1. NP, NP-Hard and NP-Complete

These are complexity classes: when a problem is categorized in a complexity class, it means that any algorithm to solve that problem will have the time complexity respect the rules of that class or worse – in other words it is considered that for that problem it is impossible to create an algorithm that has better complexity than described by the complexity class.

In complexity theory there are many complexity classes, but we will only learn about 4 of them, the most relevant complexity classes: P, NP, NP-Complete and NP-Hard. We cannot directly define them without some background in complexity theory.

First, it is important to mention that the problems themselves can be part of different types: decision problem, optimization problem, function problems, search problems etc.

In complexity theory, most study is done for **decision problems** as most problems can be solved through a decision problem one way or another. **Decision problems** are problems for which the result is binary: True/False, Accept/Reject. But don't be misled to think that they are limited, in general a *certificate* is required, basically a proof that the answer is correct, and that certificate might actually be the searched solution.

Depending on the problem the certificate might be simpler or not. For example, the problem: does the graph have an Eulerian circuit (a circuit that uses all edges)? As you recall we can test this by checking the degrees of vertices and the degrees of vertices is a good enough certificate to prove the answer. Does the graph have a Hamiltonian cycle (a cycle which uses all vertices)? On the other hand, for this problem there is no property of the graph that can be easily checked, so a certificate of proof might be just an actual Hamiltonian cycle itself. That means that if we say that the decision problem "Does this graph have a Hamiltonian cycle?" is in a certain complexity class, that actually means that there is no algorithm to *find* a Hamiltonian cycle with better complexity than that complexity class.

So, we will discuss a lot about decision problems, but we will also consider **optimization problems**. These problems involve finding the best solution according to some criteria, such as finding the shortest path in a graph or the maximum flow in a network. Many graph problems are optimization problems, but they can be often changed/solved by decision problems and the complexity classes are similar (e.g. is there a path with this length between these vertices? can be used to solve the shortest path problem).

Before defining the complexity classes, we need to define the notion of a **Turing Machine.**

A Turing machine is, practically, an ideal computer that has an input (like an infinite tape on which a finite number of symbols are written) and produces an output (what modifications it makes on that tape). To understand the tape thing, think about how anything with modern computers has to be transformed in bits – in 0s and 1s. You give your algoithm some part of memory as the input, that part of memory is basically just bits, and if you put them in a line they are a long tape with 0s and 1s. So it is kind of like a Turing machine. The turing machine just has an infinite amount of tape (i.e. memory) and is not restricted to work with only 0s and 1s (it is an ideal computer afterall).

The computer also has certain states while it runs – some finite amount of internal states based on which it makes decisions – the details do not really matter, what matters is that the final

state can be considered the output (e.g., Accept/Reject or yes/no for decision problems). You can consider this like a returned code (from a finite set of available codes) of a function that says how the function ended. Basically, the output of a Turing machine is the changes that it makes on the tape (basically what is saves in memory) and the final state (basically the return code).

The Turing machine is helpful to talk about algorithms in general, without any mentions about hardware specifics. It is an ideal computer and a very simple idea, and this is used in different proofs for complexity theory (and other computer science fields).

From now on when we talk about "**machine**" we refer to a Turing machine.

Definition of 2 complexity classes:

P – **Polynomial Time** – solvable by a deterministic machine in polynomial time.

NP – **Non-Deterministic Polynomial Time** – solvable by a non-deterministic Turing machine in polynomial time.

So, there is more than just the simple Turing machine: There are the deterministic and non-deterministic Turing machines.

Deterministic vs. non-deterministic Turing machines:

- In the **deterministic** case, the machine does one step at a time until it reaches a result.
- The **non-deterministic** machine does multiple steps at a time until it reaches a result. Or, from a different perspective, between multiple steps it chooses the best one each time until it reaches a result – and it chooses magically and luckily always the best step and it cannot be determined how it does that, hence the "non-deterministic" part of the name.

Ok, why do we need imaginary magic machines like the "Non-deterministic Turing machine"?

Well, because they can help us define the complexity classes but, more importantly, they are used to create proofs for these complexity classes (e.g. proofs that specific problems are in a specific complexity class). We will not talk about how they are used in proofs, since everything is weird enough already, but we will use them to define the complexity classes.

To better understand the deterministic/nondeterministic machines, please consider Figure 1. On the left side is presented how a deterministic machine takes the steps to reach a result and on the right side a non-deterministic machine.

The deterministic machine always takes one step, it is basically a list of steps to take.

The non-deterministic machine has more and more options at each step, based on the steps before it. Basically, it has a *tree* of steps to choose from. At one computation it does *all* the steps from one level of the tree at once and then it proceeds to the next tree (or it chooses the best step on each level, depending on your viewpoint). Why this is important is that with a tree representation you can see the difference between finding a solution and verifying if a solution is correct: the number of nodes in the tree is the number of checks needed to find the solution while the number of levels is the number of checks needed to verify the solution (once we find it we can go on the path from the root to the end node).

That “f(n)” you see on the side is the number of steps. “n” is the size of the input (the number of symbols on the tape at input), so the number of steps taken can be defined as a function of the size of the input. Finding an upper bound or lower bound for f(n) will define the complexity of the algorithm!

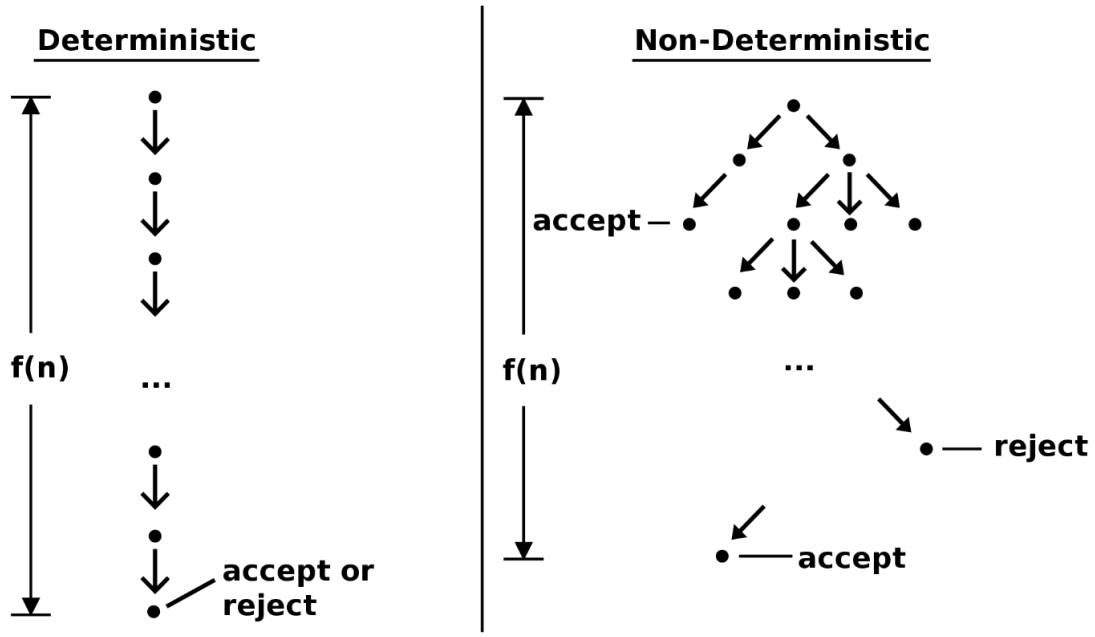

Figure 1

For both of these machines f(n) is the number of steps, but for the non-deterministic machine f(n) is also the number of steps a *deterministic* machine would take to *verify* the solution.

So, again, the definitions:

P – **Polynomial Time** – solvable by a deterministic machine in polynomial time.

NP – **Non-Deterministic Polynomial Time** – solvable by a non-deterministic Turing machine in polynomial time.

So, to clarify: NP is the class where a non-deterministic machine can *find* a solution in polynomial time, but is also the class where a deterministic machine can *verify* the solution in polynomial time.

Also, $P \subset NP$, since a non-determinist machine can surely solve any problem at least in the same number of steps as a deterministic machine.

Ok, what is polynomial time?

We say that an algorithm is in polynomial time if there is a polynomial that is an upper bound for the number of steps (i.e. as n increases, that polynomial increases faster than f(n)).

In other words, an algorithm is **polynomial time** if $f(n) \in O(n^x)$ for some constant x.

For example $O(1)$, $O(n)$, $O(n^2)$, $O(n^3)$ are all polynomial times. $O(\log(n))$ is also polynomial time. But $\log(n)$ is not a polynomial, so why? Because $O(\log(n)) < O(n)$ so it can be bounded by a polynomial => it is polynomial time.

In the same ideea O(n*log(n)) is a polynomial and any combination thereof. With 2 variables, n and k, O(n + k) is polynomial time, as well as O(n*k) – and these with any n and/or k risen to any powers, like $O(n + k^2)$, $O(n^3k^4)$ etc. (and this is true for any number of variables and similar combinations). On the other hand, $O(n^k)$ is **not** polynomial, since k is not a constant. $O(n^k)$ is exponential with respect to k.

To define the other 2 complexity classes we need another notion:

Problem A **reduces** to problem B if B can be used to solve A.

More precisely, problem A **reduces** to problem B if there is a way of solving A as follows:

1. the input for A is transformed, through a polynomial-time algorithm, into a valid input for B;
2. a solution for B is applied;
3. the output from B is transformed, through a polynomial-time algorithm, into an output for A;
4. the result from the above sequence is the correct answer of the original problem A.

An **NP-hard** problem is a problem such that all NP problems reduce to it (in polynomial time). Therefore, they are a class of problems which are at least as hard as the hardest problems in NP. Problems that are NP-hard do not have to be elements of NP; indeed, they may not even be decidable. (they are called NP-hard because they are as hard as any NP, not because they are NP).

An **NP-complete** problem is a problem that is both NP and NP-hard. That is NP-Complete are the NP-Hard problems for which a deterministic machine can *verify* a solution in polynomial time.

So, NP-Hard and NP-Complete cannot have an algorithm that solves them in polynomial complexity (i.e. an $O(n^x)$, for any x constant). For example, $O(2^n)$ cannot be bounded by any $O(n^x)$ for whatever x.

A solution for a NP-Complete must be verifiable in polynomial time, while no such requirement exists for NP-hard problems.

Example: finding an independent set with size k **is** an NP-Complete problem.

3SAT problem

Any decision problem can be reduced to a propositional satisfiability problem (**SAT**), where a propositional expression (e.g. $(x_1 \wedge x_2) \vee (x_2 \wedge \neg x_3)$) is given and we must find values x1, . . . , x_n that make it true. This can be proven that is NP-Complete using Turing Machines.

Any propositional formula can be written in Conjunctive Normal Form (CNF):

$$C_1 \wedge C_2 \wedge ... \wedge C_n$$

Where C_i, called clauses:

$$C_i = a_1^i \vee a_2^i \vee ... \vee a_k^i$$

3-SAT is a special case of SAT where the clauses are limited to 3 terms. (i.e. a_1, a_2 and a_3). It can be proven that any SAT problem can be rewritten as a 3-SAT problem (i.e. STA reduces to 3-SAT).

Example of reducing one problem to another

A problem is NP-Hard if we can reduce any NP problem to it. To prove that a problem A is NP-Hard, we can just prove that problem B reduces to problem A, if we *already know* that problem B is NP-Hard (since all reduce to B and B to A that means all reduce to A – we can reduce to B then to A).

For example, we can prove that finding an independent set of size k is NP-Hard by reducing 3-SAT problem to it.

Reminder: An **independent set** is a set of vertices in a graph, where no 2 vertices are adjacent.

Let's consider the following proposition as an example:

$$\varphi = (x_1 \vee \overline{x}_3 \vee x_4) \wedge (\overline{x}_2 \vee x_3 \vee \overline{x}_4) \wedge (\overline{x}_1 \vee x_2 \vee x_3)$$

To reduce it to independent sets problem we have to model it to a graph. To solve this we need to decide if variables x_1, x_2, x_3 and x_4 are true or false. We notice that we have to set one literal (symbol representing a variable or its negation) in each clause to true.

So we can trnsform it to a graph like this: all literals in the propositions are a vertex (including duplicates and negations, for example x_3 appears both in clause 2 and clause 3 so we will have 2 vertixes x_3). We group these vertices by their clause. Choosing 1 vertex from each group will solve the problem: these literals should be true, the others false. Now, if we link by an edge any to literals which should **not** be true at the same time, then finding an independent set of size 3 will select the 3 literals which should be true, therefore solving the problem.

So, what literals should not be true at the same time? Well, we need only one in each clause, so 2 literals in the same clause should not be in the same set. Also, 2 literals representing a variable and its negation (e.g. x_1 and $\overline{x}_1$ cannot be true at the same time, as x_1 cannot be both true and false at the same time).

Sor our example the graph looks like the one in Figure 2 ($\overline{x}$ was replaced with !x because of technical reasons):

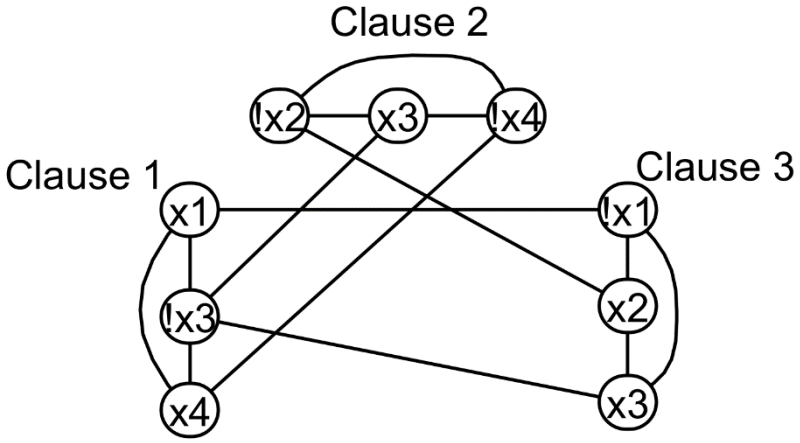

Figure 2

And the independent set {C1/x_4, C2/x_3, C3/x_2) gives us a solution to the problem: x_1 should be false while x_2, x_3 and x_4 should be true.

So, if the problem "find an independent set of size k" can be use to solve a 3-SAT problem with k clauses, then that means the independent set problem must be as hard as 3–SAT – therefore the independent set problem is in NP-Hard.

Checking if a set of k vertices is an independent set can be done in polynomial time ($O(k^2)$, just check if all pairs of vertices are adjacent or not), that means it is verifiable in polynomial time. So "find an independent set of size k" is an NP-Complete problem.

The optimization problem "Find an independent set of maximum size" is also NP-Hard, but is not NP – given an subset of vertices, we cannot check in polynomial time if it is indeed the *maximum* independent set.

Since the independent set problem is NP-Complete, that means that if anyone can find a polynomial time algorithm ($O(n^x$ +/* $k^y)$ where x,y are constants) then they proved that ***any*** NP problem can be done in polynomial time, therefore proving that NP = P (and they would also win 1 million $ as proving that NP=P is a Millenium prize problem, while also probably breaking cryptography and some other consequences).

2. NP-Complete problems in graph theory

Hamiltonian cycle

Reminder:

- A **trail** is a walk with no repeated edges.
- A *trail* that starts and ends at the same vertex is called a **circuit**.
- A **path** is a walk with no repeated vertices (except the initial and terminal vertices).
- A *path* that starts and ends at the same vertex is called a **cycle**.

A **Hamiltonian *path*** in a graph is a cycle that goes once through each vertex of the graph. A **Hamiltonian *cycle*** is a Hamiltonian path that starts and ends in the same vertex. A graph with such a cycle is called a **Hamiltonian graph**.

Consider the graph in Figure 3 for example. A Hamiltonian cycle for that graph might be: (6,5,4,3,1,2,9,10,8,7,6).

The Hamiltonian decision problem: is a given graph a Hamiltonian Graph? I.e. does the graph have a Hamiltonian cycle? This problem is proven to be NP-Complete, 3-SAT can be reduced to it.

Note: this is both in a directed graph and in an undirected graph.

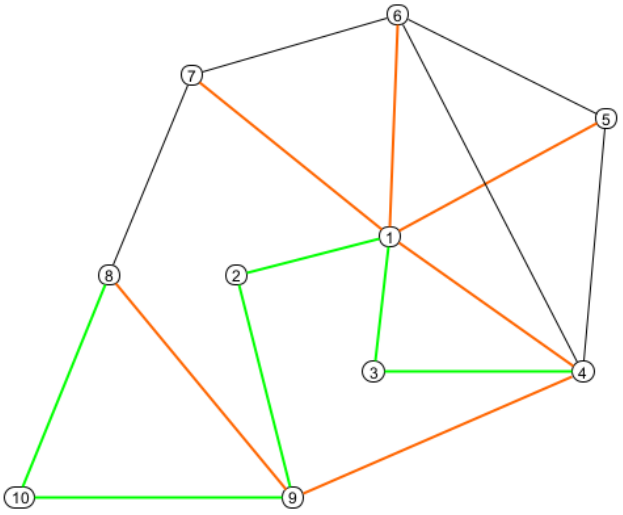

Figure 3

How to solve the Hamiltonian cycle problem?

Brute force: check all paths that might exist. If we consider that a path is a list of vertices, then all possible paths are just all the possible permutations. Checking all possible permutations is in O(n!).

O(n!) is the default upper bound for a generic algorithm that search for a Hamiltonian cycle, since, in worst case, all of them can reach n!. But it is possible to implement much better algorithms than brute force, even if they still are O(n!).

The best Hamiltonian algorithms are backtracking algorithms, with different rules and conditions, to prune and reduce the searching space. The obvious rule is, if we found 2 vertices that are not adjacent, ignore all further permutations that use them as consecutive elements (since it would not be a path).

Other optimization example: before starting the algorithm, decide which are forced edges (edges that must be in the Hamiltonian path – edges incident to vertices with degree 2, since the path must enter by some edge and exit by another, so vertices that connect to only 2 edges means that those edges must be used) and remove unneeded edges (not forced edges incident to a vertex that is incident to 2 forced edges – if a vertex is incident to 2 forced edges it means that those edges must be used to enter and to exit the vertex, therefore all the other edges should not be used). For example, in Figure 3, forced edges are in green and unneeded edges are in orange.

Traveling Salesman Problem (TSP) – This problem is about Hamiltonian cycles in weighted graphs.

Optimization problem: find a Hamiltonian cycle with minimum cost.

To make it a decision problem it can be phrased as a yes/no problem by putting an upper limit on the cost: given a weighted graph, and an integer k, is there a Hamiltonian cycle of cost at most equal to k?

To prove that the traveling Salesman problem is NP-Complete (the decision version, the optimization is NP-Hard) we can show trivially that the Hamiltonian cycle problem reduces to TSP: simply put a cost of 1 on all edges, then TSP just needs to find a Hamiltonian cycle.

Therefore, TSP is also NP-Hard. The decision problem version is NP-Complete, but the optimization version (find the minimum cost cycle) is NP-Hard (cannot check if a Hamiltonian cycle is the one with least cost in polynomial time).

It is obvious that the TSP problem is even harder than the Hamiltonian since we need to find the minimum Hamiltonian cycle, so instead of finding *a* Hamiltonian cycle, we need to find *all* of them and take the minimum. The upper bound is still the same O(n!).

Minimum Cost Path, with negative cycles – the minimum cost path, with *negative cycles* is NP-Hard.

Indeed, the Hamiltonian *path* problem can easily reduce to it (the Hamiltonian path problem – "is there a Hamiltonian path between these 2 vertices?" – is also NP-Complete, 3-SAT and Hamiltonian cycle can reduce to it).

Just put all edges at -1, then the shortest path **must** go through all the vertices to minimize the cost (more vertices mean smaller cost, if edges are at -1). Therefore, if the minimum cost path is -(n-1) then that is also a Hamiltonian path (since we added n-1 edges of cost -1, and there are n-1 edges in a Hamiltonian path).

Solving this problem relies on finding all arrangements of vertices that start with s and end with t. Backtracking is also the technique of choice for more efficient algorithms, with the same obvious rule as Hamiltonian: if 2 vertices are not adjacent remove the arrangements where they are consecutive from the search space.