COMP 314 Class 23: P=NP, and a first look at NP-completeness

1 The complexity classes P and NP

We've now spent a lot of time thinking about the complexity classes Poly and PolyCheck/NPoly. (I'll occasionally use the cumbersome description "PolyCheck/NPoly" to emphasize the fact that PolyCheck and NPoly are exactly the same class.) The computational problems in these complexity classes are mathematical functions that accept an ASCII string as input and produce another ASCII string as output. If we restrict attention to decision problems only—that is, functions that accept an ASCII string as input and produce either "`yes`" or "`no`" as output—we get the world's two most famous complexity classes: P and NP. Formally:

- P is the set of all decision problems in Poly.
- NP is the set of all decision problems in PolyCheck/NPoly.

All of the problems we've discussed so far can be altered in obvious ways to produce decision problems. For example, the decision version of HamiltonianCircuit outputs "`yes`" if a Hamiltonian circuit exists, and "`no`" otherwise. In the rest of the course, we won't usually bother to distinguish between general problems and their decision variants—the context should make it clear whether or not we are restricting to decision problems. In the rare cases where ambiguity could arise, we will use problem names like HamiltonianCircuitGen (for the general problem) and HamiltonianCircuitDec (for the decision variant).

Note that the decision variants of all of our favorite PolyCheck problems are in NP. This includes Factor, TravelingSalesperson, SubsetSum, and HamiltonianCircuit. (Defining a decision version of Factor requires a little more care, as described in Class 20.) For each of these problems, we can verify "`yes`" answers in polynomial time by using the full solution (i.e. the solution to the general, non-decision variant) as a hint.

2 P=NP

Why are P and NP the world's most famous complexity classes? Because they feature in the most famous unsolved mystery in computer science—a mystery known as "P=NP?". A formal statement of this problem is:

P=NP?

(version 1)

Are P and NP the same complexity class? That is, do they each contain the same set of computational problems?

We have already seen that two apparently different complexity classes (PolyCheck and NPoly), defined in completely different ways, turned out to be identical. So even though the definitions of P and NP are different, it's not completely unreasonable to wonder if they are equivalent.

It's obvious that P is a subset of NP. Why? Any decision problem D in P can be solved by some deterministic polytime Python program, say `D.py`. But a deterministic program is just a special case of a nondeterministic one—it's a "multithreaded" program that uses only one thread. So `D.py` can also be regarded as a nondeterministic polytime program that solves D, and hence D is in NP. So P is a subset of NP, and we can therefore reformulate the P=NP question as:

P=NP?
(version 2)
Is there any decision problem D that lies outside P but inside NP?

Note that finding even one problem D that's in NP but not in P would resolve the whole P=NP question. As I write these words, computer scientists have catalogued thousands of problems that are (i) known to be in NP, and (ii) generally believed to be outside P. But not one of these problems has been *proved* to lie outside P.

Yet another way of looking at this is to think about converting nondeterministic programs into deterministic ones. If P=NP, then it must be possible to convert any polytime, nondeterministic program `ND.py` into a polytime, deterministic program `D.py` that computes the same answers. Defining programs to be *equivalent* if they produce the same answers on the same inputs, we can again reformulate the P=NP question:

P=NP?
(version 3)
Is it always possible to convert a polytime, nondeterministic program into an equivalent polytime, deterministic program?

By definition, a polytime nondeterministic program launches a tree of threads like the one in Figure 1. Each leaf of the tree corresponds to a potential solution. The *depth* of the tree is bounded by a polynomial (because every thread finishes in polynomial time). But the *width* of the bottom layer of tree could be exponential (because the number of threads can grow by a constant factor at each layer). The nondeterministic program can check all the leaves (i.e. the potential solutions) in parallel, and return any successful result. Therefore, the notion of converting a nondeterministic program into a deterministic one is roughly equivalent to finding a deterministic method of evaluating all the leaves. It is, of course, possible to do this: we just examine the leaves one by one. But this could take an extraordinarily long time, since the number of leaves can be exponential as a function of the input size. So the real question is, can we check the leaves in polynomial time? This results in our fourth reformulation of P=NP?:

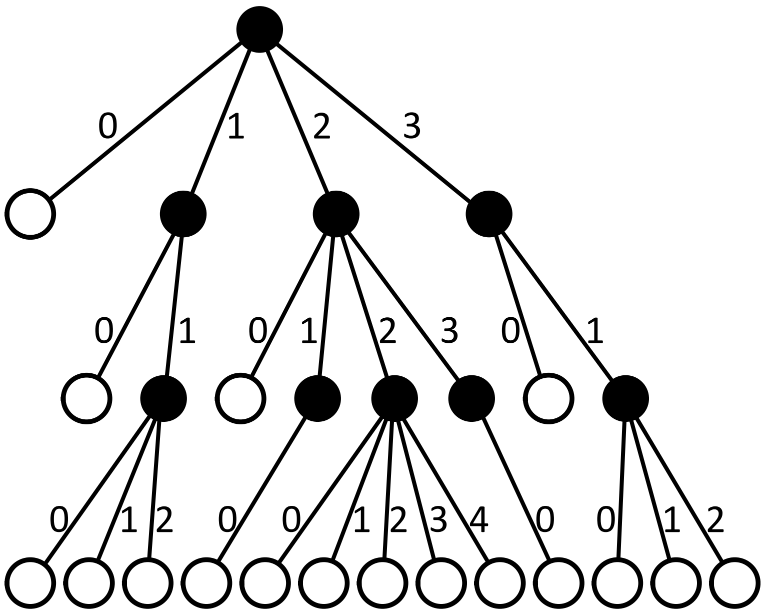

Figure 1: A tree representing the threads launched by a nondeterministic program. Leaves are represented by unfilled circles.

P=NP?
(version 4—informal)
Is it always possible to search the leaves of a tree in polynomial time?

This version of the P=NP question is labeled "informal", because it uses concepts that are not defined rigorously. But it captures the spirit of the question quite nicely. Problems in NP can always be solved by examining exponentially-many leaves in a tree. Problems in P can always be solved by some other, faster technique. A program that actually visits every leaf must take exponential time, and therefore a polytime program *cannot* visit every leaf. Instead, it somehow searches the whole tree by cleverly ignoring the vast majority of possible solutions, and focusing on a tiny, polynomial-sized minority of the possible solutions. This is what we mean by "search the leaves of a tree" in version 4 of the P=NP question above.

3 NP-completeness

Recall that in the previous lecture, we proved that UndirectedHamiltonianCircuit (Uhc) polyreduces to DirectedHamiltonianCircuit (Dhc), and vice versa. In symbols, we have Uhc $\leqslant_{\text{Poly}}$ Dhc and Dhc $\leqslant_{\text{Poly}}$ Chc. We can write this more succinctly as

UHC $\equiv_{\text{Poly}}$ DHC, and say that UHC and DHC are *poly-equivalent.* The formal definition should be obvious:

> Problems P and Q are *poly-equivalent* (written $P \equiv_{\text{Poly}} Q$) if $P \leqslant_{\text{Poly}} Q$ and $Q \leqslant_{\text{Poly}} P$.

Roughly speaking, poly-equivalent problems are "equally hard". More precisely, their solutions have the same running times, if we ignore polynomial factors.

Thus, UHC and DHC either rise together or fall together: either both can be solved in polynomial time, or neither can. In the last lecture, we also saw that CIRCUITSAT $\equiv_{\text{Poly}}$ SAT $\equiv_{\text{Poly}}$ 3-SAT. So these three problems also rise or fall together: either all can be solved in polynomial time, or none can. And in fact, although we have not yet proved it, it turns out that we can do polynomial time reductions between SAT, UHC, and DHC too. So all five problems rise or fall together:

$$\text{CIRCUITSAT} \equiv_{\text{Poly}} \text{SAT} \equiv_{\text{Poly}} \text{3-SAT} \equiv_{\text{Poly}} \text{UHC} \equiv_{\text{Poly}} \text{DHC}$$

Over the last few decades, computer scientists have identified thousands of other problems that are poly-equivalent to these five problems. These problems are called *NP-complete* problems, and the set of all such problems is a complexity class that we'll refer to as NPComplete. (Note: In the regular font, the word "NP-complete" is an adjective, as in the statement "SAT is NP-complete". In a sans-serif font, the word "NPComplete" is a complexity class, as in the statement "SAT is a member of NPComplete".)

The NP-complete problems all rise together, or fall together: either all can be solved in polynomial time, or none can. But NP-complete problems have an even more extraordinary property: the NP-complete problems are the "hardest" problems in NP. That is, given any decision problem D in NP, and any problem C in NPComplete, there's a polynomial time reduction from D to C. We will see a sketch proof of this amazing fact in the next lecture. Until then, let's take it on faith, and write it out in more formal detail:

> **Fact (to be proved in the next lecture):**
> For all $D \in$ NP and $C \in$ NPComplete, we have $D \leqslant_{\text{Poly}} C$.

Actually, the above property is usually used to define NP-completeness in the first place. Here is the classical, formal definition:

> **Definition 1 of NP-complete (the "ridiculously difficult" definition):**
> Let Q be a problem in NP. We say Q is *NP-complete* if, for all D in NP, $D \leqslant_{\text{Poly}} C$.

This definition has a strange disadvantage: the requirement of having to find a polyreduction from any conceivable problem D in NP seems ridiculously difficult, so you might be tempted to think there are no NP-complete problems. Fortunately, back in the 1970s, two computer scientists (Stephen Cook and Leonid Levin) achieved this "ridiculously difficult" goal: they each found a Q to which any other problem in NP can be polyreduced. And once you have a single Q that fits the definition of NP-completeness, you can spread the NP-completeness to other problems by a much easier method.

For example, suppose we have managed to prove, as Stephen Cook did in 1971, that $\textsc{Sat}$ is NP-complete according to Definition 1 above. That is, Cook tells us that

$$\text{For all } D \in \mathsf{NP}, D \leqslant_{\text{Poly}} \textsc{Sat} \tag{1}$$

But we already know that

$$\textsc{Sat} \leqslant_{\text{Poly}} \textsc{CircuitSat}. \tag{2}$$

(We proved this in the previous lecture.) Chaining together equations (3) and (2), we conclude that

$$\text{For all } D \in \mathsf{NP}, D \leqslant_{\text{Poly}} \textsc{CircuitSat}. \tag{3}$$

In other words, $\textsc{CircuitSat}$ also satisfies the ("ridiculously difficult") Definition 1 for NP-completeness.

In general, to show that a problem Q is NP-complete, we just need to find some other problem C that is already known to be NP-complete, and give a polytime reduction from C to Q. This works because we can chain together two equations like (3) and (2), with C and Q in place of $\textsc{Sat}$ and $\textsc{CircuitSat}$.

Hence, because the really hard work was done for us in the early 1970s, we can adopt the following equivalent—but much more convenient—definition of NP-completeness:

> **Definition 2 of NP-complete: (easier)**
> Let Q be a problem in NP. We say Q is *NP-complete* if $\textsc{Sat} \leqslant_{\text{Poly}} Q$.

And there's nothing special about the use of $\textsc{Sat}$ in this definition—$\textsc{Sat}$ just happens to be the problem that Stephen Cook first proved NP-complete. This gives us an even more convenient definition:

> **Definition 3 of NP-complete: (even easier)**
> Let Q be a problem in NP. We say Q is *NP-complete* if $C \leqslant_{\text{Poly}} Q$, for some problem C that is already known to be NP-complete.

Finally, it's obvious that any NP-complete problem (according to any of the above equivalent definitions) can be polyreduced to any other NP-complete problem—this follows from Definition 1. In other words, the NP-complete problems are all poly-equivalent to each other, and this leads to one more way of defining them:

> **Definition 4 of NP-complete: (via poly-equivalence)**
> A problem is *NP-complete* if it is poly-equivalent to $\textsc{Sat}$, or to any other problem that is already known to be NP-complete.

4 Reformulations of "P=NP?" using NP-completeness

So, now we know that NP-complete problems like SAT are the "hardest" problems in NP. This fact has some remarkable consequences. For example, if someone invented a polynomial time algorithm for SAT, it would immediately yield polynomial time algorithms for every other problem in NP. (Why? Because all those other problems are "easier" than SAT. More formally, the other problems P all polyreduce to SAT, so we can transform an instance of P to an instance of SAT in polynomial time, and then solve the SAT instance in polynomial time.) This gives us yet another way of asking "P=NP?":

P=NP?
(version 5)
Is SAT in P?

And finally, there's nothing special about SAT in version 5 of "P=NP?" above. We know that all the NP-complete problems are poly-equivalent, so "P=NP?" can be reformulated yet again as:

P=NP?
(version 6)
Is any NP-complete problem in P?

5 NP-hardness

Informally, a problem is NP-hard if it is at least as hard as the NP-complete problems. Formally, problem Q is *NP-hard* if there exists some NP-complete problem P with $P \leqslant_{\text{Poly}} Q$. We also define the complexity class NPHard as the set of all NP-hard problems. Obviously, $\mathsf{NPComplete} \subset \mathsf{NPHard}$. But the converse does not hold: it's easy to come up with problems that are NP-hard but not NP-complete. This is because all NP-complete problems are, by definition, decision problems. Therefore, the general, non-decision variants of NP-complete problems are NP-hard but not NP-complete. As specific examples, the general variants of SAT and TRAVELINGSALESPERSON are NP-hard but not NP-complete.

But these examples are a little unsatisfying. Instinctively, we feel that the general and decision variants of SAT and TRAVELINGSALESPERSON are equally hard. Are there more interesting examples of problems that are NP-hard but not NP-complete? Specifically, are there problems that are strictly harder than NP-complete problems? The answer is yes, and we can fall back on our knowledge of computability to find examples—it seems obvious that uncomputable and undecidable problems are strictly harder than NP-complete problems, and this is indeed true.

For a specific example, think back to the problem YESONSOMEINPUT, which takes a program `P.py` as input and outputs "`yes`" if and only if `P.py` outputs "`yes`" on at least one input. We will now prove:

Claim: YESONSOMEINPUT is NP-hard.

Proof of the claim: We will reduce SAT to YESONSOMEINPUT. Given an instance of SAT, with k Boolean variables $x_1, x_2, \ldots, x_k$ write a program `EvaluateSat.py` that takes as input a string of k `1`s and `0`s, indicating the values of the x_i. The output is "`yes`" if the Boolean formula is true for the given values of the x_i. It's reasonably obvious that `EvaluateSat.py` can be constructed in polynomial time, and we won't investigate the details of that here. But now we can ask the question: does `EvaluateSat.py` return "`yes`" on some input? If so, we have found a satisfying assignment to the Boolean formula, so the answer to our instance of SAT is also "`yes`". Hence, sending `EvaluateSat.py` as the input to YESONSOMEINPUT completes the polytime reduction from SAT to YESONSOMEINPUT. ■

6 Consequences of P=NP

See the Fortnow reading.