Harvard University
Computer Science 121

Problem Set 9

Due Tuesday, December 7, 2010 at 1:20 PM.
Submit a single PDF (lastname+ps9.pdf) of your solutions to cs121+ps9@seas.harvard.edu
Late problem sets may be turned in until NO LATE DAYS at 1:20 PM with a 20% penalty.
See syllabus for collaboration policy.

Name

Problem set by !!! Your Name Here !!!

with collaborators !!! Collaborators' names here !!!!

PROBLEM 1 (5 + 5 points)

(A) Let DOUBLE-SAT $= \{\langle\phi\rangle : \phi$ is a boolean formula with at least two satisfying assignments $\}$. Show that DOUBLE-SAT is NP-complete.

(B) Let k be a natural number, and let $\mathcal{C}$ be a finite collection of finite sets. Then $\mathcal{C}$ is said to be a k-HITTING SET if there is a set H of size at most k that intersects every member of $\mathcal{C}$. That is, a finite collection C of finite sets is a k-HITTING SET if there exists a set H of size at most k such that $S \cap H \neq \emptyset$ for every $S \in \mathcal{C}$. Prove deciding k-HITTING SET is NP-complete. [*Hint*: Reduce from VERTEX COVER.]

(A) First we prove DOUBLE-SAT is in NP and then show it's NP-complete by reducing from 3CNF. A witness for DOUBLE-SAT is the two satisfying assignments. These assignments are twice the length of the input, which is polynomial, and both can be checked in linear time, so DOUBLE-SAT is in NP.

To determine if a 3CNF formula has at least one satisfying assignment we need to double the number of satisfying assignments so an unsatisfiable formula remains unsatisfiable (it would also be OK to map these to formulas with only one satisfying assignment) but a formula with even only one satisfying assignment has two in its doubled version. We can do this by adding a new clause with a dummy variable—$(x_d \vee \bar{x_d} \vee x_d)$ to the original formula. If the original formula is unsatisfiable the new formula is, to—adding constraints does not allow unsatisfiable formula to become satisfiable—but every satisfying assignment in the original has now been "doubled" since assigning either true or false to the new dummy variable satisfies its clause, allowing us to keep the rest of the satisfying assignment the same. Thus we can decide 3CNF using DOUBLE-SAT, so DOUBLE-SAT is NP-complete. $\square$

(B) We prove k-HITTING SET is NP-complete by first proving it's in NP, then demonstrating how a k-Vertex cover problem can be reduced to k-Hitting Set. k-Hitting Set is in NP since a witness is a set of elements of size $\leq k$, which is at most one element per input set and thus polynomial in the size of the input, and checkable in at most quadratic time by intersecting each input set with the witness.

Now we reduce from k-Vertex cover. We take a vertex cover problem and for each edge in the graph create a set of vertices it's incident upon. These sets form the input. For a k cover we also check if there's a hitting set of size k, which by construction means there's at most k vertices s.t. each edge in the graph is incident at least one vertex. $\square$

PROBLEM 2 (5+5+2 points)

Consider the following two languages:

$$A = \{\langle D\rangle \ : \ D \text{ is a DFA and } D \text{ accepts at least one string}\}$$

$$B = \{\langle D_1, D_2, ..., D_k\rangle \ : \ \text{Each } D_i \text{ is a DFA and all of the } D_i \text{ accept at least one string in common}\}$$

(A) Show that $A \in$ P.

(B) Show that B is NP-hard by giving a reduction from 3-SAT to B.

(C) We can convert an instance of B into an instance of A by applying the product construction (See p.45-46 of *Sipser*) $k-1$ times in succession. Does this show that P = NP? Why or why not?

(A) We prove that we can determine if a DFA D accepts at least one string in polynomial time by demonstrating that if there exists a path from the start state to the accept state of a DFA it accepts a string. We know PATH is in P, so this completes the proof. We demonstrate a path from the start to an accept state causes the DFA to accept at least one string by construction. Consider a path from the start to the accept state of a DFA. The path must have transitions labelled with characters in the alphabet. Therefore the DFA must accept the string composed of the characters in order on the path from the start to the accept state. $\square$.

(B) We prove that B is NP-hard by reducing 3CNF to it. For each clause in our 3CNF formula we create a DFA which accepts a string of length equal to the number of variables in the 3CNF formula and at least one of the variables in its clause is true. This is done by creating one state per variable and for the salient variables if they satisfy a clause going to an accept state which loops back to itself on all future strings. If the end of the string is walked off we go to a special rejecting sink (if we hadn't hit an accepting state previously). In this way each DFA only accepts truth assignments which satisfy its clause. The only common string this set of DFAs will accept is a string satisfying every clause in the original formula by construction. Thus if they all accept one common string there was a satisfying assignment to the 3CNF formula. $\square$

(C) No, since the product construction is not necessarily a polynomial time operation, in some cases resulting in an exponential blowup.

PROBLEM 3 (5+2 points)

Recall that Co-NP $= \{L : \overline{L} \in \text{NP}\}$. It is unknown whether or not NP = Co-NP. Note that NP = Co-NP if and only if NP is closed under complement.

(A) Prove that if NP $\neq$ Co-NP, then P $\neq$ NP. (Aside: Nonetheless, we can't rule out the possibility that NP = Co-NP and yet P $\neq$ NP.)

(B) To prove that NP = Co-NP, it would suffice to show that for every $L \in$ NP, $\overline{L} \in$ NP. Suppose $L \in$ NP. Then there exists a nondeterministic Turing machine M that decides L in polynomial time. Consider the new Turing machine M', which is identical to M except that its accept and reject states are reversed.

What language does M' decide? Explain *briefly*.

(A) Proof by contradiction. Assume P = NP and NP $\neq$ co-NP. Since P is closed under complement and co-NP is the complement of NP by definition then P = NP = co-NP. A contradiction. $\square$

(B) M accepted every string for which there existed an accepting computation, M' accepts every string for which there existed a rejecting computation in M. Thus M and M' may accept the same set of strings, since only strings always accepted by M are rejected by M'.

PROBLEM 4 (5+5+10 points)

A *search problem* is a mapping $S : \Sigma^* \to P(\Delta^*)$ from strings ("instances") to sets of strings ("valid solutions"). An algorithm M *solves* a search problem S if for every input $x \in \Sigma^*$ such that $S(x) \neq \emptyset$, M outputs some solution in $S(x)$.

An NP *search problem* is one for which there exists a polynomial p and a polynomial-time algorithm V such that for every x and y

1. $y \in S(x) \Rightarrow |y| \leq p(|x|)$, and
2. $y \in S(x) \Leftrightarrow V$ accepts $\langle x, y \rangle$.

Informally, these conditions say that valid solutions are short and can be verified efficiently.

(A) Show that the SAT-SEARCH problem "given a satisfiable boolean formula φ, find a satisfying assignment" is an NP search problem

(B) Prove that SAT $\in$ P if and only if the SAT-SEARCH problem can be solved in polynomial time.

(C) Prove that P = NP if and only if every NP search problem can be solved in polynomial time.

(A) We prove finding a satisfying assignment to a satisfiable boolean formula is a SAT-search problem by verifying the two criteria above are satisfied. First, we know that boolean formulas have witnesses linear in their representation—the satisfying assignment itself—so the first condition holds. Second, we can verify an assignment in linear time, too, just as we've done for boolean formulas previously, so condition two holds. $\square$

(B) We first prove that if SAT $\in$ P we can solve SAT-SEARCH in polynomial time; then we prove if we can solve SAT-SEARCH in polynomial time we can solve SAT in polynomial time.

If SAT is solvable in polynomial time we can find solutions for it as follows—

1. Test if there exists a valid assignment, if not reject
2. If there are no unassigned literals, accept
3. Set the first unassigned literal to True
4. Test if there exists a valid assignment, if not set the first literal to False
5. Repeat

This algorithm finds an assignment with only a linear increase in the polynomial time to solve SAT.

The ability to find a satisfying assignment in polynomial time lets us decide SAT in polynomial time, too. First we find and assignment, then use it as the witness to our regular polynomial time verifier. Because P is closd under composition we know the resulting algorithm is in P, and verifies the witness per our previous construction. $\square$

(C) First we prove if P = NP then every NP-search problem can be solved in polynomial time, then we show if every NP search problem can be solved in polynomial time P = NP.

If P = NP then SAT is solvable in polynomial time. Further, any problem in NP is reducible to SAT in polynomial time. A witness, then, is a reduction to SAT plus a satisfying assignment to that SAT formula. This can be verified in polynomial time by a polynomial verifier performing the same reduction and then checking the assignment. The witness's length is only polynomial since the reduction running in only polynomial time can only touch a polynomial number of bits, and the satisfying assignment to SAT is linear in the size of its variables.

If every NP-search problem can be solved in polynomial time then we can solve any NP problem in polynomial time, too. From (B) we know solving SAT-Search in polynomial time allows us to solve SAT in polynomial time. Since SAT is NP-hard we can solve any problem in NP in polynomial time, too.

$\square$

PROBLEM 5 (BONUS +1 points)

A language is in $\mathrm{NST}(S, T)$ if it is accepted by a nondeterministic Turing Machine that runs simultaneously in space $S(n)$ and time $T(n)$. Show that $\mathrm{NST}(S(n), T(n)) \subseteq \mathrm{DSPACE}(S(n) \log T(n))$. Then show that actually $\mathrm{NST}(S(n), T(n)) \subseteq \mathrm{DSPACE}(S(n) \log(T(n)/S(n)))$ (an improvement, for example, if $T(n) = S(n) \log S(n)$).

The bonus was not attempted.