5.4 Pattern Matching

- regular expressions
- REs and NFAs
- NFA simulation
- NFA construction
- applications

Pattern matching

- Substring search. Find a single string in text.
- Pattern matching. Find one of a specified set of strings in text.

Ex. [genomics]
- Fragile X syndrome is a common cause of mental retardation.
- Human genome contains triplet repeats of CGG or AGG, bracketed by CGG at the beginning and CTG at the end.
- Number of repeats is variable, and correlated with syndrome.

<table>
<thead>
<tr>
<th>pattern</th>
<th>text</th>
</tr>
</thead>
<tbody>
<tr>
<td>GCC(CG</td>
<td>AGG)*CTG</td>
</tr>
</tbody>
</table>

Pattern matching: applications

- Test if a string matches some pattern.
  - Process natural language.
  - Scan for virus signatures.
  - Access information in digital libraries.
  - Filter text (spam, NetNanny, Carnivore, malware).
  - Validate data-entry fields (dates, email, URL, credit card).
  - Search for markers in human genome using PROSITE patterns.

- Parse text files.
  - Compile a Java program.
  - Crawl and index the Web.
  - Read in data stored in ad hoc input file format.
  - Automatically create Java documentation from Javadoc comments.
Regular expressions

A regular expression is a notation to specify a (possibly infinite) set of strings. A “language”

<table>
<thead>
<tr>
<th>operation</th>
<th>example RE</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>concatenation</td>
<td>AAABAB</td>
<td>AAABAB</td>
<td>every other string</td>
</tr>
<tr>
<td>or</td>
<td>AA</td>
<td>BAAB</td>
<td>AA BAAB</td>
</tr>
<tr>
<td>closure</td>
<td>AB*A</td>
<td>AA ABBBABBBA</td>
<td>AB ABABA</td>
</tr>
<tr>
<td>parentheses</td>
<td>A(A</td>
<td>B)AAB</td>
<td>AAABABABABA</td>
</tr>
<tr>
<td>(A</td>
<td>B)*A</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>

Regular expression shortcuts

Additional operations are often added for convenience.

Ex. \([a-z]++\) is shorthand for \((A|B|C|D|E)(A|B|C|D|E)\) *

<table>
<thead>
<tr>
<th>operation</th>
<th>example RE</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>wildcard</td>
<td>.O..O.</td>
<td>CUMULUS</td>
<td>SUCCUBUS</td>
</tr>
<tr>
<td>at least 1</td>
<td>A(BC)+DE</td>
<td>ABCDE</td>
<td>ADE</td>
</tr>
<tr>
<td>character classes</td>
<td>[^A-EIOU]{6}</td>
<td>RHYTHM</td>
<td>DECADE</td>
</tr>
<tr>
<td>exactly k</td>
<td>[0-9]{5}-[0-9]{4}</td>
<td>08540-1321</td>
<td>111111111 166-54-111</td>
</tr>
<tr>
<td>complement</td>
<td>[*]A[0-9]{6}</td>
<td>RHYTHM</td>
<td>DECADE</td>
</tr>
</tbody>
</table>

Regular expression examples

Notation is surprisingly expressive

<table>
<thead>
<tr>
<th>regular expression</th>
<th>matches</th>
<th>does not match</th>
</tr>
</thead>
<tbody>
<tr>
<td>.<em>SPB.</em> (contains the trigraph spb)</td>
<td>RASPBERRY CRISPSPREAD</td>
<td>SUBSPACE SUBSPACER</td>
</tr>
<tr>
<td>[0-9]{3}-[0-9]{2}-[0-9]{4} (Social Security numbers)</td>
<td>166-11-4433 166-45-1111</td>
<td>11-55555555 8675309</td>
</tr>
<tr>
<td>[a-z]{3}@[a-z.].(edu</td>
<td>com) (valid email addresses)</td>
<td><a href="mailto:wayne@princeton.edu">wayne@princeton.edu</a> <a href="mailto:ra@princeton.edu">ra@princeton.edu</a> spam@nowhere</td>
</tr>
<tr>
<td>[\w_:a-zA-Z-0-9]* (valid Java identifiers)</td>
<td>Ident3 PatternMatcher ident#3</td>
<td></td>
</tr>
</tbody>
</table>

and plays a well-understood role in the theory of computation.

Regular expressions to the rescue

Regular expressions to the rescue

http://xkcd.com/208/
Can the average web surfer learn to use REs?

Google. Supports * for full word wildcard and | for union.

Regular expression caveat

Writing a RE is like writing a program.
• Need to understand programming model.
• Can be easier to write than read.
• Can be difficult to debug.

"Some people, when confronted with a problem, think ‘I know I’ll use regular expressions.’ Now they have two problems."
— Jamie Zawinski (flame war on alt.religion.emacs)

Bottom line. REs are amazingly powerful and expressive, but using them in applications can be amazingly complex and error-prone.

Can the average programmer learn to use REs?

Perl RE for valid RFC822 email addresses

http://www.ex-parrot.com/~pdw/Mail-RFC822-Address.html
Pattern matching implementation: basic plan (first attempt)

Overview is the same as for KMP!
- No backup in text input stream.
- Linear-time guarantee.

Underlying abstraction. Deterministic finite state automata (DFA).

Basic plan.
- Build DFA from RE.
- Simulate DFA with text as input.

Bad news. Basic plan is infeasible (DFA may have exponential number of states).

Pattern matching implementation: basic plan (revised)

Overview is similar to KMP.
- No backup in text input stream.
- Quadratic-time guarantee (linear-time typical).

Underlying abstraction. Nondeterministic finite state automata (NFA).

Basic plan.
- Build NFA from RE.
- Simulate NFA with text as input.

Nondeterministic finite-state automata

Pattern matching NFA.
- Pattern enclosed in parentheses.
- One state per pattern character (start = 0, accept = M).
- Red $\epsilon$-transition (change state, but don’t scan input).
- Black match transition (change state and scan to next char).
- Accept if any sequence of transitions ends in accept state.

Nondeterminism.
- One view: machine can guess the proper sequence of state transitions.
- Another view: sequence is a proof that the machine accepts the text.
Nondeterministic finite-state automata

**Ex.** Is `aaaaab` matched by NFA?

![Diagram of NFA for pattern `(A*B|AC)D`]

Note: any sequence of legal transitions that ends in state 11 is a proof.

**Ex.** Is `aaaaac` matched by NFA?

![Diagram of NFA for pattern `(A*B|AC)D`]

Note: this is not a complete proof! (need to mention the infinite number of sequences involving ε-transitions between 2 and 3)

Nondeterminism

**Q.** How to determine whether a string is recognized by an automaton?

**DFA.** Deterministic ⇒ exactly one applicable transition.

**NFA.** Nondeterministic ⇒ can be several applicable transitions; need to select the right one!

**Q.** How to simulate NFA?

**A.** Systematically consider all possible transition sequences.

Pattern matching implementation: basic plan (revised)

Overview is similar to KMP.

- No backup in text input stream.
- Quadratic-time guarantee (linear-time typical).

Underlying abstraction. Nondeterministic finite state automata (NFA).

Basic plan.

- Build NFA from RE.
- Simulate NFA with text as input.
NFA representation

State names. Integers from 0 to M.

Match-transitions. Keep regular expression in array re[].

$\epsilon$-transitions. Store in a digraph $G$.

- $0 \rightarrow 1$, $1 \rightarrow 2$, $1 \rightarrow 6$, $2 \rightarrow 3$, $3 \rightarrow 2$, $3 \rightarrow 4$, $5 \rightarrow 8$, $8 \rightarrow 9$, $10 \rightarrow 11$

NFA corresponding to the pattern $( ( A \ast B | A C ) D )$

NFA simulation

Q. How to efficiently simulate an NFA?

A. Maintain set of all possible states that NFA could be in after reading in the first i text characters.

Q. How to perform reachability?

Digraph reachability

Find all vertices reachable from a given set of vertices.

```java
public class DFS {
    private SET<Integer> marked;
    private Digraph G;

    public DFS(Digraph G) {
        this.G = G;
    }

    private void search(int v) {
        marked.add(v);
        for (int w : G.adj(v))
            if (!marked.contains(w)) search(w);
    }

    public SET<Integer> reachable(SET<Integer> s) {
        marked = new SET<Integer>();
        for (int v : s) search(v);
        return marked;
    }
}
```
NFA simulation: Java implementation

```java
public boolean recognizes(String txt) {
    DFS dfs = new DFS(G);
    SET<Integer> pc = new dfs.reachable(0);
    for (int i = 0; i < txt.length(); i++) {
        SET<Integer> match = new SET<Integer>();
        for (int v : pc) {
            if (v == M) continue;
            if ((txt.charAt(i) == txt.charAt(i) || txt.charAt(i) == '.')
                match.add(v+1);
        }
        pc = dfs.reachable(match);
    }
    return pc.contains(M);
}
```

Proposition 1. Determining whether an N-character text string is recognized by the NFA corresponding to an M-character pattern takes time proportional to NM in the worst case.

Pf. For each of the N text characters, we iterate through a set of states of size no more than M and run DFS on the graph of \( \epsilon \)-transitions. (The construction we consider ensures the number of edges is at most M.)
Building an NFA corresponding to an RE

States. Include a state for each symbol in the RE, plus an accept state.

Concatenation. Add match-transition edge from state corresponding to letters in the alphabet to next state.

Alphabet. A B C D
Metacharacters. ( ) . * |

Parentheses. Add \( \varepsilon \)-transition edge from parentheses to next state.
Building an NFA corresponding to an RE

**Closure.** Add ε-transition edges for each * operator.

**Solution.** Maintain a stack.
- Left parenthesis: push onto stack.
- | symbol: push onto stack.
- Right parenthesis: add edges for closure and or.

**Goal.** Write a program to build the ε-transition digraph.

**Challenge.** Need to remember left parentheses to implement closure and or; need to remember | to implement or.

**NFA construction: implementation**

**NFA construction: example**
Proposition 2. Building the NFA corresponding to an $M$-character pattern takes time and space proportional to $M$ in the worst case.

Pf. For each of the $M$ characters in the pattern, we add one or two $\epsilon$-transitions and perhaps execute one or two stack operations.
Generalized regular expression print

Grep. Takes a pattern as a command-line argument and prints the lines from standard input having some substring that is matched by the pattern.

```java
public class GREP {
    public static void main(String[] args) {
        String regexp = "\1\2...\3"; // Greedy
    }
}
```

Bottom line. Worst-case for grep (proportional to MN) is the same as for elementary exact substring match.

Industrial-strength grep implementation

To complete the implementation:
- Add character classes.
- Handling metacharacters.
- Add capturing capabilities.
- Extend the closure operator.
- Error checking and recovery.
- Greedy vs. reluctant matching.

Ex. Which substring(s) should be matched by the RE <blink>.*</blink>?

Typical grep application

Crossword puzzle

```
% grep s..ict.. words.txt
constrictor
stricter
```

Regular expressions in other languages

Broadly applicable programmer’s tool.
- Originated in Unix in the 1970s
- Many languages support extended regular expressions.
- Built into grep, awk, emacs, Perl, PHP, Python, JavaScript.

```
% grep NEWLINE */*.java
```

```
% perl -p -i -e 's|from|to|g' input.txt
```

PERL. Practical Extraction and Report Language.

```
% perl -n -e 'print if /[A-Za-z][\xa-z]*$/i' dict.txt
```

dictionary (standard in UNIX) also on booksite
Validity checking. Does the input match the regexp?
Java string library. Use input.matches(regexp) for basic RE matching.

```java
public class Validate {
    public static void main(String[] args) {
        String regexp = args[0];
        String input = args[1];
        StdOut.println(input.matches(regexp));
    }
}
```

Harvesting information

Goal. Print all substrings of input that match a RE.

```java
% java Harvester "gog(cgg|agg)*ctg" chromosomeX.txt
gogctg
gogctg
gogctg
% java Harvester "http://(\w+\.)*(\w+)"
http://www.cs.princeton.edu
http://www.princeton.edu
http://www.google.com
% java Harvester "[0-9]{3}-[0-9]{2}-[0-9]{4}" 166-11-4433
```

RE pattern matching is implemented in Java's Pattern and Matcher classes.

```java
import java.util.regex.Pattern;
import java.util.regex.Matcher;
public class Harvester {
    public static void main(String[] args) {
        String regexp = args[0];
        In in = new In(args[1]);
        String input = in.readAll();
        Pattern pattern = Pattern.compile(regexp);
        Matcher matcher = pattern.matcher(input);
        while (matcher.find())
            StdOut.println(matcher.group());
    }
}
```

Algorithmic complexity attacks

Warning. Typical implementations do not guarantee performance!

```java
% java RE "a*a" aaaaaaaaaaaaaaaaaaaaaaaaaac 1.6 seconds
% java RE "a*a" aaaaaaaaaaaaaaaaaaaaaaaaaac 3.7 seconds
% java RE "a*a" aaaaaaaaaaaaaaaaaaaaaaaaaac 9.7 seconds
% java RE "a*a" aaaaaaaaaaaaaaaaaaaaaaaaaac 23.2 seconds
% java RE "a*a" aaaaaaaaaaaaaaaaaaaaaaaaaac 42.2 seconds
% java RE "a*a" aaaaaaaaaaaaaaaaaaaaaaaaaac 161.6 seconds
```

SpamAssassin regular expression.

```java
% java RE "a*a+a*[a-z]+\((a-z)\)+\{a-z\}" spammer@x.............
```

- Takes exponential time on pathological email addresses.
- Troublemaker can use such addresses to DOS a mail server.
Not-so-regular expressions

• \1 notation matches sub-expression that was matched earlier.
• Supported by typical RE implementations.

Some non-regular languages.
• Set of strings of the form \(ww\) for some string \(w\): beriberi.
• Set of bitstrings with an equal number of 0s and 1s: 0110100.
• Set of Watson-Crick complemented palindromes: atttcggaaat.

Remark. Pattern matching with back-references is intractable.

Summary of pattern-matching algorithms

Programmer.
• Implement exact pattern matching via DFA simulation.
• Implement RE pattern matching via NFA simulation.

Theoretician.
• RE is a compact description of a set of strings.
• NFA is an abstract machine equivalent in power to RE.
• DFAs and REs have limitations.

You. Practical application of core CS principles.

Example of essential paradigm in computer science.
• Build intermediate abstractions.
• Pick the right ones!
• Solve important practical problems.

Context

Abstract machines, languages, and nondeterminism.
• basis of the theory of computation
• intensively studied since the 1930s
• basis of programming languages

Compiler. A program that translates a program to machine code.
• KMP string \(\Rightarrow\) DFA.
• grep RE \(\Rightarrow\) NFA.
• javac Java language \(\Rightarrow\) Java byte code.

<table>
<thead>
<tr>
<th>pattern</th>
<th>parser</th>
<th>compiler output</th>
</tr>
</thead>
<tbody>
<tr>
<td>KMP</td>
<td>grep</td>
<td>Java</td>
</tr>
<tr>
<td>string</td>
<td>unnecessary</td>
<td>DFA</td>
</tr>
<tr>
<td>RE</td>
<td>check if legal</td>
<td>NFA</td>
</tr>
<tr>
<td>program</td>
<td>check if legal</td>
<td>byte code</td>
</tr>
<tr>
<td>DFA simulator</td>
<td>NFA simulator</td>
<td>JVM</td>
</tr>
</tbody>
</table>