CS 341: ALGORITHMS (S18) — LECTURE 19
POLYNOMIAL-TIME REDUCTIONS AND NP

ERIC BLAIS

Recall that **P** is the set of all decision problems that can be solved by polynomial-time algorithms, and that a *polynomial-time reduction* from the decision problem A to B, written $A \leq_{\mathbf{P}} B$, exists when there is a polynomial-time algorithm F that transforms inputs I_A to A into inputs I_B to B that have the same answer. In today's lecture, we continue our exploration of polynomial-time reductions and of how they might be used to show that some algorithms are (probably) not in **P**.

1. More polynomial-time reductions

Let's consider the following problems on graphs:

Clique: Given the graph G and a positive integer k, determine if G has a clique of size at least k.

IndepSet: Given the graph G and a positive integer k, determine if G has a clique of size at least k.

VertexCover: Given the graph G and a positive integer k, determine if G has a vertex cover (=a set S of vertices that "cover" all the edges $(u, v) \in E$ in the sense that at least one of u or v is in S) of size at most k.

NonEmpty: Given the graph G, determine if it has at least one edge.

HamCycle: Given the graph G on at least 3 vertices, determine if it has a *Hamiltonian cycle*—a cycle that visits each vertex in G exactly once.

HamPath: Given the graph G on at least 2 vertices, determine if it has a *Hamiltonian path*—a path that visits each vertex in G exactly once.

We can prove a number of polynomial-time reductions between these problems. In the first example, we see that the transformation in a reduction is sometimes very simple.

Lemma 19.1. NonEmpty $\leq_{\mathbf{P}}$ Clique.

Proof. Consider the polynomial-time algorithm F that transforms the input graph G into the pair $(G, 2)$ that includes the same graph and the positive integer 2.

If G is nonempty, it contains at least one edge $(u, v) \in E$. Then the set $\{u, v\}$ is a clique of size 2 in the graph, so every `Yes` instance of NonEmpty is mapped to a `Yes` instance of Clique.

To prove that every `No` instance of NonEmpty is mapped to a `No` instance of Clique, we establish the contrapositive statement: that every `Yes` instance of Clique is obtained from a `Yes` instance of NonEmpty. This is straightforward: if G contains a clique $\{u, v\}$ of size 2, then $(u, v) \in E$ must be an edge so G is nonempty. □

We can also obtain non-trivial reductions without modifying the graph itself.

Lemma 19.2. IndepSet $\leq_{\mathbf{P}}$ VertexCover.

Proof. Consider the polynomial-time algorithm F that transforms the input (G, k) to IndepSet into the input $(G, n-k)$ to VertexCover.

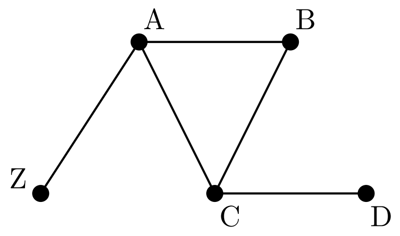
If G contains an independent set $I \subseteq V$ of size k, then the set $S = V \setminus I$ is a vertex cover of G of size $n-k$. (It is a vertex cover since no edge can have both endpoints in I since it is an independent set.)

And if G contains a vertex cover $S \subseteq V$ of size n, then the set $I = V \setminus S$ is a set of size $n-(n-k) = k$ that must be an independent set, since each edge in the graph G has at least one endpoint in S. □

We saw in the last lecture that some polynomial-time reductions, like Clique $\leq_{\mathbf{P}}$ IndepSet, are obtained by modifying the input graph itself. Here's another similar example.

Lemma 19.3. HamPath $\leq_{\mathbf{P}}$ HamCycle.

Before we prove the lemma, it's worth pausing to see that the two problems are not equivalent: there are graphs that have a Hamiltonian path but no Hamiltonian cycle. This is one example of such a graph.



Proof of Lemma 19.3. Let F be the algorithm that takes the input $G = (V, E)$ and creates the graph $G' = (V', E')$ where the set of vertices of the new graph

$$V' = V \cup \{s\}$$

is obtained by adding a new vertex s and the set of edges

$$E' = E \cup \{(s, v) : v \in V\}$$

is the original set of edges along with an edge between every vertex in V and the new vertex s. This transformation is easily computed in polynomial-time, and to complete the reduction we need to show that G has a Hamiltonian path if and only if G' has a Hamiltonian cycle.

- If G has a Hamiltonian path P with endpoints $a, b \in V$, then the cycle obtained by adding the edges (s, a) and (b, s) to the beginning and the end of the path P, respectively, forms a Hamiltonian cycle in G'.
- If G' has a Hamiltonian cycle C, that cycle contains two edges that we can call (s, a) and (s, b) that are incident to s. Removing those two edges leaves us with a Hamiltonian path in G.

□

All of these examples provide reductions between different problems on graphs, but this does not have to be the case: we can have reductions between very different types of problems as well. Consider for example the SetCover problem.

SetCover: : Given a collection $\mathcal{S}$ of subsets of $\{1, 2, \ldots, m\}$ and a positive integer k, determine if there are k sets $S_1, \ldots, S_k \in \mathcal{S}$ such that $S_1 \cup \cdots \cup S_k = \{1, 2, \ldots, m\}$.

Lemma 19.4. VertexCover $\leq_{\mathbf{P}}$ SetCover.

Proof. Let F be the algorithm that transforms a graph $G = (V, E)$ into a collection of subsets over $\{1, 2, \ldots, m\}$ with $m = |E|$. The transformation is done by labeling each edge in E with the numbers $1, \ldots, m$. Then for each vertex $v \in V$ we create the set

$$S_v = \{i \leq m : v \text{ is incident to edge } i \text{ in } G\}.$$

Let $\mathcal{S} = \{S_v\}_{v \in V}$. The result of this transformation is $(\mathcal{S}, k)$. The transformation can be completed in polynomial time. To verify that it is a reduction from VertexCover to SetCover, we need to verify that G has a vertex cover of size k if and only if $\mathcal{S}$ has a set cover of size at most k.

- If G has a vertex cover C of size k, then the family of sets S_v for each $v \in C$ satisfy $\bigcup_{v \in C} S_v = \{1, 2, \ldots, m\}$ since each edge is adjacent to at least one of the vertices in C.
- If $\mathcal{S}$ has a set cover $S_{v_1}, \ldots, S_{v_k}$ of size k, then the set $C = \{v_1, \ldots, v_k\}$ is a vertex cover for G since the edge i is adjacent to the vertex v_j corresponding to the set S_{v_j} that contained i.

□

2. Facts about polynomial-time reductions

Having now seen quite a few polynomial-time reductions, we can take a step back and try to identify some of the important basic properties of these reductions. The first one is that polynomial-time reductions are *transitive* operations on decision problems.

Theorem 19.5. *If A, B, and C are decision problems that satisfy $A \leq_{\mathbf{P}} B$ and $B \leq_{\mathbf{P}} C$, then $A \leq_{\mathbf{P}} C$.*

Proof. Let F_1 be the polynomial-time algorithm that transforms inputs for A into inputs for B, and let F_2 be the polynomial-time algorithm that transforms inputs for B into inputs for C. Let F be the algorithm that runs F_1 (on the input to A) and then F_2 (on the result of F_1). This algorithm also runs in polynomial time, and transforms inputs for A into inputs for C that satisfy the conditions of polynomial-time reductions. □

This lets us obtain new reductions by combining the ones we have already established.

Lemma 19.6. Clique $\leq_{\mathbf{P}}$ SetCover.

Proof. This result follows immediately from the transitivity of polynomial-time reductions and the fact that Clique $\leq_{\mathbf{P}}$ IndepSet (we proved this last lecture), that IndepSet $\leq_{\mathbf{P}}$ VertexCover, and that VertexCover $\leq_{\mathbf{P}}$ SetCover. □

Another important property of polynomial-time reductions is that it is *not* symmetric.

Theorem 19.7. *There are some decision problems A and B such that $A \leq_{\mathbf{P}} B$ but $B \not\leq_{\mathbf{P}} A$.*

Proof idea. We saw that NonEmpty $\leq_{\mathbf{P}}$ Clique but we have strong reasons to believe (as we will explore in the next few lectures) that there is no polynomial-time reduction from Clique to NonEmpty.[1] □

[1] You should give it a try!

This is a point worth emphasizing: the direction that you do a polynomial-time reduction really matters! And the way we will be doing these reductions will from now on usually be in the same pattern: we will reduce a known hard problem to a new problem, so that we show that the new problem is hard as well.

Theorem 19.8. *If* Hard *is a decision problem that satisfies* Hard $\notin \mathbf{P}$ *and* B *is a decision problem that satisfies* Hard $\leq_{\mathbf{P}} B$*, then* $B \notin \mathbf{P}$.

Proof. Assume for contradiction that $B \in \mathbf{P}$. Then there is a polynomial-time algorithm F that transforms inputs for Hard into inputs for B (which preserves the `Yes` and `No` answers) and there is a polynomial-time algorithm $\mathcal{A}_B$ that solves B, so we obtain a polynomial-time algorithm that solves Hard by running F and then $\mathcal{A}_B$ and outputing the result; this contradicts the fact that Hard $\notin \mathbf{P}$. □

Remark. But note that if Hard $\notin \mathbf{P}$ and we show $A \leq_{\mathbf{P}}$ Hard, this says nothing about whether A is easy or hard!

3. The class NP

Now that we have a good handle on polynomial-time reductions, we are ready to put them to good use and prove that lots of basic problems are intractable. . . except that to do so we first need to begin with at least *one* problem that we know is hard.

3.1. Verifiers.

But before we identify our hard problem, it's better to pause for a second and examine the problems Clique, Subset Sum, Vertex Cover, etc. that we would very much like to solve efficiently but can't currently solve with any polynomial-time algorithm. Do these problems have anything in common?

As it turns out, all these problems share one very interesting property: while they are all hard to solve (we believe), they are all easy to *verify*: if I claim that a graph has a clique of size k, I can convince you that this is true by identifying the k vertices that I claim form a clique. You can then check that those vertices form a clique in polynomial time with a very simple algorithm. Similarly, if I claim there is a subset of the elements that sum up to exactly the target value, I can identify the subset and you can verify that the sum is indeed the target in polynomial time. And you can verify that we can do this for every decision problem we have covered in the last two lectures.

Let's introduce some definitions to make this discussion more precise.

Definition 19.9. An algorithm A *verifies* (or *is a verifier for*) the decision problem P if

- It takes in 2 inputs, x and y, and outputs `Yes` or `No`;
- For every input x to problem P, x is a `Yes` input for P if and only if there exists a *certificate* y such that $A(x, y)$ outputs `Yes`.

So in words, a verifier is an algorithm that receives some additional "help" input y that is meant to help the algorithm verify that an input x is indeed a `Yes` input—but the algorithm must never be fooled into outputing `Yes` by *any* possible certificate when x is a `No` input.

3.2. Efficient verifiers.

The idea of an algorithm being an *efficient* verifier for a problem can now be formalized in the same way that we associated efficient algorithms with polynomial-time algorithms.

Definition 19.10. Algorithm A is a *polynomial-time verifier* for the decision problem P if it is a polynomial-time algorithm that verifies P *and* for any `Yes` input x, there is a certificate y for x of length polynomial in the length of x.

Remark. Why the second condition (after the *and*) in the definition? Remember that a polynomial-time algorithm is an algorithm whose runtime is polynomial in the size of its *input*—which for a verifier is the size of x and y; we need the condition on the length of y to guarantee that the runtime of the verifier will also be polynomial in the length of x by itself.

We now are ready to define another fundamental class of problems: **NP**.

Definition 19.11. The class **NP** is the set of all decision problems that have polynomial-time verifiers.

The famous **P** vs. **NP** problem can thus be restated as follows.

> *Can every problem that has a polynomial-time verifier also be solved with a polynomial-time algorithm?*

Answer this question—and provide a valid proof that justifies your answer!—and you will have solved one of the most significant open problems in all of computer science and mathematics today. But this is bad news for us: it means that until the **P** vs. **NP** problem has been resolved, we will not be able to show that CLIQUE $\notin$ **P**, SUBSETSUM $\notin$ **P**, etc.—since all these problems are in **NP** and it is possible that **P** = **NP**, then it's possible that all these problems are in **P** as well. But we will be able to show that if **P** $\neq$ **NP**, then CLIQUE $\notin$ **P** (and similarly for many other problems as well).