

Study guide on NP-completeness and on studying mathematical texts

CS301, Spring '09

Optional; any final exam questions on NP-completeness will be extra credit. A list of four problems, two of which will appear on the exam, are given at the end of this posting. You must look in the handout from 5/5 for the answers. This study guide is to help you make sense of the handout.

1. Let us first briefly examine the connection between nondeterminism in connection with the class NP.

Nondeterminism is a model of “limited omniscience,” where, when given a choice of examining two possibilities, one of which may turn out to be a dud, you always know which one to choose.

Unlike the case of pushdown automata, and like the case of finite state machines, nondeterministic Turing machines to accept no more languages than deterministic Turing machines accept.

We also discussed in class why deterministic Turing machines can solve a problem in polynomial time if and only if it can be solved in polynomial time on our RAM-model idealization of a modern computer.

One thing that nondeterminism does buy us is the ability to recognize words in a language in polynomial time, when they might not be recognizable in polynomial time on a deterministic Turing machine or modern computer architecture.

An example is given by the Hamiltonian cycle problem in Figure 34.2 of the in-class handout from 5/5. Once you think of a data structure for a graph and enter it into the computer’s memory, what results is a encoding of the graph as a string on the language $\{0, 1\}^*$. This is because that’s how a modern computer represents all data. Similarly, if you represent it as the input to a Turing machine, you must think of some way to encode it in the alphabet of the Turing machine.

Once you have established your encoding method, the language of encodings of Hamiltonian graphs is a language. Nobody knows a deterministic polynomial-time algorithm for recognizing strings in this language. However, there is a nondeterministic Turing machine that does this: it “guesses” a Hamiltonian cycle for the graph. Since a Hamiltonian cycle has n edges, generating one takes a polynomial number of moves (and some luck). To ensure that the machine only accepts Hamiltonian graphs, it must also contain a deterministic test of whether the guess is a legitimate hamiltonian cycle. This test can obviously be performed in polynomial time on a modern computer, and therefore on a deterministic Turing machine.

Whenever a graph is Hamiltonian, there exists a computation that gets the machine to accept, and whenever it isn’t, there doesn’t exist any computation that gets the machine to accept, since no guess will pass the test of being a Hamiltonian cycle.

The class NP of languages is the class that is accepted in a polynomial number of moves by a nondeterministic Turing machine.

According to this definition, the encodings of Hamiltonian graphs are in the class NP.

2. Notice that Section 34.2 of the handout barely mentions nondeterminism. This is because it couches the issue slightly differently. It talks about **decision problems**, where the answer is either yes or no. (In terms of languages and Turing machines, a yes answer corresponds to the encoding being in the language.)

A **certificate** is a proof of a yes answer that can be checked deterministically in polynomial time. (In terms of languages and Turing machines, the certificate corresponds to the “lucky guess.”)

The **class NP of decision problems** are those problems for which, whenever the answer is yes, there exists a certificate of this that can be checked in polynomial time.

3. I want to show you that even though this is a graduate text with a lot of material that may seem inaccessible at first, no math text is *completely* inaccessible. The trick is to identify and zero in on some parts that you can understand

Recall the problem INDEPENDENT-SET, which I talked about on 5/5. A set of vertices of a graph is **independent** if no two of them share an edge. INDEPENDENT-SET is the problem, where, given a graph G and an integer k , you must decide whether there exists a set of k independent vertices in G .

When I asked why this problem is in NP, some of you immediately remarked that if the answer is yes, then an independent set of k vertices is a polynomial certificate of this, since it takes polynomial time to check that the set has size k and that it is independent in the graph.

Look at the list of problems in Figure 34.13 on page 1004. Embedded in the text are specifications of these problems. Understanding a problem specification is an important skill for computer scientists. It should be kept separate from considerations of how the problems are solved.

Find the places in the text where the problems are specified, and understand the specifications of the problems. Figure out what issues discussed in the text are not part of the specification, and ignore them. Then try to figure out why the problem is in NP, by figuring out what kind of certificate always exists when the answer is yes and that can allow you to verify this answer in polynomial time.

Examples:

- CIRCUIT-SAT: A little skimming for bold print reveals that this is defined on page 989. *The **circuit-satisfiability problem** is, ‘Given a boolean combinational circuit composed of AND, OR, and NOT gates, is it satisfiable?’* What it means for it to be satisfiable is given in bold print on the previous page. Details about what a boolean combinational circuit consists of are also given there, though you have probably seen it in previous classes.

The problem is in NP, since if somebody tells you what boolean values to put on the input wires to cause a TRUE to appear on the output wire, you can check in polynomial time that they are telling the truth. Such an assignment is a certificate that always exists whenever the answer to the question is *yes*.

- CLIQUE: A little skimming for bold print reveals that this is defined on page 1003.

- VERTEX-COVER: page 1006. A mnemonic device is to think of the edges as streets in a bad neighborhood, the vertices as intersections, and a vertex cover as a placement of policemen such that every street has a cop at at least one end.
 - SUBSET-SUM: page 1013. A mnemonic device is to think of the numbers as the weights of items you can put in your truck and t as the weight limit of your truck.
4. We have proven that in the whole infinite world of algorithms, undiscovered ones as well as discovered ones, there isn't a single algorithm that solves the halting problem.

We also know that for every problem in NP, there is a nondeterministic Turing machine that recognizes when the answer is *yes* in a polynomial number of moves. The biggest open question in computer science is whether there is a *deterministic* one? In other words, is there a polynomial algorithm that determines this?

If there is, then there is a polynomial algorithm that *solves* the problem in polynomial time. It runs the deterministic algorithm that recognizes when the answer is *yes* in polynomial time, and declares that the answer is no if it fails to halt within this time bound.

The class P is the set of decision problems that can be solved in polynomial time by a deterministic algorithm. This big open question can therefore be rephrased as the question of whether $P = NP$.

What reason is there to think that it might be the case that $P = NP$? An astonishing result is that a polynomial algorithm for CIRCUIT-SAT would imply that $P = NP$! **Any polynomial-time algorithm for solving the CIRCUIT-SAT problem is as powerful any polynomial-time deterministic Turing machine that solves any problem in NP.**

The proof is complicated, and we won't cover it in detail. The intuition behind it is that a combinational circuit has the computing power of a modern computer. For every instance of a problem in NP, you can construct a combinational circuit in polynomial time that implements the verification of any potential certificate, outputting TRUE if the certificate checks out. The answer to the problem is *yes* if and only if there exists a certificate that checks out, in other words, if and only if the answer to the problem in NP is *yes*.

Terminology: This is what it means for CIRCUIT-SAT to be **NP-complete**.

This astonishing result was discovered by Stephen Cook in 1971, who actually showed a slight variant of this, which is that SAT is NP-complete.

5. Look again at Figure 34.13. This reflects a roadmap of subsequent work by Richard Karp and other people that uses the fact that there exists one NP-complete problem to show that there are many other NP-complete problems. The map depicts that there is a proof that any polynomial deterministic algorithm for SAT, if one exists, is at least as powerful as one for CIRCUIT-SAT, and therefore, that SAT is NP-complete. Similarly, one for 3-CNF-SAT is at least as powerful as one for SAT, so 3-CNF-SAT is NP-complete.

Similarly, one for CLIQUE or SUBSET-SUM is at least as powerful for as one for 3-CNF-SAT, so CLIQUE and SUBSET-SUM is NP-complete, etc.

6. **Who cares?** Suppose you have a set of items of different weights at a loading dock and you would like to select a set of items that maximizes your load, subject to the constraint that you not go over a certain limit, t . Let's call this the *trucker problem*.

Theorem: *If $P \neq NP$, then there is no polynomial-time algorithm for our trucker problem in the whole infinite world of discovered and undiscovered algorithms.*

Proof: *Suppose $P \neq NP$, but there is a polynomial algorithm for our trucker problem. We can use it to solve SUBSET-SUM in polynomial time by asking our trucker algorithm whether we can actually reach our target of t . That gives a polynomial-time algorithm for SUBSET-SUM. Since SUBSET-SUM is NP-complete, $P = NP$, a contradiction.*

Because of Cook's theorem, many thousands of NP-complete problems are now known. There are many thousands of industrial optimization problems of enormous economic value, for which we can show that, unless $P = NP$, there exists no polynomial algorithm for them. The proofs are similar to the one about our trucker problem.

Billions of dollars are saved every year when programmers realize that a polynomial-time algorithm for their industrial problem would imply that $P = NP$. A hurdle to solving their problem is becoming the most famous computer scientist who ever walked the planet. A prudent course of action is not to waste their employer's money looking for one. There are heuristic methods that can be used instead to get a suboptimal solution that's likely to be reasonably good.

7. **How does one show that a new problem is NP-complete?**

Suppose you know that CLIQUE is NP-complete? Why does this imply that INDEPENDENT-SET is NP-complete? Suppose there is a polynomial algorithm for INDEPENDENT-SET. We can trick this algorithm into solving any instance of CLIQUE in polynomial time, as follows. Replace every edge with a non-edge and every non-edge with an edge. This takes polynomial time. There is an independent set of size k in the new graph if and only if there is a clique of size k in the original graph. We can answer this question in polynomial time with our hypothetical polynomial algorithm for INDEPENDENT-SET in polynomial time. Therefore, our hypothetical algorithm for INDEPENDENT-SET would imply $P = NP$. INDEPENDENT-SET is NP-complete!

This is known as a **reduction** of CLIQUE to INDEPENDENT-SET. The general strategy for showing new problems are NP-complete is always a variant of this one: take a known NP-complete problem and reduce it to the new one in polynomial time.

8. **Things to figure out for the extra credit.**

The road map depicted in Figure 34.13 shows how all the problems in the picture are shown to be NP-complete in the text. CIRCUIT-SAT was shown to be NP-complete by a variant of the proof of Cook's theorem. SAT was shown to be NP-complete by reducing CIRCUIT-SAT to it. 3-CNF-SAT was shown to be NP-complete by reducing SAT to it. CLIQUE and SUBSET-SUM were shown to be NP-complete by reducing 3-CNF-SAT to them, etc.

Some of these reductions are almost as easy as the reduction of CLIQUE to INDEPENDENT-SET, above. The easiest ones are CLIQUE to VERTEX-COVER (Figure 34.15), 3-CNF-SAT to CLIQUE (Figure 34.14), and HAM-CYCLE to TSP (page 1013). One of

the extra-credit problems will be about whether you understood how these reductions work.

A more subtle reduction is the one from 3-CNF-SAT to SUBSET-SUM (Figure 34.19). This one is a little harder, but an extra credit problem about this one *will* appear on the exam.