

Lecture Notes 9:**NL-Completeness, Immerman-Szelepcsényi Theorem****Reading.**

- Arora-Barak § 4.3, 5.1-5.2

Last time: PSPACE-completeness, logspace computation

1 NL-Completeness

Recall that a language B is **NL**-complete if $B \in \mathbf{NL}$ and $A \leq_l B$ for all $A \in \mathbf{NL}$.

Our canonical **NL**-complete problem is

$$\text{PATH} = \{\langle G, s, t \rangle \mid \text{Digraph } G \text{ has a path from } s \text{ to } t\}.$$

Theorem 1. *PATH is NL-complete.*

Proof. As usual, there are two things to show.

$\text{PATH} \in \mathbf{NL}$. The idea is to nondeterministically guess a path from s to t . Let n be the number of vertices in the input graph G . Note that there is a path from s to t in the graph iff there is such a path of length at most n .

1. Set $v = s, \ell = 0$.
2. While $\ell \leq n$:
3. Update v to a nondeterministically chosen neighbor
4. If $v = t$, accept
5. Increment ℓ
6. Reject.

The space usage is $O(\log n)$ to maintain the identity of the current vertex v , plus $O(\log n)$ to maintain the step counter ℓ .

PATH is NL-hard. Let $A \in \mathbf{NL}$ and let N be an NTM deciding A in space $O(\log n)$. We give a logspace reduction f from A to PATH.

Define $f(x) = \langle G_{N,x}, s, t \rangle$ where

- $G_{N,x}$ is the configuration graph of N on x . That is, vertices are configurations, and there is an edge $C \rightarrow C'$ whenever the transition function of N can take configuration C to C' .
- $s = C_{\text{start}}$ and $t = C_{\text{acc}}$.

Correctness: We have that

$$\begin{aligned} x \in A &\iff \exists \text{ an accepting computation path of } N \text{ on } x. \\ &\iff \exists \text{ a path from } C_{\text{start}} \text{ to } C_{\text{acc}} \text{ in } G_{N,x} \\ &\iff \langle G_{N,x}, s, t \rangle \in \text{PATH} \end{aligned}$$

Space usage: Each configuration C (a vertex of $G_{N,x}$) takes $O(\log |x|)$ space to write down. (Note that the big O hides a dependence on the machine N , but that is fixed independent of x once the problem A is specified.)

Thus, we can write down (in a write-once fashion) the adjacency matrix of $G_{N,x}$ by just checking if each pair (C, C') is consistent with N 's transition function. $\square$

2 The Verifier View of NL

As with NP, we can formulate an alternative view of NL as the class of languages that admit logspace *deterministic* verifiers. However, we have to be careful about how we define a verifier.

Theorem 2. A language $A \in \mathbf{NL}$ if and only if there exists a logspace TM V with a read-once “certificate tape” and a polynomial p such that for all $x \in \{0, 1\}^*$,

$$x \in A \iff \exists u \in \{0, 1\}^{p(|x|)} \quad V(x, u) = 1.$$

Here, x is given to V on its input tape while u is given on its (again, read once) certificate tape.

Proof. For the “only if” direction: Let $A \in \mathbf{NL}$ be decided by a logspace NTM N . Then $x \in A$ iff there exists a sequence of nondeterministic choices leading N to accept on input x .

Use a certificate u to encode such a sequence of nondeterministic choices. We construct a verifier $V(x, u)$ that simulates N on x using u as the nondeterministic choices, accepting iff the simulation accepts.

For the “if” direction: Suppose A has a logspace verifier V .

First idea: using nondeterminism, guess a certificate u on some work tape, and then verify it. The problem is the same one we encountered with composing logspace reductions: Writing down u requires polynomial space, not logspace.

Better: Guess a certificate u bit-by-bit on the fly as the verifier needs them. This is ok because the verifier only needs read-once access to the certificate, so there's no need to record previous bits of u anywhere explicitly. $\square$

3 Immerman-Szelepcsényi Theorem: $NL = coNL$

We'll now prove a remarkable result about logspace computation, which one can view as the space-bounded analog of $NP = coNP$.

Theorem 3. $NL = coNL$.

The way we'll prove this is by showing that the $coNL$ -complete problem $\overline{PATH}$ is also contained in NL . Pause for a moment to think about how remarkable that is. While it's straightforward to check the existence of a path with a logspace verifier, this is saying that there's a similarly efficient way to certify the *non*-existence of such a path.

Here's the intuition for the proof. Say I want to convince you that in a digraph G with n vertices, there is no path from s to t . I can do this by convincing you of the following two statements:

1. There are exactly m_n distinct vertices reachable from s by paths of length $\leq n$.
2. The destination vertex t is *not* one of those m_n vertices.

Now how would I actually convince you of these statements? Let's start with the second one. I can do that by showing you m_n vertices *other* than t which are reachable from s by paths of length $\leq n$. Now how about the first one? The idea we'll use for this one is "inductive counting." For each $k = 0, \dots, n-1$, I'll show you that "if there are m_k vertices reachable by paths of length $\leq k$, then there are m_{k+1} vertices reachable by paths of length $\leq k+1$."

Proof. Let G be a graph with n vertices. For each $k = 0, \dots, n$, let C_k be the set of vertices reachable from s within $\leq k$ steps. Then

$$\langle G, s, t \rangle \in \overline{PATH} \iff t \notin C_n \iff \exists m_0, \dots, m_n \text{ s.t. } \left\{ \begin{array}{l} |C_0| = m_0 (= 1) \\ \text{If } |C_0| = m_0 \text{ then } |C_1| = m_1 \\ \vdots \\ \text{If } |C_{n-1}| = m_{n-1} \text{ then } |C_n| = m_n \\ \text{If } |C_n| = m_n \text{ then } t \notin C_n. \end{array} \right.$$

That is, certifying all of the statements on the right is equivalent to certifying the statement on the left. Let us now see how to describe and verify these certificates.

1. $|C_0| = m_0$. This one is easy. Since $|C_0| = \{s\}$, the only possibility is to take $m_0 = 1$ which the verifier can immediately check.
2. $|C_n| = m_n \implies t \notin C_n$. Define u_v to be a certificate for the fact that there is a path from s to v of length $\leq n$ (i.e., the list of vertices along this path). Take the certificate to be $(u_{v_1}, u_{v_2}, \dots, u_{v_{m_n}})$ where the vertices are sorted so that $v_j < v_{j+1}$ and $t \neq v_j$ for every j .

To check this certificate, scan it once while checking that a) the vertices are indeed sorted, b) m_n distinct vertices are hit, c) t does not appear in the list, and d) the individual path certificates all check out.

3. $|C_k| = m_k \implies |C_{k+1}| = m_{k+1}$. Here, the certificate will take the form $(w_1, w_2, \dots, w_n)$ where each w_i itself is a certificate for either “ $v_i \in C_{k+1}$ ” or “If $|C_k| = m_k$, then $v_i \notin C_{k+1}$.” This suffices because the verifier can count to ensure there are exactly m_{k+1} “ $\in C_{k+1}$ ” certificates while verifying each individual certificate.

Now let's see what these individual certificates look like.

- (a) To certify $v_i \in C_{k+1}$: Just exhibit a path from s to v_i of length $\leq k + 1$.
- (b) To certify “If $|C_k| = m_k$, then $v_i \notin C_{k+1}$ ”: Similar to Case 2, let u_v be a certificate for the fact that there is a path from s to v of length $\leq k$. Our certificate is now $(u_{v_1}, \dots, u_{v_{m_k}})$, where again the vertices are sorted so that every $v_j < v_{j+1}$. Checking this certificate is similar, but now we check that for every target vertex v_j in this list, there is no edge $v_j \rightarrow v_i$.

□

