Datalogi C, RUC
Forelæsning 22. november 2004
Henning Christiansen

Syntaks og syntaksgenkendelse, særligt regulære udtryk og tilstandsmaskiner og lidt om anvendelser i bioinformatik

Dagens program

- Hvad er »sprog« i denne sammenhæng?
- Regulære udtryk — væsentligt specifikationssprog
- Tilstandsmaskiner, princippet & implementation
- Javaklasser for regulære udtryk og simpel tekstsøgning
- Konteks-fri sprog + meget kort om parsing
- Anvendelser i Bioinformatik (intro)
    - Problemstillinger
    - At finde fælles mønstre i gener
    - Kort om proteinfoldninger

Sprog: Her blot sekvenser af symboler, kun syntaktisk struktur interessant

Eksempler

```
MOVE 123 3564
JMP 111
ADD 444 @2
```

```
for(int j=0; j<= s2.length()-windowSize; j++)
  {for(int i=0; i<= s1.length()-windowSize; i++)
     {int commonChars = 0;
      for(int k=0; k<windowSize; k++)
        {if(s1.charAt(i+k)==s2.charAt(j+k)) commonChars++;}
      if(commonChars>=cuttOff) mark(i,j);}}
```

ATCGACTGTATTAGACATACGTAGATCTGTAAGA

»Syntaktisk struktur«?

Regulære udtryk, regulære sprog

»Klassisk« notation

- ε er et regulært udtryk, som beskriver den tomme streng.
- for et givet tegn, a, så er a et regulært udtryk, som beskriver strengen bestående af a.
- hvis A og B er regulære udtryk, så er AB et regulært udtryk som beskriver mængden af strenge, som fremkommer ved at tage en streng beskrevet af A og sætte sammen med en streng beskrevet af B.
- hvis A og B er regulære udtryk, så er A+B et regulært udtryk som beskriver foreningen af strenge beskrevet af A og af B.
- hvis A er et regulært udtryk, så er A* et regulært udtryk, som beskriver mængden af strenge, som forekommer ved at sammensætte 0, 1, 2, . . . strenge, hver især beskrevet af A,
- A^+ er defineret som A*, men her skal være mindst 1 forekomst.
- [A] er forkortelse for A + ε (»eventuelt A«)
- Parenteser bruges til gruppering

Eksempel:

(a* + ac)d beskriver denne mængde af strenge: {bd, abd, aabd, aaabd, aaaabd, . . . , acd}

Eksempler fra programmeringssprog:

Definer først følgende *forkortelser*

ciffer = 0 + 1 + ... + 9
bogstav = a + b + ... + å

Identifier: *bogstav*(*ciffer*+*bogstav*+_)*

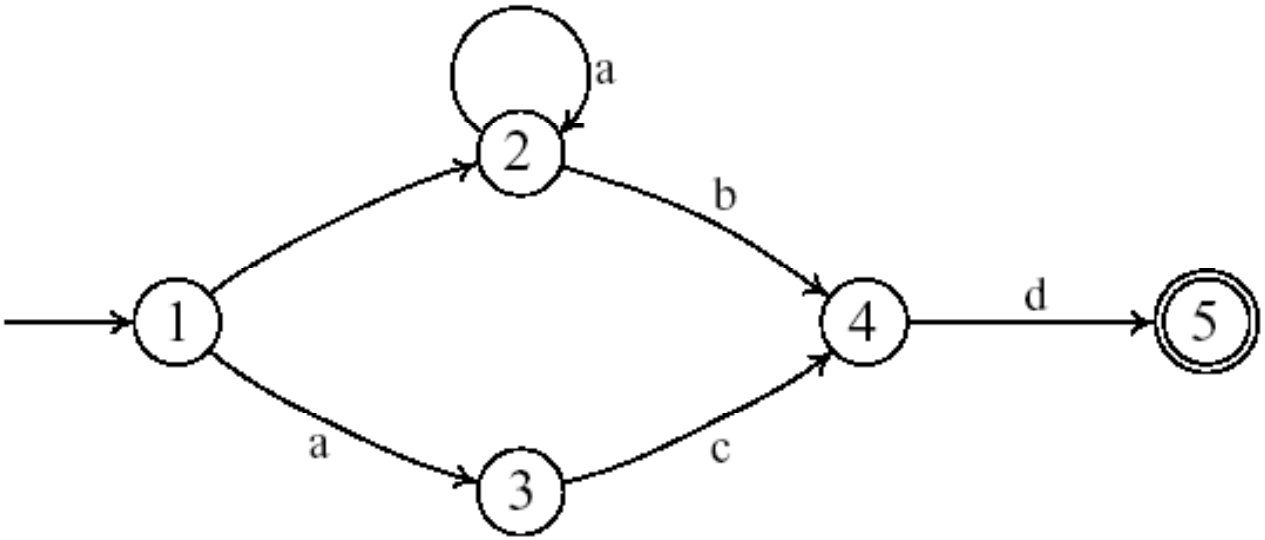
Reelt tal: [’+’ + ’–’] *ciffer*$^{+}$ [. *ciffer*$^{+}$] [e *ciffer*$^{+}$]

(Endelige) tilstandsmaskiner

Abstrakte beskrivelser til genkendelse af regulære sprog
(også: generel teknik til at holde styr på »status« for et system, f.eks. GUI, processtyring ...)

Defineret ved et endeligt antal *tilstande*, heraf én *start* og en eller flere *slut*, samt *tilstandsovergange*

Eksempel:



Genkender (a* + ac)d

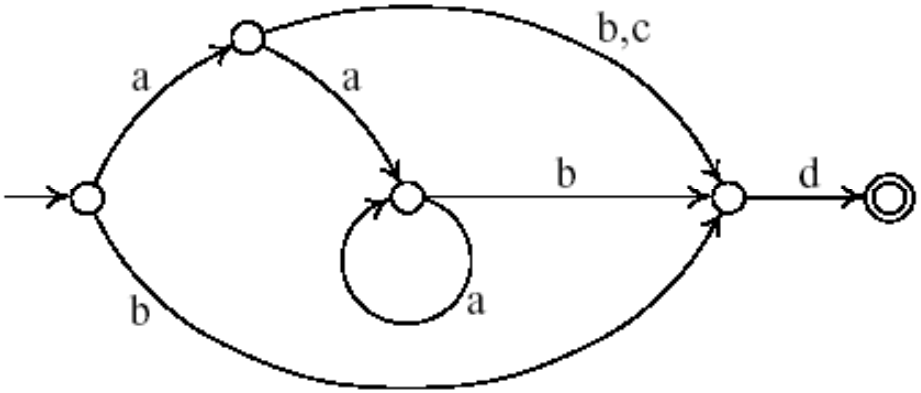
Determininistiske tilstandsmaskiner er bedre som algoritmer

Definition. En tilstandsmaskine er *ikke-deterministisk*, såfremt den har en tilstand, hvor enten
- samme tegn forekommer i etiketten på mere end én pil væk fra tilstanden, eller
- der går mere end én pil væk fra tilstanden, og en af dem har tom etikette.

En tilstandsmaskine, som ikke er ikke-deterministisk, kaldes *deterministisk*.

Etableret faktum:
Der findes algoritmer, som konverterer regulære udtryk til tilstandsmaskiner of fjerner nondeterminisme.

Eksempel: Deterministisk maskine for (a* + ac)d



Implementation af tilstandsmaskiner: 1. Generel, tabelstyret algoritme

tilstand\ tegn	*a*	*b*	*c*	*d*	EOF	andre
1	2	4	-	-	-	-
2	3	4	4	-	-	-
3	3	4	-	-	-	-
4	-	-	-	5	-	-
5	-	-	-	-	*stop*	-

```
t:= start;
repeat
        read(ch);
        t:= tabel[t, ch];
        if t = fejl then goto 1313
until t = stop ;
goto 9999
```

Implementation af tilstandsmaskiner: 2. Oversætte til kontrolstrukturer

```
1: read(ch);
case ch of
        'a': goto 2;
        'b': goto 4
otherwise goto 1313;
```

```
2: read(ch);
case ch of
        'a': goto 3;
        'b': goto 4;
        'c': goto 4
otherwise goto 1313;
```

```
3: read(ch);
case ch of
        'a': goto 3;
        'b': goto 4
otherwise goto 1313;
```

```
4: read(ch);
if ch = 'd' then goto 5 else goto 1313;
```

```
5: goto 9999;
```

Strengsøgning: et lidt andet problem

Finde bestemt streng et-eller-andet-sted i en tekst
– eller generelt, finde delstreng genereret af et givet regulær udtryk et-eller-andet-sted i en tekst

Knuth-Morris-Pratt-algoritmen

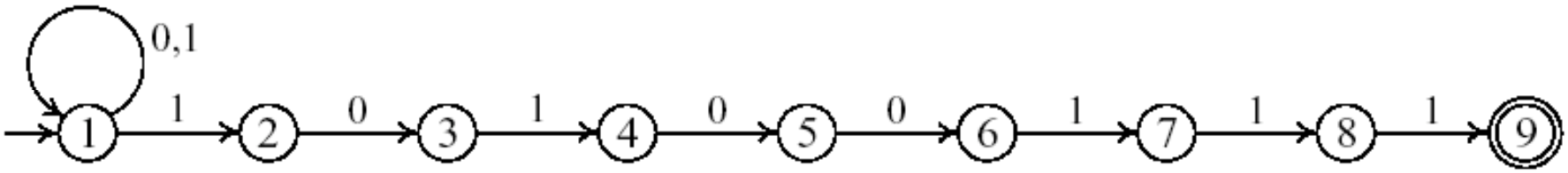
Princip:

- Givet søgestreng (eller regulært udtryk) R
- Konstruér
  1. Regulært udtryk for tekster som slutter med r, dvs. R-smart = *hvad-som-helst* * R
  2. Kør R-smart igennem standard-algoritmerne så vi får en deterministisk tilstandsmaskine

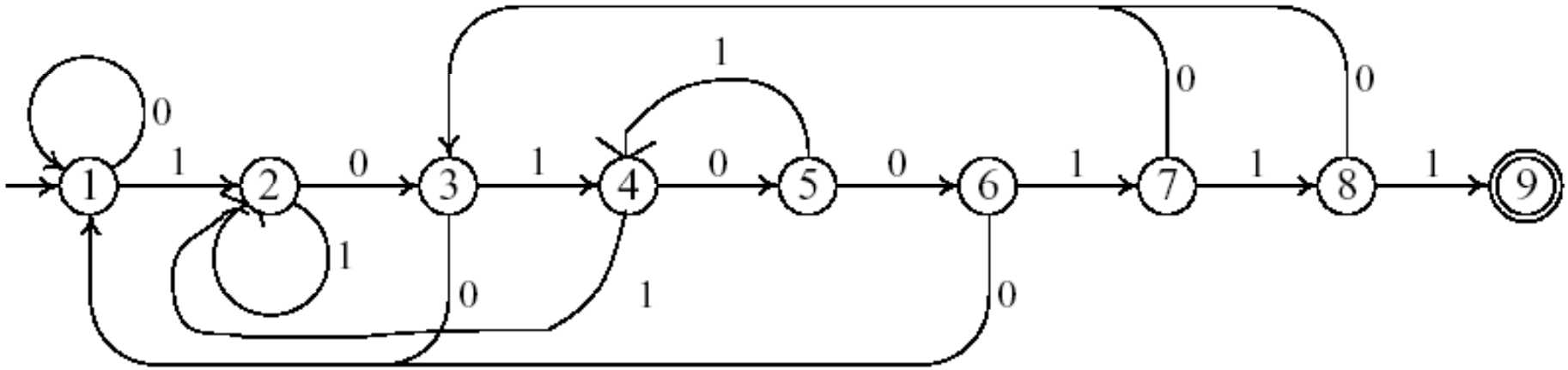
Eksempel...

Eksempel på *Knuth-Morris-Pratt-alg.*: Find forekomst af 10100111 i en tekst

Nondeterministisk tilstandsmaskine



Den tilsvarende deterministiske:



java.util.regex

http://java.sun.com/j2se/1.4.2/docs/api/index.html
Klasserne Pattern, Matcher

http://java.sun.com/j2se/1.5.0/docs/api/index.html
Klasserne Pattern, Matcher, Scanner

Kontekst-fri sprog, en mere generel klasse end de regulære

- indlejrede strukturer a la programmeringssprog { { {} {{}} } {} }
- men ikke check for korrekt type og erklæringer

Eksempel på konteks-fri grammatik for aritmetiske udtryk:

$$
\begin{aligned}
\langle udtryk \rangle &::= \quad \langle sekundært\ udtryk \rangle \\
&\quad | \quad \langle udtryk \rangle + \langle sekundært\ udtryk \rangle \\
\langle sekundært\ udtryk \rangle &::= \quad \langle tertiært\ udtryk \rangle \\
&\quad | \quad \langle sekundært\ udtryk \rangle * \langle tertiært\ udtryk \rangle \\
\langle tertiært\ udtryk \rangle &::= \quad \langle heltal \rangle \\
&\quad | \quad (\langle udtryk \rangle)
\end{aligned}
$$

NB: Udtrykker også prioritet og præcedens

Metoder til genkendelse af kontekst-fri sprog

Rekursivt nedstigende parsing

- rekursive metoder svarende til rekursion i grammatikregler
  - bygger syntaks-træer top.down
- kræver entydigt »look-ahead«
- elegant, men grammatikken må ofte vrides for at det virker

»Shift-reduce«

- Bottom-up konstruktion af syntakstræer
- formuleret ved eksplicit stak
- kræver også »look-ahead«
- ... men er væsentlig mere tolerant overfor grammatikken
- mere robust/informativ end top-down i tilfælde af syntaks-fejl

Læs selv hvis du får brug for det!

Bioinformatik, eksempler på problemstillinger, som minder om sproggenkendelse

(stærkt forenklet!!)

Proteiner, DNA, gener, genomer ...

... er informationsbærere,; kan ofte forstås udfra sekvens af aminosyrer C, G, A, T

Altså, alfabet bestående af {C, G, A, T}

Eksempel. »Hvordan nedstammer 2 fra 1?«

1. AGATCATCAT
2. CGATCATAGCAT

Mere generelt: at identificere delstrenge (=gener), som går igen i forskellige sekvenser

Problem: Givet to strenge, identificere delstrenge som optræder i begge.

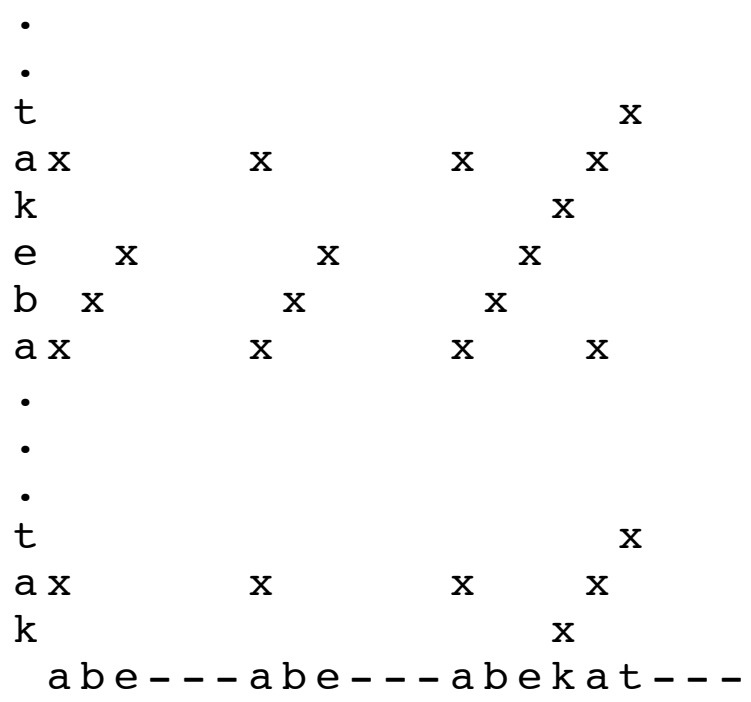
Dot-Plot-metoden:

- Lad streng 1 lege *x*-akse og streng 2 *y*-akse.
- Sæt en prik i (x,y), såfremt streng1[x] == streng[2]
- Fælles delstrenge viser sig som skå streger parallel med diagonalen

?!?!?!?!

Eksempel ....

Streng 1 ≈ x-akse: `abe---abe---abekat---`
Streng 2 ≈ y-akse: `kat...abekat...`



Smart, ikk?

Ulempe: Kræver manuel betragtning, men princippet danner udgangspunkt for (forståelse af) andre algoritmer

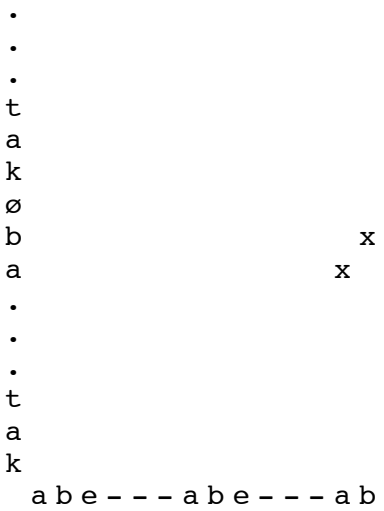
Dot-plot kan forbedres mht. støj og små mutationer

Parametre

windowSize ~ minimumslængde for at sammenfald er interessant
cutOff ~ hvor mange »rigtige« der skal være indenfor windowSize for at vi sætte prik.

Eksempel

Streng 1 ≈ x-akse: `abe---abe---abekat---`
Streng 2 ≈ y-akse: `kat...abøkat...`
windowSize = 5, cutOff = 4,



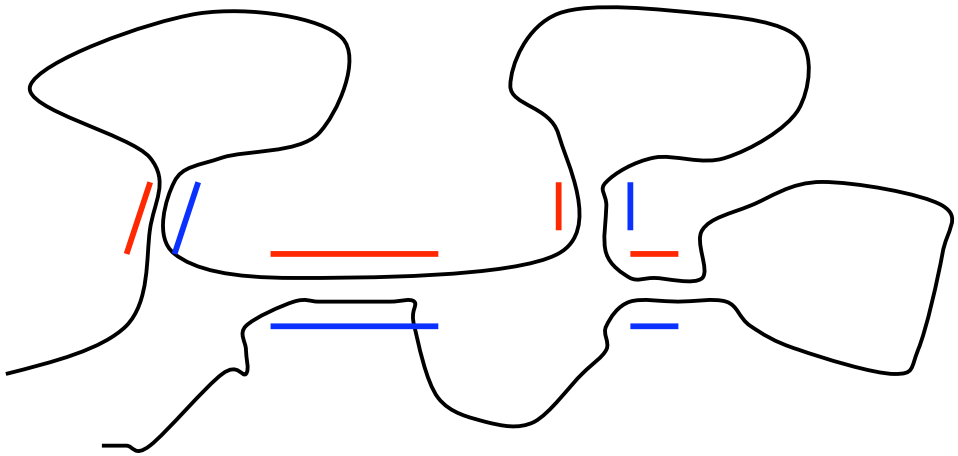
Kræver kun få linjer Java:

```
class DotPlot{
 static String s1 =  "abe---abe---abekat---";
 static String s2 =  "kat...abøkat...";
 static char [][] plot = new char[s1.length()+1][s2.length()+1];
 static final int windowSize = 5;
 static final int cuttOff = 4;

 static void initPlot()
   { plot[0][0]=' ';
     for(int i=0; i< s1.length(); i++) plot[i+1][0]=s1.charAt(i);
     for(int j=0; j< s2.length(); j++) plot[0][j+1]=s2.charAt(j);
     for(int j=1; j< s2.length(); j++)
       {for(int i=1; i< s1.length(); i++) plot[i][j]=' ';}}

 static void mark(int i, int j) {plot[i+1][j+1]='x';}

 static void printPlot()
    { for(int j=s2.length(); j>= 0; j--)
        {for(int i=0; i<= s1.length(); i++)
           {System.out.print(plot[i][j]);}
         System.out.println();}}
```

```
public static void main(String [] args){
     initPlot();
     for(int j=0; j<= s2.length()-windowSize; j++)
       {for(int i=0; i<= s1.length()-windowSize; i++)
          {int commonChars = 0;
           for(int k=0; k<windowSize; k++)
             {if(s1.charAt(i+k)==s2.charAt(j+k)) commonChars++;}
           if(commonChars>=cuttOff) mark(i,j);}}
     printPlot();}}
```

Sammenhæng mellem proteiners amino-sekvens og deres form

Forskellige delstrenge kan tiltrække hinanden (elektrisk ...), og derved opstår komplicerede faconer



Der gælder følgende lovmæssigheder:
G ⇔ C
A ⇔ T

G	**C**
A	**T**
T	**A**
A	**T**

Vi kan bruge Dot-Plot til at finde potentielle kandidater for »klisterstrenge«

Definition: Den omvendte af givet streng fås ved:

- erstat alle G med C,
- alle C med G,
- alle A med T,
- alle T med A,
- og skriv resultatet op baglæns.

For givet streng, stiller vi et Dot-Plot op for den og dens omvendte:

Eksempel:

AAAAAAAAAAAAAAAGACATAAAAAAATCTGTAAAAAAAA

⇓

TTTTTTTTAGACATTTTTTTTTCTGTTTTTTTTTTTTTTT