Languages and Algorithms for Artificial Intelligence (Third Module)

Polynomial Time Computable Problems

Ugo Dal Lago

University of Bologna, Academic Year 2022/2023

Complexity Classes

- A **complexity class** is a set of *tasks* which can be computed within some prescribed resource bounds.
  - It is *not* a set of TMs, although it is defined based on TMs.
  - Typically, the task we are interested at are decision problems, or equivalently languages (i.e. subsets of $\{0,1\}^*$).

Complexity Classes

- A **complexity class** is a set of *tasks* which can be computed within some prescribed resource bounds.
  - It is *not* a set of TMs, although it is defined based on TMs.
  - Typically, the task we are interested at are decision problems, or equivalently languages (i.e. subsets of $\{0,1\}^*$).
- Let $T : \mathbb{N} \to \mathbb{N}$. A language $\mathcal{L}$ is in the class $\mathbf{DTIME}(T(n))$ iff *there is* a TM deciding $\mathcal{L}$ and running in time $n \mapsto c \cdot T(n)$ for some constant c.

Complexity Classes

- A **complexity class** is a set of *tasks* which can be computed within some prescribed resource bounds.
    - It is *not* a set of TMs, although it is defined based on TMs.
    - Typically, the task we are interested at are decision problems, or equivalently languages (i.e. subsets of $\{0,1\}^*$).
- Let $T : \mathbb{N} \to \mathbb{N}$. A language $\mathcal{L}$ is in the class $\mathbf{DTIME}(T(n))$ iff *there is* a TM deciding $\mathcal{L}$ and running in time $n \mapsto c \cdot T(n)$ for some constant c.
- The letter "D" in $\mathbf{DTIME}(\cdot)$ refers to *determinism*: the machines on which the class is based work deterministically.
- Should we study efficiently solvable tasks by way of classes in the form $\mathbf{DTIME}(T(n))$?
    - The answer is bound to be negative, because these classes are not **robust**, they depend too much on the underlying computational model.
    - We need a larger class.

The Class $\mathbf{P}$

- The class $\mathbf{P}$ is defined as follows:

$$\mathbf{P} = \bigcup_{c \geq 1} \mathbf{DTIME}(n^c).$$

- In other words, the class $\mathbf{P}$ includes all those languages $\mathcal{L}$:
  1. which can be decided by a TM;
  2. working in time P;
  3. where P is a any polynomial.

The Class $\mathbf{P}$

- The class $\mathbf{P}$ is defined as follows:

$$\mathbf{P} = \bigcup_{c \geq 1} \mathbf{DTIME}(n^c).$$

- In other words, the class $\mathbf{P}$ includes all those languages $\mathcal{L}$:
  1. which can be decided by a TM;
  2. working in time P;
  3. where P is a any polynomial.
- Indeed, for any any polynomial P there are $c, d > 0$ such that $P(n) \leq c \cdot n^d$ for sufficiently large n.

The Class $\mathbf{P}$

- The class $\mathbf{P}$ is defined as follows:

$$\mathbf{P} = \bigcup_{c \geq 1} \mathbf{DTIME}(n^c).$$

- In other words, the class $\mathbf{P}$ includes all those languages $\mathcal{L}$:
  1. which can be decided by a TM;
  2. working in time P;
  3. where P is a any polynomial.
- Indeed, for any any polynomial P there are $c, d > 0$ such that $P(n) \leq c \cdot n^d$ for sufficiently large n.
- Please observe that c and d can be arbitrarily large, so a TM deciding $\mathcal{L}$ and working in time $10^{20} \cdot n^{10^{30}}$ is a witness of $\mathcal{L}$ being in $\mathbf{P}$.

The Class $\mathbf{P}$

- The class $\mathbf{P}$ is defined as follows:

$$\mathbf{P} = \bigcup_{c \geq 1} \mathbf{DTIME}(n^c).$$

- In other words, the class $\mathbf{P}$ includes all those languages $\mathcal{L}$:
  1. which can be decided by a TM;
  2. working in time P;
  3. where P is a any polynomial.
- Indeed, for any any polynomial P there are $c, d > 0$ such that $P(n) \leq c \cdot n^d$ for sufficiently large n.
- Please observe that c and d can be arbitrarily large, so a TM deciding $\mathcal{L}$ and working in time $10^{20} \cdot n^{10^{30}}$ is a witness of $\mathcal{L}$ being in $\mathbf{P}$.
- $\mathbf{P}$ is generally considered as *the* class of efficiently decidable languages.

The (Strong) Church-Turing Thesis

- But, again, why basing complexity theory on TMs? They are a rather simplicistic model!

The (Strong) Church-Turing Thesis

- But, again, why basing complexity theory on TMs? They are a rather simplicistic model!
- **The Church-Turing Thesis**
  - Every physically realizable computer can be simulated by a TM with a (possibly *very large*) overhead in time.
  - The class of computable tasks *would not be larger* (actually, equal!) if formalized in a realistic way, but differently.
  - Most scientists believe in it.

The (Strong) Church-Turing Thesis

- But, again, why basing complexity theory on TMs? They are a rather simplicistic model!
- **The Church-Turing Thesis**
  - Every physically realizable computer can be simulated by a TM with a (possibly *very large*) overhead in time.
  - The class of computable tasks *would not be larger* (actually, equal!) if formalized in a realistic way, but differently.
  - Most scientists believe in it.
- **The Strong Church-Turing Thesis**
  - Every physically realizable computer can be simulated by a TM with a *polynomial* overhead in time (n steps on the computer requires n^c on TMs, where c only depends on the computer), and viceversa.
  - The class $\mathbf{P}$ would be *the same* if defined based on other realistic models of computation.
  - This is more controversial (due to, e.g., quantum computation).

Why Polynomials?

- **P is Robust**
  - As already mentioned, polynomials seem to be the smallest class of bounds which make **P** a robust class.

Why Polynomials?

- **P is Robust**
  - As already mentioned, polynomials seem to be the smallest class of bounds which make **P** a robust class.
- **Exponents are Often Small**
  - In principle, the exponent c bounding the time of any machine deciding $\mathcal{L} \in \mathbf{P}$ can be huge.
  - For many problems of interest and in **P**, there are TMs working within quadratic or cubic bounds.

Why Polynomials?

- **P is Robust**
  - As already mentioned, polynomials seem to be the smallest class of bounds which make **P** a robust class.
- **Exponents are Often Small**
  - In principle, the exponent c bounding the time of any machine deciding $\mathcal{L} \in \mathbf{P}$ can be huge.
  - For many problems of interest and in **P**, there are TMs working within quadratic or cubic bounds.
- **Nice Closure Properties**
  - The class is closed various operations on programs, e.g. composition and bounded loops (with some restrictions!).
  - As a consequence, it is relatively easy to prove that a given problem/task is *in* the class: it suffices to give an algorithm solving the problem and working in polynomial time, without constructing the TM explicitly.

Some Criticisms on **P**

- **Worst-Case is Not Realistic**
  - The definition of **P** is intrinsically based on worst-case complexity: there must be *a* polynomial and *a* TM such that *for every input*. . .
  - It is good enough even if our problem takes little time *on the types of inputs* which arise in practice, and not on *all* of them.
  - Solutions: Average-case Complexity, Approximation Algorithms

Some Criticisms on $\mathbf{P}$

- **Worst-Case is Not Realistic**
  - The definition of $\mathbf{P}$ is intrinsically based on worst-case complexity: there must be *a* polynomial and *a* TM such that *for every input...*
  - It is good enough even if our problem takes little time *on the types of inputs* which arise in practice, and not on *all* of them.
  - Solutions: Average-case Complexity, Approximation Algorithms
- **Alternative Computational Models**
  - Feasibility can also be defined for classes dealing with arbitrary precision computation, with randomized computation, or with quantum computation.
  - Solutions: the class $\mathbf{P}$ can be redefined with other computational models in mind, giving rise to other classes (e.g. $\mathbf{BPP}$ or $\mathbf{BQP}$).
- **Why Just Decision Problems?**
  - As alreday pointed out, not all tasks can be modeled this way.

The Complexity Class **FP**

- Sometime, one would like to classify *functions* rather than *languages*. This can be done by slightly generalizing a couple of concepts we have previously introduced:
  - Let $T : \mathbb{N} \to \mathbb{N}$. A function f is in the class $\mathbf{FDTIME}(T(n))$ iff *there is* a TM computing f and running in time $n \mapsto c \cdot T(n)$ for some constant c.
  - The class **FP** is defined as follows, very similarly to **P**:

$$\mathbf{FP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(n^c).$$

The Complexity Class **FP**

- Sometime, one would like to classify *functions* rather than *languages*. This can be done by slightly generalizing a couple of concepts we have previously introduced:
  - Let $T : \mathbb{N} \to \mathbb{N}$. A function f is in the class $\mathbf{FDTIME}(T(n))$ iff *there is* a TM computing f and running in time $n \mapsto c \cdot T(n)$ for some constant c.
  - The class **FP** is defined as follows, very similarly to **P**:

$$\mathbf{FP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(n^c).$$

- For every $\mathcal{L} \in \mathbf{P}$, the characteristic function f of $\mathcal{L}$ is trivially in **FP**.

The Complexity Class FP

- Sometime, one would like to classify *functions* rather than *languages*. This can be done by slightly generalizing a couple of concepts we have previously introduced:
  - Let $T : \mathbb{N} \to \mathbb{N}$. A function f is in the class $\mathbf{FDTIME}(T(n))$ iff *there is* a TM computing f and running in time $n \mapsto c \cdot T(n)$ for some constant c.
  - The class **FP** is defined as follows, very similarly to **P**:

$$\mathbf{FP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(n^c).$$

- For every $\mathcal{L} \in \mathbf{P}$, the characteristic function f of $\mathcal{L}$ is trivially in **FP**.
- For certain classes of functions (e.g. those corresponding to optimization problems), there are canonical ways to turn a function f into a language $\mathcal{L}_f$
  - In general, however, it is not true that $f \in \mathbf{FP}$ implies $\mathcal{L}_f \in \mathbf{P}$.

Example Problems in **P** or **FP**

- **Lists**: inverting a list, sorting a list, finding the maximum or minimum element in a list, etc.

Example Problems in **P** or **FP**

- **Lists**: inverting a list, sorting a list, finding the maximum or minimum element in a list, etc.
- **Graphs**: reachability, shortest paths, minimum spanning trees, etc.

Example Problems in **P** or **FP**

- **Lists**: inverting a list, sorting a list, finding the maximum or minimum element in a list, etc.
- **Graphs**: reachability, shortest paths, minimum spanning trees, etc.
- **Numbers**: primality test, exponentiation, etc.

Example Problems in **P** or **FP**

- **Lists**: inverting a list, sorting a list, finding the maximum or minimum element in a list, etc.
- **Graphs**: reachability, shortest paths, minimum spanning trees, etc.
- **Numbers**: primality test, exponentiation, etc.
- **Strings**: string matching, approximate string matching, etc.

Example Problems in **P** or **FP**

- **Lists**: inverting a list, sorting a list, finding the maximum or minimum element in a list, etc.
- **Graphs**: reachability, shortest paths, minimum spanning trees, etc.
- **Numbers**: primality test, exponentiation, etc.
- **Strings**: string matching, approximate string matching, etc.
- **Optimization Problems**: linear programming, maximum cost flow, etc.

How to Prove a Task Being in **P** or **FP**

- In theory, one should give a TM working within some polynomial bounds, and prove that the machines decides the language (or computes the function).

How to Prove a Task Being in **P** or **FP**

- In theory, one should give a TM working within some polynomial bounds, and prove that the machines decides the language (or computes the function).
- This is however too cumbersome, and instead of going through TMs, one often goes informal and uses the so called pseudocode.
- Example.
  - Suppose you want to show the following problem to be computable in polynomial time: given two strings $x, y \in \{0, 1\}^*$. determine if x contains an instance of y.
  - A pseudocode solving the problem above is the following:

```
i ← 1;
while i ≤ |x| − |y| + 1 do
    if x[i : i + |y| − 1] = y then
        return True
    else
        i ← i + 1
    end
end
return False
```

How to Prove a Task Being in **P** or **FP**

- How could we be sure that the algorithm above indeed works in *polynomial time*?

$i \leftarrow 1$;
while $i \leq |x| - |y| + 1$ **do**
  if $x[i : i + |y| - 1] = y$ **then**
    return `True`
  else
    $i \leftarrow i + 1$
  end
end
return `False`

How to Prove a Task Being in **P** or **FP**

- How could we be sure that the algorithm above indeed works in *polynomial time*?

$i \leftarrow 1$;
while $i \leq |x| - |y| + 1$ **do**
  if $x[i : i + |y| - 1] = y$ **then**
    return `True`
  else
    $i \leftarrow i + 1$
  end
end
return `False`

- The input can be easily encoded as a binary string.

How to Prove a Task Being in **P** or **FP**

- How could we be sure that the algorithm above indeed works in *polynomial time*?

```
i ← 1;
while i ≤ |x| − |y| + 1 do
    if x[i : i + |y| − 1] = y then
        return True
    else
        i ← i + 1
    end
end
return False
```

- The input can be easily encoded as a binary string.
- The total number of instructions is polynomially bounded.
  - Indeed it is $O(|x|)$.

How to Prove a Task Being in $\mathbf{P}$ or $\mathbf{FP}$

- How could we be sure that the algorithm above indeed works in *polynomial time*?

```
i ← 1;
while i ≤ |x| − |y| + 1 do
    if x[i : i + |y| − 1] = y then
        return True
    else
        i ← i + 1
    end
end
return False
```

- The input can be easily encoded as a binary string.
- The total number of instructions is polynomially bounded.
  - Indeed it is $O(|x|)$.
- All intermediate results are polynomially bounded in length.
  - Indeed, i cannot be greater than $O(|x|)$, thus its length is $O(\lg |x|)$.

How to Prove a Task Being in **P** or **FP**

- How could we be sure that the algorithm above indeed works in *polynomial time*?

$i \leftarrow 1$;
while $i \leq |x| - |y| + 1$ **do**
  if $x[i : i + |y| - 1] = y$ **then**
    return `True`
  else
    $i \leftarrow i + 1$
  end
end
return `False`

- The input can be easily encoded as a binary string.
- The total number of instructions is polynomially bounded.
  - Indeed it is $O(|x|)$.
- All intermediate results are polynomially bounded in length.
  - Indeed, i cannot be greater than $O(|x|)$, thus its length is $O(\lg |x|)$.
- Each instruction takes polynomial time to be simulated.
  - Comparing two strings of length $|y|$ can be done in polynomial time in $|y|$, thus polynomial in $|\llcorner(x, y)\lrcorner|$.

The Class **EXP**

- What can we find if we try to go *beyond* the class **P**, i.e., if we allow TMs to have more time at their disposal?

The Class **EXP**

- What can we find if we try to go *beyond* the class **P**, i.e., if we allow TMs to have more time at their disposal?
- The *next* class of functions $T : \mathbb{N} \to \mathbb{N}$, beyond the polynomials and having nice closure properties is the class of exponential functions.

The Class **EXP**

- What can we find if we try to go *beyond* the class **P**, i.e., if we allow TMs to have more time at their disposal?
- The *next* class of functions $T : \mathbb{N} \to \mathbb{N}$, beyond the polynomials and having nice closure properties is the class of exponential functions.
- The classes **EXP** and **FEXP** are defined as follows:

$$\mathbf{EXP} = \bigcup_{c \geq 1} \mathbf{DTIME}(2^{n^c}) \quad \mathbf{FEXP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(2^{n^c})$$

The Class $\mathbf{EXP}$

- What can we find if we try to go *beyond* the class $\mathbf{P}$, i.e., if we allow TMs to have more time at their disposal?
- The *next* class of functions $T : \mathbb{N} \to \mathbb{N}$, beyond the polynomials and having nice closure properties is the class of exponential functions.
- The classes $\mathbf{EXP}$ and $\mathbf{FEXP}$ are defined as follows:

$$\mathbf{EXP} = \bigcup_{c \geq 1} \mathbf{DTIME}(2^{n^c}) \qquad \mathbf{FEXP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(2^{n^c})$$

- The tasks in these classes *can* be solved mechanically, but *possibly cannot* be solved efficiently.

The Class **EXP**

- What can we find if we try to go *beyond* the class $\mathbf{P}$, i.e., if we allow TMs to have more time at their disposal?
- The *next* class of functions $T : \mathbb{N} \to \mathbb{N}$, beyond the polynomials and having nice closure properties is the class of exponential functions.
- The classes **EXP** and **FEXP** are defined as follows:

$$\mathbf{EXP} = \bigcup_{c \geq 1} \mathbf{DTIME}(2^{n^c}) \quad \mathbf{FEXP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(2^{n^c})$$

- The tasks in these classes *can* be solved mechanically, but *possibly cannot* be solved efficiently.
- Of course, it holds that

$$\mathbf{P} \subseteq \mathbf{EXP} \qquad \mathbf{FP} \subseteq \mathbf{FEXP}$$

The Class **EXP**

- What can we find if we try to go *beyond* the class **P**, i.e., if we allow TMs to have more time at their disposal?
- The *next* class of functions $T : \mathbb{N} \to \mathbb{N}$, beyond the polynomials and having nice closure properties is the class of exponential functions.
- The classes **EXP** and **FEXP** are defined as follows:

$$\mathbf{EXP} = \bigcup_{c \geq 1} \mathbf{DTIME}(2^{n^c}) \qquad \mathbf{FEXP} = \bigcup_{c \geq 1} \mathbf{FDTIME}(2^{n^c})$$

- The tasks in these classes *can* be solved mechanically, but *possibly cannot* be solved efficiently.
- Of course, it holds that

$$\mathbf{P} \subseteq \mathbf{EXP} \qquad \mathbf{FP} \subseteq \mathbf{FEXP}$$

Theorem

The two inclusions above are strict.

Thank You!

Questions?