2003/11/17

NP-completeness

Most important subject that most students don’t learn well enough.

- Given a set of $S$ integers, is there a way to partition $S$ into two subsets whose sums are equal?

- Given a graph $G$, is there a path in $G$ that goes though each vertex exactly once?

- Given a graph $G$ and an integer $K$, is there a set $C$ of no more than $K$ vertices such that every edge in $G$ is incident to at least one vertex in $C$?
• Given a Boolean formula $\varphi$, is there a truth-setting of the variables that makes $\varphi$ true?

• Given a graph $G$ and an integer $K$, does $G$ have a complete subgraph of size at least $K$?

No algorithms are known for any of these problems that run in less than exponential time (essentially by exhaustive search).

BUT, a fast algorithm for any one of them will immediately give fast algorithms for the rest of them.

All these problems, and many others, are NP-complete.

The theory of NP-completeness is the best tool available to show that various interesting problems are (most likely) inherently difficult.
So far, no one has been able to prove mathematically that NP-complete problems *cannot* be solved by fast algorithms, but this hypothesis is supported by a huge amount of empirical evidence, namely, the failure of *anybody* to find a fast algorithm for *any* NP-complete problem despite intense and prolonged effort.
If your boss asks you to find a fast algorithm for a problem and you cannot find one, you may be able to show your boss that the problem is NP-complete, and hence equivalent to the problems above, which the smartest minds in the field have failed to crack.

At least your boss would know that she won’t do any better by firing you and hiring someone else.
Decision Problems

We restrict our attention to decision (i.e., yes/no) problems for convenience.

**Definition 1** A decision problem is specified by two ingredients:

1. *a description of an instance of the problem (always a finitely representable object),* and

2. *a yes-no question regarding the instance."

All the questions above are decision problems. When stating a decision problem, we name it, then explicitly give its two ingredients. For example,
VERTEX COVER
Instance: an undirected graph $G$ and an integer $K$.
Question: is there are set of vertices $C$ of size at most $K$ such that every edge in $G$ is incident to a vertex in $C$?

All instances of a decision problem are either yes-instances or no-instances, depending on the answer to the corresponding question.
We can apply these techniques to other kinds of problems, e.g., search problems, if we wanted.
P and NP

We say that an algorithm $A$ solves a decision problem $L$ if, given any instance of $L$ as input, $A$ outputs the correct answer to the corresponding question.

**Definition 2** We define $P$ to be the class of all decision problems that are solvable in polynomial time. That is, $P$ is the class of all decision problems $L$ for which there exists a constant $k$ and an algorithm that solves $L$ and runs in time $O(n^k)$, where $n$ is the size (in bits) of the input.

Thus $P$ is the class of all decision problems that are “easily decidable.” (“easy” = polynomial time; we don’t need any finer granularity)
Without loss of generality, we will assume that all algorithms must read their input sequentially, as if from a file on disk, say. Likewise, all outputs must be written sequentially (e.g., to a disk file). Thus reading input and writing output take time proportional to the size of each. This will simplify much of the discussion below.
P and NP, continued

A decision problem is in the class $\text{NP}$ if all its yes-instances can be easily verified, given the right extra information.

For example, if a graph $G$ does have a vertex cover $C$ of size $\leq K$, this fact can be verified easily if the actual set $C$ is presented as extra information (we simply check that each edge in $G$ is incident to a vertex in $C$).

Such extra information is called a proof or witness.

**Definition 3** $\text{NP}$ is the class of all decision problems $L$ for which there exists an algorithm $A$ that behaves as follows for all instances $x$ of $L$:
• $x$ is a yes-instance of $L$ if and only if there is a $y$ such that $A$ outputs “yes” on input $(x,y)$.

• $A$ runs in time polynomial in the length of $x$.

Such a $y$ (when it exists) is a witness, and $A$ is the algorithm that verifies, using the witness, that $x$ is a yes-instance of $L$.

Since $A$ must stop within time $O(n^k)$ for some constant $k$, it can only read the first $O(n^k)$ bits of $y$, where $n$ is the length of $x$. Thus we can limit the size of $y$ to be polynomial in $n$.

All the problems listed above are in $\text{NP}$. Also, it is clear that $\text{P} \subseteq \text{NP}$. 
Reductions

Definition 4 Given two decision problems $L_1$ and $L_2$, we say that $L_1$ polynomially reduces to $L_2$ ($L_1 \leq_p L_2$) if there is a function $f$ such that

- $f$ maps each instance of $L_1$ to an instance of $L_2$,
- $f$ can be computed in polynomial time, and
- for each instance $x$ of $L_1$,

\[ x \text{ is a yes-instance of } L_1 \iff f(x) \text{ is a yes-instance of } L_2. \]

$f$ is called a polynomial reduction of $L_1$ to $L_2$. 
This captures the notion that $L_1$ is “no harder than” $L_2$, or, $L_2$ is “at least as hard as” $L_1$. The $\leq^p$ relation is reflexive and transitive.

**Theorem 1 (Pretty easy)** Suppose $L_1 \leq^p L_2$. Then,

- if $L_2 \in \mathsf{P}$ then $L_1 \in \mathsf{P}$ and

- if $L_2 \in \mathsf{NP}$ then $L_1 \in \mathsf{NP}$.

**Definition 5** Two decision problems $L_1$ and $L_2$ are polynomially equivalent ($L_1 \equiv^p L_2$) if both $L_1 \leq^p L_2$ and $L_2 \leq^p L_1$. 
Proof of the first item: Suppose $A$ is a poly-
time algorithm solving $L_2$ and $f$ is a polynomial
reduction from $L_1$ to $L_2$. Then consider the
following algorithm that solves $L_1$:

1. Read as input an instance $x$ of $L_1$.

2. Compute $y = f(x)$.

3. Run algorithm $A$ on input $y$ and output the
   result.

This algo solves $L_1$ since the correct answer
for $x$ is the same as $A$’s answer for $f(x)$. Also
the algorithm runs in polynomial time: Let $n$
be the length of $x$. Item 1 takes time $O(n)$. Since $f$
runs in polynomial time, item (2) takes polynomial
time, and in addition, $y$ has length
polynomial in $n$. Thus item 3 runs the polynomial time algorithm $A$ on an input which has polynomial size. Since the composition of two polynomials is a polynomial, item 3 takes polynomial time (in $n$). Thus the whole algorithm takes time polynomial in $n$. 
NP-Hardness and NP-Completeness

Definition 6 A decision problem $L$ is NP-hard if, for every problem $L' \in \text{NP}$, we have $L' \leq_p L$.

Thus a problem is NP-hard iff it is at least as hard as any problem in NP.

Definition 7 A decision problem is NP-complete if it is in NP and it is NP-hard.

Theorem 2 (Easy) Any two NP-complete problems are polynomially equivalent.

Proof: If $L_1$ and $L_2$ are NP-complete, then in particular, $L_1 \in \text{NP}$ and everything in NP reduces to $L_2$. Thus $L_1 \leq_p L_2$. Likewise, $L_2 \leq_p L_1$.

All the problems listed above are NP-complete, and hence polynomially equivalent.
The Standard Technique

The standard technique that we will use for showing that a problem \( L \) is NP-complete takes two steps:

1. Show that \( L \in \text{NP} \). This is usually obvious.

2. Find a polynomial reduction of \( L' \) to \( L \) for some known NP-complete problem \( L' \).

Since all NP problems are reducible to \( L' \), and since the \( \leq^p \) relation is transitive, it follows that all NP problems are reducible to \( L \), and thus \( L \) is NP-complete.
Obviously, the first natural NP-complete problem could not be proved NP-complete using this method. The first such problem was

**SATISFIABILITY (SAT)**
Instance: A Boolean formula \( \varphi \).
Question: Is \( \varphi \) satisfiable, i.e., is there a setting of the Boolean variables of \( \varphi \) that makes \( \varphi \) true?

In the 1970s, Steve Cook (Waterloo, Ontario, Canada) and Leonid Levin (then in the USSR, now at Boston U.) independently came up with an ingenious proof that SAT is NP-complete.

**Theorem 3 (Cook, Levin) SAT is NP-complete.**

The Cook-Levin Theorem provides the starting point we need to use our technique. By now, there are hundreds (if not thousands) of known NP-complete problems to start from, and there is much variety (computer science, operations research, game playing, etc.).
CNF-SAT

Cook and Levin actually showed that the following restriction of SAT is NP-complete:

CNF-SAT
Instance: A Boolean formula \( \varphi \) in conjunctive normal form (CNF).
Question: Is \( \varphi \) satisfiable?

A formula is in CNF if it is a conjunction

\[ C_1 \land \cdots \land C_n \]

of clauses \( C_i \), where a clause is defined as a disjunction

\[ (l_1 \lor \cdots \lor l_m) \]

of literals \( l_j \), and where a literal is defined as either a Boolean variable (e.g., \( x \)) or the negation of a Boolean variable (e.g., \( \neg x \)).
CNF-SAT may appear easier than SAT at first, since we only need to worry about formulas in CNF instead of arbitrary formulas. But CNF-SAT is NP-complete, so it is polynomially equivalent to SAT.
3-SAT

We can restrict CNF-SAT even further:

3-SAT
Instance: A Boolean formula $\varphi$ in CNF where each clause in $\varphi$ has at most three literals.
Question: Is $\varphi$ satisfiable?

3-SAT is NP-complete, and we can prove this using our standard technique by finding a polynomial reduction from CNF-SAT to 3-SAT.

Interesting fact: If we restrict $\varphi$ to have at most two literals per clause, the resulting problem, 2-SAT, can be solved in polynomial time.