CS 170: Efficient Algorithms and Intractable Problems

Spring 2017

•

Homework 9

Due on Tuesday, April 18th, 2017 at 11:59am

•

Solutions by

Michael Fan

26697596

In collaboration with

Alex Gao

Problem 1: Basic Complexity Concepts

★ **Level**

a. Suppose we reduce a problem A to another problem B. This means that if we have an algorithm that solves ___, we immediately have an algorithm that solves ___.

Solution B, A

b. Again, suppose we show a (polynomial-time) reduction from A to B. This implies that up to polynomial factors, ___ cannot be any easier than ___.

Solution B, A

c. Define the class **NP** in one sentence. *There are several correct characterizations. One of them is the most intuitive, and the definition we use in this class. Give this definition.*

Solution We can check the validity of solutions to this question in polynomial time.

d. Define the class **NP**-hard in one sentence.

Solution Search problems that are at least as hard as the hardest problems in NP.

e. Define the class **NP**-complete in one sentence.

Solution The hardest problems in NP that are reducible to each other and if all other search problems reduce to it.

f. Show that for any problem Π in **NP**, there is an algorithm which solves Π in time $\mathcal{O}(2^{p(n)})$, where n is the size of the input instance and $p(n)$ is a polynomial (which may depend on Π).

Solution A NP-Complete question such as SAT can be solved in $\mathcal{O}(2^{p(n)})$ by checking every combination of true/false assignments to its variables (which is exponential time as there are an exponential number of assignments that each take linear time to check). Then, every other search problem reduces to SAT. Thus, every problem in NP can be solved in $\mathcal{O}(2^{p(n)})$.

Problem 2: Proving NP-completeness by generalization

★★ Level

Proving **NP**-completeness by generalization. For each of the problems below, prove that it is **NP**-complete by showing that it is a generalization of some **NP**-complete problem we have seen in the lecture or in the book. You only need to show the reduction part in this problem (there is no need to show that the problem is in **NP**).

a. SUBGRAPH ISOMORPHISM: Given as input two undirected graphs G and H, determine whether G is a subgraph of H (that is, whether by deleting certain vertices and edges of H we obtain a graph that is, up to renaming of vertices, identical to G), and if so, return the corresponding mapping of $V(G)$ into $V(H)$.

> **Solution** This is a generalization of the clique problem, which finds a clique of size k for some graph H. Preprocessing: create a graph G that is just an interconnected set of k vertices (connecting all vertices to each other). Then, solve the subgraph problem given our graph G and H. The given algorithm will tell us whether or not the K-size clique (graph G) exists within graph H, which would also tell us the solution to the clique problem.

b. LONGEST PATH: Given a graph G and an integer g, find in G a simple path of length g.

> **Solution** Longest path solves the Hamiltonian Path problem. Simply change all length inputs in the graph to have length 1. If there exists a simple path of length equal to the number of vertices, then there also exists a Hamiltonian path.

c. MAX SAT: Given a CNF formula and an integer g, find a truth assignment that satisfies at least g clauses.

> **Solution** MAX SAT solves the regular SAT problem. The CNF formula is exactly like the one that regular SAT would solve; the pre-processing is simply to set g equivalent to the number of existing clauses in the CNF, indicating that all the clauses would need to be satisfied, which is what SAT would normally solve.

d. DENSE SUBGRAPH: Given a graph and two integers a and b, find a set of a vertices of G such that there are at least b edges between them.

> **Solution** This can be used to solve a regular clique problem. Given a clique problem of finding a clique of size k in graph G, we can use Dense Subgraph. Set $a = k$, and set $b = n(n-1)/2$ which is the number of edges between n vertices if they were all connected using the maximal number of vertices (AKA the definition of a clique). Then, running Dense Subgraph will find the clique in G.

e. SPARSE SUBGRAPH: Given a graph and two integers a and b, find a set of vertices of G such that there are at most b edges between them.

> **Solution** This solves an independent set problem, for which we want to find an independent set of size k in graph G. just set $a = k$ and $b = 0$. This returns vertices with no edges between them, which is an independent set.

Problem 3: Reduction, Reduction, Reduction

★★★ Level

Instruction: In this problem, you will prove three problems are **NP**-Complete. Please be aware that to prove a problem is **NP**-Complete, you need to argue it is in **NP** and then briefly justify a reduction that shows it is **NP**-hard.

1. Alice and Bob go out on a date at a nice restaurant. At the end of the meal, Alice has eaten c_A dollars worth of food, and has in her wallet a set of bills $A = \{a_1, a_2, \ldots, a_n\}$. Similarly, Bob owes the restaurant c_B dollars and has bills $B = \{b_1, b_2, \ldots, b_m\}$. Now, Alice and Bob are very calculating people, so they agree that each of them should pay their fair share (c_A and c_B, respectively).

   One thing they don't mind doing, however, is fairly trading bills. That is, Alice can exchange a subset of $A' \subseteq A$ of her bills for a subset $B' \subseteq B$ of Bob's bills, so long as $\sum_{a \in A'} a = \sum_{b \in B'} b$. Under the above conditions, Alice and Bob wish to find, after trading as many times as desired, subsets of their bills A^*, B^* such that $\sum_{a \in A^*} a = c_A$ and $\sum_{b \in B^*} b = c_B$.

   Show that Fair Date is **NP**-complete.

   Hint: You may assume SUBSET SUM is **NP**-complete.

   Solution Let us first justify the necessary reductions. The reduction looks like this - if you have a SUBSET SUM problem with some set of elements S, then just represent all the elements S as bills that Alice owns, and let Bob be a poor and hungry sap who owns no money and eats nothing. Then, let Alice's target c_A equal the target of SUBSET SUM. The FAIR DATE algorithm then just sums up a subset of S to reach the target c_A.

   The proof requires two propositions to be proven - 1. If there is a solution to FAIR DATE there is a solution to SUBSET SUM. Also, 2. if there is no solution to FAIR DATE there is no solution to SUBSET SUM. The first proposition is clear - if FAIR DATE finds a solution for the constraints on Alice and Bob as previously mentioned, then the set $A*$ will also be a solution to SUBSET SUM. The second proposition is also clear by contraposition - if there is a solution to SUBSET SUM, then we know there is some set that adds up to some target - all we need to do is formulate FAIR DATE so that no trades occur.

   The NP-ness of the problem can be seen by how much time it takes to verify FAIR DATE. The verifying of sum of A' = sum of B', along with the sum of A* and B* meet their targets, is linear. Need to also prove A* exists in either the original set A or the traded set B', and the same for B*. All of these are linear operations, so the verification time is linear, which is a subset of Polynomial.

2. Consider the following problem Public Funds:

   > You are looking to build a new fence for your mansion, to keep out pesky people protesting profligate purchases. You have m bank account at your disposal to use to pay for your fence; each account i has a balance of b_i. You must choose one of n options for your fence; each fence j costs c_j dollars. You would like to withdraw from at most k of the bank accounts to build the fence, and due to peculiar UC accounting rules, if you use a particular bank account, you must use the whole balance (all b_m dollars.)
   >
   > Determine whether it is possible to exactly pay for some fence j; that is, whether there is a j between 1 and n such that you can withdraw exactly c_j dollars given the bank account balances $b_1, \ldots, b_m$, the fence costs $c_1, \ldots, c_n$ and k, and if so return the corresponding choice of fence and set of of bank accounts that you withdraw from.

   Show that Public Funds is **NP**-Complete.

   Solution Verification of this involves making sure the sum of the bank account values is equivalent to the target c amount of dollars, that you only withdraw from k or fewer bank accounts, and that the target c corresponds to a fence that you can buy. Each of these processes is linear time to verify in the number of bank accounts - just need to iterate through the number of bank accounts one time, and

Solution (cont.)

then iterate through your set of fences one time. This is linear time, which is a subset of polynomial time, it's in NP.

For the proof we need to reduce some other NP-complete problem to this. This problem generalizes to just the knapsack problem, where each bank account is like an "item", each value of the account is like a "weight", and the capacity of the knapsack is some exactly value j, and testing the knapsack for all j values where you maximize the value of the knapsack. A solution to one of the j knapsack problems is also an indicator that PUBLIC FUNDS has a solution - everything in the Knapsack would just be the set of bank accounts you draw from. In addition, if there is no solution to any of the j knapsack problems, then there is no exact combination of bank accounts which sum up to any j value for the fence.

3. Consider the search problem MAX-ACYCLIC-INDUCED-SUBGRAPH:
   INPUT: A *directed* graph $G = (V, E)$, and a positive integer k.
   OUTPUT: A subset $S \subseteq V$ of size k such that the graph G_S obtained from G by keeping exactly those edges both whose endpoints are in S is a DAG.
   Show that MAX-ACYCLIC-INDUCED-SUBGRAPH is **NP**-complete.

Solution Verification of this just checks to see that the vertices do not include a cycle, and there is no possible larger acyclic subgraph. The second task is the real bottleneck on the verification time. You would need to test larger subgraphs to see if they are acyclic as well, which involves testing out other potential 'branches' of the sub-DAG you have, which is polynomial time in the number of edges.

This is reducible to the maximum independent set problem. If we have some graph G that we want to find the max independent set on, we can alter it by adding directionality in both ways to each edge (doubling the number of edges) and making it the input to Max Acyclic Subgraph. For acyclic subgraphs to exist in the modified G', there can't be any edges between the vertices to begin with, meaning that they were independent in the original graph G. Thus, the largest acyclic subgraph in G' would reduce to the largest independent set in the original G. We kknow independent set is NP-complete, so this reduction checks out.

Problem 4: Graphs and Matrix Multiplication.

★★★★ Level

Consider the MATRIX MULTIPLICATION problem:

Input: Two $n \times n$ matrices M_1 and M_2.
Output: The matrix product $M_1 M_2$.

Suppose there is an algorithm which solves MATRIX MULTIPLICATION in time $t(n)$.

(a) Show that given M and an integer $p > 0$, you can computer M^p in time $\mathcal{O}(t(n) \log p)$. You only need to prove a main idea and show how to satisfy the time complexity requirement.

Solution
To solve M^p, we can begin by splitting the problem into $M^{p/2} * M^{p/2}$. We assume WLOG that $p = 2^k$. We note that it is redundant to solve both subproblems as they give the same solution. The cost of merging the 2 subproblems is $t(n)$ where n is the dimension of the square matrix. If we consider this question bottom-up, we can solve M^p in $\mathcal{O}(t(n) \log p)$ by starting from the subproblem $M^1 = M$ and squaring the output $k = \log p$ times in order to solve subsequent subproblems until we have hit M^p. Note that it will take $\log p$ squarings and a cost of $t(n)$ per squaring. Thus, our runtime is $\mathcal{O}(t(n) \log p)$.

(b) Given a directed graph G in adjacency matrix representation and two nodes s and t, you want to know whether there is a path from s to t. This is called the GRAPH REACHABILITY problem. Show that you can solve GRAPH REACHABILITY in time $\mathcal{O}(t(n) \log n)$, where n is the number of nodes in the graph. You need to first give the main idea, and justify the time complexity requirement. Also you need to prove that your algorithm is correct.
Hint 1: It may help to use part (a).
Hint 2: In matrix multiplication, what does $(A^k)_{ij}$ mean?

Solution
Main Idea: Let M be the adjacency matrix for G. M is a $n \times n$ matrix as G has n vertices. Entry (i, j) of M is 1 if nodes i and j are adjacent to each other (connected by an edge). To solve GRAPH REACHABILITY, we will compute M^n using our matrix multiplication algorithm. If entry (s, t) of M^n is nonzero, then there is path from s to t. If it is zero, then there is no path.
Proof of Correctness: Let us consider the matrix M^2. Entry (i, j) of M^2 is computed by taking the dot product of row i of M and column j of M. Then, (i, j) will be nonzero if the kth value of row i and kth value of column j are both nonzero for $1 \leq k \leq n$. In the context of M's function as an adjacency matrix, entry (i, j) of M^2 will be nonzero if both nodes i and j are adjacent to some common node k (or if they are already connected by an edge), or in other words, if they have a degree of separation of 2 (being directly adjacent is a degree of separation of 1). For every additional node that i and j are both adjacent to, the value of (i, j) will increase by 1, but this is not important for the purpose of this question. Entry (i, j) in M^3 will then be nonzero if there is a common node for which any of i and j's adjacent nodes are both adjacent to. In other words, it will be nonzero if i and j are have a degree of separation of 3. M^n will then test for "adjacency to the nth degree of separation": if i and j are connected after traveling n edges, then entry (i, j) will be nonzero. However, the longest possibly path in a graph without repeating vertices is n - a cycle of a graph. Thus, by taking M^n, we check if there is any length l path up to a cycle of length n connecting s and t. If there is, then entry (s, t) of M^n will be nonzero.
Runtime Analysis: In order to solve GRAPH REACHABILITY, we just have to compute M^n and read entry (s, t) to check if it is nonzero. We can compute M^n in $\mathcal{O}(t(n) \log n)$ (replace p with n) from part (a) and check (s, t) in constant time. Thus, the runtime is $\mathcal{O}(t(n) \log n)$.

(c) Now, suppose you know that graph reachability cannot be solved in time $\mathcal{O}(T(n) \log n)$. Show that MATRIX MULTIPLICATION cannot be solved in time $\mathcal{O}(T(n))$. This should be a very simple proof.

Solution We will show a reduction from GRAPH REACHABILITY to MATRIX MULTIPLICATION. The input to GRAPH REACHABILITY is a graph G. We can construct the adjacency matrix to G in polynomial time and use it as the input for M to MATRIX MULTIPLICATION. The other input, n, will be the number of vertices in G which can be extracted in polynomial time. The inverse function to transform an answer to MATRIX MULTIPLICATION to one for GRAPH REACHABILITY is simple: we read entry (s, t) of the computed matrix, M^n. If it is zero, we return FALSE as an answer to GRAPH REACHABILITY. If it is nonzero, we return TRUE. This is a constant time read (which is a polynomial time transformation). Thus, we have proven that there is a reduction from GRAPH REACHABILITY to MATRIX MULTIPLICATION. This means that MATRIX MULTIPLICATION is at least as hard as GRAPH REACHABILITY. Thus, if GRAPH REACHABILITY cannot be solved in $\mathcal{O}(T(n) \log n)$, then there is no way that MATRIX MULTIPLICATION can be solved in an even faster bound of $\mathcal{O}(T(n))$ as it is at least of equal difficulty as GRAPH REACHABILITY.

Problem 5: Finding Zero(s)

★★★★★ **Level**

Consider the problem INTEGER-ZEROS.

INPUT: A multivariate polynomial $P(x_1, x_2, x_3, \ldots, x_n)$ with integer coefficients, specified as a sum of monomials.
OUTPUT: Integers $a_1, a_2, \ldots, a_n$ such that $P(a_1, a_2, a_3, \ldots, a_n) = 0$.

Show that 3-SAT reduces in polynomial time to INTEGER-ZEROS. (You do not need to show that INTEGER-ZEROS is in **NP**: in fact, it is know *not* to be in **NP**).
Hint 1: Given a 3-SAT formula ϕ in the variables $x_1, x_2, \ldots, x_n$, your reduction f will produce a polynomial P in the same variables such that satisfying assignments correspond to 0, 1 valued zeros of P.
Hint 2: If your polynomial constructed above has exponential amount of terms, it is not optimal! You need to reduce it to polynomial amount. Think about the equality $a^2 + b^2 = 0$ if and only if $a = b = 0$ when a, b are real to achieve it. [1]

Solution In order to show a reduction from 3-SAT to INTEGER-ZEROS, we have to find a polynomial time algorithm A to transform an instance of 3-SAT to an instance of INTEGER-ZEROS and a polynomial time algorithm B that maps any solution of INTEGER-ZEROS back into a solution of 3-SAT.
To start, note that any instance of 3-SAT is composed of "and"-conjunctions of 3 variables linked by binary "or" operators: $(a \vee b \vee \bar{c}) \wedge (\bar{b} \vee c \vee d) \wedge \ldots$. We note that each clause of 3 variables is made true if at least 1 variable is true, and any variable x will have the opposite assignment to its complement $\bar{x}$. Thus, we arrive at algorithm A: every unique, bar-less variable a will be represented as a variable x_i of degree 1, and all complements (variables with bars) $\bar{a}$ will be represented as $1 - x_i$, where x_i is this variable's corresponding complement. All $\vee$ operations will be directly mapped to integer multiplication and all $\wedge$ operations will be directly mapped to integer addition. Then, $(a \vee b \vee \bar{c}) \wedge (\bar{b} \vee c \vee d) \wedge \ldots \rightarrow x_1 x_2 (1 - x_3) + (1 - x_2) x_3 x_4 + \ldots = 0$, and we have a generation function A for transforming instances of 3-SAT into instances of INTEGER-ZEROS. Note that this algorithm is linear time with respect to the length of the 3-SAT input as we have a 1 to 1 mapping of 3-SAT variables to numerical variables and $\vee \rightarrow \times$, $\wedge \rightarrow +$.
Upon examining the new INTEGER-ZEROS instance, we see that for each monomial, at least one of the 3 sub-components have to be equal to 0 for the statement to be true. We solve this instance of INTEGER-ZEROS. In the case that there is a solution, we note that any $x_i = 0$ implies that $1 - x_i = 1$, $1 - x_i = 0$ implies that $x_i = 1$. This leads to algorithm B: $x_i = 0$ implies that its 3-SAT variable counterpart a is assigned TRUE. Conversely, $x_i \neq 0$ implies that its 3-SAT variable counterpart a is assigned FALSE. In this way, we guarantee that there is at least one TRUE in each 3-SAT sub-clause as there has to be a zero in every monomial in the corresponding INTEGER-ZEROS instance. Note that B is also polynomial time - we go through each integer variable, find its corresponding 3-SAT variable, and assign it TRUE if the integer variable is 0, and false if the integer variable is true.
Because we have found A and B as defined, we have found a polynomial time reduction from 3-SAT to INTEGER-ZEROS.

[1] Fun fact: This problem INTEGER-ZEROS is *really* hard: the decision version of the problem is *undecidable*. This means that there is provably no algorithm which, given a multivariate polynomial, will decide correctly whether or not it has integer zeros. However, if we relax the problem to ask if there are *real* numbers at which the given polynomial vanishes, then the problem surprisingly becomes decidable! For Blue and Gold jingoists: the above decidability result was proven by Alfred Tarski, a Berkeley professor, in 1949. The undecidability of INTEGER-ZEROS was proven by Yuri Matiyasevich (at the ripe old age of 23!) in 1970, building upon previous work of another Berkeley professor, Julia Robinson.