Chapter 6
Synthesis of Majority Expressions Through Primitive Function Manipulation

Evandro C. Ferraz, Jeferson de Lima Muniz, Alexandre C. R. da Silva, and Gerhard W. Dueck

6.1 Introduction

Majority logic allows the creation of nanoelectronic circuits for several different technologies, which justifies the search for majority based algorithms that generates optimized circuits. Among the first works that deal with majority logic are Lindaman [11], Cohn [8], and Akers [1]. Lindaman [11] proposed the first theorem for applying majority logic in binary decision problems, introducing the majority operator to classical Boolean algebra. The theorem, shown in Eq. (6.1), proposes a Boolean function equivalent to a majority operation.

$$M(A, B, C) = A \cdot B + A \cdot C + B \cdot C \tag{6.1}$$

Subsequently, a set of axioms that defines the majority algebra independently of the classical Boolean algebra was presented in [8], creating the basis for current majority algebra axiomatization (Ω).

Moreover, the authors in [21] presented a method that performs the mapping of all 3-input Boolean functions into a 3-dimensional cube, generating 13 possible patterns, where each pattern has a different formula to convert a classical Boolean function into a majority equivalent.

Similarly, the authors in [19] presented a method that uses a 4-dimensional cube to map 4-input functions, generating a total of 143 representation patterns. All 143 patterns also have a specific formula to find their equivalent majority functions.

E. C. Ferraz (✉) · J. de Lima Muniz · A. C. R. da Silva
Department of Electrical Engineering, FEIS - São Paulo State University, Ilha Solteira, SP, Brazil

G. W. Dueck
Faculty of Computer Science, University of New Brunswick, Fredericton, NB, Canada

R. Drechsler, M. Soeken (eds.), *Advanced Boolean Techniques*,
https://doi.org/10.1007/978-3-030-20323-8_6

In majority algebra, simplification algorithms based on primitive functions are widely used. Primitive functions are functions with at most one majority gate in their optimized form. An algorithm that maps each of the primitive functions and uses the obtained maps to generate more complex functions was proposed in [18]. The mapping of functions is realized with Karnaugh maps, a graphical method proposed by Maurice Karnaugh in 1953, which aims to simplify a classic Boolean function by mapping its truth table [10].

In [12] a similar algorithm was developed, the $B2M$ (Boolean to Majority). The $B2M$ receives a Boolean function as input and generates a majority function that covers the same set of minterms. The generation of an output function is also done with the combination of primitives, selected by their MLD (Modified Levenshtein Distance).

The authors in [22] developed the program denominated as $MALS$ (Majority Logic Synthesizer). It was the first program to minimize majority functions with more than three inputs. The $MALS$ receives an algebraically minimized Boolean function as input and returns an equivalent majority function. The algorithm starts by preprocessing the input function, this process aims to decompose the input function in a way that no node has more than three input variables. To do this process the program $SISTOOL$ is used [14].

After the preprocessing, the algorithm converts each node in a reduced majority function. This is done by the method presented in [18]. But this method was not able to find minimal solutions for all functions given that the algorithm works individually on each node instead of the function as a whole.

The authors in [20] proposed a methodology that combines lower level majority functions, starting from primitives, to form higher level majority functions. The goal of this method is to build a majority expressions Look-Up Table ($MLUT$) that stores the majority equivalent for all possible 4-input Boolean functions. Using the $MLUT$, the algorithm will then search the equivalent majority expression for every node in the input network, generating a majority network as output.

The authors in [16] proposed the *exact_mig* algorithm, which is considered state of the art. As input, the algorithm receives a truth table or a Majority Inverter Graph (MIG) [3], with a maximum of six input variables, and returns a majority function that covers the same set of minterms. A MIG is a graph that represents a majority function. The most important characteristic of this algorithm is the proposal of an exact synthesis for majority functions. The function is built from a set of constraints (K) that shape a given problem accordingly to the definitions of the majority Boolean algebra. The majority output function is generated with the application of K to an SMT (Satisfiability Module Theory) solver [9]. As cost criteria the *exact_mig* takes into consideration the number of levels and gates in the output function, making it possible to choose which of these criteria will be prioritized.

In [7] the authors developed a decomposition methodology that uses XOR and majority operators as a base. The input function is converted into a XOR-Majority Graph (XMG), a MIG with the addition of the XOR operator, and decomposed into simplified sub-functions. To perform the decomposition, the

algorithm combines theories of majority algebra, Shannon decomposition [15], and disjoint-support decomposition (DSD)[5].

In [17] the authors proposed adaptations of the exact synthesis used in the *exact_mig*, applied to normal Boolean functions. New technologies based on constraints and SMT solvers are also presented and compared.

In this work the MPC algorithm is proposed. Similar to the methodology proposed in [20], the algorithm checks all possible combinations among primitive functions and creates a table to store them. For each function, the covered set of minterms is also stored. If there are two functions that cover the same set of minterms, the lowest cost function is kept and the other function is discarded. As a result, we have a table (M_2) that lists all the sets covered by majority functions with two levels. As cost criteria the algorithm considers the depth, followed by the number of gates, the number of inverters, and the number of gate inputs in the output function.

The MPC can be used to synthesize Boolean functions with a maximum of 5-input variables. For 3-input variables the algorithm returns an optimal solution for all possible functions. For 4 and 5-input variables the algorithm guarantees an optimal solution for functions covered by M_2 or by a primitive, and uses a specific synthesis to cover functions with a higher number of levels. For five variables however, functions with four or more levels are generated by the application of the Shannon theorem.

This article is organized as follows: In Sect. 6.2, we present an explanation about majority algebra, including its axiomatization and the concept of primitive majority functions. Section 6.3 presents the MPC algorithm, explaining how it works for 3, 4, and 5-input variables. Section 6.4 presents the results obtained comparing the MPC and the *exact_mig*. Section 6.5 presents the conclusion of what was realized in the paper.

6.2 Majority Boolean Algebra

The majority Boolean algebra is composed by the set $\{\mathbb{B}, \neg, M\}$. The elements $\mathbb{B}$ and $\neg$, as in classical Boolean algebra, represent the binary values $\{0, 1\}$ and the inversion operator, respectively, and M represents the majority operator [6].

A majority function returns as output the most present binary value among its inputs. Therefore, an operator M that has a total of three input variables will return a true value only if two or more inputs are true. The truth table presented in Table 6.1 exemplifies a majority operation for the variables X, Y, and Z.

From a majority operation it's also possible to obtain AND and OR functions, performed by fixing one of the input variables to a constant binary value.

As an example, the function $M(A, B, C)$ is considered. Setting the value of A to 0, we have an AND function between B and C. Setting the value of A to 1, we have an OR function between B and C. This example is shown in Table 6.2.

Table 6.1 Example of a majority operation

X	Y	Z	$M(X, Y, Z)$
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

Table 6.2 Generation of functions AND and OR

B	C	$B \cdot C$	$M(0, B, C)$	$B + C$	$M(1, B, C)$
0	0	0	0	0	0
0	1	0	0	1	1
1	0	0	0	1	1
1	1	1	1	1	1

Table 6.3 Equivalence between $M(A, B, C)$ and its dual form

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

Majority functions are also self-dual functions, 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 [13]. For example, the function $(X \cdot Y) + (X \cdot Z)$ is equal to its dual form $\overline{(\overline{X} + \overline{Y}) \cdot (\overline{X} + \overline{Z})}$.

Table 6.3 shows the equivalence between a majority function $M(A, B, C)$ and its dual form $\overline{M}(\overline{A}, \overline{B}, \overline{C})$.

6.2.1 Axiomatization of Majority Functions (Ω)

The set of axioms that defines the majority algebra is represented by Ω and can be divided into axioms of Commutativity, Associativity, Distribution, Inverter Propagation, and Majority [4]. Every axiom in Ω can be proved by perfect induction.

Table 6.4 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

The Commutativity axiom ($\Omega.C$), represented in Eq. (6.2), determines that the input order doesn't change the output value.

$$M(A, B, C) = M(A, C, B) = M(C, B, A) \tag{6.2}$$

Table 6.4 proves $\Omega.C$ by perfect induction.

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 Eq. (6.3).

$$M(A, D, M(B, D, C)) = M(C, D, M(B, D, A)) \tag{6.3}$$

Note that the variable shared between levels is D. Therefore, it's 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 6.5 proves $\Omega.A$ by perfect induction.

The Distribution axiom ($\Omega.D$) determines that it's possible to distribute a set of variables to gates in subsequent levels. In Eq. (6.4) an example of this theorem is given, where the distributed set is $\{A, B\}$.

$$M(A, B, M(D, E, C)) = M(M(A, B, D), M(A, B, E), M(A, B, C)) \tag{6.4}$$

Table 6.6 proves $\Omega.D$ by perfect induction.

The Inverter Propagation axiom ($\Omega.I$), represented in Eq. (6.5), determines that a majority function is self-dual [2].

$$\overline{M}(A, B, C) = M(\overline{A}, \overline{B}, \overline{C}) \tag{6.5}$$

Table 6.7 proves $\Omega.I$ by perfect induction.

The Majority ($\Omega.M$) can be divided in two equations. Equation (6.6) shows that the output of a majority gate is equal to the most common value among its inputs.

Table 6.5 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(1,0,M(0,0,1)) = 0	M(1,1,M(0,1,0)) = 0
1	0	1	0	M(1,0,M(0,1,0)) = 0	M(1,0,M(1,0,1)) = 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

Equation (6.7) shows that the output value will be equal to the tie-breaking variable in functions with the same number of true and false values.

$$M(A, A, B) = A \tag{6.6}$$

$$M(A, \overline{A}, B) = B \tag{6.7}$$

Table 6.8 proves $\Omega.M$ by perfect induction.

6.2.2 *Primitive Majority Functions*

Primitive functions can be obtained by a single gate. In the majority algebra, primitive functions (also called primitives) can be used as a base for the construction of more complex functions. All primitives can be obtained from the sets C, V, G, and T, where each set corresponds to functions with a specific number of inputs. The total number of primitives is obtained by summing the functions in C, V, G, and T [20].

The set C represents functions with no input variables, covering the constants 0 and 1. Therefore, $|C| = 2$.

Table 6.6 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(1,0,0), 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

The set V represents all functions formed by a single input variable, in its complemented form or not. Equation (6.8) shows how to calculate the number of functions in V.

$$|V| = 2 \cdot n \tag{6.8}$$

In Table 6.9, we can observe the listing of V for three input variables. The number of input variables are represented by n. Note that the classical functions and their

Table 6.7 Proof of $\Omega.I$ by perfect induction

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

Table 6.8 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

Table 6.9 List of set V for $n = 3$

Classic function	Majority function
A	A
B	B
C	C
$\overline{A}$	$\overline{A}$
$\overline{B}$	$\overline{B}$
$\overline{C}$	$\overline{C}$

corresponding majority forms are equal because the V set is composed only by functions without operators.

The set G is formed by functions with a single AND or OR operator, having a total of 2 input variables. The number of functions in G can be calculated by Eq. (6.9). The variables E and O represent the possible combinations of inputs, for AND and OR operations, respectively. For $n = 3$, we have $E = \{A \cdot B, A \cdot C, B \cdot C\}$ and $O = \{A + B, A + C, B + C\}$. Each combination has 4 inversion variations, the combination $A + B$, for example, has the variations $\left\{A + B, \overline{A} + B, A + \overline{B}, \overline{A} + \overline{B}\right\}$.

$$|G| = (4 \cdot |E|) + (4 \cdot |O|) \tag{6.9}$$

In Table 6.10, we present the functions in G for $n = 3$.

The set T represents functions with a single majority gate, no constant value and no repeated variable as input. Equation (6.10) calculates the number of functions in T. The variable t represents the number of possible combinations among the input

Table 6.10 List of set G for $n = 3$

Classic function	Majority function
$A \cdot B$	$M(A, B, 0)$
$\overline{A} \cdot B$	$M(\overline{A}, B, 0)$
$A \cdot \overline{B}$	$M(A, \overline{B}, 0)$
$\overline{A} \cdot \overline{B}$	$\overline{M}(A, B, 1)$
$A \cdot C$	$M(A, 0, C)$
$\overline{A} \cdot C$	$M(\overline{A}, 0, C)$
$A \cdot \overline{C}$	$M(A, 0, \overline{C})$
$\overline{A} \cdot \overline{C}$	$\overline{M}(A, 1, C)$
$B \cdot C$	$M(0, B, C)$
$\overline{B} \cdot C$	$M(0, \overline{B}, C)$
$B \cdot \overline{C}$	$M(0, B, \overline{C})$
$\overline{B} \cdot \overline{C}$	$\overline{M}(1, B, C)$
$A + B$	$M(A, B, 1)$
$\overline{A} + B$	$M(\overline{A}, B, 1)$
$A + \overline{B}$	$M(A, \overline{B}, 1)$
$\overline{A} + \overline{B}$	$\overline{M}(A, B, 0)$
$A + C$	$M(A, 1, C)$
$\overline{A} + C$	$M(\overline{A}, 1, C)$
$A + \overline{C}$	$M(A, 1, \overline{C})$
$\overline{A} + \overline{C}$	$\overline{M}(A, 0, C)$
$B + C$	$M(1, B, C)$
$\overline{B} + C$	$M(1, \overline{B}, C)$
$B + \overline{C}$	$M(1, B, \overline{C})$
$\overline{B} + \overline{C}$	$\overline{M}(0, B, C)$

Table 6.11 List of set T for $n = 3$

Classic function	Majority function
$AB + AC + BC$	$M(A, B, C)$
$\overline{A} \cdot \overline{B} + \overline{A} \cdot \overline{C} + \overline{B} \cdot \overline{C}$	$\overline{M}(A, B, C)$
$\overline{A} \cdot B + \overline{A} \cdot C + B \cdot C$	$M(\overline{A}, B, C)$
$A \cdot \overline{B} + A \cdot \overline{C} + \overline{B} \cdot \overline{C}$	$\overline{M}(\overline{A}, B, C)$
$A \cdot \overline{B} + A \cdot C + \overline{B} \cdot C$	$M(A, \overline{B}, C)$
$\overline{A} \cdot B + \overline{A} \cdot \overline{C} + B \cdot \overline{C}$	$\overline{M}(A, \overline{B}, C)$
$A \cdot B + A \cdot \overline{C} + B \cdot \overline{C}$	$M(A, B, \overline{C})$
$\overline{A} \cdot \overline{B} + \overline{A} \cdot C + \overline{B} \cdot C$	$\overline{M}(A, B, \overline{C})$

variables, considering three inputs per combination. Note that each combination has eight variations of inverters and, for $n = 3$, there is only one possible combination.

$$|T| = t \cdot 8 \tag{6.10}$$

Table 6.11 shows the list of functions in T, for $n = 3$.

Table 6.12 Complete list of primitives for $n = 3$

N	Classic function	Majority function	N	Classic function	Majority function
1	0	0	21	$A + B$	$M(A, B, 1)$
2	1	1	22	$\overline{A} + B$	$M(\overline{A}, B, 1)$
3	A	A	23	$A + \overline{B}$	$M(A, \overline{B}, 1)$
4	B	B	24	$\overline{A} + \overline{B}$	$\overline{M}(A, B, 0)$
5	C	C	25	$A + C$	$M(A, 0, C)$
6	$\overline{A}$	$\overline{A}$	26	$\overline{A} + C$	$M(\overline{A}, 1, C)$
7	$\overline{B}$	$\overline{B}$	27	$A + \overline{C}$	$M(A, 1, \overline{C})$
8	$\overline{C}$	$\overline{C}$	28	$\overline{A} + \overline{C}$	$\overline{M}(A, 0, C)$
9	$A \cdot B$	$M(A, B, 0)$	29	$B + C$	$M(1, B, C)$
10	$\overline{A} \cdot B$	$M(\overline{A}, B, 0)$	30	$\overline{B} + C$	$M(1, \overline{B}, C)$
11	$A \cdot \overline{B}$	$M(A, \overline{B}, 0)$	31	$B + \overline{C}$	$M(1, B, \overline{C})$
12	$\overline{A} \cdot \overline{B}$	$\overline{M}(A, B, 1)$	32	$\overline{B} + \overline{C}$	$\overline{M}(0, B, C)$
13	$A \cdot C$	$M(A, 0, C)$	33	$AB + AC + BC$	$M(A, B, C)$
14	$\overline{A} \cdot C$	$M(\overline{A}, 0, C)$	34	$\overline{A} \cdot B + \overline{A} \cdot C + B \cdot C$	$M(\overline{A}, B, C)$
15	$A \cdot \overline{C}$	$M(A, 0, \overline{C})$	35	$A \cdot \overline{B} + A \cdot C + \overline{B} \cdot C$	$M(A, \overline{B}, C)$
16	$\overline{A} \cdot \overline{C}$	$\overline{M}(A, 1, C)$	36	$A \cdot B + A \cdot \overline{C} + B \cdot \overline{C}$	$M(A, B, \overline{C})$
17	$B \cdot C$	$M(0, B, C)$	37	$\overline{A} \cdot \overline{B} + \overline{A} \cdot C + \overline{B} \cdot C$	$M(\overline{A}, \overline{B}, C)$
18	$\overline{B} \cdot C$	$M(0, \overline{B}, C)$	38	$\overline{A} \cdot \overline{B} + \overline{A} \cdot \overline{C} + \overline{B} \cdot \overline{C}$	$\overline{M}(A, B, C)$
19	$B \cdot \overline{C}$	$M(0, B, \overline{C})$	39	$A \cdot \overline{B} + A \cdot \overline{C} + \overline{B} \cdot \overline{C}$	$M(A, \overline{B}, \overline{C})$
20	$\overline{B} \cdot \overline{C}$	$\overline{M}(1, B, C)$	40	$\overline{A} \cdot B + \overline{A} \cdot \overline{C} + B \cdot \overline{C}$	$M(\overline{A}, B, \overline{C})$

Table 6.12 shows the complete primitives table for $n = 3$. Note that $|C| + |V| + |G| + |T| = 40$.

6.3 The *MPC* Algorithm

In this section we propose the MPC algorithm. The MPC receives a truth table f as input and returns a majority function that covers the same set of minterms. To generate a valid output function we use the expression $M(X_1, X_2, X_3)$. Each variable X_c, where $1 \leq c \leq 3$, represents a majority primitive or a 2-level majority function.

6.3.1 Tables Formulation

The first step of MPC is the tables formulation phase, where the functions used to build $M(X_1, X_2, X_3)$ are formulated. The algorithm receives an input truth table f, identifies the number of input variables, represented by n, and generates the

primitives table based on the sets C, V, G, and T. We also store the set of minterms covered by every primitive function. Note that each primitive function is the optimal solution of its respective set of minterms.

The second table built by the MPC is the M_2 table, formed by the application of all possible combinations among primitive functions in the expression $M(X_1, X_2, X_3)$, without considering repeated primitives. For each generated function the set of covered minterms is also stored. If a set is covered by two or more functions, the one with the lowest cost is kept and the others are discarded. Therefore, the table M_2 lists all sets of minterms that can be covered by a 2-level majority function and, since they are obtained exhaustively, M_2 functions are also an optimal solution for their respective set of minterms. It is also important to point out that, for computational performance optimization, the M_2 is stored as a LUT (Look-Up Table) in the MPC code.

As an example of a M_2 function, we have $M(X_1, X_2, X_3) = M(A, M(A, \overline{B}, 0), \overline{M}(A, B, C))$, where $X_1 = A$, $X_2 = M(A, \overline{B}, 0)$, and $X_3 = \overline{M}(A, B, C)$.

The cost criteria used by the MPC is primarily the number of levels and gates in the output function, followed by the number of inverters and gate inputs.

To ensure the minimization of inverters, the single gate primitives follow four possible patterns:

- $M(A, B, C)$, no inverters;
- $M(\overline{A}, B, C)$, a single complemented input;
- $\overline{M}(A, B, C)$, a single inverter applied to the output value;
- $\overline{M}(\overline{A}, B, C)$, a single input and the output complemented.

Note that in cases where the gate has two inverters, even though the number of inverters stay the same, it's better to complement the output and only one input, since $M(\overline{X}, Y, \overline{Z}) = \overline{M}(X, \overline{Y}, Z)$. This allows the application of $\Omega.I$ to minimize the number of inverters when the primitives are being used to build functions with two or more levels.

To exemplify this application we consider: $M(\overline{M}(A, \overline{B}, C), \overline{D}, 0)$, which has 2 levels, 2 gates, and 3 inverters. By applying $\Omega.I$ we have $M(\overline{M}(A, \overline{B}, C), \overline{D}, 0) = \overline{M}(M(A, \overline{B}, C), D, 1)$, which has the same number of levels and gates, but has one less inverter.

It's also important to point out that repeated gates are not considered in the cost calculation. In the function $M(M(0, A, C), M(1, A, M(B, C, D)), M(1, C, M(B, C, D)))$, for example, given that the gate $M(B, C, D)$ appears twice, we count a total of five gates in the function cost.

The total of possible functions for a specific number of inputs is represented by the variable S, and can be calculated by 2^m. Note that $m = 2^n$, and represents the number of minterms in the input truth table f.

For $n = 3$, $S = 256$. The primitives table covers 40 of these functions. The 216 left are covered by the M_2 table. Therefore, S can be completely covered by majority expressions with at most two levels, which makes the table formulation phase enough for obtaining all optimal solutions for $n = 3$.

For $n = 4$, S = 65,536 and 90 of these functions are primitives, with at most one majority gate. In the formulation of M_2, only 10,260 functions can be covered. For the remaining 55,186, 55,184 can be covered by majority expressions with three levels. The remaining two functions need a majority expression with four levels to be covered.

6.3.2 *MPC Synthesis for* 4*-Input Functions*

This section presents the synthesis used in MPC for the construction of majority functions where $n = 4$. The objective of this synthesis is to formulate $M(X_1, X_2, X_3)$ with the combination of primitives and M_2 functions, generating a majority function that covers the same minterms of f. Note that this synthesis is only applied if f can't be covered by any function in the M_2 table or by any primitive.

The synthesis is composed by two different loops, each one having their own characteristics. If an output function couldn't be found in the first loop the second starts.

The first loop is composed by the following steps:

1. Any primitive or M_2 function that doesn't cover at least one minterm of f is discarded from its respective table.
2. Build a new table P, selecting every pair of primitives $(p_1 + p_2)$, where:
   - Every minterm covered by f is also covered at least once by $p_1 + p_2$;
   - The pair $p_1 + p_2$ only covers minterms covered by f.
3. Select a pair of primitives from P, as X_1 and X_2.
4. Create a vector v with 2^n elements that will be used to build the truth table for X_3. Every element in v represents a minterm in f. The vector v is updated according to the set of minterms covered by X_1 and X_2. If a minterm i is covered by both functions, $v_i = 2$. If it's covered by only one function, $v_i = 1$. And if it isn't covered by any function, $v_i = 0$. For example, given $f = \{0, 1, 5, 8\}$, $X_1 = \{0, 1, 4, 5\}$ and $X_2 = \{0, 1, 2, 8, 10\}$. Then v has the values shown in Table 6.13.
5. Create the truth table for X_3, represented by the vector $X_3 f$. Positions where $v_i = 2$ or $v_i = 0$ are considered as don't care states (represented by x). For positions where $v_i = 1$ and i is also covered by f, we have $X_3 f_i = 1$. If $v_i = 1$ and i isn't covered by f, we have $X_3 f_i = 0$. Therefore, for the example presented in Table 6.13, we have $X_3 f = [xx0x01xx1x0xxxxx]$.
6. Generate every possible truth table manipulating the don't care states in $X_3 f$. Each possibility is searched in the $M2$ table. From the functions, a new table, P_3, is constructed.

Table 6.13 Generation of vector v

Minterms	$f = \{0, 1, 5, 8\}$	$X_1 = \{0, 1, 4, 5\}$	$X_2 = \{0, 1, 2, 8, 10\}$	v
0	1	1	1	2
1	1	1	1	2
2	0	0	1	1
3	0	0	0	0
4	0	1	0	1
5	1	1	0	1
6	0	0	0	0
7	0	0	0	0
8	1	0	1	1
9	0	0	0	0
10	0	0	1	1
11	0	0	0	0
12	0	0	0	0
13	0	0	0	0
14	0	0	0	0
15	0	0	0	0

7. For every function in P_3 composed by a gate that also composes X_1 or X_2, we reduce its cost by 1. This rule exists because each gate is counted only once in the calculation of a majority function size.
8. Select the lowest cost function in P_3, that hasn't been selected yet, as X_3. If there's no valid X_3, we go back to step 3 and find a new primitive pair.
9. With the selection of X_3 we now have a valid output $M(X_1, X_2, X_3)$. To minimize inverters, $\Omega.I$ is applied to every level of the function built. If the function post $\Omega.I$ application has a lower cost, the previous function is substituted.
10. The loop ends when every possible pair in P has been combined with a function from M_2, and every $M(X_1, X_2, X_3)$ found is stored in table Z.
11. By the end of the loop, the algorithm returns the function with the lowest cost in Z. If no function could be found the second loop starts.

To exemplify an iteration of the first loop, consider $n = 4$ and $f = \{4, 5, 6, 9, 15\}$. A valid output function can be found in the iteration where $X_1 = M(A, D, 0)$ and $X_2 = M(\overline{A}, B, C)$, X_1 covering the minterms $\{9, 11, 13, 15\}$ and X_2 covering $\{2, 3, 4, 5, 6, 7, 14, 15\}$. Table 6.14 shows vector v updated from X_1 and X_2.

The minterms considered don't care states, where $v_i = 2$ or $v_i = 0$, are $\{0, 1, 8, 10, 12, 15\}$. The minterms where $v_i = 1$ and $f_i = 1$ are $\{4, 5, 6, 9\}$, and the minterms where $v_i = 1$ and $f_i = 0$ are $\{2, 3, 7, 11, 13, 14\}$. Therefore, $X_3 f = xx001110x1x0x00x$.

We select as X_3, from the M_2 table, the lowest cost function that fits the truth table pattern formed by $X_3 f$. We select $X_3 = \overline{M}(C, M(\overline{B}, D, 1), M(A, B, 0))$

Table 6.14 Vector v updated from X_1 and X_2, for the first loop example

Minterms	f	X_1	X_2	v
0	0	0	0	0
1	0	0	0	0
2	0	0	1	1
3	0	0	1	1
4	1	0	1	1
5	1	0	1	1
6	1	0	1	1
7	0	0	1	1
8	0	0	0	0
9	1	1	0	1
10	0	0	0	0
11	0	1	0	1
12	0	0	0	0
13	0	1	0	1
14	0	0	1	1
15	1	1	1	2

that covers the minterms $\{0, 1, 4, 5, 6, 8, 9, 12\}$, and has 1100111011001000 as truth table. Accordingly, we have $M(X_1, X_2, X_3) = M(M(A, D, 0), M(\overline{A}, B, C), \overline{M}(C, M(\overline{B}, D, 1), M(A, B, 0)))$.

The dual form of $M(M(A, D, 0), M(\overline{A}, B, C), \overline{M}(C, M(\overline{B}, D, 1), M(A, B, 0)))$ is equal to $\overline{M}(\overline{M}(\overline{A}, \overline{D}, 1), \overline{M}(A, \overline{B}, \overline{C}), M(\overline{C}, \overline{M}(B, \overline{D}, 0), \overline{M}(\overline{A}, \overline{B}, 1)))$, which has a higher amount of inverters. Therefore, for $f = \{4, 5, 6, 9, 15\}$, the MPC algorithm adds $M(M(A, D, 0), M(\overline{A}, B, C), \overline{M}(C, M(\overline{B}, D, 1), M(A, B, 0)))$ to its table of possible outputs Z. The loop ends when every pair of functions in P are selected and combined with a function from M_2, and returns the lowest cost function in Z as output.

From all 55,184 sets of minterms that can be covered by a 3-level function, 50,016 can be covered by functions where two elements of X_c are primitives. Those functions are found by the first loop.

Among the 5168 remaining sets, 5056 can be covered by functions where only one element of X_c is a primitive. The 112 remaining sets can only be covered by functions where all elements of X_c are 2-level functions from M_2. Those functions are found by the second loop.

The second loop is composed by the following steps:

1. Select X_1 from the primitives table. If every primitive function has been selected as X_1 and a valid output function could not be found, X_1 is selected from a group of functions R. The group R is formed by every M_2 function with size r, where r represents the number of gates in a M_2 function. Therefore, r starts at 2, the lowest number of gates that a 2-level majority function can have, and is incremented if a group R with higher size functions must be defined.

2. Create two new vectors, v_0 and v_{-1}. The vector v_0 contains the positions of f that haven't been covered yet; therefore, $v_0 = f - X_1$. The vector v_{-1} has the positions of v that can't be covered one more time; therefore, $v_{-1} = X_1 - f$.
3. From v_0 and v_{-1} the truth tables for X_2, represented by the variable $X_2 f$ are generated. $X_2 f$ represents a truth table, with the same size of f, that can have binary values or don't care states. For the minterms stored in v_0, $X_2 f_i = 1$. For minterms stored in v_{-1}, $X_2 fi = 0$. The other minterms are all considered don't care states.
4. Every possible truth table manipulating the don't care states in $X_2 f$ is generated. Each possibility is searched in the M_2 table. From these functions a new table, P_2, is created.
5. For every function in P_2 that is composed by a gate that also composes X_1, its cost is reduced by one.
6. Select the lowest cost function in P_2, that was not selected yet, as X_2. If there's no valid X_2, go back to the first step and select a new X_1.
7. To find X_3 create $X_3 f$ based on v_{-1} and a new vector v_1. The vector v_1 stores the minterms of f covered only once by X_c. Therefore, the minterms in v_1 must be covered by X_3. For the minterms stored in v_{-1}, $X_3 f = 0$. For the minterms stored in v_1, $X_3 f = 1$.
8. To find all possibilities for $X_3 f$, search the respective functions in the M_2 table and build P_3 from them.
9. Again, update the cost of the functions in P_3 based on the gates in X_1 and X_2.
10. Select the lowest cost function in P_3, that hasn't been selected yet, as X_3. If there's no valid X_3, go back to step 6 and select a new X_2.
11. With the selection of X_3 we now have a valid output $M(X_1, X_2, X_3)$. For the minimization of inverters we also apply $\Omega.I$ to every level of the function built and we substitute it if the function post $\Omega.I$ application has a lower cost.
12. Every $M(X_1, X_2, X_3)$ found is stored in table Z and the loop stops when all primitive functions are selected as X_1. If no function could be found, the algorithm goes back to the first step and restarts selecting X_1 from a group R, stopping when all functions in R were selected as X_1. If yet no function could be found, the algorithm increments r and restarts the loop with a new group R. The algorithm returns the lowest cost function stored in Z as output.

For the two sets that need a function with four levels to be covered, we first select X_1 from the primitives table, then we build X_2 and X_3 as 3-level functions using the explained synthesis.

6.3.3 MPC Synthesis for 5-Input Functions

The synthesis for 5-input ($n = 5$) functions also uses the primitives and the M_2 table as a base to build functions with a higher number of levels.

For $n = 5$, S = 4,294,967,296 and 172 of these sets can be covered by primitives, with at most one majority gate. The M_2 table stores the 253,560 sets that can be covered by majority functions with 2 levels. The remaining sets need more than 2 levels to be covered.

To build 3-level functions the algorithm also uses the expression $M(X_1, X_2, X_3)$, realizing the combination of primitives and M_2 functions, selected by their lowest cost.

The complete synthesis for 3-level functions is composed by the following steps:

1. Order by cost every function from the primitives and M_2 tables.
2. Select the function with the lowest cost as X_1.
3. Reduce the cost by one for every primitive or M_2 function that is composed by a gate that also composes X_1.
4. Create v_0 and v_{-1}, where $v_0 = f - X_1$ and $v_{-1} = X_1 - f$.
5. Select X_2, firstly from the primitives, as the lowest cost function that:
   - Covers all minterms in v_0.
   - Doesn't cover any minterm in v_{-1}.

   If no valid X_2 could be found among the primitives, select X_2 from the M_2 table. If still no valid X_2 could be found, go back to step 2 and select a new X_1.
6. Again, update the cost of the primitives and M_2 functions based on the gates in X_1 and X_2.
7. Create v_1, where v_1 stores the minterms covered by f and only once by X_c.
8. Select X_3, firstly from the primitives, the lowest cost function that:
   - Covers all minterms in v_1.
   - Doesn't cover any minterm in v_{-1}.

   If no valid X_3 could be found among the primitives, select X_3 from the M_2 table. If still no valid X_3 could be found, go back to step 5 and select a new X_2.
9. With the selection of X_3 we now have a valid output. Next apply $\Omega.I$ to every level of $M(X_1, X_2, X_3)$ and return the lowest cost version as output.

To exemplify the second loop, consider $n = 5$ and $f = \{2, 4, 6, 7, 8, 11, 13, 14, 15\}$. A valid output function can be found in the iteration where $X_1 = C$, covering the minterms $\{4, 5, 6, 7, 12, 13, 14, 15, 20, 21, 22, 23, 28, 29, 30, 31\}$.

Updating the vector v based on X_1, we have $v_0 = \{2, 8, 11\}$ and $v_{-1} = \{5, 12, 20, 21, 22, 23, 28, 29, 30, 31\}$. Table 6.15 shows v after the selection of X_1.

As X_2, we select the lowest cost function from M_2 that covers every minterm in v_0 but doesn't cover any of the minterms in v_{-1}.

We select $X_2 = \overline{M}(M(C, D, 1), M(B, \overline{E}, 0), M(A, \overline{B}, E))$ that covers $\{0, 1, 2, 4, 6, 8, 9, 11, 13, 15, 16, 17, 24, 25\}$. From X_2, we update v again, generating $v_1 = \{2, 7, 8, 11, 14\}$ and $v_{-1} = \{0, 1, 5, 9, 12, 16, 17, 20, 21, 22, 23, 24, 25, 28, 29, 30, 31\}$. Table 6.16 shows v updated after X_2's selection.

As X_3, we select the lowest cost function from M_2 that doesn't cover any minterms in v_{-1} and covers all minterms in v_1.

Table 6.15 Vector v updated from X_1, for the second loop example

Minterms	f	X_1	v
0	0	0	0
1	0	0	0
2	1	0	0
3	0	0	0
4	1	1	1
5	0	1	−1
6	1	1	1
7	1	1	1
8	1	0	0
9	0	0	0
10	0	0	0
11	1	0	0
12	0	1	−1
13	1	1	1
14	1	1	1
15	1	1	1
16	0	0	0
17	0	0	0
18	0	0	0
19	0	0	0
20	0	1	−1
21	0	1	−1
22	0	1	−1
23	0	1	−1
24	0	0	0
25	0	0	0
26	0	0	0
27	0	0	0
28	0	1	−1
29	0	1	−1
30	0	1	−1
31	0	1	−1

We have $X_3 = M(M(\overline{A}, D, 0), M(B, \overline{C}, 0), \overline{M}(A, B, E))$, covering the minterms $\{2, 3, 6, 7, 8, 10, 11, 14\}$.

With the selection of X_3, MPC returns $M(X_1, X_2, X_3) = M(C, \overline{M}(M(C, D, 1), M(B, \overline{E}, 0), M(A, \overline{B}, E)), M(M(\overline{A}, D, 0), M(B, \overline{C}, 0), \overline{M}(A, B, E)))$ as output, which has less inverters than its dual form.

Table 6.16 Vector v updated from X_2, for the second loop example

Minterms	f	X_2	v
0	0	1	−1
1	0	1	−1
2	1	1	1
3	0	0	0
4	1	1	2
5	0	0	−1
6	1	1	2
7	1	0	1
8	1	1	1
9	0	1	−1
10	0	0	0
11	1	1	1
12	0	0	−1
13	1	1	2
14	1	0	1
15	1	1	2
16	0	1	−1
17	0	1	−1
18	0	0	0
19	0	0	0
20	0	0	−1
21	0	0	−1
22	0	0	−1
23	0	0	−1
24	0	1	−1
25	0	1	−1
26	0	0	0
27	0	0	0
28	0	0	−1
29	0	0	−1
30	0	0	−1
31	0	0	−1

For functions that need more than three levels to be covered we apply the reduction of fan-ins by Shannon expansion. Equation (6.11) shows the equivalent majority version of the Shannon theorem, applied to the set of inputs $\{A, B, C, D, E\}$.

$$M(A, B, C, D, E) = M(M(F_1, 0, A), M(F_2, 0, \overline{A}), 1) \tag{6.11}$$

The variable A represents the isolated variable and F_1 and F_2 represent functions built with the remaining inputs {B,C,D,E}.

Table 6.17 Example of f_1's and f_2's generation by Shannon theorem

Minterms	B	C	D	E	f_1	f_2
0	0	0	0	0	0	0
1	0	0	0	1	1	0
2	0	0	1	0	1	0
3	0	0	1	1	1	0
4	0	1	0	0	0	1
5	0	1	0	1	0	1
6	0	1	1	0	0	1
7	0	1	1	1	1	0
8	1	0	0	0	1	0
9	1	0	0	1	0	1
10	1	0	1	0	0	0
11	1	0	1	1	0	1
12	1	1	0	0	1	0
13	1	1	0	1	0	1
14	1	1	1	0	1	0
15	1	1	1	1	0	1

The first step to apply this equation in the MPC algorithm is to isolate the first input (A). Then split the input truth table f in two pieces to form 2 new truth tables, f_1 and f_2.

Table 6.17 shows an example of f_1's and f_2's generation. For this example, f = [01110001100010100000111001010101] and the set of inputs are $\{A, B, C, D, E\}$ ($n = 5$).

Note that, by splitting f in 2 equal size tables, we have f_1 = [0111000110001010] and f_2 = [0000111001010101], where the set of inputs became $\{B, C, D, E\}$ ($n = 4$) and the variable A is isolated.

To find F_1 and F_2 we apply the MPC synthesis for $n = 4$, explained in the previous section, to f_1 and f_2, respectively.

Note that the functions built by the Shannon Theorem aren't an optimal solution for f, since Eq. (6.11) adds two levels and three gates by itself.

6.4 Results

In this section results obtained from the comparison of the algorithms MPC and *exact_mig* are presented. For $n = 4$ both algorithms were executed for all 65,536 possible functions. The obtained results were then compared based on the cost criteria used by the MPC that prioritizes first the number of levels in the output function, followed by the number of gates, the number of inverters, and the number of gate inputs. In Table 6.18 each column corresponds to a group S_i, where $0 \leq i \leq 2^n$. Each S_i represents a total of functions that covers a specific number of minterms. S_4, for example, represents every function that covers 4 minterms among the 65,536

Table 6.18 Cost comparison between MPC and *exact_mig*

i	S_i	$MPC < exactmig$	$MPC < exactmig$	$MPC < exactmig$
0	1	0	0	1
1	16	16	0	0
2	120	41	8	71
3	560	324	60	176
4	1820	808	708	304
5	4368	2906	583	879
6	8008	4493	2276	1239
7	11,440	7188	3300	952
8	12,870	8108	3474	1288
9	11,440	7536	3022	882
10	8008	6273	1121	614
11	4368	3334	512	522
12	1820	1373	279	168
13	560	482	0	78
14	120	93	8	19
15	16	12	0	4
16	1	0	0	1
Total	65,536	42,987	15,351	7198

possibilities. Equation (6.12) shows the calculation of S_i. Note that $m = 2^n$, and represents the total of minterms for a specific number of inputs. For $n = 4$, $m = 16$.

$$S_i = \frac{m!}{i! \cdot (m - i)!} \tag{6.12}$$

For each S_i the table shows the quantity of functions where MPC generated results with a lower, higher, and equal cost than *exact_mig*.

The MPC generates lower cost results for 42,987 (66%) functions, generates results with equal cost for 7198 (11%) functions, and generates results with higher cost for the remaining 15,351 (23%).

Note that MPC is able to generate better results because *exact_mig* aims for the exact synthesis of only depth and size, while MPC considers also the number of inverters and the number of gate inputs as cost criteria. In this comparison, the *exact_mig* functions were generated with the prioritization of depth, followed by the function size, differing from MPC only by the addition of the number of inverters and gate inputs as third and fourth criteria, respectively.

Functions where the MPC returns a higher cost than *exact_mig* exist because the MPC builds $M(X_1, X_2, X_3)$ prioritizing X_1 and X_2 as primitives, only using functions from M_2 if needed. This rule is essential for the formation of optimized functions in the majority of the cases, but there are cases where the prioritization of 2-level functions would generate a lower cost result.

As an example consider $f = 1000000000000001$. The MPC returns the function $M(\overline{M}(B, C, 1), M(B, D, 0), M(1, M(A, C, 0), \overline{M}(A, D, 1)))$ as output that has 3 levels and 6 gates. The *exact_mig* algorithm returns $\overline{M}(M(0, C, \overline{D}), M(1, D, M(A, B, \overline{D})), \overline{M}(0, C, M(A, B, \overline{D})))$, which has 3 levels and 5 gates.

In the presented case, the MPC generated a result prioritizing the use of two primitive functions ($X_1 = \overline{M}(B, C, 1)$ and $X_2 = M(B, D, 0)$), and a single 2-level function ($X_3 = M(1, M(A, C, 0), \overline{M}(A, D, 1))$), while *exact_mig* was able to generate better results with a single primitive and 2-level functions, since the combination of both functions uses the gate $M(A, B, \overline{D})$ twice and the cost of repeated gates is disconsidered.

As example of a function where the MPC generated better results, we have $f = 0000001001000100$, where the MPC generated the function $M(M(A, \overline{C}, 0), M(C, D, 1), M(0, B, \overline{M}(A, D, 1)))$ as output that has 3 levels, 5 gates, and 2 inverters. The *exact_mig* generated the function $M(\overline{M}(0, \overline{B}, C), M(0, D, \overline{M}(\overline{A}, C, D)), M(\overline{D}, M(\overline{A}, C, D), M(0, \overline{B}, C)))$ for the same truth table f that has 3 levels, 5 gates, and 5 inverters. Note that the MPC generates a function with the same number of levels and gates, but with less inverters.

In Tables 6.19 and 6.20, comparisons about the runtime of both algorithms are presented. Table 6.19 shows the total and average runtime for every S_i. Table 6.20 shows the total and average runtime for all functions with a specific depth. The comparisons were made in a computer with 8 GB RAM and a 1.7 GHZ CPU.

Table 6.19 Runtime comparison between MPC and *exact_mig* by S_i

		MPC		$exact mig$	
i	S_i	Total runtime	Avg. runtime	Total runtime	Avg. runtime
0	1	0.01 s	0.01 s	0.01 s	0.01 s
1	16	1.10 s	0.06 s	1.65 s	0.10 s
2	120	30.02 s	0.25 s	17.85 s	0.14 s
3	560	8.29 min	0.88 s	4.57 min	0.49 s
4	1820	23.41 min	0.77 s	21.44 min	0.70 s
5	4368	3.10 h	2.55 s	2.09 h	1.72 s
6	8008	11.39 h	5.12 s	3.84 h	1.73 s
7	11,440	24.76 h	7.79 s	21.74 h	6.84 s
8	12,870	19.64 h	5.49 s	9.36 h	2.61 s
9	11,440	24.85 h	7.82 s	20.76 h	6.53 s
10	8008	11.69 h	5.25 s	3.54 h	1.59 s
11	4368	3.67 h	3.03 s	2.27 h	1.89 s
12	1820	29.81 min	0.98 s	22.12 min	0.73 s
13	560	7.42 min	0.79 s	4.81 min	0.51 s
14	120	31.56 s	0.26 s	18.68 s	0.15 s
15	16	0.99 s	0.06 s	2.24 s	0.14 s
16	1	0.01 s	0.01 s	0.01 s	0.01 s
Total	65, 536	100.26 h	5.50 s	64.49 h	3.54 s

Table 6.20 Runtime comparison between MPC and *exact_mig* by Depth

		MPC		$exactmig$	
Depth	Total functions	Total runtime	Avg. runtime	Total runtime	Avg. runtime
0	10	0.12 s	0.01 s	0.23 s	0.02 s
1	80	1.07 s	0.01 s	1.62 s	0.02 sec
2	10,260	103.81 s	0.01 s	318.34 s	0.03 s
3	55,184	91.80 h	5.98 s	50.26 h	3.27 s
4	2	8.46 h	4.23 h	14.22 h	7.11 h

Table 6.21 Comparison of average memory usage for $n = 4$

i	S_i	MPC	$exactmig$
0	1	3.36 MB	0.01 MB
1	16	4.38 MB	3.00 MB
2	120	4.72 MB	3.01 MB
3	560	5.06 MB	3.28 MB
4	1.820	5.27 MB	3.26 MB
5	4.368	5.34 MB	3.55 MB
6	8.008	5.51 MB	3.39 MB
7	11.440	5.64 MB	3.61 MB
8	12.870	5.87 MB	3.63 MB
9	11.440	5.61 MB	3.61 MB
10	8.008	5.53 MB	3.38 MB
11	4.368	5.32 MB	3.55 MB
12	1.820	5.21 MB	3.27 MB
13	560	5.23 MB	3.28 MB
14	120	4.67 MB	3.02 MB
15	16	4.41 MB	3.00 MB
16	1	3.39 MB	0.01 MB
Total	65,536	5.56 MB	3.52 MB

Note that even though the MPC can generate faster results for functions with 0, 1, 2, or 4 levels, in most cases it is still slower than *exact_mig*.

In Table 6.21, we present the average memory usage in the synthesis of every group S_i, in megabytes (MB).

Note that the MPC has an average memory usage of 5.36 MB, while the *exact_mig* has an average memory usage of 3.52 MB.

For $n = 5$ a sample of 1000 randomly generated functions was used and the MPC algorithm was able to achieve lower cost results for 477 (48%) functions, and equal cost results for 112 (11%).

The MPC's total runtime for the generated sample was 11.62 h, with an average runtime of 41.63 s. The *exact_mig*'s total runtime was 19.33 h, with an average runtime of 1.15 min.

Therefore, the MPC was able to generate results 66% faster than *exact_mig*. The MPC's average memory usage was 40.32 MB, while the *exact_mig*'s was only 5.05 MB.

Note that results for $n = 3$ are not presented because both algorithms return optimal solutions for all 256 possible functions.

6.5 Conclusions

In this paper we present the MPC algorithm, which aims to generate majority functions based on an input truth table. We also present a study on the main concepts of majority Boolean algebra and primitive functions. With the proposed cost criteria the MPC presented, in the most part, results better or equal to *exact_mig*. It's important to point out that the MPC is able to find better results only considering two additional cost criteria: the number of inverters and gate inputs.

For functions with $n = 4$, from a total of 65,536 possible functions, the MPC generated functions with lower cost in 42,987 (66%) cases and functions with equal cost in 7198 (11%) cases, reaching a total of 50,185 (77%) functions with equal or lower cost than *exact_mig*. The MPC had an average computational time of 5.50 s and an average memory usage of 5.56 MB, while the *exact_mig* had an average computational time of 3.54 s and an average memory usage of 3.52 MB.

For functions with $n = 5$, from a sample of 1000 functions, the MPC found better or equal results for a total of 589 (59%) functions, where 477 (48%) had lower cost and 112 (11%) had equal cost. The MPC's average computational time and memory usage were 41.63 s and 40.32 MB, while *exact_mig*'s average computational time and memory usage were 1.15 min and 5.05 MB, respectively.

The MPC's code is available at: https://github.com/EvandroFerraz/mpc. The list of functions used to compare MPC and *exact_mig* for 5-input functions can also be find in the link.

References

1. Akers, S.B.: A truth table method for the synthesis of combinational logic. IRE Trans. Electron. Comput. **4**, 604–615 (1961)
2. Akers, S.B.: Synthesis of combinational logic using three-input majority gates. In: Proceedings of the Third Annual Symposium on Switching Circuit Theory and Logical Design, 1962. SWCT 1962, pp. 149–158. IEEE, Sri Lanka (1962)
3. Amarú, L., Gaillardon, P.E., De Micheli, G.: Majority-inverter graph: a novel data-structure and algorithms for efficient logic optimization. In: Proceedings of the 51st Annual Design Automation Conference, pp. 1–6. ACM, New York (2014)
4. Amaru, L., Gaillardon, P.E., Chattopadhyay, A., De Micheli, G.: A sound and complete axiomatization of majority-*n* logic. IEEE Trans. Comput. **65**(9), 2889–2895 (2016)
5. Bertacco, V., Damiani, M.: The disjunctive decomposition of logic functions. In: International conference on Computer-aided design (ICCAD), pp. 78–82. IEEE, San Jose (1997)

6. Chattopadhyay, A., Amarú, L., Soeken, M., Gaillardon, P.E., De Micheli, G.: Notes on majority Boolean algebra. In: 2016 IEEE 46th International Symposium on Multiple-Valued Logic (ISMVL), pp. 50–55. IEEE, Sri Lanka (2016)
7. Chu, Z., Soeken, M., Xia, Y., De Micheli, G.: Functional decomposition using majority. In: Asia and South Pacific Design Automation Conference, pp. 676–681. IEEE, Jeju (2018)
8. Cohn, M., Lindaman, R.: Axiomatic majority-decision logic. IRE Trans. Electron Comput. (1), 17–21 (1961)
9. De Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340. Springer, New York (2008)
10. Karnaugh, M.: The map method for synthesis of combinational logic circuits. Trans. Am. Inst. Electr. Eng. Part I Commun. Electron. **72**(5), 593–599 (1953)
11. Lindaman, R.: A theorem for deriving majority-logic networks within an augmented Boolean algebra. IRE Trans. Electron. Comp. (3), 338–342 (1960)
12. Mishra, V.K., Thapliyal, H.: Heuristic based majority/minority logic synthesis for emerging technologies. In: 2017 30th International Conference on VLSI Design and 2017 16th International Conference on Embedded Systems (VLSID), pp. 295–300. IEEE, Jeju (2017)
13. Sasao, T.: Switching Theory for Logic Synthesis, 1st edn. Springer Science & Business Media, Berlin (2012)
14. Sentovich, E.M., Singh, K.J., Lavagno, L., Moon, C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P.R., Brayton, R.K., Sangiovanni-Vincentelli, A.: Sis: a system for sequential circuit synthesis. Tech. rep., EECS Department, University of California, CA (1992). http://www2.eecs.berkeley.edu/Pubs/TechRpts/1992/2010.html
15. Shannon, C.E.: The synthesis of two-terminal switching circuits. Bell Syst. Tech. J. **28**(1), 59–98 (1949)
16. Soeken, M., Amaru, L.G., Gaillardon, P.E., De Micheli, G.: Exact synthesis of majority-inverter graphs and its applications. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **36**(11), 1842–1855 (2017)
17. Soeken, M., Haaswijk, W., Testa, E., Mishchenko, A., Amarù, L.G., Brayton, R.K., De Micheli, G.: Practical exact synthesis. In: 2018 Design, Automation and Test in Europe Conference and Exhibition (DATE), pp. 309–314. IEEE, Dresden (2018)
18. Walus, K., Schulhof, G., Jullien, G., Zhang, R., Wang, W.: Circuit design based on majority gates for applications with quantum-dot cellular automata. In: Conference Record of the Thirty-Eighth Asilomar Conference on Signals, Systems and Computers 2004, vol. 2, pp. 1354–1357. IEEE, Dresden (2004)
19. Wang, P., Niamat, M., Vemuru, S.: Minimal majority gate mapping of 4-variable functions for quantum cellular automata. In: 2011 11th IEEE Conference on Nanotechnology (IEEE-NANO), pp. 1307–1312. IEEE, Dresden (2011)
20. Wang, P., Niamat, M.Y., Vemuru, S.R., Alam, M., Killian, T.: Synthesis of majority/minority logic networks. IEEE Trans. Nanotechnol. **14**(3), 473–483 (2015)
21. Zhang, R., Walus, K., Wang, W., Jullien, G.A.: A method of majority logic reduction for quantum cellular automata. IEEE Trans. Nanotechnol. **3**(4), 443–450 (2004)
22. Zhang, R., Gupta, P., Jha, N.K.: Majority and minority network synthesis with application to QCA-, SET-, and TPL-based nanotechnologies. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **26**(7), 1233–1245 (2007)