Lecture 7

Satisfiability and Quantified Boolean Formulas

January 27, 1998
Lecturer: Richard Ladner
Notes: Mark Friedman

Today we have

- the reduction of SAT to 3-CNFSAT,
- the Cook-Levin Theorem showing SAT to be NP-complete, and
- the similar theorem showing Quantified Boolean Formulas to be PSPACE-complete.

7.1. Propositional Logic Continued

Theorem 7.1: SAT $\leq_m^P$ 3-CNFSAT.

For any formula α, we will construct, in polynomial time, a formula $F(\alpha)$ that is in conjunctive normal form, with at most three literals per clause, which is satisfiable if and only if α is. We will introduce a new variable V_β for each subformula β and write down clause $C(\beta)$ which relates V_β and β.

- $C(x)$ is $V_x \equiv x$
- $C(\alpha \wedge \beta)$ is $V_{\alpha\wedge\beta} \equiv V_\alpha \wedge V_\beta \wedge C(\alpha) \wedge C(\beta)$
- $C(\alpha \vee \beta)$ is $(V_{\alpha\vee\beta} \equiv V_\alpha \vee V_\beta) \wedge C(\alpha) \wedge C(\beta)$
- $C(\alpha \equiv \beta)$ is $(V_{\alpha\equiv\beta} \equiv (V_\alpha \equiv V_\beta)) \wedge C(\alpha) \wedge C(\beta)$
- $C(\neg\alpha)$ is $(V_{\neg\alpha} \equiv \neg V_\alpha) \wedge C(\alpha)$

The other operators are similarly done.

For example, let α be $(x \vee y) \wedge (\neg x \vee y)$. Then $C(\alpha)$ is

$$(V_\alpha \equiv V_{x\vee y} \wedge V_{\neg x \vee y})\wedge$$
$$(V_{x\vee y} \equiv V_x \vee V_y) \wedge$$
$$(V_{\neg x\vee y} \equiv V_{\neg x} \vee V_y) \wedge$$
$$(V_{\neg x} \equiv \neg V_x) \wedge$$
$$(V_x \equiv x) \wedge$$
$$(V_y \equiv y).$$

Define: $F(\alpha) = V_\alpha \wedge C(\alpha)$.

F is in 3-CNF. Each RHS in the recursive definition of C is a conjunction of 3-CNF formulas and a single new clause, which has at most three variables. This clause can be replaced by its 3-CNF-conversion in constant space.

We now show that α is satisfiable if and only if $F(\alpha)$ is satisfiable.

This follows from the following two observations:

1. $\tau(C(\alpha)) = T$ if and only if $\tau(V_\beta) = \tau(\beta)$ for all subformulas β of α.
2. Given an assignment σ of T or F to the original variables, you can extend that assignment naturally to σ' such that $\sigma'(C(\beta)) = T$ for all subformulas β of α.

Now to finish the proof if α is satisfiable by an assignment σ, then extend σ to σ' as in 2. By 1, $\sigma(V_\alpha \wedge C(\alpha)) = T$. If $\sigma(F(\alpha)) = T$ then then both $\sigma(V_\alpha) = T$ and $\sigma(C(\alpha)) = T$. By 1, $\sigma(\alpha) = T$.

It may be helpful to view α as a network of gates. Then it is easy to see that the V_i label intermediate values. $F(\alpha)$ now merely 'says' that α is satisfied exactly when its output is true and each gate works properly.

7.2. The Cook-Levin Theorem

Theorem 7.2: SAT is NP-complete.

SAT is clearly in NP. Given a propositional formula, and an assignment, you can check the truth value of the formula in polynomial time.

Suppose we have a nondeterministic Turing machine T which decides L, in n^k time. We will show completeness by constructing, in polynomial time, a formula that is true if and only if T accepts.

Fix input w, which is of length n. The formula F will have variables $x_{\sigma ij}$ representing "the ith symbol of the jth configuration is the letter σ." For simplicity we will often drop the last subscript when the configuration is understood. We will not distinguish between a configuration X and the vector of variables representing it.

We constrain T to be a nondeterministic Turing machine with one tape and a single accepting state, from which there is a self-transition that does nothing. A computation can be of length at most n^k, so a configuration will look like:

$$\#\sigma_1\sigma_2\ldots\sigma_h q\sigma_{h+1}\sigma_{h+2}\ldots\sigma_{m-3}\#$$

where $m = cn^k + d$. m is the length of a configuration, and an upper bound on the number of steps in the shortest accepting computation of T. If we define $\Gamma' = \Gamma \cup Q \cup \{\#\}$, a configuration is a string in Γ'^*.

We can define a six-place relation $R \subseteq \Gamma'^6$ which identifies all legal triples of successive symbols in two successive configurations of a legal computation in T. That is, we claim that the legality constraint for two successive turing machine configurations can be expressed as a conjunction of constraints on successive triples, such as in C and C' below.

```
C       xqyzxyzxyzXYZxyzxyz
C'      xwpzxyzxyzXYZxyzxyz
```

The relation R indicates that when a machine moves, it may only change symbols in the three-symbol window surrounding the state, and only by the legal rules in the finite state control of T.

- $R(aqbacp)$ whenever $(qbpcR)$ is a transition of T.
- $R(aqbpcb)$ whenever $(qbpcL)$ is a transition of T.
- $R(abcdbe)$ whenever a, b, c are not symbols in Q.
- $R(\#ab\#cd)$ for all symbols a, b, c, d.
- $R(ab\#cd\#)$ for all symbols a, b, c, d.
- $R(aAbaAb)$ where A is the accept state.

Now we can write down declaratively the constraints on variables that describe an accepting computation of input w.

- x is a string, with exactly one symbol per postion.

$$S(x) = \left(\bigwedge_{i=1}^{m} \bigvee_{\sigma\in\Gamma'} x_{\sigma i}\right) \wedge \left(\bigwedge_{i=1}^{m-2} \bigwedge_{\sigma\in\Gamma'} \bigwedge_{\substack{\tau\in\Gamma' \\ \tau\neq\sigma}} x_{\sigma i} \Rightarrow x_{\tau i}\right)$$

- T can compute y from x in one step.

$$C(x,y) = \bigwedge_{i=1}^{m-2} \bigvee_{\substack{\sigma_0,\sigma_1,\sigma_2,\tau_0,\tau_1\tau_2\in\Gamma' \\ R(\sigma_0\sigma_1\sigma_2\tau_0\tau_1\tau_2)}} x_{\sigma_0 i} \wedge x_{\sigma_1(i+1)} \wedge x_{\sigma_2(i+2)} \wedge y_{\tau_0 i} \wedge y_{\tau_1(i+1)} \wedge y_{\tau_2(i+2)}$$

- x_0 is the initial configuration of T on input $w = w_1 \ldots w_n$, where s is the start state of T, and the rest of the tape is blank.

$$Init_w(x) = x_{\#1} \wedge x_{\#m} \wedge x_{s2} \wedge x_{w_1 3} \wedge \ldots \wedge x_{w_n(n+2)} \bigwedge_{i=n+3}^{m-1} x_{\sqcup i}$$

Now we define one big hairy formula F_w, which 'says' that at time m we accept, at time zero we have the initial configuration, and every step is legal.

$$F_w = Init_w(x_0) \wedge \bigvee_{i=1}^{m-1} x_{Aim} \wedge \bigwedge_{t=0}^{m} S(x_t) \wedge \bigwedge_{t=0}^{m-1} C(x_t, x_{t+1})$$

Note that F_w is quadratic in w, and is in CNF, if you distribute the disjunction in $C(x,y)$. In reality, there is a logarithmic expansion factor as well in the worst case, because the number of new variables is proportional to the length of α, But the number of different variables in α may be very small. So encoding these different symbols incurs a logarithmic penalty.

7.3. Quantified Boolean Formulas

A SAT problem maps to a problem of the form $\exists\overline{x}F$ for a vector of boolean variables $\overline{x}$ and a propositional formula F. *Quantified Boolean Formulas*, or QBF, are of the form $Q\overline{x}F$, where Q is a quantifier ($\exists$ or $\forall$), and F is a propositional *or* QBF formula. One example of a QBF is

$$\exists x \exists y \forall z (x \equiv z \vee y \equiv z)$$

A few side notes. We will be lax about syntactic parentheses rules. The quantifiers do not need to be all at the beginning. When they are, the QBF is said to be in *prenex* form. To convert to prenex form is easy. First rename away any quantified variable from all other variables to prevent variable capture. Then distribute $\vee$'s and $\wedge$'s over quantifiers. For a quantifier to cross a negation, it must flip from an existential to a universal, or vice versa. For example, the prenex form of $\forall y \exists x (x \wedge \neg(\exists x (x \equiv y))$ is

$$\forall y \exists x \forall z (x \wedge \neg(z \equiv y))$$

A QBF is well-formed if all variables are quantified. We adopt the convention that any unquantified variable is implicitly existentially quantified, and then all QBF formulae are well-formed. In

QBF there is no distinction between truth and validity. We can define what it means to be true by enumerating the possible combinations of truth assignments to quantifiers, going left to right. Define B_ω to be the set of all true quantified Boolean formulas.

Theorem 7.3: B_ω is PSPACE-complete.

AP = PSPACE, so we will show B_ω is AP-complete. B_ω is in AP. Reading the input formula left to right, when we reach $\exists x$, we branch existentially and choose a truth assignment for x. Similarly, when we reach $\forall y$, we branch universally and choose a truth assignment for y. Then we check the formula for truth, given truth assignments for all the variables.

Completeness. Now we proceed as in Cook's theorem. Let T be an alternating Turing machine that runs in n^k time, on input x of length n. The initial state is existential. Without loss of generality, we can add $O(n^k)$ length to the computation to make every state alternate.

We use the same boolean variables, and configurations as in Cook's theorem. We also use $Init$ and C as defined there. We define the ACCEPT predicate on configurations recursively:

- T accepts,
$$ACCEPT_m = \bigvee_{i=1}^{m} x_{Aim}.$$
- A state is accepting if its successor state(s) is(are),
If t is even,
$$ACCEPT_t(x_t) = \exists x_{t+1}(C(x_t, x_{t+1}) \wedge ACCEPT_{t+1}(x_{t+1})).$$
If t is odd,
$$ACCEPT_t(x_t) = \forall x_{t+1}(C(x_t, x_{t+1}) \Rightarrow ACCEPT_{t+1}(x_{t+1})).$$

Now we write down a big hairy QBF that 'says' whether the computation of T accepts input w.

$$F_w = \exists x_0(Init_w(x_0) \wedge ACCEPT_0(x_0)).$$