Notes for Lecture 3

Today we will prove that **NL** = **coNL** and discuss the polynomial hierarchy.

1 NL = coNL

In order to prove that these two classes are the same, we will show that there is an **NL** Turing machine which solves $\overline{\text{STCONN}}$. $\overline{\text{STCONN}}$ is the problem of deciding, given a directed graph G, together with special vertices s and t, whether t is *not* reachable from s. Note that $\overline{\text{STCONN}}$ is **coNL**-complete.

Once we have the machine, we know that **coNL** $\subseteq$ **NL**, since any language A in **coNL** can be reduced to $\overline{\text{STCONN}}$, and since $\overline{\text{STCONN}}$ has been shown to be in **NL** (by the existence of our machine), so is A. Also, **NL** $\subseteq$ **coNL**, since if $\overline{\text{STCONN}} \in$ **NL**, by definition STCONN $\in$ **coNL**, and since STCONN is **NL**-complete, this means that any problem in **NL** can be reduced to it and so is also in **coNL**. Hence **NL** = **coNL**.

1.1 A simpler problem first

Now all that remains to be shown is that this Turing machine exists. First we will solve a simpler problem than $\overline{\text{STCONN}}$. We will assume that in addition to the usual inputs G, s and t, we also have an input r, which we will assume is equal to the number of vertices reachable from s in G, including s.

Given these inputs, we will construct a non-deterministic Turing machine which decides whether t is reachable from s by looking at all subsets of r vertices in G, halting with YES if it sees a subset of vertices which are all reachable from s but do not include t, and halting with NO otherwise. Here is the algorithm:

input: $G = (V, E)$, s, t, r
output: YES if it discovers that t is not reachable from s, and NO otherwise
assumption: there are exactly r distinct vertices reachable from s

$c \leftarrow 0$
for all $v \in (V - \{t\})$ **do**
  non-deterministically guess if v is reachable from s
  if guess = YES **then**
    non-deterministically guess the distance k from s to v
    $p \leftarrow s$
    for $i \leftarrow 1$ to k **do**
      non-deterministically pick a neighbor q of p
      $p \leftarrow q$
    if $p \neq v$, reject
    $c \leftarrow c + 1$
if $c = r$ **then** return YES, otherwise return NO

It is easy to verify that this algorithm is indeed in **NL**. The algorithm only needs to maintain the five variables c, k, p, q, v, and each of these variables can be represented with $\log |V|$ bits.

Regarding correctness, notice that, in the algorithm, c can only be incremented for a vertex v that is actually reachable from s. Since there are assumed to be exactly r such vertices, c can be at most r at the end of the algorithm, and if it is exactly r, that means that there are r vertices other than t which are reachable from s, meaning that t by assumption cannot be reachable form s. Hence the algorithm accepts if and only if it discovers that t is not reachable from s.

1.2 Finding r

Now we need to provide an **NL**-algorithm that finds r. Let's first try this algorithm:

input: $G = (V, E)$, s
output: the number of vertices reachable from s (including s in this count)

$c \leftarrow 0$
for all $v \in V$ **do**
    non-deterministically guess if v is reachable from s in k steps
    if guess = YES **then**
        $p \leftarrow s$
        for $i \leftarrow 1$ to k **do**
            non-deterministically guess a neighbor q of p (possibly not moving at all)
            $p \leftarrow q$
        if $p \neq v$ reject
        $c \leftarrow c + 1$
return c

This algorithm has a problem. It will only return a number c which is at most r, but we need it to return *exactly* r. We need a way to force it to find all vertices which are reachable from s. Towards this goal, let's define r_i to be the set of vertices reachable from s in at most i steps. Then $r = r_{n-1}$, where n is the number of vertices in G. The idea is to try to compute r_i from r_{i-1} and repeat the procedure $n - 2$ times. Now here is another try at an algorithm:

input: $G = (V, E)$, s, i, r_{i-1}
output: the number of vertices reachable from s in at most i steps (including s in this count)
assumption: r_{i-1} is the exact number of vertices reachable from s in at most $i - 1$ steps

$c \leftarrow 0$
for all $v \in V$ **do**
    $d \leftarrow 0$
    $flag \leftarrow FALSE$
    for all $w \in V$ **do**
        non-deterministically guess if w is reachable from s in at most $k - 1$ steps

if guess = YES **then**
  $p \leftarrow s$
  for $i \leftarrow 1$ to $k-1$ **do**
    non-deterministically pick a neighbor q of p (possibly not moving at all)
    $p \leftarrow q$
  if $p \neq v$ **then** reject
  $d \leftarrow d + 1$
  if v is a neighbor of w, or if $v = w$ **then**
    $flag \leftarrow TRUE$
if $d < r_{i-1}$ reject
if $flag$ **then** $c \leftarrow c + 1$
return c

Here is the idea behind the algorithm: for each vertex v, we need to determine if it is reachable from s in at most i steps. To do this, we can loop over all vertices which are a distance at most $i - 1$ from s, checking to see if v is either equal to one of these vertices or is a neighbor of one of them (in which case it would be reachable in exactly i steps). The algorithm is able to force all vertices of distance at most $i - 1$ to be considered because it is given r_{i-1} as an input.

Now, putting this algorithm together with the first one listed above, we have shown that $\overline{\text{STCONN}} \in \mathbf{NL}$, implying that $\mathbf{NL} = \mathbf{coNL}$. In fact, the proof can be generalized to show that if a decision problem A is solvable in non-deterministic space $s(n) = \Omega(\log n)$, then $\overline{A}$ is solvable in non-deterministic space $O(s(n))$.

2 The polynomial hierarchy

One way to look at the difference between **NP** and **coNP** is that a decision problem in **NP** is asking a sort of "does there exist" question, where the existence of the answer can by definition be efficiently represented. On the other hand, **coNP** asks "is it true for all" questions, which do not seem to have simple, efficient proofs.

Formally, a decision problem A is in **NP** if and only if there is a polynomial time procedure $V(\cdot, \cdot)$ and a polynomial time bound $p()$ such that

$$x \in A \text{ if and only if } \exists y.|y| \leq p(|x|) \wedge V(x, y) = 1$$

and a problem A is in **coNP** if and only if there is a polynomial time procedure $V(\cdot, \cdot)$ and a polynomial bound $p()$ such that

$$x \in A \text{ if and only if } \forall y : |y| \leq p(|x|), V(x, y) = 1$$

2.1 Stacks of quantifiers

Now suppose you had a decision problem A which asked something of the following form:

$$x \in A \Leftrightarrow \exists\, y_1 \text{ s.t. } |y_1| \leq p(|x|)\ \forall\, y_2 \text{ s.t. } |y_2| \leq p(|x|)\ V(x, y_1, y_2)$$

(where $p()$ is a polynomial time bound and $V(\cdot, \cdot, \cdot)$ is a polynomial time procedure.)

In other words, an algorithm solving problem A should return YES on an input x if an only if there exists some string y_1 such that for all strings y_2 (both of polynomial length), the predicate $V(x, y_1, y_2)$ holds. An example of such a problem is this: given a Boolean formula ϕ over variables $x_1, \ldots, x_n$, is there a formula ϕ' which is equivalent to ϕ and is of size at most k? In this case, y_1 is the formula ϕ', y_2 is an arbitrary assignment to the variables $x_1, \ldots, x_n$, and $V(x, y_1, y_2)$ is the predicate which is true if and only if $x[y_2]$ and $y_1[y_2]$ are both true or both false, meaning that under the variable assignment y_2, ϕ and ϕ' agree. Notice that ϕ' is equivalent to ϕ if and only if it agrees with ϕ under all assignments of Boolean values to the variables.

As we will see, the problem A is a member of the class Σ_2 in the second level of the polynomial hierarchy.

2.2 The hierarchy

The polynomial hierarchy starts with familiar classes on level one: $\Sigma_1 = \mathbf{NP}$ and $\Pi_1 = \mathbf{coNP}$. For all $i \geq 1$, it includes two classes, Σ_i and Π_i, which are defined as follows:

$$A \in \Sigma_i \Leftrightarrow \exists y_1.\ \forall y_2.\ \ldots\ .Qy_i.\ V_A(x, y_1, \ldots, y_i)$$

and

$$B \in \Pi_i \Leftrightarrow \forall y_1.\ \exists y_2.\ \ldots\ .Q'y_i.\ V_B(x, y_1, \ldots, y_i)$$

where the predicates V_A and V_B depend on the problems A and B, and Q and Q' represent the appropriate quantifiers, which depend on whether i is even or odd (for example, if $i = 10$ then the quantifier Q for Σ_{10} is $\forall$, and the quantifier Q' for Π_{10} is $\exists$). For clarity, we have also omitted the $p(\cdot)$ side conditions, but they are still there.

One thing that is easy to see is that $\Pi_k = \text{co}\Sigma_k$. Also, note that, for all $i \leq k-1$, $\Pi_i \subseteq \Sigma_k$ and $\Sigma_i \subseteq \Sigma_k$. These subset relations hold for Π_k as well. This can be seen by noticing that the predicates V do not need to "pay attention to" all of their arguments, and so can represent classes lower on the hierarchy which have a smaller number of them.

2.3 An alternate characterization

The polynomial hierarchy can also be characterized in terms of "oracle machines." The idea here is that, instead of a standard Turing machine, we consider one which is augmented with an oracle of a certain power which can be consulted as many times as desired, and using only one computational step each time. Syntactically, this can be written as follows.

Let A be some decision problem and $\mathcal{M}$ be a class of Turing machines. Then $\mathcal{M}^A$ is defined to be the class of machines obtained from $\mathcal{M}$ by allowing instances of A to be solved in one step. Similarly, if $\mathcal{M}$ is a class of Turing machines and $\mathcal{C}$ is a complexity class, then $\mathcal{M}^{\mathcal{C}} = \bigcup_{A \in \mathcal{C}} \mathcal{M}^A$. If L is a complete problem for $\mathcal{C}$, and the machines in $\mathcal{M}$ are powerful enough to compute polynomial-time computations, then $\mathcal{M}^{\mathcal{C}} = \mathcal{M}^L$.

Theorem 1 $\Sigma_2 = \mathbf{NP}^{3SAT}$.

PROOF: Let $A \in \Sigma_2$, then for some polynomial $p()$ and polynomial-time computable $V()$ we have

$$x \in A \text{ if and only if } \exists y_1 \text{ s.t. } |y_1| \leq p(|x|).\forall y_2 \text{ s.t. } |y_2| \leq p(|x|).V(x, y_1, y_2) = 1$$

Then we define a non-deterministic machine with an **NP**-oracle as follows: on input x, the machine guesses a string y_1 of length at most $p(|x|)$, and then asks the oracle whether $\exists y_2.|y_2| \leq p(|x|) \wedge V(x, y_1, y_2) = 0$. The above question is an existential question about a polynomial-time computation, so, by Cook's theorem, it is possible to construct in polynomial time a 3SAT instance that is satisfiable if and only if the answer to the above question is YES. The machine accepts if and only if the answer from the oracle is NO. It is immediate that the machine has an accepting computation if and only if

$$\exists y_1.|y_1| \leq p(|x|).(\neg \exists y_2.|y_2| \leq p(|x|) \wedge V(x, y_1, y_2) = 0)$$

that is, the machine accepts if and only if $x \in A$.

Notice that, in the above computation, only one oracle query is made, even though the definition of $\mathbf{NP}^{3SAT}$ allows us to make an arbitrary number of oracle queries.

Let now $A \in \mathbf{NP}^{3SAT}$, and let M be the oracle machine that solves A. We first show that there is a machine M' that also solves A, only makes one oracle query, and accepts if and only if the answer to the oracle query is NO. On input x, M' guesses an accepting computation of $M(x)$, that is, M' guesses all the non-deterministic choices of $M(x)$, all the oracle questions, and all the answers. Then, for each question that was answered with a YES, M' guesses a satisfying assignment to verify that the guess was correct. Finally, M' is left with a certain set of oracle questions, say, the formulae $\phi_1, \ldots, \phi_k$, for which it has guessed that the correct oracle answer is NO. Then M' asks its oracle whether (a formula equivalent to) $\phi_1 \vee \cdots \vee \phi_k$ is satisfiable, and it accepts if and only if the answer is NO.

Consider the computation of $M'(x)$ when $x \in A$: there is a valid accepting computation of $M(x)$, and $M'(x)$ can guess that computation along with the valid oracle answers; it can also guess valid assignments for all the queries for which the answer is YES; finally, it is left with unsatisfiable formulae $\phi_1, \ldots, \phi_k$, the answer to the single oracle query of M' is NO, and M' accepts.

Conversely, if $M'(x)$ has an accepting computation, then there must be a valid accepting computation of $M(x)$, and so $x \in A$. $\square$

In fact, a more general result is known, whose proof works along similar lines.

Theorem 2 *For every* $i \geq 2$, $\Sigma_i = \mathbf{NP}^{\Sigma_{i-1}}$.

2.4 Additional Properties

Here are some more facts about the polynomial hierarchy, which we will not prove:

1. Π_i and Σ_i have complete problems for all i.
2. A Σ_i-complete problem is not in Π_j, $j \leq i-1$, unless $\Pi_j = \Sigma_i$, and it is not in Σ_j unless $\Sigma_j = \Sigma_i$.
3. Suppose that $\Sigma_i = \Pi_i$ for some i. Then $\Sigma_j = \Pi_j = \Sigma_i = \Pi_i$ for all $j \geq i$.
4. Suppose that $\Sigma_i = \Sigma_{i+1}$ for some i. Then $\Sigma_j = \Pi_j = \Sigma_i$ for all $j \geq i$.
5. Suppose that $\Pi_i = \Pi_{i+1}$ for some i. then $\Sigma_j = \Pi_j = \Pi_i$ for all $j \geq i$.

We will just prove the following special case of part (3).

Theorem 3 *Suppose* $\mathbf{NP} = \mathbf{coNP}$. *Then, for every* $i \geq 2$, $\Sigma_i = \mathbf{NP}$.

PROOF: Let us first prove that, under the assumption of the theorem, $\Sigma_2 = \mathbf{NP}$. Let $A \in \Sigma_2$ and let M be the non-deterministic oracle machine that decides A using oracle access to 3SAT. Let also M' be the non-deterministic polynomial time Turing machine that decides the complement of the 3SAT problem. We now describe a non-deterministic polynomial time Turing machine M'' to decide A: on input x, M'' guesses an accepting computation of $M(x)$, along with oracle queries and answers; for each oracle question ϕ for which a YES answer has been guessed, M'' guesses a satisfying assignment; for each oracle question ψ for which a NO answer has been guessed, M'' guesses an accepting computation of $M'(\psi)$. It is easy to verify that $M''(x)$ has an accepting computation if and only if $M^{3SAT}(x)$ has an accepting computation.

We can prove by induction on i that $\Sigma_i = \mathbf{NP}$. We have covered the base case. Let us now suppose that $\Sigma_{i-1} = \mathbf{NP}$; then $\Sigma_i = \mathbf{NP}^{\Sigma_{i-1}} = \mathbf{NP}^{\mathbf{NP}} = \Sigma_2 = \mathbf{NP}$. $\Box$

While it seems like an artificial construction right now, in future lectures we will see that the polynomial hierarchy helps us to understand other complexity classes.

Exercises

1. Consider the following decision problem: given a directed graph $G = (V, E)$, two vertices $s, t \in V$, and an integer k, determine whether the shortest path from s to t is of length k. Is this problem in **NL**?

2. In the MAX SAT problem we are given a formula ϕ in conjunctive normal form and we want to find the assignment of values to the variables that maximizes the number of satisfied clauses. (For example, if ϕ is satisfiable, the optimal solution satisfies all the clauses and the MAX SAT problem reduces to finding a satisfying assignment.) Consider the following decision problem: given a formula ϕ in conjunctive normal form and an integer k, determine if k is the number of clauses of ϕ satisfied by an optimal assignment.

   - Prove that this problem is in **NP** if and only if **NP** = *co***NP**.
     [Hint: prove that it is both **NP**-hard and *co***NP**-hard.]
   - Prove that this problem is in Σ_2.