

SUCHALGORITHMEN IN ZEICHENKETTEN

Suche in Texten oder anderen Zeichenfolgen ist eine häufige — wenn nicht die häufigste — Computeranwendung:

- Editoren, Textverarbeitungssysteme,
- Durchsuchen von Textkorpora: Basis für “Text Mining” und “Information Retrieval” (unstrukturierte Texte),
- Bioinformatik: DNS-Sequenzierung,
- Suche in semi-strukturierten Daten (Teile — “Elemente” — sind markiert mithilfe von Auszeichnungssprachen (“markup languages”), z.B. XML), ggf. mit Analyse von Metadaten (z.B. codiert in RDF(S)),
- Suchmaschinen im Internet und Intranetzen (Baukasten: Lucene/Apache)

Literatur:

Aho/Ullman, *Schöning*, Cormen et al. (*detaillierte theoretische Untersuchung!*), Manber, Skiena;

R. Sedgewick, *Algorithmen*. Bonn: Addison-Wesley, 2004

Mit spezieller Berücksichtigung der Bioinformatik:
D. Gusfield: *Algorithms on Strings, Trees, and Sequences*. Cambridge: Cambridge Univ. Press, 1995

Klassifikation von “String-Matching”–Aufgaben

Grundbegriffe: *Zeichen, Alphabet, Zeichenkette, Formale Sprache*

Gegeben: Eine (kurze) Zeichenkette, der “Suchstring” (Muster, “Pattern”) P und eine längere Zeichenkette T bzw. eine Menge von diesen (mehrere Dateien).

Gesucht: Ein oder alle Vorkommen des Suchstrings P in T

Resultate: passende gefundene Zeichenkette (im Kontext); Position der Passung.

Seiteneffekte: z.B. Anzeige durch visuelle Hervorhebung

- **Exakte Passung:** Wo tritt P in T auf?
▷ `fgrep`
- **Passung von Wortmengen:** Wo tritt P_i aus einer Menge von Strings $\{P_i | i = 1, \dots, k\}$ in T auf?
▷ `agrep`
- **Passung regulärer Ausdrücke:** Welche Stellen in T passen auf einen regulären Ausdruck P ?
▷ `grep`, `egrep`
- **Approximative Passung:** Welche Stellen in T passen am besten (gemäß einer geg. Metrik) bzw. bis auf d Fehler auf ein Muster P ?
▷ `agrep`

Basisalgorithmus

Im folgenden werden nur die Basisversionen der Algorithmen vorgestellt (nach Schönig). Für eine ausführliche Diskussion und Optimierung sei auf die zitierte Literatur verwiesen.

Notation: imperativer "Pseudocode".

Umsetzung in spezielle Programmiersprachen entsprechend den vorhandenen Ausdrucksmitteln, z.B. in Scheme: ADT String, Index läuft ab 0!

Text und Pattern werden als Felder von Zeichen aus einem gegebenen Alphabet Σ notiert: Text $T[1..n]$, Pattern $P[1..m]$ mit $m \leq n$.

Def.: P kommt in T mit Verschiebung ("Shift") s vor, wenn gilt:

$$T[s + 1, \dots, s + m] = P[1, \dots, m]$$

Aufgabe: Angabe aller (bzw. eines) Shifts, so dass P in T mit Shift s vorkommt.

Basisalgorithmus:

```

for  $s := 0$  to  $n - m$  do
if  $test(s)$  then output  $s$ 

procedure  $test(s)$ 
for  $j := 1$  to  $m$  do
if  $P[j] \neq T[s + j]$  then return false
return true
    
```

Aufwand: $\mathcal{O}((n - m + 1) \cdot m) = \mathcal{O}(n \cdot m)$

Gibt es Algorithmen mit linearem Aufwand bezüglich Eingabelänge, d.h. $\mathcal{O}(n + m)$?

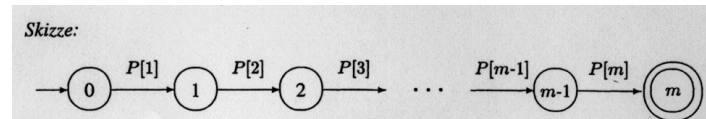
String-Matching mit endlichen Automaten

Ineffizienz des naiven Algorithmus rührt daher, dass bei gefundener Nicht-Übereinstimmung ("Mismatch") zwischen P und dem Textfenster $T[s + 1, \dots, s + m]$ nur um *eine* Position weitergeschoben wird.

Läßt sich aus dem bereits gelesenen Teil von $T[s + 1, \dots, s + m]$ schließen, dass ein größerer Shift möglich ist?

Lösungsansatz: Anhand des Patterns wird ein endlicher Automat konstruiert. Er hat $m + 1$ Zustände 0 (Start), ..., m (Ende); im Endzustand wurde das Pattern erkannt.

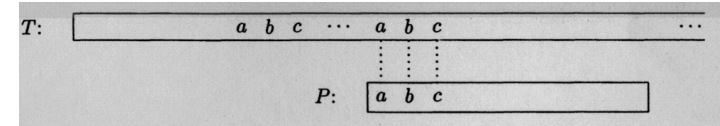
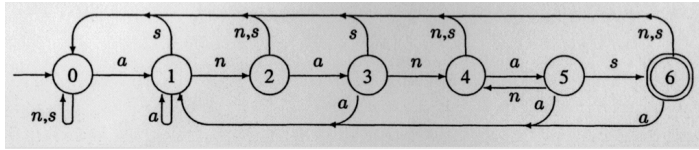
Übergänge: $\delta(i - 1, P[i]) = i$; ($i = 1, \dots, m$)



In diesem *Skelettautomaten* fehlen noch die Übergänge $\delta(i - 1, a)$ für $a \neq P[i]$ sowie die von m ausgehenden: Mismatch-Situationen. Da ein Suffix von $P[1] \dots P[i - 1]a$ wieder ein Präfix des Patterns sein kann, muss zu einem Zustand, der diese partielle Passung berücksichtigt, zurückgesprungen werden:

$$\delta(i, a) = \begin{cases} \max\{k \leq i \mid P[1..k] \text{ Endstück von } P[1..i]a\} & \leftarrow \max \text{ existiert} \\ 0 & \leftarrow \text{sonst} \end{cases}$$

Beispiel: $\Sigma = \{a, n, s\}$ und Pattern = *ananas*



Algorithmus:

```

Konstruiere den endlichen Automaten
z := 0
for i := 1 to n do {
  z := δ(z, T[i])
  if z = m then output i - m }

```

Aufwand: $\mathcal{O}(|\Sigma|m + n)$ mit der Voraussetzung, dass der Algorithmus zur Konstruktion der Übergangstabelle nur $\mathcal{O}(|\Sigma|m)$ benötigt!

Der Teil des Patterns vor der Mismatch-Position ist identisch mit einem Anfangsstück des Patterns.

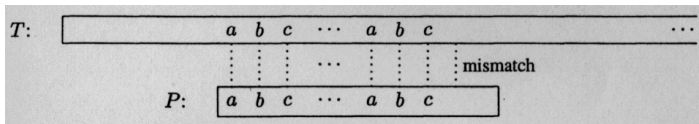
Berechnung der Verschiebetabelle $\Pi[1..m]$ mit

$$\Pi[q] = \max\{k < q \mid P[1..k] = P[q - k + 1..q]\} \quad (q = 1, \dots, m)$$

Der Algorithmus von Knuth, Morris und Pratt

Statt der zweidimensionalen Tabelle mit $|\Sigma| \cdot (m + 1)$ Einträgen wird bei MKP eine einfachere eindimensionale Tabelle mit m Einträgen verwendet.

Sei bereits ein Teil des Patterns mit dem Text verglichen, bis ein Mismatch auftritt:



Nun kann das Pattern so weit verschoben werden, bis der Teil *abc* wieder zur Deckung gebracht wird. Positionen dazwischen können ausgeschlossen werden; zudem braucht man die ersten drei Zeichen nicht noch einmal mit dem Text zu vergleichen:

Beispiel: Tabelle für das Pattern *ananas*

<i>i</i>	1	2	3	4	5	6
<i>P</i>	a	n	a	n	a	s
Π	0	0	1	2	3	0

Algorithmus:

```

Berechne die Verschiebetabelle  $\Pi[1..m]$ 
q := 0
for i := 1 to n do { while q > 0 ∧ (P[q + 1] ≠ T[i]) do q :=  $\Pi[q]$ 
  if P[q + 1] = T[i] then q := q + 1
  if q = m then {output i - m; q :=  $\Pi[q]$  } }

```

Anmerkungen:

In der while-Schleife wird bei einem Mismatch der Wert von q anhand der Verschiebetabelle reduziert; danach wird die fragliche Pattern-Position $q + 1$ wieder mit dem Text verglichen, etc.
 Wird Übereinstimmung zwischen $P[q + 1]$ und $T[i]$ festgestellt, kann das Fenster um eine Position erweitert werden.
 Match ist erreicht, wenn $q = m$

Algorithmus zum Aufbau der Verschiebetabelle:

```

Π[1] := 0
k := 0
for q := 2 to m do {
  while k > 0 ∧ (P[k + 1] ≠ P[q]) do k := Π[k]
  if P[k + 1] = P[q] then k := k + 1
  Π[q] := k }
  
```

Anmerkungen:

q ist aktuelle Pattern-Position, k Position des Anfangsabschnitts, der möglicherweise mit dem Endstück von $P[1..q]$ übereinstimmt.

Die Π -Werte werden anhand der bereits berechneten bestimmt:

Sei $q > 1$ und k maximal gewählt (while-Schleife) derart, dass

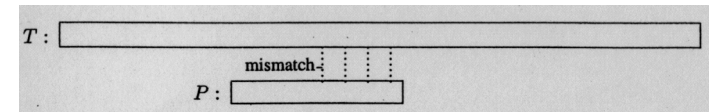
$P[1..k + 1] = P[q - k..q]$; dann ist $\Pi[q] = k + 1$

Ergebnis zur Komplexität: $\mathcal{O}(m + n)$ im ungünstigsten Fall

Der Algorithmus von Boyer und Moore

Auch hier wird das Pattern am Text entlang geschoben, um im Falle eines Mismatch das Pattern um einen möglichst großen Abschnitt weiter zu schieben.

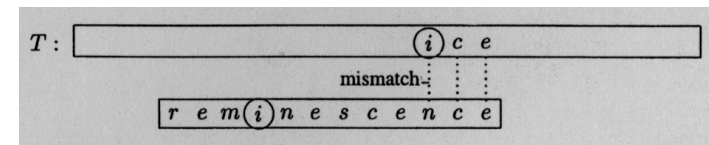
Neue Idee: Vergleiche zwischen Textabschnitt und Pattern werden *von rechts nach links* durchgeführt.



Zur Verschiebung des Patterns bei Mismatch gibt es zwei Strategien; es wird diejenige eingesetzt, die die größte Verschiebung ergibt.

1. Die "Bad Character"-Strategie

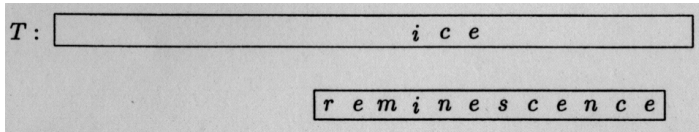
Gebe es im folgenden Beispiel beim dritten Vergleich einen Mismatch:



Hierbei ist das i (oben) der "bad character". Nun wird **vor** der Mismatch-Position im Pattern nach dem **am weitesten rechts** vorkommenden i gesucht.

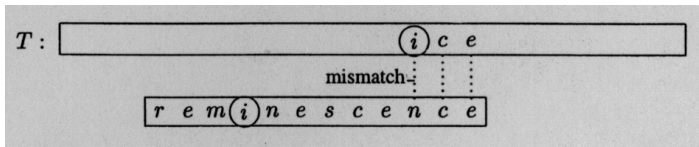
Das Pattern kann so weit verschoben werden, bis die beiden i zur Deckung kommen — dazwischen ist keine Passung möglich. Andernfalls kann das Pattern sogar vollständig über die Mismatch-Position geschoben werden.

Nach der Verschiebung:

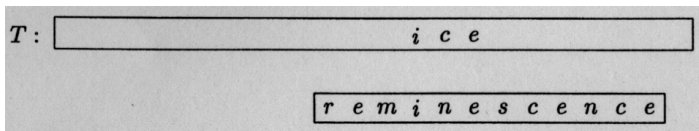


Benötigt wird eine Tabelle $T[i, a]$ mit $1 \leq i \leq m$ und $a \in \Sigma$, so dass $T[i, a] = \max\{j < i \mid P[j] = a\}$. Existiert das Maximum nicht, wird $T[i, a] := 0$. Aus der Tabelle sind die Verschiebelängen $(i - T[i, a])$ zu entnehmen.

2. Die "Good Suffix"-Strategie



Es wird der am weitesten rechts von der Mismatch-Position vorkommenden Suffix gesucht, in dem Text und Pattern übereingestimmt haben (hier: *ce*). Nun kann das Pattern so weit verschoben werden, bis diese Textabschnitte zur Deckung gebracht werden:



Die hierzu benötigte Tabelle kann analog zu KMP erfolgen, allerdings von rechts nach links durch P laufend.

Anmerkung zum Aufwand: In der Praxis ist Boyer-Moore der schnellste Algorithmus — in durchschnittlichen Fällen $\mathcal{O}(m/n)$; im ungünstigsten Fall gilt allerdings wie beim naiven Algorithmus $\mathcal{O}(m \cdot n)$! (Z.B., wenn $T = a^n$ und $P = a^m$).

Anmerkungen zum Algorithmus von Robin und Karp

Grundidee des Algorithmus:

Das Pattern wird mittels einer **Hash-Funktion** auf ein wesentlich kleineres Wort (oder Zahl) $h(P)$ abgebildet, so dass es in eine Speicherzelle passt und mit Aufwand $\mathcal{O}(1)$ verarbeitet werden kann.

Anstelle der Original-Textabschnitte der Länge m werden deren Hashwerte mit $h(P)$ verglichen. Bei Nicht-Übereinstimmung kann das Vergleichsfenster weitergeschoben werden. Bei Übereinstimmung muss jedoch ein zeichenweiser Vergleich erfolgen, da eine Kollision mit einem Text ungleich P stattgefunden haben kann.

Aufwand: Im ungünstigsten Fall $\mathcal{O}(m \cdot n)$. Kommt das Pattern im Text nicht sehr häufig vor, kann im Mittel $\mathcal{O}(m + n)$ erreicht werden.

Suffix-Bäume

Bisher: Vorverarbeitungsphase, die eine geeignete Datenstruktur (z.B. Tabelle) für das Pattern aufbaut, so dass der eigentliche Suchvorgang effizient — ideal mit Zeitaufwand $\mathcal{O}(n)$ — durchgeführt werden kann.

Liegt aber ein Text ein für alle Mal fest (bzw. sehr langfristig relativ zur Rate der Anfragen) und soll häufig durchsucht werden, lohnt sich eine Vorverarbeitungsphase, die eine für besonders schnelle Suche geeignete Datenstruktur für den Text aufbaut.

Typische Anwendungen: "Information Retrieval", Bioinformatik

Wir betrachten hier die Datenstruktur des *Suffix-Baums* (engl. "suffix trie", Kunstwort aus "tree" und "retrieval").

Sei der Text $T = \text{atacatgc}\$$ gegeben, wobei $\$$ ein besonderes Zeichen zur Markierung des Textendes ist. Für jede Textposition $i = 1, \dots, n$

(hier $n = 10$) wird die kürzeste Teilkette notiert, die bei der Position i beginnt und nur ein Mal in T vorkommt. Damit identifiziert diese Teilkette die Position i .

Position	Teilstring
1	atac
2	tac
3	ac
4	ca
5	atag
6	tag
7	ag
8	g
9	c\$
10	\$

Text $T = atacatagc\$$

Der Suffix-Baum besitzt die Blätter $1, 2, \dots, n$, wobei der Pfad von der

Beispiele:

at kommt an den Positionen 1 und 5 vor;

aca: *ac* kommt nur an Position 3 vor — Fortsetzung wird durch direkten Vergleich überprüft;

tg kommt nicht vor.

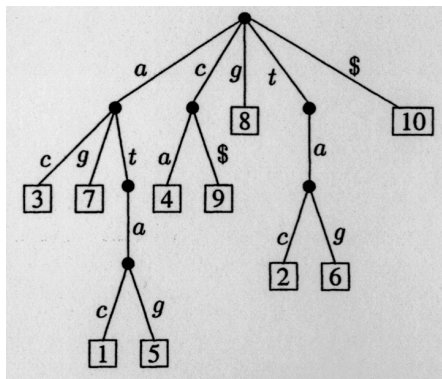
⇒ Suche nach Pattern der Länge m mit $\mathcal{O}(m)$ möglich.

Konstruktion des Suffix-Baums: s. Schönig, S. 316f.

Weitere Verbesserung des Verfahrens: PATRICIA

(Morrison: Practical Algorithm To Retrieve Information Coded In Alphanumeric, 1968)

Wurzel zum Blatt i mit den betreffenden Buchstaben der identifizierenden Teilkette markiert ist.



Hinweis zur approximativen Suche

In vielen Anwendungen möchte man alle Stellen in einem Text finden, die dem Pattern sehr ähnlich sind. Hierzu muss eine Metrik auf Zeichenketten definiert werden

- Hamming-Distanz für zwei Strings x und y gleicher Länge: Anzahl der Positionen, an denen sich x und y unterscheiden.
- Editierdistanz für zwei Strings x und y : Kleinste Anzahl an Einfüge- und Löschooperationen, die notwendig ist, um x in y zu überführen.
- Levenshtein-Distanz: Zusätzlich wird Ersetzung von Zeichen zugelassen.
- Damerau-Levenshtein-Distanz: Zusätzlich wird die Vertauschung zweier benachbarter Zeichen (Transposition) zugelassen. (Speziell für Tippfehlerkorrektur)

Berechnung der String-Distanz(en): **Dynamische Programmierung**

In Unix steht für approximative Suche `agrep` zur Verfügung.

Eine Variante der approximativen Suche ist die sog. "phonetische Suche", bei der die Varianten in einem reduzierten Alphabet dargestellt werden, das aufgrund phonetischer Kriterien konstruiert wurde.

Grundidee: Verbindung von Schreibvarianten mit Ausspracheregeln.
Algorithmen zur Codierung und Decodierung: SOUNDEX, SIMILAREX, u.a.

"Pattern Matching" mit regulären Ausdrücken

In vielen Fällen ist es wünschenswert, das Pattern nicht nur "wörtlich", sondern als Patternmenge, abgekürzt durch einen regulären Ausdruck, anzugeben.

Neben den Zeichen des Alphabets gibt es eine Menge grammatischer Sonderzeichen mit besonderer Bedeutung — darunter eines zur "Quotierung" grammatischer Sonderzeichen, um auch nach diesen im Text suchen zu können.

Ausdrucksmittel

- Zeichenklassen: Auswahlliste, Zeichenausschluss, Zeichenbereiche
- "Wildcards": einzelnes bel. Zeichen, Ziffer, Layoutzeichen
- Optionalität: Zeichen und Muster
- Disjunktion
- Verankerungen: Positionierung am Anfang, Ende einer Zeichenkette, an "Wortgrenze"
- Wiederholungen: einmal, n mal, mindestens n (und höchstens m) mal, beliebig oft

Die für Unix-Dienstprogramme (`grep` & Co. u.a.) eingeführte Notation hat sich weitgehend durchgesetzt.

Für viele Programmiersprachen stehen entsprechende Prozeduren in Programmbibliotheken zur Verfügung, in manchen (z.B. Perl) sogar als Standardprozeduren.

`lex`, `flex`: Werkzeug zur Generierung "lexikalischer" Analyseprogramme ("Scanner")

Theoretischer Hintergrund und Basis aller Implementationen ist der Satz, dass alle regulären Sprachen von **endlichen Automaten** erkannt werden können (und umgekehrt).

Suche mit linguistischer Vorverarbeitung

Einsatz-Szenarien: Suchmaschinen, Kataloge (OPAC), etc., ggf. auch in Verbindung mit Suchverfahren für semi-strukturierte Daten.

Vorverarbeitung der Anfragen über Wörterbuchabgleich mithilfe computerlinguistischer Verfahren:

- Morphologische Komponente: Segmentierung in Stamm und Suffixe; Zerlegung von Komposita; Generierung flektierter Wortformen — hierfür reichen endliche Automaten aus!
- Ergänzung durch Synonyme (ggf. Antonyme) aus einem hierarchischen Lexikon, z.B. WordNet
- Spezialwörterbücher für verschiedene Fachgebiete

Strings in Scheme

Sowohl **Zeichen** (characters) als auch **Zeichenketten** (strings) sind Standard-Datentypen in R⁵RS.

Grundfunktionen für Strings (Auswahl):

- Typ-Prädikat `string?`
- Konstruktoren: `make-string?`, `string`, `string-append`, `string-copy`,
- Selektoren: `string-ref`, `substring`
- Modifikatoren: `string-set!`, `string-fill!`
- Typ-Konvertierung: `string->list`, `list->string`
- Vergleichsoperatoren: `string=?`, `string<?`, etc.

- "Normdateien": Namen von Personen, Körperschaften, geographischen Orten
- Erkennung spezieller Einheiten: Abkürzungen, Zeit- und Datums-Angaben, Maßangaben, etc.
- Mehrsprachigkeit!

Erweiterte Suche unter Zuhilfenahme von Thesauri und *terminologischen Hierarchien*: z.B. Hierarchisches Lexikon, das weitere lexikalische Relationen darstellt, u.a. engere und weitere Termini, Teil-Ganzes-Beziehungen, etc.

Damit ist der Bereich der Stringsuche i.e.S. verlassen. . .

Die portable Programmbibliothek **SLIB** enthält leistungsfähige Prozedurpakete zur Verarbeitung von Zeichenketten, darunter auch zur Stringsuche. Vgl. auch `pregexp` von Dorai Sitaram.