BCCS: Computational methods for complex systems Wednesday, 9 June 2010

Lecture 2

Lecturer: Ashley Montanaro[1] *Computational complexity (2)*

1 Overview

In this lecture, we move on to the study of intractability, and in particular NP-*completeness*, perhaps the most important concept in computer science.

2 The complexity class NP

NP ("non-deterministic polynomial time"; **not** "non-polynomial time") is the class of decision problems that have certificates which can be verified efficiently.

Let $\mathcal{L}$ be a language, and let M be a Turing machine that takes as input a pair of words (x, c). We say that M is a *verifier* for $\mathcal{L}$ if:

- For all $x \in \mathcal{L}$, there exists a c such that M halts in the ACCEPT state on input (x, c);
- For all $x \notin \mathcal{L}$, then for all c, M halts in the REJECT state on input (x, c).

We say that c is a *certificate* for $\mathcal{L}$.

M is a *polynomial-time* verifier for $\mathcal{L}$ if it runs in time polynomial in the length of x (not c!).

Definition 1. NP *is the set of all languages* $\mathcal{L}$ *that have a polynomial-time verifier.*

Intuitively:

- P is the class of problems that can be *solved* efficiently;
- NP is the class of problems whose solutions can be *checked* efficiently.

It's obvious that $\mathsf{P} \subseteq \mathsf{NP}$. The most important open problem in computer science is to determine whether this inclusion is strict (this is the famous "P versus NP" problem). As well as having an untold number of practical applications, solving this would win you $1 million from the Clay Mathematics Institute[2]. More importantly, this is a question which is at the heart of understanding mathematical thought itself. It is conjectured that the classes are different – informally this would mean "there are problems which are easier to check than to solve". However, there are many obstacles to resolving this conjecture.

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

[2] http://www.claymath.org/millennium/P_vs_NP/

2.1 Examples

All of the example problems from the last lecture are of course in NP. Some additional problems which are in NP (and not known to be in P) are:

- FACTORING: given an integer x, and an integer c, does x have a prime factor smaller than c?
- BOOLEAN SATISFIABILITY (SAT): given a boolean formula in Conjunctive Normal Form (CNF), is the formula satisfiable?

This second problem is quite fundamental. Here are some definitions to make it make sense.

- A boolean variable x takes values 0 or 1 (false or true).
- Boolean operations AND ($x_1 \wedge x_2$), OR ($x_1 \vee x_2$), and NOT ($\neg x$) are defined as you might expect: $x_1 \wedge x_2 = 1$ if and only if $x_1 = x_2 = 1$; $x_1 \vee x_2 = 1$ if either x_1, x_2 or both are 1; $\neg x = 1 - x$.
- A boolean formula is an expression containing boolean variables and operations, such as $\phi(x_1, x_2, x_3) = x_1 \wedge ((\neg x_2 \vee x_3) \wedge \neg x_3)$.
- A boolean formula is in Conjunctive Normal Form (CNF) if it is written as $c_1 \wedge c_2 \wedge \cdots \wedge c_n$, where each c_i is a *clause* which contains only ORs and negations (e.g. the formula

  $$\phi(x_1, x_2, x_3, x_4) = (x_1 \vee \neg x_2) \wedge (\neg x_2 \vee \neg x_3 \vee x_4) \wedge (x_1)$$

  is in CNF).
- A boolean formula is *satisfiable* if there is an assignment to the variables that makes it evaluate to TRUE. For example, the previous formula is satisfiable (e.g. set $x_1 = 1, x_2 = 0, x_3 = 1, x_4 = 0$).

Why is SAT in NP? If we are given an assignment to the variables which is claimed to make some formula ϕ evaluate to true, we can just plug it in and check. However, it is not clear how to (efficiently) *find* an assignment that makes ϕ true. If n is the number of variables used in ϕ, we can test every possible true/false assignment to the variables in $O(2^n)$ time, so we have SAT $\in$ EXP. In fact, it holds that NP $\subseteq$ EXP (easy proof omitted). The fastest known algorithms for SAT are not much better than this trivial algorithm, and run in time $2^{O(n)}$.

3 NP-completeness

We say that a language $\mathcal{A}$ is NP-complete if:

- $\mathcal{A} \in$ NP, and
- For all $\mathcal{B} \in$ NP, $\mathcal{B} \leq_P \mathcal{A}$.

That is, every problem in NP reduces to $\mathcal{A}$, or in other words deciding membership in $\mathcal{A}$ is "as hard as" the hardest problem in NP. This would imply that it is very unlikely that there exists a polynomial-time algorithm for $\mathcal{A}$, because if this were true, there would be a polynomial-time algorithm for every problem in NP, and we would have P = NP.

It is not obvious that there should exist *any* NP-complete problems, let alone any "natural" ones. The remarkable fact that there do is called the Cook-Levin theorem.

Theorem 2 (Cook-Levin theorem)**.** *SAT is* NP*-complete.*

In order to prove this theorem, we will need to understand another, equivalent, definition of the class NP, as the class of languages recognised by a *nondeterministic* Turing machine.

4 Nondeterministic Turing machines

A standard deterministic Turing machine consists of a *configuration* (h, t) comprising a head state h and a tape state t, and a *transition rule* $(h, t) \mapsto (h', t', d)$, where d is "left" or "right" and h', t' are the new head and tape states, respectively. Nondeterministic Turing machines (NDTMs) have the property that, instead of having one transition rule, they may have several – $(h, t) \mapsto \{(h'_1, t'_1, d_1), (h'_2, t'_2, d_2), \ldots, (h'_k, t'_k, d_k)\}$. We think of the machine as making all of these transitions in parallel, leading to many different potential computational paths. Note that this is an unrealistic and non-physical model of computation!

We say that an NDTM M decides a language $\mathcal{L}$ if:

- All M's computational paths halt.
- For all $w \in \mathcal{L}$, at least one path halts in the Accept state.
- For all $w \notin \mathcal{L}$, every path halts in the Reject state.

Theorem 3. *A language $\mathcal{L}$ is in* NP *if and only if it's decided by some polynomial-time NDTM.*

Proof sketch. First, suppose $\mathcal{L} \in$ NP. Then there is a certificate c for $\mathcal{L}$ of size poly(n) and a verifier V such that V accepts on input (x, c) if and only if $x \in \mathcal{L}$. To decide $\mathcal{L}$, on input x our NDTM simply guesses c nondeterministically and runs V on (x, c).

Second, suppose M is a polynomial-time NDTM that decides $\mathcal{L}$. Then, for each $x \in \mathcal{L}$, there is a computational path of length poly(n) leading to the Accept state, but for all $x \notin \mathcal{L}$, there is no such path. This serves as a certificate: the verifier takes as input (x, p), where p identifies a computational path, and just simulates the path p on input x. □

5 Proof of the Cook-Levin theorem (sketch)

We already know that SAT $\in$ NP. To show that SAT is in fact NP-complete, we need to show that every language in NP reduces to SAT. What we will show is slightly stronger: that, for any

language $\mathcal{L} \in \mathsf{NP}$, given an x, we can construct (in polynomial time) a CNF formula ϕ such that ϕ is satisfiable if and only if $x \in \mathcal{L}$.

As $\mathcal{L} \in \mathsf{NP}$, there is a polynomial-time NDTM M that decides $\mathcal{L}$. Given a description of M, we will encode this as a formula ϕ, which will evaluate to true if and only if M has an accepting path on x.

We associate a $\text{poly}(n) \times \text{poly}(n)$ tableau with each computational path of M, where row t contains the configuration of M at step t of the path. Imagine that the first cell of each row contains the head state, with the remainder containing the tape state, and let q_0 be the initial head state. Then our CNF formula is

$$\phi = \phi_{\text{cell}} \wedge \phi_{\text{start}} \wedge \phi_{\text{move}} \wedge \phi_{\text{accept}},$$

where

- ϕ_{cell} evaluates to true if all squares in the tableau are uniquely filled;
- ϕ_{start} evaluates to true if the first row is the correct starting configuration;
- ϕ_{move} evaluates to true if the tableau is a valid computation path of M (according to its transition rules);
- ϕ_{accept} evaluates to true if the tableau contains an accepting state.

So there is a satisfying assignment to ϕ if and only if there is a tableau (i.e. computational path) that accepts x. Somewhat more formally, introduce the boolean variable $c_{i,j,s}$, which evaluates to true if cell (i, j) contains symbol s. Then

$$\begin{aligned}
\phi_{\text{accept}} &= \bigvee_{i,j} c_{i,j,Accept} \\
\phi_{\text{start}} &= c_{1,1,q_0} \wedge c_{1,2,x_1} \wedge c_{1,3,x_2} \wedge \cdots \wedge c_{1,n+1,x_n} \\
\phi_{\text{cell}} &= \bigwedge_{i,j} \left(\left(\bigvee_{s} c_{i,j,s} \right) \bigwedge_{s \neq t} (\neg c_{i,j,s} \vee \neg c_{i,j,t}) \right)
\end{aligned}$$

The last of these encodes the constraint that every cell has a value, and no cell has two values. For the final constraint ϕ_{move}, we need to encode the validity of a computational path. The key insight is that this can be done using the *locality* of Turing machines. Consider a 2×3 "window" (submatrix) in a tableau. If S is the number of configuration symbols used, there are only $\text{poly}(S)$ possible windows – a fixed, finite number. However, some of these are disallowed ("illegal") because they correspond to transitions which cannot take place (e.g. a head moving two positions in one step).

The constraint that a window w is legal can be written as

$$\phi_w = \bigvee_{\text{legal windows } v} [w_{ij} = v_{ij} \text{ for all } i, j].$$

Note that this constraint is a boolean formula with a fixed, finite number of terms, and can easily be converted into CNF. It turns out that a tableau corresponds to a valid computational path if

and only if all windows are legal. Intuitively, this is because the head can only move one step at a time, so every possible transition will occur within some window. We can therefore write

$$\phi_{\text{move}} = \bigwedge_{\text{windows } w} \phi_w,$$

which completes the sketch of the proof.

Note that we've skipped some technical details here, such as needing delimiters to the left and right of the tableau; for a more detailed account, see the references from the previous lecture.

6 Some more NP-complete problems

It turns out that many, many interesting problems are known to be NP-complete. Garey and Johnson list over 300! The Cook-Levin theorem allows us to prove a given problem to be NP-complete in a much more straightforward manner than was the case for SAT. Indeed, if we can show, for some language $\mathcal{L} \in \mathsf{NP}$, that SAT $\leq_P \mathcal{L}$, then $\mathcal{L}$ is NP-complete. Here are a few examples of other NP-complete problems.

- 3-SAT: the Satisfiability problem restricted to CNF formulae with 3 variables per clause.
- Hamiltonian Path: given a graph G, does it contain a path visiting each vertex exactly once?
- Subset Sum: given a set S of n integers, is there a subset of S that sums to a "target" t?
- 3-Colouring: given a graph G, can the vertices each be assigned one of three colours, such that adjacent vertices receive different colours?

We finish with a simple example of an NP-completeness proof.

Theorem 4. Subset Sum *is* NP*-complete.*

Proof sketch. It should be obvious that Subset Sum is in NP (the certificate is simply the subset of S that sums to the target). To prove that it's NP-complete, we reduce SAT to Subset Sum. Imagine we are given a formula ϕ with clauses $C_1, \ldots, C_m$ where each clause uses at most ℓ variables. We create some $(n+m)$-digit numbers (in base $\ell+1$) that sum to a particular target t if and only if ϕ is satisfiable. These numbers have one column corresponding to each variable, and one to each clause.

- For each variable v, create two integers n_{vt} and n_{vf} which are both 1 in the column corresponding to v. n_{vt} also has a 1 digit in the columns corresponding to clauses that are satisfied by v being true, and 0 digits elsewhere. n_{vf} also has a 1 digit in the columns corresponding to clauses that are satisfied by v being false, and 0 digits elsewhere.
- For each clause C_i containing ℓ_i variables, create $\ell_i - 1$ numbers that are 1 in the column corresponding to C_i, and 0 elsewhere.

- The target t has 1 digits in all the columns corresponding to variables, and a digit equal to ℓ_i in the column corresponding to C_i.

Now, if there is a subset of integers summing to t, this must use exactly one of each of the pairs $\{n_{vt}, v_{vf}\}$ (corresponding to each variable being either true or false) to make the first n columns correct. Also, each clause of ϕ must have been satisfied by at least one variable, or the last m columns couldn't be correct. Therefore, there is a satisfying assignment to ϕ if and only if there is a subset of the numbers that sums to t. □