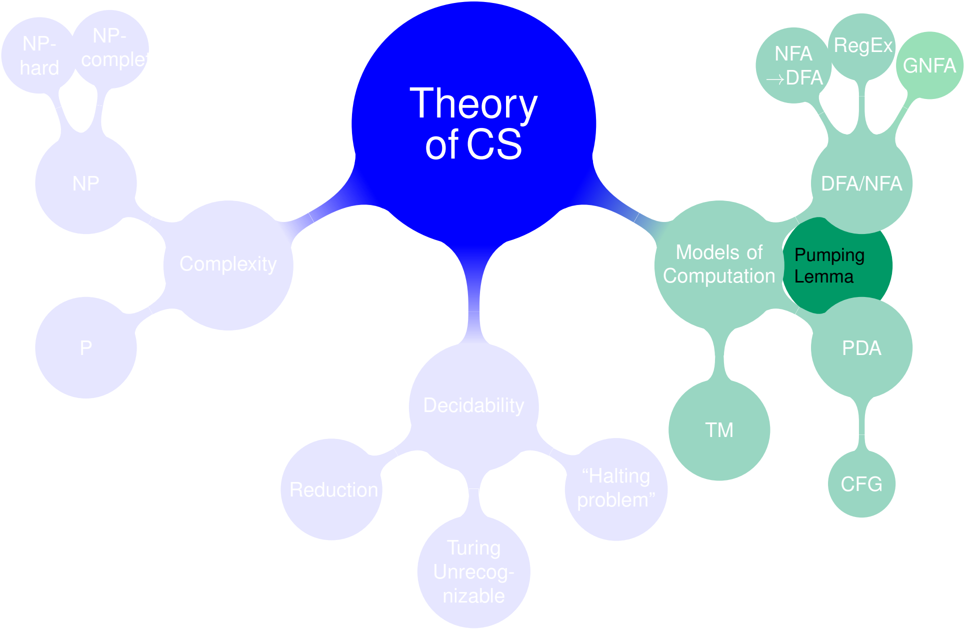
Limitations of the Regular Languages

The Pumping Lemma

Dr Kamal Bentahar

School of Science, Coventry University

Lecture 4


NP-hard
NP-complete
NP
Complexity
P
Theory of CS
Decidability
Reduction
Turing Unrecog-nizable
“Halting problem”
Models of Computation
NFA →DFA
RegEx
GNFA
DFA/NFA
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.

Types of 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.

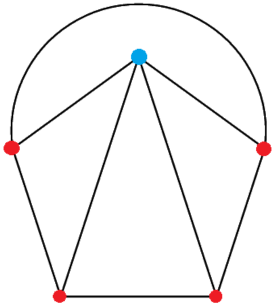
However, if a languages is *not regular* then how can we show that?!

Is it raining now? – example of proof by contradiction

- Is it raining now?
- Suppose it is.
- Let us go outside where it is supposed to be raining.
  - If it is raining then we should get wet.
    (No umbrella, etc.)
- However, we did not get wet!
- Thus, it is **not** raining!

Eulerian paths – example of proof by contradiction

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



- Suppose it is possible.
- How many times would each vertex be visited?
  - Every time a vertex is entered, it is also exited.
  - So, each vertex must have an **even number** of neighbours.
  - The **starting** and **ending** vertices are exceptions: **odd number** of neighbours.
  - There can only be 0 or 2 such exceptions.
- However, this graph has 4 exceptions!
- Thus, it is impossible to traverse this graph by travelling along each path exactly once.

Types of proofs – Proof by contradiction

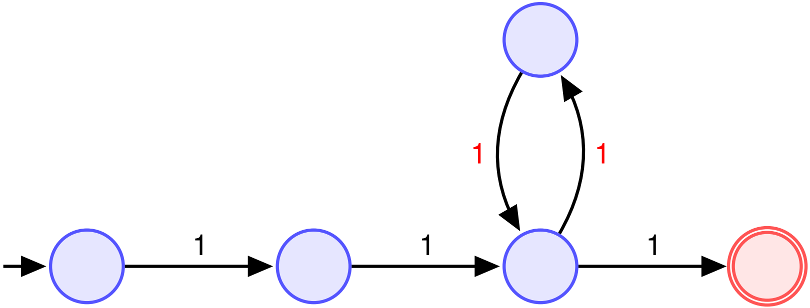
To prove a language is not regular, we can use **proof by contradiction.**

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

Let us try to understand the regular languages (RLs) a bit more...

- Let us examine some examples in the next few slides...
- For each automaton, let us think about 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 takes a “straight path” to the accept state
- 11111 goes through a loop.
- Repeating the looped part produces longer strings:
  11 | 11 | 11 | 1, 11 | 11 | 11 | 11 | 1, 11 | 11 | 11 | 11 | 11 | 1, . . .
- In fact, we can also omit the 11 loop to get: 111.

We say: we **pump** the substring 11.
2, the length of 11, is called: **pumping length.**

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.

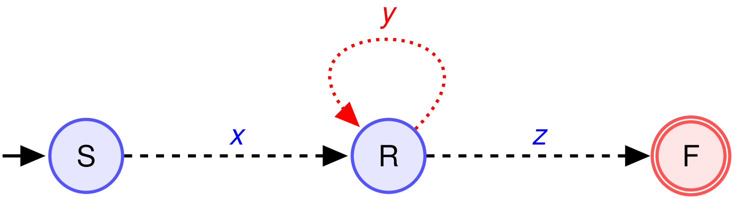
Let L be a regular language over a unary alphabet $\Sigma = \{1\}$.
The language L is:

- either **finite**, in which case it is regular, trivially;
- or **infinite**, in which case its DFA will have to **loop**:
  - The DFA that recognizes 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.

A property satisfied by all RLs

Finite number of states $\rightarrow$ DFA repeats one or more states if the string is long.



- When a DFA repeats a state R, divide the string into 3 parts:
  1. The substring x before the first occurrence of R
  2. The substring y between the first and last occurrence of R
  3. The substring z after the last occurrence of R
- x, z can be ε **but** y cannot be ε. (y forms a genuine loop.)
- Then, if the DFA accepts xyz then it accepts all of:

  $xz, xyz, xyyz, xyyyz, \ldots$

For any RL L, it is possible to divide an **accepted string**, that is "long enough", into 3 substrings xyz, in such a way that xy^*z is a subset of L.

The Pumping Lemma (PL)

- We will denote a **pumping length** by p.
- The precise meaning of “long enough” will be: $|w| \geq p$.
- y has to be in the first p symbols of w.

Pumping Lemma (PL)

Let L be a regular language. Then, there exists a constant p such that every string w from L, with $|w| \geq p$, can be broken into three substrings xyz such that

1. $y \neq \varepsilon$ (or equivalently: $|y| \neq 0$ or $|y| > 0$)
2. $|xy| \leq p$ (y is in the first p symbols of w)
3. For any $k \geq 0$, the string xy^kz is also in L ($xy^*z \subset L$)

Its main purpose in practice is to prove that a language is **not** regular.

That is, if we can show that a language L does not have this property, then we conclude that L cannot be recognized by a DFA/NFA or expressed as a regular expression.

The PL Game!

When the PL is used to prove that a language L is **not regular**, the proof can be viewed as a “game” between a **Prover** and a **Falsifier** as follows:

❶ **Prover** claims L is regular and fixes a 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 **Falsifier** always wins then L is **not regular**.
If **Prover** always wins then L **may be** regular.

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

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

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

$$w = \underbrace{a \ldots\ldots\ldots\ldots a}_{p \text{ symbols}} \underbrace{b \ldots\ldots\ldots b}_{p \text{ symbols}}$$

❸ **Prover** tries to split $w = a \ldots ab \ldots b$ into xyz such that $|xy| \leq p$

$$\underbrace{a \ldots}_{x} \underbrace{\ldots\ldots\ldots}_{y} \underbrace{\ldots ab \ldots\ldots b}_{z}$$

Since y must be within the first p symbols then y is made of a's only.

❹ **Falsifier** now can for example build

$$xy^2z = xyyz = \underbrace{a \ldots\ldots\ldots\ldots a}_{\text{more than } p \text{ symbols}} \underbrace{b \ldots\ldots\ldots b}_{\text{still } p \text{ symbols}}$$

Hence $xy^2z \notin L$, and L is not regular.

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

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

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

$$w = \underbrace{0\ldots\ldots\ldots\ldots0}_{p\text{ symbols}}1\underbrace{0\ldots\ldots\ldots0}_{p\text{ symbols}}1$$

❸ **Prover** tries to split $w = \underbrace{0\ldots01}\underbrace{0\ldots01}$ into xyz such that $|xy| \leq p$

$$\underbrace{0\ldots}_{x}\underbrace{\ldots\ldots\ldots}_{y}\underbrace{\ldots010\ldots\ldots01}_{z}$$

Since y must be within the first p symbols then y is made of 0's only.

❹ **Falsifier** pumps y to produce

$$xy^2z = \underbrace{0\ldots\ldots\ldots\ldots0}_{\text{more than } p \text{ symbols}}1\underbrace{0\ldots\ldots\ldots0}_{\text{still } p \text{ symbols}}1$$

Hence $xy^2z \notin L$, and L is not regular.

Example ($L = \{a^i b^j \mid i > j\}$)

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

❷ **Falsifier** challenges **Prover** and chooses $w = a^{p+1}b^p$.
Here $|w| = (p+1) + p \geq p$.

$$w = \underbrace{a \ldots\ldots\ldots\ldots a}_{p+1 \text{ symbols}} \underbrace{b \ldots\ldots\ldots b}_{p \text{ symbols}}$$

❸ **Prover** splits $w = a \ldots aab \ldots b$ into xyz:

$$\underbrace{a \ldots}_{x} \underbrace{\ldots\ldots}_{y} \underbrace{\ldots aab \ldots\ldots b}_{z}$$

So y is made of a's only.

❹ **Falsifier** pumps y **down** and forms $xy^0z = xz$

$$xy^0z = \underbrace{a \ldots\ldots\ldots\ldots a}_{\text{at most } p \text{ symbols}} \underbrace{b \ldots\ldots\ldots b}_{\text{still } p \text{ symbols}}$$

Hence $xy^0z \notin L$, and L is not regular.

Food for thought

If “modern computer” = Finite Automaton then:

- We can only store a fixed 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!
- So, our “modern computer” is not even able to recognize the (entire) language $a^n b^n$!
  - At some point, our “modern computer” can no longer keep track of how many a’s there are in the input.
    This occurs when the number of a’s becomes greater than $2^{2^{43}}$.
- We have assumed that the input string is not stored in the computer. . . (otherwise, it would just run out of memory anyway).
- However, at 3GHz for example, this would take. . . a length of time so inconceivably huge that the age of the universe would be negligible by comparison. (So, do we care?)

Space Complexity: Constant Space $\longleftrightarrow$ NFAs

- Finite Automaton: good model for algorithms which require **constant space** (i.e. space used does not grow with respect to the input size).
- Some languages cannot be recognized by NFAs.

  *Space used in computation must **grow** with respect to the input size.*
- We will see a more powerful model of computation next week!