Kapitel 6

Weitere Anfrage-Paradigmen

Viele Folien in diesem Abschnitt sind eine deutsche Übersetzung der Folien von Raymond J. Mooney (http://www.cs.utexas.edu/users/mooney/ir-course/). Ein weiterer großer Anteil wurde mit freundlicher Genehmigung von Peter Becker übernommen (http://www2.inf.fh-rhein-sieg.de/~pbecke2m/retrieval/).

Übersicht

- Boolesche Anfragen
- Näherungsanfragen
- String Matching
- Reguläre Ausdrücke
- Approximatives String Matching
- Phonetische Suche

Boolesche Anfragen

- Schlüsselworte kombiniert mit Booleschen Operatoren:
  - ODER: (e_1 OR e_2)
  - UND: (e_1 AND e_2)
  - ABER: (e_1 BUT e_2) erfüllt e_1 aber **nicht** e_2
- Negation nur eingeschränkt als ABER erlaubt, um den invertierten Index nutzen zu können: effektives Bearbeiten von e_1 durch inv. Index, dann Filtern der Ergebnisse.
- Problem: Untrainierte Anwender haben Probleme mit Boolescher Logik.

Boolsche Retrieval mit invertierten Indizes

- Einzelnes Schlüsselwort: Finde die gesuchten Dokumente durch Verwendung des invertierten Index.
- ODER: Finde auf rekursive Weise e_1 und e_2 bilde die Vereinigung der Ergebnisse.
- UND: Finde auf rekursive Weise e_1 und e_2 und bilde den Durchschnitt der Ergebnisse.
- ABER: Finde auf rekursive Weise e_1 und e_2 und bilde die Mengendifferenz der Ergebnisse.

"Natürlichsprachige" Anfragen

- Volltext-Anfragen als beliebige Strings.
- Typischerweise werden sie nur wie ein "bag of words" im Vektorraum-Modell behandelt und mit Standard-Vektor-Retrieval-Methoden verarbeitet.

Phrasen-Anfragen

- Findet Dokumente, die spezifische Phrasen (geordnete Listen zusammenhängender Wörter) enthalten
  - "information theory"
- Kann intervenierende Stopworte und/oder Stemming zulassen.
  - "buy camera" matches:
    "buy a camera"
    "buying the cameras"
    etc.

Phrasen-Retrieval mit invertierten Indizes

- Muss einen invertierten Index verwenden, der auch die *Positionen* der Schlüsselworte in den Dokumenten speichert.
- Finde Dokumente und Positionen für jedes individuelle Wort, bilde Durchschnitt der Treffermengen, und prüfe abschließend auf den richtigen Zusammenhang der Schlüsselwort-Positionen.
- Am besten beginnt man die Prüfung der Zusammenhänge mit dem am wenigsten häufigen Wort in der Phrase.

Phrasensuche

Aufgabe: Finde die Menge D von Dokumenten, bei denen alle Schlüsselworte ($k_1 \ldots k_m$) in einer Phrase vorkommen (verwende UND-Anfrageverarbeitung).

Intitialisiere eine leere Menge, R, von gefundenen Dokumenten.
Für jedes Dokument d in D:
  Suche für jedes k_i die Folge der Positionen P_i der Vorkommen in d
  Bestimme die kürzeste Folge P_s der P_i's
  Für jede Position p des Schlüsselworts k_s in P_s
    Für jedes Schlüsselwort k_i außer k_s
      Verwende die binäre Suche, um eine Position ($p - s + i$) in der Folge P_i zu finden.
    Falls für jedes Schlüsselwort die korrekte Position gefunden wurde, füge d zu R hinzu
Gib R aus.

Näherungsanfragen

- Liste von Termen mit zusätzlichen maximalen Abstandsbeschränkungen zwischen den Termen.
- Beispiel: “dogs” und “race” in 4 Worten passt zu “…dogs will begin the race…”
- Man kann zusätzlich Stemming verwenden und/oder Stopwörter ignorieren.

Näherungs-Retrieval mit invertiertem Index

- Benutze Ansatz ähnlich wie bei Phrasensuche, um alle Dokumente zu finden, die die Schlüsselwörter enthalten.
- Finde bei der binären Suche nach den Positionen der verbleibenden Schlüsselwörter die nächste Position von k_i nach p und überprüfe, dass diese innerhalb des maximal zulässigen Abstands ist.

String-Matching

- Erlaubt allgemeinere Stringvergleiche in Anfragen, während der vorher diskutierte Ansatz die Worte als atomare Einheiten (Tokens) betrachtet.
- Erfordert für effiziente Bearbeitung komplexere Datenstrukturen und Algorithmen als invertierte Indizes.

String-Matching

- Die Suche von einem Muster in einem Text wird auch als *String Matching* oder *Pattern Matching* bezeichnet.
- Generell besteht die Aufgabe darin, einen String (das *Muster, Pattern*) der Länge m in einem Text der Länge n zu finden, wobei $n > m$ gilt.
- Je nach Freiheitsgraden bei der Suche unterscheidet man verschiedene Arten von String-Matching-Problemen.

Arten von String-Matching-Problemen

- **Exaktes String-Matching:** Wo tritt ein String *pat* in einem Text *text* auf? Beispiel: fgrep
- **Matching von Wortmengen:** Gegeben sei eine Menge *S* von Strings. Wo tritt in einem Text ein String aus *S* auf? Beispiel: agrep
- **Matching regulärer Ausdrücke:** Welche Stellen in einem Text passen auf einen regulären Ausdruck? Beispiel: grep, egrep
- **Approximatives String-Matching:** Welche Stellen in einem Text passen am besten auf ein Muster (Best-Match-Anfrage)?

Arten von String-Matching-Problemen

- Welche Stellen in einem Text stimmen mit einem Muster bis auf *d* Fehler überein (Distance-Match-Anfrage)? Beispiel: agrep
- **Editierdistanz:** Wie kann man am "günstigsten" einen String *s* in einen String *t* überführen? Beispiel: diff

Einfache Strukturen

- Prefix: Struktur, die zum Wortanfang passt.
  - "anti" passt zu "antiquity", "antibody", etc.
- Suffix: Struktur, die zum Wortende passt:
  - "ix" passt zu "fix", "matrix", etc.
- Substring: Struktur, die zu einer willkürlichen Teilfolge von Zeichen passt.
  - "rapt" passt zu "enrapture", "velociraptor" etc.
- Range: Stringpaar, das zu jedem Wort passt, das lexikographisch (alphabetisch) dazwischen liegt.
  - "tin" to "tix" passt zu "tip", "tire", "title", etc.

Anwendungen

- Wo braucht man String-Matching-Verfahren? Z.B.:
  - Volltextdatenbanken
  - Retrievalsysteme
  - Suchmaschinen
  - Bioinformatik
- In diesem Abschnitt lernen wir effiziente Algorithmen für String-Matching kennen.

Bezeichnungen

- Ein *Alphabet* ist eine endliche Menge Σ von Symbolen. $|\Sigma|$ bezeichnet die Kardinalität von Σ.
- Ein *String (Zeichenkette,Wort) s* über einem Alphabet ist eine endliche Folge von Symbolen aus Σ. $|s|$ bezeichnet die *Länge* von *s*.
- ε bezeichnet den *leeren String*.
- Wenn *x* und *y* Strings sind, dann bezeichnet *xy* die *Konkatenation* von *x* und *y*.

Bezeichnungen

- $s[i]$ bezeichnet das i-te Element eines Strings s ($1 \leq i \leq |s|$).
- $s[i...j]$ bezeichnet den String $s[i]s[i+1]...s[j]$.
- Für $i > j$ gelte $s[i...j] = \varepsilon$
- Für einen String s (mit $m = |s|$) bezeichnet s^{-1} die *Umkehrung* $s[m]s[m-1] ...s[1]$ von s.
- Für zwei Strings x und y gilt $x = y$ genau dann, wenn $|x| = |y| = m$ und $x[i] = y[i]$ f ür alle $1 \leq i \leq m$ gilt.
- Wenn $w = xyz$ ein String ist, dann ist x ein *Präfix* und z ein *Suffix* von w.
- Gilt $w \neq x$ ($w \neq z$), dann ist x (z) ein *echter Präfix (echter Suffix)* von w.

- Ein String x (mit $m = |x|$) heißt *Substring (Faktor)* von y, wenn ein i existiert mit $x = y[i...i+m-1]$. Andere Sprechweisen: x tritt in y an Position i auf bzw. Position i ist ein *Match* für x in y.
- x (mit $m = |x|$) heißt *Subsequenz* von y, wenn Positionen $i_1 < i_2 < ... < i_m$ existieren mit $x = y[i_1]y[i_2]...y[i_m]$.

Exaktes String-Matching

- **Problem 6.1. [Exaktes String-Matching]** Gegeben sind die Strings *pat* und *text*.
  (a) Man bestimme, ob *pat* ein Substring von *text* ist.
  (b) Man bestimme die Menge aller Positionen, an denen *pat* in *text* auftritt. Diese Menge wird mit MATCH(*pat*; *text*) bezeichnet.
- Im folgenden wird nur die Variante (a) von Problem 6.1 betrachtet.
- Algorithmen für die Variante (b) erhält man durch einfache Modifikationen der Algorithmen für (a).
- Im folgenden sei $m = |pat|$ und $n = |text|$.

Naiver Ansatz

- Der naive Ansatz besteht darin, für jede Position von *text* (bzw. solange *pat* ab der aktuellen Position in *text* passt) von neuem zu testen, ob *pat* an dieser Position auftritt.
- Das allgemeine Schema für solch einen naiven Algorithmus lautet:

- **for** $i := 1$ **to** n-m+1 **do**
  man prüfe, ob $pat = text[i...i+m-1]$ gilt

- Die Prüfung kann nun "von links nach rechts" oder "von rechts nach links" erfolgen.
- Dies führt zu unterschiedlichen naiven Algorithmen und darauf aufbauend zu unterschiedlichen Ansätzen der Verbesserung:
  - „von links nach rechts" → Algorithmus von Morris und Pratt
  - „von rechts nach links" → Algorithmus von Boyer und Moore
- Wir betrachten hier nur die zweite Variante.

Der Algorithmus von Boyer und Moore

- Der Algorithmus von Boyer und Moore kann als eine verbesserte Variante eines naiven String-Matching-Algorithmus angesehen werden, bei dem *pat* mit *text* von rechts nach links verglichen wird.

- **Algorithmus 6.1 [naives String-Matching von rechts nach links]**

```
i := 1
while i ≤ n-m+1 do
    j := m
    while j ≥ 1 and pat[j] = text[i+j-1] do j := j-1 end
    if j = 0 then return true
    i := i + 1
    end
return false
```


Analyse des naiven Algorithmus

- **Satz 6.1.** *Der naive Algorithmus 6.1 löst Problem 6.1 in Zeit* $O(nm)$ *und Platz* $O(m)$.

- Für $pat = ba^{m-1}$ und $text = a^n$ benötigt Algorithmus 6.1 $(n-m+1)m = nm - m^2 + m$ Zeichenvergleiche. (Dies ist der ‚worst case'.)

- Bei einem binären Alphabet und zufällig erzeugten *pat* und *text* (jedes Zeichen unabhängig und jedes Symbol mit Wahrscheinlichkeit 1/2) ergibt sich für die durchschnittliche Anzahl an Zeichenvergleichen: $(2-2^{-m})n + O(1)$

Analyse des naiven Algorithmus

- Trotz der im Durchschnitt linearen Laufzeit lohnt sich der Einsatz von "besseren" String-Matching-Algorithmen, denn:
  - die nachfolgenden String-Matching-Algorithmen haben sich nicht nur in der Theorie, sondern auch in der Praxis als erheblich effizienter erwiesen, und
  - die Realität gehorcht nicht immer den Gesetzen der Wahrscheinlichkeitstheorie.

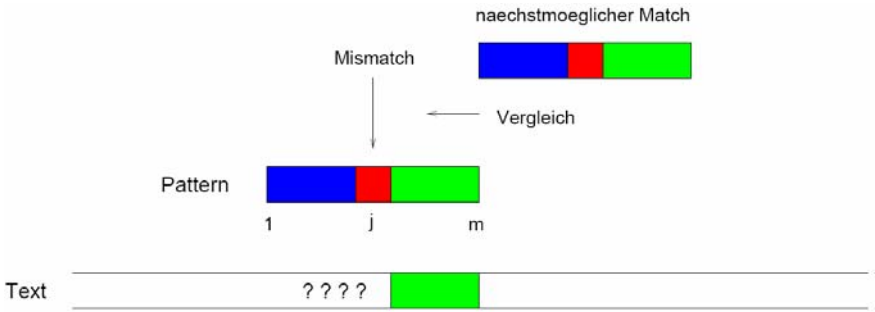
Der Algorithmus von Boyer und Moore

Der Algorithmus von Boyer und Moore basiert auf der folgenden Überlegung:

- Tritt in Algorithmus 6.1 an Stelle j ein Mismatch auf und kommt $pat[j+1...m]$ nicht ein weiteres mal in pat als Substring vor, so kann pat gleich um m Zeichen nach rechts verschoben werden.

- (Vergleicht man dagegen von links nach rechts, kann pat nach einem Mismatch an Position j nie um mehr als j Positionen nach rechts verschoben werden. Dies ist das Prinzip des Algorithmus von Morris und Pratt).

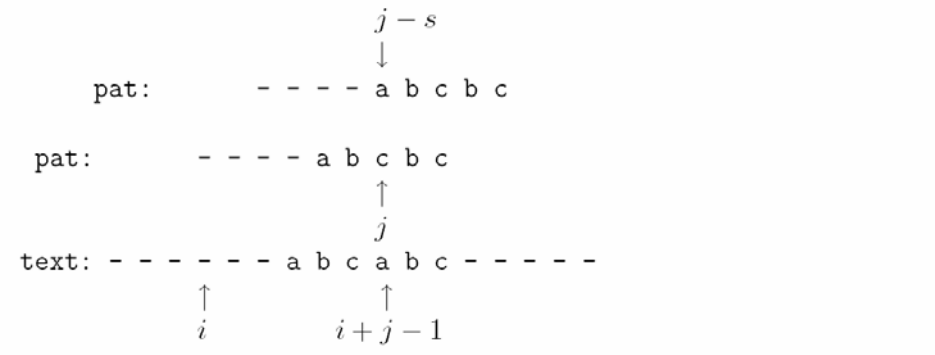
Veranschaulichung



- Kommt es in Algorithmus 6.1 an Stelle j von pat zu einem Mismatch, so gilt
  $pat[j+1...m] = text[i+j \ldots i+m-1]$ und
  $pat[j] \neq text[i+j-1]$.

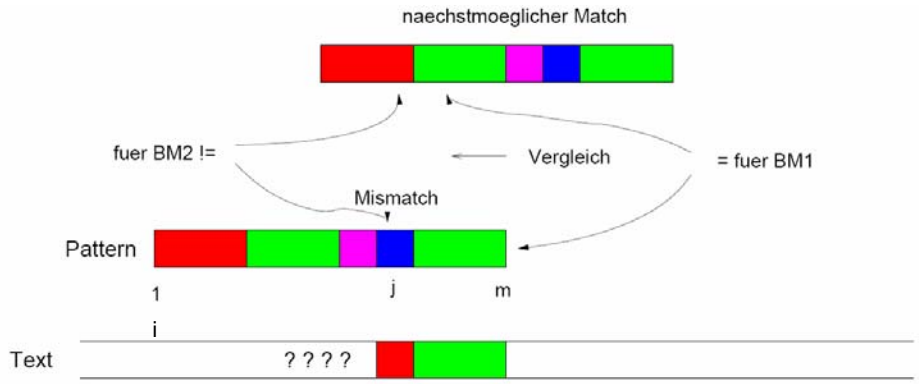
- Dies kann wie folgt ausgenutzt werden: Angenommen, pat tritt in $text$ an einer Position $i+s$ mit $i < i+s < i+m$ auf. Dann müssen die beiden folgenden Bedingungen gelten:
  (BM1) $\forall j < k \leq m$: $k \leq s \ \vee \ pat[k-s] = pat[k]$
  (BM2) $s < j \Rightarrow pat[j-s] \neq pat[j]$

Veranschaulichung



Fazit: Wir brauchen also erst wieder bei einer Stelle zu testen, bei der die beiden Bedingungen gelten. Damit man keinen Match verpasst, muss s möglichst klein gewählt werden.

Veranschaulichung



Wdh.: Angenommen, es gibt einen Mismatch an Stelle j von *pat*, und *pat* tritt in *text* an einer Position $i+s$ mit $i < i+s < i+m$ auf. Dann gelten die beiden folgenden Bedingungen:

(BM1) $\forall j < k \leq m$: $k \leq s \ \vee \ pat[k\text{-}s] = pat[k]$

(BM2) $s < j \rightarrow \ pat[j\text{-}s] \neq pat[j]$

Forts. Bayer-Moore-Algorithmus

- Die entsprechenden Werte werden in einer Preprocessingphase ermittelt und in der *Shift-Tabelle D* abgelegt. (Hierzu kann wiederum eine Variante des Boyer-Moore-Algorithmus genutzt werden.)

  $D[j] := \min_{s>0}\{s \mid$ (BM1) und (BM2) gilt für j und $s\}$

- Der Algorithmus von Boyer und Moore verwendet nun im Falle eines Mismatches an Position j den in $D[j]$ abgelegten Wert, um *pat* nach rechts zu verschieben.

Algorithmus 6.2 [Algorithmus von Boyer und Moore]

$i := 1$

while $i \leq n - m + 1$ **do**

  $j := m$

  while $j \geq 1$ **and** $pat[j] = \text{text}[i + j - 1]$ **do**

    $j := j - 1$ **end**

  if $j = 0$ **then return true**

  $i := i + D[j]$

end

return false

Beispiel

Definition 6.1 Der n-te *Fibonacci-String* Fn ($n \geq 0$) ist wie folgt definiert:

- $F_0 = \varepsilon$
- $F_1 = b$
- $F_2 = a$
- $F_n = F_{n-1}F_{n-2}$ für $n > 2$

- $D[j]$ lautet für F_7:

j	1	2	3	4	5	6	7	8	9	10	11	12	13
$pat[j]$	*a*	*b*	*a*	*a*	*b*	*a*	*b*	*a*	*a*	*b*	*a*	*a*	*b*
$D[j]$	8	8	8	8	8	8	8	3	11	11	6	13	1

Weitere Verbesserung von Boyer-Moore

Algorithmus 6.2 kann noch weiter verbessert werden: Tritt an Stelle m von *pat* (also bereits beim ersten Vergleich) ein Mismatch auf, so wird *pat* momentan nur um eine Stelle nach rechts verschoben.

Es sei

$$last[c] := \max_{1 \le j \le m} \{j \mid pat[j] = c\}$$

und $last[c] := 0$ falls c nicht in *pat* auftritt.

$last[c]$ gibt für ein $c \in \Sigma$ die jeweils letzte Position von c in *pat* an. Kommt es nun an Stelle j zu einem Mismatch, kann statt

$$i := i + D[j]$$

die Anweisung

$$i := i + \max(D[j]; j - last[text[i + j - 1]])$$

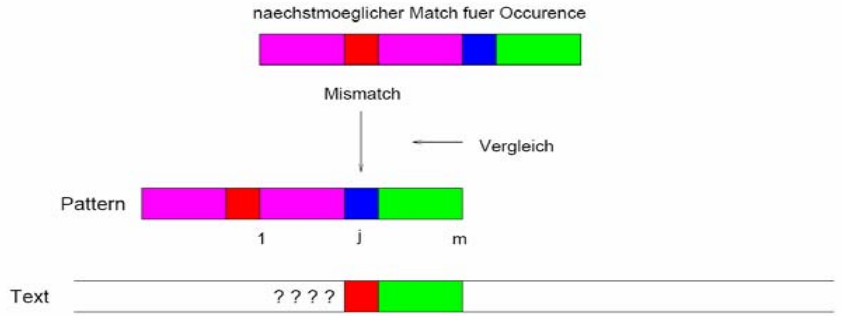
verwendet werden. Damit ergeben sich noch größere Verschiebungen.

Bemerkungen

- Verschiebungen der Länge $j - last[text[i+j-1]]$ heißen *Occurrence-Shift*.
- Wird nur der Occurence-Shift verwendet, d.h. die Verschiebe-Anweisung lautet
  $i := i + \max(1, j - last[text[i + j - 1]])$
  so spricht man von einem *vereinfachten Boyer-Moore-Algorithmus*.
- DieWorst-Case-Laufzeit des vereinfachten Boyer-Moore-Algorithmus beträgt $O(nm)$.

Bemerkungen

- Auf gewöhnlichen Texten verhält sich die vereinfachte Version i.d.R. nur marginal schlechter als die ursprüngliche Version.
- Bei kleinem $|\Sigma|$ ist die Occurence-Heuristik i.d.R. nutzlos.



Beispiel: Algorithmus von Boyer und Moore

Der String F_7 wird in dem String abaababaabacabaababaabaab gesucht.

```
                    a b a a b a b a a b a a b
              a b a a b a b a a b a a b
          a b a a b a b a a b a a b
  a b a a b a b a a b a a b
a b a a b a b a a b a a b

a b a a b a b a a b a c a b a a b a b a a b a a b
```

Beispiel [Shift-Tabellen für Boyer/Moore]

```
datenbank       retrieval          compiler
999999991       999999991          88888881

kuckuck      rokoko   papa     abrakadabra
3336661      662641   2241     77777771131
                                        00
```

Satz 4.5. *Algorithmus 6.2 löst Problem 6.1(a) in Zeit $O(n + m)$ und Platz $O(m)$.*

Bemerkungen

- Als scharfe obere Schranke für die Anzahl an Zeichenvergleichen ergibt sich $3n$.
- Würde man statt (BM1) und (BM2) nur (BM1) verwenden, so wäre keine lineare Laufzeit mehr gewährleistet ($O(mn)$).
- Sucht man mit dem Algorithmus von Boyer und Moore nach allen Matches für *pat* in *text*, so ist die Laufzeit ebenfalls $O(mn)$.

Reguläre Ausdrücke

- Sprache zur Bildung von komplexeren Such-Mustern:
  - Ein individuelles Zeichen ist ein regex (regulärer Ausdruck) .
  - Vereinigung: Wenn e_1 und e_2 regexes sind, dann ist $(e_1 \mid e_2)$ ein regex, der zu allem passt, was zu e_1 oder zu e_2 passt.
  - Verknüpfung: Wenn e_1 und e_2 regexes sind, dann ist $e_1\, e_2$ ein Regex, der zu einem String passt, der aus einem Substring besteht, der zu e_1 passt, sofort gefolgt von einem Substring, der zu e_2 passt.
  - Wiederholung (Kleene-Abschluss): Wenn e_1 ein regex ist, dann ist e_1* ein regex, der zu einer Folge von null oder mehr Strings passt, die zu e_1 passen.

Beispiele regulärer Ausdrücke

- (u|e)nabl(e|ing) passt zu
  - unable
  - unabling
  - enable
  - enabling
- (un|en)*able passt zu
  - able
  - unable
  - unenable
  - enununenable
  - ...

Erweiterte Regex's (Perl)

- Perl enthält spezielle Terme für bestimmte Zeichentypen, wie alphabetische oder numerische oder allgemeine "Wildcards".
- Spezieller Wiederholungsoperator (+) für 1 oder mehrere Vorkommen.
- Spezieller optionaler Operator (?) für 0 oder 1 Vorkommen.
- Spezieller Wiederholungsoperator für spezifische Anzahl von Vorkommen : {min,max}.
  - A{1,5} - ein bis fünf A's.
  - A{5,} - fünf oder mehr A's
  - A{5} - genau fünf A's

Perl

- Zeichenklassen:
  - \w (word char) jedes alpha-numerische Zeichen (Negation: \W)
  - \d (digit char) jede Zahl (Negation: \D)
  - \s (space char) jeder Zwischenraum (Negation: \S)
  - . (wildcard) alles
- Ankerpunkte:
  - \b (boundary) Wortgrenze
  - ^ Beginn eines Strings
  - $ Ende eines Strings

Perl-Regex-Beispiele

- U.S. Tel.Nr. ohne optionalen Bereichscode:
  - /\b(\(\d{3}\)\s?)?\d{3}-\d{4}\b/
- (amer.) Email-Addresse:
  - /\b\S+@\S+(\.com|\.edu|\.gov|\.org|\.net)\b/

Prinzip und Implementierung

- Reguläre Ausdrücke werden mit endlichen Automaten analysiert. (Siehe Vorlesung "Theoretische Informatik I")
- Pakete zur Unterstützung von Perl regex's sind in Java verfügbar.

Approximatives String-Matching

- Was ist, wenn ein Dokument Tippfehler oder falsche Buchstaben enthält?
- Ähnlichkeitsmaße zwischen Wörtern (beliebigen Strings):
  - Editier-Abstand (Levenstein distance)
  - Längste gemeinsame Teilfolge (longest common substring, LCS)
- Suche alle Strings, die näher sind als ein vorgebener Schwellwert.

Approximatives String-Matching

- Bisher haben wir String-Matching-Probleme betrachtet, bei denen das Muster *pat* exakt mit einem Substring von *text* übereinstimmen musste.
- Beim Matching von regulären Ausdrücken lässt man zwar Varianten zu, aber ebenfalls keine Fehler.
- In vielen praktischen Fällen ist es wünschenswert, die Stellen von *text* zu finden, die mit *pat* "nahezu" übereinstimmen, d.h. man erlaubt Abweichungen zwischen *pat* und *text*.

Anwendungsbeispiele

- Molekularbiologie (Erkennung von DNA-Sequenzen)
- Ausgleich verschiedener Schreibweisen (Grafik vs. Graphik)
- Ausgleich von Beugungen
- Toleranz gegenüber Tippfehlern
- Toleranz gegenüber OCR-Fehlern

String-Metriken

- Der Begriff "nahezu" wird durch eine Metrik auf Strings formalisiert.
- Zur Erinnerung: Sei M eine Menge. Eine Funktion $d : M \times M \rightarrow \mathbb{R}$ heißt *Metrik*, wenn die folgenden Bedingungen erfüllt sind:
  - $d(x,y) \geq 0$ für alle $x,y \in M$
  - $d(x,y) = 0 \Leftrightarrow x = y$ für alle $x, y \in M$
  - $d(x,y) = d(y,x)$ für alle $x,y \in M$
  - $d(x,z) \leq d(x,y) + d(y,z)$ für alle $x.y,z \in M.$

  (M,d) ist dann ein *metrischer Raum*.

String-Metriken

Problem 6.2. Gegeben seien ein String *pat*, ein String *text*, eine Metrik *d* für Strings und ein ganze Zahl $k \geq 0$. Man finde alle Substrings *y* von text mit $d(pat, y) \leq k$.

Bemerkungen:

- Für $k = 0$ erhält man das exakte String-Matching Problem.
- Problem 4.2 ist zunächst ein "abstraktes" Problem, da nichts über die Metrik *d* ausgesagt wird.
- Zur Konkretisierung von Problem 6.2 und zur Entwicklung von entsprechenden Algorithmen müssen zunächst sinnvolle Metriken betrachtet werden.

Längste gemeinsame Teilfolge (LCS)

- Länge der längsten Teilfolge von Zeichen, die zwei Strings gemeinsam ist.
- Eine *Teilfolge* eines String wird durch das Löschen von null oder mehreren Zeichen erreicht.
- Beispiele:
  - "misspell" to "mispell" is 7
  - "misspelled" to "misinterpretted" is 7
    "mis…p…e…ed"

Hamming-Distanz

Definition 6.2. Für zwei Strings *x* und *y* mit $|x| = |y| = m$ ergibt sich die *Hamming-Distanz (Hamming Distance)* durch:

$$d(x, y) = |\{1 \leq i \leq m | x[\mathrm{i}] \neq y[i]\}|$$

Bemerkungen:
Die Hamming-Distanz ist die Anzahl der Positionen, an denen sich *x* und *y* unterscheiden. Sie ist nur für Strings gleicher Länge definiert. Wird in Problem 6.2 für *d* die Hamming-Distanz verwendet, so spricht man auch von "string matching with *k* mismatches".

Beispiel: *Die Hamming-Distanz der Strings* abcabb *und* cbacba *beträgt* 4.

Editier- (Levenstein-)Abstand

- Die minimale Anzahl von *Löschungen*, *Hinzufügungen* und *Änderungen* von Zeichen, um zwei Strings gleich zu machen.
  - "misspell" zu "mispell" hat Abstand 1
  - "misspell" zu "mistell" hat Abstand 2
  - "misspell" zu "misspelling" hat Abstand 3

Editier-/Levenstein-Distanz

Definition 6.3.
Für zwei Strings x und y ist die *Editierdistanz (Edit Distance)* $edit(x, y)$ definiert als die kleinste Anzahl an Einfüge- und Löschoperationen, die notwendig sind, um x in y zu überführen.

Läßt man zusätzlich auch die Ersetzung eines Symbols zu, so spricht man von einer *Levenstein-Metrik (Levenshtein Distance)* $lev(x, y)$.

Nimmt man als weitere Operation die Transposition (Vertauschung zweier benachbarter Symbole) hinzu, so erhält man die *Damerau-Levenstein-Metrik* $dlev(x, y)$.

Editier-/Levenstein-Distanz

Bemerkungen:

- Offensichtlich gilt stets
  $dlev(x, y) \leq lev(x, y) \leq edit(x, y)$.

- Die Damerau-Levenstein-Metrik wurde speziell zur Tippfehlerkorrektur entworfen.

- Wird in Problem 6.2 für d eine der Metriken aus Definition 6.3 verwendet, dann spricht man auch von "string matching with k differences" bzw. von "string matching with k errors".

Beispiel:

Für x = abcabba und y = cbabac gilt:

$edit(x, y) = 5$

- $abcabba \rightarrow bcabba \rightarrow cabba \rightarrow cbba \rightarrow cbaba \rightarrow cbabac$

$dlev(x, y) = lev(x, y) = 4$

- $abcabba \rightarrow cbcabba \rightarrow cbabba \rightarrow cbaba \rightarrow cbabac$
- $abcabba \rightarrow bcabba \rightarrow cbabba \rightarrow cbabab \rightarrow cbabac$

Berechnung der String-Distanz

Problem 6.3. Gegeben seien zwei Strings x und y. Man ermittle $edit(x, y)$ bzw. $lev(x, y)$ bzw. $dlev(x, y)$ sowie die zugehörigen Operationen zur Überführung der Strings.

Bemerkungen:

- Wenn x und y Dateien repräsentieren, wobei $x[i]$ bzw. $y[j]$ die i-te Zeile bzw. j-te Zeile darstellt, dann spricht man auch vom *File Difference Problem*.
- Unter UNIX steht das Kommando diff zur Lösung des File Difference Problems zur Verfügung.

- Da die Metriken *edit*, *lev* und *dlev* sehr ähnlich sind, wird im folgenden nur die Levenstein-Metrik betrachtet.
- Algorithmen für die anderen Metriken erhält man durch einfache Modifikationen der folgenden Verfahren.

Berechnung der String-Distanz

- Im folgenden sei $m = |x|$ und $n = |y|$ und es gelte $m \leq n$.

- Lösungsansatz: dynamische Programmierung
- Genauer: berechne die Distanz der Teilstrings $x[1 \ldots i]$ und $y[1 \ldots j]$ auf Basis bereits berechneter Distanzen.

Berechnung der String-Distanz

Die Tabelle LEV sei definiert durch:
$LEV[i, j] := lev(x[1 \ldots i], y[1 \ldots j])$ mit $0 \leq i \leq m$, $0 \leq j \leq n$

Die Werte für $LEV[i, j]$ können mit Hilfe der folgenden Rekursionsformeln berechnet werden:

- $LEV[0, j] = j$ für $0 \leq j \leq n$,
- $LEV[i, 0] = i$ für $0 \leq i \leq m$

- $LEV[i, j] = \min\{LEV[i-1, j] + 1,$
  $LEV[i, j-1] + 1,$
  $LEV[i-1, j-1] + \delta(x[i], y[j])\}$

mit $\delta(a, b) = \begin{cases} 0, & \text{falls } a = b \\ 1, & \text{sonst} \end{cases}$

Berechnung der String-Distanz

Bemerkungen:

- Die Rekursionsformel spiegelt die drei Operation Löschen, Einfügen und Substitution wider.
- Die Stringdistanz ergibt sich als $LEV[m, n]$.
- Möchte man nur die Stringdistanz berechnen, so genügt es, sich auf Stufe i der Rekursion die Werte von LEV der Stufe $i - 1$ zu merken.
- Benötigt man die zugehörigen Operationen, speichert man LEV als Matrix und ermittelt die zugehörigen Operationen in einer "Rückwärts-rechnung".

Berechnung der String-Distanz

Algorithmus 6.4. [Berechnung der Stringdistanz]

for $i := 0$ **to** m **do** $LEV[i, 0] := i$ **end**
for $j := 1$ **to** n **do** $LEV[0, j] := j$ **end**
for $i := 1$ **to** m **do**
  for $j := 1$ **to** n **do**
    $LEV[i, j] := \min\{LEV[i-1, j] + 1,$
      $LEV[i, j-1] + 1,$
      $LEV[i-1, j-1] + \delta(x[i], y[j])\}$
  end
end
return $LEV[m, n]$

Berechnung der String-Distanz

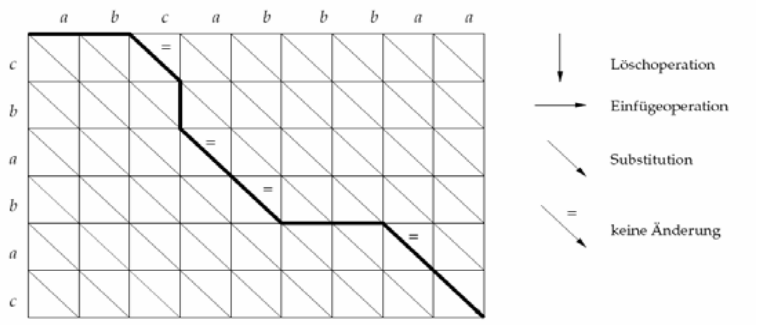
- **Beispiel:**
- Darstellung von *LEV* als Matrix für $x = cbabac$ und $y = abcabbbaa$:

		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

- Die zugehörigen Umwandlungen lauten:
  cbabac → ababac → abcabac → abcabbac → abcabbbac → abcabbbaa

Berechnung der String-Distanz

- **Veranschaulichung:** Die Berechnung der Stringdistanz kann als Pfad in einem Graphen veranschaulicht werden.



Der dargestellte Pfad entspricht der folgenden (nicht optimalen) Umwandlung:
cbabac → acbabac → abcbabac → abcabac → abcabbac → abcabbbac → abcabbbaa

Berechnung der String-Distanz

Aus der Rekursionsformel und den Bemerkungen folgt:

Satz 6.2. *Die Stringdistanz (für edit, lev und dlev) kann in Zeit $O(mn)$ und Platz $O(m)$ berechnet werden. Problem 6.3 kann mit Platz $O(mn)$ gelöst werden.*

Berechnung der String-Distanz

Mit einer kleinen Änderung kann die angegebene Rekursionsformel auch zur Lösung von Problem 6.2 eingesetzt werden:

Es sei *MLEV* definiert durch:

$$\text{MLEV}[i,j] := \min_{1 \le l \le j} \{lev(pat[1 \ldots i], text[l \ldots j])\}$$

d.h., *MLEV* $[i, j]$ ist die kleinste Distanz zwischen $pat[1 \ldots i]$ und einem Suffix von $text[1, j]$.

Es gilt nun: *MLEV* $[0, j] = 0$ für $0 \le j \le n$, denn $pat[1 \ldots 0] = \varepsilon$ und ε ist stets in $text[1 \ldots j]$ ohne Fehler enthalten.

Berechnung der String-Distanz

Ansonsten berechnet sich $MLEV[i, j]$ wie $LEV[i,; j]$, d.h.:

$MLEV[i, 0] = i$ für $0 \leq i \leq m$

$$MLEV[i,j] = \min \{MLEV[i-1,j]+1,\ MLEV[i,j-1]+1,\ MLEV[i-1,j-1]+\delta(x[i], y[j])\}$$

Gilt nun $MLEV[m, j] \leq k$, so endet in Position j ein Substring y von *text* mit $lev(pat, y) \leq k$ (wobei m die Patternlänge ist).

Berechnung der String-Distanz

Beispiel: Die Tabelle *MLEV* für *pat* = *ABCDE* und *text* = *ACEABPCQDEABCR*.

	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
i			A	C	E	A	B	P	C	Q	D	E	A	B	C	R
0		0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	A	1	0	1	1	0	1	1	1	1	1	1	0	1	1	1
2	B	2	1	1	2	1	0	1	2	2	2	2	1	0	1	2
3	C	3	2	1	2	2	1	1	1	2	3	3	2	1	0	1
4	D	4	3	2	2	3	2	2	2	2	2	3	3	2	1	1
5	E	5	4	3	2	3	3	3	3	3	3	2	3	3	2	2

Für $k = 2$ ergeben sich die Positionen 3, 10, 13 und 14. Die zugehörigen Substrings von text sind *ACE*, *ABPCQDE*, *ABC* und *ABCR*.

Satz 6.3. *Problem 6.2 kann für die Metriken edit, lev und dlev in Zeit* O(mn) *gelöst werden.*

Das Soundex-Verfahren

Problem: Suche nach phonetisch ähnlichen Wörtern.

- Wörter werden hierzu auf interne Codes abgebildet.
- Phonetisch ähnliche Wörter sollen auf möglichst gleiche Codes abgebildet werden.
- Das bekannteste Verfahren ist der SOUNDEX-Algorithmus.

Das Soundex-Verfahren

SOUNDEX-Algorithmus

Der SOUNDEX-Algorithmus besteht aus den folgenden Schritten:

1. Nimm den ersten Buchstaben des Wortes und transformiere die restlichen Buchstaben (unabhängig von Groß- und Kleinschreibung) nach der folgenden Tabelle.

a e i o u h w y	→ 0
b f p v	→ 1
c g j k q s x z	→ 2
d t	→ 3
l	→ 4
m n	→ 5
r	→ 6

2. Streiche alle Nullen.
3. Reduziere alle hintereinander vorkommenden gleichen Zahlen auf eine Zahl.
4. Beschränke den ganzen Code auf maximal vier Stellen.

Beispiel

Die Namen *Neumann* und *Newman* werden als gleich erkannt:

- Neumann → N005055 → N555 → N5
- Newman → N00505 → N55 → N5

Bemerkungen

- Der SOUNDEX-Algorithmus dient insbesondere der Suche nach Namen.
- Der Code ist nicht immer erfolgreich, denn ähnlich geschriebene Wörter werden oft auf unterschiedliche Codes abgebildet:
  - Beispiel: Rodgers und Rogers.
  - Rodgers → R032062 → R3262 → R326
  - Rogers → R02062 → R2262 → R262
- In der Praxis wird der SOUNDEX-Algorithmus z.T. in leicht abgeänderter Form eingesetzt.