Introduction, P and NP

A main objective of theoretical computer science is to understand the amount of resources (time, memory, communication, randomness , . . .) needed to solve computational problems that we care about. While the design and analysis of algorithms puts upper bounds on such amounts, computational complexity theory is mostly concerned with lower bounds; that is we look for *negative results* showing that certain problems require a lot of time, memory, etc., to be solved. In particular, we are interested in *infeasible* problems, that is computational problems that require impossibly large resources to be solved, even on instances of moderate size. It is very hard to show that a particular problem is infeasible, and in fact for a lot of interesting problems the question of their feasibility is still open. Another major line of work in complexity is in understanding the relations between different computational problems and between different "modes" of computation. For example what is the relative power of algorithms using randomness and deterministic algorithms, what is the relation between worst-case and average-case complexity, how easier can we make an optimization problem if we only look for approximate solutions, and so on. It is in this direction that we find the most beautiful, and often surprising, known results in complexity theory.

Before going any further, let us be more precise in saying what a computational problem is, and let us define some important classes of computational problems. Then we will see a particular incarnation of the notion of "reduction," the main tool in complexity theory, and we will introduce **NP**-completeness, one of the great success stories of complexity theory. We conclude by demonstrating the use of diagonalization to show some separations between complexity classes. It is unlikely that such techniques will help solving the **P** versus **NP** problem.

1.1 Computational Problems

In a *computational problem*, we are given an *input* that, without loss of generality, we assume to be encoded over the alphabet $\{0, 1\}$, and we want to return in *output* a solution satisfying some property: a computational problem is then described by the property that the output has to satisfy given the input.

In this course we will deal with four types of computational problems: *decision* problems, *search* problems, *optimization* problems, and *counting* problems.[1] For the moment, we will discuss decision and search problem.

In a *decision* problem, given an input $x \in \{0, 1\}^*$, we are required to give a YES/NO answer. That is, in a decision problem we are only asked to verify whether the input satisfies a certain property. An example of decision problem is the 3-coloring problem: given an undirected graph, determine whether there is a way to assign a "color" chosen from $\{1, 2, 3\}$ to each vertex in such a way that no two adjacent vertices have the same color.

A convenient way to *specify* a decision problem is to give the set $L \subseteq \{0, 1\}^*$ of inputs for which the answer is YES. A subset of $\{0, 1\}^*$ is also called a *language*, so, with the previous convention, every decision problem can be specified using a language (and every language specifies a decision problem). For example, if we call 3COL the subset of $\{0, 1\}^*$ containing (descriptions of) 3-colorable graphs, then 3COL is the language that specifies the 3-coloring problem. From now on, we will talk about decision problems and languages interchangeably.

In a *search* problem, given an input $x \in \{0, 1\}^*$ we want to compute some answer $y \in \{0, 1\}^*$ that is in some relation to x, if such a y exists. Thus, a search problem is specified by a relation $R \subseteq \{0, 1\}^* \times \{0, 1\}^*$, where $(x, y) \in R$ if and only if y is an admissible answer given x.

Consider for example the search version of the 3-coloring problem: here given an undirected graph $G = (V, E)$ we want to find, if it exists, a coloring $c : V \to \{1, 2, 3\}$ of the vertices, such that for every $(u, v) \in V$ we have $c(u) \neq c(v)$. This is different (and more demanding) than the decision version, because beyond being asked to determine whether such a c exists, we are also asked to construct it, if it exists. Formally, the 3-coloring problem is specified by the relation R_{3COL} that contains all the pairs (G, c) where G is a 3-colorable graph and c is a valid 3-coloring of G.

1.2 P and NP

In most of this course, we will study the *asymptotic* complexity of problems. Instead of considering, say, the time required to solve 3-coloring on graphs with 10,000 nodes on some particular model of computation, we will ask what is the best asymptotic running time of an algorithm that solves 3-coloring on all instances. In fact, we will be much less ambitious, and we will just ask whether there is a "feasible" asymptotic algorithm for 3-coloring. Here feasible refers more to the rate of growth than to the running time of specific instances of reasonable size.

A standard convention is to call an algorithm "feasible" if it runs in polynomial time, i.e. if there is some polynomial p such that the algorithm runs in time at most $p(n)$ on inputs of length n.

[1]This distinction is useful and natural, but it is also arbitrary: in fact every problem can be seen as a search problem

We denote by **P** the class of decision problems that are solvable in polynomial time.

We say that a search problem defined by a relation R is a **NP** search problem if the relation is efficiently computable and such that solutions, if they exist, are short. Formally, R is an **NP** search problem if there is a polynomial time algorithm that, given x and y, decides whether $(x, y) \in R$, and if there is a polynomial p such that if $(x, y) \in R$ then $|y| \leq p(|x|)$.

We say that a decision problem L is an **NP** decision problem if there is some **NP** relation R such that $x \in L$ if and only if there is a y such that $(x, y) \in R$. Equivalently, a decision problem L is an **NP** decision problem if there is a polynomial time algorithm $V(\cdot, \cdot)$ and a polynomial p such that $x \in L$ if and only if there is a y, $|y| \leq p(|x|)$ such that $V(x, y)$ accepts.

We denote by **NP** the class of **NP** decision problems.

Equivalently, **NP** can be defined as the set of decision problems that are solvable in polynomial time by a non-deterministic Turing machine. Suppose that L is solvable in polynomial time by a non-deterministic Turing machine M: then we can define the relation R such that $(x, t) \in R$ if and only if t is a transcript of an accepting computation of M on input x and it's easy to prove that R is an **NP** relation and that L is in **NP** according to our first definition. Suppose that L is in **NP** according to our first definition and that $R$is the corresponding **NP** relation. Then, on input x, a non-deterministic Turing machine can guess a string y of length less than $p(|x|)$ and then accept if and only if $(x, y) \in R$. Such a machine can be implemented to run in non-deterministic polynomial time and it decides L.

For a function $t : N \to N$, we define by $DTIME(t(n))$ the set of decision problems that are solvable by a deterministic Turing machine within time $t(n)$ on inputs of length n, and by $NTIME(t(n))$ the set of decision problems that are solvable by a non-deterministic Turing machine within time $t(n)$ on inputs of length n. Therefore, $\mathbf{P} = \bigcup_k DTIME(O(n^k))$ and $\mathbf{NP} = \bigcup_k DTIME(O(n^k))$.

1.3 NP-completeness

1.3.1 Reductions

Let A and B be two decision problems. We say that A reduces to B, denoted $A \leq B$, if there is a polynomial time computable function f such that $x \in A$ if and only if $f(x) inB$.

Two immediate observations: if $A \leq B$ and B is in **P**, then also $A \in \mathbf{P}$ (conversely, if $A \leq B$, and $A \notin \mathbf{P}$ then also $B \notin \mathbf{P}$); if $A \leq B$ and $B \leq C$, then also $A \leq C$.

1.3.2 NP-completeness

A decision problem A is **NP**-hard if for every problem $L \in \mathbf{NP}$ we have $L \leq A$. A decision problem A is **NP**-complete if it is **NP**-hard and it belongs to **NP**.

It is a simple observation that if A is **NP**-complete, then A is solvable in polynomial time if and only if $\mathbf{P} = \mathbf{NP}$.

1.3.3 An NP-complete problem

Consider the following decision problem, that we call U : we are given in input (M, x, t, l) where M is a Turing machine, $x \in \{0, 1\}^*$ is a possible input, and t and l are integers encoded in unary[2], and the problem is to determine whether there is a $y \in \{0, 1\}^*$, $|y| \leq l$, such that $M(x, y)$ accepts in $\leq t$ steps.

It is immediate to see that U is in **NP**. One can define a procedure V_U that on input (M, x, t, l) and y accepts if and only if $|y| \leq l$, and $M(x, y)$ accepts in at most t steps.

Let L be an **NP** decision problem. Then there are algorithm V_L, and polynomials T_L and p_L, such that $x \in L$ if and only if there is y, $|y| \leq p_L(|x|)$ such that $V_L(x, y)$ accepts; furthermore V_L runs in time at most $T_L(|x| + |y|)$. We give a reduction from L to U. The reduction maps x into the instance $f(x) = (V_L, x, T_L(|x| + p_L(|x|)), p_L(|x|))$. Just by applying the definitions, we can see that $x \in L$ if and only $f(x) \in U$.

1.3.4 The Problem SAT

In SAT (that stands for *CNF-satisfiability*) we are given Boolean variables $x_1, x_2, \ldots, x_n$ and a Boolean formula φ involving such variables; the formula is given in a particular format called *conjunctive normal form*, that we will explain in a moment. The question is whether there is a way to assign Boolean (True / False) values to the variables so that the formulais satisfied.

To complete the description of the problem we need to explain what is a Boolean formula in conjunctive normal form. First of all, Boolean formulas are constructed starting from variables and applying the operators $\vee$ (that stands for OR), $\wedge$ (that stands for AND) and

$\neg$ (that stands for NOT).

The operators work in the way that one expects: $\neg x$ is True if and only if x is False; $x \wedge y$ is True if and only if both x and y are True; $x \vee y$ is True if and only at least one of x or y is True.

So, for example, the expression $\neg x \wedge (x \vee y)$ can be satisfied by setting x to False and y to True, while the expression $x \wedge (\neg x \vee y) \wedge \neg y$ is impossible to satisfy.

A *literal* is a variable or the negation of a variable, so for example $\neg x_7$ is a literal and so is x_3. A *clause* is formed by taking one or more literals and connecting them with a OR, so for example $(x_2 \vee \neg x_4 \vee x_5)$ is a clause, and so is (x_3). A *formula in conjunctive normal form* is the AND of clauses. For example

$$(x_3 \vee \neg x_4) \wedge (x_1) \wedge (\neg x_3 \vee x_2)$$

is a formula in conjunctive normal form (from now on, we will just say "CNF formula" or "formula"). Note that the above formula is satisfiable, and, for example, it is satisfied by setting all the variables to True (there are also other possible assignments of values to the variables that would satisfy the formula).

On the other hand, the formula

$$x \wedge (\neg x \vee y) \wedge \neg y$$

is not satisfiable, as it has already been observed.