Introduction to Complexity Theory*
Lecture 6: Inside Non-Deterministic Logarithmic Space

Notes taken by Amiel Ferman and Noam Sadot

November 7, 1999

Summary: We start by considering space complexity of (decision and search) problems solved by using oracles with known space complexities. Then we study the complexity class $\mathcal{NL}$ (the set of languages decidable within Non-Deterministic Logarithmic Space); We show a problem which is complete for $\mathcal{NL}$, namely the Connectivity problem (Deciding for a given directed graph $G = (V, E)$ and two vertices $u, v \in V$ whether there is a directed path from u to v). Then we prove the somewhat surprising result: $\mathcal{NL} = \text{co}\mathcal{NL}$ (i.e., $\mathcal{NL}$ class is closed under complementation).

1 The composition lemma

$\boxed{6.1} \Rightarrow$

The following lemma was used implicitly in the proof of Savitch's Theorem:

Lemma 1.1 (composition lemma – decision version): *Suppose that machine M solves problem Π while using space $s(\cdot)$ and having oracle access to* decision tasks $\Pi_1, ..., \Pi_t$. *Further suppose that for every i, the task Π_i can be solved within space $s_i(\cdot)$. Then, Π can be solved within space $s'(\cdot)$, where $s'(n) \stackrel{\text{def}}{=} s(n) + \max_i\{s_i(\exp(s(n)))\}$.*

Proof: Let us fix a certain input x of length n for the machine M. First, it is clear from the definition of M that a space of length at most $s(n)$ is used on M's work-tape for the computation to be carried out. Next, we must consider all possible invocations of the decision tasks which M has oracle access to. Let M_i be (a determinstic) Turing Machine, computing decision task Π_i. Since at each step of its computation, M may query some oracle M_i, it is clear that the contents of each such query depends on the different configurations that M went through until it reached the configuration in which it invoked the oracle. In this sense, the input to M_i is a query that M "decided on". We may deduce that an input to an oracle is bounded by the size of the set of all configurations of machine M on the (fixed) input x (this is the maximal length of a such a query). Let us bound the maximal size of such a query: It is the number of all different configurations of M on input of size n: $|\Sigma_M|^{s(n)} \times s(n) \times n$, where we multiply the number of all possible contents of the work-tape (whose length is bounded by $s(n)$) with the number of possible positions (of the head) on the work-tape and with the number of possible positions on the input-tape (whose length is n) respectively (Σ_M is the work alphabet defined for the machine M).

Since the number of configurations of the machine M on input of length n is $exp(s(n))$, it is clear that the simulation of M_i would require no more than $s_i(exp(s(n)))$. Since we do not need to

* Lecture Notes for a course given by Oded Goldreich at the Weizmann Institute of Science, Israel.

store the contents of the work-tape after each such simulation, but rather invoke each M_i whenever we need it and erase all contents of the work-tape related to that simulation, it is clear that in addition to the space $s(n)$ of work-tape mentioned above, we need to consider the maximum space that a certain M_i would need during its simulation, hence the result. ■

We stress that the above lemma refers to decision problems, where the output is a single bit. Thus, in the simulation of the M_i's the issue of storing parts of the output of M_i does not arise. Things are different if we compose search problems. (Recall that above and below we refer to deterministic space-bounded computations)

Lemma 1.2 (composition lemma – search version): *Suppose that machine M solves problem Π while using space $s(\cdot)$ and having oracle access to* search tasks $\Pi_1, ..., \Pi_t$. (As we shall see, it does not matter if machine M has a one-way or two-way access to the oracle-reply tape.) *Further suppose that all queries of machine M have length bounded by exp(s(n)) and that the answers are also so bounded.*[1] *Further suppose that for every i, the task Π_i can be solved within space $s_i(\cdot)$. Then, Π can be solved within space $s'(\cdot)$, where $s'(n) \stackrel{\text{def}}{=} s(n) + \max_i\{s_i(\exp(s(n)))\}$.*

The emulation of the oracles here is more complex than before since these oracles may return strings rather than single bits. Furthermore, the replies to different oracles Π_i's may be read concurrently. In the emulation we cannot afford to run M_i on the required query and store the answer, since storing the answer would use too much space. In order to avoid this, every time we need any bit in the answer to $\Pi_i(q)$, (where q is a query) we need to run M_i again on q and fetch the required bit from the on-line generated output, scanning (and omitting) all other bits; i.e., the answer that would be received from the oracle would be written on the output one bit at a time and by using a counter, the machine could tell when did it reach the desired bit of the answer (this process would halt since the length of the answer is bounded). Note that this procedure is applicable regardless if M has one-way or two-way access to the oracle-reply tape. Note that unlike in Lemma 1, here we cannot bound the length of the query by the number of possible configurations since this number is too large (as it includes the number of possible oracle answers). Instead, we use the hypothesis in the lemma.

Anallagous, but much simpler, result holds for the time complexity:

Lemma 1.3 (composition lemma – time version): *Suppose that machine M solves problem Π while using time $t(\cdot)$ and having oracle access to* decision tasks $\Pi_1, ..., \Pi_k$. *Further suppose that for every i, the task Π_i can be solved within time $t_i(\cdot)$. Then, Π can be solved within time $t'(\cdot)$, where $t'(n) \stackrel{\text{def}}{=} t(n) \times \max_i\{t_i(t(n))\}$.*

Proof: Similarly to the proof regarding Space, we shall fix a certain input x of length n for the machine M. First, it is clear from the definition of M that time $t(n)$ is suffices for the computation of M on x to be carried out. Next, we must consider all possible invocations of the decision tasks which M has oracle access to. Here at each step of the computation M could invoke an oracle M_i and so it is clear that the time complexity of the computation would be $t(n)$ multiplied by the maximal time complexity of an oracle. In order to find the time complexity of some oracle M_i, we have to consider the possible length of a query to M_i; since there are $t(n)$ time units during the computation of M on x, the size of the query to M_i could be at most $t(n)$.

[1] Without this assumption, we cannot bound the number of configurations of machine M on a fixed input, as the configuration depends on the location of M's head on the oracle-reply tape.

We deduce that the time complexity of some oracle M_i which is invoked at some point of the computation of M on x, could be at most $t_i(t(n))$. According to what was said above, the time complexity of M on x would be the number of time units of its computation - $t(n)$ - multiplied by the maximal time complexity of some oracle M_i $(1 \leq i \leq k)$, hence the result. ■

2 A complete problem for $\mathcal{NL}$

$\boxed{6.2} \Rightarrow$ The complexity class $\mathcal{NL}$ is defined to be simply $NSPACE(O(log(n)))$. More formally we have:

Definition 1 *$\mathcal{NL}$: A language L belongs to $\mathcal{NL}$ if there is a nondeterminstic Turing machine M that accepts L and a function $f(n) = O(log(n))$ such that for every input x and for every computation of M at most $f(|x|)$ different work-tape cells are used.*

Our goal in this section and the following one would be to study some properties of the class $\mathcal{NL}$. To that end we define the following:

Definition 2 *A log-space reduction of L_1 to L_2 is a log-space computable function f such that $\forall x,\ x \in L_1 \ \Leftrightarrow\ f(x) \in L_2$*

Note that a log-space reduction is analagous to a Karp-reduction (where space corresponds to time and the logarithmic number of cells correspond to polynomial number of steps). Actually, since each function that can be computed in space $s(\cdot)$, can also be computed in time $exp(s(\cdot))$, we have that a log-space reduction is a special case of a polynomial time reduction. The next definition would define a notion analagous to $\mathcal{NP}$-completeness (as we will see, this would prove useful in proving a proposition about $\mathcal{NL}$ which is analagous to a proposition in $\mathcal{NP}$):

Definition 3 *L is $\mathcal{NL}$-Complete if:*

(1) *$L \in \mathcal{NL}$; and*

(2) *$\forall L' \in \mathcal{NL}$, L' is log-space reducible to L.*

2.1 Discussion of Reducibility

As implied from the definitions above, our goal would be to find a problem which is complete for the class $\mathcal{NL}$. Prior to that, we must make sure that the concept of completness is indeed meaningful
$\boxed{6.3} \Rightarrow$ for the class $\mathcal{NL}$. The following propositions ensure exactly that.

Proposition 2.1 *If L is log-space reducible to L' and L' is solvable in log-space then L is solvable in log-space.*

Proof: Since L' is solvable in logarithmic space, there exists a machine M' which decides L' using logarithmic space. Furthermore, since L is log-space reducible to L', there exists a function $f(\cdot)$ computable in log-space, such that $x \in L \ \Leftrightarrow\ f(x) \in L'$ and so there exists a machine M such that for every input x would first compute $f(x)$ and then would simulate M' in order to decide on $f(x)$, both actions demanding log-space (as $lg(|f(x)|) \leq lg(exp(lg(|x|))) = O(lg(|x|))$) and ensuring that M would accept x iff $x \in L$. ■

Interestingly, such reduction also preserve non-deterministic space:

Proposition 2.2 *If L is log-space reducible to L' and $L' \in \mathcal{NL}$ than $L \in \mathcal{NL}$*

Instead of proving the last proposition, we will prove a related proposition regarding Non-Deterministic Time:

Proposition 2.3 *If L if Karp-Reducible to L' and $L' \in \mathcal{NP}$ then $L \in \mathcal{NP}$*

Proof: Since L is Karp-Reducible to L', there is a many-to-one function $f(\cdot)$, computable in polynomial time, such that: $x \in L \;\Leftrightarrow\; f(x) \in L'$. Furthermore, since L' is in $\mathcal{NP}$, there is a Non-Deterministic Turing machine M' that can guess a witness y for an input z (the length of y is a polynomial in the size of z) in polynomial time such that $R_{L'}(z, y)$ holds (where $R_{L'}$ is the relation that defines the language L' in $\mathcal{NP}$). We will construct a Non-Determinstic machine M for deciding L in the following way: For a given input $x \in L$, M will compute $f(x)$ (deterministically in polynomial time) and then would just simulate M' (mentioned above) on input $f(x)$ to find a witness y (non-deterministically in polynomial time) such that $R_{L'}(f(x), y)$ would hold. Thus, M defines a relation R_L such that for every input $x \in L$, it guesses a witness y (non-determinstically in polynomial time) such that $R_L(x, y)$ holds (i.e., $R_L(x, y) = R_{L'}(f(x), y)$). So by definition, L is in $\mathcal{NP}$. ■

We can use the proof of Proposition 2.1 to prove Proposition 2.1: Instead of a function $f(\cdot)$ computable in polynomial time, we are guarnteed to have a function $f(\cdot)$ which is computed in logarithmic space. Furthermore we may presume the existence of a machine M' deciding the language L' in logarithmic space (instead of non-determinstic polynomial time). It is now clear, that one may construct a non-deterministic machine M which may decide the language L in logarithmic space (which is analagous to the machine M which decided L in non-deterministic polynomial time).

$\boxed{6.4} \Rightarrow$

Note that requiring the existence of a Cook-Reduction instead of a Karp-Reduction in Proposition 2.1 would probably make this proposition false: This stems from the fact that if a language L is Cook-Reducible to a language $L' \in \mathcal{NP}$ it does not necessarily mean that $L \in \mathcal{NP}$. In particular, any co$\mathcal{NP}$ language is Cook-Reducible to its complement. Still, if $\mathcal{NP} \neq$ co$\mathcal{NP}$ we have that $\overline{SAT} \notin \mathcal{NP}$ (and yet $\overline{SAT}$ is reducible to SAT). We conclude that if $\mathcal{NP} \neq$ co$\mathcal{NP}$ then Cook reductions are strictly more powerful than Karp reductions (since the class of languages which are Cook-reducible to $\mathcal{NP}$ contains co$\mathcal{NP}$, whereas the languages which are Karp-reducible to $\mathcal{NP}$ are exactly $\mathcal{NP}$). A more trivial example of this difference in power is the fact that any language in $\mathcal{P}$ is Cook-reducible to the empty set, whereas only the empty set is Karp-reducible to the empty set.

However, in the next proposition, as well as in Proposition 2.1, a Cook-reduction would do:

Proposition 2.4 *If L is polynomial-time reducible to L' and $L' \in P$ then $L \in P$*

In this last proposition, if L would be Cook-Reducible to L' then it is clear that the machine that emulates the oracle machine and answeres the queries by simulating the macine that decides L' (and runs in polynomial time), would be a polynomial-time machine that decides L (here the use of the oracle on L' and its actual simulation didn't make a difference in the running time). An analouge argument applies to Proposition 2.1. That is, if there exists a log-space oracle machine which deecides L by making polynomially-bounded queries to L', and L' is solvable in log-space then so is L (actually this follows from Lemma 1).

2.2 The complete problem: directed-graph connectivity

The problem that we will study would be graph connectivity (denoted as CONN) which we define
$\boxed{6.5} \Rightarrow$ next:

Definition 4 (directed connectivity – CONN): *CONN is the is defined as a set of triples,* (G, v, u)*, where* $G = (V, E)$ *is a directed graph,* $v, u \in V$ *are two vertices in the graph so that there is is a directed path from* v *to* u *in* G

As we shall see, the problem CONN is a natural problem to study in the context of space complexity. Intuitively, a computation of a Turing machine (deterministic or not) on some fixed input, could allways be pictured as a graph with nodes realting to the machine configurations and edges relating to transitions between configurations. Thus the question of whether there exists a certain accepting computation in the machine reduces to the question of the existence of a certain directed path in a graph: that path which connects the node which corresponds to the initial configuration and the node which corresponds to an accepting configuration (on a certain input). We note that in a deterministic machine the out-degree of each node in the graph would be exactly one, while in a non-determinstic machine the out-degree of each node could be any non-negative number (because of the possibilty of the non-determinstic machine to move to any one of a certain configurations), however in both cases the out-degree of each node is constant (depending only on the machine). Continuing this line of thought, it's not hard to see that CONN could be proved to be complete for the class $\mathcal{NL}$, i.e. it is itself in $\mathcal{NL}$ and every machine in $\mathcal{NL}$ could be reduced to it. The details are proved in the following:

Theorem 5 *CONN is* $\mathcal{NL}$*-Complete.*

> Oded's Note: *The following proof is far too detailed to my taste. The basic ideas are very simple. Firstly, it is easy to design a non-deterministic log-space machine which accepts* $CONN$ *by just guessing an adequate directed path. Secondly, it is easy to reduce any language* $L \in \mathcal{NL}$ *to* $CONN$ *by just considering the directed graph of configurations of a log-space machine (accepting* L*) on the given input, denoted* x*. Each such configuration consists of a location on the input-tape, a location of the work-tape, the contents of the work-tape and the state of the machine. A directed edge leads from one configurations to another iff they are possible consequetive configuration of a computation on input* x*. The key point is that the edge relation can be determined easily by examining the two configurations and the relevant bit of* x *(pointed to in the first configuration).*

Proof: First we show that $CONN \in \mathcal{NL}$ (see Definition 2). We will build a machine M that would decide for the input $G = (V, E)$ and $v, u \in V$ whether there exists a path in G from v to u. Of course, M would do so non-determinstically in $O(log(n))$ space where n is the size of the input. The outline of the algorithm is as follows: We start with the node v (given at the input) and a counter which is initialized to the number of nodes in G. At each step we decrement the counter and guess a node which is adjacent to the current node we have (initially v). If the node we have guessed is not adjacent to the node we hold, we just reject. No harm is done since this is a non-deterministic computation: it suffices that there will be some computation that would accept. This procedure concludes when either the counter reaches 0 (the path shold not be longer than the number of nodes) or the last node we have guessed is u (the other node specified in the input). The actual guessing of a node could be done in several ways, one would be to implement a small procedure that would non-deterministically write on the work tape symbols that would encode some node (by scanning the list of edges or adjacency matrix which are part of the input). Then, it will just check wheather the node it had written is adjacent to the current node we hold. Correctness and complexity analysis follow the formal specification of the algorithm:

Input: $G = (V, E)$, $v, u \in V$

Task: Find whether there exists a directed path from v to u in G.

1. $x \leftarrow v$
2. counter $\leftarrow |V|$
3. repeat
4. decrement counter by 1
5. guess a node $y \in V$ s.t. $(x, y) \in E$
6. if $y \neq u$ then $x \leftarrow y$
7. until $y = u$ or counter $= 0$
8. if $y = u$ then accept, else reject

First we will prove the correctness of this algorithm: On the one hand, suppose that the algorithm accepts the input $G = (V, E), v, u \in V$. This implies that during the repeat-until loop, the algorithm has guessed a sequence of nodes such that each one of them had a directed edge to its successor and that the final node in this sequence is u and the initial node is v (from the first line and the check that is made in the last line). Clearly, this implies a directed path from v to u in G (Note that from the existence of the counter, the number of steps in this computation is bounded by $O(n)$). On the other hand, suppose that there is a directed path in G from v to u. This path is a sequence $\{v, x_1, \ldots, x_k, u\}$ where $k \leq (n-2)$ and there is a directed edge from each node in this sequence to its successor. In this case it is clear that the computation of the algorithm above in which it guesses each one of the nodes in the sequence starting from v (from the first line of the algorithm) and ending with u (from the last line of the computation) is an accepting computation, and thus the algorithm would accept the input $G = (V, E), v, u \in V$.

We conclude that there is an accepting computation of the algorithm above on the input $G = (V, E), v, u \in V$ iff there is a directed path in G from v to u.

All that is left to be shown is that the implementation of such an algorithm in a non-determinstic machine would require no more than a logarithmic space in the size of the input: First, it is clear that each one of the variables required to represent a node in the graph need not be represented in more than logarithmic number of cells in the size of the input (for example, in order to represent a number n in binary notation, we need no more than $lg(n)$ bits). The same argument applies to the counter which has to count a number which is bounded by the size of the input. Secondly, all other data besides the variables may be kept at a constant number of cells of the work-tape (for example a bit that would indicate whether $y = u$ etc.). As was specified above regarding the implementation of step 5 (the guessing of the node), the actuall guessing procedure, which would be done non-determinstically, uses number of cells which is equal exactly to the length required to the representation of a node (which is again logarithmic in the size of the input). We conclude that the implementation of the algorithm above on a non-determinstic machine M requires a logarithmic space in the size of the input, and so we may conclude that the machine M decides CONN in non-deterministic logrithmic space, i.e., $CONN \in \mathcal{NL}$.

Now we need to show that every language $L \in \mathcal{NL}$ is log-space reducible to CONN. Let L be a language in $\mathcal{NL}$, then there is a non-deterministic logarithmic space machine M that decides L. We will show that for every input x, we can build in nondeterministic logarithmic space an input $(G = (V, E), start \in V, end \in V)$ (which is a function of machine M and input x) such that there is a path in G from *start* to *end* if and only if M accepts input x. The graph G we will construct would simply be the graph of all possible configurations of M given x as an input. That is, the nodes denote different configurations of M while computing on input x, and the arcs denote possible immediate transitions between configurations.

The graph is constructed (deterministic) log-space as follows;

Input: An input string x (the machine M is fixed)

Task: Output a graph $G = (V, E)$ and two nodes $v, u \in V$ such that there is a path from v to u in the graph iff x is accepted by M.

1. compute n, the number of different configurations of M while computing input x
2. for $i = 1$ to n
3. for $j = 1$ to n
4. if there is a transition (by a single step of M) from configuration number i to configuration number j ouput 1 otherwise output 0
5. output 1 and n

First we will show that this procedure indeed outputs the representation of a graph and two nodes in that graph such that there exists a directed path between those two nodes iff the input x is accepted by machine M. In the first line we compute the number of all possible configurations of machine M while computing on input x. Then, we consider every ordered pair of configurations (represented by numbers between 1 and n) and output 1 iff there is indeed a directed transition between those two configurations in the computation of M on x. Our underlying assumption is that 1 represents the initial configuration and n represents the (only) accepting configuration (if there were several accepting configurations we define a new one and draw edges from the previous accepting configurations to the new one). Thus, the output of the above procedure is simply an adjacency matrix of a graph in which each of its nodes correspond to a unique configuration of M while computing on input x , and a directed edge exists between two nodes i and j iff there is a (direct) transition in M between the configuration represented by x and the configuration represented by y. It is now clear that a directed path from the first node (according to our enumeration) to the last node in the graph would correspond to an accepting computation of machine M on input x, and that such a path would not exists should there be no such accepting computation.

Next, we must show that the above procedure could indeed be carried out using no more than a logarithmic space in the length of the input (i.e., the input x). In order to do that, we will show that the number of different configurations of M while computing x is polynomial in the length of x. This would imply that in order to count these configurations we need no more than a logarithmic space in the length of x. So, we will count the number of possible configurations of M while computing on a given input x. That number would be the number of possible states (a constant determined by M) multiplied by the number of possible contents of the work-tape which is $|\Sigma_M|^{O(log(n))}$ where Σ_M is the alphabet of M and is also a constant determined by M (let us remember that since M is log-space bounded, the number of used work-tape squares could not surpass $O(log(n))$), multiplied by the number of different positions of the reading head on the input tape which is n and finally, mutiplied by the number of different possible positions of the reading head on the work-tape which is $O(log(n))$. All in all, the number of different configurations of M while computing input x is: $|States_M| \times |\Sigma_M|^{O(log(n))} \times n \times O(log(n)) = O(n^k)$ (where k is a constant). That is, the number of different configurations of M while computing input x is polynomial in the length of x.

We may conclude now that the initial action in the procedure above, that of counting the number of different configurations of M while computing on input x could be carried out in logarithmic space in the length of x.

Secondly, we show that the procedure of checking whether there exists a (direct) transition between two configurations represented by two integers can be implemented as well in logarithmic space: We show that there is a machine M'' that receives as inputs two integers and would return a positive answer if and only if there is a (direct) transition of M between the two configurations which are represented by those integers. Note first that integers correspond to strings over some convinient alphabet and that such strings correspond to configurations. (The correspondance is by trivial computations.) Thus, all we need to determine is whether the fixed machine M can pass in one (non-deterministic) step, on input x, between a given pair of configurations. This depends only on the transition function of M, which M'' has hard-wired, and on a single bit in the input x; that is, the bit the location of which is indicated in the first of the two given configurations. That is, suppose that the first configuration is of the form (i, j, w, s), where i is a location on the input-tape, j a location on the work-tape, w the contents of the work-tape and s the machine's state. Same for the second configuration, denoted (i', j', w', s'). Then we check if M when reading symbol x_i (the i^{th} bit of x) from its input tape, and w_j (the j^{th} symbol of w) from its work-tape can make a single transition resulting in the configuration (i', j', w', s'). In particular, it must hold that $i' \in \{i-1, i, i+1\}$, $j' \in \{j-1, j, j+1\}$, and w' differs from w at most on location j. Furthermore, these small changes must depend on the transition function of M. Since there is a constant number of possible transitions (in M's transition function), we may just check all of them.

We have shown that the above procedure outputs a representation of a graph and two nodes in that graph for a machine M and input x, such that there is a directed path between the nodes iff there is an accepting computation of M on x. Furthermore we have shown that this procedure may be implemented requiring no more than a log-space in the size of the input (the input string x) which concludes the proof. ■

3 Complements of complexity classes

$\boxed{6.6} \Rightarrow$

Definition 6 (complement of a language): *Let $L \subseteq \{0,1\}^*$ be a language. The* complement of a language *L, denoted $\overline{L}$ is the language $\{0,1\}^* \backslash L$.*

To make this definition more accurate, we assume that every word in $\{0,1\}^*$ represents an instance of the problem.

Example 7 *: $\overline{CONN}$ is the following set: $\{(G, u, v)$: G is a directed graph, $u, v \in V(G)$, there is no directed path from u to v in G $\}$.*

Definition 8 (complement of class): *Let $\mathcal{C}$ be a complexity class. The* complement of the class $\mathcal{C}$ *is denoted* co$\mathcal{C}$ *and is defined to be* $\{\overline{L} : L \in \mathcal{C}\}$

It is immediately obvious that if $\mathcal{C}$ is a deterministic time or space complexity class, then co$\mathcal{C} = \mathcal{C}$, in particular, $\mathcal{P} =$ co$\mathcal{P}$. This is true, since we can change the result of the derministic machine from 'yes' to 'no' and vice versa.

However, in non-deterministic complexity classes, this method does not work. Let M be a Turing Machine that accepts a language L non-deterministicly. If $x \in L$, then there is at least one successful computation of M on x (i.e., there is a succinct verification that $x \in L$). We denote by $\overline{M}$ the non-deterministic Turing Machine that does the same as M, but replaces the output from "yes" to "no" and vice versa. Hence, if the new machine $\overline{M}$ accepts an input z, there is one accepting computation for $\overline{M}$ on z, i.e. non-accepting computation in M (by definition). In other words, $\overline{M}$ will accept z, if M has some unsuccessful guesses to prove that $z \in L$. This, however, does not mean that $\overline{M}$ accepts $\overline{L}$, since z could possibly be in L by other guesses of the machine M. For example we don't know whether co$\mathcal{NP}$ is equal to $\mathcal{NP}$. The conjecture is that $\mathcal{NP} \neq$ co$\mathcal{NP}$

Yet, in the particular case of nondeterministic space equality does hold. It can be proven that any non-deterministic space $NSPACE(s(n))$ for $s(n) \geq log(n)$ is closed under complementation. This result which was proven by Neil Immerman in 1988, is going to be proven here for the case $\mathcal{NL}$. By the following proposition, it suffices to show that $CONN \in$ co$\mathcal{NL}$ (or equivalently $\overline{CONN} \in \mathcal{NL}$.

Proposition 3.1 *: If for an $\mathcal{NL}$-complete language L it holds that $L \in$ co$\mathcal{NL}$ then $\mathcal{NL} =$ co$\mathcal{NL}$.*

$\boxed{6.7} \Rightarrow$

Proof: Let L' be a language in $\mathcal{NL}$. Since $L \in \mathcal{NL}$-complete, we have a log-space reduction f from L' to L (see Definition 2). The function f satisfies:

$$x \in L' \Leftrightarrow f(x) \in L$$

Taking the opposite direction, we get:

$$x \in \overline{L'} \Leftrightarrow f(x) \in \overline{L}$$

By the definition of the reduction, f is also a reduction from $\overline{L'}$ to $\overline{L}$. By proposition 2.1 we know that since $\overline{L} \in \mathcal{NL}$ (because by hypothesis $L \in$ co$\mathcal{NL}$) then $\overline{L'} \in \mathcal{NL}$ (i.e. $L' \in$ co$\mathcal{NL}$). We conclude that for every $L' \in \mathcal{NL}$, $\overline{L'} \in \mathcal{NL}$, thus $\mathcal{NL} =$ co$\mathcal{NL}$. ■

4 Immerman Theorem: $\mathcal{NL} =$ co$\mathcal{NL}$

In this section, we are going to prove a surprising theorem, which claims that non-deterministic log-space is closed under complementation. Due to the proof in Theorem 2.2 that $CONN \in \mathcal{NL}$-complete, and using proposition 3, we only need to prove that $\overline{CONN} \in \mathcal{NL}$, where $\overline{CONN}$ is the complementary problem of $CONN$ as defined in Example 3. Formally, The decision problem of the language $\overline{CONN}$, is obtained as the following:

Input: a directed graph $G = (V, E)$ and two nodes $u, v \in V(G)$.

Question: Is there no directed path from v to u ?

In order to show that $\overline{CONN} \in \mathcal{NL}$, we use the following theorem, which will be proven later:

$\boxed{6.9} \Rightarrow$

Theorem 9 *Given a directed graph $G = (V, E)$ and a node $v \in V(G)$, the number of nodes reachable from v in G can be computed by a non-deterministic Turing Machine within log-space.*

A non-deterministic Turing Machine that computes a function f of the input, as in the theorem above, is defined as follows:

Definition 10 (non-deterministic computation of functions): *A non-deterministic Turing Machine M is said to compute a function f if on any input x, the following two conditions hold:*

1. *either M halts with the right answer $f(x)$, or M halts with output "failure"; and*
2. *at least one of the machine computations halts with the right answer.*

4.1 Theorem 4 implies $\mathcal{NL} = \text{co}\mathcal{NL}$

$\boxed{6.8} \Rightarrow$

Lemma 4.1 *Assuming Theorem 4, $\overline{CONN} \in \mathcal{NL}$*

Assuming Theorem 4, we have a non-deterministic Turing Machine denoted CR ($CR \stackrel{\text{def}}{=}$ Count Reachable) that counts all the nodes that are reachable in a directed graph G from a single node v in non-deterministic log-space. The idea of the proof is that once we know to compute using CR the number of nodes reachable from v in G, we can also non-deterministically scan all the vertices reachable from v, using this value. This is done non-deterministically by guessing connected paths to each of the reachable nodes from v. Once the machine discovered all reachable nodes from v (i.e. the number of reachable nodes it found equals the output of CR) and the node u isn't reachable, it can decide that there is no connected path between v and u in G.

Proof: Let $x = (G, u, v)$ be an input for the problem $\overline{CONN}$. We fix (G, v) and give it as an input to the machine CR, which is the non-deterministic machine that enumerates all the nodes in the graph G reachable from the node v as was assumed before to work in non-deterministic log-space. In other words, we use CR, as a "black box".

We construct a non-deterministic machine here that uses the following simulation that solves this problem:

- Firstly, it simulates CR on the input (G, v). If the run fails, the machine rejects. Otherwise, we denote the answer by N.
- For each vertex w in the graph, it guesses whether w is reachable from v and if yes, it guesses non-deterministicly a directed path from v to w, by guessing at most $n-1$ vertices (we know by a simple combinatorial fact, that each two connected nodes in a grpah $G = (V, E)$ are connected within path of length less than or equal to $n-1$) and verifies that it is a valid path. For each correct path, it increments a counter.
- If $w = u$, and it founds a valid path then it rejects.
- If *counter* $\neq N$ then the machine rejects the input, otherwise the machine accepts the input. (*counter* $\neq N$ means that not all reachable vertices were found and verified, whereas *counter* $= N$ means that all were examined. If none of these equals u then we should indeed accept).

Formally, we have the following algorithm:

Input: $G = (V, E)$, $v, u \in V(G)$
Task: Find whether there is no connected path between u and v.

1. Simulating CR. If CR fails, the algorithm rejects, else $N \leftarrow CR((G, v))$.

2. $counter \leftarrow 0$

3. for $w = 1$ to n do *(w is a candidate reachable vertex)*

4. guess if w is reachable from v. If not, proceed to next iteration of step 3.
(we continue in steps 5-17 only if we guessed that w is reachable from v)

5. $p \leftarrow 0$ *(counter for path length)*

6. $v_1 \leftarrow v$ *(v_1 is initially v)*

7. repeat *(guess and verify a path from v to w)*

8. $p \leftarrow p + 1$

9. guess a node v_2 *(v_1 and v_2 are the last and the current nodes)*

10. if $(v_1, v_2) \notin E$ **then reject**

11. if $v_2 \neq w$ **then** $v_1 \leftarrow v_2$

12. until $(v_2 = w)$ **or** $(p = n - 1)$

13. if $(v_2 = w)$ **then** *(counting all reachable $w \neq u$)*

14. begin

15. $counter \leftarrow counter + 1$

16. if $w = u$ **then reject**

17. end

18. if $N \neq counter$ then reject, else accept.

We know that CR works in $O(log(|G|))$. In each step of the simulation, our algorithm uses only 6 variables in addition of those of CR, namely the counters $counter, w, p$, the representations of the nodes v_2, v_1 and N. the counters, and N are bounded by the number of vertices, n. Every new change of one of this variables will be written again on the work tape by reusing space. Therefore, they can be implemented in $O(log(n))$ space. The nodes, clearly, are represented in $O(log(|G|))$ space. Thus, we use no more then $O(log(|G|))$ space in the work tape in this machine (where x is the input). The correctness is proved next:

To show correctness we need to show that it has a computation that accepts the input if and only if there is no direct path from v to u in G.

Consider first the case that the machine accepts. A necessary condition (for this event) is that $counter = N$ (line 18); that is, the number of vertices that were found to be reachable is exactly the correct one (i.e., N). This means that every possible vertex that is reachable from v was counted. But, if u was found to be one of them the machine should have rejected before (in line 16). Therefore, u cannot be reachable from v by a directed path.

Suppose, on the other hand, that there is no directed path between u and v in G. Then if all guesses made are correct then the machine will necessarily accept. Specifically, we look at a computation in which (1) the machine correctly guesses (in line 4) for each vertex w whether it is reachable from v; (2) for each reachable vertex w it guesses well a directed path from v; and (3) machine CR did not fail (and thus N equals the number of vertices reachable from v). In this case $N = counter$, and since u is not connected to v the machine accepts.

Therefore, we proves the lemma. ■

$\boxed{6.10} \Rightarrow$ Using this result (under the assumption that Theorem 4 is valid), we obtain $\mathcal{NL} = \text{co}\mathcal{NL}$.

Theorem 11 (Immerman 88'): $\mathcal{NL} = \text{co}\mathcal{NL}$

Proof: We proved in Theorem 2.2 that $CONN \in \mathcal{NL}$-complete. In Lemma 4.1, we proved that $\overline{CONN} \in NL$ (or $CONN \in \text{co}\mathcal{NL}$). Using Proposition 3, we get, that $\mathcal{NL} = \text{co}\mathcal{NL}$. ■

An extension of this theorem can show that for any $s(n) \geq log(n)$, $NSPACE(s(n)) = coNSPACE(s(n))$.

4.2 Proof of Theorem 4

To conclude this proof we are only left with the proof of Theorem 4, i.e. the existness of a machine CR, that computes the number of nodes reachable from a vertex v in a directed graph $G = (V, E)$.

We use the following notations for a fixed point v in a fixed directed graph $G = (V, E)$:

Definition 12 *R_j is the set of vertices which are reachable from v by a path of length less than or equal to j. In addition, N_j is defined to be the number of nodes in R_j, namely $|R_j|$.*

It can be seen that,

$$\{v\} = R_0 \subseteq R_1 \subseteq \cdots \subseteq R_{n-1} = R$$

where n denotes the number of nodes in G, and R denotes the set of vertices reachable from v.

There is a strong connection between R_j and R_{j-1} for $j \geq 1$, since any path of length j is a path of length $j-1$ with an additional edge. The following claim will be used later in the development of the machine CR:

Claim 4.2 *The following equation holds:*

$$R_j = \begin{cases} R_{j-1} \cup \{u : w \in R_{j-1}, (w,u) \in E(G)\} & \text{if } j \geq 1 \\ \{v\} & \text{if } j = 0 \end{cases}$$

Proof: For $j = 0$: Clear from definition.
For $j \geq 1$: $R_{j-1} \subseteq R_j$ by definition. $\{u : w \in R_{j-1}, (w,u) \in E(G)\}$ represents all the nodes which are adjacent to R_{j-1}, i.e. have length at most $j - 1 + 1 = j$. This set is also contained in R_j. Thus, $R_{j-1} \cup \{u : w \in R_{j-1}, (w,u) \in E(G)\} \subseteq R_j$.

In the opposite direction, every node $u \in R_j$, which is not v, is reachable from v along a path with length less or equal to j. Thus, its predecessor in this path has length less or equal to $j-1$. Thus, $R_j \subseteq \{u : w \in R_{j-1}, (w,u) \in E(G)\} \cup \{v\} \subseteq R_{j-1} \cup \{u : w \in R_{j-1}, (w,u) \in E(G)\}$ (since $\{v\} \subseteq R_{j-1}$ for any $j \geq 1$).

Therefore, the claim follows. ■

Corollary 13 *For any $j \geq 1$, a node $w \in R_j$, if and only if there is a node $r \in R_{j-1}$ such that $r = w$ or $(r, w) \in E(G)$.* ■

We now construct a non-deterministic Turing Machine, CR, that counts the number of nodes in a directed graph G, reachable from a node in the graph v.

Our purpuse in this algorithm is to compute N_{n-1} where n is the number of nodes in G, to find the number of all reachable nodes from v. This recursive idea in Claim 4.2 is the main idea behind the following algorithm, which is build iteratively. In each stage, the algorithm computes N_j by using N_{j-1}. It has an initial value N_0, which we know to be $|\{v\}| = 1$. The itrations use the non-deterministic power of the machine.

The high-level description of the algorithm is as follows:

- For each j from 1 to $n-1$, it tries to calculate recursively N_j from N_{j-1}. This is done from N_0 to N_{n-1}, which is the desired output. Here is how N_j is computed.
  - For each node w in the graph,
    - For each node r in the graph, it guesses if $r \in R_{j-1}$ and if the answer is yes, it guesses a path with length less than or equal to $j-1$, from v to r. It verifies that the path is valid. If it is, it knows that r is a node in R_{j-1}. Otherwise, it rejects. It counts each node r, such that $r \in R_{j-1}$, by $counter_{j-1}$. The machine checks whether $w = r$ or $(r, w) \in E(G)$. If it is, (by using Corrolary 4.2), $w \in R_j$, and then it indicates by a flag, $flag_w$ that w is in R_j. ($flag_w$ is initially 0).
  - It counts the number of vertices r in R_{j-1} we found, and verifies that it is equal to N_{j-1}. Otherwise, it rejects. If the machine does not reject, we know that every node $r \in R_{j-1}$ was found. Therfore using Corrolary 4.2, the membership of w in R_j is decided properly (i.e. $flag_w$ has the right value).
  - At the end of this process it sums up the flags, $flag_w$'s into a counter, $counter_j$ (i.e. counts the number of nodes that were found to be in R_j).
- It stores the value of $counter_j$ for N_j to begin a new iteration (or to give the result in case we reach $j = n-1$).

We stress that all counters are implemented using the same space. That is, the only thing which passes from iteration $j-1$ to iteration j is N_{j-1}.

The detailed code follows:

Input: $G = (V, E)$, $v \in V(G)$

Task: Find the number of reachable nodes from v in G.

1. **Computing** $n = |V(G)|$

2. $N_0 \leftarrow 1 = |R_0|$

3. **for** $j = 1$ **to** $n-1$ **do**

4. $\quad counter_j \leftarrow 0$

5. $\quad$ **for** $w = 1$ **to** n **do** $\qquad$ *(lines 5-24 compute N_j)*

6. $counter_{j-1} \leftarrow 0$ *(w is a potential member in R_j)*

7. $flag_w = 0$

8. **for** $r = 1$ **to** n **do** *(We try to enumerate R_{j-1} using N_{j-1})*

9. **guess if** $r \in R_{j-1}$**. If not, proceed to next iteration of step 8.**
(we continue in steps 10-21 only if we guessed that $r \in R_{j-1}$)

10. $v_1 \leftarrow v$ *(v_1 is initially v)*

11. $p \leftarrow 0$

12. **repeat** *(guess and verify a path from v to r, such that $r \in R_{j-1}$)*

13. $p \leftarrow p + 1$

14. **guess a node** v_2 *(v_1 and v_2 are the last and current nodes)*

15. **if** $(v_1, v_2) \notin E$ **then halt(failure)**

16. **if** $v_2 \neq r$ **then** $v_1 \leftarrow v_2$

17. **until** $(v_2 = r)$ **or** $(p = j - 1)$

18. **if** $v_2 \neq r$ **then halt(failure)**

19. $counter_{j-1} \leftarrow counter_{j-1} + 1$

20. **if** $(r = w)$ **or** $((r, w) \in E(G))$ **then** *(check that $w \in R_j$)*

21. $flag_w = 1$

22. **if** $counter_{j-1} \neq N_{j-1}$ **then halt(failure)**

23. $counter_j \leftarrow counter_j + flag_w$

24. $N_j = counter_j$

25. **output** N_{n-1}

Lemma 4.3 *The machine CR uses $O(log(n))$ space.*

$\boxed{6.11} \Rightarrow$

Proof: Computing the number of nodes (line 1) can be made in at most $O(log(n))$, by simply counting the number of nodes with a counter on the input tape. In each other step of the running of the machine, it only needs to know at most ten variables, i.e. the counters for the 'for' loops: j, w, r, p, the value of N_{j-1} for the current j, the two counters for the size of the sets $counter_j, counter_{j-1}$, two nodes of the guessing part v_2, v_1, and the indicating flag $flag_w$.

> Oded's Note: *The proof might be more convincing if the code was modified so that N_{PREV} is used instead of N_{j-1}, $counter_{CURR}$ instead of $counter_j$, and $counter_{PREV}$ instead of $counter_{j-1}$. In such a case, line 24 will prepare for the next iteration by setting $N_{PREV} \leftarrow counter_{CURR}$. Such a description will better emphasise the fact that we use only a constant number of variables, and that their storage space is re-used in the iterations.*

Whenever is needed to change information, like increasing a counter, or changing a variable, it reuses the space it needs for this goal. Every counter we use counts no more than the number of nodes in the graph, hence we can implement each one of them in $O(log(n))$ space. Each node is assumed to take $O(log(n))$ to store, i.e. its number in the list of nodes. And the $flag_w$ clearly takes only 1 bit. Therefore, to store these variables, it is enough to use $O(log(n))$ space.

Except for these variables, we don't use any additional space. All that is done is comparing nodes with the input tape, and checking whether two nodes are adjacent in the adjecancy matrix that represents the graph. These operations can be done only by scanning the input tape, and take no more then $O(log(n))$ space, for counters that scan the matrix.

Therefore, this non-deterministic Turing Machine uses only $O(log(n))$ or $O(log(|x|))$ where $x = (G, v)$ is the input to the machine. ■

Lemma 4.4 *If the machine CR outputs an integer, then it correctly gives the result of N_{n-1}.*

Proof: We'll prove it by induction on the iteration of computing N_j:
For $j = 0$: It is obviously correct.
If it computes correctly N_{j-1}, and it did not halt while computing N_j, then it computes correctly N_j as well: By the assumption of the induction we have a computation that computes N_{j-1}, that is stored correctly. All we have to prove is that $counter_j$ is incremented if and only if the current w is indeed in R_j (line 23), since then N_j will have correctly the number of nodes in R_j. Since the machine didn't failed till now, $counter_{j-1}$ has to be equal to N_{j-1} (line 22), by the assumption of the induction. This means that the machine indeed found all $r \in R_{j-1}$, since $counter_{j-1}$ is incremented for each node that is found to be in R_{j-1} (line 19). Therefore, using Corrolary 4.2, we know that the machine changes the flag, $flag_w$, of a node w if only if $w \in R_j$. And this flag is the value that is added to $counter_j$ (line 23). Therefore, the counter is incremented if and only if $w \in R_j$. ■

Corollary 14 *Machine CR satisfies Theorem 4.*

Proof: We have shown in Lemma 4.2 that if the machine doesn't fail, it gives the right result. It is left to prove that there exists a computation in which the machine doesn't fail.

The correct computation is done as follows. For each node $r \notin R_{j-1}$, the machine guesses well in line 9 that indeed $r \notin R_{j-1}$ and stops working on this node. For each node $r \in R_{j-1}$, the machine guesses in line 9 so, and in addition it guesses correctly the nodes that form the directed path from v to r in line 14. In this computation, the machine will not fail. In lines 15 and 18, there is no failure, since only $r \in R_{j-1}$ nodes get to these lines, and in these lines it guesses corectly the connected path from v. Therefore, in line 22, all nodes $r \in R_{j-1}$ were counted, since the machine guesses them corectly, and the machine will not halt either. Thus, the machine doesn't fail on the above computation.

Using Lemma 4.2, we know that the machine uses $O(log(n))$ space. Therefore, CR is a non-deterministic machine that satisfies Theorem 4. ■