Graph Algorithms (Spring 2025)

4. Basics of Computational Complexity

Teacher: Arnaud Casteigts *Assistant: Matteo De Francesco*

In this course, we define the usual asymptotic notations. Then, we define two generic complexity classes for time and space and review classes that are particular cases of these, in particular the class P. Then, we spend some time discussing the class NP, NP-hard, and NP-complete. Finally, we discuss a number of classical graph problems and their complexity.

4.1 Asymptotic Notations

We often encounter the notations O, Ω, Θ (pronounce "big O", "big Omega", and "big Theta") and sometimes also the notations o and ω ("little o" and "little omega"). These notations are not exclusive to computer science. They allow us to manipulate mathematical expressions while ignoring details deemed insignificant, in order to simplify calculations.

These notations all have in common the following features:

- They ignore dominated terms (when $n \to \infty$)
- They ignore constant factors (factors that do not involve n)

Example: $2n^3 + 3n^2 + 4$. We ignore the dominated terms: $2n^3 \ \cancel{+3n^2 + 4}$, then we ignore the constant factors: $\cancel{2}n^3$, which leaves us with n^3. Another example: $n + \log n$ simplifies as n, because $\log n$ becomes negligible compared to n when $n \to \infty$. However, $n \cdot \log n$ cannot be simplified, because we do *not* ignore dominated *factors* (be careful!).

Up to these simplifications, the notations are simple to define as follows:

Notation	$o(b)$	$O(b)$	$\Theta(b)$	$\Omega(b)$	$\omega(b)$
Meaning	$< b$	$\leq b$	$= b$	$\geq b$	$> b$

Thus, in this case, we can write $2n^3 + 3n^2 + 4 \in \Theta(n^3)$. We could also write $\in \Omega(n^3)$, $\in o(n^{10})$, or even $\in O(n^3)$ or $\in O(n^{10})$, although these statements are less precise. On the other hand, we cannot write $\in o(n^3)$ nor $\in \Omega(n^4)$. Note that the symbol "=" is often used instead of $\in$, with the same meaning, although it does not denote standard equality. In particular, $a = O(b)$ does not imply $b = O(a)$.

Adjectives

Constant	$\Theta(1)$	Polynomial	$n^{O(1)}$
Logarithmic	$\Theta(\log n)$	Quasi-polynomial	$n^{\log^{O(1)} n}$
Linear	$\Theta(n)$	Exponential	$2^{n^{O(1)}}$
Quasi-linear	$\Theta(n \log n)$	Factorial	$\Theta(n!) = O(n^n)$
Quadratic	$\Theta(n^2)$		

The same adjectives apply to O and Ω (saying "at most" and "at least", respectively). For o and ω, we rather use super-X and sub-X (where X is the adjective of your choice), e.g., $\omega(n)$ is super-linear and $o(n^2)$ is sub-quadratic. These adjectives can apply to algorithms as well as the problems they solve. Note that logarithm bases are not specified, because logarithms in different bases differ only by constant factors (which is very convenient!).

There is some flexibility in the use of these adjectives, their usage being slighly varied among communities.

4.2 Time complexity and space complexity

There are many complexity classes, corresponding to sets of problems whose resolution requires a certain level of resources, generally in terms of time or space, and sometimes other things (randomness, non-determinism, *etc.*).

The classification focuses mostly on *decision problems*, which are problems defined by a question whose answer is YES or NO (a.k.a. formal languages). For example: "Is this graph connected?" is a decision problem. The main two classes are:

- $\mathsf{TIME}(f(n))$: Decision problems solvable in time $O(f(n))$, regardless of space.
- $\mathsf{SPACE}(f(n))$: Decision problems solvable in space $O(f(n))$, regardless of time.

The exact definition of these classes requires specifying the computational model used (Turing machines, RAM machines, circuits, etc.). We'll assume a standard RAM machine (i.e. modern computers), which can address memory directly at any location.

Example

Let $\textsc{Find}(s, c)$ be the problem of deciding whether a certain character c appears in a string s of length n. A simple algorithm consists of scanning the string, character after character, checking if the current character matches c. The check for each character can be done in constant time, so the problem is solvable in time $n \cdot O(1) = O(n)$ and $\textsc{Find} \in \mathsf{TIME}(n)$.

What about space? Apart from the input size (which does not count) and the output bit (YES/NO), this problem uses no extra space, so it is solvable in constant space $O(1)$. Thus, $\textsc{Find} \in \mathsf{SPACE}(1)$, and altogether, $\textsc{Find} \in \mathsf{TIME}(n) \cap \mathsf{SPACE}(1)$.

4.3 Important Complexity Classes

The following classes are well-known special cases of TIME and SPACE:

- $\mathsf{LOGSPACE} = \mathsf{SPACE}(\log n)$ (logarithmic space)
- $\mathsf{P} = \mathsf{TIME}(n^{O(1)})$ (polynomial time)
- $\mathsf{PSPACE} = \mathsf{SPACE}(n^{O(1)})$ (polynomial space)
- $\mathsf{EXP} = \mathsf{TIME}(2^{n^{O(1)}})$ (exponential time)

The most studied is certainly P, the set of problems solvable in polynomial time. It is seen as the class that captures what is doable in practice (arguably, with some debates, is an n^{10} algorithm efficient?).

4.4 Class NP

The class NP contains all decision problems that can be verified in polynomial time. It is not concerned with the time it takes to solve the problem, only the time it takes to verify a solution, provided that one exists. More precisely,

NP: decision problems for which, whenever the answer is YES, there exists a proof of this fact that can be verified in polynomial time.

4.4.1 Examples

- 3-Coloring: Given a graph G, can the vertices of G be colored in red, green, or blue such that all neighbors have different colors?

  No known algorithms exist to answer this question efficiently, so we don't know whether 3-Coloring is in P. On the other hand, if the answer is YES, there exist a proof that can be checked efficiently; namely, the coloring itself, so 3-Coloring is in NP.

- SAT: Given a boolean formula ϕ in conjunctive normal form, for example $\phi = (x_1 \vee \neg x_2 \vee x_4) \wedge (x_2 \vee \neg x_4) \wedge \cdots$, does there exist a boolean assignment for the variables $x_1, x_2, \ldots$ such that ϕ is true? In other words, is the formula satisfiable?

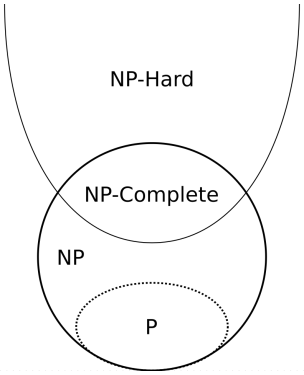
  Again, no efficient algorithm is known for this problem, but if a satisfiable assignment exists, then this assignment can be checked efficiently, so $\mathrm{SAT} \in \mathsf{NP}$.

Of course, all the problems in P are also in NP, for example using the algorithm itself to verify that the answer is YES. However, the converse is not known. Deciding whether $\mathsf{P} \neq \mathsf{NP}$ is a major open question in computer science.

4.5 NP-hard, NP-complete

Some problems are "at least as difficult" as any problem in NP. These are called NP-hard problems. Concretely, a problem Π is NP-hard if for any problem $\Pi' \in \mathsf{NP}$, Π' could be solved efficiently if we were given an efficient algorithm for Π; in other words, there exists a polynomial-time *reduction* from Π' to Π.

Interestingly, there exist NP-hard problems that are themselves in NP. In some sense, these are the hardest problems in NP. A problem that is both NP-hard and in NP is called NP-complete. Here is the general picture, assuming $\mathsf{P} \neq \mathsf{NP}$.



4.6 Examples of graph problems

- Shortest Path (G, u, v, k): Is there a path of length at most k from u to v in G? $\in \mathsf{P}$
- Longest Path (G, u, v, k): Is there a path of length at least k from u to v in G? NP-complete
- Matching (G, k): Is there a set of k edges in G that share no vertex in common? $\in \mathsf{P}$
- Clique (G, k): Does G admit a clique of size k? NP-complete
- Independent Set (G, k): Are there k vertices in G, none of which neighbors? NP-complete
- Dominating Set (G, k): Is there a set of k vertices in G s.t. all nodes are either in the set or have a neighbor in the set? NP-complete
- Vertex Cover (G, k): Are there k vertices that collectively touch every edge? NP-complete
- Coloring (G, k): Can G be properly colored with k colors? $\in \mathsf{P}$ (if $k < 3$), NP-complete (if $k \geq 3$)
- Hamiltonian Cycle (G): Does G admit a simple cycle that visits every vertex? NP-complete
- TSP (G, k): Does G admit a simple cycle of cost $\leq k$ that visits every vertex? NP-complete
- Graph isomorphism (G_1, G_2): Are G_1 and G_2 structurally identical? NP-intermediate?

You'll play with some of these problems in exercises and we'll use them in subsequent classes.

Reduction from 3-Coloring to 4-Coloring:

3-Coloring $\leq$ 4-Coloring:

Given a graph G for which we want to determine whether it is 3-colorable, we construct a graph G' as a copy of G, to which we add a universal vertex (a vertex connected to all others). Since this vertex is connected to all other vertices, it must have a unique color. Thus, G is 3-colorable if and only if G' is 4-colorable and we can use this transformation to solve 3-Coloring efficiently if we are given an efficient algorithm for 4-Coloring.

Assuming we already know that 3-Coloring is NP-hard, this establishes that 4-Coloring is also NP-hard. If we want to show that 4-Coloring is NP-complete, we must also show that it is in NP. This is easy: If a graph is 4-colorable, then there exists a short proof that the answer is YES; namely, the coloring itself. It is indeed easy to verify the coloring by checking that every edge has a different color on both sides and that the total number of colors is ≤ 4.