Solution to Problem Set 4

April 18, 2013

1. (a) Let \( p(n) \) be the upper bound on how long the witness finder takes for strings \( x \in L \) for a witness finder. Then we can decide \( L \) by running the witness finder on the input for \( p(n) \) steps and seeing if it has outputed a valid witness by then.

(b) If we assume \( P = NP \), then we can decide \( SAT \) (or any other NP-complete problem) in polynomial time. If we had a witness finder for \( SAT \), then we could use it to find a witness finder for any other problem in NP.

Given a decider for \( SAT \), a witness finder for \( SAT \) can be built as follows: If \( \phi \) is a \( SAT \) instance, select without loss of generality the first literal that appears in \( SAT \) and create two new formulas, \( \phi' \) and \( \phi'' \), that set that literal to \( true \) and \( false \), respectively. If \( \phi \) was an accepting instance of \( SAT \), then at least one of \( \phi' \) and \( \phi'' \) will be as well; this process can be repeated until we have an assignment for each variable. Since we only run two operations per literal, this process runs in polynomial (indeed linear) time.

2. To show that \( EXACT4SAT \) is NP-complete, we have to show that it is in NP and that it is NP-hard.

The first part is easier — to show that \( EXACT4SAT \in NP \), we just need to give a polynomial-time witness for an accepting instance; a witness can just be a valid assignment of variables.

To show that \( EXACT4SAT \) is NP-hard, we can reduce from \( 3SAT \). Given an instance \( \phi \) of \( 3SAT \), we can convert to an instance \( \phi' \) of \( EXACT4SAT \) as follows:

(a) Define \( f_1, f_2, f_3 \ldots \) as dummy literals that do not appear in \( \phi \) which will always have the value \( false \);

(b) For each clause of \( \phi \), if there are 3 unique literals, just add \( f_1 \) to the clause. If there are repeated literals, then replace each literal that already has another copy with \( f_2, f_3 \), etc. until each literal appears once. If we replaced \( k \) literals this way, add \( f_{k+1} \) to the clause.

The resulting formula \( \phi' \equiv \phi \), because neither adding a false literal to a conjunctive clause nor replacing repeated literals with false literals will change whether the clause has a valid assignment. Each clause will have exactly 4 unique literals, so \( \phi' \) will be an accepting instance of \( EXACT4SAT \) if and only if \( \phi \) was an accepting instance of \( 3SAT \).
3. To show that DOUBLESAT is NP-complete, we have to show that it is in NP and that it is NP-hard.

The first part is easier — to show that DOUBLESAT ∈ NP, we just need to give a polynomial-time witness for an accepting instance; a witness can just be the two valid assignments of variables.

To show that DOUBLESAT is NP-hard, we can give a reduction from CIRCUITSAT. Let φ be an instance of CIRCUITSAT and x be a literal not in φ. Then, let φ′ = φ ∧ (x ∨ ¬x).

If φ has a valid assignment, then φ′ will have two valid assignments — one where x is true and one where x is false. On the other hand, if φ does not have a valid assignment, then φ′ will have no valid assignment either (since regardless of what x is, the other side of the AND gate will be false).

4. (a) This just follows from the definitions of “cycle” and “path”. If we start from some vertex in the cycle, then we could just not include the edge that comes into it and form a path.

(Formally, let v₁, v₂, v₃, ..., vₙ, v₁ be the vertices involved in the Hamiltonian cycle; then v₁, v₂, ..., vₙ are the vertices of a Hamiltonian path.)

(b) Here is an example of such a graph:

(c) Given an instance G of HAMCYCLE, pick an arbitrary edge that connects vertices u and v and create two new vertices, u' and v' that are connected only to u and v, respectively. The resulting graph will have a Hamiltonian path if and only if the original graph had a Hamiltonian cycle (the path would necessarily have to begin and end with u' and v', so removing them leaves a cycle). We can perform this operation for each edge, using the oracle each time to determine whether the new graph has a Hamiltonian path. If any of them do, the original graph had a cycle, and if none of them do, the original graph did not.

(d) Given an instance G of HAMPATH, add a vertex v and create an edge from each vertex in G to v. Call the resulting graph G'.
If there exists a path in $G$ from some vertex $s$ to some other vertex $t$ (without loss of generality), then that path trivially corresponds to a cycle passing through $v$. If $G'$ has a cycle, it must pass through $v$; the corresponding path in $G$ is the same cycle with $v$ removed.

5. (a) As suggested in the hint, we reduce 3COLORING to QUADPROG. For each assignment $x_1, \ldots, x_n$, we will treat its sign as the “color” — whether it’s negative, positive, or zero. Our constraint will be that no two adjacent vertices have the same sign. This is a valid instance of QUADPROG, and since 3COLORING is NP-complete, this is sufficient to show that QUADPROG is NP-hard.

For every edge $e_{ab}$ connecting vertices $x_{ab}$, we have to add constraints to ensure that their signs are different. We can ensure that they are neither both positive nor both negative by adding the constraint $x_a x_b \leq 0$, and we can ensure that they are not both 0 by adding the constraint $x_a^2 + x_b h^2 > 0$. Since we are only creating two edges for each vertex, this procedure is polynomial, and the assignment is valid if and only if the vertices are 3-colorable.

(b) If QUADPROG were in NP, it would need a witness. The “standard” witness would be the assignment to the variables. However, the assignments may need precision that is exponentially larger than the number of variables (because the values may be arbitrary irrational numbers), which would make it take more than polynomial time to verify.

6. Problem X would has to have time complexity $\Omega\left(\epsilon^{n^{\frac{1}{3}}}\right)$; otherwise, SAT could be solved in $2^{o(n)}$, which contradicts our assumption.

7. First we will describe a method for creating a graph which represents instances of 2-Sat, then we will give a polynomial time algorithm that solves 2-Sat using this graph.

Let $g$ be the function which creates the graph, and $I$ an arbitrary instance of 2-Sat having $n$ literals, then

$$g(I) = G = (V, E) = \{(x_1, \ldots, x_n, \neg x_1, \ldots, \neg x_n), \{(x_i, x_j) | \text{ either } (\neg x_i \lor x_j) \text{ or } (\neg x_j \lor x_i) \text{ is a clause in } I\}$$

The edges in the graph are thus the implications formed by each clause.

**Lemma 1.** I is satisfiable if and only if there is no $x_i \in V$ such that there is a path from $x_i$ to $\neg x_i$ and from $\neg x_i$ to $x_i$ in $g(I)$.

Proof. If $I$ is unsatisfiable then for any truth assignment there is some clause for which the truth assignment is $(F \lor F)$, or in our formulation, $(T \Rightarrow F)$. Note also by the symmetry that if we have an edge $(\alpha, \beta)$ then we have also $(\neg \beta, \neg \alpha)$.

First we show that if there is a path $x \rightsquigarrow \neg x$ and a path $\neg x \rightsquigarrow x$ in $g(I)$ then $I$ is unsatisfiable. If $x$ is true, then by the transitivity of $\Rightarrow$ if there is a path $x \Rightarrow \ldots \Rightarrow \neg x$ then there must be some literal on that path that is false, which means $T \Rightarrow F$ must have appeared in $I$; if $x$ is false
then we have the same occurrence in $\neg x \Rightarrow \ldots \Rightarrow x$. Since there is at least one unsatisfiable clause, $I$ is unsatisfiable.

Now suppose $g(I)$ contains no such paths, we will show that $I$ has a satisfying truth assignment. Begin with an arbitrary vertex $\alpha$ such that there is no path $\alpha \rightsquigarrow \neg \alpha$. Give all $\beta$ such that $\beta$ is reachable from $\alpha$ the assignment true, and give $\neg \beta$ the assignment false. This is always legitimate: if there was a path $\alpha \rightsquigarrow \beta$ and $\alpha \rightsquigarrow \neg \beta$ then by symmetry there would be a path $\beta \rightsquigarrow \neg \alpha$ which contradicts our assumption. Therefore we can create a satisfying assignment for every clause which contains a literal reachable from $\alpha$. Repeating this process for every unassigned literal in $I$ creates a satisfying assignment.

Now we can show that 2-Sat is in $P$. First we use DFS to find the strongly-connected components of $g(I)$, then we check if any literal appears in the same strongly-connected component as its negation. A strongly-connected component is a set of vertices such that for any two vertices in the set there is a path between them. Therefore the correctness of the algorithm follows directly from the above lemma.

The solution is from [1].

8. Construct a boolean array $A$ of length $k + 1$ whose cells are labeled 0, 1, 2, $\ldots$, $k$. In this array we will keep track of which sums we are able to form by adding up a subset of the elements $x_1, x_2, \ldots, x_n$. Initially, set $A[0]$ to true and $A[i]$ to false for every $i = 1, 2, \ldots, k$. We then process the elements $x_1, x_2, \ldots, x_n$ one by one. For each element $x_i$, we go over all the cells of the array, and whenever we see a "true" cell (say at position $y$) we go to position $x_i + y$ and write true. In other words, if we already know that it is possible to form the sum $y$ using a subset of the elements $x_1, x_2, \ldots, x_{i-1}$, we can conclude that it is possible to form the sum $y + x_i$ using a subset of the elements $x_1, x_2, \ldots, x_i$.

Once we finish this process for all $x_1, x_2, \ldots, x_n$, we simply check if the cell at position $k$ contains "true". If so, then there is a subset that sums to $k$, and we output true. If not, we output false.

This algorithm accesses the array $O(k)$ times per element, and also performs $O(k)$ additions per element. Since the numbers $x_1, x_2, \ldots, x_n$ can be assumed to be at most $k$, each addition takes $O(k)$ time, so the algorithm uses $O(k^2)$ operations per element. There are $n$ elements, so the total running time is $O(nk^2)$. Since the size of the input is at least $n$ and at least $k$, this running time is polynomial in the input size.

References