NP Completeness

We're going to spend the next few lectures studying a set of computational problems called the **NP-complete** problems. The essential facts are these:

- Any **NP**-complete problem can be solved by a simple, but exponentially slow algorithm.
- We don't have polynomial-time solutions to any **NP**-complete problem.
- We can prove that either *all* **NP**-complete solutions have polynomial-time solutions, or *none* of them do.
- It is generally believed that none of them do. But proving this would require solving the **P versus NP problem**, one of the best known unsolved problems in mathematics, much less theoretical computer science.

In this lecture we"ll go over the basic definitions and results of the theory of **NP**-completeness. In the next two lectures we'll see examples of **NP**-complete problems and how to prove that they are **NP**-complete. This material has considerable overlap with that of CMPSCI 601, but we'll try here to present things from a more algorithmic perspective.

Let's first review what we mean by a **problem**. The algorithms we've seen so far in the course have addressed several different kinds of problems:

- **Decision Problems:** The input is some piece of data (without loss of generality, a string) and the output is a bit. A decision problem defines a **formal language**, the set of strings for which the desired output is "true". For example, we might input an undirected graph and output whether it has a perfect matching.
- **Search Problems:** Here the desired output is some piece of data related to the input. Often we have more than one acceptable output string. For example, we might input an undirected graph and output a perfect matching in the graph, if one exists.
- **Optimization Problems:** We have an input, a set of possible **options** related to it, and a **reward function** on those options. The output is an option that maximizes the possible reward for the input. (If we want to minimize the "score", we call it a **cost function**.)
- **Approximation Problems:** These are like optimization problems, but we are satisfied with an option that gets *close to* the maximum reward or minimum cost, rather than the absolute best option.

Here are two problem domains where we can define each of these four kinds of problems.

A **boolean formula** is an expression where the atomic elements are **boolean constants** and **input variables**, and the operators are AND ($\wedge$), OR ($\vee$), and NOT ($\neg$). A boolean formula is said to be in **conjunctive normal form (CNF)** if it is the AND of zero or more **clauses**, and each clause is the OR of zero or more **literals**, where a literal is an input variable or a negated input variable. For example, the formula

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_4) \wedge (\neg x_1 \vee x_4)$$

is in CNF. It has three clauses, and the clauses have three, one, and two literals respectively.

A boolean formula has a **truth value** that depends on the truth values of its inputs. The formula is **satisfiable** if there is at least one setting of the inputs that makes it true. (This means that the negation of the formula is not a tautology.)

The decision problem SAT is to input a boolean formula and decide whether it is satisfiable. The decision problem CNF-SAT is the same but requires that the input is in CNF. The decision problem 3-SAT is the same but requires that the input be in CNF with at most three variables per clause.

Given a boolean formula, a natural search problem is to find a satisfying assignment, if one exists. If we had a decision procedure for SAT, we could use it to solve the search problem as follows:

- Test the input formula for satisfiability.
- Substitute a constant 0 for x_1 in the formula and test it again. If it is still satisfiable, set x_1 to 0, otherwise set x_1 to 1.
- With x_1 set this way, try substituting 0 for x_2. If the formula is still satisfiable, set x_2 to 0, otherwise set x_2 to 1.
- Continue in this way, setting each variable to a constant in a way that keeps the formula satisfiable.
- When all the variables are set, return the setting.

If the decision procedure runs in time $t(n)$, the search procedure runs in $O(nt(n) + n^2)$.

A natural optimization version of CNF-SAT or 3-SAT is to input the formula and find a setting that satisfies (makes true) as many clauses as possible. These problems are called MAX-SAT and MAX-3-SAT respectively. In terms of "polynomial" or "not-polynomial", the search and decision problems are also of equal difficulty, as we'll see later.

Our second problem domain is **graph colorability**. Given an undirected graph, a **valid vertex coloring** is an assignment of a color to each vertex such that there is no edge whose two endpoints are the same color. The **graph coloring problem** is to input graph and find a coloring that uses as few colors as possible. (This is closely related to the **map coloring problem**, most familiar from the theorem that every planar map can be colored with four colors.) Again we have multiple versions of the problem:

- **Decision:** Given G and k, does G have a k-coloring?
- **Search:** Given G and k, find a k-coloring if one exists.
- **Optimization:** Given G, find a coloring with as few colors as possible.
- **Approximation:** Given G, find a coloring that uses close to the minimum number of colors.

Once again if we had polynomial-time solutions to any of the decision, search, or optimization problems, we could find polynomial time solutions to the others:

- The search problem immediately answers the decision problem.
- The optimization problem also solves the decision problem.
- Given a search procedure, we can use binary search to find the optimal k and then find a k-coloring.
- To solve the search problem as we did for satisfiability, we need a decision procedure that tells us whether a *partial* k-coloring can be extended to a complete one. There is a simple trick to convert a partial-coloring decision problem into an equivalent ordinary decision problem.

To study whether a problem is solvable in polynomial time, then, it usually suffices to look at the simplest case, that of decision problems. We therefore define the class **P** to be the set of decision problems for which there exists an algorithm solving them in $O(n^k)$ time for some constant k.

As we discussed in the first lecture, **P** is a potentially problematic definition for “quickly solvable problems”. But if we could show a problem to *not* be in **P**, we would show at least that its running time does not scale with increasing input size. And **P** has the advantage that it is robust across models – several different ways of formalizing “algorithm” and “running time” give us the same class.

We’re now ready to define **NP**-completeness. The first step is to define **NP**, a class of decision problems:

Definition: A formal language A is in **NP** if there exists another language B in **P**, such that for any string x, x is in A iff there exists a string y, with $|y| = |x|^{O(1)}$, such that (x, y) is in B.

An equivalent, more algorithmic definition of **NP** is as follows. An **NP-procedure** consists of a **guess phase** and a **verification phase**. Given an input x, the guess phase chooses an arbitrary string y such that $|y| = |x|^{O(1)}$. The verification phase is an algorithm that takes both x and y as input and returns a bit. We say that x is in the language of the **NP**-procedure iff *it is possible* for the procedure to guess a y making the verification phase output “true”.

The two definitions are equivalent because if y exists, it is possible for the guess procedure to guess it, and vice versa. Earlier we mentioned the idea of *randomly* guessing a string – this gives us a Monte Carlo algorithm for any **NP** language. But the success probability of this Monte Carlo algorithm could be very small. If there is exactly one y such that the verification phase accepts (x, y), the probability of guessing it is $2^{-|x|^{O(1)}}$. This probability is too small for amplification to be useful.

There *is* an obvious deterministic decision procedure for any **NP** language – simply cycle through all possible strings y and see whether the verification procedure accepts (x, y). The problem is that there are $2^{n^{O(1)}}$ possible y's, of course. Our question is whether there is a better way to decide membership in the **NP** language, one that might run in polynomial time.

Note that proving a language to be in **NP** is generally simple. We need to define a string to be guessed, and define a poly-time verification procedure that accepts the input and a guess iff the guess *proves* that the input is in the language. For SAT, our guess is a setting of the n input variables and the verification procedure evaluates the input formula with that setting. For 3-COLOR, the guess is a three-coloring of the vertices and the verification procedure checks every edge and accepts if all of them have endpoints of two different colors.

The class **NP** includes all the languages in **P**, of course, because we can have an **NP** procedure with no guess phase that just tests for membership in the verification phase. We need to determine relationships among the languages in **NP**, using the notion of **reduction**. Above we argued, for example, that *if* the decision problem SAT were in **P**, so would be the corresponding search problem. We did this by describing an algorithm for the search problem that would run in polynomial time if it were allowed to make calls to a hypothetical poly-time algorithm for the decision problem. This is called a **Cook reduction**. For defining **NP**-completeness, though, we will need a slightly different notion of reduction.

Let A and B be two formal languages, possibly over different alphabets. A **Karp reduction** from A to B is a poly-time computable function f such that for any string x, $x \in A$ if and only if $f(x) \in B$. If such a reduction exists we say that A is **poly-time reducible** to B and write "$A \leq_p B$". We also sometimes read this as "A is no harder than B".

The key point of this definition is that if B is in **P** and $A \leq_p B$, we can be sure that $A \in$ **P** as well. This is because we can decide whether x is in A in poly-time by:

- Computing $f(x)$,
- Testing whether $f(x) \in B$, and
- Reporting that answer as our answer for whether $x \in A$.

This is a poly-time decision procedure for A because f is assumed to be poly-time computable, which means not only that the first step takes poly-time but also that the string $f(x)$ has a length polynomial in that of x. So the time of the second step is polynomial *in the length of* $f(x)$ because B has a poly-time decision procedure, and this is polynomial *in the length of* x because a polynomial of a polynomial is just another polynomial.

The relation $\leq_p$ is a **preorder** on the languages in **NP**, because it is reflexive and transitive. It is not antisymmetric because it is possible that both $A \leq_p B$ and $B \leq_p A$ are true – in this case we say that A and B are **poly-time equivalent**. Then $\leq_p$ is a partial order on the equivalence classes of this relation.

With the exceptions of $\emptyset$ and Σ^*, all **P** languages are poly-time equivalent to each other. Since $\mathbf{P} = \mathbf{NP}$ might be true, it might be that there is only one nontrivial equivalence class in **NP**. But if $\mathbf{P} \neq \mathbf{NP}$, as is generally believed, there is another important equivalence class:

Definition: A language B is **NP**-complete if (1) it is in **NP**, and (2) for any language A in **NP**, $A \leq_p B$.

Thus the **NP**-complete languages, if they exist, are the *hardest* languages in **NP**. It should be easy to see that they form an equivalence class, and that if any **NP**-complete language is also in **P**, it follows that $\mathbf{P} = \mathbf{NP}$.

If a language is **NP**-complete, then, we have strong evidence that it is not in **P**. We can use the **NP**-completeness of a language to talk about non-decision problems, even though by definition these cannot be **NP**-complete.

Definition: A problem X (with boolean output or otherwise) is said to be **NP-hard** if there is a Cook reduction from X to some **NP**-complete problem B. That is, there is a poly-time algorithm, with access to a hypothetical poly-time algorithm for X, that decides B.

It should be clear that if X is **NP**-hard and there actually *is* a poly-time algorithm for X, then $\mathbf{P} = \mathbf{NP}$.

Our general methodology will be to develop a library of **NP**-complete problems. Once we have one problem, there is a straightforward way to get more:

Lemma: Let B be an **NP**-complete language and A be a language. If we prove:

- $A \in \mathbf{NP}$
- $B \leq_p A$

then we may conclude that A is **NP**-complete.

But how do we start this process? In CMPSCI 601, we prove the **Cook-Levin Theorem**, that the language SAT is **NP**-complete. That proof goes further into the details of the model of computation than we want to go here, so we will take its result on faith. But if we do that, the concept of **NP**-completeness becomes somewhat mysterious – why should there be any **NP**-complete languages at all?

In the rest of this lecture we'll give an unconditional proof that a particular language is **NP**-complete. We'll still need the Cook-Levin Theorem, because SAT is a much more convenient starting place to build up a library of **NP**-complete languages. (The discovery that there are *interesting* **NP**-hard problems, that people really wanted to solve, is what made the theory applicable – this was the great achievement of Karp.)

To define our **NP**-complete language, we need to fix a model of computation that includes **NP** procedures. If Z is an **NP**-procedure, we need there to be a string z that indicates, given an input x, how long a string y should be guessed by Z and what the verification procedure of Z will do given a certain number of steps to run.

Our language U will be the set of tuples $(z, x, 1^g, 1^t)$ such that if z denotes the **NP** procedure Z in this way, there is a string y of length g such that the verification phase of Z accepts (x, y) in at most t steps. The inputs g and t are given in unary so that the running time of the procedure will be only polynomial in the size of the input to U.

Theorem: U is **NP**-complete.

Proof: We first must show that U is in **NP**. We define an **NP**-procedure that on input $(z, x, 1^g, 1^t)$, guesses a string y of length g and then runs the verification phase of Z for up to t steps on (x, y). If this accepts, we accept the input tuple. Otherwise (it rejects, it runs out of time, or z isn't valid) we reject. Clearly it is *possible* for this procedure to accept $(z, x, 1^g, 1^t)$ iff it is in U.

Now let A be an arbitrary language in **NP**. We will show $A \leq_p U$. There must exist a string a denoting an **NP**-procedure for A, where on an input of length n the procedure will guess a string of length at most $p(n)$ and run for at most $q(n)$ steps, where p and q are polynomials. We need a function f such that for any string x, $x \in A$ iff $f(x) \in U$. To do this, we define $f(x)$ to be the tuple $(a, x, 1^{p(|x|)}, 1^{q(|x|)})$. This is easy to compute if a, p, and q are known, and takes around $n + p(n) + q(n)$ time to write down the answer, a polynomial in the input size n. By the definition of U and A, this $f(x)$ is in U iff $x \in A$.

Next time we'll see several examples of **NP**-completeness proofs, starting from the Cook-Levin Theorem about SAT.