

CMPT 710 - Complexity Theory: Week 1

Valentine Kabanets

September 7, 2006

1 Why Complexity Theory?

Ancient Greeks considered and solved the following two problems.

Problem 1: Is a given number N prime?

Problem 2: Do given numbers N and M have a common prime factor?

Problem 1 was solved by Eratosthenes (via his “Sieve”): Write down all integers less than or equal to N . Then cross out all even numbers, all multiples of 3, of 5, and so on, for each prime less than \sqrt{N} . If N remains on the list, then N is prime.

Problem 2 was solved by Euclid via what is now known as Euclid’s Algorithm for finding GCD of M and N : Initially set $r_0 = M$, $r_1 = N$, and $i = 1$. While $r_i \neq 0$, assign $r_{i+1} = r_{i-1} \bmod r_i$ and $i = i + 1$. Return r_{i-1} .

The important difference between the two algorithms is that Euclid’s algorithm is very efficient (polynomial-time in the sizes of its inputs), whereas Eratosthenes’ algorithm is extremely *inefficient*. Many other problems in mathematics had/still have inefficient algorithmic solutions (factoring numbers, factoring polynomials, etc.) **Remark:** Only very recently, in 2003, the first efficient (deterministic polynomial-time) algorithm was found for the problem of primality testing, by Agrawal, Kayal, and Saxena. So now both problems mentioned above have efficient solutions.

For some problems, no algorithms could be found for a long time. Turing (1930’s) formalized the notion of an algorithm (via Turing machines), and showed that certain problems have no algorithmic solution! This is truly one of the deepest results in mathematics.

Then first computers were built, and people realized that, in practice, efficient algorithms are what one needs. The notion of an *efficient* algorithm was formalized in the 1960’s: the class P of problems solvable in polynomial time was defined. Thus, complexity theory was born!

Informally, Complexity Theory = Computability Theory with Limited Resources (e.g., time, space, randomness, ...)

Main goal of Complexity Theory: *obtain a complete taxonomy of interesting problems according to the amount of computational resources needed to solve them.*

It is still a very distant goal. The field is still young, with many exciting open problems. It addresses some of the most fundamental questions in Computer Science and all of Mathematics.

Hardness is also useful in practice. For example, for Cryptography, where security of cryptographic protocols is based upon assumed hardness of certain problems (like factoring, for the case of RSA).

In this course we'll see what (little) is known, what is conjectured, and why it's so hard to answer the many open questions in complexity theory.

2 Review

1. Problems and Languages
2. Turing machines
3. Reductions
4. Complexity classes
5. Completeness

2.1 Problems and Languages

A *function* problem is a function from strings to strings

$$f : \Sigma^* \rightarrow \Sigma^*$$

For example, given an integer, find its prime factorization; or, given a graph G and two vertices s and t , find a shortest path from s to t in G .

A *decision* problem is a function from strings to Boolean values

$$f : \Sigma^* \rightarrow \{yes, no\}$$

For example, given integer, is it prime?

A *language* associated with a given decision problem f is the set $\{x \in \Sigma^* \mid f(x) = yes\}$. For example, SAT = { satisfiable Boolean formulas }.

2.2 Turing Machines

A Turing machine (TM) consists of an infinite tape, divided into cells; each cell can hold one symbol from a given alphabet Σ . We denote the blank symbol by “-”, and assume it is contained in Σ . TM has a finite control that can be in any of (finitely) many states from the set Q . There are three special states: $q_{start}, q_{accept}, q_{reject}$ in Q . The behaviour of a TM is specified by the transition function $\delta : Q \times \Sigma \rightarrow Q \times \Sigma \times \{L, R, -\}$.

TM starts in q_{start} scanning the leftmost symbol of the input, and proceeds according to δ . It accepts by entering q_{accept} , and rejects by entering q_{reject} . TM may also run forever (i.e., loop).

A k -tape Turing machine (TM) consists of k infinite tapes. Usually, one tape is the input tape; one tape is the output tape; the remaining $k - 2$ tapes are work tapes. The transition function is $\delta : Q \times \Sigma^k \rightarrow Q \times \Sigma^k \times \{L, R, -\}^k$.

Examples (1) Consider a language $PARITY = \{x \in \{0, 1\}^* \mid x \text{ has an odd number of 1s}\}$. It is very easy to define a 1-tape TM (in fact, a finite automaton) that accepts exactly this language. (2) Consider $PAL = \{x \in \{0, 1\}^* \mid x = x^R\}$ (a language of palindromes). There is a simple 1-tape TM accepting PAL , with running time $O(n^2)$, where n is the input length. (The idea is to match the first and the last symbol of the input; if they are different, stop and reject; otherwise, erase these two symbols, and recursively continue with the remaining (shorter) string.) If we allow ourselves to use a 2-tape TM, we can decide PAL in time $O(n)$, linear time! (Can you see how?) Is it possible to invent a 1-tape TM for PAL with running time better than quadratic? It turns out the answer is No (as you'll be asked to show in the homework).

Fact A k -tape TM can be *efficiently* simulated by a one-tape TM (with only a polynomial, in fact quadratic, slowdown). The details are left as an exercise.

Note that thanks to the quadratic lower bound for deciding PAL on a 1-tape TM, the quadratic slowdown stated in the Fact above is optimal. That is, it is impossible in general to convert any k -tape TM to a 1-tape TM with less than quadratic slowdown. (Do you see why?)

There are many other models of computation (multi-head TMs, multi-dimensional tape TMs, computers with RAM, etc.) All of them can be simulated by a 1-tape TM with only polynomial slowdown. This means that if we ignore polynomial speedups, any efficient (polytime) computation on any reasonable model of computation can be performed by a polytime 1-tape TM.

An extension of the Church-Turing thesis is as follows:

Extended Church–Turing Thesis *Everything efficiently computable on a physical computer (in the intuitive sense) is efficiently computable by a Turing machine (in the formal sense).*

(Quantum computers pose a potential challenge to this thesis, but they haven't been built yet.)

3 Reductions

We say that language L_1 is reducible to L_2 , denoted by $L_1 \leq L_2$ if there is an *efficiently* computable function $f : \Sigma^* \rightarrow \Sigma^*$ such that

$$x \in L_1 \Rightarrow f(x) \in L_2,$$

$$x \notin L_1 \Rightarrow f(x) \notin L_2.$$

For now, “efficiently” means “in polytime”.

Interpretations of $L_1 \leq L_2$:

- L_1 is at most as hard as L_2 : can solve L_1 , given an algorithm for L_2 [algorithm design technique]
- L_2 is at least as hard as L_1 : if L_1 is known to be hard, then the reduction $L_1 \leq L_2$ yields that L_2 is also hard [lower bound technique]

Example

- $3SAT = \{\phi \mid \phi \text{ is a satisfiable 3CNF formula}\}$ (Recall that a 3CNF is a conjunction of clauses, where each clause is a disjunction of three literals; a literal is a variable or its negation.)
- $IS = \{(G, k) \mid G \text{ is a graph with an independent set of size } \geq k\}$ (Recall that an independent set is a subset of vertices such that no two vertices in the subset are connected by an edge.)

Reduction: $3SAT \leq IS$

Given $\phi = (x \vee y \vee \neg z) \wedge (\neg x \vee w \vee z) \wedge \dots ()$ with m clauses, produce the graph G_ϕ that contains a triangle for each clause, with vertices of the triangle labeled by the literals of the clause. Plus, add an edge between any two complementary literals from different triangles. In our example, we have triangles on $x, y, \neg z$ and on $\neg x, w, z$, plus the edges $(x, \neg x)$ and $(\neg z, z)$. Finally, set $k = m$.

Theorem 1. ϕ is satisfiable iff G_ϕ has an independent set of size at least k .

Proof. Exercise. □