BCCS: Computational methods for complex systems Tuesday, 8 June 2010

Lecture 1

Lecturer: Ashley Montanaro[1] *Computational complexity (1)*

1 Overview

When is a problem tractable, and when is it intractable? In the next couple of lectures, I'll discuss ways in which we can attempt to answer this question rigorously. Remarkably, it will turn out that many problems which we want to solve can be grouped together into *complexity classes* of similar difficulty. As "the complexity of a problem" seems like a vague and woolly notion, the first step is to write down some definitions to enable us to study this idea mathematically.

We'll restrict ourselves to *decision problems* for now: problems where we have to output "yes" or "no". Such problems can be characterised in terms of *languages*. An *alphabet* Σ is a set of letters (e.g. the binary alphabet $\{0,1\}$). A *word* in Σ is a finite length string whose letters are picked from Σ. The set of words in Σ is denoted Σ^*. Finally, a language $\mathcal{L} \subseteq \Sigma^*$ is a set of words. For example:

- The language of prime numbers $\mathcal{L} = \{m : m \text{ is prime}\}$, written in binary;
- The language of halting Turing machines, encoded in some suitable way;
- The language of true mathematical statements in English.

Note that the alphabet we choose doesn't usually make much difference, and indeed in future we'll assume that the alphabet used is $\{0,1\}$.

Let M be a Turing machine. We say that $\mathcal{L}$ is *decided* by M if:

- On all input words $x \in \mathcal{L}$, M halts in the ACCEPT state;
- On all input words $x \notin \mathcal{L}$, M halts in the REJECT state.

Decision problems and languages are essentially interchangeable, and we will do so throughout these lectures.

2 Time complexity

Most computational tasks that we want to perform require effort that grows with the input size, and one of the most important resources that we can expend is time.

Imagine that M is a Turing machine that decides some language $\mathcal{L}$. The time complexity or running time of M is the function $f : \mathbb{N} \to \mathbb{N}$ such that $f(n)$ is the largest number of steps that M makes on any input of size n. This is a *worst case* notion of complexity.

[1] http://www.cs.bris.ac.uk/~montanar/

We would like to understand the intrinsic time complexity of a problem – the amount of time required to solve it – while abstracting away the details of the hardware we use to solve it, and so on. To understand the "true" complexity of solving a problem while avoiding "unimportant" details, an important tool is asymptotic (or big-O) notation, which is a tool for understanding the rate of growth of a function. This notation works as follows.

Let $f, g : \mathbb{N} \to \mathbb{R}$ be two functions from the natural numbers to the reals. We say that $f(n) = O(g(n))$ if there exists a positive real number c and integer n_0 such that:

$$\text{for all } n \geq n_0,\ f(n) \leq c\, g(n).$$

For example, consider $f(n) = 100n^3 + 3n^2 + 7$. It's easy to convince yourself that $f(n) = O(n^3)$. In fact, any polynomial of degree k is $O(n^k)$, and in future we say "$f(n) = \text{poly}(n)$" as shorthand for "$f(n) = O(n^k)$ for some k". On the other hand, consider $g(n) = 1.0001^n$. Then one can show that $g(n)$ is not $O(n^k)$ for any constant k.

3 Polynomial and exponential time

We now come to a key distinction in computational complexity: between polynomial and exponential time.

Definition 1. *Define the following classes:*

- $\mathsf{TIME}(f(n))$ *is the class of all languages decided by a Turing machine running in time* $O(f(n))$.
- P *is the class of languages that are decided by a Turing machine in polynomial time, i.e.* $\mathsf{P} = \bigcup_k \mathsf{TIME}(n^k)$.
- EXP *is the class of languages that are decided by a Turing machine in exponential time, i.e.* $\mathsf{EXP} = \bigcup_k \mathsf{TIME}(2^{n^k})$.

Mathematically, P and EXP are sets of languages, and it should be clear that $\mathsf{P} \subseteq \mathsf{EXP}$ (in fact, one can prove that this inclusion is strict, i.e. there are problems which can be solved in exponential time that cannot be solved in polynomial time). There is an even stronger result, called the *time hierarchy theorem*, which implies that $\mathsf{TIME}(n^k) \subset \mathsf{TIME}(n^{k+1})$ for any $k \geq 0$; however, we won't prove this here.

We think of P as the class of languages which we have a hope of deciding with an "efficient" algorithm, or in other words the class of problems we might be able to solve in practice. This seems plausible for problems where we have an algorithm whose running time is $O(n)$, but what about a problem where the best algorithm we have runs in time $O(n^{100})$? All we can say is that such problems don't seem to occur very often; usually, when we can prove that a problem is in P, it turns out that the exponent in the polynomial isn't too big.

Here are some examples of problems in P:

- Circuit Value: given a circuit C, what is the first bit of the output of C on input x?
- Primes: given an integer m, is m a prime number?

- Linear Programming: find $z = \max_x(c^T x)$ subject to the constraint $Ax \leq b$.

Strictly speaking, the last of these is a functional, rather than decision problem, but it should be easy to see how it can be modified to make it a decision problem. Note that the second and third of these were only proven to be in P quite recently (2002 and 1979, respectively). So it's often not obvious a priori whether a problem is in P, i.e. whether it has a polynomial-time algorithm.

4 Reductions between languages

We would like to formalise the notion of one decision problem being at least as hard to solve as another. In particular, we would like to understand when a polynomial-time solution to problem A implies a polynomial-time solution to problem B.

- We say that $f : \Sigma^* \to \Sigma^*$ is a *polynomial-time computable function* if there is a polynomial time Turing machine that halts with $f(x)$ on its tape when started with input x on the tape.
- We say that the language $\mathcal{A}$ is polynomial time reducible to $\mathcal{B}$ if there is a polynomial-time computable function f such that $w \in \mathcal{A}$ if and only if $f(w) \in \mathcal{B}$. Here f is called a polynomial-time reduction from $\mathcal{A}$ to $\mathcal{B}$ and we write $\mathcal{A} \leq_P \mathcal{B}$.

It's clear from these definitions that, if $\mathcal{A} \leq_P \mathcal{B}$ and $\mathcal{B} \in \mathsf{P}$, $\mathcal{A} \in \mathsf{P}$. A very nice aspect of polynomial-time reductions is that they *compose*: if $\mathcal{A} \leq_P \mathcal{B}$, and $\mathcal{B} \leq_P \mathcal{C}$, then $\mathcal{A} \leq_P \mathcal{C}$. This is a major motivation for choosing P as our class of "reasonable" problems.

5 Further reading

There's a vast amount of information about computational complexity on the Internet and elsewhere. These lectures are mostly based on the lecture notes for the course "Computational Complexity Theory" by Richard Jozsa[2], which are highly recommended for a gentle introduction. The following books are also good background reading.

- *Introduction to the Theory of Computation*, Michael Sipser. An undergraduate-level book; part 3 is about complexity theory.
- *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Michael R. Garey and David S. Johnson. A classic of the field containing descriptions of many interesting problems.
- *Computational Complexity*, Christos H. Papadimitriou. Dense but mathematically precise and full of interesting details.
- *Computational Complexity: A Modern Approach*, Sanjeev Arora and Boaz Barak. A very recent and comprehensive text containing many important developments in the field.

The lecture next week on hardness of approximation is based on the latter two books.

[2] http://www.cs.bris.ac.uk/Teaching/Resources/COMS30126/