Assignment 4 Solutions

April 15, 2002

1. [9.5.4]

   (a) [(a)] Suppose we have a set $S = \{C_1, C_2, \ldots, C_n\}$ of clauses, with each clause being a set of literals. A truth assignment sets a clause to true if and only if at least one of the literals in the clause is true. Thus, an empty clause is unsatifiable. The following is a polynomial–time algorithm for SAT, provided that each variable appears at most twice.

   Our algorithm works as follows: we will systematically remove clauses from S. If we ever remove all the clauses from S, (and so S becomes empty) then S is satisfiable. If, at any time, S contains an empty clause, then S is not satisfiable. In order to remove clauses, we apply the following rules. Any time a clause is removed, we go back to the first rule (so the second rule will only be applied if the first rule failed to remove a clause, etc)

   - If no occurance of a variable x in S is negated, remove each clause which contains x.
   - If each occurance of a variable x in S is negated, remove each clause which contains x.
   - If both occurrances of a variable x occur in the same clause, remove that clause.
   - Each remaining variable x must occur positively in exactly one clause C and negated in exactly one clause D. Remove C and D, and add $(C \cup D) - \{x, \neg x\}$.

   To see that application of these rules preserve satifiability, note that:

   - Suppose that x does not occur in A, B, or S'. Then $(x \vee A) \wedge (x \vee B) \wedge S'$ is satisfiable if and only if S' is satisfiable, since any truth assignment for S' may safely set x to true.
   - Suppose that x does not occur in A, B, or S'. Then $(\neg x \vee A) \wedge (\neg x \vee B) \wedge S'$ is satisfiable if and only if S' is satisfiable, since any truth assignment for S' may safely set x to false.
   - $(x \vee \neg x \vee A) \wedge S'$ is satisfiable if and only if S' is satisfiable, since $(x \vee \neg x \vee A)$ is a tautology. (Note that by the time this rule is applied, the first two rules have assured that each variable x occurs exactly twice, once positively and once negated).

- Suppose that x does not occur in A, B, or S'. Then $(x \vee A) \wedge (\neg x \vee B) \wedge S'$ is satisfiable if and only if $(A \vee B) \wedge S'$ is satisfiable. This is because, in order for $(x \vee A) \wedge (\neg x \vee B) \wedge S'$ to be true, S' must be true, and if x is true, B must be true. Otherwise, if x is false, then A must be true.

Note that the last rule will create an empty clause if there are ever two clauses (x) and $(\neg x)$ in S, in which case the algorithm should terminate and answer "no". Otherwise S will eventaully become empty, in which case the algorithm should terminate and answer "yes".

This clearly runs in polynomial time, since each rule can be checked in quadratic time, and each step reduces the size of S by one.

(b) [(b)] First, repeatedly apply the 3rd rule from part (a), so that each clause contains exactly 3 variables (the requirements were that each clause contain exactly 3 literals, with no repetitions)

Next, construct a bipartite graph $G = (V, E)$ as follows: Let V_1 be the set of clauses in our input, and let V_2 be the set of variables. For a clause C and a variable x, $(C, x) \in E$ if and only if the variable x appears in C. Let B be some set of clauses. Each clause has (exactly) degree 3, since each clause contains exactly 3 variables. But, each variable has degree no more than 3 (since it can appear in no more than 3 clauses) so $g(B) \geq B$. Thus, by the result of 9.5.25, G has a matching. This matching is a satisfying truth assignment for S, since each clause is matched up with exactly one variable, and no variable is matched up with more than one clause. Thus, if a clause C is matched with a variable x, simply set x to true if x appears positively in C. Otherwise, if x appears negatively in C, set x to false. By doing this for each clause C, we have built a satisfying truth assignment for S.

2. [9.5.6] Clearly it is in **NP**, since it is a special case of SAT.

So show hardness, we will reduce from 3SAT. Suppose $S = C_1 \vee C_2 \vee \ldots C_n$ is an instance of 3SAT, that is, each clause contains at most three literals. The reduction will work as follows: for each clause C_i:

If C_i contains only one literal, it is already a Horn clause, so it can be left as is.

If C_i contains exactly two literals, is can also be left as is.

We must consider the case where C_i contains exactly three literals, that is, it is in the form $(x \vee y \vee z)$. So, we introduce two new variables, a_i and b_i. Then, we replace the clause $C_i = (x \vee y \vee z)$ with the three clauses $(x \vee a_i)$, $(y \vee b_i)$, and $(z \vee \neg a_i \vee \neg b_i)$.

Note that the resulting clauses either contain only two variables, or are Horn clauses. It only remains to be shown that $(x \vee y \vee z)$ is true if and only if, for some assignment of the variables a_i and b_i, the sentence $(x \vee a_i) \wedge (y \vee b_i) \wedge (z \vee \neg a_i \vee \neg b_i)$ is true. However, this is obvious. This completes the reduction.

3. [10.4.7] To compute $x + y \pmod z$ in time $O(l^2)$, first compute $x + y$ in time $O(l)$ by using the regular "grade school" addition algorithm. Then use the division algorithm to find the

remainder $\pmod z$ in time $O(l^2)$.

To compute $x \cdot y \pmod z$ in time $O(l^2)$, first compute $x \cdot y$ in time $O(l^2)$ by using the multiplication algorithm. Then use the division algorithm to find the remainder $\pmod z$ in time $O(l^2)$.

To compute $x^y \pmod z$ in time $O(l^3)$: First note that $y = 2^{i_1} + 2^{i_2} + \ldots + 2^{i_j}$ for a set of unique i's, each no more than l (these correspond to the "1" bits in the binary representation of y. So, $x^y = x^{2^{i_1}+\ldots+2^{i_j}} = x^{2^{i_1}} \cdot x^{2^{i_2}} \cdot \ldots \cdot x^{2^{i_j}}$.

Also, $x^{2^{i+1}} = \left(x^{2^i}\right)^2$. Thus, we can compute each $x^{2^i} \pmod z$ inductively, by squaring $x^{2^{i-1}} \pmod z$ and computing the remainder $\pmod z$. There are at most l squarings and divisions to compute, and each one requires $O(l^2)$ time. Thus we can compute all the $x^{2^i} \pmod z$ in $O(l^3)$ time. We can then multiply all the required x^{2^i}'s together $\pmod z$, with each multiplication (and subsequent division) requiring $O(l^2)$ time. Thus the total running time is $O(l^3)$, as desired.

4. [10.4.8] Suppose N is not prime. Suppose that x divides N, and x is not equal to 1 or N. So, for some y (also not equal to N or 1), $N = xy$. Clearly, one of x or y must be no larger than $\sqrt{N}$, since if both x and y are greater than $\sqrt{N}$, then $xy > N$.

   By the result of problem 10.4.7, we can compute the value of $xy \pmod N$ in time $O(\log^2 N)$, and so it follows that we can compute $x \pmod N$ in time $O(\log^2 N)$ (by simply setting $y = 1$). So, we can test for primality of N in time $O(\sqrt{N} \log^2 N)$ by checking all integers x between 2 and $\sqrt{N}$ to see if $x \pmod N$ is 0.

5. [10.4.19]

   (a) [(a)] The idea here is to build a circuit which acts as an adder. Given a set $\{a_1, a_2, \ldots, a_n\}$ whose sum is less than $2^n - 1$ of positive integers, we will build a circuit with inputs $\{I_1, I_2, \ldots, I_n\}$ and n outputs. The value of the n outputs will be the n–bit integer which represents $\sum_{I_k=\mathbf{true}} a_k$.

   (b) [(b)] Suppose that the circuit C computes the monotone function F. Then, start by using the circuit to compute $F(\mathbf{false}, \mathbf{false}, \ldots, \mathbf{false})$. If the result is $\mathbf{true}^n$ then we have found an input that outputs $\mathbf{true}^n$ as desired. Otherwise, change one of the inputs to **true** and evaluate the circuit again. If the output is the same as the previous output, then we have found two different inputs which give the same output. Otherwise, some of the outputs must have changed from **false** to **true**, and none of the outputs changed from **true** to **false** (due to the monotonicity of the function. Change another input from **false** to **true** and repeat this process. Since there are n inputs and n outputs, it is clear that after repeating this process at most n times, we will have either found an input which yields the output $\mathbf{true}^n$, or we will have found two inputs which yield the same output. Since we can evaluate the circuit in polynomial time, the whole algorithm takes polynomial time.

6. [11.5.4] Sketch:

- Notice that the terms in the determinant of an (symbolic) adjacency matrix partitions the graph into disjoint cycles
- Therefore, we need a disjoint cycle to represent a Hamiltonian path
- Take the original graph G and make n copies of it "layered" on top of one another
- Add edges x_i between levels i and $i+1$ for verticies v and w if they were connected within G
- Add a special point external to all these
- Argue that a disjoint (covering) cycle implies a Hamiltonian path in the original graph

7. [11.5.14] Suppose that L_1 and L_2 are languages in **RP**, accepted by **RP** machines M_1 and M_2 respectively. We will define RP machines $M_\cup$ and $M_\cap$ which accept $L_1 \cup L_2$ and $L_1 \cap L_2$ respectively.

   M_{cup} simply simulates M_1 and M_2, and accepts if and only if one of M_2 and M_2 accepted. Clearly, if the input x is not in either L_1 or L_2, then all of the paths through $M_\cup$ are rejecting. If, on the other hand, x is in L_1 or L_2 then at least 1/2 of the paths will be accepting.

   M_{cap} works similarly, except that it accepts if and only if both the simulation of M_1 and M_2 accept. Again, at least 1/4 of the paths will be accepting if x is in both L_1 and L_2, otherwise, none of the paths will be accepting.

   For **BPP**, our machines $M_\cup$ and $M_\cap$ will work in exactly the same way. Suppose that our original **BPP** machines M_1 and M_2 were correct at least 3/4 of the time for any input. Then both $M_\cup$ and $M_\cap$ are correct at least $1 - (1/4 + (3/4)(1/4)) = 9/16$ of the time for any input, which is sufficient for the language to be in **BPP**.

8. [11.5.15] To see that **PP** is closed under complementation, note that we can simply invert the accepting and non–accepting configurations.

   To see that **PP** is closed under symmetric difference, suppose that L_1 and L_2 are languages accepted by **PP** machines M_1 and M_2. A machine M can accept the symmetric difference of L_1 and L_2 by simply simulating both M_1 and M_2. M should accept if and only if *exactly* one of M_1 and M_2 accepts.

   To see that this is correct, note that a path through M will accept a particular input x if both the simulations of M_1 and M_2 give the correct result, or if both give the incorrect result. Suppose that $M_1(x)$ is gives the correct answer in $1/2 + \epsilon_1$ of its paths, and $M_2(x)$ is gives the correct answer in $1/2 + \epsilon_2$ of its paths. Note that both ϵ_1 and ϵ_2 must be positive. So, M will answer correctly when both M_1 and M_2 are correct, and when both M_1 and M_2 are incorrect. So the fraction of paths on which M will be correct is:

$$\begin{aligned}(1/2+\epsilon_1)(1/2+\epsilon_2)+(1/2-\epsilon_1)(1/2-\epsilon_2) &= 1/4+\epsilon_1/2+\epsilon_2/2+\epsilon_1\epsilon_2+1/4-\epsilon_1/2-\epsilon_2/2+\epsilon_1\epsilon_2 \\ &= 1/2+2\epsilon_1\epsilon_2 \\ &> 1/2\end{aligned}$$

   as desired.

9. [11.5.17] Since our machines are balanced and make a "coin flip" at each step, there are exactly $2^{f(n)}$ paths through a machine that runs in time $f(n)$. Thus, we can assume without loss of generality that $\epsilon = a/2^b$ for integers a and b.

   Suppose $L \in \mathbf{PP}$, with $\mathbf{PP}$ machine M. To see that $L \in \mathbf{PP}_\epsilon$, we can run b simulations of M, keeping track of which of the simulations of M accepted, and which rejected. At the end, we are left with a b–bit number N, where the ith bit of N is a "1" if and only if the ith simulation of M accepted. We will accept if and only if $2^{b+1} - 1 - N < a$. At least an $\epsilon = a/2^b$ fraction of the computations of an input x will accept if and only if at least $\frac{1}{2}$ of the paths in M accept. So $L \in \mathbf{PP}_\epsilon$, and so $\mathbf{PP} \subseteq \mathbf{PP}_\epsilon$.

   Now suppose $L \in \mathbf{PP}_\epsilon$, with $\mathbf{PP}_\epsilon$ machine M. To see that $L \in \mathbf{PP}$, we can construct the following $\mathbf{PP}$ machine M': The first step is to "flip a coin" to decide which "branch" of our machine to take. The left branch is simply a simulation of M. The right branch is a machine of the same depth as M, which ignores the input x and accepts in exaclty a $1 - \epsilon$ fraction of its computations. So, on input x, if M accepts on an ϵ' fraction of its computations, then the fraction of computations on which M accepts is $(1 - \epsilon + \epsilon')/2$. Clearly, this is greater than $\frac{1}{2}$ if and only if $\epsilon' > \epsilon$. Thus $L \in \mathbf{PP}$, as desired. So $\mathbf{PP}_\epsilon \subseteq \mathbf{PP}$. This completes the proof.