prev

18 Beyond NP

The theory of NP-completess is a bedrock of computer science because there are so many NP-complete problems, and they crop of everywhere. There are NP-complete problems from mathematics, the theory of databases, the theory of compilers and even from politics.

But even though NP-completeness is central, there is more to the world than that. This section looks at some other classes of problems.

18.1 Co-NP and the Validity Problem

We started looking at difficult problems in Section 14.3 with the Validity Problem for Propositional Logic (VALIDPL). But we have not said anything more about it yet. We have not shown that it is NP-complete, nor have we shown that it is in P.

There is a good reason for that. The validity problem is conjectured to be neither in P nor NP-complete. That is a consequence of the asymmetry of NP: if $A \in$ NP, then there are short, easily checkable proofs that things are in A, but there is no requirement that there are short, easily checkable proofs that things are *not* in A.

But VALIDPL has short, easily checkable proofs of *nonmembership*. To show that ϕ is not valid, show that $\neg\phi$ is satisfiable by finding a truth-value assignment that makes $\neg\phi$ true.

Let's define $\overline{\text{SATPL}}$ to be the set of propositional formulas that are not satisfiable. Then $f(\phi) = \neg\phi$ is a polynomial-time reduction from VALIDPL to $\overline{\text{SATPL}}$. The same function is a polynomial-time reduction from $\overline{SATPL}$ to VALIDPL. So VALIDPL is equivalent in difficulty to $\overline{\text{SATPL}}$.

To deal with VALIDPL, we need a class of languages that are complements of languages that are in NP:

$$\text{Co-NP} = \{X \mid \overline{X} \in \text{NP}\}.$$

Pay close attention to the definition of Co-NP. Co-NP is not the complement of NP.

Now we can define the class of hardest problems in Co-NP in a way that is analogous to the way NP-complete problems are defined. Say that a language L is *Co-NP-complete* provided $L \in$ Co-NP and $X \leq_p L$ for every language $X \in$ Co-NP.

It is easy to show that A is NP-complete if and only if $\overline{A}$ is Co-NP-complete, for every language A. For example, since VALIDPL is equivalent to $\overline{SATPL}$, VALIDPL is Co-NP-complete.

It is also easy to show that, if P $\neq$ NP, then a Co-NP-complete problem has no polynomial-time algorithm. (Imagine, for example, a polynomial-time Turing-reduction from SATPL to VALIDPL. That shows that VALIDPL $\in$ P $\rightarrow$ SATPL $\in P$, or, by taking the contrapositive, SATPL $\notin P \rightarrow$ VALIDPL $\notin P$.)

18.2 NP Intersect Co-NP and Factoring

We know that P $\subseteq$ NP. By symmetry, P $\subset$ Co-NP. So P $\subset$ NP $\cap$ Co-NP.

In Section 13.3 we saw co-partially computable languages and found that the intersection of the class of partially computable sets with the class of co-partially computable sets is exactly the class of computable sets. An obvious question is whether an analogous thing happens here: Is P = NP $\cap$ Co-NP?

Surprisingly, it is conjectured that P $\neq$ NP $\cap$ Co-NP.

Conjecture 18.1 P $\neq$ (NP $\cap$ Co-NP).

What would lead people to make Conjecture 18.1? There must be some decision problem that is conjectured to be in NP$\cap$Co-NP but not in P. And there is: factoring integers. Quick, what are the prime factors of 109,938,432,277?

Actually, the problem of factoring a given integer cannot be in NP $\cap$ Co-NP because it is not a decision problem; the result is a list of factors. But there is a decision problem that is has the same level of difficulty.

Definition 18.2. FACTOR is the following decision problem.

Input. Two positive integers x and k.
Question. Does there exist a factor y of x where $1 < y < k$?

If you have a polynomial-time algorithm that finds the factors of an integer then it is easy to decide FACTOR. And if you have a polynomial-time algorithm for FACTOR then you can find the smallest factor of an integer using binary search. Having found the smallest factor, you divide x by that factor and continue finding factors, stopping when the number that you have is prime. (As mentioned in Section 14, there is a known algorithm that determines whether a given integer is prime in polynomial time.)

There is no known polynomial-time algorithm for FACTOR and FACTOR is conjectured to be in $NP \cap$ Co-NP but not in P.

18.3 Public Key Cryptograpy

Cryptographic systems are based on keys. To encipher a message you use the encipher key, and to decipher a message you use the associated decipher key. A person who has the decipher key is said to *decipher* a message. A person who attempts to do the same job without the benefit of the decipher key is said to *decrypt* the message.

Traditional cryptography is based on the idea that someone attempting to decrypt an enciphered message does not have enough *information* to do so. The standard traditional cryptosystem is a one-time pad, where a randomly chosen sequence of bits called the *pad* is used as a key. To encipher a message (a sequence of bits), you do a bitwise exclusive-or of the message with the pad. To decipher a message, you also do a bitwise exclusive-or of the enciphered message with the pad, which gives the original message back. The pad must only be used once. Using it more than once risks giving away information to an adversary.

Public key cryptography takes a different viewpoint. The encipher and decipher keys are different, and the strength of the system is based on the idea that someone trying to decrypt a message does not have enough *time* to do that. And that depends on the problem of decrypting a message being computationally difficult.

Definition 18.3. A public key cryptosystem is described by two functions $E(k, x)$ and $D(j, y)$ where k is a public encipher key and j is a private decipher key. Functions $E(k, x)$ and $D(j, y)$ must have the following properties.

1. For every x in a limited range, $D(j, E(k, x)) = x$ and $E(k, D(j, x)) = x$. That is, deciphering an enciphered message gives the original message, and enciphering a deciphered message also gives the original message.

2. There is a polynomial-time algorithm to compute $E(k, x)$ and another to compute $D(j, y)$.

3. The *decryption function* $C(k, y)$ is defined to take the encipher key k and an enciphered message y and yield a value x such that $E(k, x) = y$; $C(k, y)$ decrypts without the benefit of j; it is only told k. There should be no polynomial-time algorithm that computes $C(k, y)$.

   Because the encipher key k is public, we write $C(y)$ rather than $C(k, y)$. That slightly simplifies what follows.

The strength of a public key cryptosystem is tied to the (at least apparent) computational difficult of computing $C(y)$.

Several public key cryptosystems are known, and it is not our concern here to describe one. Rather, let's ask whether a public-key cryptosystem exists. Assume that function pair ($E(k, x)$, $D(j, y)$) is such a cryptosystem. For simplicity, assume that messages (x and y) are integers. Text can always be encoded using integers.

Consider the following decision problem DECRYPT determined by the decryption function $C(y)$.

$$\text{DECRYPT} = \{(y, i) \mid C(y) < i\}.$$

DECRYPT must be in NP. As evidence, use the decipher key j. First compute $x = D(j, y)$, then compute $z = E(k, x)$. Accept j as evidence that $(y, i) \in$ DECRYPT provided $z = y$ and $x < i$. Requirement $z = y$ tells you that j is the correct decipher key and requirement $x < i$ tells you that $C(y) < i$.

DECRYPT must also be in Co-NP. The complement of DECRYPT is equivalent to language $\{(y, i) \mid D(y) \geq i\}$, and a similar evidence checker works for that.

So DECRYPT is in NP $\cap$ Co-NP. But if DECRYPT is in P then there is a polynomial-time algorithm to compute $C(y)$. Simply use binary search to

search for the smallest i such that $D(y) < i$. Then $C(y) = i - 1$. That leads to the following conclusion.

Theorem 18.4. A public key cryptosystem can only exist if P $\neq$ NP $\cap$ Co-NP.

It is no accident that public key cryptosystems are based on factoring or on other problems that are in NP $\cap$ Co-NP.

18.4 Polynomial Space

PSPACE is the class of all decision problems that can be solved using $O(n^k)$ bits of memory for some fixed k, where n is the length of the input.

It is known that NP $\subseteq$ PSPACE and Co-NP $\subseteq$ PSPACE, and it is conjectured that NP $\neq$ PSPACE (and Co-NP $\neq$ PSPACE). Polynomial space allows a lot of room for computations. Typical exponential-time algorithms only use a polynomial amount of memory.

A PSPACE-complete problem is one of the hardest problems in PSPACE.

Definition 18.5. A decision problem A if *PSPACE-complete* if

(a) A is in PSPACE and

(b) $X \leq_p A$ for every problem $X \in$ PSPACE.

Several PSPACE-complete problems are related to two-person games. An example is *Generalized Checkers*, defined as follows.

> **Input.** A placement of red and black kings on an $n \times n$ checkerboard.
> **Question.** Assuming that it is red's move, does red have a winning strategy from the given configuration?

Geography is a game that children can play without any props. A child thinks of the name of a country (or other chosen category). If the first child selects Sweden, then the next child must select a country name that begins with N, the last letter of Sweden. Suppose that child chooses Nepal. Now the next player must choose a country name that starts with L. Countries cannot be reused, and the first child who cannot think of a country name loses.

There is a version of Geography that is played on a directed graph. A vertex is selected as the start vertex s. The first player selects a vertex u where there is a directed edge from s to u. The second player selects a vertex v where there is a directed edge from u to v. Play alternates. No vertex that was previously selected can be selected again.

The *Generalize Geography* decision problem is as follows.

> **Input.** A directed graph G with a selected start vertex.
> **Question.** Does the first player have a winning strategy on the game of Geography played on G?

Generalize Geography is known to be PSPACE-complete.

In practice, PSPACE-complete problems appear very difficult to solve. Not only do they appear not to have polynomial-time algorithms, but they do not appear to have polynomial-time evidence checkers. Even if someone knows the answer to a particular input of PSPACE-complete problem, he or she cannot convince you that the answer is correct using a short, easy to check proof!

It is worth thinking about how you would write an evidence checker for Generalized Geography. What would the evidence be? The most obvious evidence that the first player has a winning strategy is the strategy. But that is really huge! It must not only say what the first player's first move is, it must say how the first player responds to each move of the second player. As the number of moves grows, the strategy grows exponentially in size. An exponential-size piece of evidence clearly cannot be checked in polynomial time.

But, even though PSPACE appears to be larger than NP, surprisingly, nobody knows whether PSPACE = P. Even the huge jump from polynomial time to polynomial space is not enough for us to demonstrate a separation. If you want to prove that P $\neq$ NP, you might warm up by proving that P $\neq$ PSPACE; that ought to be easier.

18.5 Exponential Time

Definition 18.6. *EXPTIME* is the class of decision problems that are solvable in time $O(2^{n^k})$ for some fixed k, where n is the length of the input.

It is known that PSPACE $\subseteq$ EXPTIME, and PSPACE is conjectured to be a proper subset of EXPTIME.

With EXPTIME, we finally have a provable separation! It is known that P $\neq EXPTIME$. So at least one of the subset relations P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ EXPTIME is surely a proper subset relation. Of course, they are all conjectured to be proper.

There is a notion of an EXPTIME-complete problem, defined in the usual way as a hardest problem in EXPTIME.

Definition 18.7. A decision problem A is *EXPTIME-complete* if

(a) A is in EXPTIME and

(b) $X \leq_p A$ for every problem $X \in$ EXPTIME.

There are EXPTIME-complete problems. There is a typed programming language called ML. The ML Type Checking problem is as follows.

Definition 18.9. The *ML Type Checking Problem* is the following decision problem.

Input. An ML program p.
Question. Is p well-typed (free of type errors)?

The ML Type Checking Problem is known to be EXPTIME-complete. Fortunately, ML programmers tend not to write programs that are difficult to type check. But if you want to, you can write a short ML program that will bring an ML compiler to its knees; in practice, the memory requirements overwhelm the compiler, and it gives up.

prev