Complexity Theory - Tutorial

Ivan Gavran

November 29th 2016

1. STRONGLY-CONNECTED $:= \{G = (V, E) : G$ is strongly connected directed graph$\}$. Prove that STRONGLY-CONNECTED is NL-complete.
   Solution Let's show first that STRONGLY-CONNECTED $\in NL$. We offer two ways of showing this.

   - proving that $\overline{\text{STRONGLY-CONNECTED}} \in NL$. (From this, since NL = coNL the claim follows). Nondeterministically choose a pair of vertices (s, t) of the input graph and output the decision of $\overline{\text{PATH}}$ on those.
   - iterating over all pairs of vertices: we iterate over all $(u, v) \in E$. We run PATH(G, s, t). If it rejects, we reject. If the iteration was finished without rejecting, we accept.

   Next, we want to show that STRONGLY-CONNECTED is NL-hard. We do it by reducing PATH to STRONGLY-CONNECTED. Let (G, s, t) be an instance of PATH problem. We construct graph G' by adding edges so that s has an incoming edge from every other vertex and that t has an outgoing edge to every other vertex. We claim that STRONGLY-CONNECTED$(G) \Leftrightarrow$ PATH(G, s, t). Assume there is a path π from s to t in G and consider any two vertices $u, v \in G'$. Then there is a path $u - s - t - v$ Conversely, assume that there is now path between s and t in graph G. But this also means they are not connected in G', as all added edges were coming into s and going out of t, hence G' is not connected.

2. Prove $NP \neq$ SPACE(n)
   Solution We say that a complexity class C is closed under a reduction if $A \leq B \wedge B \in C \Rightarrow A \in C$. Now we want to show that NP is closed under log-space reduction, while SPACE(n) is not. We start from $R(L) = \{R(x) : x \in L\}$ and $R(L) \in NP$, with R logspace computable function. In order to see that $L \in NP$, we note that $R(x)$, for $x \in L$, can be computed in poly-time ($LSPACE \subset P$) and $R(x)$ can be decided by a nondeterministic polynomial Turing machine, Therefore, $L \in NP$.
   Now observe any language $L_1 \in SPACE(n^2)$. There is a Turing machine M deciding L_1 in quadratic space. For $x \in L$ define $\tilde{x} := x\underbrace{* * \ldots *}_{n^2-n}$, where $n = |x|$. (note that $R(x) = \tilde{x}$ is implicitly logspace computable) Those

$\tilde{x}$ form a new language, L_2. A machine that checks this language, call it M', first checks whether it has an appropriate format (last $n^2 - n$ symbols are *, this can be done in $\mathcal{O}(log|\tilde{x}|))$ space. Afterwards, machine M is simulated on the first n input symbols, and this uses $\mathcal{O}(n^2) = \mathcal{O}(|\tilde{x}|)$ space. Therefore, $L_2 \in SPACE(n)$. By the space hierarchy theorem, $\exists L_1$ such that $L_1 \in SPACE(n^2)$ and $L_1 \notin SPACE(n)$. For the defined log-space reduction R we know it holds $R(L_1) \in SPACE(n)$. Therefore, the closeness property we mentioned at the beginning does not hold.
This leads us to the conclusion $NP \neq$ SPACE(n). We don't - however - know whether one is contained in the other.

3. Assume $P = NP$. Then there is a polynomial-time algorithm for solving SAT. Find in polynomial time an explicit algorithm that outputs a satisfying assignment to Boolean formulas whenever such an assignment exists.
Solution Let $M_1, M_2, \ldots$ be a list of all polynomial-time Turing machines that take a SAT problem instance as their input. Run them in a dovetail manner: $([M_1])([M_1][M_2])([M_1][M_2][M_3]) \ldots$ where $[M_i]$ means *a next step of the machine* M_i. If a satisfying assignment exists, one of the machines would find it in the polynomial time, causing the whole computation to take polynomial time. (Whenever a machine halts, we could check whether the assignment it suggested as a correct is actually correct in polynomial time). The fact that we're simulating *gazillion* of other machines as well merely slows things down *by a polynomial factor*, independent of the input size.

4. Let A be an algorithm that's supposed to solve SAT in polynomial time (that is, find a satisfying assignment whenever one exists), but that actually fails on some SAT instance of size n. Then if someone gives you the source code of A, you can, in time polynomial in n, find a specific SAT instance that actually witnesses A's failure.
Solution If A claims that there is a satisfying assignment for formula x, then it can be found using a polynomial machine C that simulates asking A: "Is there a satisfying assignment $1x_2x_3 \ldots x_n$, and finding the correct value for the first bit (c_1). Afterwards, it would ask "Is there a satisfying assignment $a_1 1 x_3 \ldots x_n$"... Proceeding this way it would find y, an assignment that A is claiming to be satisfying for formula x. In the same way as in the proof of Lemma 2.12 (Arora, Barak), we encode the computation of machine A on the input x in formula $\phi(x) =$ "x is a SAT instance of size n on which A fails (that is, either there's a satisfying assignment A fails to find, or A outputs an assignment for x that isn't satisfying)". More formally, let $\phi(x, y, z) =$ x is a formula $\wedge ((A(x) = 1 \wedge y = C(A, x) \wedge Eval(x, y) = 0) \vee (A(x) = 0 \wedge Eval(x, z) = 1)$. When running A on the input ϕ, A would either fail, or successfully compute the solution. As we know that there must be an input x for which A fails, then succeeding on ϕ would mean finding this assignment. If A succeeds, x is our witness. If it fails - well, ϕ is the witness we were looking for.

5. We say that a function is *write-once computable* if it can be computed by an $\mathcal{O}(logn)$-space Turing machine M whose output-tape is "write once", meaning that M can either keep its head in the same position on the tape or write to it a symbol and move to the right. The used space on the output tape is not counted against M's space bound.
   On the other hand, the *implicitly logspace computable function* is defined as a function that is polynomially bounded and the languages $L_f = \{(x,i) : f(x)_i = 1\}$ and $L'_f = \{(x,i) : i < |f(x)|\}$ are in L.
   Prove that f is write-once computable if and only if f is implicitly logspace computable.
   Solution Assume f is "write-once" computable. There exists a machine M that uses this output tape. Whenever M is about to write on the output tape, it instead increases counter. Once the counter reaches i, it outputs the bit it was about to write on the output tape. It can also check whether $i \geq |f(x)|$ by seeing whether M halts before the counter reaches i. (The length of i is not a problem as i is part of the input.). Finally, to see that f is logspace bounded ($|f(x)| \leq |x|^c$), we should observe configuration graph of the computation. It is polynomially bounded (log space used on working tape), so that is the biggest possible length of the output. (note that without condition that M is write-once, we'd have a possibly bigger f) This means f is also implicitly logspace computable.
   For the other direction - if f is implicitly logspace computable - there are two machines, M_1 that recognizes $L_f = \{(x,i) : f(x)_i = 1\}$ and M_2 that recognizes $L'_f = \{(x,i) : i < |f(x)|\}$. Define machine M the following way: after receiving x as an input, run $M_1(x,1)$ and write the bit on the output tape. Afterwards, run $M_2(x,1)$ to see whether we're done or not. Continue the same way. The first time that machine M_2 rejects, we're done with $f(x)$ written on the output tape. In the process we have to have a counter for the "current bit". Its size is guaranteed to be in logspace, as - according to the definition of implicitly logspace computable function - f is polynomially bounded.

6. $M = \{0^k 1^k : k \geq 0\}$. Prove that $M \in L$.
   Solution Given an input α, a deterministic Turing machine might first check whether it consists of some zeros and some ones. (no space needed there). Afterwards, it can scan the input again, this time counting zeros and ones. The counters require logarithmic space (in the size of input.

NOTE: Sources of the problems and more problems of the same kind:

- an interesting blogpost by Scott Aaronson `http://www.scottaaronson.com/blog/?p=392`.
- `http://cse.iitkgp.ac.in/~abhij/course/theory/CC/Spring04/chap3.pdf`