Lecture 5

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

$c := 0$
for all $v \in (V - \{t\})$ **do**
    non-deterministically guess if v is reachable from s in k steps
    if guess = YES **then**
        $p := s$
        for $i := 1$ to k **do**
            non-deterministically pick a neighbor q of p
            p := q
        if p is not equal to v, reject
        $c := c + 1$

if $c = r$ **then** return YES, otherwise return NO

Exercise: verify that this algorithm is indeed in **NL**.

Notice that in the algorithm above, 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)

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

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

$c := 0$
for all $v \in V$ **do**
  $d := 0$
  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**
  $p := s$
  for $i := 1$ to $k-1$ **do**
    non-deterministically pick a neighbor q of p (possibly not moving at all)
    $p := q$
  if $p \neq v$ **then** reject
  $d := d + 1$
  if v is a neighbor of w, or if $v = w$ **then**
    $c := c + 1$
    break out of the inner loop and start the next iteration of the outer loop
if $d < r_{i-1}$ reject

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. In this way, complexity theorists think of there being a $\exists$ associated with **NP** and a $\forall$ associated with **coNP**.

2.1 Stacks of quantifiers

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

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

In other words, a Turing machine 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 $\Pi_k = \mathrm{co}\Sigma_k$. 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.

Here are some more facts about the polynomial hierarchy (proof left to the reader):

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 $j \geq i$.

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.

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. For example, $\mathbf{NP}^{\mathrm{3SAT}} = \Sigma_2$.

In general, let L be some Σ_{i-1}-complete problem. Then it can be verified that $\Sigma_i = \mathbf{NP}^L$ and $\Pi_i = \mathbf{coNP}^L$.