Lecture 22: NP problems

The concept of NP is important: a problem is in NP if this problem can be verifiable in polynomial-time. Note: we are not asking for efficient finding the solution, but rather just verifying whether a proposed solution is legal or not.

In the last lecture, we saw two NP problems: graph 3-coloring and the HAM-CYCLE problem. Our next problem is the satisfiability problem. We are given a boolean formula in so-called Conjunctive normal form (CNF). A CNF formula contains several clauses (connected by logical and), where in each clause we apply logical or to connect several literals. Literal means a boolean variable or its negation \( x \) or \( \neg x \).

Example: \[ \Phi = (x \lor y \lor z) \land (x \lor \neg y) \land (y \lor \neg z) \land (z \lor \neg x) \land (\neg x \lor \neg y \lor \neg z) \]

Satisfying truth assignment: a setting of boolean variables s.t. each clause is evaluated to be true (i.e. for each clause, at least one literal is true). In class, I explained that this formula does not have satisfying truth assignment. Study this if you forget about the argument. The satisfiability problem asks whether there exists satisfying truth assignment for a given formula. Again, no efficient algorithm is known for satisfiability (SAT) problem. But verifying a proposed truth assignment can be easily done in polynomial time.

Our next problem is the traveling salesman problem. We are given \( n \) cities, where we can go from any cite \( i \) to any city \( j \) by traveling \( d(i,j) \) miles. Our goal is to find a tour (cycle) of cities so that the total distance of the tour is minimized. This is an optimization problem, which is slightly different from a search problem (which just find a legal solution). To make verification easier, we re-phrase the problem: can we find a tour that costs no more than \( b \) (some budget limit)? This becomes a search problem. Often if we can solve the search problem we will be able to solve the optimization problem as well. Think about what this means. Caution: we need to carefully choose how to frame the search problem. For example, we can not use the following for the TSP problem: can we find a tour that costs no smaller than \( b \)? This will allow trivial solution like any way of ordering the cities.

Our next example is the independent set problem. Given a graph, find the largest subset of nodes of \( V \) s.t. no two nodes in the set are connected by an edge of \( G \). Again, we can convert this to a search problem, and it is easy to see the search problem is in NP. Now we look at the vertex cover problem. Given graph \( G \), does there exist \( k \) nodes in \( G \) s.t. each edge in \( G \) is touched (covered) by one of the \( k \) nodes. This again, is in NP.

Now is one of the most important concepts in this chapter: polynomial-time reduction. Suppose we have a black box efficient code that can solve problem \( X \), and we can use it to solve problem \( Y \). Which is harder: \( X \) or \( Y \)? Since \( X \) lets solve \( Y \), so \( X \) is at least as hard as \( Y \). More precisely: can we solve any instance of \( Y \) (say a graph) using poly-time to prepare input and read output for black box for \( X \) and make polynomial number of calls to the black box of \( X \)? If yes, we say \( Y \) is poly-time reducible to \( X \), denoted \( s \ Y \leq_p X \). Consequence: if \( X \) does have a poly-time algorithm, then \( Y \) also has a polynomial time algorithm. Conversely, if \( Y \leq_p X \), and \( Y \) can not be solved in poly-time, then \( X \) can not be solved in poly-time either.

Lecture 23: NP-completeness.

We now show our first reduction for two problems in NP: independent set (IS) problem and vertex cover (VC) problem. Note we will use IS to denote both the IS problem and some independent set for a graph. Same usage of VC. We will show \( IS \leq_p VC \) and \( VC \leq_p IS \). Here is a useful observation. Given a graph \( G = (V, E) \), a subset \( S \) is an IS of \( G \) if \( V - S \) is a VC. This can be easily verified by the definition of IS and VC. Check this out if you do not understand it. Thus, given a black box code for the VC problem, we will find solution for the following question on the IS problem: given a graph \( G = (V, E) \), does \( G \) have an IS with \( k \) nodes or more? Based on our previous observation, we note that this question can be easily answered by passing this question to the VC black box: does \( G \) (the same graph) contain a VC with \( n - k \) nodes or
smaller? Here \( n = |V| \). Clearly the reduction is done in polynomial-time: we actually just make one call of the black box and the cost of preparing the input is none. I omit the other reduction, which is similar.

NP-complete: the hardest problems in NP. A problem is NP-complete if (1) it is in NP and (2) every problem in NP is polynomial-time reducible to it. In class, we went over the implication of NP completeness (1) if problem \( X \) is NP complete, and \( X \) has a polynomial time algorithm, then every (yes every) problem in NP has polynomial-time algorithm. (2) if any problem in NP has no polynomial time algorithm, the each NP complete problem has no polynomial algorithm.

Does NP complete problem exist? The 1971 Cook-Levin theorem says yes: circuit satisfiability problem (CIRUIT-SAT) is NP complete. Our textbook has a nice introduction to CIRUIT-SAT. Read it if you still do not understand it.

From now on, we have one starting problem: CIRUIT-SAT. Now how to prove problem \( X \) is NP complete? The recipe for NP-completeness: (1) show \( X \) is in NP (2) Find a known NP complete problem \( Y \), (3) describe how to convert instances for \( Y \) into an instance for \( X \) (4) Prove the instance for \( Y \) can be efficient solved iff \( X \) can be solved. (5) Ensure the conversion can be done in poly-time.

We then went over the proof of 3-CNF-SAT (or simply 3SAT) is NP complete (where each clause has exactly 3 literals). Read the textbook for more details. But here is the important points. We need to first show 3SAT is in NP. Then, we will reduce from CIRUIT-SAT. The basic idea is to construct equivalent formula for each of three types of logical gates in the circuits. Read the textbook for more details. Finally, some of reductions will lead to clauses with only two literals. But in class, I showed how to convert such clauses into equivalent clauses with 3 literals each. Again, read the book if you still miss something.