Pattern Matching

Regular expressions

FSAs
grep

nondeterministic machines

parsing

Regular expression pattern matching

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!

Natural way to describe multiple patterns in a compact manner

**Concatentation**

```
ab cd a
```

**Or**

```
a+b a b
c(a+b)d cad cbd
```

**Closure**

```
ap a aa aaa aaaa aaaaa ...
ca*b cb cab caab caaab ...
(a+b)* a b aa ab ba bb aaa aab aba abb baa ...
c(a+b)*d cd cad cbd caad cbdd cbbd cadd ...
```

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

<table>
<thead>
<tr>
<th>Text</th>
<th>State Transitions</th>
</tr>
</thead>
<tbody>
<tr>
<td>ababcdef</td>
<td>1 2 0</td>
</tr>
<tr>
<td>ababab</td>
<td>1 2 3 4 0</td>
</tr>
<tr>
<td>ababca</td>
<td>1 2 3 4 5 6</td>
</tr>
</tbody>
</table>

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

**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

NFSAs are imaginary, but we can simulate their operation

**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

**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:

- concatenation
- closure
- nondeterministic state

represents null deterministic states which we add or omit as convenient
Example of deriving an NFSA from an RE

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)

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

Simulating an NFSA

Idea: Keep track of all possible states for the 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

typedef struct { char ch; int next1; int next2; } state;
state nfsa[M+1];
Simulating an NFSA is remarkably easy to do!

```c
#define scan = '|'  
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)
        {
            j++; DQput(scan);
        }
        else if (nfsa[st].ch == a[j])
        {
            DQpush(nfsa[st].next1);
        }
        else if (nfsa[st].ch == '
        { DQpush(nfsa[st].next1); DQpush(nfsa[st].next2); }
        return j;
    }
}
```

**NFSA simulation example**

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

```
<table>
<thead>
<tr>
<th>st</th>
<th>action</th>
<th>deque contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>aabd</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aabd</td>
<td>push 2 6</td>
<td></td>
</tr>
<tr>
<td>aabd</td>
<td>push 1 3</td>
<td></td>
</tr>
<tr>
<td>aabd</td>
<td>put 2</td>
<td></td>
</tr>
<tr>
<td>aabd</td>
<td>no match</td>
<td></td>
</tr>
<tr>
<td>aabd</td>
<td>put 7</td>
<td></td>
</tr>
<tr>
<td>aabd</td>
<td>put 1</td>
<td></td>
</tr>
<tr>
<td>aabd</td>
<td>put 2</td>
<td></td>
</tr>
<tr>
<td>aabd</td>
<td>no match</td>
<td></td>
</tr>
<tr>
<td>aabd</td>
<td>put 1</td>
<td></td>
</tr>
<tr>
<td>aabd</td>
<td>put 2</td>
<td></td>
</tr>
<tr>
<td>aabd</td>
<td>put 1</td>
<td></td>
</tr>
<tr>
<td>aabd</td>
<td>put 2</td>
<td></td>
</tr>
<tr>
<td>aabd</td>
<td>put 8</td>
<td></td>
</tr>
<tr>
<td>aabd</td>
<td>put 9</td>
<td></td>
</tr>
<tr>
<td>aabd</td>
<td>put 0</td>
<td></td>
</tr>
</tbody>
</table>
```

**NFSA simulation improvement 1**

**Problem:** Running time might be quadratic in \(N\)

**Example:** find pattern \(a^*b\) in text `aaaaaaaaaaaaaaaaaaaaaa`

<table>
<thead>
<tr>
<th>start</th>
<th>cost</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N</td>
<td>push 6</td>
</tr>
<tr>
<td>1</td>
<td>N-1</td>
<td>push 1</td>
</tr>
<tr>
<td>2</td>
<td>N-2</td>
<td>push 2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>N-3</td>
<td></td>
<td>push 4</td>
</tr>
<tr>
<td>N-2</td>
<td></td>
<td>push 5</td>
</tr>
<tr>
<td>N-1</td>
<td></td>
<td>push 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push 4</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>push 4</td>
</tr>
</tbody>
</table>

**Solution:** support \(*\)

- wild-card \(.*\) matches any char
- prepend \(.*\) to every search
- do just one search in grep

**NFSA simulation improvement 2**

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

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

```
<table>
<thead>
<tr>
<th>st</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaaaab</td>
<td></td>
</tr>
<tr>
<td>aaaaab</td>
<td>push 5 2</td>
</tr>
<tr>
<td>aaaaab</td>
<td>no match</td>
</tr>
<tr>
<td>aaaaab</td>
<td>push 1 3</td>
</tr>
<tr>
<td>aaaaab</td>
<td>push 2</td>
</tr>
<tr>
<td>aaaaab</td>
<td>put 2</td>
</tr>
<tr>
<td>aaaaab</td>
<td>no match</td>
</tr>
<tr>
<td>aaaaab</td>
<td>push 1 3</td>
</tr>
<tr>
<td>aaaaab</td>
<td>push 2</td>
</tr>
<tr>
<td>aaaaab</td>
<td>put 2</td>
</tr>
<tr>
<td>aaaaab</td>
<td>no match</td>
</tr>
<tr>
<td>aaaaab</td>
<td>push 1 3</td>
</tr>
<tr>
<td>aaaaab</td>
<td>push 2</td>
</tr>
<tr>
<td>aaaaab</td>
<td>put 4</td>
</tr>
<tr>
<td>aaaaab</td>
<td>push 5 2</td>
</tr>
<tr>
<td>aaaaab</td>
<td>push 2</td>
</tr>
<tr>
<td>aaaaab</td>
<td>push 1</td>
</tr>
<tr>
<td>aaaaab</td>
<td>put 2</td>
</tr>
<tr>
<td>aaaaab</td>
<td>put 4</td>
</tr>
</tbody>
</table>
```

**Easy solution:** disallow duplicate states on deque

```
<table>
<thead>
<tr>
<th>st</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaaaab</td>
<td></td>
</tr>
<tr>
<td>aaaaab</td>
<td>push 5 2</td>
</tr>
<tr>
<td>aaaaab</td>
<td>no match</td>
</tr>
<tr>
<td>aaaaab</td>
<td>push 1 3</td>
</tr>
<tr>
<td>aaaaab</td>
<td>push 2</td>
</tr>
<tr>
<td>aaaaab</td>
<td>put 2</td>
</tr>
<tr>
<td>aaaaab</td>
<td>no match</td>
</tr>
<tr>
<td>aaaaab</td>
<td>push 1 3</td>
</tr>
<tr>
<td>aaaaab</td>
<td>push 2</td>
</tr>
<tr>
<td>aaaaab</td>
<td>put 4</td>
</tr>
</tbody>
</table>
```

Size doubles for each \(a\) scanned.
A language is a set of strings

A grammar is a metalanguage for specifying languages

- terminal symbols
- nonterminal symbols
- set of replacement rules

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

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

A quick overview of parsing

NFSA construction for simple text strings

state *nfsa; array of states
int st; index of current state
void makestate(char c, int n1, int n2)
{
    nfsa[st].ch = c;
    nfsa[st].next1 = n1;
    nfsa[st].next2 = n2;
    st++;
}

NFSA construction step 0: concatenation

NFSA for ababc

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;
}

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);

NFSA for ab*c*d

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

NFSA for a(b+c)d

NFSA construction for simple text strings

state *nfsa; array of states
int st; index of current state
void makestate(char c, int n1, int n2)
{
    nfsa[st].ch = c;
    nfsa[st].next1 = n1;
    nfsa[st].next2 = n2;
    st++;
}

NFSA construction step 0: concatenation

NFSA for ababc

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;
}

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);

NFSA for ab*c*d

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);
    resetsucc(left, st);
    makestate('(', left+1, st+1);
}
else if (p[i] == '+')
{
    plus = st;
    makestate('(', 0, 0);
    resetsucc(plus, st);
    makestate('(', left+1, st+1);
}
else makestate(p[i], st+1, st+1);
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
- use recursive descent (mutually recursive functions) to build nfsa

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

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

represent nfsa

int match(state *nfsa, char a[])
{ int M = strlen(a);
  int st, i = 0;
  DQinit(); DQput(scan);
  for (st = nfsa[0].next1; st; st = DQpop())
    if (a[i++] == nfsa[st].ch)
      nfsa[st].next1 = nfsa[st].next1 + 1;
  for (i = 0; i < M; i++)
    if (a[i] == ')')
      nfsa[st].next2 = nfsa[st].next2 + 1;
  return st;
}

build nfsa

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

context-free grammar:  
<expr> := <term> | <term> + <expr>
<term> := <fctr> | <fctr><term>
<fctr> := ( <expr> ) | c | ( <expr> )* | c*

build nfsa(char p[])
{ int i, left, plus, M = strlen(p);
  nfsa = malloc((M+2)*sizeof(state)); st = 0;
  for (i = 0; i < M; i++)
    if (p[i] == ')')
      nfsa[st].next2 = nfsa[st].next2 + 1;
    else if (p[i] == '+')
      nfsa[st].next1 = nfsa[st].next1 + 1;
    else if (p[i] == ' ')
      st = DQpop(nfsa[st].next1);
    else if (p[i] == c)
      DQpush(nfsa[st].next1);
  return nfsa;
}

simulate nfsa

int match(state *nfsa, char a[])
{ int M = strlen(a);
  int st = 0;
  DQinit(); DQput(scan);
  for (st = nfsa[0].next1; st; st = DQpop())
    if (a[i++] == nfsa[st].ch)
      nfsa[st].next1 = nfsa[st].next1 + 1;
  return st;
}

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

- 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

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?

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

<table>
<thead>
<tr>
<th>machine</th>
<th>language</th>
<th>nondeterministic version</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSA</td>
<td>RE</td>
<td>more powerful?</td>
</tr>
<tr>
<td>pushdown FSA</td>
<td>context-free</td>
<td>yes</td>
</tr>
<tr>
<td>bounded TM</td>
<td>context-sensitive</td>
<td>unknown</td>
</tr>
<tr>
<td>Turing machine</td>
<td>any replacement</td>
<td>no</td>
</tr>
</tbody>
</table>

Why study imaginary machines?
- virtually all machines are imaginary
- can simulate imaginary machines with real ones