

# CMPT 710 - Complexity Theory: Week 4

Valentine Kabanets

September 27, 2007

## 1 Co-NP

We say that a language  $L \in \text{coNP}$  if the complement of  $L$  is in NP.

In other words, if  $L \in \text{coNP}$ , then there is a NTM  $M$  with the property: if  $x \in L$ , then every computation path of  $M$  on  $x$  is accepting; if  $x \notin L$ , then at least one computation path of  $M$  on  $x$  is rejecting.

Another, equivalent definition of  $\text{coNP}$  is as follows. A language  $L \in \text{coNP}$  if there exist a constant  $c$  and a polynomial-time computable relation  $R \in \text{P}$  such that

$$L = \{x \mid \forall y, |y| \leq |x|^c, R(x, y)\}$$

**Open Question:**  $\text{NP} \stackrel{?}{=} \text{coNP}$

Define the language

$$\text{TAUT} = \{\phi \mid \phi \text{ is a tautology (i.e., identically true)}\}$$

**Theorem 1.** *TAUT is coNP-complete.*

*Proof.* The proof is easy once you realize that TAUT is essentially the same as the complement of SAT. Since SAT is NP-complete, its complement is coNP-complete. (Check this!)  $\square$

The “NP vs. coNP” question is about the existence of short proofs that a given formula is a tautology.

The common belief is that  $\text{NP} \neq \text{coNP}$ , but it is believed less strongly than that  $\text{P} \neq \text{NP}$ .

**Lemma 2.** *If  $\text{P} = \text{NP}$ , then  $\text{NP} = \text{coNP}$ .*

*Proof.* First observe that  $\text{P} = \text{coP}$  (check it!) Now we have  $\text{NP} = \text{P}$  implies  $\text{coNP} = \text{coP}$ . We know that  $\text{coP} = \text{P}$ , and that  $\text{P} = \text{NP}$ . Putting all this together yields  $\text{coNP} = \text{NP}$ .  $\square$

The contrapositive of this lemma says that:  $\text{NP} \neq \text{coNP}$  implies  $\text{P} \neq \text{NP}$ .

Thus, to prove  $\text{P} \neq \text{NP}$ , it is enough to prove that  $\text{NP} \neq \text{coNP}$ . This means that resolving the “NP vs. coNP” question is probably even harder than resolving the “P vs. NP” question.

## 2 Some NP-complete problems

There are literally hundreds if not thousands NP-complete problems; a good reference is the monograph by Garey and Johnson “Computers and Intractability: A guide to the theory of NP-completeness”. We’ll look at only a few “classic” NP-complete problems.

3-SAT =  $\{\phi \mid \phi \text{ is a satisfiable 3-CNF}\}$  (recall that a 3-CNF is a conjunction of clauses, where each clause of size at most 3)

**Theorem 3.** *3-SAT is NP-complete.*

*Proof.* We need to prove that

1. 3-SAT is in NP (it is easy), and
2. 3-SAT is NP-hard (i.e., every language  $L \in \text{NP}$  reduces to 3-SAT).

We accomplish (2) by a reduction  $\text{SAT} \leq \text{3-SAT}$ . We need to define a polynomial-time computable map taking a CNF formula  $\phi$  into a 3-CNF  $\psi$  such that  $\phi$  is satisfiable iff  $\psi$  is satisfiable. We will map each clause of  $\phi$  of size  $> 3$  into a conjunction of new size-3 clauses as follows.

Consider a clause  $C = (l_1 \vee l_2 \vee \cdots \vee l_m)$ . The first idea may be to introduce a new variable  $y$  and require that the two conditions be satisfied:

1.  $y$  is logically equivalent to  $(l_1 \vee \cdots \vee l_{m-2})$ , and
2. the clause  $(y \vee l_{m-1} \vee l_m)$  is true.

It is easy to check that the original clause  $C$  is satisfiable iff both conditions (1) and (2) hold: a satisfying assignment for  $C$  can be extended to a satisfying assignment for the conjunction of (1) and (2) (by assigning  $y$  the value  $l_1 \vee \cdots \vee l_{m-2}$ ), and conversely, a satisfying assignment for the conjunction of (1) and (2) is also satisfying for  $C$  (if we just ignore the  $y$ ).

Now, the idea is to convert the conjunction of (1) and (2) into a 3-cnf. Note that condition (2) is already a 3-clause. Condition (1) can be written as a conjunction of two implications:

3.  $y \rightarrow (l_1 \vee \cdots \vee l_{m-2})$  and
4.  $(l_1 \vee \cdots \vee l_{m-2}) \rightarrow y$ .

The first of these two (implication (3)) can be written as the clause  $\bar{y} \vee l_1 \vee \cdots \vee l_{m-2}$  (using the fact that  $a \rightarrow b$  is equivalent to  $\bar{a} \vee b$ ); the second (implication (4)) can be written as a conjunction of 2-clauses  $\bar{l}_i \vee y$ , for  $i = 1..m - 2$ .

Note that after this transformations, we got one clause of size  $m - 1$  plus  $m - 2$  clauses of size 2. We can now transform the size  $m - 1$  clause into a conjunction of a  $m - 2$  clause,  $m - 3$  size 2 clauses, and a single size 3 clause. We can continue this recursively. At the end, we’ll have a conjunction of clauses of size at most 3, and the number of such clauses will be at most  $O(m^2)$ , polynomial in the size of  $C$ .

However, we can do slightly better, and get a shorter 3-cnf from  $C$ . The observation is that of the implications (3) and (4) above, only one is actually required: if we discard

implication (4) (and keep implication (3)), then we still have that  $C$  is satisfiable iff the conjunction of (2) and (3) is satisfiable.

Indeed, if  $C$  is satisfied, then, as argued above, (2), (3), and (4) are satisfied (and so (2) and (3) are satisfied). Conversely, suppose that both (2) and (3) are satisfied. There are two cases: either  $y$  is assigned True by this satisfying assignment, or  $y$  is assigned False. In the first case, since implication (3) is satisfied, we get that one of the  $l_i$ 's, for  $i = 1..m - 2$ , is satisfied, and so  $C$  is satisfied. In the second case, since (2) is satisfied, it must be that  $l_1$  or  $l_2$  is True, and so again  $C$  is satisfied.

Recursively applying this more efficient transformation results in the following cnf for  $C$ , where  $y_1, y_2, \dots, y_{m-3}$  are new variables:

$$(l_1 \vee l_2 \vee \bar{y}_1) \wedge (l_3 \vee y_1 \vee \bar{y}_2) \wedge (l_4 \vee y_2 \vee \bar{y}_3) \wedge \dots \wedge (l_{m-2} \vee y_{m-4} \vee y_{m-3}^-) \wedge (l_{m-1} \vee l_m \vee y_{m-3})$$

□

### 3 More NP-Complete problems

The next problem will be useful in showing more problems NP-complete.

$NAE - SAT = \{\phi \mid \phi \text{ is a 3-CNF with a satisfying assignment that makes at least one literal false in every clause}\}$

NAE-SAT stands for “Not All Equal” SAT, meaning that an assignment exists under which no clause of the formula has all equal literals. (e.g.,  $(x \vee y \vee z) \wedge (x \vee \bar{z})$  has a satisfying assignment  $x = T, y = F, z = T$  of the desired type).

The useful property of formulas in NAE-SAT is this: if an assignment  $a$  satisfies  $\phi$  in the NAE-sense, then  $\bar{a}$  is also a satisfying assignment for  $\phi$ . (Check this!)

**Theorem 4.** *NAE-SAT is NP-complete.*

*Proof.* As usual, NAE-SAT is in NP (easy). To show that NAE-SAT is NP-hard, we will slightly modify our previous reduction from Circuit-SAT to SAT.

Recall that this reduction associated a variable  $y_i$  with each gate of a given circuit, and produced a formula for each gate as follows. If  $i$  is an AND gate with inputs  $j$  and  $k$ , then we create the formula expressing that “ $y_i \equiv y_j \wedge y_k$ ”. The last formula can be written as the CNF

$$(\bar{y}_i \vee y_j) \wedge (\bar{y}_i \vee y_k) \wedge (\bar{y}_k \vee \bar{y}_j \vee y_i) \tag{1}$$

Now, our modified formula will be the formula produced by the “Circuit-SAT $\leq$ SAT” reduction where each clause of size 1 or 2 gets a new literal  $z$  added to it (the same for all clauses). We claim that the new formula is satisfiable in the NAE-SAT sense iff the original formula (without the  $z$ ) is satisfiable.

Suppose the modified formula is satisfied by assignment  $a$  in the NAE-SAT sense. Then  $\bar{a}$  is also a satisfying assignment for this formula. Let’s pick the one of these two satisfying assignments that makes  $z$  False. This assignment will also satisfy the original formula (without the  $z$ ) (Check this!)

For the other direction, starting with a satisfying assignment for the original formula, we create a satisfying assignment for the modified formula by setting  $z$  to False. We need to

argue that this is a satisfying assignment in the NAE-SAT sense, i.e., that every clause has at least one false literal. Here we use the fact that the original formula has a very particular form. It has groups of clauses associated with every gate of the circuit from which this formula was constructed (by the “Circuit-SAT to SAT” reduction). For example, an AND gate will be associated with three clauses of the type given above in formula (1). The two clauses of size 2 in formula (1) will have  $z$  added to them in the modified formula, and since  $z$  is set to False, they both will have a false literal. The remaining clause of size 3 must also have a false literal. Suppose it does not. Then it is easy to see that both size-2 clause would need to be false, which contradicts the fact that we started with a satisfying assignment. The case of an OR gate, and a NOT gate can be argued similarly.  $\square$