Lecture 12:
P, NP, and "search to decision" reductions

Valentine Kabanets

November 2, 2016

1 Complexity Theory

We take a closer look at the class of decidable problems, and want to classify these problems according to the amount of time/space they require.

Example: Checking if two numbers are relatively prime can be done in polytime, using the Euclidean algorithm for GCD computation.

Here, "polytime" means "polynomial in the size of the input." For input a, where a is a natural number presented in binary, the input size of a is at most $\lceil \log_2 a \rceil$ bits long. So, for an algorithm to be polytime, it must run in time $poly(\log_2 a)$.

On the other hand, the problem of factoring a given integer number a appears to require more that polynomial time. Currently, there is no $poly(\log_2 a)$-time algorithm for factoring a given number a. (This excludes the quantum computers, for which a polynomial-time factoring algorithm is known. But, sufficiently large quantum computers haven't been built yet, and so we still can't factor numbers efficiently in practice.)

2 Polynomial Time

Recall that computation of a DTM is a path (sequence of configurations), whereas the computation of an NTM is a tree (of configurations). The time on input x is the height of a computation tree on input x. For a given TM M, $time(n)$ is the maximum (over all inputs x of size n) of the time TM M takes on x.

For a function $t : \mathbb{N} \to \mathbb{N}$, we define complexity class

$$\mathsf{TIME}(t(n)) = \{L \mid L \text{ is decided by some deterministic TM running in time } O(t(n))\}.$$

We define

$$\mathsf{P} = \cup_{k>0} \mathsf{TIME}(n^k).$$

> The intuitive notion of practical algorithm is captured by the formal notion of a polytime Turing machine, i.e., Practical Algorithm = Polytime TM.

Feasibility Thesis: If a problem is in P, then there is a "fast" polytime algorithm for this problem. (That is, if you have an algorithm for some problem A whose running time is n^{10}, then probably, with some more work, you can get a better algorithm, whose running time is n^2, or better.)

2.1 Why P is a "good" class

We consider P a good class because

1. P is robust: other models of computation (k-tape TM, RAM computers, etc.) can be simulated by 1-tape TM with only polynomial slowdown.

2. P contains problems that can be solved in reasonable amount of time.

2.2 Why we ignore constants when we define the class $\mathsf{TIME}(t(n))$

For any TM M of running time $t(n)$ and any constant c, we can design a new TM M' whose running time is $n + t(n)/c$. That is, we can speed up the computation of a TM by any constant factor. The idea is to "compress" each c-tuple of symbols of M into a single "super-symbol" of M'. In other words, one symbol of M' will contain the information about c symbols of M. Now, such a TM M' can simulate c steps of M in just a couple of steps. Hence, TM M' is about c times faster than the old TM M.

What the above argument says is that TMs are not sensitive enough to take into account constant factors. That is why we can just ignore these constant factors, and simply use the "Big O" notation.

3 Nondeterministic Polynomial Time (NP)

We define NP as follows:

Definition 1: $\mathsf{NP} = \cup_{k>0}\mathsf{NTIME}(n^k)$. That is, L is in NP if there is a nondeterministic polytime TM M such that $L(M) = L$.

For the second definition, we need the notion of a *verifier* V. A verifier V is a deterministic polytime algorithm.

Definition 2: NP contains all those languages L such that there exists a constant c, and a polytime verifier V so that, for every input x,

$$x \in L \Leftrightarrow \exists y,\ |y| \leq |x|^c,\ V(x, y) \text{ accepts.}$$

Interpretation: think of a string y as a *witness/certificate* of membership in L. So, language L is in NP if there is a polynomial bound on the size of certificates of membership, and there is an efficient (polytime) algorithm for testing certificates.

Theorem 1. *The two definitions of* NP *are equivalent.*

Proof Idea. Given a polytime NTM M deciding a language L, we can define the certificate y for $x \in L$ to be a (description of) accepting computation path of M on input x. (Note that this certificate is polynomially bounded in $|x|$ since M is a polytime TM. Also note that we can test in polytime whether a given sequence is indeed an accepting computation path of M on input x.) Thus, we can say that

$$L = \{x \mid \exists y,\ |y| \leq |x|^c,\ \ V(x, y) \text{ accepts}\},$$

where $V(x, y)$ accepts iff y is an accepting path of NTM M on input x.

For the other direction. Given

$$L = \{x \mid \exists y,\ |y| \leq |x|^c,\ \ V(x,y) \text{ accepts}\},$$

we define an NTM M for L as follows: "On input x, nondeterministically guess a string y of length at most $|x|^c$, and simulate $V(x,y)$. Accept if $V(x,y)$ accepts, and Reject otherwise." Note that the described NTM runs in polytime, and that $L = L(M)$. □

3.1 Examples

SAT

$$SAT = \{\langle\phi\rangle \mid \phi(x_1, \ldots, x_n) \text{ is a satisfiable propositional formula}\}$$

SAT is in NP: Given $\phi(x_1, \ldots, x_n)$, nondeterministically guess an assignment $a_1 \ldots a_n$ to the variables, and check that this assignment makes the formula evaluate to true.

The certificate is the truth assignment to the n variables (and hence is of size at most that of the input formula ϕ). Checking the certificate is in polytime because it reduces to evaluating a given propositional formula.

Hamiltonian Path

$$HamPath = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a Hamiltonian path from vertex } s \text{ to vertex } t\}$$

HamPath is in NP: Given $\langle G, s, t\rangle$, nondeterministically guess a sequence of n vertices, where n is the total number of vertices in G; check if the guessed sequence of vertices is a Hamiltonian path from s to t (i.e., if it is a path that starts in s, ends in t, and visits every vertex of the graph G exactly once).

So, in this case, certificate = sequence S of n vertices; Verifier $V(\langle G, s, t\rangle, S)$ accepts iff S is a Hamiltonian path from s to t in G.

Clique

$$Clique = \{\langle G, k\rangle \mid G \text{ is a graph containing a clique of size } k\}$$

(clique in a graph is a set of vertices such that every pair of vertices in the set is connected by an edge).

Clique is in NP: Given $\langle G, k\rangle$, nondeterministically guess a set S of k vertices of G; deterministically check if the set S is a clique. If S is a clique, then Accept; else, Reject.

In this case, the certificate is a set of k vertices (note that the size of the certificate is at most the size of the input graph). Testing the certificate involves checking that each pair of vertices in S is connected by an edge. This testing can be done in polytime.

Subset Sum

$$SubsetSum = \{\langle a_1, \ldots, a_n; t\rangle \mid \exists S \subseteq \{1, \ldots, n\},\ \sum_{i \in S} a_i = t\}$$

SubsetSum is in NP: Nondeterministically guess a subset S of $\{1, \ldots, n\}$; check if the numbers a_i, for $i \in S$, add up to t.

3.2 Importance of NP

Note that the class NP (using the second Definition above) captures exactly those problems that have "short" and "quickly verifiable" solutions. This doesn't of course mean that finding such solutions is easy. In fact, the famous "P vs. NP" conjecture states that finding solutions is hard in general.

What this definition reflects, though, is that, usually, people are interested in questions that have "short" answers and the answers that can be "easily understood" (verified for correctness). Lots of optimization problems (e.g., finding a good schedule, finding a good route, etc.) are of this nature. Precisely this abundance of natural important problems in the class NP is what makes the class NP so extremely important!

The main open problem in Complexity Theory (and one of the main open problems in all of mathematics) is the question:

Is $\mathsf{NP} = \mathsf{P}$?

It is conjectured that $\mathsf{NP} \neq \mathsf{P}$.

4 Search to decision reductions

Every language L can be uniquely associated with a decision problem $f : \Sigma^* \to \{0, 1\}$. We say that $x \in L$ iff $f(x) = 1$.

The complexity classes P and NP are classes of *languages*. What people are usually interested in are "search" problems.

4.1 Examples

SAT :

- **Decision problem:** Given a propositional formula $\phi(x_1, \ldots, x_n)$, decide if ϕ is satisfiable.
- **Search problem:** Given a propositional formula $\phi(x_1, \ldots, x_n)$, find a satisfying assignment for ϕ, if it exists.

Clique

- **Decision problem:**

  $$Clique = \{\langle G, k\rangle \mid G \text{ is a graph containing a clique of size } k\}$$

  (clique in a graph is a set of vertices such that every pair of vertices in the set is connected by an edge)
- **Search problem:** Given $\langle G, k\rangle$, find a clique of size k in G, if it exists.

Hamiltonian Path

- **Decision problem:**

  $$HamPath = \{\langle G, s, t\rangle \mid G \text{ is a digraph that has a Hamiltonian path from } s \text{ to } t\}$$

- **Search problem:** Given $\langle G, s, t\rangle$, find a Hamiltonian path in G from s to t, if such a path exists.

Subset Sum

- **Decision problem:**

$$SubsetSum = \{\langle a_1, \ldots, a_n; t\rangle \mid \exists S \subseteq \{1, \ldots, n\}, \sum_{i \in S} a_i = t\}$$

- **Search problem:** Given $\langle a_1, \ldots, a_n; t\rangle$, find a subset of a_i's that adds up to t, if such a subset exists.

4.2 Search-to-decision reductions for our examples

Obviously, if one can solve *CliqueSearch* problem in polytime, then one can also solve the *CliqueDecision* problem in polytime. Similarly for all the other pairs of Search-Decision problems on our list.

What about the converse? Suppose we're given a polytime algorithm for *CliqueDecision*. Can we use this algorithm to solve the *CliqueSearch* in polytime?

It turns out that Yes, we can! Let's call an algorithm solving *CliqueDecision* problem CD. Then we can specify an algorithm for *CliqueSearch* as follows:

> On input $\langle G, k\rangle$,
>
> 1. If $CD(\langle G, k\rangle)$ =No, then return "No clique"; otherwise, continue.
> 2. For every vertex v of G, if $CD(\langle G - v, k\rangle)$ =Yes then $G = G - v$.
> 3. Return G.

The idea of this algorithm is this: we eliminate all those vertices of G that do not occur in a clique of size k. In the end, we are left just with k vertices that form a clique in G.

Similar algorithms can be designed for other search problems on our list.

SATSearch: set one variable at a time, and continue with that partial assignment which still makes the original formula satisfiable.

HamPathSearch: remove one edge at a time if the remaining graph still has a Hamiltonian path.

SubsetSumSearch: remove one number a_i at a time if the remaining numbers a_j's still have a subset that adds up to t.

Thus, for each of the decision problems on our list (SAT, HamPath, SubsetSum, Clique), we have a *search version*, and we can show the "search-to-decision" reduction:

> If a decision version of problem A is solvable in polytime, then the search version of problem A is also solvable in polytime.

4.3 Sometimes no "search-to-decision" reduction is known

Finally, we note that it's not always easy (or even possible) to reduce search problems to the corresponding decision problems. For example, consider the problem

Composite: Given a number n (in binary), decide if n has a nontrivial factor (i.e., if n is a composite number).

The corresponding search version would be:

CompositeSearch: Given a number n (in binary), find a nontrivial factor of n, if one exists.

Here, we even know a deterministic polytime algorithm for Composite (due to Agrawal, Kayal, and Saxena, discovered in 2003), but we don't know any polytime algorithm for CompositeSearch. Note that CompositeSearch is the same as the problem of factoring a given integer! If there were an efficient reduction of Search to Decision in this case, we would have an efficient algorithm for factoring numbers, which is still unknown (and is a well-known difficult open question[1]).

[1] A polytime *quantum* algorithm for factoring is known, and some mathematicians conjecture that a classical polytime algorithm should also exist. However, other mathematicians conjecture factoring is hard, and even use the assumed hardness of factoring to build cryptosystems, such as RSA.