

Solutions for cs154 Homework #6

To show that a problem is NP-Complete, you need to show four things: (a) A non-deterministic polynomial-time algorithm that solves the problem, (b) A reduction from a known NP-Complete problem, (c) Proof that the reduction works in polynomial time, and (d) Proof that the original problem has a solution if and only if the new problem has a solution.

Easy Problems (2 Points each)

Part (A) The *Set Partition Problem* takes as input a set S of numbers. The question is whether the numbers can be partitioned into two sets A and $\bar{A} = S - A$ such that $\sum_{x \in A} x = \sum_{x \in \bar{A}} x$. Show that Set Partition is NP-Complete. Hint: Reduce SUBSET-SUM.

Set Partition \in NP: Guess the two partition and verify that the two have equal sums.

Reduction: The SUBSET-SUM problem is: Given a set X of integers and a target number t , find a subset $Y \subseteq X$ such that its members add up to exactly t . Let s be the sum of members of X . Feed $X' = X \cup \{s - 2t\}$ into *Set Partition*. Accept iff *Set Partition* accepts. The reduction clearly works in polynomial time.

Proof: We will prove that SUBSET-SUM has a solution iff Set Partition has a solution on the reduced problem. Note that the sum of members of X' is $2s - 2t$.

\Rightarrow If there exists a set of numbers in X that sum to t , then there exists a partition of X' into two such that each partition sums to $s - t$.

\Leftarrow Let's say that there exists a pair of partitions such that the sum of both partitions is $s - t$ each. One of these partitions contains the number $s - 2t$. Removing this number, we get a set of numbers whose sum is t .

Part (B) Show that the *subgraph-isomorphism problem* is NP-complete: Given two graphs G_1 and G_2 , does G_1 contain a copy of G_2 as a subgraph? That is, can we find a subset of the nodes of G_1 that, together with the edges among them in G_1 , forms an exact copy of G_2 when we choose the correspondence between nodes of G_2 and nodes of the subgraph of G_1 properly?

Hint: Reduce CLIQUE.

Subgraph-isomorphism \in NP: Simply guess the mapping of vertices of G_2 to a subset of G_1 and verify that the edges in both are identical.

The CLIQUE problem is: Given a graph G and positive integer k , find if it contains a clique of size k in G . Reduction: Given a graph G and k , construct a second graph G' which is a complete graph with k nodes (i.e., it has all possible edges). Feed $\langle G, G' \rangle$ to *Subgraph-Isomorphism*. Accept iff *Subgraph-Isomorphism* accepts.

The reduction clearly requires polynomial time. Or does it? Well, if k is large, then our construction is flawed. The encoding for k require only $\log k$ bits while the complete graph G' has $O(k^2)$ edges. Thus, the size of G' could be exponential in the size of the input $\langle G, k \rangle$. Okay, the fix is: Before generating G' , count the number of edges in G . Construct G' only if there at least $\binom{n}{k}$ edges in G .

Proof of correctness is straightforward.

(C) Consider the *graph-isomorphism problem*: Given two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, is there a 1 - 1 mapping $\pi : V_1 \rightarrow V_2$ such that $(u, v) \in E_1$ iff $(\pi(u), \pi(v)) \in E_2$? (a) Is the graph-isomorphism problem reducible to subgraph-isomorphism problem? (b) Is the

subgraph-isomorphism problem reducible to the graph-isomorphism problem? (c) Is the graph-isomorphism problem in NP?

(a) Yes, Just feed the pair $\langle G_1, G_2 \rangle$ to *Subgraph Isomorphism*.

(b) It is not known whether such a reduction is possible. It is an open problem whether Graph-Isomorphism is an NP-complete problem or not.

(c) Yes, simply guess the mapping π and verify that all edges respect the mapping. This requires polynomial time.

(D) Given an integer $m \times n$ matrix A and an integer m -vector b , the 0-1 Integer Programming Problem asks whether there is an integer n -vector x with elements in the set $\{0, 1\}$ such that $Ax \leq b$. Prove that 0-1 Integer Programming is NP-Complete. Hint: Reduce 3SAT.

0-1 Integer Programming \in NP: Simply guess the assignment of variables to x and verify the condition $Ax \leq b$.

Reduction from 3SAT: For each literal x in 3SAT, we have two variables x and x' . The pair of constraints: $x + x' \leq 1$ and $-x - x' \leq -1$ ensures that the only possible solution in integers is $x = 1 - x'$ where $x \in \{0, 1\}$.

Let's say that there are k clauses with the j^{th} clause having the form $(\ell_{j1} \cup \ell_{j2} \cup \ell_{j3})$, where $\ell_{j1}, \ell_{j2}, \ell_{j3}$ are three literals. For each clause, we have a pair of constraints $\ell_{j1} + \ell_{j2} + \ell_{j3} \leq 3$ and $-\ell_{j1} - \ell_{j2} - \ell_{j3} \leq -1$. This ensures that for each clause, at most 3 and at least 1 literal has value 1.

Proof: If the 3SAT instance has a solution, then there is an assignment to the various x and x' variables that satisfies all the constraints. If the set of constraints has a solution, then all pairs of x and x' variables are assigned complementary values in the set $\{0, 1\}$ and all clauses have at least one true variable.

(E) Let $DOUBLE - SAT = \{\langle \phi \rangle \mid \phi \text{ has at least two satisfying assignments}\}$. Show that $DOUBLE - SAT$ is NP-Complete. Hint: Reduce 3SAT.

DOUBLE - SAT \in NP: Simply guess two different assignments to all variables and verify that each clause is satisfied in both cases.

Reduction from 3SAT: We add a new clause $(x \cup \bar{x})$ and feed the instance to *DOUBLE - SAT*, where x is a new variable.

Proof: If the original 3SAT formula is unsatisfiable, the new formula is also unsatisfiable. Otherwise, both $x = 0$ and $x = 1$ are valid assignments. Thus there are at least two satisfying assignments of the augmented CNF formula fed to *DOUBLE - SAT*.

Problem 5(a)

(Sipser 7.23) MAX-CUT is NP-Hard

The reduction from \neq -3SAT is described in Sipser. It takes a formula ϕ (with n variables and k clauses) and converts it into a graph G . We will show that the reduction works by proving the following claims:

\Rightarrow *If ϕ has a satisfying assignment with at most two TRUE literals per clause, then G has a MAX-CUT of size $9k^2n + 2k$.*

\Leftarrow *If G has a MAX-CUT of size $9k^2n + 2k$, then ϕ has a satisfying assignment with at most two TRUE literals per clause.*

The proofs in both directions are straightforward once we establish the following:

Claim: Let T and F denote the partition of vertices of G into two disjoint sets that corresponds to the MAX-CUT. Then for each variable x , the set of $3k$ nodes corresponding to x lie either in T or F . The same holds for the set of $3k$ nodes corresponding to \bar{x} .

Proof: By contradiction. Consider a cut in which some vertex for x lies in T and another vertex for x lies in F . Then by moving the first vertex into F , we introduce $3k$ edges into the cut. However, we reduce the number of edges (due to the triangles corresponding to clauses) in the cut by at most two. Thus, the cut we started out with was not of maximum size.

Problem 2

(Sipser 7.24) Prove that $D = \{ \langle p \rangle \mid p \text{ is a polynomial in several variables with an integral root} \}$ is NP-Hard.

A problem is *NP-Hard* if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself. This means that all NP-Complete problems are also NP-Hard.

Reduction from 3SAT: A 3SAT formula is a conjunction of clauses, where each clause is a disjunction of at most three literals. First, we introduce n new clauses where for each variable x , we have a clause $(x \vee \bar{x})$. Next, we carry out the following transformation:

- (a) Replace each disjunction symbol \vee by the product symbol \times .
- (b) Replace each conjunction symbol \wedge by sum symbol $+$.
- (c) Replace each uncomplemented literal x by $(1 - x)^2$.
- (d) Replace each complemented literal \bar{x} by x^2 .

For example, the 3SAT formula

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_2 \vee x_4)$$

is first transformed into the formula

$$(x_1 \vee \bar{x}_2 \vee x_3) \wedge (x_2 \vee x_4) \wedge (x_1 \vee \bar{x}_1) \wedge (x_2 \vee \bar{x}_2) \wedge (x_3 \vee \bar{x}_3) \wedge (x_4 \vee \bar{x}_4)$$

which is then transformed into the polynomial

$$(1 - x_1)^2 x_2^2 (1 - x_3)^2 + (1 - x_2)^2 (1 - x_4)^2 + (1 - x_1)^2 x_1^2 + (1 - x_2)^2 x_2^2 + (1 - x_3)^2 x_3^2 + (1 - x_4)^2 x_4^2$$

Note that we have omitted the \times symbols for clarity.

Proof of Correctness:

\Rightarrow If 3SAT has a satisfying assignment, then the polynomial becomes zero for integral values of its constituent variables. It is easy to see why this is true.

\Leftarrow If the polynomial has an integral root, then the 3SAT formula has a satisfying assignment. Note that our polynomial is a sum of products. Each product has squared terms. So for the entire polynomial to evaluate to zero, each of the products must be zero. Products of the form $(1 - x)^2 x_2$ (that we introduced per variable) ensure that each variable takes the value 1 or 0. Since each of the other products is also true, every disjunction in the original formula is true.

Problem 3

(Sipser 7.34) Show that $3\text{COLOR} = \{ \langle G \rangle \mid \text{the nodes of undirected graph } G \text{ can be colored with three colors such that no two nodes joined by an edge have the same color} \}$ is NP-Complete.

We will reduce 3SAT to 3COLOR . For a 3SAT formula with k clauses and n variables, we will create an undirected graph with $3 + 2n + 6k$ nodes.

- There is a small-sized graph called the *palette*. It is simply a 3-clique over three vertices $\{F, T, Z\}$.
Note: F denotes False and T denotes True. Z is a Dont-Care.
- For each variable x in 3SAT , we have two nodes in the graph, denoted by x and \bar{x} respectively. There is an edge (x, \bar{x}) between the two nodes. Thus there are $2n$ nodes corresponding to variables in 3SAT . Moreover, for every variable x , node Z in the palette (described above) is connected to nodes x and \bar{x} . This means that there are two edges: (Z, x) and (Z, \bar{x}) respectively. *This forces x and \bar{x} to be colored T or F .*
- For every clause with exactly 3 literals, there is a 6-node gadget: A graph with six vertices $\{p, q, r, s, t, u\}$ which is equivalent to 3-cliques over $\{p, q, r\}$ and $\{s, t, u\}$ with an edge (q, s) connecting them. This means that there are exactly seven edges: $\{(p, q), (q, r), (r, p), (q, s), (s, t), (t, u), (u, s)\}$.

The p node in each gadget is connected to the *palette* by two edges: (F, p) and (Z, p) , where F and Z are vertices in the palette (described above). *Note that this forces p to be colored T .*

Every gadget corresponds to a clause with 3 literals. The vertices r, t and u are connected to 3 nodes corresponding to 3 variables in a clause. For example, let's consider the clause $(x_1 \vee \bar{x}_2 \vee x_3)$. Then, there are 3 edges: $(r, x_1), (t, \bar{x}_2)$ and (u, x_3) .

Any clause with 1 or 2 literals can easily be transformed into a 3-literal clause because $x = x \vee x$.

\Rightarrow If the 3SAT formula has a satisfying assignment, then the graph has a 3-coloring.

\Leftarrow If the graph has a 3-coloring, the 3SAT formula has a satisfying assignment.

The above claims can be proved by a case-by-case analysis. The crucial observations are: (a) in any gadget, node p is always colored T , (b) all variable nodes (x and \bar{x}) are always colored either T or F , and (c) each gadget is connected to three literals; it is not possible to color all three literals F when p has been forced to be T .

Problem 4(a)

(Sipser 7.21) **HALF-CLIQUE** = $\{\langle G \rangle \mid G \text{ is an undirected graph having a complete subgraph with at least } n/2 \text{ nodes, where } n \text{ is the number of nodes in } G\}$.

HALF-CLIQUE \in NP: Just guess a subset of vertices (with size at least $n/2$) and verify that each pair of vertices in this subset is connected by an edge.

Reduction from CLIQUE: Given $\langle G, k \rangle$ with G having n nodes, first test whether $k > n/2$ or not. There are three cases:

- $k = n/2$: G has a clique iff HALF-CLIQUE accepts $\langle G \rangle$.
- $k > n/2$: Add $2k - n$ new vertices with degree zero to G so as to obtain a new graph G' with exactly $2k$ vertices. It is clear that G has a clique of size k iff G' has a clique of size k or more.
- $k < n/2$: Add $n - 2k$ new vertices to G . New vertices have two kinds of edges: (i) Each pair of new vertices has an edge between them and (ii) Each new vertex has an edge with each old vertex. Graph G has a clique of size k iff G' has a clique of size $n - k$ or more.

Note that if G has n vertices, we add at most $O(n)$ vertices and $O(n^2)$ edges to create G' . Therefore, our reduction is polynomial time.

Problem 4(b)

Let **MAX-CLIQUE** = $\{\langle G, k \rangle \mid \text{the largest clique of } G \text{ has } k \text{ vertices}\}$. Whether **MAX-CLIQUE** is in NP is unknown. Show that if $P = NP$, then **MAX-CLIQUE** is in P, and a polynomial time algorithm exists that, for a graph G , finds one of the largest cliques.

CLIQUE = $\{\langle G, k \rangle \mid G \text{ is a graph with a } k\text{-clique}\}$.

If $P = NP$, then *CLIQUE* is in P. We invoke *CLIQUE* with $\langle G, i \rangle$ with i ranging from 1 to $|V|$, where V is the set of vertices of G . The largest value of i for which *CLIQUE*(G, i) is true happens to be the size of the largest clique.

How do we actually discover the vertices that belong to the largest clique? We initialize $H \leftarrow G$. Next, we iterate over all vertices of G . When the iteration encounters vertex v , we create a graph H_{without_v} which is simply H with vertex v (and all edges incident upon it) removed. If *CLIQUE*(H_{without_v}, k) is TRUE, we set $H \leftarrow H_{\text{without}_v}$; otherwise we don't change H . Interestingly, this procedure works even if there are multiple (possibly disjoint) cliques in the original graph G .

Overall time is polynomial because we iterate over $|V|$ vertices. In each iteration, we create a new graph by removing an existing vertex. This can be done in polynomial time. Further, each iteration invokes *CLIQUE*() exactly once. Assuming $P = NP$, the overall procedure takes polynomial time.

Problem 5(a)

(Sipser 7.29) Show that, if $P = NP$, we can factor integers in polynomial time.

Let $\text{LARGE_FACTOR} = \{\langle n, t \rangle \mid n \text{ and } t \text{ are positive integers, and } n \text{ has a factor } f \text{ satisfying } t \leq f < n\}$

Claim: $\text{LARGE_FACTOR} \in NP$. A non-deterministic TM for LARGE_FACTOR simply guesses an integer f and verifies that (a) $t \leq f < n$ and (b) f divides n .

By invoking $\text{LARGE_FACTOR}(n, 2)$, we check if n is composite or not. If n is composite, we use LARGE_FACTOR to actually discover the prime factorization of n as follows:

Let us denote the largest factor of n that lies in the range $(1, n)$ by $f_{largest}$. Then $\text{LARGE_FACTOR}(n, f)$ is TRUE for all $f \in [2, f_{largest}]$ but FALSE for all $f \in [f_{largest} + 1, n]$. How do we discover $f_{largest}$? If we iterate f from 1 thru n , we would be doing exponential amount of work in the size of the input (because we have $O(n)$ iterations when the size of the input is only $O(\log n)$). A faster procedure is to use binary search to identify $f_{largest}$. This would involve at most $O(\log n)$ invocations of LARGE_FACTOR .

Having discovered the largest factor of n , say $f_{largest}$, we compute $p = n/f_{largest}$, which happens to be the smallest prime factor of n . We now discover the largest factor of n/p , and so on (using the above procedure for discovering the largest factor).

Any number n can have at most $O(\log n)$ prime factors. As we showed earlier, each prime factor requires $O(\log n)$ invocations of LARGE_FACTOR for its discovery. Thus, LARGE_FACTOR is invoked $O(\log^2 n)$ times. Since $P = NP$, the whole procedure is polynomial.

Problem 5(b)

(Sipser 7.29) Show that, if $P = NP$, a polynomial time algorithm exists that, given a Boolean formula ϕ , actually produces a satisfying assignment for ϕ if it is satisfiable.

Let the set of variables in the Boolean formula be x_1, x_2, \dots, x_k .

```
FIND_ASSIGNMENT ( $\phi$ )
{
    Array  $A[1 \dots k]$  ;

    IF ( $3SAT(\phi)$  is FALSE)
        RETURN NO-ASSIGNMENT

    FOR  $i = 1$  TO  $k$ 
         $\phi_0 \leftarrow \phi \wedge (\overline{x_i} \vee \overline{x_i} \vee \overline{x_i})$ 
         $\phi_1 \leftarrow \phi \wedge (x_i \vee x_i \vee x_i)$ 
        IF ( $3SAT(\phi_0)$  is TRUE)
             $\phi \leftarrow \phi_0$      $A[i] \leftarrow 0$ 
        ELSE
             $\phi \leftarrow \phi_1$      $A[i] \leftarrow 1$ 

    RETURN  $A$  ;
}
```

Assuming $P = NP$, $3SAT$ is in P . There are at most k invocations of $3SAT$. The formula ϕ grows gradually but remains polynomial in the size of the input. Overall, the algorithm runs in polynomial time.