Algorithmen auf Sequenzen

Vorlesung von Prof. Dr. Sven Rahmann
im Sommersemester 2008

Kapitel 4
Reguläre Ausdrücke

Webseite zur Vorlesung
http://ls11-www.cs.tu-dortmund.de/people/rahmann/teaching/ss2008/AlgorithmenAufSequenzen

Sprechstunde
Mo 16-17 in OH14, R214

Übersicht

- Definition von regulären Ausdrücken
- Einfaches Parsen von regulären Ausdrücken
- NFAs für reguläre Ausdrücke

Reguläre Ausdrücke

Ein regulärer Ausdruck *R* beschreibt eine (endliche oder unendliche) Menge von Strings, d.h. eine Sprache *L*(*R*), über einem gegebenen Alphabet *A*.

Ein regulärer Ausdruck (rA) über dem Alphabet *A* ist ein String über $A \cup \{ \epsilon, |, \cdot, *, (,) \}$, der den folgenden Regeln gehorcht:

- ϵ ist ein rA. Die beschriebene Sprache ist $L(\epsilon) := \{\epsilon\}$.
- Jedes Zeichen $a \in A$ ist ein rA mit $L(a) := \{a\}$ für alle $a \in A$.
- Sind *R* und *S* rAs, so auch (*R* | *S*) mit $L((R \mid S)) := L(R) \cup L(S)$.
- Sind *R* und *S* rAs, so auch (*R*·*S*) mit $L((R \cdot S)) := L(R) \cdot L(S) = \{rs : r \in R \text{ und } s \in S\}$.
- Ist *R* ein rA, so ist auch (*R*) ein rA mit $L((R)) := L(R)$.
- Ist *R* ein rA, so ist auch (*R**) ein rA mit $L((R^*)) := \bigcup_{i \geq 0} L(R)^i$.

Vereinfachung der Notation:

- Den Konkatenierungsoperator · lässt man wie üblich weg.
- Klammern, die zwischen Elementen, die über denselben assoziativen Operator verbunden werden, lässt man ebenfalls weg.
- Man vereinbart, dass * stärker bindet als ·, und · stärker als |, so dass man ggf. weitere Klammern weglassen kann.

Tafelbeispiel: (AT|GA)(AG|AAA)*

Erinnerung: Grundlagen Theoretische Informatik

Reguläre Sprachen

Eine Sprache *L* heißt regulär,
wenn eine der folgenden äquivalenten Bedingungen zutrifft.

- Es gibt einen regulären Ausdruck *R* mit *L*(*R*) = *L*.
- Es gibt einen DFA, der genau die Wörter aus *L* akzeptiert.
- Es gibt einen NFA, der genau die Wörter aus *L* akzeptiert.

Suchproblem für reguläre Ausdrücke

Gegeben ein Text *T* und ein regulärer Ausdruck *R* über demselben Alphabet,
finde die (Start- und) End-Positionen aller Teilstrings *t* von *T* mit $t \in L(R)$.

Häufig ist man nur an den End-Positionen der Matches interessiert. Dann gilt:
Es gibt $i \leq j$ mit $T_i \ldots T_j \in L(R) \Leftrightarrow T_1 \ldots T_j \in L(A^*R)$.
Man benutzt also den regulären Ausdruck A^*R, um die Teilstrings in T zu finden,
die auf *R* matchen.

Klassische Strategien – Übersicht

0. Der rA *R* ist als Zeichenkette gegeben.

1. Erstelle einen Syntaxbaum (parse tree) von *R*

2. Erzeuge einen NFA, der genau L(*R*) erkennt
   - Thompson-Konstruktion
   - Glushkov-Konstruktion

3. Möglichkeiten nach NFA-Konstruktion zur rA-Suche:
- Benutze NFA zum Suchen im Text;
  nutze Bit-Parallelismus, wenn möglich.
- Erzeuge äquivalenten DFA, benutze DFA zum Suchen im Text.

Syntaxbaum

Gegeben sei ein rA R über dem Alphabet A.
Sei m die Gesamtzahl der Zeichen $a \in A$ oder ϵ in R.
Sei M die Länge von R als String.

Wir lesen R von links nach rechts und erstellen dabei den Syntaxbaum S = S(R):
- In den Blättern von S stehen Symbole aus A, oder ϵ.
- In den inneren Knoten von S steht jeweils ein Operator |, ·, *.
- Innere Knoten haben jeweils 1 Kind (*) oder 2 Kinder (|, ·).

Der Syntaxbaum ist wegen der Assoziativität der binären Operatoren nicht eindeutig.

Tafelbeispiel:

Syntaxbäume zu (G|ϵ)A(CGG|A*C)*G
Es macht Sinn, die binären Operatoren als n-är zuzulassen.

Erstellung des Syntaxbaums

Einfacher rekursiver Parser: rA $R = R_1 ... R_M \rightarrow$ Syntaxbaum S.
Aufruf: (S,*last*) := parse(R\$,1) [das $ markiert das Ende des rA]
Der Zeitaufwand ist O(M).

Rekursive Erstellung des Syntaxbaums

```
function parse(R,last):
v := null
while (R_last != $)
    if (R_last is ϵ or in A):
        w := new node('leaf', R_last)
        if v=null then v := w else v := new node('·', v,w)
        last ++
    elsif (R_last = '|'):
        (w, last) := parse(R, last+1)
        v := new node('|', v,w)
    elsif (R_last = '*'):
        v := new node('*', v)
        last ++
    elsif (R_last = '(' ):
        (w, last) := parse(R, last+1)
        if v=null then v := w else v := new node('·', v,w)
        last ++
    elsif (R_last = ')' ):
        return (v, last)
return (v, last)
```

Tafelbeispiel:

```
        11111111
12345678901234567
(G|e)A(CGG|A*C)*G
```

Thompson-NFA

Aus dem Syntaxbaum erhält man durch Bottom-up Auswertung
direkt einen NFA (mit Epsilon-Transitionen), der genau L(R) erkennt:
den Thompson-NFA (Thompson, 1968).
Er verfügt über maximal $2m$ Zustände und maximal $4m$ Transitionen.

Idee

- Zu jedem Knoten *v* des Syntaxbaumes wird rekursiv ein NFA Thompson(*v*) definiert, der genau die Sprache erkennt, die dem rA mit *v* als Wurzel entspricht.
- Thompson(v) verfügt über genau einen Start- und einen Endzustand.
  Der Startzustand hat keine eingehenden, der Endzustand keine ausgehenden Kanten.
- Durch Bottom-up Auswertung erhält man in der Wurzel *r* des Syntaxbaums des rA *R* den NFA Thompson(*r*) mit $L(\text{Thompson}(r)) = L(R)$.

Basisfälle (Blätter)

- Der Automat für ϵ besteht aus 2 Zuständen (Start- und End-Zustand), die mit einer Epsilon-Transition miteinander verbunden sind.
- Der Automat für $a \in A$ besteht aus 2 Zuständen (Start- und End-Zustand), mit mit einer *a*-Transition miteinander verbunden sind.

Thompson-NFA

Konstruktion für innere Knoten des Syntaxbaums

- Konkatenations-Knoten ('·',*v*,*w*):
  Verbinde die Automaten für *v* und *w*, indem der Endzustand von *v* mit dem Anfangszustand von *w* fusioniert wird.
  Neuer Anfangszustand ist der von *v*; neuer Endzustand der von *w*.

- Oder-Knoten ('|', *v*,*w*):
  Erzeuge 2 neue Knoten: den neuen Startzustand *i* und Endzustand *f*.
  Erzeuge 4 neue Epsilon-Transitionen:
  - von *i* zu den Startzuständen von *v* und *w*
  - von den Endzuständen von *v* und *w* nach *f*

- Stern-Knoten ('*', *v*):
  Erzeuge 2 neue Knoten: den neuen Startzustand *i* und Endzustand *f*.
  Erzeuge 4 neue Epsilon-Transitionen:
  - von *i* zum Startzustand von *v*
  - von *i* nach *f*
  - vom Endzustand von *v* nach *f*
  - vom Endzustand von *v* zum Startzustand von *v*

Warum neue Knoten? Um die o.g. Eigenschaften zu erhalten!

Naives rA-Matching mit dem Thompson-NFA

Algorithmus

- Füge dem Startzustand eine *A*-Schleife hinzu, damit der NFA *A*R* statt *R* erkennt.
- Setze *active* := EpsilonAbschluss({Start})
- Wiederhole bis Text erschöpft
  - Lies das nächste Textzeichen *c*
  - Führe *c*-Übergänge auf *active* durch; nenne das Ergebnis *active*'
  - Berechne *active* := EpsilonAbschluss(*active*')
  - Wenn der Endzustand in *active* enthalten ist, wurde ein Match gefunden.

Epsilon-Abschluss

Der Epsilon-Abschluss einer Zustandsmenge *Q* besteht aus allen Zuständen, die durch (evtl. mehrere) Epsilon-Transitionen (hintereinander) von Zuständen in *Q* erreicht werden können.

Laufzeit

Der Thompson-NFA hat linear viele Zuständen in der Pattern-Länge *m*.
Man kann für jeden seinen Epsilon-Abschluss vorberechnen (Größe O(*m*)).
Dies führt auf eine Laufzeit von $O(nm^2)$.

Bit-paralleles Matching mit dem Thompson-NFA

Verbesserung 1

- Nummeriere Zustände in der Reihenfolge ihrer Erzeugung.
- Achte darauf, dass Symbol-Kanten Zustände mit aufeinanderfolgenden Nummern verbinden!
- Speichere aktive Zustände als Bit-Vektor (ein Bit pro Zustand).
- Symbol-Transitionen können jetzt wie bei Shift-and mit << und (& mask[*c*]) durchgeführt werden.
- Epsilon-Abschlüsse werden ebenfalls als Bit-Vektoren vorberechnet.

- Schleife des Algorithmus:

  Lies das nächste Textzeichen *c*
  active' := ((*active* << 1) & mask[*c*]) | 1
  active := $OR_{a \text{ in } active'}$ EpsilonAbschluss(*a*)
  Test auf Endzustand

Laufzeit

- Vorberechnung: $O(m^2)$
- Matching: $O(nm)$ durch Bit-parallele Operationen, eigentlich kommt ein Faktor m/w hinzu (w die Integer-Wortlänge)

Bit-paralleles Matching mit dem Thompson-NFA

Verbesserung 2

- Wie kann man den Faktor *m* bei der Epsilon-Abschluss-Berechnung reduzieren?
- Berechne Epsilon-Abschluss für alle 2^z Kombinationen von *z* Zuständen!

- Schleife des Algorithmus:
  Lies das nächste Textzeichen *c*
  active' := ((*active* << 1) & mask[*c*]) | 1
  active := EpsilonAbschluss(*active'*)
  Test auf Endzustand
- Problem: Tabelle der Grösse 2^z!

Laufzeit

- Vorberechnung: $O(2^z)$
- Matching: $O(n)$,
  eigentlich kommt ein Faktor m/w hinzu (*w* die Integer-Wortlänge)

Bit-paralleles Matching mit dem Thompson-NFA

Kompromiss

- Fasse Zustände zu Blöcken der Länge k zusammen.
- Zu jedem Block b berechnet man eine Tabelle $\text{EpsilonAbschluss}_b$,
  die jeder der 2^k Zustandskombinationen in b ihren Epsilon-Abschluss zuordnet.

- Schleife des Algorithmus:
  Lies das nächste Textzeichen c
  $active'$:= (($active$ << 1) & mask[c]) | 1
  $active$:= $\text{OR}_{\text{Blöcke } b}$ $\text{EpsilonAbschluss}_b(active' |_b)$
  Test auf Endzustand
- $active' |_b$ bezeichnet $active'$, auf die k bits in b reduziert und auf $\{0,...2^k\text{-}1\}$ abgebildet.

Laufzeit

- Vorberechnung: $O(2^k m/k)$
- Matching: $O(nm/k)$,
  eigentlich kommt ein Faktor m/w hinzu (w die Integer-Wortlänge)
- Vorteil: k kann problemspezifisch gewählt werden.
  k=1 und k=z=#States=$O(m)$ sind die eben diskutierten Extremfälle.

rA-Matching mit DFAs

Erinnerung – Theoretische Informatik

- Zu jedem NFA (mit oder ohne Epsilon-Transitionen) mit *z* Zuständen gibt es einen äquivalenten DFA mit maximal 2^z Zuständen.
- Unter allen äquivalenten DFAs gibt es genau einen minimalen.

Vorgehen

- Gegeben Thompson-NFA, konstruiere äquivalenten DFA über die Teilmengenkonstruktion:
  - DFA-Zustand q $\Leftrightarrow$ Menge von NFA-Zuständen
  - Betrachte nur solche Mengen, die im NFA vom Startzustand überhaupt erreichbar sind!
  - Mengen können bit-parallel als Integer codiert werden.
  - Akzeptierende DFA-Zustände: diejenigen, deren NFA-Menge einen akzeptierenden NFA-Zustand enthält
- Wenn gewünscht, minimiere anschließend DFA (zunächst nicht nötig)

Laufzeit

- DFA-Berechnung („Kompilieren des rA“): $O(m2^z)$
- Matching: $O(n)$
- Wird der rA nicht mehrfach verwendet, lohnt sich die DFA-Berechnung eher nicht.

Die Teilmengenkonstruktion

Algorithmus

Sei NFA = (Q,A,I,F,Δ) gegeben.
E := EpsilonAbschluss(NFA)
I_{DFA} := $E(I)$ (Menge von NFA-Zuständen = 1 DFA-Zustand)
F_{DFA} := { }
Q_{DFA} := {I_{DFA}}
BuildState(I_{DFA})
return DFA = (Q_{DFA}, A, I_{DFA}, F_{DFA}, δ)

function BuildState(S)
if $S \cap F \neq \emptyset$ **then** F_{DFA} := F_{DFA} ∪ { S }
for each c in A:
  T := ∅
  for each s in S:
    for each s' in $\Delta(s,c)$: T := T ∪ $E(s')$
  $\delta(S,c)$:= T
  if $T \notin Q_{DFA}$:
    Q_{DFA} := Q_{DFA} ∪ { T }
    BuildState(T)

Abbildung der Menge S
auf eine Zustandsnummer
z.B. mit Hilfe von Hash-Funktionen