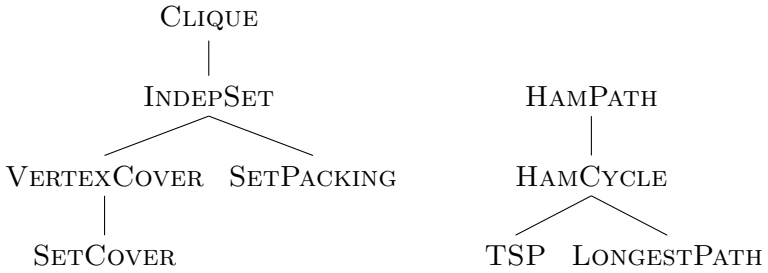
CS 341: ALGORITHMS (S18) — LECTURE 20

NP-COMPLETENESS

ERIC BLAIS

Last week, we saw a number of polynomial-time reductions between various problems.



Today, we will use these reductions to show that if any of these problems are intractable, then all of them are.

1. Polynomial-time verifiers

All the problems that we have examined last week have two interesting properties in common: first, we believe that they are hard to solve (or, at least, we haven't found any efficient algorithms to solve them yet!), and they are all easy to *verify*. We can make this idea precise by introducing a new type of algorithm that doesn't try to solve decision problems by itself, but instead expects to be given some help in the form of a *certificate* that some input x to the problem should have the answer `Yes` for the problem.

Definition 20.1. A *verifier* for the decision problem X is an algorithm A that takes in as input an input x to problem X and an additional input (that we will call a *potential certificate*) y and satisfies two conditions

(1) For every `Yes` input x to X,[1], there is a *certificate* y that causes $A(x, y)$ to output `Yes`; and
(2) For every `No` input x to X, for any possible input y the algorithm $A(x, y)$ outputs `No`.

In other words, when x is a `Yes` input *there exists* a certificate y that helps convince A that it is indeed a `Yes` input, but when x is a `No` input *there does not exist* any claimed certificate that incorrectly causes A to output `Yes`. The idea of an *efficient* verifier can be defined formally by again associating efficiency with polynomial-time complexity.

Definition 20.2. An algorithm A is a *polynomial-time verifier* for the decision problem P if it is a verifier for X with time complexity that is polynomial in the size of the input x.

[1] i.e., any input x where an algorithm that solves problem X should output `Yes` on input x.

Remark. Note that having a time complexity that is polynomial in the size of x only means that for every problem X that has a polynomial-time verifier A and every `Yes` input x, there must be a certificate y of size polynomial in the size of x that causes $A(x, y)$ to accept (since the verifier does not have enough time to even *read* longer certificates!). You will sometimes see this condition explicitly stated in the definition.

NP (for *nondeterministic polynomial-time*) is the name that was given to the class of all decision problems that can be efficiently verified.

Definition 20.3. NP is the set of all decision problems that have polynomial-time verifiers.

Let's see some examples.

Lemma 20.4. CLIQUE $\in$ **NP**.

Proof. We need to show that there is a polynomial-time verifier for CLIQUE. Consider the following algorithm that takes in as input the graph $G = (V, E)$ and the positive integer k, and as a potential certificate a set $S \subseteq V$ of size $|S| = k$.

Algorithm 1: CLIQUEVERIFIER$(G = (V, E), k, S)$

```
for each u ∈ S do
    for each v ∈ S \ {u} do
        if (u, v) ∉ E return No;
return Yes;
```

When G has a clique of size k and this clique is provided as the certificate S to the algorithm, then $(u, v) \in E$ for every $u \neq v \in S$ so the algorithm will return `Yes` when provided with this valid certificate.

When G does not contain a clique of size k, then for any set S of size k, there must exist two vertices $u, v \in S$ such that $(u, v) \notin E$ (otherwise S would be a clique of size k) and so the algorithm returns `No` for all potential certificates. □

Lemma 20.5. SUBSETSUM $\in$ **NP**.

Proof. Consider the following algorithm that takes in as input the set of n integers $a_1, \ldots, a_n$ and the target value t, and as a potential certificate a subset $S \subseteq \{1, 2, \ldots, n\}$.

Algorithm 2: SUBSETSUMVERIFIER$(a_1, \ldots, a_n, t, S)$

```
if ∑_{i∈S} a_i = t then
    return Yes;
else
    return No;
```

When there is a subset $S \subseteq \{1, 2, \ldots, n\}$ of the integers that satisfy $\sum_{i \in S} a_i = t$, providing this set S as the certificate causes the verifier to return `Yes`.

When there is *no* subset $S \subseteq \{1, 2, \ldots, n\}$ of the integers that satisfy $\sum_{i \in S} a_i = t$, then no matter what set S is provided as a potential certificate, the algorithm returns `No`. □

We can consider all the problems we have covered in the past week and see that they are all in **NP** as well. And we obtain a proof that many more problems are in **NP** by making a simple observation.

Theorem 20.6. *Every decision problem in* **P** *is also in* **NP**. *That is,* $\mathbf{P} \subseteq \mathbf{NP}$.

Proof. Fix any problem X in **P** and let A be a polynomial-time algorithm that solves X. Then A is also a polynomial-time *verifier* for X as well: all it needs to do when provided an input x and a potential certificate y is to ignore y entirely and solve X on input x and output `Yes` or `No` appropriately. This algorithm runs in polynomial-time and always provides the correct answer (no matter what potential certificate is given), so $X \in \mathbf{NP}$. □

This result should make intuitive sense: it is no harder to verify an answer to some problem than it is to solve the problem from scratch! So if we can solve a problem efficiently on our own, we expect to be able to be able to verify it efficiently as well.

2. NP-Completeness

So we now see that there are many different types of problems in **NP**: some that can be solved efficiently (namely, the problems in **P**), and others that we don't believe are efficiently solvable. The remarkable fact about **NP** is that we can formalize the idea that **NP** contains some "hardest" problems.

Definition 20.7. The decision problem X is **NP**-*complete* if

- $X \in \mathbf{NP}$, and
- For every problem $A \in \mathbf{NP}$, $A \leq_{\mathbf{P}} X$.

Proving that a decision problem X is **NP**-complete has some significant immediate implications.

Theorem 20.8. *Let* X *be any* **NP**-*complete problem. Then*

- *if* $X \in \mathbf{P}$ *then* $\mathbf{P} = \mathbf{NP}$ *and so* every *problem in* **NP** *can be solved with a polynomial-time algorithm; and*
- *if* $X \notin \mathbf{P}$, *then* $\mathbf{P} \neq \mathbf{NP}$ *and* no **NP**-*complete problem can be solved with a polynomial-time algorithm.*

As a result, proving that a problem is **NP**-complete is a very strong indication that a problem cannot be solved efficiently: the only way that the problem can be solved with a polynomial-time algorithm is if every other **NP**-complete problem can also be solved that efficiently—and computer scientists have spent countless hours trying to find efficient algorithms for many of those problems, without any success so far!

3. 3SAT & Our First NP-Completeness Proofs

When we are working with a specific decision problem X, proving that every $A \in \mathbf{NP}$ satisfies $A \leq_{\mathbf{P}} X$ is typically a rather arduous task. Thankfully, we don't have to do all this work every time; because of the transitivity of polynomial-time reductions, all we have to do is to show that *one* **NP**-complete problem A satisfies $A \leq_{\mathbf{P}} X$.

Theorem 20.9. *Let* A *be an* **NP**-*complete problem and let* $X \in \mathbf{NP}$ *be a decision problem such that* $A \leq_{\mathbf{P}} X$. *Then* X *is* **NP**-*complete.*

This is the approach we will take to prove that the problems we have been considering since the beginning of last week are **NP**-complete. But to do this, we must first identify *one* **NP**-complete problem. Cook (1971) and Levin (1973) were the first to do so, by showing explicitly that every problem in **NP** has a polynomial-time reduction to *satisfiability*

problems. The full proof of this theorem is one of the gems of CS 360/365. For reductions, the variant on the satisfiability problems that is most useful is known as *3SAT*.

Definition 20.10. In the 3SAT *problem*, we are given a Boolean CNF formula (an OR of ANDs) on n variables in which each clause has at most 3 literals; we must determine whether there is an assignment of `True` or `False` values to the n variables that makes the formula true (i.e., that *satisfies* the formula).

For example, the formula

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_4) \wedge (x_3 \vee x_4 \vee x_5)$$

is an input to the 3SAT problem for which the answer is `True`. (Set x_1 and x_4 to be `True`, assign any value to the remaining three variables.)

Cook–Levin Theorem. 3SAT *is* **NP***-complete.*

(As an exercise: can you prove that 3SAT $\in$ **NP**?) Let's see how we can now use this result to show that CLIQUE is **NP**-complete as well.

Theorem 20.11. CLIQUE *is* **NP***-complete.*

Proof. We already showed that CLIQUE $\in$ **NP**.

We want to show that 3SAT $\leq_{\mathbf{P}}$ CLIQUE. Let φ be a CNF formula with c clauses that each contain at most 3 literals. Define G to be the graph G where we create one vertex for each literal in each clause (for a total of at most $3c$ vertices) and we connect two vertices with an edge if

- the vertices represent literals in *different* clauses; and
- the literals represented by the vertices do not contradict each other (e.g., they do not represent x_i and $\neg x_i$, respectively).

This transformation is easily completed in polynomial time. And we have the following two observations.

(1) If G contains a clique $C \subseteq V$ of size c, the clique must contain one vertex that represents a literal in each of the c different clauses (since two vertices from the same clause are never connected by an edge), and none of those literals contradict each other. So we can assign values to the variables that make all those literals `True` to satisfy all the clauses in φ.

(2) If φ is satisfiable, take any satisfying assignment. This makes at least one literal in each clause evaluate to `True`. Consider the set S of vertices obtained by choosing any one `True` literal from each clause. This set S is a clique of size c since all its vertices correspond to literals in different clauses that don't contradict each other.

Therefore, the transformation we defined gives a polynomial-time reduction from 3SAT to CLIQUE. □