Algorithmen und Datenstrukturen

Praktische Einführung und Programmierung

Stefan Bosse

Universität Koblenz - FB Informatik

Textverarbeitung

Wie werden Texte maschinell verarbeitet?

Textverarbeitung

Wie werden Texte maschinell verarbeitet?

Welche Algorithmen und Datenstrukturen werden gebraucht?

Textverarbeitung

Wie werden Texte maschinell verarbeitet?

Welche Algorithmen und Datenstrukturen werden gebraucht?

Was sind reguläre Ausdrücke und wie und wofür verwendet man sie?

Anwendungen für Textverarbeitung

Übersetzer

- Übersetzung einer natürlichen Sprache in eine andere natürliche
- Übersetzung einer natürlichen Sprache in eine maschinelle Speache oder Notation (symbolische Repräsentation, symbolische KI)
- Übersetzung einer maschinellen (Programmier-) Sprache in eine andere
  - Meistens Reduktion der Ausdrucksfähigkeit und Komplexität
  - Beispiel: C++ in konkrete Maschinensprache eines Mikroprozessors (strukturiert und hierarchisch auf lineare Struktur)
  - Beispiel: Java in abstralte Maschinensprache einer Virtuellen Maschine

Suche

- Bisher haben wie in strukturierten Daten gescuht. Jetzt Suche un unstrukturierten Daten - den Texten.
  - Ein Text besteht aus Zeichen mit einem Alphabet, ggfs. unterteilt uin Worte mit Trennzeichen
- Suche in Texten bedarf vor allem effiziente Algorithmen, häufig mit Zustandsautomaten

Reguläre Ausdrücke

Wozu braucht man RA?

[https://files.ifi.uzh.ch/cl/hess/classes/le/regex.0.1.pdf]

- Die Verwendung regulärer Ausdrücke (RA) ist heute weit verbreitet im Bereich grundlegender, einfacherer Verfahren der Computerlinguistik (z.B. Tokenisierung und Tagging), Suchmaschinen oder in einfachen Interpretern, der sogenannten **flachen Textverarbeitung**.
- Mit Hilfe von regulären Ausdrücken lassen sich Muster für einfache sprachliche Ausdrücke spezifizieren, nach denen automatisch in Texten gesucht werden kann.
- Ein Beispiel hierfür, umgangssprachlich ausgedrückt, wäre "Wort, das mit einem Grossbuchstaben anfängt und mit einem 'k' aufhört".
  - Je nach Textkorpus, in dem man sucht, lässt sich damit das Wort "Computerlinguistik" finden.
  - Die gefundenen Instanzen (Matches) eines solchen Mustervergleichs (Pattern matching) stehen dann zum Teil direkt der weiteren Verarbeitung zur Verfügung.

Woher kommen RA?

- Die Verwendung regulärer Ausdrücke ist vor allem durch Suchfunktionen (egrep) und Skriptsprachen (lex und flex) auf Unix-Ebene sowie durch plattformübergreifende Skriptsprachen (vor allem Perl, JavaScript) bekannt geworden
- Mittlerweile können RA in vielen gängigen Programmiersprachen verwendet werden.

Was sind RA?

- Reguläre Ausdrücke sind zunächst einmal einfach zu spezifizieren.
  - Einfach heisst dabei nicht unbedingt leicht verständlich, ganz im Gegenteil!, sondern gemäss formaler Kriterien wie Repräsentations- und Verarbeitungskomplexität einfach - das macht sie so attraktiv.
  - Entsprechend lassen sich Sprachen, die RA beschreiben (sogenannte reguläre Sprachen), mithilfe des einfachsten Grammatiktyps (sogenannten Typ-3-Grammatiken) beschreiben.
  - Solche Sprachen können nachweislich in effizient zu verarbeitende Strukturen (sogenannte Endliche Automaten) übersetzt werden, die durch RA spezifizierte Ausdrücke schnell aufspüren.

Was kann man mit RA machen?

- Programmiersprachen, die die Verwendung von RA implementieren, bieten in der Regel mindestens die folgenden Möglichkeiten:
  - checken, ob ein RA in einem Text vorkommt
  - angeben, wo ein RA in einem Text vorkommt
  - den Zugriff auf Teile eines Matches im Text
  - den Zugriff auf alle zu findenden Matches im Text
  - die Möglichkeit zur Ersetzung eines (oder aller) Vorkommen in einem Text durch etwas anderes
  - die Möglichkeit zur Zerlegung eines Textes an den Stellen, an denen ein gegebener RA vorkommt.

Konstruktion regulärer Ausdrücke

- Im folgenden sollen die wesentlichen Möglichkeiten zur Konstruktion von RA vorgestellt werden.
- Hierzu werden RA in den Beispielen stets durch Schrägstriche ("/") (slashes) eingerahmt (wie es z.B. in Perl oder JavaScript üblich ist).
- Anhand eines solchermassen angegebenen Suchmusters wird ein Text von Anfang bis Ende durchsucht und es wird der zuerst gefundene Match als Ergebnis geliefert.
- Im Folgenden gehen wir davon aus, dass jeweils alle Ergebnisse gesammelt werden.

Direkte Angabe von Zeichenketten

Direkte Angabe von Zeichenketten, also Abfolgen von Zeichen, sind der einfachste Fall:

```
/einzig/
/einzig|winzig/
```

Suche nach dem Vorkommen von "einzig" im Text oder alternativ durch "|" getrennt nach "einzig" und "winzig" (**Disjunktion**).

Zeichenklassen

- Zeichenklassen ("character classes") sind eine wichtige Möglichkeit, reguläre Ausdrücke kürzer zu formulieren
  - In "[]" eingeschlossen lassen sich Zeichen ange-ben, die an einer bestimmten Position auftreten sollen/dürfen.
  - Sie dienen im wesentlichen der Vereinfachung von RA (um nicht jeweils die Zeichenalternativen an einer Position explizit angeben zu müssen).
- weitere Möglichkeiten, Zeichenklassen zu spezifizieren:
  - `/[A-Z]/` ein beliebiger Grossbuchstabe (aber ohne Umlaute, ß und Buchstaben mit Akzenten)
  - `/[A-Za-z]/` ein beliebiger Gross- oder Kleinbuchstabe (ditto)
  - `/[0-9]/` eine beliebige Ziffer
  - Mischungen und Teilbereichsangaben sind möglich, z.B. `/[ab17G-K]/`
  - Ausschluss: `/[^abc]/` passt auf alle Zeichen außer a,b und c.

Man kann bei der Suche das Flag "g" hinter den Ausdruck setzen. Dann wird dieser mehrfach angewendet und liefert alle passenden Ergebnisse.

Wiederholungen

- Häufig sollen Muster mehrfach expandiert werden, d.h. z.B. würde /a/ nur ein Zeichen "a" im Text "aabbba" suchen.
- Sollen Sequenzen von Zeichen die aus einzelnen Teilen zusammengesetzt sind gesucht werden braucht man Wiederholungen.
- Quantifikatoren geben an, wie häufig das direkt vorangehende Zeichen bzw. die direkt vorangehende Gruppe oder Zeichenklasse vorkommen können soll.
- Es gibt einige allgemeine und eine Reihe von numerischen Wiederholungen.
- Allgemeine Wiederholungen sind:
  - `*` Keinmal oder beliebig oft
  - `+` Mindestens einmal
  - `?` Einmal oder keinmal

Beispiel:

```
s="aabbba"
match(/a+|b+/g,"abbaab") => [ 'a', 'bb', 'aa', 'b' ]
```

Reguläre Ausdrücke und Automaten

Reguläre Ausdrücke bzw. deren Anwendung ist i.A. zustandslos und kontextfrei.

Folgendes Beispiel:

```
if a=0 then a=1 end
```

Das "then" ist nur nach einem "if" zulässig, dazwischen befindet sich ein Ausdruck. Es gibt also einen Kontext. Das Gleichheitszeichen hat hier zwei Bedeutungen: Vergleich und Zuweisung (wie z.B. in der Programmiersprache Basic). Es gibt also einen Kontext. Ist `=` in einem Ausdruck dann ist es ein Vergleich und liefert einen Booleschen Wert, ist es Bestandteil einer Anweisung dann ist es eine Zuweisung. Das kann ein regulärer Ausdruck nicht unterscheiden.

Reguläre Ausdrücke und Automaten

Mit einer Ausnahme:

Rückverweise sind ein sehr mächtiges Instrument beim Schreiben von regulären Ausdrücken.

- Für gruppierte (d.h. in runde Klammern eingeschlossene) Teilmuster werden systematisch Register angelegt, so dass mit Hilfe von Variablen der Form "$i" (mit i aus [1-9]) ein Rückbezug auf Matches dieser Teilstrukturen möglich ist.
- Dies ist insbesondere bei Ersetzungen sehr hilfreich.

Innerhalb des RA kann mit "\i" auf Teilmuster(matches) zurückverwiesen werden:

```
/\b([egin]{3,3}).*?\b\s\1/
```

Suche nach dem Vorkommen einer Zeichenkette, die mit einer Wortgrenze beginnt, gefolgt von genau 3 Zeichen, die jeweils entweder "e", "g", "i" oder "n" sind (dieses Muster wird als erstes Muster gespeichert), gefolgt von beliebigen Zeichen bis zur ersten Wortgrenze, gefolgt von einem white space, gefolgt vom ersten Muster

Reguläre Ausdrücke und Automaten

Man kann jeden regulären Ausdruck in einen Endlichen Zsutandsautomaten (EA) übersetzen, möglicherweise aber ein nicht determinisistischer Automat! RA sind aber viel mächtiger und ausdrucksstärker.

[https://inf-schule.de/automaten-sprachen/sprachenundautomaten/spracherkennung/regulaeresprachen/theorie_regulaereausdrueckeendlicheautomaten]

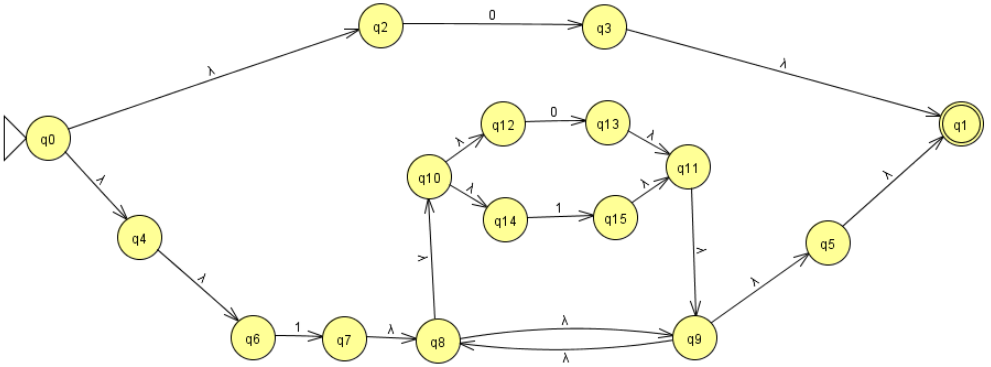


Abb. 1. Beispiel eines Endlichen Automaten zum regulären Ausdruck `0+1(0+1)*`

Reguläre Ausdrücke und Automaten

- RA können Graphen beschreiben, die sog. Zustandsübergangsdiagramme eines EA.
- Reguläre Ausdrücke dienen dazu, Sprachen zu beschreiben. So lässt sich die Sprache

$$L_{\text{BIN}} = \{0, 1, 10, 11, 100, 101, 110, 111, 1000, \ldots\}$$
$$\Sigma = \{0, 1\}$$

mit Hilfe des folgenden regulären Ausdruckes beschreiben:

```
0+1(0+1)*
```

Ein EA eines RA arbeitet einen Zeichenstrom (Zeichen für Zeichen einer Textzeichenkette) ab bis alle Zeichen konsumiert oder der Ausdruck erfüllt ist (dann nur Teil einer Textzeichenkette).

Reguläre Ausdrücke und Automaten

Der Knackpunkt bei der Erstellung eines EA bzw. des Zustandsübergangsgraphen sind Wiederholungen und Alternativen von komplexeren RA. Wenn es nur einzelne Zeichen $a \in \Sigma$ des Alphabets als nächstes Zeichen gibt ist das eine einfache Verzweigung im EA Zustandsübergangsdiagramm, bei komplexeren Ausdrücken α muss der EA verschiedene Pfade "probieren" (Lookahead Verhalten).

Reguläre Ausdrücke über dem Alphabet Σ werden wie folgt definiert:

- Ø ist ein regulärer Ausdruck.
- λ ist ein ggfs. zusammengesetzer regulärer Ausdruck.
- Für jedes $a \in \Sigma$ ist a ein regulärer Ausdruck.
- Wenn α und β reguläre Ausdrücke sind, dann sind auch die Konkatenation αβ, die Alternative α|β und die Iteration (α)* sowie (α)+ reguläre Ausdrücke.

Ein Automatenbaukasten

Zu einem regulären Ausdruck kann man recht einfach einen endlichen Automaten konstruieren, der die vom regulären Ausdruck beschriebene Sprache akzeptiert. Der Automat kann dabei aus den Bestandteilen des regulären Ausdrucks nach dem Baukastenprinzip konstruiert werden. Im Folgenden wird erst einmal der Automatenbaukasten vorgestellt.

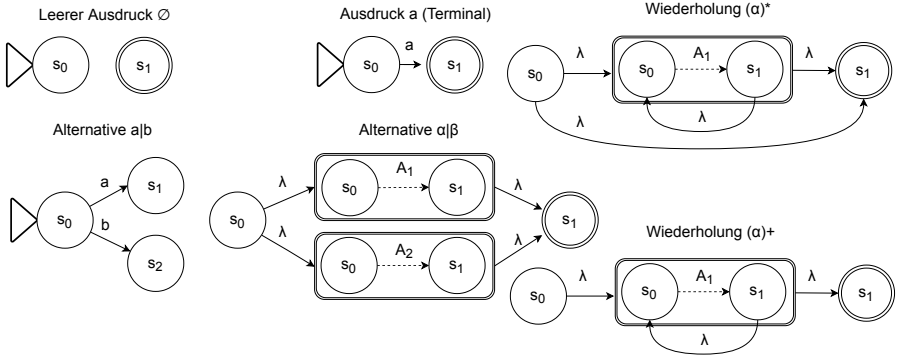


Abb. 2. Automatenbaukasten: A_i sind Automaten, α und β Teilausdrücke, a,b, usw. sind einzelne Zeichen des Alphabets Σ, λ ist eine Einleitung eines Teil-RA, uns s_i sind numerierte Zustände des EA (freie Wahl).

Ein Automatenbaukasten

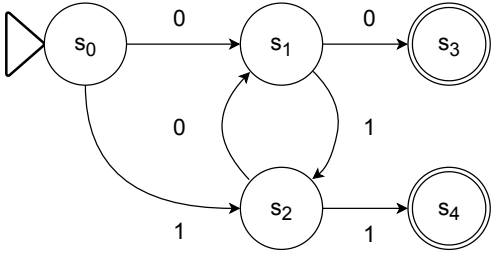


Abb. 3. Beispiel eines EA für den statischen RA `00|11`. Die Zustände s_3 und s_4 könnten noch zusammengeführt werden.

$$s_0 \rightarrow s_2 \rightarrow s_1 \rightarrow s_2 \rightarrow s_4$$

Bsp. 1. Ablauf des obigen EA für 10110

Suche in Texten

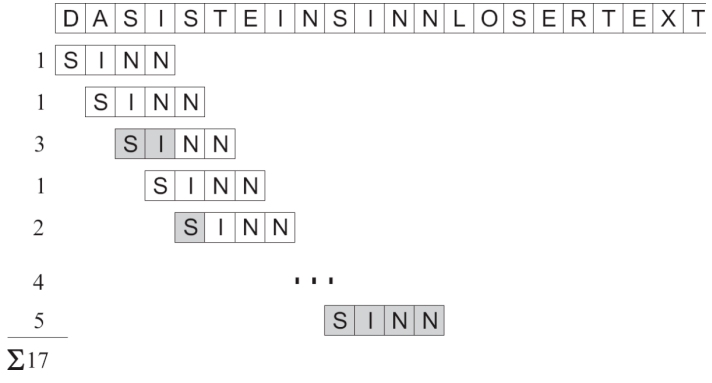
Jetzt wollen wir uns einen Algorithmus für eine Suche eines einfachen Musters in einem Text anschauen.

Brute-Force

```
function BruteForceSearch (text, pat) {
  // Eingabe: zu durchsuchender Text text der Länge n,
  // gesuchtes Muster pat der Länge m
  for (i= 0; i<= n - m;i++) {
    j = 0;
    while (j < m && pat[j] == text[i + j]) {
      j = j + 1
    }
    if (j >= m) return i
  }
  return -1
}
```

Alg. 1. Zwei geschachtelte Schleifen für die Brute-Force-Suche in Texten

Suche in Texten



[addp]

Abb. 4. Ablauf beim Brute-Force-Algorithmus

- Die Laufzeitkomplexität beträgt im ungünstigsten Fall $O((n - m)\, m) = O(nm)$ bei einer Textlänge n und einer Musterlänge m, der zusätzliche Platzbedarf ist aber konstant und damit $O(1)$.
- Insbesondere für lange Muster wünscht man sich daher eine Verbesserung der Laufzeitkomplexität, gegebenenfalls auch bei einer Verschlechterung des Platzbedarfs.

Suche in Texten

Das Problem des Brute-Force-Verfahrens ist im Wesentlichen, dass bei Feststellen einer Nichtübereinstimmung sowohl der Zeiger im Muster als auch der Zeiger im Text zurückgesetzt werden.

An dieser Stelle setzt der Algorithmus von Knuth, Morris und Pratt an, der die Suche mit einer Komplexität von O(n + m) ermöglicht.

```
function KMPSearch (text, pat) {
  // Eingabe: zu durchsuchender Text text der Länge n,
  // gesuchtes Muster pat der Länge m
  j = 0;
  for (i = 0;i<n;i++) {
    while (j >= 0 && pat[j] != text[i]) j = next[j]
    j = j + 1;
    if (j = m) return i - m + 1
  }
  return -1
}
```

Alg. 2. Zwei geschachtelte Schleifen für die Brute-Force-Suche in Texten. Es wird ein voraus berechnetes Hilfsfeld *next* verwendet was Rücksrpungpositionen bei Nichtübereinstimmung angibt

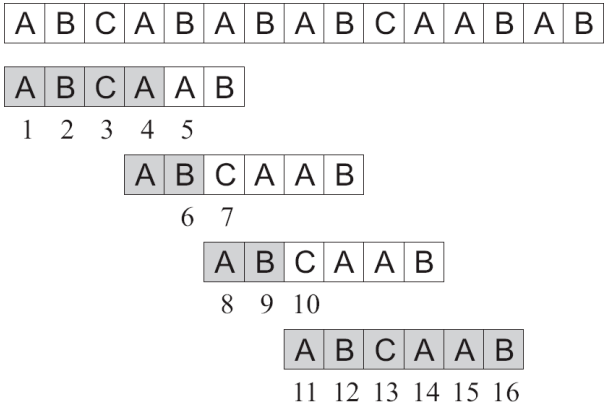
Suche in Texten

```
function initNext (pat) {
  i = 0, j = - 1
  next[0] = - 1
  while (i < pat.length() - 1) {
    while (j > = 0 && pat[i] != pat[j])
      j = next[j]
    i++; j++
    next[i] = j
  }
  return next;
}
```

Alg. 3. Berechung des Hilfsfeldes *next*

Die Anzahl der zu verschiebenden Stellen ist dabei nur vom Muster abhängig. Der Knuth-Morris-Pratt-Algorithmus (kurz »KMP«) nutzt dies aus, indem in einem Vorverarbeitungsschritt die Struktur des Musters analysiert wird. Daraus kann abgeleitet werden, wie weit im Fall einer Nichtübereinstimmung zurückgegangen werden muss. Diese Informationen werden in einem Feld gespeichert, das als next- bzw. border-Feld oder manchmal auch als Fehlerfunktion bezeichnet wird. Es enthält für jede Position des Musters die Distanz für eine sichere Verschiebung. Diese Werte werden bestimmt, indem für jedes j das längste Präfix des Musters pat ermittelt wird, das auch Suffix von pat[1… j] ist.

Suche in Texten



j	0	1	2	3	4	5	6
$next[j]$	-1	0	0	0	1	1	2

Abb. 5. Berechnung des next Feldes für das Muster `ABCAAB`