

CMPT 710 - Complexity Theory: Week 5

Valentine Kabanets

October 11, 2005

1 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 ϕ in the NAE-sense, then \bar{a} is also a satisfying assignment for ϕ . (Check this!)

Theorem 1. *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

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

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

2 NP-completeness of 3-COL

3-COL = $\{G \mid G \text{ is a 3-colorable graph}\}$ (recall that a 3-colorable graph is a graph whose vertices may be colored with colors 0,1, and 2 in such a way that the endpoints of every edge receive different colors).

Theorem 2. *3-COL is NP-complete.*

Proof. We need to prove that

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

We prove (1) by giving the following NP algorithm for 3-COL: Given a graph G , nondeterministically guess an assignment of colors 0,1,2 to the vertices of G ; check (in deterministic polytime) that the guessed coloring is proper, i.e., that no edge has both of its endpoints colored with the same color.

To prove (2), we reduce NAE-SAT to 3-COL. Given a 3-CNF formula $\phi(x_1, \dots, x_n)$, we construct a graph G_ϕ such that $\phi \in \text{NAE-SAT}$ iff $G_\phi \in \text{3-COL}$. Our graph G_ϕ will have

vertices:

- $x_1, \bar{x}_1, x_2, \bar{x}_2, \dots, x_n, \bar{x}_n$ (i.e., one vertex for each literal),
- a vertex u , and
- a triple of vertices for each clause $C_j = (l_{j_1} \vee l_{j_2} \vee l_{j_3})$ labeled by $v(j, l_{j_1}), v(j, l_{j_2}), v(j, l_{j_3})$, respectively.

Our graph will have

edges:

- (x_i, \bar{x}_i) for each $1 \leq i \leq n$,
- (u, x_i) and (u, \bar{x}_i) for each i ,
- the triple of vertices corresponding to a clause will be connected to each other (i.e., every clause $C_j = (l_{j_1} \vee l_{j_2} \vee l_{j_3})$ corresponds to a triangle on the vertices $v(j, l_{j_1}), v(j, l_{j_2}), v(j, l_{j_3})$, and

- for every clause $C_j = (l_{j_1} \vee l_{j_2} \vee l_{j_3})$, there are three edges $(v(j, l_{j_1}), l_{j_1})$, $(v(j, l_{j_2}), l_{j_2})$, and $(v(j, l_{j_3}), l_{j_3})$ (i.e., each vertex in a clause-triangle is connected to the corresponding literal-vertex).

We now prove the correctness of our reduction. First, assume that G_ϕ is 3-colorable. Without loss of generality, the vertex u is colored with color 2. So, each of the literal-vertices connected to u will get colors 0 or 1. Our truth assignment will set variable x_i to True, if vertex x_i is colored with color 1; and to False, if vertex x_i is colored with color 0. Now we argue that this assignment is satisfying for ϕ in the NAE-SAT sense, i.e., that every clause of ϕ has at least one true literal and at least one false literal.

Consider any clause $C_j = (l_{j_1} \vee l_{j_2} \vee l_{j_3})$ corresponding to the triangle on vertices $v(j, l_{j_1}), v(j, l_{j_2}), v(j, l_{j_3})$ of G_ϕ . Suppose that all literals in C_j are assigned True by our truth assignment. Then it means that the vertices $l_{j_1}, l_{j_2}, l_{j_3}$ are all colored with color 1. So color 1 cannot be used to color the vertices of the triangle on $v(j, l_{j_1}), v(j, l_{j_2}), v(j, l_{j_3})$. But we cannot color a triangle with just two colors! A contradiction. So, at least one literal in clause C_j is assigned False. A similar argument shows that at least one literal in C_j is assigned True. So ϕ is satisfied in the NAE-SAT sense.

Now we prove the other direction. Given an assignment a to ϕ which satisfies ϕ in the NAE-SAT sense, we define a coloring for G_ϕ as follows. The vertex u gets color 2. A vertex x_i gets color 1, if x_i is set to True by the assignment a , and color 0 otherwise. Consider the triangle corresponding to each clause $C_j = (l_{j_1} \vee l_{j_2} \vee l_{j_3})$. Since assignment a is satisfying in the NAE-SAT sense, there is at least one true literal and at least one false literal in C_j . W.l.o.g., assume that l_{j_1} is True, and l_{j_2} is False. Then we color the vertex $v(j, l_{j_1})$ with color 0, the vertex $v(j, l_{j_2})$ with color 1, and the vertex $v(j, l_{j_3})$ with color 2. It is not hard to verify that this coloring is indeed a proper 3-coloring of our graph. \square

Another NP-complete problem is the Subset Sum problem: Given numbers a_1, \dots, a_n, T in binary, decide if there is a subset $S \subseteq \{1, \dots, n\}$ such that $\sum_{i \in S} a_i = T$.

Theorem 3. *Subset Sum problem is NP-complete.*

Proof. The fact that it is in NP is obvious. To show NP-hardness, we do a reduction from 3-SAT. Given a 3cnf on n variables and m clauses, we define the following matrix of decimal digits. The rows are labeled by literals (i.e., x and \bar{x} for each variable x), the first n columns are labeled by variables, and another m columns by clauses.

For each of the first n columns, say the one labeled by x , we put 1's in the two rows labeled by x and \bar{x} . For each of the last m columns, say the one corresponding to the clause $\{x, \bar{y}, z\}$, we put 1's in the three rows corresponding to the literals occurring in that clause, i.e., rows x, \bar{y} , and z . We also add $2m$ new rows to our table, and for each clause put two 1's in the corresponding column so that each new row has exactly one 1. Finally, we create the last row to contain 1's in the first n columns and 3 in the last m columns.

The $2n + 2m$ rows of the constructed table are interpreted as decimal representations of $k = 2n + 2m$ numbers a_1, \dots, a_k , and the last row as the decimal representation of the number T . The output of the reduction is a_1, \dots, a_k, T .

The proof of correctness of the reduction is left as an exercise. \square

3 “Search-to-Decision” Reductions

Suppose that $P = NP$. That would mean that all NP languages can be decided in deterministic polytime. For example, given a graph, we could decide in deterministic polytime whether that graph is 3-colorable. But could we find an actual 3-coloring? It turns out that yes, we can.

In general, we can define an NP *search problem*: Given a polytime relation R , a constant c , and a string x , find a string y , $|y| \leq |x|^c$, such that $R(x, y)$ is true, if such a y exists. . As the following theorem shows, if $P = NP$, then every NP search problem can also be solved in deterministic polytime.

Theorem 4. *If $NP = P$, then there is a deterministic polytime algorithm that, given a formula $\phi(y_1, \dots, y_n)$, finds a satisfying assignment to ϕ , if such an assignment exists.*

Proof. We use a kind of binary search to look for a satisfying assignment to ϕ . First, we check if $\phi(x_1, \dots, x_n) \in SAT$. Since we assumed that $P = NP$, this can be done in deterministic polytime. Then we check if $\phi(0, x_2, \dots, x_n) \in SAT$, i.e., if ϕ with x_1 set to False is still satisfiable. If it is, then we set a_1 to be 0; otherwise, we make $a_1 = 1$. In the next step, we check if $\phi(a_1, 0, x_3, \dots, x_n) \in SAT$. If it is, we set $a_2 = 0$; otherwise, we set $a_2 = 1$. We continue this way for n steps. By the end, we have a complete assignment a_1, \dots, a_n to variables x_1, \dots, x_n , and by construction, this assignment must be satisfying.

The amount of time our algorithm takes is polynomial in the size of ϕ : we have n steps, where at each step we must answer a SAT question. Since, by our assumption, $P = NP$, each step takes polytime. \square

Theorem 4 shows the true importance of proving that $NP = P$. If $NP = P$, we could efficiently generate a correct solution for any problem with an efficient recognition algorithm for correct solution. For instance, if $P = NP$, then we could efficiently find a login password of any user of a network, since checking if a password matches a login name can be done efficiently. Thus, if $P = NP$, essentially any secret could be found out efficiently.

As another example of the “search-to-decision” reduction, consider the problem Hamiltonian Cycle: Given an undirected graph G , decide if G has a Hamiltonian cycle (i.e., a cycle that visits every vertex of G exactly once). The corresponding search problem is: Given a graph G , find a Hamiltonian cycle in G , if such a cycle exists.

Assuming that we have access to a subroutine solving the decision version of Hamiltonian Cycle, here is an efficient algorithm for solving the search version: If G has no Hamiltonian cycle, then output “No” and halt. Otherwise, for each edge e of the graph G , if $G - e$ has a Hamiltonian cycle then $G = G - e$. After all the edges have been checked, the remaining graph is exactly a Hamiltonian cycle of G .

It should be stressed that we are interested in *efficient* (i.e., polytime) search-to-decision reductions. Such efficient reductions allow us to say that if the decision version of our problem is in P , then there is also a polytime algorithm solving the corresponding search version of the problem.

4 Nondeterministic Time Hierarchy

We want to argue that in more nondeterministic time, we can accept more languages. Recall how we argued that in the case of *deterministic* Turing machines. Given a proper complexity function $t(n)$, we constructed a language $Diag_{t(n)}$ that cannot be in $\text{Time}(t(n))$ by “diagonalizing” against every deterministic TM running in time $t(n)$. That is, we considered an enumeration of all TM’s $M_1, M_2, \dots, M_i, \dots$ and all inputs $x_1, x_2, \dots, x_i, \dots$, and defined

$$Diag_{t(n)} = \{x_i \mid M_i \text{ does not accept } x_i \text{ in } t(|x_i|) \text{ steps}\}$$

Then we argued that

1. The language $Diag_{t(n)}$ is not in $\text{Time}(t(n))$ (since it differs from the language of any $t(n)$ -time TM on at least one input).
2. The language $Diag_{t(n)}$ is in $\text{Time}(t^3(n))$ (since we can simulate a deterministic TM on a given input, and then flip its answer).

For the case of nondeterministic TM’s, we may try to follow the same approach. We can define

$$NDiag_{t(n)} = \{x_i \mid \text{NTM } M_i \text{ does not accept } x_i \text{ in } t(|x_i|) \text{ steps}\}$$

As before, it is possible to show (with exactly the same proof as in the Time case) that the new language $NDiag_{t(n)}$ is not in $\text{NTIME}(t(n))$. But, it is not at all clear if $NDiag_{t(n)}$ is in $\text{NTIME}(t^c(n))$ for some constant c . The difficulty is that, unlike the case of *deterministic* TM’s, we cannot flip the answers of a NTM deciding language L to get an NTM deciding the complement of L . This is related to the big open question $\text{NP} \stackrel{?}{=} \text{coNP}$.

Therefore, we must use a different approach. It is still based on diagonalization, but a different kind of diagonalization - so-called “lazy” diagonalization. We will see the construction next time.