Applications to string and pattern matching

Informatics 2A: Lecture 6

John Longley

School of Informatics
University of Edinburgh
jrl@inf.ed.ac.uk

29 September 2017

Recap of Lecture 5

- Regular languages are closed under the operations of concatenation and Kleene star.
- This is proved using ϵ-NFAs, which can be easily converted to ordinary NFAs.
- Regular expressions provide a textual representation of regular languages.
- Kleene's Theorem: regular expressions define exactly the regular languages.
- We haven't proved the difficult half of the theorem formally, but equation solving using Kleene algebra and Arden's Rule gives the idea.

Applications of regular language technology

In this lecture and the next, we'll look at several practical applications of the theory we've covered:

- efficient string searching
- more general pattern searching
- data validation, e.g. for XML documents
- lexical analysis for computer languages (first stage of the language processing pipeline)
- automated verification of safety and liveness properties for complex interacting systems — typically via model checking.

Further applications will be discussed in the Natural Language part of the course.

String and pattern matching with Grep tools

Important practical problem: Search a large file (or batch of files) for specific strings, or strings of a certain form.

Most UNIX/Linux-style systems since the '70s have provided a bunch of utilities for this purpose, known as Grep (Global Regular Expression Print).

Extremely useful and powerful in the hands of a practised user. Make serious use of the theory of regular languages.

Typical uses:

```
grep "[0-9]*\.[0-9][0-9]" document.txt
```

−− searches for prices in pounds and pence

```
egrep "(^|[^a-zA-Z])[tT]he([^a-zA-Z]|$)" document.txt
```

−− searches for occurrences of the word "the"

grep, egrep, fgrep

There are three related search commands, of increasing generality and correspondingly decreasing speed:

- `fgrep` searches for one or more fixed strings, using an efficient *string matching* algorithm.
- `grep` searches for strings matching a certain pattern (a simple kind of regular expression).
- `egrep` searches for strings matching an extended pattern (these give the full power of regular expressions).

All three of these make use of the ideas we've been studying.

Efficient string matching

Suppose we want to search for occurrences of a shortish string s in a very long document D.

Obvious method: For each position p within D, check whether there's an occurrence of s starting at p, by working through s one character at a time until:

- either there's a character mismatch
- or we reach the end of s (search successful).

(Example on next slide.)

Can we do better?

Naive string search: an example

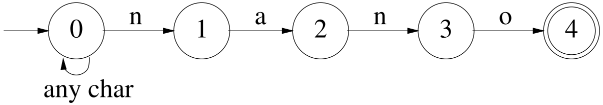
Suppose we're searching for nano within a long document D containing nanobananas.

n	a	n	o	b	a	n	a	n	a	s
n	a	n	✓							
	X									
		n	X							
			X							
				X						
					X					
						n	a	n	X	
							X			
								n	a	X
									X	
										X

Notice that several characters in D are visited more than once.

Better method: The Knuth-Morris-Pratt algorithm

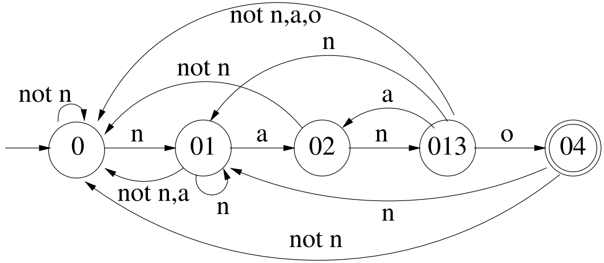
The following NFA clearly accepts all strings ending in nano:



So we can . . .

1. First convert this to an equivalent DFA M. (Costs some time—but worth it if s is short and D is very long.)
2. Run D through M. (Each character of D processed just once; no buffering required.)
3. Every time we enter an accepting state of M, signal a hit.

The corresponding DFA



- For NFAs of this kind, the subset construction behaves nicely: no state explosion. Can even optimize the construction a bit for this class of NFAs.
- Useful in practice for documents with a lot of repetition. E.g. imagine searching for `www.inf.ed.ac.uk/inf2a/` in a long list of web addresses, where most begin with `www.` and many begin with `www.inf.ed.ac.uk/`

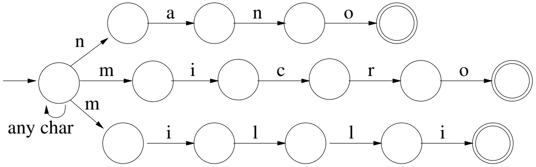
Optional exercise

1. Suppose now that the search string is nana. Construct the appropriate DFA in this case.
2. Convince yourself that this will detect all occurrences of nana, even overlapping ones!

Multiple strings

Suppose now we want to find all occurrences of any of the strings nano, micro, milli in D.

No problem! Just do the same starting from the following NFA:



(The gain over the naive method is here readily apparent.)

To do more powerful searches, can use regular expressions . . .

Machine syntax for regular expressions

`a`	Single character
`[abc]`	Choice of characters
`[A-Z]`	Any character in ASCII range
`[^Ss]`	Any character except those given
`.`	Any single character
`^`, `$`	Beginning, end of line
`*`	zero or more occurrences of preceding pattern
`?`	optional occurrence of preceding pattern
`+`	one or more occurrences of preceding pattern
`|`	choice between two patterns ('union')

(The last three are allowed by `egrep`, but not by plain `grep`.)

This kind of syntax (with further bells and whistles) is very widely used. In Perl/Python (including NLTK), patterns are delimited by `/.../` rather than `"..."`.

Mathematical versus machine notation

We've now seen two notations for writing regular expressions:

- Mathematical notation, e.g. $(a+b)(a+b)^*$. This notation is intended to have as few operations as possible, for convenience in setting up the theory (e.g. Kleene algebra).
- Machine notation (regex), e.g. `(a|b)+`. This has a more generous set of operations, for convenience when writing complicated regular expressions.

The clashes between these are unfortunate, but we're stuck with them.

- Union of two languages is written using | in machine syntax, and $+$ in mathematical syntax.
- In machine syntax, $+$ is a unary operation representing concatenation of one or more strings of a given form.
- Dot `.` means concatenation in the mathematical syntax, and 'any character' in the machine syntax.

Example

How suitable are the patterns below for specifying the form of non-negative decimal integers?

1. `[0-9]*`
2. `[0-9]+`
3. `0 | [1-9][0-9]*`
4. `0 | [1-9][0-9]?[0-9]?(,[0-9][0-9][0-9])*`

Example

How suitable are the patterns below for specifying the form of non-negative decimal integers?

1. `[0-9]*`
2. `[0-9]+`
3. `0 | [1-9][0-9]*`
4. `0 | [1-9][0-9]?[0-9]?(,[0-9][0-9][0-9])*`

Answer: Pattern 1 is bad, because it admits the empty string.
Pattern 2 is fine if we don't mind leading zeros, e.g. 023.
Pattern 3 is just right for non-neg integers without leading zeros.
Pattern 4 is good for a common way of writing integers within English text, e.g. 1,024 or 578,000,000,000.

How egrep (typically) works

`egrep` will print all lines containing a match for the given pattern. How can it do this efficiently?

- Every machine regexp is clearly equivalent to a mathematical one.
- So we can convert a pattern into a (smallish) NFA.

  (More precisely, the number of states of the NFA grows linearly in the length of the regular expression.)
- We then run the NFA , using the just-in-time simulation discussed in Lecture 4.

  We don't determinize the NFA to construct the full DFA, because of the potential exponential state-space blow-up.

`grep` can be a bit more efficient, exploiting the fact that there's 'less non-determinism' around in the absence of $+, ?, |$.

Regular expressions in data validation

Regexp's are used not just in searching, but also in checking whether data is of the expected form:

- Within XML documents, can enforce constraints on parts of the data:

  ```
  <xs:simpleType name="ProductNumberType">
    <xs:restriction base="xs:string">
      <xs:pattern value="\d{3}-[A-Z]{2}|\d7"/>
    </xs:restriction>
  </xs:simpleType>
  ```

  (Example from P. Walmsley, *Definitive XML Schema*, 2012.)

- For text fields in web forms, check that the input text has the correct form. (See `regexlib.com` for hundreds of regexp's for validating email addresses, URLs, UK mobile phone numbers, postcodes, . . .)

Challenge question

Regular expressions and the pattern language have operations that correspond to the closure of regular languages under union, concatenation and Kleene star.

However, we have seen other closure properties of regular languages too: closure under intersection and complement.

Question: Why does the (basic) regex language not include operations for intersection and complement?

Challenge question

Regular expressions and the pattern language have operations that correspond to the closure of regular languages under union, concatenation and Kleene star.

However, we have seen other closure properties of regular languages too: closure under intersection and complement.

Question: Why does the (basic) regex language not include operations for intersection and complement?

Answer: If we included these, even a smallish regex could lead to a state space explosion. (Complement especially bad: need to determinize first!) The design of the regex language protects the unwary user from such nasty surprises.

Challenge question

Regular expressions and the pattern language have operations that correspond to the closure of regular languages under union, concatenation and Kleene star.

However, we have seen other closure properties of regular languages too: closure under intersection and complement.

Question: Why does the (basic) regex language not include operations for intersection and complement?

Answer: If we included these, even a smallish regex could lead to a state space explosion. (Complement especially bad: need to determinize first!) The design of the regex language protects the unwary user from such nasty surprises.

WARNING: Some modern versions of so-called ‘regex’ (e.g. in Perl) include wild constructs that actually go way beyond the power of regular languages (back-references, backtracking, lookahead, recursive regexes) . . . and which definitely don’t protect the unwary user from nastiness!