CSE 347 Lecture 14

Intractability and NP-completeness

1. Gadget reductions: 3-SAT reduces to Independent Set
2. NP: verifiable problems
3. NP-completeness

Problems that appear very different may be very closely related. For example, last time we saw the Independent Set problem: given a graph $G = (V, E)$ and an integer k, does there exist a set $S \subseteq V$ such that no pair $u, v \in S$ is joined by an edge (an "independent set") containing at least k vertices? We also saw the 3-SAT problem: given a Boolean formula that is an AND of ORs of at most three literals/terms (variables or negated variables) — that is, a "3-CNF" — is there a setting of the variables that makes the formula true (a "satisfying assignment")? These problems seem to have nothing in common, and yet, we will see that they are very closely related.

Theorem: $3\text{-SAT} \leq_p \text{Independent Set}$.

Idea: We will build graph structures in the Independent Set problem that correspond to the pieces of a 3-SAT instance; in particular, these "gadgets" will force any independent set to obey constraints that are analogous to those imposed by the formula on its satisfying assignments. The reduction is then merely a matter

of "translating" the 3-CNF into an appropriately assembled graph. This is known as a "gadget reduction."

In particular, we take the perspective that the 3-CNF is simply demanding that in each clause (OR), we can identify a literal that our assignment makes true; as long as we are consistent across the clauses, and never commit the same variable to be both true and false at the same time, we'll be able to recover a combined, global assignment that makes the formula true. Supposing that these satisfied literals are translated into vertices in the independent set problem, this key "mutual exclusion" constraint can be translated into an edge joining a variable and its negations in other parts of the formula. More carefully, now:

Proof: Consider the following algorithm: on input a CNF $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, for each C_i and literal ℓ_j in C_i, we create a vertex in G; we add edges between each ℓ_j in C_i, and an edge between ℓ_j in C_i and any $\overline{\ell_j}$ in $C_{i'}$. We call IndependentSet(G, m) and return its answer.

Observe that adding the edges takes time $O(m^2)$, which dominates the running time. The algorithm is correct because? if the formula is satisfiable, we can fix any satisfying assignment $x \in \{0,1\}^n$, and add a vertex corresponding to one of the satisfied literals

in each C_i to S. This gives an S of size m, since we can only choose satisfied literals, we don't choose both x and $\overline{x}$ (so these edges don't connect vertices in S) and we only choose one vertex from each clause, so the intra-clause edges also don't join vertices in S. Thus, S is an independent set of size m, and thus IndependentSet$(G,m)=1$. Conversely, if G has an independent set S of size m, the intra-clause edges ensure that we choose one literal l from each clause, and the edges between x and $\overline{x}$ ensure that we only choose one "setting" for each variable. Therefore, an assignment to the variables that satisfies the chosen literals (say, false on the variables that weren't chosen anywhere) satisfies every clause, so φ is satisfiable, and returning 1 is correct. ▨

NP: verifiable problems

We don't have efficient algorithms for any of these problems, but there is still one feature that limits their potential difficulty: we can check candidate "solutions" — e.g., independent sets, vertex covers, satisfying assignments — pretty easily. This enables, at a minimum, the design of "brute-force search" algorithms that iterate over candidate solutions and check them one by one. (Without verification algorithms, this "check" part is impossible.) More formally:

Definition: A Boolean (decision) problem X is in **NP** if the following kind of **certifier** algorithm B exists: On input a pair (x, c), B runs in time bounded by a polynomial in the length of its input, and returns a Boolean answer; the answer of X on x is "1" if and only if there is a polynomial-size c such that B returns "1" on (x, c). (c is a "certificate" or "proof")

Note that for our problems, B respectively...

- Interprets c as a set $S \subseteq V$, and checks that no edges exist between u,v ∈ S (in time $|V|^2$), and that it has ≥ k vertices, in time $O(|V|^2)$ overall
- Vertex-cover? c encodes a set $S \subseteq V$ and checks that every edge has some endpoint in S (in time $|E||V|$) and has size ≤ k, in time $O(|E||V|)$ overall.
- 3-SAT? c lists a Boolean setting for each variable, and we evaluate the input 3-CNF on this assignment to see that it is true, in time $O(m)$ overall.

There are many more NP problems - for example, Integer Factoring: is there a number in the range [2,k] that divides N? This nontrivial factor d itself is a certificate, since we can divide N by d to see if there is zero remainder. (We can use binary search on k to identify the smallest divisor d exactly, as before). Mathematicians have searched for centuries for a good algorithm for factoring, but still none is known.

So, some NP problems seem to be quite hard. Note also that most engineering problems are of the form, "find a design that meets the following requirements ...", where these requirements can be verified by an efficient algorithm. These problems also have equivalent "decision" formulations in NP.

The "P vs. NP" question asks if there are polynomial time algorithms for every problem in NP. It is very natural to conjecture "no," as some famously hard problems are in NP, and algorithms that solve all engineering tasks seems too fantastic to be possible. But, we still really don't know.

NP-completeness

The P vs. NP question is clearly very important, but probably seems very remote and disconnected from our immediate concerns. But, this is not so: the P vs. NP question is actually equivalent to the question of whether or not specific, natural problems have efficient algorithms. Precisely:

Definition: A computational problem X is NP-complete if X is in NP and for every other problem Y in NP, $Y \leq_p X$.

So, an NP-complete X is related, like our example of 3SAT and Independent Set, to a vast family

of seemingly unrelated problems. It is at least as hard as all of the many famously hard NP problems, and, a truly efficient algorithm for any such problem would allow us to replace most engineering effort with computer time. So, although we do not know for certain whether or not NP-complete problems have efficient algorithms it seems unlikely.

It's natural to wonder, at this point, how an NP-complete problem can exist. The most intuitive example is Circuit-SAT: Given an acyclic Boolean (AND/OR/NOT) circuit with one output wire, is there a setting of the input wires that causes the output to be "1"? This problem is in NP since? We can take c as encoding an input to the circuit, and then evaluate the circuit to see if it outputs "1".

Cook-Levin Theorem: Circuit-SAT is NP-complete.

We won't be able to prove this theorem in this class, but here is the idea:

1. If X is in NP, it has a certifier algorithm B.
2. We can implement the algorithm B in a circuit. Actually, more precisely, there is an algorithm that takes as input an integer n, and prints out a circuit that computes B on x of size n (that runs in poly. time)
3. The reduction generates this circuit, and hard-wires in the input x of interest. Then the circuit

is satisfiable if and only if there is a c that B would accept with x — if and only if X outputs 1 on x. So, Circuit SAT on this circuit gives the answer for X on x. Thus, we find $X \leq_p$ Circuit SAT for any X in NP. "□"

(What's the problem? Step 2, actually — it requires us to show that <u>any</u> algorithm B can be implemented by a small, easy-to-generate circuit. This is not hard, but it is tedious and requires a concrete programming model. CSE 547† contains the details...)