Notes 21 for CS 170

1 Tractable and Intractable Problems

So far, almost all of the problems that we have studied have had complexities that are *polynomial*, i.e., whose running time $T(n)$ has been $O(n^k)$ for some fixed value of k. Typically k has been small, 3 or less. We will let **P** denote the class of all problems whose solution can be computed in polynomial time, i.e., $O(n^k)$ for some fixed k, whether it is 3, 100, or something else. We consider all such problems efficiently solvable, or *tractable*. Notice that this is a very relaxed definition of tractability, but our goal in this lecture and the next few ones is to understand which problems are *intractable*, a notion that we formalize as *not being solvable in polynomial time*. Notice how *not being in* **P** is certainly a strong way of being intractable.

We will focus on a class of problems, called the *NP-complete problems*, which is a class of very diverse problems, that share the following properties: we only know how to solve those problems in time much larger than polynomial, namely *exponential time*, that is $2^{O(n^k)}$ for some k; and if we could solve one NP-complete problem in polynomial time, then there is a way to solve *every* **NP**-complete problem in polynomial time.

There are two reasons to study NP-complete problems. The practical one is that if you recognize that your problem is NP-complete, then you have three choices:

1. you can use a known algorithm for it, and accept that it will take a long time to solve if n is large;
2. you can settle for *approximating* the solution, e.g., finding a nearly best solution rather than the optimum; or
3. you can change your problem formulation so that it is in **P** rather than being NP-complete, for example by restricting to work only on a subset of simpler problems.

Most of this material will concentrate on recognizing NP-complete problems (of which there are a large number, and which are often only slightly different from other, familiar, problems in **P**) and on some basic techniques that allow to solve some NP-complete problems in an *approximate* way in polynomial time (whereas an exact solution seems to require exponential time).

The other reason to study NP-completeness is that one of the most famous open problem in computer science concerns it. We stated above that "we *only know* how to solve NP-complete problems in time much larger than polynomial" not that we *have proven* that NP-complete problems require exponential time. Indeed, this is the million dollar question,[1] one of the most famous open problems in computer science, the question whether "**P** = **NP**?", or whether the class of NP-complete problems have polynomial time solutions. After

[1] This is not a figure of speech. See `http://www.claymath.org/prizeproblems`.

decades of research, everyone believes that **P**$\neq$**NP**, i.e., that no polynomial-time solutions for these very hard problems exist. But no one has proven it. If you do, you will be very famous, and moderately wealthy.

So far we have not actually defined what NP-complete problems are. This will take some time to do carefully, but we can sketch it here. First we define the larger class of problems called **NP**: these are the problems where, if someone hands you a potential solution, then you can *check* whether it is a solution in polynomial time. For example, suppose the problem is to answer the question "Does a graph have a simple path of length $|V|$?". If someone hands you a path, i.e., a sequence of vertices, and you can *check* whether this sequence of vertices is indeed a path and that it contains all vertices in polynomial time, then the problem is in **NP**. It should be intuitive that any problem in **P** is also in **NP**, because are all familiar with the fact that checking the validity of a solution is easier than coming up with a solution. For example, it is easier to get jokes than to be a comedian, it is easier to have average taste in books than to write a best-seller, it is easier to read a textbook in a math or theory course than to come up with the proofs of all the theorems by yourself. For all this reasons (and more technical ones) people believe that **P**$\neq$**NP**, although nobody has any clue how to prove it. (But once it will be proved, it will probably not be too hard to understand the proof.)

The NP-complete problems have the interesting property that if you can solve any one of them in polynomial time, then you can solve *every* problem in **NP** in polynomial time. In other words, they are at least as hard as any other problem in **NP**; this is why they are called *complete*. Thus, if you could show that *any one* of the NP-complete problems that we will study *cannot* be solved in polynomial time, then you will have not only shown that **P**$\neq$**NP**, but also that none of the **NP**-compete problems can be solved in polynomial time. Conversely, if you find a polynomial-time algorithm for just one NP-complete problem, you will have shown that **P**=**NP**.[2]

2 Decision Problems

To simplify the discussion, we will consider only problems with Yes-No answers, rather than more complicated answers. For example, consider the *Traveling Salesman Problem* (TSP) on a graph with nonnegative integer edge weights. There are two similar ways to state it:

1. Given a weighted graph, what is the minimum length cycle that visits each node exactly once? (If no such cycle exists, the minimum length is defined to be ∞.)

2. Given a weighted graph and an integer K, is there a cycle that visits each node exactly once, with weight at most K?

Question 1 above seems more general than Question 2, because if you could answer Question 1 and find the minimum length cycle, you could just compare its length to K to answer Question 2. But Question 2 has a Yes/No answer, and so will be easier for us to consider. In

[2]Which still entitles you to the million dollars, although the sweeping ability to break every cryptographic protocol and to hold the world banking and trading systems by ransom might end up being even more profitable.

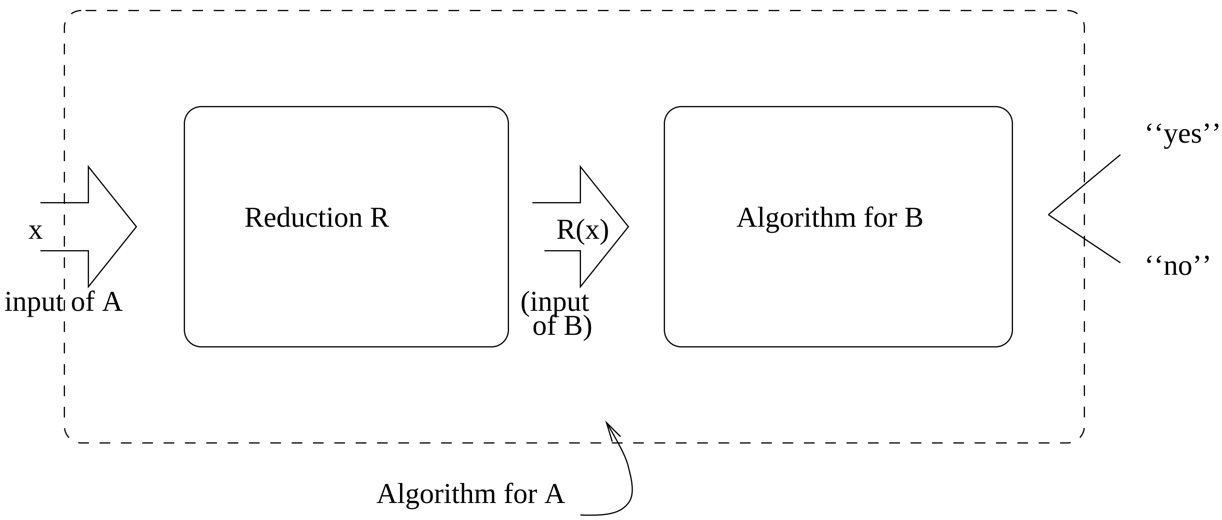

Figure 1: A reduction.

particular, if we show that Question 2 is NP-complete (it is), then that means that Question 1 is at least as hard, which will be good enough for us.[3]

Another example of a problem with a Yes-No answer is *circuit satisfiability* (which we abbreviate CSAT). Suppose we are given a Boolean circuit with n Boolean inputs $x_1, ..., x_n$ connected by AND, OR and NOT gates to one output x_{out}. Then we can ask whether there is a set of inputs (a way to assign True or False to each x_i) such that x_{out} =True. In particular, we will not ask what the values of x_i are that make x_{out} True.

If A is a Yes-No problem (also called a *decision* problem), then for an input x we denote by $A(x)$ the right Yes-No answer.

3 Reductions

Let A and B be two problems whose instances require Yes/No answers, such as TSP and CSAT. A *reduction* from A to B is a polynomial-time algorithm R which transforms inputs of A to equivalent inputs of B. That is, given an input x to problem A, R will produce an input $R(x)$ to problem B, such that x is a "yes" input of A if and only if $R(x)$ is a "yes" input of B. In a compact notation, if R is a reduction from A to B, then for every input x we have $A(x) = B(R(x))$.

A reduction from A to B, together with a polynomial time algorithm for B, constitute a polynomial algorithm for A (see Figure1). For any input x of A of size n, the reduction R takes time $p(n)$—a polynomial—to produce an equivalent input $R(x)$ of B. Now, this input $R(x)$ can have size at most $p(n)$—since this is the largest input R can conceivably construct in $p(n)$ time. If we now submit this input to the assumed algorithm for B, running in time $q(m)$ on inputs of size m, where q is another polynomial, then we get the right answer of x, within a total number of steps at most $p(n) + q(p(n))$—also a polynomial!

[3]It is, in fact, possible to prove that Questions 1 and 2 are equally hard.

We have seen many reductions so far, establishing that problems are easy (e.g., from matching to max-flow). In this part of the class we shall use reductions in a more sophisticated and counterintuitive context, in order to prove that certain problems are hard. If we reduce A to B, we are essentially establishing that, *give or take a polynomial, A is no harder than B.* We could write this as

$$A \leq B$$

an inequality between the complexities of the two problems. If we know B is easy, this establishes that A is easy. If we know A is hard, this establishes B is hard. It is this latter implication that we shall be using soon.

4 Definition of Some Problems

Before giving the formal definition of **NP** and of **NP**-complete problem, we define some problems that are **NP**-complete, to get a sense of their diversity, and of their similarity to some polynomial time solvable problems.

In fact, we will look at pairs of very similar problems, where in each pair a problem is solvable in polynomial time, and the other is presumably not.

- **minimum spanning tree:** Given a weighted graph and an integer K, is there a tree that connects all nodes of the graph whose total weight is K or less?
- **travelling salesman problem:** Given a weighted graph and an integer K, is there a cycle that visits all nodes of the graph whose total weight is K or less?

Notice that we have converted each one of these familiar problems into a decision problem, a "yes-no" question, by supplying a goal K and asking if the goal can be met. Any optimization problem can be so converted

If we can solve the optimization problem, we can certainly solve the decision version (actually, the converse is in general also true). Therefore, proving a negative complexity result about the decision problem (for example, proving that it cannot be solved in polynomial time) immediately implies the same negative result for the optimization problem.

By considering the decision versions, we can study optimization problems side-by-side with decision problems (see the next examples). This is a great convenience in the theory of complexity which we are about to develop.

- **Eulerian graph:** Given a directed graph, is there a closed path that visits each edge of the graph exactly once?
- **Hamilitonian graph:** Given a directed graph, is there a closed path that visits each *node* of the graph exactly once?

A graph is Eulerian if and only if it is strongly connected and each node has equal in-degree and out-degree; so the problem is squarely in **P** There is no known such characterization—or algorithm—for the Hamilton problem (and notice its similarity with the TSP).

- **circuit value:** Given a Boolean circuit, and its inputs, is the output T?
- **circuit SAT:** Given a Boolean circuit, is there a way to set the inputs so that the output is T? (Equivalently: If we are given *some* of its inputs, is there a way to set the remaining inputs so that the output is T.)

We know that **circuit value** is in **P**: also, the naïve algorithm for that evaluates all gates bottom-up is polynomial. How about **circuit SAT?** There is no obvious way to solve this problem, sort of trying all input combinations for the unset inputs—and this is an exponential algorithm.

General circuits connected in arbitrary ways are hard to reason about, so we will consider them in a certain standard form, called *conjunctive normal form (CNF)*: Let $x_1, ..., x_n$ be the input Boolean variables, and x_{out} be the output Boolean variable. Then a Boolean expression for x_{out} in terms of $x_1, ..., x_n$ is in CNF if it is the AND of a set of *clauses*, each of which is the OR of some subset of the set of *literals* $\{x_1, ..., x_n, \neg x_1, ..., \neg x_n\}$. (Recall that "conjunction" means "and", whence the name CNF.) For example,

$$x_{out} = (x_1 \vee \neg x_1 \vee x_2) \wedge (x_3 \vee x_2 \vee \neg x_1) \wedge (x_1 \vee x_2) \wedge (x_3)$$

is in CNF. This can be translated into a circuit straightforwardly, with one gate per logical operation. Furthermore, we say that an expression is in **2-CNF** if each clause has two distinct literals. Thus the above expression is not 2-CNF but the following one is:

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

3-CNF is defined similarly, but with 3 distinct literals per clause:

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

- **2SAT:** Given a Boolean formula in **2-CNF**, is there a satisfying truth assignment to the input variables?
- **3SAT:** Given a Boolean formula in **3-CNF** is there a satisfying truth assignment to the input variables?

2SAT can be solved by graph-theoretic techniques in polynomial time. For **3SAT**, no such techniques are available, and the best algorithms known for this problems are exponential in the worst case, and they run in time roughly $(1.4)^n$, where n is the number of variables. (Already a non-trivial improvement over 2^n, which is the time needed to check all possible assignments of values to the variables.)

- **matching:** Given a boys-girls compatibility graph, is there a complete matching?
- **3D matching:** Given a boys-girls-homes compatibility relation (that is, a set of boy-girl-home "triangles"), is there a complete matching (a set of disjoint triangles that covers all boys, all girls, and all homes)?

We know that matching can be solved by a reduction to max-flow. For **3D matching** there is a reduction too. Unfortunately, the reduction is *from 3SAT to* **3D matching**—and this is bad news for **3D matching**...

- **unary subset-sum:** Given integers $a_1, \ldots, a_n$, and another integer K *in unary*, is there a subset of these integers that sum exactly to K?
- **subset-sum:** Given integers $a_1, \ldots, a_n$, and another integer K *in binary*, is there a subset of these integers that sum exactly to K?

unary subset-sum is in **P**—simply because the input is represented so wastefully, with about $n + K$ bits, so that a $O(n^2 K)$ dynamic programming algorithm, which would be exponential *in the length of the input* if K were represented in binary, is bounded by a polynomial in the length of the input. There is no polynomial algorithm known for the real **subset-sum** problem. This illustrates that you have to represent your input in a sensible way, binary instead of unary, to draw meaningful conclusions.

5 NP, NP-completeness

Intuitively, a problem is in **NP** if it can be formulated as the problem of whether there is a solution

- They are *small*. In each case the solution would never have to be longer than a polynomial in the length of the input.
- They are *easily checkable*. In each case there is a polynomial algorithm which takes as inputs the input of the problem and the alleged solution, and checks whether the solution is a valid one for this input. In the case of **3SAT**, the algorithm would just check that the truth assignment indeed satisfies all clauses. In the case of *Hamilton cycle* whether the given closed path indeed visits every node once. And so on.
- Every "yes" input to the problem has at least one solution (possibly many), and each "no" input has none.

Not all decision problems have such certificates. Consider, for example, the problem **non-Hamiltonian graph:** Given a graph G, is it true that there is no Hamilton cycle in G? How would you prove to a suspicious person that a given large, dense, complex graph has *no* Hamilton cycle? Short of listing all cycles and pointing out that none visits all nodes once (a certificate that is certainly not succinct)?

These are examples of problems in NP:

- Given a graph G and an integer k, is there a simple path of length at least k in G?
- Given a set of integers $a_1, \ldots, a_n$, is there a subset S of them such that $\sum_{a \in S} a = \sum_{a \notin S} a$?

We now come to the formal definition.

Definition 1 *A problem A is* **NP** *if there exist a polynomial p and a polynomial-time algorithm $V()$ such that x is a YES-input for problem A if and only if there exists a solution y, with $length(y) \leq p(length(x))$ such that $V(x, y)$ outputs YES.*

We also call $\mathbf{P}$ the set of decision problems that are solvable in polynomial time. Observe every problem in $\mathbf{P}$ is also in $\mathbf{NP}$.

We say that a problem A is $\mathbf{NP}$-hard if for every N in $\mathbf{NP}$, N is reducible to A, and that a problem A is $\mathbf{NP}$-complete if it is $\mathbf{NP}$-hard *and* it is contained in $\mathbf{NP}$. As an exercise to understand the formal definitions, you can try to prove the following simple fact, that is one of the fundamental reasons why $\mathbf{NP}$-completeness is interesting.

LEMMA 1
If A is $\mathbf{NP}$-complete, then A is in $\mathbf{P}$ if and only if $\mathbf{P} = \mathbf{NP}$.

So now, if we are dealing with some problem A that we can prove to be $\mathbf{NP}$-complete, there are only two possibilities:

- A has no efficient algorithm.
- All the infinitely many problems in $\mathbf{NP}$, including factoring and all conceivable optimization problems are in $\mathbf{P}$.

If $\mathbf{P} = \mathbf{NP}$, then, given the statement of a theorem, we can find a proof in time polynomial in the number of pages that it takes to write the proof down.

If it was so easy to find proof, why do papers in mathematics journal have theorems *and* proofs, instead of just having theorems. And why theorems that had reasonably short proofs have been open questions for centuries? Why do newspapers publish solutions for crossword puzzles? If $\mathbf{P} = \mathbf{NP}$, whatever exists can be found efficiently. It is too bizarre to be true.

In conclusion, it is safe to assume $\mathbf{P} \neq \mathbf{NP}$, or at least that the contrary will not be proved by anybody in the next decade, and it is *really* safe to assume that the contrary will not be proved by us in the next month. So, if our short-term plan involves finding an efficient algorithm for a certain problem, and the problem turns out to be $\mathbf{NP}$-hard, then we should change the plan.