Recitation 10: L, NL, and Logspace Reductions

Logspace Reductions

The question of $L \stackrel{?}{=} NL$ is still open. How do we answer it? We can start by borrowing the tools developed for understanding P v. NP and adapt them to L vs. NL. One such tool is "completeness," where NL-complete languages are in some ways the most difficult languages in NL.

Our previous definition of completeness used polynomial time reducibility, but since NL is contained in P that would make all languages in NL (except $\emptyset$ and Σ^*) reducible to one another. Instead we will use *log space reducibility*, denoted $\leq_L$.

Machines that compute log space reductions are deterministic Turing machines with three tapes: a read-only input tape; a write-only output tape; and a read/write work tape that contains at most $O(\log n)$ symbols. They are called *log space transducers*, shown in Figure .

Note that the output tape may contain a polynomial number of symbols!

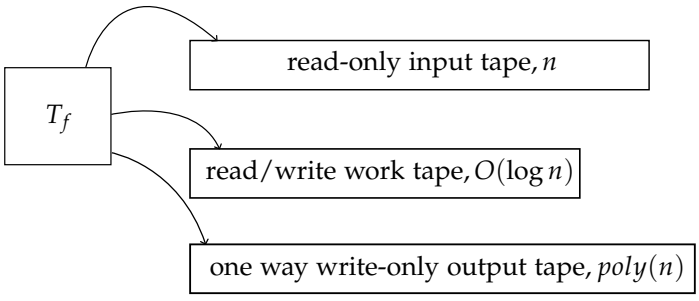

Figure 1: Figure of a log space transducer.

Theorem 1. *If $A \leq_L B$ and $B \in L$, then $A \in L$.*

Proof. Given that $A \leq_L B$, this means there exists a log-space transducer T_f which computes the reduction f. Furthermore, let M_B be the log-space Turing Machine that decides B. To show that $A \in L$ we will construct a log-space Turing Machine M_A that decides A.

A naive construction of M_A would be to have M_A simulate T_f on input w and store its output, $f(w)$, on the work tape. Then it would

simulate M_B on $f(w)$ and accept if M_B accepts. Otherwise, it rejects. The issue is that $f(w)$ can have length polynomial in the input, and our work tape only has $O(\log n)$ space.

Instead, we construct M_A such that it simulates M_B on $f(w)$, but it recomputes the symbols of $f(w)$ on an as-needed basis. When M_A simulates M_B it keeps track of where M_B's head would be on $f(w)$. Each time M_B's head moves, that means it sees a different symbol of $f(w)$. In order for the simulation to continue, M_A must know what that symbol is, so it simulates T_f on w from the beginning. But importantly, it discards all of the output except for the desired symbol. Now that M_A knows the symbol under M_B's head, the simulation can proceed. This entire process is illustrated in Figure . Lastly, if M_B accepts then accept. Otherwise, reject.

The "recomputation trick" might seem strange because the algorithm is repeating a lot of work! That's rather inefficient, no? Certainly it's inefficient in terms of time, but that's not what we're optimizing for, we're trying to minimize space!

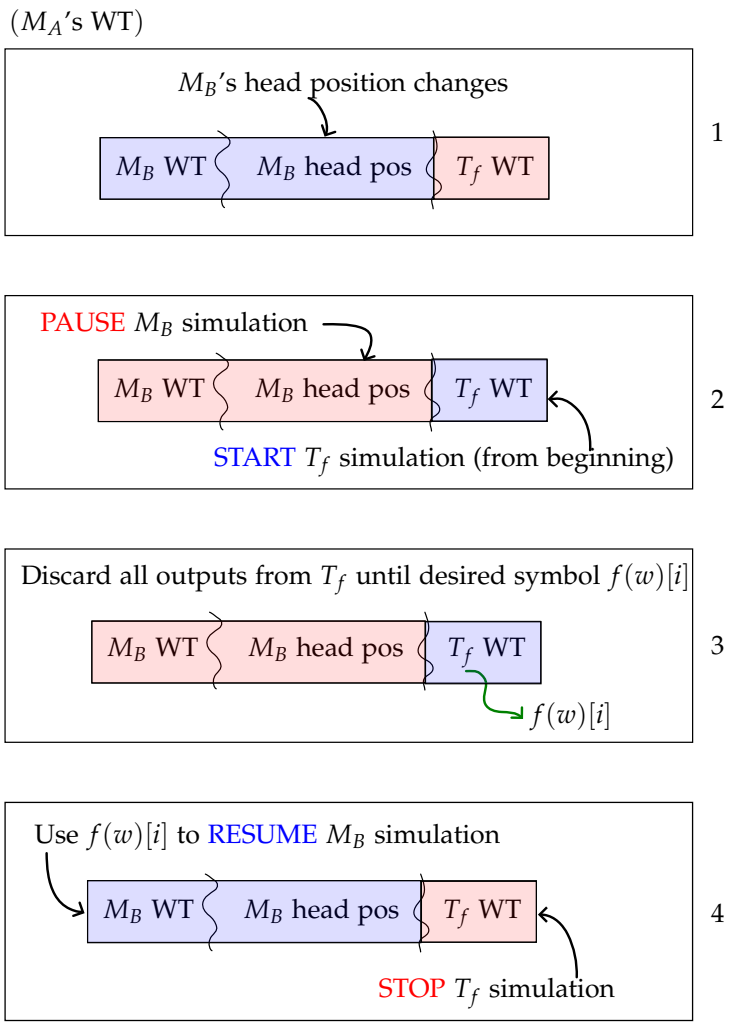

Figure 2: Illustration of the "recomputation" trick on M_A's working tape (WT). A section of M_A's WT is used for the simulation of M_B on $f(w)$, and another section is used for the simulation of T_f on w. Parts of the tape that are active are colored blue, and those that are inactive are colored red.

[1-2] When M_B's head position changes, the simulation for M_B is paused and the simulation for T_f on w starts from the beginning.

[3] All outputs from the simulation of T_f are discarded until the desired symbol, $f(w)[i]$, is obtained.

[4] Use that symbol to resume the M_B simulation and stop the T_f simulation.

At any point during the simulation, M_A's working tape will contain the contents of M_B's and T_f's working tapes, a counter keeping track of M_B's head position, and a single symbol of $f(w)$, all of which take logarithmic space to store. □

Theorem 2 (transitivity of logspace reductions)**.** *If $A \leq_L B$ and $B \leq_L C$, then $A \leq_L C$.*

Proof. Since $A \leq_L B$ there exists a log-space transducer T_f which computes the reduction f, where for all strings w,

$$w \in A \iff f(w) \in B.$$

Similarly, since $B \leq_L C$ there exists T_g computing the log-space reduction g, where for all strings w,

$$w \in B \iff g(w) \in C.$$

To show that $A \leq_L C$, we will construct a log-space transducer $T_{g \circ f}$ that computes $g \circ f$. This is because for input w,

$$w \in A \iff g(f(w)) \in C.$$

The machine $T_{g \circ f}$ cannot simply run T_f and then T_g on the input because that would involve writing $f(w)$ on the working tape, but $f(w)$ may be polynomial in length. Using the "recomputation trick" described in the proof of Theorem 1, we will instead recompute the symbols of $f(w)$ on an as-needed basis.

On input w, $T_{g \circ f}$ works by simulating T_g on $f(w)$ without storing the entirety of it on the working tape. The machine does this by keeping track of T_g's head position on $f(w)$. Every time a new symbol is needed, we fire up T_f and run it on w from the beginning while discarding the output until we get the desired symbol in $f(w)$. The working tape of $T_{g \circ f}$ will store the contents of T_g and T_f's working tapes, a counter for the head position of T_g, and a symbol of $f(w)$, which takes up logarithmic space. □

Strongly-Connected *is NL-complete*

Definition 1. *For a directed graph G, if for every pair of vertices $u, v \in V(G)$ there is a path from u to v and a path from v to u, then G is said to be* ***strongly connected****.*

We can then define the language Strong-Conn as follows,

$$\text{Strong-Conn} = \{\langle G \rangle : G \text{ is a strongly connected directed graph}\}.$$

Theorem 3. Path $\leq_L$ Strong-Conn.

Proof. Construct a log-space reduction f from PATH to STRONG-CONN, $f(\langle G, s, t\rangle) = \langle G'\rangle$. The idea is to construct G' by making a copy of G and adding some additional edges such that if there exists an s-t-path in G, then we can use this path to get between any two vertices in G'.

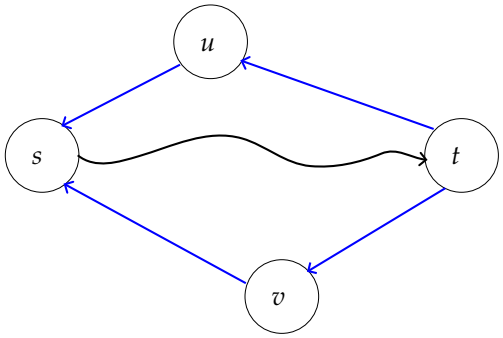

Figure 3: Construct G' by adding additional edges, colored blue, to the graph G and using the s-t-path in G (if it exists) to get between any two vertices.

Concretely, G' has the same vertices as G and every edge in G is also in G'. For every vertex $v \in V(G)$, create edges $t \to v$ and $v \to s$ in G'. This reduction is computable in logarithmic space because it copies the graph G from the input to the output tape. Then it steps through the vertices of G and puts two new edges $(v \to s, t \to v)$ onto the output tape for each v.

To prove correctness, if $\langle G, s, t\rangle \in$ PATH then there is an s-t-path in G (denoted $s \rightsquigarrow t$). Now given any vertices $u, v \in V(G')$ there are paths

$$u \to s \rightsquigarrow t \to v,$$
$$v \to s \rightsquigarrow t \to u.$$

Therefore, G' is strongly connected and $\langle G'\rangle \in$ STRONG-CONN.

For the other direction, if $f(\langle G, s, t\rangle) = \langle G'\rangle \in$ STRONG-CONN, then by definition there exists a path from s to t in G'. Furthermore, this path is made of edges that were originally in G. To see this, note that edges added as part of the reduction are either directed into s or out from t. These edges cannot be included in an s-t-path because they would be part of a loop starting and ending at s or t, which can always be excluded. Therefore, G has a path from s to t and we can conclude that $\langle G, s, t\rangle \in$ PATH. □

Theorem 4. STRONG-CONN $\in NL$.

Proof. Construct a nondeterministic Turing machine N that decides STRONG-CONN in log space. On input $\langle G\rangle$, M iterates *deterministically* over all pairs of vertices u, v in G. For each pair it *nondeterministically*

guesses a path of length at most $|V|$ from u to v vertex-by-vertex. **Reject** if any two vertices do not have a path connecting them. Otherwise, **accept**.

If the graph is strongly connected, then some sequence of guesses will find the path for each pair of vertices, so N accepts. Otherwise, no sequence of guesses will succeed for a pair of vertices, so N rejects. Keep in mind that we only guess a path of length at most $|V|$ to keep our number of guesses finite.

Keeping track of vertices u, v requires two pointers which takes logarithmic space to store. Furthermore, when guessing the path vertex-by-vertex all that is stored is a pointer to the current vertex as well as counter keeping track of the path length. Therefore, N runs in log space. □

Corollary 1. STRONG-CONN *is NL-complete.*

NL = coNL

A major open question in complexity theory is whether *NP = coNP*. It is widely believed that these two complexity classes are different, though we do not know how to prove this. Unlike the polynomial time regime, however, it turns out that in the log space regime, we actually know how to prove that the classes *NL* and *coNL* are equal!

Our starting point will be the following theorem that we showed in class.

Theorem 5. $\overline{PATH} \in NL$.

Using this fact, let us show that $PATH$ is $coNL$-complete.

Theorem 6. *$PATH$ is $coNL$-hard.*

Proof. We need to show that $PATH \in coNL$, and that $PATH$ is $coNL$-hard.

- $PATH \in coNL$. From $\overline{PATH} \in NL$, we get that $PATH \in coNL$.
- $PATH$ is $coNL$-hard.

  Equivalently, we'd like to show that $\overline{PATH}$ is NL-hard. It suffices to show that $PATH \leq_L \overline{PATH}$. To see why this is sufficient, consider an application of Theorem 2: $A \leq_L PATH$ and $PATH \leq_L \overline{PATH}$ implies $A \leq_L \overline{PATH}$ for all $A \in NL$. But now $PATH \leq_L \overline{PATH} \iff \overline{PATH} \leq_L PATH$ (to see this, think about what the transducer does to instances inside and outside $PATH$). But since we showed in class that $\overline{PATH} \in NL$, this means that there in fact does exist a logspace reduction from $\overline{PATH}$ to $PATH$, and we are done!

□

Now, finally, we can show that $NL = coNL$.

Corollary 2. *$NL = coNL$.*

Proof. We need to show that every $A \in NL$ is also in $coNL$, and that every $B \in coNL$ is also in NL.

For any $A \in NL$, we know that $A \leq_L PATH$, which is also $coNL$-complete, so $A \in coNL$ as well. By the exact same reasoning, we also know that any $B \in coNL$ satisfies $B \leq_L PATH$ which is NL-complete, so $B \in NL$. □