CSE 431
Introduction to Theory of Computation
Homework #6
Due: Friday, May 21, 2010

W. L. Ruzzo 15 May 2010

Homework Assignment:

1. In lecture I showed a reduction from 3SAT to Vertex Cover. Apply it to the Boolean formula:

   $$(x_1 \vee x_2 \vee x_4) \wedge (x_1 \vee \neg x_2 \vee x_3) \wedge (x_2 \vee \neg x_3 \vee \neg x_4)$$

   Note that my reduction is *different* from the one given in Theorem 7.44 in the text.

   Show us the resulting graph and integer "k." Although the graph is technically just an ordinary undirected graph, with no vertex- or edge labels of any kind, in the reduction there is a natural correspondence between literals/clauses in the formula and vertices/vertex groups in the graph; show this somehow on your graph drawing. (Also, please try to draw your graph neatly and clearly; e.g., it's a planar graph, I think. It's hard to grade a hairball!)

   The formula has many satisfying assignments, and the resulting graph has many vertex covers. The correctness proof for the reduction defines a correspondence between the two. In particular, given a satisfying assignment, it explains how to choose one (or more) vertex covers. For your graph, what cover corresponds to the assignment $x_1 = x_2 = x_3 = F, x_4 = T$? (There should only be one.) Given that vertex cover, what assignments correspond to it? (There should be two.) Are they both satisfying assignments?
2. 7.17
3. Let $A_{\mathrm{TBNTM}} = \{\langle M, w, \$^t\rangle \mid M$ is a nondeterministic TM accepting w in t steps on at least one computation path $\}$. Prove that A_{TBNTM} is NP-complete. (A_{TBNTM} is "the acceptance problem for time-bounded nondeterministic TMs," an analog of A_{TM}, which we showed to be complete within the class of recognizable languages.)
4. 7.21. It may (or may not) be helpful to note that SAT, as defined in the text, allows arbitrary Boolean formulas over $\wedge, \vee, \neg$, not just CNF formulas.
5. Most of the NP-completeness theory we've covered deals with language recognition problems, like SAT: "Does Boolean formula w have a satisfying assignment?" In contrast, for most practical purposes, we'd really like to have an algorithm that *generates* such a truth assignment, if one exists. So isn't the theory, swell though it is, really addressing the wrong problem? In this exercise you will show that this is *not* the case.
   (a) Show that generating a solution is at least as hard as checking whether a solution exists. In particular, let *satassign*(w) be a function giving some (simple encoding of a) satisfying assignment for w, if one exists, or giving the empty string if either w isn't a syntactically valid formula, or isn't satisfiable. Show that if *satassign* is computable by a deterministic polynomial time Turing machine, then $P = NP$.
   (b) That was too easy. Let *satassign*$'(w)$ be as above, but let it output anything it wants, in particular something that looks like a truth assignment, in case either w isn't a syntactically valid formula, or isn't satisfiable. Show that, unless $P = NP$, *satassign*$'$ still isn't computable by a deterministic polynomial time Turing machine.

(c) Now for the converse — show that if you had a polynomial time deterministic Turing machine M_{SAT} solving the *language recognition problem* SAT, then you could build a polynomial time deterministic TM computing the function *satassign*. Hint: Use M_{SAT} as a subroutine to tell whether $x_1 = 0$ or $x_1 = 1$ is part of a satisfying assignment. [This property of SAT is known as *self-reducibility*. Most, but perhaps not all, NP-complete problems behave similarly.]

6. [Extra credit:] 7.33. Note: be careful about formula length. E.g., if x is a subformula you have, then $x * x$ is twice as long; this is perhaps OK, but if it is part of a construction being applied recursively, then you might end up with a *much* longer result.