23 Easy and Hard Problems

The theory of NP-completeness is an attempt to draw a line between tractable and intractable problems. The most important question is whether there is indeed a difference between the two, and this question is still unanswered. Typical results are therefore relative statements such as "if problem B has a polynomial-time algorithm then so does problem C" and its equivalent contra-positive "if problem C has no polynomial-time algorithm then neither has problem B". The second formulation suggests we remember hard problems C and for a new problem B we first see whether we can prove the implication. If we can then we may not want to even try to solve problem B efficiently. A good deal of formalism is necessary for a proper description of results of this kind, of which we will introduce only a modest amount.

What is a problem? An *abstract decision problem* is a function $I \to \{0,1\}$, where I is the set of problem instances and 0 and 1 are interpreted to mean FALSE and TRUE, as usual. To completely formalize the notion, we encode the problem instances in strings of zeros and ones: $I \to \{0,1\}^*$. A *concrete decision problem* is then a function $Q : \{0,1\}^* \to \{0,1\}$. Following the usual convention, we map bit-strings that do not correspond to meaningful problem instances to 0.

As an example consider the shortest-path problem. A problem instance is a graph and a pair of vertices, u and v, in the graph. A solution is a shortest path from u and v, or the length of such a path. The decision problem version specifies an integer k and asks whether or not there exists a path from u to v whose length is at most k. The theory of NP-completeness really only deals with decision problems. Although this is a loss of generality, the loss is not dramatic. For example, given an algorithm for the decision version of the shortest-path problem, we can determine the length of the shortest path by repeated decisions for different values of k. Decision problems are always easier (or at least not harder) than the corresponding optimization problems. So in order to prove that an optimization problem is hard it suffices to prove that the corresponding decision problem is hard.

Polynomial time. An algorithm *solves* a concrete decision problem Q *in time* $T(n)$ if for every instance $x \in \{0,1\}^*$ of length n the algorithm produces $Q(x)$ in time at most $T(n)$. Note that this is the worst-case notion of time-complexity. The problem Q is *polynomial-time solvable* if $T(n) = \mathrm{O}(n^k)$ for some constant k independent of n. The first important complexity class of problems is

$$\mathsf{P} = \text{set of concrete decision problems that are polynomial-time solvable.}$$

The problems $Q \in \mathsf{P}$ are called *tractable* or *easy* and the problems $Q \notin \mathsf{P}$ are called *intractable* or *hard*. Algorithms that take only polynomial time are called *efficient* and algorithms that require more than polynomial time are *inefficient*. In other words, until now in this course we only talked about efficient algorithms and about easy problems. This terminology is adapted because the rather fine grained classification of algorithms by complexity we practiced until now is not very useful in gaining insights into the rather coarse distinction between polynomial and non-polynomial.

It is convenient to recast the scenario in a formal language framework. A *language* is a set $L \subseteq \{0,1\}^*$. We can think of it as the set of problem instances, x, that have an affirmative answer, $Q(x) = 1$. An algorithm $A : \{0,1\}^* \to \{0,1\}$ *accepts* $x \in \{0,1\}^*$ if $A(x) = 1$ and it *rejects* x if $A(x) = 0$. The language *accepted* by A is the set of strings $x \in \{0,1\}^*$ with $A(x) = 1$. There is a subtle difference between accepting and *deciding* a language L. The latter means that A accepts every $x \in L$ and rejects every $x \notin L$. For example, there is an algorithm that accepts every program that halts, but there is no algorithm that decides the language of such programs. Within the formal language framework we redefine the class of polynomial-time solvable problems as

$$\begin{aligned}\mathsf{P} &= \{L \subseteq \{0,1\}^* \mid L \text{ is accepted by a polynomial-time algorithm}\} \\ &= \{L \subseteq \{0,1\}^* \mid L \text{ is decided by a polynomial-time algorithm}\}.\end{aligned}$$

Indeed, a language that can be accepted in polynomial time can also be decided in polynomial time: we keep track of the time and if too much goes by without x being accepted, we turn around and reject x. This is a non-constructive argument since we may not know the constants in the polynomial. However, we know such constants exist which suffices to show that a simulation as sketched exists.

Hamiltonian cycles. We use a specific graph problem to introduce the notion of verifying a solution to a problem, as opposed to solving it. Let $G = (V, E)$ be an undirected graph. A *hamiltonian cycle* contains every vertex

$v \in V$ exactly once. The graph G is *hamiltonian* if it has a hamiltonian cycle. Figure 108 shows a hamiltonian cycle of the edge graph of a Platonic solid. How about the edge graphs of the other four Platonic solids? Define $L =$

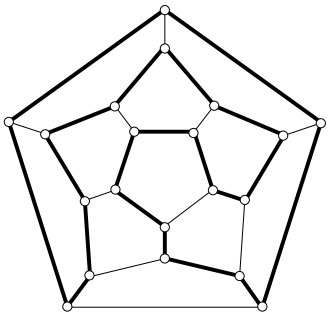

Figure 108: The edge graph of the dodecahedron and one of its hamiltonian cycles.

$\{G \mid G \text{ is hamiltonian}\}$. We can thus ask whether or not $L \in \mathsf{P}$, that is, whether or not there is a polynomial-time algorithm that decides whether or not a graph is hamiltonian. The answer to this question is currently not known, but there is evidence that the answer might be negative. On the other hand, suppose y is a hamiltonian cycle of G. The language $L' = \{(G, y) \mid y \text{ is a hamiltonian cycle of } G\}$ is certainly in P because we just need to make sure that y and G have the same number of vertices and every edge of y is also an edge of G.

Non-deterministic polynomial time. More generally, it seems easier to verify a given solution than to come up with one. In a nutshell, this is what NP-completeness is about, namely finding out whether this is indeed the case and whether the difference between accepting and verifying can be used to separate hard from easy problems.

Call $y \in \{0,1\}^*$ a *certificate*. An algorithm A *verifies* a problem instance $x \in \{0,1\}^*$ if there exists a certificate y with $A(x, y) = 1$. The language *verified* by A is the set of strings $x \in \{0,1\}^*$ verified by A. We now define a new class of problems,

$$\mathsf{NP} \quad = \quad \{L \subseteq \{0,1\}^* \mid L \text{ is verified by a polynomial-time algorithm}\}.$$

More formally, L is in NP if for every problem instance $x \in L$ there is a certificate y whose length is bounded from above by a polynomial in the length of x such that $A(x, y) = 1$ and A runs in polynomial time. For example, deciding whether or not G is hamiltonian is in NP.

The name NP is an abbreviation for **n**on-deterministic **p**olynomial time, because a non-deterministic computer can guess a certificate and then verify that certificate. In a parallel emulation, the computer would generate all possible certificates and then verify them in parallel. Generating one certificate is easy, because it only has polynomial length, but generating all of them is hard, because there are exponentially many strings of polynomial length.

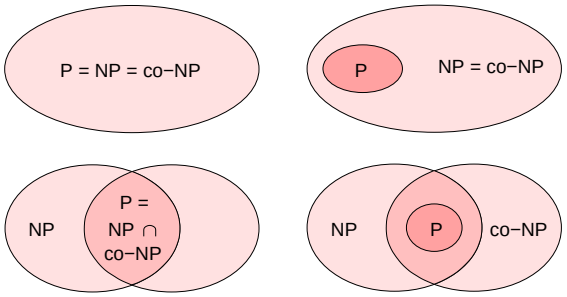

Figure 109: Four possible relations between the complexity classes P, NP, and co-NP.

Non-deterministic machine are at least as powerful as deterministic machines. It follows that every problem in P is also in NP, $\mathsf{P} \subseteq \mathsf{NP}$. Define

$$\text{co-NP} \quad = \quad \{L \mid \overline{L} = \{x \notin L\} \in \mathsf{NP}\},$$

which is the class of languages whose complement can be verified in non-deterministic polynomial time. It is not known whether or not $\mathsf{NP} = \text{co-NP}$. For example, it seems easy to verify that a graph is hamiltonian but it seems hard to verify that a graph is not hamiltonian. We said earlier that if $L \in \mathsf{P}$ then $\overline{L} \in \mathsf{P}$. Therefore, $\mathsf{P} \subseteq \text{co-NP}$. Hence, only the four relationships between the three complexity classes shown in Figure 109 are possible, but at this time we do not know which one is correct.

Problem reduction. We now develop the concept of reducing one problem to another, which is key in the construction of the class of NP-complete problems. The idea is to map or transform an instance of a first problem to an instance of a second problem and to map the solution to the second problem back to a solution to the first problem. For decision problems, the solutions are the same and need no transformation.

Language L_1 is *polynomial-time reducible* to language L_2, denoted $L_1 \leq_P L_2$, if there is a polynomial-time computable function $f : \{0,1\}^* \to \{0,1\}^*$ such that $x \in L_1$ iff $f(x) \in L_2$, for all $x \in \{0,1\}^*$. Now suppose that

L_1 is polynomial-time reducible to L_2 and that L_2 has a polynomial-time algorithm A_2 that decides L_2,

$$x \xrightarrow{f} f(x) \xrightarrow{A_2} \{0,1\}.$$

We can compose the two algorithms and obtain a polynomial-time algorithm $A_1 = A_2 \circ f$ that decides L_1. In other words, we gained an efficient algorithm for L_1 just by reducing it to L_2.

REDUCTION LEMMA. If $L_1 \leq_P L_2$ and $L_2 \in \mathsf{P}$ then $L_1 \in \mathsf{P}$.

In words, if L_1 is polynomial-time reducible to L_2 and L_2 is easy then L_1 is also easy. Conversely, if we know that L_1 is hard then we can conclude that L_2 is also hard. This motivates the following definition. A language $L \subseteq \{0,1\}^*$ is NP-*complete* if

(1) $L \in \mathsf{NP}$;

(2) $L' \leq_P L$, for every $L' \in \mathsf{NP}$.

Since *every* $L' \in \mathsf{NP}$ is polynomial-time reducible to L, *all* L' have to be easy for L to have a chance to be easy. The L' thus only provide evidence that L might indeed be hard. We say L is NP-*hard* if it satisfies (2) but not necessarily (1). The problems that satisfy (1) and (2) form the complexity class

$$\mathsf{NPC} \quad = \quad \{L \mid L \text{ is NP-complete}\}.$$

All these definitions would not mean much if we could not find any problems in NPC. The first step is the most difficult one. Once we have one problem in NPC we can get others using reductions.

Satisfying boolean formulas. Perhaps surprisingly, a first NP-complete problem has been found, namely the problem of satisfiability for logical expressions. A *boolean formula*, φ, consists of variables, $x_1, x_2, \ldots$, operators, $\neg, \wedge, \vee, \Longrightarrow, \ldots$, and parentheses. A *truth assignment* maps each variable to a boolean value, 0 or 1. The truth assignment *satisfies* if the formula evaluates to 1. The formula is *satisfiable* if there exists a satisfying truth assignment. Define SAT $= \{\varphi \mid \varphi \text{ is satisfiable}\}$. As an example consider the formula

$$\psi \quad = \quad (x_1 \Longrightarrow x_2) \Longleftrightarrow (x_2 \vee \neg x_1).$$

If we set $x_1 = x_2 = 1$ we get $(x_1 \Longrightarrow x_2) = 1$, $(x_2 \vee \neg x_1) = 1$ and therefore $\psi = 1$. It follows that $\psi \in$ SAT. In fact, all truth assignments evaluate to 1, which means that ψ is really a tautology. In other words, its negation is not in SAT.

SATISFIABILITY THEOREM. We have SAT $\in$ NP and $L' \leq_P$ SAT for every $L' \in \mathsf{NP}$.

That SAT is in the class NP is easy to prove: just guess an assignment and verify that it satisfies. However, to prove that every $L' \in \mathsf{NP}$ can be reduced to SAT in polynomial time is quite technical and we omit the proof. The main idea is to use the polynomial-time algorithm that verifies L' and to construct a boolean formula from this algorithm. To formalize this idea, we would need a formal model of a computer, a Touring machine, which is beyond the scope of this course.