CS278: Computational Complexity
Homework 1

Jongmin Baek, 25306749

September 15, 2016

Question 1

(a) $UNBEATABLE - PROGRAM$ is in coNP. The certificate is the sequence of moves that makes M lose. The polynomial-time algorithm simply has to play Geography game on G, making the moves as the certificate dictates.

(b) $TWO - PATH$ is in NL. The certificate: $P_1 = (x_1, x_2, ..., x_n)$ and $P_2 = (y_1, y_2, ..., y_n)$. The polynomial verifier does the following: check that each x_i is equal to y_i, check that both paths are valid, i.e. there exists an edge from x_i to x_{i+1} and also from y_i to y_{i+1}, and then check that $x_1 = y_1 = s$ and $x_n = y_n = t$. This can be done in logarithmic space, because we need to be looking at at most two vertices at any given time.

Question 2

(a) $L_{majority}$ is in NP. It is easy to verify that $L_{majority}$ is in NP if it is: the certificate just has to encode at least three certificates on how x belongs in L_i. This certificate is clearly polynomial in length, because $L_i \in NP \forall i$. It clearly takes polynomial time to check that $L_{majority}$ is in NP, because we simply need to run the polynomial-time verifiers for each L_i.

(b) $L_{=3}$ is in P^{NP}. To check that x is in exactly three of the languages, we must be able to check that (1) x is not in some languages and (2) x is in some other languages. (1) makes $L_{=3}$ coNP-hard, and (2) makes it NP-hard. Since we don't know if NP = coNP, that $L_{=3}$ requires the power of both languages means it needs a language powerful than both of them.

Question 3

We wish to show that $coNP \not\subset NP \Rightarrow 3-SAT \notin P^{FACTORING}$. We demonstrate a proof by contraposition.

Assume that $3-SAT \in P^{FACTORING}$. Since $3-SAT$ is NP-complete, $NP \subset P^{FACTORING}$. If we can find a satisfying formula to an instance of $3-SAT$ in polynomial time with a factoring oracle, we must be able to say that a satisfying formula does not exist at all if that is the case the same time. So $\overline{3-SAT}$ must also be in $P^{FACTORING}$. Since $\overline{3-SAT}$ is coNP-complete, $coNP \subset P^{FACTORING}$.

Now consider some language $l \in P^{FACTORING}$. Assume that, to determine l, we need to factor some set of numbers, and then need to do some polynomial-time bookkeeping on those numbers. Then we can write out all those factors in a certificate u which can then be fed into a polynomial-time verifier to check that l is indeed in $P^{FACTORING}$. Note that this certificate is guaranteed to be polynomial length because an N-bit number has at most N factors. Whatever added power $P^{FACTORING}$ has over P has all been encoded in the certificate; $P^{FACTORING}$ did all the heavy lifting, and P just has to look at its certificate, read the nicely factored numbers, and do the polynomial-time bookkeeping to verify that l is indeed in $P^{FACTORING}$.

But then all languages in $P^{FACTORING}$ can be verified for membership in polynomial time. So $P^{FACTORING} \subset NP$. But $NP \subset P^{FACTORING}$. So $NP = P^{FACTORING}$. But $coNP \subset P^{FACTORING} = NP$. This contradicts our assumption that $coNP \not\subset NP$. This completes the proof. ■

CS278: Computational Complexity
Homework 2

Jongmin Baek, 25306749
Collaborators: Minho Park, Emily Sharff, Sherdil Niyaz

September 21, 2016

Question 1

(a) If there is a polynomial time algorithm A that can decide 3-SAT, there is also a polynomial time algorithm that can search for a satisfying assignment for 3-SAT. First, ask A if some 3-SAT formula $\phi(x_1, x_2, ..., x_n)$ is satisfiable. If so, fix the value of x_1 as 1. Then ask A if $\phi(1, x_2, ..., x_n)$ is satisfiable. If so, fix the value of x_2 as 1. If not, fix the value of x_1 as 0, and ask A if ϕ is satisfiable. And so on and so forth: keep fixing the value of some literal and asking A if the proposition with the fixed value is satisfiable, and if it isn't satisfiable flip the value of the aforementioned literal. This clearly takes polynomial time, in fact linear in n.

(b) If there is a BPP algorithm A that can decide 3-SAT, a similar argument as the one above holds. If A outputs the correct satisfiability decision for some $\phi(x_1, x_2, ..., x_n)$ with probability larger than $\frac{1}{2}$, by the amplification theorem A can output the correct satisfiability decision with arbitrarily high probability. So fix the value of some x_1 as 1, and ask A if $\phi(1, x_2, ..., x_n)$ is satisfiable, repeating as much as possible until we have the probability we want. So if there is a BPP algorithm A that can decide 3-SAT, there is also a BPP algorithm that can search for a satisfying assignment for 3-SAT.

Question 2

BPP is defined as the set of all languages l for which a machine M exists such that if $x \in l, M(x, y) = 1$ with probability $> 2/3$ and if $x \notin l, M(x, y) = 1$ with probability $< 1/3$. But what is really causing the "randomness" here? After all, the machine is a deterministic TM, and there's nothing random about its behavior. What is really random is the inputted random bits, y. That $M(x, y) = 1$ with probability $> 2/3$ is another way of saying that for more than two-thirds of random bits y out of all possible y's, $M(x, y) = 1$. Since we have an oracle that takes in a circuit C and give the number of satisfying assignments for C, we want to get a representation of M as a circuit and get B to spit out the number of y's for which $M(x, y) = 1$. To clarify,

- Let C_M be the circuit representation of M. C_M always exists, by proof of Theorem 6.6 in page 109 of *Computational Complexity*.
- For all $x \in l$, it is possible to hard-code x into C_M, so that C_M is satisfiable if and only if $M(x, y) = 1$.
- If $x \in l$ and there exists a machine M such that $Pr[M(x, y) = 1] > 2/3$, then $M(x, y) = 1$ for two-thirds of all possible y's. Since $B(C_M)$ gives the number of y's for which $M(x, y) = 1$, by verifying that this number is larger than $2^n \cdot 2/3$ (there are 2^n possible n-bit y's), we have simultaneously verified that $Pr[M(x, y) = 1] > 2/3$.
- Thus P^B can check whether x is in some BPP-language l, and so $BPP \in P^B$.

CS278: Computational Complexity
Homework 3

Jongmin Baek
Collaborators: Minho Park, Emily Sharff, Sherdil Niyaz

September 22, 2016

Question 1

Each clause in a 2-SAT formula can be expressed as a pair of implications: $\neg A \vee B \equiv A \Rightarrow B \equiv \neg B \Rightarrow \neg A$. One can draw a directed graph using these implications such that $A \Rightarrow B$ is encoded as two nodes A and B with an edge outgoing from A to B. Now consider the co-2-SAT problem: determine if the formula is unsatisfiable. This is easy with the graph, because all one has to do is find a path from A to $\neg A$, if it exists. So it is easy to see that $2 - SAT \in coNL$. Because $coNL = NL$, it also holds that $2 - SAT \in NL$.

Question 2

Construct a directed graph with nodes $w_0, w_1, w_2, ..., w_n$ and an edge from w_i to w_j for all $i < j$. So there is a path from w_0 to each of $w_1, w_2, ..., w_n$, there is a path from w_1 to each of $w_2, w_3, ..., w_n$, and so on. It is clear that each of the paths in this graph constitute a subset of $w_1, w_2, ..., w_n$ (w_0 is added in the graph for convenience; when looking at the subsets, ignore w_0, because w_0 is not part of the set of strings.) For each of the paths, walk down the path and write down the number of 0's, number of 1's, and number of 2's you encounter on the way. Consider the co-problem: there exists a string obtained by concatenating a subset such that the string is balanced. So a certificate for the co-problem is just a path in the graph such that the number of 0's, 1's, and 2's are identical. Since all the space we need is for the number of 0's, 1's, and 2's, the problem can clearly be solved in log-space, assuming our machine is nondeterministic and can walk down multiple paths at the same time. So the problem is clearly in coNL. But NL = coNL. So the problem is in NL.

CS278: Computational Complexity
Homework 4

Jongmin Baek
Collaborators: Minho Park, Emily Sharff, Sherdil Niyaz

September 29, 2016

Q1

If $NP = P$, then $P = PH$. As stated in p151 of *Computational Complexity*, $AM[2] \subseteq \Sigma_3^p$. Clearly, $P \subseteq AM[2]$; to solve a P problem in $AM[2]$, let the prover solve the problem and the verifier can verify it in polynomial time. Therefore, if $NP = P$, $AM[2] = P$.

Q2

Before we begin, note that if $y = \mathcal{E}^t(x)$, $t \leq 2^n$. I mean, t could be bigger, but the verifier knows *a priori* that if $y \in \{0,1\}^n$ is indeed connected to $x \in \{0,1\}^n$ it must be connected at at most 2^n steps. So the verifier may *demand* for a $t \leq 2^n$.

- Verifier asks prover for the midpoint of $\mathcal{E}^t(x)$, $\mathcal{E}^{t/2}(x)$.
- Prover provides an alleged midpoint, $\tilde{m_1}$.
- With probability $1/2$, verifier asks for the midpoint between $\tilde{m_1}$ and $E^t(x)$. With probability $1/2$, verifier asks for the midpoint between $\tilde{m_1}$ and x.
- Prover provides an alleged midpoint, $\tilde{m_2}$.
- With probability $1/2$, verifier chooses a midpoint to the left of $\tilde{m_2}$. With probability $1/2$, verifier chooses a midpoint to the right of $\tilde{m_2}$.
- Prover provides an alleged midpoint, $\tilde{m_3}$.
- And so on...
- Prover provides an alleged midpoint $\tilde{m_{\log(t)}}$. Observe: the verifier may now check if this midpoint is indeed accurate, or if it is not, by simply applying E to the alleged midpoint. The key is that if the prover lied at any point, and the verifier (with probability $1/2$) caught the lie, the prover

must continue to lie, and its lie is easily checkable by verifier at the very end.

To make the proof a little more lucid, let's assume the prover is lying, that there is no path from x to y, and the verifier, by pure chance, catches him lying each time. Of course, the chances of this happening is exponentially small, $1/2^n$ given n interactions. But non-zero soundness is all we need.

- Verifer asks for the midpoint.
- Prover gives $\tilde{m_1}$. Observe: $\tilde{m_1}$ cannnot be connected to both x and y. If it were, then there exists a path from x to y, which contradicts our assumption that the prover is lying. Wlog let's assume $\tilde{m_1}$ is connected to x but not y.
- Verifier asks, by pure chance, for midpoint between $\tilde{m_1}$ and x.
- Prover gives $\tilde{m_2}$. Observe: $\tilde{m_2}$ cannot be connected to both x and $\tilde{m_1}$, for the same reason as before. Wlog let's assume $\tilde{m_2}$ is connected to x but not $\tilde{m_1}$.
- Verifier asks, by pure chance, for midpoint between $\tilde{m_2}$ and x.
- Prover gives $\tilde{m_3}$...
- and so on...
- Prover gives $m_{\tilde{\log(t)}}$, where originally $y = \mathcal{E}^t(x)$. Observe: $m_{\tilde{\log(t)}}$ cannot be connected to both x and $m_{\tilde{\log(t)-1}}$. Wlog let's assume that it is connected to x but not $m_{\tilde{\log(t)-1}}$.
- Verifier chooses, by pure chance, to compute $\mathcal{E}(m_{\tilde{\log(t)-1}})$, and sees that it is not equal to $m_{\tilde{\log(t)}}$. Verifier rejects.

As stated above, verifier chooses the unconnected side with at least half probability each time, and there are at most n interactions. Therefore the verifier rejects with probability at least $\frac{1}{2^n}$. It is trivial to see that the verifier accepts with perfect probability if x is indeed connected to y: verifier cannot reject if the midpoint is indeed the midpoint, and its last computation of applying $\mathcal{E}$ doesn't reveal a lie.

CS278: Computational Complexity
Homework 6

Jongmin Jerome Baek
Collaborators: Minho Park, Sherdil Niyaz, Emily Sharff

October 12, 2016

(a) $PCP_{1,s}[O(\log n, 2)] \in P \ \forall s < 1$ because of the number of bits queried by the PCP Verifier, 2, corresponds to the arity of constraints in the corresponding CSP instance. So the PCP Verifier has a corresponding 2-CSP instance, and every 2-CSP instance can be reduced to a 2-SAT instance. So every problem in $PCP_{1,s}[O(\log n, 2)] \forall s < 1$ may be reduced to 2-SAT, which implies that it must be in P.

(b) Because $s < \frac{1}{2^q}$, there is a corresponding $qCSP$ instance such that if it is not satisfiable, no more than $\frac{1}{2^q}$ clauses may be satisfied. It is known that the greedy algorithm for $3SAT$ can be generalized to $qCSP$ such that for any satisfiable $qCSPw$ instance ϕ with m constraints, the algorithm will output an assignment satisfying $\frac{val(\phi)m}{w^q}$ clauses, where $w = 2$ in our case because our alphabet is $\{0, 1\}$. This implies the following. If it is satisfiable, $val(\phi) = 1$, and therefore $\frac{m}{2^q}$ clauses are satisfied. If it is not satisfiable, $val(\phi) < \frac{1}{2^q}$, so less than $\frac{m}{2^{2q}}$ clauses are satisfiable. Then our algorithm is simple:

```
ALG SATISFIABLE?(phi):
    create an assignment using greedy algorithm
    k = number of satisfied clauses with this assignment
    if k == m/(2^q):
        output YES
    else if k <= m/(2^(2q)):
        output NO
```

(c) Respectively: 0.9, 2, OR

CS278: Computational Complexity
Homework 7

Jongmin Jerome Baek
Collaborators: Emily Sharff, Sherdil Niyaz, Minho Park

October 20, 2016

Question 1

Every boolean function $f : \{0,1\}^n \to \{0,1\}$ admits an exponential sized DNF formula. Consider the boolean function expressed as a truth table where the input is a bitstring of length n and the output is either 0 or 1. This truth table can be converted into a DNF by encoding each row as a clause such that the clause is satisfied if and only if the bitstring corresponding to the input column of the row is a satisfying assignment to the clause. For example, if the input column bitstring is 010, the clause may look like $\not{x}_1 x_2 \not{x}_3$. Then, since a DNF is a disjunction over all these clauses, the DNF is satisfied if and only if at least one of the clauses is satisfied. Whether the DNF is satisfied or not can then be considered as the output of the boolean function f.

Question 2

The branching program is of width 3. This suggests that we can use each row as serving a special need. Using part 1, we can convert any boolean function f into a DNF. Consider the fact that a DNF proposition is a disjunction over conjunctions: that is, if just one of the clauses is satisfied, the entire proposition is satisfied. Hence we can have the first row (say) to be the "done" row: once you are in this row, the computation proceeds in this row regardless of what the assignment to the variable is – once you are in this row, the two edges outgoing from any node in this row go to the next-layer node in the same row. In this way we can encode the fact that if just one clause is satisfied, we don't care about the rest of the computation. Then, consider the fact that each clause is a conjunction. This is where we use the other two rows. The second row (say) can be a "work in progress" row: if you are in a clause and the literals have not all been satisfied yet, you stay on this row. The third row (say) can be a "clause unsatisfied" row: if you are in a clause and if just one literal isn't satisfied, you direct edges to the third row for the remainder of the clause until you hit the start of the next clause. So, three rows: "done", "work in progress", and "clause unsatisfied".

CS278: Computational Complexity
Homework 8

Jongmin Jerome Baek
Collaborators: Emily Sharff, Sherdil Niyaz, Minho Park

Thursday, October 27, 2016

The proposition *Every function f that has a logarithmic depth and polynomial size circuit also has a poly-sized formula* is false. Here is a counterexample. Consider a circuit for a function f such that the circuit has logarithmic depth and each level has $poly(n)$ nodes. For the ith level, draw an arrow from each node in that level to every other node in the $i+1$th level. A concern here is that this may not be the most efficient way to construct a circuit for a function f. Specifically, if each node is either an AND, OR, or NOT gate, there must be at least two nodes with identical labels in any level with more than three nodes. Moreover, these nodes take the exact same input: all of the nodes in the previous level. Therefore, it can be argued that only three nodes are needed in any given level if each node is connected to every other node of the next level. This concern can be avoided like so: for each node, in a level i, in addition to connecting it with every node in level $i+1$, also connect it with some arbitrary node in some aribtrary level $i+k$ where $k > 1$. Then it is no longer guaranteed that two nodes of identical labels compute the same output, for they may receive different inputs. For such a circuit, a source node has $O(n^{\log n})$ paths to get to any sink node. To construct a formula for the circuit, each of these paths must use distinct nodes and edges. Therefore, each source node needs to be duplicated $O(n^{\log n})$ times. Therefore, the size of the formula is not polynomial in n. This completes the proof. ■

CS278: Computational Complexity
Homework 10

Jongmin Jerome Baek
Collaborators: Sherdil Niyaz, Minho Park, Emily Sharff

November 10, 2016

(1) (a) $H(X, g(X)) = H(X)$ If X is known, there is no information gain in knowing $g(X)$ also, because one can always take X and compute $g(X)$.

(b) $H(X, g(X)) \geq H(g(X))$ Equality holds only if g is a bijection. Otherwise, it is more informative to know X than to know $g(X)$.

(c)

$$\begin{aligned} I(X : g(X)) &= H(g(X)) - H(g(X)|X) \\ H(g(X)|X) &= 0 \\ I(X : g(X)) &= H(g(X)) \end{aligned}$$

(2)

$$\begin{aligned} H(Z|XY) - H(Z|X) &\geq 0 \\ H(Z) - H(Z|XY) &\geq H(Z) - H(Z|X) \\ I(XY : Z) &\geq I(X : Z) \end{aligned}$$

(3) (a)

$$\begin{aligned} X = Y = Z \\ H(X|Y) = H(X|Z) = H(X|YZ) = 0 \\ H(X) > 0 \\ I(X : Y) > I(X : Y|Z) \\ H(X) - H(X|Y) > H(X|Z) - H(X|YZ) \end{aligned}$$

(b)

$$
\begin{aligned}
&X, Y \ independent \\
&Z = X \ XOR \ Y \\
&H(X) = H(X|Y) \\
&H(X|YZ) = 0 \\
&I(X:Y) < I(X:Y|Z) \\
&H(X) - H(X|Y) < H(X|Z) - H(X|YZ) \\
&0 < H(X|Z)
\end{aligned}
$$

CS278: Computational Complexity
Final Exam

Jongmin Jerome Baek

December 9, 2016

Question 1

For each $i \in \{\frac{n}{2}+1, n\}$, the tape-head of the machine M traverses between the i^{th} and $i+1^{st}$ positions on the work-tape at least $\Omega(n)$ times.

Proof. Let us break the tape at position i into two pieces. Give the left piece ($\leq i$) to Alice and give the right piece ($> i$) to Bob. The left piece looks like $x_1...x_{n/2}0^i$, and the right piece looks like $0^{n-i}x_{n/2}...x_1$. Now consider the function $f : \{0,1\}^* \times \{0,1\}^* \to \{0,1\}$,

$$f = \begin{cases} 1 & a_1...a_{n/2} = b_{|b|}...b_{|b|-n/2} \\ 0 & else \end{cases}$$

where $n = |a| + |b|$. Clearly, $EQ(a,b)$, the equality function, reduces to f; $EQ(a,b) = f(a,b')$ where $b' = b_{|b|}...b_1$.

Lemma. *The communication complexity of equality, $C(EQ)$, is $\Omega(n)$.*

Because EQ reduces to f, so $C(f) = \Omega(n)$. Proof by contradiction: suppose $C(f)$ is sublinear. Then f can be used to solve EQ with sublinear communication complexity. But this is absurd. Therefore, $C(f)$ is not sublinear.

Now consider the following: traversing the Turing machine tape from the left side ($\leq i$) to the right side ($> i$) can be seen as equivalent to Alice transmitting a bit to Bob, and traversing the Turing machine tape from the right side ($> i$) to the left side ($\leq i$) can be seen as equivalent to Bob transmitting a bit to Alice. So, the tape-head of M must traverse from the left side to the right side, and from the right side to the left side, $\Omega(n)$ times. So, for each i in $\{\frac{n}{2}+1, n\}$, the tape head traverses from the i^{th} to the $i+1^{th}$ positions at least $\Omega(n)$ times. □

Question 3

(a) Assume that Alice can use the cloud pattern in the sky to get a $1000n$-bit random string. How can Alice detect an anomaly with probability greater than 0.99 using at most $O(\log n)$ bits of memory?

Claim. *Alice can detect an anomaly, where a car went in but did not come out, or went out but did not come in, with probability greater than 0.99 using at most* $O(\log n)$ *bits of memory.*

It is natural to use a streaming algorithm for this problem. Specifically, we will compute the second frequency moments of the instream and the outstream separately, then check if they are equal. If they are equal, there is no anomaly; if they are different, there is an anomaly. (This method assumes that only one car is an anomaly.)

Lemma. *There exists a randomized streaming algorithm that* $(1+\epsilon)$*-approximates* $F_2 = \sum_{i=1}^{n} x_i^p$ *in space* $O(\log n)$*.*

Proof. A proof is presented in the lecture notes on streaming algorithms. [1] □

Leveraging our $1000n$ random bits, we can use the randomized algorithm to approximate F_2 with 0.99 accuracy. We approximate F_2's of the instream $(a_1, a_2, ..., a_n)$ and the outstream $(b_1, b_2, ..., b_n)$ separately and call them $F_2^I = \sum_{i=1}^{n} a_i^2$ and $F_2^O = \sum_{i=1}^{n} b_i^2$.

Claim. *There is an anomaly if and only if* $F_2^I \neq F_2^O$*.*

Proof. We give the proof in both directions. Forward direction: if there is an anomaly, then $F_2^I \neq F_2^O$. Because there is an anomaly, there is a car h that, without loss of generality, came in once more than it went out. Then, clearly $F_2^I > F_2^O$, because $a_h > b_h$. So $F_2^I \neq F_2^O$. Backward direction: if $F^I 2 \neq F_2^O$, then there is an anomaly. We give a proof by contraposition. Assume that there is no anomaly. Then all cars went in the same number of times they went out. That is, $a_i = b_i$ for all i. Therefore, $\sum_{i=1}^{n} a_i^2 = \sum_{i=1}^{n} b_i^2$. Therefore, $F_2^I = F_2^O$. □

(b) How can Alice perform this same task, without using the cloud pattern but with $O(\log^2 n)$ bits of memory?

Claim. *The streaming algorithm presented above can be derandomized using* $O(\log^2 n)$ *bits of memory.*

"Extractors can be used to obtain a pseudorandom generator for space-bounded randomized computation, which allows randomized logspace computations to be run with $O(\log^2 n)$ random bits." [2] By replacing the random bits with a pseudorandom generator that requires $O(\log^2 n)$ memory, we can derandomize the log-space computation of approximating F_2.

Remark. *Thank you for this great course, professor Raghavendra. :)*

Citations

[1]: The Space Complexity of Approximating the Frequency Moments. Alon, Matias & Szegedy. *Journal of Computer and System Sciences*, 1999.
[2]: *Computational Complexity, 21.6.* Arora & Barak. *Cambridge University Press*, 2009.