Computational Approaches for Life Scientists

Faculty of Biology
Technion
Spring 2014

Amir Rubinstein

Strings (2) – Regular Expressions

(and Deterministic Finite Automata)

Lecture Outline

- We'll consider the **pattern matching** problem

- Then introduce the concepts of
  - Regular Expressions (RE)
    - The **re** module in Python
  - Deterministic Finite Automata (DFA)

Motivation – pattern matching

- Pattern matching:

  Given a string ***s*** and another string ***p*** (called pattern),

  does ***s*** contain ***p*** ?

- Is this different from the substring problem we have just learned?
  - Not really, unless you consider this important generalization:

    the pattern ***p*** may not be a unique string,

    it could represent multiple possible strings (even ∞)

    for example: *G then C or T then between 2-3 A's*

  - Another variation is that today, we will assume we have the pattern for pre-processing

- Can suffix trees help us here?

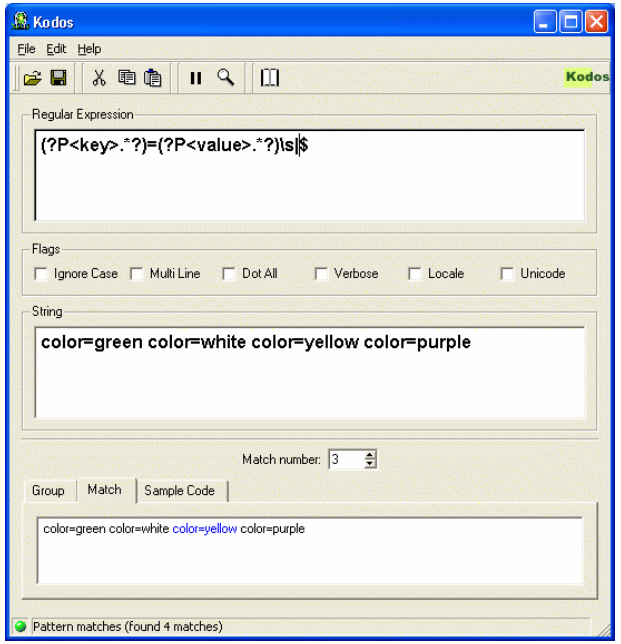
Regular expressions

- Regular expressions are a common way of representing patterns within strings.
  - However, RE are not omnipotent – not all patterns can be represented as RE.
  - We will see an example later.

- Python has a module called **re** , which enables to define regular expressions.
  we will now introduce it, mostly by examples.

- For a more thorough introduction, see http://docs.python.org/3.2/howto/regex.html

- Good to know:
  Kodos: a nice GUI utility (made in Python)
  that allows you to test and debug
  your regular expressions:
  http://kodos.sourceforge.net



RE – basic search

```
>>> import re
>>> mo = re.search("hello", "Hello world, hello Python!")  #mo = match object
>>> mo.group()
'hello'
>>> mo.start()
13
>>> mo.end()
18
>>> mo.span()
(13, 18)

>>> help(re.search)
Help on function search in module re:

search(pattern, string, flags=0)
    Scan through string looking for a match to the pattern, returning
    a match object, or None if no match was found.
```

RE – findall / finditer

```
>>> import re
>>> mo = re.search("[Hh]ello", "Hello world, hello Python!")  #mo = match object
>>> mo.span()
(0, 5)

>>> re.findall("[Hh]ello","Hello world, hello Python!")
['Hello', 'hello']

>>> re.finditer("[Hh]ello","Hello world, hello Python!")
<callable-iterator object at 0xb6f43d8c>
>>> mit = re.finditer("[Hh]ello","Hello world, hello Python!")  #mit = match object iterator
>>> for mo in mit:
        print (mo.group(), mo.span())

Hello (0, 5)
hello (13, 18)
```

what's the advantage of *finditer* over *findall*?

Python **re** basic syntax

- Characters represent themselves, except:

- Meta-characters: [ ] . | ^ $ * + ? { \ ( )

- [] (square brackets): Indicates a set of characters.

  "[ABC]" will match A or B or C. "[A-Z]" will match any uppercase letter and "[a-z0-9]" will match any lowercase letter or digit.

  The ^ inside a set will math the complement of a set. "[^R]" will match any character but "R".

  Other meta-characters are not active inside [ ]. "[AT*]" will match "A", "T" or "*".

- . (dot): Matches any character, except new-line:

  "ATT.T" will match "ATTCT", "ATTFT" but not "ATTTCT".

- | (vertical bar): As in logic, it reads as "or".

  "ATT|CG|C" will match "ATT", "CG" or "C".

Example – TATA box

- Does a given DNA string contain a TATA-box-like pattern?

  “TATAA” followed by 3 nucleotides and ends with “TT”

```
def hasTataLike(string):
    if (re.search("TATAA[ACGT][ACGT][ACGT]TT", string)):
        return True
    return False
```

Python **re** basic syntax (2) - repeats

- Meta-characters for repeats
- * (star): Matches 0 or more repetitions of the preceding token:

  "AT*" will match "AAT", "A", but not "TT".

- + (plus): Matches 1 or more repetitions of the preceding token:

  "AT+" will match "GATT", but not "GA".

- ? (question mark): Matches 0 or 1 repetitions of the preceding token.

  "AT?" will match either "A" or "AT".

- {n}: Exactly n copies.

  "(ATTG){3}" will match "ATTGATTGATTG" but not "ATTGATTG".

- {m,n}: from m to n repetitions.

  "(AT){3,5}" will match "ATATTATATAT" but not "ATATTATAT".

  Omitting m is interpreted as a lower limit of 0, while omitting n results in an upper bound of infinity.

Example – TATA box again

- Does a given DNA string contain a TATA-box-like pattern?

  “TATAA” followed by 3 nucleotides and ends with “TT”

```
def hasTataLike(string):
    if (re.search("TATAA[ACGT][ACGT][ACGT]TT", string)):
        return True
    return False
```

```
def hasTataLike(string):
    if (re.search("TATAA[ACGT]{3}TT", string)):
        return True
    return False
```

Python **re** basic syntax (3)

- Meta-characters for anchors:

- ^(caret): Matches the beginning of the chain: "^AUG" will match "AUGAGC" but not "AAUGC". Using ^ inside [ ] means "opposite".

- $(dollar): Matches the end of the chain or just before a new line at the end of the chain: "UAA$" will match "AGCUAA" but not "ACUAAG".

Python **re** basic syntax (4)

- Pre-defined classes

String	Class	Equivalent
\d	Decimal digit	[0-9]
\D	Non-digit	[^0-9]
\s	Any whitespace	[\t\n\r\f\v]
\S	Non-whitespace	[^ \t\n\r\f\v]
\w	Any alphanumeric	[a-zA-Z0-9_]
\W	Non alphanumeric	[^a-zA-Z0-9_]

```
# Match two digits, then any character, then two non digits
>>> re.search("\d\d.\D\D", "98\tAD").span()
(0, 5)
```

Python **re** basic syntax (5)

- Parenthesis ( )

  Used for grouping:

  e.g. GC{4} vs. (GC){4}

  Use them as follows:

```
>>> re.findall("(?:GC){4}", "GCGCGCGCATGCGCGCGC")
['GCGCGCGC', 'GCGCGCGC']
```

not like this:

```
>>> re.findall("(GC){4}", "GCGCGCGCATGCGCGCGC")
['GC', 'GC']
```

Compiling a pattern (preprocessing)

- "Compiling" a pattern means **preprocessing** it, storing it in some way in the computer's memory, to speed up future searches.

- The internal representation is somewhat based on the idea of Deterministic Finite Automata (DFA) that we will see next, but involves many more details, and we won't get into it.

```
>>> rgx = re.compile("[Hh]ello")
>>> rgx.findall("Hello world, hello Python!")
['Hello', 'hello']
>>> rgx.findall("Hello world, bye Python!")
['Hello']
```

- Compiled objects also have the methods search, findall and finditer.

- We do not have to use compiled re, but if we search the same pattern many times, this could speed up things significantly.

Another example

- What will this code do?

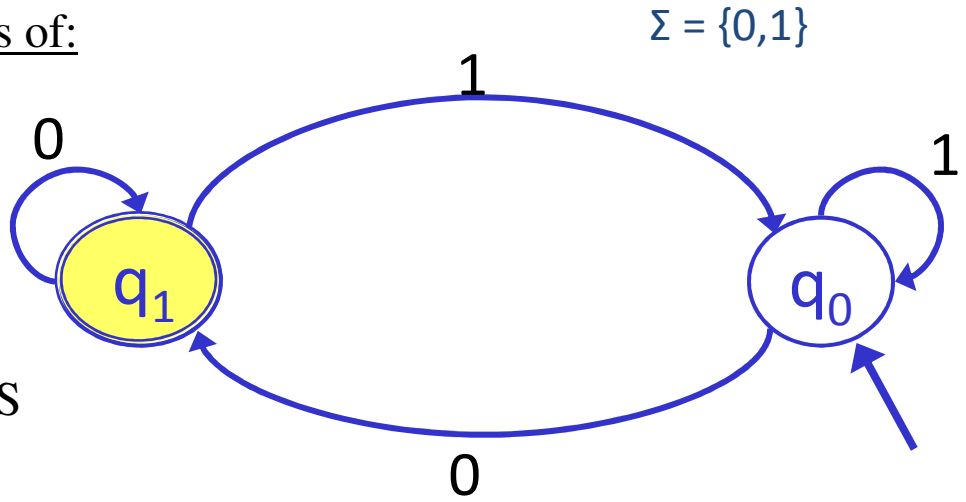
```
seq = "ATATAAGATGCGCGCGCTTATGCGCGCA"
rgx = re.compile("TA.")
i = 1
for mo in rgx.finditer(seq):
    print("Occurrence", i, mo.group() )
    print("Position: From", mo.start(), "to" ,mo.end() )
    i += 1
```

Deterministic Finite Automata (DFA)

Regular expressions can be represented by DFA.

An automaton is an object that consists of:

- alphabet Σ
- finite set of states S
- transition function f: SxΣ→ S
- an initial state ∈S
- a set of final ("accepting") states ⊆S



We say that automaton **accepts** a string

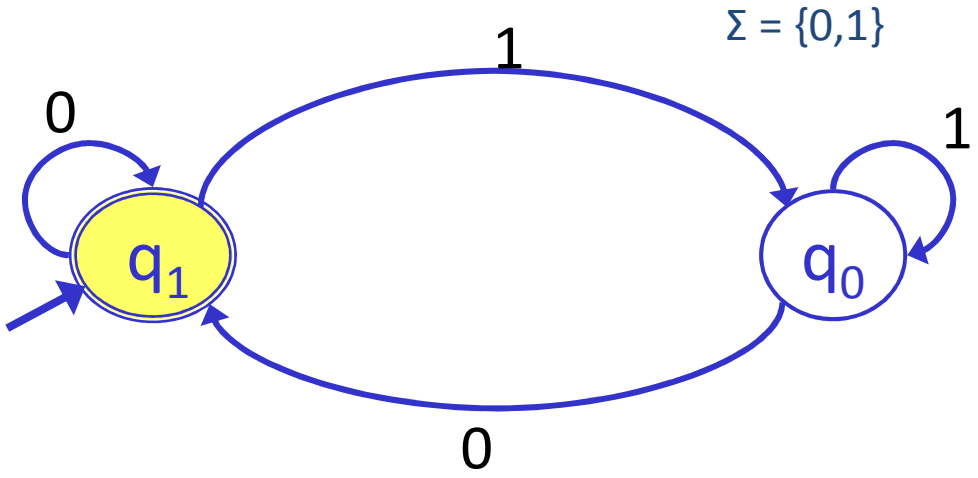
if "walking the string through the automaton" ends up in an accepting state.

automaton = pattern

- Which strings does the automaton in the figure accept?

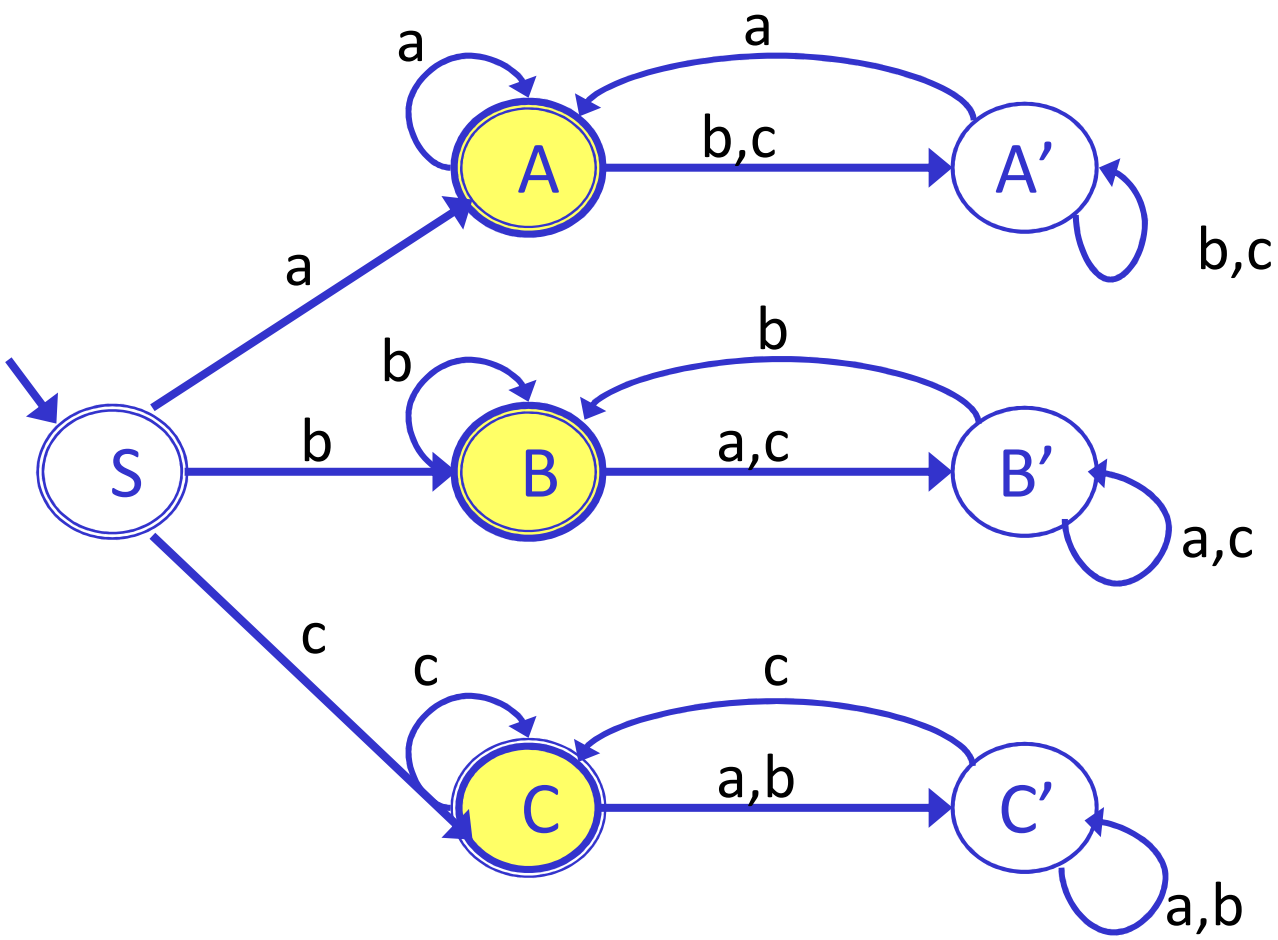
DFA

What would change if we made q_1 the initial state?



DFA – another example

What patterns does this automaton identify (over {a,b,c})?

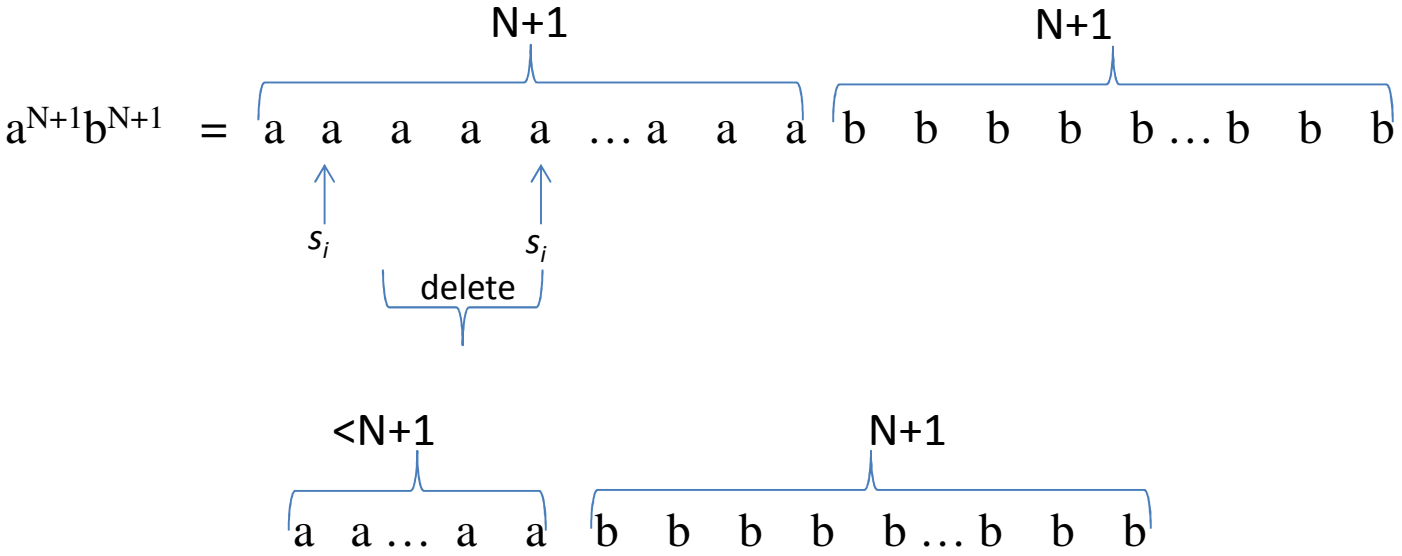


Limitations of DFA

We said that an automaton represents a regular expression and accepts a pattern.
But not all patterns can be defined this way.

for example: no finite automata can identify strings of the form $a^n b^n$ for unknown n.

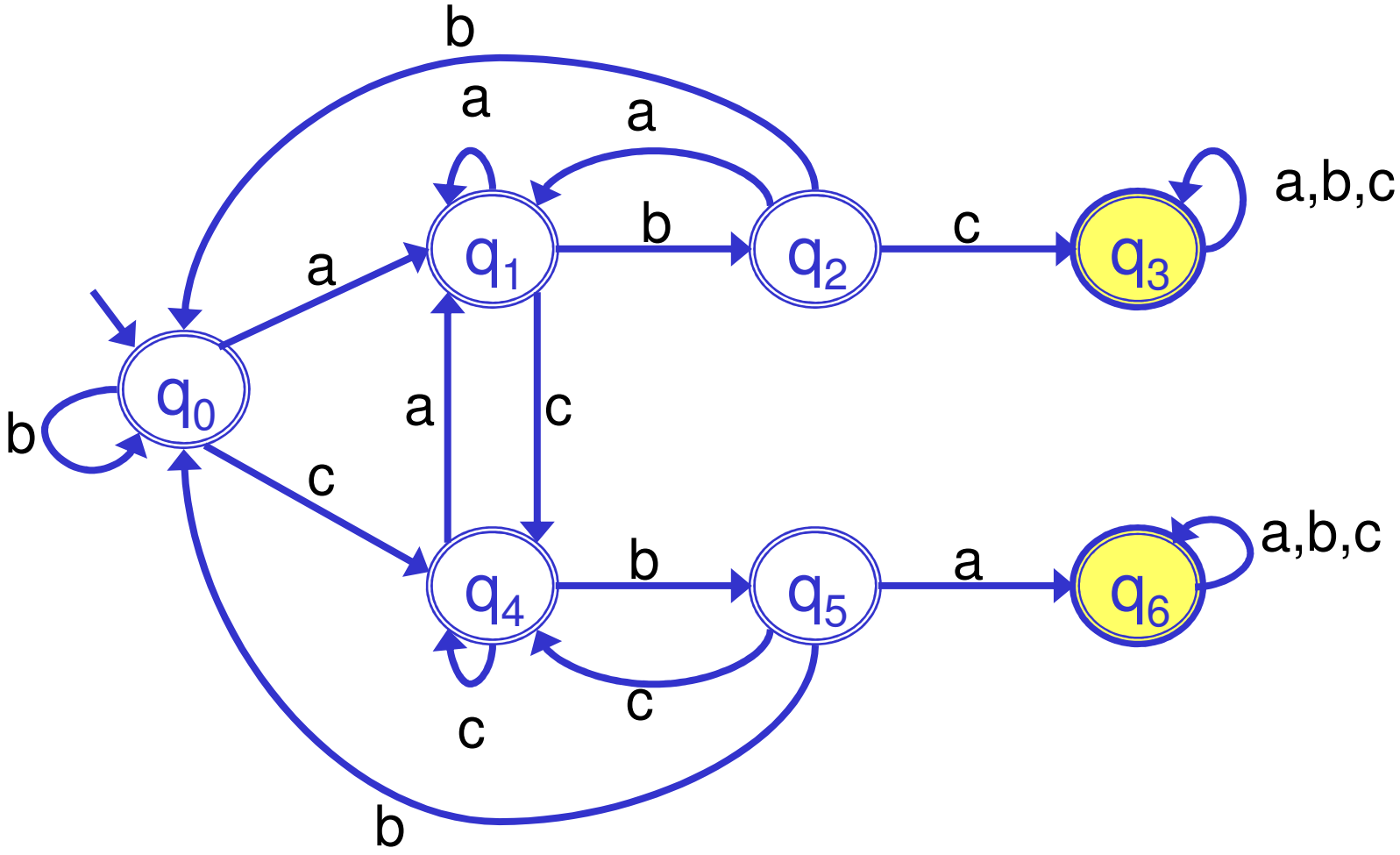
proof:
assume there is such an automaton, with N states.
Then for $a^{N+1}b^{N+1}$ it must repeat the same state s_i at least twice (pigeon hole principle).
But then it must also accept strings with <N+1 a's , contradiction.



* comment: for any finite known n, there IS such an automaton… show one for a^2b^2.

Exercises

1. What pattern does this automaton identify (over {a,b,c})?
   what is the regular expression defined by this automaton?



Exercises

2. What does the following function do ?

```
def replace(seq, re, new):
    return re.sub(new, seq)
```

Try to call it with:

```
seq = "Hello my dear, hello!"
regex = re.compile("[hH]ello")
new = "Goodbye"
```

Conclude what re.sub does.

3. Write a function count_re(seq, re),
which gets a sequence seq and a regular expression re, and returns how many times re appears in seq. (hint: it's a 1 line function…)

4. Write a function twiceTataLike(seq),
which gets a sequence seq, and returns True if it contains at least two TATA-like boxes, else returns False (a 3 line function…)