Recitation 9: Space Complexity

In this recitation we'll be gaining familiarity with the space complexity classes: *PSPACE*, *NPSPACE*, *L*, and *NL*. So far in this class, we've focused on constructing Turing machines that run within a specific *time* bound. The techniques we used sometimes traded space for time (i.e. memoization). Now we're adjusting our perspective and instead considering *space* bounds!

Space Complexity Classes

Define $SPACE(s(n))$ to be the set of languages decided by an $O(s(n))$ space *deterministic* Turing machine. Likewise, define $NSPACE(s(n))$ to be the set of languages decided by an $O(s(n))$ space *nondeterministic* Turing machine.

Definition 1.

$$PSPACE = \bigcup_k SPACE(n^k)$$
$$NPSPACE = \bigcup_k NSPACE(n^k)$$

We know that: $L \subseteq NL = coNL \subseteq P \subseteq NP \subseteq PSPACE = NPSPACE$.

The first equality, $NL = coNL$, is by Immerman-Szelepcsényi '87. The second, $PSPACE = NPSPACE$, is by Savitch '70.

For *sublinear* space complexity bounds, the machine does not have enough space to store the entire input. Therefore, we adjust our computation model to be a Turing machine with a read-only input tape and a read/write work tape. The amount of space used is the number of cells used on the work tape. An example of a logarithmic space Turing machine is shown in Figure 1. We can now use this new computation model to define the classes *L* and *NL*.

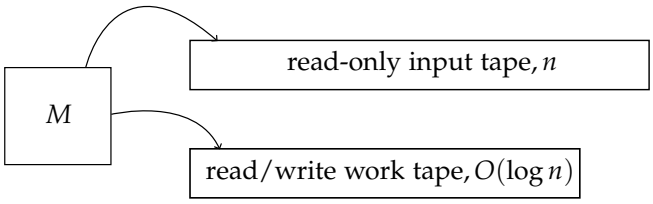

Figure 1: 2-tape Turing machine that uses logarithmic space.

Definition 2.

$$L = SPACE(\log n)$$
$$NL = NSPACE(\log n)$$

The reason why logarithmic space is interesting is because it contains problems that can be solved using a constant number of pointers (that point into the input) and counters (counting up to a polynomial in the length of the input).

Seeing how we're defining sublinear space bounds, a natural question might be why we don't have something similar for time? Turns out we do! Note that with a sublinear time bound, the machine doesn't have enough time to read the input. So when considering sublinear time we're generally talking about *approximation* algorithms.

Generalized Geography

Geography is a game where two players take turns naming locations. Each location named must start with the last letter of the previous location and locations cannot be repeated. So if Alice and Bob are playing the game and Alice says "Boston," then Bob can say "Naples," Alice can then say "Shanghai," so on and so forth. A player *loses* when they are unable to respond, which means the other player wins.

This game can be modeled as a directed graph! Each vertex in the graph corresponds to the name of a location. For two locations l_i and l_j, if l_j starts with the same letter that l_i end with, then there is a directed edge $l_i \to l_j$. Alice and Bob take turns selecting vertices in the graph to produce a path. If a player ever ends up in a state where they cannot extend the path without repeating a previously selected vertex, then they lose and the other player wins.

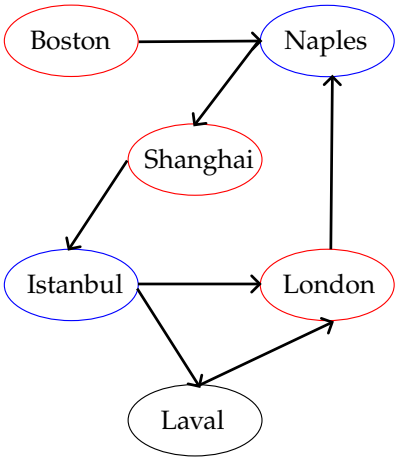

Figure 2: Geography game modeled as a directed graph, players Alice and Bob take turns selecting vertices. Alice is represented as red and Bob is represented as blue.

The game played is: Boston, Naples, Shanghai, Istanbul, London. After Alice selects London, Bob loses because he cannot select any other vertices.

The game Generalized-Geography (GG) has the same rules, except Alice and Bob are now playing on an *arbitrary* directed graph G from the starting vertex s. We say that a player has a *winning strategy* if they can always win even if the other player plays optimally. In other

words, Alice has a winning strategy if regardless of what Bob does, Alice can always counter Bob's moves to force a win.

$$\text{GG} = \{\langle G, s\rangle : \text{Alice has a winning strategy starting from } s\}$$

Theorem 1. GG $\in$ *PSPACE*

It's also worth thinking about this game in terms of quantifiers. There exists a first move for Alice, such that whatever the first move of Bob, Alice can counter it, whatever the second move of Bob, Alice can counter it, ... all the way until Alice counters Bob's last move.

The main idea is to reframe "Alice has a winning strategy" into the equivalent statement "There exists a move that Alice can make such that Bob does not have a winning strategy." This gives us the following recursive algorithm:

> M: On input $\langle G, s\rangle$:
>
> 1. If s has no outgoing edges, Alice cannot win, so **Reject**.
> 2. For every vertex v such that there exists edge $s \to v$ in G:
>   i. Remove all edges that include s in G to form G'.
>   ii. Recursively call M on $\langle G', v\rangle$.
> 3. If all of the calls accept, **Reject**. Otherwise (if at least one call rejects), **Accept**.

Players take turns selecting vertices in the game, so the recursive call at Step 2.ii shifts the perspective from Alice to Bob. The value returned from each call answers whether *Bob has a winning strategy given that Alice selected vertex v*. If all of the calls accept, that means Alice cannot select any vertex to prevent Bob from having a winning strategy, so she does not have a winning strategy. If at least one call rejects, that means there is a way for Alice to select a vertex to prevent Bob from having a winning strategy, so she has a winning strategy.

To show that GG is in *PSPACE*, M must run in polynomial space. This algorithm is a depth first search, so the space used depends on what is stored on the recursion stack. The recursion depth is $O(|V|)$ because we do not allow vertices to be repeated and at each level of the recursion we keep track of one additional vertex, which means the recursion stack uses a linear amount of space. We also need to store original graph G which also takes a linear amount of space. Therefore, M uses linear space.

Theorem 2. GG *is PSPACE-hard.*

Proof. We proved this during lecture via polynomial time reduction from *PSPACE*-complete problem Formula-Game. □

Corollary 1. GG *is PSPACE-complete.*

L and NL

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 5.

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

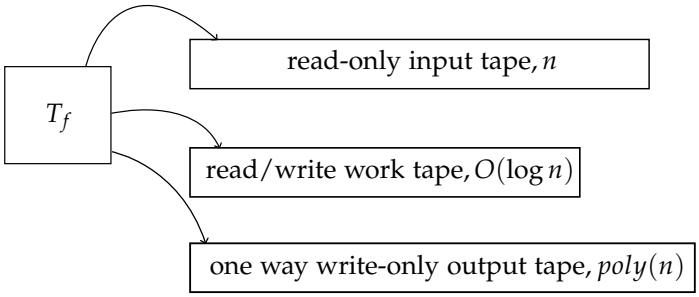

Figure 3: Figure of a log space transducer.

Theorem 3. *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.

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

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!

proceed. This entire process is illustrated in Figure 4. Lastly, if M_B accepts then accept. Otherwise, reject.

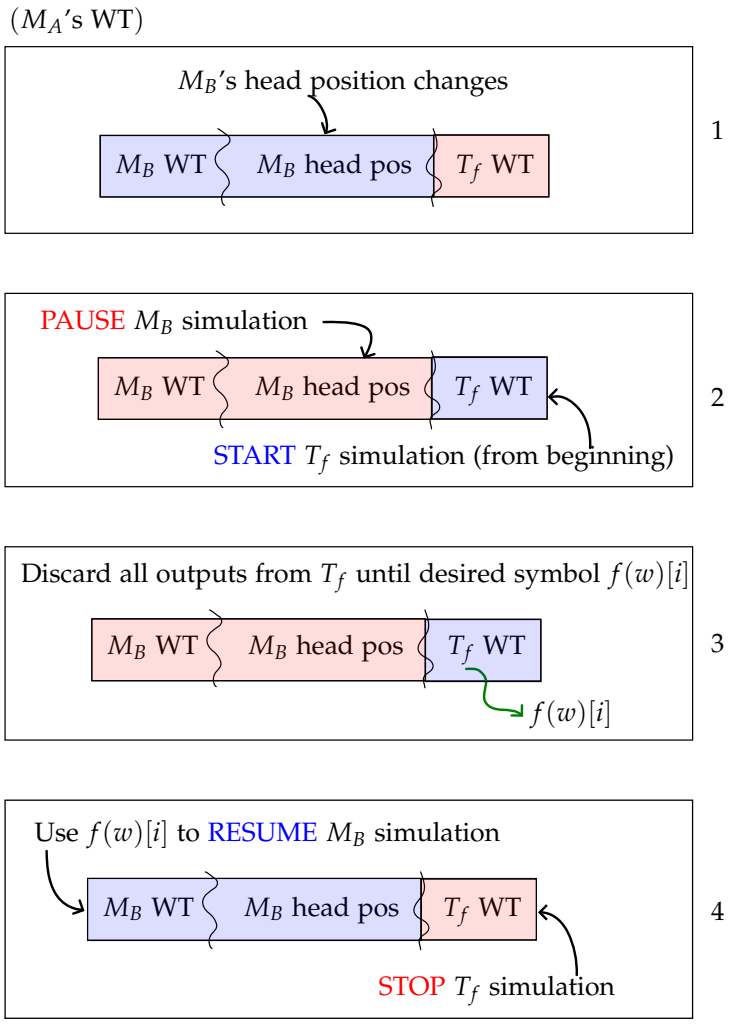

Figure 4: Illustration of the "recomputation" trick on M_A's working tape (WT). A section of M_A's WT is used for the simulation of M_B on $f(w)$, and another section is used for the simulation of T_f on w. Parts of the tape that are active are colored blue, and those that are inactive are colored red.

[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 4. *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 3, 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 3. *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,

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

Theorem 5. 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'.

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.

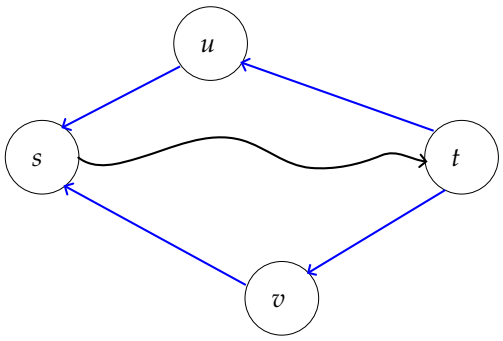

Figure 5: Construct G' by adding additional edges, colored blue, to the graph G and using the s-t-path in G (if it exists) to get between any two vertices.

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 6. 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.

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 2. STRONG-CONN *is NL-complete.*