Notes on Computer Theory Last updated: September, 2015

More $\mathcal{NP}$ reductions, co$\mathcal{NP}$

Dov Gordon, taken from Sipser [1], and Jonathan Katz

1 More NP reductions

1.1 $SAT \leq_p 3SAT$

We know that we can convert any formula ϕ into conjunctive normal form by using basic laws of Boolean algebra. However, we have to ensure that there is an *efficient* reduction, and, in general, the trivial mapping might blow up the number of clauses at an exponential rate. For example, consider $(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee \cdots \vee (x_n \wedge y_n)$. It is easy to see that this is equivalent to $(x_1 \vee x_2 \vee \cdots \vee x_n) \wedge (y_1 \vee x_2 \vee \cdots \vee x_n) \wedge (x_1 \vee y_2 \vee \cdots \vee x_n) \wedge (y_1 \vee y_2 \vee x_3 \vee \cdots \vee x_n) \wedge \cdots$. This will have 2^n clauses, though, so it is not a valid Karp reduction! Instead, we give a good Karp reduction, which can be found in pages 2-4 of these lecture notes.

1.2 $3SAT \leq_p SUBSET\text{-}SUM$ (Sipser, Chapter 7 [1])

The language $SUBSET\text{-}SUM = \{(S,t) \mid t \in \mathcal{N}, S \subseteq \mathcal{N} \text{ is a multi-set}, \exists Y \subset S, \text{s.t. } \sum_{y_i \in Y} y_i = t\}$ We show a reduction from $3SAT$ by constructing the set S and the number t based on the formula ϕ over variables $x_1, \ldots, x_\ell$ and clauses $c_1, \ldots, c_n$. For each variable $x_i \in \phi$, we create two numbers $y_i^{(1)}, y_i^{(2)} \in S$, each with 2 parts: the left part (more significant digits) are identical, having a single 1, followed by $\ell - i$ 0s. The right hand side of $y_i^{(1)}$ is n digits long, and has a 1 in the jth place iff x_i is found in clause c_j. $y_i^{(2)}$ is defined similarly, but has a 1 in the jth place iff $\overline{x}_i$ is found in clause c_j. Every other digit is set to 0. In addition, for each clause c_j, we add two more numbers to S: $y_j^{(3)}, y_i^{(4)}$, each identical to one another, and each containing a 1, followed by $n - j$ 0s. As an example, imagine we have

$$\phi = (x_1 \vee \overline{x_2} \vee x_3) \wedge (x_2 \vee x_3 \vee \cdots) \wedge \cdots \wedge (\overline{x_3} \vee \cdots \vee \cdots)$$

Then we add the following set of numbers to our set S (see the table below).

	1	2	3	4	$\cdots$	ℓ	c_1	c_2	$\cdots$	c_n
$y_1^{(1)}$	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
$y_1^{(2)}$	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
$y_2^{(1)}$		1	0	0	$\cdots$	0	0	1	$\cdots$	0
$y_2^{(2)}$		1	0	0	$\cdots$	0	1	0	$\cdots$	0
$y_3^{(1)}$			1	0	$\cdots$	0	1	1	$\cdots$	0
$y_3^{(2)}$			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
$y_1^{(3)}$							1	0	$\cdots$	0
$y_1^{(4)}$							1	0	$\cdots$	0
$y_2^{(3)}$								1	$\cdots$	0
$y_2^{(4)}$								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
$y_n^{(3)}$										1
$y_n^{(4)}$										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3

We must prove two things:
If $\phi \in$ 3-SAT, then $(\{y_1^{(1)}, \ldots, y_\ell^{(1)}, y_1^{(2)}, \ldots y_\ell^{(2)}, y_1^{(3)}, \ldots, y_n^{(3)}, y_1^{(4)}, \ldots, y_n^{(4)}\}, t) \in$ SUBSET-SUM (where $t = 1^\ell 3^n$). Suppose we have some satisfying assignment to the variables of ϕ. We construct Y, a subset of S that adds up to t, as follows. For each $i \in \{1, \ldots, \ell\}$, if x_i is True in the satisfying assignment, then place $y_i^{(1)}$ in Y, and otherwise place $y_i^{(2)}$ in Y. Notice that we include exactly one of each, and so the sum of our elements in Y thus far is 1^ℓ, followed by some other values in the n least significant digits. Furthermore, because we know that we started with a satisfying assignment, every clause has at least one true literal in it, so we know that for each $j \in \{1, \ldots n\}$, the set Y includes at least one number with a 1 in the column labeled c_j. On the other hand, the set Y will never include more than 3 numbers containing a 1 in column c_j, because there are only 3 literals in each clause, and thus only 3 values among $y_i^{(1)}$ and $y_i^{(2)}$ containing a 1 in the jth column. We now add a few more numbers to our set Y: if we have exactly a single 1 above the line in column c_j, we include both $y_j^{(3)}$ and $y_j^{(4)}$ in our subset. If we have two values of 1 above the horizontal line, we include only $y_j^{(3)}$ in our subset. If we have three values, we include neither $y_j^{(3)}$ nor $y_j^{(4)}$. The resulting subset adds up to exactly t.

It remains to show that if there is some subset Y of S that adds up to t, then ϕ has a satisfying assignment. For each $i \in \{1, \ldots, \ell\}$, if $y_i^{(1)} \in Y$, set x_i to True, and if not, set it to False. We claim that this constitute a valid assignment, and a satisfying assignment. To see that this is a valid assignment, note that for every $i \in \{1, \ldots \ell\}$, exactly one of $y_i^{(1)}$ and $y_i^{(2)}$ must appear in Y, or else the numbers cannot sum up to t (specifically, the first ℓ digits of the sum cannot possibly

be 1^ℓ). Therefore, exactly one of x_i and $\bar{x}_i$ will be set to True. To see that this is a satisfying assignment, note that for every $j \in \{1, \ldots, n\}$, the subset Y must include at least one number among $\{y_1^{(1)} \ldots, y_\ell^{(1)}, y_1^{(2)}, \ldots, y_\ell^{(2)}\}$ that has a 1 in the column labeled c_j (or else that column could not add up to 3). Since this value appears in Y, its corresponding variable is assigned True, and jth clause is satisfied.

2 The Class co$\mathcal{NP}$ (Jonathan Katz)

For any class $\mathcal{C}$, we define the class co$\mathcal{C}$ as co$\mathcal{C} \stackrel{\text{def}}{=} \{L \mid \bar{L} \in \mathcal{C}\}$, where $\bar{L} \stackrel{\text{def}}{=} \{0,1\}^* \setminus L$ is the complement of L. Applied to the class $\mathcal{NP}$, we get the class co$\mathcal{NP}$ of languages where *non*-membership can be efficiently verified. In other words, $L \in$ co$\mathcal{NP}$ if there exists a Turing machine M_L and a polynomial p such that (1) $M_L(x, w)$ runs in time[1] $p(|x|)$, and (2) $x \in L$ iff for all w we have $M_L(x, w) = 1$. Note why this (only) implies efficiently verifiable proofs of *non*-membership: a single w where $M_L(x, w) = 0$ is enough to convince someone that $x \notin L$, but a single w where $M_L(x, w) = 1$ means nothing.

A co$\mathcal{NP}$ language is easily obtained by taking the complement of any language in $\mathcal{NP}$. So, for example, the complement of IndSet is the language

$$\text{NoIndSet} = \left\{ (G, k) \ : \ \begin{array}{c} G \text{ does } \textit{not} \text{ have} \\ \text{an independent set of size } k \end{array} \right\}.$$

Let us double-check that this is in co$\mathcal{NP}$: we can prove that $(G, k) \notin$ NoIndSet by giving a set of k vertices that *do* form an independent set in G (this assumes the obvious verification algorithm); note that (assuming we use the obvious verification algorithm) we can never be "fooled" into believing that (G, k) is not in NoIndSet when it actually is.

As another example, we have SAT $\in \mathcal{NP}$ and $\overline{\text{SAT}} \in$ co$\mathcal{NP}$. Or, consider the language TAUT of tautologies:

$$\text{TAUT} = \{\phi : \phi \text{ is satisfied by every assignment}\}.$$

TAUT is also in co$\mathcal{NP}$.

The class co$\mathcal{NP}$ can also be defined in terms of non-deterministic Turing machines. This is left as an exercise.

Note that $\mathcal{P} \subseteq \mathcal{NP} \cap$ co$\mathcal{NP}$. (Why?) Could it be that $\mathcal{NP} =$ co$\mathcal{NP}$? Once again, we don't know the answer but it would be surprising if this were the case. In particular, there does not seem to be any way to give an efficiently verifiable proof that, e.g., a boolean formula does *not* have any satisfying assignment (which is what would be implied by $\overline{\text{SAT}} \in \mathcal{NP}$).

Conjecture 1 $\mathcal{NP} \neq$ co$\mathcal{NP}$.

3 Self-Reducibility and Search vs. Decision (Jonathan Katz)

We have so far been talking mainly about decision problems, which can be viewed as asking whether a solution *exists*. But one often wants to solve the corresponding search problem, namely to *find* a solution (if one exists). For many problems, the two have equivalent complexity.

[1] See footnote **??**.

Let us define things more formally. Say $L \in \mathcal{NP}$. Then there is some polynomial-time Turing machine M such that $x \in L$ iff $\exists w : M(x, w) = 1$. The *decision problem* for L is: given x, determine if $x \in L$. The *search problem for* L is: given $x \in L$, find w such that $M(x, w) = 1$. (Note that we should technically speak of the *search problem for* L *relative to* M since there can be multiple non-deterministic Turing machines deciding L, and each such machine will define its own set of "solutions". Nevertheless, we stick with the inaccurate terminology and hope things will be clear from the context.) The notion of reducibility we want in this setting is *Cook-Turing reducibility*. We define it for decision problems, but can apply it to search problems via the natural generalization.

Definition 1 *Language L is* Cook-Turing reducible to *L' if there is a poly-time Turing machine M such that for any oracle $\mathcal{O}'$ deciding L', machine $M^{\mathcal{O}'(\cdot)}$ decides L. (I.e., $M^{\mathcal{O}'(\cdot)}(x) = 1$ iff $x \in L$.)*

Note that if L is *Karp*-reducible to L', then there is also a Cook-Turing reduction from L to L'. In general, however, the converse is not believed to hold. We'll come back to this below.

Returning to the question of search vs. decision, we have:

Definition 2 *Let $L \in \mathcal{NP}$, and let $M_L(\cdot, \cdot)$ be a machine that verifies membership in L. We say L is* self-reducible *if there is a Cook-Turing reduction from the search problem for L to the decision problem for L. Namely, there is polynomial-time Turing machine M such that for any oracle $\mathcal{O}_L$ deciding L, and any $x \in L$ we have $M_L(x, M^{\mathcal{O}_L(\cdot)}(x)) = 1$.*

(One could also ask about reducing the decision problem to the search problem. For languages in $\mathcal{NP}$, however, such a reduction always exists.)

Theorem 2 SAT *is self-reducible.*

Proof Assume we have an oracle that tells us whether any CNF formula is satisfiable. We show how to use such an oracle to find a satisfying assignment for a given (satisfiable) CNF formula ϕ. Say ϕ is a formula on n variables $x_1, \ldots, x_n$. If $b_1, \ldots, b_\ell \in \{0, 1\}$ (with $\ell \leq n$), then by $\phi|_{b_1,\ldots,b_\ell}$ we mean the CNF formula on the variables $x_{\ell+1}, \ldots, x_n$ obtained by setting $x_1 = b_1$, $\ldots$, $x_\ell = b_\ell$ in ϕ. ($\phi|_{b_1,\ldots,b_\ell}$ is easily computed given ϕ and $b_1, \ldots, b_\ell$.) The algorithm proceeds as follows:

- For $i = 1$ to n do:
  - Set $b_i = 0$.
  - If $\phi_{b_1,\ldots,b_i}$ is not satisfiable, set $b_i = 1$. (Note: we determine this using our oracle for SAT.)
- Output $b_1, \ldots, b_n$.

We leave it to the reader to show that this always returns a satisfying assignment (assuming ϕ is satisfiable to begin with). ■

The above proof can be generalized to show that *every* $\mathcal{NP}$-complete language is self-reducible.

Theorem 3 *CLIQUE is self-reducible.*

Proof Assume we have an oracle O that, on input (G, k), outputs 1 iff G has a clique of size k. We define a machine M^O that, on input G, finds the largest clique in the graph.[2]

M^O:

For $i = n$ to 1:

[2] It might be more natural to instead show that M^O, on input (G, k), finds some clique of size k, if it exists. But finding the largest clique is even stronger, so we define the search problem this way.

- Query (G, i) to O.
  - If the output is 1 (i.e. G has a clique of size i), set $k = i$ and break;

For $j = 1$ to n

- Query $(G \setminus \{x_j\}, k)$ to O.
  - If the output is 1, set $G = G \setminus \{x_j\}$.

Output G.

■

Theorem 4 *Every $\mathcal{NP}$-complete language L is self-reducible.*

Proof The idea is similar to above, with one new twist. Let M be a polynomial-time non-deterministic Turing machine such that

$$L = \{x \mid \exists w : M(x, w) = 1\}.$$

We first define a new language L':

$$L' = \{(x, b) \mid \exists w' : M(x, bw') = 1\}.$$

I.e., $(x, b) \in L'$ iff there exists a w *with prefix* b such that $M(x, w) = 1$. Note that $L' \in \mathcal{NP}$; thus, there is a Karp reduction f such that $x \in L'$ iff $f(x) \in L$. (Here is where we use the fact that L is $\mathcal{NP}$-complete.)

Assume we have an oracle deciding L; we design an algorithm that, given $x \in L$, finds w with $M(x, w) = 1$. Say the length of w (given x) is $n = \mathsf{poly}(|x|)$. The algorithm proceeds as follows:

- For $i = 1$ to n do:
  - Set $b_i = 0$.
  - If $f((x, b_1, \ldots, b_i)) \notin L$, set $b_i = 1$. (We run this step using our oracle for L.)
- Output $b_1, \ldots, b_n$.

We leave it to the reader to show that this algorithm gives the desired result. ■

Other languages in $\mathcal{NP}$ (that are not $\mathcal{NP}$-complete) may be self-reducible as well. An example is given by graph isomorphism, a language that is not known (or believed) to be in $\mathcal{P}$ or $\mathcal{NP}$-complete. On the other hand, it is believed that not all languages in $\mathcal{NP}$ are self-reducible. One conjectured example is factoring: although compositeness can be decided in polynomial time, we do not believe that polynomial-time factoring algorithms exist.

Interestingly, any language in $\mathsf{co}\mathcal{NP}$ is Cook-Turing reducible to any $\mathcal{NP}$-complete language, but there is no Karp-reduction from a $\mathsf{co}\mathcal{NP}$-complete language to a language in $\mathcal{NP}$ unless $\mathsf{co}\mathcal{NP} = \mathcal{NP}$.

References

[1] M. Sipser. *Introduction to the Theory of Computation* (2nd edition). Course Technology, 2005.