Boolean Functions, Boolean Maps, and Boolean Circuits

Klaus Pommerening
Fachbereich Physik, Mathematik, Informatik
der Johannes-Gutenberg-Universität
Saarstraße 21
D-55099 Mainz

February 2, 2003—English version August 30, 2003
last change March 6, 2021

1 Elementary Operations on Bits

On the lowest software level computers process bits or groups of bits. Examples of such groups are bytes that usually consist of 8 bits, or "words", usually 32 or 64 bits, depending on the processor architecture.

Bits have a logical interpretation as truth values "true" (T) or "false" (F). They also have an algebraic interpretation as values 0 (corresponding to F) or 1 (corresponding to T). As mathematical objects they are elements of the two element set $\{0, 1\}$, denoted by $\mathbb{F}_2$. This notation comes from the algebraic context:

Consider the residue class ring of $\mathbb{Z}$ modulo 2. This ring has two elements and is a field since 2 is a prime number. Addition in this field is the same as the logical operation XOR, multiplication is the same as the logical operation AND, see Table 1.

The algebraic structure as field is of fundamental importance in Cryptography. Therefore, as usual in Algebra, we use the notation $\mathbb{F}_q$ for finite fields where q is the number of elements (often also written as $\mathrm{GF}(q)$ for "Galois Field"). In this context we also use the algebraic symbols $+$ (for XOR), and $\cdot$ (for AND) for the operations, and often omit the multiplication dot. Cryptographers sometimes like to use the symbols $\oplus$ and $\otimes$ that unfortunately have a quite different meaning in Mathematics (direct sum or tensor product of vector spaces). We use these circled symbols only in diagrams.

logical					**algebraic**			
bits		operation			bits		operation	
x	y	OR	AND	XOR	x	y	$+$	$\cdot$
F	F	F	F	F	0	0	0	0
F	T	T	F	T	0	1	1	0
T	F	T	F	T	1	0	1	0
T	T	T	T	F	1	1	0	1

Table 1: The elementary operations on bits

2 Description of Boolean Functions

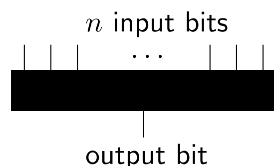
Let's start with the naive definition: A **Boolean function** is a rule (or a formula or an algorithm) that takes a certain number of bits and produces a new bit. Before making this definition mathematically precise we try to depict it in an appealing manner.

A first simple example is the function AND: It takes two bits and produces a new bit by the logical operation AND, see Table 1.

For a slightly more complicated example we consider the function f_0 that takes three bits x_1, x_2 und x_3 and produces the value

$$f_0(x_1, x_2, x_3) = x_1 \text{ AND } (x_2 \text{ OR } x_3). \tag{1}$$

An illustration of an (abstract) Boolean function is a "black box":



What happens inside this "black box" has different specifications:

- **mathematically** by a formula,
- **informatically** by an algorithm,
- **technically** by a circuit (or plugging diagramm),
- **pragmatically** by a truth table (the value table of the function).

For the sample function f_0 the formula is in the definition (1). The algorithm likewise is conveniently described by this formula because it has no branch points or conditional instructions. A visualization of f_0 as a circuit is in Figure 1. The truth table simply lists the values of f_0 for each input triple, see Table 2. (It is called "truth table" because it tells whether the complete expression is true (= 1) or false (= 0) depending on the input bits.)

The connection between logical calculus and electric circuits essentially goes back to Shannon.

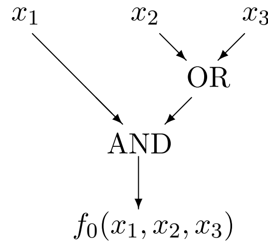


Figure 1: Circuit for f_0

x_1	x_2	x_3	$f_0(x_1, x_2, x_3)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Table 2: Truth table for f_0

3 The Number of Boolean Functions

The truth table of f_0 suggests a simple enumeration of all Boolean functions: For three variables there are $8 = 2^3$ different input triples—each single input bit can assume the values 0 or 1 independently of the other two bits. Furthermore a Boolean function f can assume the values 0 or 1 for each triple independently of the other seven triples. Hence there are $256 = 2^8$ Boolean functions of three variables.

The general formula is:

Theorem 1 *There are exactly 2^{2^n} different Boolean functions of n variables.*

For four variables this number is $2^{16} = 65536$. The number grows super-exponentially with n, even the exponent grows exponentially.

For a list of all 16 Boolean functions of two variables see Table 3 in Section 7.

4 Bitblocks and Boolean functions

For arrangements of bits there are different denotations depending on the context: vector, list, (n-) tuple, ... For certain special sizes we even have special names such as bytes (for 8 bits), words (for 32 or 64 bits depending on the processor architecture). In this text we prefer the denotation "bitblock" that is common in Cryptography. A **bitblock** of length n is an ordered list $(x_1, \ldots, x_n)$ of bits. There are eight different bitblocks of length 3. Here they are:

$$(0,0,0),(0,0,1),(0,1,0),(0,1,1),(1,0,0),(1,0,1),(1,1,0),(1,1,1).$$

Sometimes we write them as bitstrings without parantheses or commas:

$$000, 001, 010, 011, 100, 101, 110, 111,$$

sometimes as columns ($n \times 1$-matrices) when the interpretation as vector is the main aspect. Often we abbreviate $(x_1, \ldots, x_n)$ by x. The 2^n different bitblocks of length n form the cartesian product $\mathbb{F}_2^n = \mathbb{F}_2 \times \cdots \times \mathbb{F}_2$. This has a "natural" structure as vector space: We can add bitblocks x and $y \in \mathbb{F}_2^n$, and multiply them by scalars $a \in \mathbb{F}_2$:

$$(x_1, \ldots, x_n) + (y_1, \ldots, y_n) = (x_1 + y_1, \ldots, x_n + y_n),$$

$$a \cdot (x_1, \ldots, x_n) = (a \cdot x_1, \ldots, a \cdot x_n).$$

Here is the mathematically exact definition:

Definition A **Boolean function of n variables** is a map

$$f \colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2.$$

We denote the set of all Boolean functions on $\mathbb{F}_2^n$ by $\mathcal{F}_n$. By Theorem 1 it has 2^{2^n} elements.

Convention When we describe a Boolean function by its truth table in general we order it lexicographically for $x \in \mathbb{F}_2^n$, as seen in the example above. This order is the natural order of the integers $a = 0, \ldots, 2^n - 1$ in binary representation

$$a = x_1 \cdot 2^{n-1} + \cdots + x_{n-1} \cdot 2 + x_n$$

corresponding to the bitblocks $(x_1, \ldots, x_n) \in \mathbb{F}_2^n$.

5 Logical Expressions and Conjunctive Normal Form

The mathematical description of Boolean functions—by formulas—essentially follows two ways:

- Logic expresses Boolean functions by disjunctions (the operation OR, also written $\vee$), conjunctions (the operation AND, also written $\wedge$), and negations (the operation NOT, also written $\neg$). Compositions of these operations are called **logical expressions**.
- Algebra expresses Boolean functions by additions $+$ and multiplications $\cdot$ in the field $\mathbb{F}_2$. Compositions of these operations are called **(binary) polynomial expressions**.

We'll soon see that both ways lead to a description of all Boolean functions, and that in both cases additional requirements are possible leading to so called normal forms.

For cryptologic purposes the algebraic form seems to be somewhat more useful due to its structure (that is yet to explore). On the other hand the logical form leads to an easy implementation in hardware by circuits because the elementary Boolean operations have direct realizations in logic gates.

In this text the logical form plays a minor role. Therefore we state the following result without proof. The weaker statement of representability by logical operations (without normalization) will come out as a side result in Section 7, see Theorem 5.

Theorem 2 *Each Boolean function f of n variables $x_1, \ldots, x_n$ admits a representation (conjunction) as*

$$f(x) = s_1(x) \wedge \ldots \wedge s_r(x)$$

with a suitable r where the $s_j(x)$ for $j = 1, \ldots, r$ each have the form (disjunction)

$$s_j(x) = t_{j1}(x) \vee \ldots \vee t_{jn_j}(x)$$

with a certain number n_j of terms $t_{jk}(x)$ ($j = 1, \ldots, r$ and $k = 1, \ldots, n_j$) that are of the form x_i (input bit) or $\neg x_i$ (negated input bit) for a suitable index i.

In particular $n_j \leq n$ for $j = 1, \ldots, r$. Each individual input bit x_i occurs in each of the $t_{jk}(x)$ either directly or negated or not at all.

In other words: We can build an arbitrary Boolean function by forming some expressions (the $s_j(x)$) by ORing some of the input bits or negations thereof, and then composing these expressions by AND ("conjunction of disjunctions"). This "normal form" cleanly separates AND- and

OR-operations into two layers without further mixing. The defining equation $f_0(x_1, x_2, x_3) = x_1 \wedge (x_2 \vee x_3)$ of our example f_0 from Section 2 is already in "conjunctive" form, but not when written in expanded form $f_0(x_1, x_2, x_3) = (x_1 \wedge x_2) \vee (x_1 \wedge x_3)$. This example has no negated input bits. But Table 3 contains some.

A Boolean function is in **conjunctive normal form (CNF)** if it has the form of Theorem 2. This form is not unique. Without further explanation we remark that the CNF may be further refined to a "canonical CNF" that gives some kind of uniqueness. In analogy there is a disjunctive normal form (**DNF**) ("disjunction of conjunctions").

6 Polynomial Expressions and Algebraic Normal Form

For (binary) polynomial expressions in the variables $x_1, \ldots, x_n$, such as $x_1^2 x_2 + x_2 x_3 + x_3^2$, we need as coefficients the constants 0 and 1 only since we operate in the field $\mathbb{F}_2$, and we need not write down these coefficents explicitly. Furthermore we observe that $a^2 = a$ for all elements $a \in \mathbb{F}_2$ (note $0^2 = 0$ and $1^2 = 1$) and so $a^e = a$ for all exponents $e \geq 1$. This leads to another simplification of binary polynomial expressions: The variables $x_1, \ldots, x_n$ occur at most with exponent 1. Thus an equivalent expression of our sample polynomial is $x_1 x_2 + x_2 x_3 + x_3$. Another example: $x_1^3 x_2 + x_1 x_2^2 = x_1 x_2 + x_1 x_2 = 0$.

In general a **monomial expression** (or simply "monomial") has the form

$$x^I := \prod_{i \in I} x_i \quad \text{with a subset } I \subseteq \{1, \ldots, n\},$$

in other words, it is the product of some of the variables where the subset I describes the choice of "some of the variables". An example with $n = 3$ illustrates this:

$$I = \{2, 3\} \Longrightarrow x^I = x_2 x_3,$$

There are exactly 2^n of such monomial expressions—as many as subsets we can extract from an n-element set, or form partial products from n potential factors. The empty set corresponds to the product of 0 factors, and this we consider, as common, as being equal to 1. Thus:

$$I = \emptyset \Longrightarrow x^I = 1.$$

A monomial expression admits a direct interpretation as a Boolean function. We don't yet know whether these functions are different (but we'll see it immediately).

A polynomial expression is a sum of monomial expressions (as coefficients we only need 0 or 1 because we are in the binary case). Thus the most general

(binary) polynomial expression has the form

$$\sum_{I\subseteq\{1,\ldots,n\}} a_I x^I$$

where all coefficients a_I are 0 or 1. This means that we have to take the sum over a subset of all 2^n possible monomial expressions. The number of possibilities is 2^{2^n}. The Boolean functions generated by this method are all different, but we have to prove this fact. As a first step we prove that we get all Boolean functions in this way.

Theorem 3 *Let $f : \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ be a Boolean function. Then there are coeffizients $a_I \in \mathbb{F}_2$ (hence $= 0$ oder 1) where I runs through all subsets of $\{1, \ldots, n\}$ such that f can be written as a polynomial expression in n variables of the form:*

$$f(x_1, \ldots, x_n) = \sum_{I\subseteq\{1,\ldots,n\}} a_I x^I. \tag{2}$$

Proof. (Induction on n) Let $n = 1$. The four possible Boolean functions of a single variable x are the constants 0 and 1, and the non-constants x and $1 + x$ (= the negation of x). They all have the desired form.

Now let $n \geq 1$. In the following we abbreviate $x' = (x_2, \ldots, x_n) \in \mathbb{F}_2^{n-1}$ when $x = (x_1, \ldots, x_n) \in \mathbb{F}_2^n$. Then $x = (x_1, \ldots, x_n)$ may be written as $x = (x_1, x')$.

Take a function $f \in \mathcal{F}_n$. For an arbitrary fixed value b for the first variable x_1 (i. e. $b = 0$ or 1) we consider the function $x' \mapsto f(b, x')$ of the remaining $n - 1$ variables in x'. By induction these two functions (for $b = 0$ as well as for $b = 1$) have the form

$$f(b, x') = p_b(x') \qquad \text{for all } x' \in \mathbb{F}_2^{n-1}$$

where p_0, p_1 are polynomial expressions in x' of the required form. Therefore

$$f(x_1, x') = \begin{cases} p_0(x'), & \text{if } x_1 = 0, \\ p_1(x'), & \text{if } x_1 = 1, \end{cases} \qquad \text{for all } x = (x_1, x') \in \mathbb{F}_2^n,$$

since x_1 can only assume the values 0 or 1. Write this as

$$f(x_1, x') = (1 + x_1)p_0(x') + x_1 p_1(x') \qquad \text{for all } x \in \mathbb{F}_2^n, \tag{3}$$

to get a polynomial expression in x. Expand this and remove monomials that occur twice, since these cancel out. ◇

The second column of Table 3 illustrates the mathematically compact statement of this theorem. Note that the variables in the table are denoted

by x and y instead of x_1 and x_2, and the coefficients by a, b, c, d instead of $a_\emptyset$, $a_{\{1\}}$, $a_{\{2\}}$, $a_{\{1,2\}}$. Each row of the table describes a Boolean function of two variables as sum of those terms 1, x, y, xy that have coefficient 1 in the representation of Equation (2), suppressing terms with coefficients 0.

The representation of a Boolean function as polynomial expression given by Theorem 3 is called the **algebraic normal form (ANF)**. Note that it is unique: Since there are 2^{2^n} polynomial expressions, and these represent all the 2^{2^n} different Boolean functions, first all these polynomial expressions must be different as functions, secondly the representation of a Boolean function as a polynomial expression in ANF must be unique. We have shown:

Theorem 4 *The representation of a Boolean function in algebraic normal form is unique.*

Definition The degree of a Boolean function $f \in \mathcal{F}_n$ as polynomial expression in algebraic normal form,

$$\deg f = \max\{\#I \mid a_I \neq 0\},$$

is called the **(algebraic) degree** of f. It is always $\leq n$.

The degree is the maximum number of variables that occur in a monomial of the ANF.

Example The Boolean function given by $x \mapsto x_1x_2 + x_2x_3 + x_3$ has degree 2.

Remark A high degree of a Boolean function is not caused by high powers of variables but "only" by products of different variables. Each single variable occurs at most in the first power in each monomial of the ANF. In other words all partial degrees—that are the degrees in the single variables x_i without accounting for the remaining variables—are ≤ 1.

7 Boolean Functions of Two Variables

The $2^4 = 16$ Boolean functions of two variables x and y are listed in Table 3. The table contains the polynomial expressions in algebraic normal form $a + bx + cy + dxy$ as well as logical expressions in conjunctive form.

As we saw already each Boolean function in any number of variables has a polynomial expression. To prove the existence of a logical expression we only have to show that the algebraic operations $+$ and $\cdot$ can be expressed by the logical operations $\vee$, $\wedge$, $\neg$. This is evident from the corresponding rows of Table 3. This shows (a weak form of the here unproven Theorem 2):

Theorem 5 *Each Boolean function admits a logical expression, i. e. may be defined by a formula in the logical operations* $\vee$, $\wedge$, $\neg$.

a	b	c	d	ANF	logical operation		CNF
0	0	0	0	0	False	constant	$x \wedge \neg x$
1	0	0	0	1	True	constant	$x \vee \neg x$
0	1	0	0	x	x	projection	x
1	1	0	0	$1 + x$	$\neg x$		$\neg x$
0	0	1	0	y	y	projection	y
1	0	1	0	$1 + y$	$\neg y$		$\neg y$
0	1	1	0	$x + y$	x XOR y	XOR	$(x \vee y) \wedge (\neg x \vee \neg y)$
1	1	1	0	$1 + x + y$	$x \iff y$	equivalence	$(x \vee \neg y) \wedge (\neg x \vee y)$
0	0	0	1	xy	$x \wedge y$	AND	$x \wedge y$
1	0	0	1	$1 + xy$	$\neg(x \wedge y)$	NAND	$(\neg x) \vee (\neg y)$
0	1	0	1	$x + xy$	$x \wedge (\neg y)$		$x \wedge (\neg y)$
1	1	0	1	$1 + x + xy$	$x \Longrightarrow y$	implication	$(\neg x) \vee y$
0	0	1	1	$y + xy$	$(\neg x) \wedge y$		$(\neg x) \wedge y$
1	0	1	1	$1 + y + xy$	$x \Longleftarrow y$		$x \vee (\neg y)$
0	1	1	1	$x + y + xy$	$x \vee y$	OR	$x \vee y$
1	1	1	1	$1 + x + y + xy$	$\neg(x \vee y)$	NOR	$(\neg x) \wedge (\neg y)$

Table 3: The 16 operations on two bits (= Boolean functions of 2 variables)

Hint The logical negation $\neg$ corresponds to the addition of 1 in the algebraic interpretation.

Remark In analogy we may write the ANF of a Boolean function of three variables x, y, z in the form

$$(x, y, z) \mapsto a + bx + cy + dz + exy + fxz + gyz + hxyz.$$

This involves 8 coefficients $a, \ldots, h$, and fits the observation that there are $2^{2^3} = 2^8 = 256$ such functions.

Example What does the ANF of the function f_0 from Section 2 look like (here written as $f_0(x, y, z) = x \wedge (y \vee z)$ using the variables x, y, z)? By Table 3 we have $y \vee z = y + z + yz$, whereas the AND operation $\wedge$ simply is the product in the field $\mathbb{F}_2$. Hence

$$f_0(x, y, z) = x \cdot (y + z + yz) = xy + xz + xyz.$$

By the way this shows that f_0 has degree 3.

Remark The table shows:

- Every logical operation may be replaced by at most 3 algebraic operations.

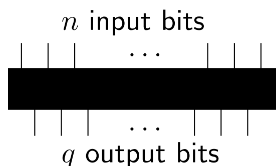
- Every algebraic operation may be replaced by at most 5 logical operations.
- Every binary operation (= function of 2 bits) may be replaced by at most 4 algebraic operations.

8 Boolean Maps

Most cryptographic applications involve processes that produce several bits at once, not only a single one. An abstract description uses the concept of a **Boolean map**, that is a map

$$f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$$

with natural numbers n and q, illustrated by the following figure



Note that the conceptual differentiation between "function" and "map" is somewhat arbitrary. We follow a common usage in Mathematics for expressing the property that the range is one- or multidimensional where "map" is the superordinate concept.

The images of f are bitblocks of length q. If we decompose them into their components

$$f(x) = (f_1(x), \ldots, f_q(x)) \in \mathbb{F}_2^q,$$

then we see that a Boolean map to $\mathbb{F}_2^q$ in a canonical way simply is a q-tuple of Boolean functions

$$f_1, \ldots, f_q\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2.$$

Definition The **(algebraic) degree** of a Boolean map $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$ is the maximum of the algebraic degrees of its components,

$$\deg f = \max\{\deg f_i \mid i = 1, \ldots, q\}.$$

Theorem 6 *Each Boolean map $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$ has a unique representation in the form*

$$f(x_1, \ldots, x_n) = \sum_{I \subseteq \{1,\ldots,n\}} x^I a_I$$

with $a_I \in \mathbb{F}_2^q$ and monomials x^I as in Theorem 3.

Like for functions this representation of a Boolean map is called **algebraic normal form**. It arises by combining the algebraic normal forms of the component functions $f_1, \ldots, f_q$. Compared with Theorem 3 the x^I and a_I changed their positions for by convention usually the "scalars" (the $x^I \in \mathbb{F}_2$) precede the "vectors" (the $a_I \in \mathbb{F}_2^q$). The a_I simply are the combinations of the corresponding coefficients of the component functions.

Example

Consider the Boolean map $g\colon \mathbb{F}_2^3 \longrightarrow \mathbb{F}_2^2$ defined by the following pair of logical expressions in three variables x, y, z:

$$g(x,y,z) := \begin{pmatrix} x \wedge (y \vee z) \\ x \wedge z \end{pmatrix}$$

where we write the components below each other, that is, in a column. In the first component we recognize the function f_0, in the second one the product $x \cdot z$. Thus the ANF of g is

$$g(x,y,z) = \begin{pmatrix} xy + xz + xyz \\ xz \end{pmatrix} = xy \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} = xz \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = xyz \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix}.$$

The algebraic degree is 3, and the value table is in Table 4. In the table we write the values $g(x,y,z) \in \mathbb{F}_2^2$ of g as bitstrings of length 2—the different notations of bitblocks, sometimes as column vectors, sometimes as bitstrings are chosen by convenience and are considered as equivalent, see the first subsection of Section 11.

x	y	z	$g(x,y,z)$
0	0	0	00
0	0	1	00
0	1	0	00
0	1	1	00
1	0	0	00
1	0	1	11
1	1	0	10
1	1	1	11

Table 4: The value table of a Boolean map

9 Linear Forms and Linear Maps

A Boolean function $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ is called a **linear form**, if it has degree 1 and absolute term 0. This means that the algebraic normal form contains

only linear terms, thus it has the form

$$f(x) = \sum_{i=1}^{n} s_i x_i \qquad \text{for all } x = (x_1, \ldots, x_n) \in \mathbb{F}_2^n$$

where $s_i \in \mathbb{F}_2$ for $i = 1, \ldots, n$. Since the s_i can only be 0 or 1 a linear form is a partial sum

$$\alpha_I(x) = \sum_{i \in I} x_i \qquad \text{for all } x = (x_1, \ldots, x_n) \in \mathbb{F}_2^n$$

over a subset $I \subseteq \{1, \ldots, n\}$ of the set of all indices, namely

$$I = \{i \mid s_i = 1\}.$$

We immediately conclude that there are exactly 2^n Boolean linear forms in n variables, and they correspond to the power set $\mathfrak{P}(\{1, \ldots, n\})$ in a natural way.

Other common notations are (for $I = \{i_1, \ldots, i_r\}$):

$$\alpha_I(x) = x[I] = x[i_1, \ldots, i_r] = x_{i_1} + \cdots + x_{i_r}.$$

The following theorem connects linear forms with the usual notation of Linear Algebra:

Theorem 7 *A Boolean function $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2$ is a linear form if and only if the following two conditions hold:*
(i) *$f(x + y) = f(x) + f(y)$ for all $x, y \in \mathbb{F}_2^n$.*
(ii) *$f(ax) = af(x)$ for all $a \in \mathbb{F}_2$ and $x \in \mathbb{F}_2^n$.*

Proof. For each linear form the conditions hold as immediately seen by the representation as partial sum.

For the reverse direction let f be a Boolean function that fulfills (i) and (ii). Let $e_1 = (1, 0, \ldots, 0)$, $\ldots$, $e_n = (0, \ldots, 1)$ be the "canonical unit vectors". Then each $x = (x_1, \ldots, x_n) \in \mathbb{F}_2^n$ is a sum

$$x = x_1 e_1 + \cdots + x_n e_n.$$

From this we get

$$f(x) = f(x_1 e_1) + \cdots + f(x_n e_n) = x_1 f(e_1) + \cdots + x_n f(e_n),$$

a partial sum of the x_i for which the constant value $f(e_i)$ is not 0, hence 1. Thus f is a linear form in the sense of our definition. $\diamond$

A Boolean map $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$ is called linear if all of its component functions $f_1, \ldots, f_q$ are linear forms. As in the case $q = 1$ we see:

Theorem 8 *A Boolean map* $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$ *is linear if and only if the following two conditions hold:*
(i) $f(x+y) = f(x) + f(y)$ *for all* $x, y \in \mathbb{F}_2^n$.
(ii) $f(ax) = af(x)$ *for all* $a \in \mathbb{F}_2$ *and* $x \in \mathbb{F}_2^n$.

Theorem 9 *A Boolean map* $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}_2^q$ *is linear if and only if it has the form*

$$f(x) = \sum_{i=1}^{n} x_i s_i$$

with $s_i \in \mathbb{F}_2^q$.

A Boolean map is called **affine** if its algebraic degree is ≤ 1 or, equivalently, if it is the sum of a linear map and a constant.

In the case $q = 1$, for functions, the only constants are 0 and 1. Adding the constant 1 is the same as the logical negation that "toggles" the bits. In other words: *The affine Boolean functions are the linear forms and their negations.*

10 Systems of Boolean Linear Equations

Algebra over the field $\mathbb{F}_2$ is quite easy. Many complications known from other mathematical domains break down. This is the case for the solution of systems of linear equations, systems of the form

$$\begin{array}{ccccccc} a_{11}x_1 & + & \cdots & + & a_{1n}x_n & = & b_1 \\ \vdots & & & & \vdots & & \vdots \\ a_{m1}x_1 & + & \cdots & + & a_{mn}x_n & = & b_m \end{array}$$

where a_{ij} and $b_i \in \mathbb{F}_2$ are given and the x_j are the unknowns to be solved for. In matrix notation the equations become

$$Ax = b$$

where x and b are column vectors, that is $(n \times 1)$- or $(m \times 1)$-matrices.

Systems of Linear Equations in Sage

As a link with "ordinary" Linear Algebra we consider an example over the rational numbers $\mathbb{Q}$, the system

$$\begin{array}{ccccccc} x_1 & + & 2x_2 & + & 3x_3 & = & 0 \\ 3x_1 & + & 2x_2 & + & x_3 & = & -4 \\ x_1 & + & x_2 & + & x_3 & = & -1 \end{array}$$

This is treated by Sage Example 1. The single steps are:

1. Define the "coefficient matrix" $A = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \\ 1 & 1 & 1 \end{pmatrix}$.

2. Define the "image vector" $b = (0, -4, 1)$.

3. Let Sage find a "solution vector" x.—Having written the left-hand side of the system as matrix product Ax we have to apply the method `solve_right()`.

4. The system could have other solutions. We find these by solving the corresponding "homogeneous" system $Az = 0$. If z is a solution of the homogeneous system, then $A \cdot (x + z) = Ax + Az = b + 0 = b$, hence $x + z$ is another solution of the original ("inhomogeneous") system. In this way we get all solutions: For if $Ax = b$ and $Ay = b$, then $A \cdot (y - x) = 0$, hence the difference $y - x$ is a solution of the homogeneous system. The Sage method `right_kernel()` determines all solutions of the homogeneous system.

5. The output of `right_kernel()` is somewhat cryptic. It says that all solutions of the homogeneous system are multiples of the vector $z = (1, -2, 1)$. (Since all coefficients are integers Sage considers the system as defined over $\mathbb{Z}$ (= `Integer Ring`).)

6. Finally we verify that $y = x - 4z$ really is a solution, i. e. $Ay = b$.

In the general case (over an arbitrary field) the solution of a system of linear equations is constructed by Gaussian elimination. The Sage method `solve_right()` also uses this algorithm.

```
sage: A = Matrix([[1,2,3],[3,2,1],[1,1,1]])
sage: b = vector([0,-4,-1])
sage: x = A.solve_right(b); x
(-2, 1, 0)
sage: K = A.right_kernel(); K
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[ 1 -2  1]
sage: y = x - 4*vector([1,-2,1]); y
(-6, 9, -4)
sage: A*y
(0, -4, -1)
```

Sage Example 1: Solution of a system of linear equations over $\mathbb{Q}$

Systems of Linear Equations in the Boolean Case

In the Boolean case (over the field $\mathbb{F}_2$) the solution by Gaussian elimination is extremely simple since the only coefficients are 0 and 1, and multiplications or divisions are completely obsolete. There are no complicated coefficients (like fractions over $\mathbb{Q}$) or inexact coefficients (like floating point numbers). So simple is the method that even for six unknowns the solution by "pencil and paper" is faster then writing the corresponding program in Sage. Here is an illustrative example.

The basic idea of elimination is reducing to a system with only $n-1$ unknowns by "elimination" of one unknown.

1. case: All coefficients $a_{in} = 0$ for $i = 1, \ldots, m$, that is x_n doesn't occur at all. Then the system is already reduced.

2. case: x_n has coefficient 1 in one of the equations. Then we solve this equation for x_n:

$$x_n = a_{i1}x_1 + \cdots a_{i,n-1}x_{n-1} + b_i,$$

and substitute the resulting term for x_n in the other $m-1$ equations. These then only contain the unknowns $x_1, \ldots, x_{n-1}$. (If x_n occurs in more then one equation it doesn't matter which one we choose—in contrast to the situation over other fields where finding an optimal "pivot element" is an essential part of the algorithm.)

We continue this approach recursively until only one unknown or one equation is left (whatever happens first). Now a concrete example.

Example

$$\begin{array}{cccccccc}
x_1 & & +x_3 & & & +x_6 & = & 1 \\
x_1 & +x_2 & & +x_4 & & +x_6 & = & 0 \\
& x_2 & +x_3 & & +x_5 & +x_6 & = & 0 \\
x_1 & & & +x_4 & +x_5 & & = & 1 \\
& x_2 & & +x_4 & +x_5 & & = & 1
\end{array}$$

The first equation yields $x_6 = x_1 + x_3 + 1$. The remaining system from equations 2 to 5 after elimination is (using $x_1 + x_1 = 0$ etc.):

$$\begin{array}{ccccccc}
& x_2 & +x_3 & +x_4 & & = & 1 \\
x_1 & +x_2 & & & +x_5 & = & 1 \\
x_1 & & & +x_4 & +x_5 & = & 1 \\
& x_2 & & +x_4 & +x_5 & = & 1
\end{array}$$

We solve the second equation of the remaining system for $x_5 = x_1 + x_2 + 1$ and substitute, getting

$$\begin{array}{cccccc}
& x_2 & +x_3 & +x_4 & = & 1 \\
& x_2 & & +x_4 & = & 0 \\
x_1 & & & +x_4 & = & 0
\end{array}$$

The last two equations yield $x_4 = x_2 = x_1$, and the first one then gives $x_3 = 1$. Now we have the complete solution:

$$x_1 = x_2 = x_4 = x_6 = a \quad \text{with arbitrary } a \in \mathbb{F}_2, \quad x_3 = 1, \quad x_5 = 1.$$

Since a may assume the values 0 or 1 there are exactly two solutions: $(0, 0, 1, 0, 1, 0)$ and $(1, 1, 1, 1, 1, 1)$.

The Example in Sage

Sage code for this example is in Sage Example 2. The Sage method `solve_right()` gives only the one solution $(0, 0, 1, 0, 1, 0)$. To get all solutions we have to solve the homogeneous system: Its solutions are the multiples of the vector $(1, 1, 0, 1, 0, 1)$, hence the two vectors $(0, 0, 0, 0, 0, 0)$ and $(1, 1, 0, 1, 0, 1)$. Thus the second solution of the original system is $(0, 0, 1, 0, 1, 0) + (1, 1, 0, 1, 0, 1) = (1, 1, 1, 1, 1, 1)$.

```
sage: M = MatrixSpace(GF(2), 5, 6) # GF(2) = field with two elements
sage: A = M([[1,0,1,0,0,1],[1,1,0,1,0,1],[0,1,1,0,1,1],[1,0,0,1,1,0],\
      [0,1,0,1,1,0]]); A
[1 0 1 0 0 1]
[1 1 0 1 0 1]
[0 1 1 0 1 1]
[1 0 0 1 1 0]
[0 1 0 1 1 0]
sage: b = vector(GF(2),[1,0,0,1,1])
sage: x = A.solve_right(b); x
(0, 0, 1, 0, 1, 0)
sage: K = A.right_kernel(); K
Vector space of degree 6 and dimension 1 over Finite Field of size 2
Basis matrix:
[1 1 0 1 0 1]
```

Sage Example 2: Solution of a system of Boolean linear equations

Cost Estimate

What about the costs for a solution of a system of Boolean linear equations? Consider m equations for n unknowns. The matrix A of coefficients has size $m \times n$, the extended matrix (A, b) has size $m \times (n + 1)$.

Since we only want a coarse estimate we don't bother about optimizations, and assume that $m = n$. In the case of $m > n$ we ignore surplus equations (at the end one has to check whether the found solutions fulfill

the additional equations). In the case $m < n$ we add "null equations" (of type $0 \cdot x_1 + \cdots + 0 \cdot x_n = 0$).

The elimination step, the reduction of the problem size from n to $n-1$, needs one run through all the n columns of the *extended* matrix:

- At first in column n, that contains the coefficients of x_n, we search the first entry 1. This takes at most n bit comparisions.
- Then the row we found (with the first entry 1 in column n) is added to all rows below it that also contain a 1 in column n. This costs one bit comparision and at most n bit addition for each row—we may omit the n-th entry because we know there will be a 0.

On the whole we need n bit comparisons and at most $n \cdot (n-1)$ bit additions, a total of at most n^2 bit operations. Let us denote the number of needed bit operations for the complete solution of the system by $N(n)$. Then we have the inequality:

$$N(n) \leq n^2 + N(n-1) \quad \text{for all } n \geq 2.$$

Clearly $N(1) = 1$: We check the one coefficient of the one unknown whether it is 0 or 1 and decide whether the equation has a unique solution (coefficient 1), or whether it is fulfilled for no or for arbitrary values of the unknown (coefficient 0, right-hand side $b = 1$ or 0).

Then we conclude $N(2) \leq 2^2 + 1$, $N(3) \leq 3^2 + 2^2 + 1$ etc. By induction we immediately get

$$N(n) \leq \sum_{i=1}^{n} i^2.$$

The value of this sum is explicitly known, and we have shown:

Theorem 10 *The number $N(n)$ of bit comparisons and bit additions needed for the solution of a system of n binary linear equations with n unknowns has the upper bound*

$$N(n) \leq \frac{1}{6} \cdot n \cdot (n+1) \cdot (2n+1).$$

A slightly coarser statement is that the cost is $O(n^3)$. In any case it is "polynomial of small degree" in the size n of the problem. Note that the asymptotic complexity over $\mathbb{F}_2$ is not smaller than over any other field, although the single steps are much cheaper.

11 The Representation of Boolean Functions and Maps

Some Interpretations of Bitblocks

We used bitblocks $b = (b_1, \ldots, b_n) \in \mathbb{F}_2^n$ to represent different objects. A bitblock can stand for:

- a vector $b \in \mathbb{F}_2^n$ (that is itself as a bitblock, written as row or as column),
- the argument of a Boolean function or map, for example as the row key in a value table (or truth table),
- a bitstring of length n,
- a subset $I \subseteq \{1, \ldots, n\}$ that is defined by the indicator b via the equivalence $i \in I \Leftrightarrow b_i = 1$,
- a linear form on $\mathbb{F}_2^n$, expressed as sum of the variables x_i for which $b_i = 1$,
- a monomial in n variables $x_1, \ldots, x_n$ with all partial degrees ≤ 1; here b_i is the exponent 0 or 1 of the variable x_i,
- an integer between 0 and $2^n - 1$ in binary representation; the sequence of the binary "digits" (= bits) is identical with the corresponding bitstring. Conversely the integer is the index (starting with 0) of the bitstring if the bitstrings are ordered alphabetically in increasing order.

Of course there could be yet other interpretations—each information eventually has a binary encoding. The bitblocks for $n = 3$ are listed in Table 5. Some conversion routines are in Sage Example A.1 (part of the module `Bitblock.sage`).

Representation of the Truth Table of a Boolean Function

The interpretation of bitblocks as integers in binary representation leads to an economical representation of the truth table of a Boolean function $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}$ as bitblock $b = (b_0, \ldots, b_{2^n-1})$ of length 2^n:

$$f(x) = b_{i(x)}, \quad \text{where } i(x) = x_1 \cdot 2^{n-1} + \cdots + x_{n-1} \cdot 2 + x_n \quad \text{for} \quad x = (x_1, \ldots, x_n) \in \mathbb{F}_2^n.$$

At first sight this looks complicated, however it simply means: "Interpret x as the binary representation of an integer $i(x)$, look at the bitblock b, and

Integer	Bitstring	Subset	Linear form	Monomial
0	000	$\emptyset$	0	1
1	001	$\{3\}$	x_3	x_3
2	010	$\{2\}$	x_2	x_2
3	011	$\{2,3\}$	$x_2 + x_3$	x_2x_3
4	100	$\{1\}$	x_1	x_1
5	101	$\{1,3\}$	$x_1 + x_3$	x_1x_3
6	110	$\{1,2\}$	$x_1 + x_2$	x_1x_2
7	111	$\{1,2,3\}$	$x_1 + x_2 + x_3$	$x_1x_2x_3$

Table 5: Interpretations of bitblocks of length 3

x_1	x_2	x_3	$i(x)$	$f_0(x_1, x_2, x_3)$
0	0	0	0	0
0	0	1	1	0
0	1	0	2	0
0	1	1	3	0
1	0	0	4	0
1	0	1	5	1
1	1	0	6	1
1	1	1	7	1

Table 6: An extended truth table

pick up the bit at position $i(x)$". This correspondence is illustrated by an additional column in the truth table, Table 2, of the function f_0, see the extended Table 6.

Thus for example the truth table of f_0 is simply given by the bitblock $(0,0,0,0,0,1,1,1)$, or even more economically by the bitstring

$$00000111$$

of length $2^3 = 8$.

Representation of the Algebraic Normal Form

For the description of the algebraic normal form (ANF) also 2^n bits suffice: the coefficients of the 2^n different monomials. The monomials also are in the list of interpretations of bitblocks, see Table 5. Using this we may interpret a bitblock $a = (a_0, \ldots, a_{2^n-1})$ as representation of the ANF of a Boolean function $f\colon \mathbb{F}_2^n \longrightarrow \mathbb{F}$ in the following way:

$$f(x) = \sum_{i=0}^{2^n-1} a_i x_1^{e_1(i)} \cdots x_n^{e_n(i)} \quad \text{where } i = e_1(i)\cdot 2^{n-1} + \cdots + e_n(i)$$
$$\text{with} \quad e_1(i), \ldots, e_n(i) = 0 \text{ or } 1.$$

Expressed in words this means: "Interpret the n-tuple e of exponents of a monomial as the binary representation of an integer i. Look at position i in the bitblock a whether this monomial occurs in the ANF of f or not."

For the example f_0 we already saw that the ANF is

$$f_0(x) = x_1x_3 + x_1x_2 + x_1x_2x_3,$$

the sum of the monomials with exponent triples $101, 110, 111$, corresponding to the integers $5, 6, 7$. Thus the economical representation of the ANF by a bitstring is

$$00000111.$$

Caution! This is the same bitstring as for the truth table *by accident*—a special property of the function f_0! The function $f(x_1, x_2) = x_1$ has the truth table 0011 and the ANF 0010.

For determining the ANF from the truth table in the general case we use the Sage module `boolF.sage`, see Appendix A. This transformation that transforms a bitstring of length 2^n (the truth table) into another bitstring of length 2^n (the list of coefficients of the ANF) is sometimes called the Reed-Muller transformation or the binary Moebius transformation. Its application to f_0 is demonstrated in Sage Example 3.

```
sage: attach('Bitblock.sage')
sage: attach('boolF.sage')
sage: bits = "00000111"
sage: x = str2bbl(bits); x
[0, 0, 0, 0, 0, 1, 1, 1]
sage: f = BoolF(x)
sage: y = f.getTT(); y
[0, 0, 0, 0, 0, 1, 1, 1]
sage: z = f.getANF(); z
[0, 0, 0, 0, 0, 1, 1, 1]
```

Sage Example 3: A Boolean function with truth table and ANF

Remark Naively evaluating a Boolean function f for all arguments $x \in \mathbb{F}_2^n$ costs 2^n evaluations $f(x)$ each with at most 2^n summands with at most $n-1$ multiplicatios. Therefore the total cost is about $n \cdot 2^n \cdot 2^n$. If we relate the cost to the size of the input—that is $N = 2^n$—we get the quasi-quadratic cost function $N^2 \cdot \log_2(N)$. As is often the case also here a binary recursion—dividing the problem into two partial problems of half the size—leads to a significantly more efficient algorithm. The starting point is Equation (3). The end effect is the quasi-linear cost $3N \cdot \log_2 N$. This algorithm (also called fast binary Moebius transformation) is implemented in the module `boolF.sage`, see Appendix A.2.

Object Oriented Implementation

An implementation of Boolean functions in Sage (or Python) is given in Appendix A.2 (module `boolF.sage`). Sage itself has a class `sage.crypto.boolean_function` that contains many of the needed methods, but not the fast Moebius transformation. Our implementation is independent.

In general in the object oriented paradigma one defines a class as an abstraction of the structure of an object "Boolean function":

Class `BoolF`:

Attributes:

- `blist`: truth table as a list of bits (= bitblock in the "natural" order); we use this as the internal representation of the Boolean function.
- `dim`: the dimension of the domain

Methods:

- `setTT`: fill the truth table with a bitblock ("TT" for Truth Table)
- `setANF`: read the ANF and internally transform it to a truth table
- `setDim`: read the dimension of the domain
- `getTT`: return the truth table as a bitblock
- `value`: return the value of the Boolean function for a given argument
- `getDim`: return the dimension of the domain
- `getANF`: return the algebraic normal form (ANF) as bitblock (in the "natural" order)
- `deg`: return the algebraic degree

The first three of these, the "`set` methods", are only implicitly needed during initialization. For a "human readable" output we additionally define the methods `printTT` and `printANF`.

Functions for transforming bitlists into integers or bitstrings, or vice versa, are in the module `Bitblock.sage`.

The implementation of Boolean maps builds on functions: Define a class `BoolM` as a list of objects of the class `BoolF` with (at least) the analogous methods.

12 Boolean Circuits

A Boolean circuit describes an algorithm in the form of a flow chart that connects the single bit operations. This concept leads to an alternative approach to complexity theory, based on circuit complexity and somewhat different from TURING complexity, that goes back to SHANNON in 1949.

A **circuit** is an acyclic directed labeled graph all of whose nodes have indegree 0 or 2.

> The nodes are connected by arrows. There is no closed directed path. All nodes have labels. Each node is the target of 0 or 2 arrows.

An **input node** is a node of indegree 0. An **output node** is a node of outdegree 0. Thus an output node is not the tail (or starting point) of any arrow. Each node has one of the labels $\oplus$ or $\otimes$, corresponding to addition or multiplication of two bits in the field $\mathbb{F}_2$, or "variable" or "constant" for an input node.

> Thus a circuit represents what in Section 2 was called the technical specification of a Boolean function or map.

Some input nodes are defined as constant: They always represent a constant bit 0 or 1. Since we allow for unbounded outdegrees in principle two constant input nodes 0 and 1 suffice, even a single constant node 1. However it is convenient to allow more constant input bits, say a constant integer expressed in base-2 representation.

Notes

- Alternatively we could also label the nodes by the logical operations or even by any of the 16 Boolean functions of two variables, see Section 7. For complexity results this wouldn't make a noteworthy difference: The size of the circuit would at the worst expand (or shrink) by a small constant factor, for example by a factor of at most 5 if logical nodes would replace algebraic nodes, or 3 for the opposite direction, see Figure 2 for $x \vee y$.
- For building circuits of logical operations it would be convenient to allow nodes with indegree 1 to account for the NOT-operation.
- A circuit may re-use intermediate results many times due to the unbounded outdegree. This corresponds to unbounded storage on a real computer. A more narrow concept that avoids this is a Boolean formula—or a circuit whose underlying graph is a tree. Then each node has outdegree 1 (except that the output nodes have outdegree 0). A circuit of this kind reflects the structure of a one-line formula composed of operations on two bits where each intermediate result can be used only once. In particular each bit must be repeatedly calculated as many times as needed.
- Sometimes the definition of circuits allows for an unbounded indegree. However bounding it by 2 makes sense since real machines process only a bounded number of bits in each step. Whether we take 2 as bound or some larger number is irrelevant for the following complexity assessments.

An **input** is an assignment of a bit tuple $x = (x_1, \ldots, x_r) \in \mathbb{F}_2^r$ to the r nonconstant (labelled "variable") input nodes. An internal node (or "gate") adds or multiplies (mod 2) its two input bits, depending on its label. After traversing the complete circuit the s output nodes contain the **output** $y \in \mathbb{F}_2^s$. Thus the circuit defines a Boolean map

$$C \colon \mathbb{F}_2^r \longrightarrow \mathbb{F}_2^s$$

representing its algorithmic decomposition into single bit operations and immediately leading to the truth table.

Conversely, since every Boolean map $\mathbb{F}_2^r \longrightarrow \mathbb{F}_2^s$ is polynomial, it has a description by a circuit.

Every Boolean map has an algorithmic description as a Boolean circuit. The Algebraic Normal Form easily translates to such a representation.

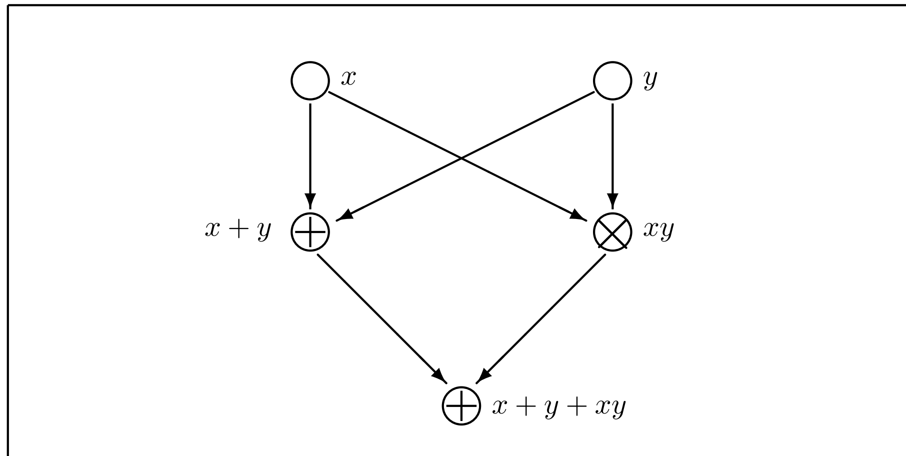


Figure 2: Boolean circuit for $x \vee y = x + y + xy$

Figure 3 shows an example how an ANF translates to a circuit.

The data structure "Boolean circuit" is represented by a finite set $\mathcal{N}$, interpreted as the set of nodes, each node a having the attributes

- label: one of "$\oplus$", "$\otimes$", "variable", "constant",
- predecessors: $\emptyset$ or a pair of other nodes $P(a) \in \mathcal{N}^2$.

The set of arrows (= directed edges) is implicitly given by all the predecessor relations. The acyclicity is a side condition that has to be guaranteed, as well as the compatibility of labels and indegrees.

13 Circuits for Basic Integer or Logical Operations

As illustrations we consider circuits for some basic arithmetic operations of integers that serve as building blocks of more complicated operations. In this way we can express as a Boolean circuit every algorithm that, beside bit manipulations, involves integer arithmetic and comparisions.

1. Adding two one-bit integers a, b with $\bmod 2$ sum c and carry u, the "primitive" addition in $\mathbb{N}$ of binary digits, is a function $C : \mathbb{F}_2^2 \longrightarrow \mathbb{F}_2^2$. Figure 4 shows a corresponding circuit.

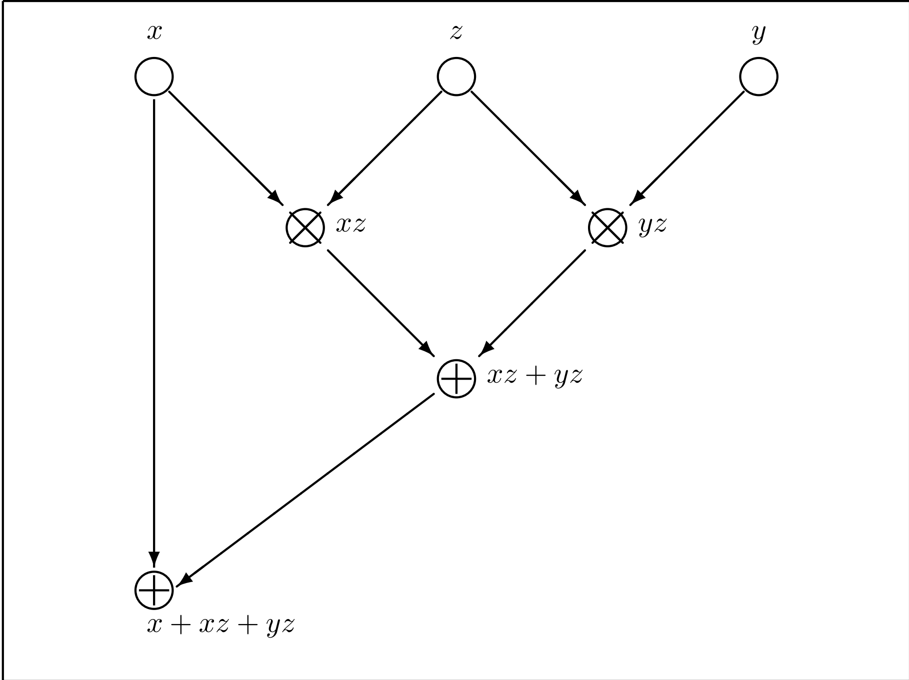


Figure 3: Boolean circuit from ANF, example $f(x, y, z) = x + xz + yz$

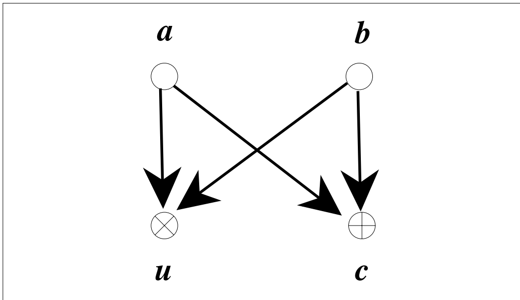


Figure 4: Circuit for adding two one-bit integers

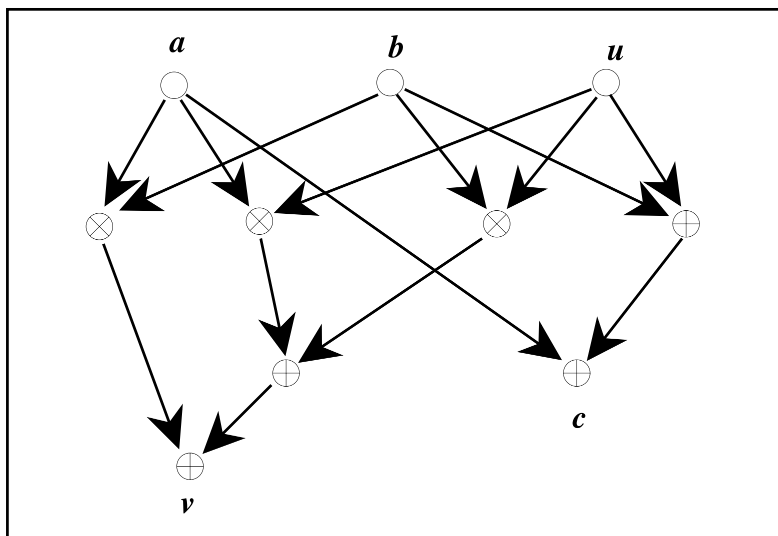


Figure 5: Circuit for adding three one-bit integers

2. The basic building block for describing the addition of arbitrary integers is the addition of three bits a, b, u with mod 2 sum c and carry v. It is a function $C : \mathbb{F}_2^3 \longrightarrow \mathbb{F}_2^2$ represented by the circuit in Figure 5. A formula for v is:

$$v = a \cdot b + a \cdot u + b \cdot u = \begin{cases} 1 & \text{if at least 2 input bits are 1,} \\ 0 & \text{else.} \end{cases}$$

3. Adding a one-bit integer and an s-bit integer follows a similar, somewhat more bulky scheme and has a representation by a circuit with $3s + 1$ nodes, $s + 1$ of which are output nodes.

4. The multiplication of two bits is trivial (one $\otimes$-gate). More interesting is the multiplication of two two-bit integers $2a_1 + a_0$ and $2b_1 + b_0$ with four-bit result $8c_3 + 4c_2 + 2c_1 + c_0$. The classical algorithm follows the scheme

$$\begin{aligned} & c_0 = a_0 \cdot b_0, \\ & t_1 = a_0 \cdot b_1, \quad t_2 = a_1 \cdot b_0, \quad c_1 = t_2 + t_1, \quad u_1 = t_2 \cdot t_1, \\ & t_3 = a_1 \cdot b_1, \quad c_2 = t_3 + u_1, \quad c_3 = t_3 \cdot u_1 \end{aligned}$$

with auxiliary (intermediate) bits t_1, t_2, t_3, and u_1. The corresponding circuit is shown in Figure 6.

5. Logical operations reduce to additions and multiplications in $\mathbb{F}_2$ according to Table 3.

6. The simplest case of branching is represented by the function $f\colon \mathbb{F}_2^3 \longrightarrow \mathbb{F}_2$,
$$f(x,y,z) = \begin{cases} x & \text{if } z = 0, \\ y & \text{if } z = 1. \end{cases}$$
To construct a representation by a closed formula we note that
$$\begin{aligned} z \cdot y &= \begin{cases} 0 & \text{if } z = 0, \\ y & \text{if } z = 1, \end{cases} \\ (\neg z) \cdot x &= \begin{cases} x & \text{if } z = 0, \\ 0 & \text{if } z = 1. \end{cases} \end{aligned}$$
Hence $f(x,y,z) = x \cdot (1+z) + y \cdot z$. Figure 7 shows the corresponding circuit. It contains a constant input node for computing the logical negation $1+z$.

7. To describe a more general branching we take three circuits $C_0, C_1 : \mathbb{F}_2^r \longrightarrow \mathbb{F}_2^s$, and $E\colon \mathbb{F}_2^r \longrightarrow \mathbb{F}_2$ and ask for a circuit $C\colon \mathbb{F}_2^r \longrightarrow \mathbb{F}_2^s$ such that
$$C(x) = \begin{cases} C_0(x) & \text{if } E(x) = 0, \\ C_1(x) & \text{if } E(x) = 1. \end{cases}$$
Substituting C_0, C_1, and E into the formula of Example 6 we get
$$C(x) = (1 + E(x)) \cdot C_0(x) + E(x) \cdot C_1(x).$$
Figure 8 shows the corresponding circuit.

8. The comparision $x \geq y$ of two one-bit integers is one of the 16 operations on two bits, namely $1 + y + x \cdot y$. More generally we cobble together the comparision of two $n+1$-bit integers $x = (x_n \ldots x_0)$ and $y = (y_n \ldots y_0)$ in the following way:
$$C(x,y) = \begin{cases} 1 & \text{if } x_n > y_n \\ & \text{or } x_n = y_n \text{ and } x' \geq y', \\ 0 & \text{if } x_n = y_n \text{ and } x' < y' \\ & \text{or } x_n < y_n \end{cases}$$
with $x' = (x_{n-1} \ldots x_0)$ and $y' = (y_{n-1} \ldots y_0)$.

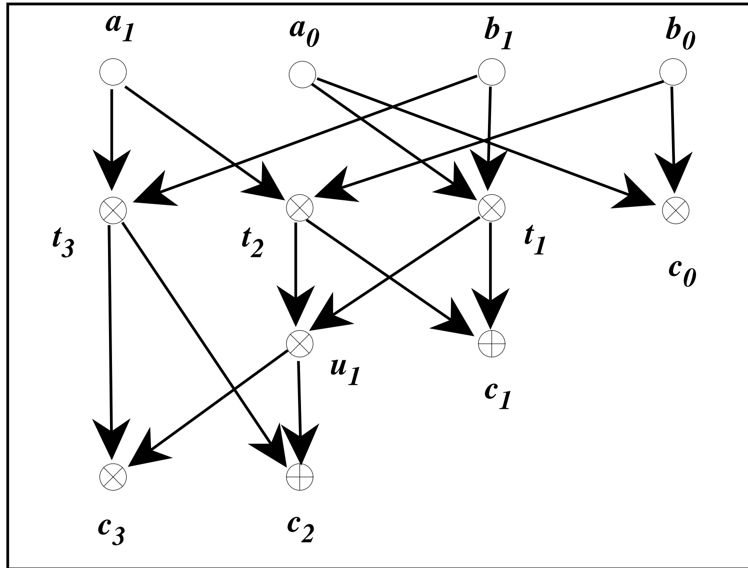


Figure 6: Circuit for the multiplication of two two-bit integers

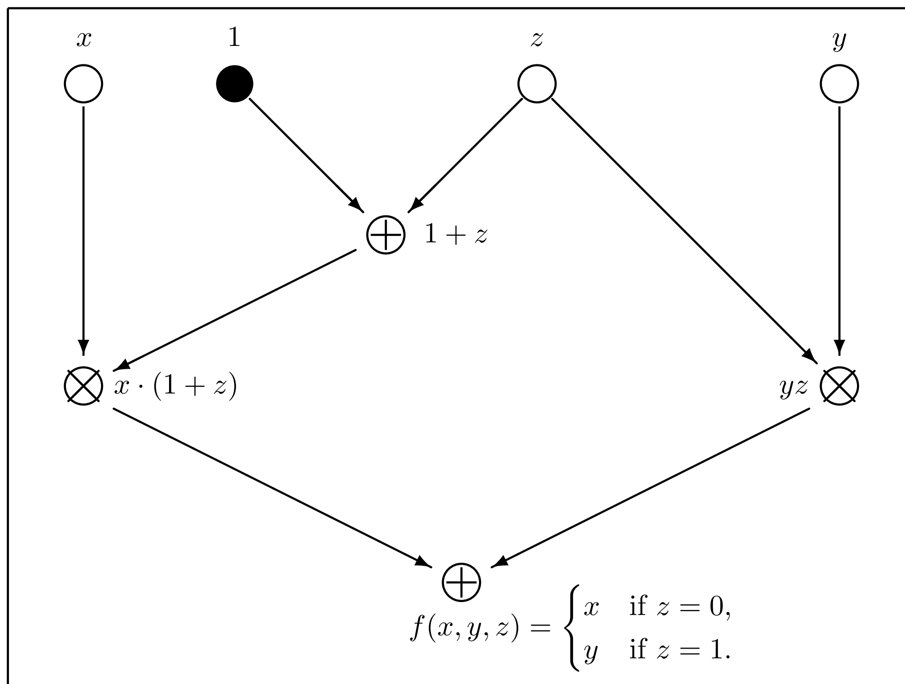


Figure 7: Boolean circuit for a simple branch

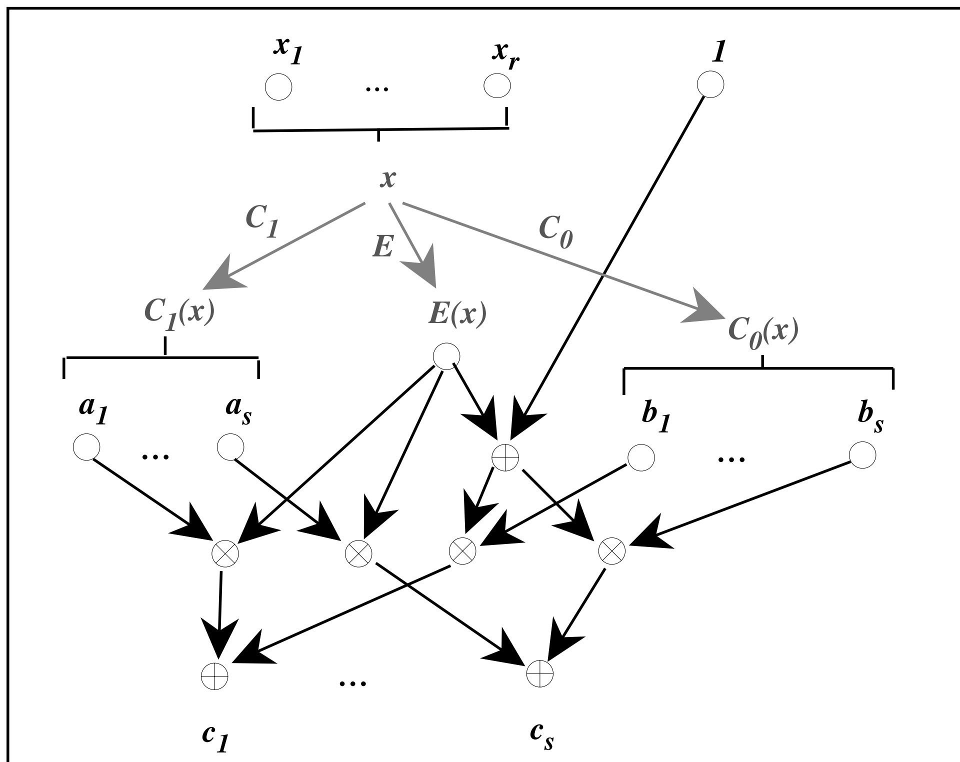


Figure 8: Circuit for branching in general

14 Circuits and Programs

Intuitively it is clear, but also easy to prove, that each algorithm has a description by a circuit. Note that this corresponds to how computers work on the lowest level by bit operations. Thus “the circuit” may serve as a universal machine model—we don’t need this result in this text but observe that this model involves a different circuit for each computation. More on this subject in Parts III and IV of these lecture note. An advantage over other machine models is that a circuit allows for a fairly realistic count of elementary operations.

A circuit computes a Boolean map $C\colon \mathbb{F}_2^r \longrightarrow \mathbb{F}_2^s$. To derive an algorithm, say expressed in a programming language, we proceed as follows:

1. Enumerate the nodes (the set $\mathcal{N}$) in a consistent way.
2. Run through all nodes according to this enumeration and perform the corresponding bit operation.

To see that there is a consistent enumeration we partition the circuit into layers, that is we exhibit a canonical function $L\colon \mathcal{N} \longrightarrow \mathbb{N}$, and then choose an enumeration $N\colon \mathcal{N} \longrightarrow \{1, \ldots, \#\mathcal{N}\} \subseteq \mathbb{N}$ that is compatible with L in the sense that $L(a) < L(b)$ implies $N(a) < N(b)$. We proceed as follows:

- Layer 0 consists of the input nodes.
- For $i \geq 1$ layer i consists of the nodes that can be reaches from each input node via a path of length at most i or via no path at all, and don’t belong to layer $i - 1$.
- Run through the layers 1, 2, ... in order, and in each layer enumerate the nodes in any order, using the smallest numbers that are not yet used.

(In graphical illustrations the “high” layers usually lie below the “low” layers.) Obviously, if a is a predecessor of b, then $L(a) < L(b)$, and a forteriori $N(a) < N(b)$. If $s = 1$, that is if C defines a Boolean function, then there is only one output node, and this must be the unique node in the highest layer.

Now the algorithm defined by the circuit runs as follows:

- Assume an enumeration N compatible with the layer function L such that $1, \ldots, n_0$ correspond to the variable input nodes, and $n_0{+}1, \ldots, n_1$ to the constant input nodes.
- Assign values $x_h \in \mathbb{F}_2$ to the variable input nodes with numbers $1, \ldots, n_0$.

- Loop over $h = n_1 + 1, \ldots, \#\mathcal{N}$ (omitting the input layer 0): For each h select the node $a \in \mathcal{N}$ with $N(a) = h$ and its pair (b, c) of predecessors with numbers $j, k < h$ and assign the value $x_h = x_j * x_k$ to a, where $*$ is the operation corresponding to the label "$\oplus$" or "$\otimes$" of a.
- Output the bits from the output nodes.

Example 1

For the circuit of Figure 3 we have

Layer 0: input nodes

- 1 (filled with x)
- 2 (filled with y)
- 3 (filled with z)

Layer 1: nodes

- 4 (evaluating to xz)
- 5 (evaluating to yz)

Layer 2: node 6 (evaluating to $xz + yz$)

Layer 3: output node 7 (evaluating to $x + xz + yz$)

Example 2

For the circuit of Figure 7 we have

Layer 0: variable input nodes

- 1 (filled with x)
- 2 (filled with y)
- 3 (filled with z)

plus the constant input node 4 (occupied by 1)

Layer 1: nodes

- 5 (evaluating to $1 + xz$)
- 6 (evaluating to yz)

Layer 2: node 7 (evaluating to $x \cdot (1 + z)$)

Layer 3: output node 8

15 Measuring the Complexity of Circuits

Two key figures for assessing the complexity of a circuit are $\#C$, its **size**, = the number $\#\mathcal{N}$ of its nodes, and $d(C)$, its **depth**, or the length of the largest path = the index of the highest layer. The size counts the number of bit operations and thus measures the complexity of serial execution, the depth measures the complexity of unbounded parallel execution of the algorithm, where in each step all the nodes of a complete layer are evaluated.

Examples

1. The circuit for adding two bits, Figure 4, has size 4 and depth 1.
2. The circuit for adding three bits, Figure 5, has size 10 and depth 3.
3. From Example 3 from Section13 above we get a simple (not yet optimal) circuit for adding s one-bit integers with $A(s)$ nodes, among them $a(s)$ output nodes. By Example 1 $A(2) = 4$, $a(2) = 2$, by Example 3 from Section13 we have the recursive formulas $a(s) = a(s-1)+1$ and $A(s) = A(s-1) + 2a(s-1) + 1$. Hence by induction $a(s) = s$ and $A(s) = s^2$.
4. The circuit for multiplying two two-bit integers, Figure 6, has size 12 and depth 3.
5. To evaluate a Boolean function of r bits in ANF we need at most 1 constant input node (for the constant monomial 1), r input nodes and at most $\binom{r}{k}$ multiplicative nodes for calculating all monomials of degree k. Then we have to add all the monomials, using at most $2^r - 1$ additive nodes. In summary we can represent each Boolean function of r variables by a circuit of size at most $2^r + 2^r - 1 = 2^{r+1} - 1$, noting that
$$\sum_{k=0}^{r} \binom{r}{k} = 2^r.$$
And for a Boolean map $\mathbb{F}_2^r \longrightarrow \mathbb{F}_2^s$ we need at most $s \cdot (2^{r+1} - 1)$ nodes. Clearly this general result leaves ample room for optimization in each single case.
6. For the circuit for branching, Figure 8, we have the formulas
$$\begin{aligned} \#C &= \#C_1 + \#C_2 + \#E - 2r + 3s + 2, \\ d(C) &= \max\{d(C_0) + 2, d(C_1) + 2, d(E) + 3\}. \end{aligned}$$
7. The size $V(n)$ of the circuit for comparing two $n+1$-bit integers in Example 8 from Section13 satisfies the recursive formula $V(n) = V(n-1) + 11$ with V(2) = 6, hence $V(n) = 11n - 5$.

8. Estimating the costs of the integer operations in the binary number system we see that pairs of n-bit integers can be added or subtracted with circuits of size $\mathrm{O}(n)$, and multiplied or divided with circuits of size $\mathrm{O}(n^2)$.

These results allow us to re-interpret results on the cost of algorithms in terms of integer operations as statements in terms of Boolean operations or, equivalently, of circuit complexity. Note that the O's are due to laziness. With some additional effort we could derive more concrete estimates.

A Appendix: Boolean Maps in Sage

A.1 Conversion of Bitblocks

```
def int2bbl(number,dim):
  """Converts number to bitblock of length dim via base-2
  representation."""
  n = number
  b = []
  for i in range(0,dim):
    bit = n % 2
    b = [bit] + b
    n = (n - bit)//2
  return b

def bbl2int(bbl):
  """Converts bitblock to number via base-2 representation."""
  ll = len(bbl)
  nn = 0
  for i in range(0,ll):
    nn = nn + bbl[i]*(2**(ll-1-i))
  return nn

def str2bbl(bitstr):
  """Converts bitstring to bitblock."""
  ll = len(bitstr)
  xbl = []
  for k in range(0,ll):
    xbl.append(int(bitstr[k]))
  return xbl

def bbl2str(bbl):
  """Converts bitblock to bitstring."""
  bitstr = ""
  for i in range(0,len(bbl)):
    bitstr += str(bbl[i])
  return bitstr
```

Sage Example 4: Conversion of bitblocks (in `Bitblock.sage`)

A.2 Class for Boolean Functions

```
class BoolF(object):
  """Boolean function
  Attribute: a list of bits describing the truth table of the function
  Attribute: the dimension of the domain"""

  __max = 4096                              # max dim = 12

  def __init__(self,blist,method="TT"):
    """Initializes a Booelan function with a truth table
    or by its algebraic normal form if method is ANF."""
    ll = len(blist)
    assert ll <= self.__max, "BoolF_Error: Bitblock too long."
    dim = 0                                 # dimension
    m = 1                                   # 2**dim
    while m < ll:
      dim = dim+1
      m = 2*m
    assert ll == m, "booltestError: Block length not a power of 2."
    self.__dim = dim
    if method=="TT":
      self.__tlist = blist
    else:
      self.__tlist=self.__convert(blist)

  def __convert(self,xx):
    """Converts a truth table to an ANF or vice versa."""
    x = copy(xx)                  # initialize auxiliary bitblock
    y = copy(xx)                  # initialize auxiliary bitblock
    mi = 1                        # actual power of 2
    for i in range(0,self.__dim): # binary recursion
      for k in range(0,2**(self.__dim)):
        if ((k//mi) % 2 == 1): # picks bit nr i
          y[k] = (x[k-mi] + x[k]) % 2 # XOR
        else:
          y[k] = x[k]
      for k in range(0,2**(self.__dim)):
        x[k] = y[k]
      mi = 2*mi                   # equals 2**i in the next step
    return x
```

Sage Example 5: Class for Boolean functions (in `boolF.sage`)

```
def getTT(self):
  """Returns truth table as bitlist."""
  return self.__tlist

def value(self,xx):
  """Evaluates Boolean function."""
  ll = len(xx)
  assert ll == self.__dim, "booltestError: Block has false length."
  index = bbl2int(xx)
  return self.__tlist[index]

def getDim(self):
  """Returns dimension of definition domain."""
  return self.__dim

def getANF(self):
  """Returns algebraic normal form as bitlist."""
  y = self.__convert(self.__tlist)
  return y

def deg(self):
  """Algebraic degree of Boolean function"""
  y = self.__convert(self.__tlist)
  max = 0
  for i in range (0,len(y)):
    if y[i] != 0:
      b = int2bbl(i,self.__dim)
      wt = sum(b)
      if wt > max:
        max = wt
  return max
```

Sage Example 6: Boolean functions—continued

```
def printTT(self):
  """Prints truth table to stdout."""
  for i in range(0,2**(self.__dim)):
    bb = int2bbl(i,self.__dim)
    print "Value at " + bbl2str(bb) + " is " + repr(self.__tlist[i])

def printANF(self):
  """Prints algebraic normal form to stdout."""
  y = self.__convert(self.__tlist)
  for i in range(0,2**(self.__dim)):
    monom = int2bbl(i,self.__dim)
    print "Coefficient at " + bbl2str(monom) + " is " + repr(y[i])
```

Sage Example 7: Boolean functions—human readable output

A.3 Class for Boolean Maps

```
class BoolMap(object):
  """Boolean map
  Attribute: a list of Boolean functions
  Attribute: the dimensions of domain and range"""

  __max = 8                               # max dim = 8

  def __init__(self,flist):
    """Initializes a Boolean map with a list of Boolean functions."""
    qq = len(flist)
    assert qq <= self.__max, "BoolMap_Error: Too many components."
    ll = len(flist[0].getTT())
    dim = 0                               # dimension
    m = 1                                 # 2**dim
    while m < ll:
      dim = dim+1
      m = 2*m
    assert ll == m, "BoolMap_Error: Block length not a power of 2."
    assert dim <= self.__max, "BoolMap_Error: Block length exceeds maximum."
    self.__dimd = dim
    self.__dimr = qq
    for i in range(1,qq):
      li = len(flist[i].getTT())
      assert li == ll,  "BoolMap_Error: Blocks of different lengths."
    self.__flist = flist

  def getFList(self):
    """Returns component list."""
    return self.__flist

  def getDim(self):
    """Returns dimension of definition domain."""
    return [self.__dimd, self.__dimr]
```

Sage Example 8: Class for Boolean maps (in `boolF.sage`)

```
def getTT(self):
  """Returns truth table as list of bitlists."""
  nn = 2**(self.__dimd)
  qq = self.__dimr
  clist = []
  for j in range(0,qq):
    clist.append(self.__flist[j].getTT())
  transp = []
  for j in range(0,nn):
    trrow = []
    for i in range(0,qq):
      trrow.append(clist[i][j])
    transp.append(trrow)
  return transp

def value(self,xx):
  """Evaluates Boolean map."""
  ll = len(xx)
  assert ll == self.__dimd, "boolF_Error: Block has false length."
  index = bbl2int(xx)
  vlist = []
  for j in range(0,self.__dimr):
    vlist.append(self.__flist[j].getTT()[index])
  return vlist
```

Sage Example 9: Boolean maps—continued