Maggie Johnson

Handout #XX
CS103A

Boolean Algebra

Key Topics

- Introduction
- Boolean Operators
- Boolean Expressions and Functions
- Boolean Identities
- More on Boolean Functions
- Functional Completeness
- Simplification of Boolean Functions
- Karnaugh Maps

Introduction

The circuits in computers have inputs (0 or 1) and produce outputs (0 or 1). Circuits can be constructed using any basic unit that has two different states, one for the 0 input/output, and one for the 1 input/output. Typically, such units are in the form of switches that can be either on or off.

In 1938, Claude Shannon showed how the rules of propositional logic could be used to design circuits (in his Master's thesis at MIT). These rules form the basis for **Boolean algebra**. A Boolean algebra is just the operations and rules used for working with the set {0,1} where 0 and 1 can take on different meanings. Propositional Logic (which he have been studying) is a Boolean algebra.

Boolean Operators

The main Boolean operators we will use are **complement**, **sum** and **product**. The complement of an element (which always has a value of 0 or 1) is denoted with a single quote: 0' = 1 and 1' = 0. The Boolean sum denoted by + or OR has the following values:

$$1 + 1 = 1$$
$$1 + 0 = 1$$
$$0 + 1 = 1$$
$$0 + 0 = 0$$

The Boolean product denoted by • or by AND has the following values:

$$1 \cdot 1 = 1$$
$$1 \cdot 0 = 0$$
$$0 \cdot 1 = 0$$
$$0 \cdot 0 = 0$$

Typically, the • is dropped so x • y = xy. The rules of precedence for these operators are: complement, product, sum. Therefore, the value of

$$\begin{aligned} 1 \cdot 0 + (0 + 1)' &= 1 \cdot 0 + 1' \\ &= 1 \cdot 0 + 0 \\ &= 0 + 0 \\ &= 0 \end{aligned}$$

Boolean Expressions and Functions

x is a **Boolean variable** if it can only assume the value of either 0 or 1. A **Boolean function** is a function whose domain is a set of n-tuples of 0's and 1's, and whose range is an element of the basic Boolean set {0,1}. We always display the values of a Boolean function in a truth table. A **Boolean expression** on the Boolean variables $\{x_1, x_2, ...,$

$x_n\}$ is an expression using those variables and the operations of a boolean algebra.

Every Boolean expression defines a Boolean function. The values of this function are obtained by substituting 0 and 1 for the variables in the expression. For example, we can define a Boolean expression xy + xy' by a Boolean function F(x,y) = xy + xy'. The values of this function are displayed in the table below - all we did was substitute all possible values for the variables.

x	y	xy	xy'	F(x,y)
0	0	0	0	0
0	1	0	0	0
1	0	0	1	1
1	1	1	0	1

The domain in this function is the 2-tuple which represents the values of x and y. The range is {0,1} in the last column. The n-tuple of a Boolean function is just the possible values of the variables. Two Boolean expressions are **equivalent** if they represent the same function (i.e., have the same truth table).

Boolean Identities

Identity	Name
(x')' = x	Involution Law
x + x' = 1 x • x' = 0	Complementarity
x + x = x x • x = x	Idempotent Laws
x + 0 = x x • 1 = x	Identity Laws
x + 1 = 1 x • 0 = 0	Dominance Laws
x + y = y + x xy = yx	Commutative Laws
x + (y + z) = (x + y) + z x(yz) = (xy)z	Associative Laws
x + yz = (x + y)(x + z) x(y + z) = xy + xz	Distributive Laws
(xy)' = x' + y' (x + y)' = x'y'	DeMorgans Laws
x + (xy) = x x(x + y) = x	Absorption Laws
x + x'y = x + y x(x' + y) = xy	Redundancy Laws
xy + x'z + yz = xy + x'z (x+y)(x'+z)(y+z) = (x+y)(x'+z)	Consensus Laws

There is a duality principle which applies to all Boolean algebras. In the definition of the identities above, we always include two parts that represents the dual identities. The only different is we interchange • and +, and 0 and 1. Any proven theorem is automatically true for the dual of the theorem.

In Boolean algebra, we can prove the identities using truth tables or using other identities. So, for example the distributive law is proven true by the last two columns of the following table:

x	y	z	y+z	xy	xz	x(y+z)	xy+xz
1	1	1	1	1	1	1	1
1	1	0	1	1	0	1	1

1	0	1	1	0	1	1	1
1	0	0	0	0	0	0	0
0	1	1	1	0	0	0	0
0	1	0	1	0	0	0	0
0	0	1	1	0	0	0	0
0	0	0	0	0	0	0	0

A proof of the absorption law $x(x + y) = x$ using identities:

$x(x + y)$	$= (x + 0)(x + y)$	identity law
	$= x + 0 \bullet y$	distributive law
	$= x + y \bullet 0$	commutative law
	$= x + 0$	dominance law
	$= x$	identity law

More on Boolean Functions

In preparation for using all the above information to build circuits, we need to be able to solve two basic problems:

1) Given a table of values for a Boolean function, how do we derive the corresponding Boolean expression?
2) Is there a smaller set of operators that can be used to represent a Boolean function?

First, question 1. Given the following table, how do we figure out the corresponding Boolean expression?

x	y	z	F	G
1	1	1	0	0
1	1	0	0	1
1	0	1	1	0
1	0	0	0	0
0	1	1	0	0
0	1	0	0	1
0	0	1	0	0
0	0	0	0	0

Notice that the only time that F is 1 is when x = z = 1 and y = 0. This translates to xy'z meaning that the expression xy'z has the value 1 if and only if x = y' = z = 1. G is 1 in two cases: x = y = 1 and z = 0, and x = z = 0 and y = 1. When we have to deal with two or more cases, we represent the sum of the two products: xyz' + x'yz'. This is the Boolean expression for G.

This illustrates a basic method for constructing an expression from the values of a Boolean function. A **minterm** of the Boolean variables $x_1, x_2, ..., x_n$ is a Boolean product $y_1y_2...y_n$ where $y_i = x_i$ or $y_i = x_i'$. A minterm has a value of 1 if and only if all the values of its variables are 1. So, if x = y = 1 and z = 0 then xyz' is the minterm that equals 1.

By taking Boolean sums of minterms, we can build up a Boolean expression that represents the values of a function represented by a table. The minterms in this sum correspond to those combinations of the values for which the function has a value of 1. This Boolean sum is sometimes called a **sum of products expansion** or **disjunctive normal form**.

There is an dual entity called a **maxterm** which is a **product of sums expansion** (**conjunctive normal form**). The table below shows the duality:

x	y	z	minterm	maxterm
0	0	0	x'y'z'	x+y+z
0	0	1	x'y'z	x+y+z'
0	1	0	x'yz'	x+y'+z
0	1	1	x'yz	x+y'+z'
1	0	0	xy'z'	x'+y+z
1	0	1	xy'z	x'+y+z'
1	1	0	xyz'	x'+y'+z
1	1	1	xyz	x'+y'+z'

For the following Boolean function F(x,y,z) = (x+z) y

x	y	z	x+z	(x+z) y
0	0	0	0	0
0	0	1	1	0
0	1	0	0	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1

xyz + xyz' + x'yz is the minterm expression; (x+y+z)(x+y+z')(x+y'+z)(x'+y+z)(x'+y+z') is the maxterm expression. Every Boolean function has both a minterm and maxterm expansion.

Functional Completeness

The other question posed in the previous section was: Do we need all three operators to express Boolean functions? As shown above, it's easy with {' + •} - since every Boolean function can be expressed with these three operators, we say the set is **functionally complete**. Is there a smaller set of functionally complete operators? There is a smaller set if one of the three operators can be expressed in terms of the other two. For example:

$$x + y = (x'y')'$$

which means we can get rid of the + operator (this is an application of DeMorgan's Law). So, the set of {' •} is functionally complete. In a similar way we can eliminate Boolean products:

$$xy = (x' + y')'$$

Is the set {+ •} functionally complete? There is no way to represent the Boolean function F(x) = x' using these two operators, so this set is not functionally complete.

Finally, is there a set of one operator that is functionally complete? Only if we define a new operator. We could define a **NAND** operator "|" as follows:

x	y	x | y
0	0	1
0	1	1
1	0	1
1	1	0

This set is functionally complete. To prove this, we know that the set {• '} is functionally complete so all we have to do is show that these two operators can be expressed using |:

x' = x | x
xy = (x | y) | (x | y)

A similar operator that is functionally complete is the **NOR** operator: 0

x	y	x 0 y
0	0	1
0	1	0
1	0	0
1	1	0

Simplification of Boolean Functions

We know that any Boolean function can be built from ANDs, ORs, and NOTs using minterm expansion. However, a practicing computer engineer will very rarely be satisfied with a minterm expansion, because as a rule, it requires more gates than necessary. The laws and identities of Boolean algebra will almost always allow us to simplify a minterm expansion. For example, the minterm expansion for a Boolean function f of three variables might be represented as follows (taking the values directly from the truth table):

$$f = x'y'z' + x'y'z + x'yz' + x'yz + xyz' + xyz$$

This would require a circuit with maximum gates: 12 ANDs, 5 ORs and 9 NOTs. Using the identities of Boolean algebra, this minterm expansion can be simplified considerably:

f	= x'y'z' + x'y'z + x'yz' + x'yz + xyz' + xyz	
	= x'y'(z' + z) + x'y(z' + z) + xy(z' + z)	distributive law
	= x'y' + x'y + xy	complementarity & identity
	= x'(y' + y) + xy	distributive law
	= x' + xy	complementarity & identity
	= x' + y	redundancy

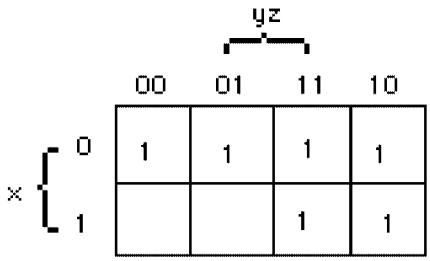
So, that big long minterm reduces down to x' + y which can be built with 1 OR and 1 NOT. This is an important point, but reducing minterms can require a lot of luck knowing which identities to apply when. Therefore, we will look at a very simple technique that usually leads to a significant simplification of minterms. It won't always produce the simplest form, but it's close enough for most engineers considering the difficulty of the alternative method.

Karnaugh Maps

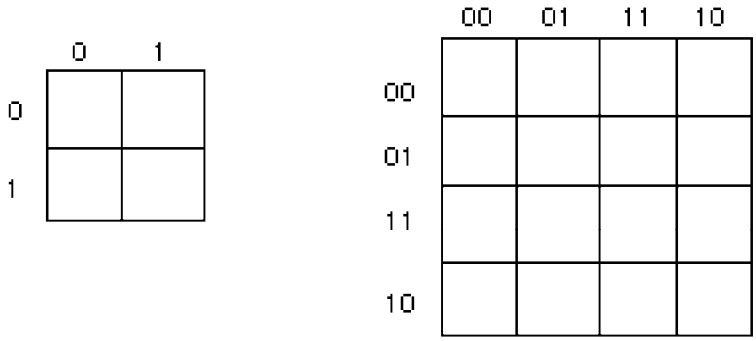
Karnaugh maps were invented by Maurice Karnaugh, a telecommunications engineer. He developed them at Bell Labs in 1953 while studying the application of digital logic to the design of telephone circuits. This method is typically used on Boolean functions of two, three or four variables - past that, it gets quite cumbersome and other techniques are frequently used.

A Karnaugh map is a 2-dim representation of the truth table for a Boolean function. For example:

x	y	z	f(x,y,z)
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

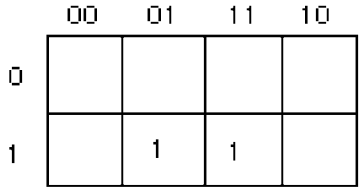


Each of the eight cells in the map corresponds to one of the eight possible combinations of x, y, & z. The templates for 2 and 4 variable Karnaugh maps are given below.

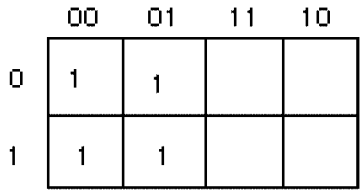


Once we have placed the 1's in the map, there is a simple procedure that we apply. Before doing that though, it is necessary to understand the basis for the procedure. The reason Karnaugh maps are useful is because certain simple Boolean functions have simple Karnaugh maps. These simple functions are called **product functions**; they are products of some or all of the variables *and* their complements. For example, x_1, $x_2'x_4$ and xyz are all product functions but $x_1 + x_2'$ and xy + zw are not.

Take the product function f(x,y,z) = xz. This function accepts two inputs 101 and 111. Its Karnaugh map:

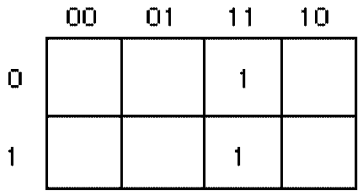


Notice that the 1's lie in a 1x2 rectangular block. Another example is g(x,y,z) = y'. This function accepts 000,001,100,101 (since for all the other possibilities y = 1 and therefore y' = 0). Its Karnaugh map:



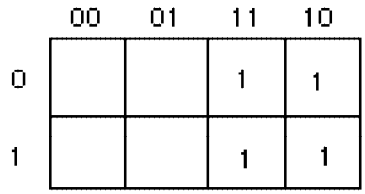
The minterms for this one lie in a 2x2 rectangular block. It turns out that every product function has a Karnaugh map whose minterms are confined to a block whose sides are 1, 2 or 4 cells long. Thus, it becomes important to recognize what product function is represented by a particular Karnaugh map.

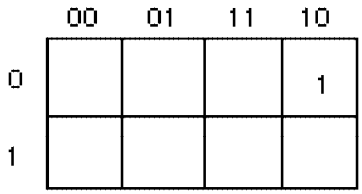
To do this, first write down the truth set (the set of combinations with a value of 1 from the map). For example:

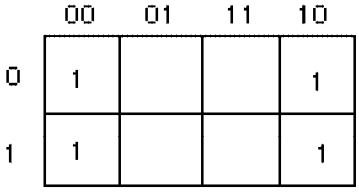


The truth set is {011, 111}. Next, we analyze the values of the truth set. If the variables are x, y, and z, we notice that x can be 0 or 1; while y and z can be only 1. We characterize this analysis as follows: {*11} where the "*" is a wildcard. If the unknown product function involved x, it would only accept inputs for which x assumed some particular value. Since x can be either 0 or 1, we conclude that x is not involved. y, on the other hand, must be involved since y = 1 for all inputs. So y must be a term (not y'); the same is true for z. The product function is yz.

Just for practice, what are the product functions associated with the following Karnaugh maps?





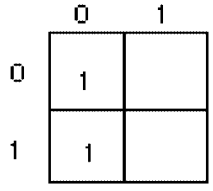


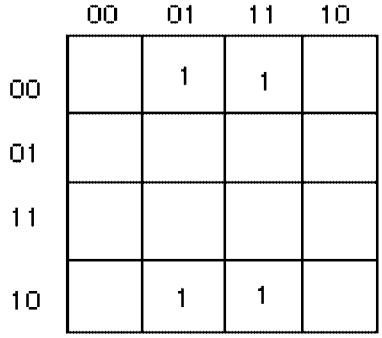
1) truth set: {011, 010, 111, 110} = {*1*}. The product function never includes wildcard variables; the function is: y.

2) truth set: {010} so the function is x'yz'.

3) truth set: {000, 100, 010, 110} = {**0} = z'. Notice that the rectangular block wraps around.

Finally, try some 2 and 4 variable maps:



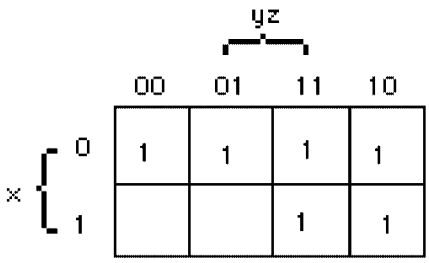


1) truth set = {00, 10} = {*0} = y'

2) (variables are wxyz): truth set = {0001, 0011, 1001, 1011} = {*0*1} = x'z.

The reason this is all so important is any Boolean function with the same Karnaugh map as, for example, the ones above, can be represented by the same expression. Another expression with the same Karnaugh map as the two-variable one above: x'y' + xy' + x'yxy; therefore, we can represent this as y' and build a circuit as such.

Unfortunately, not all expressions work out to be equivalent to product function Karnaugh maps. The first example we looked at did not break down into a 1, 2, or 4 cell rectangular block.



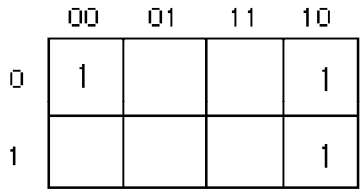
We can handle these too by decomposing the blocks into a union of two product blocks, one representing the varialbe x' (the ones across the top indicate a 0 value for x) and one representing y (the two ones on the bottom indicate a 1 value for y). The function is x' + y, which is exactly the conclusion we came to earlier. All we are doing is ORing the two product functions together. We did not bother to show the truth sets for these two blocks ({000, 001, 011, 010} = {0**} = x') - you will reach the same conclusion.

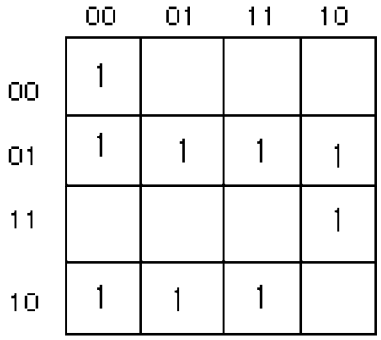
This technique can be applied to any Boolean function. The idea is to cover the terms using as few product blocks as possible. Then, write the function as a sum of these product blocks. It's important that the blocks are as large as possible or you may not end up with the simplest expression. For example, in the map above, we could have used three blocks: one horizantal for the first two ones in the top row, and then two vertical blocks. This would give the expression: x'y' + yz + yz'. This is equivalent, but inferior.

What if you block off the first two ones in the first row and then the four ones across the two rows? The expression is: x'y' + y; again it's equivalent but not as simple as the first. (An application of the redundancy law will simplify this further). It's important to carefully pick the blocks or you may not end up with the simplest expression.

The product blocks that make up a representation for an expression are called **implicants**. x'y' , yz and yz' are all implicants the expression x'y' + yz + yz'. Implicants that cover as many cells of the map as possible are called **prime implicants**. The rule of the game, therefore, is *cover the minterms with as few prime implicants as possible*.

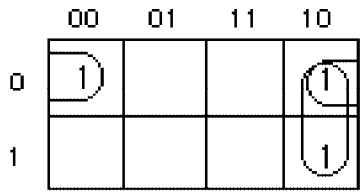
As additional practice, simplify the following functions represented by Karnaugh maps:

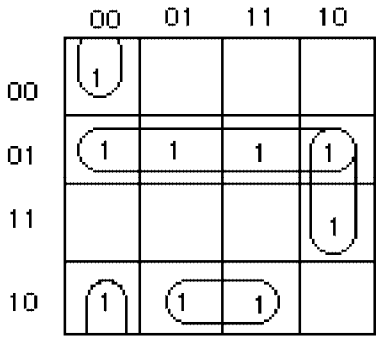




1) x'z' + yz'
2) w'x + xyz' + wx'z x'y'z'

Notice that you can overlap the blocks if ncessary:





Karnaugh maps are certainly easier to deal with than using identities to simpify Boolean functions. However, the trick is finding the right set of blocks to get the simplest expression. This takes practice; there is no simple rule that tells how it should be done.

Bibliography and Historical Notes

* The study of deduction in logic dates back to Aristotle. Boole developed the algebra of propositions, and it is from this work that Boolean algebra comes.

G. Boole, *An Investigation of the Laws of Thought*, 1854, reprinted by New York: Dover Press, 1958.

* For more on Boolean algebra:

S. Epp, *Discrete Mathematics with Applications*, Belmont, CA: Wadsworth, 1990.

R. Grimaldi, *Discrete and Combinatorial Mathematics, 2nd ed*., Reading, MA: Addison- Wesley, 1989.

F.E. Hohn, *Applied Boolean Algebra, 2nd ed*., New York: Macmillan, 1966.

Z. Kohavi, *Switching and Finite Automata Theory, 2nd ed*., New York: McGraw-Hill, 1978.

K. Rosen, *Discrete Mathematics and its Applications 4nd Ed.*, New York: McGraw-Hill, 1999.

* Karnaugh maps were introduced in:

M. Karnaugh, "The Map Method for Synthesis of Combinatorial Logic Circuits," *Transactions of the AIEE*, 72 (1953), 593-599.