Problem Set 1, Answers

Kevin Lacker

May 18, 2020

Problem 1. A language L is in $\Sigma_2^{\mathbf{P}}$ iff there is a polynomial time TM M such that:

$$x \in L \iff \exists u_1 \forall u_2 M(x, u_1, u_2) \tag{1}$$

where the u_i are polynomial size and we treat M as returning true or false. This is equivalent to:

$$x \in L \iff \exists u_1 \neg(\exists u_2(\neg M(x, u_1, u_2)) \tag{2}$$

The answer to $\exists u_2(\neg M(x, u_1, u_2))$ can be found in a single call to an **NP** oracle, so $\Sigma_2^{\mathbf{P}} \in \mathbf{NP}^{\mathbf{NP}}$.

The other direction is similar. For any language L in $\mathbf{NP}^{\mathbf{NP}}$, there is a Turing machine M, where M has access to an **NP** oracle, such that:

$$x \in L \iff \exists v_1 M(x, v_1) \tag{3}$$

There are different equivalent formats for the oracle. It is convenient to think of the input as being a SAT problem, and the output is either a solution or an indication that there is no solution.

To write this as a $\Sigma_2^{\mathbf{P}}$ language we need to convert the oracle calls to a regular Turing machine under a $\forall$ quantifier. So, let v_2 represent the oracle outputs, and v_3 represent a string of bits long enough to hold all assignments to the SAT problems where the oracle reported no solution. M makes polynomially many calls to the oracle, so the lenght of v_2 and v_3 can be polynomially bounded.

Then, we can represent L as:

$$x \in L \iff \exists v_1 v_2 \forall v_3 M'(x, v_1, v_2, v_3) \tag{4}$$

Where M' works like:

- It performs the same operations as M, except when there is an oracle call
- When the oracle gives a solution, M' validates the solution
- When the oracle reports no solution, M' uses bits from v_3 to check that the particular inputs do not represent a solution

Since this is quantified over all v_3, the quantifiers do the job of the NP oracle, and this is an equivalent representation for L. Thus $\mathbf{NP^{NP}} \in \mathbf{\Sigma_2^P}$.

Problem 2. Assume for the sake of contradiction that $NP = SPACE(n)$. For any $L \in SPACE(n^2)$, we can pad the length to n^2 to get a language that is in $SPACE(n)$. By our assumption, this language is in NP.

However, if this padded language is in NP, it must also be in NP without the padding, since a nondeterministic machine can add the padding, and runtime polynomial in n^2 is also polynomial in n. So L is in NP as well. But since $NP = SPACE(n)$, this shows that $SPACE(n^2) \subset SPACE(n)$, which contradicts the space hierarchy theorem.

Problem 3. TODO

Problem 4. Assume for the sake of contradiction that $\Sigma_2^P \in SIZE[n^k]$. Since $NP \in \Sigma_2^P$, this implies $NP \in P/poly$, and so by the Karp-Lipton theorem, the polynomial hierarchy collapses, with $\Sigma_2^P = \Sigma_3^P = PH$. We can then substitute into our assumption to get $\Sigma_3^P \in SIZE[n^k]$.

However, Σ_3^P cannot have polynomial circuits, because it has enough power to find a circuit with a given complexity. It will be useful to have a total ordering of all circuits, first by number of gates, with tiebreaks broken lexicographically. So a "smaller" circuit can either be one with fewer gates, or one earlier lexicographically. Then consider this chain of problems:

- Given two circuits, is there any input where their result differs? This problem is in NP.
- Given a circuit, is there any smaller circuit that computes the same result on all inputs? This can be solved in NP with an oracle for the previous problem, so it is in $NP^{NP} = \Sigma_2^P$.
- Find the smallest circuit of size n^{k+1} that has no smaller circuit that computes the same result on all inputs. (This exists, by a counting argument.) This can be solved in NP with an oracle for the previous problem, so it is in $NP^{NP^{NP}} = \Sigma_3^P$.

A function that computes the result of the circuit output by the last algorithm has no circuits smaller than n^{k+1}, but it is in Σ_3^P, which leads us to a contradiction.

Problem 5. Assume we have $P = NP$ and consider the $NEXP$-complete problem of $SUCCINCT\text{-}SAT$. In exponential time, we can expand out the truth table of the compressed SAT problem into an exponentially long SAT problem. We can then use our polynomial-time algorithm for solving NP problems on this exponentially long SAT problem. It takes time that is the polynomial of an exponential, but this is still contained in EXP. So $P = NP$ implies $EXP = NEXP$.

In particular, when $P = NP$ you can find a circuit that requires at least x gates with time polynomial in x. By the reasoning in the previous problem, this task is in Σ_3^P, and when $P = NP$ the hierarchy collapses and this task is in P.

We know from Shannon's counting argument that there exists some function on n bits that requires $2^n/n$ gates. So we just have to find it. We can use this algorithm, and it takes time polynomial in $2^n/n$. A polynomial function of 2^n is bounded by 2^{cn} for some c, and these algorithms are contained in EXP. So if $P = NP$, the language determined by the first such circuit is in EXP.

(This didn't use $EXP = NEXP$ directly, as hinted, but I think the reasoning is simpler this way.)