Computational Complexity Theory **Summer HSSP 2018**

Week 3: Reductions and Completeness

Dylan Hendrickson *MIT Educational Studies Program*

3.1 Reductions

Suppose I know how to solve some problem quickly. How can I use this ability to solve other problems quickly? A reduction is one way to do this.

Definition 3.1. Let A and B be decision problems. A *reduction* f from A to B is a Turing machine which takes as input instances of A, and outputs instances of B, such that the answers to the input and output questions are the same. That is, for any instance x of A, $x \in A$ iff $f(x) \in B$.

If there is a reduction from A to B, we say A is *reducible* to B, and write $A \leq_m B$.

This kind of reduction is called a mapping reduction or many-one reduction, to contrast with other types of reductions such as oracles (which we probably won't discuss).

If we know how to solve B, and have a reduction from A to B, we can solve A. On the other hand, if you can solve A, the reduction from A to B doesn't help you solve B. So you can think of B being 'at least as hard' to solve as A.

Lemma 3.2. *If $A \leq_m B$ and B is decidable, then A is decidable. If $A \leq_m B$ and B is recognizable, then A is recognizable.*

Exercise 1. Prove lemma 3.2.

Lemma 3.3. *$\leq_m$ is a* preorder, *meaning it satisfies these properties:*

- *$\leq_m$ is reflexive: for any decision problem A, $A \leq_m A$.*
- *$\leq_m$ is transitive: for any decision problems A, B, and C, if $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$.*

Exercise 2. Prove lemma 3.3

The *Post Correspondence Problem* (PCP) is the following decision problem: given a collection of 'dominoes,' each of which has a string of symbols on each side, is there a nonempty sequence of these dominoes such that the total top string and bottom string are the same?

For example, we might have these dominoes:

$$\begin{bmatrix} b \\ ca \end{bmatrix}, \begin{bmatrix} a \\ ab \end{bmatrix}, \begin{bmatrix} ca \\ a \end{bmatrix}, \begin{bmatrix} abc \\ c \end{bmatrix}.$$

One valid sequence of dominoes is

$$\begin{bmatrix} a \\ ab \end{bmatrix} \begin{bmatrix} b \\ ca \end{bmatrix} \begin{bmatrix} ca \\ a \end{bmatrix} \begin{bmatrix} a \\ ab \end{bmatrix} \begin{bmatrix} abc \\ c \end{bmatrix}.$$

The strings on the top and bottom are both $abcaaabc$.

A few notes: the empty sequence, which has the empty string on the top and bottom, doesn't count. You can dominoes multiple times. You can't flip dominoes over or anything like that; they have to be upright. There may be multiple valid sequences (in fact, if there's one, there are infinitely many); the question is whether there are any.

Theorem 3.4. *PCP is undecidable.*

Proof. We give a reduction from $HALT$ to PCP, showing $HALT \leq_m PCP$. Since $HALT$ is undecidable, by lemma 3.2, PCP is undecidable. We actually use a slight variation on PCP, where one domino is designated as the start domino (it's possible to reduce this version of PCP to the original).

Given a Turing machine and input, we want to find a collection of dominoes that only have a solution if the Turing machine halts. We'll do this by making a successful sequence of dominoes correspond to the computation history of the Turing machine. The winning string on the dominoes will have the form

$$\#—C_1—\#—C_2—\# \cdots \#—C_f—\#$$

where C_i is the configuration of the Turing machine at the ith time step, and $\#$ is a symbol to separate them. Each configuration is a sequence of tape symbols, with one symbol indicating the Turing machine's head. For example, $01q_40010-10-001$ would indicate that the tap has $010010-10-001$, and the Turing machine is in state q_4 looking at the third cell of the tape (containing a 0).

We'll always have the bottom row of the dominoes one configuration ahead of the top row. In order to match the bottom row, we have to pick dominoes with the correct top row, with the bottom row of these dominoes designed to produced the next configuration.

Let's start building dominoes. The first configuration has the Turing machine it its start state looking at the left end of the input, so we'll make the start domino do that, assuming the start state is q_0 and the input is $w_1w_2\dots w_n$:

$$\begin{bmatrix} \# \\ \#q_0w_1\dots w_n\# \end{bmatrix}$$

Since most of the tape doesn't change each step, we want dominoes to copy each symbol that can be on the tape:

$$\begin{bmatrix} 0 \\ 0 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \end{bmatrix}, \begin{bmatrix} - \\ - \end{bmatrix}$$

Now we want to simulate the Turing machine head. Suppose it's in state q and reads a on the tape. If it writes b, moves right, and switches to state q', we put a domino

$$\begin{bmatrix} qa \\ bq' \end{bmatrix}$$

If it writes b, moves left, and switches to q', then for each symbol c, we put a domino

$$\begin{bmatrix} cqa \\ q'cb \end{bmatrix}$$

For separations between configurations, we want to be able to add extra blanks to extend the tape, so we have the dominoes

$$\begin{bmatrix} \# \\ \# \end{bmatrix}, \begin{bmatrix} \# \\ -\# \end{bmatrix}, \begin{bmatrix} \# \\ \#- \end{bmatrix}$$

So far, these dominoes will simulate the Turing machine until it halts. Once it halts, we want to get the top and bottom strings to match. We do this by having the halt state 'eat' the entire tape, so for each symbol a we have:

$$\begin{bmatrix} aq_{halt} \\ q_{halt} \end{bmatrix}, \begin{bmatrix} q_{halt}a \\ q_{halt} \end{bmatrix}$$

After the Turing machine halts, this gives us a sequence of configurations with the tape shrinking by one cell each time. At the end of this, the final configuration is just q_{halt}. So the domino strings look like

$$\begin{matrix} \cdots \# \\ \cdots \# q_{halt} \# \end{matrix}$$

We add one more domino

$$\begin{bmatrix} q_{halt}\#\# \\ \# \end{bmatrix}$$

Putting this domino on the end makes the strings match.

If the Turing machine halts, this collection of dominoes will have a matching sequence. If the Turing machine doesn't halt, trying to build a matching sequence will take forever, as we continue to simulate the Turing machine forever. So this is a reduction from $HALT$ to PCP, and $HALT \leq_m PCP$. By lemma 3.2, if PCP is decidable then $HALT$ is decidable; since we know $HALT$ is undecidable, PCP is undecidable. □

Exercise 3. Suppose A and B are both decidable. When is $A \leq_m B$?

3.1.1 Polynomial-time reductions

Since we care about what we can do quickly, not just what we can do at all, we'll usually restrict ourselves to polynomial-time reductions (the are other restrictions, such as log-space reductions, but they aren't as useful). A polynomial-time reduction is simply a reduction that runs in polynomial time (in the input length). If there is a polynomial-time reduction from A to B, we write $A \leq_P B$. For example, the reduction from $HALT$ to PCP takes only polynomial time, so $HALT \leq_P PCP$.

If we have a polynomial-time reduction from A to B, and we can solve B easily, then we can solve A easily.

Lemma 3.5. *Suppose $A \leq_P B$.*

- *If $B \in \mathbf{P}$, then $A \in \mathbf{P}$*
- *If $B \in \mathbf{NP}$, then $A \in \mathbf{NP}$*
- *If $B \in \mathbf{coNP}$, then $A \in \mathbf{coNP}$*
- *If $B \in \mathbf{PSPACE}$, then $A \in \mathbf{PSPACE}$*

Exercise 4. Prove lemma 3.5.

Lemma 3.6. *$\leq_P$ is a preorder (it satisfies the properties in lemma 3.3).*

Exercise 5. Prove lemma 3.6.

Exercise 6. Suppose $A, B \in \mathbf{P}$. When is $A \leq_P B$?

3.2 Completeness

We now have a way to compare the relative difficulty of decision problems. This raises the natural question of what the hardest problems in **NP** or **PSPACE** are. It's not obvious that there exist 'hardest' problems; maybe there are two hard problems in **NP**, neither of which is reducible to the other. It turns out that many (but not all, e.g. **PH**) complexity classes have maximally difficult problems.

Definition 3.7. Let $\mathbf{C}$ be a complexity class and A be a decision problem. We say that A is *hard for* $\mathbf{C}$, or $\mathbf{C}$-*hard* if for every problem $B \in \mathbf{C}$, $B \leq_P A$. If A is $\mathbf{C}$-hard and $A \in \mathbf{C}$, we say that A is $\mathbf{C}$-*complete.*

Technically, this is only *complete under polynomial-time reductions.* You might care about completeness under arbitrary reductions, or under log-space reductions, etc.

Theorem 3.8. *$HALT$ is* $\mathbf{RE}$*-complete.*

Proof. We know $HALT \in \mathbf{RE}$. To show that $HALT$ is $\mathbf{RE}$-hard, let $A \in \mathbf{RE}$, and let M be a Turing machine which recognizes A. We show $A \leq_P HALT$ by giving a polynomial-time reduction from A to $HALT$.

Construct a new Turing machine M' which is the same as M except that whenever M would reject, M' enters an infinite loop. Given an instance x of A, we construct the instance (M', x) of $HALT$. This clearly takes only polynomial time.

For this to be a reduction, we need $x \in A$ exactly when $(M', x) \in HALT$. If $x \in A$, then M accepts x, so M' also accepts and thus halts on x. If $x \notin A$, then M either rejects or loops on x, so M' doesn't halt on x. Hence this is a reduction from A to $HALT$, completing the proof. □

Corollary 3.9. *PCP is* $\mathbf{RE}$*-complete.*

Proof. Since it's easy to check whether a sequence of dominoes works, we can use it as a certificate to verify PCP. Thus $PCP \in \mathbf{RE}$. Let $A \in \mathbf{RE}$. We know that $A \leq_P HALT$ and $HALT \leq_P PCP$, so $A \leq_P PCP$. Hence PCP is $\mathbf{RE}$-hard. □

Lemma 3.10. *If $A \leq_P B$ and A is* $\mathbf{C}$*-hard, then B is* $\mathbf{C}$*-hard.*

Exercise 7. Prove lemma 3.10.

Exercise 8. Which problems are $\mathbf{P}$-complete?

Lemma 3.10 lets us show lots of problems complete for a complexity class once we have one such problem. So we want to find complete problems for the complexity classes we care about.

Definition 3.11. A *boolean formula* is a collection of variables joined with & (and), $\vee$ (or), and $\neg$ (not), such as $(a \vee b)$ & $(\neg a \vee \neg b)$.

A *satisfying assignment* of a boolean formula is an assignment of 'true' (T) or 'false' (F) to each variable that makes the formula true. The formula above has the two satisfying assignments $a = T, b = F$ and $a = F, b = T$.

SAT is the following decision problem: given a boolean formula, does it have a satisfying assignment?

Theorem 3.12 (Cook-Levin Theorem)**.** *SAT is* $\mathbf{NP}$*-complete.*

Proof. It's easy to check whether an assignment of the variables makes a formula true, so SAT is in $\mathbf{NP}$. Showing that SAT is $\mathbf{NP}$-hard is more complicated. Let $A \in \mathbf{NP}$, and let M be a Turing machine which verifies A; i.e. it takes in pairs (w, c) where w is an instance of A, and $w \in A$ iff there is a c such that $M(w, c)$ accepts. We'll find a complicated boolean formula that simulates M so that it has a satisfying assignment if and only if $w \in A$.

Consider a table showing M's computation; each row is one timestep, and each column is one cell of the tape. Since M runs in polynomial time, say $\sim n^k$, the table has size $\sim n^{2k}$; big but polynomial. Our formula will have a variable x_{ijs} for each row i, column j, and symbol s, representing whether cell j contains s and time i. It will also have a variable y_{ijq} for each row i, column j, and state q, representing whether the Turing machine is in state q looking at cell j at time i.

Now we make a bunch of subformulas, and combine them with ANDs.

- The first row is the initial configuration, so we take y_{11q_0}, and x_{1jw_j} for each j, where w_j is the jth symbol in the input. We allow the portion of the first row corresponding to the certificate c to be anything.
- For each i, j, and symbols $r \neq s$, $\neg x_{ijr} \vee \neg x_{ijs}$. For each row i, columns j and k, and states p and q, $\neg y_{ijp} \vee \neg y_{ikq}$ unless $j = k$ and $p = q$. This enforces that each cell only has one symbol at a time, and the head of the Turing machine is in only one place and state at a time.
- $y_{i_f 1 q_{acc}} \vee y_{i_f j_f q_{acc}}$, where i_f is the last row of the table and the middle subscript ranges through all cells. This makes sure that the Turing machine accepts.
- For each i, j, and s, we have $\neg x_{ijs} \vee x_{(i+1)js} \vee y_{ijq_0} \vee \cdots \vee y_{ijq_m}$, with the final subscripts on y ranging through all states. This makes the tape not change between timesteps away from the Turing machine head.
- For each i, j, q, and s, we have $\neg(x_{ijs} \ \& \ y_{ijq}) \vee (x_{ijr} \ \& \ y_{(i+1)kp})$, where r, k, and p depend on what the Turing machine does when it reads s while it state q. r is the symbol it writes, p is the state it switches to, and k is $j + 1$ or $j - 1$ if it moves right or left, respectively. This simulates the Turing machine's head, by forcing a new head position and symbol when the head is at position j, state q, reading s.

A satisfying assignment of the resulting formula corresponds to an accepting computation history of M on (w, c) for some value of c. This the formula has a satisfying assigment if and only if there is a c that makes M accept, which is true exactly when $w \in A$. So this is a reduction from A to SAT.

Each portion of the formula is polynomially long, and there are polynomially many portians, each of which is trivial to construct. So we construct the formula in polynomial time. Thus $A \leq_P SAT$, completing the proof that SAT is **NP**-hard. □

Now that we have an **NP**-complete problem, we can use lemma 3.10 to show more problems **NP**-complete. To begin, there is a variant of SAT that's easier to work with

Definition 3.13. A *literal* is a variable or its negation. A *clause* is an OR of variables. A formula is in *conjunctive normal form (cnf)* if it is an AND of clauses. A formula is *3cnf* if it is cnf and each clause has at most three variables.

$3SAT$ is the following decision problem: given a 3cnf boolean formula, does it have a satisfying assignment?

Exercise 9. Show that $3SAT$ is **NP**-complete. You can do this either by giving a reduction from SAT, or by modifying the proof that SAT is **NP**-complete so that the formula that comes out is 3cnf.

Having the structure of clauses makes it easier to reduce $3SAT$ to other decision problems. For example, $VertexCover$ is the following decision problem: given a graph G and a number k, is there a set of k vertices in G such that every vertex is either in the set or connected by an edge to a vertex in the set?

Lemma 3.14. *$VertexCover$ is* **NP***-complete.*

Proof. The set of k vertices serves as a certificate, so $VertexCover \in \mathbf{NP}$. To show that it's **NP**-hard, we provide a reduction from $3SAT$ (using lemma 3.10).

Given a 3cnf formula ϕ, we construct a graph G and number k such that G has a vertex cover of size k iff ϕ is satisfiable. For each variable x of ϕ, there are there vertices x, $\neg x$, and x_0, all connected. For each clause C of ϕ, there is a vertex C connected to the vertices labelled with literals in C. Set k to be the number of variables of ϕ.

Let's see whether $(G, k) \in VertexCover$. For each variable x, to satisfy x_0, we need to pick one of x_0, x, or $\neg x$. Since there are k variables, that's all the vertices we get. x_0 isn't connected to anything other than x and $\neg x$, so we might as well pick either x or $\neg x$. This is equivalent to picking an assigment for ϕ. The set of vertices we pick works of all the clauses get hit: that is, for each clause vertex, we need to pick at least one of the vertices connected to it. This is equivalent to the assigment satisfying that clause. So $(G, k) \in VertexCover$ iff $\phi \in 3SAT$. Since the construction of G and k takes only polynomial time, $3SAT \leq_P VertexCover$, so $VertexCover$ is **NP**-hard. □

As you can see, it's much easier to prove problems **NP**-complete once we know SAT is **NP**-complete. As we prove more problems **NP**-complete, we get more options for further such proofs. We could give a reduction from $VertexCover$ instead of from $3SAT$ to prove another problem **NP**-hard, if we thought it would be easier.

NP-completeness relates to the **P** vs **NP** problem.

Lemma 3.15. *Let A be* **NP***-complete. Then $A \in P$ if and only if* $\mathbf{P} = \mathbf{NP}$.

Exercise 10. Prove lemma 3.15.

Exercise 11. If $\mathbf{P} = \mathbf{NP}$, which problems are **NP**-complete?

Exercise 12. Pick one (or more) of the following decision problems, and show it's **NP**-complete.

- $3COLOR$: given a graph G, can you color each vertex of G with one of three colors so that no two vertices of the same color are connected by an edge?
- $CLIQUE$: given a graph G and number k, is there a set of k vertices of G that are all connected by edges (called a k-clique)?
- $HAMPATH$: given a directed graph G, is there a path through G that visits every vertex exactly once?
- $SUBSETSUM$: given a numbers n_i and a number k, can you write k as a sum of some n_i? (You can't reuse n_i. We care about polynomial in the *length* of the input, not in the *value* of the input, which can be much larger. The length of a number is the number of digits it has.)

Exercise 13. Find some examples of **coNP**-complete problems.

Exercise 14. Pick your favorite logic puzzle (Sudoku, Nurikabe, Slitherlink, etc.). Is it **NP**-complete? You can assume that all of the problems in exercise 12 are **NP**-complete.