CS 170
Spring 2017

Efficient Algorithms and Intractable Problems
Prasad Raghavendra and Sanjam Garg

Homework 9

Instructions: You are welcome to form small groups (up to 4 people total) to work through the homework, but you **must** write up all solutions by yourself. List your study partners for homework on the first page, or "none" if you had no partners.

If using LaTeX (which we recommend), you may use the homework template linked on this Piazza post to get started.

Begin each problem on a new page. Clearly label where each problem and subproblem begin. The problems must be submitted in order (all of P1 must be before P2, etc). For questions asking you to give an algorithm, respond in what we will refer to as the *four-part format* for algorithms: main idea, pseudocode, proof of correctness, and running time analysis.

Read the Homework FAQ Piazza post on Piazza before doing the homework for more explanation on the four-part format and other clarifications for our homework expectations.

No late homeworks will be accepted. No exceptions. This is not out of a desire to be harsh, but rather out of fairness to all students in this large course. Out of a total of approximately 12 homework assignments, the lowest two scores will be dropped.

Special Questions:

- *Shortcut questions*: Short questions are usually easy questions that give you opportunities to practice basic materials. However, we understand that some of you are very familiar with the topics and do not want to spend too much time on easy questions. Therefore, we design shortcut questions for this purpose. A shortcut question usually has multiple parts that build upon each other and are ordered by their difficulty level. You can work on those in order or start from wherever you like. However you only need to submit the last part you are able to solve. For example, if a question has 5 parts (a, b, c, d, e), you are confident about part e, you should submit part e without any of the previous four parts. If you are confident about d but not sure about e, you should submit d for grading purposes. Please clearly indicate in your submission which part you are submitting.
- *Redemption questions*: It is important that you carefully read the posted solutions, even for problems you got right. To encourage this, you have the option of submitting a redemption file, a few paragraphs in which you explain, for each problem you choose to cover, what you did wrong and what the right idea was in your own words (not cutting and pasting from the solution!), and appending it to your homework. For example, suppose that as you review your solutions to HW1, you realize you had misunderstood question 3 and answered it incorrectly. You would write down what you just learned, and then submit it in your HW2 assignment the following week. Because these are mainly for your benefit, feel free to format them however is most useful for you.
- *Extra credit questions*: We might have some extra credit questions in the homework for people who really enjoy the materials. However, please note that you should do the extra credit problems only if you really enjoy working on these problems and want an extra challenge. It is likely not the most efficient manner in which to maximize your score.

Due Tuesday, April 18, at 11:59am

1. (★ level) Basic Complexity Concepts

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: The class of problems for which we can verify any proposed solution in polynomial time.
Another acceptable answer: "The class of search problems."

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

Solution: The class of problems to which all problems in **NP** reduce.

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

Solution: The set of problems that are **NP**-hard and also in **NP**.
"The set of search problems to which all search problems reduce" is also acceptable.

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

Solution: Since $\Pi \in$ **NP**, $\exists$ a verifier V that can verify solutions in $O(q(n))$ time for some polynomial $q(\cdot)$. Since V must read a solution to verify it, V can read at most $q(n)$ input bits. Therefore, we can simply run V on all binary strings of size $2^{q(n)}$. If the instance of Π has a solution, then one of these inputs is a solution. This solves any instance of Π in $O(q(n)2^{q(n)}) = O(2^{2q(n)}) = O(2^{p(n)})$, where $p(n) = 2q(n)$.
Alternatively, we can reduce Π to an NP complete problem that has an obvious exponential solution (i.e. CircuitSAT, 3-SAT, SAT). Because reductions are polynomial time, the CircuitSAT instance in the reduction has size $q(n)$, for some polynomial q where $n = |\Pi|$. Therefore, it suffices to show that there exists an algorithm that solves CircuitSAT in time $O(2^{p_1(n)})$, where p_1 is a polynomial. If this holds, then through the reduction we will get an algorithm to solve Π in $O(2^{p(n)})$ time, where $p(n)$ is the polynomial $p_1(q(n))$. Consider any CircuitSAT instance I. The instance I can have only a finite number k of input wires, which are either 0 or 1, so there are 2^k possible inputs. Therefore, we can try all of the 2^k possible inputs to see if the circuit is satisfiable. Let $n = |I|$ be the size of the circuit, so we know that $k \leq n$. Let $p_2(n)$ be the time it takes to evaluate the circuit on a given input. Trying all possible inputs will take $O(p_2(n)2^k)$ time. However, since $p_2(n)$ is a polynomial, and $k \leq n$, $O(p_2(n)2^k)$ is $O(2^{2n})$. Therefore, the brute force algorithm solves CircuitSAT in $O(2^{2n})$ time, so this gives us an algorithm to solve Π in $O(2^{2q(n)})$ time.

2. (★★ level) Proving NP-completeness by generalization.

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: We can view this as a generalization of the CLIQUE problem. Given an input (G,k) for CLIQUE, let H be a graph consisting of k vertices with every pair connected by an edge (i.e. a clique of size k). Then G contains a clique of size k if and only if H is a subgraph of G.

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

Solution: This is a generalization of RUDRATA-PATH. Given a graph G with n vertices, let $g = n-1$. Then $(G, n-1)$ is an instance of LONGEST PATH. However, a simple path of length $n-1$ must contain n vertices and hence must be a Rudrata path. Also, any Rudrata path is of length $n-1$. Hence for any graph with n vertices, LONGEST-PATH$(G, n-1)$ is precisely the asking for the Rudrata path.

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

Solution: Given a formula ϕ with m clauses, setting $g = m$ gives SAT as a special case of MAX-SAT.

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 also a generalization of CLIQUE. Given an instance (G,k) let $a = k$, $b = k(k-1)/2$. Any subgraph of G with k vertices and containing $k(k-1)/2$ edges, must have an edge between every possible pair out of these k vertices and hence must be a clique. Similarly, a clique on k vertices must contain $k(k-1)/2$ edges.

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

Solution: This is a generalization of INDEPENDENT SET. Given (G,k), an instance for independent set, let $a = k$, $b = 0$. Then (G,a,b) is an instance for SPARSE SUBGRAPH. Also, any subgraph of G with k vertices and 0 edges must be an independent set of size k and vice-versa.

3. (★★★ level) Reduction, Reduction, and Reduction

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 $A' \subseteq A$ of her bills for 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: We can reduce SUBSET SUM to this problem. Given a weight capacity W and the item weights $w_1, \ldots, w_n$, we convert to an instance of this problem as follows:

Alice has the set of paper bills $\{w_1, \ldots, w_n\}$ and owes W dollars. Her date is an imaginary Bob, who naturally has no bills and eats nothing.

Now Alice and Bob can't trade, and Alice must pay the exact amount W with $\{w_1, \ldots, w_n\}$, which is precisely the SUBSET SUM problem.

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 accounts 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 bank accounts that you withdraw from.

   Show that PUBLIC FUNDS is **NP**-Complete.

   Solution: We can trivially verify a solution for PUBLIC FUNDS in polynomial time by verifying that the sum of the balances of the chosen bank accounts adds up exactly to the price of the fence, therefore it is in NP.

   To show that it is in NP-Hard, we reduce from SUBSET SUM. Let each of the m elements in the set of numbers correspond to a bank account's balance b_m. Let the desired sum s correspond to the price of the only fence c_1. Also, set the maximum number of bank accounts used, k, to the number of elements in the set of numbers m. If an algorithm for PUBLIC FUNDS finds a solution, we can interpret the bank accounts chosen as a solution for SUBSET SUM. Furthermore, all instances of SUBSET SUM correspond to a valid instance of PUBLIC FUNDS by construction.

   One common mistake was to reduce in the wrong direction. To show that PUBLIC FUNDS is NP-Hard, we need to show that it can be used to solve a known NP-Complete problem (with polynomial time modifications), not the other way around.

   Another common mistake was to reduce from KNAPSACK WITHOUT REPETITIONS. All NP-Complete problems can theoretically be reduced to each other (through a chain of reductions), but direct reductions from KNAPSACK WITHOUT REPETITIONS don't have a way to handle the weights of the items. Furthermore, you're not allowed to modify problem you're reducing from; some people attempted to modify knapsack to force the weights to be equivalent to the values of the item.

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: We reduce from MAX-INDEPENDENT-SET. Let (H, k) be an instance of MAX-INDEPENDENT-SET, where H is an undirected graph. Suppose $H = (V, E)$. We construct a directed graph $H = (V, E')$ as follows. For each edge $\{u, v\}$ in H, we include in E the pair of directed edges (u, v) and (v, u). We then return the instance (H, k) of MAX-ACYCLIC-INDUCED-SUBGRAPH. Given a solution S to this instance, we simply return S as a solution to the original MAX-INDEPENDENT-SET instance. Thus, our reduction is clearly polynomial time.

We now consider correctness. Suppose there was an independent set I of size k in H. Then, there are no edges between vertices in I in H, and hence, by construction, the induced subgraph on I in G is completely disconnected and trivially a DAG. Thus, there is an induced DAG of size k in G.

Now suppose that G has an induced DAG G_S of size k. If $u \in S$ and $v \in S$ such that $\{u,v\} \in E$, then both the edges (u,v) and (v,u) are present in G_S, and hence it is not a DAG. Thus, we see that S is an independent set in H.

4. (★★★★ level) Graphs and Matrix Multiplication.

Consider the MATRIX MULTIPLICATION problem:

Input: Two $n \times n$ matries 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 compute M^p in time $O(t(n) \log p)$. You only need to prove a main idea and show how to satisfy the time complexity requirement.

Solution: Use repeated squaring.

(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 $O(t(n) \log n)$, 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: Let A be the adjacency matrix of G with 1 on the diagonal: so for all nodes i, $A_{ii} = 1$, and whenever $i \neq j$, $A_{ij} = 1$ if there is an edge between nodes i and j, and $A_{ij} = 0$ otherwise.

Fact: For any integer $k \geq 0$, $(A^k)_{ij}$ is the number of lazy paths from i to j which have length k. A lazy path is a sequence of vertices $i = v_1, v_2, \ldots, v_k = j$ such that at every step ℓ, either $v_\ell = v_{\ell+1}$, or v_ℓ and $v_{\ell+1}$ are connected by an edge.

Based on the fact, our algorithm is simple: compute A^n, where n is the number of vertices, and check whether $(A^n)_{st} > 0$. (There is a lazy path from s to t iff there is a simple path.)

We can prove the fact by induction on k. For nodes i and j and integer $k \geq 0$, let $P(i,j,k)$ be the number of lazy paths of length k from i to j: we wish to show $P(i,j,k) = (A^k)_{ij}$.

A^0 is the identity matrix, so the base case is only saying that there is a length-zero path from i to, j iff $i = j$.

For the induction step, assume that for all nodes i and j, $(A^k)_{ij} = P(i,j,k)$. Now, every lazy path of length $k+1$ from i to j consists of a length-k lazy path followed by one more step. So $P(i,j,k+1)$ equals the sum over every node connected to j, including j itself, of $P(i,j,k)$:

$$P(i,j,k+1) = \sum_{\ell=1}^{n} A_{\ell j} P(i,\ell,k).$$

By the induction hypothesis, the right-hand side is equal to $\sum_{\ell=1}^{n} (A^k)_{i\ell} A_{\ell j}$. Does it look familiar? That's the formula for multiplying the matrices A^k and A! So we've just proved that $P(i,j,k+1) = (A^{k+1})_{ij}$, and our induction argument is complete.

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

Solution: By part (b), if MATRIX MULTIPLICATION could be solved in time $O(T(n))$, then graph reachability could be solved in time $O(T(n)\log n)$.

5. (★★★★★ level) Finding Zero(s)

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 known *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:

Given a 3-SAT formula ϕ in the variables $x_1, x_2, \ldots, x_n$, our reduction f will produce a polynomial P in the same variables such that satisfying assignments correspond to $0, 1$ valued zeros of P. We can use $1 - x_i$ to denote $\neg x_i$ and multiplication to denote $\wedge$. However naïvely translating clauses using this method and multiplying can lead to a polynomial P with 3^m terms, where m is the number of clauses. Thus, we will instead add use the observation that $a^2 + b^2 = 0$ iff $a = b = 0$ when a, b are real to achieve this. We now proceed to describe f. Let the clauses of ϕ be $c_1, c_2, \ldots, c_m$ where $c_i = (y_{i1} \vee y_{i2} \vee y_{i3})$ where y_{ij} are literals. Corresponding to the ith clause we consider the polynomial $p_i(x_1, x_2, \ldots, x_n) := (1 - y_{i1})(1 - y_{i2})(1 - y_{i3})$. For example, corresponding to the clause $x_1 \vee \neg x_2 \vee x_3$, we will get the polynomial $(1 - x_1)x_2(1 - x_3)$. Notice that if x_i are $0, 1$ valued, then p_i is zero if any only if the ith clause is satisfied (interpreting 0 as FALSE and 1 as TRUE). We now need to ensure that x_i are $0, 1$ valued. To do this we need to ensure that the polynomials $q_i(x_1, x_2, \ldots, x_n) := x_i(1 - x_i)$ are all 0. Thus, we need to enforce that all the q_i and p_i polynomials are 0 at a zero of P. This is achieved by setting

$$P(x_1, x_2, \ldots, x_n) := \sum_{i=1}^{m} p_i(x_1, x_2, \ldots, x_n)^2 + \sum_{i=1}^{n} q_i(x_1, x_2, \ldots, x_n)^2.$$

By construction, p can be fully expanded in time polynomial in the size of the clause, and hence the reduction runs in polynomial time.

Now suppose ϕ has a satisfying assignment α. Setting $x_i = 1$ when $\alpha(x_i) = \text{TRUE}$ and $x_i = 0$ otherwise, we see that all the polynomials q_i constructed above are 0. Further, since α is a satisfying assignment, all the polynomials p_i are 0 too. Thus, P has the integral zero as constructed above.

[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.

Now suppose P has integral zeros: suppose $P(s_1, s_2, \ldots, s_n) = 0$ for integral s_i. This implies that all the p_i and q_i polynomials are 0. By the latter, we get $s_i \in \{0, 1\}$ for all i. Now setting $x_i = \text{TRUE}$ if $s_i = 1$ and $x_i = \text{FALSE}$ otherwise, we see that all the clauses of ϕ are satisfied since p_i's are 0. Thus, given an integral solution to P, we can construct in polynomial time a satisfying assignment to ϕ.

***Note**: A construction of P which has exponentially many terms should probably lose some points.*