

UNIVERSIDADE ESTADUAL PAULISTA “JÚLIO DE MESQUITA FILHO”

FACULDADE DE ENGENHARIA

CAMPUS DE ILHA SOLTEIRA

EVANDRO CATELANI FERRAZ

**Optimization Models for Majority Logic Synthesis**



Ilha Solteira  
2022

EVANDRO CATELANI FERRAZ

**Optimization Models for Majority Logic Synthesis**

Thesis presented to the Faculdade de Engenharia – UNESP – Campus de Ilha Solteira as part of the necessary requirements to obtain the PhD title in Electrical Engineering. Knowledge Area: Automation.

Prof. Dr. Alexandre César Rodrigues da Silva  
Supervisor

Ilha Solteira  
2022

FICHA CATALOGRÁFICA

Desenvolvido pelo Serviço Técnico de Biblioteca e Documentação

F381o Ferraz, Evandro Catelani.  
Optimization models for majority logic synthesis / Evandro Catelani Ferraz. -  
- Ilha Solteira: [s.n.], 2022  
76 f. : il.

Tese (doutorado) - Universidade Estadual Paulista. Faculdade de Engenharia  
de Ilha Solteira. Área de conhecimento: Automação , 2022

Orientador: Alexandre César Rodrigues da Silva  
Inclui bibliografia

1. Lógica majoritária. 2. Otimização linear. 3. Solucionadores de otimização .

  
Raiane da Silva Santos

Supervisora Técnica de Seção  
Seção Técnica de Referência, Atendimento ao usuário e Documentação  
Diretoria Técnica de Biblioteca e Documentação  
CRB/8 - 9999

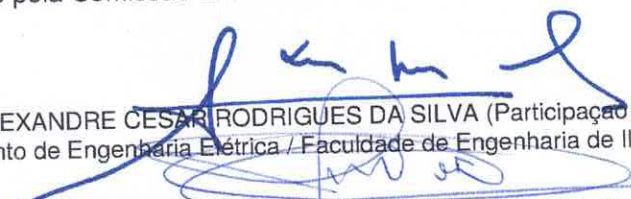
**CERTIFICADO DE APROVAÇÃO**

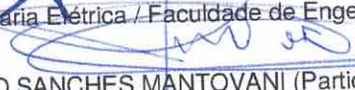
TÍTULO DA TESE: Optimizaton Models for Majority Logic Synthesis

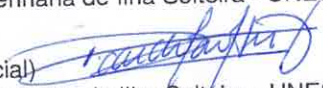
**AUTOR: EVANDRO CATELANI FERRAZ**

**ORIENTADOR: ALEXANDRE CESAR RODRIGUES DA SILVA**

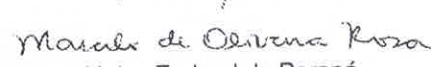
Aprovado como parte das exigências para obtenção do Título de Doutor em Engenharia Elétrica, área:  
Automação pela Comissão Examinadora:

  
Prof. Dr. ALEXANDRE CESAR RODRIGUES DA SILVA (Participação Presencial)  
Departamento de Engenharia Elétrica / Faculdade de Engenharia de Ilha Solteira - UNESP

  
Prof. Dr. JOSE ROBERTO SANCHES MANTOVANI (Participação Presencial)  
Departamento de Engenharia Elétrica / Faculdade de Engenharia de Ilha Solteira - UNESP

  
Prof. Dr. RICARDO TOKIO HIGUTI (Participação Presencial)  
Departamento de Engenharia Eletrica / Faculdade de Engenharia de Ilha Solteira - UNESP

  
Prof. Dr. MANOEL GOMES DE MENDONÇA NETO (Participação Virtual)  
Departamento de Ciência da Computação / Universidade Federal da Bahia

  
Prof. Dr. MARCELO DE OLIVEIRA ROSA (Participação Virtual)  
Departamento Acadêmico de Eletrotécnica (DAELT) / Universidade Tecnológica Federal do Paraná

Ilha Solteira, 21 de novembro de 2022

## **ACKNOWLEDGMENTS**

I thank my family, Marcos, Elis and Ana for all their support.

I thank my brother, Murilo, for his empathy and companionship.

I thank my partner, Aline, for all the words of encouragement, company, and moments of joy.

I thank my supervisor, Prof. Alexandre, for all the teachings and advice during this 6 years working together.

I thank all those who, directly or indirectly, contributed to the development of this work.

I am grateful for the support of the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Financing Code 001.

## RESUMO

Neste trabalho apresenta-se o algoritmo 3MS (Majority Math Model Solver), utilizado para síntese lógica de funções majoritárias com operadores de 3 ou 5 entradas. O novo algoritmo proposto recebe um vetor de números binários como entrada, representando a saída de uma tabela verdade, e retorna uma função majoritária otimizada para sua cobertura. A característica principal desta abordagem é a formulação de restrições que codificam problemas de lógica majoritária em problemas de otimização linear. O conjunto de restrições é então aplicado a um solucionador de otimização e os resultados transcritos em uma função majoritária de saída. No algoritmo 3MS, considera-se tanto o número de níveis quanto de operadores como primeiro ou segundo critérios de custo, possibilitando a escolha de qual destes critérios será priorizado. Como terceiro e quarto critérios de custo, considera-se a minimização de inversores e de literais. O desempenho do algoritmo 3MS foi avaliado a partir de uma comparação com 2 algoritmos de síntese exata para funções majoritárias com operadores de 3 e 5 entradas. Em ambas as sínteses, considera-se apenas o número de níveis e de operadores como critérios de custo. Tendo em vista que no algoritmo 3MS considera-se 2 critérios de custo adicionais, tem-se como objetivo gerar funções que também sejam exatas em relação ao número de níveis e operadores, porém possuam menos inversores e literais. Quando comparado a ambos algoritmos de síntese exata, testes mostraram que o algoritmo 3MS gerou melhores resultados para 64% de todas as 220.376 funções testadas, enquanto atingiu resultados equivalentes para as 36% restantes. O algoritmo 3MS também foi avaliado a partir de uma comparação com o *benchmark* EPFL (École Polytechnique Fédérale de Lausanne), sendo capaz de gerar resultados competitivos para todos os 10 circuitos que compõem o *benchmark*.

**Palavras-chave:** lógica majoritária; funções primitivas; síntese lógica; solucionadores de otimização; otimização linear.

## ABSTRACT

This work presents the 3MS (Majority Math Model Solver) algorithm, used for majority-of-three and majority-of-five logic synthesis. The new proposed algorithm receives a vector of binary numbers as input, representing the output of a truth table, and returns an optimized majority function for its coverage. Key in this approach is the formulation of constraints that encodes majority logic problems into linear optimization problems. The resulting set of constraints is then applied to an optimization solver and the results translated into the output majority function. The 3MS algorithm considers both depth and size as first and second cost criteria, making it possible to choose which of these criteria will be prioritized. As third and fourth cost criteria, the minimization of inverters and literals is considered. The 3MS algorithm was evaluated based on a comparison with 2 exact synthesis algorithms for both majority-of-three and majority-of-five networks. Both synthesis have only depth and size as cost criteria. Since the 3MS considers 2 additional cost criteria, the goal of the algorithm is to generate functions that are also exact in relation to depth and size, but with less inverters and literals. When compared to both algorithms, simulation studies have shown that the 3MS was able to further improve 64% of all 220,376 compared functions, and achieved equal results for the remaining 36%. The 3MS algorithm was also evaluated based on the EPFL (École Polytechnique Fédérale de Lausanne) benchmark suites, where the algorithm was able to generate competitive results for all 10 circuits that composes the benchmark.

**Keywords:** majority logic; primitive functions; logic synthesis; optimization solver; linear optimization.

## LIST OF FIGURES

Figure 1 – Flowchart for the overall 3MS algorithm.	28
Figure 2 – Example of QCA cells and wire.	29
Figure 3 – QCA layout for $Z = M(A, B, C)$ and $Z = M(A, B, C, D, E)$	29
Figure 4 – QCA layout for basic inverter and $Z = \overline{M}(A, B, C)$	30
Figure 5 – QCA layout for $Z = \overline{M}(1, \overline{M}(A, \overline{D}, 0), \overline{M}(B, C, 0))$	31
Figure 6 – QCA layout for $Z = M(0, M(A, \overline{D}, 0), M(B, C, 0))$	31
Figure 7 – Flowchart for the overall 3MS-Depth synthesis.	34
Figure 8 – Flowchart for the overall 3MS-Size synthesis.	46
Figure 9 – Comparison among constant, logarithmic, linear and linearithmic orders.	50
Figure 10 – Comparison among quadratic, cubic and exponential orders.	51
Figure 11 – Example of $O(1)$ .	51
Figure 12 – Example of $O(\log(N))$ .	52
Figure 13 – Example of $O(N)$ .	52
Figure 14 – Example of $O(N \cdot \log(N))$ .	53
Figure 15 – Example of $O(N^2)$ .	53
Figure 16 – Example of $O(N^3)$ .	54
Figure 17 – Example of $O(2^N)$ .	54
Figure 18 – Cost comparison (by number of improved and equal functions) between the 3MS-Size and the exact_m5ig, for criteria S5.	62
Figure 19 – Cost comparison (by number of improved and equal functions) between the 3MS-Depth and the exact_mig, for criteria D3.	63
Figure 20 – Cost comparison (by number of improved and equal functions) between the 3MS-Size and the exact_mig, for criteria S3.	64
Figure 21 – Average runtime comparison (in seconds) for criteria D3.	67
Figure 22 – Average runtime comparison (in seconds) for criteria S3.	68
Figure 23 – Average runtime comparison (in seconds) for criteria S5.	69



## LIST OF TABLES

Table 1 – Truth table of an <i>AND</i> operation.	15
Table 2 – Truth table of an <i>OR</i> operation.	15
Table 3 – Truth table of a <i>NOT</i> operation.	16
Table 4 – Example of a majority function.	16
Table 5 – Formulation of an <i>AND</i> function using a maj-3 function.	17
Table 6 – Formulation of an <i>OR</i> function using a maj-3 function.	17
Table 7 – Formulation of an <i>AND</i> function using a maj-5 function.	17
Table 8 – Formulation of an <i>OR</i> function using a maj-5 function.	17
Table 9 – Proof of $\Omega.C$ by perfect induction.	18
Table 10 – Proof of $\Omega.A$ by perfect induction.	19
Table 11 – Proof of $\Omega.D$ by perfect induction.	20
Table 12 – Proof of $\Omega.I$ by perfect induction.	21
Table 13 – Proof of $\Omega.M$ by perfect induction.	21
Table 14 – List of primitives in set $P_1$ for $n = 3$ .	22
Table 15 – List of primitives in set $P_2$ for $n = 3$ .	23
Table 16 – List of primitives in set $P_3$ for $n = 3$ .	24
Table 17 – Complete list of maj-3 primitives for $n = 3$ .	25
Table 18 – Complete list of maj-3 primitives for $n = 4$ .	26
Table 19 – Matrix $Y_{it}$ for $n = m = 3$ .	33
Table 20 – Example of $f_t$ coverage.	37
Table 21 – Example of vector $H_t$ .	44
Table 22 – Main orders of magnitude.	50
Table 23 – Constraint complexity orders for the 2-level optimization model.	56
Table 24 – Complexity orders of $X_{ij}$ bounding and $\Omega.I$ application, for the 3-level optimization model.	57
Table 25 – Complexity for the 3-level optimization model remaining constraints.	58
Table 26 – Constraint complexity orders for the size optimization model.	59
Table 27 – Cost comparison for $n = 3$ and $n = 4$ , considering criteria D3.	61
Table 28 – Cost comparison for $n = 3$ and $n = 4$ , considering criteria S3.	61
Table 29 – Cost comparison for $n = 3$ and $n = 4$ , considering criteria S5.	61
Table 30 – Cost comparison for $n = 5$ and $n = 6$ , considering criteria D3.	62

Table 31 – Cost comparison for $n = 5$ and $n = 6$ , considering criteria S3.	63
Table 32 – Runtime comparison between Gurobi and CPLEX, for criteria D3.	65
Table 33 – Runtime comparison between Gurobi and CPLEX, for criteria S3.	65
Table 34 – Average memory usage of Gurobi and CPLEX, for $m = 3$ .	65
Table 35 – Runtime comparison between Gurobi and CPLEX, for criteria D5.	66
Table 36 – Runtime comparison between Gurobi and CPLEX, for criteria S5.	66
Table 37 – Average memory usage of Gurobi and CPLEX, for $m = 5$ .	66
Table 38 – Runtime comparison between 3MS-Depth and exact_mig, for criteria D3.	67
Table 39 – Runtime comparison between 3MS-Size and exact_mig, for criteria S3.	67
Table 40 – Average memory usage of 3MS-Depth and exact_mig, for criteria D3.	68
Table 41 – Average memory usage of 3MS-Size and exact_mig, for criteria S3.	68
Table 42 – Runtime comparison between 3MS-Size and exact_m5ig, for criteria S5.	69
Table 43 – Average memory usage of 3MS-Size and exact_m5ig, for criteria S5.	69
Table 44 – EPFL benchmarks for Size optimization.	70
Table 45 – EPFL benchmarks for Depth optimization.	70

## **LIST OF ABBREVIATIONS**

API	Application Programming Interface
CMOS	Complementary Metal-Oxide-Semiconductor
DSD	Disjoint-Support Decomposition
EPFL	École Polytechnique Fédérale de Lausanne
M5IG	Majority-of-five Inverter Graph
MIG	Majority Inverter Graph
QCA	Quantum-Dot Cellular Automata
SMT	Satisfiability Modulo Theories
XMG	XOR-Majority Graph

## CONTENTS

<b>1</b>	<b>INTRODUCTION</b>	<b>10</b>
<b>2</b>	<b>MAJORITY BOOLEAN ALGEBRA</b>	<b>14</b>
2.1	CLASSICAL BOOLEAN ALGEBRA	14
2.2	FEATURES OF MAJORITY BOOLEAN ALGEBRA	16
2.3	AXIOMATIZATION OF MAJORITY FUNCTIONS ( $\Omega$ )	18
<b>2.3.1</b>	<b>Commutativity (<math>\Omega.C</math>)</b>	<b>18</b>
<b>2.3.2</b>	<b>Associativity (<math>\Omega.A</math>)</b>	<b>18</b>
<b>2.3.3</b>	<b>Distribution (<math>\Omega.D</math>)</b>	<b>19</b>
<b>2.3.4</b>	<b>Inverter Propagation (<math>\Omega.I</math>)</b>	<b>19</b>
<b>2.3.5</b>	<b>Majority (<math>\Omega.M</math>)</b>	<b>20</b>
2.4	MAJORITY PRIMITIVE FUNCTIONS	21
<b>3</b>	<b>3MS ALGORITHM</b>	<b>27</b>
3.1	APPLICABILITY OF THE 3MS ALGORITHM	28
3.2	3MS PRE-SYNTHESIS	31
3.3	3MS-DEPTH SYNTHESIS	33
<b>3.3.1</b>	<b>Optimization model for 2-level majority functions</b>	<b>34</b>
<b>3.3.2</b>	<b>Optimization model for 3-level majority functions</b>	<b>38</b>
<b>3.3.3</b>	<b>Heuristic for majority functions with 4 or more levels</b>	<b>42</b>
3.4	3MS-SIZE SYNTHESIS	45
<b>3.4.1</b>	<b>Size optimization model</b>	<b>45</b>
<b>4</b>	<b>COMPLEXITY OF ALGORITHMS</b>	<b>49</b>
4.1	COMPLEXITY OF THE 3MS ALGORITHM	54
<b>4.1.1</b>	<b>Complexity of the 2-level optimization model</b>	<b>55</b>
<b>4.1.2</b>	<b>Complexity of the 3-level optimization model</b>	<b>56</b>
<b>4.1.3</b>	<b>Complexity of the size optimization model</b>	<b>58</b>

<b>5</b>	<b>TESTS AND RESULTS</b>	<b>60</b>
<b>6</b>	<b>CONCLUSION</b>	<b>71</b>
	<b>REFERENCES</b>	<b>72</b>
	<b>APPENDIX A -- THE 3MS ALGORITHM CONDITIONAL</b>	
	<b>CONSTRAINTS</b>	<b>75</b>

# 1 INTRODUCTION

This chapter introduces the main algorithms and methodologies used for majority logic optimization, aiming for a literature review that points out the different approaches used by each algorithm and the importance of majority logic synthesis.

The major part of electronic circuits are currently built with CMOS (Complementary Metal-Oxide Semiconductor) technology, considered a revolutionary advance in the design of integrated circuits. However, this technology is already reaching its physical limits and the need for circuits miniaturization is increasing steadily (ZHANG *et al.*, 2019). Consequently, the study of nanotechnologies with potential to replace CMOS technology is also growing. Majority logic allows the creation of nanoelectronic circuits for several different technologies, presenting a possible solution for CMOS limitations (MISHRA *et al.*, 2021). Examples of emerging majority based technologies include QCA (Quantum-Dot Cellular Automata) (MAJEED; HUSSAIN; ALKALDY, 2022), Spin Wave Devices (CHUMAK *et al.*, 2022) and DNA Strand Displacement (WANG; YUAN; SUN, 2020).

CMOS based circuits are designed from classical boolean functions and the minimization of these functions has a direct impact on reducing hardware cost. Likewise, majority based circuits will also benefit from the minimization of its corresponding majority functions, justifying the study of majority synthesis algorithms that can generate optimized majority circuits. Among the first works that deal with the concept of majority logic, the works of Richard Lindaman (LINDAMAN, 1960) and Marius Cohn (COHN; LINDAMAN, 1961) stand out. Lindaman (1960) proposed the first theorem for applying majority logic in binary decision problems, introducing the majority operator to classical boolean algebra. The theorem, shown in Equation 1, proposes a boolean function equivalent to a majority operation.

$$M(A, B, C) = A \cdot B + A \cdot C + B \cdot C \quad (1)$$

Note that  $M(A, B, C)$  represents a 3-input majority function with the variables  $A$ ,  $B$

and  $C$ . Subsequently, Cohn and Lidaman (1961) presented a set of axioms that defines the majority algebra independently of the classical boolean algebra. The proposed set of axioms, represented by  $\Omega$ , became the basis for current majority algebra axiomatization.

With the goal to improve majority logic applicability, the authors Amaru, Gaillardon e Micheli (2014) proposed the MIG (Majority Inverter Graph). MIGs are graphic systems, in ternary tree format, used to describe majority-of-three networks. Being an efficient data structure, the use of MIGs can be justified for its low computational complexity, allowing lower runtimes and clearer analogies to study and create majority optimization algorithms. Soeken *et al.* (2017) proposed the `exact_mig` algorithm, used for majority-of-three exact synthesis. As input, the algorithm receives a truth table or a MIG and returns an optimized majority function that covers the same set of minterms. The most important characteristic of this algorithm is the proposal of an exact synthesis for majority functions. This synthesis is built from a set of constraints that shape a given problem accordingly to the definitions of the majority boolean algebra. The majority output function is generated with the application of this set to a SMT (Satisfiability Modulo Theories) solver. As cost criteria, the `exact_mig` can prioritize both the reduction of levels (depth) and gates (size) in the output function. The authors in Soeken *et al.* (2018) proposed adaptations of the exact synthesis used in the `exact_mig`, applying it to classical boolean functions. New technologies based on constraints and SMT solvers are presented and compared.

Chu *et al.* (2018) proposed a decomposition algorithm based on XOR and majority gates. The input function is transcribed into a XMG (XOR-Majority Graph), a new format also proposed by the authors as an alternative to the previously used MIG. The decomposition algorithm combines characteristics of majority algebra, Shannon expansion (SHANNON, 1949) and DSD (Disjoint-Support Decomposition) (BERTACCO; DAMIANI, 1997). Posteriorly, the authors in Chu *et al.* (2019) proposed an extension of this decomposition method for general boolean functions, an approach also based on majority operations and XOR-Majority Graphs. In Riener *et al.* (2019) the authors proposed a methodology to decompose monotone boolean functions into 3-input majority operations. This methodology decomposes binary decision diagrams for monotone functions into Majority Inverter Graphs, also using a decomposition logic based on Shannon’s expansion.

The authors Chu *et al.* (2019) proposed the `exact_m5ig`, another exact synthesis approach for majority functions. This synthesis is built to optimize majority-of-five networks, differing from the commonly used majority-of-three networks. The functions are represented by using M5IG (Majority-of-five Inverter Graph) as data structure, an extension of normal MIGs that uses 5-input majority gates instead of 3-input majority

gates. As cost criteria the `exact_m5ig` prioritizes the reduction of size, followed by depth.

Neutzling *et al.* (2020) also presented an approach that differs from majority-of-three networks. The proposed algorithm allows majority logic synthesis considering gates with an arbitrary number of inputs, being based on the relationship between majority functions and threshold functions and aiming to optimize size as cost criteria.

This work presents the 3MS (Majority Math Model Solver) algorithm, used for majority logic synthesis. The 3MS algorithm can synthesize functions with 3-input or 5-input majority gates, being able to optimize both majority-of-three and majority-of-five networks. As input, the algorithm receives a truth table represented by a vector of binary numbers. As output, the algorithm returns an optimized majority function that covers the same minterms. The output majority function is generated from a constraints set that encodes majority logic problems into linear optimization problems. The built set of constraints is then applied to an optimization solver and the results translated into the output majority function. The 3MS can be divided into 2 synthesis: 3MS-Depth and 3MS-Size, which will vary based on cost criteria. The 3MS-Depth synthesis prioritizes depth optimization, followed by size, while the 3MS-Size synthesis will prioritize size optimization, having depth as second cost criteria. Both synthesis have the minimization of inverters and literals as third and fourth cost criteria, where literals are the input variables, in their complemented form or not. Therefore, the goal of the algorithm is not to optimize circuits of a specific technology, the 3MS algorithm was built to cover a wide range of possible cost criteria prioritizations, aiming to optimize circuits of any technology that can benefit from this range.

Among all emerging technologies based on majority circuits, QCA stands out as a technology that would benefit from inverter optimization, besides depth and size optimization, the commonly used criteria (SHI; CHU, 2019). In QCA, the number of cells needed to form an inverter is at least 2, almost half the cost needed to build a majority gate, which is 5 (KO *et al.*, 2022). Therefore, even though depth and size optimization are still priority, to also consider the reduction of inverters results in a great decrease in the circuits total cost.

This work is organized as follows: Chapter 2 presents the main elements of majority algebra, including its axiomatization and the concept of primitive functions. Chapter 3 presents the 3MS algorithm, explaining how it works for each of its possible synthesis. An example of the 3MS algorithm applicability is also presented. Chapter 4 presents an study about complexity of algorithms, aiming to explain the complexity of each 3MS



optimization model. Chapter 5 presents the results obtained comparing the 3MS with the algorithms `exact_mig` and `exact_m5ig`. Tests to evaluate the 3MS algorithm to large circuits are also presented, using the EPFL (École Polytechnique Fédérale de Lausanne) arithmetic benchmarks. Lastly, Chapter 6 presents the conclusion of what was realized in the work.

## 2 MAJORITY BOOLEAN ALGEBRA

This chapter presents the theory behind majority boolean algebra, including its axiomatization and the concept of majority primitive functions. The purpose of this chapter is to introduce the reader to the theory transcribed into an optimization model in Chapter 3, which will address the algorithm proposed in this work and the logic used to define the set of constraints that makes this transcription possible. The first section presents the main concepts of classical boolean algebra, aiming to form a knowledge base to understand majority algebra, explained in the subsequent sections.

### 2.1 CLASSICAL BOOLEAN ALGEBRA

The classical boolean algebra, or simply boolean algebra, was introduced in the literature through the book "An Investigation of the Laws of Thought", published in 1854 and written by George Boole. In his work, Boole presents a new type of algebra that simulates logic decisions of the human thought. Logic decisions are made through a series of arguments, which can be either true or false. In boolean algebra, these arguments are represented by variables and the decisions by logic operations (BOOLE, 1854). Each boolean variable can assume 2 possible values: 0 or 1, representing logic states False and True, respectively. These logic states express the relationship between the inputs and the output of an operation, where the output value is determined by the input variables (TOCCI; WIDMER; MOSS, 2017).

The 3 basic operations in boolean algebra are *AND*, *OR* and *NOT* (FLOYD, 2015). Operation *AND*, also known as conjunction, is equivalent to an operation of multiplication and can be represented by the symbols  $\cdot$  or  $\wedge$ . Likewise, the operation *OR*, also known as disjunction, is equivalent to an addition and can be represented by the symbols  $+$  or  $\vee$ . The operation *NOT*, also known as complement or inversion, is represented by the symbol  $\neg$  and will always result in the opposite value of a given variable. A variable  $X$  can be written in its complemented state as  $\overline{X}$ . Therefore, boolean algebra is composed by the elements in set  $\{\mathbb{B}, \wedge, \vee, \neg\}$ , where  $\mathbb{B}$  represents the set of binary numbers  $\{0, 1\}$ .

Table 1 shows an example of an *AND* operation between the variables  $A$  and  $B$ , through a truth table. A truth table is the listing of every possible combination between input variables and its corresponding operation output.

**Table 1** – Truth table of an *AND* operation.

$A$	$B$	$A \cdot B$
0	0	0
0	1	0
1	0	0
1	1	1

**Source: Author.**

Note that an *AND* operation will return a True value only if all input variables are also True. If at least 1 variable is False, the output value will also be False. Table 2 shows an example of an *OR* operation between the variables  $A$  and  $B$ .

**Table 2** – Truth table of an *OR* operation.

$A$	$B$	$A + B$
0	0	0
0	1	1
1	0	1
1	1	1

**Source: Author.**

Note that an *OR* operation will return a True value if at least 1 of the input variables are True. The only False output occurs when all input variables are also False. It is important to point out that, in boolean algebra, an addition will never result in a value higher than 1. This is true because boolean variables can only assume binary values  $\{0, 1\}$ , and this statement remains true regardless the number of True values in the sum.

Table 3 shows an example of a *NOT* operation for variable  $A$ , generating the output  $\overline{A}$ . Note that the value of  $\overline{A}$  is always the opposite of  $A$ . A *NOT* operation is also known as an inverter.

**Table 3** – Truth table of a *NOT* operation.

$A$	$\overline{A}$
0	1
1	0

**Source: Author.**

## 2.2 FEATURES OF MAJORITY BOOLEAN ALGEBRA

Majority boolean algebra is composed by the set  $\{\mathbb{B}, \neg, M\}$ . Elements  $\mathbb{B}$  and  $\neg$ , as in classical boolean algebra, represent the boolean constants  $\{0, 1\}$  and the inversion operator. The remaining element of the set,  $M$ , represents the majority operator (CHATTOPADHYAY *et al.*, 2016). A majority operation returns the most present binary value among its inputs. The 3-input majority function  $M(A, B, C)$ , for instance, will return a True value if and only if at least 2 of its 3 inputs are True. If generalized to  $m$  inputs (maj- $m$ ), the majority function will return a True value if  $\lceil \frac{m}{2} \rceil$  inputs are True, where  $m$  is an odd number. Table 4 shows the truth table of  $M(A, B, C)$ , where  $m = 3$ .

**Table 4** – Example of a majority function.

$A$	$B$	$C$	$M(A, B, C)$
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

**Source: Author**

From a majority function it is also possible to formulate *AND* and *OR* functions, performed by setting  $m - 2$  inputs to constant values (RIENER *et al.*, 2019). For maj-3 functions ( $m = 3$ ), the formulation is performed by setting 1 input to a constant value. Considering  $M(A, B, C)$  as example, if the value of  $C$  is set to 0, an *AND* function between  $A$  and  $B$  is created. Similarly, setting  $C$  to 1 creates an *OR* function between  $A$  and  $B$ . Truth tables showing this formulation are presented in Tables 5 and 6, for functions *AND* and *OR*, respectively.

The formulation of *AND* and *OR* functions through a maj-5 function ( $m = 5$ ) is made with a combination of 3 constants: a pair of equal constants and a single constant

**Table 5** – Formulation of an *AND* function using a maj-3 function.

$A$	$B$	$A \cdot B$	$M(A, B, 0)$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

**Source: Author****Table 6** – Formulation of an *OR* function using a maj-3 function.

$A$	$B$	$A + B$	$M(A, B, 1)$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

**Source: Author**

of opposite value. Considering variables  $A$  and  $B$  as example, the equivalent *AND* and *OR* functions, respectively, are  $M(A, B, 0, 0, 1)$  and  $M(A, B, 1, 1, 0)$ . Truth tables showing this formulation are presented in Tables 7 and 8.

**Table 7** – Formulation of an *AND* function using a maj-5 function.

$A$	$B$	$A \cdot B$	$M(A, B, 0, 0, 1)$
0	0	0	0
0	1	0	0
1	0	0	0
1	1	1	1

**Source: Author****Table 8** – Formulation of an *OR* function using a maj-5 function.

$A$	$B$	$A + B$	$M(A, B, 1, 1, 0)$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	1

**Source: Author**

A maj-5 function can be written in its optimized maj-3 format as shown in Equation 2, considering depth as primary cost criteria. Note that a maj-3 function with 3 levels and 4 gates is needed to simulate the logic of a maj-5 function, which justifies the study of maj-5 logic synthesis when considering depth and size optimization.

$$M(A, B, C, D, E) = M(E, M(B, C, D), M(M(C, D, E), A, B)) \quad (2)$$

### 2.3 AXIOMATIZATION OF MAJORITY FUNCTIONS ( $\Omega$ )

The set of axioms that defines the majority algebra is represented by  $\Omega$  and can be divided into axioms of Commutativity, Associativity, Distribution, Inverter Propagation and Majority. Every axiom in  $\Omega$  can be proved by perfect induction, with the listing of every possible result for each term of a truth table (CHATTOPADHYAY *et al.*, 2016).

#### 2.3.1 Commutativity ( $\Omega.C$ )

The Commutativity axiom ( $\Omega.C$ ), represented in Equation 3, determines that the input order does not change the output value. Table 9 proves  $\Omega.C$  by perfect induction.

$$M(A, B, C) = M(A, C, B) = M(C, B, A) \quad (3)$$

**Table 9** – Proof of  $\Omega.C$  by perfect induction.

$A$	$B$	$C$	$M(A, B, C)$	$M(A, C, B)$	$M(C, B, A)$
0	0	0	$M(0, 0, 0) = 0$	$M(0, 0, 0) = 0$	$M(0, 0, 0) = 0$
0	0	1	$M(0, 0, 1) = 0$	$M(0, 1, 0) = 0$	$M(1, 0, 0) = 0$
0	1	0	$M(0, 1, 0) = 0$	$M(0, 0, 1) = 0$	$M(0, 1, 0) = 0$
0	1	1	$M(0, 1, 1) = 1$	$M(0, 1, 1) = 1$	$M(1, 1, 0) = 1$
1	0	0	$M(1, 0, 0) = 0$	$M(1, 0, 0) = 0$	$M(0, 0, 1) = 0$
1	0	1	$M(1, 0, 1) = 1$	$M(1, 1, 0) = 1$	$M(1, 0, 1) = 1$
1	1	0	$M(1, 1, 0) = 1$	$M(1, 0, 1) = 1$	$M(0, 1, 1) = 1$
1	1	1	$M(1, 1, 1) = 1$	$M(1, 1, 1) = 1$	$M(1, 1, 1) = 1$

Source: Author

#### 2.3.2 Associativity ( $\Omega.A$ )

The Associativity axiom  $\Omega.A$  states that the exchange of variables between two functions is possible, as long as they are at subsequent levels and have one variable in common. An example of an  $\Omega.A$  application is presented in Equation 4.

$$M(A, D, M(B, D, C)) = M(C, D, M(B, D, A)) \quad (4)$$

Note that the variable shared between levels is  $D$ . Therefore, it is possible to substitute the remaining variable in the upper level for one in the subsequent level. In the presented example, we had an exchange between the variables  $A$  and  $C$ . Table 10 proves  $\Omega.A$  by perfect induction.

**Table 10** – Proof of  $\Omega.A$  by perfect induction.

$A$	$B$	$C$	$D$	$M(A, D, M(B, D, C))$	$M(C, D, M(B, D, A))$
0	0	0	0	$M(0, 0, M(0, 0, 0)) = 0$	$M(0, 0, M(0, 0, 0)) = 0$
0	0	0	1	$M(0, 1, M(0, 1, 0)) = 0$	$M(0, 1, M(0, 1, 0)) = 0$
0	0	1	0	$M(0, 0, M(0, 0, 1)) = 0$	$M(1, 0, M(0, 0, 0)) = 0$
0	0	1	1	$M(0, 1, M(0, 1, 1)) = 1$	$M(1, 1, M(0, 1, 0)) = 1$
0	1	0	0	$M(0, 0, M(1, 0, 0)) = 0$	$M(0, 0, M(1, 0, 0)) = 0$
0	1	0	1	$M(0, 1, M(1, 1, 0)) = 1$	$M(0, 1, M(1, 1, 0)) = 1$
0	1	1	0	$M(0, 0, M(1, 0, 1)) = 0$	$M(1, 0, M(1, 0, 0)) = 0$
0	1	1	1	$M(0, 1, M(1, 1, 1)) = 1$	$M(1, 1, M(1, 1, 0)) = 1$
1	0	0	0	$M(1, 0, M(0, 0, 0)) = 0$	$M(0, 0, M(0, 0, 1)) = 0$
1	0	0	1	$M(1, 1, M(0, 1, 0)) = 1$	$M(0, 1, M(0, 1, 1)) = 1$
1	0	1	0	$M(1, 0, M(0, 0, 1)) = 0$	$M(1, 0, M(0, 0, 1)) = 0$
1	0	1	1	$M(1, 1, M(0, 1, 1)) = 1$	$M(1, 1, M(0, 1, 1)) = 1$
1	1	0	0	$M(1, 0, M(1, 0, 0)) = 0$	$M(0, 0, M(1, 0, 1)) = 0$
1	1	0	1	$M(1, 1, M(1, 1, 0)) = 1$	$M(0, 1, M(1, 1, 1)) = 1$
1	1	1	0	$M(1, 0, M(1, 0, 1)) = 1$	$M(1, 0, M(1, 0, 1)) = 1$
1	1	1	1	$M(1, 1, M(1, 1, 1)) = 1$	$M(1, 1, M(1, 1, 1)) = 1$

Source: Author

### 2.3.3 Distribution ( $\Omega.D$ )

The Distribution axiom ( $\Omega.D$ ) determines that it is possible to distribute a set of variables to gates in subsequent levels. In Equation 5 an example of this theorem is given, where the distributed set is  $\{A, B\}$ . Table 11 proves  $\Omega.D$  by perfect induction.

$$M(A, B, M(D, E, C)) = M(M(A, B, D), M(A, B, E), M(A, B, C)) \quad (5)$$

### 2.3.4 Inverter Propagation ( $\Omega.I$ )

The Inverter Propagation axiom ( $\Omega.I$ ), represented in Equation 6, determines that a majority function is self-dual, meaning that a majority function is always equivalent to its dual form. A function's dual form can be obtained by complementing all input variables and gates (SASAO, 2012). Note that  $M(A, B, C)$  and  $\overline{M}(\overline{A}, \overline{B}, \overline{C})$  are the dual form of

**Table 11** – Proof of  $\Omega.D$  by perfect induction.

$A$	$B$	$C$	$D$	$E$	$M(A, B, M(D, E, C))$	$M(M(A, B, D), M(A, B, E), M(A, B, C))$
0	0	0	0	0	$M(0, 0, M(0, 0, 0)) = 0$	$M(M(0, 0, 0), M(0, 0, 0), M(0, 0, 0)) = 0$
0	0	0	0	1	$M(0, 0, M(0, 1, 0)) = 0$	$M(M(0, 0, 0), M(0, 0, 1), M(0, 0, 0)) = 0$
0	0	0	1	0	$M(0, 0, M(1, 0, 0)) = 0$	$M(M(0, 0, 1), M(0, 0, 0), M(0, 0, 0)) = 0$
0	0	0	1	1	$M(0, 0, M(1, 1, 0)) = 0$	$M(M(0, 0, 1), M(0, 0, 1), M(0, 0, 0)) = 0$
0	0	1	0	0	$M(0, 0, M(0, 0, 1)) = 0$	$M(M(0, 0, 0), M(0, 0, 0), M(0, 0, 1)) = 0$
0	0	1	0	1	$M(0, 0, M(0, 1, 1)) = 0$	$M(M(0, 0, 0), M(0, 0, 1), M(0, 0, 1)) = 0$
0	0	1	1	0	$M(0, 0, M(1, 0, 1)) = 0$	$M(M(0, 0, 1), M(0, 0, 0), M(0, 0, 1)) = 0$
0	0	1	1	1	$M(0, 0, M(1, 1, 1)) = 0$	$M(M(0, 0, 1), M(0, 0, 1), M(0, 0, 1)) = 0$
0	1	0	0	0	$M(0, 1, M(0, 0, 0)) = 0$	$M(M(0, 1, 0), M(0, 1, 0), M(0, 1, 0)) = 0$
0	1	0	0	1	$M(0, 1, M(0, 1, 0)) = 0$	$M(M(0, 1, 0), M(0, 1, 1), M(0, 1, 0)) = 0$
0	1	0	1	0	$M(0, 1, M(1, 0, 0)) = 0$	$M(M(0, 1, 1), M(0, 1, 0), M(0, 0, 0)) = 0$
0	1	0	1	1	$M(0, 1, M(1, 1, 0)) = 1$	$M(M(0, 1, 1), M(0, 1, 1), M(0, 1, 0)) = 1$
0	1	1	0	0	$M(0, 1, M(0, 0, 1)) = 0$	$M(M(0, 1, 0), M(0, 1, 0), M(0, 1, 1)) = 0$
0	1	1	0	1	$M(0, 1, M(0, 1, 1)) = 1$	$M(M(0, 1, 0), M(0, 1, 1), M(0, 1, 1)) = 1$
0	1	1	1	0	$M(0, 1, M(1, 0, 1)) = 1$	$M(M(0, 1, 1), M(0, 1, 0), M(0, 1, 1)) = 1$
0	1	1	1	1	$M(0, 1, M(1, 1, 1)) = 1$	$M(M(0, 1, 1), M(0, 1, 1), M(0, 1, 1)) = 1$
1	0	0	0	0	$M(1, 0, M(0, 0, 0)) = 0$	$M(M(1, 0, 0), M(1, 0, 0), M(1, 0, 0)) = 0$
1	0	0	0	1	$M(1, 0, M(0, 1, 0)) = 0$	$M(M(1, 0, 0), M(1, 0, 1), M(1, 0, 0)) = 0$
1	0	0	1	0	$M(1, 0, M(1, 0, 0)) = 0$	$M(M(1, 0, 1), M(1, 0, 0), M(1, 0, 0)) = 0$
1	0	0	1	1	$M(1, 0, M(1, 1, 0)) = 1$	$M(M(1, 0, 1), M(1, 0, 1), M(1, 0, 0)) = 1$
1	0	1	0	0	$M(1, 0, M(0, 0, 1)) = 0$	$M(M(1, 0, 0), M(1, 0, 0), M(1, 0, 1)) = 0$
1	0	1	0	1	$M(1, 0, M(0, 1, 1)) = 1$	$M(M(1, 0, 0), M(1, 0, 1), M(1, 0, 1)) = 1$
1	0	1	1	0	$M(1, 0, M(1, 0, 1)) = 1$	$M(M(1, 0, 1), M(1, 0, 0), M(1, 0, 1)) = 1$
1	0	1	1	1	$M(1, 0, M(1, 1, 1)) = 1$	$M(M(1, 0, 1), M(1, 0, 1), M(1, 0, 1)) = 1$
1	1	0	0	0	$M(1, 1, M(0, 0, 0)) = 1$	$M(M(1, 1, 0), M(1, 1, 0), M(1, 1, 0)) = 1$
1	1	0	0	1	$M(1, 1, M(0, 1, 0)) = 1$	$M(M(1, 1, 0), M(1, 1, 1), M(1, 1, 0)) = 1$
1	1	0	1	0	$M(1, 1, M(1, 0, 0)) = 1$	$M(M(1, 1, 1), M(1, 1, 0), M(1, 1, 0)) = 1$
1	1	0	1	1	$M(1, 1, M(1, 1, 0)) = 1$	$M(M(1, 1, 1), M(1, 1, 1), M(1, 1, 0)) = 1$
1	1	1	0	0	$M(1, 1, M(0, 0, 1)) = 1$	$M(M(1, 1, 0), M(1, 1, 0), M(1, 1, 1)) = 1$
1	1	1	0	1	$M(1, 1, M(0, 1, 1)) = 1$	$M(M(1, 1, 0), M(1, 1, 1), M(1, 1, 1)) = 1$
1	1	1	1	0	$M(1, 1, M(1, 0, 1)) = 1$	$M(M(1, 1, 1), M(1, 1, 0), M(1, 1, 1)) = 1$
1	1	1	1	1	$M(1, 1, M(1, 1, 1)) = 1$	$M(M(1, 1, 1), M(1, 1, 1), M(1, 1, 1)) = 1$

**Source: Author**

each other. Table 12 shows the equivalence between the majority functions, proving  $\Omega.I$  by perfect induction.

$$M(A, B, C) = \overline{M}(\overline{A}, \overline{B}, \overline{C}) \quad (6)$$

### 2.3.5 Majority ( $\Omega.M$ )

The Majority axiom ( $\Omega.M$ ) can be divided into 2 equations. Equation 7 shows that the output of a majority gate is equal to the most common value among its inputs. Equation



**Table 12** – Proof of  $\Omega.I$  by perfect induction.

$A$	$B$	$C$	$M(A, B, C)$	$\overline{M}(\overline{A}, \overline{B}, \overline{C})$
0	0	0	$M(0, 0, 0) = 0$	$\overline{M}(1, 1, 1) = 0$
0	0	1	$M(0, 0, 1) = 0$	$\overline{M}(1, 1, 0) = 0$
0	1	0	$M(0, 1, 0) = 0$	$\overline{M}(1, 0, 1) = 0$
0	1	1	$M(0, 1, 1) = 1$	$\overline{M}(1, 0, 0) = 1$
1	0	0	$M(1, 0, 0) = 0$	$\overline{M}(0, 1, 1) = 0$
1	0	1	$M(1, 0, 1) = 1$	$\overline{M}(0, 1, 0) = 1$
1	1	0	$M(1, 1, 0) = 1$	$\overline{M}(0, 0, 1) = 1$
1	1	1	$M(1, 1, 1) = 1$	$\overline{M}(0, 0, 0) = 1$

**Source: Author**

8 shows that the output value will be equal to the tie-breaking variable in functions with the same number of True and False values. Table 13 proves  $\Omega.M$  by perfect induction.

$$M(A, A, B) = A \quad (7)$$

$$M(A, \overline{A}, B) = B \quad (8)$$

**Table 13** – Proof of  $\Omega.M$  by perfect induction.

$A$	$B$	$M(A, A, B) = A$	$M(A, \overline{A}, B) = B$
0	0	$M(0, 0, 0) = 0$	$M(0, 1, 0) = 0$
0	1	$M(0, 0, 1) = 0$	$M(0, 1, 1) = 1$
1	0	$M(1, 1, 0) = 1$	$M(1, 0, 0) = 0$
1	1	$M(1, 1, 1) = 1$	$M(1, 0, 1) = 1$

**Source: Author**

## 2.4 MAJORITY PRIMITIVE FUNCTIONS

Majority primitive functions (also known as primitives) are functions formed by at most one majority gate, often used in the formulation of more complex functions (WANG *et al.*, 2015). The complete list of primitives can be obtained from the sets  $P_q$ , where  $0 \leq q \leq m$ . Each set  $P_q$  comprises functions with  $q$  literals, formed by maj- $m$  gates.

Set  $P_0$  comprises the constants 0 and 1 and, therefore, the number of functions in  $P_0$

will always be 2. Set  $P_1$  comprises functions with a single literal, being simply the list of input variables, in its complemented form or not. The number of input variables is represented by  $n$  and the number of functions in  $P_1$  can be calculated by Equation 9.

$$|P_1| = 2 \cdot n \quad (9)$$

Table 14 shows the list of functions in set  $P_1$ , considering  $n = 3$ . Note that both classical and majority forms are the same, since set  $P_1$  is composed only by functions with a single literal and no gates.

**Table 14** – List of primitives in set  $P_1$  for  $n = 3$ .

Classical Function	Majority Function
$A$	$A$
$B$	$B$
$C$	$C$
$\overline{A}$	$\overline{A}$
$\overline{B}$	$\overline{B}$
$\overline{C}$	$\overline{C}$

**Source: Author**

Set  $P_2$  is formed by maj- $m$  gates with 2 literals and  $m - 2$  constant values, comprising the formulation of *AND* and *OR* gates. The number of functions in  $P_2$  can be calculated by the number of possible 2-to-2 combinations among input variables, without repeated pairs, multiplied by 8. This calculation is shown in Equation 10.

$$|P_2| = \frac{n!}{2! \cdot (n-2)!} \cdot 8 \quad (10)$$

The multiplication by 8 refers to the number of possible inversion variations, which is always 4 for each *AND* and *OR* simulations. Considering  $n = 3$ , the possible 2-to-2 combinations are  $AB$ ,  $AC$  and  $BC$ . Combination  $AB$  and constant 0, for instance, generates the maj-3 function  $M(A, B, 0)$  with the possible inversion variations:  $M(A, B, 0)$ ,  $M(\overline{A}, B, 0)$ ,  $M(A, \overline{B}, 0)$  and  $M(\overline{A}, \overline{B}, 0)$ . Table 15 shows the list of functions in set  $P_2$  for  $n = 3$ , considering both  $m = 3$  and  $m = 5$ . Note that for  $m = 5$  the formulation of *AND* and *OR* functions is done with a combination of 3 constants ( $m - 2 = 3$ ).

Set  $P_3$  is formed by majority gates with 3 literals, also without considering repeated pairs. The number of functions in  $P_3$  can be calculated by the number of possible 3-to-3

**Table 15** – List of primitives in set  $P_2$  for  $n = 3$ .

Classical Function	Maj-3 Function	Maj-5 Function
$A \cdot B$	$M(A, B, 0)$	$M(A, B, 0, 0, 1)$
$\overline{A} \cdot B$	$M(\overline{A}, B, 0)$	$M(\overline{A}, B, 0, 0, 1)$
$A \cdot \overline{B}$	$M(A, \overline{B}, 0)$	$M(A, \overline{B}, 0, 0, 1)$
$\overline{A} \cdot \overline{B}$	$\overline{M}(A, B, 1)$	$\overline{M}(A, B, 1, 1, 0)$
$A \cdot C$	$M(A, C, 0)$	$M(A, C, 0, 0, 1)$
$\overline{A} \cdot C$	$M(\overline{A}, C, 0)$	$M(\overline{A}, C, 0, 0, 1)$
$A \cdot \overline{C}$	$M(A, \overline{C}, 0)$	$M(A, \overline{C}, 0, 0, 1)$
$\overline{A} \cdot \overline{C}$	$\overline{M}(A, C, 1)$	$\overline{M}(A, C, 1, 1, 0)$
$B \cdot C$	$M(B, C, 0)$	$M(B, C, 0, 0, 1)$
$\overline{B} \cdot C$	$M(\overline{B}, C, 0)$	$M(\overline{B}, C, 0, 0, 1)$
$B \cdot \overline{C}$	$M(B, \overline{C}, 0)$	$M(B, \overline{C}, 0, 0, 1)$
$\overline{B} \cdot \overline{C}$	$\overline{M}(B, C, 1)$	$\overline{M}(B, C, 1, 1, 0)$
$A + B$	$M(A, B, 1)$	$M(A, B, 1, 1, 0)$
$\overline{A} + B$	$M(\overline{A}, B, 1)$	$M(\overline{A}, B, 1, 1, 0)$
$A + \overline{B}$	$M(A, \overline{B}, 1)$	$M(A, \overline{B}, 1, 1, 0)$
$\overline{A} + \overline{B}$	$\overline{M}(A, B, 0)$	$\overline{M}(A, B, 0, 0, 1)$
$A + C$	$M(A, C, 1)$	$M(A, C, 1, 1, 0)$
$\overline{A} + C$	$M(\overline{A}, C, 1)$	$M(\overline{A}, C, 1, 1, 0)$
$A + \overline{C}$	$M(A, \overline{C}, 1)$	$M(A, \overline{C}, 1, 1, 0)$
$\overline{A} + \overline{C}$	$\overline{M}(A, C, 0)$	$\overline{M}(A, C, 0, 0, 1)$
$B + C$	$M(B, C, 1)$	$M(B, C, 1, 1, 0)$
$\overline{B} + C$	$M(\overline{B}, C, 1)$	$M(\overline{B}, C, 1, 1, 0)$
$B + \overline{C}$	$M(B, \overline{C}, 1)$	$M(B, \overline{C}, 1, 1, 0)$
$\overline{B} + \overline{C}$	$\overline{M}(B, C, 0)$	$\overline{M}(B, C, 0, 0, 1)$

**Source: Author**

combinations among inputs variables multiplied by 8, where 8 refers to the number of possible inversion variations for each 3-to-3 combination. This calculation is shown by Equation 11. For  $m = 5$ , the combination of constants 01 is considered as input aside from the 3 literals. Table 16 shows the list of functions in set  $P_3$  for  $n = 3$ . Note that the only possible 3-to-3 combination is  $ABC$ .

$$|P_3| = \frac{n!}{3! \cdot (n-3)!} \cdot 8 \quad (11)$$

The total number of primitives, represented by  $p$ , is given by the sum of the elements in all sets, as shown in Equation 12.

**Table 16** – List of primitives in set  $P_3$  for  $n = 3$ .

Classical Function	Maj-3 Function	Maj-5 Function
$A \cdot B + A \cdot C + B \cdot C$	$M(A, B, C)$	$M(A, B, C, 0, 1)$
$\overline{A} \cdot \overline{B} + \overline{A} \cdot \overline{C} + \overline{B} \cdot \overline{C}$	$\overline{M}(A, B, C)$	$\overline{M}(A, B, C, 0, 1)$
$\overline{A} \cdot B + \overline{A} \cdot C + B \cdot C$	$M(\overline{A}, B, C)$	$M(\overline{A}, B, C, 0, 1)$
$A \cdot \overline{B} + A \cdot \overline{C} + \overline{B} \cdot \overline{C}$	$\overline{M}(\overline{A}, B, C)$	$\overline{M}(\overline{A}, B, C, 0, 1)$
$A \cdot \overline{B} + A \cdot C + \overline{B} \cdot C$	$M(A, \overline{B}, C)$	$M(A, \overline{B}, C, 0, 1)$
$\overline{A} \cdot B + \overline{A} \cdot \overline{C} + B \cdot \overline{C}$	$\overline{M}(A, \overline{B}, C)$	$\overline{M}(A, \overline{B}, C, 0, 1)$
$A \cdot B + A \cdot \overline{C} + B \cdot \overline{C}$	$M(A, B, \overline{C})$	$M(A, B, \overline{C}, 0, 1)$
$\overline{A} \cdot \overline{B} + \overline{A} \cdot C + \overline{B} \cdot C$	$\overline{M}(A, B, \overline{C})$	$\overline{M}(A, B, \overline{C}, 0, 1)$

**Source: Author**

$$p = \sum_{q=0}^m P_q \quad (12)$$

Considering  $m = 3$ , the maximum value of  $q$  is 3. Therefore, for  $n = 3$ ,  $|P_0| + |P_1| + |P_2| + |P_3| = 40$ , and for  $n = 4$ ,  $|P_0| + |P_1| + |P_2| + |P_3| = 90$ . Tables 17 and 18 shows, respectively, the complete list of primitive functions for  $n = 3$  and  $n = 4$ , considering  $m = 3$ .

For  $m = 5$  the maximum value of  $q$  is 5, resulting in the additional sets  $P_4$  and  $P_5$ . Set  $P_4$  is formed by majority functions with 4 literals and a single constant. Set  $P_5$  is formed by majority functions with 5 literals. Both sets  $P_4$  and  $P_5$  allows the repetition of literals as long as this repetition is a pair of the same literal with the same complement. An example of a valid pair would be the combination  $AABC$ , while  $\overline{A}ABC$  exemplifies an invalid combination. The majority function  $M(\overline{A}, A, B, C, 0)$ , formed based on the invalid combination  $\overline{A}ABC$ , is disregarded because it can be simplified (based on the  $\Omega.M$  axiom) to  $M(B, C, 0, 0, 1)$ , a lower cost primitive when considering the number of literals as cost criteria. Equation 13 shows how to calculate the number of elements in  $P_4$ .

$$|P_4| = \left( \frac{n!}{3! \cdot (n-3)!} \cdot 3 \right) \cdot 16 + \frac{n!}{4! \cdot (n-4)!} \cdot 24 \quad (13)$$

The calculation  $\frac{n!}{3! \cdot (n-3)!} \cdot 3$  comprises functions formed by 2 literals and a pair of a different literal, totalizing 4 inputs. The multiplication by 16 refers to 8 possible inversion variations for each constant 0 and 1. The rest of the equation calculates the number of 4-to-4 combinations among input variables, multiplied by 24 and meaning 12 possible inversion variations for each constant. For  $n = 3$ , only the elements in  $\left( \frac{n!}{3! \cdot (n-3)!} \cdot 3 \right) \cdot 16$

**Table 17** – Complete list of maj-3 primitives for  $n = 3$ .

N <sup>o</sup>	Classical Function	Majority Function	N <sup>o</sup>	Classical Function	Majority Function
0	0	0	20	$A + B$	$M(A, B, 1)$
1	1	1	21	$\overline{A} + B$	$M(\overline{A}, B, 1)$
2	$A$	$A$	22	$A + \overline{B}$	$M(A, \overline{B}, 1)$
3	$B$	$B$	23	$\overline{A} + \overline{B}$	$\overline{M}(A, B, 0)$
4	$C$	$C$	24	$A + C$	$M(A, 0, C)$
5	$\overline{A}$	$\overline{A}$	25	$\overline{A} + C$	$M(\overline{A}, 1, C)$
6	$\overline{B}$	$\overline{B}$	26	$A + \overline{C}$	$M(A, 1, \overline{C})$
7	$\overline{C}$	$\overline{C}$	27	$\overline{A} + \overline{C}$	$\overline{M}(A, 0, C)$
8	$A \cdot B$	$M(A, B, 0)$	28	$B + C$	$M(1, B, C)$
9	$\overline{A} \cdot B$	$M(\overline{A}, B, 0)$	29	$\overline{B} + C$	$M(1, \overline{B}, C)$
10	$A \cdot \overline{B}$	$M(A, \overline{B}, 0)$	30	$B + \overline{C}$	$M(1, B, \overline{C})$
11	$\overline{A} \cdot \overline{B}$	$\overline{M}(A, B, 1)$	31	$\overline{B} + \overline{C}$	$\overline{M}(0, B, C)$
12	$A \cdot C$	$M(A, 0, C)$	32	$A \cdot B + A \cdot C + B \cdot C$	$M(A, B, C)$
13	$\overline{A} \cdot C$	$M(\overline{A}, 0, C)$	33	$\overline{A} \cdot \overline{B} + \overline{A} \cdot \overline{C} + \overline{B} \cdot \overline{C}$	$\overline{M}(A, B, C)$
14	$A \cdot \overline{C}$	$M(A, 0, \overline{C})$	34	$\overline{A} \cdot B + \overline{A} \cdot C + B \cdot C$	$M(\overline{A}, B, C)$
15	$\overline{A} \cdot \overline{C}$	$\overline{M}(A, 1, C)$	35	$A \cdot \overline{B} + A \cdot \overline{C} + \overline{B} \cdot \overline{C}$	$\overline{M}(\overline{A}, B, C)$
16	$B \cdot C$	$M(0, B, C)$	36	$A \cdot \overline{B} + A \cdot C + \overline{B} \cdot C$	$M(A, \overline{B}, C)$
17	$\overline{B} \cdot C$	$M(0, \overline{B}, C)$	37	$\overline{A} \cdot B + \overline{A} \cdot \overline{C} + B \cdot \overline{C}$	$\overline{M}(A, \overline{B}, C)$
18	$B \cdot \overline{C}$	$M(0, B, \overline{C})$	38	$A \cdot B + A \cdot \overline{C} + B \cdot \overline{C}$	$M(A, B, \overline{C})$
19	$\overline{B} \cdot \overline{C}$	$\overline{M}(1, B, C)$	39	$\overline{A} \cdot \overline{B} + \overline{A} \cdot C + \overline{B} \cdot C$	$\overline{M}(A, B, \overline{C})$

**Source: Author**

are added to the primitives list. For  $n \geq 4$ , all elements in  $P_4$  are added to the primitives list.

The number of elements in set  $P_5$  can be calculated by Equation 14. The calculation  $\frac{n!}{4! \cdot (n-4)!} \cdot 4$  comprises functions formed by 3 literals and a pair of a different literal, whereas  $\frac{n!}{5! \cdot (n-5)!}$  calculates the number of 5-to-5 combinations among input variables.

$$|P_5| = \left( \frac{n!}{4! \cdot (n-4)!} \cdot 4 \right) \cdot 24 + \frac{n!}{5! \cdot (n-5)!} \cdot 48 \quad (14)$$

The multiplication by 24, as in set  $P_4$ , represents 12 possible inversion variations for each constant 0 and 1. The multiplication by 48 represents the possible inversion variations among the 5 different literals. For  $n = 4$ , only the elements in  $\left( \frac{n!}{4! \cdot (n-4)!} \cdot 4 \right) \cdot 24$  are added to the primitives list. For  $n \geq 5$ , all elements in  $P_5$  are added to the primitives list.

**Table 18** – Complete list of maj-3 primitives for  $n = 4$ .

N <sup>o</sup>	Classical Function	Majority Function	N <sup>o</sup>	Classical Function	Majority Function
0	0	0	45	$\bar{A} + \bar{D}$	$\bar{M}(A, 0, D)$
1	1	1	46	$B + C$	$M(B, 1, C)$
2	$A$	$A$	47	$\bar{B} + C$	$M(\bar{B}, 1, C)$
3	$B$	$B$	48	$B + \bar{C}$	$M(B, 1, \bar{C})$
4	$C$	$C$	49	$\bar{B} + \bar{C}$	$\bar{M}(B, 0, C)$
5	$D$	$D$	50	$B + D$	$M(B, 1, D)$
6	$\bar{A}$	$\bar{A}$	51	$\bar{B} + D$	$M(\bar{B}, 1, D)$
7	$\bar{B}$	$\bar{B}$	52	$B + \bar{D}$	$M(B, 1, \bar{D})$
8	$\bar{C}$	$\bar{C}$	53	$\bar{B} + \bar{D}$	$\bar{M}(B, 0, D)$
9	$\bar{D}$	$\bar{D}$	54	$D + C$	$M(1, C, D)$
10	$A \cdot B$	$M(A, B, 0)$	55	$\bar{D} + C$	$M(1, C, \bar{D})$
11	$\bar{A} \cdot B$	$M(\bar{A}, B, 0)$	56	$D + \bar{C}$	$M(1, \bar{C}, D)$
12	$A \cdot \bar{B}$	$M(A, \bar{B}, 0)$	57	$\bar{D} + \bar{C}$	$\bar{M}(0, C, D)$
13	$\bar{A} \cdot \bar{B}$	$\bar{M}(A, B, 1)$	58	$A \cdot B + A \cdot C + B \cdot C$	$M(A, B, C)$
14	$A \cdot C$	$M(A, 0, C)$	59	$\bar{A} \cdot \bar{B} + \bar{A} \cdot \bar{C} + \bar{B} \cdot \bar{C}$	$\bar{M}(A, B, C)$
15	$\bar{A} \cdot C$	$M(\bar{A}, 0, C)$	60	$\bar{A} \cdot B + \bar{A} \cdot C + B \cdot C$	$M(\bar{A}, B, C)$
16	$A \cdot \bar{C}$	$M(A, 0, \bar{C})$	61	$A \cdot \bar{B} + A \cdot \bar{C} + \bar{B} \cdot \bar{C}$	$\bar{M}(\bar{A}, B, C)$
17	$\bar{A} \cdot \bar{C}$	$\bar{M}(A, 1, C)$	62	$A \cdot \bar{B} + A \cdot C + \bar{B} \cdot C$	$M(A, \bar{B}, C)$
18	$A \cdot D$	$M(A, 0, D)$	63	$\bar{A} \cdot B + \bar{A} \cdot \bar{C} + B \cdot \bar{C}$	$\bar{M}(A, \bar{B}, C)$
19	$\bar{A} \cdot D$	$M(\bar{A}, 0, D)$	64	$A \cdot B + A \cdot \bar{C} + B \cdot \bar{C}$	$M(A, B, \bar{C})$
20	$A \cdot \bar{D}$	$M(A, 0, \bar{D})$	65	$\bar{A} \cdot \bar{B} + \bar{A} \cdot C + \bar{B} \cdot C$	$\bar{M}(A, B, \bar{C})$
21	$\bar{A} \cdot \bar{D}$	$\bar{M}(A, 1, D)$	66	$A \cdot B + A \cdot D + B \cdot D$	$M(A, B, D)$
22	$B \cdot C$	$M(0, B, C)$	67	$\bar{A} \cdot \bar{B} + \bar{A} \cdot \bar{D} + \bar{B} \cdot \bar{D}$	$\bar{M}(A, B, D)$
23	$\bar{B} \cdot C$	$M(0, \bar{B}, C)$	68	$\bar{A} \cdot B + \bar{A} \cdot D + B \cdot D$	$M(\bar{A}, B, D)$
24	$B \cdot \bar{C}$	$M(0, B, \bar{C})$	69	$A \cdot \bar{B} + A \cdot \bar{D} + \bar{B} \cdot \bar{D}$	$\bar{M}(\bar{A}, B, D)$
25	$\bar{B} \cdot \bar{C}$	$\bar{M}(1, B, C)$	70	$A \cdot \bar{B} + A \cdot D + \bar{B} \cdot D$	$M(A, \bar{B}, D)$
26	$B \cdot D$	$M(0, B, D)$	71	$\bar{A} \cdot B + \bar{A} \cdot \bar{D} + B \cdot \bar{D}$	$\bar{M}(A, \bar{B}, D)$
27	$\bar{B} \cdot D$	$M(0, \bar{B}, D)$	72	$A \cdot B + A \cdot \bar{D} + B \cdot \bar{D}$	$M(A, B, \bar{D})$
28	$B \cdot \bar{D}$	$M(0, B, \bar{D})$	73	$\bar{A} \cdot \bar{B} + \bar{A} \cdot D + \bar{B} \cdot D$	$\bar{M}(A, B, \bar{D})$
29	$\bar{B} \cdot \bar{D}$	$\bar{M}(1, B, D)$	74	$A \cdot C + A \cdot D + C \cdot D$	$M(A, C, D)$
30	$D \cdot C$	$M(0, D, C)$	75	$\bar{A} \cdot \bar{C} + \bar{A} \cdot \bar{D} + \bar{C} \cdot \bar{D}$	$\bar{M}(A, C, D)$
31	$\bar{D} \cdot C$	$M(0, \bar{D}, C)$	76	$\bar{A} \cdot C + \bar{A} \cdot D + C \cdot D$	$M(\bar{A}, C, D)$
32	$D \cdot \bar{C}$	$M(0, D, \bar{C})$	77	$A \cdot \bar{C} + A \cdot \bar{D} + \bar{C} \cdot \bar{D}$	$\bar{M}(\bar{A}, C, D)$
33	$\bar{D} \cdot \bar{C}$	$\bar{M}(1, D, C)$	78	$A \cdot \bar{C} + A \cdot D + \bar{C} \cdot D$	$M(A, \bar{C}, D)$
34	$A + B$	$M(A, B, 1)$	79	$\bar{A} \cdot C + \bar{A} \cdot \bar{D} + C \cdot \bar{D}$	$\bar{M}(A, \bar{C}, D)$
35	$\bar{A} + B$	$M(\bar{A}, B, 1)$	80	$A \cdot C + A \cdot \bar{D} + C \cdot \bar{D}$	$M(A, C, \bar{D})$
36	$A + \bar{B}$	$M(A, \bar{B}, 1)$	81	$\bar{A} \cdot \bar{C} + \bar{A} \cdot D + \bar{C} \cdot D$	$\bar{M}(A, C, \bar{D})$
37	$\bar{A} + \bar{B}$	$\bar{M}(A, B, 0)$	82	$B \cdot C + B \cdot D + C \cdot D$	$M(B, C, D)$
38	$A + C$	$M(A, 1, C)$	83	$\bar{B} \cdot \bar{C} + \bar{B} \cdot \bar{D} + \bar{C} \cdot \bar{D}$	$\bar{M}(B, C, D)$
39	$\bar{A} + C$	$M(\bar{A}, 1, C)$	84	$\bar{B} \cdot C + \bar{B} \cdot D + C \cdot D$	$M(\bar{B}, C, D)$
40	$A + \bar{C}$	$M(A, 1, \bar{C})$	85	$B \cdot \bar{C} + B \cdot \bar{D} + \bar{C} \cdot \bar{D}$	$\bar{M}(\bar{B}, C, D)$
41	$\bar{A} + \bar{C}$	$\bar{M}(A, 0, C)$	86	$B \cdot \bar{C} + B \cdot D + \bar{C} \cdot D$	$M(B, \bar{C}, D)$
42	$A + D$	$M(A, 1, D)$	87	$\bar{B} \cdot C + \bar{B} \cdot \bar{D} + C \cdot \bar{D}$	$\bar{M}(B, \bar{C}, D)$
43	$\bar{A} + D$	$M(\bar{A}, 1, D)$	88	$B \cdot C + B \cdot \bar{D} + C \cdot \bar{D}$	$M(B, C, \bar{D})$
44	$A + \bar{D}$	$M(A, 1, \bar{D})$	89	$\bar{B} \cdot \bar{C} + \bar{B} \cdot D + \bar{C} \cdot D$	$\bar{M}(B, C, \bar{D})$

**Source: Author**

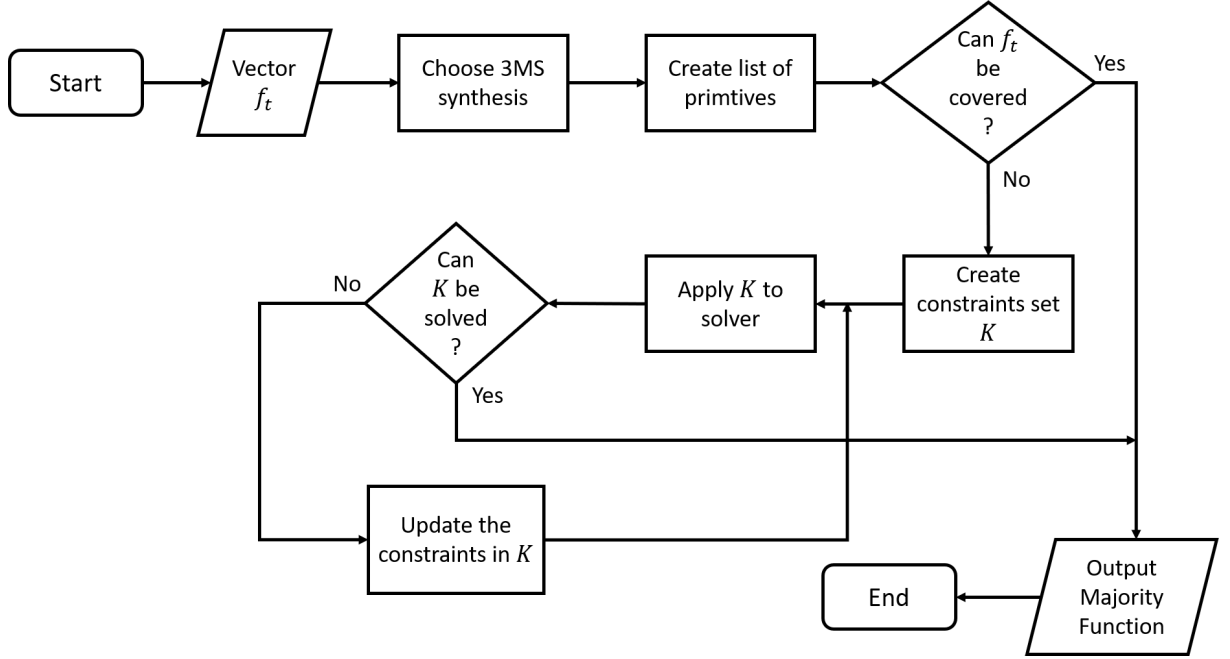
### 3 3MS ALGORITHM

In this chapter the 3MS (Majority Math Model Solver) algorithm, proposed in this work, is presented. The 3MS algorithm receives a vector of binary values as input, representing the output of a truth table, and returns the minimum cost majority function for its coverage. The input binary vector is represented by  $f_t$ , where  $0 \leq t < 2^n$  and index  $t$  a to the input table's terms. The coverage of  $f_t$  is based on the formulation of a constraints set, represented by  $K$ , that encodes a majority logic problem into a linear optimization model. The built set of constraints  $K$  is then applied to an optimization solver, generating results that will be posteriorly interpreted by the algorithm and translated into the output majority function.

The 3MS algorithm can be divided into two different synthesis, which varies depending on the chosen cost criteria. The 3MS-Depth synthesis prioritizes the minimization of levels (depth), followed by the minimization of gates (size) in the output majority function. The 3MS-Size synthesis prioritizes the minimization of size, followed by depth. Both synthesis consider the minimization of inverters and literals as third and fourth cost criteria. In addition, for both synthesis the cost of repeated gates is disregarded.

The first step of the 3MS algorithm is the creation of the primitives list, using the logic described in subsection 2.4. If  $f_t$  could not be covered by a primitive, the chosen synthesis is started. For each synthesis the 3MS algorithm builds a different set of constraints, molding the minimization problem for the corresponding cost criteria. Both synthesis can also consider majority gates with 3 (maj-3) or 5 (maj-5) inputs when building its constraints. Figure 1 presents a flowchart for the overall 3MS logic.

Section 3.1 presents a brief explanation about Quantum-Dot Cellular Automata (QCA), aiming to exemplify a majority based nanotechnology that would benefit from the exact synthesis approach presented in this work. Section 3.2 presents the first steps of the 3MS algorithm, explaining how the primitives table is stored and the variables generated from it. 3MS-Depth and 3MS-Size synthesis are presented, respectively, in sections 3.3 and 3.4. All constraints are explained individually in detail for each synthesis.

**Figure 1** – Flowchart for the overall 3MS algorithm.**Source:** Author

### 3.1 APPLICABILITY OF THE 3MS ALGORITHM

The main goal of this work is to present a new exact synthesis algorithm for optimal depth and size minimization, for both maj-3 and maj-5 functions, that can also optimize 2 additional cost criteria, the number of inverters and literals, without losing the optimal results for depth and size optimization.

In QCA technology, circuits are formed by quantum cells composed by 4 quantum dots and 2 electrons. Electrons can not leave the QCA cell but are able to travel between the dots. Due to Coulomb's law, electrons can only occupy opposite diagonals in the cells, generating the 2 possible polarizations  $P = +1$  and  $P = -1$ . The polarization of a cell defines its logical value:  $P = -1$  represents the logic state False (constant 0) and  $P = +1$  represents the logic state True (constant 1) (CHUNG *et al.*, 2017).

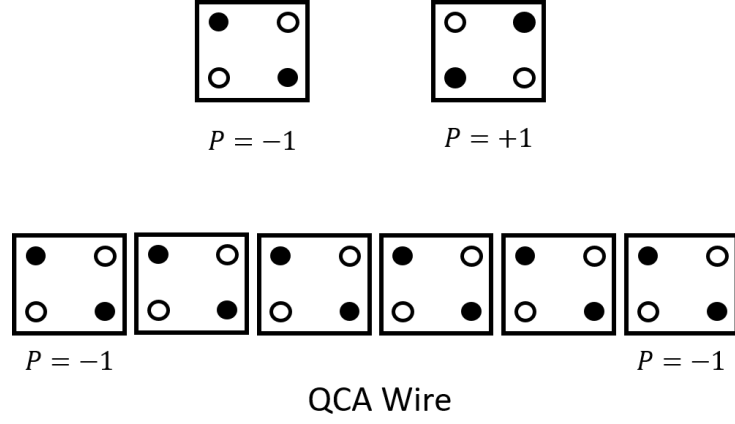
The main QCA components include the QCA cell, the QCA wire, the QCA majority gate and the QCA inverter. The QCA wire is simple a line of cells that holds the polarization. Figure 2 shows an example of a QCA cell for each polarization and an example of a QCA wire, holding the polarization  $P = -1$ .

A QCA majority gate represents the main logic operation of a QCA circuit. The totals of cells needed to build a majority gate are, respectively, 5 for a maj-3 gate and 10



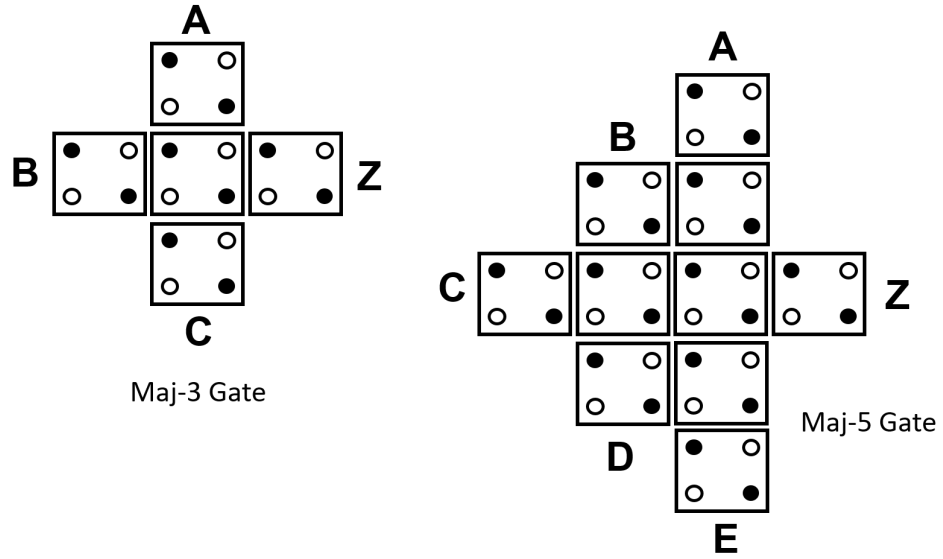
for a maj-5 gate. Figure 3 shows the layout of both maj-3 and maj-5 gates, considering functions  $Z = M(A, B, C)$  and  $Z = M(A, B, C, D, E)$ .

**Figure 2** – Example of QCA cells and wire.



Source: Author

**Figure 3** – QCA layout for  $Z = M(A, B, C)$  and  $Z = M(A, B, C, D, E)$



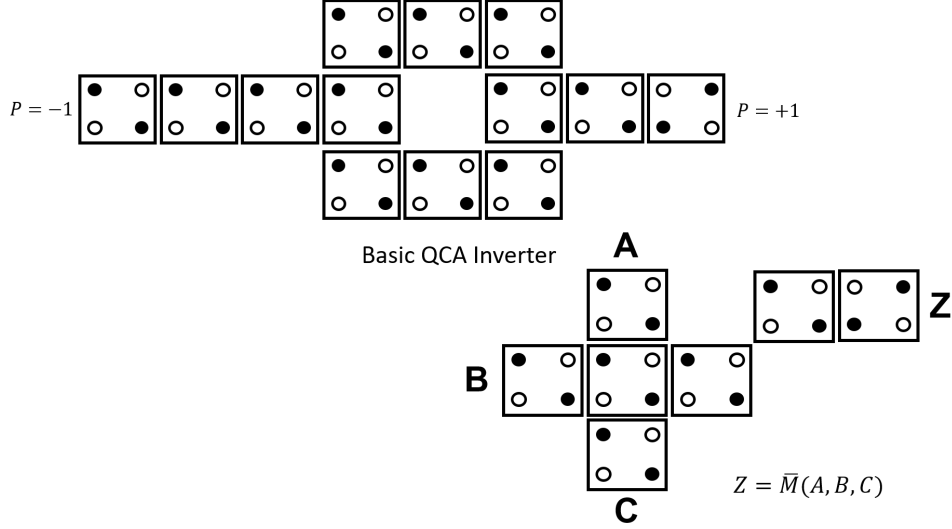
Source: Author

The basic QCA inverter needs a total of 11 cells to be built. However, in QCA technology, cells placed diagonally to each other have reverse polarizations. This characteristic can be exploited to implement an inverter using only 2 cells (KONG; SHANG; LU, 2010). Figure 4 shows the basic QCA inverter and the majority function  $Z = \overline{M}(A, B, C)$ , exemplifying both inverter layouts.

Note that in QCA the number of cells needed to build an inverter is at least 2, almost

half the cost needed to build a maj-3 gate, which is 5 (KO *et al.*, 2022). Therefore, even though depth and size optimization are still priority, to also consider the reduction of inverters results in a great decrease in the circuits total cost.

**Figure 4** – QCA layout for basic inverter and  $Z = \overline{M}(A, B, C)$

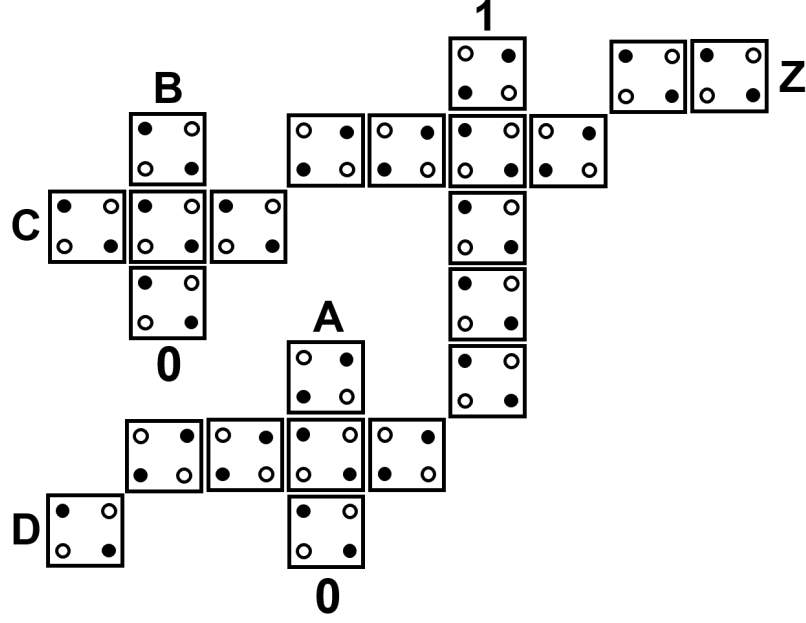


Source: Author

To exemplify the impact of inverter optimization in QCA circuits, consider the majority function  $\overline{M}(1, \overline{M}(A, \overline{D}, 0), \overline{M}(B, C, 0))$ , which has 2 levels, 3 gates, 4 inverters and 4 literals. Even though this function is already optimized for depth and size, it can still be further optimized to  $M(0, M(A, \overline{D}, 0), M(B, C, 0))$ , which has the same depth, size and literals but with 3 less inverters. Figure 5 shows the layout and amount of needed QCA cells to build the function before inverter optimization, where  $Z = \overline{M}(1, \overline{M}(A, \overline{D}, 0), \overline{M}(B, C, 0))$ .

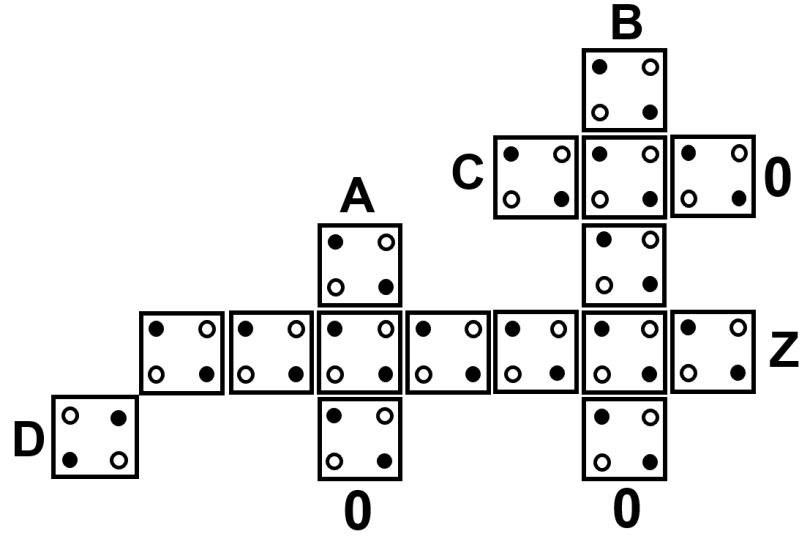
Before inverter optimization, a total of 22 QCA cells were needed to build the circuit. Figure 6 shows the layout and the amount of needed QCA cells to build the function after inverter optimization, where  $Z = M(0, M(A, \overline{D}, 0), M(B, C, 0))$ . After inverter optimization, the number of QCA cells needed to build  $Z$  was reduced to 16 cells, generating a 28% reduction.

**Figure 5** – QCA layout for  $Z = \overline{M}(1, \overline{M}(A, \overline{D}, 0), \overline{M}(B, C, 0))$



Source: Author

**Figure 6** – QCA layout for  $Z = M(0, M(A, \overline{D}, 0), M(B, C, 0))$



Source: Author

### 3.2 3MS PRE-SYNTHESIS

The primitives list is created based on the variables  $n$  and  $m$ , that represents, respectively, the number of input variables in  $f_t$  and the number of majority gate inputs.

Since the 3MS can consider maj-3 or maj-5 functions, variable  $m$  will always be 3 or 5. From the primitives table, vector  $C_i$  is created. Vector  $C_i$  stores the cost of each primitive  $i$ , where  $0 \leq i < p$  and  $p$  represents the amount of functions in the primitives table. Positions  $C_0$  and  $C_1$  corresponds to the constants 0 and 1, respectively, and have a cost equal to 0. The cost of the remaining primitives can be calculated by Equation 15.

$$C_i = 10^4 \cdot G_i + 100 \cdot I_i + L_i \quad \forall i \mid 2 \leq i < p. \quad (15)$$

Vectors  $G_i$ ,  $I_i$  and  $L_i$  corresponds, respectively, to the number of gates, inverters and literals of a primitive  $i$ . Literals are the input variables, in their complemented form or not. The weights assigned to each vector defines the prioritization of each cost criteria. The primitive  $M(\overline{A}, B, 0)$ , for instance, is composed by 1 gate, 1 inverter and 2 literals, resulting in a total cost of 10,102. The numerical difference of each weight is intended to avoid cases where stacking a cost criteria would imply addition to another cost criteria, when building higher level functions.

Matrix  $Y_{it}$  stores the truth table of every primitive function in binary vector format. To exemplify, table 19 shows the matrix  $Y_{it}$  considering  $n = m = 3$ , where  $0 \leq t < 8$ . Set  $e$  is also created, storing the index  $i$  of all primitives with an inverted root. Primitives with an inverted root can be single complemented literals or a complemented gate. Examples are  $\overline{A}$  and  $\overline{M}(A, B, C)$ . Based on set  $e$ , set  $k$  is also created, which stores all elements in  $e$  along with the indexes that represents the constants 0 and 1:  $i = 0$  and  $i = 1$ , respectively.

**Table 19** – Matrix  $Y_{it}$  for  $n = m = 3$ .

Index $i$	Primitive Function	Line $i$ of $Y_{it}$	Index $i$	Primitive Function	Line $i$ of $Y_{it}$
0	0	00000000	20	$M(A, B, 1)$	00111111
1	1	11111111	21	$M(\bar{A}, B, 1)$	11110011
2	$A$	00001111	22	$M(A, \bar{B}, 1)$	11001111
3	$B$	00110011	23	$\bar{M}(A, B, 0)$	11111100
4	$C$	01010101	24	$M(A, 1, C)$	01011111
5	$\bar{A}$	11110000	25	$M(\bar{A}, 1, C)$	11110101
6	$\bar{B}$	11001100	26	$M(A, 1, \bar{C})$	10101111
7	$\bar{C}$	10101010	27	$\bar{M}(A, 0, C)$	11111010
8	$M(A, B, 0)$	00000011	28	$M(1, B, C)$	01110111
9	$M(\bar{A}, B, 0)$	00110000	29	$M(1, \bar{B}, C)$	11011101
10	$M(A, \bar{B}, 0)$	00001100	30	$M(1, B, \bar{C})$	10111011
11	$\bar{M}(A, B, 1)$	11000000	31	$\bar{M}(0, B, C)$	11101110
12	$M(A, 0, C)$	00000101	32	$M(A, B, C)$	00010111
13	$M(\bar{A}, 0, C)$	01010000	33	$\bar{M}(A, B, C)$	11101000
14	$M(A, 0, \bar{C})$	00001010	34	$M(\bar{A}, B, C)$	01110001
15	$\bar{M}(A, 1, C)$	10100000	35	$\bar{M}(\bar{A}, B, C)$	10001110
16	$M(0, B, C)$	00010001	36	$M(A, \bar{B}, C)$	01001101
17	$M(0, \bar{B}, C)$	01000100	37	$\bar{M}(A, \bar{B}, C)$	10110010
18	$M(0, B, \bar{C})$	00100010	38	$M(A, B, \bar{C})$	00101011
19	$\bar{M}(1, B, C)$	10001000	39	$\bar{M}(A, B, \bar{C})$	11010100

**Source: Author**

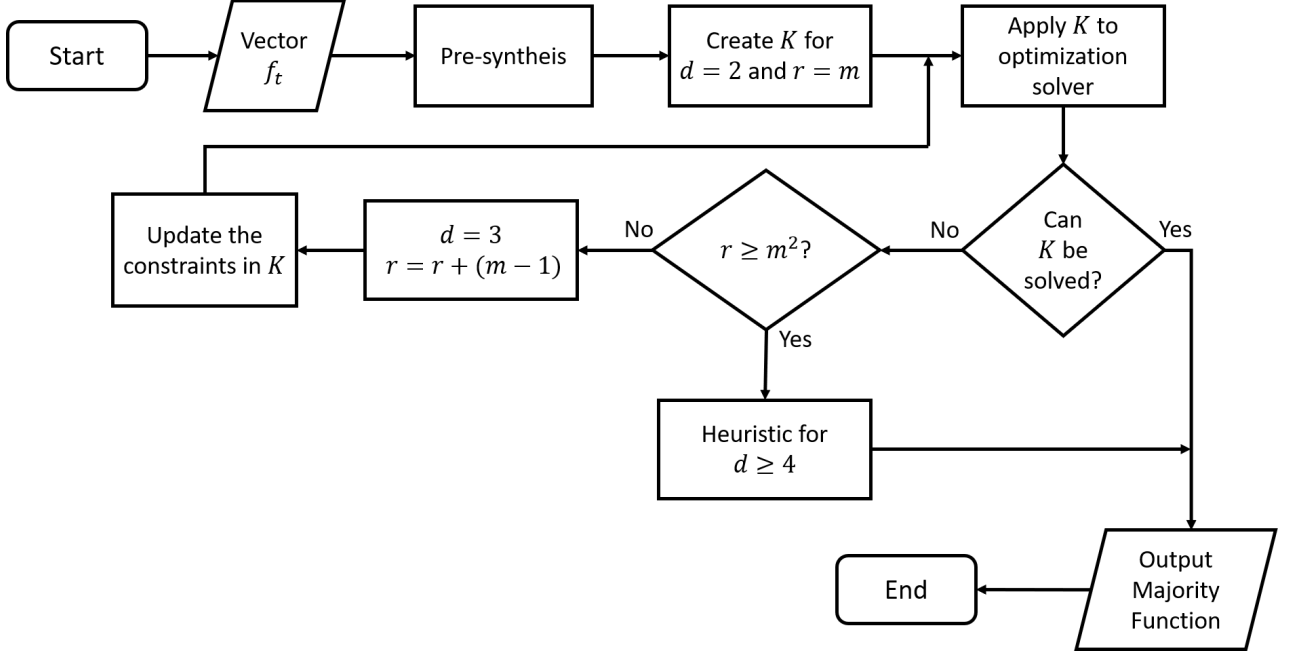
### 3.3 3MS-DEPTH SYNTHESIS

If 3MS-Depth is the chosen synthesis and  $f_t$  could not be covered by a primitive function, the initial formulation of  $K$  considers a 2-level output majority function. Every time  $K$  is applied to the optimization solver and a valid output function could not be found, the set of constraints  $K$  is rebuilt considering an output function with one additional level, until a maximum of 3 levels. If  $f_t$  can not be covered by an output function with up to 3 levels, a heuristic is used to cover  $f_t$  considering functions with 4 or more levels.

Set  $K$  is constructed based on the variables  $d$  and  $r$ , where  $d$  and  $r$  represents, respectively, the number of levels (depth) and primitives in the output function. Since a 2-level majority function is always formed by  $m$  primitives,  $d = 2$  will always imply  $r = m$ . The synthesis for 3-level functions ( $d = 3$ ) is subdivided based on the value of  $r$ , that will range from  $(2 \cdot m) - 1$  to  $m^2$  and be incremented by  $m - 1$  every

time  $K$  could not be solved. Therefore,  $m = 3$  implies  $5 \leq r \leq 9$ , where  $r$  is incremented by 2, and  $m = 5$  implies  $9 \leq r \leq 25$ , where  $r$  is incremented by 4. Function  $M(M(A, D, M(B, C, 1)), 0, \overline{M}(B, C, D))$ , for instance, is a 3-level maj-3 function where  $r = 5$ , since its formed by 5 primitives. Figure 7 shows a flowchart for a better understanding of the overall 3MS-Depth synthesis.

**Figure 7** – Flowchart for the overall 3MS-Depth synthesis.



Source: Author

### 3.3.1 Optimization model for 2-level majority functions

This subsection presents the formulation of constraints set  $K$  when the coverage of  $f_t$  by a 2-level majority function is being considered. First step is the creation of the integer vector  $X_i$ , where the value of each position  $i$  can range from 0 to  $\lfloor \frac{m}{2} \rfloor$ . Each position of  $X_i$  pertains to a primitive  $i$  and the value assigned to that position represents the number of times that a primitive  $i$  appears in the output majority function. Since a 2-level maj- $m$  function will always be formed by a combination of  $m$  primitives, the constraint shown in Equation 16 is added to  $K$ .

$$\sum_{i=0}^{p-1} X_i = m \quad (16)$$

Next step is the creation of the integer variables  $S1$  and  $S2$ , followed by the addition

of the constraints shown in Equations 17 and 18 to  $K$ .

$$\sum_{i \in e} X_i = m \rightarrow S1 = \left\lfloor \frac{m}{2} \right\rfloor \quad (17)$$

$$\sum_{i \in k} X_i = m \rightarrow S2 = \left\lfloor \frac{m}{2} \right\rfloor \quad (18)$$

Note that the constraints consider the elements of set  $e$  and  $k$ , respectively. This pair of constraints guaranties the reduction of inverters based on the  $\Omega.I$  axiom, which states that a majority function is self dual. Set  $k$  considers the constants 0 and 1 along with all primitives with and inverted root, stored by set  $e$ . Therefore, variable  $S2$  will value to  $\left\lfloor \frac{m}{2} \right\rfloor$  if all selected primitives are either in set  $e$  or are constants. Variable  $S1$  will value to  $\left\lfloor \frac{m}{2} \right\rfloor$  if all selected primitives are in set  $e$ , but none is a constant. For  $m = 3$  and  $m = 5$  we have, respectively,  $\left\lfloor \frac{3}{2} \right\rfloor = 1$  and  $\left\lfloor \frac{5}{2} \right\rfloor = 2$ .

To exemplify consider  $m = n = 3$  and the solution  $X_i = \{X_5 = 1, X_{11} = 1, X_{39} = 1\}$ . For  $n = 3$ , index  $i = 5$ ,  $i = 11$  and  $i = 39$  corresponds, respectively, to the primitives  $\bar{A}$ ,  $\bar{M}(A, B, 1)$  and  $\bar{M}(A, B, \bar{C})$ . Without considering  $\Omega.I$ , the 2-level output function would be  $M(\bar{A}, \bar{M}(A, B, 1), \bar{M}(A, B, \bar{C}))$ , having a total cost of 30,406 based on the weights of Equation 15, which calculates the cost vector  $C_i$ . Considering  $\Omega.I$ ,  $S1$  will value to 1 because  $X_5 + X_{11} + X_{39} = 3$  and all 3 primitives are no constants and contained in  $e$ . Since  $S1 = 1$  also implies  $S2 = 1$ , the number of inverters will be decreased by 2, generating a total cost of 30,206 and updating the output function to its optimized version  $\bar{M}(A, M(A, B, 1), M(A, B, \bar{C}))$ .

To exemplify a case where only  $S2$  values to 1, we replace  $X_5 = 1$  for  $X_0 = 1$  in the previous example, forming  $M(0, \bar{M}(A, B, 1), \bar{M}(A, B, \bar{C}))$  with a total cost of 30,305. The condition  $X_0 + X_{11} + X_{39} = 1$  implies  $S1 = 0$  and  $S2 = 1$ , since only  $S2$  considers constants in its sum. The optimized output function will then be  $\bar{M}(1, M(A, B, 1), M(A, B, \bar{C}))$ , with 1 less inverter and a total cost of 30,205.

The objective function for the optimization model is defined by Equation 19. Note that  $Z$  is composed simply by the sum of  $X_i$  multiplied by each respective cost  $C_i$ , with the subtraction of the excluded inverters signaled by  $S1$  and  $S2$ .

$$\min Z = \sum_{i=0}^{p-1} X_i \cdot C_i - 100 \cdot (S1 + S2) \quad (19)$$

The coverage constraints are built from the binary matrix  $Y_{it}$ , defined in the pre-synthesis phase. Each line  $i$  of  $Y_{it}$  stores the truth table of a primitive  $i$ , in binary vector format. Likewise, each column  $t$  of  $Y_{it}$  stores a vector of  $p$  elements representing the coverage of a term  $t$  by each primitive  $i$ . If  $Y_{it} = 1$ , then term  $t$  is covered by primitive  $i$ . If  $Y_{it} = 0$ , then term  $t$  is not covered by primitive  $i$ . From  $Y_{it}$ , a set of  $2^n$  constraints is created, each constraint corresponding to a term  $t$  of  $f_t$ . For minterms, where  $f_t = 1$ , the constraint shown in Equation 20 is added to  $K$ .

$$\sum_{i=0}^{p-1} X_i \cdot Y_{it} > \left\lfloor \frac{m}{2} \right\rfloor \quad \forall f_t = 1 \mid 0 \leq t < 2^n \quad (20)$$

For maxterms, where  $f_t = 0$ , the constraint shown in Equation 21 is added to  $K$ . Note that the bound is replaced for  $\leq \left\lfloor \frac{m}{2} \right\rfloor$ , since  $f_t = 0$  implies that the term  $t$  should not be covered by the majority of the selected primitives in  $X_i$ .

$$\sum_{i=0}^{p-1} X_i \cdot Y_{it} \leq \left\lfloor \frac{m}{2} \right\rfloor \quad \forall f_t = 0 \mid 0 \leq t < 2^n \quad (21)$$

These  $2^n$  constraints ensures that the  $m$  selected primitives will, together, cover all minterms ( $f_t = 1$ ) more than  $\left\lfloor \frac{m}{2} \right\rfloor$  times and, at the same time, will not cover maxterms ( $f_t = 0$ ) more than  $\left\lfloor \frac{m}{2} \right\rfloor$  times. As an example, consider  $n = 4$ ,  $m = r = 5$ , the input binary vector  $f_t = 0100000011000101$  and vector  $X_i$  with the solution  $\{X_0 = 2, X_1 = 1, X_{40} = 1, X_{86} = 1\}$ , all other positions of  $X_i$  being equal to 0. The indexes  $i = 0$ ,  $i = 1$ ,  $i = 40$  and  $i = 86$  represents, respectively, the primitives 0, 1,  $M(B, \overline{C}, 1, 1, 0)$  and  $M(A, \overline{B}, D, 0, 1)$ , resulting in  $Z = 20,205$  and forming the output majority function  $M(0, 0, 1, M(B, \overline{C}, 1, 1, 0), M(A, \overline{B}, D, 0, 1))$ . Therefore, for  $m = 5$ , the combined primitives must cover every minterm of  $f_t$  at least 3 times and can not cover any maxterm more than 2 times. Table 20 shows the coverage of  $f_t$  in truth table format. Note that the combination of constants 001 was simplified to 0, since  $M(0, 0, 1) = 0$ .

Equation 22 shows the complete optimization model for 2-level majority functions. The set of constraints  $K$  is then applied to the optimization solver, being translated into the output function if  $K$  could be solved. If  $K$  could not be solved, all constraints are reset and  $K$  is rebuild to cover  $f_t$  using a 3-level majority function.



**Table 20** – Example of  $f_t$  coverage.

$t$	0	$M(B, \overline{C}, 1, 1, 0)$	$M(A, \overline{B}, D, 0, 1)$	$f_t = M(0, 0, 1, M(B, \overline{C}, 1, 1, 0), M(A, \overline{B}, D, 0, 1))$
0	0	1	0	0
1	0	1	1	1
2	0	0	0	0
3	0	0	1	0
4	0	1	0	0
5	0	1	0	0
6	0	1	0	0
7	0	1	0	0
8	0	1	1	1
9	0	1	1	1
10	0	0	1	0
11	0	0	1	0
12	0	1	0	0
13	0	1	1	1
14	0	1	0	0
15	0	1	1	1

**Source: Author.**

$$\begin{aligned}
\min Z &= \sum_{i=0}^{p-1} X_i \cdot C_i - 100 \cdot (S1 + S2) \\
\text{s.t.} \\
\sum_{i=0}^{p-1} X_i &= m \\
\sum_{i=0}^{p-1} X_i &\geq 0 \\
\sum_{i \in e} X_i = m &\rightarrow S1 = \left\lfloor \frac{m}{2} \right\rfloor \\
\sum_{i \in k} X_i = m &\rightarrow S2 = \left\lfloor \frac{m}{2} \right\rfloor \\
S1, S2 &\in \{0, \left\lfloor \frac{m}{2} \right\rfloor\} \\
\sum_{i=0}^{p-1} X_i \cdot Y_{it} &> \left\lfloor \frac{m}{2} \right\rfloor \quad \forall f_t = 1 \mid 0 \leq t < 2^n \\
\sum_{i=0}^{p-1} X_i \cdot Y_{it} &\leq \left\lfloor \frac{m}{2} \right\rfloor \quad \forall f_t = 0 \mid 0 \leq t < 2^n
\end{aligned} \tag{22}$$

### 3.3.2 Optimization model for 3-level majority functions

This subsection presents the formulation of  $K$  for the coverage of  $f_t$  with a 3-level majority function ( $d = 3$ ). The first step is to update vector  $X_i$  into the matrix  $X_{ij}$ , where  $1 \leq j \leq 1 + \lfloor \frac{r}{m} \rfloor$ . Each position of  $X_{ij}$  represents a primitive  $i$  assigned to a majority function  $j$  that composes the output function. Position  $j = 1$  refers to the first level of the output function, while positions  $j \geq 2$  represents 2-level functions, formed by  $m$  primitives each, in the second level of the output function.

As shown in the 3MS-Depth flowchart, presented in Figure 7, for  $d = 3$  the value of  $r$  will range from  $2 \cdot m - 1$  to  $m^2$  and be incremented by  $m - 1$  every time  $K$  could not be solved. Considering  $m = 3$ , the initial value of  $r = 5$  implies  $1 \leq j \leq 2$ . Therefore, considering  $r = 5$ ,  $X_{ij}$  is composed by vectors  $X_{i1}$  and  $X_{i2}$ , where 2 inputs of  $X_{i1}$  are primitives and the third input is a 2-level function, formed by 3 primitives, defined by  $X_{i2}$ . As an example, consider the 3-level output function  $M(\overline{M}(1, D, M(B, C, 1)), A, M(B, C, \overline{D}))$ . The 2-level function  $\overline{M}(1, D, M(B, C, 1))$ , formed by 3 primitives, corresponds to  $X_{i2}$ , which is an input for  $X_{i1}$  in conjunction with the primitives  $A$  and  $M(B, C, \overline{D})$ .

The constraints shown in Equations 23 and 24 define the number of primitives in each function  $j$  of  $X_{ij}$ . Since all 2-level functions are formed by  $m$  primitives,  $X_{ij}$  is equal to  $m$  for each  $j > 1$ . The number of primitives in the first level of the output function, represented by  $X_{i1}$ , is given by  $r - m \cdot \lfloor \frac{r}{m} \rfloor$ .

$$\sum_{i=0}^{p-1} X_{ij} = m \quad \forall j > 1 \quad (23)$$

$$\sum_{i=0}^{p-1} X_{i1} = r - m \cdot \left\lfloor \frac{r}{m} \right\rfloor \quad (24)$$

Therefore, for all values of  $r$ ,  $j > 1$  corresponds to 2-level majority functions in the second level of the output function and  $j = 1$  refers to the primitives in the first level, represented by vector  $X_{i1}$ .

The solution of  $K$  consists on a sequence of values for matrix  $X_{ij}$  and the total cost of the selected primitives, represented by  $Z$ . Each position of  $X_{ij}$  represents how many times a primitive function  $i$  was selected to build a determined function  $j$ . The 3MS algorithm reads the  $X_{ij}$  matrix and builds the output majority function based on  $i$  and

$j$ , adding every primitive to its respective level and creating a new maj- $m$  operator to comprise all created functions. For each level  $j > 1$ , a maj- $m$  function is also created for each  $m$  selected primitives.

As an example consider  $m = d = 3$ ,  $n = r = 5$  and the matrix  $X_{ij}$  with the solution  $\{X[0, 1] = 1, X[57, 1] = 1, X[2, 2] = 1, X[6, 2] = 1, X[31, 2] = 1\}$ , all other positions of  $X_{ij}$  being equal to 0. The indexes  $i = 0$ ,  $i = 2$ ,  $i = 6$ ,  $i = 31$  and  $i = 57$  represents, respectively, the primitives 0,  $A$ ,  $E$ ,  $M(B, \overline{C}, 0)$  and  $M(D, \overline{E}, 1)$ .

From index  $j$ , function  $M(D, \overline{E}, 1)$  and constant 0 are set to level 1 ( $j = 1$ ), while the primitives  $A$ ,  $E$  and  $M(B, \overline{C}, 0)$  are set to level 2 ( $j = 2$ ). Since there are 3 primitives in level 2, a new maj-3 operator is created, resulting in the 2-level majority function  $M(A, E, M(B, \overline{C}, 0))$  that will be an input for level 1 along with 0 and  $M(D, \overline{E}, 1)$ . Lastly, the maj-3 operator for level 0 is created, resulting in the 3-level output majority function  $M(0, M(D, \overline{E}, 1), M(A, E, M(B, \overline{C}, 0)))$ .

Next step is to update the variables  $S1$  and  $S2$  into the vectors  $S1_j$  and  $S2_j$ , adding Equations 25 and 26 to  $K$ . This constraints represent the application of  $\Omega.I$ 's logic to every function where  $j > 1$ .

$$\sum_{i \in e} X_{ij} = m \rightarrow S1_j = \left\lfloor \frac{m}{2} \right\rfloor \quad \forall j > 1 \quad (25)$$

$$\sum_{i \in k} X_{ij} = m \rightarrow S2_j = \left\lfloor \frac{m}{2} \right\rfloor \quad \forall j > 1 \quad (26)$$

The constraints shown in Equations 27 and 28 ensures the application of  $\Omega.I$  for the first level of the output function, considering the inverted root primitives and the constants in  $X_{i1}$  together with  $S1_j$  and  $S2_j$ , respectively.

$$\sum_{i \in e} X_{i1} + \sum_{j=2}^{1+\left\lfloor \frac{r}{m} \right\rfloor} S1_j = m \rightarrow S1_1 = \left\lfloor \frac{m}{2} \right\rfloor \quad (27)$$

$$\sum_{i \in k} X_{i1} + \sum_{j=2}^{1+\left\lfloor \frac{r}{m} \right\rfloor} S2_j = m \rightarrow S2_1 = \left\lfloor \frac{m}{2} \right\rfloor \quad (28)$$

To exemplify the application of  $\Omega.I$  through vectors  $S1_j$  and  $S2_j$ , consider  $d = m = 3$ ,  $n = 4$ ,  $r = 7$  and the matrix  $X_{ij}$  with the solution  $\{X[61, 1] = 1, X[17, 2] = 1, X[7, 2] = 1, X[0, 2] = 1, X[57, 3] = 1, X[8, 3] = 1, X[85, 3] = 1\}$ , all other positions of  $X_{ij}$  being

equal to 0. Considering  $r = 7$  we have  $j = 3$ , meaning that the output function is formed by 1 primitive in the first level ( $j = 1$ ) and by 2 functions formed by 3 primitives each, corresponding to  $2 \leq j \leq 3$ .

The positions  $X[17, 2]$ ,  $X[7, 2]$  and  $X[0, 2]$  corresponds, respectively, to the functions  $\overline{M}(A, 1, C)$ ,  $\overline{B}$  and 0, forming the 2-level function  $M(\overline{M}(A, 1, C), \overline{B}, 0)$  for  $j = 2$ . For  $m = 3$  we have  $\lfloor \frac{m}{2} \rfloor = 1$ . Therefore, for  $j = 2$ ,  $S2_2$  will value to 1 because all 3 selected primitives are in set  $k$ , being primitives with an inverted root or a constant. The function  $M(\overline{M}(A, 1, C), \overline{B}, 0)$ , which has a total cost of 20,203, can be simplified to  $\overline{M}(M(A, 1, C), B, 1)$ , updating its cost to 20,103 since the updated form has 1 less inverter.

The positions  $X[57, 3] = 1$ ,  $X[8, 3] = 1$  and  $X[85, 3] = 1$  corresponds, respectively, to the functions  $\overline{M}(0, C, D)$ ,  $\overline{C}$  and  $\overline{M}(\overline{B}, C, D)$ , forming the 2-level function  $M(\overline{M}(0, C, D), \overline{C}, \overline{M}(\overline{B}, C, D))$  for  $j = 3$ . For  $j = 3$ , both  $S1_3$  and  $S2_3$  will value to 1, since all 3 selected primitives are in set  $e$ , which is contained by set  $k$ . The function  $M(\overline{M}(0, C, D), \overline{C}, \overline{M}(\overline{B}, C, D))$  will have its cost of 30,405 updated to 30,205, since it can be simplified to  $\overline{M}(M(0, C, D), C, M(\overline{B}, C, D))$ , which has 2 less inverters.

The position  $X[61, 1]$  refers to the primitive  $\overline{M}(\overline{A}, B, C)$  in the first level of the output function, where  $j = 1$ . Combining all  $j$  functions we have the output 3-level function  $M(\overline{M}(\overline{A}, B, C), \overline{M}(M(A, 1, C), B, 0), \overline{M}(M(0, C, D), C, M(\overline{B}, C, D)))$ , which has a total cost of 70,511. Through Equations 27 and 28, that defines  $\Omega$ 's logic to the first level of the output function, we have  $\sum_{j=2}^{1+\lfloor \frac{r}{m} \rfloor} S1_j = 1$  and  $\sum_{j=2}^{1+\lfloor \frac{r}{m} \rfloor} S2_j = 2$ , signaling the already removed inverters, along with the inverted root primitives in the first level. Since  $\overline{M}(\overline{A}, B, C)$  is both on set  $e$  and  $k$ , we have  $S1_1$  and  $S2_1$  valuing to 1, resulting in a decrease of 2 more inverters and updating the output majority function to  $\overline{M}(M(\overline{A}, B, C), M(M(A, 1, C), B, 0), M(M(0, C, D), C, M(\overline{B}, C, D)))$ , which has a total cost of 70,311.

Next step is the creation of binary vector  $W_i$  and set  $c$ . Set  $c$  stores the index of all primitives with a majority gate, where  $C_i \geq 10,000$ . Vector  $W_i$  stores the weight applied to every  $C_i$  where  $i \in c$ , resulting in  $W_i = 1$  if that primitive is assigned at least once to the output function and to  $W_i = 0$  otherwise. This logic is defined by the constraints shown in Equations 29 and 30, ensuring that the cost of repeated primitives is disregarded in the total cost of the output function.

$$\sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \geq 1 \rightarrow W_i = 1 \quad \forall i \in c \quad (29)$$

$$\sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \leq 0 \rightarrow W_i = 0 \quad \forall i \in c \quad (30)$$

The objective function for  $d = 3$  is defined by Equation 31.

$$\min Z = \sum_{i \in c} W_i \cdot C_i + \sum_{i \notin c} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot C_i - 100 \cdot \left( \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} S1_j + \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} S2_j \right) \quad (31)$$

Note that  $Z$  is composed by the sum of all  $C_i$  multiplied by  $W_i$  if  $i \in c$  or by  $X_{ij}$  otherwise, followed by the subtraction of all inverters signaled by  $S1_j$  and  $S2_j$ .

The coverage of  $f_t$  for  $d = 3$  follows the same logic as  $d = 2$ . Based on the matrix  $Y_{it}$ ,  $2^n$  constraints are added to  $K$ , each for a term  $t$  of  $f_t$ . The constraint shown in Equation 32 is added to  $K$  if  $f_t = 1$ . If  $f_t = 0$ , the bound is replaced by  $\leq \lfloor \frac{m}{2} \rfloor$ , as shown in Equation 33.

$$\sum_{i=0}^{p-1} X_{i1} \cdot Y_{it} + \frac{\sum_{i=0}^{p-1} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot Y_{it}}{\lfloor \frac{m}{3} \cdot 2 \rfloor} > \lfloor \frac{m}{2} \rfloor \quad \forall f_t = 1 \mid 0 \leq t < 2^n \quad (32)$$

$$\sum_{i=0}^{p-1} X_{i1} \cdot Y_{it} + \frac{\sum_{i=0}^{p-1} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot Y_{it}}{\lfloor \frac{m}{3} \cdot 2 \rfloor} \leq \lfloor \frac{m}{2} \rfloor \quad \forall f_t = 0 \mid 0 \leq t < 2^n \quad (33)$$

The division by  $\lfloor \frac{m}{3} \cdot 2 \rfloor$  ensures the proper coverage weight to all 2-level functions that composes the 3-level output function, where  $j > 1$ . Also, this logic is not applied to  $X_{i1}$  because primitives in the first level does not form a 2-level function, having a higher weight in the coverage of  $f_t$ . Equation 34 shows the complete optimization model for 3-level majority functions.

The set of constraints  $K$  is then applied to the optimization solver and translated into the output function if  $K$  could be solved. If  $K$  could not be solved, the constraints are updated considering the increment  $r = r + (m - 1)$ . If  $r = m^2$  and  $K$  still could not be solved, an heuristic is used to cover  $f_t$  considering functions with 4 or more levels.

$$\begin{aligned}
\min Z &= \sum_{i \in c} W_i \cdot C_i + \sum_{i \notin c} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot C_i - 100 \cdot \left( \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} S1_j + \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} S2_j \right) \\
\text{s.t.} \\
\sum_{i=0}^{p-1} X_{ij} &= m \quad \forall j > 1 \\
\sum_{i=0}^{p-1} X_{i1} &= r - m \cdot \left\lfloor \frac{r}{m} \right\rfloor \\
X_{ij} &\geq 0 \\
\sum_{i \in e} X_{ij} = m &\rightarrow S1_j = \left\lfloor \frac{m}{2} \right\rfloor \quad \forall j > 1 \\
\sum_{i \in k} X_{ij} = m &\rightarrow S2_j = \left\lfloor \frac{m}{2} \right\rfloor \quad \forall j > 1 \\
\sum_{i \in e} X_{i1} + \sum_{j=2}^{1+\lfloor \frac{r}{m} \rfloor} S1_j = m &\rightarrow S1_1 = \left\lfloor \frac{m}{2} \right\rfloor \\
\sum_{i \in k} X_{i1} + \sum_{j=2}^{1+\lfloor \frac{r}{m} \rfloor} S2_j = m &\rightarrow S2_1 = \left\lfloor \frac{m}{2} \right\rfloor \\
S1_j, S2_j &\in \{0, \left\lfloor \frac{m}{2} \right\rfloor\} \\
\sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \geq 1 &\rightarrow W_i = 1 \quad \forall i \in c \\
\sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} = 0 &\rightarrow W_i = 0 \quad \forall i \in c \\
\sum_{i=0}^{p-1} X_{i1} \cdot Y_{it} + \frac{\sum_{i=0}^{p-1} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot Y_{it}}{\left\lfloor \frac{m}{3} \cdot 2 \right\rfloor} &> \left\lfloor \frac{m}{2} \right\rfloor \quad \forall f_t = 1 \mid 0 \leq t < 2^n \\
\sum_{i=0}^{p-1} X_{i1} \cdot Y_{it} + \frac{\sum_{i=0}^{p-1} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot Y_{it}}{\left\lfloor \frac{m}{3} \cdot 2 \right\rfloor} &\leq \left\lfloor \frac{m}{2} \right\rfloor \quad \forall f_t = 0 \mid 0 \leq t < 2^n
\end{aligned} \tag{34}$$

### 3.3.3 Heuristic for majority functions with 4 or more levels

This subsection presents an heuristic to cover  $f_t$  considering majority functions with 4 or more levels ( $d \geq 4$ ). The main logic of the heuristic consists in subdividing the input table  $f_t$  into truth tables that can be covered by majority functions with up to 3 levels,

using the optimization models presented in the previous subsections. Table  $f_t$  will be subdivided into  $m$  truth tables represented by  $F_{st}$ , where  $1 \leq s \leq m$ .

The first step is to apply don't care state status to half ( $2^{n-1}$ ) the terms of  $f_t$ , generating the first truth table to be covered, represented by the integer vector  $F_{1t}$ . A term is a don't care state if its coverage is irrelevant in the formulation of the output function. Don't care states are applied by setting the position of the chosen terms to 2, while all other positions of  $F_{1t}$  have the same value of  $f_t$ . The number 2 is a random value, any single digit different than 0 or 1 can also be used. The goal is to skip the don't care state term in the coverage Equations 32 and 33. With an assigned value different than 0 or 1, a term  $t$  will neither be considered a minterm ( $f_t = 1$ ) nor a maxterm ( $f_t = 0$ ), being simply ignored in the coverage constraints.

Since the initial value of  $d = 4$  implies that a 3-level function is needed in the output majority function,  $F_{1t}$  will be covered based on the optimization model for 3-level majority functions. If  $F_{1t}$  can not be covered by a 3-level function, new  $2^{n-1}$  terms will be randomly selected as don't care states and a new optimization model will be created. Once  $F_{1t}$  has been covered, the cost  $C_i$  of every primitive used to build  $F_{1t}$  will be set to 0, since the cost of repeated gates is disregarded in the output function.

Every non don't care state term in  $F_{1t}$  will be considered a don't care state in  $F_{2t}$ , all other terms being equal to  $f_t$ . Therefore,  $F_{1t}$  considers half the terms of  $f_t$  as don't care states while  $F_{2t}$  considers the other half. The coverage of  $F_{2t}$  is also done through a 3-level optimization model, along with the updated vector  $C_i$ . Since the cost of the primitives used to cover  $F_{1t}$  are set to 0, the optimization solver will prioritize the use of those primitives in the coverage of  $F_{2t}$ , reducing the overall size cost of the output function.

Next step is to update  $F_{1t}$  and  $F_{2t}$  with the truth table of the newly generated functions. From the updated vectors  $F_{1t}$  and  $F_{2t}$ , a new vector  $H_t$  is created. Vector  $H_t$  stores the number of times that each term  $t$  is covered by all  $F_{st}$  functions. From vector  $H_t$ , it is possible to define all remaining truth tables  $F_{st}$ , where  $s > 2$ . The combined  $F_{st}$  functions must cover all minterms ( $f_t = 1$ ) at least  $\lceil \frac{m}{2} \rceil$  times and, at the same time, can not cover any maxterms ( $f_t = 0$ ) more than  $\lceil \frac{m}{2} \rceil$  times. Therefore, if  $f_t = 1$  and term  $t$  is covered less than  $\lceil \frac{m}{2} \rceil$  times, it must be covered by the next  $F_{st}$  function. Likewise, if  $f_t = 0$  and term  $t$  is covered  $\lceil \frac{m}{2} \rceil - 1$  times, it must not be covered by the next  $F_{st}$  function. For any other case, term  $t$  is considered a don't care state. For  $s > 2$ ,  $F_{st}$  can also be covered by primitives or 2-level functions, so the standard 3MS-Depth sequence

of optimization models is used. The heuristic ends when all  $m$  tables  $F_{st}$  are covered, completing the output majority function.

As an example, consider  $m = 3$ ,  $d = n = 4$  and  $f_t = 0110100110010110$ . This truth table refers to the boolean function  $A \oplus B \oplus C \oplus D$ , one of the only 2 functions for  $n = 4$  (from a total 65,536) that needs a 4-level majority function to be covered. From  $f_t$ , suppose  $F_{1t} = 2112122210010222$  and  $F_{2t} = 0220200122222110$ , where value 2 represents the randomly assigned don't care states. Applying  $F_{1t}$  to the 3-level optimization model generates the majority function  $\overline{M}(0, A, M(C, \overline{M}(B, C, D), M(B, \overline{C}, D)))$ . Applying  $F_{2t}$  to the 3-level optimization model generates the majority function  $M(1, A, M(C, \overline{M}(B, C, D), M(B, \overline{C}, D)))$ . Note that the majority functions share 3 primitives, greatly decreasing the total cost of the output function.

Next step is to update  $F_{1t}$  and  $F_{2t}$  to the truth table of the generated majority functions, resulting in  $F_{1t} = 1111111110010110$  and  $F_{2t} = 0110100111111111$ . Using the vectors  $f_t$ ,  $F_{1t}$  and  $F_{2t}$  as a base, vector  $H_t$  is created. Table 21 shows vector  $H_t$  for the presented example.

**Table 21** – Example of vector  $H_t$ .

$t$	$F_{1t}$	$F_{2t}$	$f_t$	$H_t$
0	1	0	0	1
1	1	1	1	2
2	1	1	1	2
3	1	0	0	1
4	1	1	1	2
5	1	0	0	1
6	1	0	0	1
7	1	1	1	2
8	1	1	1	2
9	0	1	0	1
10	0	1	0	1
11	1	1	1	2
12	0	1	0	1
13	1	1	1	2
14	1	1	1	2
15	0	1	0	1

**Source: Author.**

The positions where  $H_t = 2$  or  $H_t = 0$  represents don't care states for the next



function  $F_{3t}$ , since this positions refers to maxterms that were never covered and minterms that were already covered twice. The positions where  $H_t = 1$  refers to maxterms of  $f_t$  that were already covered once. Therefore, since  $\lceil \frac{m}{2} \rceil = 2$  for  $m = 3$ , function  $F_{3t}$  can not cover any term where  $H_t = 1$ . The resulting truth table would be  $F_{3t} = 0220200222220220$ , where value 2 represents the don't care state terms. The lowest cost function that can effectively cover  $F_{3t}$  is the constant 0. Therefore, the complete output majority function is  $M(0, \overline{M}(0, A, M(C, \overline{M}(B, C, D), M(B, \overline{C}, D))), M(1, A, M(C, \overline{M}(B, C, D), M(B, \overline{C}, D))))$ , with a total cost of 6 gates (disregarding repeated gates), 3 inverters and 9 literals.

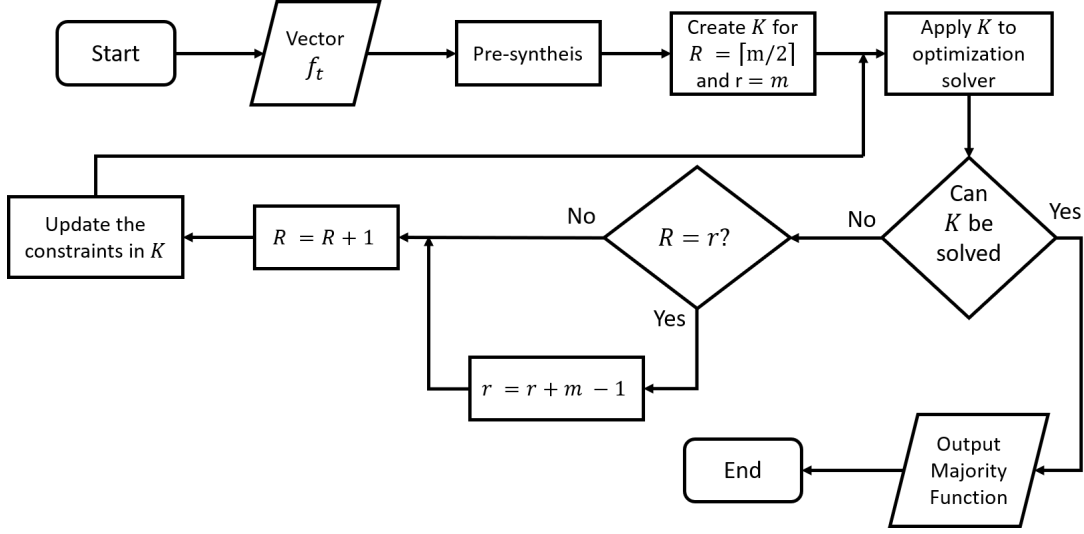
If  $f_t$  can not be covered by the resulting 4-level majority function, the heuristic considers the newly formed 4-level majority function as  $F_{1t}$ , incrementing  $d$  to  $d = 5$ . The terms with different values between  $F_{1t}$  and  $f_t$  are considered don't care states for  $F_{2t}$  and, from  $F_{1t}$  and  $F_{2t}$ , vector  $H_t$  and the  $s > 2$  truth tables are created. It's important to point out that this heuristic was tested only for the coverage of functions with up to 5 levels. Even though, in theory, this heuristic could work for  $d > 5$  if extended to a higher number of iterations, the high computational time required to synthesize functions with more than 5 levels limits the heuristic viability.

### 3.4 3MS-SIZE SYNTHESIS

The 3MS-Size synthesis introduces a new variable  $R$  to the construction of set  $K$ . The variable  $R$  represents the limit of different gates in the output majority function. Since all functions with up to 1 gate are primitives, the initial value of  $R$  is  $R = \lceil \frac{m}{2} \rceil$ , meaning that  $K$  is only solved if the combined selection of primitives covers  $f_t$  and, at the same time, do not have more than  $\lceil \frac{m}{2} \rceil$  different gates. Variable  $r$  has a initial value of  $r = m$ , representing the limit of selected primitives, including repeated gates. Every time  $K$  could not be solved, the variable  $R$  is incremented by 1, up to a maximum of  $r$ , and the constraints in  $K$  are rebuilt. If  $R = r$ , variable  $r$  is incremented to  $r = r + m - 1$ . Figure 8 shows a flowchart for the overall 3MS-Size synthesis.

#### 3.4.1 Size optimization model

The first step in the formulation of  $K$  is the creation of matrix  $X_{ij}$ , representing the assignment of a primitive  $i$  into a maj- $m$  2-level function  $j$  that composes the output function, following the same logic of primitives distribution presented in the 3-level optimization model. Likewise, variable  $j$  ranges from 1 to  $1 + \lfloor \frac{r}{m} \rfloor$ , where  $r$  bounds

**Figure 8** – Flowchart for the overall 3MS-Size synthesis.**Source: Author.**

the number of selected primitives in the coverage of  $f_t$ , as shown in Equation 35.

$$\sum_{i=0}^{p-1} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} = r \quad (35)$$

The 3MS-Size synthesis utilizes the same  $W_i$  vectors as the 3-level optimization model to disregard the cost of repeated gates in the output majority function, as shown in Equations 36 and 37. The main goal of the size optimization model is to combine the same primitives as much as possible, aiming to find a solution with the least possible number of different primitives, represented by  $R$ .

$$\sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \geq 1 \rightarrow W_i = 1 \quad \forall i \in c \quad (36)$$

$$\sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \leq 0 \rightarrow W_i = 0 \quad \forall i \in c \quad (37)$$

Vector  $W_i$  will value to 1 if a primitive  $i$  in set  $c$  (where  $C_i \geq 10,000$ ) is selected at least once to cover  $f_t$ , valuing to 0 otherwise. Therefore, the sum of the elements in  $W_i$  represents the number of different primitives (considering only non literals since literals are not in set  $c$ ) that composes the majority output function. Equation 38 bounds the total of elements in  $W_i$  to  $R$ .

$$\sum_{i \in c} W_i \leq R \quad (38)$$

The objective function, represented by  $Z$ , can be calculated by Equation 39.

$$\min Z = \sum_{i \in c} W_i \cdot C_i + \sum_{i \notin c} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot C_i \quad (39)$$

Next step is the creation of vector  $T_j$ , storing the weight in the coverage of  $f_t$ , for each maj- $m$  function  $j$ . For the functions in the first level of the output function ( $j = 1$ ), the assigned weight is  $T_j = m^3$ . If  $j > 1$  and  $j \leq m$ , meaning that the function  $j$  is in the second level of the output function, the assigned weight will be  $T_j = m^2$ . Likewise,  $m < j \leq m^2$  refers to functions on the third level of the output function and the assigned weight is  $T_j = m$ . Lastly, a function  $j$  is on the fourth level of the output function if  $m^2 < j \leq m^3$  and the assigned weight is  $T_j = 1$ . Note that functions in the higher levels have lower weights in the coverage of the output function. It is important to point out that a higher number of levels is not considered because the computational viability of the size optimization model is limited to the coverage of functions with up to 5 levels.

Based on the matrix  $Y_{it}$ , built in the pre-synthesis phase, the coverage of each term  $t$  of  $f_t$  is shown in Equations 40 and 41, for minterms and maxterms, respectively.

$$\sum_{i=0}^{p-1} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot T_j \cdot Y_{it} > \lfloor \frac{m}{2} \rfloor \cdot m^3 \quad \forall f_t = 1 \mid 0 \leq t < 2^n \quad (40)$$

$$\sum_{i=0}^{p-1} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot T_j \cdot Y_{it} \leq \lfloor \frac{m}{2} \rfloor \cdot m^3 \quad \forall f_t = 0 \mid 0 \leq t < 2^n \quad (41)$$

The bound  $\lfloor \frac{m}{2} \rfloor \cdot m^3$  represents the coverage by the majority of the selected primitives, considering the weights defined by the vector  $T_j$ . The complete size optimization model is shown in Equation 42. If  $f_t$  could not be covered, variable  $R$  is incremented and the algorithm builds a new optimization model considering an output majority function with greater size.

$$\min Z = \sum_{i \in c} W_i \cdot C_i + \sum_{i \notin c} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot C_i$$

s.t.

$$\sum_{i=0}^{p-1} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} = r$$

$$X_{ij} \geq 0$$

$$\sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \geq 1 \rightarrow W_i = 1 \quad \forall i \in c$$

$$\sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \leq 0 \rightarrow W_i = 0 \quad \forall i \in c$$

$$\sum_{i \in c} W_i \leq R$$

$$\sum_{i=0}^{p-1} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot T_j \cdot Y_{it} > \left\lfloor \frac{m}{2} \right\rfloor \cdot m^3 \quad \forall f_t = 1 \mid 0 \leq t < 2^n$$

$$\sum_{i=0}^{p-1} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot T_j \cdot Y_{it} \leq \left\lfloor \frac{m}{2} \right\rfloor \cdot m^3 \quad \forall f_t = 0 \mid 0 \leq t < 2^n$$

(42)

## 4 COMPLEXITY OF ALGORITHMS

This chapter presents an introduction about complexity of algorithms, including the concept of orders of magnitude and the complexity calculation for the optimization models in both 3MS-Depth and 3MS-Size synthesis. The study of computational complexity can be defined as the study of the computational effort needed to solve a determined problem through an algorithm. The performance of an algorithm is measured by how the size of its input, represented by  $N$ , impacts the overall runtime, whereas runtime is measured by the number of operations to be performed by the algorithm (AGARWAL; BAKA, 2018). Since the runtime of an algorithm depends on the input size, as  $N$  increases, the runtime also increases. For example, a sorting algorithm will need more time to sort a list of 1,000 elements as compared to the time needed to sort a list of 100 elements. Therefore, the runtime depends on the more complex operation to be executed. In a sorting algorithm, the operation to compare elements have a higher complexity and thus will take more time to execute, time that will be increased based on the value of  $N$ .

Orders of magnitude, represented by  $O$ , are the mathematical relationship between the input size  $N$  and the necessary time for the algorithm to execute. There are 3 possible ways to characterize an order of magnitude: The worst-case scenario considers the upper-bound complexity, resulting in the maximum runtime required for the algorithm to run. In this case, the higher complexity operations should be executed the maximum number of times. Best-case scenario is the opposite, being the lower-bound complexity where the minimum runtime is considered. Lastly, the average-case scenario considers the average runtime required for an algorithm to execute (ASCENCIO; ARAÚJO, 2010).

With an worst-case scenario approach, its possible to ignore variables and operations that, at large values of  $N$ , do not impact the overall complexity of the algorithm. This makes the complexity calculation mathematically easier, while also allowing to focus on more relevant operations. Therefore, worst-case scenario is the most useful and the most used analysis to generate complexity orders of magnitude (DEDOV, 2020). In this work, the complexity of the 3MS algorithm is calculated based on worst-case scenarios.

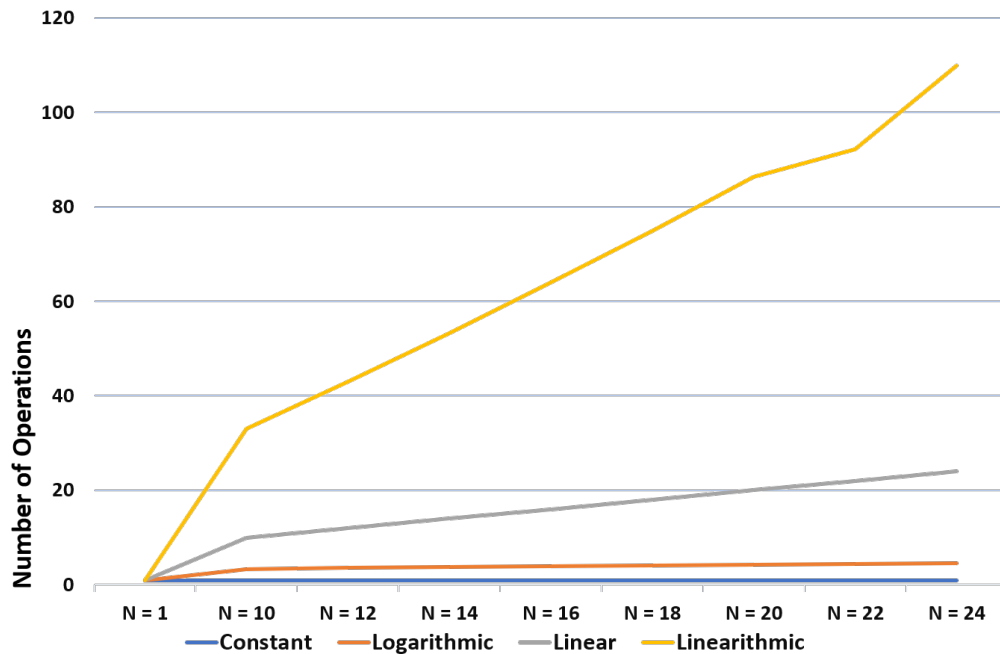
The main possible complexity orders are shown in Table 22. Figure 9 presents a complexity comparison among the constant, logarithmic, linear and linearithmic orders of magnitude, aiming to exemplify the increase in the number of operations for each order. In this example, we consider logarithmic base 2 for  $O(\log(N))$  and  $O(N \cdot \log(N))$ . Figure 10 presents a complexity comparison among the remaining, and more complex, orders of magnitude: quadratic, cubic and exponential.

**Table 22** – Main orders of magnitude.

Order of Magnitude	Complexity
$O(1)$	Constant
$O(\log(N))$	Logarithmic
$O(N)$	Linear
$O(N \cdot \log(N))$	Linearithmic
$O(N^2)$	Quadratic
$O(N^3)$	Cubic
$O(2^N)$	Exponential

Source: Author.

**Figure 9** – Comparison among constant, logarithmic, linear and linearithmic orders.

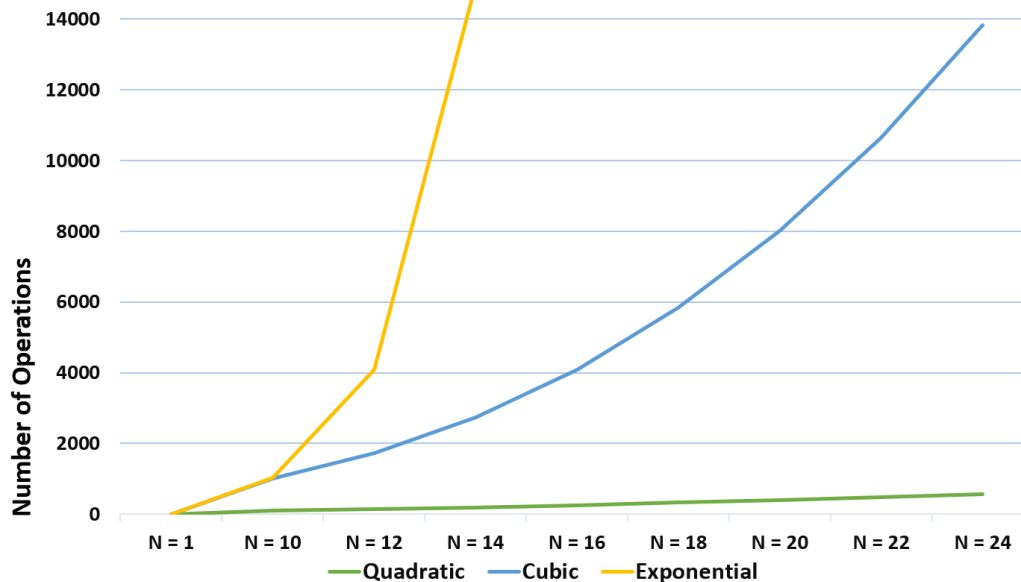


Source: Author

According to Wilf (2019), an algorithm is considered efficient if composed mostly by polynomial functions. Therefore, to be considered efficient, an algorithm must have a complexity order of at most  $O(N^3)$ . Exponential orders of magnitude,  $O(2^N)$ , are only

considered efficient if the algorithm can not be further simplified.

**Figure 10** – Comparison among quadratic, cubic and exponential orders.



Source: Author

The constant order of magnitude, represented by  $O(1)$ , means that the number of operations of an algorithm is not related to the size of  $N$ . An example of  $O(1)$  would be an algorithm that receives a number as input and prints it on the screen. In this example, independently of the input, the number of operations in the algorithm would always be the same: one operation to read the number and another to print it. Figure 11 shows this example coded in Python language.

**Figure 11** – Example of  $O(1)$ .

```
1 #Constant Complexity
2 number = input("Type a number:")
3 print(number)
```

Source: Author

An algorithm has a logarithmic order of magnitude,  $O(\log(N))$ , if each iteration divides the input, performing the inverse operation of a exponentiation. Binary search is a classic example of logarithmic complexity. In a binary search, the goal is to find a specific item in an already sorted sequence by dividing the input in half with each iteration. Therefore, binary search algorithms have a logarithmic complexity of base 2,  $O(\log_2(N))$ . Figure 12 presents the code of a binary search algorithm.

**Figure 12** – Example of  $O(\log(N))$ .

```

1  #Logarithmic Complexity
2  def binary_search(sequence, item):
3      start_index = 0
4      end_index = len(sequence) - 1
5
6      while start_index <= end_index:
7          mid_index = start_index + (end_index - start_index) // 2
8          mid_value = sequence[mid_index]
9          if mid_value == item:
10             return mid_value
11         elif item < mid_value:
12             end_index = mid_index - 1
13         else:
14             start_index = mid_index + 1

```

**Source:** Author

The linear order of magnitude  $O(N)$  means that the number of operations in an algorithm is linearly dependant of its input size. An example of linear complexity is an algorithm that receives a number as input and prints all numbers from 0 up to that number. This algorithm is formed by a repetition loop that performs the print operation  $N$  times, where  $N$  is the input number. Figure 13 shows an example of this algorithm code.

**Figure 13** – Example of  $O(N)$ .

```

1  #Linear Complexity
2  number = input("Type a number:")
3  for i in range(0,number):
4      print(i)

```

**Source:** Author

Note that the operation with higher complexity will define the overall complexity of the algorithm. Even through the input operation on line 2 has a constant complexity  $O(1)$ , in large values of  $N$  this constant loses relevance when compared to the linear complexity  $O(N)$  of the repetition loop. If  $N = 10,000$ , for instance, the repetition loop would be responsible for 10,000 of the total 10,001 operations realized by the algorithm.

Similar to logarithmic complexity, algorithms with linearithmic complexy,  $O(N \cdot \log(N))$ , also utilize a sequence of divisions to travel the search space. The key difference is that in a linearithmic algorithm both halves of the input are retained after each division. Quick sorts are a classic example of linearithmic algorithms. In a quick sort algorithm, the goal is to sort a sequence of numbers by selecting a random number, called pivot, that will be used as parameter. The correct index for the pivot is calculated by dividing the other



numbers in the sequence into 2 subsets, one receiving values lower than the pivot and the other receiving the numbers with a higher value. This logic will be applied to every element in the sequence. Therefore, linearithmic complexity means that an operation of logarithmic complexity will be realized a linear number of times. Figure 14 shows the code of a quick sort function.

**Figure 14** – Example of  $O(N \cdot \log(N))$ .

```

1  #Linearithmic Complexity
2  def quick_sort(sequence):
3      size = len(sequence)
4      if size <= 1:
5          return sequence
6      else:
7          pivot = sequence.pop()
8
9      items_greater = []
10     items_lower = []
11
12     for item in sequence:
13         if item > pivot:
14             items_greater.append(item)
15         else:
16             items_lower.append(item)
17     return quick_sort(items_lower) + [pivot] + quick_sort(items_greater)

```

Source: Author

Quadratic complexity,  $O(N^2)$ , represents an algorithm whose performance is directly proportional to the squared size of  $N$ . An example of  $O(N^2)$  complexity is an algorithm that prints all elements of an  $N \times N$  matrix. This algorithm can be programmed with 2 repetition loops, as shown in Figure 15.

**Figure 15** – Example of  $O(N^2)$ .

```

1  #Quadratic Complexity
2  def print_matrix(matrix, size):
3      for i in range(0, size):
4          for j in range(0, size):
5              print(matrix[i,j])

```

Source: Author

Likewise, cubic complexity represents an algorithm where the number of operations is proportional to the cube value of the input size. An example of  $O(N^3)$  complexity is an algorithm that receives 3 vectors of  $N$  elements and has to print all 3-to-3 combinations of those vectors. Figure 16 shows this example in Python code.

**Figure 16** – Example of  $O(N^3)$ .

```

1  #Cubic Complexity
2  def print_3to3(vector1, vector2, vector3, size):
3      for i in range(0, size):
4          for j in range(0, size):
5              for k in range(0, size):
6                  print(vector1[i], vector2[j], vector3[k])

```

**Source:** Author

A classic example of exponential complexity,  $O(2^N)$ , is the Fibonacci algorithm. Exponential complexity refers to algorithms where the number of operations doubles with each increase to the input size  $N$ . Figure 17 shows the recursive programming of a Fibonacci function. Note that for each iteration the function is called 2 more times until  $N = 1$  or  $N = 0$ , generating an exponential growth in the number of operations.

**Figure 17** – Example of  $O(2^N)$ .

```

1  #Exponential Complexity
2  def fibonacci(N):
3      if (N == 1):
4          return 1
5      elif (N == 0):
6          return 0
7      return fibonacci(N-1) + fibonacci(N-2)

```

**Source:** Author

#### 4.1 COMPLEXITY OF THE 3MS ALGORITHM

This section presents the complexity calculation for both 3MS-Depth and 3MS-Size synthesis, the complexity order of each optimization model is addressed and explained individually. The input data received by the 3MS algorithm are the variable  $m$  and the binary vector  $f_t$ . The variable  $m$  represents the number of majority gate inputs considered to generate the output function. Vector  $f_t$  represents a truth table of  $2^n$  terms, where  $n$  refers to the number of input variables. Based on  $n$ ,  $m$  and  $f_t$ , the pre-synthesis of the 3MS algorithm, described in section 3.2, builds the primitives table and stores the necessary information to build all optimization constraints. Both 3MS-Depth and 3MS-Size synthesis considers the complete table of primitives to build its constraints and the size of each constraint is mainly dependant of the amount of primitive functions, represented by  $p$ . Therefore, since the number of primitives  $p$  has the higher impact in every 3MS

optimization model, we consider  $N = p$  to calculate the complexity orders.

It is also important to point out that only the complexity of the constraints formulation is being considered. The complexity of the solution provided by the optimization solvers is not being considered, since the solution will vary based not only for the optimization solver but also for each optimization model created.

#### 4.1.1 Complexity of the 2-level optimization model

The complexity of an optimization model is given by its highest complexity constraints. In this section we explain the individual programming complexity for each constraint that composes the 2-level optimization model and then combine these complexities to form the overall complexity of the optimization model. Equation 16 bounds the sum of elements in  $X_i$  to  $m$ . The number of elements in the sum will increase linearly based on the value of  $p$ , resulting in the linear complexity  $O(N)$ .

Equations 17 and 18, that simulates the  $\Omega.I$  axiom, can be programmed by using a loop of  $p$  iterations. Since set  $e$  does not include the constants 0 and 1, the initial index is  $i = 2$ . Each iteration is composed by at least 1 operation to verify if a primitive  $i$  is in set  $e$  or in set  $k$ . The attributions  $S1 = \lfloor \frac{m}{2} \rfloor$  or  $S2 = \lfloor \frac{m}{2} \rfloor$  are done once after the loop, thus having a constant complexity. Therefore, the constraint defined by Equation 17 has a total complexity of  $O(N - 2) + O(1)$ . For Equation 18, set  $k$  includes all  $p$  primitives, resulting in the complexity  $O(N) + O(1)$ . Both constraints can be simplified to  $O(N)$ .

The objective function  $Z$ , shown in Equation 19, can be divided into 2 operations. The operation  $\sum_{i=0}^{p-1} X_i \cdot C_i$  has a linear complexity, since the multiplication is done once for each primitive  $i$ , totalizing  $p$  multiplications. The operation  $-100 \cdot (S1 + S2)$  is done a single time independently of the value of  $p$ , having a constant complexity. Therefore, the complexity of the objective function is given by  $O(N) + O(1)$ , simplified to  $O(N)$ .

The total number of multiplications in the coverage constraints, represented by Equations 20 and 21, is given by  $p \cdot 2^n$ , since the operation  $\sum_{i=0}^{p-1} X_i \cdot Y_{it}$  implies one sum of  $p$  multiplications for each term  $t$  of  $f_t$ , where  $0 \leq t < 2^n$ . Additionally, each term  $t$  has an operation to verify if  $t$  is a minterm or a maxterm and another to verify the condition which its corresponding sum is bound to. Therefore, the complexity of each coverage constraint is defined by a repetition loop of  $2^n$  iterations and  $p + 2$  operations in each iteration, resulting in the complexity order  $2^n \cdot O(N + 2)$ .

Table 23 shows each constraint and its respective complexity order. The complete

complexity of the 2-level optimization model is given by the sum  $2 \cdot (2^n \cdot O(N+2)) + O(N-2) + 3 \cdot O(N) + 3 \cdot O(1)$ . Based on the worst case scenario approach, that considers the highest order of magnitude as the overall complexity, the 2-level optimization complexity can be further simplified to  $O(N)$ . Therefore, considering  $N = p$ , the 3MS-Depth optimization model for 2-level majority functions has a linear complexity.

**Table 23** – Constraint complexity orders for the 2-level optimization model.

Constraints	Order of Magnitude
$\sum_{i=0}^{p-1} X_i = m$	$O(N)$
$\sum_{i \in e} X_i = m \rightarrow S1 = \lfloor \frac{m}{2} \rfloor$	$O(N-2) + O(1)$
$\sum_{i \in k} X_i = m \rightarrow S2 = \lfloor \frac{m}{2} \rfloor$	$O(N) + O(1)$
$\min Z = \sum_i X_i \cdot C_i - 100 \cdot (S1 + S2)$	$O(N) + O(1)$
$\sum_{i=0}^{p-1} X_i \cdot Y_{it} > \lfloor \frac{m}{2} \rfloor \quad \forall f_t = 1 \mid 0 \leq t < 2^n$	$2^n \cdot O(N+2)$
$\sum_{i=0}^{p-1} X_i \cdot Y_{it} \leq \lfloor \frac{m}{2} \rfloor \quad \forall f_t = 0 \mid 0 \leq t < 2^n$	$2^n \cdot O(N+2)$

Source: Author.

#### 4.1.2 Complexity of the 3-level optimization model

The first constraints of the 3-level optimization model bounds the values of  $X_{ij}$ . Equation 23 bounds the value of  $X_{ij}$  for all  $j > 1$ . The total of elements in the constraint is defined by  $p \cdot \lfloor \frac{r}{m} \rfloor$ , representing  $\lfloor \frac{r}{m} \rfloor$  sums of  $p$  elements. Therefore, the complexity can be calculated by  $O(N) \cdot \lfloor \frac{r}{m} \rfloor$  and, since  $\lfloor \frac{r}{m} \rfloor$  is a fixed value in the model and not a variable, it can be simplified to  $O(N)$ . Equation 24 bounds the value  $X_{i1}$  for  $j = 1$ , being a single sum of  $p$  elements and thus also having a linear complexity.

The complexity of the  $\Omega.I$  constraints for  $j > 1$ , defined by Equations 25 and 26, can be calculated by considering the same complexity applied to the 2-level optimization model, multiplied by  $\lfloor \frac{r}{m} \rfloor$ , since the logic is applied for every  $j > 1$  and  $1 \leq j \leq 1 + \lfloor \frac{r}{m} \rfloor$ . Therefore, Equation 25 have a total complexity of  $\lfloor \frac{r}{m} \rfloor \cdot (O(N-2) + O(1))$ . Likewise, the total complexity of Equation 26 is given by  $\lfloor \frac{r}{m} \rfloor \cdot (O(N) + O(1))$ . Both complexities can be simplified to  $O(N)$ , considering the worst case scenario approach.

The remaining  $\Omega.I$  constraints, represented by Equations 27 and 28, that defines the values of  $S1_1$  and  $S2_1$ , have a individual complexity order of  $O(N) + (2 + \lfloor \frac{r}{m} \rfloor) + O(1)$ . This complexity can be calculated by considering the  $p$  operations of  $X_{i1}$ , to verify if the primitive  $i$  is in set  $e$  or set  $k$ , followed by the sum of all values in  $S1_j$  and  $S2_j$ . Lastly, an operation to assign the correct values to  $S1_1$  or  $S2_1$  is also considered. Table 24 shows the equation and the individual complexity of the constraints for  $X_{ij}$  bounding and  $\Omega.I$

application.

**Table 24** – Complexity orders of  $X_{ij}$  bounding and  $\Omega$ .I application, for the 3-level optimization model.

Constraints	Order of Magnitude
$\sum_{i=0}^{p-1} X_{ij} = m \ \forall j > 1$	$O(N) \cdot \lfloor \frac{r}{m} \rfloor$
$\sum_{i=0}^{p-1} X_{i1} = r - m \cdot \lfloor \frac{r}{m} \rfloor$	$O(N)$
$\sum_{i \in e} X_{ij} = m \rightarrow S1_j = \lfloor \frac{m}{2} \rfloor \ \forall j > 1$	$\lfloor \frac{r}{m} \rfloor \cdot (O(N - 2) + O(1))$
$\sum_{i \in k} X_{ij} = m \rightarrow S2_j = \lfloor \frac{m}{2} \rfloor \ \forall j > 1$	$\lfloor \frac{r}{m} \rfloor \cdot (O(N) + O(1))$
$\sum_{i \in e} X_{i1} + \sum_j S1_j = m \rightarrow S1_1 = \lfloor \frac{m}{2} \rfloor$	$O(N) + (2 + \lfloor \frac{r}{m} \rfloor) + O(1)$
$\sum_{i \in k} X_{i1} + \sum_j S2_j = m \rightarrow S2_1 = \lfloor \frac{m}{2} \rfloor$	$O(N) + (2 + \lfloor \frac{r}{m} \rfloor) + O(1)$

**Source: Author.**

Equations 29 and 30, responsible for the assignment values of  $W_i$ , share the same complexity and both can be build using similar calculations. For each primitive  $i$  the algorithm sums all  $1 + \lfloor \frac{r}{m} \rfloor$  elements in  $X_{ij}$  (since the maximum value of  $j$  is  $1 + \lfloor \frac{r}{m} \rfloor$ ) and, after the sum, verifies the condition of each constraint and assigns the value of  $W_i$  accordingly, totalizing 2 operations for each  $i$ . Therefore, the complexity of both constraints is given by  $(3 + \lfloor \frac{r}{m} \rfloor) \cdot O(N)$ .

the by Equation 31, can be divided into 3 calculations. The first calculation consists in multiplying vector  $W_i$  with its respective cost  $C_i$ , totalizing at least  $p$  operations to verify if each  $i$  pertains to  $c$ . Therefore, for  $\sum_{i \in c} W_i \cdot C_i$ , we have linear complexity of  $O(N)$ . The complexity of the second calculation,  $\sum_{i \notin c} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot C_i$ , is defined by the sum of  $p \cdot (1 + \lfloor \frac{r}{m} \rfloor)$  multiplications, since  $0 \leq i < p - 1$  and  $1 \leq j \leq 1 + \lfloor \frac{r}{m} \rfloor$ . A second set of  $p$  multiplications is also needed to multiply  $C_i$ , along with a constant operation in every multiplication to verify if the primitive  $i$  does not pertains to  $c$ . The complexity for this calculation is  $O(N) \cdot (\lfloor \frac{r}{m} \rfloor + 2)$ . The last calculation,  $-100 \cdot (\sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} S1_j + \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} S2_j)$ , is composed by  $2 \cdot (1 + \lfloor \frac{r}{m} \rfloor)$  calculations that does not depend on the value of  $N = p$ , being of complexity  $2 \cdot \lfloor \frac{r}{m} \rfloor \cdot O(1)$ . The complete complexity order for the objective function is  $O(N) \cdot (\lfloor \frac{r}{m} \rfloor + 3) + 2 \cdot \lfloor \frac{r}{m} \rfloor \cdot O(1)$ .

The coverage constraints, represented by Equations 32 and 33, can be divided into 2 base calculations. The first calculation refers to  $\sum_{i=0}^{p-1} X_{i1} \cdot Y_{it}$ , which generates the complexity  $2^n \cdot O(N+2)$ , following the same logic as the coverage of the 2-level optimization model. The second calculation refers to the application of the coverage logic to every function where  $j > 1$ . Additionally, the weight calculation is considered as an operation of constant complexity for each  $j > 1$ . Therefore, the total complexity for each coverage constraint is  $(2^n \cdot O(N + 2)) \cdot (1 + \lfloor \frac{r}{m} \rfloor) + (O(1) \cdot \lfloor \frac{r}{m} \rfloor)$ .

Table 25 shows the equation and the individual complexity of the remaining constraints for the 3-level optimization model. Based on the worst case scenario approach, since  $O(N)$  is the highest order of magnitude, the 3MS-Depth optimization model for 3-level majority functions also has a linear complexity.

**Table 25** – Complexity for the 3-level optimization model remaining constraints.

Constraints	Order of Magnitude
$\sum_j X_{ij} \geq 1 \rightarrow W_i = 1 \forall i \in c$	$(3 + \lfloor \frac{r}{m} \rfloor) \cdot O(N)$
$\sum_j X_{ij} = 0 \rightarrow W_i = 0 \forall i \in c$	$(3 + \lfloor \frac{r}{m} \rfloor) \cdot O(N)$
$\min Z = \sum_{i \in c} W_i \cdot C_i + \sum_{i \notin c} \sum_j X_{ij} \cdot C_i - 100 \cdot (\sum_j S1_j + \sum_j S2_j)$	$O(N) \cdot (\lfloor \frac{r}{m} \rfloor + 3) + 2 \cdot \lfloor \frac{r}{m} \rfloor \cdot O(1)$
$\sum_{i=0}^{p-1} X_{i1} \cdot Y_{it} + \frac{\sum_i \sum_j X_{ij} \cdot Y_{it}}{\lfloor \frac{m}{3} \cdot 2 \rfloor} > \lfloor \frac{m}{2} \rfloor \forall f_t = 1 \mid 0 \leq t < 2^n$	$(2^n \cdot O(N + 2)) \cdot (1 + \lfloor \frac{r}{m} \rfloor) + (O(1) \cdot \lfloor \frac{r}{m} \rfloor)$
$\sum_{i=0}^{p-1} X_{i1} \cdot Y_{it} + \frac{\sum_i \sum_j X_{ij} \cdot Y_{it}}{\lfloor \frac{m}{3} \cdot 2 \rfloor} \leq \lfloor \frac{m}{2} \rfloor \forall f_t = 0 \mid 0 \leq t < 2^n$	$(2^n \cdot O(N + 2)) \cdot (1 + \lfloor \frac{r}{m} \rfloor) + (O(1) \cdot \lfloor \frac{r}{m} \rfloor)$

**Source: Author.**

#### 4.1.3 Complexity of the size optimization model

This section presents the complexity calculation when the reduction of size is being prioritized. Since the size optimization model utilizes several constraints also used by the 3-level optimization model, which can all be simplified to a linear complexity  $O(N)$ , in this section we will address only the constraints that are exclusive to the size optimization model. The first constraint, represented by Equation 35, bounds the sum of  $X_{ij}$  to  $r$ . The total of elements in the constraint is defined by  $p \cdot (1 + \lfloor \frac{r}{m} \rfloor)$ , representing  $(1 + \lfloor \frac{r}{m} \rfloor)$  sums of  $p$  elements. Therefore, the complexity can be calculated by  $O(N) \cdot (1 + \lfloor \frac{r}{m} \rfloor)$ , being simplified to  $O(N)$ .

Equations 38 limits the sum of elements in  $W_i$  to the upper bound  $R$ . Since at least 1 iteration for each primitive  $i$  is needed to verify if the primitive pertains to set  $c$ , the number of iterations can be given by  $p$ . Therefore, Equation 38 have a linear complexity  $O(N)$ .

The objective function  $Z$ , shown in Equation 39, is formed by the  $\sum_{i=0}^{p-1} X_i \cdot C_i$ , having linear complexity since the multiplication is done once for each primitive  $i$ , totalizing  $p$  multiplications.

The coverage constraints in the size optimization model, represented by Equations 40 and 41, are composed by  $p \cdot (1 + \lfloor \frac{r}{m} \rfloor)$  calculations, since  $0 \leq i < p - 1$  and  $1 \leq j$ , multiplied by  $j$  (application of weight  $T_j$  for each function  $j$ ) and  $2^n$  (since a constraint is created for each term  $t$ ), respectively. Additionally, the weight calculation is considered

as an operation of constant complexity for each  $j$ . Therefore, the coverage constraints have an individual complexity of  $2^n \cdot 2 \cdot (1 + \lfloor \frac{r}{m} \rfloor) \cdot O(N) + (1 + \lfloor \frac{r}{m} \rfloor) \cdot O(1)$ , which can be simplified to  $2^n \cdot 2 \cdot (1 + \lfloor \frac{r}{m} \rfloor) \cdot O(N)$  and then further simplified to  $O(N)$ . Table 26 presents the complexity orders of the size optimization model.

**Table 26** – Constraint complexity orders for the size optimization model.

Constraints	Order of Magnitude
$\sum_{i=0}^{p-1} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} = r$	$O(N) \cdot \lfloor \frac{r}{m} \rfloor$
$\sum_{i \in c} W_i \leq R$	$O(N)$
$\min Z = \sum_{i \in c} W_i \cdot C_i + \sum_{i \notin c} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot C_i$	$O(N)$
$\sum_{i=0}^{p-1} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot T_j \cdot Y_{it} > \lfloor \frac{m}{2} \rfloor \cdot m^3 \forall f_t = 1 \mid 0 \leq t < 2^n$	$2^n \cdot 2 \cdot (1 + \lfloor \frac{r}{m} \rfloor) \cdot O(N)$
$\sum_{i=0}^{p-1} \sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \cdot T_j \cdot Y_{it} \leq \lfloor \frac{m}{2} \rfloor \cdot m^3 \forall f_t = 0 \mid 0 \leq t < 2^n$	$2^n \cdot 2 \cdot (1 + \lfloor \frac{r}{m} \rfloor) \cdot O(N)$

**Source: Author.**

## 5 TESTS AND RESULTS

This chapter presents the results obtained from the comparisons between the 3MS algorithm and other exact synthesis algorithms for maj-3 and maj-5 functions. The `exact_mig` algorithm (SOEKEN *et al.*, 2017), used for maj-3 synthesis ( $m = 3$ ), is viable for functions with up to 6 input variables, while the `exact_m5ig` algorithm (CHU *et al.*, 2019), used for maj-5 synthesis ( $m = 5$ ), is viable for functions with up to 4 input variables. Both algorithms are exact when optimizing size, but only the `exact_mig` can also optimize depth as primary cost criteria. Currently there are no algorithms other than the 3MS that can optimize depth for maj-5 functions.

The 3MS algorithm can optimize inverters and literals, without losing optimal minimization for depth and size. Since 2 additional cost criteria are considered, the goal of the algorithm is to further reduce the production cost of technologies where inverters and gate inputs are expensive to build. Considering  $m = 3$ , the 3MS algorithm is viable for functions with up to 8 input variables. Considering  $m = 5$ , the 3MS can synthesize functions with a maximum of 5 input variables. For both  $m = 3$  and  $m = 5$ , the 3MS algorithm can synthesize functions with a higher number of input variables when compared to the `exact_mig` and `exact_m5ig`.

The 3MS algorithm can be divided into 2 synthesis, the 3MS-Depth, prioritizing depth over size, and the 3MS-Size, prioritizing size over depth. Both synthesis also considers the number of inverters and literals as third and fourth cost criteria. From the possibilities of prioritization, 4 cost criteria were selected. Criteria D3 considers  $m = 3$  and refers to the prioritization of depth, followed by size, inverters and literals. Criteria D5 follows the same sequence of prioritization, but considering  $m = 5$  instead. Similarly, criteria S3 considers  $m = 3$  and prioritizes the minimization of size, followed by depth, inverters and literals. Criteria S5 considers the same sequence for  $m = 5$ . All tests were made using a computer with a 16 GB (gigabytes) RAM memory and a 2.2 GHZ CPU. The 3MS algorithm was implemented in Python, using Gurobi (Gurobi Optimization LLC, 2021) and CPLEX (IBM CPLEX LOG, 2021) optimizers to solve the optimization models. Both



optimizers were implemented through their respective API (Application Programming Interface). Additionally, both optimizers were able to generate optimal results for all tested functions.

For  $n = 3$  and  $n = 4$ , the 3 algorithms were executed for all possible functions, totalizing 256 and 65,536 functions, respectively. Table 27 shows the cost comparison between the 3MS-Depth synthesis and the `exact_mig` for criteria D3.

**Table 27** – Cost comparison for  $n = 3$  and  $n = 4$ , considering criteria D3.

	$n = 3$	$n = 4$
<b>3MS-Depth &lt; exact_mig</b>	155	51,799
<b>3MS-Depth = exact_mig</b>	101	13,737
<b>Total Functions</b>	256	65,536

Source: Author

For  $n = 3$ , the 3MS-Depth was able to further reduce the cost of 155 (61%) functions and achieved equivalent results for the remaining 101 (39%). For  $n = 4$ , the 3MS-Depth achieved a lower cost for 51,799 (79%) and an equal cost for 13,737 (21%) functions. Table 28 shows the comparison between the 3MS-Size synthesis and the `exact_mig` for criteria S3, also considering  $n = 3$  and  $n = 4$ .

**Table 28** – Cost comparison for  $n = 3$  and  $n = 4$ , considering criteria S3.

	$n = 3$	$n = 4$
<b>3MS-Size &lt; exact_mig</b>	143	33,657
<b>3MS-Size = exact_mig</b>	113	31,879
<b>Total Functions</b>	256	65,536

Source: Author

For  $n = 3$ , the 3MS-Size achieved better results for 143 (56%) functions and equal cost results for 113 (44%). For  $n = 4$ , the 3MS-Size was able to further improve 33,657 (51%) and generated equal cost results for the remaining 31,879 (49%). Table 29 shows the comparison of the 3MS-Size synthesis and the `exact_m5ig`, considering criteria S5,  $n = 3$  and  $n = 4$ .

**Table 29** – Cost comparison for  $n = 3$  and  $n = 4$ , considering criteria S5.

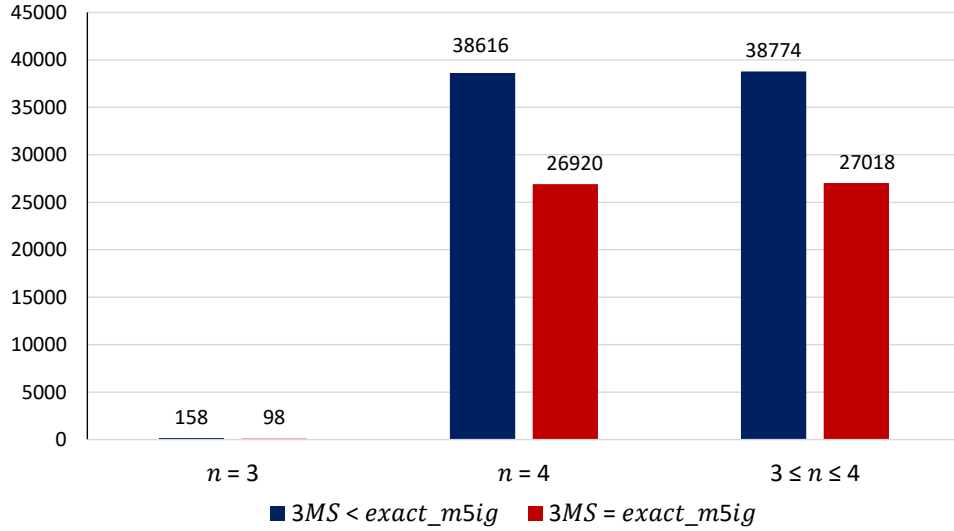
	$n = 3$	$n = 4$
<b>3MS-Size &lt; exact_m5ig</b>	158	38,616
<b>3MS-Size = exact_m5ig</b>	98	26,920
<b>Total Functions</b>	256	65,536

Source: Author

For  $n = 3$ , the 3MS-Size was able to achieve a lower cost for 158 (62%) and an equal

cost for 98 (38%) functions. For  $n = 4$ , the 3MS-Size was able to improve 38,616 (59%) functions and achieved equal cost for the remaining 26,920 (41%). Therefore, considering  $3 \leq n \leq 4$  and criteria S5, the 3MS-Size was able to further reduce the cost of 38,774 (59%) functions out of a total of 65,792 compared functions. Figure 18 shows the comparison between the 3MS-Size and the exact\_m5ig in graph format.

**Figure 18** – Cost comparison (by number of improved and equal functions) between the 3MS-Size and the exact\_m5ig, for criteria S5.



Source: Author

Since the total number of possible functions is calculated by  $2^{2^n}$ , comparing the algorithms for all functions where  $n > 4$  is impracticable. Considering  $n = 5$ , for instance, we have  $2^{32} = 4,294,967,296$  possible functions. For that reason, samples of 10,000 and 1,500 functions were generated for  $n = 5$  and  $n = 6$ , respectively. Table 30 shows the comparison between the 3MS-Depth synthesis and the exact\_mig for criteria D3.

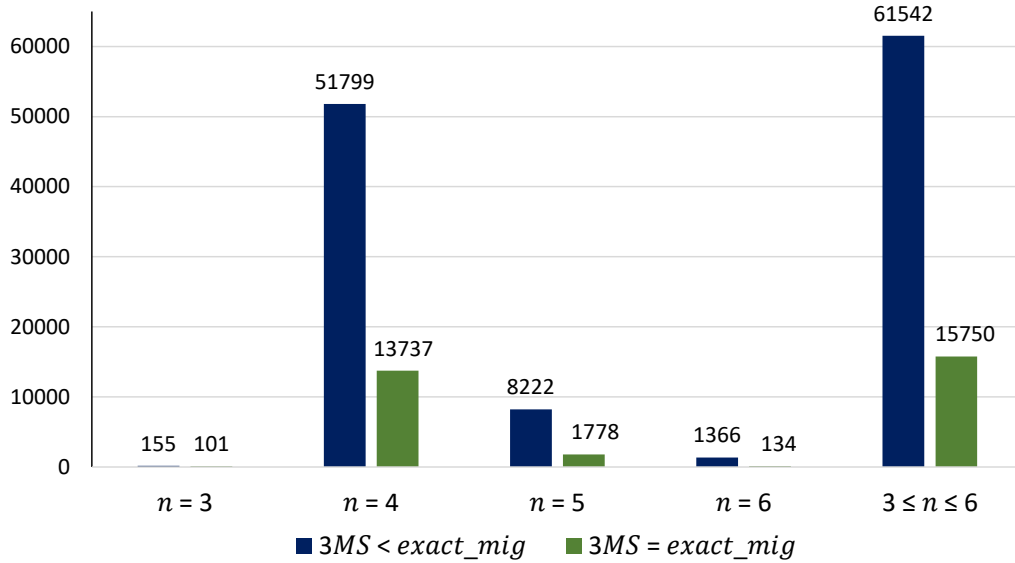
**Table 30** – Cost comparison for  $n = 5$  and  $n = 6$ , considering criteria D3.

	$n = 5$	$n = 6$
3MS-Depth < exact_mig	8,222	1,366
3MS-Depth = exact_mig	1,778	134
Total Functions	10,000	1,500

Source: Author

Considering  $n = 5$  and  $n = 6$ , the 3MS-Depth was able to further reduce the cost of 8,222 (82%) and 1,366 (91%) functions, respectively, and generated equivalent results for 1,778 (18%) and 134 (9%). Figure 19 shows the cost comparison between the 3MS-Depth and the exact\_mig for criteria D3 and  $3 \leq n \leq 6$ . Note that the 3MS-Depth was able to further reduce the cost of 61,542 (79%) functions out of a total of 77,292 functions.

**Figure 19** – Cost comparison (by number of improved and equal functions) between the 3MS-Depth and the exact\_mig, for criteria D3.



Source: Author

Table 31 shows the comparison between the 3MS-Size synthesis and the exact\_mig for criteria S3, for both  $n = 5$  and  $n = 6$ .

**Table 31** – Cost comparison for  $n = 5$  and  $n = 6$ , considering criteria S3.

	$n = 5$	$n = 6$
3MS-Size < exact_mig	7,407	1,208
3MS-Size = exact_mig	2,593	292
<b>Total Functions</b>	10,000	1,500

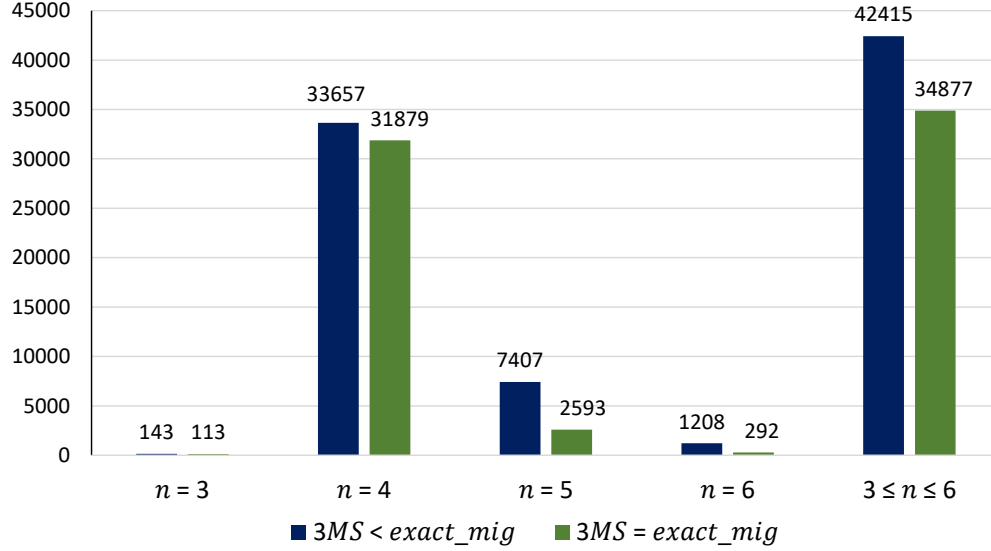
Source: Author

For  $n = 5$ , the 3MS-Size was able to further improve 7,407 (74%) of the compared functions, achieving equal cost results for the remaining 2,593 (26%). For  $n = 6$ , the number of improved and equal cost functions was, respectively, 1,208 (81%) and 292 (19%). Figure 20 shows the cost comparison between the 3MS-Size and the exact\_mig for criteria S3 and  $3 \leq n \leq 6$ . Note that, from a total of 77,292 compared functions, the 3MS-Size was able to improve the cost of 42,415 (55%) functions.

Therefore, considering all comparisons regarding the 3 cost criteria variations D3, S3 and S5, the 3MS algorithm was able to generate lower cost results for 142,731 (64%) of all 220,376 compared functions. It is important to point out that comparisons using the cost criteria D5 were not presented because, to our knowledge, there are no other synthesis than the 3MS-Depth that can prioritize depth as primary cost criteria for maj-5 functions. Additionally, all 220,376 compared functions could be covered using at maximum 5 levels

in the output function, for both  $m = 3$  and  $m = 5$ .

**Figure 20** – Cost comparison (by number of improved and equal functions) between the 3MS-Size and the `exact_mig`, for criteria S3.



**Source:** Author

To apply the 3MS to maj-3 functions where  $n > 6$ , two samples of 1,000 randomly generated functions each were created for  $n = 7$  and  $n = 8$ . The 3MS was able to generate optimal solutions for all functions in the sample, using both 3MS-Depth and 3MS-Size. Likewise, to apply the 3MS algorithm to maj-5 functions where  $n = 5$ , a sample of 10,000 functions was created. The 3MS was also able to effectively cover all 10,000 functions, using both 3MS-Depth and 3MS-Size synthesis.

Both CPLEX and Gurobi optimizers were able to generate optimal solutions for all tested functions. However, when considering runtime and memory usage, the CPLEX optimizer was able to generate optimal solutions using less memory and in a lower amount of time, for both  $m = 3$  and  $m = 5$ . Table 32 shows the runtime comparison (in seconds and hours) between the Gurobi and the CPLEX optimizers, for criteria D3 and  $3 \leq n \leq 8$ . Likewise, Table 33 shows the same comparison for criteria S3.

Considering criterias D3 and S3, along with all 158,584 generated results for  $3 \leq n \leq 8$ , the total runtime of the 3MS algorithm was 101.85 hours when using the CPLEX API and 121.06 hours when using the Gurobi API. Therefore, for  $m = 3$ , the CPLEX optimizer resulted in a 16% lower runtime. Table 34 shows the memory usage comparisons (in megabytes) between the CPLEX and the Gurobi optimizers, for  $3 \leq n \leq 8$  and criterias D3 and S3.

**Table 32** – Runtime comparison between Gurobi and CPLEX, for criteria D3.

		<b>Gurobi</b>		<b>CPLEX</b>	
$n$	Total Functions	Total Runtime	Avg. Runtime	Total Runtime	Avg. Runtime
3	256	20.48 s	0.08 s	17.49 s	0.06 s
4	65,536	35.49 h	1.95 s	31.23 h	1.71 s
5	10,000	7.52 h	2.71 s	6.22 h	2.23 s
6	1,500	1.34 h	3.23 s	1.07 h	2.56 s
7	1,000	5.70 h	20.55 s	5.20 h	18.75 s
8	1,000	7.13 h	25.70 s	6.12 h	22.04 s

**Source: Author****Table 33** – Runtime comparison between Gurobi and CPLEX, for criteria S3.

		<b>Gurobi</b>		<b>CPLEX</b>	
$n$	Total Functions	Total Runtime	Avg. Runtime	Total Runtime	Avg. Runtime
3	256	15.36 s	0.06 s	14.82 s	0.05 s
4	65,536	38.04 h	2.09 s	31.49 h	1.73 s
5	10,000	9.66 h	3.48 s	8.02 h	2.89 s
6	1,500	1.98 h	4.75 s	1.79 h	4.29 s
7	1,000	6.21 h	22.39 s	4.82 h	17.36 s
8	1,000	7.99 h	28.83 s	5.89 h	21.23 s

**Source: Author****Table 34** – Average memory usage of Gurobi and CPLEX, for  $m = 3$ .

		<b>Gurobi</b>		<b>CPLEX</b>	
$n$	Total Functions	D3	S3	D3	S3
3	256	5.13 MB	4.60 MB	4.91 MB	4.27 MB
4	65,536	7.47 MB	6.89 MB	7.09 MB	6.16 MB
5	10,000	9.25 MB	8.42 MB	8.82 MB	7.33 MB
6	1,500	10.58 MB	10.16 MB	10.63 MB	9.51 MB
7	1,000	21.03 MB	19.77 MB	20.12 MB	18.06 MB
8	1,000	36.74 MB	33.26 MB	33.91 MB	23.57 MB

**Source: Author**

The average memory usages for the 3MS algorithm, using the Gurobi and CPLEX APIs respectively, were 7.78 megabytes and 7.30 megabytes. Therefore, for  $m = 3$ , the CPLEX optimizer resulted in a 6% lower memory usage.

Tables 35 and 36 shows, respectively, the runtime and memory usage comparisons between the CPLEX and the Gurobi optimizers for criteria D5 and S5, considering  $3 \leq n \leq 5$ . Considering criterias D5 and S5, along with all 151,584 generated results for  $3 \leq n \leq 5$ , the total runtime of the 3MS algorithm was 106.81 hours when using the CPLEX API and 118.08 hours when using the Gurobi API. Therefore, for  $m = 5$ , the CPLEX optimizer had a 10% lower runtime when compared to the Gurobi optimizer.

**Table 35** – Runtime comparison between Gurobi and CPLEX, for criteria D5.

		<b>Gurobi</b>		<b>CPLEX</b>	
$n$	Total Functions	Total Runtime	Avg. Runtime	Total Runtime	Avg. Runtime
3	256	2.17 m	0.51 s	1.75 m	0.41 s
4	65,536	47.13 h	2.58 s	41.35 h	2.27 s
5	10,000	11.41 h	4.11 s	10.88 h	3.92 s

Source: Author

**Table 36** – Runtime comparison between Gurobi and CPLEX, for criteria S5.

		<b>Gurobi</b>		<b>CPLEX</b>	
$n$	Total Functions	Total Runtime	Avg. Runtime	Total Runtime	Avg. Runtime
3	256	2.04 m	0.48 s	1.97 m	0.46 s
4	65,536	48.42 h	2.66 s	44.72 h	2.45 s
5	10,000	11.05 h	3.98 s	9.83 h	3.54 s

Source: Author

Table 37 shows the memory usage comparisons between the CPLEX and the Gurobi optimizers, considering  $3 \leq n \leq 5$  and  $m = 5$ .

**Table 37** – Average memory usage of Gurobi and CPLEX, for  $m = 5$ .

		<b>Gurobi</b>		<b>CPLEX</b>	
$n$	Total Functions	D5	S5	D5	S5
3	256	23.04 MB	22.36 MB	19.12 MB	17.44 MB
4	65,536	36.53 MB	33.61 MB	31.05 MB	35.28 MB
5	10,000	41.07 MB	42.15 MB	39.11 MB	38.33 MB

Source: Author

The average memory usages for the Gurobi and CPLEX optimizers were 29.99 megabytes and 35.89 megabytes, respectively. Therefore, for  $m = 5$ , the CPLEX optimizer resulted in a 17% lower memory usage when compared to the Gurobi optimizer.

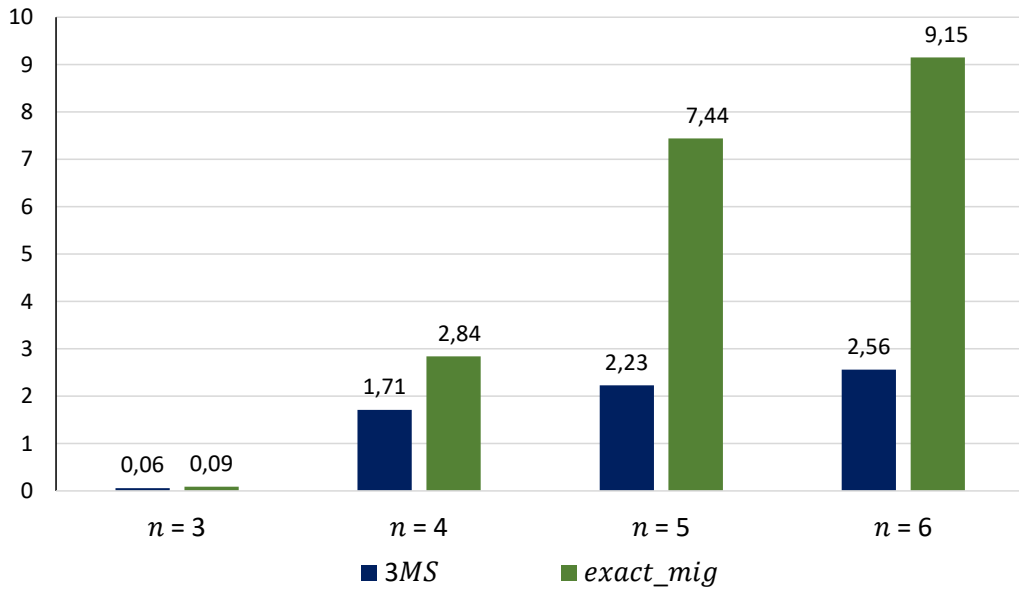
Since CPLEX generated better results for both runtime and memory usage, the following comparisons considers the 3MS algorithm runtime and memory usage with the CPLEX optimizer as the chosen API. Table 38 shows the runtime comparison between the 3MS-Depth synthesis and the exact\_mig, for criteria D3 and  $3 \leq n \leq 6$ . Figure 21 shows the average runtime comparison in graph format for each  $n$ .

The exact\_mig algorithm had a total runtime of 76,17 hours for criteria D3 and  $3 \leq n \leq 6$ , whereas the 3MS-Depth had a 97% faster total runtime of 38,52 hours. Table 39 shows the runtime comparison between the 3MS-Size and the exact\_mig, for criteria S3 and  $3 \leq n \leq 6$ . Figure 22 shows this comparison in graph format.

**Table 38** – Runtime comparison between 3MS-Depth and exact\_mig, for criteria D3.

		3MS-Depth		exact_mig	
$n$	Total Functions	Total Runtime	Avg. Runtime	Total Runtime	Avg. Runtime
3	256	17.49 s	0.06 s	23.04 s	0.09 s
4	65,536	31.23 h	1.71 s	51.70 h	2.84 s
5	10,000	6.22 h	2.23 s	20.66 h	7.44 s
6	1,500	1.07 h	2.56 s	3.81 h	9.15 s

Source: Author

**Figure 21** – Average runtime comparison (in seconds) for criteria D3.

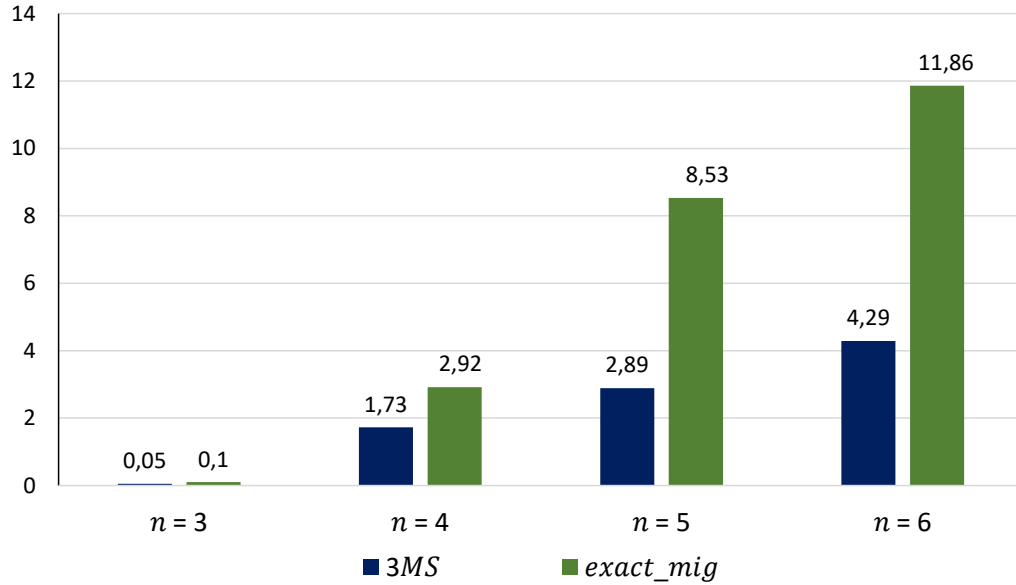
Source: Author

**Table 39** – Runtime comparison between 3MS-Size and exact\_mig, for criteria S3.

		3MS-Size		exact_mig	
$n$	Total Functions	Total Runtime	Avg. Runtime	Total Runtime	Avg. Runtime
3	256	14.82 s	0.05 s	25.6 s	0.10 s
4	65,536	31.49 h	1.73 s	53.15 h	2.92 s
5	10,000	8.02 h	2.89 s	23.69 h	8.53 s
6	1,500	1.79 h	4.29 s	4.94 h	11.86 s

Source: Author

Tables 40 and 41 shows, respectively, the average memory usage of the 3MS and exact\_mig algorithms for criterias D3 and S3. The average memory usage, considering  $3 \leq n \leq 6$  and both criterias D3 and S3, was 3.51 megabytes for the exact\_mig algorithm and 6.87 megabytes for the 3MS algorithm. Therefore, the 3MS algorithm had a 96% higher average memory usage when compared to the exact\_mig.

**Figure 22** – Average runtime comparison (in seconds) for criteria S3.

Source: Author

**Table 40** – Average memory usage of 3MS-Depth and exact\_mig, for criteria D3.

$n$	Total Functions	3MS-Depth	exact_mig
3	256	4.91 MB	2.65 MB
4	65,536	7.09 MB	3.52 MB
5	10,000	8.82 MB	4.57 MB
6	1,500	10.63 MB	6.34 MB

Source: Author

**Table 41** – Average memory usage of 3MS-Size and exact\_mig, for criteria S3.

$n$	Total Functions	3MS-Size	exact_mig
3	256	4.27 MB	2.18 MB
4	65,536	6.16 MB	3.17 MB
5	10,000	7.33 MB	3.98 MB
6	1,500	9.51 MB	5.72 MB

Source: Author

The runtime and average memory usage comparisons between the 3MS-Size and the exact\_m5ig, for criteria S5 and  $3 \leq n \leq 4$ , are presented, respectively, in Tables 42 and 43. Figure 23 shows the average runtime comparison in graph format.

Considering criteria S5, to cover all 65,792 functions where  $3 \leq n \leq 4$ , the 3MS and exact\_m5ig algorithms had a total runtime of, respectively, 44.75 and 71.63 hours, while having an average memory usage of 35.21 megabytes and 26.32 megabytes. Therefore, the 3MS algorithm was 60% faster when compared to the exact\_m5ig algorithm, but had



a 33% higher memory usage.

**Table 42** – Runtime comparison between 3MS-Size and exact\_m5ig, for criteria S5.

$n$	Total Functions	3MS-Size		exact_m5ig	
		Total Runtime	Avg. Runtime	Total Runtime	Avg. Runtime
3	256	1.97 m	0.46 s	5.41 m	1.27 s
4	65,536	44.72 h	2.45 s	71.54 h	3.93 s

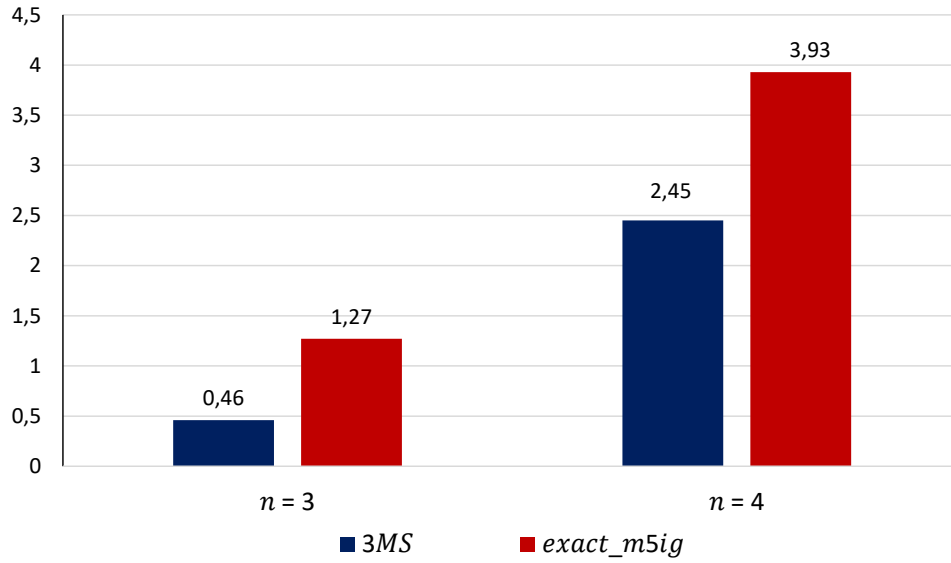
Source: Author

**Table 43** – Average memory usage of 3MS-Size and exact\_m5ig, for criteria S5.

$n$	Total Functions	3MS-Size	exact_m5ig
3	256	17.44 MB	13.59 MB
4	65,536	35.28 MB	26.37 MB

Source: Author

**Figure 23** – Average runtime comparison (in seconds) for criteria S5.



Source: Author

From all 220,376 comparisons, considering all criterias D3, S3 and S5, it is important to point out that even though the 3MS has a lower runtime, it also has a higher average memory usage. To evaluate the 3MS application to large circuits, experiments on the 10 EPFL arithmetic benchmarks were conducted (AMARU; MICHELI, 2015). Size and depth are the commonly used cost criteria to compare algorithms through the EPFL bechmarks. Without considering inverters and literals, the 3MS algorithm generates majority networks with the same cost, when compared to the exact\_mig and the exact\_m5ig, for all 10 EPFL circuits. To find a valid output majority network we first apply 4-LUT mapping, meaning that each circuit network will be divided into smaller functions with up to 4 inputs. Each

4-LUT function is synthesized and the equivalent majority function is added to the output network. When equivalent majority functions have been found for every 4-LUT, those functions are merged together to form the optimized output majority network. Table 44 shows the EPFL benchmark results for size optimization. The 3MS-Size was compared with the `exact_mig` for  $m = 3$  and with the `exact_m5ig` for  $m = 5$ , being able to generate equivalent results in both comparisons.

**Table 44** – EPFL benchmarks for Size optimization.

Benchmark	Inputs/Outputs	Initial Cost		3MS-Size = <code>exact_mig</code>		3MS-Size = <code>exact_m5ig</code>	
		Size	Depth	Size	Depth	Size	Depth
Adder	256/129	1,020	255	512	130	386	129
Barrel Shifter	135/128	3,336	12	3,238	14	2,496	14
Divisor	128/128	57,247	4,372	44,331	4,381	47,147	4,231
Hypotenuse	256/128	214,335	24,801	160,678	9,518	141,850	9,334
Log2	32/32	32,060	444	27,645	383	22,314	338
Max	512/130	2,865	287	2,535	294	2,302	237
Multiplier	128/128	27,062	274	22,720	188	19,362	186
Sine	24/25	5,416	225	4,768	169	3,822	157
Square-root	128/64	24,618	5,058	19,746	6,043	16,972	4,097
Square	64/128	18,484	250	15,670	156	13,855	129

Source: Author

Note that, when considering the same cost for `maj-3` and `maj-5` gates, the utilization of `maj-5` logic synthesis is able to further improve the size and depth of 9 out of 10 EPFL benchmarks. Table 45 shows the EPFL benchmark results for depth optimization, where the 3MS-Depth results, for  $m = 3$ , are equivalent to `exact_mig`'s. Results considering  $m = 5$  for depth optimization are also presented. Note that `maj-5` logic synthesis also generates improved results when compared to `maj-3` for depth optimization. All 10 circuits have a lower depth if synthesized to `maj-5` networks and 9 out of 10 have a lower size.

**Table 45** – EPFL benchmarks for Depth optimization.

Benchmark	Inputs/Outputs	Initial Cost		3MS-Depth = <code>exact_mig</code>		3MS-Depth for $m = 5$	
		Size	Depth	Size	Depth	Size	Depth
Adder	256/129	1,020	255	583	107	512	85
Barrel Shifter	135/128	3,336	12	3,357	9	2,811	7
Divisor	128/128	57,247	4,372	49,579	3,066	51,291	2,962
Hypotenuse	256/128	214,335	24,801	181,415	6,091	167,318	5,728
Log2	32/32	32,060	444	28,361	322	23,117	285
Max	512/130	2,865	287	2,906	255	2,647	205
Multiplier	128/128	27,062	274	24,183	167	21,990	159
Sine	24/25	5,416	225	4,959	141	4,871	123
Square-root	128/64	24,618	5,058	22,994	5,266	22,780	3,553
Square	64/128	18,484	250	17,892	102	16,045	84

Source: Author

## 6 CONCLUSION

In this work the 3MS (Majority Math Model Solver) algorithm is presented, which aims to synthesize majority functions through linear optimization models. As cost criteria the reduction of depth and size is prioritized, followed by the reduction of inverters and literals. Considering the proposed cost criteria, the 3MS achieved at least equivalent results when compared to other depth and size exact synthesis algorithms for majority-of-three ( $m = 3$ ) and majority-of-five ( $m = 5$ ) functions, being able to further improve their results with the optimization of inverters and literals, without losing optimal results for depth and size minimization.

Considering all cost criteria variations, for  $n = 3$  the 3MS was compared to a total of 768 functions, being able to reduce the cost of 456 (59%) functions while achieving equal cost results for the remaining 312 (41%). For  $n = 4$ , from a total of 196,608 functions, the 3MS further improved 124,072 (63%) functions and achieved equivalent results for 72,536 (37%). For  $n = 5$  and  $n = 6$ , from totals of 20,000 and 3,000 functions, the 3MS achieved better results for 15,629 (78%) and 2,574 (86%) functions, while also achieving equivalent results for the remaining 4,371 (22%) and 426 (14%). Therefore, from the total of 220,376 comparisons, the 3MS was able to further reduce the cost of 142,731 (64%) functions. The 3MS algorithm was also evaluated for functions where  $m = 3$  and  $n > 6$ , being able to surpass `exact_mig`'s limit of 6 input variables. Samples of 1,000 randomly generated functions were created, for both  $n = 7$  and  $n = 8$ , and the 3MS was able to effectively cover all 2,000 functions. The 3MS algorithm was also able to surpass `exact_m5ig`'s limit of 4 input variables. A sample of 10,000 functions where  $m = n = 5$  was created and effectively covered. Additionally, the EPFL arithmetic benchmark was used to evaluate the performance of the 3MS algorithm on large circuits, where the 3MS was able to generate competitive results for all 10 circuits in the benchmark, for both size and depth optimization.

## REFERENCES

- AGARWAL, B.; BAKA, B. **Hands-On Data Structures and Algorithms with Python: Write complex and powerful code using the latest features of Python 3.7**. Mumbai: Packt Publishing Ltd, 2018. 183–186 p.
- AMARU, L.; GAILLARDON, P.-E.; MICHELI, G. D. Majority-inverter graph: a novel data-structure and algorithms for efficient logic optimization. In: ANNUAL DESIGN AUTOMATION CONFERENCE, 51., 2014, San Francisco. **Proceedings...** San Francisco: IEEE. 2014. p. 1–6.
- AMARU, P. G. L.; MICHELI, G. D. The epfl combinational benchmark suite. In: INTERNATIONAL WORKSHOP ON LOGIC AND SYNTHESIS, 1., 2015, Bremen. **Proceedings...** Bremen: IWLS. 2015. p. 55–63.
- ASCENCIO, A. F. G.; ARAÚJO, G. S. d. **Estruturas de Dados: algoritmos, análise da complexidade e implementações em JAVA e C/C++**. 3. ed. São Paulo: Pearson Prentice Halt, 2010. 20-24 p.
- BERTACCO, V.; DAMIANI, M. The disjunctive decomposition of logic functions. In: INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN - ICCAD, 15., 1997, San Jose. **Proceedings...** San Jose: IEEE. 1997. p. 78–82.
- BOOLE, G. **An investigation of the laws of thought: on which are founded the mathematical theories of logic and probabilities**. Cork: Dover Publications, 1854. 24-38 p.
- CHATTOPADHYAY, A. *et al.* Notes on majority boolean algebra. In: MULTIPLE-VALUED LOGIC - ISMVL, 46., 46., 2016, Sapporo. **Proceedings...** Sapporo: IEEE. 2016. p. 50–55.
- CHU, Z. *et al.* Exact synthesis of boolean functions in majority-of-five forms. In: 2019 IEEE INTERNATIONAL SYMPOSIUM ON CIRCUITS AND SYSTEMS (ISCAS), 1., 2019, Sapporo. **Proceedings...** Sapporo: IEEE. 2019. p. 1–5.
- CHU, Z. *et al.* Functional decomposition using majority. In: ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE, 23., 2018, Jeju. **Proceedings...** Jeju: IEEE. 2018. p. 676–681.
- CHU, Z. *et al.* Advanced functional decomposition using majority and its applications. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, Lausanne, v. 39, n. 8, p. 1621–1634, 2019.
- CHUMAK, A. V. *et al.* Advances in magnetics roadmap on spin-wave computing. **IEEE Transactions on Magnetics**, Giza, v. 58, n. 6, p. 1–72, 2022.

- CHUNG, C.-C. *et al.* Majority logic circuits optimization by node merging. In: 22ND ASIA AND SOUTH PACIFIC DESIGN AUTOMATION CONFERENCE (ASP-DAC), 22., 2017, Chiba. **Proceedings...** Chiba: IEEE. 2017. p. 714–719.
- COHN, M.; LINDAMAN, R. Axiomatic majority-decision logic. **IRE Transactions on Electronic Computers**, New York, EC-10, n. 1, p. 17–21, 1961.
- DEDOV, F. **The Bible of Algorithms and Data Structures: A Complex Subject Simply Explained (Runtime Complexity, Big O Notation, Programming)**. 1. ed. California: Neural Nine, 2020. 24-27 p.
- FLOYD, T. L. **Digital fundamentals**. 11. ed. Dallas: Pearson Education, 2015. 125–149 p.
- Gurobi Optimization LLC. **Gurobi Optimizer Reference Manual**. Available in: <https://www.gurobi.com>. 2021.
- IBM CPLEX LOG. **12.8 User's Manual for CPLEX**. Available in: <https://www.ibm.com/docs/en/icos/12.8.0.0?topic=cplex-users-manual>. 2021.
- KO, C.-C. *et al.* Majority logic circuit minimization using node addition and removal. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 41, n. 3, p. 642–655, 2022.
- KONG, K.; SHANG, Y.; LU, R. An optimized majority logic synthesis methodology for quantum-dot cellular automata. **IEEE Transactions on Nanotechnology**, Delft, v. 9, n. 2, p. 170–183, 2010.
- LINDAMAN, R. A theorem for deriving majority-logic networks within an augmented boolean algebra. **IRE Transactions on electronic computers**, New York, v. 9, n. 3, p. 338–342, 1960.
- MAJEED, A. H.; HUSSAIN, G. A.; ALKALDY, E. Design a low-cost majority gate in qca nanotechnology. In: 2022 MUTHANNA INTERNATIONAL CONFERENCE ON ENGINEERING SCIENCE AND TECHNOLOGY (MICAST), 1., 2022, Samawah. **Proceedings...** Samawah: IEEE. 2022. p. 6–9.
- MISHRA, V. K. *et al.* A heuristic-driven and cost effective majority/minority logic synthesis for post-cmos emerging technologies. **IEEE Access**, New York, v. 1, n. 1, p. 1–16, 2021.
- NEUTZLING, A. *et al.* Maj-n logic synthesis for emerging technology. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, La Jolla, v. 39, n. 3, p. 747–751, 2020.
- RIENER, H. *et al.* Logic optimization of majority-inverter graphs. In: MBMV 2019: 22ND WORKSHOP-METHODS AND DESCRIPTION LANGUAGES FOR MODELLING AND VERIFICATION OF CIRCUITS AND SYSTEMS, 22., 2019, Kaiserslautern. **Proceedings...** Kaiserslautern: VDE. 2019. p. 1–4.
- SASAO, T. **Switching theory for logic synthesis**. Berlin: Springer Science & Business Media, 2012. 93-106 p.

SHANNON, C. E. The synthesis of two-terminal switching circuits. **Bell System Technical Journal**, Kansas, v. 28, n. 1, p. 59–98, 1949.

SHI, L.; CHU, Z. Inversions optimization in xor-majority graphs with an application to qca. In: 2019 CHINA SEMICONDUCTOR TECHNOLOGY INTERNATIONAL CONFERENCE (CSTIC), 12., 2019, Shangai. **Proceedings...** Shangai: IEEE. 2019. p. 1–3.

SOEKEN, M. *et al.* Exact synthesis of majority-inverter graphs and its applications. **IEEE Transactions on Computer-aided Design of Integrated Circuits and Systems**, La Jolla, v. 36, n. 11, p. 1842–1855, 2017.

SOEKEN, M. *et al.* Practical exact synthesis. In: 2018 DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE AND EXHIBITION - DATE, 1., 2018, Dresden. **Proceedings...** Dresden: IEEE. 2018. p. 309–314.

TOCCI, R. J.; WIDMER, N. S.; MOSS, G. L. **Digital systems: principles and applications**. 12. ed. Dallas: Pearson Education, 2017. 57–58 p.

WANG, P. *et al.* Synthesis of majority/minority logic networks. **IEEE Transactions on Nanotechnology**, Boston, v. 14, n. 3, p. 473–483, 2015.

WANG, Y.; YUAN, G.; SUN, J. Four-input multi-layer majority logic circuit based on dna strand displacement computing. **IEEE Access**, Adelaide, v. 8, p. 3076–3086, 2020.

WILF, H. S. **Algorithms and complexity**. [S.l.]: AK Peters/CRC Press, 2019. 5–10 p.

ZHANG, T. *et al.* Design and analysis of majority logic based approximate radix-4 booth encoders. In: 2019 IEEE/ACM INTERNATIONAL SYMPOSIUM ON NANOSCALE ARCHITECTURES, 14., 2019, Qingdao. **Proceedings...** Qingdao: IEEE. 2019. p. 1–6.

## APPENDIX A -- THE 3MS ALGORITHM CONDITIONAL CONSTRAINTS

This appendix presents an study about the logic behind the conditional constraints used to build the optimization models in this work. As conditional constraints we consider constraints that will imply a certain condition based on another condition, being logically equivalent to an “if” operation in programming languages.

When conditions are imposed on a linear programming model, binary auxiliary variables (also called indicators) are included to indicate certain states. Suppose that  $x$  represents the quantity of an ingredient to be included in a blend. We may wish to use an indicator variable  $\delta$  to distinguish between the state where  $x = 0$  and the state where  $x > 0$ . Logically, we have achieved the condition shown in Equation 43, where “ $\rightarrow$ ” stands for “implies”.

$$x > 0 \rightarrow \delta = 1 \quad (43)$$

However, this logic can be formulated as a linear programming constraint by adding a new coefficient  $M$  to  $\delta$ , where  $M$  is a fixed value that represents a known upper bound for  $x$ . The constraint shown in Equation 44 is logically equivalent to Equation 43.

$$x - M \cdot \delta \leq 0 \quad (44)$$

In this work, conditional constraints are used to build the logic behind  $\Omega$ .I’s application along with the vector  $W_i$ , used to disregard repeated gates. Examples of this constraints are listed as follows:

- $\sum_{j=1}^{1+\lfloor \frac{x}{m} \rfloor} X_{ij} \geq 1 \rightarrow W_i = 1, \forall i \in c$
- $\sum_{j=1}^{1+\lfloor \frac{x}{m} \rfloor} X_{ij} \leq 0 \rightarrow W_i = 0, \forall i \in c$
- $\sum_{i \in e} X_{ij} = m \rightarrow S1_j = \lfloor \frac{m}{2} \rfloor, \forall j > 1$

Note that we are considering the more complex version of  $\Omega$ .I's application, utilized in the 3-level optimization model. Equations 45 and 46 shows the non conditional form of the first and second constraints listed, where  $W_i$  is the binary indicator and a new coefficient  $M$  was added.

$$\sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \geq 1 - M \cdot (1 - W_i), \forall i \in c \quad (45)$$

$$\sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij} \leq M \cdot W_i, \forall i \in c \quad (46)$$

Note that for both Equations 45 and 46, the value of  $M$  must be at least equal to the sum  $\sum_{j=1}^{1+\lfloor \frac{r}{m} \rfloor} X_{ij}$ . Since the  $\Omega$ .I's constraint uses an equality verification ( $=$ ), a pair of constants is needed to rewrite it in linear programming format. The pair of constraints is shown by Equations 47 and 48, where  $S_j$  is the indicator. Note that both constraints only vary on the inequality ( $\leq$  or  $\geq$ ). Additionally, even though  $S_j$  is an integer variable, it can be used as an indicator since it can assume only 2 possible values: 0 or  $\lfloor \frac{m}{2} \rfloor$ .

$$\sum_{i \in e} X_{ij} \geq m - M \cdot (\lfloor \frac{m}{2} \rfloor - S1_j), \forall j > 1 \quad (47)$$

$$\sum_{i \in e} X_{ij} \leq m - M \cdot (\lfloor \frac{m}{2} \rfloor - S1_j), \forall j > 1 \quad (48)$$