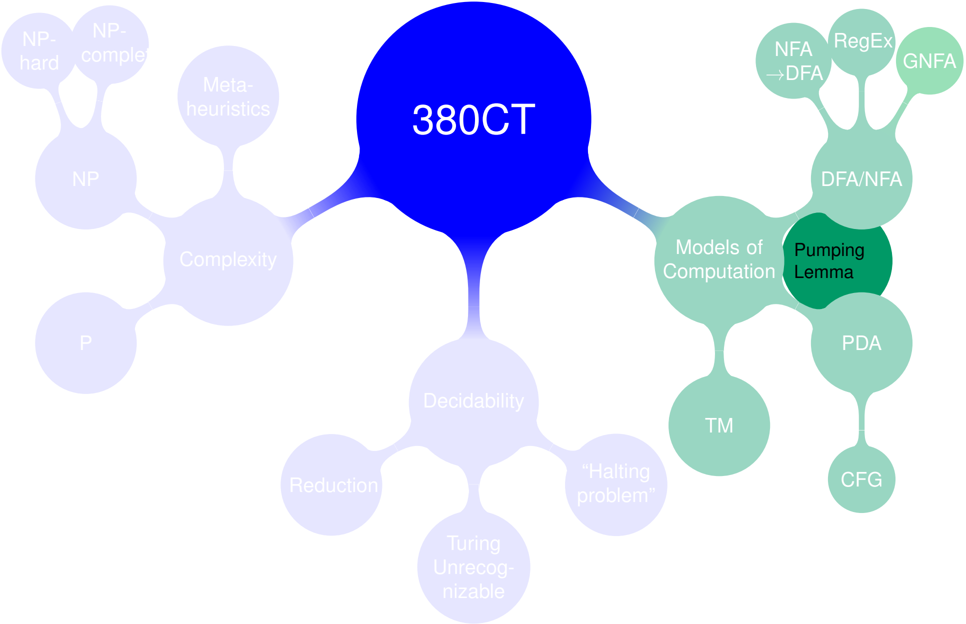
Models of Computation: Limitations of the Regular Languages

The Pumping Lemma
Grammars

Dr Kamal Bentahar

School of Computing, Electronics and Mathematics
Coventry University

Week 4 – 12/2/2019


380CT
Complexity
NP
NP-hard
NP-complet
Meta-heuristics
P
Decidability
Reduction
Turing Unrecog-nizable
"Halting problem"
Models of Computation
DFA/NFA
NFA →DFA
RegEx
GNFA
Pumping Lemma
PDA
CFG
TM

Last week...

Regular Languages

The class of regular languages can be:

1. Recognized by NFAs. (equiv. GNFA or ε-NFA or NFA or DFA).
2. Described using **Regular Expressions**.

Today:

1. See the limit of regular languages.
2. How to show a language is not regular.
3. Generate regular languages using **Regular Grammars**.

Proofs

We show a language is regular using **Proof by existence**:

- Construct an NFA recognizing it.
- Write a Regular Expression for it.

  Using closure under the **union**, **concatenation** and **star** operations.

However, *not all languages are regular*; so how can we show that a given languages is **not** regular?!

To prove a language is not regular, we use **Proof by contradiction.**

- We need a property that all regular languages must satisfy.
- So if a given language does not satisfy it then it cannot be regular.

Proof by contradiction in general – example 1

The sum of two positive numbers is always positive.

- Suppose: The sum of two positive numbers is not always positive.
- So there exist positive numbers a and b that sum to a negative number.

$$a + b < 0$$

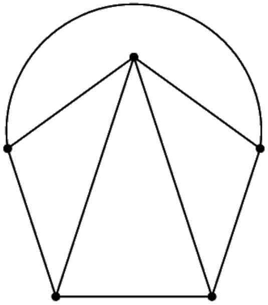
- Rearranging we get:

$$a < -b$$

- But since b is a positive number then $-b$ must be a negative number.
- This means that a, which is positive, is less than a negative number!
- $\rightarrow$ Contradiction.
- So the supposition above must be false.

Proof by contradiction in general – example 2

Is it possible to traverse this graph by travelling along **each path exactly once**?



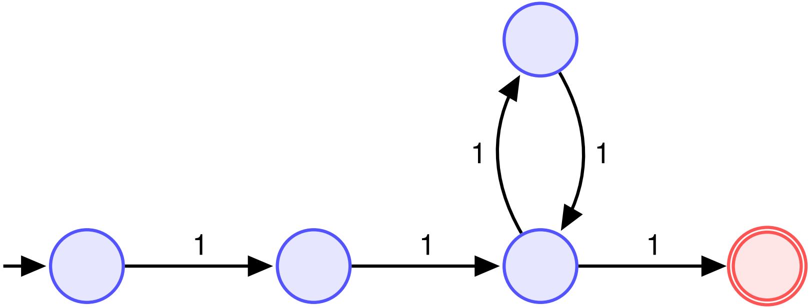
- Suppose it is possible.
- How many times would each vertex be visited?
  - Every time a vertex is entered, it is also exited.
  - Therefore, each vertex should have an **even number** of neighbours.
  - The exceptions to this are the starting vertex and ending vertex: these should have an **odd number** of paths coming from them.
  - There can only be one starting vertex and one ending vertex.
- However, this graph has 4 vertices with an odd number of paths coming from them.
- Thus, it is impossible to traverse the above graph by travelling along each path exactly once.

A look back

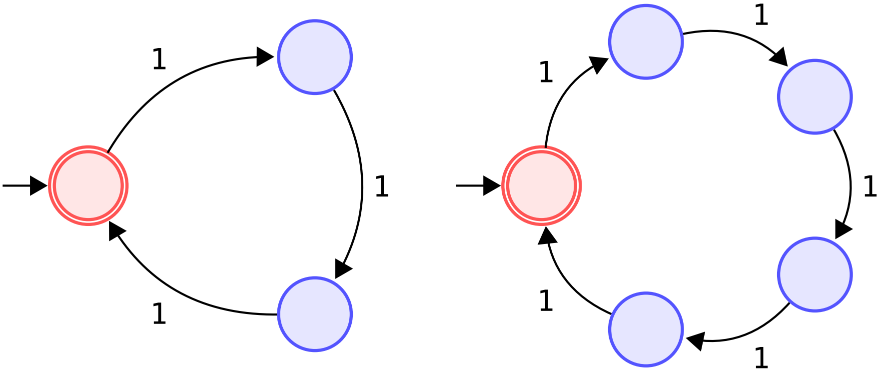
Let us try to understand RLs a bit more. . . let us look back at some examples — for each automaton in the next slides, let us think about **RegEx** and the **path taken by an accepted string** (is it “straight” or does it loop?).

Unary alphabet $\{1\}$

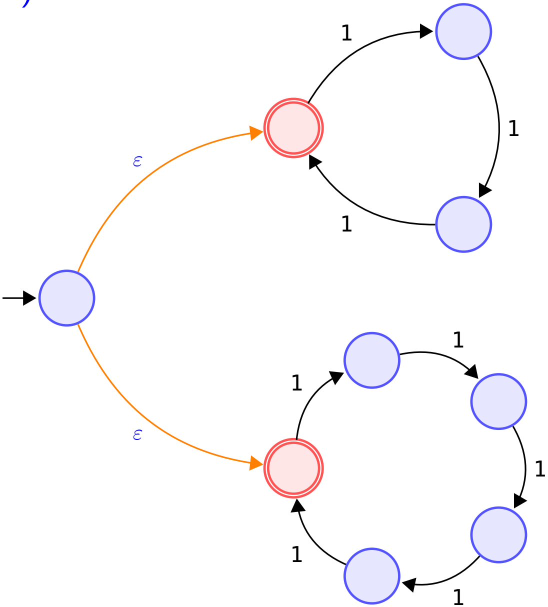
Strings of length $3, 5, 7, 9, \ldots$



$(111)^*$, $(11111)^*$



$(111)^* + (11111)^*$



Unary alphabet – Special case

Let L be a regular languages over a unary alphabet $\Sigma = \{1\}$.

The language L is:

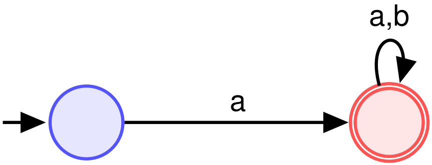
- either **finite**, in which case it is regular trivially,
- or **infinite**, in which case its DFA will have to **loop**:
  - The DFA that recgnizes L has a finite number of states.
  - Any string in L determines a path through the DFA.
  - So any sufficiently long string must visit a state twice.
  - This forms a loop.

This looped part can be repeated any arbitrary number of times to produce other strings in L.

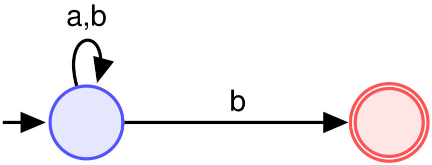
Pigeon-hole principle

If we put **more than** n pigeons into n holes then there must be a hole with more than one pigeon in.

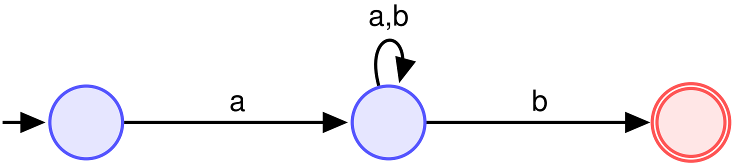
$a\Sigma^*$



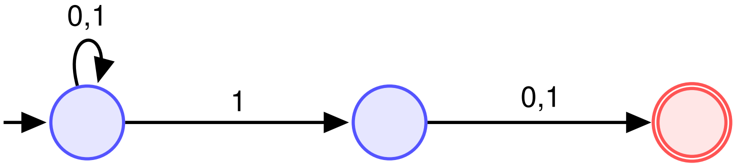
$\Sigma^* b$



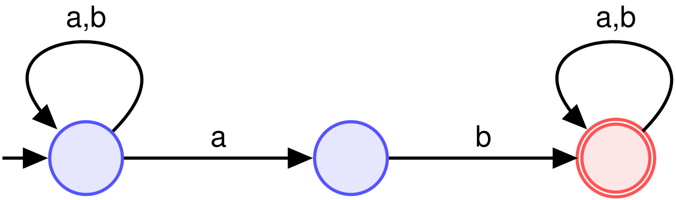
$a\Sigma^*b$



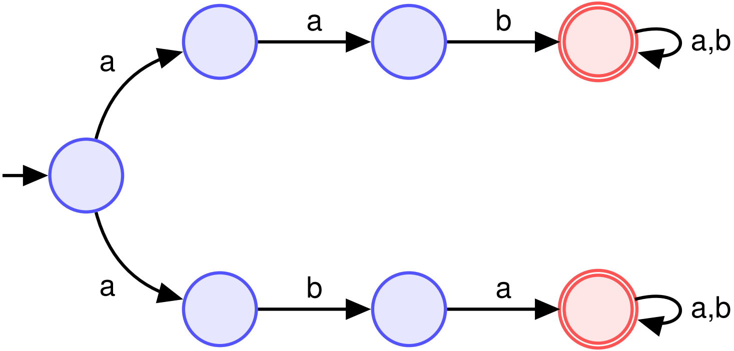
$\Sigma^*1\Sigma$



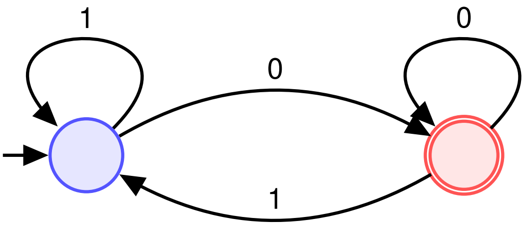
$\Sigma^*ab\Sigma^*$



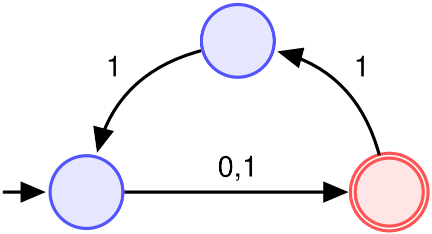
$aab\Sigma^* + aba\Sigma^*$



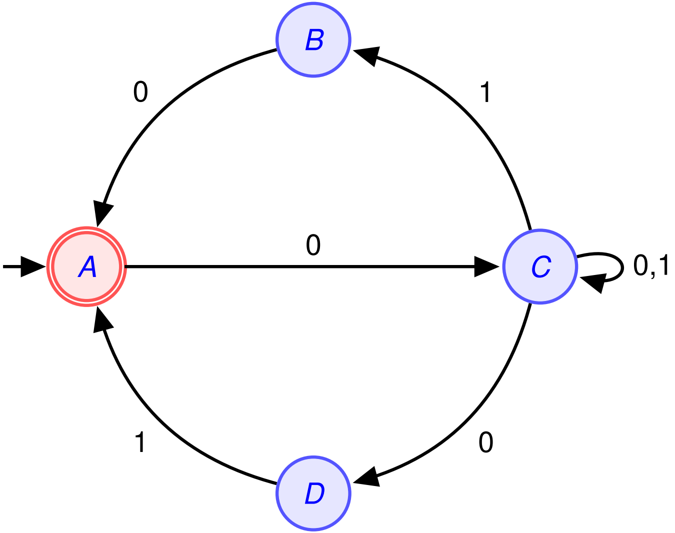
Σ^*0



$(\Sigma 11)^* \Sigma$



$(0\Sigma^*(01+10))^*$



DFAs have a finite number of states, so once a string gets beyond a certain length, the DFA must repeat one or more states.

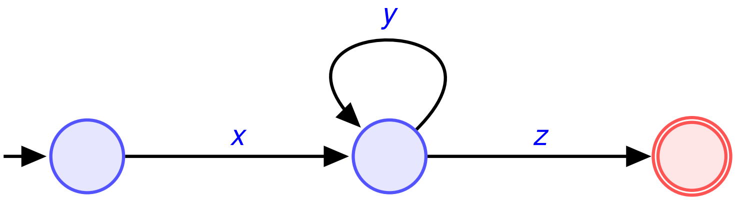
- When a DFA repeats a state (say q_8), we may divide the input string up into three substrings:
  1. The substring x before the first occurrence of q_8
  2. The substring y between the first and last occurrence of q_8
  3. The substring z after the last occurrence of q_8
- It follows that if the DFA accepts xyz, then it will also accept $xz, xyz, xyyz, xyyyz, \ldots$

Therefore, for any RL, once a string extends above a certain length (the pumping length p) it becomes possible to divide the string up into three substrings xyz, in such a way that xy^*z is also a member of that language

- x, z can be ε
- y cannot be ε
- $|xy| \leq p$

The Pumping Lemma – informal

Observation: path from the start to the accept state for a string xyz:



The strings x and z can be ε, but **not** y.

Idea of the Pumping Lemma

Any "sufficiently long" string in a regular language can be broken into three parts such that if we **"pump" the middle part** (repeat it zero or more times) then the result would still be in the language.

The Pumping Lemma – formal

Pumping Lemma

Let L be a regular language. Then there exists a constant p such that for every string w in L, with $|w| \geq p$, we can break w into three strings $w = xyz$ such that

1. $y \neq \varepsilon$ (or equivalently $|y| > 0$ or $|y| \neq 0$)
2. $|xy| \leq p$
3. For all $k \geq 0$, the string xy^kz is also in L

The length p is called **the pumping length**.

Its main purpose in practice is to prove that a language is not regular.
That is, if we can show that a language does not have the required property, then we can conclude that it cannot be expressed as a regular expression or recognized by a DFA.

Game!

The Pumping Lemma when used to prove that a language L is **not regular** can be viewed as a “game” between a **Prover** and a **Falsifier** as follows:

❶ **Prover** claims L is regular and fixes the pumping length p.

❷ **Falsifier** challenges **Prover** and picks a string $w \in L$ of length at least p symbols.

❸ **Prover** writes $w = xyz$ where $|xy| \leq p$ and $y \neq \varepsilon$.

❹ **Falsifier** wins by finding a value for k such that xy^kz is **not** in L. If it cannot then it fails and **Prover** wins.

The language L is not regular if **Falsifier** can always win systematically.

Example ($L = \{a^n b^n \mid n \geq 0\}$)

❶ **Prover** claims L is regular and fixes the pumping length p.

❷ **Falsifier** challenges **Prover** and picks $w = a^p b^p \in L$ $(|w| = 2p \geq p)$.

❸ **Prover** tries to write w as $w = xyz$ but sees that the condition $|xy| \leq p$ forces x and y to only contain the symbol a. Also, y cannot just be the empty string because of the condition $y \neq \varepsilon$. So the only option available is to have $xy = a^m$ for some $m \geq 1$, and then we get $z = a^{p-m} b^p$.

❹ **Falsifier** now sees that $xy^0z, xy^2z, xy^3z, \ldots$ all do not belong to L because they either have less or more a's than there are b's. So, any such string will be enough for **Falsifier** to win the game.

Example ($L = \{ww \mid w \in \{0,1\}^*\}$)

❶ **Prover** claims L is regular and fixes the pumping length p.

❷ **Falsifier** challenges **Prover** and Choose $w = (0^p1)(0^p1) \in L$. This has length $|w| = (p+1)+(p+1) = 2p+2 \geq p$.

❸ **Prover** The PL now guarantees that w can be split into three substrings $w = xyz$ satisfying $|xy| \leq p$ and $y \neq \varepsilon$.

❹ **Falsifier** Since $w = (0^p1)(0^p1) = xyz$ with $|xy| \leq p$ then we must have that y only contains the symbol 0. We can then pump y and produce $xy^2z = xyyz \notin L$, causing a contradiction.
So L is not regular.

Food for thought

- If modern computer = finite state machine: a finite amount of data, say $1\text{TB} = 1024^4 \times 8 = 2^{43}$ bits of information, i.e. a maximum of $2^{2^{43}} \approx 10^{2,647,887,844,335}$ states – a finite number still!
- Consequently, it is unable to recognize the (entire) language $a^n b^n$
- This means that at some point, my computer can no longer count the number of a's in a string. This occurs when the number of a's becomes greater than $2^{2^{43}}$.
- We are assuming that the computer is not storing the string (in which case it would just run out of memory anyway)
- At 3GHz, this would take…a length of time so inconceivably huge that the age of the universe would be negligible by comparison

Space Complexity: Constant Space $\longleftrightarrow$ NFAs

- Finite State Automaton: good model for algorithms which require **constant space**.
  Space complexity $O(1)$, i.e. space used does not grow with respect to the input size.
- Some languages cannot be recognized by NFAs.
  Space used must grow with respect to input size.
- We will see a more powerful model of computation next week!

“Language recognition” and “Language generation”

	Regular Languages
Recognizer:	NFA/DFA
Generator:	RegEx / Regular **Grammar**

Grammars:

- more powerful at describing languages than RegEx’s.
  Can be used to describe all RLs, as well as some non regular ones
- first used in the study of natural languages.

Grammars

Grammars are defined by **production rules** such as

$$\begin{aligned} A &\to \mathrm{a}\,A\,\mathrm{b} \\ A &\to B \\ B &\to \varepsilon \end{aligned}$$

The rules of the grammar represent possible *replacements*
e.g. $A \to \mathrm{a}\,A\,\mathrm{b}$ means the variable A may be replaced with the string $\mathrm{a}\,A\,\mathrm{b}$.

- **Lower case symbols** a and b are **terminals** (like symbols for NFAs). They constitute the alphabet for the grammar.
- **Upper case symbols** A and B are **variables** (or **non-terminals**). They are to be replaced by terminals or strings.
- A is the **start variable**.

Derivation of strings – generation of a language

$$\begin{array}{lcl} A & \rightarrow & \mathrm{a}A\mathrm{b} \\ A & \rightarrow & B \\ B & \rightarrow & \varepsilon \end{array}$$

Commencing with the start variable, these replacements can be used iteratively to produce strings e.g.

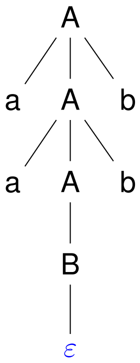
$$A \rightarrow \mathrm{a}A\mathrm{b} \rightarrow \mathrm{aa}A\mathrm{bb} \rightarrow \mathrm{aa}B\mathrm{bb} \rightarrow \mathrm{aa}\varepsilon\mathrm{bb} = \mathrm{aabb}$$

This is called a **derivation** of the string aabb.

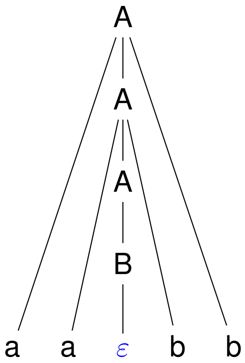
Parse Trees

Diagrammatic way of representing the derivation process.

$$A \to aAb \to aaAbb \to aaBbb \to aa\varepsilon bb = aabb$$



or



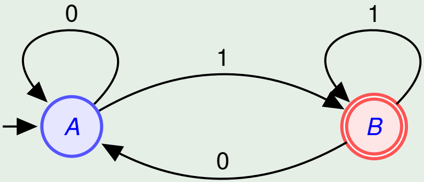
RL $\leftrightarrow$ DFA/NFA/RegEx $\leftrightarrow$ Regular Grammar

- Make a variable V_i for each state q_i
- Add a rule $V_i \rightarrow aV_j$ for each transition from q_i to q_j on symbol a.
- Add a rule $V_i \rightarrow \varepsilon$ if q_i is an accepting state

Example

Variables: A, B.

$$\begin{aligned} A &\rightarrow 0A \\ A &\rightarrow 1B \\ \\ B &\rightarrow 1B \\ B &\rightarrow 0A \\ \\ B &\rightarrow \varepsilon \end{aligned}$$



Notation

To make writing grammars compact, we combine rules starting with the same variable:

$$\left.\begin{array}{lcl} A & \to & 0A \\ A & \to & 1B \\ \\ B & \to & 1B \\ B & \to & 0A \\ B & \to & \varepsilon \end{array}\right\} \quad \text{become} \quad \left\{\begin{array}{lcl} A & \to & 0A \mid 1B \\ B & \to & 1B \mid 0A \mid \varepsilon \end{array}\right.$$

Here the $|$ symbol means “or” or “union”.