CS221: Computational Complexity	Prof. Salil Vadhan

Lecture 7: Completeness

10/4 — *Scribe: Chris Crick*

Contents

1 Announcements

- Scribe notes can be written using Microsoft Word and Microsoft Equation Editor, though LaTeXis preferable — it's simpler and easier to use, as well.
- Reading: Papadimitriou §8.1, §8.2 (scan §9)

2 Relations Between Resources

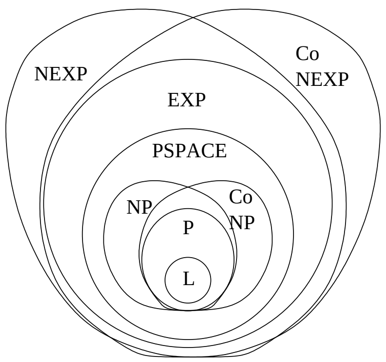

Figure 1: Time and space hierarchies

Theorem: For all proper functions f,

- $\text{TIME}(f(n)) \subseteq \text{SPACE}(f(n))$
- $\text{SPACE}(f(n)) \subseteq \bigcup_c \text{TIME}(2^{cf(n)})$

- TIME$(f(n)) \subseteq$ NTIME$(f(n))$
- NTIME$(f(n)) \subseteq$ SPACE$(f(n))$

We know that NTIME is strictly contained by PSPACE. Given an NTM M and TIME $f(n)$, we can enumerate over all the sequences of $f(n)$'s nondeterministic choices and check if it ever accepts. This uses $O(f(n))$ space, because we can reuse space in each iteration through the sequence.

In Figure 1 above, the existence of *every line* dividing and strictly containing the various complexity domains is an open question. We do, however, know that some of them have to exist, because the hierarchy theorems show us that $L \neq PSPACE$, $P \neq EXP$, and $NP \neq NEXP$.

3 Reductions

So far, we've only talked about relations *between* resources and classes. Reductions are the tool that allow us to discuss the complexity of natural problems.

"A reduces to B" $\equiv$ "A is easier than B;" in other words, if we know how to solve B within a certain complexity domain, we have a mechanism for solving A within the same bounds. The most general way to formalize this is via a Cook reduction.

DEFINITION: A *Cook reduction* from A to B is a poly-time algorithm for A using B as an "oracle" – a black box which solves B in one step. This is denoted $A \in P^B$. If such a reduction exists, it means that if $B \in P$, A must be as well.

Cook reductions are good if "easy" $\equiv P$, but sometimes we'll want "easy" to be L or NP, etc., and this kind of reduction doesn't work. For instance, it implies that its answers are closed under complement, which we don't know to be true for all complexity classes.

DEFINITION: A LOGSPACE *many-one reduction* from language A to language B is a function of $f : \Sigma^\star \to \Sigma^\star$ such that

- f is computable in LOGSPACE
- $x \in A \Leftrightarrow f(x) \in B$

PROPOSITION: If A reduces to B and B reduces to C, then A reduces to C.

PROOF: Let R_1 = reduction from A to B, and R_2 = reduction from B to C. We need to show $R = R_1 \circ R_2$. It's clear that $x \in A \Leftrightarrow R(x) \in C$, but how can we compute R in LOGSPACE? A naive algorithm would compute $R_1(x)$, then use the output y to begin computing $R_2(y)$. However, y can be polynomially big, and we would have to write it down, thus violating the LOGSPACE constraint. Here's a better algorithm:

- Compute the first symbol of $R_1(x)$
- Simulate R_2. Keep track of the location of its input cursor; at this stage, we have its first input symbol available.
- When R_2 moves its cursor (to position i), run R_1 again from the start, discarding symbols until it outputs the needed symbol (the ith).

Since the pointer keeping track of the location of R_2's input cursor requires only LOGSPACE, the whole algorithm never uses more than this. The length $|R_1(x)| \leq$poly$(|x|)$ (by the fact that SPACE$(f(n)) \subseteq \bigcup_c$ TIME$(2^{cf(n)})$).

DEFINITION: A *complexity class* $\mathcal{C}$ is closed under reductions when, if A reduces to B, $B \in \mathcal{C} \Rightarrow A \in \mathcal{C}$.

PROPOSITION: L, NL, P, NP, co-NP, etc. are all closed under reductions.

4 Completeness

DEFINITION: A language L is complete for $\mathcal{C}$ if $L \in \mathcal{C}$ and every language $L' \in \mathcal{C}$ reduces to L. Completeness says that the complexity of $\mathcal{C}$ is equivalent to that of L. For examble, $\mathcal{C} \not\subseteq P \Rightarrow L \notin P$ if L is $\mathcal{C}$-complete. Completeness can therefore be used to give evidence or proof of L's intractiability. On the other hand, $L \in P \Rightarrow \mathcal{C} \subseteq P$, which means that completeness can be used to prove something about an entire class, merely by proving it for L.

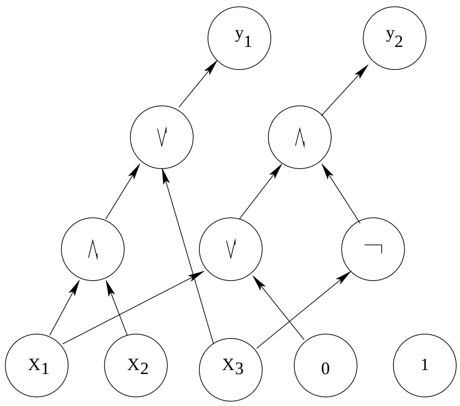

Figure 2: A boolean circuit

DEFINITION: A *boolean circuit* C is a directed acyclic graph with the following properties:

- input nodes $(x_1, \ldots, x_n)$
- internal gates $(\wedge, \vee, \neg)$
- fan-in of 2
- output nodes $(y_1, \ldots, y_n)$
- constant nodes

The circuit defines a function $C : \{0,1\}^n \rightarrow \{0,1\}^m$.

DEFINITION: The CIRCUIT-VALUE problem is defined as a decision problem on the circuit $\{C\}$, where C is built from arbitrary constants and logic gates, but no input nodes and only one output node. The decision is defined as the value of this node.

THEOREM: CIRCUIT-VALUE is P-complete.

LEMMA: Every boolean function $f : \{0,1\}^n \to \{0,1\}$ can be computed by a circuit (albeit of size 2^n).

PROOF: A CNF representation can be trivially transformed into a circuit. If fan-in is bigger than 2, then the CNF clause can be replaced with the appropriate binary tree.

PROOF: CIRCUIT-VALUE $\in P$, obviously. We can show that it is P-hard, meaning that given any $L \in P$, there is a TM M running in time n^k deciding L. We'll make a few simplifying assumptions. Without loss of generality, M is a one-work-tape TM. First it copies its input onto its work tape, and never looks at the input tape again. It writes its output on the worktape at the end. Now, given x, we want to construct a circuit C such that $C \equiv 1$ iff $M(x)$ outputs 1. How?

We create an $n^k \times n^k$ tableau, where the i axis represents time and the j axis represents space. In cell(i, j), we can construct circuit nodes to represent

- the symbol in tape position j at time i
- whether the cursor is at j at time i
- (if the cursor is at j at time i) the state of M

So at each cell location, we're using a fixed number of nodes to represent this information. We fill in constants for the bottom row of the tableau (time 0) to represent the TM input x and the initial configuration. The contents of the upper left square of the tableau represent the single output node. The key observation is that cell(i, j) *only* depends on the three cells below it: $(i-1, j-1)$, $(i-1, j)$, and $(i-1, j+1)$. These are the only Turing Machine possibilities – at each time step, either the cursor moves right, left or stays put. Thus the CIRCUIT-VALUE problem can model any computation that can be peformed on a Turing machine in polynomial time.