Universität Potsdam
Institut für Informatik
Lehrstuhl Maschinelles Lernen

Indexieren und Suchen

Tobias Scheffer

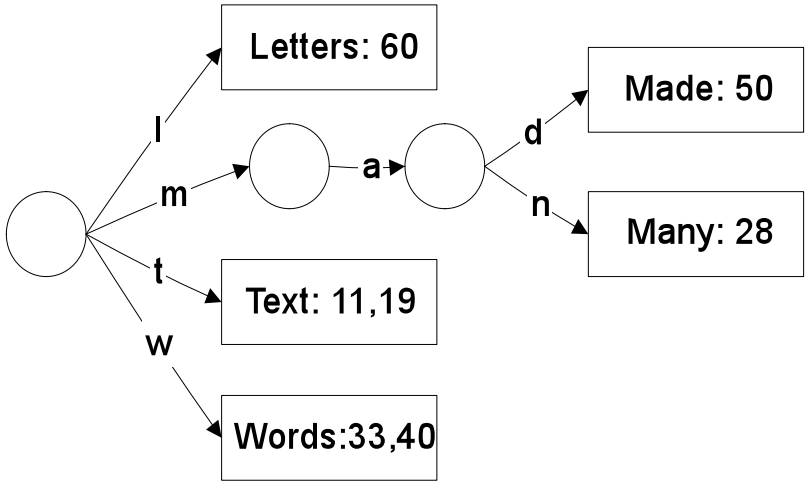
Index-Datenstrukturen, Suchalgorithmen

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

Tries

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



Konstruktion eines Tries

- Einfügen eines Wortes:
- 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.

Invertierte Indizes

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.

Invertierte Indizes

- 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.
- Das ist ziemlich viel!
- Speicherreduktion durch Blockadressierung: z.B. Aufteilung der Texte in 256 Blöcke, funktioniert bis ca. 200 MB Text und belegt 0.05 x Textlänge.

Invertierte Indizes 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

- 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 abspeicher, mit nächstem Unterbaum weitermachen, dazu Text neu iterieren.
- Traversiere den Trie und schreibe die Wörter in den invertierten Index.

Aufbau eines invertierten Index für große Sammlungen

- Iteriere über Dokumente, parsiere 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 für große Sammlungen

- 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 invertierten Indizes

- 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.

Rechtschreibung

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

Kontextanfragen und invertierte Indizes

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

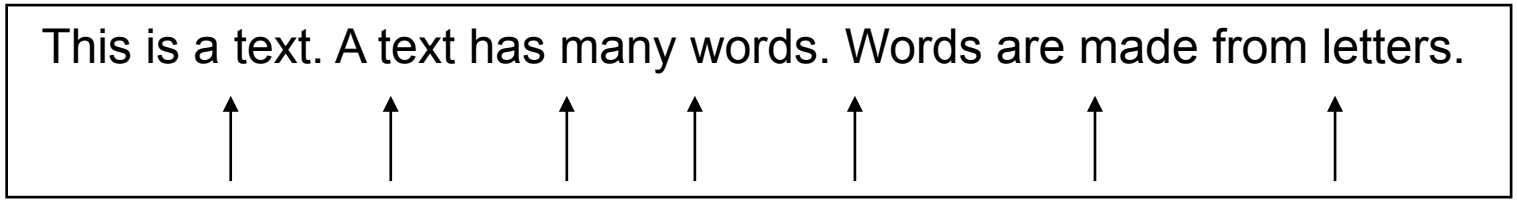
Invertierte Indizes: Probleme

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

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

Suffix-Bäume

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

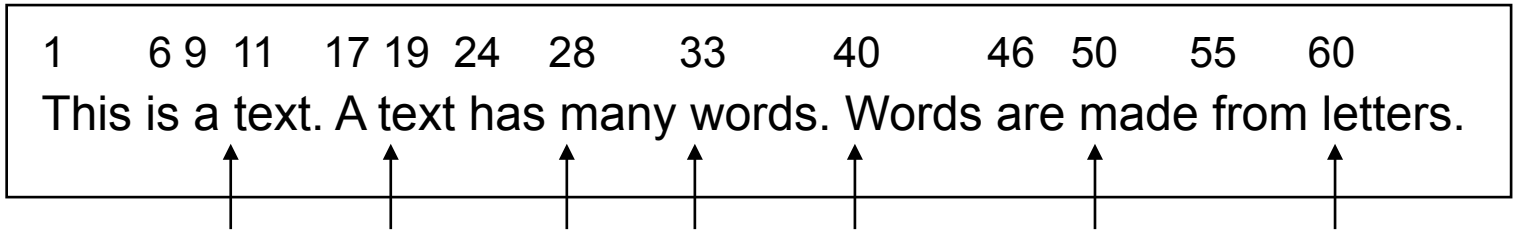


Suffizes:

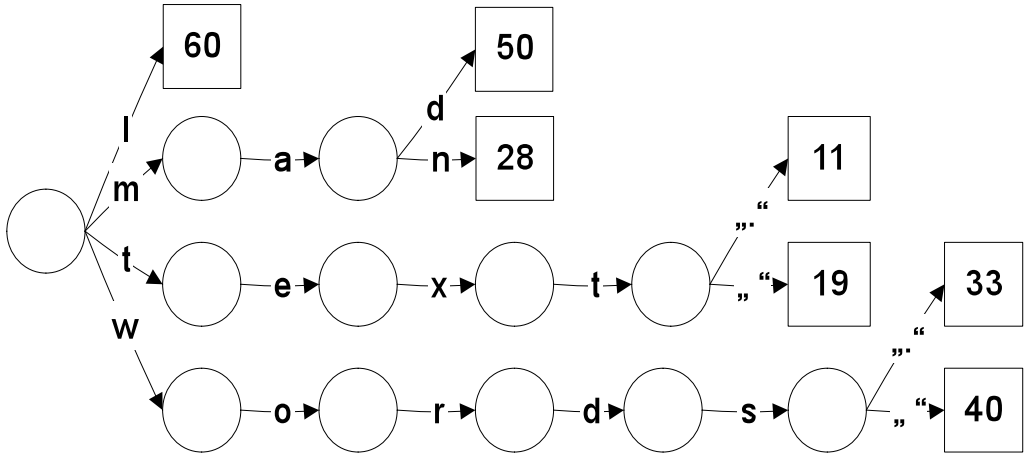
text. A text has many words. Words are made from letters.
text has many words. Words are made from letters.
many words. Words are made from letters.
words. Words are made from letters.
Words are made from letters.
made from letters.
letters.

Suffix-Tries

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



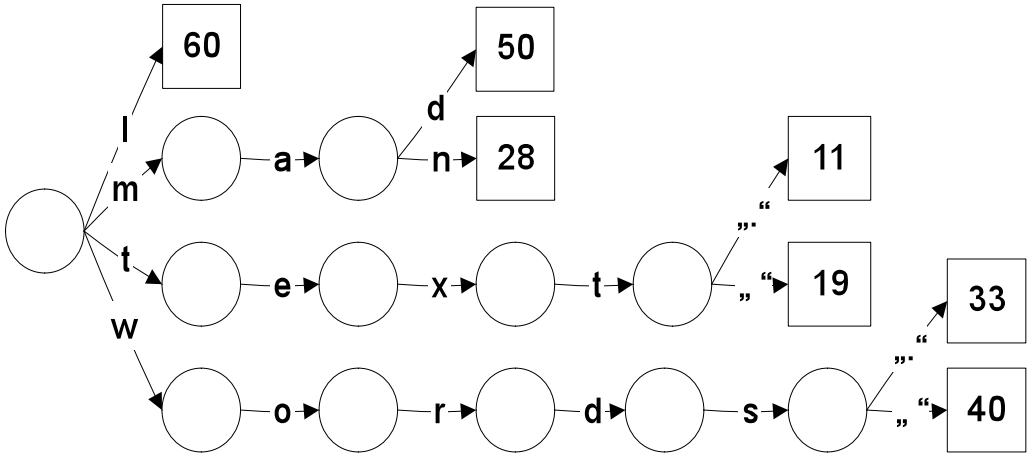
Suffix-Trie:



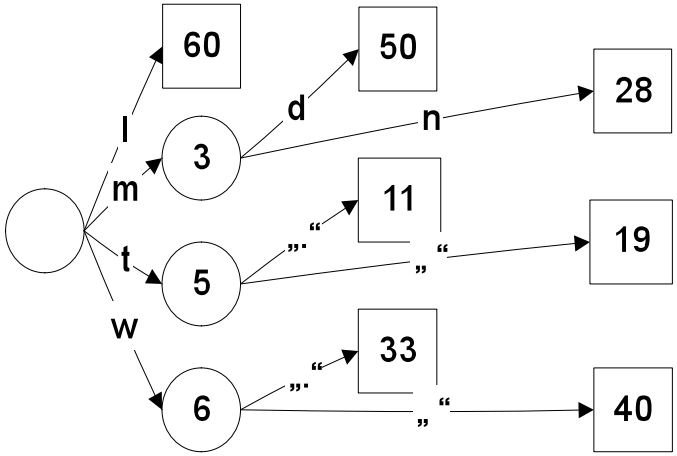
Patricia-Trees

- 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: Suchstring, Wurzelknoten.

1. Wiederhole
    1. Wenn Terminalknoten, liefere Position zurück, überprüfe, ob Suchstring 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.
2. Bis zum Abbruch in Schritt 1.1.

Suffix-Bäume

- 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.

- Suffix-Arrays: kompaktere Speicherung.

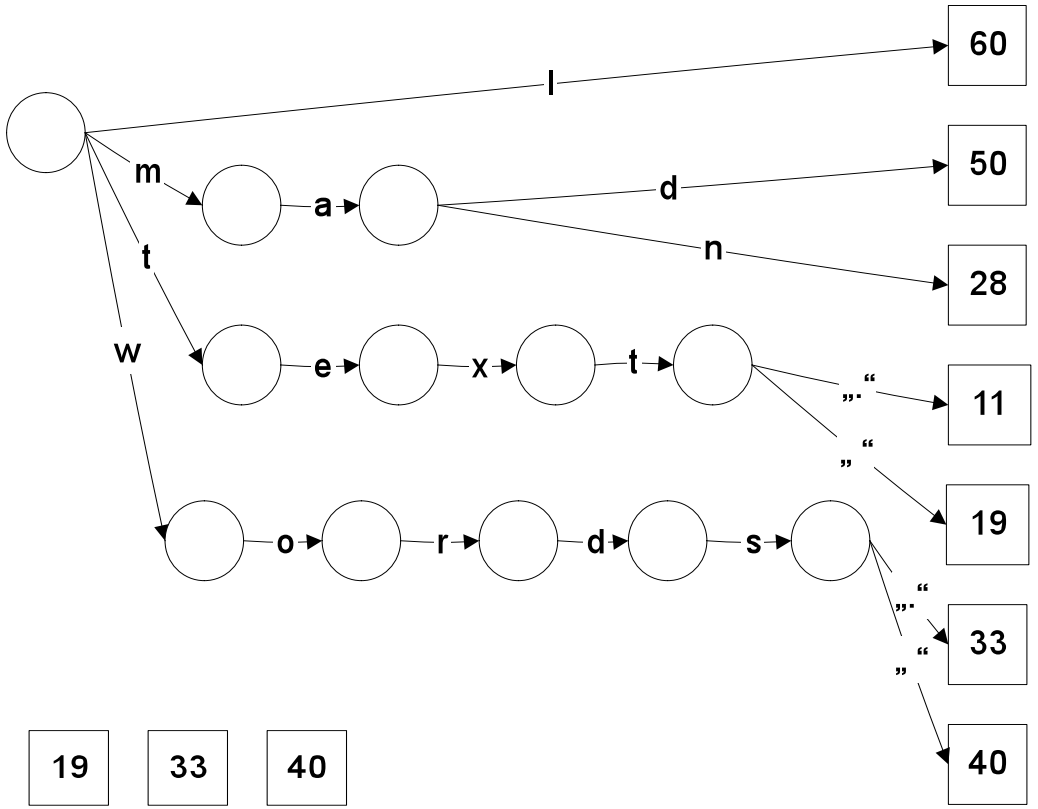
Suffix-Arrays

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

Suffix-Trie,

Lexikographisch

sortiert:



Suffix-Array:

60	50	28	11	19	33	40

Suche in Suffix-Arrays: Binärsuche

- Eingabe: Suchstring s, Suffix-Array A

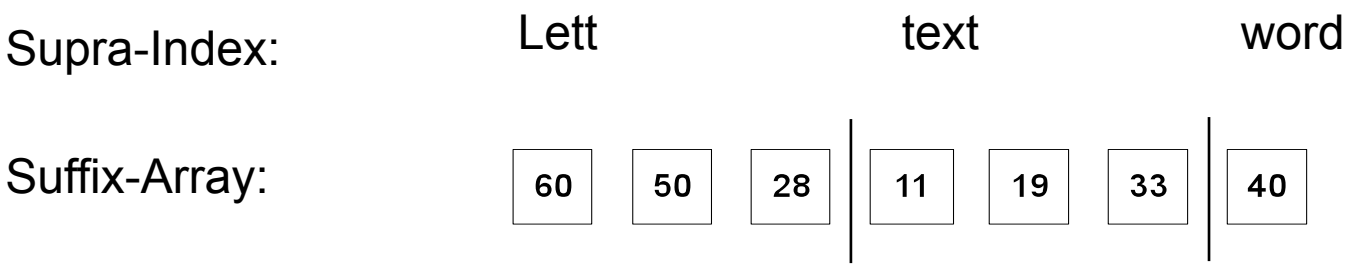
1. A = 1
2. E = # Elemente des Arrays.
3. Wiederhole bis A = E
    1. M = Mittelpunkt (A, E).
    2. Wenn s = Text(A[M]), liefere Textposition
    3. Wenn s < Text(A[M]), E = Mittelpunkt (A, M).
    4. Sonst A = Mittelpunkt (M, E).

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

Suche in Suffix-Arrays

- Binärsuche: log(Textlänge) viele Datenbankzugriffe.
- 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 Datenbankzugriff.
- Trade-Off zwischen Größe der Suffix-Struktur und Anzahl der Datenbankzugriffe.

Suffix-Arrays mit Supra-Index



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

- Speicherbedarf ähnlich wie invertierter Index.

Aufbau von Suffix-Arrays aus großen Sammlungen

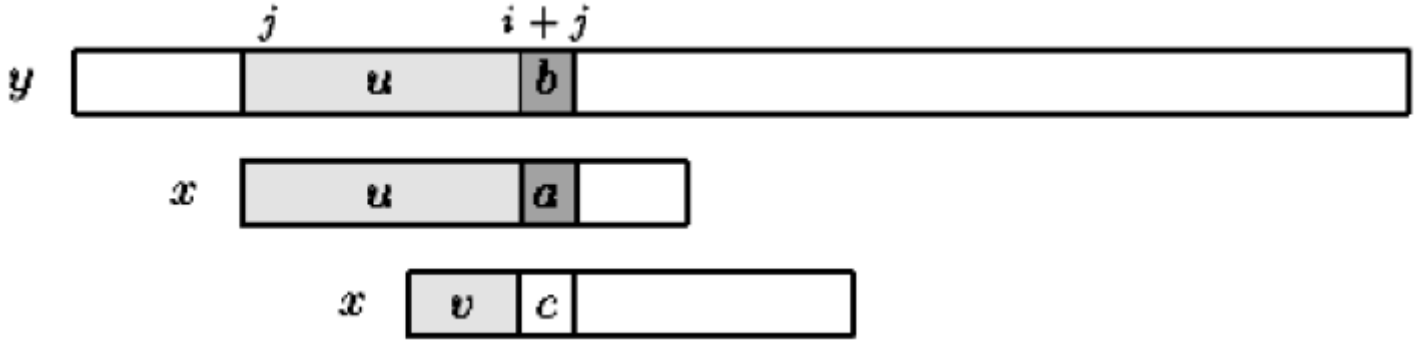
- Lies Text blockweise ein, bilde lokale Suffix-Arrays, speichere.
- Für alle lokalen Suffix-Arrays:
  - Lies inkrementell den gesamten Text ein,
  - Suche alle Suffizes im lokalen Array, zähle, wie viele Suffizes zwischen den Einträgen „fehlen“. Daraus ergeben sich die Positionen im globalen Suffix-Array, 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



- Vergleich von x und y an Fenster-Position y [j…j+m-1].
- y[i+j] ≠ x[i], aber der Substring u hat gepasst.
- Wenn der Anfang von u noch mal in u vorkommt (v = Anfang von u), dort aber mit einem anderen Zeichen weiter geht, dann könnten wir es an dieser Stelle noch mal versuchen.
- Nächster Vergleich zwischen x[kmpNext [i]] und y[i+j].

Knuth-Morris-Pratt

- Demo

The C code

```
void preKmp(char *x, int m, int kmpNext
   int i, j;

   i = 0;
   j = kmpNext[0] = -1;
   while (i < m) {
      while (j > -1 && x[i] != x[j])
         j = kmpNext[j];
      i++;
      j++;
      if (x[i] == x[j])
         kmpNext[i] = kmpNext[j];
      else
         kmpNext[i] = j;
   }
}
```

```
void KMP(char *x, int m, char *y, int n) {
   int i, j, kmpNext[XSIZE];

   /* Preprocessing */
   preKmp(x, m, kmpNext);

   /* Searching */
   i = j = 0;
   while (j < n) {
      while (i > -1 && x[i] != y[j])
         i = kmpNext[i];
      i++;
      j++;
      if (i >= m) {
         OUTPUT(j - i);
         i = kmpNext[i];
      }
   }
}
```

Weitere String-Matching-Algorithmen

- Eigenes, umfangreiches Thema. → Bioinformatik.
- Aho-Corasick-Trie, Idee: mehrere Suchstrings
  - Suchstrings 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 Suchstrings entspricht.
- 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 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] = 0
- C[i, 0] = i
- C[i, j] = Wenn x[i] = y[j], dann C[i-1, j-1]

  Sonst 1+min {C[i-1, j], C[i, j-1], C[i-1, j-1]}
- O (nm)

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.

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.
- Das funktioniert nicht so besonders toll.
- Möglichkeit: Indexieren von n-Grammen (interessant für Sprachen ohne klare Wortgrenzen).

Suffix-Bäume 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