Regular Expressions

Regular Expressions

> A notation to specify a set of strings

> pattern matching---find one of a specific set of strings

Applications

Test if a string matches some pattern.

- Process natural language.
- Scan for virus signatures.
- Specify a programming language.
- Access information in digital libraries.
- Search genome using PROSITE patterns.
- Filter text (spam, NetNanny, Carnivore, malware).
- Validate data-entry fields (dates, email, URL, credit card).
- ...

Parse text files.

- Compile a Java program.
- Crawl and index the Web.
- Read in data stored in ad hoc input file format.
- Create Java documentation from Javadoc comments.

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

REs and NFAs

> Machine to recognize whether a given string is in a given text

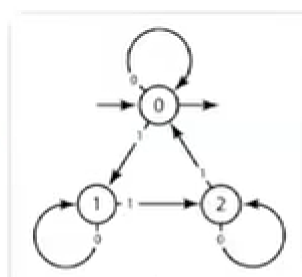
Kleene's theorem

- For any DFA, there exists a RE that describes the same set of srings
- For any RE, there exists a DFA that recognizes the same set of things

RE `0* | (0*10*10*10*)*`

number of 1's is a multiple of 3

DFA



number of 1's is a multiple of 3

NFA

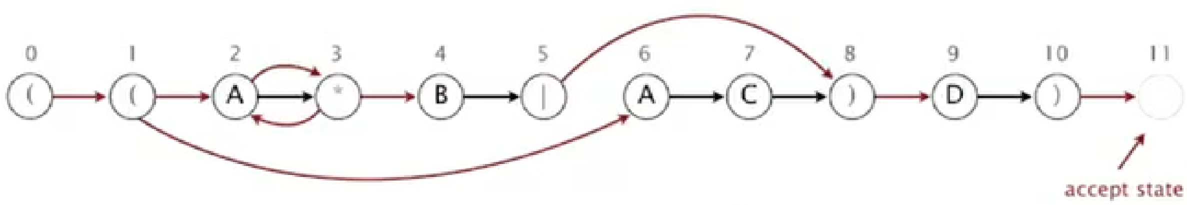
> DFA---deterministic finite state automata, easy because exactly one applicable transition

> NFA---nondeterministic finite state automata, can be several applicable transitions and need to select the right one

- Nondeterminism
  - one view---machine can guess the proper sequence of state transitions
  - another---sequence is a proof that the machine accepts the text

RE matching NFA

- RE enclosed in parentheses
- One state per RE character (start=0, accept=M)
- Red ϵ-transition (change state, but don't scan text)
- Black match transition (change state and can to next text char)
- Accept if any sequence of transitions ends in accept state



NFA corresponding to the pattern ((A * B | A C) D)

NFA Simulation

Elements

- state names---integers from 0 to M
- match transitions---keep RE in array re[]
- ϵ-transitions---store in a digraph G

Simulation

> maintain a set of all possible states that NFA could be in after reading in the first i text characters

Complexity

Proposition. Determining whether an N-character text is recognized by the NFA corresponding to an M-character pattern takes time proportional to $M N$ 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 ε-transitions.
[The NFA construction we will consider ensures the number of edges $\leq 3M$.]

Digraph reachability

> find all vertices reachable from a given source or set of vertices

```java
public class NFA {

    private Digraph graph;     // digraph of epsilon transitions
    private String regexp;     // regular expression
    private final int m;       // number of characters in regular expression

    public NFA(String regexp) {}

    public boolean recognizes(String txt) {
        DirectedDFS dfs = new DirectedDFS(graph, 0);
        Bag<Integer> pc = new Bag<Integer>();
        for (int v = 0; v < graph.V(); v++)
            if (dfs.marked(v)) pc.add(v);

        // Compute possible NFA states for txt[i+1]
        for (int i = 0; i < txt.length(); i++) {
            if (txt.charAt(i) == '*' || txt.charAt(i) == '|' || txt.charAt(i) == '(' || txt.charAt(i) == ')')
                throw new IllegalArgumentException("text contains the metacharacter '" + txt.charAt(i) + "'");

            Bag<Integer> match = new Bag<Integer>();
            for (int v : pc) {
                if (v == m) continue;
                if ((regexp.charAt(v) == txt.charAt(i)) || regexp.charAt(v) == '.')
                    match.add(v+1);
            }
            if (match.isEmpty()) continue;

            dfs = new DirectedDFS(graph, match);
            pc = new Bag<Integer>();
            for (int v = 0; v < graph.V(); v++)
                if (dfs.marked(v)) pc.add(v);

            // optimization if no states reachable
            if (pc.size() == 0) return false;
```

```java
        }

        // check for accept state
        for (int v : pc)
            if (v == m) return true;
        return false;
    }

    public static void main(String[] args) {
        String regexp = "(" + args[0] + ")";
        String txt = args[1];
        NFA nfa = new NFA(regexp);
        StdOut.println(nfa.recognizes(txt));
    }

}
```

NFA Construction

Stack

Challenges. Remember left parentheses to implement closure and or; remember | to implement or.

Solution. Maintain a stack.

- (symbol: push (onto stack.
- | symbol: push | onto stack.
-) symbol: pop corresponding (and possibly intervening |; add ε-transition edges for closure/or.

Implementation

```java
    public NFA(String regexp) {
        this.regexp = regexp;
        m = regexp.length();
        Stack<Integer> ops = new Stack<Integer>();
        graph = new Digraph(m+1);
        for (int i = 0; i < m; i++) {
            int lp = i;
            if (regexp.charAt(i) == '(' || regexp.charAt(i) == '|')
                ops.push(i);
            else if (regexp.charAt(i) == ')') {
                int or = ops.pop();

                // 2-way or operator
                if (regexp.charAt(or) == '|') {
                    lp = ops.pop();
```

```java
                    graph.addEdge(lp, or+1);
                    graph.addEdge(or, i);
                }
                else if (regexp.charAt(or) == '(')
                    lp = or;
                else assert false;
            }

            // closure operator (uses 1-character lookahead)
            if (i < m-1 && regexp.charAt(i+1) == '*') {
                graph.addEdge(lp, i+1);
                graph.addEdge(i+1, lp);
            }
            if (regexp.charAt(i) == '(' || regexp.charAt(i) == '*' || regexp.charAt(i) == ')')
                graph.addEdge(i, i+1);
        }
        if (ops.size() != 0)
            throw new IllegalArgumentException("Invalid regular expression");
    }
```

Applications

Completion

To complete the implementation:

- Add wildcard.
- Add multiway or.
- Handle metacharacters.
- Support character classes.
- Add capturing capabilities.
- Extend the closure operator.
- Error checking and recovery.
- Greedy vs. reluctant matching.

Generalized RE Print

> Take a RE as a command-line argument and print the lines from standard input having some substring that is matched by the RE

```java
public class GREP {

    // do not instantiate
    private GREP() { }

    public static void main(String[] args) {
        String regexp = "(.*" + args[0] + ".*)";
        NFA nfa = new NFA(regexp);
```

```
        while (StdIn.hasNextLine()) {
            String line = StdIn.readLine();
            if (nfa.recognizes(line)) {
                StdOut.println(line);
            }
        }
    }
}
```

Java string library

> java.util.regexp.Patern

> java.util.regexp.Matcher

```
/******************************************************************************
 *  % java Validate "..oo..oo." bloodroot
 *  true
 *
 *  % java Validate "..oo..oo." nincompoophood
 *  false
 *
 *  % java Validate "[^aeiou]{6}" rhythm
 *  true
 *
 *  % java Validate "[^aeiou]{6}" rhythms
 *  false
 ******************************************************************************/

public class Validate {

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

Harvesting information

> print all substrings of input that match a RE

Not-so-regular expressions

> pattern matching with back-references is intractable

Back-references.

- \1 notation matches subexpression that was matched earlier.
- Supported by typical RE implementations.

```
(.+)\1            // beriberi couscous
1?$|^(11+?)\1+    // 1111 111111 111111111
```

Some non-regular languages.

- Strings of the form ww for some string w: beriberi.
- Unary strings with a composite number of 1s: 111111.
- Bitstrings with an equal number of 0s and 1s: 01110100.
- Watson-Crick complemented palindromes: atttcggaaat.

Summary

- 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

	KMP	grep	Java
pattern	string	RE	program
parser	unnecessary	check if legal	check if legal
compiler output	DFA	NFA	byte code
simulator	DFA simulator	NFA simulator	JVM

- programmer
  - implement substring search via DFA simulation
  - implement RE pattern matching via NFA simulation
- Theoretician
  - RE is a compact descripion of a set of strings
  - NFA is an abstract machine equivalent in power of RE
  - DFAs and REs have limitations