EE224: Handout-2
Formulas, Switching Algebras and Minimization

Madhav P. Desai

January 23, 2018

1 Boolean Functions to Boolean Formulas

We have already seen that a finite Boolean algebra is isomorophic to some power set of a finite set. Thus, a finite Boolean algebra must have 2^n elements for some $n \geq 0$. The simplest Boolean algebra is the trivial one with a single-element, which corresponds to a set algebra consisting of just the empty set.

The smallest non-trivial Boolean algebra consists of two elements and we will denote it by $\mathbf{B}_2$, consisting of just the two elements $\{0, 1\}$. The elements of $\mathbf{B}_2$ will be called *bits*. Any larger, finite Boolean algebra will then be isomorphic to $\mathbf{B}_2^n$, whose elements are $n-$tuples of bits (called bit-vectors).

Now consider any function $f : \mathbf{B}_2^n \rightarrow \mathbf{B}_2$. Such a function maps bit-vectors to bits. A bit of reflection shows that the set of such functions is also a Boolean algebra with 2^{2^n} elements. Each Boolean function f can be identified with the subset f_{ON} (this is called the ON-set of f) of $\mathbf{B}_2^n$ consisting of elements u such that $f(u) = 1$. The set of atoms in this Boolean algebra consists of those functions which have a single element in the ON-set. Remember that in a Boolean algebra

$$u = \sum_{a \in A_u} a \tag{1}$$

and

$$u = \prod_{a \in A_{\overline{u}}} \overline{a} \tag{2}$$

wher A_x represents the set of atoms $\leq x$.

We introduce a convenient way to represent Boolean functions, namely, the notion of a *formula*. A formula on n variables $x_1, x_2, \ldots x_n$ is a *string* constructed using the following rules:

- 0, 1, $x_1,\ x_2, \ldots x_n$ are formulas.
- if A is a formula then $\overline{A}$ is a formula[1].
- if $A,\ B$ are formulas, then $(A + B)$ and $(A.B)$ are formulas[2].

A formula is thus a finite string, and there are infinitely many formulas.

To give meaning to a formula (as representing a Boolean function), we introduce the notion of evaluation by substitution. This means that if $\mathbf{a} = (a_1, a_2, \ldots a_n) \in \mathbf{B_2}^n$, then an evaluation of a formula $f(x_1, x_2, \ldots x_n)$ at the point $\mathbf{a}$ is obtained by substituting $x_i = a_i$ and then computing the expression thus obtained in $\mathbf{B_2}$. It is clear that each formula thus defines a unique function.

Conversely, given an arbitrary Boolean function f, it is always possible to obtain a formula for it. To see this, let f_{ON} be the set of elements in $\mathbf{B_2}^n$ on which f evaluates to 1 (this is called the ON-set of f). Now the set of atoms which are $\leq f$ consists of functions which map exactly one element in f_{ON} to 1, while all other elements in $\mathbf{B}_2^n$ are mapped to 0. Let $\mathbf{a} = (a_1, a_2, \ldots a_n)$ be an element of f_{ON}. Then suppose $f_{\mathbf{a}}$ is the atom $\leq f$ which maps $\mathbf{a}$ to 1. Consider the formula $y_1.y_2.\ldots.y_n$ where each $y_i = x_i$ if $a_i = 1$ or $y_i = \overline{x_i}$ if $a_i = 0$. It is easy to check that this formula evaluates to 1 exactly on $\mathbf{a}$. Such a formula is called a *min-term*. Thus every atom has a formula (a min-term). It follows that the sum of such min-terms corresponding to atoms $\leq f$ will yield a formula for f (Eq. 1). A formula for f is then obtained as a sum of minterms corresponding to the elements of f_{ON}. This formula is called the DNF (disjunctive normal form) formula for the function f.

An alternate formula can be obtained by starting with atoms $\leq \overline{f}$. If $\mathbf{a} = (a_1, a_2, \ldots a_n)$ is an element of $\overline{f_{ON}}$. Then suppose $f_{\mathbf{a}}$ is the atom $\leq \overline{f}$ which maps $\mathbf{a}$ to 1. Consider the formula $(y_1 + y_2 + \ldots + y_n)$ where each $y_i = \overline{x_i}$ if $a_i = 1$ or $y_i = x_i$ if $a_i = 0$. It is easy to check that this formula evaluates to 0 exactly on $\mathbf{a}$ and to 1 everywhere else. Thus the formula represents $\overline{f_{\mathbf{a}}}$. Such a formula is called a *max-term*. Thus, from Eq. 2, f can be written as product of such max-terms. This is called the CNF (conjunctive normal form) formula for f.

[1] sometimes we write this as $\neg A$.

[2] if the context is clear, we will drop the parentheses.

2 A Switching Algebra and the need for Formula Minimization

Consider the infinite set of formulas constructed above, with $+$, . operations and the 0, 1 elements. We say that two formulas are considered identical if they correspond to the same Boolean function (thus, each Boolean function defines an equivalence class of Boolean formulas). As a result the infinite set of formulas can be viewed as a finite set of equivalence classes, with each equivalence class corresponding to a Boolean function. The set of such equivalence classes is called the switching algebra on n variables which is isomorphic to the Boolean algebra of functions from $\mathbf{B_2}^n$ to $\mathbf{B}_2$. Putting it another way, a switching algebra gives a convenient way of representing and manipulating Boolean functions. Note that a switching algebra has a finite number of elements even though the number of formulas is infinite.

It is easy to see that all identities in a Boolean algebra can be converted to identities on elements of a switching algebra, because each such element can be interpreted as a function by substitution. For, example if A, B are formulas then

- $(A + B) = (B + A)$.
- $(A.B) = (B.A)$.
- $A.(B + C) = ((A.B) + (A.C))$.
- $\overline{A.B} = (\overline{A} + \overline{B})$.
- etc.

That is, the switching algebra itself can be interpreted as a Boolean algebra.

It is evident that the same Boolean function can have many formulas. Note that corresponding to a formula, there is a derivation tree which describes how the formula was constructed. If we have AND, OR, NOT gates available, then this derivation tree gives us a direct implementation of the Boolean function. Therefore, we are interested in *small* formulas: because we will be implementing such formulas using logic gates such as two-input AND, two-input OR and NOT-gates (what are logic gates?). The number of two input AND/OR gates needed to implement a formula is equal to the number of literals in the formula minus one...

Thus, a central problem in logic design is the following: Given a formula, find an equivalent formula which is as simple as possible (for example, has the smallest number of literals). This is a difficult problem for which an exact algorithm exists only in a couple of restricted cases which we discuss below.

3 Sum of Products Minimization

Given an $n-$variable function f, we can always write it as a sum of products.

$$p_1 + p_2 + \dots p_m$$

The number of minterms covered by a product is 2^{n-k}, where k is the number of literals occuring in the product. Thus, larger products have smaller formulas.

Thus, if we are interested in finding a small sum-of-products expression for f, we should look for large products inside f_{ON} and then cover f_{ON} in the best way possible. Typically, we are looking to use the minimum number of products to cover f (with a smaller number of literals being a tie-breaker). The method of Karnaugh maps as well as the tabular method are approaches which try to do this. The basic idea is

- Construct the set of prime implicants of the function (a prime implicant is a maximal product which is contained in f_{ON}).
- Choose the smallest subset of the set of prime implicants which will cover all elements of f_{ON}.

Both methods have been discussed in class and explained in the excellent text by Kohavi. Please read up.

4 Problem set

1. A NAND gate is a logic gate with two inputs (say $a, b \in \mathbf{B_2}$) which computes $\overline{a.b}$. Show that any Boolean function can be implemented using ony NAND gates.

2. Show that any Boolean function can be implemented using only 2 to 1 multiplexors (a three input gate whose inputs are $s, u0, u1$ and whose output is the function $s.u1 + \overline{s}.u0$). You are allowed to tie gate inputs to 1 or 0.

3. Find all the prime implicants for the following function:

$$(\overline{w}.(x+y)+x.\overline{y}+w.y).z$$

4. Suppose the function f_1 has the formula

$$f_1=(\overline{w}.(x+y)+x.\overline{y}+w.y).z$$

and the function f_2 has the formula

$$f_2=(\overline{p}.(q+r)+q.\overline{r}+q.r).s$$

Find the prime implicants of the function with formula $f_1.f_2$. Note that the functions f_1 and f_2 depend on disjoint sets of variables.

5. Consider $Z_5=\{0,1,2,3,4\}$, the set of integers modulo 5. Using AND, OR, NOT gates, implement a logic network which computes the square (modulo 5) of a number in Z_5 (if you use the standard coding using 3 bits, this will be a Boolean function with 3 outputs).