Non-Standard-Datenbanken

Zeichenkettenabgleich

Prof. Dr. Ralf Möller
Universität zu Lübeck
Institut für Informationssysteme

Danksagung

- Das folgende Präsentationsmaterial wurde von Sven Groppe für das Modul Algorithmen und Datenstrukturen erstellt und mit Änderungen hier übernommen
(z.B. werden Algorithmen im Pseudocode präsentiert)

Motivation Zeichenkettenabgleich

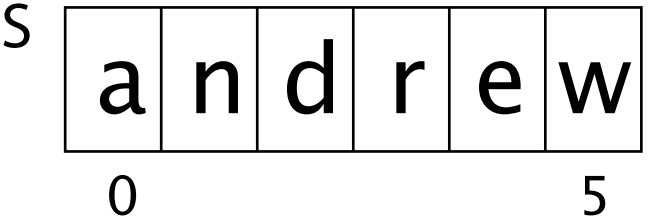
- Gegeben eine Folge von Zeichen (Text), in der eine Zeichenkette (Muster) gefunden werden soll
- Varianten
  - Alle Vorkommen des Musters im Text
  - Ein beliebiges Vorkommen im Text
  - Erstes Vorkommen im Text
- Anwendungen
  - Nachverarbeitung bei Bigramm-Indexen
  - XQuery
  - Suchen von Mustern in DNA-Sequenzen (begrenztes Alphabet: A, C, G, T)

Teilzeichenkette, Präfix, Suffix

- S sei eine Zeichenkette der Länge m
- S[i..j] ist dann eine Teilzeichenkette von S zwischen den Indizes i und j ($0 \leq i \leq j \leq m-1$)
- Ein Präfix ist eine Teilzeichenkette S[0..i] ($0 \leq i$ m-1)
- Eine Suffix ist eine Teilzeichenkette S[i..m-1] ($0 \leq i \leq m-1$)

Beispiele

- Teilzeichenkette S[1..3] = "ndr"

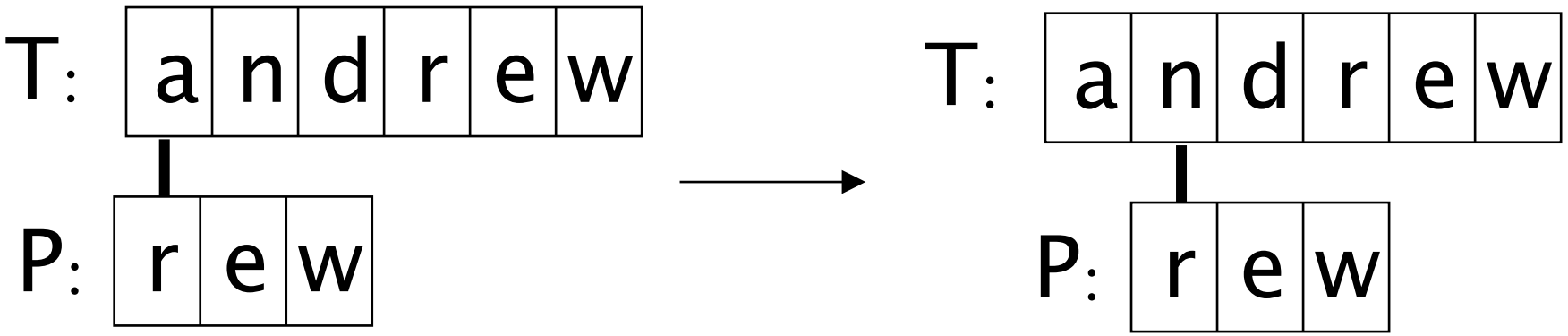


- Alle möglichen Präfixe von S:
  - "andrew", "andre", "andr", "and", "an”, "a"

- Alle möglichen Suffixe von S:
  - "andrew", "ndrew", "drew", "rew", "ew", "w"

Der Brute-Force-Algorithmus

- Problem: Bestimme Position des ersten Vorkommens von Muster P in Text T oder liefere -1, falls P nicht in T vorkommt
- Idee: Überprüfe jede Position im Text, ob das Muster dort startet



P bewegt sich jedes Mal um 1 Zeichen durch T

. . . .

Brute-Force-Suche

```
function BFsearch(text, pattern: String): Integer
    n := length(text); m := length(pattern)
    for i from 0 to (n-m) do
        j := 0
        while j<m and text[i+j] = pattern[j] do    // passende Teilkette
            j := j +1
        if j = m then
            return i // erfolgreiche Suche
    return -1 // erfolglose Suche
```

- Datentyp String entspricht Array [0..length-1] of Character
- Character umfasse hier 128 Zeichen (ASCII)

Analyse der Komplexität für Suche

- Schlechtester Fall für erfolglose Suche
  - Beispiel
    - Text: „aaaaaaaaaaaaaaaaaaaaa“
    - Muster: „aaaah“
  - Das Muster wird an jeder Position im Text durchlaufen: O(n·m)
- Bester Fall für erfolglose Suche
  - Beispiel
    - Text: „aaaaaaaaaaaaaaaaaaaa“
    - Muster: „bbbbbb“
  - Das Muster kann an jeder Position im Text bereits am ersten Zeichen des Musters falsifiziert werden: O(n)
- Komplexität erfolgreiche Suche im Durchschnitt
  - Meist kann das Muster bereits an der ersten Stelle des Musters falsifiziert werden und in der Mitte des Textes wird das Muster gefunden: O(n+m)

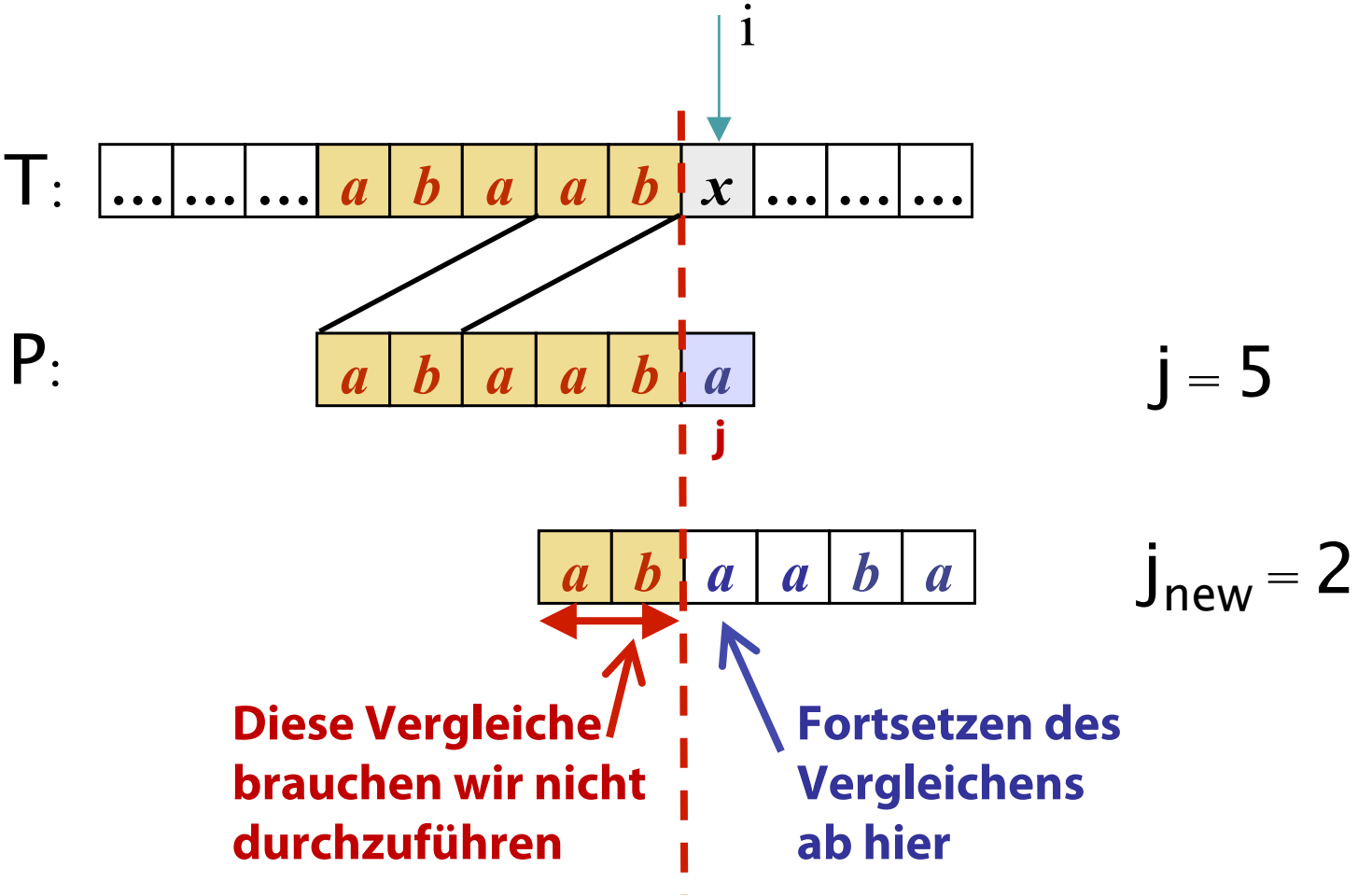
Weitere Analyse

- Brute-Force-Algorithmus ist um so schneller, je größer das Alphabet ist
  - Größere Häufigkeit, dass das Muster bereits in den ersten Zeichen falsifiziert werden kann

- Für kleine Alphabete (z.B. binär 0,1) ungeeigneter

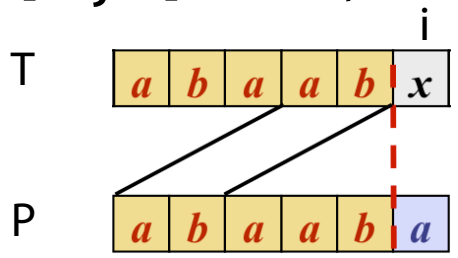
- Bessere Verschiebung des Musters zum Text als bei Brute-Force möglich?

Beispiel


i
T:
a b a a b x
P:
a b a a b a
j
a b a a b a
j = 5
j_new = 2
Diese Vergleiche brauchen wir nicht durchzuführen
Fortsetzen des Vergleichens ab hier

Knuth-Morris-Pratt-Algorithmus (KMP)

- Vergleich des Musters im Text von links nach rechts
  - Wie der Brute-Force-Ansatz
- Bessere Verschiebung des Musters zum Text als bei Brute-Force
  - **Frage:** Falls das Muster an der Stelle j falsifiziert wird, was ist die größtmögliche Verschiebung, um unnötige Vergleiche zu sparen?
  - **Antwort:** um den längsten Präfix P[0..j-1] von P, der ein Suffix T[i-j..i-1] von T ist

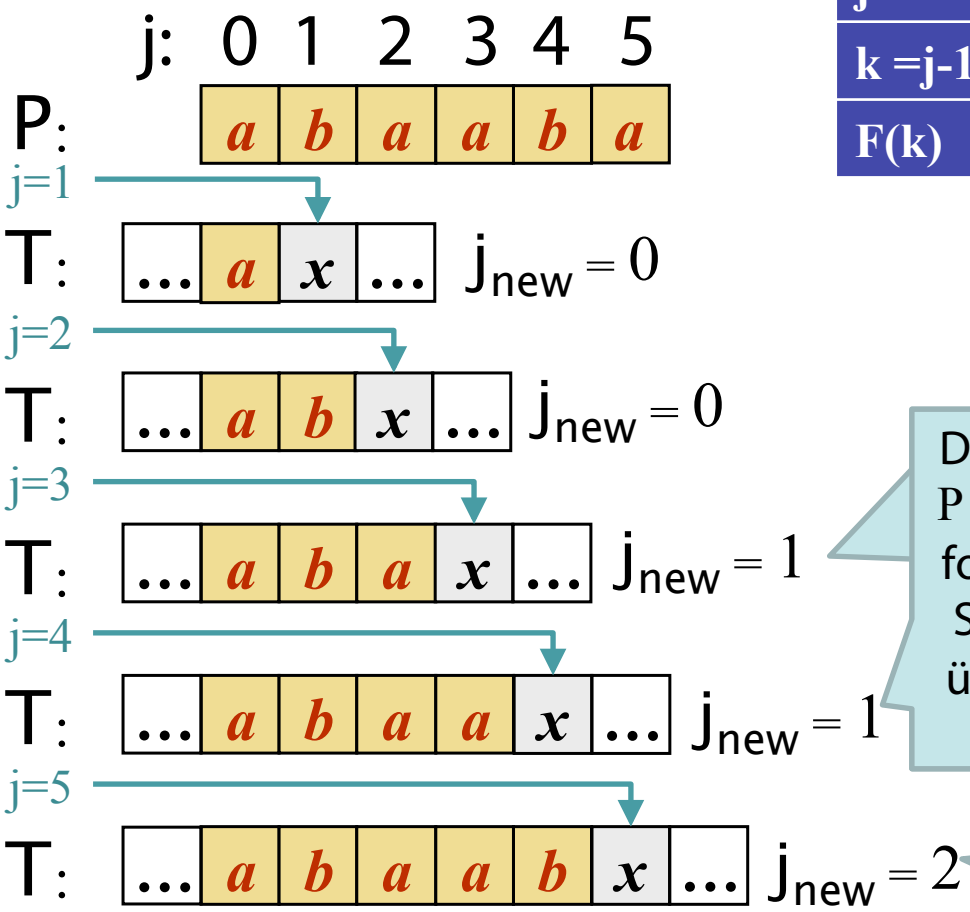


- Wie können solche Präfixe mit vertretbarem Aufwand bestimmt werden?

KMP Fehlerfunktion

- KMP verarbeitet das Muster vor, um Übereinstimmungen zwischen den Präfixen des Musters mit sich selbst zu finden
- j = Position der Ungleichheit in P
- k = Position vor der Ungleichheit (k = j-1).
- Die sog. Fehlerfunktion F(k) ist definiert als die Länge des längsten Präfixes von P[0..k], welcher auch ein Suffix T[i-j..i-1] von T ist

Beispiel Fehlerfunktion



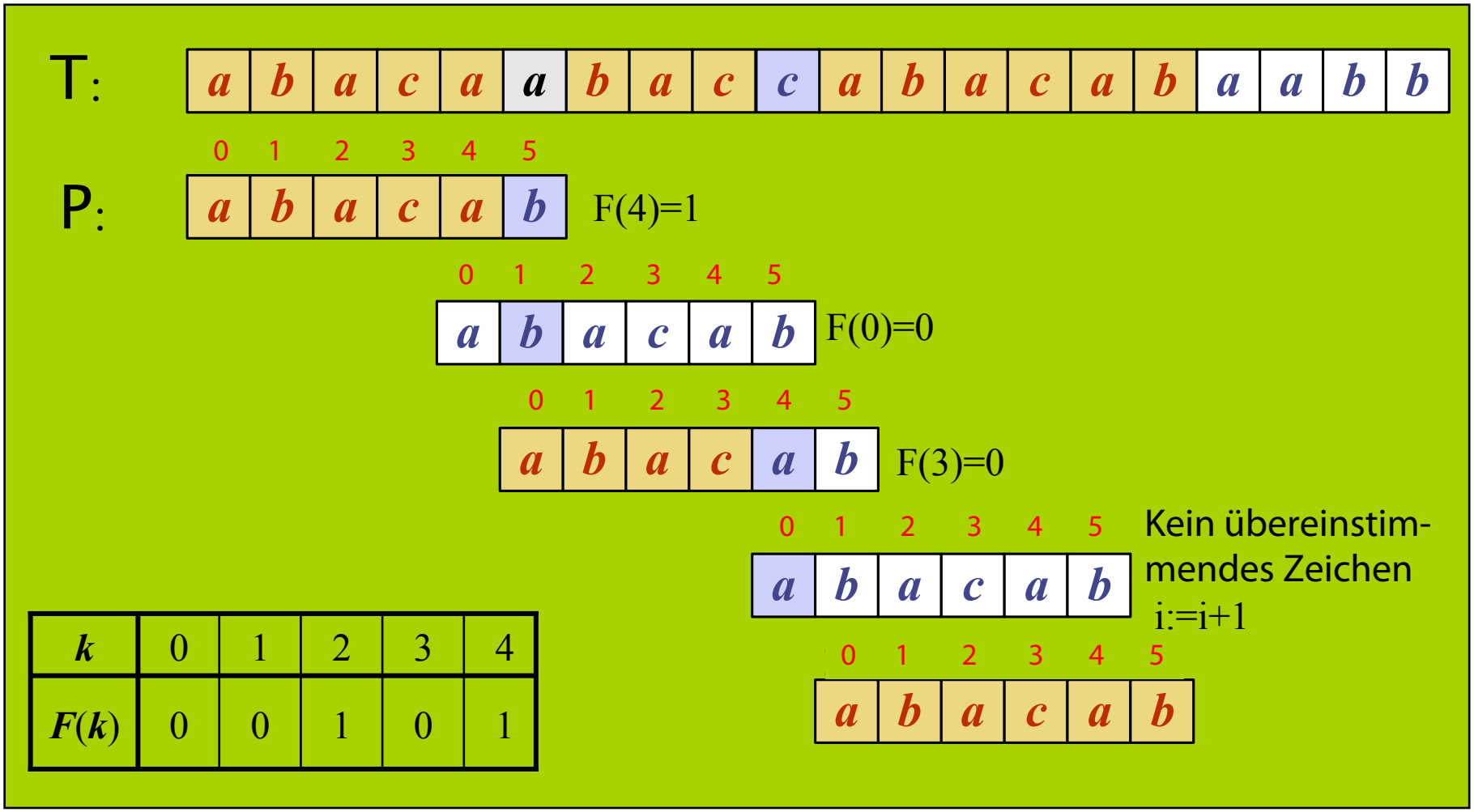
j	1	2	3	4	5
k =j-1	0	1	2	3	4
F(k)	0	0	1	1	2

- F(k) ist die Länge des längsten Präfix (j_{new} für j=k+1)
- Im Code wird F(k) als ein Feld gespeichert

Das erste a in P braucht bei fortgesetzter Suche nicht überprüft zu werden!

a und b in P braucht bei fortgesetzter Suche nicht überprüft zu werden!

Beispiel



k	0	1	2	3	4
$F(k)$	0	0	1	0	1

Unterschiede im Code zum Brute-Force-Algorithmus

- Knuth-Morris-Pratt Ansatz modifiziert den Brute-Force-Algorithmus
- Falls keine Übereinstimmung bei P[j]
  (d.h. P[j] ≠ T[i]), dann

```
k := j-1;
j := F(k);    // berechne neues j
```

Verfahren Knuth-Morris-Pratt

```
Procedure KMPsearch(text, pattern: String): Integer
    n := length(text); m := length(pattern)
    F[] := computeF(pattern)
    i := 0; j := 0
    while i<n do    // passende Teilkette
      if pattern[j] = text[i] then
        if j = m - 1 then
          return i – m + 1 // erfolgreiche Suche
        i := i + 1
        j := j + 1
      else if j > 0 then
              j := F[j – 1]
           else i := i + 1
    return -1 // erfolglose Suche
```

Fehlerfunktion computeF

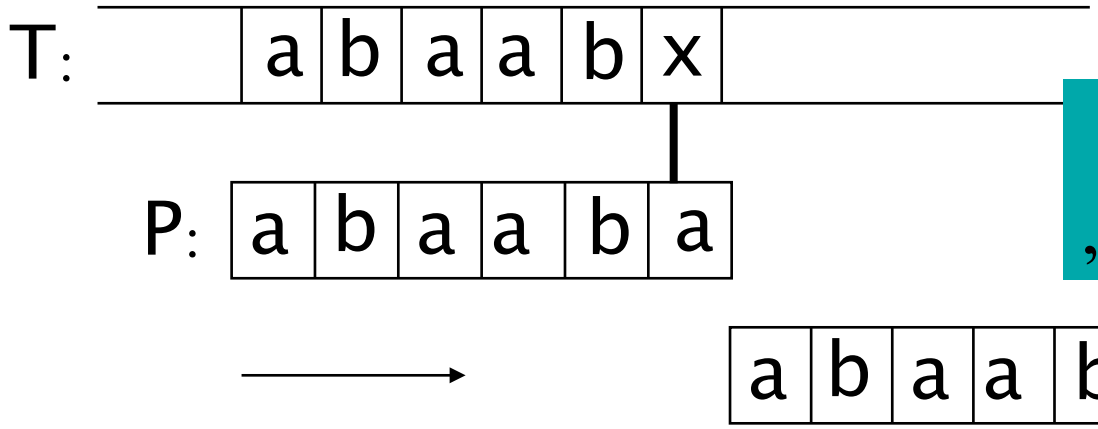
```
function computeF(pattern: String): Array[] of IN
    F:= <0,…,0>: Array [0..length(pattern)-1 - 1] of IN
    F[0] := 0
    m := length(F)
    i := 1; j := 0
    while i < m do
      if pattern[j] = pattern[i] then     // j+1 Zeichen stimmen überein
        F[i] := j + 1
        i := i + 1; j := j +1
      else if j > 0 then   // j folgt dem übereinstimmenden Präfix
              j := F[j – 1]
           else     // keine Übereinstimmung
              F[i] := 0
              i := i + 1
    return F
```

Vorteile vom Knuth-Morris-Pratt-Algorithmus

- Der Algorithmus springt niemals zurück im Text
  - Geeignet für Datenströme
- Komplexität O(m+n)
- Algorithmus wird langsamer, wenn das Alphabet größer ist
  - Werte der Fehlerfunktion werden tendenziell kleiner

Erweiterungen von Knuth-Morris-Pratt

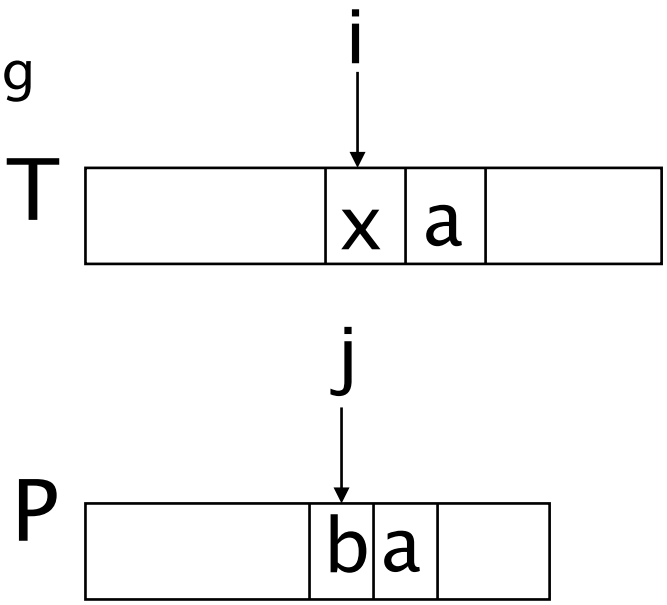
- Der Original-Algorithmus berücksichtigt bei der Verschiebung nicht das Zeichen auf Grund dessen keine Übereinstimmung gefunden wurde



Original-KMP „macht“ dieses **nicht**!

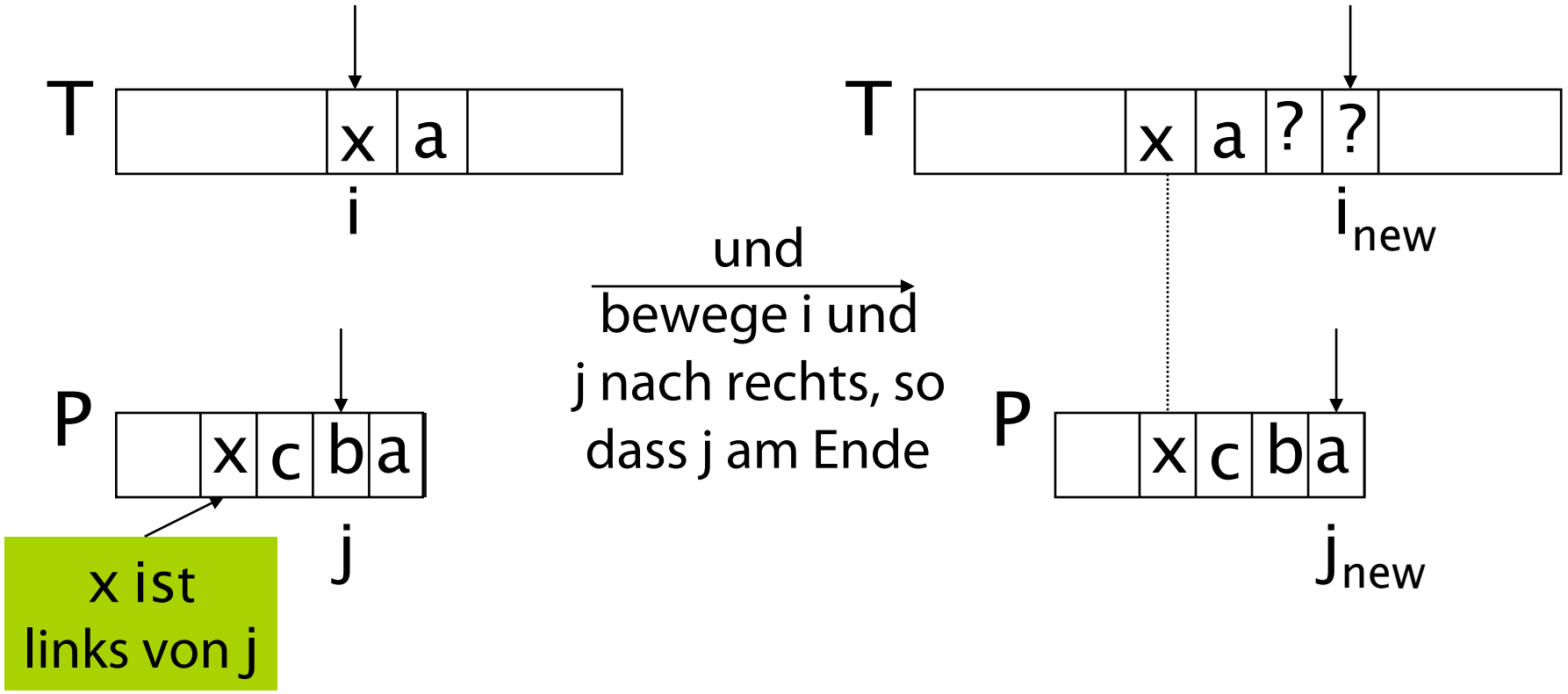
Boyer-Moore-Algorithmus

- Basiert auf 2 Techniken
  - Spiegeltechnik
    - Finde P in T durch Rückwärtslaufen durch P, am Ende beginnend
  - Zeichensprungtechnik
    - Im Falle von Nichtübereinstimmung des Textes an der i-ten Position (T[i]=x) und des Musters an der j-ten Position (P[j]≠T[i])
    - 3 Fälle…



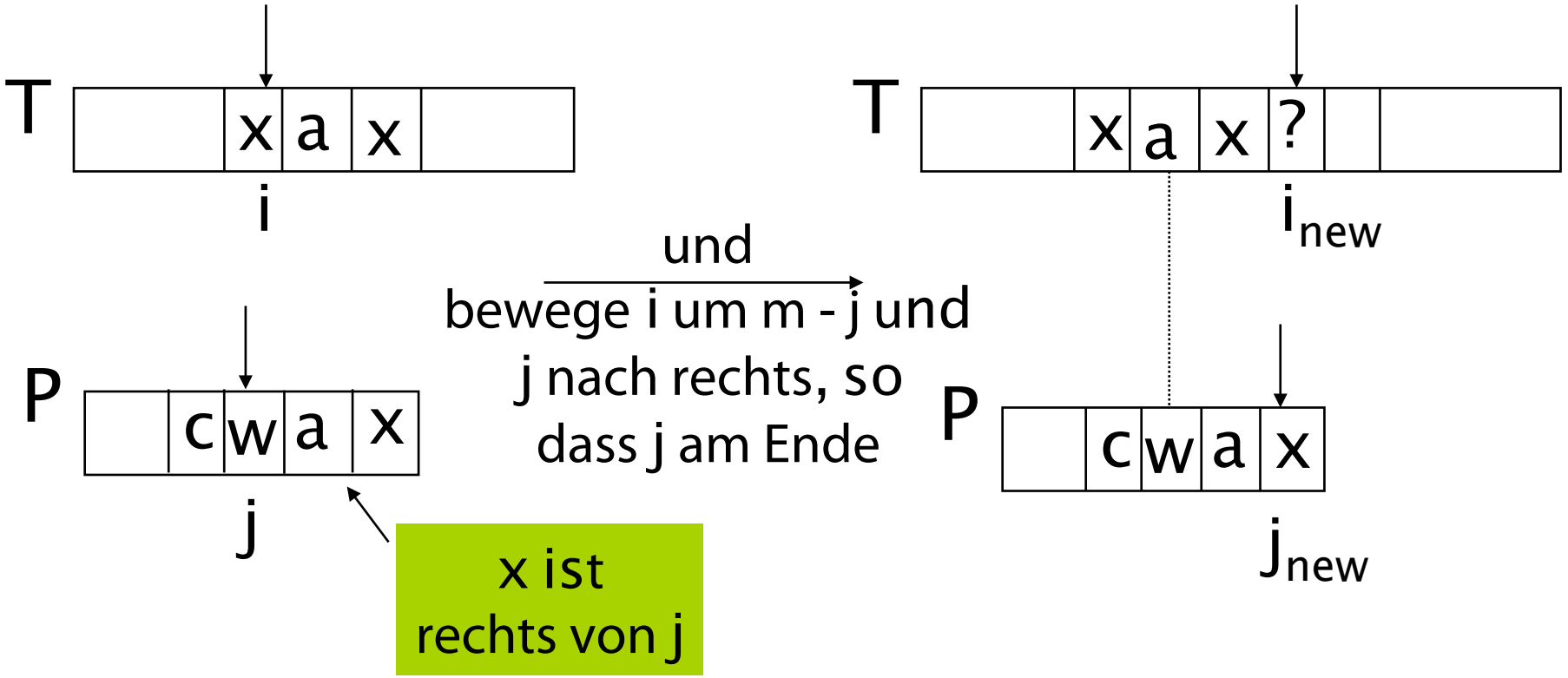
Fall 1: P enthält x nur links von j

- Bewege P nach rechts, um das letzte Vorkommen von x in P mit T[i] abzugleichen



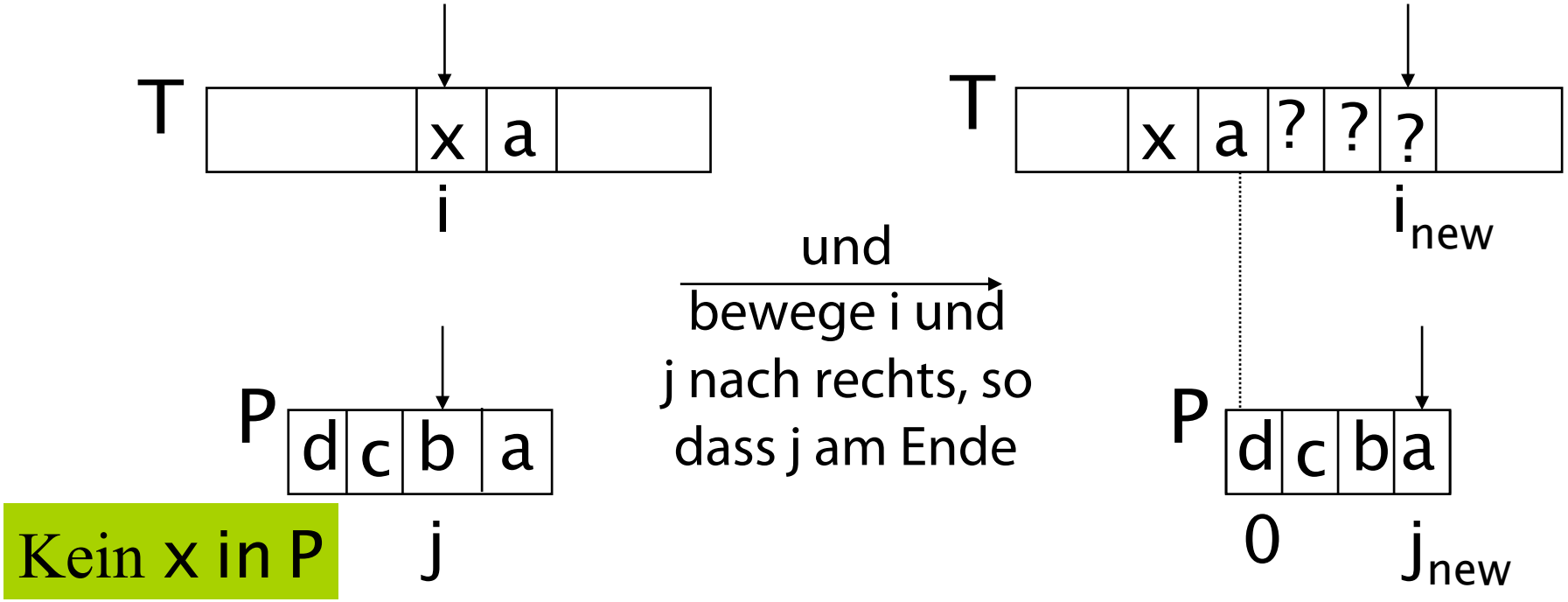
Fall 2: P enthält x rechts von j

- Bewege P um 1 Zeichen nach T[i+1]

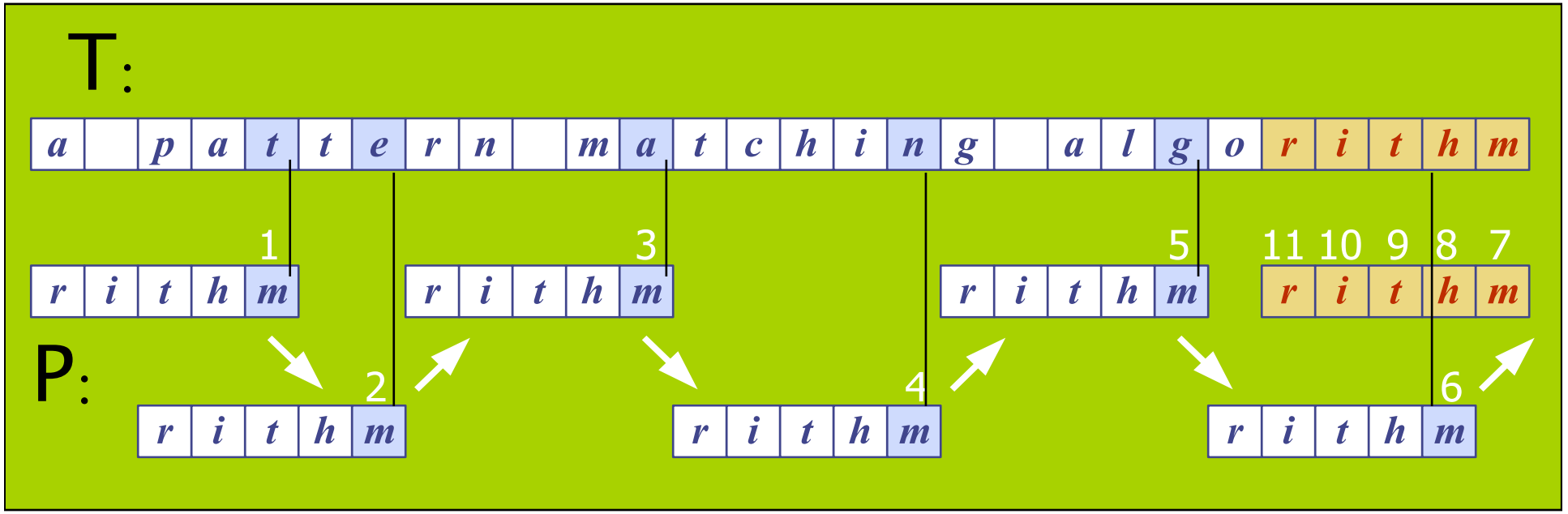


Fall 3: Falls Fall 1 und 2 nicht anzuwenden sind (x ist *nicht* in P enthalten)

- Bewege P nach rechts, um P[0] und T[i+1] abzugleichen



Beispiel (Boyer-Moore)



Funktion des letzten Vorkommens

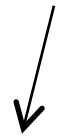
- Der Boyer-Moore Algorithmus verarbeitet das Muster P und das Alphabet A vor, so dass eine Funktion L des letzten Vorkommens berechnet wird
  - L bildet alle Zeichen des Alphabets auf ganzzahlige Werte ab
- L(x) (mit x ist Zeichen aus A) ist definiert als
  - den größten Index i, so dass P[i]=x, oder
  - -1, falls solch ein Index nicht existiert
- Implementationen
  - L ist meist in einem Feld der Größe |A| gespeichert

Beispiel L

A = {a, b, c, d}
P = „abacab“

P

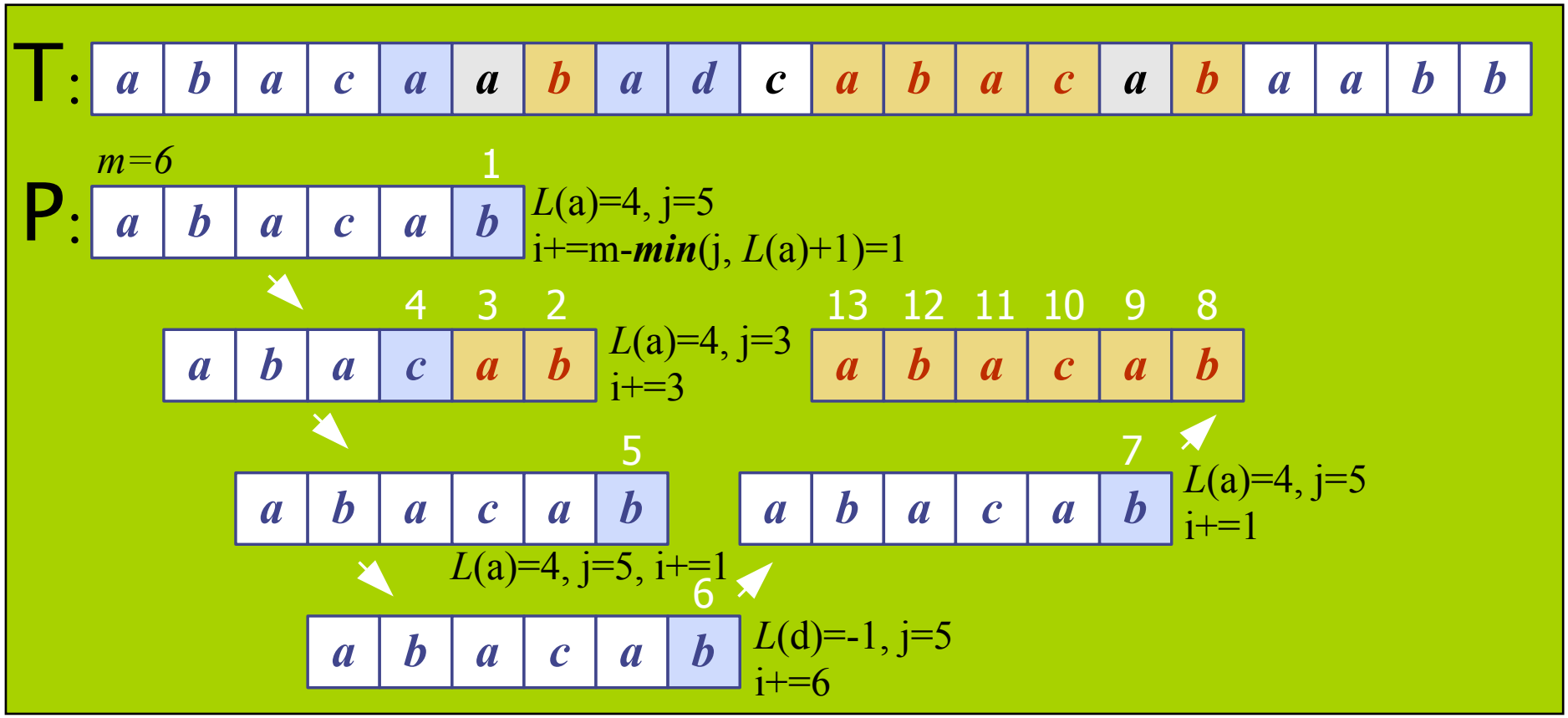
a	b	a	c	a	b
0	1	2	3	4	5



x	a	b	c	d
$L(x)$	4	5	3	-1

L speichert Indexe von P

Zweites Boyer-Moore-Beispiel



x	a	b	c	d
$L(x)$	4	5	3	−1

Funktion des letzten Vorkommens

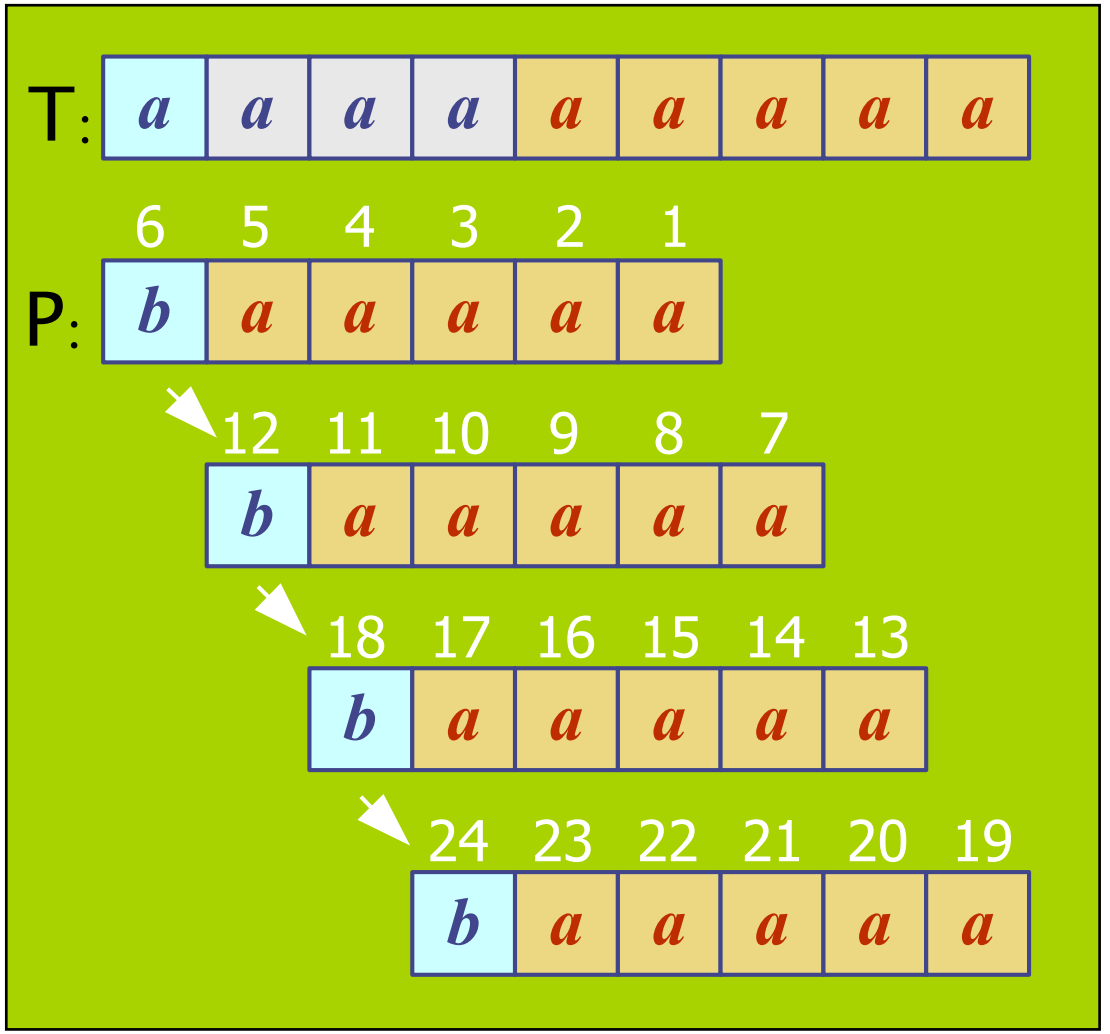
```
function buildL(pattern: String): Array[0..127] of Integer
    l := <-1,…,-1>: Array [0..127] of Integer // nur ASCII unterstützt
    for i from 0 to length(pattern)-1 do
        l[pattern[i]] := i
    return l
```

Boyer-Moore-Verfahren

```
Procedure BMsearch(text, pattern: String): Integer
    l := buildL(pattern)
    n := length(text); m := length(pattern)
    i := m - 1
    if i > n - 1 then
        return -1       // Muster länger als Text, Suche erfolglos
    j := m – 1
    repeat
      if pattern[j] = text[i] then
        if j = 0 then
          return i    // Suche erfolgreich
        else            // Spiegeltechnik
          i := i -1; j := j -1
      else              // Zeichensprungtechnik
        lo := l[text[i]]; i := i + m – min(j, lo+1); j := m – 1
    until j ≤ n – 1
    return -1           // Suche erfolglos
```

Analyse der Komplexität

- Schlechtester Fall
  - Beispiel
    - T: „aaaaaaaaaa…a“
    - P: „baa…a“
  - O(m·n + |A|)
  - Wahrscheinlichkeit hoch für schlechtesten Fall bei kleinem Alphabet



Analyse der Komplexität

- Bester Fall
  - Immer Fall 3, d.h. P wird jedes Mal um m nach rechts bewegt
  - Beispiel
    - T: „aaaaaaaa…a“
    - P: „bbb…b“
  - $O(n/m + |A|)$
  - Wahrscheinlichkeit für besten Fall höher bei großem Alphabet
- Durchschnittlicher Fall
  - nahe am besten Fall: $O(n/m + |A|)$

Rabin-Karp-Algorithmus

- Idee
  - Ermittle eine Hash-Signatur des Musters
  - Gehe durch den zu suchenden Text durch und vergleiche die jeweilige Hash-Signatur mit der des Musters
    - Mit geeigneten Hash-Funktionen ist es möglich, dass die Hash-Signatur iterativ mit konstantem Aufwand pro zu durchsuchenden Zeichen berechnet wird
    - Falls die Hash-Signaturen übereinstimmen, dann überprüfe noch einmal die Teilzeichenketten

Hash-Funktion für Rabin-Karp-Algorithmus

- Für ein Zeichen
  - h(k)=k·p mit k z.B ASCII-Code des betrachteten Zeichens und q eine Primzahl
- Für eine Zeichenkette
  - $h'(k_1..k_m) = h(k_1) + ... + h(k_m)$
- Beispiel
  - q=5 (in der Praxis sollte allerdings eine möglichst große Primzahl gewählt werden)
  - A={1, 2, 3, 4}
  - der Einfachheit halber sei der Code des Zeichens i wiederum i
- Berechnung der Hash-Signatur des Musters 1234:
  - h'(1234) = 1·5 + 2·5 + 3·5 +4·5 = 50

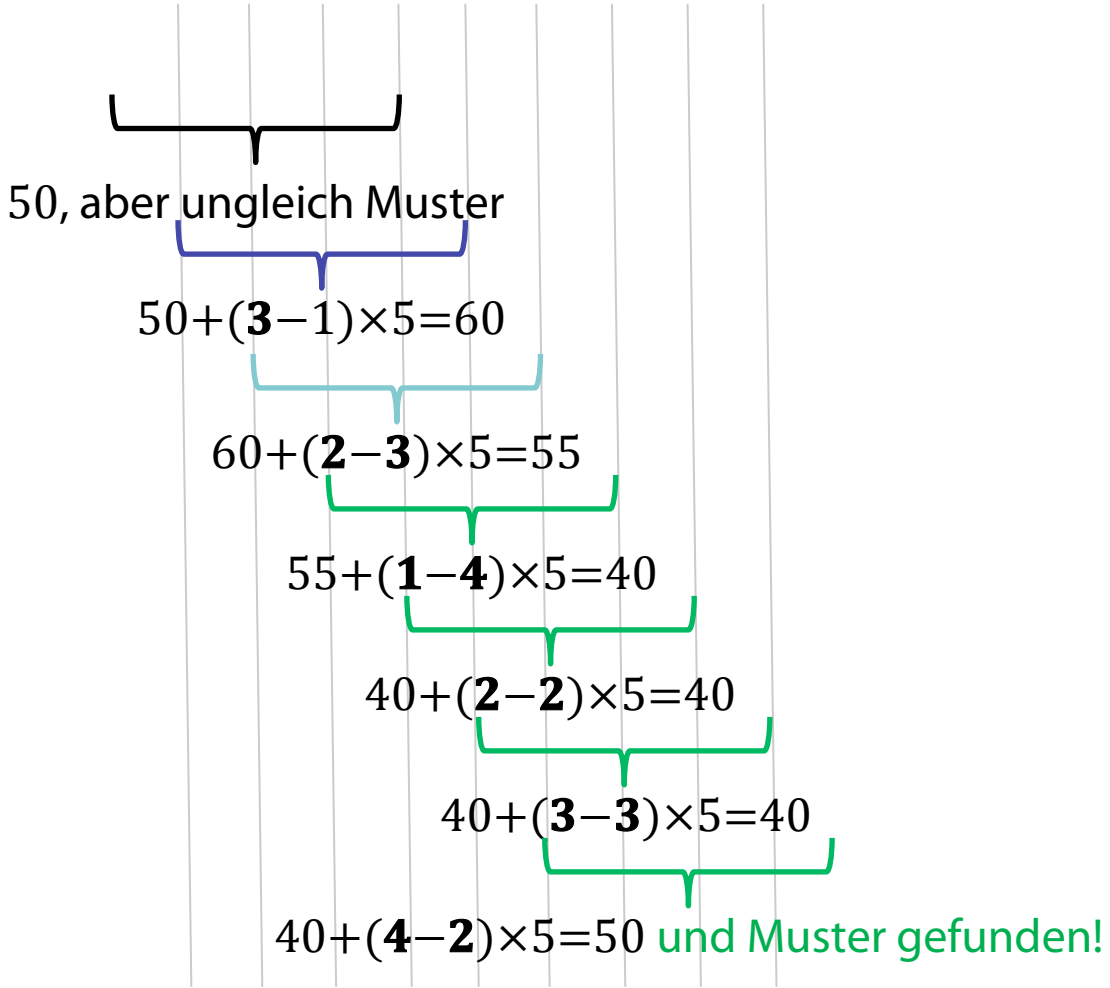
Suche nach der Hash-Signatur

- Einmaliges Durchlaufen des zu durchsuchenden Textes und Vergleich der aktualisierten Hash-Signatur mit Hash-Signatur des Musters
- Hash-Signatur kann iterativ gebildet werden:

$$h'(k_2..k_m\ k_{m+1}) = h'(k_1\ k_2\ ...\ k_m) - k_1 \cdot q + k_{m+1} \cdot q$$
$$= h'(k_1\ k_2\ ...\ k_m) + (k_{m+1} - k_1) \cdot q$$

- **Man beachte:** Konstanter Aufwand pro Zeichen

Beispiel: Suche nach der Hash-Signatur



Komplexitätsanalyse Rabin-Karp-Algorithmus

- Ermittle die Hash-Signatur des Musters: O(m)
- Gehe durch den zu suchenden Text durch und vergleiche die jeweilige Hash-Signatur mit der des Musters
  - Best (und Average) Case: O(n)
    - Hash-Signaturen stimmen nur bei einem Treffer überein
  - Worst Case: O(n · m)
    - Hash-Signaturen stimmen immer überein, auch bei keinem Treffer
- Insgesamt O(n + m) im Best/Average Case und O(n · m) im schlimmsten Fall

Variante des Rabin-Karp-Algorithmus für Suche nach vielen Mustern

- Alle Hash-Signaturen (für Muster bis zu einer gegebenen maximalen Länge) können mit einem Durchlauf durch den Text in eine geeignete Datenstruktur z.B. eine Hashtabelle abgelegt werden
- Anschließende Suche nach vielen Mustern schnell möglich
  - im durchschnittlichen Fall in konstanter Zeit pro Muster
- **Aber:** Großer Speicherbedarf!

Zusammenfassung

- Textsuche
  - Brute-Force
  - Knuth-Morris-Pratt
  - Boyer-Moore
  - Rabin-Karp

Acknowledgements

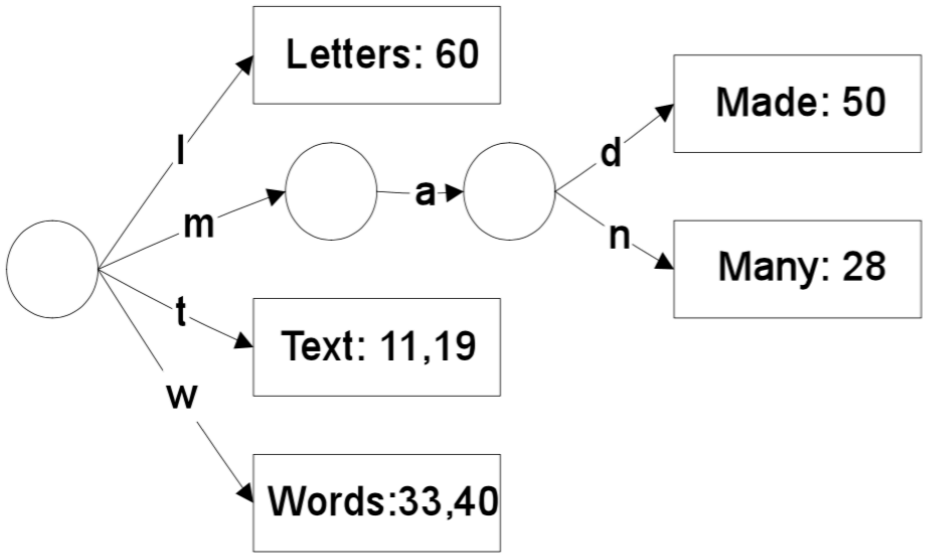
- Präsentationen im nachfolgenden Teil sind entnommen aus dem Material zur Vorlesung Indexierung und Suchen von Tobias Scheffer, Univ. Potsdam

Indexstrukturen für eindimensionale Daten

Wiederholung: Tries

1	6 9	11	17 19	24	28	33	40	46	50	55	60

This is a text. A text has many words. Words are made from letters.



Indexstrukturen für eindimensionale Daten

Wiederholung: Invertierter Index

1 6 9 11 17 19 24 28 33 40 46 50 55 60

This is a text. A text has many words. Words are made from letters.

Terme	Vorkommen
Letters	60
Made	50
Many	28
Text	11, 19
words	33, 40

Invertierter Index mit Blockadressierung

1	2	3	4
This is a text.	A text has many	words. Words are	made from letters.

Terme	Vorkommen
Letters	4
Made	4
Many	2
Text	1, 2
words	3

Aufbau eines invertierten Index mit Tries

- Iteriere über alle Texte, iteriere über alle Positionen des Textes, an denen ein neues Wort beginnt.
  - Füge (Wort, Position) sortiert in einen Trie ein.
  - Wenn Speicher voll ist, dann speichere Trie, lade einen Unterbaum in den Speicher, berücksichtige nur Wörter, die in diesen Unterbaum gehören. Dann abspeichern, mit nächstem Unterbaum weitermachen, dazu Text neu iterieren.
- Traversiere den Trie und schreibe die Wörter in den invertierten Index.

Aufbau eines invertierten Index mit Listen

- Iteriere über Dokumente, parse Tokens,
- Füge Tokens mit Textpositionen in unsortierte Liste ein.
- Hänge alle unsortierten Listen aneinander.
- Sortiere die Listen.
- Eliminiere doppelt vorkommende Terme, füge die Vorkommen zu einer Liste zusammen.
- Liste muss in den Speicher passen.

Aufbau eines invertierten Index mit Blöcken

- Lies Dokumente in k Blöcken ein.
- Bilde Termliste für jeweiligen Block.
- Sortiere lokalen Index für den Block, speichere.
- Merge lokale Indizes in $\log_2$ k Ebenen.
  - Für alle Ebenen, für alle Paare, sequentiell einlesen, mergen, abspeichern.

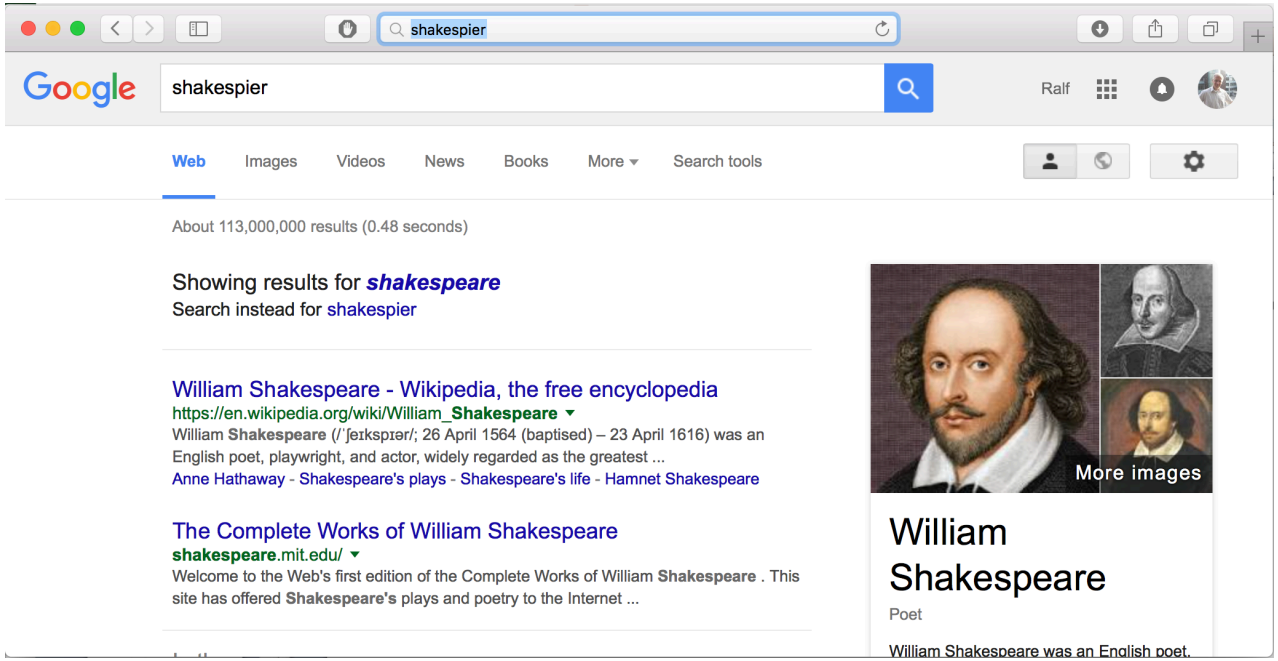
Suche mit invertiertem Index

- Gegeben: Suchanfrage q.
- Schlage einzelne Terme der Suchanfrage im Index nach, dazu Binärsuche in der Termliste.
- Greife auf die gefundenen Dokumente zu.
- Löse Phrasen-, Nachbarschafts- und Boolesche Anfrage anhand der Dokumente auf.
- (Bei Blockadressierung, Suche in den Blöcken.)

- Liste der Terme passt in der Regel in den Hauptspeicher, Listen der Vorkommen aller Terme häufig nicht.

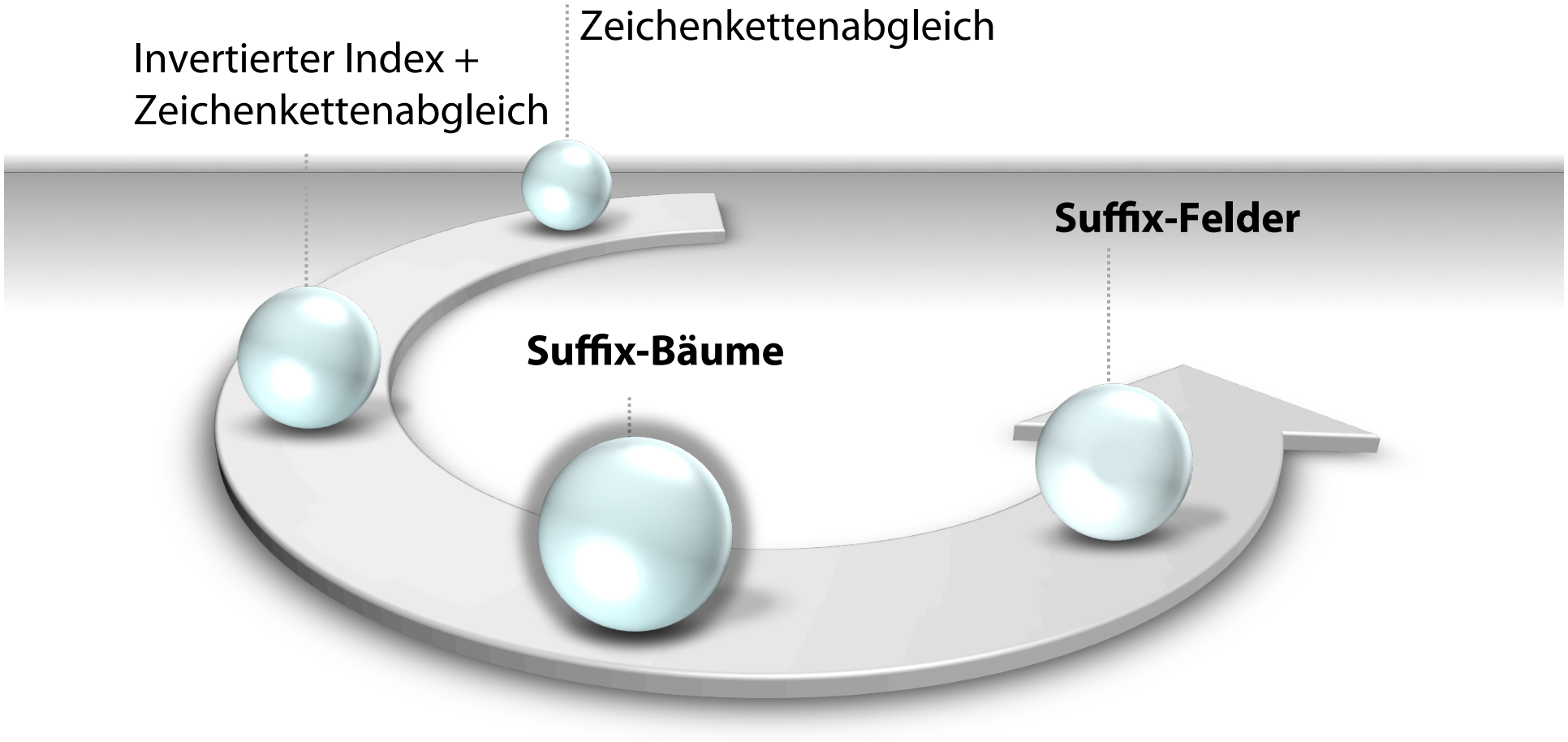
Umgang mit Rechtschreibfehlern

- Erweitere Suchanfrage um Terme, die Edit-Abstand von höchstens x haben.
- Verlangsamt die Suche sehr. Lösung: nur, wenn ursprüngliche Anfrage keinen Treffer liefert.



Non-Standard-Datenbanken

Zeichenkettenabgleich ohne Wortgrenzen

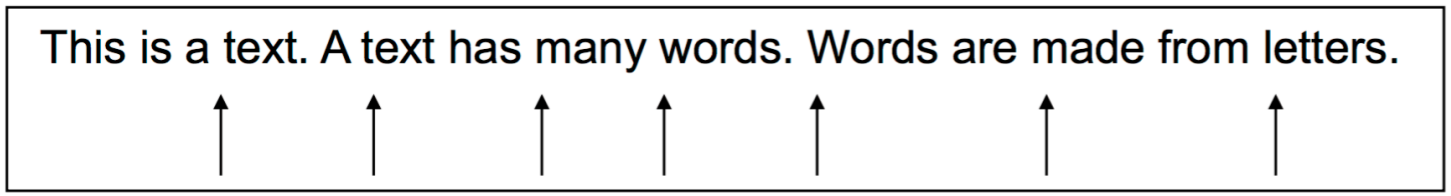


Motivation

- Invertierter Index für Nachbarschafts-anfragen nicht so gut geeignet
- Erfordert Tokenisierung
  (geht nicht für DNA-Sequenzen)
- Idee von Suffix-Bäumen: Text ist eine einzige Zeichenkette, jede Position
  ist ein Suffix (von „hier“ bis zum Ende)
- Trie-Struktur über alle Suffixe

Suffix-Bäume

- Indexpunkte können Wortanfänge oder alle Zeichenkettenpositionen sein.
- Text ab Position: Suffix.

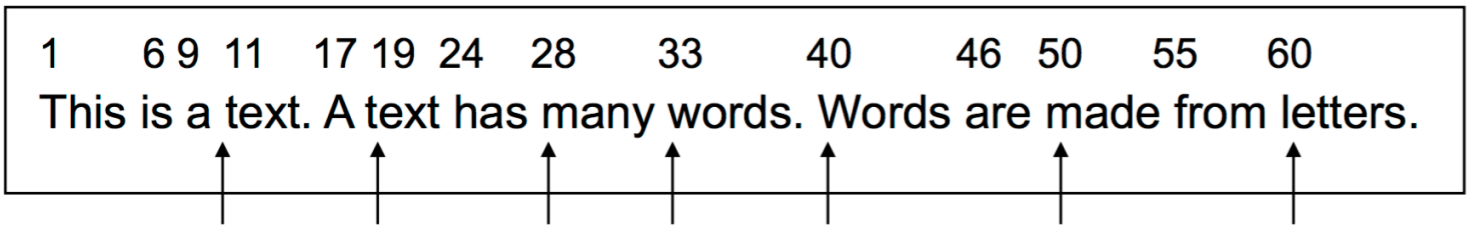


Suffizes:

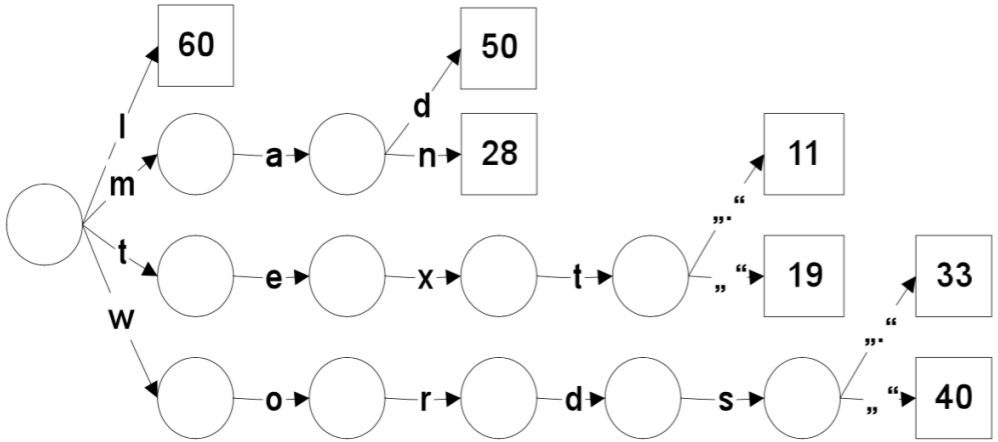
text. A text has many words. Words are made from letters.
text has many words. Words are made from letters.
many words. Words are made from letters.
words. Words are made from letters.
Words are made from letters.
made from letters.
letters.

Suffix-Tries

- Aufbau eines Suffix-Tries:
- Für alle Indexpunkte:
  - Füge Suffix ab Indexpunkt in den Trie ein.



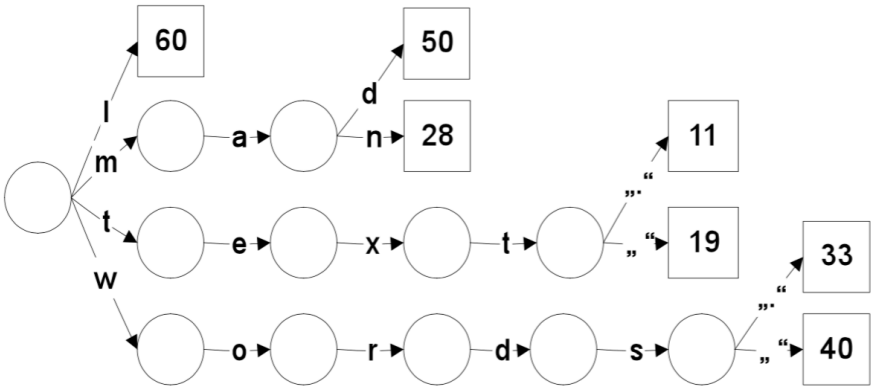
Suffix-Trie:



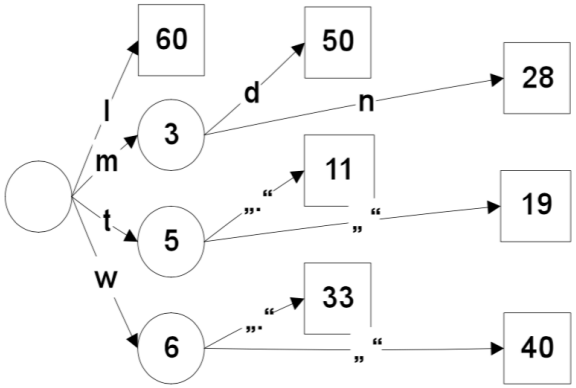
Patricia-Tries

- Ersetze interne Knoten mit nur einer ausgehenden Kante durch „Überleseknoten“, beschrifte sie mit der nächsten zu beachtenden Textposition.

Suffix-Trie:



Patricia-Tree:



Suche im Suffix-Baum

Eingabe: Suchzeichenkette, Wurzelknoten.

Wiederhole

1. Wenn Terminalknoten, liefere Position zurück, überprüfe, ob Suchzeichenkette an dieser Position steht.
2. Wenn „Überleseknoten“, spring bis zur angegebenen Textposition weiter.
3. Folge der Kante, die den Buchstaben an der aktuellen Position akzeptiert.

Suffix-Bäume

- Konstruktion: O(Länge des Textes).
- Algorithmus funktioniert schlecht, wenn die Struktur nicht in den Hauptspeicher passt.
- Problem: Speicherstruktur wird ziemlich groß, ca. 120-240% der Textsammlung, selbst wenn nur Wortanfänge (Längenbegrenzung) indexiert werden.
- Suffix-Felder (Arrays): kompaktere Speicherung.

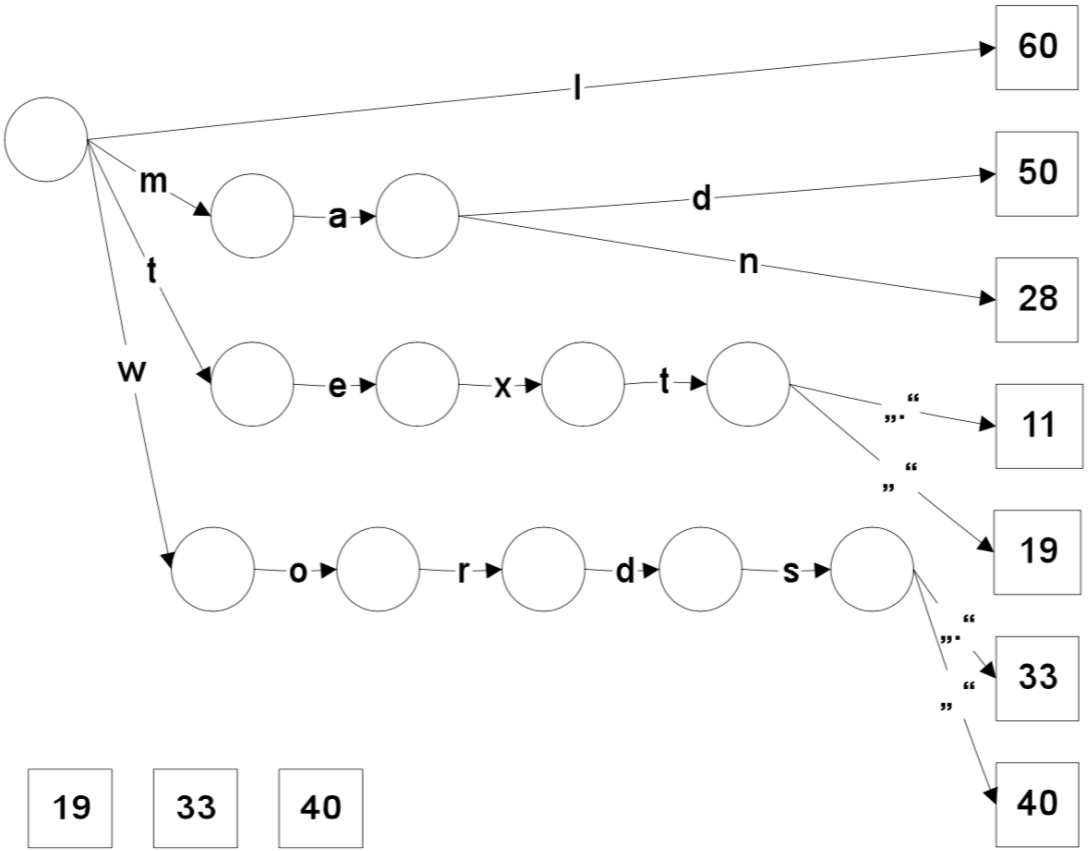
Suffix-Felder (Aufbau naiv)

- Suffix-Trie in lexikographische Reihenfolge bringen.
- Suffix-Feld= Folge der Indexpositionen.

Suffix-Trie,

Lexikographisch

sortiert:



Suffix-Array:

60	50	28	11	19	33	40

Suche in Suffix-Feldern: Binärsuche

```
procedure find(p, A): Integer
    // Suchzeichenkette p, Suffix-Feld A
    // liefert Position von p in A oder -1, falls p nicht in A
    s := 1
    e := length(A)
    while s < e do
        m := ⌈(e + s) / 2⌉  // nach oben runden
        if p = Text(A[m]) then
            return m
        if p < Text(A[m])
         then e := m
         else s := m
     return -1
```

U. Manber and G.W. Myers "Suffix arrays: A new method for on-line string searches". In Proceedings of the 1st ACM-SIAM Symposium on Discrete Algorithms, **1990**

Beispiel

Suffixsortierung → Binäre Suche

SA	10	7	4	1	0	9	8	6	3	5	2
	1	2	3	4	5	6	7	8	9	10	11

lenght(SA) = 11

Beispiel: Suche „sip“ in „mississippi“

Beispiel von Karsten Klein, Uni Dortmund,
Vorlesung Algorithmen und Datenstrukturen

SA	L	7	4	1	0	M	8	6	3	5	R
	i	ippi	issippi	ississippi	mississippi	pi	ppi	sippi	sissippi	ssippi	ssissippi

sip < ssi

sip > i

sip > pi

Suche: sip

SA	10	7	4	1	0	L	8	6	M	5	R
	i	i	i	i	m	p	p	s	s	s	s
		p	s	s	i	i	p	i	i	s	s
		p	s	s	s		i	p	s	i	i
		i	i	i	s			p	s	p	s
			p	s	i			i	i	p	s
			p	s	s				p	i	i
			i	i	s				p		p
				p	i				i		p
				p	p						i
				i	p						
					i						

sip < sis

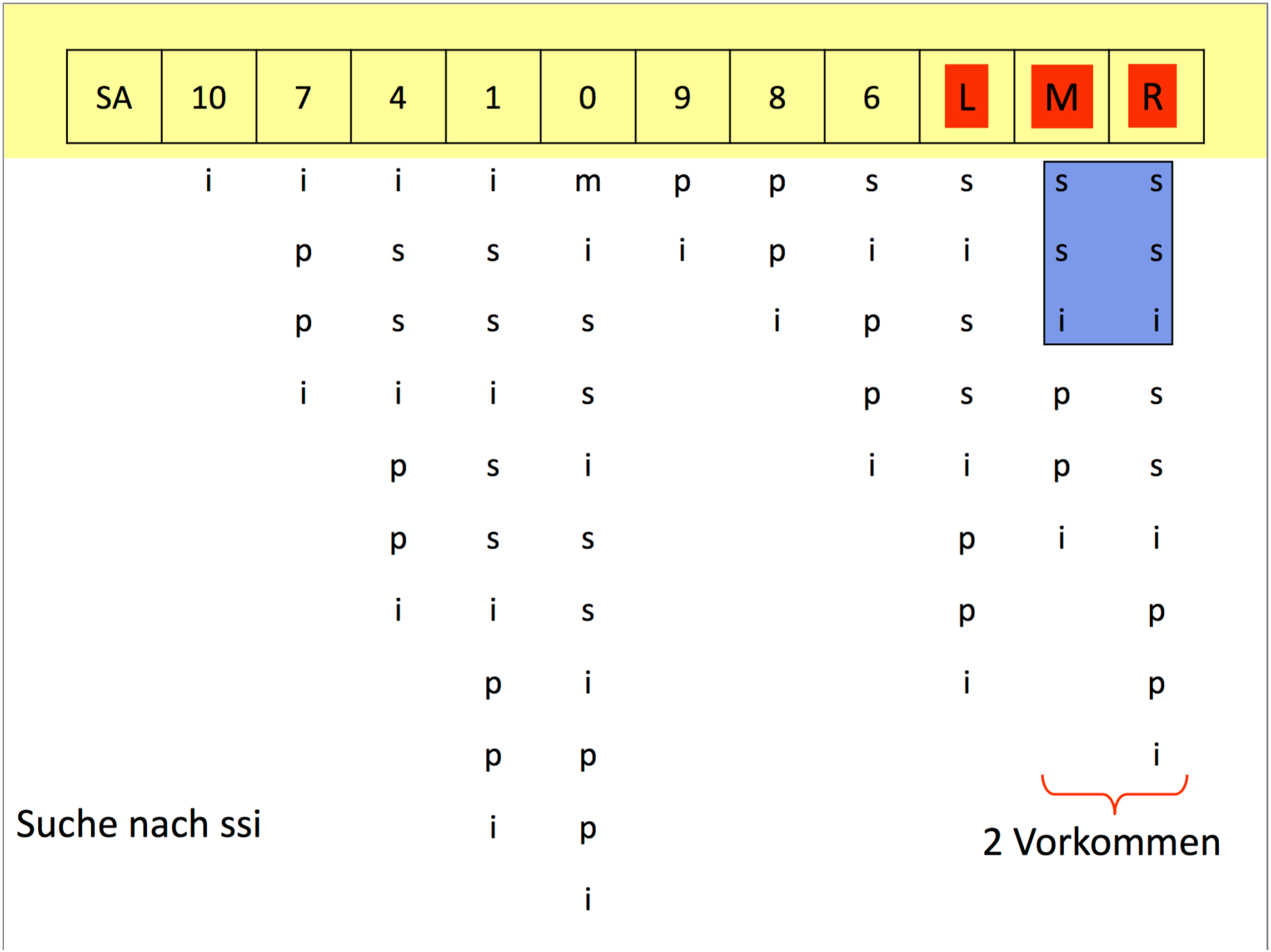
SA	10	7	4	1	0	L	8	M	R	5	2
	i	i	i	i	m	p	p	s	s	s	s
		p	s	s	i	i	p	i	i	s	s
		p	s	s	s		i	p	s	i	i
		i	i	i	s			p	s	p	s
			p	s	i			i	i	p	s
			p	s	s				p	i	i
			i	i	s				p		p
				p	i				i		p
				p	p						i
				i	p						
					i						

1 Vorkommen

Mehrere Vorkommen?

Vorkommen benachbart!

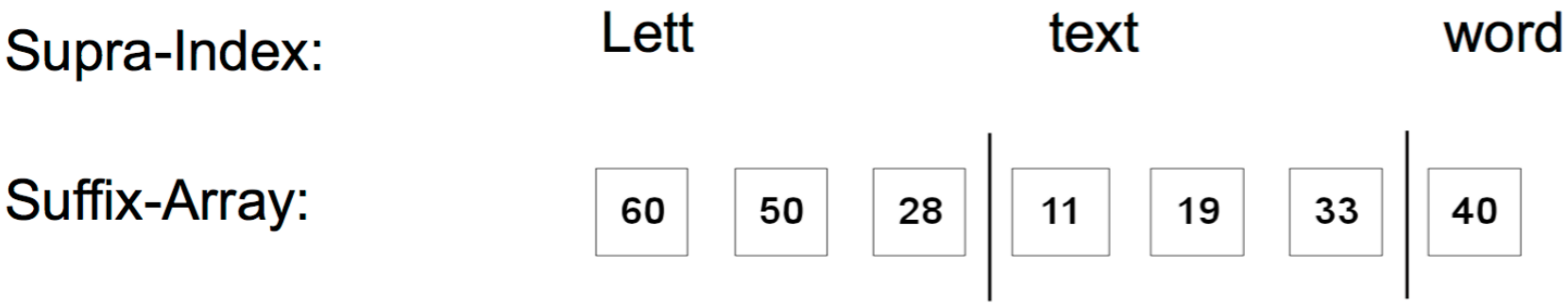
sip = sip


SA
10
7
4
1
0
9
8
6
L
M
R
Suche nach ssi
2 Vorkommen

Suche

- Suffix-Bäume: O(Länge der Suchzeichenkette)
- Suffix-Felder: O(log(Länge der Textsammlung)) und viele Datenbankzugriffe.
- Beschleunigung durch Supra-Index (im Hauptspeicher)
  - Von jedem k-ten Eintrag werden die ersten Buchstaben in einem Feld (Array) gespeichert.
  - Zuerst Binärsuche in Supra-Index, ohne Datenbankzugriff
  - Danach Binärsuche in Unterabschnitt des Suffix-Felder mit Datenbankzugriff
- Trade-Off zwischen Größe der Suffix-Struktur und Anzahl der Datenbankzugriffe

Suffix-Felder mit Supra-Index



- Binärsuche im Supra-Index (ohne Datenbankzugriff)
- Dann Binärsuche im entsprechenden Abschnitt des Suffix-Felder (mit Datenbankzugriffen).

- Speicherbedarf ähnlich wie invertierter Index.

Aufbau von Suffix-Feldern

- Ukkonens Algorithmus (1995, Hauptspeicher)
- Später: Datenbank-basierte Verfahren (ab 2001)
  z.B. für Bioinformatik-Anwendungen

Weiner, Peter (1973). Linear pattern matching algorithms, 14th Annual Symposium on Switching and Automata Theory, pp. 1–11, **1973**

McCreight, Edward Meyers, A Space-Economical Suffix Tree Construction Algorithm". Journal of the ACM 23 (2): 262–272, **1976**

Ukkonen, E., On-line construction of suffix trees, Algorithmica 14 (3): 249–260, **1995**

Hunt, E., Atkinson, M. and Irving, R. W. "A Database Index to Large Biological Sequences". VLDB **2001**

Tata, Hankins, Patel: „Practical Suffix Tree Construction“, VLDB **2004**

Weitere Abgleichsalgorithmen

- Eigenes, umfangreiches Thema. → Bioinformatik.
- Aho-Corasick-Trie, Idee: mehrere Suchzeichenketten
  - Suchzeichenketten werden in Automaten umgebaut,
  - Kanten akzeptieren Buchstaben,
  - Wenn keine passende Kante mehr existiert, Sprung über „failure transition“ in Zustand, der der größten Übereinstimmung zwischen Text und einem Prefix eines Suchzeichenketten entspricht.
- Suffix-Automaten

Approximativer Zeichenkettenabgleich

- Suche in y Unterzeichenketten, die Edit-Abstand von höchstens k von x haben.
- Dynamische Programmierung

  Berechnung von C[0...m, 0...n]; C[i,j] = minimale # Fehler beim Abgleich von x[1...i] mit y[1...j]

$$C[0, j] = j$$

$$C[i, 0] = i$$

$$C[i, j] = \begin{cases} C[i-1, j-1] \text{ , wenn } x[i] = y[j] \\ 1+\min \{C[i-1, j], C[i, j-1], C[i-1, j-1]\} \text{ , sonst} \end{cases}$$

$$O(nm)$$

Beispiel

- C[0, j] = j
- C[i, 0] = i
- C[i, j] = { C[i-1, j-1] , wenn x[i] = y[j]
  1+min {C[i-1, j], C[i, j-1], C[i-1, j-1]} , sonst

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

Beispiel

- C[0, j] = j
- C[i, 0] = i
- C[i, j] = { C[i-1, j-1] , wenn x[i] = y[j]
  1+min {C[i-1, j], C[i, j-1], C[i-1, j-1]} , sonst

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

Beispiel

- C[0, j] = j
- C[i, 0] = i
- C[i, j] = { C[i-1, j-1] , wenn x[i] = y[j]
  1+min {C[i-1, j], C[i, j-1], C[i-1, j-1]} , sonst

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

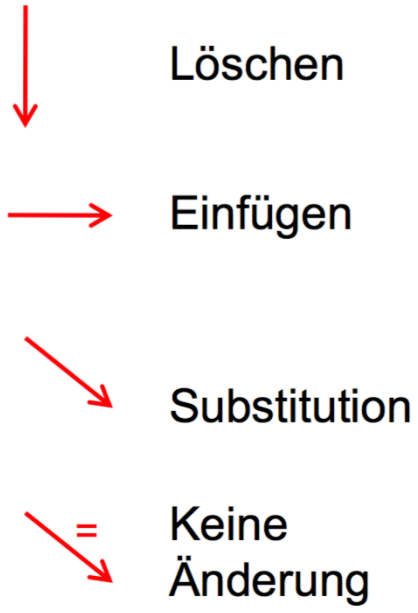
Beispiel

- C[0, j] = j
- C[i, 0] = i
- C[i, j] = { C[i-1, j-1] , wenn x[i] = y[j]
  1+min {C[i-1, j], C[i, j-1], C[i-1, j-1]} , sonst

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

Beispiel

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



cbabac -> ababac -> abcabac -> abcabbac -> abcabbbac -> abcabbbaa

Weitere approximative Verfahren

- Endliche Automaten für approximativen Zeichkettenabgleich.
- Suche nach regulären Ausdrücken: Konstruktion eines endlichen Automaten, der die Anzahl der Fehler beim Akzeptieren zählt.

Suffix-Bäume und Suffix-Felder

- Zeichenkettensuche mit Suffix-Bäumen und –Feldern kein Problem, aber jede Zeichenkettenposition muss indexiert werden. Speicherbedarf empirisch 1200-2400% der Textsammlung.
- Aktuelle Themen: Suche in komprimiertem Text, Suche mit komprimierten Indexdateien