Languages and Algorithms for Artificial Intelligence (Third Module)

Historical, Conceptual, and Mathematical Preliminaries

Ugo Dal Lago

University of Bologna, Academic Year 2023/2024

Section 1

A Bit of History

Computation and Computers

- The notion of computation has existed for **thousands of years**, thus prior to the advent of computers.
  - Example: Euclid's algorithm is an effective computational procedure, even if almost 2300 years old.

Computation and Computers

- The notion of computation has existed for **thousands of years**, thus prior to the advent of computers.
  - Example: Euclid's algorithm is an effective computational procedure, even if almost 2300 years old.
- Before the twentieth century, however, computation seen as the process of producing outputs from inputs was defined in many **different** (often inconsistent) ways, and without the appropriate degree of precision.
  - Computation was seen as a **form of art**, rather than a concept to be studied scientifically.

Computation and Computers

- The notion of computation has existed for **thousands of years**, thus prior to the advent of computers.
  - Example: Euclid's algorithm is an effective computational procedure, even if almost 2300 years old.
- Before the twentieth century, however, computation seen as the process of producing outputs from inputs was defined in many **different** (often inconsistent) ways, and without the appropriate degree of precision.
  - Computation was seen as a **form of art**, rather than a concept to be studied scientifically.
- The modern **theory of computation** (ToC) is one of the many gifts humanity has received from science during the first half of the twentieth century.
  - A *precise* definition of the computable has been given, together with many results about it.
  - Actually, many alternative definitions of the computable have been given, but have been proved to be essentially *equivalent*.

Computation and Computers

- The notion of computation has existed for **thousands of years**, thus prior to the advent of computers.
  - Example: Euclid's algorithm is an effective computational procedure, even if almost 2300 years old.
- Before the twentieth century, however, computation seen as the process of producing outputs from inputs was defined in many **different** (often inconsistent) ways, and without the appropriate degree of precision.
  - Computation was seen as a **form of art**, rather than a concept to be studied scientifically.
- The modern **theory of computation** (ToC) is one of the many gifts humanity has received from science during the first half of the twentieth century.
  - A *precise* definition of the computable has been given, together with many results about it.
  - Actually, many alternative definitions of the computable have been given, but have been proved to be essentially *equivalent*.
- In the second half of the twentieth century, ToC has evolved into a **fully fledged** scientific field.

Computation, Science and Technology

- The outfalls of modern theory of computation, since the 1940s, have been twofold:
  1. **From ToC to the other Sciences**
     - Examples: Genomics, Physics, ...
     - Results from the other sciences have had a very strong impact to the theory of computation, as well.
  2. **From ToC to ICT**
     - The theoretical notion of computation was already there when the first modern, electronic, computers appeared.
     - Concepts from the theory of computation (e.g. universality) informed the design of computer from the very beginning.
     - Computation theory has since been catching up with the way ICT is done, in practice.

Computation, Science and Technology

- The outfalls of modern theory of computation, since the 1940s, have been twofold:
  1. **From ToC to the other Sciences**
     - Examples: Genomics, Physics, ...
     - Results from the other sciences have had a very strong impact to the theory of computation, as well.
  2. **From ToC to ICT**
     - The theoretical notion of computation was already there when the first modern, electronic, computers appeared.
     - Concepts from the theory of computation (e.g. universality) informed the design of computer from the very beginning.
     - Computation theory has since been catching up with the way ICT is done, in practice.
- One can easily observe a trend, in the last thirty years:
  - **Concepts** from ToC have more and more influence on the other sciences.
  - **Problems** from ToC are considered worth being solved by, e.g., researchers in mathematics and physics.

Computability and Complexity

- Until the Late 60s, ToC was mainly concerned with understanding **computability**.
  - Main Question: is a certain task computable?
  - If the answer is negative, the task is said to be *uncomputable* (and there can be many ways in which this can happen).
  - This is the so-called **computability theory**.

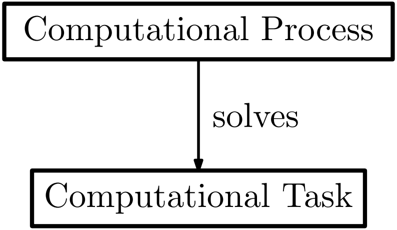
Computability and Complexity

- Until the Late 60s, ToC was mainly concerned with understanding **computability**.
  - Main Question: is a certain task computable?
  - If the answer is negative, the task is said to be *uncomputable* (and there can be many ways in which this can happen).
  - This is the so-called **computability theory**.
- Since the pioneering works by Hartmanis, Stearns, and Cobham (all from the late 60s), a new branch of ToC has emerged, which deals with **efficiency**.
  - Main Question: is a certain task solvable *in a reasonable amount* of time, or space (i.e. working memory)?
  - If the answer is negative, the task is maybe computable, but requires so much time or space, that it pecomes *practically* uncomputable.
  - This new branch of ToC has been named **computational complexity theory**.
- This module will mainly deal with computational complexity theory.

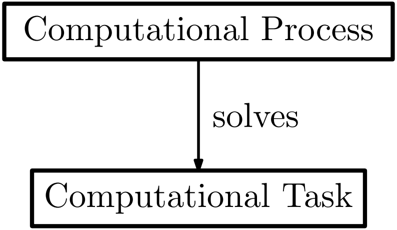
Section 2

Modelling Computation

The Standard Paradigm



The Standard Paradigm



- Processes and tasks **are different**.
- In some cases, there could be **many** distinct processes solving the same tasks.
- In some other cases, the task is so hard that **no** process can solve it.

The Multiplication Problem

- **Task**: multiplying two natural numbers a and b both expressible with n digits, by performing operations on digits.

The Multiplication Problem

- **Task**: multiplying two natural numbers a and b both expressible with n digits, by performing operations on digits.
- One **process** solving the task above consists in reducing multiplication to addition:

$$a \cdot b = \underbrace{a + a + \ldots + a}_{b \text{ times}}$$

Since addition of two n-digit numbers is a relatively easy process which can easily be carried out in a linear amount of steps, the total number of steps is proportional to $b \cdot n$. Call this process `RepeatedAddition`.

The Multiplication Problem

- **Task**: multiplying two natural numbers a and b both expressible with n digits, by performing operations on digits.
- One **process** solving the task above consists in reducing multiplication to addition:

$$a \cdot b = \underbrace{a + a + \ldots + a}_{b \text{ times}}$$

  Since addition of two n-digit numbers is a relatively easy process which can easily be carried out in a linear amount of steps, the total number of steps is proportional to $b \cdot n$. Call this process `RepeatedAddition`.
- Another **process** solving the task above consists in rather multipling a and b with the elementary school direct algorithm, which takes an amount of steps which is proportional to $n \cdot n$. Call this process `GridMethod`

The Multiplication Problem

- There is an exponential difference between the performances of `RepeatedAddition` and `GridMethod`.
  - Indeed, b can be exponential in n, i.e. at most $10^n - 1$.
  - When, e.g., n is 100, there is a huge different between $100 \cdot 100$ and $(100^{100} - 1) \cdot 100$.
  - If each basic instruction takes a millisecond, `GridMethod` would take one second, while `RepeatedAddition` would take more than 10^{80} years.
- Distinct processes can solve the same task in very different ways, and not all of them are acceptable.
- `GridMethod` witnesses the fact that the task is in a class of tasks called $\mathbf{P}$

The Wedding Problem

- **Task**: given a set L of candidate invitees, and a list $I \subseteq L \times L$ of incompatible pairs of candidate invitees, find the largest subset of L composed of compatible invitees.

The Wedding Problem

- **Task**: given a set L of candidate invitees, and a list $I \subseteq L \times L$ of incompatible pairs of candidate invitees, find the largest subset of L composed of compatible invitees.
- A **process** that is for obvious reasons a correct way to accomplish the task above consists in considering *all* possible subsets of L, and check for the presence of incompatibilities in each of them, at the same tracking their size.

The Wedding Problem

- **Task**: given a set L of candidate invitees, and a list $I \subseteq L \times L$ of incompatible pairs of candidate invitees, find the largest subset of L composed of compatible invitees.
- A **process** that is for obvious reasons a correct way to accomplish the task above consists in considering *all* possible subsets of L, and check for the presence of incompatibilities in each of them, at the same tracking their size.
- If L has n elements, how many subset does L have?
  - This turns out to be 2^n, which is impractical.

The Wedding Problem

- **Task**: given a set L of candidate invitees, and a list $I \subseteq L \times L$ of incompatible pairs of candidate invitees, find the largest subset of L composed of compatible invitees.
- A **process** that is for obvious reasons a correct way to accomplish the task above consists in considering *all* possible subsets of L, and check for the presence of incompatibilities in each of them, at the same tracking their size.
- If L has n elements, how many subset does L have?
  - This turns out to be 2^n, which is impractical.
- A decisional version of this problem is known to be in the class **NP**, and we will talk about that!

The Wedding Problem

- **Task**: given a set L of candidate invitees, and a list $I \subseteq L \times L$ of incompatible pairs of candidate invitees, find the largest subset of L composed of compatible invitees.
- A **process** that is for obvious reasons a correct way to accomplish the task above consists in considering *all* possible subsets of L, and check for the presence of incompatibilities in each of them, at the same tracking their size.
- If L has n elements, how many subset does L have?
  - This turns out to be 2^n, which is impractical.
- A decisional version of this problem is known to be in the class **NP**, and we will talk about that!
- The real question is **can we do better** than exhaustive enumeration?
  - If we manage to do significantly better, we would have solved the question of whether **NP** is equal to **P**.

What *is* a Process?

- In the Theory of Computation, a process is taken to be an **algorithm**.

What *is* a Process?

- In the Theory of Computation, a process is taken to be an **algorithm**.
- Giving a precise definition of an algorithm is not possible *at this point*, but any algorithm must satisfy the following constraints:
  - It must be a *finite* description of a series of computation *steps*
  - Each step must be *elementary*.
  - The way the next step is determined must be *deterministic*

What *is* a Process?

- In the Theory of Computation, a process is taken to be an **algorithm**.
- Giving a precise definition of an algorithm is not possible *at this point*, but any algorithm must satisfy the following constraints:
  - It must be a *finite* description of a series of computation *steps*
  - Each step must be *elementary*.
  - The way the next step is determined must be *deterministic*
- Any program (say, written in `Python`) can be seen as a description of an algorithm!
  - It is a very **high-level** description, and the resources necessary to execute each instruction are hard to estimate.

What *is* a Process?

- In the Theory of Computation, a process is taken to be an **algorithm**.
- Giving a precise definition of an algorithm is not possible *at this point*, but any algorithm must satisfy the following constraints:
  - It must be a *finite* description of a series of computation *steps*
  - Each step must be *elementary*.
  - The way the next step is determined must be *deterministic*
- Any program (say, written in `Python`) can be seen as a description of an algorithm!
  - It is a very **high-level** description, and the resources necessary to execute each instruction are hard to estimate.
- In this course, we will give a different, more **abstract** and **low level**, definition.

Proving the Nonexistence of Algorithms

- Proving tasks **not solvable** by processes beyond a certain level of efficiency be necessary for a proper classification of tasks.

Proving the Nonexistence of Algorithms

- Proving tasks **not solvable** by processes beyond a certain level of efficiency be necessary for a proper classification of tasks.
- But **how could we proceed**? There are *infinitely many* processes solving a certain task, so we cannot exhaustively examine them.

Proving the Nonexistence of Algorithms

- Proving tasks **not solvable** by processes beyond a certain level of efficiency be necessary for a proper classification of tasks.
- But **how could we proceed**? There are *infinitely many* processes solving a certain task, so we cannot exhaustively examine them.
- We are **very rarely** able to prove the nonexistence of efficient algorithm.
    - Finding mathematical techniques which allow us to do so is *the crux* of computational complexity theory.

Proving the Nonexistence of Algorithms

- Proving tasks **not solvable** by processes beyond a certain level of efficiency be necessary for a proper classification of tasks.
- But **how could we proceed**? There are *infinitely many* processes solving a certain task, so we cannot exhaustively examine them.
- We are **very rarely** able to prove the nonexistence of efficient algorithm.
  - Finding mathematical techniques which allow us to do so is *the crux* of computational complexity theory.
- Rather than proving the non-existence of certain algorithms, complexity theory **interrelates** different tasks.

Some Research Questions in Complexity Theory

- The **P** vs. **NP** question.
- Can the use of randomness help in speeding up computation?
- Can hard tasks become easier if we allow algorithms to err on a relatively small subset of the inputs?
- Can we exploit the hardness of certain tasks?
- Can we make use of some counterintuitive properties of quantum mechanics to build faster computing devices?

Section 3

Some Mathematical Preliminaries

Sets and Numbers

- The cardinality of any set X is indicated as $|X|$.
  - It can be finite or infinite.
- We will mainly work with discrete numbers.
  - $\mathbb{N}$ is the set of natural numbers, while $\mathbb{Z}$ is the set of integers.
- We say that a condition $P(n)$ depending on $n \in \mathbb{N}$ holds for **sufficiently large** n if there is $N \in \mathbb{N}$ such that $P(n)$ holds for every $n > N$.
- Some common (and useful notation):
  - For a given real number x, $\lceil x \rceil$ is the the smallest element of $\mathbb{Z}$ such that $\lceil x \rceil \geq x$. Whenever a real number x is use in place of anatural number, we implicitly read it as $\lceil x \rceil$
  - For a natural number n, $[n]$ is the set $\{1, \ldots, n\}$.
  - Contrarily to the common mathematical notation, $\log x$ is **base** 2 logarithm.

Strings

- If S is a finite set, then a **string** over the alphabet S is a finite, ordered, possibly empty, tuple of elements from S.
  - Most often, the alphabet S will be the set $\{0, 1\}$.
- The set of all strings over S of length exactly $n \in \mathbb{N}$ is indicated as S^n (where S^0 is the set containing only the empty string ε).
- The set of *all strings* over S is $\bigcup_{n=0}^{\infty} S^n$ and is indicated as S^*.
- The concatenation of two strings x and y over S is indicated as xy or as $x \cdot y$. The string over S obtained by concatenating x with itself $k \in \mathbb{N}$ times is indicated as x^k.
  - Strings form a monoid!
- The **length** of a string x is indicated as $|x|$.
- Examples

$$\{0, 1\}^2 = \{00, 01, 10, 11\} \qquad |000101| = 6$$

Tasks as Functions

- One of the principles of computational complexity is that any task of interest consists in *computing a function* from $\{0,1\}^*$ to itself.
  - Is it too strong an assumption?

Tasks as Functions

- One of the principles of computational complexity is that any task of interest consists in *computing a function* from $\{0,1\}^*$ to itself.
  - Is it too strong an assumption?
- The notion of function is sufficiently general, indeed.
  - Most tasks (but not all!) consists in turning an input to an output.

Tasks as Functions

- One of the principles of computational complexity is that any task of interest consists in *computing a function* from $\{0,1\}^*$ to itself.
  - Is it too strong an assumption?
- The notion of function is sufficiently general, indeed.
  - Most tasks (but not all!) consists in turning an input to an output.
- How about the emphasis on strings?
  - If the input and/or the output are not strings, but they are taken from a discrete set, they can be **represented** as strings, following some encoding, which however should be as simple as possible.

Tasks as Functions

- One of the principles of computational complexity is that any task of interest consists in *computing a function* from $\{0,1\}^*$ to itself.
  - Is it too strong an assumption?
- The notion of function is sufficiently general, indeed.
  - Most tasks (but not all!) consists in turning an input to an output.
- How about the emphasis on strings?
  - If the input and/or the output are not strings, but they are taken from a discrete set, they can be **represented** as strings, following some encoding, which however should be as simple as possible.
- Summing up, we always assume that the task we want to solve **is given** as a function $f : A \to B$ where both the domain A and B are discrete (i.e. countable) sets, sometime leaving the encoding of A and B into $\{0,1\}^*$ implicit.
- The encoding of any element x of A as a string is often indicated as $\llcorner x \lrcorner$ or simply as x.

Representing Objects as Strings: Examples

- Strings in S^*, where $S = \{a, b, c\}$.
  - a can be encoded as 00, b becomes 01 and c becomes 10.
  - this way, e.g., $abbc$ becomes 00010110.

Representing Objects as Strings: Examples

- Strings in S^*, where $S = \{a, b, c\}$.
  - a can be encoded as 00, b becomes 01 and c becomes 10.
  - this way, e.g., $abbc$ becomes 00010110.
- Natural numbers.
  - We can take of the many encodings of natural numbers as strings.
  - As an example, 12 becomes 1100.

Representing Objects as Strings: Examples

- Strings in S^*, where $S = \{a, b, c\}$.
  - a can be encoded as 00, b becomes 01 and c becomes 10.
  - this way, e.g., $abbc$ becomes 00010110.
- Natural numbers.
  - We can take of the many encodings of natural numbers as strings.
  - As an example, 12 becomes 1100.
- Pairs of binary strings, i.e. $\{0, 1\}^* \times \{0, 1\}^*$.
  - One can encode the pair (x, y) as the string $x\#y$ over the alphabet $\{0, 1, \#\}$, and the proceed as before.

Representing Objects as Strings: Examples

- Strings in S^*, where $S = \{a, b, c\}$.
  - a can be encoded as 00, b becomes 01 and c becomes 10.
  - this way, e.g., $abbc$ becomes 00010110.
- Natural numbers.
  - We can take of the many encodings of natural numbers as strings.
  - As an example, 12 becomes 1100.
- Pairs of binary strings, i.e. $\{0,1\}^* \times \{0,1\}^*$.
  - One can encode the pair (x, y) as the string $x\#y$ over the alphabet $\{0, 1, \#\}$, and the proceed as before.

$\vdots$

Languages and Decision Problems

- An important class of functions from $\{0,1\}^*$ to $\{0,1\}^*$ are those whose range are strings of length exactly one, called **boolean functions**.
  - They are the so -called *characteristic* functions.

Languages and Decision Problems

- An important class of functions from $\{0,1\}^*$ to $\{0,1\}^*$ are those whose range are strings of length exactly one, called **boolean functions**.
  - They are the so -called *characteristic* functions.
- We identify such a function f with the subset $\mathcal{L}_f$ of $\{0,1\}^*$ defined as follows:

$$\mathcal{L}_f = \{x \in \{0,1\}^* \mid f(x) = 1\}.$$

- Any subset of S^* (the set of all strings over an alphabet S) is usually called a **language**.
- This way, a **decision problem** for a given language $\mathcal{M}$ (i.e. does $x \in \{0,1\}^*$ is in $\mathcal{M}$?) can be seen as the task of computing f such that $\mathcal{M} = \mathcal{L}_f$.

Asymptotic Notation

- A function $f : \mathbb{N} \to \mathbb{N}$ is $O(g)$ if there is a positive real constants c such that $f(n) \leq c \cdot g(n)$ for sufficiently large n.
  - Example: the function $n \mapsto 3 \cdot n^2 + 4 \cdot n$ is $O(n^2)$, but also $O(n^3)$, and certainly $O(2^n)$. It is not, however, $O(n)$.
- A function $f : \mathbb{N} \to \mathbb{N}$ is $\Omega(g)$ if there if a positive real constants c such that $f(n) \geq c \cdot g(n)$ for sufficiently large n
  - Example: the function $n \mapsto 3 \cdot n^2 + 4 \cdot n$ is $\Omega(n^2)$, but also $\Omega(n)$, but not $\Omega(n^3)$.
- A function f is $\Theta(g)$ if f is both $O(g)$ and $\Omega(g)$.
  - Example: the function $n \mapsto 3 \cdot n^2 + 4 \cdot n + 7$ is $\Theta(n^2)$.

Asymptotic Notation

- A function $f : \mathbb{N} \to \mathbb{N}$ is $O(g)$ if there is a positive real constants c such that $f(n) \leq c \cdot g(n)$ for sufficiently large n.
  - Example: the function $n \mapsto 3 \cdot n^2 + 4 \cdot n$ is $O(n^2)$, but also $O(n^3)$, and certainly $O(2^n)$. It is not, however, $O(n)$.
- A function $f : \mathbb{N} \to \mathbb{N}$ is $\Omega(g)$ if there if a positive real constants c such that $f(n) \geq c \cdot g(n)$ for sufficiently large n
  - Example: the function $n \mapsto 3 \cdot n^2 + 4 \cdot n$ is $\Omega(n^2)$, but also $\Omega(n)$, but not $\Omega(n^3)$.
- A function f is $\Theta(g)$ if f is both $O(g)$ and $\Omega(g)$.
  - Example: the function $n \mapsto 3 \cdot n^2 + 4 \cdot n + 7$ is $\Theta(n^2)$.
- Studying in which relation two functions f and g are can be done by studying the limit of $\frac{f(n)}{g(n)}$ for n tending to infinity.

Thank You!

Questions?