Universität Potsdam
Institut für Informatik
Lehrstuhl Maschinelles Lernen

Indexieren und Suchen

Tobias Scheffer
Paul Prasse
Michael Großhans

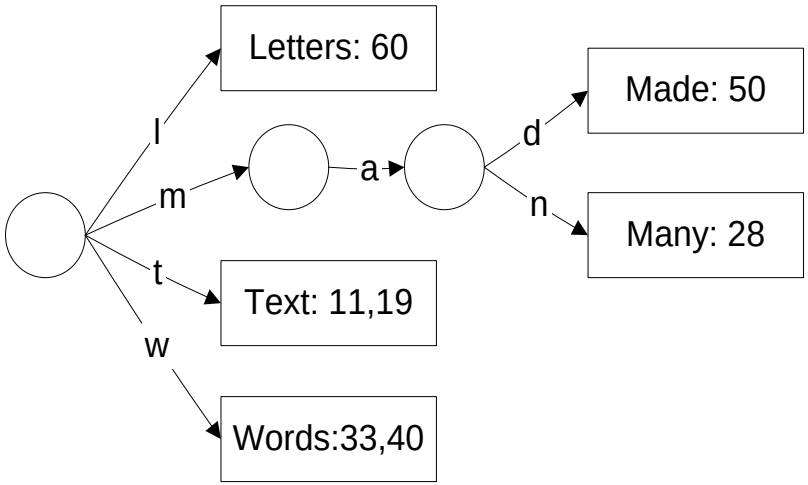
Überblick

- Index-Datenstrukturen
  - Invertierte Indizes
  - Suffix-Bäume und -Arrays
  - Signaturdateien
- Suchalgorithmen
  - Boolesche Suche
  - Sequentielle Suche ohne Indexstruktur
  - String-Matching
  - String-Matching mit Indexstruktur

Invertierter Index
Tries

- Elementare Datenstruktur für Re*trie*val-Aufgaben.
- Kanten sind mit Buchstaben beschriftet.
- Knoten sind leer oder mit Wörtern und Fundstellen beschriftet.

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.



Invertierter Index
Konstruktion eines Tries

- Beginne mit einem leeren Baum und füge iterativ die Wörter in den Trie ein:
- Starte an Wurzel, iteriere über Buchstaben,
  - Wenn Terminalknoten mit gleich Wort erreicht ist, füge Position hinzu, fertig.
  - Wenn Terminalknoten mit anderem Wort erreicht wird, ersetze durch interne Knoten, bis Unterschied zwischen altem und neuem Wort erreicht ist, dann Verzweigung und zwei neue Terminalknoten, fertig.
  - Wenn eine Kante den Buchstaben akzeptiert, folge der Kante.
  - Sonst lege eine neue Kante an, beschrifte mit aktuellem Buchstaben und lege Terminalknoten an, der mit aktuellem Wort und Position beschriftet ist.

Invertierter Index
Was ist das?

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

- Abbildung von Termen auf Dokumente / Textpositionen.
- Oft werden noch Zusatzinformationen gespeichert:
  - Schriftgrad, HTML-Tag

Invertierter Index

- Speicherbedarf für Vokabular: zwischen $n^{0.4}$ und $n^{0.6}$.
- Speicherbedarf für Liste der Vorkommen: zwischen 0.3 x Textlänge und 0.4 x Textlänge.
- Wenn nur Dokumente (nicht Wortpositionen) indiziert werden, 0.2 bis 0.3 x Textlänge.

- **Problem**: Hoher Speicherbedarf.
- **Lösung**: Speicherreduktion durch Blockadressierung: z.B. Aufteilung der Texte in 256 Blöcke, belegt 0.05 x Textlänge.

Invertierter Index
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

- Bestimmung der genauen Textposition mit sequentieller Suche.

Invertierter Index
Aufbau eines invertierten Index

- 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.

Invertierter Index
Große Datensammlungen

- Iteriere über Dokumente, parsiere Tokens (Wörter).
- Füge Tokens mit Textpositionen in unsortierte Liste ein.
- Hänge alle unsortierten Listen aneinander.
- Eliminiere doppelt vorkommende Terme, füge die Vorkommen zu einer Liste zusammen.
- Sortiere die Listen.

- **Voraussetzung**: Liste muss in den Speicher passen.

Invertierter Index
Große Datensammlungen

- 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.

Invertierter Index
Suche

- 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. (Beispiel: Suchmaschiene)

Invertierter Index
Anwendung: Rechtschreibung

- Oft hat genauer Suchbegriff keinen Treffer:
  - Z.B. bei falscher Rechtschreibung.
- Erweitere Suchanfrage um Terme, die Edit-Abstand von höchstens x haben.
- Verlangsamt die Suche sehr. Lösung: nur, wenn ursprüngliche Anfrage keinen Treffer liefert.

Invertierter Index
Kontextanfragen

- Finde Listen der Vorkommen aller Anfrageterme.
  - Listen sind nach Position sortiert.
- Setze Zeiger auf die ersten Listenelemente.
- Wiederhole bis eine der Listen leer ist:
  - Wenn Positionen aller aktuellen Listenelemente hintereinander folgen bzw. dicht genug beieinander liegen, dann haben wir eine Fundstelle.
  - Setze den Listenzeiger, der auf die niedrigste Textposition zeigt, weiter.

- Aufwand: $O(n^{0.4 \ldots 0.8})$

Invertierter Index
Probleme

- Für Nachbarschaftsanfragen nicht so gut geeignet.
- Erfordert Tokenisierung (geht nicht für DNA-Sequenzen).

- Betrachte nicht einzelne Token, sondern alle Suffixe eines Textes.

- Idee von Suffix-Bäumen: Text ist ein einziger String, jede Position ist ein Suffix (von hier bis zum Ende).
- Trie-Struktur über alle Suffixe.

Invertierter Index
Suffix-Trie

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

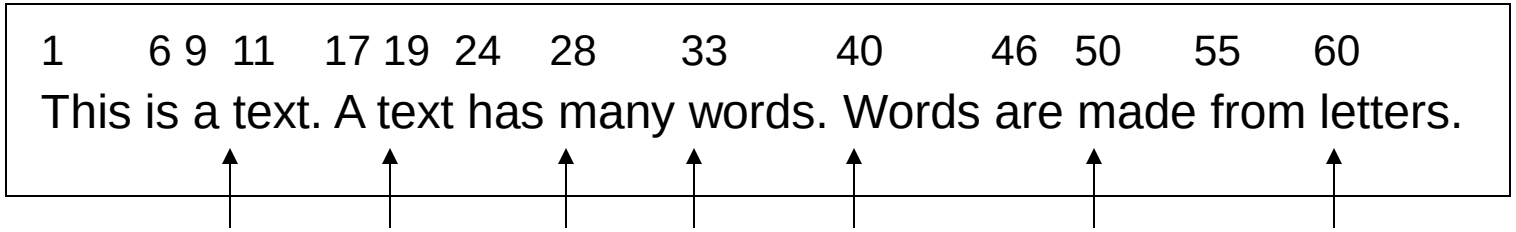
This is a text. A text has many words. Words are made from letters.

Suffixe:

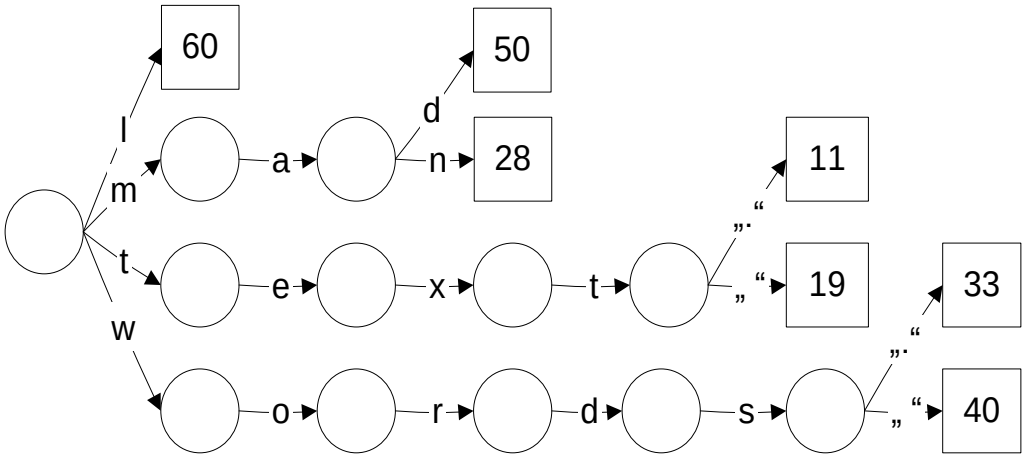
text. A text has many words. Words are made from letters.
text has many words. Words are made from letters.
many words. Words are made from letters.
words. Words are made from letters.
Words are made from letters.
made from letters.
letters.

Invertierter Index
Suffix-Trie

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



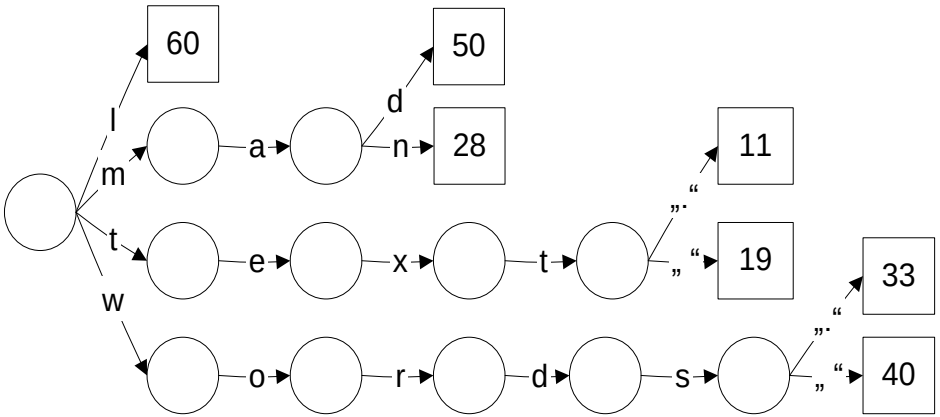
Suffix-Trie:



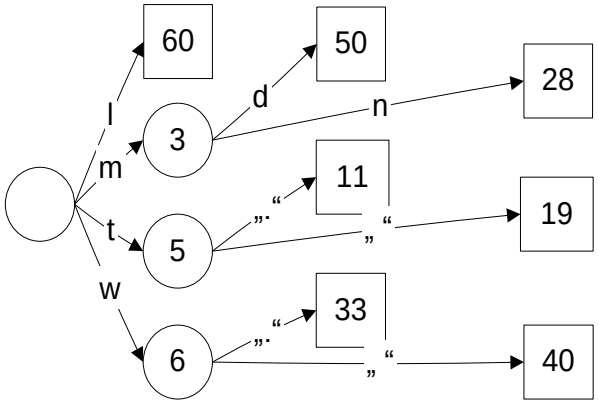
Invertierter Index
Patricia-Tree

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

Suffix-Trie:



Patricia-Tree:



Invertierter Index
Suche im Suffix-Trie

- **Eingabe**: Suchstring, Wurzelknoten.

1. Wiederhole
   1. Wenn Terminalknoten, liefere Position zurück, überprüfe, ob Suchstring an dieser Position steht.
   2. Wenn „Überleseknoten“, springe bis zur angegebenen Textposition weiter.
   3. Wenn „Verzweigungsknoten“, folge der Kante, die den Buchstaben an der aktuellen Position akzeptiert.
2. Bis zum Abbruch in Schritt 1.1.

Invertierter Index
Suffix-Trie

- Konstruktion: O(Länge des Textes).
- Algorithmus funktioniert schlecht, wenn die Struktur nicht in den Speicher passt.

- **Problem**: Speicherstruktur wird ziemlich groß
  - ca. 120-240% der Textsammlung, selbst wenn nur Wortanfänge indexiert werden.

- **Lösung**: Suffix-Arrays
  - Kompaktere Speicherung.

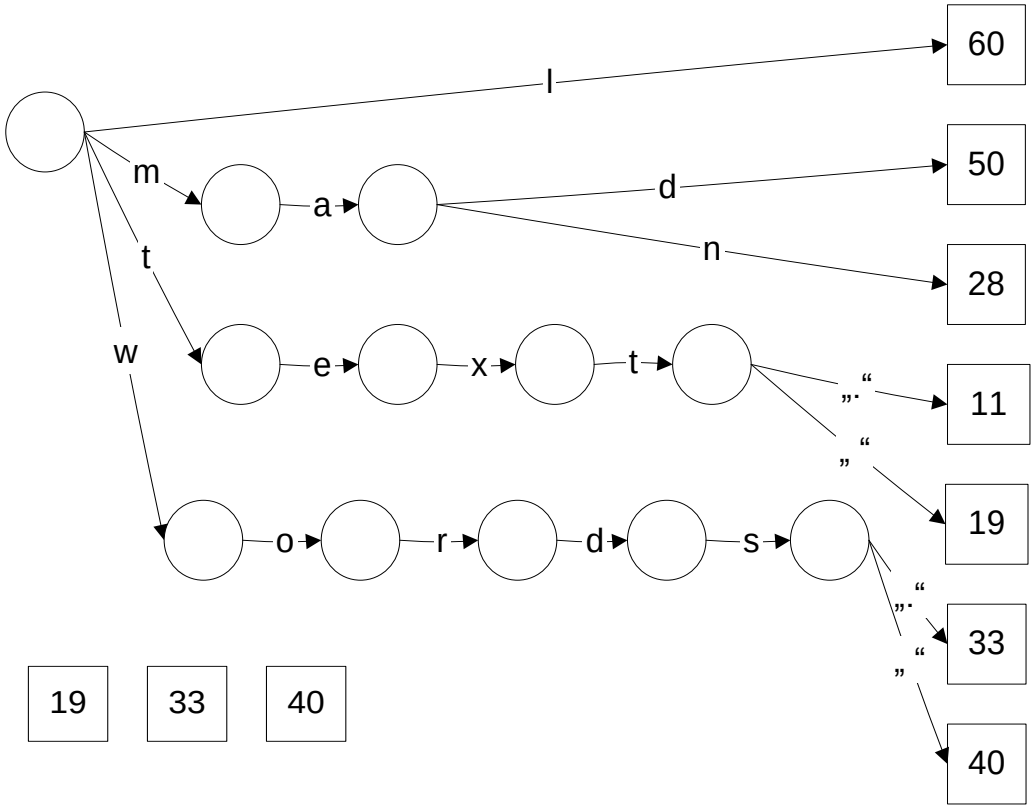
Invertierter Index
Suffix-Array

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

Suffix-Trie,

Lexikographisch

sortiert:



Suffix-Array:

60	50	28	11	19	33	40

Invertierter Index
Suche in Suffix-Arrays: Binärsuche

- Eingabe: Suchstring s, Suffix-Array S

1. A = 1
2. E = # Elemente des Arrays.
3. Wiederhole bis A = E
   1. M = Mittelpunkt (A, E).
   2. Wenn s = Text(S[M]), liefere Textposition
   3. Wenn s < Text(S[M]), E = Mittelpunkt (A, M).
   4. Sonst A = Mittelpunkt (M, E).

- Suffix-Bäume: O(Länge des Suchstrings)
- Suffix-Arrays: O(log(Länge der Textsammlung))

Laufzeit ist schlechter!!

Invertierter Index
Suche in Suffix-Arrays

- Binärsuche: log(Textlänge)
- **Problem**: Viele Datenbankzugriffe.
- **Lösung**: Beschleunigung durch Supra-Index.

- Von jedem k-ten Eintrag werden die ersten Buchstaben in einem Array gespeichert.
- Zuerst Binärsuche in Supra-Index, ohne Datenbankzugriff,
- Danach Binärsuche in Unterabschnitt des Suffix-Arrays mit Datenbankzugriffen.
- Trade-Off zwischen Größe der Suffix-Struktur und Anzahl der Datenbankzugriffe.

Invertierter Index
Suffix-Arrays mit Supra-Index

Supra-Index: Lett text word

Suffix-Array:

60	50	28	11	19	33	40

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

- Speicherbedarf ähnlich wie invertierter Index.

Invertierter Index
Suffix-Arrays für großen Datensammlungen

- Lies Text blockweise ein, bilde lokale Suffix-Arrays, speichere.
- Für alle lokalen Suffix-Arrays:
  - Lies inkrementell den gesamten Text ein,
  - Suche alle Suffixe im lokalen Array, zähle, wie viele Suffixe zwischen den Einträgen „fehlen“. Daraus ergeben sich die Positionen im globalen Suffix-Array, die jeder Eintrag des lokalen Arrays am Ende einnehmen muss. Speichere Position in lokalem Array.
- Merge jetzt die Arrays inkrementell, ohne Zugriff auf den Text, berücksichtige nur auf Grundlage der gespeicherten Positionen.

Sequentielle Suche

- Suche eines Strings der Länge m in Text der Länge n ohne Indexstruktur.
- Brute Force:
  - Für alle Positionen des Textes, iteriere über Suchstring, vergleiche,
    - Suchstring passt: Fundstelle.
    - Unterschied: Vergleich abbrechen.

- O(n x m).
- O(n) möglich?

Knuth-Morris-Pratt

- Algorithmus, um ein Muster in einer Zeichenkette zu finden.
  - Nutzt Struktur des Suchmusters aus, indem teilweise Zeichen nach Missmatch übersprungen werden.
- Besteht aus zwei Teilen:
  - Vorlaufalgorithmus:
    - Bestimmt Präfixtabelle, die angibt, um wie viele Positionen das Muster bei einem Missmatch verschoben werden kann.
  - Suchalgorithmus:
    - Durchsucht Zeichenkette nach dem Muster mithilfe der Präfixtabelle.

Knuth-Morris-Pratt
Vorlaufalgorithmus

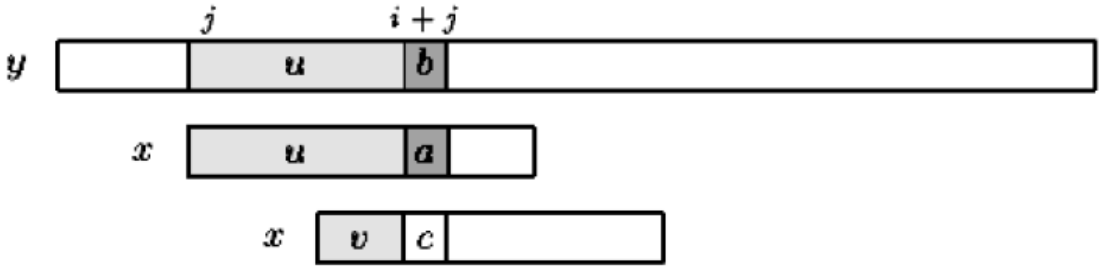
```
kmpProprocess(p)
i = 1; j = 0;
b[0] = 0;
while (i < m) do
   if (p[j] == p[i]) // Match
      b[i] = j + 1;
      i = i + 1; j = j + 1;
   else        // Missmatch
      if (j > 0)
         j = b[j-1];
      else
         b[i] = 0;
         i = i + 1;
```

Beispiel:

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Muster

Präfixarray



- Laufzeit O(m)
- b[j] ist längster Suffix von p[0…j] der Präfix von p[0... m] ist.

Knuth-Morris-Pratt
Suchalgorithmus

```
kmpSearch(t,p)
b = kmpPreprocess(p);
i = 0; j = 0;
while (i < n) do
  if (p[j] == t[i]) // Match
    if (j == m-1)
      return i-m+1;
    i = i+1; j = j+1;
  else              // Mismatch
    if (j > 0)
      j = b[j-1];
    else
      i = i + 1;
```

vorberechnet

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Beispiel:

i	0	1	2	3	4	5	6	7	8	9
t	d	a	d	a	c	d	a	d	a	b
p	d	a	d	a	b					

- Laufzeit O(n)

Knuth-Morris-Pratt
Suchalgorithmus

```
kmpSearch(t,p)
b = kmpPreprocess(p);
i = 0; j = 0;
while (i < n) do
  if (p[j] == t[i]) // Match
    if (j == m-1)
      return i-m+1;
    i = i+1; j = j+1;
  else              // Mismatch
    if (j > 0)
      j = b[j-1];
    else
      i = i + 1;
```

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Beispiel:

i	0	1	2	3	4				
t	d	a	d	a	c	d	a	a	b
p	d	a	d	a	b				

Mismatch bei i = 4 und j = 4:
j = b[4-1] = 2

- Laufzeit O(n)

Knuth-Morris-Pratt
Suchalgorithmus

```
kmpSearch(t,p)
b = kmpPreprocess(p);
i = 0; j = 0;
while (i < n) do
  if (p[j] == t[i]) // Match
    if (j == m-1)
      return i-m+1;
    i = i+1; j = j+1;
  else              // Mismatch
    if (j > 0)
      j = b[j-1];
    else
      i = i + 1;
```

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Beispiel:

i	0	1	2	3	4	5	6	7	8	9
t	d	a	d	a	c	d	a	d	a	b
p	d	a	d	a	b					
			d	a	d	a	b			

- Laufzeit O(n)

Knuth-Morris-Pratt
Suchalgorithmus

```
kmpSearch(t,p)
b = kmpPreprocess(p);
i = 0; j = 0;
while (i < n) do
  if (p[j] == t[i]) // Match
    if (j == m-1)
      return i-m+1;
    i = i+1; j = j+1;
  else              // Mismatch
    if (j > 0)
      j = b[j-1];
    else
      i = i + 1;
```

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Beispiel:

i	0	1	2	3	4
t	d	a	d	a	c
p	d	a	d	a	b
			d	a	d	a	b

Mismatch bei i = 4
und j = 2:
j = b[2-1] = 0

- Laufzeit O(n)

Knuth-Morris-Pratt
Suchalgorithmus

```
kmpSearch(t,p)
b = kmpPreprocess(p);
i = 0; j = 0;
while (i < n) do
  if (p[j] == t[i]) // Match
    if (j == m-1)
      return i-m+1;
    i = i+1; j = j+1;
  else              // Mismatch
    if (j > 0)
      j = b[j-1];
    else
      i = i + 1;
```

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Beispiel:

i	0	1	2	3	4	5	6	7	8	9
t	d	a	d	a	c	d	a	d	a	b
p	d	a	d	a	b					
			d	a	d	a	b			
					d	a	d	a	b	

- Laufzeit O(n)

Knuth-Morris-Pratt - Suchalgorithmus

```
kmpSearch(t,p)
b = kmpPreprocess(p);
i = 0; j = 0;
while (i < n) do
  if (p[j] == t[i]) // Match
    if (j == m-1)
      return i-m+1;
    i = i+1; j = j+1;
  else              // Mismatch
    if (j > 0)
      j = b[j-1];
    else
      i = i + 1;
```

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Beispiel:

i	0	1	2	3	4				
t	d	a	d	a	d				
p	d	a	d	a	b				
			d	a	d	a	b		
					d	a	d	a	b

Mismatch bei i = 4 und j = 0:
i = 5

- Laufzeit O(n)

Knuth-Morris-Pratt - Suchalgorithmus

```
kmpSearch(t,p)
b = kmpPreprocess(p);
i = 0; j = 0;
while (i < n) do
  if (p[j] == t[i]) // Match
    if (j == m-1)
      return i-m+1;
    i = i+1; j = j+1;
  else              // Mismatch
    if (j > 0)
      j = b[j-1];
    else
      i = i + 1;
```

j	0	1	2	3	4
p	d	a	d	a	b
b[j]	0	0	1	2	0

Beispiel:

i	0	1	2	3	4	5	6	7	8	9
t	d	a	d	a	c	d	a	d	a	b
p	d	a	d	a	b					
			d	a	d	a	b			
					d	a	d	a	b	
						d	a	d	a	b

- Laufzeit O(n)

Knuth-Morris-Pratt - Laufzeitanalyse

- Vorlaufalgorithmus hat Laufzeit von O(m).
  - Es werden höchstens 2m -1 Vergleiche durchgeführt.
- Suchalgorithmus hat Laufzeit von O(n).
  - Es werden höchstens 2n-m+1 Vergleiche durchgeführt.
- Da m in der Regel viel kleiner als n, ergibt sich eine Gesamtlaufzeit von O(n).
- Insgesamt werden maximal 2n + m Vergleiche durchgeführt.

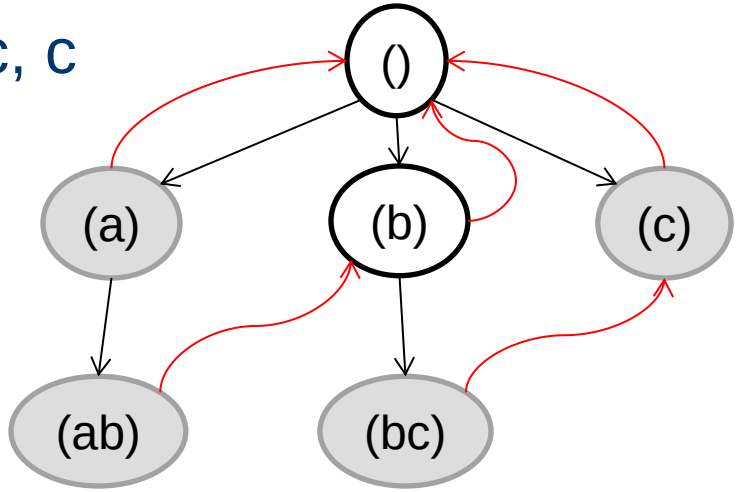
Aho-Corasick-Trie

- Aho-Corasick-Trie
- **Idee**: mehrere Suchstrings
  - Suchstrings werden in Automaten gespeichert,
  - Kanten akzeptieren Buchstaben,
  - Wenn keine passende Kante mehr existiert, Sprung über „failure transition“ in Zustand, der der größten Übereinstimmung zwischen Text und einem Präfix eines Suchstrings entspricht.

Aho-Corasick-Trie - Konstruktion

- 1. Aufbau eines Präfixbaums, der alle Suchwörter enthält.
  - Markiere Zustände, die Suchwörter akzeptieren.
- 2. Konstruiere „failure transitions“:
  - Füge zu jedem Knoten (außer Wurzel) eine failure transition zu dem Knoten hinzu, der längster Suffix des Strings dieses Knotens ist.

Beispiel: a, ab, bc, c

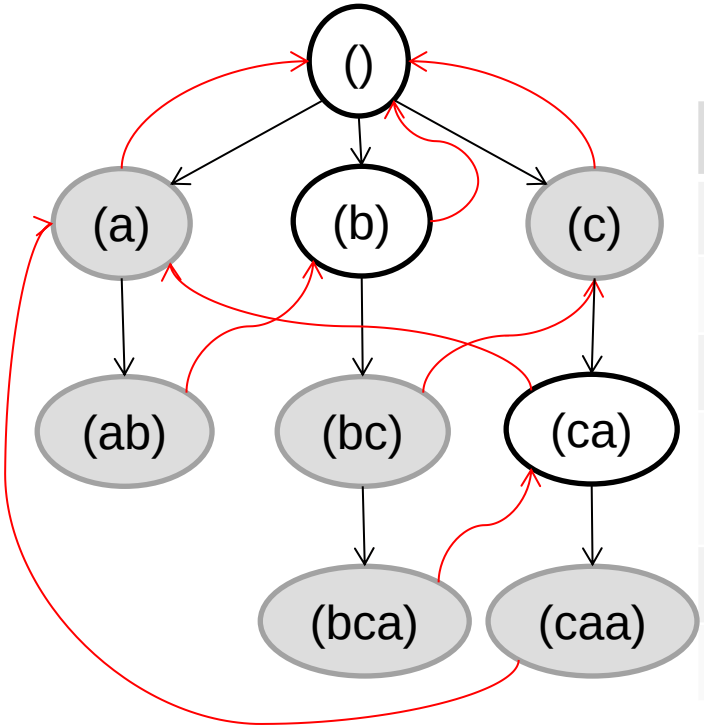


Aho-Corasick-Trie - Suche von Strings

- Starte an Wurzel, iteriere über Buchstaben
  - Folge solange failure transitionen, bis ein Folgezustand erreicht wird, der aktuellen Buchstaben als Suffix enthält, wenn möglich.
  - Wechsel zur Wurzel, sonst.
  - Ausgabe: Gib alle Strings von akzeptierenden Zuständen als akzeptierte Suchstrings aus, die von dem aktuellem Knoten aus durch failure transitionen hin zur Wurzel besucht werden.

Aho-Corasick-Trie – Suche am Beispiel

- Automat akzeptiert Strings a,ab,bc,bca,c,caa



String	Zustand	Folgezustand	Ausgabe
abccab	()	() -> (a)	a:1
bccab	(a)	(a) -> (ab)	ab:2
ccab	(ab)	(ab) -> (b) -> (bc)	c:3, bc:3
cab	(bc)	(bc) -> (c) -> () -> (c)	c:4
ab	(c)	(c) -> (ca)	a:5
b	(ca)	(ca) -> (a) -> (ab)	ab:6

Weitere String-Matching-Algorithmen

- Eigenes, umfangreiches Thema. → Bioinformatik.
- Boyer-Moore, Idee: Strings von hinten nach vorn vergleichen. Dann sind durchschnittlich größere Sprünge möglich.
- Suffix-Automaten.

Approximatives String-Matching

- Suche in y Substrings, die Abstand von höchstens k von x haben.

- Was ist guter Abstand?
  - Edit-Abstand.
  - Vertippmodell.
  - Rechtschreibmodell.
  - Semantisches Modell.

Approximatives String-Matching

- Suche in y Substrings, die Edit-Abstand von höchstens k von x haben.
- Dynamische Programmierung (ähnlich Forward-Backward oder Viterbi).
- Berechnung von C[0…m, 0…n]; C[i,j] = minimale # Fehler beim matchen 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)

Approximatives String-Matching - Beispiel

- $C[0, j] = j$
- $C[i, 0] = i$
- $C[i, j] = \begin{cases} C[i-1, j-1] \text{ , wenn } x[i] = y[j] \\ 1+\min \{C[i-1, j], C[i, j-1], C[i-1, j-1]\} \text{ , sonst} \end{cases}$

		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									

Approximatives String-Matching - Beispiel

- $C[0, j] = j$
- $C[i, 0] = i$
- $C[i, j] = \begin{cases} C[i-1, j-1] \text{ , wenn } x[i] = y[j] \\ 1+\min \{C[i-1, j], C[i, j-1], C[i-1, j-1]\} \text{ , sonst} \end{cases}$

		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									

Approximatives String-Matching - Beispiel

- $C[0, j] = j$
- $C[i, 0] = i$
- $C[i, j] = \begin{cases} C[i-1, j-1] \text{ , wenn } x[i] = y[j] \\ 1+\min \{C[i-1, j], C[i, j-1], C[i-1, j-1]\} \text{ , sonst} \end{cases}$

		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									

Approximatives String-Matching - Beispiel

- $C[0, j] = j$
- $C[i, 0] = i$
- $C[i, j] = \begin{cases} C[i-1, j-1] \text{ , wenn } x[i] = y[j] \\ 1+\min \{C[i-1, j], C[i, j-1], C[i-1, j-1]\} \text{ , sonst} \end{cases}$

		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

Approximatives String-Matching - 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

Löschen

Einfügen

Substitution

= Keine Änderung

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

Weitere Algorithmen zur Mustersuche in Strings

- Endliche Automaten für approximatives String-Matching.
- Suche nach regulären Ausdrücken: Konstruktion eines endlichen Automaten, der die Anzahl der Fehler beim akzeptieren zählt.
  - 1. Umwandlung in NEA mit O(n) Zuständen.
  - 2. Konvertierung des NEA in DEA mittels Potenzmengenkonstruktion mit höchstens O($\sum^n$) Zuständen.
- Laufzeit für das matchen eines Strings der Länge m mit einem DEA ist O(m).

String-Matching mit Index-Strukturen

- Inverse Indizes: wortbasiert. Für Phrasensuche werden alle Suchbegriffe nachgeschlagen, die Texte aneinandergehängt und in den Texten kann dann nach dem Muster gesucht werden.
- Zur approximativen String-Suche müssen alle Wörter mit Höchst-Edit-Abstand im Index nachgesehen werden.

String-Matching in Suffix-Bäumen und Suffix-Arrays

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

Fragen?