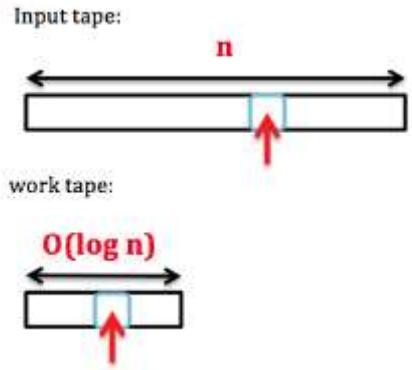
CSE 431 Theory of Computation
Scribes: Michelle Park and Julianne Brodhacker
Lecture 18 May 29

Log-Space

Review of Log-Space Turing Machines:

A log-space Turing Machine is comprised of two tapes: the input tape of size n which is cannot be written on, and the work tape of size $O(logn)$.

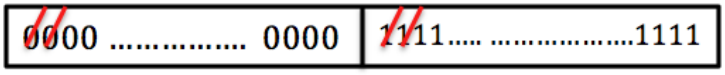


A way to think about these two tapes is in terms of memory, where the input tape is the equivalent of read-only memory on disk while the work tape is the equivalent of RAM.

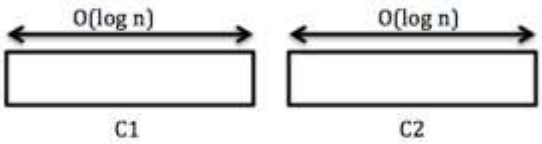
Example Early on in the quarter, we considered the following language:

$$A = \{\, 0^k 1^k : k \geq 1 \,\}$$

The algorithm we came up with was to move the header back and forth across the input, crossing off a 1 for every 0 crossed off until either there was a left over 0 or 1, or all the numbers were crossed out, as depicted below.



This algorithm does not work in log-space, however, since it requires being able to write to the entirety of the input. Since we have two tapes and, as such, two headers, we can easily count each number. It only takes $O(logn)$ bits to store each counter.



As such, the log-space algorithm is to simply count up all the 0's, count up all the 1's, and compare the counters to see if they are equal.

$$L = SPACE(logn)$$

Claim $L \subseteq P$

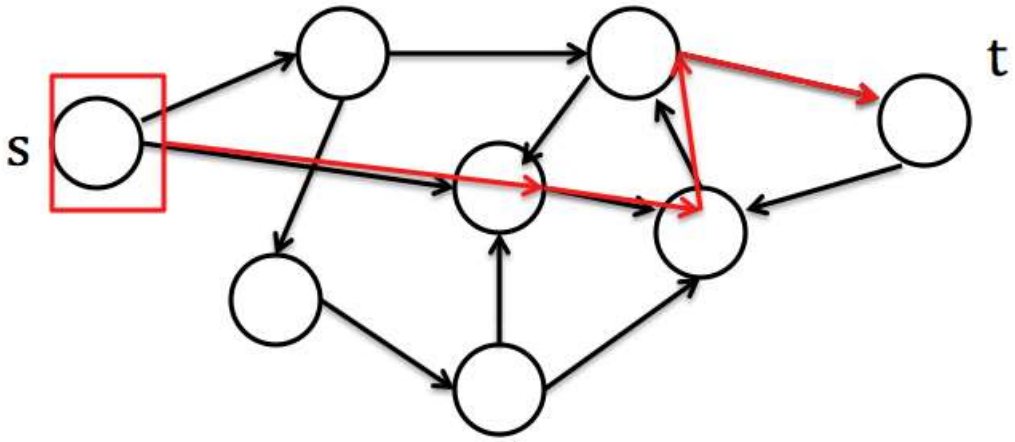
Proof Consider a log-space Turing Machine M with input w of size n. The maximum number of possible configurations that it can be in before repeating a configuration is $\leq n \times |\Gamma|^{O(logn)} \times O(logn) \leq n^{O(1)}$. Since the amount of space required to contain and the amount of time required to enumerate all possible configurations is polynomial, a machine can simply keep track of all possible configurations and emulate the behavior of L while being classified as within P. Thus, $L \subseteq P$. Like other time and space relationships, we do not know if $L = P$.

Non-Deterministic Log-Space

$$NL = NSPACE(logn)$$

One of the most representative problems found in Non-Deterministic log-space is DIRPATH. You may recall from earlier in the quarter that DIRPATH is defined as the following language:

$$DIRPATH = \{ < G, s, t > : G \text{ is a directed graph and there is an } s \to t \text{ path in } G \}$$



Claim $DIRPATH \in NL$

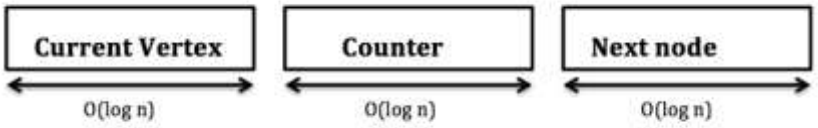
Simply remembering the name of a vertex takes $O(logn)$ space. As such, performing an algorithm such as breadth-first search is not viable. Instead we have to come up with

another algorithm. The key to this is to take advantage of the fact that the algorithm is being used by a non-deterministic Turing Machine.

Consider the following algorithm:

1. Start with node s as the current node
2. Guess a vertex at random
3. Check to see if there is an edge between the current node and the guessed node
    - If an edge does not exist, reject
    - If an edge does exist, increment the counter. If the guessed node is t, accept. If the counter reaches size n, reject. Otherwise, make the guessed node the current node and repeat steps 2 and 3.

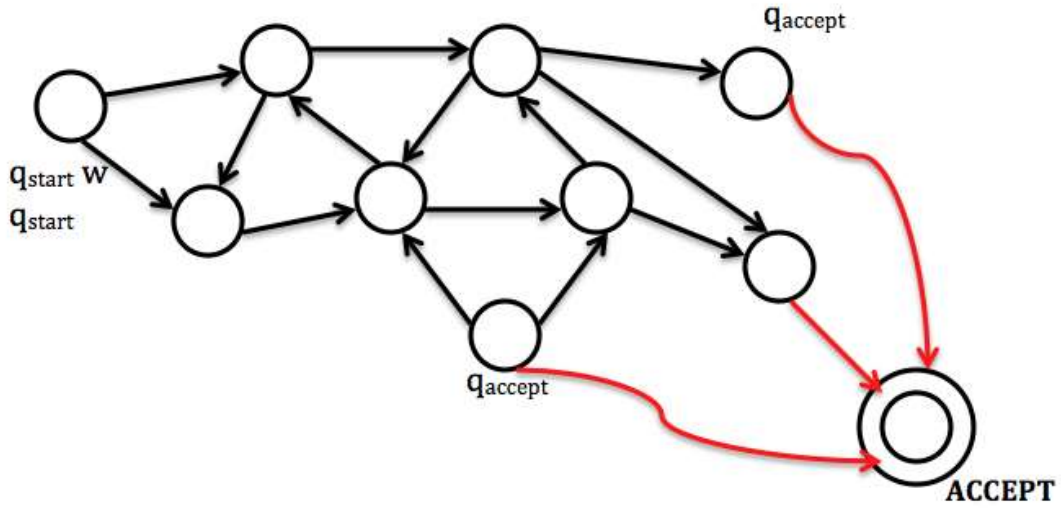
The work tape for such an algorithm would appear as such:



A non-deterministic Turing Machine that runs this algorithm would end up trying all possibilities due to its non-deterministic nature. Additionally, if any of the possibilities end up in an accept state, the Turing Machine will accept. Regardless of what it tries, the machine will always halt due to the rejecting if the counter reaches size n without finding node t. The only space required by this algorithm is the amount of space necessary to write the current node, the counter, and the next node, each of which only take $O(logn)$ space. Thus, $DIRPATH \in NL$.

Theorem DIRPATH is NL-Complete
While we are not going to prove it, DIRPATH captures all the difficulty of the NL class. The main difficulty of proving NL-Completeness is that reductions would have to run in $NSPACE(logn)$.

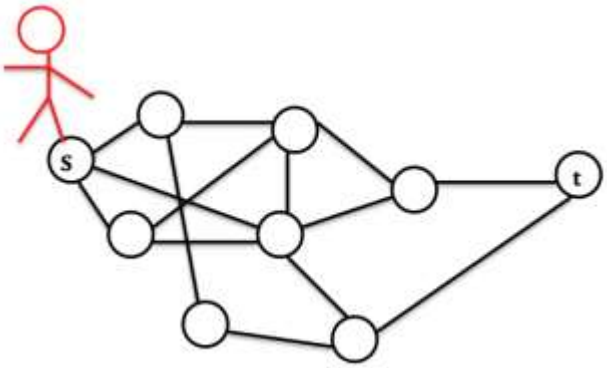
Claim $NL \subseteq P$
Proof Let M be a non-deterministic log-space Turing Machine. We want to know if on input w, does M accept w? Imagine building the following graph:



q_{start} is the node representing the start configuration of M on input w. Each node in the graph represents a different possible configuration the M can be in, while each edge represents the ability to transition from one configuration to the next. Finally, ACCEPT is our own accept node which has edges leading from any accept configuration M can be in to it. This graph can be built in polynomial time. As explained earlier, the total number of configurations possible is at most $n^{O(1)}$. As such, there will only be $n^{O(1)}$ nodes in the graph. Once the graph is built, we simply run one of the polynomial-time DIRPATH algorithms to find if there's a path between q_{start} and ACCEPT. If there is, then we accept. Else, reject.

We looked at DIRPATH and found that it was contained within the class L. However, consider the following similar language PATH:

$$PATH = \{ < G, s, t > : G \text{ is an undirected graph and there is an } s \to t \text{ path in } G \}$$

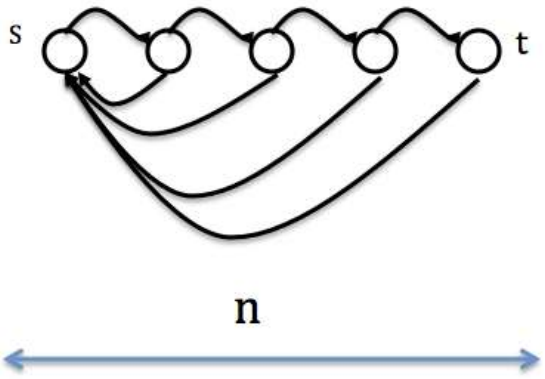


The question is, is $PATH \in L$?
One way to think of this problem is to think of it like being inside a maze. It can be done easily if you have enough memory to remember what paths had been tried already, such as using breadcrumbs. However, L does not afford enough space to be able to remember what had already been tried.

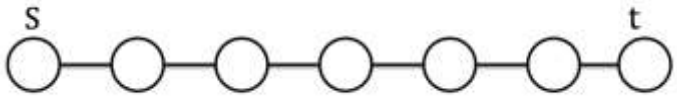
It turns out this problem can be solved in L if given the ability of randomness; such as flipping a coin. What happens is that at each point, the coin is flipped to determine which path to take. A counter is kept to keep track of how many steps are taken. A Turing Machine that does this will have the following attributes:

- If $< G, s, t > \notin PATH$, then the log-space TM will always reject.
- If $< G, s, t > \in PATH$, then the log-space TM will accept with probability $\geq \frac{1}{2}$. If the TM tries k times, then the probability that it will accept becomes $\geq 1 - \frac{1}{2^k}$.

What's important to know is that this algorithm does not work for a directed graph efficiently. Consider the following graph:



There is only a 2^n chance to get from node s to node t. As such, it would have to run for 2^n steps in order for it to have the same probability of accepting. However, even an undirected graph such as the one below does not have an exponential blow up.



In order to prove this theorem that $PATH \in L$, we need to examine one more property of the graph.

Let n = the number of vertices, m = the number of edges, V = { 1, 2, … , n }, and d_i be the degree of vertex i (i.e. the number of neighbors of i). The algorithm tends to spend more time at higher degree vertices. We can represent this through the following stationary measure: $\pi_i = \frac{d_i}{2m}$. We know that, due to what's known as the handshaking lemma, that $\sum_{i=1}^{n} d_i = 2m$ (every edge gets counted twice since each edge has two endpoints, both of which count it). This means that $\sum_{i=1}^{n} \pi_i = \sum_{i=1}^{n} \frac{d_i}{2m} = \frac{1}{2m} \sum_{i=1}^{n} d_i = 1$. We can consider each node i to have π_i mass. Every vertex has probability $\frac{1}{d_i}$ to send $\frac{1}{2m}$ mass to its neighbors. Each node should have equal amounts of mass coming into and leaving it. This is important to keep in mind for next time when we actually prove the theorem.