

# Problem Set 1, Answers

Kevin Lacker

May 21, 2020

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

$$x \in L \iff \exists u_1 \forall u_2 M(x, u_1, u_2) \quad (1)$$

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

$$x \in L \iff \exists u_1 \neg(\exists u_2 (\neg M(x, u_1, u_2))) \quad (2)$$

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  $\Sigma_2^P \in \mathbf{NP}^{\mathbf{NP}}$ .

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:

$$x \in L \iff \exists v_1 M(x, v_1) \quad (3)$$

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  $\Sigma_2^P$  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 length of  $v_2$  and  $v_3$  can be polynomially bounded.

Then, we can represent  $L$  as:

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

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  $\mathbf{NP}^{\mathbf{NP}} \in \Sigma_2^P$ .

**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.** A language in  $TISP[t, s]$  can be interpreted as a graph problem. Each node is one configuration in the configuration space of the Turing machine, with a single configuration describable by  $s$  bits. Our task is to find the path of length  $t$  through configuration space and see if it ends at a “success” state.

The standard proof involves splitting this path into a number of equal parts. We are going to split the path into  $p$  equal parts, and recursively do that  $k$  times. Call the overall path a 0-path, and in general let an  $i$ -path be split into  $p$  equal  $i + 1$ -paths. Then, our algorithm on an alternating Turing machine can work like:

- There exists a way to split the 0-path into  $p$  1-paths, defined by the endpoints of the 1-paths,  $C_{1,0}, C_{1,1}, \dots, C_{1,p}$ , such that...
- For all choices of a 1-path, defined by consecutive  $C_{1,i}$  and  $C_{1,i+1}$ ...
- (repeat recursively)
- There exists a way to split this  $k - 1$ -path into  $p$   $k$ -paths, defined by the endpoints of the  $k$ -paths,  $C_{k,0}, C_{k,1}, \dots, C_{k,p}$ , such that...
- For all choices of a  $k$ -path, defined by consecutive  $C_{k,i}$  and  $C_{k,p}$
- There exists a set of nodes that defines a  $k$ -path with those endpoints.

We also need to check that our endpoint is a “success” state but we can do that at the first recursion level.

How much time does this alternating Turing machine take? It will depend on the choice of  $p$ , so we can choose  $p$  to optimize time. The “exists” clauses are selecting  $p$  variables of size  $s$ , so they take  $ps$  time. The interior “for all” clauses are smaller, so they don’t matter.  $k$  is a constant so the  $k$ -fold repetition can be ignored as well. The final path is a  $p^{-k}$  fraction of the total path, so it takes  $tp^{-k}$  time to calculate. So the total time is

$$O(ps + tp^{-k}) \tag{5}$$

This is minimized when the two summands are equal, by choosing  $p = (t/s)^{1/(k+1)}$ . Then the runtime becomes  $O((ts^k)^{1/(k+1)})$ . Since there are  $2k$  levels of alternation in this algorithm, we have:

$$TISP[t, s] \subset \Sigma_{2k}TIME[(ts^k)^{1/(k+1)}] \quad (6)$$

To improve this, note that  $TISP$  is closed under complement. In our path-finding algorithm, instead of finding a path to a “success” state, we can also be finding the absence of a path to a “failure” state.

In our recursion, each level is finding a slightly shorter path, and adding extra  $\exists\forall$  quantifiers to the formula provided when searching for the shorter path. Instead of adding on two extra quantifiers each time, we can first negate the whole formula, so that we get one that starts with a  $\forall$  instead of a  $\exists$ . Then, when we recurse we have only added one extra switching level.

Since we negate at every step, at the end of the whole recursion, we may have found the complement of our desired language instead of the desired language. But that is equivalently hard since  $TISP$  is closed under complement.

Thus, we can use this improved recursion which adds only one rather than two alternations per recursion, to demonstrate that:

$$TISP[t, s] \subset \Pi_{k+1}TIME[(ts^k)^{1/(k+1)}] \quad (7)$$

**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,  $\Sigma_3^P$  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  $\Sigma_3^P$ , which leads us to a contradiction.

**Problem 5.** Assume we have  $P = NP$  and consider the  $NEXP$ -complete problem of  $SUCCINCT-SAT$ . 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  $\Sigma_3^P$ , 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.)

**Problem 6.** So we have a polynomial-time algorithm that can use a single query to an  $NP$  oracle to determine the satisfiability to two  $3SAT$  formulas  $\Phi_1$  and  $\Phi_2$ . We can remove the oracle and run the algorithm twice. The first time, we pretend the oracle returns a "true". The second time, we pretend the oracle returns a "false". Each time our algorithm gives us a set of two answers - whether  $\Phi_1$  is satisfiable, and whether  $\Phi_2$  is satisfiable. And we know one of these two sets is correct, although we don't have the oracle so we don't know which set it is. You can think of this as two "branches" of the algorithm. We get an answer from each branch, and we don't know which branch is the correct one.

So each branch can report one of four things:

- $\Phi_0$  and  $\Phi_1$  are both satisfiable
- neither of  $\Phi_0$  and  $\Phi_1$  are satisfiable
- $\Phi_0$  is satisfiable,  $\Phi_1$  is not
- $\Phi_1$  is satisfiable,  $\Phi_0$  is not

What will we learn from these results? If both branches report the same thing about either formula, this tells us the answer for that formula. If the branches do not agree on either formula, there are two cases to consider. One branch could be saying "both satisfiable" and the other could be saying "neither". In this case, we learn that the formulas have the same satisfiability. Or, the branches could both be saying different formulas are the satisfiable ones. In this case, we learn that the formulas have opposite satisfiability. In other words, we have a polynomial-time algorithm that tells us one of the following things:

- $\Phi_0$  is satisfiable
- $\Phi_0$  is not satisfiable
- $\Phi_1$  is satisfiable

- $\Phi_1$  is not satisfiable
- Either both or neither of the  $\Phi_i$  are satisfiable
- Precisely one of the  $\Phi_i$  are satisfiable

We can now use this recursively to solve *3SAT* problems. For a formula  $\Phi$ , create two smaller formulas. Substitute  $x_0 = 0$  to get  $\Phi_0$ , and substitute  $x_0 = 1$  to get  $\Phi_1$ . Now,  $\Phi$  is satisfiable if either  $\Phi_0$  or  $\Phi_1$  is satisfiable.

We run our algorithm on  $\Phi_0$  and  $\Phi_1$  and the way we recurse depends on the information we get.

- If either  $\Phi_i$  is satisfiable,  $\Phi$  is satisfiable and we are done.
- If either  $\Phi_i$  is not satisfiable, we can recurse on the other one.
- If the  $\Phi_i$  are either both or neither satisfiable, we can recurse on either one.
- If precisely one is satisfiable, then  $\Phi$  is satisfiable and we are done.

Each time we recurse it takes polynomial time and we remove one free variable, so the whole algorithm will take polynomial time, and we have a polynomial time algorithm for solving *3SAT*, so  $P = NP$ .