

CMPT 710 - Complexity Theory: Week 4

Valentine Kabanets

September 21, 2006

1 NP Completeness

Circuit-SAT = $\{C \mid C \text{ is a satisfiable Boolean circuit}\}$

Theorem 1 (Cook-Levin). *Circuit-SAT is NP-complete.*

Proof. We need to prove that

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

The fact that Circuit-SAT is in NP is easy: Given a circuit C on variables x_1, \dots, x_n , nondeterministically guess an assignment to x_1, \dots, x_n and verify that this assignment is satisfying; this verification can be done in time polynomial in the size of the circuit. In other words,

$$\text{Circuit-SAT} = \{C \mid \exists x, |x| \leq |C|, R'(C, x)\}$$

where $R'(C, x)$ is True iff $C(x) = 1$ (i.e., C on input x evaluates to 1).

Now we prove NP-hardness. Take an arbitrary $L \in \text{NP}$. Say

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

for some constant c and $R \in \text{P}$. Let's suppose that $R(x, y)$ is computable in time $N = (|x| + |y|)^d$, for some constant d .

Consider N steps of computation of the Turing machine deciding R on input x, y . This computation can be pictured as a sequence of N configurations. A configuration at time t is a sequence of symbols $y_1 \dots y_m$, where each y_j contains the following information: the contents of tape cell j at time t , whether or not tape cell is being scanned by the TM at time t , and if it is, then what is the state of a TM at time t .

The crucial observation is that the computation of a TM has the following "locality property": the value of symbol y_i at time $t + 1$ depends only on the values of symbols y_{i-1}, y_i, y_{i+1} at time t (as well as the transition function of the TM).

We can construct a constant-size circuit *Step* that computes the value of y_i at time $t + 1$ from the values of y_{i-1}, y_i, y_{i+1} at time t . Now, we construct a big circuit $C(x, y)$ by replacing each symbol y_i in every configuration at time t by a copy of the circuit *Step* whose inputs

are the outputs of the corresponding three copies of *Step* from the previous configuration. We also modify the circuit so that it outputs 1 on x, y iff the last configuration is accepting.

The size of the constructed circuit will be at most $N * N * |Step|$ (N configurations, at most N copies of *Step* in each), which is polynomial in $|x|$.

Our reduction from L to Circuit-SAT is the following: Given x , construct the circuit $C_x(y) = C(x, y)$ as explained above (with x hardwired into C). It is easy to verify that $x \in L$ iff there is y such that $C_x(y) = 1$. So this is a correct reduction. \square

$$SAT = \{\phi \mid \phi \text{ is a satisfiable Boolean formula}\}$$

Theorem 2 (Cook-Levin). *SAT is NP-complete.*

Proof. SAT is in NP (easy). To prove NP-hardness, we will show that Circuit-SAT is reducible to SAT.

Let C be an arbitrary Boolean circuit with gates g_1, \dots, g_m , where g_1, \dots, g_n are input gates and g_m is the output gate. For each g_j , introduce a Boolean variable y_j . For every $i > n$, define the Boolean formula $gate_i$ expressing that the value of y_i is equal to the value of the gate g_i . That is, if gate g_i is an AND gate with inputs g_{i_1} and g_{i_2} , then $gate_i$ is True iff $y_i \equiv y_{i_1} \wedge y_{i_2}$; similarly, for OR, and NOT gates.

Our final formula ϕ_C is defined as

$$\bigwedge_{i=n+1}^m gate_i \wedge "y_m \equiv 1"$$

It is left as an exercise to verify that C is satisfiable iff ϕ_C is satisfiable. \square

2 Importance of the Cook-Levin Theorem

There is a trivial NP-complete language:

$$L_u = \{(M, x, 1^k) \mid \text{NTM } M \text{ accepts } x \text{ in } \leq k \text{ steps}\}$$

Exercise: Show that L_u is NP-complete.

The language L_u is not particularly interesting, whereas SAT is extremely interesting since it's a well-known and well-studied natural problem in logic. After Cook and Levin showed NP-completeness of SAT, literally hundreds of other important and natural problems were also shown to be NP-complete. It is this abundance of natural complete problems which makes the notion of NP-completeness so important, and the "P vs. NP" question so fundamental.

3 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 **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 3. *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 4. *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.

4 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.

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

Theorem 5. *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 $\text{SAT} \leq 3\text{-SAT}$. We need to define a polynomial-time computable map taking a CNF formula ϕ into a 3-CNF ψ such that ϕ is satisfiable iff ψ is satisfiable. We will map each clause of ϕ of size > 3 into a conjunction of new size-3 clauses as follows.

Consider a clause $C = (l_1 \vee l_2 \vee \dots \vee l_m)$. 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 $(l_1 \vee \dots \vee l_{m-2})$, 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 $l_1 \vee \dots \vee l_{m-2}$), 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. $y \rightarrow (l_1 \vee \dots \vee l_{m-2})$ and
4. $(l_1 \vee \dots \vee l_{m-2}) \rightarrow y$.

The first of these two (implication (3)) can be written as the clause $\bar{y} \vee l_1 \vee \dots \vee l_{m-2}$ (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:

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

□