Certificates, Function Classes and Cryptography

We originally defined NP as the collection of languages accepted by a nondeterministic machine in polynomial time. We have also used another equivalent notion, that of decision problems that can be decided by a "generate-and-test" program that generates candidate solutions which are not too long (there is a polynomial that bounds their length in terms of the length of the input) and then test (deterministically in polynomial time) that the candidate solution really is a solution. This is essentially the definition of NP we have been using in establishing that various problems are in the class. We now look at a formal statement that characterises it in this way. A language $L \subseteq \Sigma^\star$ is in NP if, and only if, it can be expressed as:

$$L = \{x \mid \exists y R(x, y)\},$$

where R is a relation on strings satisfying two conditions:

1. R is decidable in polynomial time by a deterministic machine.
2. R is *polynomially balanced.* That is, there is a polynomial p such that if $R(x, y)$ and the length of x is n, then the length of y is no more than $p(n)$.

In such a case, if $R(x, y)$ holds, we say that y is a certificate of the membership of x in L. Or, as we might have said earlier, it is a "solution" to x. For example, in the case where L is SAT and x is a Boolean expression, y would be an assignment of truth values to the variables of x and $R(x, y)$ would be the relation that holds if y makes x true. Similarly, in the case of 3-colourability, x would be a graph, y an assignment of colours to the graph and $R(x, y)$ would hold if y was a valid colouring.

co-NP

The class of languages co-NP was defined as the complements of languages in NP. So, if L is in co-NP, there is a polynomial time decidable and polynomially balanced relation S such that $\bar{L} = \{x \mid \exists y S(x, y)\}$. In other words $\bar{L} = \{x \mid \forall y \neg S(x, y)\}$. Since P is closed under complementation, $\neg S(x, y)$ is decidable in polynomial time. So, we can equivalently characterise the class co-NP as the set of languages L for which there is a polynomial-time decidable relation R such that:

$$L = \{x \mid \forall y \, |y| < p(|x|) \to R(x, y)\}.$$

This situation is often summed up by saying that NP is the collection of those languages for which there are succinct certificates of membership and co-NP is the collection of languages for which there are succinct certificates of disqualification.

Since every language in P is also in NP, and the complement of a language in P is itself in P, it follows that every language in P is also in co-NP. In other words P $\subseteq$ NP $\cap$ co-NP. The situation is depicted in figure 9.

Clearly, if it were the case that P = NP, it would follow that NP = co-NP. Other than that, we know nothing about whether the containments pictured in figure 9 are proper. Any of the following situations is consistent with what we can prove so far:

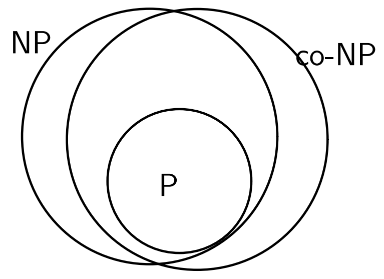

Figure 9: Relation between the classes P, NP and co-NP

- P = NP = co-NP
- P = NP $\cap$ co-NP $\neq$ NP $\neq$ co-NP
- P $\neq$ NP $\cap$ co-NP = NP = co-NP
- P $\neq$ NP $\cap$ co-NP $\neq$ NP $\neq$ co-NP

Validity There is a notion of completeness for co-NP entirely analogous to that of NP-completeness.
Definition A language L is said to be *co-NP-complete* if L is in co-NP and for every language $A \in$ co-NP, $A \leq_P L$.

Indeed, the complement of any NP-complete language is co-NP-complete. This is because if f is a a reduction from a language L_1 to a language L_2 then it is also a reduction of $\bar{L_1}$–the complement of L_1–to $\bar{L_2}$–the complement of L_2, as $x \in \bar{L_1} \Leftrightarrow x \notin L_1 \Leftrightarrow f(x) \notin L_2 \Leftrightarrow f(x) \in \bar{L_2}$. So, $\bar{\mathsf{SAT}}$ is co-NP-complete.

Recall that VAL is the set of Boolean expressions that are valid, i.e. true for any assignment of values to the variables. VAL is in co-NP. There is any reduction from $\bar{\mathsf{SAT}}$ to VAL. It is just the function that maps a Boolean expression ϕ to $\neg\phi$. It follows that VAL is co-NP-complete. Hence, if there were a polynomial time algorithm for solving VAL, this would imply that P = NP = co-NP. Similarly, if we could show that VAL is in NP it would follow that every problem in co-NP is in NP by composing the polynomial time reduction of the problem to VAL with the nondeterministic machine deciding VAL.

Prime Numbers

We now turn our attention to a particular familiar decision problem on numbers. It is simply this: given a number n, is it prime? Or, to formulate it as a language, it is

$$\mathsf{PRIME} = \{x \in 1\{0,1\}^\star \mid x \text{ is the binary representation of a prime number}\}.$$

What is interesting about this problem is that it is in NP and, unlike other problems in NP that we have encountered, it is far from obvious that it is in NP. The problem is not known (or believed) to be NP-complete, but it takes some effort to even show that it is in NP.

There is an obvious algorithm for checking whether a number n is prime. For each integer $1 < m < \sqrt{n}$, check whether m divides n. This is not a polynomial time algorithm, since it requires $\sqrt{n}$ steps—which is exponential in the number of bits in n (remember we always measure complexity in terms of the size of the input, and the fair way to measure the size of a number is the number of bits it takes to represent it). However, this simple algorithm is already sufficient to establish that the language PRIME is in co-NP. This is because it shows that the language can be defined as:

$$\forall y (y < x \rightarrow (y = 1 \vee \neg(\mathrm{div}(y, x))))$$

where $\mathrm{div}(y, x)$ represents the relation y divides x. Another way of putting this is that the language PRIME has succinct certificates of disqualification because for any number n, a factor m of n is sufficient to certify that n is not prime. Or equivalently, the language

$$\mathsf{COMP} = \{x \in 1\{0,1\}^\star \mid x \text{ is the binary representation of a composite number}\}$$

is in NP, because it has succinct certificates of membership.

However, Vaughn Pratt showed in 1976 that PRIME is in fact in NP. This is best understood as a proof that there are succinct certificates of primality. That is, it is possible to write, for each prime number p a string $C(p)$ whose length is bounded by a polynomial in the number of bits in p and such that, given p and $C(p)$ it is easy to verify that p is a prime. Here "easy" means that there is a deterministic algorithm running in polynomial time that will take inputs p and $C(p)$ and confirm that p is prime.

So, what are these certificates of primality. They are based on the following number-theoretic fact, which we will take as given without proof. A proof is instructive, but would take us too far beyond the scope of this course:

> A number $p > 2$ is *prime* if, and only if, there is a number r, $1 < r < p$, such that $r^{p-1} = 1 \bmod p$ and $r^{\frac{p-1}{q}} \neq 1 \bmod p$ for all *prime divisors* q of $p - 1$.

At first sight, it might seem that r would be an appropriate certificate of the primality of p. But, if we were given r, how would we check that it satisfied the conditions above? It is possible to check that $r^{p-1} = 1 \bmod p$, by evaluating r^{p-1} by repeated squaring and doing all arithmetic mod$p$. But, to check that $r^{\frac{p-1}{q}} \neq 1 \bmod p$ for all *prime divisors* q of $p - 1$, we have to know what the prime divisors of $p-1$ are. If we had a polynomial-time algorithm for finding the prime divisors of a number, we would hardly be worrying about whether PRIME is in NP. So, r by itself does not seem to be a good certificate.

Would it suffice to take r along with all of its prime divisors? That is, given r and all of its prime divisors $q_1, \ldots, q_k$ can one check that p is indeed prime. Once again, checking that $r^{p-1} = 1 \bmod p$ is not too difficult. What's more, one can now check that $r^{\frac{p-1}{q}} \neq 1 \bmod p$ for each $q \in \{q_1, \ldots, q_k\}$. But, to be convinced that this establishes the primality of p, we need to be sure that $q_1, \ldots, q_k$ are indeed the prime divisors of r. One can check that they multiply to give r, but how does one check that they are prime? We seem to be back to square one. The solution is to include in $C(p)$ certificates of primality for each q. We are finally ready to define the certificates of primality we use, which are defined recursively. The certificate $C(p)$ is defined as:

$$(r, q_1, C(q_1), \ldots, q_k, C(q_k)),$$

where r satisfies the condition that $r^{p-1} = 1 \bmod p$, and $r^{\frac{p-1}{q}} \neq 1 \bmod p$ for each of $q_1, \ldots, q_k$, which are the prime factors of r and $C(q_i)$ is the certificate of primality of q_i.

To complete the proof that PRIME is in NP, we need to prove two things:

1. $C(p)$ is succinct – that is there is a polynomial bound on the length of $C(p)$, in terms of the length of p.

2. There is a polynomial time algorithm that will check that $C(p)$ is a valid certificate of the primality of p.

For the first part, we can prove by induction that $|C(p)| \leq 4(\log p)^2$. This can be checked by hand for 2 and 3. Suppose then that $p \geq 5$. The length of the certificate $C(p)$ is given by:

1. the length of r – at most $\log p$ bits;

2. k commas – where k, the number of prime divisors of $p-1$ is less than $\log p$;

3. the number of bits required to represent the q_i – since the product $q_1 \cdots q_k \leq p-1$, $\log q_1 + \cdots + \log q_k \leq \log p$; and

4. the sum of the lengths of $C(q_i)$ which, by induction hypothesis is at most $4\sum_i (\log q_i)^2$.

Since $p-1$ must be even, $q_1 = 2$. We can therefore write the sum in (4) as $|C(2)| + 4\sum_{i=2}^{k}(\log q_i)^2$. What's more, since $q_2 \cdots q_k \leq (p-1)/2$, it follows that $\sum_{i=2}^{k} \log q_i \leq (\log p) - 1$ and therefore $\sum_{i=2}^{k}(\log q_i)^2 \leq (\log p - 1)^2$. Putting this in for (4), and adding the other terms, it is easy to see that the total length of $C(p)$ is at most $4(\log p)^2$.

Finally, we need to check that there is a polynomial time algorithm which given $C(p)$ and p will verify that $C(p)$ certifies that p is prime. Note that, by repeated squaring, $r^x \bmod p$ can be computed with $O(\log x)$ multiplications. Since each of the multiplications is done mod$p$, we know that it involves only $O(\log p)$-bit numbers, and can be done in $O((\log p)^2)$ time. Finally, $r^x \bmod p$ has to be computed for fewer than $O(\log p)$ distinct numbers x, since the number of distinct prime factors of $p-1$ is less than $\log p$. Overall, our verification algorithm is $O((\log p)^4)$ – polynomial in the length of p.

Function Classes

So far, in discussing complexity of problems we have only considered decision problems—those where there is a yes/no answer for every possible input. In some cases, as with the travelling salesman problem, this was somewhat artificial. We forced what is naturally an optimisation problem into the mould of a decision problem in order to make it fit our theory. The same could be said of the problems Clique, IND and a variety of the scheduling problems we considered. The justification for this is that we were concerned with establishing lower bounds. And, if we could prove that there is no polynomial time algorithm for the decision problem TSP, we would know that there is none for the optimisation problem either. Similarly, the proof of NP-completeness of TSP shows that if there were a polynomial-time algorithm for the optimisation problem, then $P = NP$. Furthermore, there is a connection in the other direction as well. A polynomial-time algorithm for the decision problem could also be used to obtain a polynomial-time algorithm for the optimisation problem. This is true for problems (TSP, Clique and IND are among them) where the maximum possible value of the measure being optimised can be represented by a string that is at most polynomial in the length of the input. This is because a black-box solving the decision problem could be used by a binary search algorithm to find the optimum value. For instance, given a black box that given a pair G, K gives a yes/no answer to the question whether G has a clique with at least K elements, we can find out the size of the largest clique in G with $\log |G|$ queries to the black box.[4]

Still, there is something interesting one can say about the complexity of *functions* as opposed to decision problems and one can develop a complexity theory around such things. While we know what the function computed by a deterministic machine is, it is not easy to speak of the function computed by a nondeterministic machine since the string that is produced as output for a given input x is not determined. Instead, we will talk of the complexity of the witness functions of languages in NP.

Suppose we have a language L in NP. Then, we know

$$L = \{x \mid \exists y R(x, y)\}$$

where R is a polynomially-balanced, polynomial time decidable relation.

Definition

A *witness function* for L is any function f such that:

- if $x \in L$, then $f(x) = y$ for some y such that $R(x, y)$;
- $f(x) =$ "no" otherwise.

The class FNP is the collection of all witness functions for languages in NP.

It is reasonably clear that if an NP-complete problem L had a polynomial-time computable witness function, then P would be the same as NP. For instance, a witness function for SAT would be a function which, for any Boolean expression ϕ returns a satisfying truth assignment for ϕ if there is one and the string "no" otherwise.

Conversely, if P = NP, then every function in FNP is computable in polynomial time, by a binary search algorithm. We could define a notion of polynomial-time reduction between functions under which one can show that the witness function for SAT is FNP-complete. If we write FP for the collection of functions that are computable by a deterministic machine in polynomial time, the question of whether FP = FNP is equivalent to the question whether P = NP. Still, there are interesting functions in FNP, beside those that are associated with NP-complete problems. We look at one such next.

Factorisation The factorisation problem is that of finding, given a positive integer n, its prime factors. Since we also want to be able to verify that a given factorisation is correct, we define the factorisation function as the function that maps n to a tuple

$$(2, k_1; 3, k_2; \ldots; p_m, k_m; C(2), \ldots, C(p_m)),$$

[4]Actually finding a clique is a different matter.

where $n = 2^{k_1}3^{k_2}\cdots p_m^{k_m}$ and $C(p)$ is a certificate of primality of p.

This function is in FNP because it is a witness function for a trivial problem in NP, namely the set of all positive integers. Still, it is not known whether this function is computable in polynomial time. Indeed, public key cryptographic systems are built on the assumption that the factorisation cannot be done quickly.

Cryptography

The fundamental aim of cryptography is to enable Alice and Bob to communicate (as in figure 10) without Eve being able to eavesdrop.

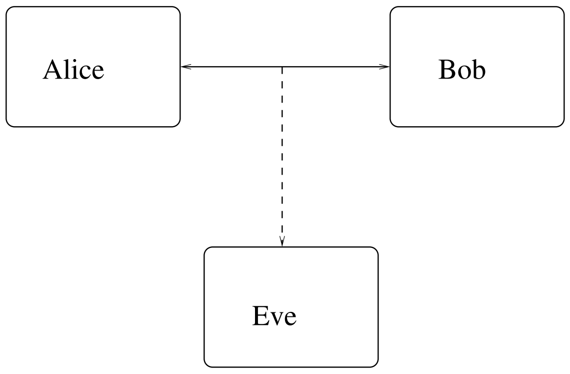

Figure 10: Alice, Bob and Eve

In any cryptographic system, there are two keys: e – the encryption key and d – the decryption key. There are also two functions D and E such that: for any x, $D(E(x,e),d) = x$. A private key system relies for its security on keeping both e and d secret.

For instance, we could take $d = e$ and let both D and E be the *exclusive or* function. This is known as the one-time pad. Since $(x \oplus e) \oplus e = x$, this satisfies the required condition for a cryptographic system. Furthermore, the one time pad is provably secure in that the only way that Eve would be able to decode a message is by knowing the key. For, if Eve knows both the plain text x and the encrypted text y, she can work out the key $e = x \oplus y$.

In contrast, in a public key cryptographic system the key e is made public while d is kept secret. We still have functions D and E such that: for any x, $D(E(x,e),d) = x$. If the system is to work in reasonable time, then both D and E should be computable in polynomial time. However, if Eve is not to be able to eavesdrop, the function that maps $E(x,e)$ to x without knowing d should not be computable in polynomial time. However, this function is necessarily in the class FNP. This is because it is a witness function for the set:

$$\{y \mid \exists x E(x,e) = y\}.$$

Thus, public key cryptography is not provably secure in the way that the one-time pad is. Instead, it relies on the unproved hypothesis that there are functions in FNP that are not in FP.

One Way Functions To be more precise, in order for public-key cryptography to work, we would like to have a *one-way function*, which is defined as a function f meeting the following conditions:

1. f is one-to-one.
2. for each x, $|x|^{1/k} \leq |f(x)| \leq |x|^k$ for some k.
3. $f \in \mathsf{FP}$.
4. $f^{-1} \notin \mathsf{FP}$.

The reason for the first condition is obvious: we don't want to have two distinct plain text strings map to the same encrypted string. The second condition ensures that neither the map from plain text to encrypted text nor the reverse map will result in a greater than polynomial increase in the length of the string. The third condition states is to ensure that encryption can be done efficiently and the last is to ensure that decryption is difficult.

To be really useful, we would also like f to satisfy the condition that it can be easily inverted when supplied with a decryption key d as a parameter. Note, however, that we don't even know whether there are any functions that satisfy the four conditions above. Indeed to prove that such one-way functions exists would involve proving that $\mathsf{P} \neq \mathsf{NP}$. However, a great deal of money has been invested in the belief that the RSA function, defined by:

$$f(x, e, p, C(p), q, C(q)) = (x^e \bmod pq, pq, e)$$

is a one-way function. This takes the plain text x and an encryption key e along with two certified primes p and q and produces the encrypted text $x^e \bmod pq$ along with the public key (pq, e).

Given our inability to prove lower bounds, we are unable to prove that RSA is secure. It's reasonable to ask, however, whether we can place as much confidence in it as in our belief that P and NP are different. Can we prove that, assuming $\mathsf{P} \neq \mathsf{NP}$, RSA is not invertible in polynomial time? Indeed, can we show, under the assumption that $\mathsf{P} \neq \mathsf{NP}$ that some one-way function exists? Is there a function f that satisfies the first three conditions in the definition of a one-way function and for which f^{-1} is FNP-complete? The answer to these questions is, unfortunately, no. The existence of one-way functions is logically equivalent to a stronger statement than $\mathsf{P} \neq \mathsf{NP}$. To understand this, we need to introduce one further complexity class.

Definition

A nondeterministic machine is *unambiguous* if, for any input x, there is at most one accepting computation of the machine.

UP is the class of languages accepted by unambiguous machines in polynomial time.

Equivalently, we could say that UP is the class of languages L for which:

$$\{x \mid \exists y R(x, y)\},$$

where R is polynomial time computable, polynomially balanced, *and* for each x, there is *at most one* y such that $R(x, y)$. In other words, R is a partial function.

It is clear from the definitions that $\mathsf{P} \subseteq \mathsf{UP} \subseteq \mathsf{NP}$. It is considered unlikely that there are any NP-complete problems in UP. It is, in fact, difficult to think of any natural problems that are in UP but not in P. However, we can show that the existence of one-way functions is equivalent to the statement that $\mathsf{P} \neq \mathsf{UP}$.

To see why this is the case, first assume that we have a one-way function f. Then, we can show that the language L_f defined by

$$L_f = \{(x, y) \mid \exists z(z \leq x \text{ and } f(z) = y\}.$$

Here, we are assuming that f acts on positive integers represented by binary strings. It is not too difficult to see that L_f is in UP, since there is a nondeterministic machine that recognises it by "guessing" a value for z and then checking that $f(z) = y$. Since the function f is one-to-one, there is at most one such z and therefore the machine is unambiguous.

To see that L_f is not in P, we note that if it was, then we could compute f^{-1} in polynomial time, using a binary search procedure. That is, given a value y, we know that if there is a z such that $f(z) = y$, then it has $z \leq 2^{(\log y)^k}$, by condition 2 in the definition of a one-way function. So, we can find z using a binary search procedure making at most $(\log y)^k$ calls to the algorithm that checks for membership in L_f. This gives a polynomial time algorithm for computing f^{-1}, contradicting the assumption that f is one-way.

We have shown that the existence of a one-way function implies that $\mathsf{P} \neq \mathsf{UP}$. Conversely, suppose that we have a language L that is in UP but not in P and U is an unambiguous machine accepting L. Define the function f_U by the following:

> if x is a string that encodes an accepting computation of U, then $f(x) = 1y$ where y is the input string accepted by this computation;
>
> $f(x) = 0x$ otherwise.

The function f is one-to-one because the machine is unambiguous and therefore a given string y cannot have more than one accepting computation. It is polynomially bounded by the fact that machine U runs in polynomial time and therefore the lengths of computations are bounded in the length of the input. It is also easily checked that f is in FP and if f^{-1} were in FP, then the language L would be in P.