P versus NP

Benjamin Kuykendall & Hsin Pei Toh

Columbia Splash

30 March 2018

1 Introduction

1.1 Motivation

Imagine that you and your friend were tasked with solving some problem. One of you has to come up with the solution, while the other checks it to make sure it is correct. Which would you rather do?

Intuitively, we think that verifying a solution to a problem ought to be easier than solving the problem itself. However, it turns out that this is surprisingly hard to prove. We will formalize these notions of "hardness" and "easiness" and show how different problems relate to each other in terms of how hard or how easy they are to solve.

1.2 Time complexity

Any problem can be characterized by the amount of computational resources it takes to solve. Most of the time, we are interested in the amount of time and space it takes. In particular, we measure a problem's "hardness" by the former, also known as its *time complexity*.

Now, you might think that time is a somewhat subjective measure of hardness; after all, a complicated calculation might take us a few minutes to work out by, perhaps seconds on your smart phone, and merely milliseconds on a super-computer. To avoid issues like this, define the following elementary algorithmic steps:

- Arithmetic and logical operations
- Declaring and assigning variables
- Storing and retrieving values to and from memory

Definition 1. An algorithm's *time complexity* is the number of elementary steps it performs.

This notion should be related to the actual run-time of a program you might implement on your computer. Any algorithms we write in this model you could take home and write in your favorite programming language; if you timed your implementation on a stopwatch, the time in seconds would be proportional to the time complexity of the algorithm.

Furthermore, instead of counting the actual number of steps, we define time complexity only with respect to n, the length of the input: specifically, if we write the problem in binary, how many bits long it is. Sometimes we think of the input as a set of numbers, a graph, or even a sentence. But in any of these cases, there is an easy way to convert it to binary. Indeed, everything in your computer is just that: a sequence of bits.

With a clear *model of computation* and using the *complexity measure* of time, we can now define some *complexity classes*; that is, sets of problems with time complexity falling in specified ranges.

1.3 Defining P and NP

From now on, all of the problems we consider are going to be *decision problems*; that is, questions with YES or NO answers. Though computations are also important in computer science, they can always be written in terms of decision problems. For example, if we wanted to compute a number with n bits, this would correspond to n decision problems: "is the ith bit a 1?" for each bit. So without loss of generality we will only work with decision problems.

Now we are ready to define our two main complexity classes.

Definition 2. P is the set of problems that can be solved in *polynomial time*, i.e. $\leq p(n)$ for some polynomial p.

Many problems that you may be familiar with from your math or programming classes are in P. For example, consider the following problem:

> Given a set of n points with distances between each pair of points, is there a path from point A to point B with total distance $\leq k$?

This problem can in fact be solved in $\sim n^2$ time and is therefore in P. So you can think of problems in P as those that are "easy" to solve.

Definition 3. NP is the set of problems whose solutions can be verified in polynomial time.

By "verify", we mean that given some *certificate*, there exists an algorithm, or *verifier*, to solve the problem. That is, given an input x to a problem L, a verifier V satisfies

$$L(x) = \text{YES} \Leftrightarrow \exists c \text{ such that } V(x,c) = \text{YES}.$$

For example, a valid certificate for the previous problem would be a path from point A to point B with total distance $\leq k$. Given such a path, it is possible to compute the total distance, verify that it is indeed $\leq k$, and output YES, in polynomial time. Thus this problem is in NP as well.

1.4 Non-determinism

Now we are going to define a model of computation called a *non-deterministic algorithm.* In addition to the elementary steps above, a non-deterministic algorithm is allowed to "guess" values. For example, we could have a step like "non-deterministically guess the next edge of the path". Then, after guessing and carrying out the rest of our computation, we get to answer YES or NO. Given a non-deterministic algorithm, which may answer differently depending on how its guesses turn out, we say the algorithm as a whole outputs YES if there is a lucky guess, or lucky guesses, it could have made that resulted in a YES; we say it answers NO if every single guess results in a NO answer.

This seemingly magical computational model should feel closely related to NP. Indeed, we may define NP alternatively as follows:

Definition 4. NP is the set of problems that can be solved by a non-deterministic algorithm in polynomial time.

Thus the name—note that NP does not stand for "non-polynomial"!

We can show this is equivalent to the verifier definition given above. This proof requires two directions: showing any non-deterministic algorithm has a verifier, and show any verifier yields a non-deterministic algorithm:

> Given a verifier: use non-determinism to guess a certificate, then run the verifier.
>
> Given a non-deterministic algorithm M: let the certificate represent the series of non-deterministic guesses our algorithm will make. With the guesses in place, M can be completed by a normal polynomial time algorithm.

Hopefully this model helps you understand the name NP, but for the rest of today, we will stick with the more convenient verifier definition.

2 Some problems in NP

Now, let's try to prove that some problems are in NP.

First, take any problem $M \in \mathsf{P}$. Is $M \in \mathsf{NP}$? Yes! For example, if we have an algorithm telling us if there exists a path between point A and point B in polynomial time, then that algorithm serves as the verifier. Thus we've shown the following:

Theorem 5. $\mathsf{P} \subseteq \mathsf{NP}$.

Next, we will look at some problems that may or may not have applications to everyday life and show that they are in NP.

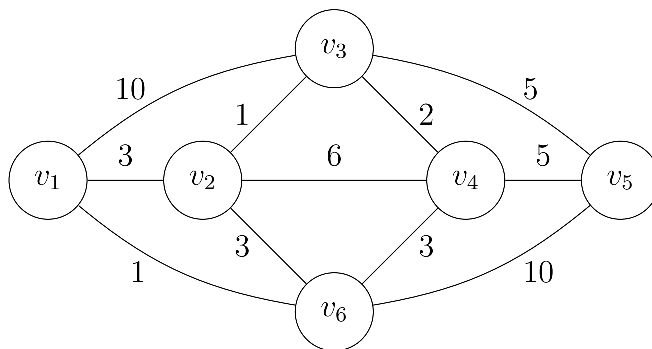
2.1 Knapsack

> We have an assortment of items $i = 1, \ldots, \ell$ each with weight w_i and value v_i. If our bag can hold at most weight W, can we carry items with total value V?

ANSWER: A certificate is a list of items; to verify, we check that their weights sum to at most W and values sum to at least V.

2.2 Graph problems

A *graph* is a set of nodes, some of which have *edges* between them. We say two nodes sharing an edge are *adjacent*. Additionally, some graphs are *weighted*, whereby each edge has an associated value, or *weight*. Here's an example:



Graphs are useful for representing relationships between objects: for example, a set of social media users form a graph, where two parties share an edge whenever they are friends.

There are lots of interesting graph problems in various complexity classes. For example, earlier we considered the problem of finding a path between point A and point B. This is a graph problem, by letting each point be a node, whereby the edge weight between each pair of nodes denotes distance.

2.2.1 Travelling salesman problem (TSP)

> We have a set of ℓ cities, and the distances $\delta(i, j)$ from one city to the next. Can we visit each city exactly once and return to our origin with a total of at most distance D?

ANSWER: A certificate is the order of the cities; to verify, we check $i_1 = i_{\ell+1}$ and distances sum to at most D.

2.2.2 Colorability (3Color)

> Given a graph, can we color each node red, blue, or yellow in a manner such that no adjacent nodes share a color?

ANSWER: A certificate is an assignment of colors; to verify, we check that each node has a color, and that each edge goes between differently-colored nodes.

2.3 Logic problems

A *Boolean formula* is a series of variables v_i and the symbols $\wedge, \vee, \neg$ (meaning "and", "or", "not" respectively). A *truth assignment* sets each v_i to either T or F; we say the assignment *satisfies* the formula if the formula evaluates to true overall: for example, $T \vee (\neg F \wedge T)$ evaluates to true, while $T \wedge F$ evaluates to false.

2.3.1 Satisfiability

Given a Boolean formula, is there a satisfying assignment?

ANSWER: A certificate is an assignment; to verify, we check that the formula with the given assignment evaluates to true.

2.3.2 3-CNF Satisfiability

A very specific format for a formula is a 3-literal conjunctive normal form (or 3-CNF). They are written

$$\phi = (a_1 \vee a_2 \vee a_3) \wedge (b_1 \vee b_2 \vee b_3) \wedge \ldots \wedge (z_1 \vee z_2 \vee z_3)$$

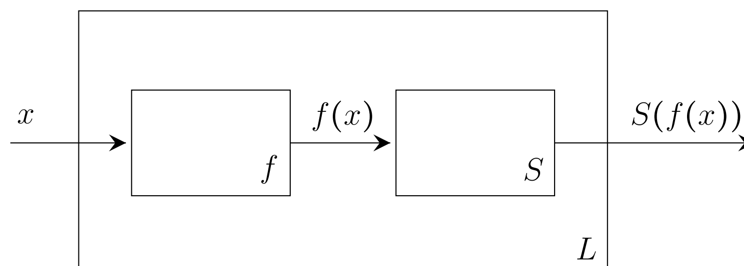
where each literal $k_i = v_i$ or $k_i = \neg v_i$. It turns out any formula can be converted to an equivalent 3-CNF without blowing up the length too much. Thus this problem is equivalent to the normal SAT problem.

3 NP-completeness

Now we will give a notion of the "hardest" problems in NP. To do so, we define a *polynomial-time reduction.* To denote a reduction we use $\leq_p$; we should read $L \leq_p S$ as "L reduces to S" or "L is easier than S". Formally, we say $L \leq_p S$ if there is some f computable in polynomial time such that

$$L(x) = S(f(x)).$$

This means that, given an input for L, we can efficiently transform it into an input for S which gives the same answer, as shown in the following diagram:



Reductions relate to NP in the following sense:

Theorem 6. If $S \in \mathsf{NP}$ and $L \leq_p S$, then $L \in \mathsf{NP}$ as well.

Proof. To prove this statement, given V a verifier for S and f a transformation, we need to construct a verifier W for L. Letting x be the input to W and c a certificate for L, claim that $W(x,c) = V(f(x),c)$ is such a verifier. Indeed, we have

$$
\begin{aligned}
W(x,c) = \text{YES} \Leftrightarrow V(f(x),c) = \text{YES} && \text{by construction} \\
= S(f(x)) && \text{by definition of verifier} \\
= L(x) && \text{by definition of reduction}
\end{aligned}
$$

Finally, since f and V run in polynomial time, W also runs in polynomial time. □

Now we can finally define NP-hardness.

Definition 7 (NP-hardness)**.** A problem S is NP-hard if, for any $L \in \mathsf{NP}$, $L \leq_p S$.

The idea here is that if S is NP-hard, then if we could somehow solve S, then we could solve any NP problem by first transforming it into an instance of S. You can think of an NP-hard problem as one that is at least as hard as any problem in NP.

Furthermore, we have:

Theorem 8. If S is NP-hard and $S \leq_p T$, then T is also NP-hard.

Proof. Since S is NP-hard, for any $L \in \mathsf{NP}$, $L \leq_p S$, i.e. there exists f so that

$$L(x) = S(f(x)).$$

Similarly, $S \leq_p T$, i.e. there exists g so that

$$S(x) = T(g(x)).$$

We can simply compose the two transformations to transform an instance of L to an instance of S to an instance of T:

$$L(x) = S(f(x)) = T(g(f(x))).$$

□

Definition 9 (NP-completeness)**.** A problem S is NP-complete if $S \in \mathsf{NP}$ and S is NP-hard.

You can think of NP-complete problems as the hardest problems in NP.

3.1 Cook-Levin Theorem

Theorem 10 (Cook-Levin Theorem)**.** SAT is NP-hard.

Proof sketch. This is the most important result about NP; unfortunately, the proof is kind of complicated. Basically, it encodes the possible states of the verifier at each time step as a set of Boolean variables. Then we write a formula ϕ over those variables: it hard-codes the value x into one of the inputs, expresses the transitions between states as constraints on which kinds of states can follow each other, and adds constraints to require the algorithm end by answering YES. Then, a satisfying assignment to ϕ will correspond to a certificate c for V and a series of steps the algorithm would go through to reach a YES state. Thus there exists a satisfying assignment for ϕ if and only if for some c, $V(x, c) =$ YES. □

Regardless of how the proof works, we now know that any NP problem can be re-written as a SAT problem. Furthermore, recall that we can actually transform any Boolean formula into an equivalent one in the form of a 3-CNF. Thus 3SAT is NP-hard as well.

3.2 3Color is NP-hard

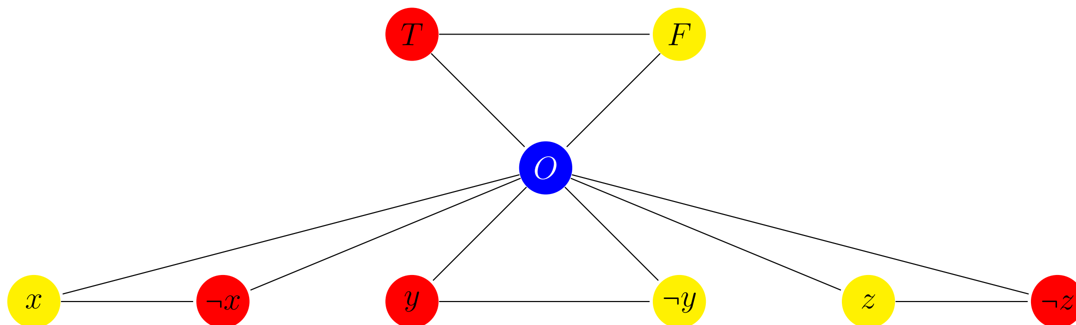
Theorem 11. 3Color is NP-hard.

Proof sketch. Since 3SAT is NP-hard, by **Theorem 7**, it suffices to show 3SAT $\leq_p$ 3Color. Given a 3-CNF formula, say

$$\phi = (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee y \vee \neg z)$$

we construct a graph[1] as follows.

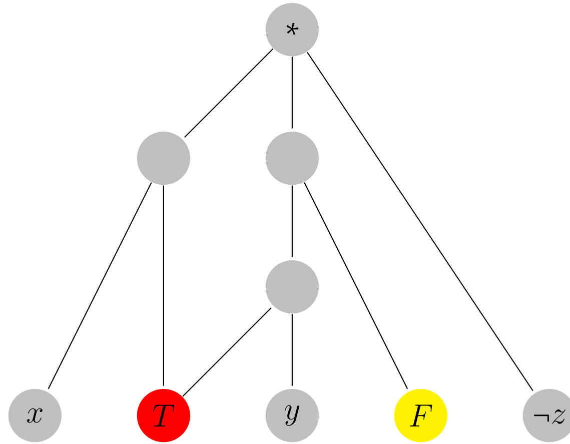
First we consider the variables. The start of our graph is the nodes



The idea here is that T, F, and O will always be different colors (without loss of generality we can illustrate them with red, yellow and blue). Then, each variable must be colored either true or false, and its negation must be colored the opposite.

[1] https://web.stanford.edu/class/archive/cs/cs103/cs103.1132/lectures/27/Small27.pdf

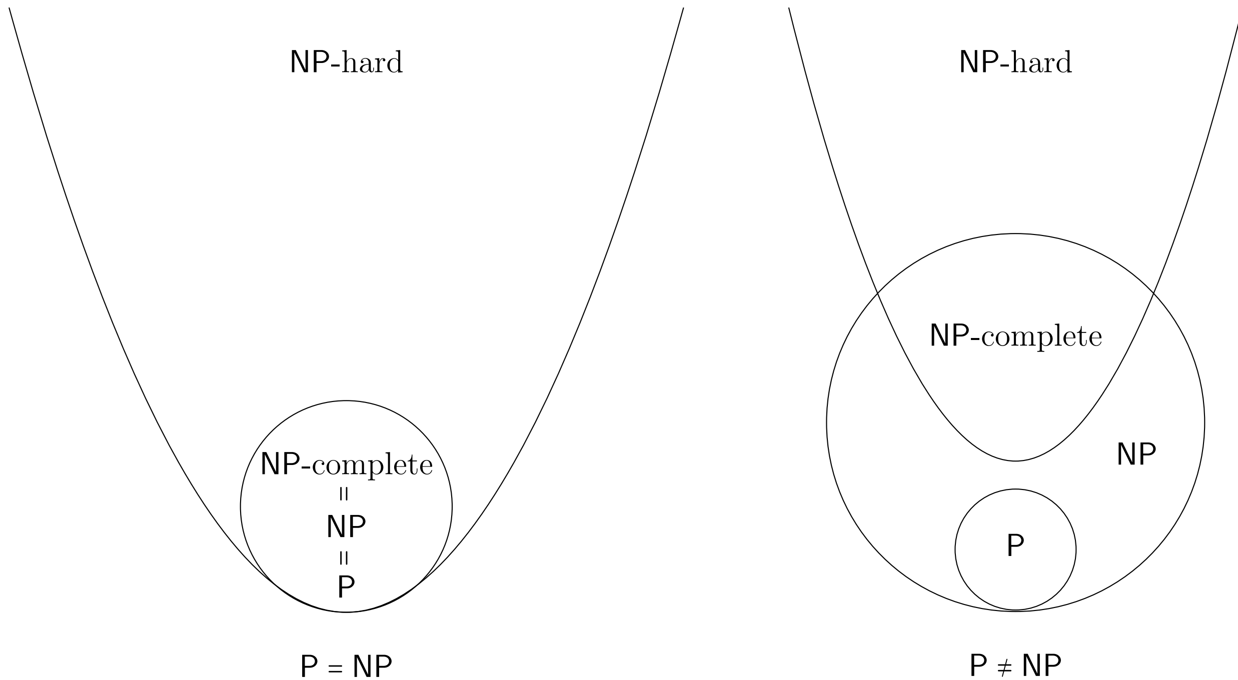
Now for each clause, say $(x \vee y \vee \neg z)$, we construct the following gadget:



The idea here is, there is a valid color for node $*$ if and only if at least one of the variables is true. We can verify this case by case. $\square$

4 Open questions in complexity

Despite everything we know about the structure of NP and NP-hard problems, we still don't know whether $\mathsf{P} \stackrel{?}{=} \mathsf{NP}$. Taking all that we know about P, NP, NP-hard, and NP-complete, the two possible scenarios stand thus:



In practice, we generally assume that $\mathsf{P} \neq \mathsf{NP}$, even if we haven't managed to prove it yet. But what if $\mathsf{P} = \mathsf{NP}$?

- Cryptography: we count on the hardness of certain problems to make encryption hard to invert without possessing the key. In fact, almost all cryptographic protocols use NP-hardness as an asset. If $\mathsf{P} = \mathsf{NP}$, then encryption would be easy to break, signatures would be easy to forge, and cryptocurrencies would be easy to mine.

- Science: some real problems in science are NP-complete, including protein folding. If we could solve these problems quickly, we could design novel proteins to solve all sorts of medical problems.

- Logistics: all sorts of logistics and operations research problems are in NP, including natural problems in scheduling, packing, and routing. If we could solve these quickly, we could save huge amounts of money and labor.

- Mathematics: proofs of theorems are easy to verify (in fact, *automated proof systems* are already in use to check real proofs). But if $\mathsf{P} = \mathsf{NP}$, this would mean proofs would be just as easy to generate; computers would become the best mathematicians!

Now, let's discuss some related problems in complexity.

4.1 P, NP, and EXP

We know that "hard" problems are those that take a long time to solve; but to formalize this, we need to decide just how long is "long". A common choice is defined as follows:

Definition 12. EXP is the set of problems that can be solved in *exponential time*, i.e. $\leq 2^{p(n)}$ for some polynomial p.

From this different characterization of hard problems, we have two important questions:

- How does EXP compare to P? Since EXP is "hard" and P is "easy", is $\mathsf{P} \subsetneq \mathsf{EXP}$?

- How does EXP compare to NP? Since both are "hard" questions, is $\mathsf{EXP} = \mathsf{NP}$?

The first question has a solid answer: yes!

Theorem 13 (Time-hierarchy theorem)**.** $\mathsf{P} \subsetneq \mathsf{EXP}$.

In general, whenever two complexity classes are defined based on run-time, they follow this relationship.

Proof. One direction is easy: if a problem takes at most polynomial time, then certainly it takes at most exponential time, since $p(n) < 2^{p(n)}$. Thus $\mathsf{P} \subseteq \mathsf{EXP}$.

The other direction is more subtle; to show $\mathsf{P} \neq \mathsf{EXP}$, we need to construct a problem in EXP but not in P. The problem ends up being very artificial and kind of meta, but it suffices for the proof, which we do not have enough time for today. □

The second question is up in the air. We know that $\mathsf{NP} \subset \mathsf{EXP}$ because the following algorithm runs in exponential time. For any problem $L \in \mathsf{NP}$ with verifier V, do the following:

> For input x and each choice of certificate c:
>
> If $V(x, c)$ is true, answer YES
>
> If there is no such c, answer NO

Since there are exponentially many choices of c, and V runs in polynomial time, the whole algorithm is exponential time.

In the other direction, the answer depends on whether $\mathsf{P} = \mathsf{NP}$:

- If $\mathsf{P} = \mathsf{NP}$, then using $\mathsf{P} \subsetneq \mathsf{EXP}$, we have $\mathsf{NP} \subsetneq \mathsf{EXP}$.
- If $\mathsf{P} \neq \mathsf{NP}$, then either $\mathsf{NP} = \mathsf{EXP}$ or $\mathsf{NP} \subsetneq \mathsf{EXP}$; we still wouldn't know the answer.

What's the lesson here? It seems like NP is a different type of hardness than EXP, but we don't know for sure, even if we find that $\mathsf{P} \neq \mathsf{NP}$.

4.2 Strong exponential time hypothesis

Another way to compare NP and EXP is to ask if we always need exponential time to solve NP-hard problems. This question is a stronger version of P versus NP called the Strong Exponential Time Hypothesis. It posits that SAT is not only not in P but in particular takes times at least 2^{kn} for some k. A proof of this statement would be very strong evidence that NP-hard problems would be just too slow to solve in practice.

4.3 NP-intermediate problems

An NP-intermediate problem is in NP and not in P but is not NP-hard; the NP-intermediate problems are the "easier" questions in NP. So far, we do not know of any provably NP-intermediate problems. It turns out there is a good reason for this.

Theorem 14 (Ladner's Theorem). NP-intermediate problems exist if and only if $\mathsf{P} \neq \mathsf{NP}$.

Thus one approach to showing $\mathsf{P} \neq \mathsf{NP}$ is to find a provably NP-intermediate problem. There are a lot of "candidate" NP-intermediate problems: we know they are in NP, but we have not been able to show either that they are NP-complete or in P.

Some examples are:

- Factoring: given N, find x, y such that $N = xy$
- Discrete log: given a, b, N find k such that $a^k \equiv b \mod n$
- Graph isomorphism: seeing if two graphs are equal up to re-arranging nodes

These are all very interesting problems, and at least for now, they seem to have "medium" hardness, that is, somewhere between P and NP-complete. This is reflected in their algorithms: these problems all have algorithms with runtimes that are super-polynomial yet sub-exponential.

4.4 Heuristics and Approximations

Just because some mathematician says a problem is too hard doesn't mean we all should give up and go home. In the real world, there are a few approaches to "almost" solve conjectured hard problems. For concreteness, consider solving TSP in the real world (note that in the real world, distances are *metric*, meaning it is shorter to walk from DC to NYC than to walk from DC to Philly to NYC). We may not be able to always find the shortest path, but we can find pretty good ones in both of these senses.

- Heuristics: sometimes the problem is easy with real-world data. The most common approach is a *greedy heuristic*: each time you need to make a decision, just go to the closest unvisited node. There are problem instances where this path is much longer than the optimal, but in practice, it can be decent, and the algorithm is very fast.

- Approximations: if the shortest path is length ℓ, we have algorithms that can find a path of length 1.5ℓ fairly quickly.

Innovations in the field of approximation algorithms make a lot of hard problems easier, as long as we do not need an exact solution all the time.