

Homework 3 Solutions

1. Shorter answer [20 pts, 5 pts each]

- (a) Suppose we know that a problem X is NP-complete. Suppose we discover a polynomial time algorithm for X . Would that imply that the SATISFIABILITY problem can be solved in polynomial time? Explain your answer.

Yes. To be NP complete means there is a poly-time transformation from any problem in the class NP, so that an instance of that problem can be transformed into an instance of problem X . Thus, if X is NP-complete, there is a way to transform an instance of SAT into an instance of X , where the transform takes poly-time. if X also takes poly time, this gives a poly-time solution for SAT.

- (b) Suppose we know that a problem X belongs to the class NP. Suppose we discover a polynomial time algorithm for X . Would that imply that the SATISFIABILITY problem can be solved in polynomial time? Explain your answer.

No. Many problems are in the class NP, such as “sorting” (you can check the certificate — an example sorted order — easily. However, it isn’t known how to transform an NP-hard problem into sorting, which is why sorting isn’t “NP-Complete”.

- (c) Suppose we discover an $O(n^3)$ algorithm for SATISFIABILITY. Would that imply that every problem in NP can be solve in $O(n^3)$ time? Why or why not?

No. There is some poly-time transformation, but it may increase the size of the problem, and the transformation itself may take a long time. For instance, if the problem size for the transform grows (an instance to some problem of size n may be transformed into a SAT instance of size n^2 , then the total run time would be $O(n^6)$.)

- (d) Given an integer $N > 1$ the Factoring problem asks for two integers $p, q > 1$ so that $N = pq$. Consider the following algorithm for Factoring:

```

For all integers i: 2 < i < sqrt{N}
  if i divides N, return i and N=i.
Return prime.

```

Is this a polynomial time algorithm for Factoring? Why or why not?

This is not. the complexity of an algorithm is based on the size of its input, which in this case is the number of bits required to encode the number NN , or $\log(N)$. The run time is \sqrt{N} . Let a be the size of the input, so $a = \log(n)$. The run time, as a function of a is $\sqrt{2^a} = 2^{\frac{a}{2}}$, which is not polynomial.

2. The following algorithm returns a satisfying assignment A for a formula ϕ over n variables $x_1 \dots x_n$. First, check whether ϕ is satisfiable; if not, fail. Otherwise, set $\phi_0 = \phi$, and for each i from 1 to n proceed as follows. Set $\phi' = \phi_{i-1} \wedge x_i$, and check whether ϕ' is satisfiable. If so, set $\phi_i = \phi'$ and add $x_i = \text{true}$ to A . Otherwise, set $\phi_i \rightarrow \phi_{i-1} \wedge \neg x_i$, and add $x_i = \text{false}$ to A . After all x_i have been processed, return the final assignment.

We claim that A is a satisfying assignment for ϕ . To show this, we first argue inductively that, after any number of variables have been assigned, the partial assignment A is *consistent* with some satisfying assignment to ϕ_i ; that is, there exists a satisfying assignment B that does not contradict the truth values in A .

Bas: Before any variables are assigned, A is empty and so does not conflict with any satisfying assignment to $\phi_0 = \phi$.

Ind: We know that there exists some satisfying assignment to ϕ_{i-1} consistent with A . If ϕ' is satisfiable, then there is a satisfying assignment B for ϕ' that sets $x_i = \text{true}$, since B must satisfy the subformula x_i of ϕ' . In this case, we set $x_i = \text{true}$ in A , which leaves it consistent with B . Otherwise, any satisfying assignment B' for ϕ_{i-1} consistent with A must set $x_i = \text{false}$; hence, B' satisfies $\phi_{i-1} \wedge \neg x_i$. In this case, we set $x_i = \text{false}$ in A , which is consistent with B' .

To conclude, we have that, after all x_i have been processed, A is consistent with a satisfying assignment to ϕ_n . Note that the final A assigns every variable of ϕ_n , so it is a complete truth assignment; moreover, A must satisfy every conjunct of ϕ_n , one of which is the original formula ϕ . Hence, A is a satisfying assignment to ϕ .

The above algorithm makes $O(n)$ calls to the SAT oracle and otherwise does $O(n)$ work.

Note: there is more than one way to constrain a formula to make certain variables true or false while leaving it a valid Boolean formula. It would also be OK to substitute “true” or “false” for a variable in ϕ , then partially evaluate ϕ until we are left with a new, smaller formula or a truth value (in the latter case, the remaining assignment does not matter). It would also be OK to make a variable true by replacing it everywhere with a tautology ($q \vee \neg q$).

3. We first show that IP is in NP. For any instance X of IP with n variables and m constraints, a certificate is the assignment of values to each variable, which has size proportional to n times the size of the numbers in the integer program. (This assumes that there exists a feasible solution in which the numbers don't get too big, but I won't worry about that here – if you like, assume the values are all 0 or 1). To verify this certificate, we must check that every constraint is satisfied, which is doable in $O(mn)$ operations using simple arithmetic.

We now argue that $3\text{-SAT} \leq_p \text{IP}$. Let ϕ be a 3-CNF formula over a set X of variables, whose j th clause C_j is a disjunction of literals $\ell_1^j \vee \ell_2^j \vee \ell_3^j$. We construct the following integer program $P(\phi)$.

For a literal ℓ , let the function $f(\ell)$ be x_i , if $\ell = x_i$, or $1 - x_i$, if $\ell = \neg x_i$. For each clause C_j , we construct the constraint

$$f(\ell_1^j) + f(\ell_2^j) + f(\ell_3^j) \geq 1.$$

For each variable $x_i \in X$, we add the additional constraints

$$0 \leq x_i \leq 1.$$

The total number of constraints is linear in $|X|$ plus the number of clauses, so the construction is polynomial in $|\phi|$.

Claim 1: if ϕ is satisfiable, then there is a feasible solution to $P(\phi)$.

Pf: let A be a satisfying assignment to ϕ . Set $x_i = 1$ if $A(x_i)$ is true, or 0 otherwise. Clearly, each x_i is between 0 and 1. Moreover, if A is satisfying, then at least one of the three literals

in each clause C_j is true. Hence, at least one of $f(\ell_k^j)$, $1 \leq k \leq 3$, must be 1. Conclude that all constraints are satisfied.

Claim 2: if there is a feasible solution to $P(\phi)$, then ϕ is satisfiable.

Pf: let V be a feasible assignment of values to the variables of P . Define a truth assignment A as follows: if $V(x_i) = 1$, set $A(x_i)$ true; otherwise, set it false. Clearly, each variable in A receives a truth value. Moreover, at least one of $f(\ell_k^j)$, $1 \leq k \leq 3$, must be 1, so the corresponding literal must be true in A . Hence, A satisfies each clause C_j , and so A satisfies ϕ .

4. The canonical decision problem for this 0-1 max flow problem is: “Given a directed, weighted graph G with source and target vertices s and t , does there exist a flow for G such that each edge is either filled to capacity or empty, and the total flow is *at least* k ?” Call this decision problem FLOW-01.

We claim that FLOW-01 is NP-complete. We first show that it is in NP. For any true instance (G, s, t, k) , a certificate is the list of edges with nonzero flow, which has size $O(|G|)$. We can check the following properties:

- For every vertex v except s and t , the total flow into v equals the total flow out.
- The total flow out of s is at least k .

if $G = (V, E)$, these checks can be done in time $O(|V||E|)$.

We now show that $\text{SUBSET-SUM} \leq_p \text{FLOW-01}$. let (X, t) be an instance of subset sum. We define G as follows. G contains three designated vertices a, b , and c , along with a vertex v_i for every $x_i \in X$. For each $1 \leq i \leq n$, there exist edges $a \rightarrow v_i$ and $v_i \rightarrow b$ with capacity equal to x_i . Finally, there exists an edge $b \rightarrow c$ with capacity t . The FLOW-01 instance is then (G, a, c, t) .

Claim 1: if X has a subset that sums to t , then G has a valid flow of at least t .

Pf: Let X' be the subset that sums to t . For each $x_i \in X'$, push x_i units of flow along the edges $a \rightarrow v_i$ and $v_i \rightarrow b$. The total flow into b is then exactly t , so push t units of flow along the edge $b \rightarrow c$. The result is a valid flow that pushes t units into c .

Claim 2: if G has a valid flow of at least t , then X has a subset that sums to t .

Pf: every valid flow on G is *at most* t , since no more than t units can flow into c . Consider a flow ϕ of at least t on G ; this flow must be *exactly* t . Because no flow is gained or lost at b , there must be a collection of edges $v_i \rightarrow b$ with nonzero flow, such that their total flow is t . Let X' be the set of all x_i such that v_i is nonzero in ϕ . By construction of G , X' sums to t .

5. The Geryon problem is a classic computer science problem. It is more commonly known as the “Scorpion problem”, and a very nice solution is available:

<http://www.cs.cornell.edu/courses/cs681/2004fa/Handouts/scorpion.pdf>