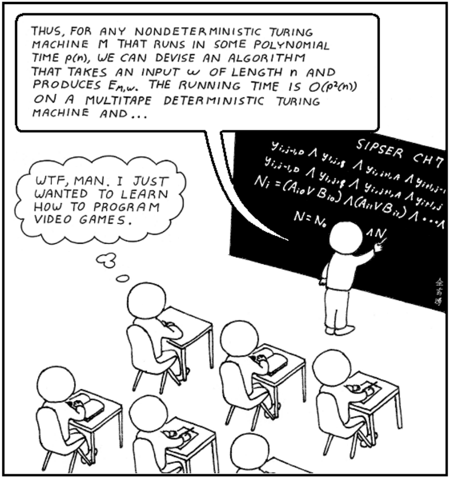
COMS12200 lecture: week #2



---

http://abstrusegoose.com/206

- (Fairly) reasonable question(s):
  1. "I thought this was CS, not Maths!", and
  2. "why does this unit duplicate material in *other* units?".

- (Fairly) reasonable question(s):
  1. "I thought this was CS, not Maths!", and
  2. "why does this unit duplicate material in *other* units?".
- Answer: it isn't, and it doesn't (well, not *too* much) ... note
  - axiomatic manipulation can be practically motivated, e.g., **optimisation**,
  - theoretical tools, such as KWuniversality, often have major practical uses or implications, and
  - Boolean algebra has wider application than circuit design.

- Question: simplify the Boolean expression

$$(\neg(a \vee b) \wedge \neg(c \vee d \vee e)) \vee \neg(a \vee b)$$

  into a form that contains the fewest operators possible.

- Question: simplify the Boolean expression

$$(\neg(a \vee b) \wedge \neg(c \vee d \vee e)) \vee \neg(a \vee b)$$

into a form that contains the fewest operators possible.

- Solution #1: more steps.

$$\begin{array}{clllllr}
 & (\neg(a \vee b) & \wedge & \neg(c \vee d \vee e)) & \vee & \neg(a \vee b) & \\
= & ((\neg a \wedge \neg b) & \wedge & \neg(c \vee d \vee e)) & \vee & \neg(a \vee b) & \text{(de Morgan)} \\
= & ((\neg a \wedge \neg b) & \wedge & (\neg c \wedge \neg d \wedge \neg e)) & \vee & \neg(a \vee b) & \text{(de Morgan)} \\
= & ((\neg a \wedge \neg b) & \wedge & (\neg c \wedge \neg d \wedge \neg e)) & \vee & (\neg a \wedge \neg b) & \text{(de Morgan)} \\
= & (\neg a \wedge \neg b) & \vee & ((\neg a \wedge \neg b) & \wedge & (\neg c \wedge \neg d \wedge \neg e)) & \text{(commutativity)} \\
= & (\neg a \wedge \neg b) & & & & & \text{(absorption)} \\
= & \neg(a \vee b) & & & & & \text{(de Morgan)}
\end{array}$$

- Question: simplify the Boolean expression

$$(\neg(a \vee b) \wedge \neg(c \vee d \vee e)) \vee \neg(a \vee b)$$

into a form that contains the fewest operators possible.

- Solution #2: less steps.

$$\begin{array}{lllllll} & (\neg(a \vee b) & \wedge & \neg(c \vee d \vee e)) & \vee & \neg(a \vee b) & \\ = & \neg(a \vee b) & \vee & (\neg(a \vee b) & \wedge & \neg(c \vee d \vee e)) & \text{(commutativity)} \\ = & \neg(a \vee b) & & & & & \text{(absorption)} \end{array}$$

- Question: simplify the Boolean expression

$$(a \wedge b \wedge c) \vee (\neg a \wedge b) \vee (a \wedge b \wedge \neg c)$$

into a form that contains the fewest operators possible.

- Question: simplify the Boolean expression

$$(a \wedge b \wedge c) \vee (\neg a \wedge b) \vee (a \wedge b \wedge \neg c)$$

into a form that contains the fewest operators possible.

- Solution:

$$\begin{array}{llllllr}
 & (a \wedge b \wedge c) & \vee & (\neg a \wedge b) & \vee & (a \wedge b \wedge \neg c) & \\
= & (a \wedge b \wedge c) & \vee & (a \wedge b \wedge \neg c) & \vee & (\neg a \wedge b) & \text{(commutativity)} \\
= & (a \wedge b) & \wedge & (c \vee \neg c) & \vee & (\neg a \wedge b) & \text{(distribution)} \\
= & (a \wedge b) & \wedge & 1 & \vee & (\neg a \wedge b) & \text{(inverse)} \\
= & (a \wedge b) & & & \vee & (\neg a \wedge b) & \text{(identity)} \\
= & b & \wedge & (a \vee \neg a) & & & \text{(distribution)} \\
= & b & \wedge & 1 & & & \text{(inverse)} \\
= & b & & & & & \text{(identity)}
\end{array}$$

Point #1: axiomatic manipulation can be practically motivated

Quote

If I designed a computer with 200 chips, I tried to design it with 150. And then I would try to design it with 100. I just tried to find every trick I could in life to design things real tiny.

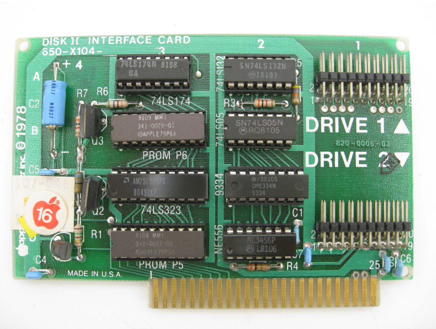
– S. Wozniak

Quote

So I took 20 chips off their board; I bypassed 20 of their chips.

– S. Wozniak

Boolean algebra: theory $\Longleftrightarrow$ practice (point #1, 4)



---

http://en.wikipedia.org/wiki/File:Shugart_SA400.jpg

Boolean algebra: theory $\Longleftrightarrow$ practice (point #1, 5)

Listing (Python)

```python
from sympy                  import *
from sympy.logic.boolalg    import bool_equal, simplify_logic

a, b, c, d, e = symbols( "a b c d e" )

f = ( b & ~a ) | ( a & ~b )
g = ( a | b ) & ~( a & b )

print f
print g
print bool_equal( f, g )
print

f = ( ~( a | b ) & ~( c | d | e ) ) | ~( a | b )

print f
print simplify_logic( f )
print

f = ( a & b & c ) | ( ~a & b ) | ( a & b & ~c )

print f
print simplify_logic( f )
print
```

Boolean algebra: theory $\iff$ practice (point #2, 1)

- We *can* add to our existing suite of logical operators (the result is often termed a **derived operator**) ...
- ... two useful additions are as follows:
  - "NOT-AND" or **NAND**, such that

    $$x \overline{\wedge} y \equiv \neg(x \wedge y)$$

    so

x	y	$x \overline{\wedge} y$
0	0	1
0	1	1
1	0	1
1	1	0

    and
  - "NOT-OR" or **NOR**, such that

    $$x \overline{\vee} y \equiv \neg(x \vee y)$$

    so

x	y	$x \overline{\vee} y$
0	0	1
0	1	0
1	0	0
1	1	0

- Question: haven't we already got enough ... this is *already* quite difficult!
- Answer: NAND and NOR turn out to be **universal**, e.g.,

$$
\begin{array}{lcccc}
\neg x & \equiv & x \overline{\wedge} x & & \\
x \wedge y & \equiv & (x \overline{\wedge} y) & \overline{\wedge} & (x \overline{\wedge} y) \\
x \vee y & \equiv & (x \overline{\wedge} x) & \overline{\wedge} & (y \overline{\wedge} y)
\end{array}
$$

which we can see from

x	y	$x \overline{\wedge} y$	$x \overline{\wedge} x$	$y \overline{\wedge} y$	$(x \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} y)$	$(x \overline{\wedge} x) \overline{\wedge} (y \overline{\wedge} y)$
0	0	1	1	1	0	0
0	1	1	1	0	0	1
1	0	1	0	1	0	1
1	1	0	0	0	1	1

- Question: haven't we already got enough ... this is *already* quite difficult!
- Answer: NAND and NOR turn out to be **universal**, e.g.,

$$
\begin{array}{lcl}
\neg x & \equiv & x \overline{\wedge} x \\
x \wedge y & \equiv & (x \overline{\wedge} y) \;\overline{\wedge}\; (x \overline{\wedge} y) \\
x \vee y & \equiv & (x \overline{\wedge} x) \;\overline{\wedge}\; (y \overline{\wedge} y)
\end{array}
$$

which we can see from

x	y	$x \overline{\wedge} y$	$x \overline{\wedge} x$	$y \overline{\wedge} y$	$(x \overline{\wedge} y) \overline{\wedge} (x \overline{\wedge} y)$	$(x \overline{\wedge} x) \overline{\wedge} (y \overline{\wedge} y)$
0	0	1	1	1	0	0
0	1	1	1	0	0	1
1	0	1	0	1	0	1
1	1	0	0	0	1	1

- Eureka: computation of *any* Boolean function can be expressed using *one* simple building block component.

- Question: translate

$$x \wedge (y \vee z)$$

into a version using NAND only.

- Question: translate

$$x \wedge (y \vee z)$$

into a version using NAND only.

- Solution #1: apply the identities naively to get

$$\begin{array}{ll} & x \wedge (y \vee z) \\ = & x \wedge ((y \overline{\wedge} y) \overline{\wedge} (z \overline{\wedge} z)) \\ = & (x \overline{\wedge} ((y \overline{\wedge} y) \overline{\wedge} (z \overline{\wedge} z))) \overline{\wedge} (x \overline{\wedge} ((y \overline{\wedge} y) \overline{\wedge} (z \overline{\wedge} z))) \end{array}$$

Boolean algebra: theory $\iff$ practice (point #2, 3)

- **Question**: translate

$$x \wedge (y \vee z)$$

into a version using NAND only.

- **Solution #2**: be a bit smarter by writing

$$\begin{array}{cl} & x \wedge (y \vee z) \\ = & x \wedge ((y \overline{\wedge} y) \overline{\wedge} (z \overline{\wedge} z)) \\ = & t \overline{\wedge} t \end{array}$$

where $t = x \overline{\wedge} ((y \overline{\wedge} y) \overline{\wedge} (z \overline{\wedge} z))$ is a common sub-expression [4].

Definition (**Boolean algebra vs. C programs**)

Boolean algebra plays a role in many C constructs:

<table>
<tr><td>r is x</td><td>$\equiv$</td><td>$r = x$</td><td>$\equiv$</td><td><code>r = x</code></td><td>$\equiv$</td><td><code>r = x</code></td></tr>
<tr><td>r is NOT x</td><td>$\equiv$</td><td>$r = \neg x$</td><td>$\equiv$</td><td><code>r = !x</code></td><td>$\cong$</td><td><code>r = ~x</code></td></tr>
<tr><td>r is x NAND y</td><td>$\equiv$</td><td>$r = x \overline{\wedge} y$</td><td>$\equiv$</td><td><code>r = !(x && y)</code></td><td>$\cong$</td><td><code>r = ~(x & y)</code></td></tr>
<tr><td>r is x NOR y</td><td>$\equiv$</td><td>$r = x \overline{\vee} y$</td><td>$\equiv$</td><td><code>r = !(x || y)</code></td><td>$\cong$</td><td><code>r = ~(x | y)</code></td></tr>
<tr><td>r is x AND y</td><td>$\equiv$</td><td>$r = x \wedge y$</td><td>$\equiv$</td><td><code>r = x && y</code></td><td>$\cong$</td><td><code>r = x & y</code></td></tr>
<tr><td>r is x OR y</td><td>$\equiv$</td><td>$r = x \vee y$</td><td>$\equiv$</td><td><code>r = x || y</code></td><td>$\cong$</td><td><code>r = x | y</code></td></tr>
<tr><td>r is x XOR y</td><td>$\equiv$</td><td>$r = x \oplus y$</td><td></td><td></td><td>$\cong$</td><td><code>r = x ^ y</code></td></tr>
</table>

Note that the latter columns capture

- **decisional** operators, e.g., && : $\mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$, and
- **computational** operators, e.g., & : $\mathbb{B}^n \times \mathbb{B}^n \rightarrow \mathbb{B}^n$.

Boolean algebra: theory ⟺ practice (point #3, 2)

Listing (C)

```
void condition_v1( int a, int b, int c, int d ) {
  if( ( ( a + b ) > c ) && ( ( d % 2 ) == 0 ) ) {
    //         ( a + b is     >  c ) and     ( d is     even )

    printf( "if -> true  : %d %d %d %d\n", a, b, c, d );
  }
  else {
    //   not ( ( a + b is     >  c ) and     ( d is     even ) )
    //
    // = not   ( a + b is     >  c ) or  not ( d is     even )
    // =       ( a + b isn't >  c ) or      ( d isn't even )
    // =       ( a + b is     <= c ) or      ( d is     odd  )

    printf( "if -> false : %d %d %d %d\n", a, b, c, d );
  }
}
```

Boolean algebra: theory $\iff$ practice (point #3, 2)

Listing (C)

```
void condition_v2( int a, int b, int c, int d ) {
  if( ( ( a + b ) > c ) && ( d = 3 ) ) {
    printf( "if -> true  : %d %d %d %d\n", a, b, c, d );
  }
  else {
    printf( "if -> false : %d %d %d %d\n", a, b, c, d );
  }
}
```

Boolean algebra: theory $\Longleftrightarrow$ practice (point #3, 2)

Listing (C)

```
void condition_v3( int a, int b, int c, int d ) {
  if( ( d = 3 ) && ( ( a + b ) > c ) ) {
    printf( "if -> true  : %d %d %d %d\n", a, b, c, d );
  }
  else {
    printf( "if -> false : %d %d %d %d\n", a, b, c, d );
  }
}
```

Boolean algebra: theory $\Longleftrightarrow$ practice (point #4, 1)

- Question: imagine we have a Boolean expression (or circuit)

$$f(x_0, x_1, \ldots x_{n-1})$$

and manipulate it into a different form

$$g(x_0, x_1, \ldots x_{n-1})$$

e.g., optimise it: we want to know if there is a bug in g.

- Question: imagine we have a Boolean expression (or circuit)

$$f(x_0, x_1, \ldots x_{n-1})$$

  and manipulate it into a different form

$$g(x_0, x_1, \ldots x_{n-1})$$

  e.g., optimise it: we want to know if there is a bug in g.
- Solution(s):
  1. *prove* that f and g are equivalent,
  2. try all 2^n possible assignments, and see if f ever differs from g, or
  3. use a (carefully defined) instance of **SAT**.

Boolean algebra: theory $\Longleftrightarrow$ practice (point #4, 2)

Definition (SAT)

The **Boolean satisfiability problem** (or **SAT**) is a decision problem. Consider some Boolean function

$$f(x_0, x_1, \ldots x_{n-1}).$$

which defines a SAT **instance**. The problem is to decide whether or not an assignment to said variables (i.e., $x_i \in \{0, 1\}$ for $0 \leq i < n$) exists st. f is **satisfiable** (i.e., we have $f(x_0, x_1, \ldots x_{n-1}) = 1$).

- Question: decide whether

$$f(a,b) = (b \wedge \neg a) \vee (a \wedge \neg b)$$

$$\equiv$$

$$g(a,b) = (a \vee b) \wedge \neg(a \wedge b).$$

- **Question**: decide whether

$$f(a,b) = (b \wedge \neg a) \vee (a \wedge \neg b)$$

$$\equiv$$

$$g(a,b) = (a \vee b) \wedge \neg(a \wedge b).$$

- **Solution #1**: manipulation via axioms, e.g.,

$$\begin{array}{rlcll}
f(a,b) & = (b \wedge \neg a) & \vee & (a \wedge \neg b) & \\
 & = (b \wedge \neg a) \vee 0 & \vee & (a \wedge \neg b) \vee 0 & \text{(identity)} \\
 & = (b \wedge \neg a) \vee (b \wedge \neg b) & \vee & (a \wedge \neg b) \vee (a \wedge \neg a) & \text{(inverse)} \\
 & = (b \wedge (\neg a \vee \neg b)) & \vee & (a \wedge (\neg b \vee \neg a)) & \text{(distribution)} \\
 & = ((\neg a \vee \neg b) \wedge b) & \vee & ((\neg a \vee \neg b) \wedge a) & \text{(commutativity)} \\
 & = (\neg a \vee \neg b) \wedge (a \vee b) & & & \text{(distribution)} \\
 & = (a \vee b) \wedge (\neg a \vee \neg b) & & & \text{(commutativity)} \\
 & = (a \vee b) \wedge \neg(a \wedge b) & & & \text{(de Morgan)} \\
 & = g(a,b) & & &
\end{array}$$

- Question: decide whether

$$f(a,b) = (b \wedge \neg a) \vee (a \wedge \neg b)$$

$$\equiv$$

$$g(a,b) = (a \vee b) \wedge \neg(a \wedge b).$$

- Solution #1: brute-force enumeration, i.e.,

a	b	$b \wedge \neg a$	$a \wedge \neg b$	$f(a,b)$	$a \vee b$	$\neg(a \wedge b)$	$g(a,b)$
0	0	0	0	0	0	1	0
0	1	1	0	1	1	1	1
1	0	0	1	1	1	1	1
1	1	0	0	0	1	0	0

- Question: decide whether

$$f(a,b) = (b \wedge \neg a) \vee (a \wedge \neg b)$$

$$\equiv$$

$$g(a,b) = (a \vee b) \wedge \neg(a \wedge b).$$

- Solution #3: define

$$h(x_0, x_1, \ldots x_{n-1}) = f(x_0, x_1, \ldots x_{n-1}) \oplus g(x_0, x_1, \ldots x_{n-1})$$

and use this as a SAT instance: if the SAT instance is satisfiable, this yields a **test vector** we can use to debug g.

Boolean algebra: theory ⟺ practice (point #4, 4)

Listing (Python)

```
from sympy                  import *
from sympy.logic.inference import satisfiable

a, b, c, d, e = symbols( "a b c d e" )

f = ( a & ~a )

print f
print satisfiable( f )
print

f = ( a & b & c ) | ( ~a & b ) | ( a & b & ~c )

print f
print satisfiable( f )
print

f = ( b & ~a ) | ( a & ~b )
g = ( a | b ) & ~( a & b )

print f
print g
print satisfiable( f ^ g )
print
```

Conclusions

Quote

In theory there is no difference between theory and practice. In practice there is.

– L.P. “Yogi” Berra

References and Further Reading

[1] Wikipedia: Boolean data type.
http://en.wikipedia.org/wiki/Boolean_data_type.

[2] Wikipedia: Boolean satisfiability problem.
http://en.wikipedia.org/wiki/Boolean_satisfiability_problem.

[3] Wikipedia: Circuit complexity.
http://en.wikipedia.org/wiki/Circuit_complexity.

[4] Wikipedia: Common sub-expression elimination.
http://en.wikipedia.org/wiki/Common_subexpression_elimination.

[5] Wikipedia: Short-circuit evaluation.
http://en.wikipedia.org/wiki/Short-circuit_evaluation.