TheoryPrep **(Due: 6/28/20)**

Assignment #6: Reductions

Name: Student name(s)

We've taken some problems/solutions from the course texts and CS 121 material. Some problems are marked as (Challenge) or (Additional Problems). Do the rest first, and if you have time, return to these problems. It's **much more important** to have a rigorous understanding of the core problems and how to prove them than to simply finish all the additional problems. This week's pset will have a few longer proofs.

Problem 1: Uncomputability Reductions

Learning goal: **This is the main problem on the pset.** It starts with some shorter questions to build intuition for uncomputability reductions and then has two longer problems that are designed to help you understand uncomputability reductions.

(a) Your friend wants to prove that some function F is uncomputable and they've heard that they can do something called "reducing from $HALT$." They've never seen a reduction before. Explain at a high level what they should do. Make sure to include the steps they need to do and what the proof implies.

Solution:
Not the original pset says "reducing **TO** HALT." You can tell students it should say "reducing **FROM** HALT.'

The overall argument they need to make is by supposing towards contradiction that F is computable and proving that $HALT$ is computable. This contradicts that $HALT$ is uncomputable, so it implies that F is not computable.

They should assume the existence of a black box program to solve F, call it P_F. Then they should use P_F to build some program P_H to compute $HALT$. It's good to break this up into two parts

1. Describe the construction of P_H using P_F.
2. Prove that P_H computes $HALT$.

(b) (Additional Problem) An uncomputability reduction could be considered an application of the contrapositive. Explain why this is true (think about the $HALT$, $HALTONZERO$ example).

Solution:
In an uncomputability reduction we want to prove that since F is uncomputable so is G. Applying the contrapositive to this gives if G is computable then F is computable, which is exactly the proof strategy we use.

(c) Let $EVENLENGTH : \{0,1\}^* \to \{0,1\}$ be the function that takes in a turing machine M and evaluates to 1 if and only if every input halts and gives an even-length output. Prove by reduction that $EVENLENGTH$ is uncomputable (we suggest you reduce from $HALT$ or $HALTONZERO$). You can use \begin{lstlisting} to write code.

Solution:
The student's paranthetical remarks says "(we suggest you reduce to $HALT$ or $HALTONZERO$)." It should read "(we suggest you reduce from $HALT$ or $HALTONZERO$)."

We will reduce $HALT$ to $EVENLENGTH$. Suppose that we have a program $EVENLENGTH\_oracle$ that computes $EVENLENGTH$. I claim that the following program computes $HALT$.

```
def compute_HALT(M,x):
    def Q(y):
        run M(x)
        return 00

    return EVENLENGTH_oracle(Q)
```

This program takes in a turing machine and input, defines another turing machine Q that disregards its input y, runs $M(x)$ and returns the string 00. We prove that $compute\_HALT(M,x) = HALT(M,x)$ by cases

1. If $M(x)$ halts then $Q(y)$ returns 00 (an even-length string) on every input and therefore $compute\_HALT(M,x) = EVENELENGTH\_oracle(Q) = 1$
2. If $M(x)$ does not halt then $Q(y)$ never halts on any inputs and therefore $compute\_HALT(M,x) = EVENELENGTH\_oracle(Q) = 0$

(d) Prove Rice's Theorem. To review, Rice's Theorem says that any $F : \{0,1\}^* \to \{0,1\}$ that is semantic and nontrivial is uncomputable.

Solution:
We will reduce from $HALT$. Fix arbitrary nontrivial semantic function F. Suppose towards a contradiction that F is computable by F_oracle.

Let M be a turing machine such that on any input it does not halt. $F(M) = b$, which is either 0 or 1, so let M' be a turing machine such that $F(M') = 1 - b$, giving the opposite result (we know such a turing machine must exist since F is nontrivial). I claim the following program computes $HALT$.

```
def compute_HALT(T,x):
    def Q(y):
        run T(x)
        return M'(y)

    if b == 1:
        return 1- F_oracle(Q)
    else:
        return F_oracle(Q)
```

To summarize, upon input T, x the program creates a new turing machine that runs $T(x)$ and if that halts then returns $M'(y)$. It then applies the oracle for F. We now prove that $compute\_HALT(x) = HALT(x)$. We can do this with cases

1. If $T(x)$ halts, then $Q(y)$ just returns $M'(y)$ on every input. Thus, Q and M' are functionally equivalent, so since F is semantic, $F\_oracle(Q) = F\_oracle(M') = 1 - b$. Thus, if $b = 1$ then we get back $1 - (1 - b) = b = 1$ and if $b = 0$ then we get back $1 - b = 1 - 0 = 1$. Thus our program gives the correct result.
2. If $T(x)$ does not halt, then $Q(y)$ is just the program that never halts. That means it's functionally equivalent to M, so $F\_oracle(Q) = F\_oracle(M) = b$. If $b = 1$ then the program returns $1 - b = 0$ and if $b = 0$ then the program returns $b = 0$. Thus, we return the correct value.

Thus, if we could compute F we could compute $HALT$ so F must be uncomputable. We chose F arbitrarily so this holds for any F semantic and nontrivial.

Problem 2: Polynomial Time Reductions (Additional Problem)

Learning goal: This section has some brief questions to help you build intuition for polynomial time reductions and then some reduction problems. This entire section is optional, and you should prioritize building a strong grasp of uncomputability reductions in the previous section first.

Solution:
Polynomial time reductions are often pretty difficult for students because they feel that you have to have some "spark of ingenuity" to come up with the reduction. However, I think by giving students more structure and helping them think about the proof techniques, we can hopefully make the process seem more directed. I've listed a couple of tips I've found

1. Students shouldn't be afraid to try things with no idea if they will work–in fact, I think often the only way to build intuition for the structure of problems in a reduction is to try small examples and a diversity of transformations. Encouraging students to draw examples and experiment can be super helpful!
2. It is often difficult to come up with reductions, and students often feel they don't even know how to approach the problem. If this happens, a helpful way of framing the problem is thinking about what you're going to use the reduction for–you're likely going to use it to prove $F(x) = 1 \Leftrightarrow G(R(x)) = 1$ which we usually prove by transforming a solution for F into a solution for G and vice versa. Thus, it can be helpful to think about "if I had a solution to $F(x)$, then how would I make that into a solution for $G(R(x))$?" This can help students think about what transformations are helpful–I've tried to put some intuition along this "transforming solutions" for each of the reductions below.

(a) (Additional Problem) Suppose that for functions $F, G : \{0,1\}^* \to \{0,1\}$, F reduces to G. Complete each of the following statements to form the strongest true statements (a statement A is stronger than B if $A \Rightarrow B$–another way to think about the strongest statement is that it's the most you can assert).

1. If G is polynomially time computable, then F
2. If F cannot be computed in polynomial time, then G
3. We'd say that F is (easier/weakly easier/equal/weakly harder/harder) compared to G.
4. If F cannot be computed in exponential time, then G
5. If G can be computed in linear time, then F

Solution:

1. If G is polynomially time computable then F is polynomially time computable.
2. If F cannot be computed in polynomial time, then G cannot either (this is the contrapositive of previous statement).
3. We say that F is weakly easier than G (since if we can solve G we can solve F–weakly is because they could both reduce to each other like $HALT$ and $HALTONZERO$).
4. If F cannot be computed in exponential time, then G can also be computed in exponential time.
5. If G can be computed in linear time, then F can be computed in **polynomial**

time (the important takeaway from this problem is that we can only say that F is computable in polynomial time since the reduction may not also be linear).

(b) (Additional Problem) In 121, we think any algorithm that runs in polynomial time is efficient. Thus, we wanted our definition of a polynomial time reduction to satisfy "F reduces to G implies that if G is polynomially time computable then so is F" (which it does by Solved Exercise 13.1 on page 15 of the reading). Why does R need to be polynomially time computable in order for this property to hold? An intuitive explanation will suffice (this question exemplifies how to interrogate a definition–like we learned last week).

Solution:
Intuitively, if we allow R to have a large runtime, then even if P_G computes G in polynomial time, we can't necessarily use $P_G(P_R(x))$ to efficiently compute F because P_R may take exponential or longer. This would be a sufficient explanation on the part of the student.

Intuitively, if we allow R to have any runtime, R can just solve the problem and return a trivial true or false example to G. This would imply that we could reduce any computable problem to any other computable problem. As an example of this, consider the function $IDENTITY : \{0,1\}^* \to \{0,1\}^*$ that is the identity function. This is clearly a linear time function, but with no restriction on R we could reduce any computable function F (even one requiring exponential time–2^n) to IDENTITY. The reduction $R(x)$ would just compute the function $F(x)$ and return $F(x)$, so $IDENTITY(R(x)) = IDENTITY(F(x)) = F(x)$.

Note that if we remove the polynomial-time restriction on R we basically end up with the definition of an uncomputability reduction.

(c) (Additional Problem) Now to do a reduction. In the readings/video we talked about the 3SAT problem, but we might wonder what happens if we have 4 variables in each clause. This motivates the 4SAT problem. $4SAT : \{0,1\}^* \to \{0,1\}$ takes in the encoding of a 4CNF (like $(x_0 \vee \bar{x_2} \vee x_3 \vee \bar{x_4}) \wedge (\bar{x_0} \vee x_1 \vee \bar{x_2} \vee x_3)$...) and returns 1 if and only if it is satisfiable. You'll prove that interestingly, even though we added more variables, it doesn't become more difficult; prove that $4SAT$ reduces to $3SAT$ (it may be helpful to think of the 01EQ, 3SAT reduction in the reading and how it used slack variables).

Solution:
Instructor Note: Following the "transforming solutions" paradigm for the below problem, you're going to want to take in a 4CNF and create a 3CNF such that if you have a solution to the 4CNF, you can create a solution to the 3CNF. This can lead students to the conclusion that they need to create an analogue for each clause in the 4CNF using 3CNF formulas. They can then use a slack variable to "bind together" two 3CNF clauses to mimic each 4CNF clause. This will enable them to basically use the same solution for each CNF, providing a convenient way to transform the solution back and forth.

We want to define a function R that takes in a 4CNF and turns it into a 3CNF. For each clause of the form $(y \vee y' \vee y'' \vee y''')$ we will put two clauses in the resulting 3CNF–$(y \vee y' \vee z) \wedge (y'' \vee y''' \vee \bar{z})$ (using a different label z for each clause). The core idea is that z will be 1 or 0 which will make one of the clauses true, and the other is true since the 4CNF is. This is a polynomial time reduction, since we are taking linear time to scan through the clauses and do a constant time operation on each of them. We now prove $4SAT(x) = 3SAT(R(x))$.

($4SAT(x) = 1 \Rightarrow 3SAT(R(x)) = 1$) Suppose there is a satisfying assignment for the initial 4CNF. Consider an assignment of the 3CNF variables which is the same as the 4CNF variables (we will explain how to set the zs soon). Since each clause of the 4CNF $(y_1 \vee y_2 \vee y_3 \vee y_4)$ is true, there is some y_i which is true. This y_i makes one of the two 3CNF clauses $(y_1 \vee y_2 \vee z), (y_3 \vee y_4 \vee \bar{z})$ true, and the other we can make true by setting z to 0 or 1 respectively. This implies there is a satisfying assignment for the transformed CNF, so $3SAT(R(x)) = 1$

($3SAT(R(x)) = 1 \Rightarrow 4SAT(x) = 1$) Suppose there is a satisfying assignment for the 3CNF. I claim taking the same assignment (without the zs) satisfies the 4CNF. For any clause $(y_1 \vee y_2 \vee y_3 \vee y_4)$ in the 4CNF, there are two clauses $(y_1 \vee y_2 \vee z), (y_3 \vee y_4 \vee \bar{z})$ in the 3CNF. If $z = 1$ then either y_3 or y_4 must be true, so $(y_1 \vee y_2 \vee y_3 \vee y_4)$ is true. If $z = 0$ then y_1 or y_2 must be true, so $(y_1 \vee y_2 \vee y_3 \vee y_4)$ is true. Thus, we see that the 4CNF has a valid solution, so $4SAT(x) = 1$.

As a side note, you can ask your students if 4SAT and 3SAT are equivalent in difficulty (ie can you reduce $3SAT$ to $4SAT$). You can by modifying each $(y \vee y' \vee y'')$ clause to be of the form $(y \vee y' \vee y'' \vee z)$ and adding a clause of the form $(\bar{z} \vee \bar{z} \vee \bar{z} \vee \bar{z})$ (since the $z = 0$ by the last clause so it's just solving the initial 3CNF).

(d) (Additional Problem) The function $VERTEXCOVER : \{0,1\}^* \to \{0,1\}$ on input a graph G and number k returns 1 if and only if there is a vertex cover of size at most k of the graph. A vertex cover is a set S of vertices such that every edge has at least one endpoint in S. $HALFCOVER : \{0,1\}^* \to \{0,1\}$ also takes in a graph G and number k and returns 1 if and only if there is a set of vertices S of size at most k such that at least half the edges have an endpoint in S. It seems that HALFCOVER should be an easier problem , but your task is to prove that VERTEXCOVER reduces to HALFCOVER.

Solution:
Instructor Note: Following the "transforming solutions" paradigm for the below problem, you're going to want to take in a graph G and create a graph G' such that if you have a vertex cover in G, you can create a half cover in G' (and vice versa). It's often difficult to think of a transformation that immediately allows us to translate solutions in both directions easily, so a tip for students is to **start by trying to create a transformation where they can prove one direction and then modify**. The vertex cover for $G \Rightarrow$ half cover for G' seems easier, and a good insight is that if we add exactly $|E|$ edges to G get G', then a vertex cover for G is exactly the half cover that we need for G'! The relevant question is thus how to add those $|E|$ edges such that the backwards direction (trying to transform a half cover for G' into a vertex cover for G) works. Students should try some different ways of adding these edges to see some of the issues that may occur. For example, it's hard to find any sort of structure if you try to only add edges and no vertices, and this may inspire students to completely separate out the new edges by putting them on new vertices. From there, if we limit each of these new vertices to degree 1, we can transform a half cover in G' to a vertex cover in G by removing added vertices and adding relevant ones from the initial graph G.

First we define the reduction R. Upon input a graph $G = (V, E)$ it will add $|E|$ "dumbells" which are pairs of vertices connected and only connected to each other. Call the new graph G', the new vertices and edges "added", and the old vertices and edges "initial". We leave k unchanged. This is clearly a polynomial time algorithm, because assuming an adjacency-list representation (in 121 students are usually free to pick any type of representation form since they're the same up to polynomial time work), we can count the number of edges, and then add a dumbell (constant amount of time) for each edge. Now we prove $HALFCOVER(R(G,k)) = VERTEXCOVER(G,k)$.

($VERTEXCOVER(G,k) = 1 \Rightarrow HALFCOVER(R(G,k)) = 1$) If $VERTEXCOVER(G,k) = 1$ then there exists a set S of size at most k such that all edges in G are covered. We only added $|E|$ edges in the reduction R, so the same set of vertices V will cover $|E|$ edges in the reduced graph, which is at least half of them. Thus $HALFCOVER(R(G,k)) = 1$

($HALFCOVER(R(G,k)) = 1 \Rightarrow VERTEXCOVER(G,k) = 1$). Suppose that there is a set S of size at most k vertices such that at least half of the edges in G' are covered. I claim we can create a set S' of at most k vertices in G' that covers all the edges in G; this implies $VERTEXCOVER(G,k) = 1$ since we can just take the initial vertices from S', and these must still cover all the initial edges in G (because all endpoints of edges in G are initial vertices, so removing added vertices does not change coverage of initial edges). Thus, all we need to do is prove this result.

If the set S does not cover all the initial edges but still covers half the edges in

G', it must cover an added edge and therefore must have an added vertex (since we only added edges between added vertices). Remove an added vertex v' and add an initial vertex v that is the endpoint of a currently uncovered initial edge (if there are no currently uncovered initial edges, this means we've proven the result already). We lose at most one edge from removing v' (since it's on a dumbell), but we add at least one edge from adding v, so the total number of edges covered does not decrease, and the number of initial edges increases. Thus, we can keep applying this strategy to maintain a set covering at least $|E|$ edges and an increasing number of initial edges–this necessarily gives us a set S' covering all initial edges. Thus $VERTEXCOVER(G, k) = 1$.

(e) (Challenge Problem) Reduce 3SAT to VERTEXCOVER. It may help to think about the reduction from 3SAT to ISET from the video.

Solution:
Instructor Note: Whenever we reduce from a CNF to a graph, we usually want to make the vertices represent the literals. VERTEXCOVER uses edges to enforce restrictions, so that's a good place to start. If we connect vertices for literals $x_i, \bar{x_i}$ then this will force a vertex cover to cover at least one of these. We then want to incorporate the restriction that all clauses have at least one literal true. If we create a vertex for every literal in a clause and connect them to the original literal vertices, this will require all literals in the clause to be true because these edges need to be covered (so close but a little overrestrictive). Connecting the literals in each clause and changing the max size of our vertex cover can ease the restriction sufficiently.

First to define the reduction. We seek to create a graph representing the 3SAT equation such that it's satisfiable if and only if there is a vertex cover of size n + 2m, where n is the number of literals in the 3SAT equation and m is the number of clauses. Consider the following construction of such a graph. For each variable x in the formula, add 2 nodes x and $\bar{x}$ and connect them with an edge, kind of like a dumbbell. For each clause x_i, x_j, x_k add a new node that represents each of the literals in that clause, and connect those new nodes together in a triangle so that each literal is connected to the others in the same clause. Then, connect each of the new literal nodes to the original variable nodes. Below we've given an example of the resulting graph for a 3CNF

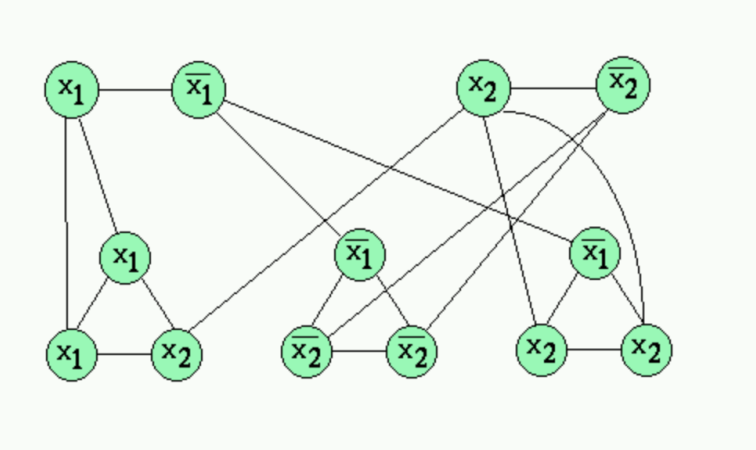

Figure 1: The graph for $(x_1 \vee x_1 \vee x_2) \wedge (\bar{x_1} \vee \bar{x_2} \vee \bar{x_2}) \wedge (\bar{x_1} \vee x_2 \vee x_2)$ source

We need to show that this reduction runs in polynomial time. We can scan the CNF to identify the literals, thus creating the variables nodes in linear time. We can then scan over the CNF and create the clause triangles, also taking linear amount of time. This means the overall algorithm R to create the graph is linear. We posit that this graph has a vertex cover of size $n + 2m$ if and only if the 3SAT expression it represents is satisfiable.

($\Leftarrow$) We see first that if there is a satisfiable solution, we can construct a vertex cover from it. We do so by selecting the n variable nodes ($\frac{1}{2}$ of each pair of the dumbbell nodes). This will cover each of the edges connecting the variable nodes, and 1 out of each of the three nodes connecting the variable nodes with the literal nodes (the cross edges between dumbbells and triangles). Now, to cover the remaining cross edges and the edges between literal nodes in each of the triangles, you will have to select up to 2 nodes for each of the clauses, which is up to $2m$ overall. Thus, there will be a vertex cover of at most $n + 2m$ nodes if the 3SAT expression is satisfiable.

($\Rightarrow$) To show the other direction, we see that given a vertex cover of size $n + 2m$, we must have an assignment that satisfies the expression. At least n of the vertices must be used to cover the literal nodes, since there are n edges between a variable and it's negation, so to cover those edges, at least one of those nodes must be covered. We also know that $2m$ vertices must cover the edges in each clause triangle because in order to cover all the 3 edges in each triangle, two of the vertices must be in the vertex cover. Since the literal nodes and the clause triangles are composed of disjoint (read different) vertices, and we can only have $n + 2m$ vertices in the vertex cover, it must have exactly n literal vertices and $2m$ clause triangle vertices. Assign the x for 3SAT based on the literal vertices that are in the vertex cover (note that we will not have conflicts because if some $x_i, \bar{x_i}$ were both in the vertex cover, this would mean that one of the n literal edges is uncovered since only n of the literal vertices are in the vertex cover). We see for any clause triangle, exactly 2 of the vertices are in the vertex cover (because the vertex cover only has a total of $2m$ clause vertex so if any clause triangle had more than 2 vertices in the vertex cover than another triangle would have fewer than 2 vertices in the vertex cover, in which case there would be an uncovered edge). Thus, there's some literal x_i in the clause triangle which is not in the vertex cover, and that literal has an edge to it's literal vertex, so that literal vertex must be in the vertex cover. Thus, we know that by assignment this literal in the clause is satisfied. This applies to every clause, so all clauses are satisfied.

Problem 3: Review

Learning goal: The goal for this section is to reflect on how you can improve your proof skills! We're probably going to use this format for Problem 4 for the rest of the course since reflecting and fixing past proofs is a great exercise–if you'd prefer more review problems from previous weeks please let us know (in a response to Problem 5 or on the Piazza).

(a) Select the problem from a previous week that you think you would learn the most from redoing. Copy the problem and your instructor's feedback below, rewrite the proof, and add a short reflection on the changes you made.

(b) (Additional Question) Choose another problem you received feedback on and copy the problem and feedback you received, rewrite the proof, and write a sentence about why you made your changes (so do Part A again) OR solve a problem you left blank from a previous week (try to pick from a week you felt least secure on).

Problem 5: Logistics

Purpose: This helps us make sure the course is going at the right speed!

(a) How long did you spend on the videos and readings this week?

(b) How long (including time in problem sessions) did you spend on this problem set?

(c) Do you have any feedback about the course in general (did the videos and readings sufficiently prepare you for the problem set)?