Lecture 14

October 5, 2005

1 Overview

In the lectures so far we have been discussing NP-complete problems, but we did not indicate any strategies for solving them. In this lecture we will focus on a very simple and intuitive method that will allow us to determine whether a formula is satisfiable.

In order to justify the correctness of the algorithm, we need several lemmas.

Lemma 1 (Substitution). *Let* $\varphi \in \text{Form}, \tau \in 2^{AP(\varphi)}$*, and* $p \in AP(\varphi)$*. Then* $\tau \models \varphi \iff \tau_{AP(\varphi)-\{p\}} \models \varphi[p \mapsto \tau(p)]$.

Essentially, this means that if we keep simplifying τ, eventually we will just get 1 or 0. The proof of Lemma 1 is one of the exercises on homework 3.

Corollary 2. *Let* $\varphi \in \text{Form}$*, and* $p \in AP(\varphi)$*. Then* φ *is satisfiable* $\iff$ $\varphi[p \mapsto 0]$ *is satisfiable or* $\varphi[p \mapsto 1]$ *is satisfiable.*

Proof.

$\boxed{\Rightarrow}$ Suppose that φ is satisfiable. Then $\tau \models \varphi$ for some $\tau \in 2^{AP(\varphi)}$. By the lemma above, $\tau_{AP(\varphi)-\{p\}} \models \varphi[p \mapsto \tau(p)]$. But either $\tau(p) = 0$ or $\tau(p) = 1$. So either $\varphi[p \mapsto 0]$ or $\varphi[p \mapsto 1]$ is satisfiable.

$\boxed{\Leftarrow}$

- Suppose $\varphi[p \mapsto 0]$ is satisfiable. Then $\tau \models \varphi[p \mapsto 0]$ for some $\tau \in 2^{AP(\varphi)-\{p\}}$. Define $\tau' \in 2^{AP(\varphi)}$ as follows:

$$\tau'(q) = \begin{cases} 0, & \text{if } q \text{ is } p \\ \tau(q), & \text{otherwise} \end{cases}$$

  By the Substitution Lemma (Lemma 1), $\tau' \models \varphi$. So φ is satisfiable.

- Suppose $\varphi[p \mapsto 1]$ is satisfiable.Then $\tau \models \varphi[p \mapsto 1]$ for some $\tau \in 2^{AP(\varphi)-\{p\}}$ Define $\tau' \in 2^{AP(\varphi)}$ as follows:

$$\tau'(q) = \begin{cases} 1, & \text{if } q \text{ is } p \\ \tau(q), & \text{otherwise} \end{cases}$$

  By the Substitution Lemma, $\tau' \models \varphi$. So φ is satisfiable.

□

Lemma 1 suggests a strategy for solving SAT problems, which follows the age-old rule "Divide and conquer"[1]. Here we present a simplified version, and the course project will be to build a very fast version.

2 Splitting Method

The algorithms works as follows:

$SAT(Form\ \varphi)$
if φ is "easily satisfiable" **then** return **true**
if φ is "easily unsatisfiable" **then** return **false**
choose $p \in AP(\varphi)$, choose $c \in \{0, 1\}$
if $SAT(\varphi[p \mapsto c])$ **then** return **true**
if $SAT(\varphi[p \mapsto 1 - c])$ **then** return **true**
return **false**

Certain steps are left open for the implementor of this algorithm. "Easily satisfiable" and "easily unsatisfiable" are not clearly defined. For instance, "easily satisfiable" could mean 1 or it could mean $(\varphi \vee \neg\varphi)$.

The running time of this procedure is still exponential, $O(2^{|\varphi|})$, 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, the running time for this procedure will be no better than that of the simple truth table algorithm.

The space requirement is equal to the size of the call stack. The worst case stack depth is $O(|\varphi|)$, and each stack frame is of linear size. This makes the space requirements polynomial with the size of the input formula.

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 a formula that has already been examined, we look up the saved result. This technique saves time, but could use exponential space. In effect, we are trading space for time.

The Splitting Method is the dominant method of solving large SAT problems right now. Reading the formula from disk often takes longer than running this algorithm. In some cases, however, this method does not work very efficiently. For those, we have another method, which will be introduced in Lecture 17.

3 Data Representation

Consider the problem of multiplying two polynomials $(a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0)(b_n x^n + b_{n-1}x^{n-1} + \ldots + b_0)$. The "traditional" method (i.e., multiplying the corresponding coefficients for each index) takes $O(n^2)$ time. How-

[1] *Divide et impera*

ever, we solve the problem in this way just because we are confined to this particular representation. It turns out that there is an algorithm which uses Fast Fourier Transform (FFT) to transform the polynomials to another representation, and altogether takes $O(n \log n)$ time. Pushing it even further, assume that the polynomials are represented in the roots form. That is, assume $(a_n x^n + a_{n-1} x^{n-1} + \ldots + a_0) = (x - z_1^1)(x - z_2^1) \ldots (x - z_n^1)$ and $(b_m x^m + b_{m-1} x^{m-1} + \ldots + b_0) = (x - z_1^2)(x - z_2^2) \ldots (x - z_m^2)$. Then finding the product becomes trivial (by simply concatenating them) and only takes linear time.

Due to this observation, we now go back to the satisfiability problem to see if we can find alternate representations with more efficient algorithms. We could represent the set of truth assignments for a formula φ by $models(\varphi)$.

3.1 Semantic Space

The discussion by far is on the syntactic space, which is defined by $\varphi \in \mathit{Form}$. We also have the semantic space defined by $\text{models}(\varphi) \in 2^{2^{\text{Prop}}}$. Then the satisfiability problem is reduced to $\varphi \mapsto \text{models}(\varphi) \stackrel{?}{=} \emptyset$.

Let's see if semantic space is a good representation:

If $\varphi = p$ then $|models(p)| = 2^{n-1}$ (exponential!)

If $\varphi = (\theta \vee \psi)$ then $models(\varphi) = models(\theta) \cup models(\psi)$. (polynomial time operation)

If $\varphi = (\theta \wedge \psi)$ then $models(\varphi) = models(\theta) \cap models(\psi)$. (polynomial time operation)

If $\varphi = \neg\theta$ then $models(\varphi) = 2^{\text{Prop}} - models(\theta)$. This operation can blow up since if $models(\theta)$ is a small set then its complement will be big. Conclusion: It is not advantageous to work with such a representation for truth assignments.

We observe that semantic space has easy SAT test ($\text{models}(\varphi) \stackrel{?}{=} \emptyset$) and hard base case ($\text{models}(p_i) = \{\tau : \tau(p_i) = 1\}$ is exponentially large). For inductive case, union and intersection are also in polynomial time. However, the complement could be a very large set (i.e., slow). So semantic space is not a good representation.

The previous representation does not degrade gracefully (it all of a sudden broke horribly). We need a representation that has the following desiderata (desired properties):

1. Easy base case.
2. Easy inductive steps.
3. Easy satisfiability test.
4. Easy conversion between representations (in polynomial time).

3.2 Deterministic Finite Automata

Consider writing all propositions consecutively $p_1 p_2 \dots p_n$, i.e. we fix the enumeration of propositions. Then, a truth assignment could be written as a string (bit vector) like $00\dots0 \in \{0,1\}^n$, which is a binary word of length n. So a set of truth assignments can also be written as a set of words, which forms a language.

Fact 1. *Finite languages are regular.*

Since the number of propositions is finite, the language must also be finite, and so the language must be regular. Therefore, we have $\varphi \mapsto A_\varphi$ (an automaton for φ), and $L(A_\varphi) = \text{models}(\varphi)$. We now define a deterministic finite automaton (DFA). The following definition is a the formal definition of the state machines that are taught in Elec 326, Comp 412, Comp 481, Comp 280 or one of a number of other classes.

Definition 1 (DFA). *A DFA A is a quintuple $(\Sigma, S, s_0, \rho, F)$ where*

$$\begin{aligned}
\Sigma &: \textit{finite alphabet}\\
S &: \textit{finite set of states}\\
s_0 \in S &: \textit{initial state}\\
\rho : S \times \Sigma \to S &: \textit{transition function}\\
F \subseteq S &: \textit{set of accepting states}
\end{aligned}$$

Definition 2 (Transitions on words). *Define $\overline{\rho} : S \times \Sigma^* \to S$ inductively.*

- $\overline{\rho}(\varepsilon) = s_0$
- $\overline{\rho}(wa) = \rho(\overline{\rho}(w), a)$

Definition 3 (Acceptance condition). *A DFA A accepts w if $\overline{\rho}(s_0, w) \in F$.*

Definition 4 (Language of an automaton). $L(A) = \{w \mid A \textit{ accepts } w\}$.

Example 1. *The following DFA accepts strings that terminate with a 1:*

$$\begin{aligned}
\Sigma &: \{0,1\}\\
S &: \{0,1\}\\
s_0 &: \mathit{0}\\
\rho &: \ \rho(s,a) = a\\
F &: \{1\}
\end{aligned}$$

Example 2. *The following DFA accepts strings of length n.*

$$\begin{aligned}
\Sigma &: \{0,1\}\\
S &: \{0,1,\dots,n,n+1\}\\
s_0 &: \mathit{0}\\
F &: \{n\}
\end{aligned}$$

where ρ *is defined as:*

$$\rho(i, a) = \begin{cases} i+1, & i \leq n \\ i, & i = n+1 \end{cases}$$

The state $n+1$ *is also known as the* sink state*.*

In the next lecture we will present a mechanical way of constructing automata that correspond to a particular logical formula.