0,j} = D(\varepsilon, B_j) = D_{0,j-1} + c(\varepsilon, b_j) = j$$

$$D_{i,0} = D(A_i, \varepsilon) = D_{i-1,0} + c(a_i, \varepsilon) = i$$

Editierdistanz – Dynamisches Programm

- Berechne $D_{i,j}$ für alle $0 \leq i \leq k$, $0 \leq j \leq \ell$:



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

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

Editierdistanz - Zusammenfassung

- Editierdistanz zwischen zwei Strings der Länge m und n kann in Zeit $O(mn)$ berechnet werden.
- Durch Speichern der *Vorgänger* kann man auch die zugehörigen Editieroperationen erhalten
- Erweiterung auf komplexere Kostenmodelle ist ohne Probleme möglich

Approximate String Matching

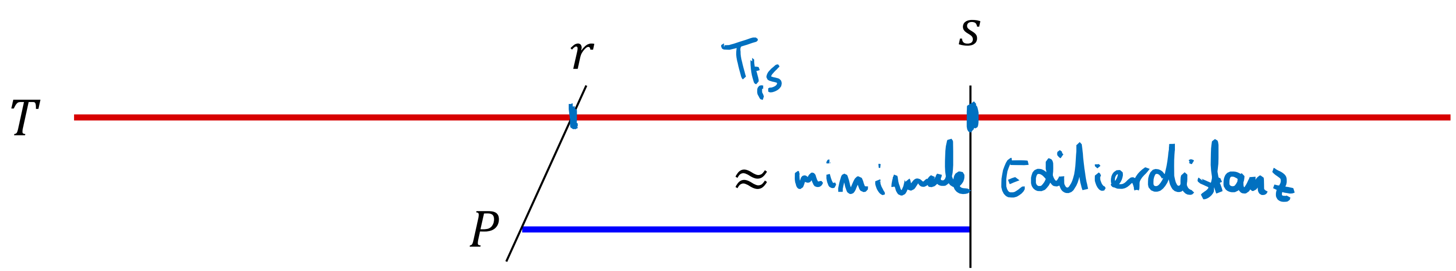
Gegeben: Strings

$T = t_1 t_2 \ldots t_n$ (Text) und

$P = p_1 p_2 \ldots p_m$ (Muster).

Ziel: Finde ein Intervall $[r, s]$, $1 \leq r \leq s \leq n$ so dass der Teilstring $T_{r,s} := t_r \ldots t_s$ die größte Ähnlichkeit zum Muster P hat:

$$\arg\min_{1 \leq r \leq s \leq n} D(T_{r,s}, P)$$



Approximate String Matching

Naive Lösung:

for all $1 \leq r \leq s \leq n$ **do**

compute $ED(T_{r,s}, P)$ $\mathcal{O}((r-s+1) \cdot m) = \mathcal{O}(n \cdot m)$

choose the minimum

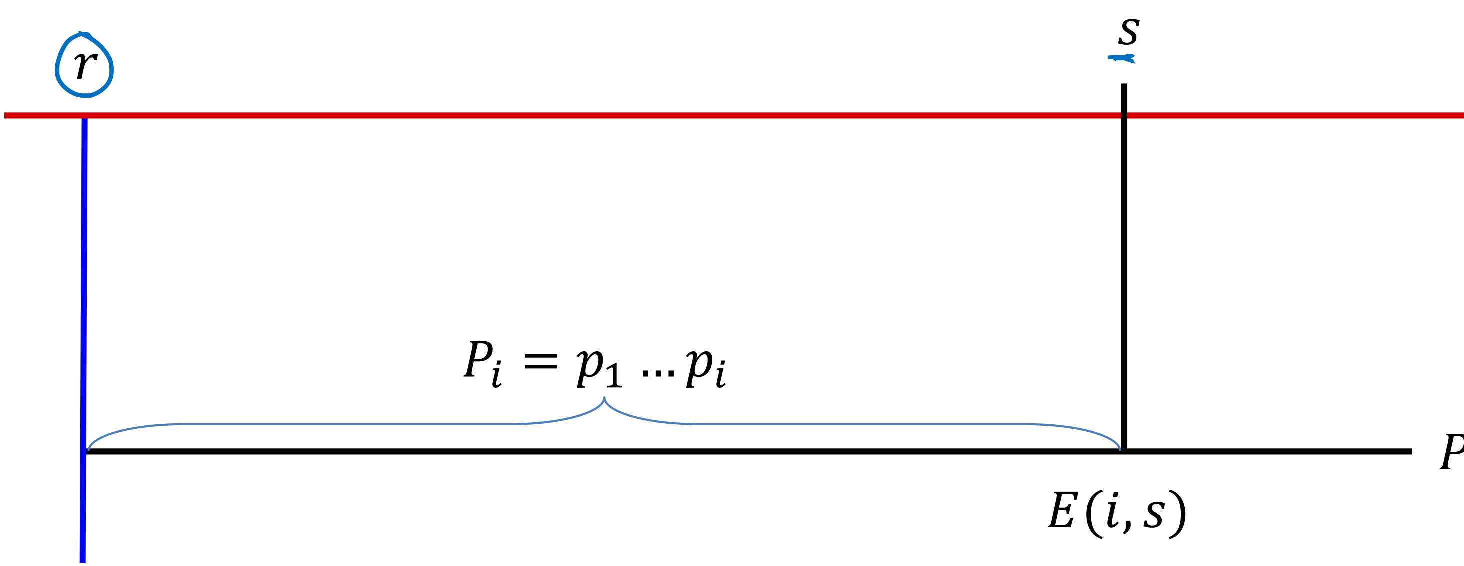
Gesamtlaufzeit ist hier

$\mathcal{O}(n^2 \, n \cdot m) = \mathcal{O}(n^3 \cdot m)$

Ein verwandtes Problem:

- Berechne für jede Position s im Text und i im Muster die kleinste Editierdistanz $E(i, s)$ zwischen $\boxed{P_i = p_1 \dots p_i}$ und einem *beliebigen Teilstring* $\boxed{T_{r,s}}$ von T, der in Position s endet / oder das ~~leere~~ Wort ist.

ist unbekannt

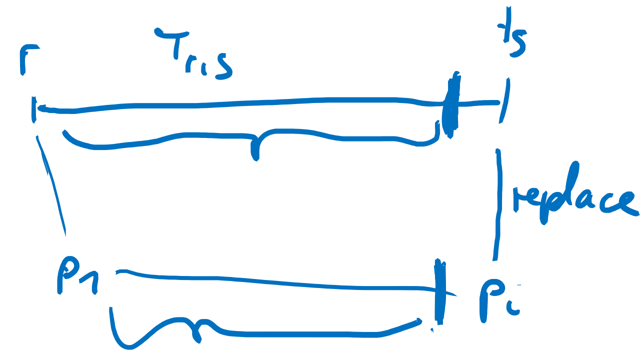


Approximate String Matching

Es gibt drei Möglichkeiten ein *Alignment von* T_s and P_i zu beenden:

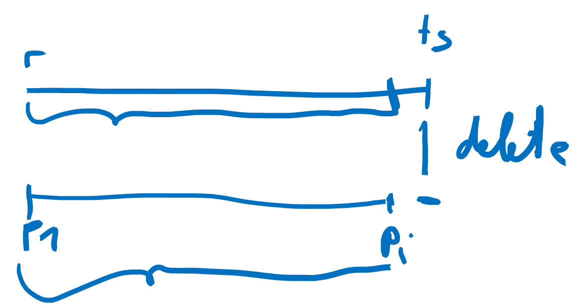
1. t_s is replaced by p_i:

$$E_{s,i} = E_{s-1,i-1} + c(t_s, p_i)$$



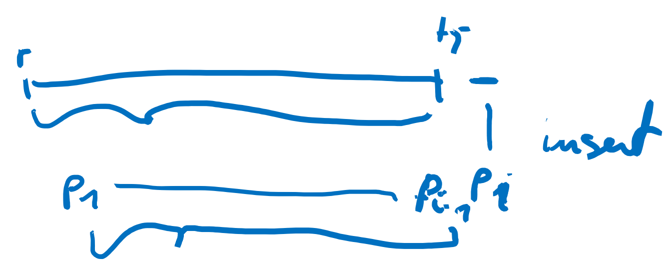
2. t_s is deleted:

$$E_{s,i} = E_{b-1,i} + c(t_s, \varepsilon)$$



3. p_i is inserted:

$$E_{s,i} = E_{s,i-1} + c(\varepsilon, p_i\ )$$



Approximate String Matching

$E_{i,j} =$ Editierdistanz eines optimalen Matchings von P_i welches an Position j im Text endet.

Basisfälle:

$$E_{0,0} = 0$$

$E_{j,0} = j$ $\cancel{E_{0,j} = j}$ leere Wort approximativ auf das leere Wort welche

Wort der Länge j approximativ auf das leere Wort match

$E_{0,i} = 0$ $E_{i,0} = 0$ leere Wort auf ein Wort der Länge i matchen ~~tex~~ wollen

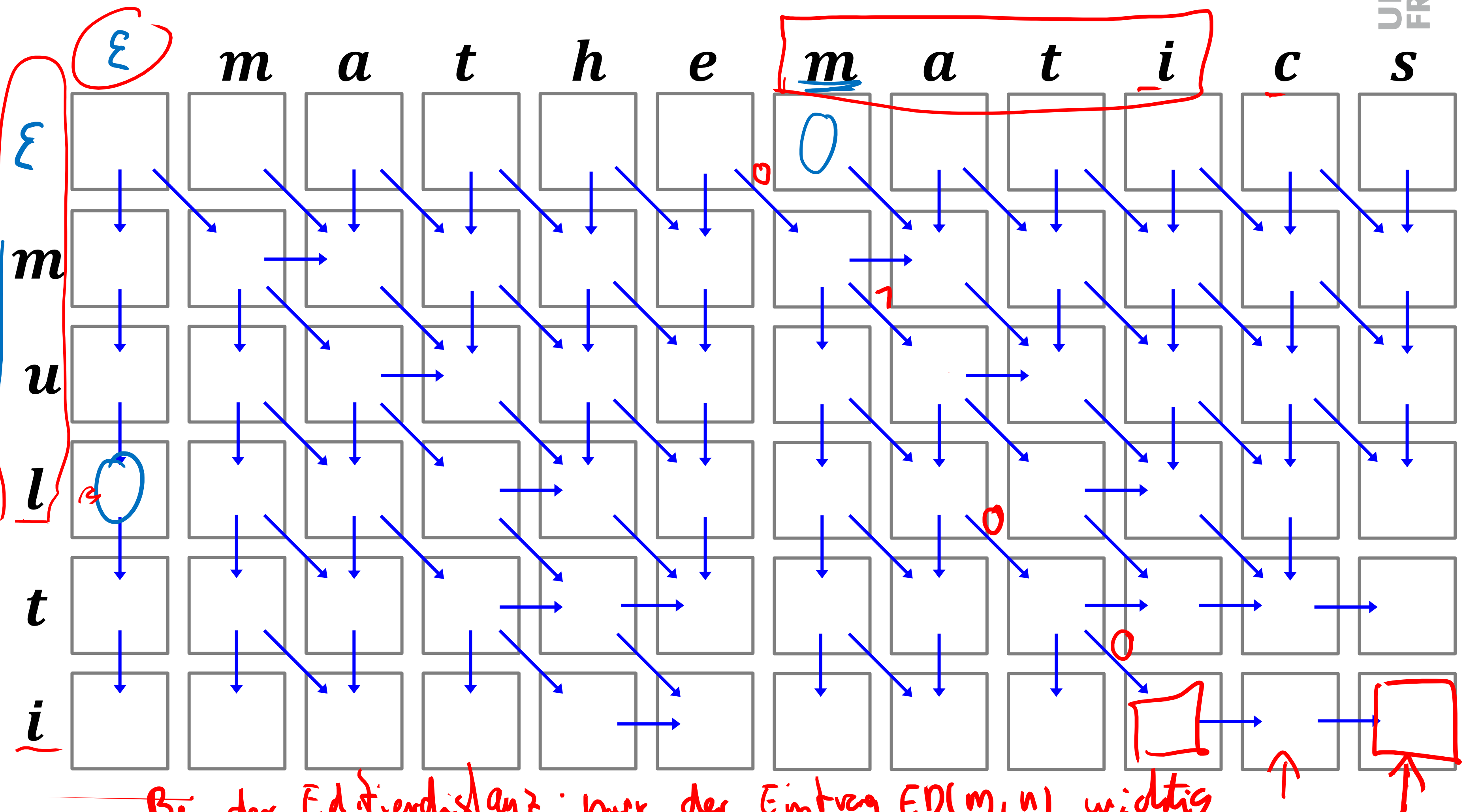
Rekursionsgleichung (unit cost model):

$$E_{s,i} = \min \begin{Bmatrix} E_{s-1,i-1} + 1/0 \\ E_{s-1,i} \quad + 1 \\ E_{s,i-1} \quad + 1 \end{Bmatrix}$$

(identisch zur Editierdistanz)

$\mathcal{E}_{3,0} = 3$

m a – t i
m u l t i



Bei der Editierdistanz: nur der Eintrag ED(m,n) wichtig

App. String Matching: Suche Minimum der letzten Zeile suchen

Approximate String Matching

- Algorithmus um $E(m, n)$ zu berechnen ist bis auf die Initialisierung von $E(i, 0)$ identisch zum Editierdistanzalgorithmus
- Bei der Editierdistanz: Nur eintrag $E(m, n)$ zählt
- Approximate String Matchting: Minimum der letzten Zeile zählt.
- Approximate String Matching ist in O(nm) Zeit lösbar