Pattern Matching

Regular expressions
FSAs
grep
nondeterministic machines
parsing

grep

generalized regular expression pattern matching:
encompass incompletely specified patterns in string search

- quintessential Unix tool
- find/replace in text editors, web search
- search in massive data sets in computational biology and other scientific applications

Approach to develop grep algorithm

- define class of abstract machines
- write simulator for machine
- write translator from REs to machines

Example of essential paradigm in computer science

- build intermediate abstractions
- pick the right ones!

Regular expressions

Natural way to describe multiple patterns in a compact manner

Concatentation

abcda	abcda

Or

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

Closure

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

Every RE defines a language: the set of all strings it describes

Regular expression pattern matching

Text with N characters
Regular expression with M characters defines language (set of patterns)

- match existence: any occurence of any pattern from language in text?
- enumerate: how many occurences?
- match: return index of any occurence ← focus of this lecture
- all matches: return indices of all occurences
-

Sample problem: find `a*f(y*t*+xy)(v+g)*ik` in

```
kvjlixapejrbxeenpphkhthbkwyrwamnugzhppfxiyjyanhapfwbghx
mshrlyujfjhrsovkvveylnbxnawavgizyvmfohigeabgksfnbkmffxj
ffqbualeytqrphyrbjqdjqavctgxjifqgfgydhoiwhrvwqbxgrixydz
bpajnhopvlamhhfavoctd[redacted]ngkwzixgjtlxkozjlefilbrboi
gnbzsudssvqymnapbpqvlubdoyxkkwhcoudvtkmikansgsutdjythzl
apawlvliygjkmxorzeoafeoffbfxuhkzukeftnrfmocylculksedgrd
```

Brute-force approach (use KMP for each pattern)?

- NO: way too many patterns

Substantially more difficult than string search?

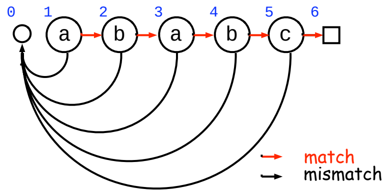
- NO (!!): theory of computation to the rescue

Finite-state automata (FSAs) for string searching

Finite-state automaton (FSA): a simple machine with M+1 states

- start in state 1
- read a text char, change to another state depending only on the text char and the current state
- continue until no more chars or states 0 or M+1 reached
- accept if in state M+1, reject if in state 0

Brute-force string search is equivalent to simulating the operation of an FSA N times, once for each text position



text	state transitions
abcdef	1 2 0
ababab	1 2 3 4 0
ababca	1 2 3 4 5 6

Knuth-Morris-Pratt string search is equivalent to a single FSA simulation

A possible approach to implementing grep

FSA view of brute-force string search:

- build FSA from pattern
- simulate operation of FSA at each text position

Possible approach to implementing grep:

- build FSA from RE
- simulate operation of FSA at each text position

Good news from theory of computation:

- there exists an FSA corresponding to any RE

Bad news:

- the FSA can be exponentially large (!)

Need more efficient abstract machines than FSAs

Nondeterministic FSAs

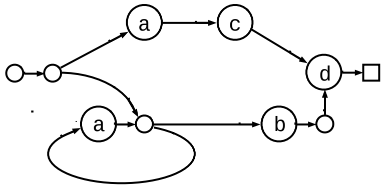
Nondeterministic FSA state

- no character
- two possible successor states: machine can choose either one

Nondeterministic FSA: an FSA with nondeterministic states

A nondeterministic FSA can guess the right answer

- can choose either successor from a nondeterministic state
- if specific choice leads to a match, NFSA will find it



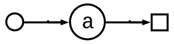
aaaaaaabd	accept
aaaaacd	reject
acd	accept
b	reject

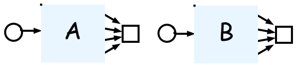
NFSAs are imaginary, but we can simulate their operation

NFSAs are as expressive as REs

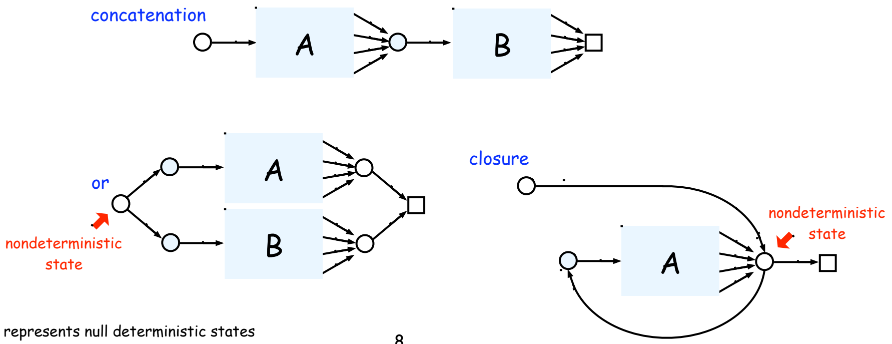
Theorem:
Given an RE, there exists an NFSA that accepts the same set of strings

Proof: [constructive, by induction]

- Base case: 

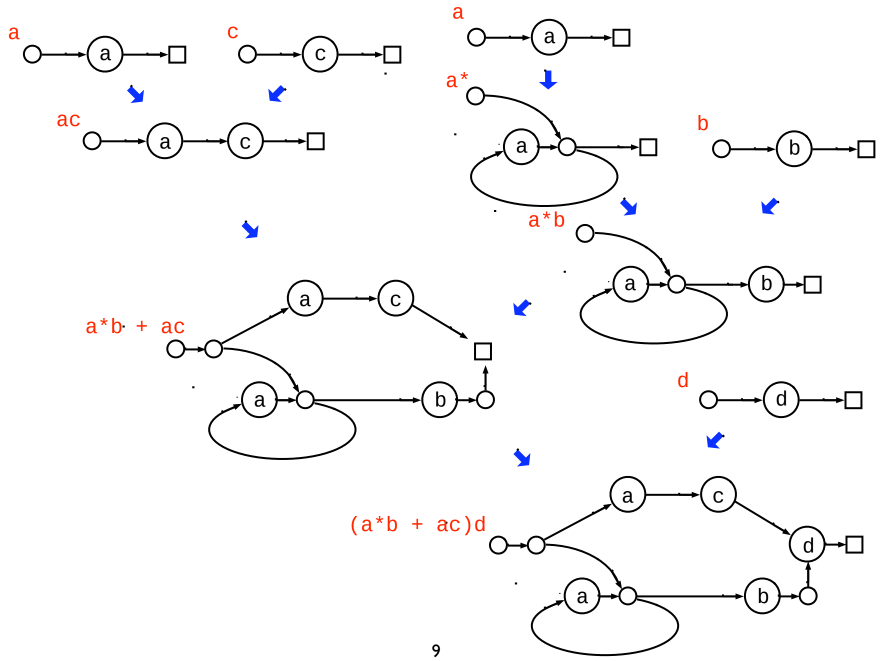
- For any NFSAs 

, use these constructions:



○ represents null deterministic states which we add or omit as convenient

Example of deriving an NFSA from an RE



grep implementation scaffolding

Input: RE, text

Output: substring in language defined by RE

Approach:

- build NFSA corresponding to RE
- simulate NFSA starting at each position in text

pattern (RE)

text

```
grep(char p[], char a[])
  { int i, j, t, N;
    state *nfsa = buildnfsa(p);
    N = strlen(a);
    for (t = 0, i = 0; i < N && !t; i++)
      t = match(nfsa, &a[i]);

    for (j = 0; j < t; j++)
      printf("%c", a[i-1+j]);
    printf("\n");
  }
```

construct NFSA from pattern

simulate NFSA at each text position

print match

NFSA representation

NFSA is an array of states

Each state is a struct with

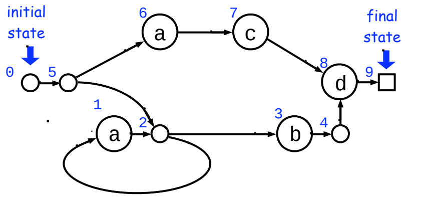
- character field ch
- one or two successor fields next

Deterministic states

- one next field specifies next state if ch matches current text char
- no match needed if ch is null

Nondeterministic states

- ch is always null
- two next fields specify two possible state transitions (no match needed)



NFSA 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			1			2				

deterministic state

nondeterministic state

```
typedef struct { char ch; int next1; int next2; } state;
state nfsa[M+1];
```

Simulating an NFSA

Idea: Keep track of all possible states for the NFSA

Same idea gives FSA for any NFSA: define state in FSA for every set of states in NFSA

Implementation: maintain a data structure with

- all possible states for current text char
- all possible states for next text char

Main loop: remove a "current-char" state

- nondeterministic: insert both next states as "current-char" states
- deterministic (match): insert next state as "next-char" state
- deterministic (mismatch): do nothing

Appropriate data structure: deque (doubly-ended queue)

- "current-char" states at front (like a stack)
- "next-char" states at back (like a queue)
- "scan" marker separating the two

```
void DQinit();
 int DQpop();
void DQpush();
void DQput();
 int DQempty();
```

combine elementary stack and queue implementations

"scan" marker

1	3	4	|	2	3	1	5	6	8

"current-char" states

"next-char" states

NFSA simulation code

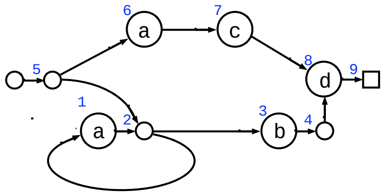
Simulating an NFSA is remarkably easy to do!

```
#define scan = ‘|’                                     simulate nfsa on string
int match(state *nfsa, char *a)
  { int st, j = 0, N = strlen(a);
    DQinit(); DQput(scan);
    for (st = nfsa[0].next1; st; st = DQpop() }
        if ((st == scan) && (DQempty() || j == N)) return 0;
        else if (st == scan)   ← move to next text char
          { j++; DQput(scan); }
        else if (nfsa[st].ch == a[j])   ← deterministic match
          DQput(nfsa[st].next1);
        else if (nfsa[st].ch == ' ')   ← nondeterministic state
          { DQpush(nfsa[st].next1); DQpush(nfsa[st].next2); }
    return j;
  }
```

NFSA simulation example

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

```
         st   action     deque contents
aabd
↑        5 | push 2 6             |
         2 | push 1 3        2  6  |
         1 | put 2        1  3  6  |
         3 | no match        3  6  |  2
         6 | put 7              6  |  2
aabd     | | put |                 |  2  7
 ↑       2 | push 1 3              2  7  |
         1 | put 2              1  3  7  |
         3 | no match              3  7  |  2
         7 | no match                 7  |  2
aabd     | | put |                       |  2
  ↑      2 | no match                       2  |
         1 | no match                    1  3  |
         3 | put 4                          3  |
aabd     | | put |                             |  4
   ↑     4 | push 8                             4  |
         8 | put 9                              8  |
aabd     | | put |                                |  9
    ↑    9 | push 0                                 9  |
         0 |                                        0  |
```



	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			1			2				

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

possible NFSA states for each text position

NFSA simulation improvement 1

Problem: Running time might be quadratic in N

Example: find pattern a*b in text aaaaaaaaaaaaaaaaaaaaaaaaa

start		cost
0	aaaaaaaaaaaaaaaaaaaaaaaaa	N
1	aaaaaaaaaaaaaaaaaaaaaaaa	N-1
2	aaaaaaaaaaaaaaaaaaaaaaa	N-2
.	.	.
.	.	.
.	.	.
N-3	aaa	3
N-2	aa	2
N-1	a	1

total ~ $N^2/2$

Solution: support .*

- wild-card . matches any char ←
- prepend .* to every search
- do just one search in grep ←
- (more work to find start of real match—search again, backwards)

```
else if (nfsa[st].ch == ‘.’)
  DQput(nfsa[st].next1);
```

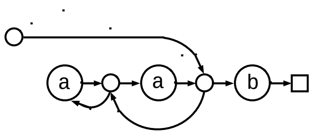
```
t = match(nfsa, a)
```

NFSA simulation improvement 2

Problem: Running time might be exponential in N (!!)

Example: NFSA for pattern (a*a)*b running on text aaaaaab

```
          st   action     deque contents
aaaab
↑         4 | push 5 2          a
          5 | no match    5  2  a
          2 | push 1 3       2  a
          1 | put 2       1  3  a
          3 | put 4          3  a  2
aaaab     | | put |             a  2  4
 ↑        2 | push 1 3             2  4  a
          1 | put 2             1  3  4  a
          3 | put 4                3  4  a  2
          4 | push 5 2                4  a  2  4
          5 | no match             5  2  a  2  4
          2 | push 1 3                2  a  2  4
          1 | put 2                1  3  a  2  4
          3 | put 4                   3  a  2  4  2
                                         a  2  4  2  4
aaaab
  ↑
```



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

```
a  2  4  2  4  2  4  2  4
```

size doubles for each a scanned

Easy solution: disallow duplicate states on deque

A quick overview of parsing

A language is a set of strings

A grammar is a metalanguage for specifying languages

- terminal symbols
- nonterminal symbols
- set of replacement rules

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

A parse of a string is a sequence of a grammar's replacement rules proving that the string is in the language specified by the grammar

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

A compiler is a program that parses a string for the purpose of translating it into another language

A program to build an NFSA from an RE is essentially a compiler

NFSA construction step 0: concatenation

NFSA construction for simple text strings

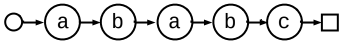
```
state *nfsa;      
int st;    
void makestate(char c, int n1, int n2)
  { nfsa[st].ch = c; nfsa[st].next1 = n1; nfsa[st].next2 = n2; st++; }
state *buildnfsa(char p[])
  { int i, M = strlen(p);
    nfsa = malloc((M+2)*sizeof(state)); st = 0;
    makestate(' ', 1, 1);
    for (i = 0; i < M; i++)
      makestate(p[i], st+1, st+1);
    makestate(' ', 0, 0);
    return nfsa;
  }
```

array of states

index of current state

Key point: st is a GLOBAL variable, incremented on each call to makestate

NFSA for ababc



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

i = 3

NFSA construction step 1: add closure

To handle *, need to reset successor of states created earlier

```
void resetsucc(int i, int next)
  {
    if (nfsa[i].next1 == nfsa[i].next2)
      nfsa[i].next1 = next;
    nfsa[i].next2 = next;
  }
```

reset both nexts for deterministic states

```
0  1  2  3  4  5  6  7
   a  b
1  2  3
```

Replace `makestate(p[i], st+1, st+1)` in step 0 with:

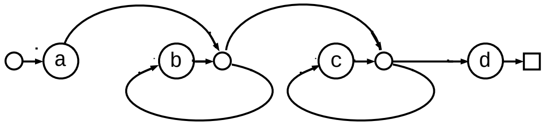
```
if (p[i] == '*')
  {
    makestate(' ', st-1, st+1);
    resetsucc(st-3, st-1);
  }
else makestate(p[i], st+1, st+1);
```

```
   a  b
1  3  3  2             ab*c*d
         4

   a  b     c
1  3  3  2  5
         4

   a  b     c          ab*c*d
1  3  3  2  5  4
         5     6
```

NFSA for ab*c*d



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

NFSA construction step 2: add parenthesized or

To handle (...+...), need to extend if statement to remember states and fill in successors later

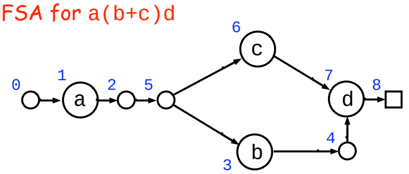
```
if (p[i] == '*')
  { makestate(' ', st-1, st+1); resetsucc(st-3, st-1); }
else if (p[i] == '(')
  { left = st; makestate(' ', 0, 0);}
else if (p[i] == '+')
  { plus = st; makestate(' ', 0, 0);
    resetsucc(left, st);
    makestate(' ', left+1, st+1);
  }
else if (p[i] == ')')
  resetsucc(plus, st);
else makestate(p[i], st+1, st+1);
```

```
0  1  2  3  4  5  6  7  8
         left
   a                        a(b+c)d
1  2  0

                  plus
   a     b                  a(b+c)d
1  2  5  4  0  3
               6

   a     b                  a(b+c)d
1  2  5  4  7  3  c
               6  7
```

NFSA for a(b+c)d



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

Full NSFA construction

To complete implementation

- add parenthesized *, unparenthesized +: `a(b+c)*d, ab+bc+de ...`
- check for errors: `a(b+cd(e+f), a(*b), +ab(), ...`
- allow nested parentheses: `a(b+cd(e*f+fg))h, ...`

use systematic approach (details in Sedgewick, second edition)

- use context-free grammar to formally specify legal REs

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

context-free grammar:
all rules have single nonterminal on lhs

- use recursive descent (mutually recursive functions) to build nfsa

```
int expr()
  { int plus, left = term();
    if (p[j] != '+') return left; else
      {
        j++; plus = st;
        makestate(' ', 0, 0);
        resetsucc(plus, expr());
        makestate(' ', st, st);
      }
  }
```

but simpler implementations do often suffice: `(grep, egrep, fgrep, ...)`

grep running time

Theorem: The cost of determining whether any substring of an N-char text is in the language defined by an M-char RE is O(MN).

Proof: Let |A| be the number of states in the NFSA for the RE A

(not counting null deterministic states)

- NSFA size is O(M)
  - single character: |x| = 1
  - concatenation: |AB| = |A| + |B|
  - closure: |A*| = |A| + 1
  - or: |A+B| = |A| + |B| + 1
  - Total states for M-char RE: M + O(M) null deterministic states
- Simulation cost is O(MN)
  - not more than M states for each text char ← assuming use of .* (see nfsa simulation improvement 1)

Surprising bottom line:

Worst case cost for grep is the same as for elementary string match!

grep summary

Solves important practical problem

- elegant, efficient, extensible

Demonstrates importance of theory

- power of abstraction
- which problems are truly difficult?

simulate nfsa

```
int match(state *nfsa, char a[])
  { int N = strlen(a);
    int st, j = 0;
    DQinit(); DQput(scan);
    for (st = nfsa[0].next1; st; st = DQpop())
      if (st == scan)
        { if (DQempty() || j == N) return 0;
          j++; DQput(scan); }
      else if (nfsa[st].ch == a[j])
        DQput(nfsa[st].next1);
      else if (nfsa[st].ch == ' ')
        { DQpush(nfsa[st].next1);
          DQpush(nfsa[st].next2); }
    return j;
  }
```

represent nfsa

```
typedef
  struct { char ch; int next1; int next2; } state;
```

build nfsa

```
int st;
state *nfsa;
void makestate(char c, int n1, int n2)
  { nfsa[st].ch = c; nfsa[st].next1 = n1;
    nfsa[st].next2 = n2; st++; }
void resetsucc(int i, int next)
  {
    if (nfsa[i].next1 == nfsa[i].next2)
      nfsa[i].next1 = next;
    nfsa[i].next2 = next;
  }
state *buildnfsa(char p[])
  { int i, left, plus, M = strlen(p);
    nfsa = malloc((M+2)*sizeof(state)); st = 0;
    makestate(' ', 1, 1);
    for (i = 0; i < M; i++)
      if (p[i] == '*')
        {  makestate(' ', st-1, st+1);
           resetsucc(st-3, st-1); }
      else if (p[i] == '(')
        { left = st; makestate(' ', 0, 0); }
      else if (p[i] == '+')
        { plus = st;  makestate(' ', 0, 0);
          resetsucc(left, st);
          makestate(' ', left+1, st+1);
        }
      else if (p[i] == ')')
        resetsucc(plus, st);
      else makestate(p[i], st+1, st+1);
    makestate(' ', 0, 0);
    return nfsa;
  }
```

Context

Abstract machines, languages, and nondeterminism

- are the basis of the theory of computation
- have been intensively studied since the 1930s

Chomsky hierarchy progresses from FSAs to Turing machines

machine	language	nondeterministic version more powerful?	 more efficient?
FSA	RE	no	yes → grep
pushdown FSA	context-free	yes	
bounded TM	context-sensitive	unknown	unknown
Turing machine	any replacement	no	???

↖ P vs NP problem

Why study imaginary machines?

- virtually all machines are imaginary
- can simulate imaginary machines with real ones