Notes on NP-completeness

Definition 1. An algorithm runs in *polynomial time* if it runs in $O(n^k)$ for some fixed k where n is the size of the input.

A *decision problem* is a problem where the answer (output) is always "yes" or "no" (depending on the input).

A decision problem is *NP-complete* if it is both *in NP* and *NP-hard*.

NP stands for *Non-deterministic Polynomial time*.

There are many equivalent ways to define problems in NP.

One way is to say that if the output is "yes" then there is a *certificate* so that given the original input and this certificate, there is a polynomial time algorithm (polynomial in the size of the original input) which can verify that the output should indeed be "yes".

Another equivalent way of defining problems in NP is to say that a problem is in NP if there is a polynomial time algorithm to solve this problem given a special computer. This special computer has all the functions of an ordinary computer plus the ability to clone itself (and thus doubling its processing power) which setting some variable to different values in this cloning process. Clones are allowed to clone themselves again. The output from such a computer is taken to be the output of the first clone to give an output. Clones cannot communicate with each other.

A problem is *NP-hard* if being able to solve this problem in polynomial time implies that all problems in NP can be solved in polynomial time.

A problem is *in P* if it can be solved in polynomial time.

Remark 1. *Technically, we should be talking about* Turing machines *rather than "computers" in the above definition but the two are close enough for the purpose of this discussion.*

One of the biggest open problems in computer science consists of determining whether $P = NP$. That is, determine if there is an NP-hard problem which can be solving in polynomial time. Of course, $P \subseteq NP$ is an easy consequence of the definitions.

Typically, a problem is shown to be NP-hard by *reducing* another NP-hard problem to it. That is, we use this algorithm as a black box (with the only guarantee that it runs in polynomial time) to solve the problem we are reducing from in polynomial time.

But this approach requires that we have a NP-hard problem in the first place (to reduce from). This was not emphasized in class but the first problem shown to be NP-complete is *satisfiability* (abbreviated SAT). This result is known as the Cook-Levin theorem.

Satisfiability is a problem which resembles the following problem briefly mentioned in class.

Problem 1. Given an input proposition P, is P not a contradiction?

The negation in the formulation of the problem is to guarantee that there is a certificate when the answer is "yes".

One reason why we do not have an exactly characterisation ("if and only if" type statement) for graphs with Hamiltonian cycles is because the following problem is NP-complete.

Problem 2. Given an input graph G, does G have a Hamiltonian cycle?

Determining if a graph has a Hamiltonian path is also NP-complete.

P and NP are called *complexity classes*. There are many more complexity classes that a problem can belong to. See `http://qwiki.stanford.edu/wiki/Complexity_Zoo` for example (although the more complex classes are rarely encountered).