Analysis of Minimal Boolean Circuits

Trabajo de Fin de Máster
Máster en Metodos Formales en Ingeniería Informática

Universidad Complutense de Madrid
2024-2025
Autor: Joseba Celaya Rodriguez
Director: Ismael Rodríguez Laguna

Convocatoria: Septiembre 2025
Nota: 5/10

Abandon normal instruments
∼ Oblique Strategies

Resumen

La complejidad de circuitos, una rama de la teoría de la complejidad computacional, ha mostrado avances limitados en la obtención de cotas inferiores para el tamaño mínimo de circuitos que resuelven problemas NP-completos. Las cotas existentes se aplican principalmente a familias restringidas de circuitos, como los circuitos de profundidad constante o monótonos. Identificar un problema NP que requiera circuitos de tamaño superpolinómico implicaría que P $\neq$ NP, subrayando la dificultad de este desafío.

En este trabajo proponemos métricas para el análisis de funciones booleanas y los circuitos que las implementan. Aplicamos estas métricas a conjuntos completos de circuitos de tamaño mínimo, con el objetivo de analizar su estructura y tratar de entender que hace que una función requiera circuitos grandes.

Palabras clave

Complejidad de circuitos, Funciones booleanas, Circuitos mínimos, Análisis de grafos, Endogamia.

Abstract

Circuit complexity, a branch of computational complexity theory, has seen limited progress in establishing lower bounds for the minimum size of circuits that solve NP-complete problems. Existing bounds primarily apply to restricted families of circuits, such as constant-depth or monotone circuits. Identifying an NP problem that requires superpolynomial-size circuits would imply that P $\neq$ NP, highlighting the difficulty of this challenge.

In this work, we propose metrics for analyzing Boolean functions and the circuits that implement them. We apply these metrics to complete sets of minimum-size circuits, with the aim of studying their structure and understanding what makes a function require large circuits.

Keywords

Circuit complexity, Boolean functions, Minimal circuits, Graph analysis, Endogamy.

Contents

1 Introduction

Circuit complexity is an area of computational complexity theory that deals with classifying Boolean functions in relation to the size and depth of the Boolean circuits that compute them. These structural notions of circuits play the role of time and space in Turing machines. In the same way complexity classes are usually characterized according to their resource requirements, circuit complexity classes do the same describing families of circuits of specific sizes. $P_{/\text{poly}}$ for example, consists of all decision problems that can be solved by polynomial-sized circuit families. P is known to be in $\text{P}_{/\text{poly}}$, which means that if we are able to find a NP problem that cannot be solved by any polynomial-size family or circuits, then $\text{P} \neq \text{NP}$.

For this reason, one of the main interests of researchers in computational complexity is to establish lower bounds for specific Boolean functions, as this contributes to the separation of complexity classes.

Finding the minimum circuits for a given boolean function can be useful in order to study its structure and the essence of its complexity. Not only in a theoretical sense, but in pragmatic one; due to its applications in physical computer circuit design.

1.1 Goals

In this work, we aim to discuss the difficulty of finding minimum Boolean circuits, along with techniques developed for the corresponding task in both the theory and practice of the field. We will review previous research, introduce and test a new approach to the concept of endogamy in the structure of Boolean circuits (borrowed from biology) and present programming tools, curated circuit databases, and precomputed metrics that may accelerate future studies.

Our main goal is to help establish some foundations for approaching the structural analysis study of Boolean circuits, such as selecting desirable properties of both the specific Boolean circuits and the metrics computed over them.

1.2 Work plan

The structure of this work is the following:

- **Prelimaries**: Some background definitions are introduced.
- **Databases and tools**: We will discuss what makes a good database of Boolean circuits for the required needs, and we will introduce the 2 databases of minimal circuits used. Some tools have also been developed in order to manage mentioned databases, computing the proposed metrics, and producing the statistical analysis experiments. These tools are briefly explained, and some use cases shown.
- **Metrics**: Endogamy and other proposed metrics are introduced and discussed.
- **Experiments**: How data has been obtained, and the performed statistical analysis are introduced.
- **Conclusions**: Results obtained on the previous section are discussed.
- **Future work**: Improvements are new research direction suggestions have been developed.

2 Preliminaries

We begin by introducing and defining the key concepts and notation that will be used throughout this work:

2.1 Functions

Definition 1 (Boolean function)**.** Let $n \in \mathbb{N}$ and $\mathbb{B}$ the Boolean set $\{0, 1\}$. A Boolean function of n variables is a mapping

$$f : \mathbb{B}^n \to \mathbb{B}^m$$

Throughout this work, we will be focusing on $f : \mathbb{B}^n \to \mathbb{B}$ functions, and we will use the binary representation of a Boolean functions, where each symbol 0 or 1 represents the value $f(x)$ for $x \in \mathbb{B}^n$, and each x is listed in lexicographic order. Consider the function defined by the following truth table:

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

This function can be represented by the vector of output values:

$$\mathbf{0001}$$

For convenience, this vector may also be identified by its decimal value:

$$\mathbf{0001_2} = 1_{10}$$

As we will see, referring to each Boolean function by the decimal value of the binary number formed by its outputs is highly practical, as it greatly facilitates computational manipulation.

2.2 Circuits

The circuit is the object that describes how to compute the result of a Boolean function from the successive application of a set of Boolean operations on

some inputs. Intuitively, it can be seen as directed acyclic graph (DAG) with the following components:[1]

- **Sources:** Vertices with no incoming edges.
- **Sinks:** Vertices with no outgoing edges.
- **Gates:** Non-source vertices labeled with the Boolean operations they compute.
- **Fan-in:** The number of inputs (incoming edges) that a gate can accept, usually 2 (e.g., AND, XOR) or 1 (e.g., NOT, identity).
- **Fan-out:** The number of outputs (outgoing edges) a non-sink vertex can have, usually greater than 1.
- **Size:** The total number of vertices in the circuit.

The value of $C(x)$ is the value computed in the sinks, where the value of each vertex is computed recursively by applying its Boolean operation to its inputs until it reaches the values at the sources, which are fixed.

More formally:

Definition 2 (Boolean Circuit)**.** Let $C = (V, E, \text{op})$ be a Boolean circuit, where:

- V is the set of vertices,
- $E \subseteq V \times V$ is the set of edges,
- $G = (V, E)$ forms a DAG,
- $S \subseteq V$ is the set of sources (vertices with no incoming edges),
- $T \subseteq V$ is the set of sinks (vertices with no outgoing edges),
- $\text{op} : V \setminus S \to \mathcal{O}$ assigns a Boolean operation to each gate.

[1]The definition of a circuit is adapted from [1], is generalized to the number of Boolean operations, the number of sinks (to represent mappings $f : \mathbb{B}^n \to \mathbb{B}^m$), and the number of inputs and outputs allowed per gate

Given $x = (x_1, \ldots, x_n)$ where n is the number of sources and each x_i the value at input i, then, define the value function $\mathrm{val}_C : V \to \{0, 1\}$ recursively by

$$\mathrm{val}_C(v) = \begin{cases} x_v, & \text{if } v \in S \text{ (source)} \\ \mathrm{op}(v)\big(\mathrm{val}_C(u_1), \ldots, \mathrm{val}_C(u_k)\big), & \text{if } v \text{ is a gate with input vertex } u_1, \ldots, u_k \end{cases}$$

Finally, the output $y \in \mathbb{B}^m$ of the circuit on input $x \in \mathbb{B}^n$ is denoted by

$$C(x) = (\mathrm{val}_C(t))_{t \in T}.$$

Here, $\mathrm{op}(v)$ denotes the Boolean operation at gate v, and $\mathrm{val}_C(v)$ is the Boolean value computed at vertex v.

Here is a example of a circuit implementing 5 input AND function:

Example 1. (AND_5 boolean circuit)

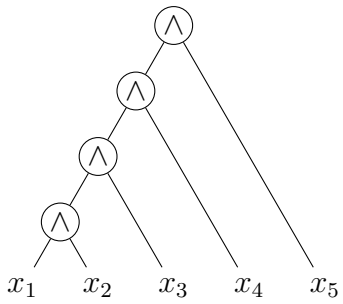


Figure 1: AND_5 Boolean circuit

There are also some other definitions that make more notational sense [1]:

Definition 3 (Boolean straight-line program)**.** A Boolean straight line program of length T with input variables $x_1, x_2, \ldots, x_n \in \{0, 1\}$ is a sequence of T statements of the form

$$y_i = z_{i1} \ \circ_{op} \ z_{i2}, \quad \text{for } i = 1, 2, \ldots, T$$

where $\circ_{op}$ is either $\vee$ or $\wedge$, and each z_{i1}, z_{i2} is either an input variable, the negation of an input variable, or y_j for some $j < i$. For every setting of values to the input variables, the straight-line computation consists in executing these simple statements in order, thereby finding values for $y_1, y_2, \ldots, y_T$. The output of the computation is the value of y_T.

Here is the Boolean straight-line program that computes AND_5:

$$y_1 = x_1 \wedge x_2$$
$$y_2 = y_1 \wedge x_3$$
$$y_3 = y_2 \wedge x_4$$
$$y_4 = y_3 \wedge x_5$$

Let's present yet another definition that allows more general gates [12]:

Definition 4 (Boolean chain)**.** For functions of n variables $(x_1, \ldots, x_n)$, a Boolean chain is a sequence of binary steps $(x_{n+1}, \ldots, x_{n+r})$ with the property that each step combines two of the preceding steps:

$$x_i = x_{j(i)} \circ_i x_{k(i)}, \quad \text{for } n+1 \leq i \leq n+r,$$

where $1 \leq j(i) < i$ and $1 \leq k(i) < i$, and where $\circ_i$ is one of the sixteen binary operators. Notice how this is similar to a *Boolean straight-line program*[2]

Later, we will see how the Boolean chain definition is going to be useful as an internal representation for circuits 3.

2.3 Boolean Expression

Definition 5 (Boolean expression)**.** Let $X = \{x_1, \ldots, x_n\}$ be a finite set of Boolean variables, and let $\mathcal{O}_2$ be a set of binary Boolean operators. A *Boolean formula* over X is any expression generated by the following grammar:

$$\phi ::= 0 \mid 1 \mid x_i \mid \neg\phi \mid (\phi \ o \ \phi), \quad x_i \in X, \ o \in \mathcal{O}_2$$

where

- 0 and 1 are the Boolean constants,
- $x_i \in X$ are Boolean variables,
- $\neg\phi$ denotes the negation of a formula ϕ,
- $(\phi \ o \ \phi)$ denotes the application of a binary operator $o \in \mathcal{O}_2$ in infix notation.

[2]Also notice how straight line programs allow the unary negation operator, while Boolean chains embed them in their operations

2.4 Size

Size matters, and for this particular work we won't only need a notion of size for the Boolean circuit. In the following section, different ways of studying the size of Boolean functions are going to be introduced. Some of these definitions were previously introduced in *The Art of Computer Programming, Volume 4* [12], by Don Knuth.

Definition 6 (Combinatorial complexity $C(f)$)**.** The *Combinatorial complexity* of a function f is the length of the shortest *Boolean chain* that computes f. This is also the number of two-input nodes in the smallest circuit that represent the function.

Definition 7 (Length $L(f)$)**.** The *Length* of a function f is the number of binary operators in the shortest formula for f.

Definition 8 (Depth $D(f)$)**.** The *Depth* of a function f is the minimum length of the longest path from any of the sources to the sink among all circuits that represent f.

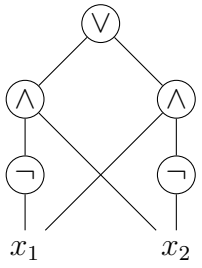
Example 2 (Size of XOR))**.** Given the binary base $\{\wedge, \vee, \neg\}$, these are the corresponding Boolean chain, Boolean formula and Boolean circuit:

$$\begin{aligned} x_3 &= \bar{x}_1 \wedge x_2 \\ x_4 &= x_1 \wedge \bar{x}_2 \\ x_5 &= x_3 \vee x_4. \end{aligned}$$

(a) Boolean Chain

$$(\neg x_1 \wedge x_2) \vee (x_1 \wedge \neg x_2)$$

(b) Boolean Formula



(c) Boolean circuit diagram

The different sizes notions of this function are: Combinatorial complexity $C(XOR)$: 3, Length $L(XOR)$: 3 and Depth $D(XOR)$: 2

2.5 NPN Classes

Changing the order of two input variables of a Boolean Function is called a swap. Replacing a variable by its complement is called a flip. There are eight possible transformations of the function performed by swapping and flipping pairs of adjacent support variables. This observation can be generalized as follows: For an N-variable Boolean Function, there are 2^N ways of transforming it by flipping its variables and $N!$ ways of transforming it by swapping its variables. Additionally, there are two polarities of the function derived by complementing its output. In total, there are $M = 2^{N+1} * N!$ transformations of the function derived by swapping its inputs and flipping its inputs and output.

Definition 9. (NPN Class)
Consider the set of all Boolean functions derived by M transformations of a Boolean function F, as described above. These functions constitute the NPN class of function F.

The *NPN canonical form* of function F is one function belonging to its NPN class, also called the representative of this class. The representative is selected uniquely among the functions belonging to the given class. For example, in some applications, the representative is selected as the function whose truth table, considered as an integer number, has the smallest numeric value. In other applications, the representative is uniquely selected by some deterministic algorithm. The number of NPN classes is much smaller than the number of Boolean functions. For example, all 2^{16} Boolean functions of 4 variables split into 222 NPN classes.

That's where NPN Classes become useful. We can work with the NPN Canonical form of a given class, and analyze its structure. This way, we can make sure that previously defined size definitions hold under inputs negation and permutation and output negation.

This is also introduces the following intuition: **A good metric defined over a Boolean circuit that characterizes a non-trivial property should hold within all NPN class members.**

3 Databases and tools

Studying all Boolean functions and the minimal circuits that implement them is, unfortunately, beyond the finite nature of machines (and humans). For $n = 4$, the number of Boolean functions we can generate is 65,536, and for $n = 5$, 4,294,967,296. Assuming $n = 5$, on a modern machine we could represent each function uniquely using a uint32, which would require approximately 17.18 GB of memory. This is relatively practical. However, if we increase n to 6, we could use a uint64 type, but we would then need 147.57 EB[3]. The functions we can observe with our own eyes are thus constrained to a perhaps very small n. Nevertheless, this should not discourage us from attempting to shed light on the problem through empirical and experimental approaches.

3.1 Databases

One approach to this problem is to use randomly generated functions and circuits according to some desired distribution, and then select a sufficiently large sample set to work with. This has been the approach taken in previous work by other students [15].

Random generation, however, does not allow us to directly obtain minimal-size circuits. To understand the lower bounds governing them, we need to be able to obtain minimal circuits. Obtaining minimal circuits is non-trivial. The problem of finding minimal circuits (MCSP) lies in NP, but it is unknown whether it is NP-hard [8].

The literature contains many solutions for circuit minimization, ranging from the well-known Quine–McCluskey algorithm [14] to industrial software such as ABC [2] and ESPRESSO, which rely heavily on heuristics (and do not guarantee minimal circuits), or SAT-based synthesis algorithms [7], which do allow obtaining minimal circuits for functions with a moderate number of inputs.

Even with a good method for generating circuits, questions remain about which functions to choose, since even for $n = 5$ the number of functions to minimize is too large to cover exhaustively, and about which constraints to impose during generation. Constraints may include the selection of allowed

[3]Exabyte, 10^9 GB.

binary gates, as well as restrictions on depth, width, fanout, fanin, or even specific topologies.

To address the issue of specific selection, we decided to use a canonical representative for each NPN class of Boolean functions. Using these representatives allows us, on one hand, to reuse circuits at zero cost, since obtaining the circuits for any other function in the class is simply a matter of inverting or permuting inputs, or inverting the output. This reduces the number of functions to consider for $n = 5$ from 4,294,967,296 to only 616,126. However, this approach is not perfect. The metrics we choose should be robust under any of the transformations mentioned above.

But this approach is not perfect, as Ernst says here [4] "Unlike the P-equivalence class [4], functions within the same NPN-equivalence class may have different minimum networks with different structures. The structure obtained by transforming a NAND2 implementation for one function in the class from a representative function may not always produce an optimal solution.". This means, that depending of the basis of the gate selection, size optimality might not be preserved along the NPN class functions.

The following are some of the databases we have used for our study. Fortunately, we have access to two databases of minimal circuits provided by Mishchenko et al. [13] and Goucher [6].

3.1.1 Minimum Fanout-Free Circuit Structures

Mishchenko et al. focus on the logic synthesis of minimum-size circuits[5] for small Boolean functions through structural enumeration [13]. The authors reproduce and independently verify known results for completely specified functions with up to five inputs, as initially presented by Donald Knuth[6].

- They provide for the first time the number of structurally different minimum boolean expressions for each NPN class of Boolean functions up to five inputs.

[4] P class is defined like NPN, but without negation of inputs and outputs.

[5] Although Mishchenko et al. refer to their contribution as minimun-size circuits, from this point onward we will refer to them as Boolean expressions rather than circuits, since they do not capture the notion of gate reuse.

[6] Knuth provides the different minimal the values for $(D(f), C(f) \text{and} L(f)$ for 4 inputs Boolean functions and some 5 input Boolean functions [12].

- A library of all minimum circuit structures for Boolean functions with up to five inputs in the *AIG* format [7].

We are interested in the first database, which has the following characteristics: First, the database uses the gate set AND, XOR, NOT. This means that any structural analysis we perform is constrained by this choice of gates. Moreover, these circuits do not allow gate reuse. Part of our interest in analyzing the circuits revolves around the idea of capturing a notion of "inbreeding" and observing what happens when it is low or high. Our notion of inbreeding should be high when gates are reused. Fortunately, although gate reuse is not allowed, input reuse does occur, so our metric remains interesting to test.

3.1.2 Optimal Five

In 2005, Donald Knuth developed a comprehensive approach to determine the minimal circuit implementations for all five-input, one-output Boolean functions across the 616,126 NPN classes. His method involved "canonizing" functions by identifying the lexicographically smallest truth table equivalent under NPN transformations, enabling constant-time lookups using hash tables [5] Through this methodology, Knuth successfully determined the optimal gate counts for all but 6 NPN classes, which required additional computational effort to resolve. The *Optimal5* [6] database was created by Goucher, to unify and optimize Knuth's solutions for five-input Boolean circuits, addressing two key objectives:

1. **Consolidation** – Knuth's results were scattered across multiple sources (databases, text files, and documentation). Optimal5 integrates these into a single, structured repository, ensuring completeness and ease of access.

2. **Efficient Canonization** – The database enhances the speed of function canonization, enabling rapid lookups.

Knuth's original approach canonized functions by evaluating all 3,840 possible input permutations and negations. Despite using bitwise tricks, this process took approximately 10 microseconds per function. To improve efficiency, Optimal5 employs:

[7] AND-inverter graph, which is a directed acyclic graph of 2-input AND gates and 1-input NOT gates.

- Hypercube-Based Reduction:
  Boolean functions are visualized as colored vertices of a 5D hypercube, where transformations correspond to rotations and reflections. By fixing a canonical "top face," the search space is reduced from 3,840 to 384 permutations, leading to a 10× speedup (1.6 microseconds).

- Group-Theoretic Pruning:
  Instead of checking all 384 remaining permutations, only those within the stabilizer subgroup of the top face are traversed. This is further optimized using the **Held-Karp algorithm**[8] for finding optimal Hamiltonian paths in the Cayley graph, reducing unnecessary computations.

- Automated Code Generation:
  Instead of manually handling 75 distinct stabilizer subgroups, a metaprogram was used to generate efficient lookup tables and hashing functions.

With these optimizations, function canonization now averages **686 nanoseconds**, a significant improvement over Knuth's original implementation at **10 microseconds**[9]
This acceleration makes *Optimal5* a powerful tool for Boolean function classification and circuit design. Goucher presents this curated database as a C++ code repository and a file named "knuthies.dat" of size 9.2M, which stores the minimal Boolean chains for each function. This code will be included as a library that we can access from our Python tools, in order to make the process of manipulating the functions as ergonomic as possible.

[8]Held-Karp-algorithm solves the Travelling salesman problem (TSP) in exponential time

[9]Goucher doesn't specify the characteristics of his testing machine while talking about time, but this might suggests that he's using a contemporary consumer PC

3.2 Tools

The convenient use and adaptation of these databases, as well as the tools for handling Boolean functions, circuits, and expressions, together with the computation of the proposed metrics and subsequent statistical analysis, have been collected in the form of utilities implemented in C++, C, and Python. The objective is to facilitate usability so that researchers who wish to do so may easily access these databases and tools.
The main components are as follows:

- `o5.py`: Provides bindings to the *Optimal5* database. It manages the low-level usage of the dynamic library `o5lib`, which acts as an interface to Goucher's *Optimal5* library, making it accessible from Python. A common usage example is shown below:

```
from o5 import O5
"""
O5() loads the dynamic library, and manages its state
"""
knuth = O5()
"""
Boolean functions are stored as python int type,
but treated as uint32 values.

Exmaple 1:
fun     -> 65535
fun_hex -> 0x0000ffff
fun_bin -> 0b00000000000000001111111111111111
"""
fun = 65535
"""
cir_str:    Boolean chain string representation of
            the minimal circuit for `fun`.
"""
cir_str = knuth.fun_to_cir(fun)
"""
Result:

    cir_str == "result = x1;"

What if we flip the function vector bits?

Example 2:
```

```
flip_fun     -> 4294901760
flip_fun_bin -> 0b11111111111111110000000000000000
flip_fun_hex -> 0xffff0000
"""
flip_fun = 4294901760
flip_cir_str = knuth.fun_to_cir(flip_fun)
"""
Result:

    flip_cir_str == "result = ~x1";

As expected, the output circuit is exactly the same,
but with the output negated.
"""
```

- `circuit_parser.py`: This module contains, among other components, the tools for parsing Goucher's circuits and Mishchenko's expressions, so that they can be evaluated given specific input values. These tools have been used primarily to independently verify that the circuits and expressions from *Optimal5* and *Lflib5*[10] correspond to the intended input functions.

```
from o5 import O5
from circuit_parser import NormalCircuit, AND_XOR_NOT_Exp


knuth = O5()
nc = NormalCircuit()

fun = 65535
cir_str = knuth.fun_to_cir(fun)

tree = nc.parse(cir_str)
fun_reconstruction = nc.tree_to_fun(tree)

print(fun == fun_reconstruction) # True


"""
Eval for input x = 1 (dec), this is,

```

[10] `Lflib5.txt` is the name of the database provided by Mishchenko et al.

```
x_bin    = 0b00001
x_inputs = {x1:0, x2:0, x3:0, x4:0, x5:1}

"""
x = 1
res = nc.eval_x(tree, x)
"""
Result:

        res == False

This corresponds to the second position of
fun = 65535 binary representation:

    0b00000000000000001111111111111111
      ^^                               ^
      ||                               |
      |--- 1 position                  31 position
      |
      0 position


AND_XOR_NOT_Expr works similarly for parsing
the provided function expressions:
"""
exp_dec = 7339785
exp_hex = 0x006fff09
exp_str = "~(e^(~d&~((~e^c)&~(a^b))))"

axn = AND_XOR_NOT_Exp()
t = axn.parse(exp_str)
f = axn.tree_to_fun(t)

exp_dec == f # True
```

- `data_manager.py`: Loading and saving data module.
- `stats.py`: Statistical experiments module.
- `metrics.py`: Metrics computing tools.
- `util.py`: Additional tools.

4 Metrics

Obtaining good metrics is one of the main objectives of this work. A metric is considered interesting when its computational cost is affordable and when, for our specific dataset, its values are preserved among members of the same NPN class. Moreover, such a metric should provide valuable information regarding the structure of minimal circuits. Finding a metric of this kind that, for instance, strongly correlates with the number of gates in minimal circuits would be of great interest. Although, as we will see in the Conclusions section, this task is not trivial.

We want to be able to determine what makes a function require large circuits. Obtaining a metric, or rather an algorithm, that determines the size of the minimal circuit would be equivalent to solving the Minimum Circuit Size Problem (MCSP) [10], and this problem is in NP.

This is perhaps the first and most fundamental observation to make: **The time required to obtain any proposed perfect metric (a metric that could unambiguously determine whether a circuit is minimal or not) will be linked to the problem of finding the minimal circuit.**

However, we can try to analyze non-perfect metrics whose computation is efficient and which serves as an approximation for the intended purpose.

The choice of size metrics computed on the structures[11] is necessary, those generated from the binary representation of the circuit feel essential, and the inbreeding metric can only be described as intuitive. The implementation of these metrics can be found in the project repository [3], in `metrics.py` module.

[11] DAGs associated to circuits and expressions.

4.1 Endogamy

Endogamy is the practice of mating within social groups or specific ethnicities, excluding individuals from outside the group for reproduction. This practice is prevalent in many species and can contribute to the transmission of genetic disorders due to **the founder effect**[12] within closed populations. To measure the genetic diversity of a species is an activity of interest, because low genetic diversity can lead to the development of diseases and deteriorate the overall quality of newly born individuals. In this context, the idea of the **coefficient of inbreeding** is developed.

The **coefficient of inbreeding** is a number that measures how inbred an individual is. Specifically, it is the probability that two alleles at any locus in an individual are identical by descent from a common ancestor of both parents. A higher coefficient will make the traits of the offspring more predictable, but also increase the risk of health problems. In dog breeding, it is recommended to keep the coefficient below 5%; however, in some breeds, this may not be possible without external crossings.

Just as two individuals of a species can mate and produce a new individual, we can draw an analogy between parents and the inputs of a particular gate, and the offspring with the outputs. We asked ourselves whether some of the properties captured by the **coefficient of inbreeding** can be transferred naturally to the circuits, or if they might be useful for making predictions about other properties of interest.

[12]The loss of genetic variation that occurs when a new population is established by a very small number of individuals from a larger population

4.1.1 Coefficient of Inbreeding

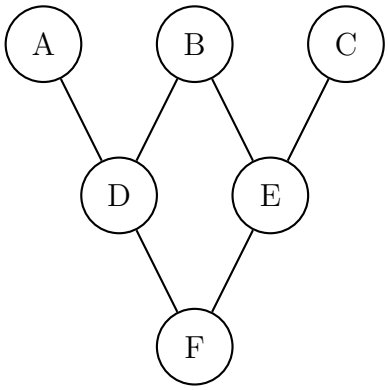
The coefficient of inbreeding was introduced by the american geneticist Sewall Wright in 1922, trying to improve previously stated methods in the field [16]:

Definition 10. (Coefficient of Inbreeding (COI))[13]

$$f_o = \sum_{i=1}^{a} \frac{1}{2}^{n+n'+1} (1 + f_i)$$

where a is the number of common ancestors shared by the parents, n and n' are the number of generations from father and mother respectively, to the common ancestor i, and f_i is the coefficient of inbreeding of the ancestor i.

Example 3. COI of F, whose parents share a parent:



First, we find the common ancestors of F´s parents: $\{B\}$. We count how many steps are there from each parent to the common ancestor. We proceed the calculation

$$(D - B) \to n = 1$$
$$(E - B) \to n' = 1$$
$$f_B = 0$$
$$f_F = \sum_{i \in \{B\}} \frac{1}{2}^{n+n'+1} (1 + f_i)$$
$$f_F = \frac{1}{2}^{1+1+1} (1 + 0) = \frac{1}{2}^{3} = 0.125$$

[13]Different formulations of COI can be found. This one belongs to the original work introducing it. [16]

4.2 Hamming weight

Definition 11 (Hamming weight)**.** The Hamming wight of a binary vector (or string) is the number of positions in which the vector has a 1. Formally, for a binary vector $x = (x_1, x_2, \ldots, x_n) \in \{0,1\}^n$, its Hamming weight $w_H(x)$ is defined as

$$w_H(x) = \sum_{i=1}^{n} x_i.$$

Example 4. Consider $x = 1011010$. The positions with 1s are 1, 3, 4, and 6, so

$$w_H(x) = 4.$$

Moreover, it also provides a natural way to describe and classify many well-known Boolean functions. Several important Boolean functions can be expressed directly in terms of the Hamming weight:

- **OR function:** $f(x) = 1$ if and only if $w_H(x) \geq 1$.
- **AND function:** $f(x) = 1$ if and only if $w_H(x) = n$.
- **Parity function:** $f(x) = 1$ if and only if $w_H(x)$ is odd.
- **Majority function:** $f(x) = 1$ if and only if $w_H(x) > n/2$.

It also provides important insight into the structure and properties of a Boolean function.

- **Balance:** If the Hamming weight of output vector is exactly 2^{n-1}, the function is *balanced*, outputting 1 for half of the inputs and 0 for the other half. Balanced functions are often more complex to implement in minimal circuits.
- **Symmetry and majority functions:** For *symmetric functions*, the output depends only on the Hamming weight of the input. For example, the *majority function*.
- **Distribution of values:** Functions with *low Hamming weight* (few ones) can often be implemented efficiently using sum-of-products forms, since each 1 corresponds to a minterm. Conversely, functions with *high Hamming weight* (few zeros) can be efficiently represented using product-of-sums forms.

This notion of balance can be interesting while analyzing data.

4.3 Entropy

Definition 12 (Binary Entropy)**.** Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function. Let

$n_0 =$ number of 0s in the outputs of f, $\quad n_1 =$ number of 1s in the outputs of f

and define probabilities

$$p_0 = \frac{n_0}{2^n}, \quad p_1 = \frac{n_1}{2^n}.$$

Then the entropy $H(f)$ of f is

$$H(f) = -\big(p_0 \log_2 p_0 + p_1 \log_2 p_1\big),$$

Entropy is used in the context of information theory, but it can be useful at revealing some interesting information about functions, similar to Hamming weight:

- $H(f) = 0$: The function is constant (all outputs 0 or all outputs 1).
- $H(f) = 1$: The function is balanced (equal number of 0s and 1s).
- $0 < H(f) < 1$: The function is biased (outputs 0 and 1 with unequal frequency).

Example 5. (Entropy of binary string $x = 1011010$)

- Count outputs: $n_0 = 3$, $n_1 = 4$
- Probabilities: $p_0 = \frac{3}{7} \approx 0.4286$, $p_1 = \frac{4}{7} \approx 0.5714$
- Entropy:

$$H(f) = -\big(p_0 \log_2 p_1 + p_1 \log_2 p_1\big) \approx 0.985 \text{ bits}$$

The function is slightly biased, outputs 0 and 1 are almost balanced.

4.4 Sensitivity and Block Sensitivity

Definition 13 (Sensitivity)**.** Let $f : \{0,1\}^n \to \{0,1\}$ be a Boolean function, and let $x \in \{0,1\}^n$. The sensitivity of f at input x, denoted $s(f,x)$, is the number of bits i such that flipping the i-th bit of x changes the value of f:

$$s(f,x) = |\{i \in \{1,\ldots,n\} : f(x) \neq f(x^{(i)})\}|,$$

where $x^{(i)}$ is x with the i-th bit flipped.

The **sensitivity of** f is the maximum over all inputs:

$$s(f) = \max_{x \in \{0,1\}^n} s(f,x).$$

Block Sensitivity: The block sensitivity of f at input x, denoted $bs(f,x)$, is the maximum number of disjoint subsets (blocks) $B_1,\ldots,B_k \subseteq \{1,\ldots,n\}$ such that flipping all bits in any block B_j changes $f(x)$:

$$f(x) \neq f(x^{B_j}),$$

where x^{B_j} is x with all bits in B_j flipped.
The **block sensitivity of** f is

$$bs(f) = \max_{x \in \{0,1\}^n} bs(f,x).$$

Example 6. ($s(f)$ and $bs(s)$ of $f(x_1,x_2,x_3) = x_1 \wedge x_2$ and $x = 110$)

- Sensitivity at x: flipping x_1 or x_2 changes $f(x)$, but flipping x_3 does not, so $s(f,x) = 2$.
- Block sensitivity at x: the singleton blocks $\{1\}$ and $\{2\}$ are disjoint and flipping each changes $f(x)$, so $bs(f,x) = 2$.

There is also an interesting theorem proven in 2019 about sensitivity [9], which states that:

Theorem 1 (**Sensitivity Theorem (Hao Huang, 2019):**)**.** *For any Boolean function f,*

$$s(f) \geq \sqrt{\deg(f)}.$$

where $deg(f)$ is the degree of the multilinear function of f.

Definition 14 (Degree of a Multilinear Function)**.** It is the maximum number of different variables that appear multiplied together in any single term of a multilinear polynomial

Example 7. (Degree of the parity function $f(x_1, x_2, x_3) = x_1 \oplus x_2 \oplus x_3$) Let's get its multilinear polynomial:

$$f(x_1, x_2, x_3) = x_1 + x_2 + x_3 - 2(x_1x_2 + x_1x_3 + x_2x_3) + 4x_1x_2x_3,$$

The largest monomial has 3 distinct variables, so $\deg(f) = 3$.

Definition 15 (**Multilinear function**)**.** It is a polynomial over n Boolean variables of the form:

$$f(x_1, \ldots, x_n) = \sum_{S \subseteq \{1,\ldots,n\}} c_S \prod_{i \in S} x_i,$$

where:

- S runs over all subsets of $\{1, \ldots, n\}$, including the empty set,
- $c_S \in \mathbb{R}$ are coefficients,
- each monomial $\prod_{i \in S} x_i$ is **multilinear**, meaning no variable appears with power higher than 1.

Example: For $f(x_1, x_2) = x_1 \wedge x_2$, the multilinear polynomial is

$$f(x_1, x_2) = x_1x_2.$$

Degree captures somehow the interaction of all the variables: Functions with higher degree must have some input where many bits influence the output, e.g., high sensitivity. So, an algebraic property like the degree, is connected with a combinatorial one, like sensitivity.

4.5 Size, depth, and length

These three notions have been previously defined 2.4. Being able to estimate any of these metrics in practice could help us shed some light on the problem. But our ultimate goal remains to be able to estimate the size.

5 Experiments

5.1 Data selection

For our experiments, we extracted all minimal circuits from **Optimal5** and processed the minimal expressions from Mishchenko's **Lflib**. In the case of expressions, we had access to multiple minimal solutions per class. Many of these multiple solutions are variations in associativity; therefore, we selected the first expression available in the database for each function, without loss of generality.

We selected one canonical representative for each NPN class (i.e., 616126 distinct functions) and assigned to it the corresponding circuit and expression. For each triplet, we computed the proposed metrics. Both circuits and expressions were converted into directed acyclic graphs, on which we computed COI-related metrics as well as size-related ones. The remaining metrics were computed directly on the binary representation of the associated Boolean function.

The advantage of the metrics we have chosen is that they are relatively robust with respect to the choice of any representative within a class, but with some caveats:

- **Graph-based metrics** remain invariant for every function in the class, since each minimal circuit is structurally equivalent across all functions of the class.[14]
- **For the remaining metrics**, entropy, sensitivity, and block sensitivity are preserved across each NPN class. This is not the case for the Hamming weight: although it is robust under input permutations and negations, it is not invariant under output negation.
- **For expressions and COI**, we decided to remove NOT gates when generating the corresponding graph, in order to avoid cases in which the two parents are clones.

[14]We do not consider input and output negations as separate gates when interpreting the circuit as a graph.

5.2 Analysis

All of the aforementioned metrics were computed, and both Pearson and Spearman correlation analyses were performed. The goal was to determine whether correlation analysis alone can provide insight into the factors driving the number of gates.

Here are finally the results of the collected data. Although some of the results were expected (such as the high correlation between Hamming weight and the entropy of the bit vectors representing the function, or between sensitivity and block sensitivity) these findings are not interesting enough, since they are measures computed explicitly on the functions themselves. What would be truly interesting is to obtain some relationship between the measures computed on the function vector and those computed on its minimal circuit or minimal expression. Unfortunately, this does not seem to be the case.

However, we can observe a clear correlation between the number of gates in the minimal circuit and the number of binary operations in the shortest expression to be 0.84. This might be worth discussing. As we mentioned earlier, our selection of circuits and functions, although minimal, is structurally biased, since the choice of their logical gate basis and operations is arbitrary. Circuits are built on the normal basis of gates $\{\wedge, \vee, \oplus, >\}$, AND, OR, XOR, and Non-implication, respectively. Meanwhile, minimal expressions use $\{\wedge, \oplus, \neg\}$.

These two basis are also **functionally complete**, so there shouldn't be any surprises.[15]. On the other hand, while in circuits it is allowed to reuse gates (that is, to reuse already computed values) in Boolean expressions this is not the case; which should affect their structure. However, they seem to correlate. This suggests that the **"irreducible" essence of the functions might not be avoided even when reusing gates**.

This is useful for us to talk about the notion of **biological inbreeding**, taken directly from biology. Let us recall that its implementation is straightforward: a Boolean circuit is interpreted as a family tree, where the output would be the descendant of the family, and the inputs of the circuit would be

[15] A **functionally complete** set of logical connectives or Boolean operators is one that can be used to express all possible truth tables by combining members of the set into a Boolean expression, e.g the one formed just by the NAND operator

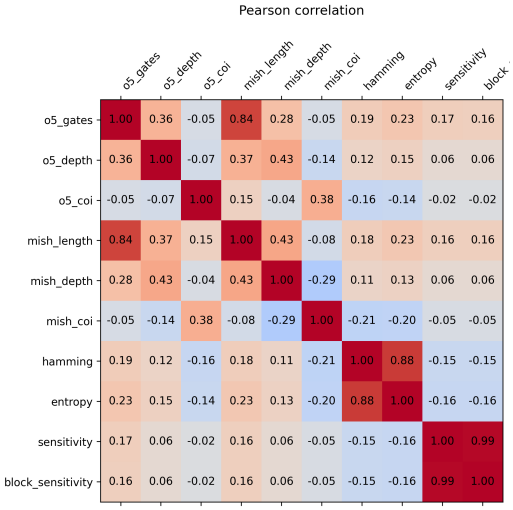


Figure 3: Pearson correlation

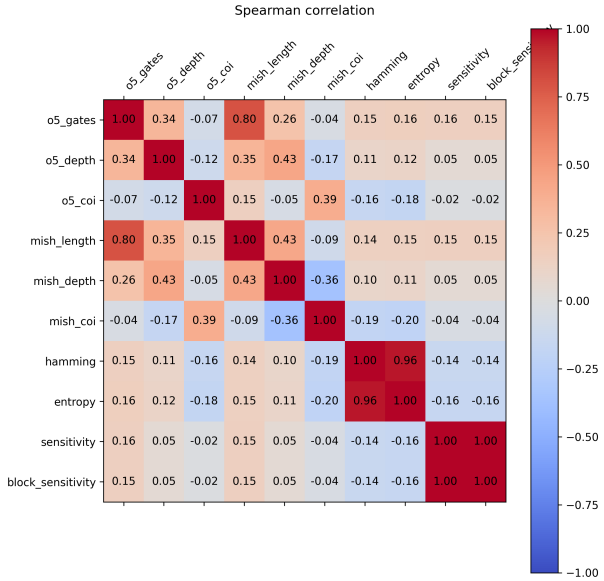


Figure 4: Spearman correlation

the most distant ancestors. We do not distinguish the type of gate; we simply consider each gate as an individual in the tree. The results were expected not to be as good as those we might have obtained from using databases of minimal circuits on a single-gate basis, such as NAND or NOR (because with no distinction of gates, the results could only be analyzed in structural terms), and indeed that was the case. The correlation of *o5_coi* was close to zero and negative for all the proposed metrics, with the exception of *mish_coi* at 0.38 and *length* at 0.15, the COI and L(f) computed on the database of minimal Boolean expressions by Mishchenko et al. Recall that *mish_coi* is the inbreeding coefficient of structures where the common descendants can only be the inputs, unlike in circuits where intermediate gates can also be ancestors, which supported our intuition that "high inbreeding" should capture gate reuse.

6 Conclusions

Although some correlation has been observed between metrics computed on circuits (o5_depth, o5_gates, o5_coi) and metrics computed on the binary representation (hamming, entropy, sensitivity, block_sensitivity), these do not turn out to be very significant.

The metrics that correlate the most are the expected ones: **o5_gates and mish_length correlate at 0.84**, which, as we mentioned in the previous chapter, was to be expected since they are both measures that capture the essence of size; even though circuits and expressions are objects generated with different structural constraints.[16]

Next, with a **correlation of 0.43, we find mish_depth and mish_length**, as well as mish_depth and o5_depth. Once again, these pairs are not surprising. We could have even expected a higher correlation. It is intuitive to see that by not allowing gate reuse, each new gate added will contribute more, on average, to increasing the height of the tree derived from the expression. On the other hand, in the same way as o5_gates and mish_length, o5_depth and mish_depth and the other o5/mish pairs preserve some correlation, because once again, they are defined analogously for their respective tree structures.

The high correlations present between the notions of entropy and hamming weight, or sensitivity and block sensitivity, will be ignored for reasons similar to those previously stated.

The values that turn out to correlate most negatively are the **notions of inbreeding, with -0.29** for the pair mish_coi and mish_depth. If our interest was that a notion capturing the abstract idea of inbreeding reflects gate reuse, and that gate reuse does not allow "complexity to be compacted", this seems like good news. Although this good news is overshadowed by the correlation between **o5_coi and o5_depth: -0.07**. In does not seem to reflect the equivalent.

[16]Recall that circuits reuse gates, while expressions only reuse inputs.

7 Future work

Some thoughts on what could be done from now on:

7.1 On the search for circuits and metrics

The results make it clear that this approach does not seem promising. Or at least, it does not appear so in the current state of the work. The idea of gathering curated databases of hard-to-obtain circuits with different structural characteristics, and producing metrics to evaluate them in order to shed light on the small-circuit problem, can be seen as either:

- A futile exercise, or
- A difficult path in which it is only a matter of time to find and refine the right data and metrics that allow us to obtain high-quality observations.

In the case that we are interested in continuing to explore this approach, the way forward is to try new datasets and refine the selected metrics.

7.2 Ideas for new metrics

Throughout the work, the idea of trying to find notions that would capture intuitions about information compression and the decomposition of Boolean functions was always present. Expanding further avenues of exploration became counterproductive at a certain point in the process. In particular, exploring the idea of how difficult it is to compress a string representing a function, and whether this relates to the size of the associated circuit. This idea has been previously stated more clearly, and explored more skillfully, of course [11].

7.3 On the right tools

My intuition regarding the obtained results is that refining the developed tools to make them as efficient as possible in terms of resource management and usability is, per se, an interesting goal, as it can be useful in the context of circuit development in the industrial domain. As the project progressed, some of the ideas that emerged were:

- Producing an extension module in C/C++ to efficiently implement some of the most computationally demanding sections of the code.

- Producing a module exclusively dedicated to the identification of types of Boolean functions, and using it for parameterized analysis, e.g., studying sets of functions that implement the majority function.
- Develop a procedure for finding minimal circuits where all binary gates are allowed, and compare them with the ones with other gate basis choice.

References

[1] Arora, S., and Barak, B. *Computational complexity: a modern approach, pp. 109*. Cambridge University Press, 2009.

[2] Brayton, R., and Mishchenko, A. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification* (2010), Springer, pp. 24–40.

[3] Celaya, J. Boolean circuit analysis. `https://github.com/blcksy/bca`, 2025.

[4] Ernst, E. A. *Optimal combinational multi-level logic synthesis*. University of Michigan, 2009.

[5] Goucher, A. P. Five-input boolean circuits. `https://cp4space.hatsya.com/2019/05/28/five-input-boolean-circuits/`, 2019.

[6] Goucher, A. P. Optimal circuits for all 5-input 1-output boolean functions repository. `https://gitlab.com/apgoucher/optimal5`, 2019.

[7] Haaswijk, W., Soeken, M., Mishchenko, A., and De Micheli, G. Sat-based exact synthesis: Encodings, topology families, and parallelism. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 39*, 4 (2019), 871–884.

[8] Hitchcock, J. M., and Pavan, A. On the np-completeness of the minimum circuit size problem. In *35th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2015)* (2015), Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 236–245.

[9] Huang, H. Induced subgraphs of hypercubes and a proof of the sensitivity conjecture. *Annals of Mathematics 190*, 3 (2019), 949–955.

[10] Kabanets, V., and Cai, J.-Y. Circuit minimization problem. In *Proceedings of the thirty-second annual ACM symposium on Theory of computing* (2000), pp. 73–79.

[11] Kabanets, V., and Kolokolova, A. Compression of boolean functions. In *Electronic Colloquium on Computational Complexity (ECCC)* (2013), vol. 20, p. 24.

[12] Knuth, D. E. *The Art of Computer Programming, Volume 4, Section 7.1.2, Boolean evaluation*. Pearson Education, Inc., 1998.

[13] Mishchenko, A. R., Brayton, Lee, S.-Y., and Jiang, J.-H. R. A library of minimum circuit structures of small boolean functions. `https://people.eecs.berkeley.edu/~alanmi/research/funenum/`, 2019.

[14] Quine, W. V. The problem of simplifying truth functions. *The American mathematical monthly 59*, 8 (1952), 521–531.

[15] Rubio Madrigal, C. Búsqueda de funciones booleanas complejas: construcciones mediante monotonía y relación entre repetitividad y endogamia. *UCM* (2022).

[16] Wright, S. Coefficients of inbreeding and relationship. *The American Naturalist 56*, 645 (1922), 330–338.