COMP 409, Lecture 10

February 18, 2003

1 Co-NP and CoNP-Completeness

Recall that φ is satisfiable if and only if $(\neg\varphi)$ is not valid and that φ is valid if and only if $(\neg\varphi)$ is not satisfiable. Thus, if it holds, $SAT \in PTIME \leftrightarrow VALID \in PTIME$. NP characterizes the difficulty of solving SAT. How can we characterize the difficulty of solving VALID? Recall that the unique feature of VALID is that it is easy to verify counterexamples to refute the validity of a formula. We need a complexity class that denotes this (*i.e.*, easy to check counterexamples).

We have previously discussed several complexity classes. We now have a new one to consider, $Co-NP$:

Definition 1 *$L \subseteq \Sigma^*$ is in $Co-NP$ is there is a polynomial P and $P-time$ algorithm A, such that for all $x \in \Sigma^*$, we have $x \notin L$ if and only if there is some $y \in \Sigma^{\leq P(|x|)}$ such that $A(x,y) = 1$.*

Observe that in NP, given an answer, it is easy to show that it is indeed a solution, while for Co-NP, if given something that is *not* an answer, it is easy to show that it is not a solution. ($SAT \in NPC$, $valid \in Co-NPC$)

Lemma 1 *$L \in Co-NP$ if and only if $\Sigma^* - L \in NP$*

Like NP-complete class, NPC, we can define the CoNP-complete class, $CoNPC$, which represents the hardest problems in $CoNP$ as follows:

Definition 2 *$L \in CoNPC$ if*

1. *$L \in CoNP$*
2. *$L' \leq_p L$ for all $L' \in CoNP$*

Lemma 2 *$L \in Co-NP-Complete$ if and only if $\Sigma^* - L$ is NP-Complete.*

Lemma 3 $P \subseteq NP \bigcap Co-NP$

In this regard consider the problem of testing PRIMALITY. It is trivial to show that $PRIMALITY \in Co-NP$. Showing that $PRIMALITY \in NP$ is attributed to Pratt, 1976. Thus, $PRIMALITY \in NP \bigcap Co-NP$. However, it was not clear until 2002 whether $PRIMALITY \in P$. In 2002, Agarwal et al showed that $PRIMALITY \in P$.

Lemma 4 *If $L \in NPC$ and $L \in Co-NP$, then $NP = Co-NP$*

Ladner's Theorem: If $P \neq NP$, then $NP - P - NPC \neq \emptyset$. If $P = NP$, then $NP = Co-NP$. If $NP \neq Co-NP$, then $P \neq NP$.

Corollary 1 $VALID \in CoNPC$

1.1 Open Problem

1. $NP \stackrel{?}{=} Co-NP$. Are SAT and VALID equally hard? This is like asking about the relationship beteen $CoNP$ and NP.
2. $P \stackrel{?}{=} NP \bigcap Co-NP$.
No one knows.... (at least as of February 18, 2003)

2 Practical Methods for SAT

The fact that SAT is in NPC does not mean that we cannot find algorithms for it which perform better in the average case.

Although there is no good standard way to compare algorithms, there are several notions of "better" that we can work with. One of these is better provable asymptotic worst-case complexity. Another, fuzzier metric is how fast the algorithms run on real problems in practice (especially large problems).

If we have two algorithms A and B such that the cost complexity of A is $O(2^{\alpha n})$ and the cost complexity of B is $O(2^{\beta n})$, where $\alpha > \beta$, then B represents an exponential improvement over A, for $\frac{2^{\alpha n}}{2^{\beta n}} = 2^{(\alpha-\beta)n}$.

For SAT, the search space is 2^{PROP}, and for a given formula φ, we wish to determine models(φ) $\stackrel{?}{=} \emptyset$. There are two widely used techniques: search methods and symbolic methods.

2.1 Search Methods

Given the formula $\varphi = ((p \rightarrow q) \rightarrow (r \rightarrow (p \wedge (s \rightarrow \neg q)))) \wedge (p \wedge \neg p)$, we can tell fairly easily that φ is not satisfiable, since $(p \wedge \neg p)$ will clearly hold for neither $p = 0$ nor $p = 1$. From this, we can see that there might be considerable benefit in employing a decomposition technique when trying to determine if some expression is satisfiable. To examine the merits of such a technique, we introduce the idea of *simplification*, which is essentially constant propogation, or partial evaluation, for formulas.

We begin by extending our notion of what a formula is:

$$FORM' = FORM \cup \{0,1\}.$$

This is a unification of the syntactical and semantical domains, i.e.

- $\neg : FORM' \to FORM'$.
- $\circ : FORM' \times FORM' \to FORM'$.

We now define what our standard connectives mean in this new context:

$$\neg(\varphi) = \begin{cases} 1, & \varphi = 0 \\ 0, & \varphi = 1 \\ (\neg\varphi), & \varphi \in FORM \end{cases},$$

$$\vee(\varphi,\psi) = \begin{cases} 1 & \varphi = 1 \\ 1 & \psi = 1 \\ 0 & \varphi = 0 \text{ and } \psi = 0 \\ (\varphi \vee \psi) & \varphi,\psi \in FORM \\ \psi & \varphi = 0 \text{ and } \psi \in FORM \\ \varphi & \psi = 0 \text{ and } \varphi \in FORM \end{cases},$$

$$\wedge(\varphi,\psi) = \begin{cases} 0 & \varphi = 0 \\ 0 & \psi = 0 \\ 1 & \varphi = 1 \text{ and } \psi = 1 \\ (\varphi \wedge \psi) & \varphi,\psi \in FORM \\ \psi & \varphi = 1 \text{ and } \psi \in FORM \\ \varphi & \psi = 1 \text{ and } \varphi \in FORM \end{cases},$$

$$\to(\varphi,\psi) = \begin{cases} 0 & \varphi = 1 \text{ and } \psi = 0 \\ 1 & \varphi = 0 \text{ or } \psi = 1 \\ (\varphi \to \psi) & \varphi,\psi \in FORM \\ \psi & \varphi = 1 \text{ and } \psi \in FORM \\ \neg\varphi & \psi = 0 \text{ and } \varphi \in FORM \end{cases},$$

$$\leftrightarrow(\varphi,\psi) = \begin{cases} 0 & \varphi = 1 \text{ and } \psi = 0 \\ 0 & \varphi = 0 \text{ and } \psi = 1 \\ 1 & \varphi = 1 \text{ and } \psi = 1 \\ 1 & \varphi = 0 \text{ and } \psi = 0 \\ (\varphi \leftrightarrow \psi) & \varphi,\psi \in FORM \\ \psi & \varphi = 1 \text{ and } \psi \in FORM \\ \varphi & \psi = 1 \text{ and } \varphi \in FORM \\ \neg\varphi & \psi = 0 \text{ and } \varphi \in FORM \\ \neg\psi & \varphi = 0 \text{ and } \psi \in FORM \end{cases}.$$

Now we define *simplification*.

Let $\varphi \in Form$. Let $c \in \{0,1\}$. Let $p \in AP(\varphi)$.

$$\varphi[p \mapsto c] = \begin{cases} c & \text{if } \varphi = p \\ q & \text{if } \text{for some } q \neq p,\ \varphi = q \\ \neg(\varphi'[p \mapsto c]) & \text{if } \varphi = (\neg\varphi') \\ \circ(\varphi_1[p \mapsto c], \varphi_2[p \mapsto c]) & \text{if } \varphi = (\varphi_1 \circ \varphi_2) \end{cases}$$

Lemma 5 *Let $\varphi \in Form$ and $\tau \in 2^{Prop}$, $p \in Prop$. Then $\tau \models \varphi$ iff $\tau\mid_{Prop-\{p\}} \models \varphi[p \mapsto \tau(p)]$.*

Corollary 2 *If $\varphi[p \mapsto 0]$ is satisfiable, then φ is satisfiable. If $\varphi[p \mapsto 1]$ is satisfiable, then φ is satisfiable.*

Corollary 3 *φ is satisfiable iff $\varphi[p \mapsto 0]$ is satisfiable OR $\varphi[p \mapsto 1]$ is satisfiable.*

The following algorithm follows from the above concepts and is called the *splitting method*. Other names include the *Davis-Logermann-Loveland* (DLL) method, the *Davis-Putnam* method, and the DPLL method. They all refer to the same algorithm. It is essentially case analysis.

$SAT(\varphi)$
if $\varphi \in \{0,1\}$ **then** return φ
choose $p \in AP(\varphi)$ and $c \in \{0,1\}$
if $SAT(\varphi[p \mapsto c])$ **then** return **true**
if $SAT(\varphi[p \mapsto 1-c])$ **then** return **true**
return **false**

The implementor has a lot of choice with this algorithm, since "easily satisfiable" and "choose" are not specified. An obvious choice for "easily satisfiable" is that only the (extended) formula 1 is easily satisfiable. We will see other choices later.

The running time of this procedure is still exponential ($2^{O(n)}$), but it will usually be much more efficient than the truth-table algorithm. Note, however, that it is important how we choose p and c in this algorithm. If they are chosen poorly enough, the running time for this procedure will be no better than that of the original truth table algorithm.

The space requirement of this algorithm is polynomial.

If we do not want to do the same work twice, we can use "memoing". That is, once we determine whether a formula is satisfiable, we record this. When we reach formula that has already been examined, we the saved result. This technique saves time, but could use exponential space. In effect, we are trading space for time.