Lecture 10: NL and coNL

Anup Rao

April 25, 2024

In this lecture, we continue our discussion of space complexity classes. We first introduce a new definition. Given any set of boolean functions S, we write coS to denote the set

$$\{f : 1 - f \in S\}.$$

Thus $co\mathbf{NP}$ is the set of functions for which there is an efficiently verifiable proof that $f(x) = 0$.

Fact 1. $\mathbf{P} = co\mathbf{P}$

Fact 2. $\mathbf{L} = co\mathbf{L}$

Fact 3. $\mathbf{EXP} = co\mathbf{EXP}$

We do not know if $\mathbf{NP} = co\mathbf{NP}$. To show that $co\mathbf{NP} \subseteq \mathbf{NP}$, it would be enough to a polynomial time algorithm that can certify that a boolean formula is *unsatisfiable*.

Fact 4. *If* $\mathbf{P} = \mathbf{NP}$, *then* $\mathbf{NP} = co\mathbf{NP}$.

On the other hand, we can show:

Theorem 5. *For space constructible* $s(n)$, $\mathsf{NSPACE}(s(n)) = co\mathsf{NSPACE}(s(n))$.

Proof As usual we focus on the configuration graph. To prove the theorem, it will be enough to be able to verify that there is *no* path from two vertices u, v in the graph, in $s(n)$ space. This would show that if $f(x) = 1$ can be certified in space $s(n)$, then $f(x) = 0$ can also be certified in space $s(n)$. The other direction is completely symmetric.

We shall prove how to do this by designing a sequence of algorithms. Let C_i denote the set of vertices that are reachable from u in i steps. Suppose the graph is of size at most 2^s.

Claim 6. *Given any vertex* v *and a number* $i \leq 2^s$, *there is a non-deterministic space* $s(n)$ *algorithm such that:*

- *If* $v \in C_i$, *then some computational path outputs* 1
- *If* $v \notin C_i$, *then every computational path outputs* 0.

The algorithm simply guesses a path from u to v and checks that the path is a valid path of the graph by checking each edge in order.

Claim 7. *Given the size of $|C_{i-1}| = c$, and a vertex v, there is a non-deterministic space $s(n)$ algorithm such that*

- *If $v \notin C_i$, there is some computational path that outputs* 1.
- *If $v \in C_i$, then every computational path outputs* 0.

Since the algorithm is given the size of $C_{i-1}i$, the algorithm guesses each of the vertices of C_{i-1} in increasing order, and for each one, it checks that the vertex is different from the last vertex that was guessed, and then uses Claim 6 to verify that the vertex is indeed a member of C_{i-1}. It also makes sure that the given vertex is not v and not a neighbor of v. It maintains a count of all the number of vertices guessed and checks that $|C_{i-1}|$ vertices are given. If any of the checks fail, the algorithm outputs 0.

Finally, we argue that given the size of C_{i-1}, we can certify the size of $|C_i|$.

Claim 8. *Given the size of $|C_{i-1}| = c'$, there is a non-deterministic space $s(n)$ algorithm such that the algorithm either aborts or outputs $|C_i|$ on every computational path, and there is some computational path on which the algorithm outputs $|C_i|$.*

For each vertex v of the graph (in increasing order), the algorithm uses Claims 6 and 7 to check whether $v \in C_i$ or $v \notin C_i$, and it maintains a count of the number of vertices in C_i.

Thus, we obtain an algorithm that can verify that $v \notin C_n$ in $O(s(n))$ space. We first compute C_n by repeatedly using Claim 8 and then we apply Claim 7 to check whether $v \notin C_n$. ■

TQBF

The TQBF function maps the set of totally quantified boolean formulas to 0 or 1. A totally quantified boolean formula is something that looks like this:

$$\psi = \exists x_1 \forall x_2 \exists x_3 \cdots \exists x_n \phi(x_1, \ldots, x_n),$$

where here ϕ is a boolean formula on the variables $x_1, \ldots, x_n$.

$$\mathsf{TQBF}(\psi) = 1 \text{ if and only } \psi \text{ is true.}$$

TQBF's help characterize **PSPACE**.

Lemma 9. TQBF(ψ) *for $\psi = \exists x_1 \forall x_2 \exists x_3 \cdots \exists x_n \phi(x_1, \ldots, x_n)$ can be computed in space $O(m \cdot n)$, where size of ϕ is m. In other words,* TQBF $\in$ **PSPACE**.

Proof First note that for every fixing of $x_1, \ldots, x_n$, ϕ can be computed in space $O(m)$. Let

$$A = \forall x_2 \exists x_3 \cdots \exists x_n \phi(0, x_2, \ldots, x_n)$$

and

$$B = \forall x_2 \exists x_3 \cdots \exists x_n \phi(1, x_2, \ldots, x_n).$$

We know that $\mathsf{TQBF}(\psi) = \mathsf{TQBF}(A) \vee \mathsf{TQBF}(B)$ (similarly we will have to compute $A \wedge B$ when the first quantifier is a $\forall$). Writing down A takes at most $O(m)$ space. Let $S(n)$ denote the space required to compute $\mathsf{TQBF}(\psi)$. Now, computing A recursively uses $S(n-1)$ space. After computing A, we can store the answer (one bit) and erase all contents of the tape that was used to compute A. We then write down B and compute $\mathsf{TQBF}(B)$ recursively. Overall, we have that $S(n) = S(n-1) + O(m)$. As we know that $S(0) = O(m)$, we can conclude that $S(n) = O(m \cdot n)$. ■

Theorem 10. *For every boolean $f \in$ **PSPACE**, there is a polynomial time computable function g mapping bits to truly quantified boolean formulas such that $f(x) = \mathsf{TQBF}(g(x))$.*

Proof We shall show how to use the formula to encode connectivity in the configuration graph of the machine that computes f. This is a graph of size $2^t = 2^{\text{poly}(n)}$.

We generate a formula $\psi_i(A, B)$ in poly(n) time that checks whether there is a path of length $\leq 2^i$ from A to B. When $i = 0$. $\psi_i(A, B)$ just needs to check that B is the configuration that comes after A. Since we know that there is a polynomial sized circuit $\mathcal{C}$ such that $\mathcal{C}(x, A)$ computes the configuration that follows from A, we can construct a circuit $\mathcal{F}$ of size poly(n) such that

$$\mathcal{F}(A, B, x) = \begin{cases} 1 & \text{if } \mathcal{C}(A, x) = B, \\ 0 & \text{else.} \end{cases}$$

Just like in the proof that SAT is **NP**-complete, we can generate a polynomial sized formula $F(y)$ such that $\exists y F(y)$ is true if and only if $\mathcal{F}(A, B, x) = 1$.

For the general case, note that there is a path of length at most 2^i from A to B if and only if there is some vertex C in the graph such that there is a path of length 2^{i-1} from A to C and a path of length 2^{i-1} from C to A. Thus we can define

$$\psi_i(A, B) = \exists C, \psi_{i-1}(A, C) \wedge \psi_{i-1}(C, B).$$

However, this doubles the size of the formula ψ_{i-1} (which means that after t steps we will be trying to generate a formula that is exponentially big and this is impossible in polynomial time).

Indeed, we haven't yet used the $\forall$ quantifiers. Let us use the same idea as before to define the smaller formula:

$$\begin{aligned}
&\psi_i(A,B)\\
&= \exists C, \forall X, \forall Y, (X = A \wedge Y = C) \vee (X = C \wedge Y = B) \Rightarrow \psi_{i-1}(X,Y)\\
&= \exists C, \forall X, \forall Y, (\neg(X = A \wedge Y = C) \wedge \neg(X = C \wedge Y = B)) \vee \psi_{i-1}(X,Y)
\end{aligned}$$

The end result is a formula of size $\text{poly}(n,t)$ that checks for a path of length 2^t in the graph as required. ■

Lower Bounds on SAT

The material in this section was not discussed in class. We include it here as you might find it interesting. Although we cannot say anything non-trivial about the running time required to compute SAT, or the space required to compute SAT, we can show that SAT cannot have an algorithm that is both linear time and log space:

Theorem 11. *There is no turing machine computing* SAT *in $O(n)$ time and $O(\log n)$ space.*

In order to prove the theorem, we shall rely on two facts that we have convinced ourselves of before:

Theorem 12. *If $t(n) \geq \Omega(n)$, any $f \in \mathsf{NTIME}(t(n))$ can be reduced in in logarithmic space and time $O(t(n)\log(t(n)))$ to computing* SAT *on a formula of size $O(t(n)\log t(n))$.*

Earlier in the course we proved that the reduction is in polynomial time, but in fact it is even in **L**. (Think about this!). The reduction works by first computing a circuit that simulates the computation of a machine, and then computing the formula that simulates the execution of the circuit.

Another theorem we shall appeal to is the deterministic time hierarchy theorem:

Theorem 13 (Time Hierarchy)**.** *If r, t are time-constructible functions satisfying $r(n)\log r(n) = o(t(n))$, then $\mathsf{DTIME}(r(n)) \subsetneq \mathsf{DTIME}(t(n))$.*

Proof of Theorem 11: Assume for the purpose of contradiction that there is a turing machine computing SAT in $O(n)$ time and $O(\log n)$ space. The idea is to use the purported SAT algorithm to get an unreasonable speed up of computations. Suppose for the sake of contradiction that SAT can be computed in linear time and logarithmic space.

Suppose that $f \in \mathsf{DTIME}(n^2)$ via the machine M_f and $f \notin \mathsf{DTIME}(n\mathsf{polylog}(n))$. Such an f exists by Theorem 13. We shall show how to compute f in time $O(n\mathsf{polylog}(n))$, giving us the desired contradiction.

By appealing to Theorem 12, consider the machine M that runs as follows on input $x \in \{0,1\}^n$:

1. Generate the formula ϕ of size $n^2 \log n$ that simulates the machine $M_f(x)$, using Theorem 12.

2. Check whether $M_f(x)$ accepts by computing $\mathsf{SAT}(\phi)$ in time $O(n^2 \log n)$ and space $O(\log(n^2 \log n)) = O(\log n)$.

M is not our final simulation. M computes f in time $O(n^2 \log n)$ and space $O(\log n)$.

Consider the configuration graph of M. This graph accepts if and only if there is an accepting path of length $t = O(n^2 \log n)$, which happens if and only if there exist $\sqrt{t}$ intermediate configurations $C_1, \ldots, C_{\sqrt{t}}$, such that there is a path of length $\sqrt{t}$ between intermediate configurations. In other words, $f(x) = 1$ if and only if

$$\exists C_1, \ldots, C_{\sqrt{t}}, \forall i, C_i \text{ follows from } C_{i-1} \text{ in } \sqrt{t} \text{ steps.}$$

Each configuration takes only $O(\log n)$ bits to write down. So once we guess all of these $\sqrt{t}$ configurations, the problem of determining whether they determine an accepting of path of length t can be encoded using a SAT formula of size $O(\sqrt{t} \cdot \log n \cdot \mathsf{polylog}(t,n))$ (by Theorem 12), so it can be solved in deterministic time $O(\sqrt{t} \cdot \mathsf{polylog}(t,n))$. Thus we can compute a formula ψ of size $O(\sqrt{t} \cdot \mathsf{polylog}(t,n))$ such that $f(x) = 1$ if and only if

$$\exists C_1, \ldots, C_{\sqrt{t}}, \exists z, \psi(C_1, \ldots, C_{\sqrt{t}}, z).$$

The above is an instance of SAT and can then be solved deterministically in time $O(\sqrt{t} \cdot \mathsf{polylog}(n,t))$. Thus, overall, we get a simulation in deterministic time $O(\sqrt{t} \cdot \mathsf{polylog}(t,n)) = O(n\mathsf{polylog}(n)) = o(n^2)$, contradicting the deterministic time hierarchy theorem. ■

Randomized Algorithm review

A *probability space* is a set Ω such that every element $a \in \Omega$ is assigned a number $0 \leq \Pr[a] \leq 1$ (called the probability of a), and $\sum_{a \in \Omega} \Pr[a] = 1$.

An *event* in this space is a subset $E \subseteq \Omega$. The probability of the event is $\sum_{a \in E} \Pr[a]$. For example, imagine we toss a fair coin n times. Then the probability space consists of the 2^n possible outcomes of the coin tosses. If E is the event that the first k coin tosses are heads, this event has probability exactly 2^{-k}. Given two events E, E', we write $\Pr[E|E']$ to denote $\Pr[E \cap E'] / \Pr[E']$. This is the probability that E happens given that E' happens. We say that E, E' are independent if $\Pr[E \cap E'] = \Pr[E] \cdot \Pr[E']$. In other words, E, E' are independent if $\Pr[E|E'] = \Pr[E]$.

A *real valued random variable* is a function $X : \Omega \to \mathbb{R}$. The number of heads in the coin tosses is a random variable. The expected value of a random variable X is defined as $\mathbb{E}[X] = \sum_{a \in \Omega} \Pr[a] \cdot X(a)$. The following lemma is a very useful fact about random variables.

Lemma 14 (Linearity of expectation)**.** *If X, Y are real random variables, then $\mathbb{E}[X+Y] = \mathbb{E}[X] + \mathbb{E}[Y]$.*

Proof

$$\begin{aligned}\mathbb{E}[X+Y] &= \sum_{a \in \Omega} \Pr[a] \cdot (X(a) + Y(a)) \\ &= \sum_{a \in \Omega} \Pr[a] \cdot Y(a) + \sum_{a \in \Omega} \Pr[a] \cdot X(a) \\ &= \mathbb{E}[X] + \mathbb{E}[Y].\end{aligned}$$

■

Here is an expectation basic magic trick: Tell your audience to generate two sequences of coin tosses—one generated using 200 flips of a coin, and the second generated by hand. You leave the room, and they write both sequences on a black board. Then you come back into the room and immediately point out the sequence that was generated by hand. The trick: a random sequence is very likely to have a run of 7 heads or tails, while people tend to not insert such a long run into a sequence that they think looks random.

For example, let us calculate the expected number of runs of seeing 7 contiguous heads or tails in a 200 coin tosses. Let X_i be 1 if there are 7 heads or tails that start at the i'th position, and 0 otherwise. If $1 \leq i \leq 194$, then $\mathbb{E}[X_i] = \Pr[X_i = 1] = 2 \cdot 2^{-7} = 1/64$. If $i \geq 196$, then $X_i = 0$. On the other hand, the total number of such runs is $\sum_{i=1}^{194} X_i$. So by linearity of expectation, the expected number of such runs is $194/64 \approx 3.031$.

In class, we discussed the waiting time to see the first heads. Suppose you keep tossing a fair coin until you see heads. Let T be the number of tosses you make. What is the expected value of T? The key observation is that if the first toss is a heads, you stop with $T = 1$. Otherwise, the rest of the experiment is exactly the same as the original random experiment. So, we get:

$$\begin{aligned}&\mathbb{E}[T] = (1/2) \cdot 1 + (1/2) \cdot (1 + \mathbb{E}[T]) \\ \Rightarrow &\mathbb{E}[T] \cdot (1 - 1/2) = 1 \\ \Rightarrow &\mathbb{E}[T] = 2.\end{aligned}$$

Randomized Algorithms

We shall give a few examples of problems where randomness helps to give very effective solutions.

Matrix Product Checking

Suppose we are given three $n \times n$ matrices A, B, C, and want to check whether $A \cdot B = C$. One way to do this is to just multiply the matrices, which will take much more than n^2 time. Here we give a randomized algorithm that takes only $O(n^2)$ time.

[H] 3 $n \times n$-matrices A, B, C Whether or not $A \cdot B = C$. Sample an n coordiante column vector $r \in \{0,1\}^{0,1}$ uniformly at random A(B(r)) = C(r)Output "Equal" Output "Not equal" Algorithm for Multiplication Checking

The algorithm only takes $O(n^2)$ time. For the analysis, observe that if $AB = C$, then the algorithm outputs "Equal" with probability 1. If $AB \neq C$, the algorithm outputs "Equal" only when $ABr = Cr \Rightarrow (AB - C)r = 0$. We shall show that this happens with probability at most $1/2$.

Let $D = AB - C$. Then $D \neq 0$, so let d_{ij} be a non-zero entry of D. Then we have that the i'th coordinate $(Dr)_i = \sum_k d_{ik} \cdot r_k$. This coordinate is 0 exactly when $r_j = (1/d_{ij}) \sum_{k \neq j} d_{ik} r_k$. Finally, observe

$$\begin{aligned}
&\Pr\left[r_j = (1/d_{ij}) \sum_{k \neq j} d_{ik} r_k\right] \\
&= \sum_a \Pr\left[a = (1/d_{ij}) \sum_{k \neq j} d_{ik} r_k\right] \cdot \Pr\left[r_j = a | a = (1/d_{ij}) \sum_{k \neq j} d_{ik} r_k\right] \\
&\leq 1/2 \sum_a \Pr\left[a = (1/d_{ij}) \sum_{k \neq j} d_{ik} r_k\right] \\
&= 1/2.
\end{aligned}$$

Exercise: Modify the above algorithm so that the probability the algorithm outputs "Equal" when $AB \neq C$ is at most $1/4$.