COS 226 Lecture 13: Pattern matching

Generalize string-searching problem to include
incompletely specified patterns

TEXT: N characters
PATTERN: M-character REGULAR EXPRESSION

Is SOME SUBSTRING of cdbcaaaaabcddbbc
in the language described by the RE (a*b + ac)d ?

Existence: Any substring in language?
Enumerate: How many substrings in language?
Search: Find a substring in language
Search all: Find all substrings in language

Pattern matching algorithm

egrep (grep -E)

Practical application of core principles
of the theory of computation

Based on simulating
NONDETERMINISTIC finite-state automata

Essential paradigm in computer science

- build intermediate abstractions
- pick the right ones!

"man grep" (excerpts)

Name

```
  grep - search file for regular expression
```

Syntax

```
  grep [option...] expression [file...]
```

Description

```
  Commands of the grep family search the
  input files (standard input default) for
  lines matching a pattern.  Normally, each
  line found is copied to the standard output.
```

Basic elements of expressions (grep)

- c   any non-special char matches itself
- r*   zero or more occurences of r

"Extended" regular expressions (egrep)

- (r)
- r1 | r2

``man grep`` (more excerpts)

Heavily used practical tool; many practical considerations

```
Take care when using the characters $ * [ ^ |
( ) and  in the expression because they are
also meaningful to the Shell. It is  safest to
enclose the entire expression argument in single
quotes ' '.
```

Ex: (now fixed in many implementations)

```
Lines longer than 256 characters are
silently truncated
```

Ex: (modern implementation)

```
See largefile(5) for the description of
the behavior of grep when encountering files
greater than or equal to 2 Gbyte.
```

Regular expressions

Theoretician: language accepted by FSA

Programmer: compact description of multiple patterns

Concatenate

abcda	abcda

OR

a+b	a b
c(a+b)d	cad cbd
(ac+b)d	acd bd
(a+c)(b+d)	ab ad cb cd

Closure

a*	. a aa aaa aaaa aaaaa
ca*b	cb cab caab caaab caaaab
(a+b)*	. a b aa ab ba bb aaa aab aba ...
c(a+b)*d	cd cad cbd caad cabd cbad cbbd ...

String searching FSAs

Recall:

- KMP algorithm: FSA for one pattern
- AC algorithm: FSA for multiple patterns
- purpose of FSA is to avoid backing up in text

RE searching is more complicated

- check each text position for RE match
- N(N-1)/2 possible matches

Ex: RE search for (a*b + ac)d in cdbcaaaaabcddbbc

Is some prefix of one of the following

- cdbcaaaaabcddbbc
- dbcaaaaabcddbbc
- bcaaaaabcddbbc
- caaaaabcddbbc
- ...

in the language described by the RE

(a*b + ac)d ?

RE FSA examples

THM: There exists an FSA for every RE

Proof: [stay tuned]

Ex: acb (string match)

.		0	1	2	3
.	a	1	1	1	3
.	b	0	0	3	3
.	c	0	2	0	3

Ex: c(a+b)d

.		0	1	2	3	4
.	a	0	3	0	0	4
.	b	0	2	0	0	4
.	c	1	1	1	1	4
.	d	0	0	3	3	4

Note: table can be huge

C program to simulate FSAs

```
struct state { char c; int next[256]; }
struct state fsa[1000];
int t = start;
while ((c = getchar()) != EOF)
   t = fsa[t].next[c];
if (t == final) printf("Accepted ");
else printf("Rejected ");
```

Different type of FSA can avoid large next table

- character in state
- one state on match
- another state on mismatch

grep implementation

Possible approach to implementing grep:

- build FSA from RE
- use C program to simulate FSA

Problems:

- FSA can be exponentially large
- how to build FSA from RE?

Nondeterministic FSAs

Appropriate intermediate abstraction for egrep

- small in size
- can build directly from RE

FSA that can guess the right answer

More than one successor state

Simplest version

- character in state with next state for match
- NULL states with TWO possible successor states

NDFSA examples

Ex: ca*b

.		0	1	2	3	4
.	ch	c		a	b	
.	next1	1	2	1	4	4
.	next2	1	3	1	4	4

Ex: c(a+b)d

.		0	1	2	3	4	5
.	ch	c		a	b	d	
.	next1	1	2	4	4	5	5
.	next2	1	3	4	4	5	5

NDFSAs and FSAs

Nondeterminism does not extend descriptive power of FSAs

THM: Given any nondeterministic FSA,
there is a deterministic FSA that recognizes
the same language.

Proof (recall 126):
one FSA state for every subset of NDFSA states

Possible approach to implement grep:

- build NDFSA from RE
- transform to FSA
- use C program to simulate FSA

Problem: FSA may be exponential in size

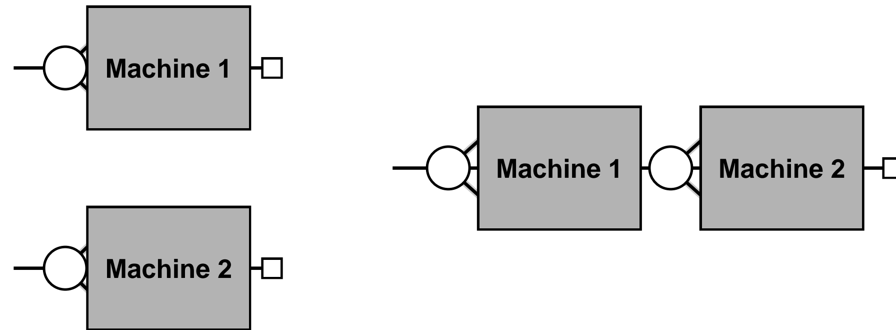
Solution: use C program to simulate NDFSA

Building a NDFSA from a regular expression

Each RE construct corresponds to a piece of the NDFSA

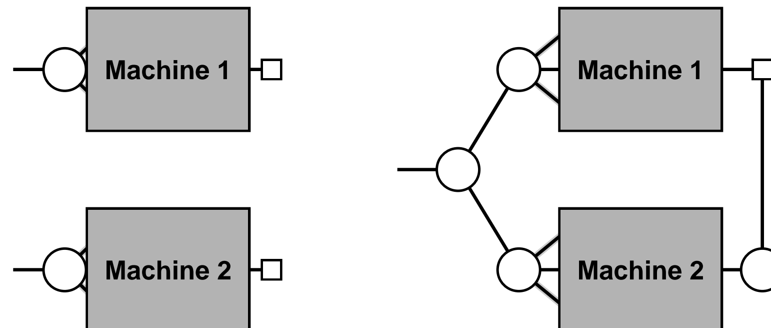
Single character:

Concatenation:



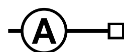
Closure (*):

OR:

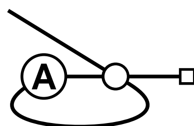


NDFSA construction example

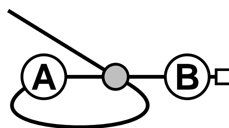
A



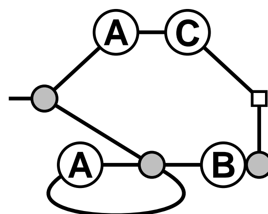
A*



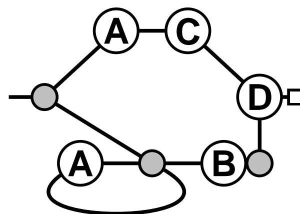
A*B



A*B+AC

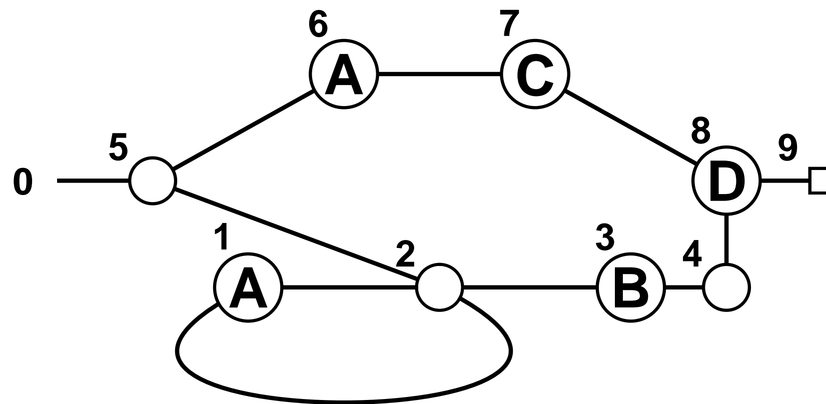


(A*B+AC)D



NDFSA representation

NDFSA for (A*B+AC)D



.		0	1	2	3	4	5	6	7	8	9
.	ch		a		b			a	c	d	
.	next1	5	2	3	4	8	6	7	8	9	0
.	next2	5	2	1	4	8	2	7	8	9	0

Recursive descent RE parser

Goal: program to build NDFSA from RE

Solve easier problem first: is RE legal?

Use context free language to describe RE

```
<expr> := <term> | <term> + <expr>
<term> := <fctr> | <fctr><term>
<fctr> := ( <expr> ) | c | ( <expr> )* | c*
```

PARSER: determine structure of legal strings

COMPILER: add semantic information to translate

Top-down recursive descent parser:

- recursive programs directly derived from CFL

Recursive descent RE parser (continued)

Expression def in CFL

```
<expr> := <term> | <term> + <expr>
```

C code for expression

```
expression()
  {
    term();
    if (p[j] == '+') { j++;  expression(); }
  }
```

Term def in CFL

```
<term> := <fctr> | <fctr><term>
```

C code for term

```
term()
  { factor();
    if ((p[j] == '[') || letter(p[j])) term(); }
  }
```

Recursive descent RE parser (continued)

Factor def in CFL

```
<fctr> := ( <expr> ) | c | ( <expr> )* | c*
```

C code for factor

```
factor()
   {
      if (p[j] == '[')
         { j++; expression();
            if (p[j] == ']') j++; else error(); }
      else if (letter(p[j])) j++; else error();
      if (p[j] == '*') j++;
   }
```

Recursive descent problem

Not as trivial as it first seems

Ex: Alternate def of expression in CFL

```
<expr> := v | <expr> + <term>
```

Corresponding C code leads to infinite recursive loop!

```
badexpression();
   {
      if (letter(p[j])) j++; else
         {
            badexpression();
            if (p[j] == '+') { j++; term(); }
            else error();
         }
   }
```

Fix: Rewrite grammar

Parsing example

RE: (A*B+AC)D

```
expression()
   term()
      factor()
         [
         expression()
            term()
               factor()  A  *
               term()
                  factor()  B
            +
            expression()
               term()
                  factor()  A
                  term()
                     factor()  C
         ]
      term()
         factor()  D
```

Recursive compiler to build NDFSA from RE

Augment parser to put out state table for NDFSA

- state: next state to be filled in
- setstate: procedure to fill in state values

Recursive routines return index of initial state

Ex: Code for expression

```
int expression()
   { int t1 = term(), t2, r = t1;
      if (p[j] == '+')
         {
            j++; state++;
            t2 = state; r = t2; state++;
            setstate(t2, ' ', expression(), t1);
            setstate(t2-1, ' ', state, state);
         }
      return r;
   }
next1[0] = expression();
setstate(0, ' ', next1[0], next1[0]);
setstate(state, ' ', 0, 0);
```

Simulating NDFSAs

Maintain a DEQUE (doubly ended queue) with

- possible states for CURRENT char
- a scan marker, *
- possible states for NEXT char

```
3   6   1   *   4   7   2
```

Take next state off beginning (stack, queue)

- null: put both states at beginning (stack)
- match: put new state at end (queue)

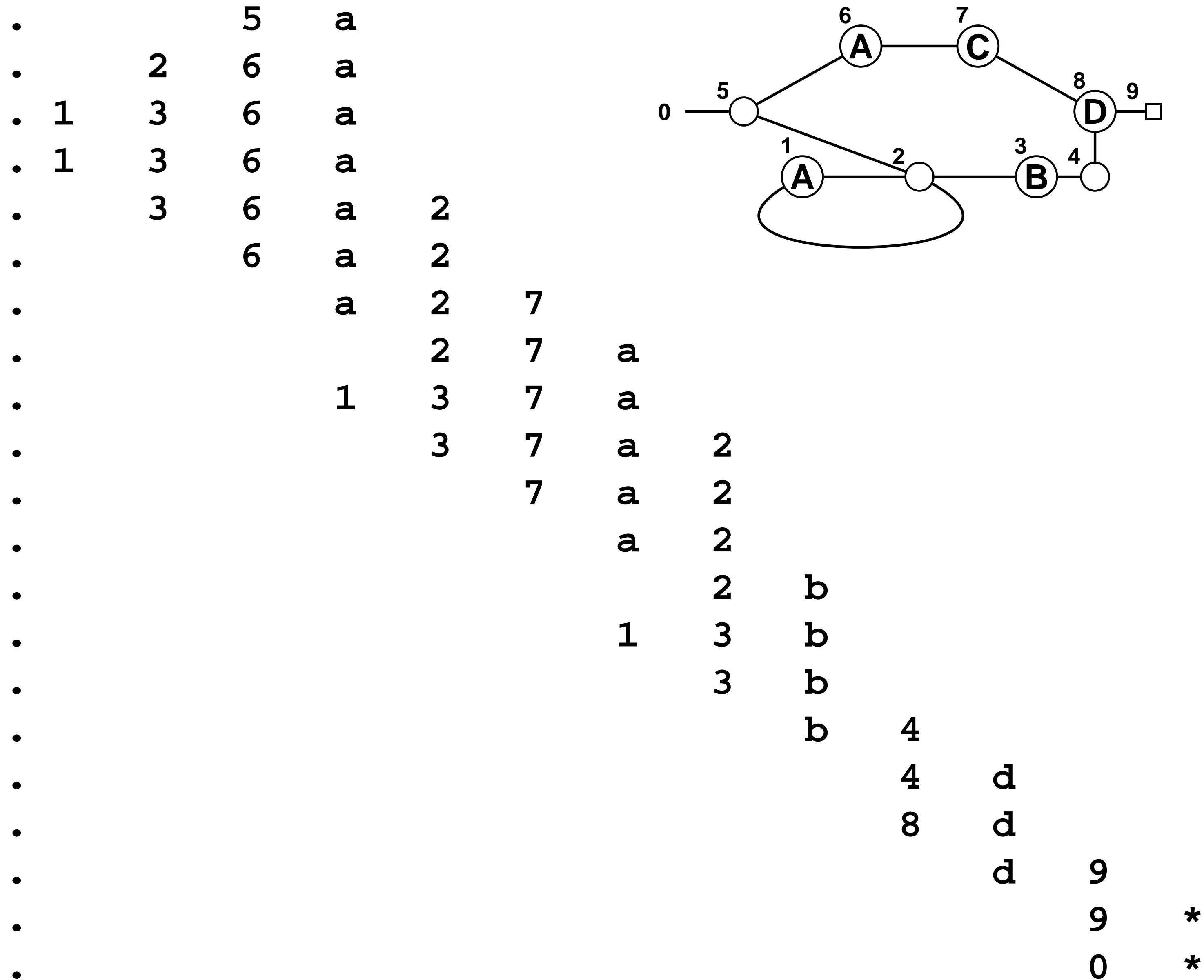
NDFSA simulation code

```
int match(char *a)
{ int j = 0, N = strlen(a), state = next1[0];
   dequeinit(); put(scan);
   while (state)
      {
         if (state == scan) { j++; put(scan); }
         else if (ch[state] == a[j])
            put(next1[state]);
         else if (ch[state] == ' ')
         { push(next1[state]); push(next2[state]); }
         if (dequeempty() || j==N) return 0;
         state = pop();
      }
   return j;
}
```

Don't bother checking next1==next2. Stay tuned.

NDFSA simulation example

NDFSA for pattern (a*b+ac)d running on text aabd



.			5	a									
.		2	6	a									
.	1	3	6	a									
.	1	3	6	a									
.		3	6	a	2								
.			6	a	2								
.				a	2	7							
.					2	7	a						
.				1	3	7	a						
.					3	7	a	2					
.						7	a	2					
.							a	2					
.								2	b				
.							1	3	b				
.								3	b				
.									b	4			
.										4	d		
.										8	d		
.											d	9	
.												9	*
.												0	*

Problem: exponential growth

Ex: NDFSA for (a*a)*b

.		0	1	2	3	4	5	6
.	ch		a		a		b	
.	next1	4	2	3	4	2	6	0
.	next2	4	2	1	4	5	6	0

simulation of machine running on aaaaaaab

```
.        4   a
.    5   2   a
.        2   a
.    1   3   a
.        3   a   2
.            a   2   4
.                2   4   a
.            1   3   4   a
.                3   4   a   2
.                    4   a   2   4
.                       ...
.                        a   2   4   2   4
.                            2   4   2   4   a
```

Easy fix: disallow duplicate states on deque

- keep "existence table", indexed by state
- also fixes next1=next2 code hack

Worst-case analysis of grep match

N: length of text

M: length of RE

Number of states in NDFSA:

- single character: 2
- concatenation AB: |A| + |B| - 1
- closure A*: |A| + 1
- OR A+B: |A| + |B| + 1

Total: less than or equal to M+1

Time for match:

- not more than M+1 states for each char
- run alg on each text position
- total not more than (M+1)(N + N-1 + N-2 + ...) = O(MN^2)
- (linear on many real problems)

Quintessential tool, classic algorithm

Context

string searching

- brute force implementation O(MN)
- KMP FSA saves a factor of M

grep implementation

- abstract machine: NDFSA
- pattern (word in CFL) --> NDFSA
- simulate NDFSA

compiler

- abstract machine: computer
- program (word in CFL) --> machine code
- run program (hardware simulation of abstract machine)

grep is a CFL -> machine code compiler!

- parser: check if pattern is in CFL
- grep: check if text is in RE

Faster algorithm than grep?

- YES, using tries (Gonnet and Baeza-Yates)