CMSC 451: Lecture 16
NP-Completeness: Reductions

Recap: We have introduced a number of concepts on the way to defining NP-completeness:

Decision Problems/Language recognition: Problems for which the answer is either yes or no. These can also be thought of as language recognition problems, assuming that the input has been encoded as a string. For example:

$$\begin{aligned} \text{HC} &= \{G \mid G \text{ has a Hamiltonian cycle}\} \\ \text{MST} &= \{(G, z) \mid G \text{ has a spanning tree of weight at most } z\}. \end{aligned}$$

Recall that a *Hamiltonian cycle* is a simple cycle (does not repeat vertices) that visits every vertex of the graph.

P: Class of all languages (equivalently, decision problems) which can be solved in (worst-case, deterministic) polynomial time. We know that MST $\in$ P, but we do not know whether HC $\in$ P (but we suspect not).

Verification: A problem L is *verifiable* in polynomial time if whenever $x \in L$ (that is, x is a "yes" instance for the decision problem) it is possible to prove this in polynomial time, given the assistance of a *certificate*. For example, the language HC above is verifiable. The certificate consists of a sequence of vertices forming the cycle. In polynomial time, we can check that this cycle visits all the vertices of the graph exactly once. (If $x \notin L$, we don't care about the result of verification.)

NP: Class of all languages that can be *verified* in polynomial time. (Formally, it stands for "*Nondeterministic Polynomial time*", since coming up with a certificate can be viewed as a nondeterministic computation.) Clearly, P $\subseteq$ NP, and it is widely believed (but not known) that P $\neq$ NP.

In this lecture, we will discuss the concept of *NP-completeness*. These are, in some sense, the hardest problems in NP. Before defining NP-completeness, we need to introduce the concept of a reduction.

Reductions: The class of NP-complete problems consists of a set of decision problems (languages) (a subset of the class NP) that no one knows how to solve efficiently, but if there were a polynomial time solution for even a single NP-complete problem, then every problem in NP would be solvable in polynomial time.

Before discussing reductions, let us just consider the following question. Suppose that there are two problems, H and U. We know (or you strongly believe at least) that H is *hard*, that is it cannot be solved in polynomial time. On the other hand, the complexity of U is *unknown*. We want to prove that U is also hard. How would we do this? Effectively, we want to show that

$$(H \notin \text{P}) \Rightarrow (U \notin \text{P}).$$

To show that U is not solvable in polynomial time, we will suppose (towards a contradiction) that a polynomial time algorithm for U did existed, and then we will use this algorithm to

solve H in polynomial time, thus yielding a contradiction. In other words, we could prove the contrapositive,

$$(U \in \mathrm{P}) \;\Rightarrow\; (H \in \mathrm{P}).$$

To make this more concrete, suppose that we had a subroutine[1] that can solve any instance of problem U in polynomial time. Given an input x for the problem H, we could translate it into an *equivalent* input x' for U. By "equivalent" we mean that $x \in H$ if and only if $x' \in U$ (see Fig. 1). Then we run our U subroutine on x' and output whatever it outputs.

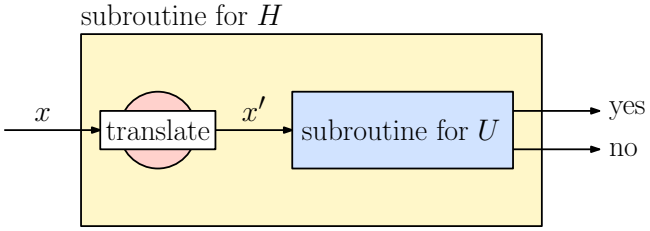

Fig. 1: Reducing H to U.

It is easy to see that if U is solvable in polynomial time, then so is H. We assume that the translation module runs in polynomial time. If so, we call this a *polynomial reduction* of problem H to problem U, which is denoted $H \leq_P U$. Richard Karp used this style of reduction in his influential paper on NP-completeness, and for this reason, it is called a *Karp reduction*.

More generally, we might consider calling the subroutine multiple times. This is called a *Cook reduction*, after Stephen Cook. While Cook reductions are theoretically more powerful, Karp reductions are simpler and work for virtually all NP-completeness proofs.

3-Colorability and Clique Cover: Let us consider an example to make this clearer. The following problem is well-known to be NP-complete, and hence it is strongly believed that the problem cannot be solved in polynomial time.

3-coloring (3Col): Given a graph G, can each of its vertices be labeled with one of three different "colors", such that no two adjacent vertices have the same label (see Fig. 2(a) and (b)).

Coloring arises in various partitioning problems. (E.g., You are arranging your relatives to sit at three big tables during a wedding reception. You have pairs that don't get along, each represented by an edge in your graph. Can you assign them to three tables avoiding warring pairs at the same table?) It is well known that planar graphs can be colored with four colors, and there exists a polynomial time algorithm for doing this. But determining whether three colors are possible (even for planar graphs) seems to be hard, and there is no known polynomial time algorithm.

[1] It is important to note here that this supposed subroutine for U is a *fantasy*. We know (or strongly believe) that H cannot be solved in polynomial time, thus we are essentially proving that such a subroutine cannot exist, implying that U cannot be solved in polynomial time.

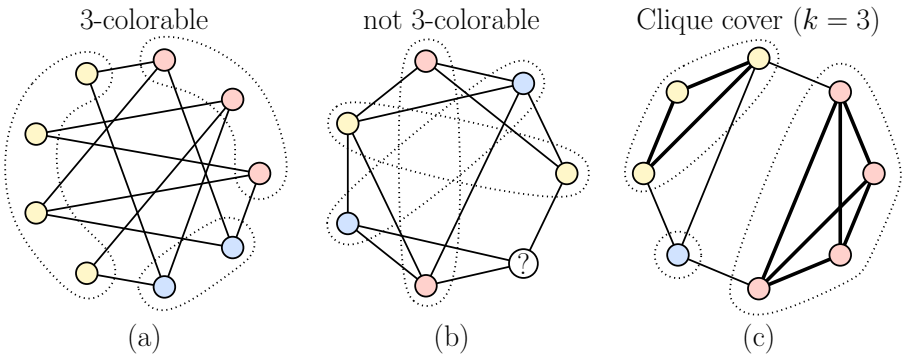

Fig. 2: 3-coloring and Clique Cover.

The 3Col problem will play the role of the known hard problem H. To play the role of U, consider the following problem. Given a graph $G = (V, E)$, we say that a subset of vertices $V' \subseteq V$ forms a *clique* if for every pair of distinct vertices $u, v \in V'$ $(u, v) \in E$. That is, the subgraph induced by V' is a complete graph.

Clique Cover (CCov): Given a graph $G = (V, E)$ and an integer k, can we partition the vertex set into k subsets of vertices $V_1, \ldots, V_k$ such that each V_i is a clique of G (see Fig. 2(c)).

The clique cover problem arises in clustering. We put an edge between two nodes if they are similar enough to be clustered in the same group. We want to know whether it is possible to cluster all the vertices into at most k groups.

We want to prove that CCov is hard, under the assumption that 3Col is hard, that is,

$$(\text{3Col} \notin \text{P}) \implies (\text{CCov} \notin \text{P}).$$

Again, we'll prove the contrapositive:

$$(\text{CCov} \in \text{P}) \implies (\text{3Col} \in \text{P}).$$

Let us assume that we have access to a polynomial time subroutine $\text{CCov}(G, k)$. Given a graph G and an integer k, this subroutine returns true (or "yes") if G has a clique cover of size k and false otherwise. How can we use this *alleged* subroutine to solve the well-known hard 3Col problem? We need to find a translation, that maps an instance G for 3-coloring into an instance (G', k) for clique cover (see Fig. 3).

Observe that both problems involve partitioning the vertices up into groups. There are two differences. First, in the 3-coloring problem, the number of groups is fixed at three. In the Clique Cover problem, the number is given as an input. Second, in the 3-coloring problem, in order for two vertices to be in the same group they should *not* have an edge between them. In the Clique Cover problem, for two vertices to be in the same group, they *must* have an edge between them. Our translation therefore, should convert edges into non-edges and vice versa.

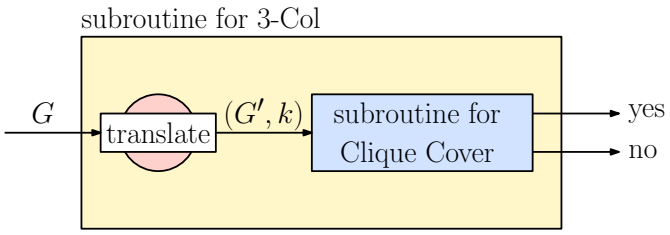

Fig. 3: Reducing 3Col to CCov.

This suggests the following idea for reducing the 3-coloring problem to the Clique Cover problem. Given a graph G, let $\overline{G}$ denote the *complement graph*, where two distinct nodes are connected by an edge if and only if they are not adjacent in G. Let G be the graph for which we want to determine its 3-colorability. The translator outputs the pair $(\overline{G}, 3)$. We then feed the pair $(G', k) = (\overline{G}, 3)$ into a subroutine for clique cover (see Fig. 4).

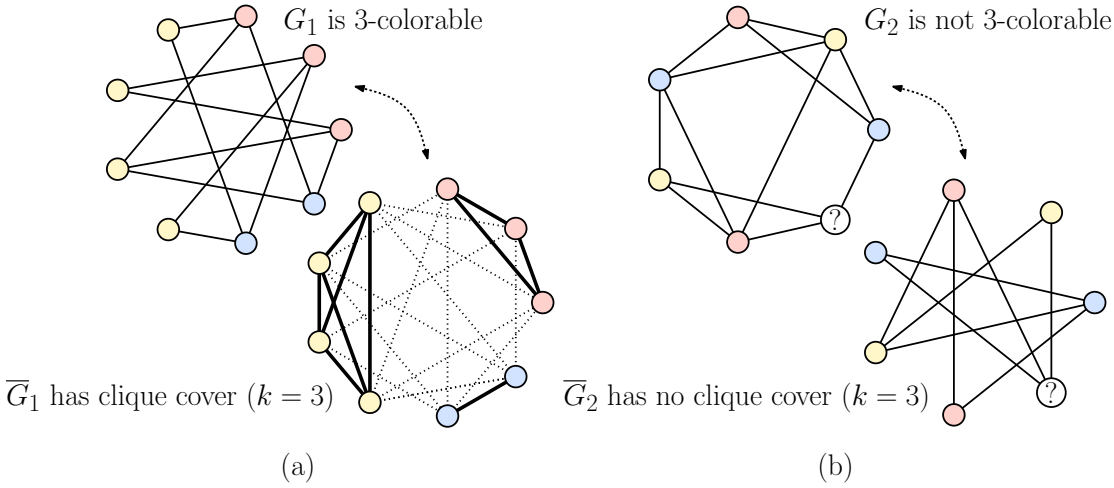

Fig. 4: Clique covers in the complement.

The following formally establishes the correctness of this reduction by showing that we have *faithfully* translated an instance of 3Col to an equivalent instance of CCov.

Claim: A graph $G = (V, E)$ is 3-colorable if and only if its complement $\overline{G} = (V, \overline{E})$ has a clique-cover of size 3. In other words,

$$G \in \text{3Col} \quad \Longleftrightarrow \quad (\overline{G}, 3) \in \text{CCov}.$$

Proof: ($\Rightarrow$) If G 3-colorable, then let V_1, V_2, V_3 be the three color classes. We claim that this is a clique cover of size 3 for $\overline{G}$, since if u and v are distinct vertices in V_i, then $\{u, v\} \notin E$ (since adjacent vertices cannot have the same color) which implies that $\{u, v\} \in \overline{E}$. Thus every pair of distinct vertices in V_i are adjacent in $\overline{G}$.

($\Leftarrow$) Suppose $\overline{G}$ has a clique cover of size 3, denoted V_1, V_2, V_3. For $i \in \{1, 2, 3\}$ give the vertices of V_i color i. We assert that this is a legal coloring for G, since if distinct

vertices u and v are both in V_i, then $\{u, v\} \in \overline{E}$ (since they are in a common clique), implying that $\{u, v\} \notin E$. Hence, two vertices with the same color are not adjacent.

It is useful to observe that the reduction was from 3Col to CCov, which means that we are at liberty to use any value of k we like, and $k = 3$ was convenient. You might wonder why we didn't need to consider other values of k. The reason is that we didn't need to. Of course, if we were trying to do the reduction in the opposite direction from CCov to 3Col, we would need to worry about this.

Polynomial-time reduction: We now take this intuition of reducing one problem to another through the use of a subroutine call, and place it on more formal footing. Notice that in the example above, we converted an instance of the 3-coloring problem (G) into an equivalent instance of the Clique Cover problem ($\overline{G}, 3$).

Definition: We say that a language (i.e. decision problem) L_1 is *polynomial-time reducible* to language L_2 (written $L_1 \leq_P L_2$) if there is a polynomial time computable function f, such that for all x, $x \in L_1$ if and only if $f(x) \in L_2$.

In the previous example we showed that 3Col $\leq_P$ CCov, and in particular, $f(G) = (\overline{G}, 3)$. Note that it is easy to complement a graph in $O(n^2)$ (i.e. polynomial) time (e.g. flip 0's and 1's in the adjacency matrix). Thus f is computable in polynomial time.

Intuitively, saying that $L_1 \leq_P L_2$ means that "if L_2 is solvable in polynomial time, then so is L_1." This is because a polynomial time subroutine for L_2 could be applied to $f(x)$ to determine whether $f(x) \in L_2$, or equivalently whether $x \in L_1$. Thus, in sense of polynomial time computability, L_1 is "no harder" than L_2.

You shouldn't read too much into the notation or make incorrect inferences. For example, this does not imply that L_1 is necessarily easier in the sense that its running time is smaller. It may very well be that $L_1 \leq_P L_2$ and L_1 takes $O(n^{10})$ time to compute while L_2 takes only $O(n)$ time. It could be that L_1 is solvable in polynomial time, but L_2 takes exponential time to compute. It could be that both take exponential time to compute. What we can infer, however, is that, if L_2 is solvable in polynomial time, then L_1 *cannot* take exponential time.

The way in which inequality is applied in NP-completeness is exactly the converse. We usually have strong evidence that L_1 is not solvable in polynomial time, and hence the reduction is effectively equivalent to saying "since L_1 is not likely to be solvable in polynomial time, then L_2 is also not likely to be solvable in polynomial time." Thus, this is how polynomial time reductions can be used to show that problems are as hard to solve as known difficult problems.

Summarizing the above, and recalling that the composition of poly-time functions is poly-time, we have the following.

Lemma: Given languages L_1, L_2, and L_3,

(i) If $L_1 \leq_P L_2$ and $L_2 \in P$, then $L_1 \in P$.
(ii) If $L_1 \leq_P L_2$ and $L_1 \notin P$, then $L_2 \notin P$.
(iii) If $L_1 \leq_P L_2$ and $L_2 \leq_P L_3$ then $L_1 \leq_P L_3$. (Transitivity of "$\leq_P$")

NP-completeness: We now have the necessary tools to define NP-completeness. This is subset of NP that are "hardest" in the sense that if it is known that if any one is solvable in polynomial time, then they all are. This is made mathematically rigorous using the notion of polynomial time reductions.

Definition: A language L is *NP-hard* if $L' \leq_P L$, for all $L' \in$ NP. (Note that L does not need to be in NP.)

Definition: A language L is *NP-complete* if:

(1) $L \in$ NP (that is, it can be verified in polymomial time), and

(2) L is NP-hard (that is, every problem in NP is polynomially reducible to it).

Unfortunately, showing that a problem is NP-hard seems nearly impossible, since it involves a property of *all* languages in NP, an infinite set. An alternative, and much easier, way to show that a problem is NP-complete is to employ transitivity.

Lemma: L is NP-complete if:

(1) $L \in$ NP, and

(2) $L' \leq_P L$, where L' is a *known* NP-hard language.

The reason is that, if L' is known to be NP-hard, then we know that all $L'' \leq_P L'$, for all $L'' \in$ NP. Thus, by transitivity, $L'' \leq_P L$, implying that L is NP-hard.

This gives us a way to prove that problems are NP-complete, once we know that *one* problem is NP-complete. But how do we do this? In our next lectures, we will do this by introducing the problem of *boolean satisfiability* (SAT) and present *Cook's theorem*, which shows that SAT is NP-complete. This is illustrated in Fig. 5 below.

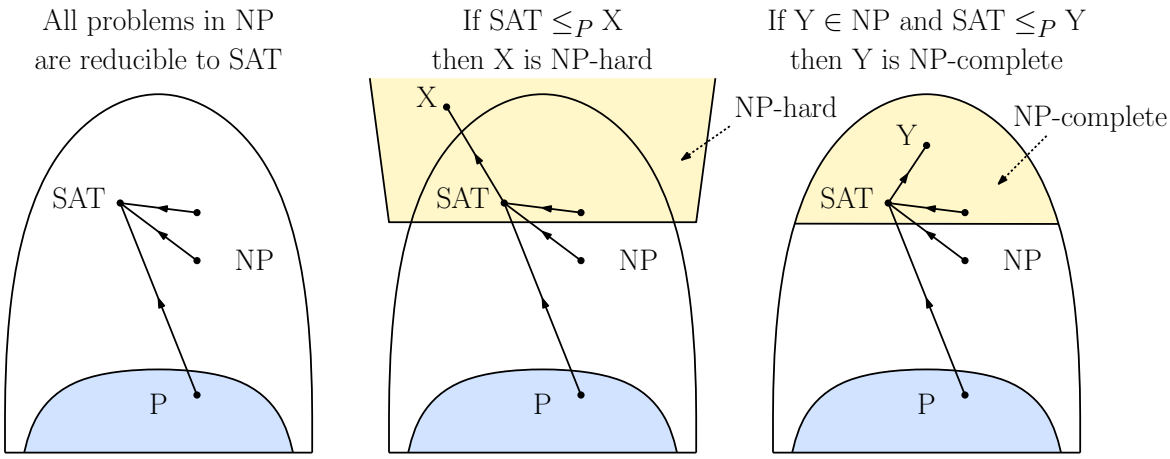

Fig. 5: Structure of NPC and reductions. Each arrowed line $L \to L'$ means that $L \leq_P L'$