The P-vs-NP problem

Andrés E. Caicedo

September 10, 2011

This note is based on lecture notes for the Caltech course Math 6c, prepared with A. Kechris and M. Shulman.

1 Decision problems

Consider a finite alphabet A, and "words" on that alphabet (the "alphabet" may consist of digits, of abstract symbols, of actual letters, etc).

We use the notation A^* to indicate the set of all "words" from the alphabet A. Here, a word is simply a finite sequence of symbols from A. For example, if A is the usual alphabet, then

$$awwweeeedddfDDkH$$

would be a word.

We are also given a set V of words, and we say that the words in V are *valid.* (V may be infinite.)

In the *decision problem* associated to V, we are given as **input** a word in this alphabet. As **output** we say yes or no, depending on whether the word is in V or not (i.e., whether it is "valid").

We are interested in whether there is an *algorithm* that allows us to decide the right answer.

For example:

1. V may be the set of *prime numbers.* As input we are given a positive integer n. An algorithm to see whether n is prime consists on dividing n by $2, 3, \dots, n-1$ and checking whether the answer is ever exact. If it is (i.e., if we find some number k that divides n), then n is not prime, and we return No. If we do not find such number k, then n is prime and we return Yes.

2. V may be the set of propositional tautologies. As input we are given a propositional formula A. An algorithm to see if A is a tautology consists in writing down the truth table for A and seeing if we always get truth value T. If so, we return Yes.

3. The "alphabet" could consists of the names for American cities. V may be a set of pairs $city_1city_2$ for which we can travel by plane from $city_1$ to $city_2$ spending under \$300. Websites like Expedia try to implement algorithms that allow us to solve the associated decision problem.

4. Using the usual alphabet together with the usual keyboard symbols, let V be the set of computer programs in language C^{++} that eventually stop if executed with input 1. In this case there is no algorithm to see whether a given computer program is in V. In computer science this is called the *halting problem*.

5. If V is the set of mathematical statements that are true, there is no algorithm either. This means that there is no mechanical way of checking, given a mathematical statement S, whether it holds. (S could be "$3 > 5$" or Fermat's last theorem or "There are integers x and y such that $x^2 - 17y^2 = -1$" or ...)

   Of course, one way of verifying the truth of S could be to find a proof of S or a proof of $\neg S$. One could simply have a program that writes down all possible proofs, and if it ever writes down a proof of either S or $\neg S$, then we stop and have our answer.

   However, this method does not work in general. A remarkable result of Gödel from 1930 says that, *no matter what axioms we begin with*, there are mathematical truths that can neither be proved nor disproved. This is his famous *Incompleteness theorem*.

If an algorithm exists for the decision problem corresponding to a set V, we say that V is *decidable*. Otherwise we call it *undecidable*.

Thus, modulo our rather vague definition of "algorithm," we have the first main classification of (decision) problems as decidable or undecidable. Our next goal will be to classify further the decidable problems according to the (time) complexity of their algorithms and distinguish between those for which an efficient (feasible) algorithm is possible, *the tractable problems*, and those for which it is not, the *intractable* problems.

There is a mathematical way of making precise the notion of algorithm, using "Turing machines." We will not need the precise definition in what follows. For information on Alan Turing, please visit

http://www.mathcomp.leeds.ac.uk/turing2012/

Other approaches are possible, and go by the common name of "models of computation." It is a remarkable fact that the very different models that have been considered are all actually equivalent.

2 The Class $\mathcal{P}$

The study of efficiency of algorithms is also called *computational complexity*. We will concentrate here on time complexity (how long it takes to solve the problem), but one can also discuss, for example, space complexity (how much "memory" storage is used in solving it).

A natural measure of time complexity for us is the number of steps an algorithm requires to solve a given problem.

Since different problems require different amounts of input, in order to compare the complexities of different algorithms and problems, we must measure complexity as a function of the input size. In addition, with computer speeds increasing rapidly, small instances of a problem can usually be solved no matter how difficult the problem is. For this reason, and also to eliminate the effect of "overhead" time, we will consider only the *asymptotic* complexity as the size of the input increases. First we recall the "big-O notation," which gives a rigorous way of comparing asymptotic growth rates.

Definition 1 *Given two functions $f, g : \mathbb{N} \to \mathbb{N}$ we define*

$$f = O(g) \textit{ iff } \exists n_0\, \exists C\, \forall n \geq n_0 (f(n) \leq Cg(n)).$$

For example, if $p(n)$ is a polynomial, then $p(n) = O(n^d)$ for large enough d.

Definition 2 *Given any $T : \mathbb{N} \to \mathbb{N}$, we let* TIME$(T)$ *consist of all decision problems V which can be decided by an algorithm in time t for some $t = O(T)$.*

More precisely, a decision problem V is in TIME(T) if there is $t = O(T)$ and an algorithm M on some alphabet $B \supseteq A \cup \{Y, N\}$ such that for $w \in A^*$:

1. $w \in P \Longrightarrow$ on input w, M halts in $\leq t(|w|)$ steps with output Y.
2. $w \notin P \Longrightarrow$ on input w, M halts in $\leq t(|w|)$ steps with output N

(here $|w|$ = length of the word w).

It is important to remark here that, even though it is standard convention, the notation $f = O(g)$ is somewhat imprecise, because there are many different functions f, all of which are $O(g)$ for the same g, so saying $f = O(g)$ and $h = O(g)$ does not mean that $f = h$, the $=$ symbol in "$f = O(g)$" is just a convention, and should not be confused with any kind of actual equality.

What sort of growth rate should we require of an algorithm to consider it manageable? After all, we must expect the time to increase somewhat with the input size. It turns out that the major distinctions in complexity are between "polynomial time" algorithms and faster-growing ones, such as exponentials.

Definition 3 *A decision problem is in the class $\mathcal{P}$ (or it is polynomially decidable) if it is* TIME(n^d) *for some $d \geq 0$, i.e. it can be decided in polynomial time.*

Problems in the class $\mathcal{P}$ are considered *tractable* (efficiently decidable) and the others *intractable*. It is clear that the class $\mathcal{P}$ provides an upper limit for problems that can be algorithmically solved in realistic terms. If a problem is in $\mathcal{P}$, however, it does not necessarily mean that an algorithm for it can be practically implemented, for example, it could have time complexity of the order of $n^{1,000,000}$ or of the order n^3 but with enormous coefficients. However, most natural problems that have been shown to be polynomially decidable have been found to have efficient (e.g., very low degree) algorithms. Moreover, the class of problems in $\mathcal{P}$ behaves well mathematically, and is independent of the model of computation, since any two formal models of computation can be mutually simulated within polynomial time.

It is important to remark that time complexity, as explained here, is a worst case analysis. If a problem P is intractable, then there is no polynomial time algorithm which for all n and all inputs of length n will decide P. But one can still search for approximate algorithms that work well on the average or for most practical (e.g., small) instances of the problem or give the correct answer with high probability, etc.

Example 1 *LINEAR PROGRAMMING is the decision problem given by:*

An input is an integer matrix (v_{ij}) for $1 \leq i \leq m$ and $1 \leq j \leq n$, along with integer vectors $D = (d_i)_{i=1}^m$ and $C = (c_j)_{j=1}^n$, and an integer B.

The question we want to decide is whether there a rational vector $X = (x_j)_{j=1}^n$ such that $\sum_{j=1}^n v_{ij}x_j \leq d_i$, for $1 \leq i \leq m$, and $\sum_{j=1}^n c_jx_j \geq B$?

LINEAR PROGRAMMING turns out to be in $\mathcal{P}$. This is a difficult result. It was proved by Khachian, in 1979.

3 The Class $\mathcal{N}P$ and the $\mathcal{P} = \mathcal{N}P$ Problem

Although a large class of problems have been classified as tractable or intractable, there is a vast collection of decision problems, many of them of great practical importance, that are widely assumed to be intractable but no one until now has been able to demonstrate it. These are the so-called *$\mathcal{N}P$-complete problems*. In order to introduce these problems, we must first define the class $\mathcal{N}P$ of *non-deterministic polynomial* decision problems.

Definition 4 *Let V be a decision problem. We say that V is in the class $\mathcal{N}P$ if there is an algorithm M on an alphabet $B \supseteq A \cup \{Y, N\}$ and a polynomial $p(n)$ such that for any $w \in A^*$:*

$$w \in P \Longleftrightarrow \begin{array}{l} \exists v \in B^* \textit{ such that } |v| \leq p(|w|) \textit{ and on input } w * v, M \textit{ stops} \\ \textit{with output } Y \textit{ after at most } p(|w|) \textit{ many steps.} \end{array}$$

In other words, $w \in P$ iff there is a "guess" v, of length bounded by a polynomial in the length of w, such that w together with v pass a polynomial acceptance test. (This can be also viewed as a non-deterministic polynomial time algorithm.)

Example 2 *SATISFIABILITY, the decision problem of whether a propositional sentence is satisfiable, is in $\mathcal{N}P$, since if we can guess a truth assignment, we can verify that it satisfies the given set of clauses in polynomial time.*

Example 3 *TRAVELING SALESMAN is the following decision problem: We are given a finite collection of cities $c_1, \ldots, c_n$, together with information about the distance between them, $d(c_i, c_j)$ being the distance between c_i and c_j. We are also given a bound B.*

We want to decide whether here is a way to travel through all the cities and returning to the original point, without actually traveling more than distance B, i.e., whether there is a way to order the cities, say $c_2, c_7, c_5, \ldots, c_3$ so that we have

$$d(c_2, c_7) + d(c_7, c_5) + d(c_5, c_3) + \cdots + d(c_3, c_2) \leq B.$$

It turns out that TRAVELING SALESMAN is also in $\mathcal{NP}$, since if we guess the order of the set of cities, we can calculate whether the length of the tour is $\leq B$ in polynomial time.

In fact a vast number of problems like these, which involve some kind of search, are in $\mathcal{NP}$. Clearly $\mathcal{P} \subseteq \mathcal{NP} \subseteq \bigcup_d \mathrm{TIME}(2^{n^d})$. The most famous problem in theoretical computer science is whether

$$\mathcal{P} = \mathcal{NP},$$

that is, whether any problem with an efficient nondeterministic algorithm also has an efficient deterministic algorithm. This is known, unsurprisingly, as the $\mathcal{P} = \mathcal{NP}$ *Problem.* Recently the Clay Mathematics Institute included the $\mathcal{P} = \mathcal{NP}$ Problem as one of its seven Millenium Prize Problems, offering \$1,000,000 for its solution. The prevailing assumption today is that $\mathcal{P} \neq \mathcal{NP}$.

There are many excellent resources on the web with detailed information on this problem. I highly recommend Scott Aaronson's blog, Shtetl-Optimized,

http://www.scottaaronson.com/blog/

See for example his lectures on "Great Ideas in Theoretical Computer Science."

4 $\mathcal{NP}$-Complete Problems

We can get a better understanding of the $\mathcal{P} = \mathcal{NP}$ problem by discussing the notion of an $\mathcal{NP}$-complete problem. The $\mathcal{NP}$-complete problems form sort of a "core" of the "most difficult" problems in $\mathcal{NP}$. The key notion is the following:

Definition 5 *A (total) function $f : A^* \to B^*$, where A, B are finite alphabets, is* polynomial-time computable *if there is an algorithm M on a finite alphabet $C \supseteq A \cup B$, and a polynomial p, such that for every input $w \in A^*$, M terminates on at most $p(|w|)$ steps with output $f(w)$.*

Unsurprisingly, a *polynomial-time reduction* is a computable reduction which is in addition polynomial-time computable.

Definition 6 *A decision problem Q in some alphabet B is $\mathcal{NP}$-complete if it is in $\mathcal{NP}$, and for every $\mathcal{NP}$ problem P (in some alphabet A) there is a polynomial-time reduction of P to Q. (That is, there is a polynomial-time computable function $f : A^* \to B^*$ such that $w \in P$ if and only if $f(w) \in Q$.)*

An $\mathcal{N}P$-complete problem is in some sense a hardest possible problem in $\mathcal{N}P$. For example, it is clear that if Q is in $\mathcal{P}$ and P can be polynomial-time reduced to Q, then also P is in $\mathcal{P}$. It therefore follows that if *any* $\mathcal{N}P$-complete problem is in $\mathcal{P}$, then $\mathcal{P} = \mathcal{N}P$. Thus the $\mathcal{P} = \mathcal{N}P$ question is equivalent to the question of whether any given $\mathcal{N}P$-complete problem is in $\mathcal{P}$.

It is clear from the definition that all $\mathcal{N}P$-complete problems are "equivalent" in some sense, since each one can be reduced to the other by a polynomial-time computable function. The first example of an $\mathcal{N}P$-complete problem is due to Cook and Levin in 1971.

Theorem 7 (Cook, Levin) SATISFIABILITY *is $\mathcal{N}P$-complete.*

In a sense, this suggests that there may not be any way of solving SATISFIABILITY than by means of the very slow algorithm of writing down truth tables. However, there are polynomial-time algorithms that solve SATISFIABILITY in many (but not all) situations. This is a problem with many practical applications, and therefore "SAT-solving engines" that are "in general" fast are of great interest.

Karp in 1972 has shown that TRAVELING SALESMAN and many other combinatorial problems are $\mathcal{N}P$-complete, and since that time hundreds of others have been discovered in many areas of mathematics and computer science.