Problem Set 2, Answers

Kevin Lacker

May 22, 2020

Problem 1. The idea is to use MAJ circuits to calculate PARITY, and thus use our lower bound on PARITY circuits to provide a lower bound on MAJ circuits.

First, we can create a circuit for the function that counts whether precisely half of the bits of the input are 1. I.e., $HALF(x) = 1$ iff $|x| = n/2$, with two MAJ circuits:

$$HALF(x) = MAJ(x) \wedge MAJ(\neg x) \tag{1}$$

Here $\neg x$ represents the bitwise negation of x. Our circuit for for HALF uses two MAJ subcircuits and one more depth.

Now that we have HALF, we can construct a circuit to count whether precisely k bits of the input are 1, $EXACT_k$, by padding the input with ones or zeros and using a single HALF circuit on at most twice the input size.

We can then take one $EXACT_K$ for each odd number less than or equal to n, and take their conjunction to create a PARITY circuit of depth $d + 2$. This PARITY circuit of depth $d+2$ is built of $O(n)$ depth-d MAJ circuits, plus $O(n)$ extra gates, where each MAJ circuit has at most $2n$ inputs.

Let $H_{MAJ}(n, d)$ denote the minimum size of a circuit of depth d calculating MAJ on a size-n input, and H_{PARITY} the same for PARITY. This construction demonstrates that

$$O(n) \cdot H_{MAJ}(n, d) \geq H_{PARITY}(n/2, d + 2) \tag{2}$$

But we know that

$$H_{PARITY}(n, d) \geq exp(\Omega(n^{2^{-d}})) \tag{3}$$

Substituting in and simplifying we get

$$H_{MAJ}(n, d) \geq exp(\Omega(n^{2^{-d-O(1)}})) \tag{4}$$

Problem 2. Let's say we have an estimate v of $\Sigma_x f(x)$ and we want to improve that estimate. We define:

$$g(x) = \begin{cases} f(0) - v & \text{for } x = 0 \\ f(x) & \text{otherwise} \end{cases} \tag{5}$$

We can then use our estimation oracle to get an estimation for $\Sigma_x g(x) = \Sigma_x f(x) - E$. Basically, we are estimating the error in our original estimate. When we add this estimate to E, we are improving our estimate to $\Sigma_x f(x)$. If our estimation oracle returns a value within a factor of $1 \pm \epsilon$ of the correct value, then the size of our error shrinks by a factor of ϵ every iteration.

Recursing until our error is below 1 thus takes time that is linear in the size of the output value, so we can use this to get an exact answer to $\Sigma_x f(x)$ in polynomial time, assuming the answer has polynomial length.

We can use this to exactly solve problems in $\sharp SAT$. Let x represent a possible solution to a $\sharp SAT$ problem, and $f(x) = 1$ when x is a solution, 0 otherwise. Since $f(x)$ is a constant, the sum has polynomial length, and our algorithm takes polynomial time.

$\sharp SAT$ is $\sharp P$-complete so this solves any $\sharp P$ problem in polynomial time.

Problem 3. Say we have a polynomial that agrees with OR over $\{0,1\}^n$ with degree less than n, $P_{OR}(x_0, x_1, ...x_n) = OR(x_0, x_1, ...x_n)$. Then we can substitute variables to get another polynomial with degree less than n that represents AND:

$$P_{AND}(x_0, x_1, ..., x_n) = 1 - P_{OR}(1 - x_0, 1 - x_1, ..., 1 - x_n) \tag{6}$$

Now consider any string of n bits, $y = y_0 y_1 ... y_n$. We can construct a polynomial of degree less than n that is 1 on this input and 0 on all the other inputs in $\{0,1\}^n$:

$$P_y(x_0, x_1, ..., x_n) = P_{AND}(x_0 - y_0, x_1 - y_1, ..., x_n - y_n) \tag{7}$$

Now these functions act as a basis if you think of this as a vector space. So any function $f(x)$ from $\{0,1\}^n \to \mathbb{Z}_q^n$ can be represented as a polynomial of degree less than n:

$$f(x) = \Sigma_{y \in \{0,1\}^n} P_y(x) f(y) \tag{8}$$

There are q^{2^n} different such functions, as defined by their values on $\{0,1\}^n$. But there are only q^{2^d} different polynomials of degree d, because such polynomials can have only 2^d different terms. This means d cannot be less than n, which gives us a contradiction.

Problem 4. Let's first do the direction that every language in PH can be computed by an exponential-size, constant-depth uniform circuit.

We will proceed by induction. For the base case, we want a constant-depth circuit for any language in P. First, we want a layer of 2^n AND gates, one gate for each possible input. Each gate is activated only on a single input. Our second and final layer is a single OR gate. When we calculate the $EDGE$ function, we just calculate whether the ith input string is in our language or not, and if so, we add an edge to the AND gate that is activated precisely by that input string. This depth-2 circuit computes our language in P.

Note that these circuit families can be negated by switching every AND gate to an OR gate, and negating every input. Thus, if we can represent Σ_k^P languages by these circuits, we can represent Π_k^P languages by these circuits, and vice versa.

Now, consider a language $L \in \Sigma_k^P$. It can be written as

$$x \in L \iff \exists u : ux \in L' \tag{9}$$

where L' is in Π_{k-1}^P. We recursively find a circuit family that calculates L', and make 2^n copies of that circuit, one for every possible x. Each of these copies can be calculated in polynomial-time, so the resulting circuit is still DC uniform. We combine all of those with a single OR gate to create a circuit that calculates L. Induction then tells us that these circuits can compute every language in PH.

For the other direction, it's simpler to see that an alternating Turing machine can evaluate a DC uniform circuit with a constant number of alternations. When we hit an AND node, use the mode that accepts when every branch accepts, and have one branch go to each child of the AND node. When we hit an OR node, use the mode that accepts when any branch accepts, and have one branch go to each child of the OR node. Since each local graph area can be computed in polynomial time, and the graph has constant depth, the whole algorithm will run in polynomial time. Since depth is constant, we only need a constant number of alternations. So a DC uniform circuit can be evaluated by an alternating Turing machine in a constant number of alternations, and PH is powerful enough to compute these circuits. Thus the models are equivalent.

Problem 5. This is a three-part problem.

1. SUCCINCT-3SAT is $NEXP$-complete, so we just have to show that SUCCINCT-3SAT is in E^{NP}. This is straightforward - it takes exponential time to expand the SAT instance from its compressed form, and then we can solve it with a single call to the NP oracle.

2. $NE \subset NEXP$, so $NEXP \not\subseteq P/poly \implies NE \not\subseteq P/poly$.

3. We can construct the truth table for SUCCINCT-3SAT. There are 2^n different compressed 3SAT problems possible on size n. So, we can nondeterministically guess an answer to each problem, and put 0 in the truth table if it is not a correct answer, 1 if it is the correct answer. This takes poly(2^n) time for each problem, so poly(2^n) time overall.

   However, this is not enough. We still need to figure out which of the branches should be the accepting branch. To do this, our advice string can tell us how many 1's should be in the truth table. Note that our algorithm has one-sided error - it only writes a 0 where there should be a 1, not vice versa. So if we only accept when we have the correct number of 1's in the truth table, that is sufficient to limit to a single accepting branch.

Since SUCCINCT-3SAT is NEXP-complete, by our assumption there is no polynomial-size circuit family that can calculate it, and these truth tables satisfy the desired circuit complexity condition.