600.363 Introduction to Algorithms / 600.463 Algorithms I **Lecturer:** Michael Dinitz
Topic: NP-Completeness I **Date:** 11/11/14

15.1 Introduction

Definition 15.1.1 *We say that an algorithm runs in* Polynomial time *if its running time is $O(n^c)$, where n is the size of the input.*

So for example the basic Ford-Fulkerson algorithm is not polynomial time, but both Edmonds-Karp algorithms are. Polynomial time isn't a perfect formalization of "fast", since running times like n^{100} aren't particularly practical, but it's a pretty good formalization. n^{100} is certainly a lot faster than 2^n. So we will be concerned with the following question: which problems admit polynomial-time algorithms?

15.2 Semi-Formal Definitions: P and NP

The last few weeks we've been talking about optimization problems, but now we're going to consider decision problems (problems in which the output is either YES or NO). So, for example, instead of trying to compute the maximum possible flow, we'll try to answer the decision problem "is there a flow of value at least k?" This is essentially for convenience, and except in extremely rare cases isn't actually important. For example, if we know how to solve the decision problem for max-flow, we can actually find the value of a maximum flow by using binary search. And (kind of like with dynamic programming) it is usually easy to figure out the actual solution once we know the value of the actual solution (there's a question about this on the homework).

So from now on we'll pretty much only talk about decision problems. Note that for a decision problem, we can split all possible instances into two categories: YES-instances (where the correct answer is YES) and NO-instances (where the correct answer is NO).

Definition 15.2.1 **P** *is the set of decision problems solvable in polynomial time.*

For example, the decision problem "Given a directed graph, edge capacities, and a source and sink, does there exist a flow of value at least k" is in **P**. We can simply compute the max-flow using Edmonds-Karp and check whether it has value at least k.

Are all problems in **P**? No! It is known that there are problems which cannot be solved at all (e.g. the halting problem), which you should have learned about in Automata. It is also know that there are problems which *can* be solved in exponential time, but cannot be solved in polynomial time (via something called the *time hierarchy*).

One setting which commonly arises (particularly in optimization problems) is that if someone *gave* us a solution, we could easily check whether or not it was valid. Consider the decision version of max-flow: if someone gave us a flow, we could check if it had value at least k, and we could also check if flow conservation and capacity constraints held everywhere. Or consider the following decision problem known as 3-coloring:

Definition 15.2.2 *In the* 3*-coloring problem we are given a graph* $G = (V, E)$*. If there is a coloring* $f : V \to \{1, 2, 3\}$ *such that* $f(u) \neq f(v)$ *for all edges* $\{u, v\}$ *then we output YES, otherwise we output NO.*

If someone have me a coloring f, I can check whether its valid by simply checking that it assigned $1, 2$, or 3 to each node, and then looking at each edge and checking whether the colors of its endpoints are different. On the other hand, it is not at all obvious how to compute such a coloring if we are simply given a graph.

Informally, **NP** is the class of decision problems for which we can verify a YES answer. We can make this formal as follows:

Definition 15.2.3 *A decision problem* Q *is in* **NP** *if there is a polynomial time algorithm* $V(I, X)$ *with the following guarantees:*

1. *If* I *is a YES-instance of* Q*, then there is some* X *with size polynomial in* $|I|$ *so that* $V(I, X) =$ *YES.*
2. *If* I *is a NO-instance of* Q*, then for all* X*,* $V(I, X) =$ *NO.*

The algorithm V in this definition is usually called a *verifier*, and X is usually called a *witness.* Let's see some examples.

1. For 3-Coloring, the witness X is the coloring and the verifier checks that it is valid.
2. For max-flow, the witness is the flow and the verifier check that it is a valid flow (obeys capacities and flow conservation) and has value at least k
3. Consider the factoring problem, in which we are given an integer M and a value k and are asked to determine if there is a factor of M in $2, 3, \ldots, k$. Then a witness would be such a factor, and the verifier can check through a simple division whether it is indeed a factor.
4. Consider the *Traveling Salesman* problem (TSP): "Given a weighted graph G and value k, does G have a tour that visits all of the vertices and has total length at most k?" For this problem the witness is the actual tour, and the verifier checks that it does visit all of the vertices and that it does have length at most k.

There's an important subtlety here: if the answer is YES then we need to have a witness, but we *do not* need to provide a witness if the answer is no. So, for example, in TSP if no such tour exists then we don't have to give you a proof of this that you can verify, we simply need to answer NO. We only need to provide a witness if the answer is YES.

Here's an obvious theorem:

Theorem 15.2.4 $\mathbf{P} \subseteq \mathbf{NP}$.

Proof: Suppose that Q is in **P**. Then given an instance I, our verifier simply ignores X and determines if I is a YES-instance or a NO-instance. ■

The single biggest open problem in computer science is whether the converse is true: does **P** = **NP**? If the answer is yes, then this would mean that whenever we can verify a solution we can also solve the problem: finding a solution is no harder than verifying a solution. This would have massive implications, since in almost all important problems verification is easy, so if **P** = **NP** then we can solve almost all important problems in polynomial time. It seems hard to imagine that finding a solution is as easy as verifying a solution, but we are not even close to being able to prove this.

15.3 NP-completeness and Reductions

Suppose that we're trying to prove that **P** = **NP**. How could we possibly do this? We would have to consider every possible problem in **NP**, and show that each of them has a polynomial-time algorithm. This seems wildly infeasible. So this naturally leads us to try to figure out what the "hardest" problem in **NP** is. To do this, we have to have a notion of reduction. We will use the following:

Definition 15.3.1 *Problem A is* polytime reducible *to problem B (written $A \leq_p B$) if, given a polynomial-time algorithm for B, we can use it to produce a polynomial-time algorithm for A.*

Why is this a reasonable definition? If $A \leq_p B$, then B is "at least as hard" as A, in the sense that if B is in **P** then also A is in **P**. It cannot be the case that B is "easy" and A is "hard".

As a side note, when we do these reductions they will almost always be "many-one" reductions: we will only use the assumed algorithm for B a single time.

Definition 15.3.2 *A* Many-one *or* Karp *reduction from A to B is a function f which takes arbitrary instances of A and transforms them into instances of B so that*

1. *If x is a YES-instance of A then $f(x)$ is a YES-instance of B.*
2. *If x is a NO-instance of A then $f(x)$ is a NO-instance B.*
3. *f can be computed in polynomial time.*

OK, so now we have a notion of "harder" problems: problems which can be reduced to (*not* problems which can be reduced from!). This naturally gives the following definitions:

Definition 15.3.3 *Problem Q is* **NP**-hard *if $Q' \leq_p Q$ for all problems Q' in* **NP**.

Definition 15.3.4 *Problem Q is* **NP**-complete *if it is* **NP***-hard and in* **NP**.

Let's think about these definitions for a minute. First, note that it's not obvious that *any* problem is NP-complete (or even NP-hard). Why should there be a problem which all other problems reduce to? Nevertheless, we will see that there is (and in fact there are many). Second, consider the implications of designing a polynomial-time algorithm for a problem Q which is **NP**-complete. Then for any other problem Q' in **NP**, since there is a polynomial-time reduction to Q we can use our polynomial-time algorithm for Q to give a polynomial-time algorithm for Q'! So we would have proved that **P** = **NP**! Thus if you believe that **P** = **NP**, all you need to do to solve it is find an algorithm for a single **NP**-complete problem.

15.4 Circuit-SAT

Let's see our first NP-complete problem.

Definition 15.4.1 Circuit-SAT*: Given a boolean circuit with a single output and no loops (some inputs might be hardwired), is there a way of setting the inputs so that the output of the circuit is* 1*?*

I'm not going to define a boolean circuit, but hopefully you all know what it is. There are n input wires, each of which is set to either 0 or 1. There are gates which have one output and whose inputs are are either input wires or the outputs of other gates. We'll assume arbitrary fan-out – the output of a gate can be split and sent to as many other gates as we want. The standard gates are AND, OR, and NOT, and it is straightforward and well-known that any function from $\{0,1\}^n$ to $\{0,1\}$ can be computes using AND, OR, and NOT gates (although this might have exponential size). Less well-known, but useful is the fact that NAND gates (gates which compute the inverse of AND) are also universal – we can compute any function just by using NAND gates. This is a good exercise to do at home, if you're not already familiar with it.

We're going to be pretty informal here. The precise proof is a little more complex, but hopefully this gives you the idea.

Theorem 15.4.2 *Circuit-SAT is* **NP***-complete.*

Proof: We begin by arguing that Circuit-SAT is in **NP**. This is pretty obvious: a witness would be an assignment to the inputs, and then we can simply go through the gates of the circuit to verify that the output is 1.

Now we want to prove that it is **NP**-hard. Consider some other problem A in **NP**. We'll do a many-one reduction from A to Circuit-SAT, i.e., we'll construct a function f which goes from inputs of A to circuits so that $f(x)$ is satisfiable if and only if x is a YES instance of A. To define the reduction, we'll need to use the fact that A is in **NP**, so there is some verifier V that check polynomial-size witnesses. This V is an algorithm – you can think of it as running on a computer (or, if you're comfortable with it, on a Turing machine). But what is an algorithm? It's just a way of transforming the memory of the machine from one state to another on every clock tick. Since we know that the verifier runs in polynomial time, we can thus think of one big circuit of "depth" equal to the running time of V, where the inputs to the circuit are the inputs to V (an instance I and a witness X). This circuit will output 1 if X is a valid witness.

So now given an input I, we construct the circuit for V and hardwire I. If I is a YES instance of A, then there is some witness W which will cause the circuit to output 1, and thus the circuit is a YES-instance for Circuit-SAT. On the other hand, if the circuit is a YES-instance for Circuit-SAT then there is some witness W which makes the verifier output YES, and thus I was a YES-instance of A.

It only remains to show that this reduction can be computed in polynomial time. This takes a little bit of work to prove formally, but is intuitively obvious. The verifier V itself is constant-size, and only runs for a polynomial amount of time, so we can construct the circuit representing this computation in polynomial time (see book for a few more details). ■