1 Tractable and Intractable Problems

So far, almost all of the problems that we have studied have had complexities that are *polynomial*, i.e. such problems admitted algorithms of running time $O(n^k)$ for some fixed value of k, where n is the size of the input. 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 (an algorithm of running time $O(n^{100})$ is not efficient in practice), 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 of interest 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).

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 decades of research, everyone believes that P≠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 have not quite enough money to buy a one-bedroom condo in San Francisco.

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 $|V| - 1$?". 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 we 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 these reasons (and more technical ones) people believe that P≠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≠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]

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

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

2 Search Problems

In these lectures, we are interested in proving things about *computational problems*, and so we need to provide precise definitions of what is a *problem* for which we want to devise an *algorithm*.

In the following, for a given input I, we will be interested in referring to the *size* or *length* of I, that we will denote by $len(I)$. This can be thought of as the length, in bits, of a binary representation of the data in I.

Very generally, we say that a *search problem* is a computational problem defined by a property (a Boolean function) $V(\cdot, \cdot)$, such that given an input I our goal is to find a solution S such that $V(I, S) = True$, or to determine that no such S exists. Here "V" stands for *validity*, since defining V means defining which solutions are valid outputs for a given input.

We say that a search problem A is an *NP search problem* if the property $V(I, S)$ that defines A is computable in time polynomial in the size of I. Notice that we ask the running time to be bounded by $len(I)$, and in particular it must be true that the length of valid solutions S must themselves be bounded by a polynomial in $len(I)$.

In some cases, we will be interested in *optimization* problems, where the goal is to find a solution that maximizes or minimizes a certain cost function, and such problems may not immediately fit the pattern of being NP search problems. In those cases, there are however NP search problems that are *computationally equivalent* to the optimization problem, in the sense that an algorithm for the search problems can be easily modified, with a small increase in running time, to an algorithm for the optimization problem, and vise versa. 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, find a cycle that visits each node exactly once, with total weight at most K, if such a cycle exists.

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 an algorithm that solves Question 2 can be easily modified to an algorithm that solves Question 1 by doing a binary search on the parameter K. Note that Question 2 defines an NP search problem, because given a graph G with weights on the edges, a cycle C, and an integer K, it is easy to check in polynomial time that C is indeed a cycle in G, that C touches each vertex of G precisely once, and that the sum of the weights of the edges of C is at most K.

3 Reductions

Let A and B be two search problems. A *reduction* from A to B is a polynomial-time algorithm f which transforms inputs of A to equivalent inputs of B and a polynomial time algorithm g that transforms solutions for the latter to solutions for the former. That is, given an input x to problem A, $f(x)$ is an imput to problem B, such that if y is valid solution for instance $f(x)$ of problem B, then $g(x, y)$ is a valid solution for instance x of problem A, and if there is no valid solution to $f(x)$ then there is no valid solution for x.

Notice that this implies that if Alg_B is a correct algorithm for solving problem B, then the algorithm $x \to g(x, Alg_B(f(x)))$ is a correct algorithm for solving problem A. Furthermore, if Alg_B runs in polynomial time, then so does $g(x, Alg_B(f(\cdot)))$.

If there is a reduction from A to B, then we write $A \leq B$, which reads "A reduces to B". We have proved the following important result.

Lemma 1 *If A and B are two search problems such that $A \leq B$, and B is solvable in polynomial time, then A is solvable in polynomial time.*

We have seen many reductions so far, establishing that problems are easy (e.g., from max-flow to linear programming). 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. That is, if $A \leq B$, and A is intractable problem, that is, a problem that does not admit any polynomial time algorithm, then B is also an intractable problem that admits no polynomial time algorithm.

4 NP, NP-completeness

We call NP the set of all NP search problems and we call P the set of all NP search problems that are solvable in polynomial time.

Note that it is common to associate to every NP search problem a *decision* problem, that is a problem with a YES-NO answer, that is to determine whether a given input has at least one valid solution. The standard definition of NP is the set of all *decision* problems associated to NP search problems in the above way. In these notes we follow the approach of the book, and we do not introduce decision problems. The two ways of developing the theory are equivalent, but keep in mind this difference if you do additional readings from other sources.

We say that a search problem A is NP-hard if for every N in NP, N is reducible to A, and that a problem A is NP-complete if it is NP-hard *and* it is contained in 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 NP-completeness is interesting.

Lemma 2 *If A is NP-complete, then A is in P if and only if P=NP.*

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

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

If P=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 P=NP, whatever exists can be found efficiently. It is too bizarre to be true.

In conclusion, it is safe to assume $P \neq NP$, 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 NP-hard, then we should change the plan.

5 Proving NP-completeness Results

In order to prove that an NP problem C is NP-complete we need to exhibit infinitely many reductions: we have to show that for every problem N in NP there is a reduction from N to C. In the next lecture, we will prove that a problem called CSAT (abbreviation of Circuit Satisfiability) is NP-complete.

Once we have found an NP-complete problem, however, proving that other problems are NP-complete becomes easier, since we now just need one more reduction.

Indeed, the following result clearly holds:

Lemma 3 *If A reduces to B, and B reduces to C, then A reduces to C.*

PROOF: Let f_{AB} and g_{BA} be the functions that define the reduction $A \leq B$ and let f_{BC} and g_{CB} be the functions that define the reduction $B \leq C$.

It now follows from the definition that the functions $x \to f_{BC}(f_{AB}(x))$ and $y \to g_{BA}(x, g_{CB}(f(x), y))$ are polynomial time computable functions that define a reduction from A to C. $\square$

This immediately implies:

Lemma 4 *Let C be an NP-complete problem and A be a problem in NP. If we can prove that C reduces to A, then it follows that A is NP-complete.*

Right now, literally thousands of problems are known to be NP-complete, and each one (except for a few "root" problems like Circuit-SAT) has been proved NP-complete by way of a single reduction from another problem previously proved to be NP-complete. By the definition, all NP-complete problems reduce to each other, so the body of work that lead to the proof of the currently known thousands of NP-complete problems, actually implies *millions* of pairwise reductions between such problems.

6 Proving NP-completeness

6.1 3SAT

We refer to the book for the definition of 3SAT. We will prove later that 3SAT is NP-complete. From now on, we will assume the NP-completeness of 3SAT and use this fact to prove the NP-completeness of several other problems, by showing reductions from 3SAT to these other problems.