CS 6810: Theory of Computing **Spring 2026**

Lecture 8: February 19, 2021

Lecturer: Eshan Chattopadhyay *Scribe: Caroline Sun*

1 Review of Space Complexity

In lecture 7, we discussed **PSPACE** completeness by discussing the following two theorems.

1. **TQBF** is **PSPACE**-complete.

   Note that **PSPACE** itself is so large that it captures **P** and **PH** $\mathbf{P} \subseteq \mathbf{PH} \subseteq \mathbf{PSPACE}$. However, similar to the reason that we still do not know $\mathbf{P} = \mathbf{NP}$, we also don't know if $\mathbf{P} = \mathbf{PSPACE}$. In particular, we aren't capable of showing if there is a polynomial-time algorithm deciding **TQBF**. If **TQBF** is **PSPACE**-complete and this polynomial-time algorithm deciding **TQBF** exists, then the Polynomial Hierarchy will collapse.

2. (**Savitch**) For any space-constructible $S : \mathbb{N} \to \mathbb{N}$ with $S(n) \geq \log n$, $\mathbf{NSPACE}(S(n)) \subset \mathbf{DSPACE}(S(n)^2)$. Knowing Savitch and the trivial relation $\mathbf{PSPACE} \subseteq \mathbf{NPSPACE}$, we can naturally infer that $\mathbf{NPSPACE} = \mathbf{PSPACE}$, .

Savitch solved the **P** vs **NP** question for space complexity. Another question we are interested in is the **NP** vs **coNP** for space complexity. Therefore, we will explore the relation between complexity classes $\mathbf{NSPACE}(S(n))$ and $\mathbf{coNSPACE}(S(n))$ in this lecture. By the end of the lecture, you will see that these classes are equivalent.

2 Equivalence of NSPACE$(S(n))$ and coNSPACE$(S(n))$

We first focus on the log-space $\mathbf{L}, \mathbf{NL}$, where **L** is the class of languages accepted by a deterministic TM using $O(\log(n))$ space, and **NL** is the analogue for nondeterministic TMs. These machines use little space, i.e. $\log(n)$. We will show that if $\mathbf{NL} = \mathbf{coNL}$, then it follows $\mathbf{NSPACE}(S(n)) = \mathbf{coNSPACE}(S(n))$.

2.1 NL completeness

Recall that for the language in **L** or **NL**, the space that their corresponding Turing machines use doesn't count the used space of the input tape. This makes it possible for Turing machines to use less space than the input size. Savitch tells us that $\mathbf{NL} \subseteq \mathbf{DSPACE}(\log^2(n))$, making this chain $\mathbf{NL} \subseteq \mathbf{DSPACE}(\log^2(n)) \subseteq \mathbf{coNSPACE}(\log^2(n))$. Simplifying it to only terms on the two ends, we get $\mathbf{NSPACE}(S(n)) \subseteq \mathbf{coNSPACE}(S^2(n))$, where $S(n) = \log(n)$. Therefore, proving $\mathbf{NSPACE}(S(n)) \subseteq \mathbf{coNSPACE}(S(n))$ is really just solving a more fine-grained question that asks about the necessity of this "square" in space usage.

The overall plan is to first find the hardest problem L in **NL**, proving it is **NL**-complete, and show $\bar{L}$, which is **coNL**-complete, can be decided by an algorithm using $\mathbf{NSPACE}(\log(n))$ space, eventually proving $\mathbf{NL} = \mathbf{coNL}$.

Question 1. *What is the hardest problem(s) in* **NL***?*

The hardest problem is the "reachability of two nodes on a directed graph".

$$s\text{-}t\textbf{-PATH} = \{\langle G, s, t\rangle : \text{There is a path from node } s \text{ to node } t \text{ in the directed graph } G \text{ of } n \text{ nodes.}\}$$

2.1.1 s-t-PATH $\in$ NL

Claim 2.1. s-t-**PATH** $\in$ **NL**

Proof. We create a nondeterministic TM M as follows: We let the machine start a "nondeterministic walk" at s, and make it store the index of the node it is at and maintain a counter for the length of the walk. It rejects when the node has no more neighbors or when the counter reaches n. Otherwise, it selects a neighbor of the node non-deterministically, overwrites the index of the current node, and increments the walk size counter by one, accepting when the latest encountered node is t.

This algorithm is $\in$ **NSPACE**$(O\log(n))$, because each walk stores just one counter and the index of the latest node, each of which takes $\log(n)$ bits. □

Note that in the proof, we need the counter to keep track of the state of each walk without explicitly storing all the nodes we already visited, as in a conventional graph search algorithm.

2.1.2 s-t-PATH is NL-hard

Definition 2.2. *A language S is* **NL***-complete if*

1. $S \in \mathbf{NL}$
2. $\forall S' \in \mathbf{NL}, S' \leq_l S$.

Definition 2.3. *(log-space reduction)*

S' is log-space reducible to S, denoted $S' \leq_l S$, if there is a log-space computable function $f : \{0,1\}^* \to \{0,1\}^*$, such that $x \in S'$ iff $f(x) \in S$ for every $x \in \{0,1\}^*$.

f is log-space computable if on any input $x \in \{0,1\}^*$, any index $i < |x|$, the i^{th} bit of $f(x)$, denoted $f(x)_i$, is computable in log-space.

To convince ourselves that Definition 2.2 of **NL**-complete is useful, we need to show that if we can solve an **NL**-complete language S' in log-space, then we can solve all problems in **NL** in log-space (like what we did with polynomial-time Karp reduction). Below, we will validate exactly this statement.

Claim 2.4. *Suppose S is* **NL***-complete and there is a log-space deterministic algorithm $\mathcal{A}_s$ for S. Then,* $\mathbf{L} = \mathbf{NL}$.

Proof. Suppose some $S' \in \mathbf{NL}$. Given $S' \leq_l S$, we know that there exists a log-space computable f, such that $x \to f(x)$ and S is solvable by some deterministic algorithm $\mathcal{A}_s$.

We want to construct an algorithm for S', by mapping its input x to $f(x)$, using the log-space computable f, and output $\mathcal{A}_s(f(x))$, and then prove this algorithm to be using only log-space.

It is not trivial that this algorithm uses only log-space, because simply writing the log-space reduction output $f(x)$ would take poly(n) space. Here, the size of $f(x)$ is bounded by the polynomial time complexity of f. Therefore, in stead of explicitly writing down $f(x)$ on a work tape, we want to utilize the definition of the log-space computable function so that during the simulation of $\mathcal{A}_s(f(x))$, the Turing machine reads one bit at a time. whenever there are queries for the ith bit of $f(x)$,

we rerun f using the same $\log(n)$-sized space on a work tape to compute this bit $f(x)_i$Note that generating one bit takes up only space of size $\log(|f(x)|)^i = O(\log(\text{poly}(|x|))) = O(\log(n))$ due to the definition of log-space computable function. Therefore, the two parts of the algorithm deciding S' –"procedurally" regenerating $f(x)$ and running $A_s(f(x))$ takes $O(\log(n))$ – both take up only $O(\log(n))$ space. This makes the algorithm use $O(\log(n))$ space in total. □

Note that we are not using polynomial-time reduction but the log-space reduction, because **NL** is a weaker complexity class than **P**. However, the polynomial-time reduction is stronger than log-space reduction, making it only capable of proving **NL** $\subseteq$ **P**, given a log-space algorithm that decides a **NL**-complete language. However, this is not telling us more information than what we have already known – that **NL** is a subset/subclass of **P**.

The proof above is a more general version of the lemma below.

Lemma 2.5. *If f and g are log-space computable, then $f \circ g$ is also log-space computable.*

We can make a proof of the general version that is similar to the one above. The detailed proof is on page 85 of Arora and Barak [1].

Claim 2.6. *For any $W \in$ **NL**, $W \leq_l$ s-t-**PATH**.*

Proof. Let NDTM M solve W in $c\log(n)$ space. We want to construct a log-space computable function f that maps from the input of W, x, to a graph $G_{M,x}$ such that there exists a path from s to t iff $\langle G_{M,x}, s, t\rangle \in$ s-t-**PATH**.

$$x \xrightarrow{f} \langle G_{M,x}, s, t\rangle$$

We construct this graph $G_{M,x}$ as the configuration graph of M, where s is the starting configuration node and t is the accepting configuration node. Now, we want to show that this function f is log-space computable, meaning that generating any bit of $f(x)$ takes log-space. The input of $f(x)$ is x, i and it should output $f(x)_i$ This bit could be part of the encoding of the configuration graph $G_{M,x}$, the starting configuration s, or the accepting configuration t. If it happens to be part of the latter two, which are fixed, constant-sized strings, they can be hard-coded into your function f, so retrieving a single bit only takes constant time. In the scenario where the bit is part of $G_{M,x}$, which can be encoded as an adjacency matrix, this bit could decide if there is an edge pointing from configuration C to C'. This bit can also be generated using log-space, because the Turing machine M uses $c\log(n)$ space, making each configuration of $\log(n)$ space. To evaluate this bit, you can write both configurations and apply the transition function to check the existence of the edge, a process that takes log-space to write both configurations. In conclusion, $f(x)_i$ generation takes up only log-space, and we constructed the log-space reduction as desired.

□

Putting claim 2.1 and claim 2.6 together, we conclude that s-t-**PATH** is **NL**-complete.

2.1.3 s-t-PATH is NL-complete

If a language L is **NL**-complete, then its complement $\bar{L}$ is trivially **coNL**-complete, like what we proved for **SAT** (**NP**-complete) and its complement **TAUTOLOGY** (**coNP**-complete). Applying this theorem here, we can show that $\overline{s\text{-}t\text{-}\textbf{PATH}}$ is **coNL**-complete.

Theorem 2.7. *If there is a $O(\log(n))$ space nondeterministic algorithm for $\overline{s\text{-}t\text{-}\textbf{PATH}}$ (which is **coNL**-complete), then it follows that **NL** = **coNL**.*

2.2 $\mathbf{NSPACE}(S(n)) = \mathbf{coNSPACE}(S(n))$

We didn't have time in the lecture to prove theorem 2.7, so let's temporarily assume it is true and explore how we may use it to derive corollary 2.8.

Corollary 2.8. $\mathbf{NSPACE}(S(n)) = \mathbf{coNSPACE}(S(n))$*, for all 'nice'* $S(n) \geq \log n$.

Proof. We will utilize the strategy of "padding". Suppose $L \in \mathbf{NSPACE}(S(n))$, for some $S(n) \geq \log(n)$. Then there is an NDTM M uses $O(S(n))$ space and computes L. We define a new language $L_{\text{pad}} = \{x\#0^{2^{S(|x|)}} : x \in L\}$. Observe that $L_{\text{pad}} \in \mathbf{NL}$, because the algorithm deciding it can just run M on x and takes $S(|x|)$ space, which is approximately log of the padded input size $n = 2^{S(|x|)} + |x|$. Given $\mathbf{NL} = \mathbf{coNL}$ from theorem 2.7 and $L_{\text{pad}} \in \mathbf{NL}$, we can derive $L_{\text{pad}} \in \mathbf{coNL}$, which is equivalent to $\overline{L_{\text{pad}}} \in \mathbf{NL}$.

Now, we want to show that $\bar{L} \in \mathbf{NSPACE}(S(n))$. Knowing that $\overline{L_{\text{pad}}} \in \mathbf{NL}$, we can tell there exists a NDTM M' that decides $\overline{L_{\text{pad}}}$ and it runs in $O(\log(n)) = O(\log(2^{S(|x|)+|x|})) = O(S(|x|))$ space, given an input of the form $x\#0^{2^{S(|x|)}}$. Then, We can use it to compute $\bar{L}$ in log-space as follows: given an input x, we simulate running M' on $x\#0^{2^{S(|x|)}}$, but we don't actually write down the redundant trailing 0s. Instead, we build the algorithm to answer 0 whenever M' queries an out-of-bound bit of x. This makes the algorithm use the same amount of space, $O(S(|x|))$, as M' on work tapes. From this, we can infer that the $\bar{L} \in \mathbf{NSPACE}(S(n))$, and correspondingly $L \in \mathbf{coNSPACE}(S(n))$, making $\mathbf{NSPACE}(S(n)) \subseteq \mathbf{coNSPACE}(S(n))$. The other direction can be proved in the same way by switching $\mathbf{NSPACE}(S(n))$ and $\mathbf{coNSPACE}(S(n))$. □

References

[1] S. Arora and B. Barak, *Computational Complexity: A Modern Approach*. 440 W. 20th St. New York, NY, United States: Cambridge University Press, April 2009.