Notes on Complexity Theory: Fall 2005 Last updated: September, 2005

Lecture 3

Jonathan Katz

1 Terminology

For any complexity class $\mathcal{C}$, we define the class co$\mathcal{C}$ as follows:

$$\text{co}\mathcal{C} \stackrel{\text{def}}{=} \left\{ L \mid \bar{L} \in \mathcal{C} \right\}.$$

One class that is worth mentioning at this time is the class co$\mathcal{NP}$. By definition, we have:

$$\text{co}\mathcal{NP} \stackrel{\text{def}}{=} \{L \mid \bar{L} \in \mathcal{NP}\},$$

but this does not give too much intuition about this class. One can show that this definition is equivalent to the following two definitions:

1. $L \in \text{co}\mathcal{NP}$ iff there exists a polynomially-bounded relation $R_L \in \mathcal{P}$ such that

$$x \notin L \Leftrightarrow \exists w : (x, w) \in R_L.$$

2. $L \in \text{co}\mathcal{NP}$ iff there exists a polynomially-bounded relation $R_L \in \mathcal{P}$ such that

$$x \in L \Leftrightarrow \forall w : (x, w) \in R_L.$$

A proof of equivalence is left as an exercise.

2 Non-Deterministic Space Complexity

When we talk about space complexity in general, it is understood that our Turing machine model includes a read-only input tape (and, in the case of machines computing functions, a read-only, unidirectional output tape); furthermore, only the space used on the *work* tape(s) is counted in determining the space used by the machine. This allows us to meaningfully talk about sub-linear space classes.

There are some subtleties in defining non-deterministic space complexity. In particular, there are (at least) three definitions/models one might consider: The first definition is simply in terms of a Turing machine with a non-deterministic transition function. (We stress that the machine must work within the allotted space bound for *every* computation path it might possibly take.) One might actually say that this is *the* definition, so other definitions are useful only if they are equivalent to this one. As in the case of $\mathcal{NP}$, we can also imagine speaking in terms of *witnesses*. Specifically, we might augment our Turing machine model to consider machines with a special, read-only "auxiliary input" tape. Then we would say that $L \in \text{NSPACE}(s)$ if there exists a *deterministic* machine M_L using space

$s(n)$ such that if $x \in L$ then there exists a witness w such that $M_L(x, w)$ accepts; if $x \notin L$ then $M_L(x, w)$ rejects for all w. (The space used is measured in terms of the length $|x|$ of the input, and the witness w is placed on the auxiliary input tape. Note that w may be much longer than x, and the space used on the auxiliary input tape is not counted in the space complexity of M_L.) An important distinction here is whether we allow M_L to have *bidirectional* access to its auxiliary input or only *unidirectional* access; the first model is called the "off-line" model, while the second is called the "on-line" model. In turns out that the off-line model is too powerful; specifically, for "nice" s:

$$\text{NSPACE}_1(s) \subseteq \text{NSPACE}_{\text{off}}(\log s),$$

where the subscript "1" refers to the original definition and the subscript "off" refers to the off-line model. On the other hand, one can show:

$$\text{NSPACE}_1(s) = \text{NSPACE}_{\text{on}}(s),$$

and so the on-line model is a good one. See [1, Chap. 5] for further discussion as well as proofs of the above statements.

These issues will not explicitly arise in what follows, but the point is that we always implicitly work in the on-line model or (equivalently) under the original definition.

2.1 A Translation Lemma

Here is as good a place as any to discuss a simple translation lemma which is often useful. The lemma is stated for non-deterministic vs. deterministic space bounds but extends to other cases as well.

> **Note:** *We did not cover this in class, but the result is good to know (though the proof is rather technical).*

Lemma 1 *Let S_1, S_2, f be space-constructible with $S_2(n) \geq \log n$ and $f(n) \geq n$. Then* $\text{NSPACE}(S_1(n)) \subseteq \text{SPACE}(S_2(n))$ *implies* $\text{NSPACE}(S_1(f(n))) \subseteq \text{SPACE}(S_2(f(n)))$.

Proof We are given $L \in \text{NSPACE}(S_1(f(n)))$ and a non-deterministic machine M_1 running in this space bound that decides L. Define a padded version of L by:

$$L^p \stackrel{\text{def}}{=} \left\{ x\$^i \mid x \in L, \text{ and } M_1 \text{ accepts } x \text{ in space } S_1(|x| + i) \right\},$$

where we assume that $L \subseteq \{0,1\}^*$. Note that if $x \in L$ then $x\$^{f(|x|)-|x|} \in L^p$.

It is not hard to see that $L^p \in \text{NSPACE}(S_1(n))$. In particular, we have the following machine M_2 to decide it:

1. On input y, first check that y is of the form $x\$^i$ with $x \in \{0,1\}^*$. If not, reject.
2. Otherwise, say $y = x\$^i$. Run $M_1(x)$. If this execution of M_1 ever exceeds space $S_1(|x| + i) = S_1(|y|)$, reject. If M_1 rejects, reject. If M_1 accepts, accept.

M_2 needs only constant space in the first step, and space $S_1(|y|)$ in the second.

Now, by assumption, $L^p \in \text{SPACE}(S_2(n))$. In particular, there exists a deterministic machine M_3 using space $S_2(n)$ which decides L^p. We now construct M_4 that decides the original language L in space $O(S_2(f(n)))$:

- On input x, simulate executions of M_3 on $x\$^j$ for $j = 0, 1, \ldots$ using space at most $2S_2(f(|x|))$. If any of these accept, accept. If the space bound is ever exceeded, reject.

Instead of explicitly storing $x\$^j$ (which may require too much space), M_4 will instead maintain a counter of length $\log j$ indicating the input-head position of M_3 in the simulation (in this way, it knows when to feed M_3 the input symbol \$ rather than a blank). If $x \notin L$, the above will eventually reject (specifically, it will reject at latest by the time $\log j$ exceeds $2S_2(f(n))$). On the other hand, if $x \in L$ then set $j^* \stackrel{\text{def}}{=} f(|x|) - |x|$. As pointed out above, $x^* \stackrel{\text{def}}{=} x\$^{j^*}$ is in L^p. Furthermore, M_3 accepts x^* in space $S_2(|x^*|) = S_2(f(|x|))$, and M_3 never uses more space than this on shorter inputs. Furthermore, the space required (by M_4) to hold the counter is at most $\log j^* \leq \log f(|x|) \leq S_2(f(|x|))$ (using the fact that $S_2(n) \geq \log n$). We conclude that M_4 will accept x for some $j \leq j^*$. ■

We remark that the analogous lemma for the case of time-bounded machines is even easier, by setting

$$L^p \stackrel{\text{def}}{=} \{x\$^{f(|x|)-|x|} \mid x \in L\},$$

and then letting M_4 compute $f(|x|)$ directly and then run $M_3(x\$^{f(|x|)-|x|})$. Two modifications need to be made: first, we need $S_1(n), S_2(n) \geq n$ also; second, M_2 needs to check that its input $x\$^j$ satisfies $j = f(|x|) - |x|$ within its allotted time bound. This may be problematic if $x\$^j$ is shorter than $f(|x|)$. To solve this, M_2 simply times itself while computing $f(|x|)$ and rejects if it exceeds its time bound.

3 Non-Deterministic Space and the Reachability Method

In this section, we will see different applications of the so-called reachability method. The basic idea is that we can view the computation of a non-deterministic machine M on input x as a directed graph with vertices corresponding to configurations of $M(x)$ and an edge from i to j iff this represents a legal move of $M(x)$. (We will sometimes refer to this as the *configuration graph* of $M(x)$.) If we assume, without loss of generality, that M has only a single accepting state, then the question of whether $M(x)$ accepts is equivalent to the question of whether there is a path from the initial configuration of $M(x)$ to the accepting configuration. We refer to this as the *reachability* problem in the graph of interest.

3.1 The Class $\mathcal{NL}$

We begin with a brief discussion of $\mathcal{L}$ and $\mathcal{NL}$, where

$$\mathcal{L} \stackrel{\text{def}}{=} \text{SPACE}(\log n) \quad \text{and} \quad \mathcal{NL} \stackrel{\text{def}}{=} \text{NSPACE}(\log n).$$

Clearly, $\mathcal{L} \subseteq \mathcal{NL}$ but, as in the case of $\mathcal{P}$ vs. $\mathcal{NP}$, we do not know whether this inclusion is strict. Let us define some terminology.

Definition 1 L is *log-space reducible* to L' if there is a function f computable in log-space such that $x \in L \Leftrightarrow f(x) \in L'$. $\diamondsuit$

A log-space reduction is a special case of a Karp reduction, since any f computable in log-space is computable in polynomial time. In particular, then, the length $|f(x)|$ of the output is at most polynomially-longer than $|x|$.

Definition 2 L is $\mathcal{NL}$-complete if: (1) $L \in \mathcal{NL}$; and (2) for all $L' \in \mathcal{NL}$, L' is log-space reducible to L. $\diamondsuit$

Lemma 2 *If L is log-space reducible to L' and $L' \in \mathcal{L}$ (resp., $L' \in \mathcal{NL}$) then $L \in \mathcal{L}$ (resp., $L \in \mathcal{NL}$).*

Proof (Sketch) Let f be a function computable in log space such that $x \in L$ iff $f(x) \in L'$. The "trivial" way of trying to prove this lemma (namely, on input x computing $f(x)$ and then determining whether $f(x) \in L'$) does *not* work: the problem is that $|f(x)|$ may potentially be larger than $O(\log |x|)$ in which case this trivial algorithm will use more than logarithmic space. Instead, we need to be a bit more clever. The basic idea is as follows: instead of computing $f(x)$, we simply compute the i^{th} bit of $f(x)$ whenever we need it. In this way, although we are wasting time (in re-computing $f(x)$ multiple times), we never use more than logarithmic space. ■

We now use the reachability method discussed earlier to show a very natural $\mathcal{NL}$-complete problem: *directed connectivity* (called CONN). Here we are given a directed graph on n-vertices (say, specified by an adjacency matrix), and two vertices s and t and we need to determine whether there is a directed path from s to t or not. Formally:

$$\text{CONN} \stackrel{\text{def}}{=} \{(G, s, t) \mid \text{there is a path from } s \text{ to } t \text{ in graph } G\}.$$

(We remark that the corresponding problem on *undirected* graphs is possibly easier — since any undirected graph can be easily transformed into a directed graph — and in particular undirected connectivity is not believed to be $\mathcal{NL}$-complete.) To see that directed connectivity is in $\mathcal{NL}$, we need to show is a non-deterministic algorithm using log-space which never accepts if there is no path from s to t, but which sometimes accepts if there is a path from s to t. Letting n be the number of vertices in our graph, the following simple algorithm achieves this:

if $s = t$ accept
set $v_{\text{current}} := s$
for $i = 1$ to n:
    guess a vertex v_{next}
    if there is no edge from v_{current} to v_{next}, reject
    if $v_{\text{next}} = t$, accept
    $v_{\text{current}} := v_{\text{next}}$
if $i = n$ and no decision has yet been made, reject

To see that CONN is $\mathcal{NL}$-complete, assume $L \in \mathcal{NL}$ and let M_L be a non-deterministic log-space machine that decides L. Our log-space reduction from L to CONN will take an

input x and output the configuration graph of $M_L(x)$. Specifically, we will output a graph (represented as an adjacency matrix) in which the nodes represent configurations of M_L on input x and edges represent allowable transitions. The number of configurations of M_L on input x (with $|x| = n$) is at most $|Q_{M_L}| \cdot n \cdot 2^{O(\log n)} \cdot O(\log n)$, where these terms correspond to the state, the position of the input head, the contents of the work-tape, and the position of the work-tape head. So, a configuration can be represented using $O(\log n)$ bits.[1] Number these from 1 to R (with $\log R = O(\log n)$). We generate the adjacency matrix in log-space as follows:

For $i = 1$ to R:
    for $j = 1$ to R:
        Output 1 if there is a legal transition from i to j, and 0 otherwise
        (if i or j is not a legal state, simply output 0)

The above algorithm requires $O(\log n)$ space to store i, j and at most logarithmic space to check for a legal transition. Also, note that it indeed gives a reduction to CONN by the properties of the configuration graph that we remarked upon earlier.

Another consequence of the reachability method is that if a non-deterministic machine running in space $s(n) \geq \log n$ has an accepting computation, then it has an accepting computation passing through at most $2^{O(s(n))}$ states and so, in particular, it has an accepting computation running for at most this much time. So, using a counter, we may always assume that a non-deterministic machine using space $s(n) \geq \log n$ never runs (in *any* computation path) for time longer than $2^{O(s(n))}$. In fact, we can prove something even stronger:

Claim 3 *For $s(n) \geq \log n$ a space-constructible function,* $\text{NSPACE}(s(n)) \subseteq \text{TIME}(2^{O(s(n))})$.

Proof (Sketch) The idea is that we can solve the reachability problem on a graph with n nodes in deterministic time $O(n^2)$. The graph derived from the non-deterministic machine M_L (which decides some language L in space $s(n)$) has at most $2^{O(s(n))}$ vertices. Combining these two facts gives the stated theorem. ■

Corollary 4 $\mathcal{NL} \subseteq \mathcal{P}$.

3.2 Savitch's Theorem

We prove Savitch's theorem stating that non-deterministic space is not *too* much more powerful than deterministic space:

Theorem 5 *Let $s(n) \geq \log n$ be space-constructible. Then* $\text{NSPACE}(s(n)) \subseteq \text{SPACE}(s^2(n))$.

Proof The crux of the proof comes down to showing that reachability on an n-node graph can be decided in space $\log^2 n$. Specifically, let M be a non-deterministic machine using space $O(s(n))$. By what we have said above, if $M(x)$ accepts then there exists an accepting sequence of configurations of length at most $\mathcal{C}_M(|x|) = 2^{O(s(n))}$. We may assume without

[1] An important point is that we do not need to include the input x in the configuration, since x is fixed throughout the computation.

loss of generality that M has a single accepting configuration. We view the computation of $M(x)$ as a directed graph $\mathcal{G}_M$ with $2^{O(s(n))}$ vertices corresponding to configurations of $M(x)$, and where there is a directed edge between vertices i and j iff going from configuration i to configuration j represents a legal move for $M(x)$. Note that:

1. Vertices in $\mathcal{G}_M$ can be represented using space $O(s(n))$ (for $s(n) < n$ note that we do not need to explicitly write the input x as part of the configuration since it always sits on the input tape).

2. There is an enumeration of vertices such that given a vertex i the next vertex in the enumeration can be generated in constant space.

3. Given two vertices in $\mathcal{G}_M$, it is possible to determine whether there is an edge between them using space $O(s(n))$. (This is actually an over-estimate.)

We will now solve a more general problem: determining reachability in directed graphs. Specifically, given an n-vertex graph and vertices s, t we show a deterministic algorithm to determine whether there is a path from s to t using space $\log n \cdot |v|$, where $|v|$ is the space required to represent a vertex. Note that this proves the theorem, since then we have the following deterministic algorithm using space $s(n)^2$: determine whether there is a path from the initial configuration of M to the accepting configuration of M (viewed as vertices in $\mathcal{G}_M$) in the graph $\mathcal{G}_M$.

We now give the promised algorithm. Specifically, we give an algorithm PATH(a, b, i) which determines whether there is a path of length at most 2^i from a to b (we solve the stated problem by calling PATH$(s, t, \log n)$). The algorithm proceeds as follows:

- If $i = 0$, output "yes" if $a = b$ or if there is an edge from a to b. Otherwise, output "no".

- Otherwise, for each vertex v:

  - If PATH$(a, v, i-1)$ and PATH$(v, b, i-1)$, output "yes".

- Return "no".

Let us analyze the space complexity of the above algorithm. It is clear that the depth of the recursion is $O(\log n)$. At each level of the recursion, the algorithm needs to store a, b, the current vertex v, and the current value of i, all of which requires space $O(|v|)$. Putting everything together shows that the algorithm runs within the claimed space. ■

Corollary 6 NPSPACE = PSPACE.

3.3 The Immerman-Szelepcsényi Theorem

Here we will show the somewhat surprising result that non-deterministic space is closed under complementation.

Theorem 7 *If $s(n) \geq \log n$ is space constructible, then* NSPACE$(s(n))$ = *co*NSPACE$(s(n))$.

Proof The crux of the proof is to show that *non*-reachability (i.e., the complement of CONN) on graphs of n vertices can be decided using non-deterministic space $\log n$. We then show how this implies the theorem. We build up our algorithm in stages. We have already shown that there exists a non-deterministic algorithm for deciding CONN. To make it precise, let $\mathsf{path}(s, t, i)$ be a non-deterministic routine which determines whether there is a path of length at most i from s to t (then CONN is decided in n-vertex graphs by $\mathsf{path}(s, t, n)$). It is easy to see that the algorithm we gave earlier for CONN extends for the case of arbitrary i. Algorithm path is a yes/no algorithm using $O(\log n)$ space[2] which sometimes outputs "yes" if t is reachable from s in at most i steps (and sometimes outputs "no" in this case), but *always* outputs "no" if t is *not* reachable from s in at most i steps.

Using path as a sub-routine, we design a yes/no/abort algorithm isReachable with different guarantees: if t is reachable from s within i steps it sometimes outputs "yes" (but *never* outputs "no"), and if t is *not* reachable from s within i steps it sometimes outputs "no" (but never outputs "yes"). The rest of the time it aborts. (I.e., when it does not abort its answer is correct.) However, there is a caveat: this algorithm takes as additional input the number of vertices c within $i-1$ steps of s (and correctness is only guaranteed when this input is correct). Nevertheless, this algorithm will form a crucial building block for our main algorithm. Algorithm $\mathsf{isReachable}(s, t, i, c)$ proceeds as follows:

$\mathsf{ctr} = 0$
for each vertex v:
    if ($\mathsf{path}(s, v, i-1)$):
        increment ctr
        if $v = t$ or there is an edge from v to t output "yes" and stop
if $\mathsf{ctr} = c$ output "no"
if $\mathsf{ctr} \neq c$ abort

Let us analyze the behavior of the above *assuming c is the number of vertices within distance $i-1$ of s*: if t is reachable from s within i steps, then either t is within distance $i-1$ of s, or there exists some vertex v^* within distance $i-1$ of s for which there is an edge from v^* to t. In either case, path will sometimes output "yes" for this vertex v^*, and isReachable will output "yes" in such a case. We claim that isReachable cannot output "no". The only way it could possibly do so is if $\mathsf{ctr} = c$. But in this case the algorithm has found every vertex within distance $i-1$ of s (including v^*), in which case we would have already output "yes".

Considering the other case: if t is not reachable from s within i steps, then there is certainly no way we can output "yes". On the other hand, we will output "no" as long as we happen to find all vertices within distance $i-1$ of s.

Finally, the space used by isReachable is the space to store ctr, v, and to run path. Since $\mathsf{ctr} \leq n$ we see that the algorithm uses space $O(\log n)$.

Now we are ready to give our main algorithm $\mathsf{notReachable}(s, t)$. The intuition is as follows: let c_i denote the number of vertices within distance i of s (including s itself). Setting $c_0 = 1$ (since only s itself is within distance 0), we then iteratively compute $c_1, \ldots, c_{n-1}$. We then call $\mathsf{isReachable}(s, t, n, c_{n-1})$ and accept iff the answer is "no" (note that if t is reachable

[2] We assume path is always called with $i \leq n$, as there is no reason to do otherwise.

from s, then it is reachable in at most n steps). If any of the sub-routines of isReachable abort, the entire algorithm simply rejects immediately. (*Note*: in the algorithm, we save space by not storing all of $c_0, \ldots c_{n-1}$ but instead only storing the values we need.)

$i = 0, c_{\text{cur}} = 1$
for $i = 1$ to $n - 1$:
    $c_{\text{next}} = 0$
    for each vertex v:
        if (isReachable$(s, v, i, c_{\text{cur}})$) increment c_{next}
    $c_{\text{cur}} = c_{\text{next}}$ // (c_{cur} now holds c_i)
if isReachable$(s, t, n, c_{\text{cur}})$ outputs "no", accept
otherwise, reject

Let's see why the above algorithm is correct (it is not hard to see that it takes space $O(\log n)$). Note first that every time we call isReachable we are calling it with a correct final argument. (This is because if any invocation of isReachable aborts, the entire algorithm halts immediately; if all invocations of isReachable do *not* abort then the correct answer was always returned and so the value stored by c_{cur} is correct.) Now, we need to show that if there is no path from s to t then the algorithm sometimes accepts, and if there *is* a path from s to t then the algorithm never accepts. If there is no path from s to t and none of the calls to isReachable abort (which is certainly possible), then — as we said above — in the last line of the algorithm we call isReachable with a correct value of $c_{\text{cur}} = c_{n-1}$; it returns the correct answer, and we accept as desired. On the other hand, if there is a path from s to t there are two cases: either some call to isReachable aborts, or not. In the first case we are fine (since we reject anyway in this case). In the second case, we again call isReachable with a correct value of $c_{\text{cur}} = c_{n-1}$ in the last line of the algorithm; it returns the correct answer, and we reject. This is exactly what we needed to show.

Finally coming to the proof of the theorem statement itself, we use the same idea as in the proof that CONN is $\mathcal{NL}$-complete. Namely, given $L \in \text{NSPACE}(s(n))$ and non-deterministic machine M_L which accepts L using space $O(s(n))$, we consider the function f which takes an input x and outputs the adjacency matrix corresponding to configurations of M_L on input x. The space required to do so is $O(s(n) + \log n) = O(s(n))$, and we end up with a graph on $2^{O(s(n))}$ vertices. We then run algorithm notReachable on this graph (setting s to be the initial configuration and t to be the unique accepting configuration), which requires space $O(\log 2^{O(s(n))}) = O(s(n))$. (A subtlety is that — exactly as in the case of Lemma 2 — we never store the entire adjacency matrix at any time, but instead implicitly decide (as needed) whether there is an edge between some two given vertices.) ■

Bibliographic Notes

The proof of the translation lemma is from [1, Lect. 5].

References

[1] O. Goldreich. Introduction to Complexity Theory (July 31, 1999).