UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

LUCAS HAGEMEISTER

Como identificar funções threshold de até oito variáveis

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof. Dr. Renato Perez Ribas

Porto Alegre
2022

Este trabalho é dedicado àqueles que mais incentivam a realização dos meus sonhos.Gratidão eterna aos meus pais e ao meu irmão.

AGRADECIMENTOS

Agradeço ao meu professor orientador por toda dedicação e amizade nestes meses de realização deste trabalho. Agradeço ao Augusto Neutzling, que me ajudou a tirar dúvidas em relação ao seu trabalho, que até 2019 foi o estado-da-arte na área desta monografia. Agradeço a todos os professores que passaram por minha trajetória acadêmica, pois foram fundamentais na minha formação profissional. E agradeço aos meus colegas (e amigos) de curso, parceiros ao longo destes anos. Ninguém chega longe sozinho.

RESUMO

Funções Threshold são um subconjunto de funções booleanas importantes em áreas como nanotecnologia. Elas também têm aplicações em redes neurais. Um campo de estudo é determinar se uma função booleana é uma função threshold, o que é chamado de identificação de funções threshold. Neste campo de estudo, até o momento da escrita deste trabalho, o trabalho "Threshold Function Identification by Redundancy Removal and Comprehensive Weight Assignments" (IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 38, NO. 12, DECEMBER 2019) é o estado-da-arte. Neste Trabalho de Graduação, foi feita a implementação em C++ de todos os algoritmos do fluxograma do trabalho mencionado e a verificação dos resultados experimentais.

Palavras-chave: Identificação de funções threshold. circuitos digitais. atribuições de pesos. nanotecnologia.

How to identify up-to-8-variables-threshold functions

ABSTRACT

Threshold functions are a subset of Boolean functions important in areas such as nanotechnology. They also have applications in neural networks. One field of study is to determine whether a Boolean function is a threshold function, which is called identifying threshold functions. In this field of study, at the time of writing this work, the work "Threshold Function Identification by Redundancy Removal and Comprehensive Weight Assignments" (IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF INTEGRATED CIRCUITS AND SYSTEMS, VOL. 38, NO. 12, DECEMBER 2019) is state-of-the-art. In this Graduation Work, all the algorithms of the flowchart of the mentioned work were implemented in C++ and the experimental results were verified.

Keywords: threshold functions identification, digital circuits, weight assignments, nanotechnology.

LISTA DE ABREVIATURAS E SIGLAS

CI	Circuitos Integrados
CMOS	Semicondutor de Óxido Metálico Complementar
CW	Peso Crítico
ILP	Programação Linear Inteira
NMOS	Semicondutor de Óxido Metálico tipo N
NP	Negação-Permutação
NPN	Negação-Permutação-Negação
TF	Função Threshold
TLG	Porta Lógica Threshold
TLN	Rede Lógica Threshold
VWO	Ordenamento de Peso das Variáveis

LISTA DE FIGURAS

LISTA DE TABELAS

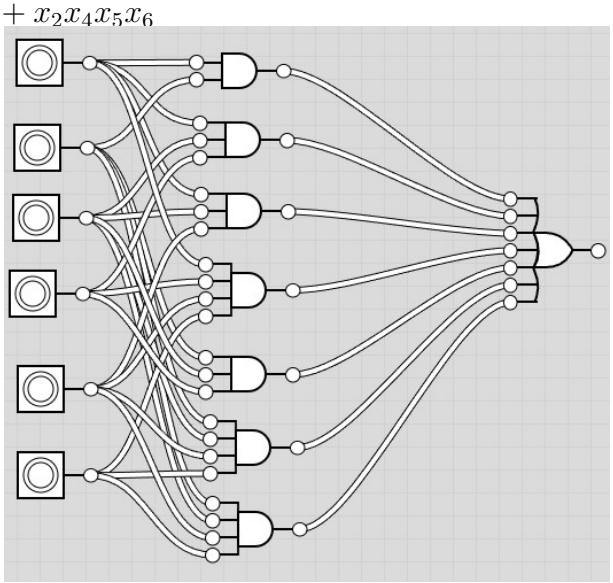
SUMÁRIO

1 INTRODUÇÃO

Circuitos Integrados (CI) são tradicionalmente implementados em tecnologia *Complementary Metal Oxide Semiconductor* (CMOS), baseados em funções booleanas convencionais e suas portas lógicas AND,OR e NOT. No entanto, para os emergentes dispositivos em nanoescala, o CMOS se mostra inconveniente, uma vez que dispositivos nanotecnológicos implicam uma área compacta demais para o CMOS (ZHANG et al., 2005).

Promissores substitutos dos transistores MOS para estes novos tipos de CIs são as portas lógicas threshold (TLGs), baseadas em funções threshold (TFs), um grupo específico de funções booleanas (ZHANG et al., 2005). Um IC em geral ocupa menos área se implementado por TLG do que o IC equivalente implementado com portas lógicas tradicionais via tecnologia CMOS. Para ilustrar a redução de espaço físico necessário, a figura 1.1 mostra a implementação em 2 níveis de portas lógicas AND e OR da função $f = x_1x_2 + x_1x_3x_4 + x_1x_3x_5 + x_1x_4x_5x_6 + x_2x_3x_4 + x_2x_3x_5x_6 + x_2x_4x_5x_6$. A figura mostra que são necessárias 8 portas lógicas (7 ANDs e 1 OR) e 30 conexões entre portas e entradas para construir este circuito. A mesma função booleana pode ser implementada em uma única TLG, como mostra a figura 1.2.

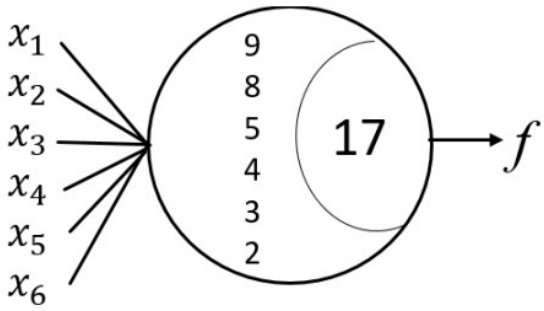
Figura 1.1: Representação em 2 níveis de portas lógicas tradicionais (que podem ser implementadas com tecnologia CMOS) de $f = x_1x_2 + x_1x_3x_4 + x_1x_3x_5 + x_1x_4x_5x_6 + x_2x_3x_4 + x_2x_3x_5x_6 + x_2x_4x_5x_6$



Fonte: https://logic.ly/demo/

Como será mostrado no capítulo de Fundamentação Teórica, uma função é dita booleana quando suas entradas e a saída são 0 (false) ou 1 (true) e sua expressão respeitam

Figura 1.2: Representação em uma TLG de $f = x_1x_2 + x_1x_3x_4 + x_1x_3x_5 + x_1x_4x_5x_6 + x_2x_3x_4 + x_2x_3x_5x_6 + x_2x_4x_5x_6$



Fonte: (LIU,2019)

as regras da álgebra booleana. Com ela é possível criar as convencionais portas lógicas, como, por exemplo, AND, OR, NOT e XOR.TF é um tipo específico de função booleana, onde cada entrada e a saída continuam sendo apenas 0 ou 1, mas cada entrada possui um peso – normalmente um valor inteiro positivo –, e a porta lógica possui um valor de limiar (threshold).

Desde a década de 1960 há pesquisas relacionadas à síntese de threshold logic (WINDER, 1962), geralmente relacionadas a redes neurais. Na década de 1970, (MUROGA, 1971) compilou diversos conceitos e teoremas matemáticos relacionado à threshold logic. No entanto, a falta de implementações eficientes de circuitos e a ascensão do NMOS (e posteriormente do CMOS) fizeram com que este ramo de estudo não tivesse muito progresso por décadas, até que se percebeu, entre o final dos anos 1990 e início dos anos 2000, sua aplicação nas emergentes nanotecnologias, uma vez que uma função booleana precisa de menos portas lógicas de threshold (TLG) do que portas lógicas booleanas tradicionais (usadas no CMOS) para ser implementada e, por consequência, ocupa menos área. Por isso, 3 décadas depois de seu lançamento, (MUROGA, 1971) foi essencial nas pesquisas que vieram a partir de então.

Hoje, os estudos das TFs se dividem em três campos. O primeiro é relacionado à identificação de TFs, isto é, como descobrir se dada função booleana é uma função threshold (que pode ser representada por uma única porta threshold). Como nem toda função booleana pode ser representada por uma única TLG, o segundo campo explora a síntese de uma TLN, ou seja, como criar um circuito de diversas TLGs para representar uma função booleana. Por fim, há pesquisas que estão focadas somente na implementação física das TFs, mais especificamente nos dispositivos em nanoescala, como diodos de tunelamento ressonante (PACHA, 1999), autômatos celulares quânticos (BLAIR; LENT, 2003) e dispositivos de tunelamento de elétron único (LIKHAREV, 1999).

Em relação ao primeiro campo de estudo, que é a identificação de TFs, já há diversas pesquisas relacionadas ao tema. Os sete principais, que são melhor descritos no capítulo 3, são: (AVEDILLO; QUINTANA, 2004), (ZHANG et al., 2005), (SUBIRATS; JEREZ; FRANCO, 2008),(GOWDA et al., 2011), (PALANISWAMY; GOPARAJU; TRAGOUDAS, 2012), (NEUTZLING et al., 2018) e (LIU et al., 2019).

1.1 Objetivo

O trabalho de (NEUTZLING et al., 2018) foi realizado na UFRGS e,até 2019, foi o estado-da-arte. Foi substituído desta posição pelo trabalho de (LIU et al., 2019), cuja implementação não temos na UFRGS. Portanto, o objetivo deste trabalho é compreender plenamente o trabalho do estado-da-arte na área de identificação de TFs, implementá-lo e validar os resultados. A partir desse objetivo, pode-se definir os seguintes objetivos específicos:

1. Investigar na literatura a evolução dos métodos propostos para identificar TFs.
2. Estudar com profundidade o trabalho de (NEUTZLING et al., 2018) feito na UFRGS.
3. Estudar as melhorias que (LIU et al., 2019) fez em relação a (NEUTZLING et al., 2018).
4. Implementar os algoritmos de (LIU et al., 2019) e validar os resultados quanto à identificação de TFs.

Ao final deste trabalho, é possível ter uma base sólida para, em uma eventual pesquisa a nível de pós-graduação, buscar melhorar o estado-da-arte.

1.2 Estrutura do texto

Este texto é dividido em seis partes além deste primeiro capítulo de introdução. O capítulo 2 traz os fundamentos teóricos para o pleno entendimento deste trabalho. Estes fundamentos são: função booleana, expressão lógica 2 níveis, unateness, classes de funções, TF, TLG, TLN, ILP,parâmetros de Chow e CW. Caso o leitor não tenha o conhecimento destes conceitos,irá aprender com este capítulo e conseguirá acompanhar o restante da monografia. O capítulo 3 aborda os trabalhos relacionados, trazendo uma linha histórica das principais pesquisas e suas contribuições na área de identificação de funções

threshold. O capítulo 4 apresenta a proposta e a metodologia deste trabalho, trazendo com profundidade os detalhes técnicos das duas pesquisas que são pilares deste trabalho: (NEUTZLING et al., 2018) e (LIU et al., 2019). O capítulo 5 detalha a implementação do trabalho de (LIU et al., 2019) feita pelo autor desta monografia, explicando cada tomada decisão na hora de escrever o código. O capítulo 6 mostra os resultados experimentais, o que foi possível validar dos resultados obtidos por (LIU et al., 2019). Por fim, o capítulo 7 mostra as conclusões obtidas acerca da implementação e o que pode ser feito em um eventual trabalho de pós-graduação.

2 FUNDAMENTAÇÃO TEÓRICA

Caso o leitor não tenha familiaridade com TFs, este capítulo é essencial para que possa ter a base teórica para poder acompanhar o restante do trabalho. Caso o leitor já conheça todos os conceitos listados, pode pular a leitura para o próximo capítulo.

2.1 Função booleana

Uma função booleana f é uma função do tipo $f(X) : B^n \rightarrow B$, onde $B = \{0, 1\}$. B também é conhecido como domínio booleano, uma vez que seus relementos representam *true* (0) ou *false* (1). As operações feitas em uma expressão booleana seguem as regras da álgebra booleana. Resumidamente, são três as principais operações:

- operação "and"(multiplicação lógica): todos os operandos precisam ser 1 (true) para que o valor da operação seja 1.
- operação "or"(adição lógica): basta que um dos operandos seja 1 (true) para que o valor da operação seja 1.
- operação "not (negação lógica): inverte o valor booleano (o que era true vira false, e vice-versa).

Um exemplo de função booleana é $f = x_1 \cdot x_2 + \overline{x_1} \cdot x_3$. Neste exemplo, há uma operação and entre x_1 e x_2 e $\overline{x_1}$ (que é a negação de x_1) e x_3. Entre estes dois termos multiplicativos, há uma operação or (representada pelo símbolo +).

2.2 Expressão Lógica 2 Níveis

Uma expressão lógica 2 níveis é uma expressão booleana na qual a distância da entrada até a saída é de duas portas lógicas AND/OR. Por exemplo, a função $f = x_1 \cdot x_2 + x_3 \cdot x_4$ é uma expressão de 2 níveis porque, para qualquer uma das entradas do circuito, o sinal vai passar por uma porta AND e depois por uma OR, até chegar na saída do circuito.

Seja a função booleana $f = \overline{a}\overline{b}\overline{c}\overline{d} + \overline{a}\overline{b}\overline{c}d + \overline{a}\overline{b}c\overline{d} + \overline{a}\overline{b}cd + \overline{a}b\overline{c}d + ab\overline{c}d + abc\overline{d}$. Podemos chamá-la de SOP (do inlgês, sum of products), pois ela é justamente uma soma de vários termos que são produtos de literais. A SOP é um dos dois tipos de expressão lógica 2 níveis: várias portas ANDs que estão conectadas a uma porta OR.

Esta expressão tem 28 literais. No entanto, ela possui termos redundantes, desnecessários. Para exemplificar, observe os dois primeiros termos: $\overline{a}\overline{b}\overline{c}\overline{d}$ e $\overline{a}\overline{b}\overline{c}d$. Quando $d = 0$, teremos, $\overline{a}\overline{b}\overline{c} \cdot 0$ e $\overline{a}\overline{b}\overline{c} \cdot 1$. Quando $d = 1$, teremos exatamente os mesmos termos. Ou seja, d é desnecessário e poderíamos juntar os dois termos em um só: $\overline{a}\overline{b}\overline{c}$.

Existem técnicas para fazer estas simplicações. Como veremos nos próximos capítulos, a usada neste trabalho para fazer simplificação foi o algortimo de Quine McCluskey. Quando conseguimos reduzir ao máximo a SOP, obtemos, então, a ISOP (irredundant sum of product), isto é, uma expressão simplificada ao máximo. No exemplo mostrado anteriormente, a sua ISOP é $f = abc\overline{d} + b\overline{c}d + \overline{a}\overline{b}$.

A outra forma de expressão lógica 2 níveis (além da SOP) é a POS, que é justamente o contrário: várias portas ORs que estão conectadas a uma porta AND. A função $g = (\overline{a} + \overline{b} + \overline{c} + \overline{d}) \cdot (\overline{a} + \overline{b} + \overline{c} + d) \cdot (\overline{a} + \overline{b} + c + \overline{d}) \cdot (\overline{a} + \overline{b} + c + d) \cdot (\overline{a} + b + \overline{c} + d) \cdot (a + b + \overline{c} + d) \cdot (a + b + c + \overline{d})$ é um exemplo de POS. De maneira análoga à ISOP, a IPOS é a POS irredutível, ou seja, que todos os termos presentes são essenciais, não é possível simplificar ainda mais a expressão.

2.3 Unateness

Uma variável de entrada de uma função booleana é unate se uma mudança em seu valor (de 0 para 1 ou de 1 para 0) provoca uma ou nenhuma mudança na saída desta função. Esta entrada é chamada de unate positiva quando consegue alterar o valor da função de 0 para 1 se ela tiver a transição de 0 para 1 também. É chamada de unate negativa se consegue alterar o valor da função de 0 para 1 se a variável tiver a transição de 1 para 0. Quando uma variável pode provocar até duas mudanças na saída, então ela é dita binate.

Quando todas as entradas da função são unates, então dizemos que a função também é unate. Funções booleanas que implementam AND2 e NOR2 são exemplos de funções unates. Esta propriedade é importante para identificação de funções de threshold porque uma propriedade necessária (mas não suficiente) para ser TF é a função ser unate (MUROGA, 1971).

A tradução mais próxima para a palavra unateness no português seria monotonicidade, mas como a literatura usa com mais frequência a expressão unateness (e unate), e por não haver uma tradução para a palavra binateness (algo como "bitonicidade"), o autor opta por usar, ao longo deste trabalho, as expressões originais em inglês.

2.4 Classes de Funções

Considere as seguintes operações que podem ser feitas em cima da função f:

1. Negação de uma ou mais variáveis de f.
2. Permutação das variáveis de f.
3. Negação da f.

Uma classe NP-equivalente é um conjunto de funções que são obtidas das combinações das operações 1 e 2 (mas não 3). Uma classe PN-equivalente é um conjunto de funções que são obtidas das combinações das operações 2 e 3 (mas não 1). Uma classe NPN-equivalente é um conjunto de funções que são obtidas das combinações das três operações.

Todas as funções booleanas que pertencem à mesma classe NPN-equivalente (ou à mesma clase NP-equivalente) têm uma característica em comum: ou todas são TFs ou nenhuma é (MUROGA, 1971).

2.5 TF

TFs são um subconjunto das funções booleanas e sua definição remonta da década de 1970 (MUROGA, 1971). Em resumo, é atribuído para cada entrada x_i um peso w_i. O valor da função será 1 caso o somatório ponderado das entradas seja maior ou igual a um valor T, chamado de threshold. Caso a soma não alcance T, então a saída da função é 0. Sua expressão é mostrada abaixo.

$$f = \begin{cases} 1, \sum_{i=1}^{n} w_i \cdot x_i \geq T, \\ 0, \text{caso contrário} \end{cases}$$

É possível usar a figura 1.2 como exemplo. Sabe-se quais são os pesos de cada entrada e o valor de threshold. Nesta situação, o somatório definido acima na expressão acima é $9x_1+8x_2+5x_3+4x_4+3x_5+2x_6$. Se esta soma for maior ou igual do que 17 (valor threshold), então o sinal da saíde é $f = 1$. Caso seja inferior ao threshold, então $f = 0$. Se as entradas forem, por exemplo, $x_1 = 0, x_2 = 0, x_3 = 1, x_4 = 1, x_5 = 1, x_6 = 0$, o somatório será $9 \cdot 0 + 8 \cdot 0 + 5 \cdot 1 + 4 \cdot 1 + 3 \cdot 1 + 2 \cdot 0 = 12 \leq 17$. Portanto, para este conjunto de entradas, $f = 0$.

2.6 TLG

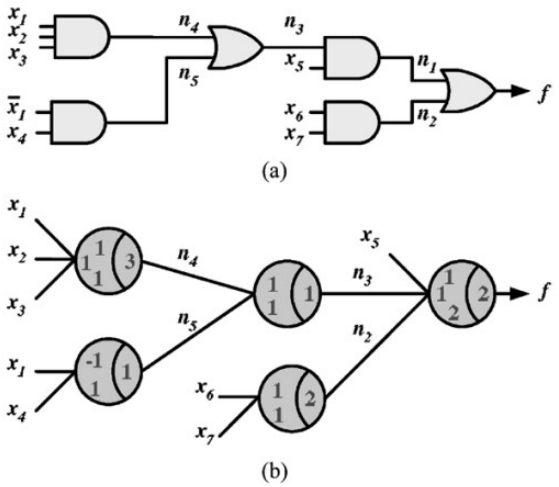
Uma TLG é um circuito eletrônico que implementa uma TF e é representada por um vetor $[w_1, w_2, ..., w_n; T]$, onde w_i representa o peso da variável de entrada x_i, e T é o valor threshold. A figura 1.2 é um exemplo de uma TLG.

2.7 TLN

Nem toda função booleana é uma TF. Nestas situações, é preciso usar mais de uma TLG para representar esta função booleana em lógica threshold. Quando se tem mais de uma TLG, há, então, uma rede de portas TLG, chamada de TLN.

Ou seja, se pode definir TLN como um circuito lógico formado por diversas TLGs. O exemplo da figura 2.1b) mostra uma TLN, equivalente ao circuito lógico tradiconal de 2.1a).

Figura 2.1: (a) Representação de uma função booleana com portas AND e OR. (b) Representação da mesma função booleana em uma TLN



Fonte: (ZHANG et al., 2005)

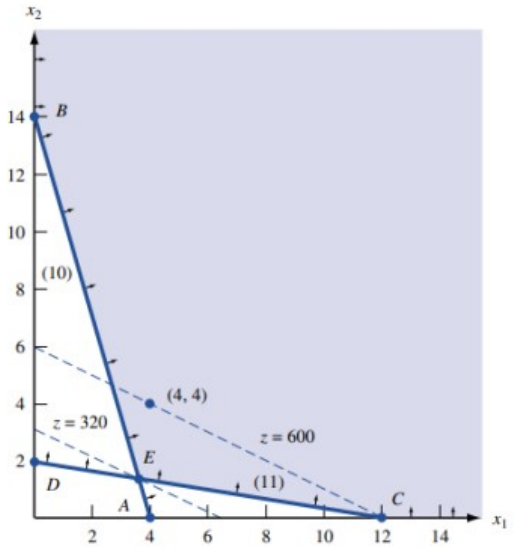
2.8 ILP

Programação Linear (LP, do inglês, Linear Programming) é qualquer problema de otimização em que a função objetivo e suas restrições são lineares. Veja o exemplo da figura 2.2. Neste caso, há uma função z que se quer minimizar, e as restrições são dadas pelas inequações. A região em azul do gráfico mostra a região com todos os pontos que

satisfazem as inequações. Como o problema quer o valor mínimo de z (em problemas de otimização, ou queremos maximizar ou minimizar algo), visualizando o gráfico é possível ver que o menor z acontece no ponto E, onde $z = 320$.

Figura 2.2: Exemplo de um problema LP

$$\begin{aligned} \min z = \; & 50x_1 + 100x_2 \\ \text{s.t.} \quad & 7x_1 + 2x_2 \geq 28 \\ & 2x_1 + 12x_2 \geq 24 \\ & x_1, x_2 \geq 0 \end{aligned}$$

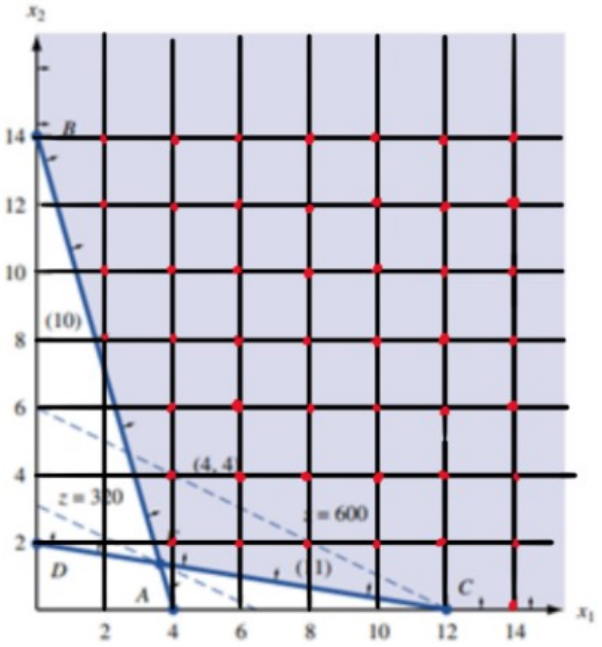


Fonte: (WINSTON, 1994)

Programação Linear Inteira (ILP, do inglês, Integer Linear Programming) é uma programação linear em que as variáveis envolvidas são obrigatoriamente inteiras. Considerando que exatamente o mesmo problema anterior seja um problema de ILP, a situação em que $z = 320$, as variáveis não são inteiras, por tanto esta não é a solução. O exemplo da figura 2.3 mostra o mesmo gráfico, mas apenas os pontos vermelhos são candidatos à solução, pois estão dentro da região de solução e suas coordenadas são inteiras. Nesta situação, o ponto que minimiza a solução é o de coordenada (4,2), e, neste caso, $z = 400$.

É possível resolver problemas de identificação de TF usando ILP (ZHANG et al., 2005). A partir da tabela-verdade (ou de seu formato ISOP), é possível gerar as inequações que serão as restrições do problema, e a função que sempre se quer minimzar é $w_1 + w_2 + ... + w_n + T$, pois se quer pesos w_i e o valor threshold T (que são inteiros) sejam os menores possíveis,

Figura 2.3: Exemplo de um problema ILP



Fonte: (WINSTON, 1994), com modificações do autor

2.9 Parâmetros de Chow

Dada uma função booleana $f(x_1, x_2, ..., x_n)$, o parâmetro de Chow de uma variável x_i é dado por $p_i = 2m_i - 2n_i$, onde m_i é o número de vezes, na tabela verdade, em que $x_i = 1$ e $f(x_i) = 1$, e n_i é o número de vezes em que $x_i = 0$ e $f(x_i) = 1$. A figura 2.4 mostra a tabela-verdade de $f = x_1x_2 + x_1x_3x_4$ e o cálculo dos parâmetros de Chow de cada uma das variáveis.

Figura 2.4: Parâmetros de Chow das variáveis de $f = x_1x_2 + x_1x_3x_4$

x_1	x_2	x_3	x_4	f
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	0
0	1	0	1	0
0	1	1	0	0
0	1	1	1	0
1	0	0	0	0
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	1
1	1	0	1	1
1	1	1	0	1
1	1	1	1	1

$m_1 = 5$ and $n_1 = 0$
$p_1 = 10$

$m_2 = 4$ and $n_2 = 1$
$p_2 = 6$

$m_3 = 3$ and $n_3 = 2$
$p_3 = 2$

$m_4 = 3$ and $n_4 = 2$
$p_4 = 2$

2.10 Peso Crítico (CW, do inglês, Critical Weight)

No processo de atribuição de valores para os pesos das variáveis, pode acontecer que algumas inequações são falsas (inconsistentes) com aqueles valores. Para reajustar os pesos e tentar tornar as inequações verdadeiras, (LIU et al., 2019) traz a definição de pesos críticos (CWs).

CWs são os pesos w_i em que d(w_i) > 0. Veja as próximas duas definições:

G(w_i): número de vezes em que w_i aparece no lado maior (greater side) em todas as inequações falsas.

L(w_i): número de vezes em que w_i aparece no lado menor (lesser side) em todas as inequações falsas.

Com isto, a fórmula d(w_i) é apresentada abaixo:

d(w_i) = G(w_i) - L(w_i)

3 TRABALHOS RELACIONADOS

Dos três campos de estudo mencionados na introdução, dois estão interligados. Para o desenvolvimento lógico de um circuito integrado baseado em lógica threshold, tanto a etapa de identificação de TF quanto a de síntese de rede de TLGs são necessárias. Se dada função booleana é identificada como TF, uma única porta TLG é suficiente para implementá-la e, portanto, não há necessidade de sintetizar uma rede. No entanto, se uma função booleana é identificada como não-TF, então é preciso criar uma rede de TLGs – chamada de TLN - para implementar esta função booleana. Por este motivo, mesmo em alguns trabalhos cujo foco é a síntese de TLN, eles de certa forma, contribuíram no estudo de identificação de TF.

Tabela 3.1: Trabalhos Relacionados

Pesquisa	*Avedillo (2004)*	*Zhang (2005)*	*Subirats (2008)*	*Gowda (2011)*	*Palaniswamy (2012)*	*Neutzling (2018)*	*Liu (2019)*
TF				X	X	X	X
TLN	X	X	X	X			

A tabela 3.1 mostra sete trabalhos e seus respectivos focos. Estes são os principais trabalhos na linha histórica do estudo de identificação de TFs. Ao longo desta seção, será explorada a contribuição de cada um dos seis primeiros para chegar até o sétimo trabalho, que é o estado-da-arte.

3.1 Avedillo (2004)

Conforme já mencionado, existem três ramificações na área de threshold logic, mas duas delas não são mutuamente excludente: identificação de TF e síntese de TLN (caso uma função booleana precise ser representada por mais de uma TF).

María J. Avedillo e José M. Quintana se voltaram ao estudo de síntese de TLN. Eles propuseram uma ferramenta de síntese de TLN voltada para circuitos RTD. No entanto, até o momento não existia um método de identificação de TF. (AVEDILLO; QUINTANA, 2004) foi o primeiro trabalho a se preocupar em, primeiro, identificar TFs. O método deles é baseado em *Multi-Valued Decision Diagrams*, que, é uma solução exaustiva, ou seja, seu custo computacional é alto. É uma solução que não foi aproveitada nem pelo trabalho seguinte (ZHANG et al., 2005), mas vale destacar por ter sido o primeiro método de identificação de TFs.

3.2 Zhang (2005)

(ZHANG et al., 2005) estava concentrado na síntese de TLN, mas foi o primeiro trabalho a criar passos na identificação de TF que foi aproveitado pelo estado-da-arte (LIU et al., 2019). Recorreu ao trabalho teórico, matemático de (MUROGA, 1971) e fez uso do conceito de unateness e de ILP para identificar TFs.

Demonstrado por (MUROGA, 1971), é sabido que uma função threshold é unate. Por tanto, se for provado que determinada função booleana não é unate, já está provado que ela não é uma TF. E este é o primeiro passo do algoritmo de (ZHANG et al., 2005).

Da definição de TF, é possível gerar inequações a partir das linhas da tabela-verdade. Para exemplificar, observe a tabela-verdade da AND2 (cuja função booleana é $f = x_1 \cdot x_2$) que está mostrada na tabela do 3.2.

Tabela 3.2: Tabela-Verdade da função f que representa AND2

x_1	x_2	f
0	0	0
0	1	0
1	0	0
1	1	1

Pela definição de TF vista no capítulo anterior, podemos criar uma inequação por linha da tabela-verdade:

Linha 0: $w_1 \cdot 0 + w_2 \cdot 0 < T$ (já que $f = 0$)

Linha 1: $w_1 \cdot 0 + w_2 \cdot 1 < T$ (já que $f = 0$)

Linha 2: $w_1 \cdot 1 + w_2 \cdot 0 < T$ (já que $f = 0$)

Linha 3: $w_1 \cdot 0 + w_2 \cdot 1 \geq T$ (já que $f = 1$)

Com isso, é possível montar o sistema de inequação abaixo.

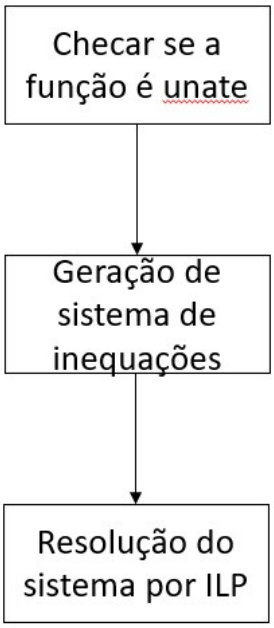
$$\begin{cases} 0 < T \\ w_2 < T \\ w_1 < T \\ w_1 + w_2 \geq T \end{cases}$$

Em geral, se quer que o valor threshold e os pesos sejam inteiros positivos e, por isso, é possível resolver este sistema com ILP. E este é o último passo de (ZHANG et al., 2005) para determinar a TLG equivalente. Conforme visto no capítulo de Fundamenta-

ção Teórica, podem ser infinitas as soluções que satisfaçam a expressão a ser minimizada. Entretanto, se busca sempre os menos valores de pesos e threshold possíveis, por isso é possível entender este sistema de inequações como restrições de um problema de minimização do ILP. A função a ser minimizada, deste exemplo, é $minz = w_1 + w_2 + T$.

A figura 3.1 mostra o fluxograma do processo de identificação de TF do trabalho de (ZHANG et al., 2005).

Figura 3.1: Fluxograma de identificação de TF feita por (ZHANG et al., 2005)



3.3 Subirats (2008)

Assim como nos dois trabalhos antecessores, José L. Subirats, José M. Jerez, e Leonardo Franco objetivavam pesquisar na área de síntese de TLN. Mesmo assim, fizeram acréscimos quanto à identificação de TF em relação à (ZHANG et al., 2005).

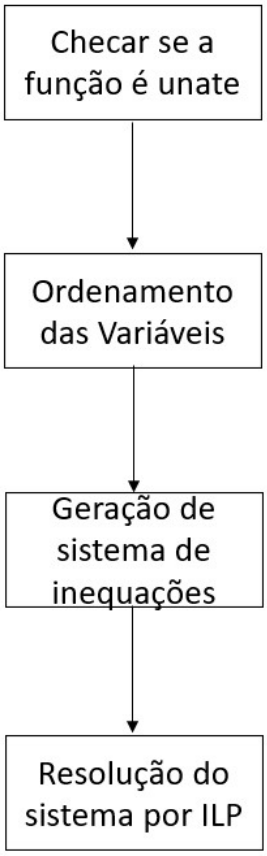
(SUBIRATS; JEREZ; FRANCO, 2008) segue o método de (ZHANG et al., 2005), mas adiciona um passo antes da resolução do sistema de inequações por ILP: ordenamento das variáveis (em relação aos seus pesos). Com isso, é possível diminuir o número de inequações a serem resolvidas, ou seja, o tempo de execução da ILP será menor, o que implica um algoritmo mais otimizado.

Suponha-se que, para uma determinada função booleana, se tenha, dentre outras, as seguintes inequações: $w_1 + w_2 \geq T$ e $w_1 + w_3 \geq T$. Caso se saiba previamente que o ordenamento dos pesos é $w_1 > w_2 > w_3$, a segunda inequação se torna desnecessária,

pois se $w_1 + w_2 \geq T$ e $w_2 > w_3$, por transitividade, $w_1 + w_3 \geq T$. Isto significa que a segunda inequação é redundante e, portanto, pode ser descartada do sistema de inequações a ser resolvida por ILP.

A figura 3.2 mostra o fluxograma do processo de identificação de TF do trabalho de (SUBIRATS; JEREZ; FRANCO, 2008). Ao compará-la com a figura 3.1, se pode conferir que a única diferença entre os dois trabalhos (em relação à identificação de TF) é o acrescimento do passo de ordenamento de variáveis, que otimiza a resolução do sistema de inequações no passo seguinte.

Figura 3.2: Fluxograma de identificação de TF feita por (SUBIRATS; JEREZ; FRANCO, 2008)



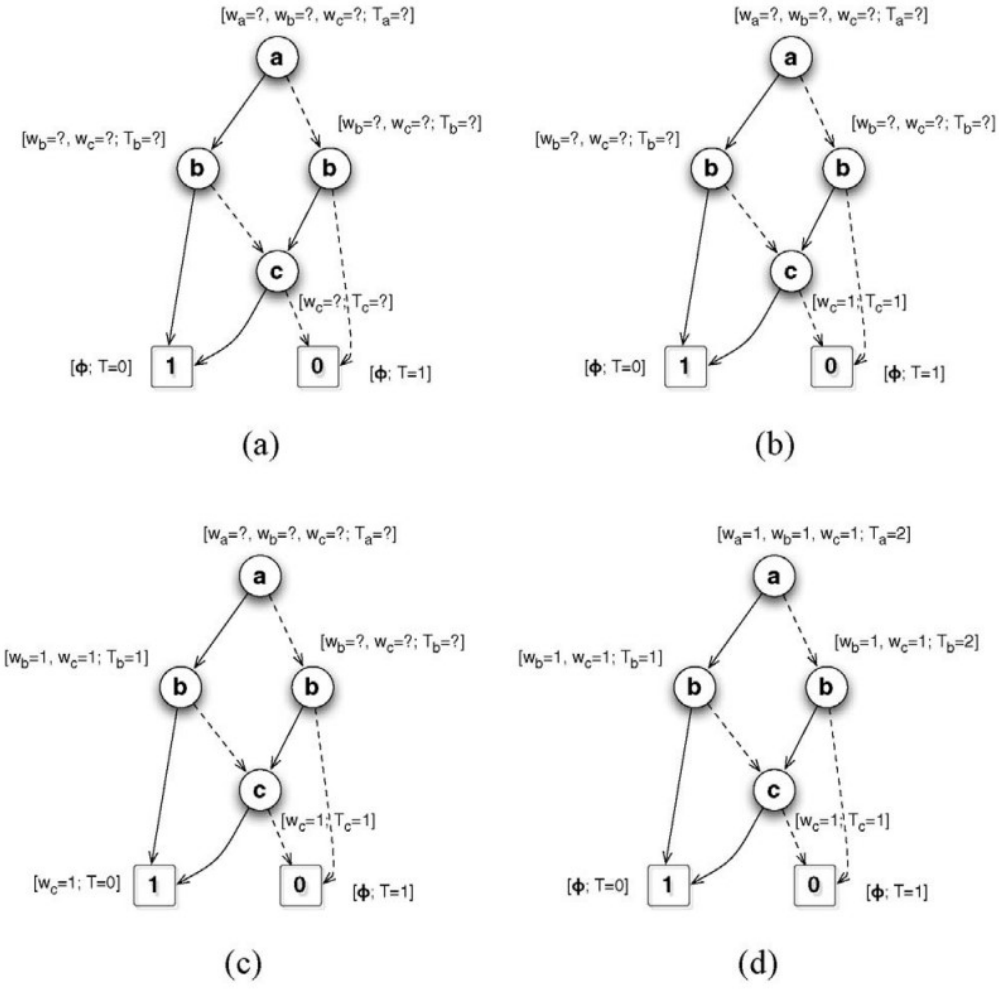
3.4 Gowda(2011)

(GOWDA et al., 2011) é o primeiro trabalho a de fato se aprofundar no estudo de identificação de TFs. Sem considerar, em um primeiro momento, as possíveis simplificações devidas ao ordenamento de variáveis, o número de inequações a serem resolvidas pela ILP equivale ao número de linhas da tabela-verdade da função. Se n é o número de entradas, então o número de linhas da tabela-verdade (e, portanto, o número de ine-

quações) é 2^n. Para valores pequenos de n, ILP é uma excelente forma de resolver as inequações. No entanto, conforme n cresce, o custo computacional cresce exponencialmente, o que faz com que a ILP seja impraticável. Para se ter uma noção, o estado-da-arte, como será visto a seguir, identifica todas as TFs de até oito variáveis. Caso quiséssemos identificar via ILP, precisaríamos resolver um sistema de inequações de $2^8 = 256$ inequações (por TF). Para identificar as 2.700.791 classes NP-equivalentes de TFs de até oito variáveis (MUROGA, 1971), seriam geradas, ao todo, quase 700.000.000 de inequações.

Para buscar contornar este problema da impraticabilidade de identificação de TFs via ILP, (GOWDA et al., 2011) propôs o primeiro método que não usa ILP e mostra uma nova heurística para determinar se uma função é threshold ou não. É baseado em BDD (diagrama de decisão binária). Incialmente, o dicionário de valores de pesos e threshold é [a:?; b:?, c:?; T:?]. Percorre-se a árvore até as folhas (que são sempre os valores binários 0 e 1), e volta-se recursivamente até a raiz com os valores dos pesos e o threshold. O procedimento todo para este exemplo é mostrado na figura 3.3.

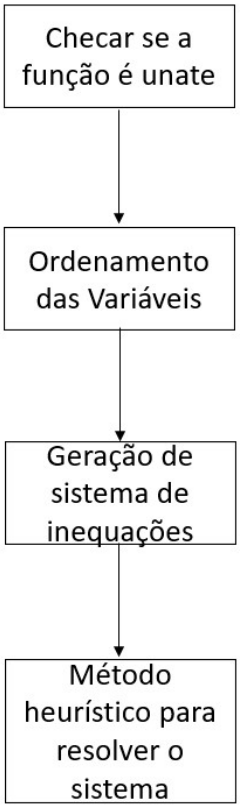
Figura 3.3: Método heurístico de (GOWDA et al., 2011) para $f = ab + bc + ca$



Fonte: (GOWDA et al., 2011)

A figura 3.4 mostra o fluxograma de como (GOWDA et al., 2011) identifica TF. Ao comparar com a figura 3.2, é possível ver que a única diferença é que as inequações não são mais resolvidas via-ILP, mas sim por um método heurístico.

Figura 3.4: Fluxograma de identificação de TF feita por (GOWDA et al., 2011)

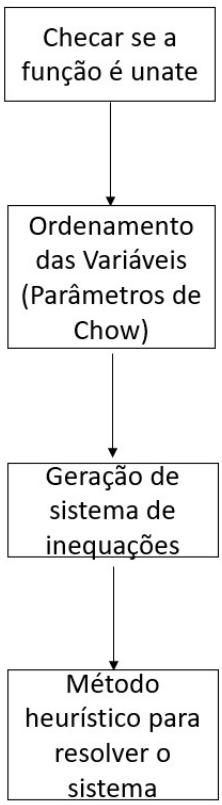


3.5 Palaniswamy (2012)

A grande contribuição de (PALANISWAMY; GOPARAJU; TRAGOUDAS, 2012), em relação ao trabalho de (GOWDA et al., 2011), é o fato de usar parâmetros de Chow para identificar o ordenamento das variáveis de entrada. O ordenamento dos parâmetros de Chow determina o ordenamento de variáveis (MUROGA, 1971). Por exemplo, se uma função booleana tem três variáveis a, b e c, cujos pesos são, respectivamente, w_a, w_b e w_c e cujos parâmetros de Chow são p_a, p_b e p_c, respectivamente. Se $p_a > p_b > p_c$, então obrigatoriamente $w_a > w_b > w_c$. Ou seja, o ordenamento das variáveis (VWO) é [a,b,c]

A figura 3.5 mostra o fluxograma do trabalho de (PALANISWAMY; GOPARAJU; TRAGOUDAS, 2012) em relação à identificação de TF. Ao comparar com a figura 3.4, se observa que a diferença é apenas na maneira de determinar o VWO.

Figura 3.5: Fluxograma de identificação de TF feita por (PALANISWAMY; GOPARAJU; TRAGOUDAS, 2012)



3.6 Neutzling (2018)

A heurística usada por (GOWDA et al., 2011) não era muito eficiente. Das classes NP-equivalentes de TF, ele conseguiu determinar todas com até 3 variáveis (que totalizam 8 classes). A partir de quatro variáveis, o método não tem 100% de aproveitamento. (NEUTZLING et al., 2018) propõe, então, uma heurística mais eficiente, capaz de identifcar todas as TFs com até seis variáveis (e quase 80% das de sete variáveis).

Basicamente, foram três as principais contribuições deste trabalho:

1. Procedimento baseado no formato ISOP para determinar o VWO;
2. Simplificação das inequações;
3. Um novo método heurístico para determinar os pesos das variáveis.

3.7 Liu (2019)

(LIU et al., 2019) atinge o estado-da-arte ao fazer duas melhorias em relação ao trabalho de (NEUTZLING et al., 2018). A primeira contribuição é o aperfeiçoamento

da simplificação das inequações. Apesar de o método de (NEUTZLING et al., 2018) simplificar significativamente o número de inequações do sistema, ainda assim é possível, em alguns casos, haver redundâncias. Para resolver estas redundâncias, (LIU et al., 2019) implementa dois passos adicionais entre os feitos por (NEUTZLING et al., 2018). A segunda evolução é referente ao procedimento heurístico de atribuir valores aos pesos das variáveis. Enquanto a heurística de (NEUTZLING et al., 2018) consegue detectar todas as TFs de até seis variáveis, a de (LIU et al., 2019) consegue até oito.

4 PROPOSTA E METODOLOGIA

Como visto no capítulo anterior, (NEUTZLING et al., 2018), que foi realizado na Universidade Federal do Rio Grande do Sul (UFRGS), foi um marco por criar uma heurística muito mais eficiente que a anterior de (GOWDA et al., 2011). (LIU et al., 2019) trabalha em cima da proposta feita na UFRGS e a melhorou em dois aspectos. A proposta deste trabalho é implementar o trabalho de (LIU et al., 2019) para que tenhamos um código com a versão mais atualizada possível em relação à identificação de TF. Com isso, em um eventual trabalho de pós-graduação, será possível, a partir do presente trabalho, melhorar o estado-da-arte.

A metodologia foi dividida em duas partes:

1. Estudo aprofundado de (NEUTZLING et al., 2018) e implementá-lo.
2. Estudo aprofundado de (LIU et al., 2019) e implementar as modificações feitas em cima do trabalho anterior.

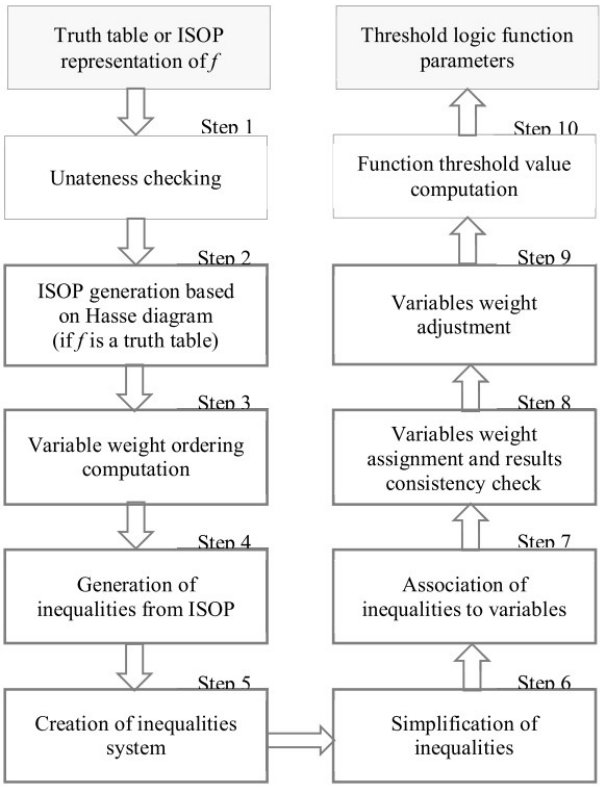
O restante deste capítulo se proporá a trazer os detalhes técnicos destes dois trabalhos.

4.1 Neutzling

A figura 4.1 resume o fluxo de identificação de TF feita por (NEUTZLING et al., 2018). O primeiro passo é checar se a função é unate. Caso a função não seja unate, já é sabido que ela não é TF (MUROGA, 1971). O segundo passo é apenas um tratamento para gerar a ISOP via diagrama de Hasse caso a entrada não seja uma ISOP. Este foi um passo em que o autor desta monografia não implementou, por duas razões:

1. Este pedaço do código não seria aproveitado, pois (LIU et al., 2019) opta por checar a "monotonicidade"(unateness) após já ter o formato ISOP.
2. O autor prefere gerar o formato ISOP pelo método Quine-McCluskey, e não por Diagrama de Hasse.

Figura 4.1: Fluxograma de identificação de TF feita por (NEUTZLING et al., 2018)



Fonte: (NEUTZLING et al., 2018)

4.1.1 Determinação do VWO

O terceiro passo é a determinação do VWO. Desde (SUBIRATS; JEREZ; FRANCO, 2008) sabe-se que, conhecendo o VWO, é possível diminuir a quantidade de inequações a serem resolvidas. Por isso, é importante descobrir o VWO antes de resolver o sistema.

Em vez de usar os Parâmetros de Chow para determinar o VWO, (NEUTZLING et al., 2018) utiliza outro critério. A variável com maior peso é associada ao literal que ocorre mais frequentemente nos cubos com menos literais no formato ISOP da função. Em caso de empate, repete-se a contagem para os próximos cubos com menos literais.

Por exemplo, dada a função $f = x_1 \cdot x_2 + x_1 \cdot x_3 \cdot x_4$, x_1 e x_2 aparecem ambos uma vez no único termo de menos literais (dois). No desempate, x_1 aparece uma vez no de três literais, enquanto x_2 não aparece nenhuma vez. x_3 e x_4 não aparecem nenhuma vez no de dois literais, mas ambos aparecem no de três. Portanto, o ordenamento fica: $w_1 > w_2 > w_3 = w_4$.

Este procedimento leva ao mesmo resultado do que fazer por parâmetros de Chow. Para este mesmo exemplo, os parâmetros de Chow seriam $p_a = 10$, $p_b = 6$, $p_c = p_d = 2$ e, portanto, $w_1 > w_2 > w_3 = w_4$.

Como será visto a seguir, o estado-da-arte (LIU et al., 2019) opta por manter o critério dos parâmetros de Chow para determinar o VWO, mesmo que, segundo (NEUTZLING et al., 2018), a complexidade de seu critério é menor.

4.1.2 Geração e Simplificação das Inequações

Para mostrar o procedimento dos passos 4 a 7 (figura 4.1), referentes à geração e simplificação das inequações, o mesmo exemplo $f = x_1 \cdot x_2 + x_1 \cdot x_3 \cdot x_4$ será usado.

A tabela-verdade e as respetivas inequações são mostradas na figura 4.2. Observe que as cinco últimas inequações são aquelas em que a soma de pesos é maior do que o valor threshold (T). Já foi determinado na subseção anterior o VWO: $w_1 > w_2 > w_3 = w_4$.

Observe que as três últimas inequações são redundantes, pois, pelo VWO, elas são maiores do que $w_1 + w_2$. Por transitividade, elas são maiores do que $w_1 + w_2$ e $w_1 + w_2 \geq T$, então o trio obrigatoriamente é maior do que T. Os dois somatórios restantes e que são maiores do que T são inseridos numa lista chamada *greater side*: $w_1 + w_2$ e $w_1 + w_3 + w_4$.

Mas como tirar estas redundâncias? Observe que as cinco últimas linhas da tabela verdade representam a SOP, pois são aquelas em que a função tem valor 1, ou seja, $f = x_1 \cdot x_3 \cdot x_4 + x_1 \cdot x_2 + x_1 \cdot x_2 \cdot x_4 + x_1 \cdot x_2 \cdot x_3 + x_1 \cdot x_2 \cdot x_3 \cdot x_4$. Ao calcular a ISOP, encontraremos o $f = x_1 \cdot x_2 + x_1 \cdot x_3 \cdot x_4$.

Observe que os termos da ISOP são $x_1 \cdot x_2$ e $x_1 \cdot x_3 \cdot x_4$ e que os somatórios que entram no *greater side* são justamente a soma dos pesos de cada termo ($w_1 + w_2$ e $w_1 + w_3 + w_4$). Ou seja, basta olhar para a ISOP para saber quais serão os somatórios que pertencerão ao greater side.

De maneira análoga, determinamos o *lesser side* para somatórios irredundantes que são menores do que T. Para encontrar, para o *lesser side*, é preciso econtrar a ISOP da f negada e repetir o processo. Para este exemplo, três somas entram nesta lista: $w_1 + w_4$, $w_1 + w_3$ e $w_2 + w_3 + w_4$. O resultado é mostrado na figura 4.3.

Por transitividade (figura 4.3), todas as somas do *greater side* são maiores do que o *lesser side*. Permutando todas as possíveis combinações de desiguldades entre as duas listas (greater side > lesser side), são geradas as 6 inequações apresentadas na figura 4.4.

A partir deste ponto, as inequações já estão preparadas para serem simplificadas. Esta simplificação se divide em quatro passos:

1. **Merge dos parâmetros com mesma ordem no VWO**. Neste exemplo, $w_1 > w_2 > w_3 = w_4$, portanto, $w_1 = A, w_2 = B, w_3 = w_4 = C$. As inequações com as variáveis atualizadas são mostradas na figura 4.5.
2. **Eliminar variáveis que aparecem em ambos os lados das inequações** (afinal, se $A + B > A + C$, por exemplo, então $B > A + C - A = C$). O resultado desta simplifação é mostrada na figura 4.6.
3. **Eliminar inequações sem elemenos no *lesser side*** (afinal, se todos os pesos são maiores do que zero, não faz sentido manter no sistema uma inequação como $A > 0$, pois isto é redundante). Por causa disso, as inequações 4 e 5 da figura 4.6 são eliminadas.
4. **Eliminar inequações com um único elemento no *lesser side*** (não é preciso resolver a inequação $B > C$, por exemplo, pois ela é redundante devido ao VWO). Neste caso, as inequações 1,2 e 6 da figura 4.6 são eliminadas.

Figura 4.2: Tabela-verdade e inequações de $f = x_1 \cdot x_2 + x_1 \cdot x_3 \cdot x_4$

x_1	x_2	x_3	x_4	f_3	Inequality
0	0	0	0	**0**	$0 < T$
0	0	0	1	**0**	$w_4 < T$
0	0	1	0	**0**	$w_3 < T$
0	0	1	1	**0**	$(w_3 + w_4) < T$
0	1	0	0	**0**	$w_2 < T$
0	1	0	1	**0**	$(w_2 + w_4) < T$
0	1	1	0	**0**	$(w_2 + w_3) < T$
0	1	1	1	**0**	$(w_2 + w_3 + w_4) < T$
1	0	0	0	**0**	$w_1 < T$
1	0	0	1	**0**	$(w_1 + w_4) < T$
1	0	1	0	**0**	$(w_1 + w_3) < T$
1	0	1	1	**1**	$(w_1 + w_3 + w_4) \geq T$
1	1	0	0	**1**	$(w_1 + w_2) \geq T$
1	1	0	1	**1**	$(w_1 + w_2 + w_4) \geq T$
1	1	1	0	**1**	$(w_1 + w_2 + w_3) \geq T$
1	1	1	1	**1**	$(w_1 + w_2 + w_3 + w_4) \geq T$

Fonte: (NEUTZLING et al., 2018)

Com isso, com esta heurística, a única inequação a ser resolvida é $A > C + C$. Observe que um ILP deveria resolver $2^4 = 16$ inequações, como seria com (ZHANG et al., 2005). Mesmo tratando redundâncias devido ao VWO, como faz (GOWDA et al., 2011) e (PALANISWAMY; GOPARAJU; TRAGOUDAS, 2012), ainda sim seria necessário resolver um sistema de 6 inequações.

Figura 4.3: Inequações geradas a partir da figura 4.2

greater side				*lesser side*
(w_1+w_2)	$\geq$	T	$>$	(w_1+w_4)
$(w_1+w_3+w_4)$	$\geq$	T	$>$	(w_1+w_3)
---		T	$>$	$(w_2+w_3+w_4)$

Fonte: (NEUTZLING et al., 2018)

Figura 4.4: Inequações geradas a partir da figura 4.3

#	Inequality		
1	(w_1+w_2)	$>$	(w_1+w_4)
2	(w_1+w_2)	$>$	(w_1+w_3)
3	(w_1+w_2)	$>$	$(w_2+w_3+w_4)$
4	$(w_1+w_3+w_4)$	$>$	(w_1+w_4)
5	$(w_1+w_3+w_4)$	$>$	(w_1+w_3)
6	$(w_1+w_3+w_4)$	$>$	$(w_2+w_3+w_4)$

Fonte: (NEUTZLING et al., 2018)

Figura 4.5: Inequações com as variáveis atualizadas

#	Inequality		
1	A+B	$>$	A+C
2	A+B	$>$	A+C
3	A+B	$>$	B+C+C
4	A+C+C	$>$	A+C
5	A+C+C	$>$	A+C
6	A+C+C	$>$	B+C+C

Fonte: (NEUTZLING et al., 2018)

Figura 4.6: Resultado das simplificações

#	Inequality		
~~1~~	B	$>$	C
~~2~~	B	$>$	C
3	A	$>$	C+C
~~4~~	C	$>$	0
~~5~~	C	$>$	0
~~6~~	A	$>$	B

Fonte: (NEUTZLING et al., 2018)

4.1.3 Determinação dos Pesos e do Valor Threshold

Os passos 8 e 9 da figura 4.1 são referentes à determinação dos pesos.Faz-se uma atribuição inicial que depende do VWO. Como no exemplo trabalhado, $A > B > C$, a atribuição inicial é $A = 3, B = 2, C = 1$. Se fossem cinco variáveis, tais que $A > B > C > D > E$, então a atribuição inicial seria $A = 5, B = 4, C = 3, D = 2, E = 1$.

A seguir, as variáveis recebem seus valores numéricos no sistema de inequações. Neste exemplo, há apenas uma inequação: $A > C + C$. Com isso, a inequação fica $3 > 1 + 1$. Como a inequação é verdade, o processo de determinação dos pesos está concluído.

No entanto, há exemplos em que se chega em uma (ou mais de uma) inequação falsa. Supondo que se tivesse, para este exemplo, a inequação $B + C > A$. Neste caso, a atribuição inicial implicaria $2 + 1 > 3$, o que não é verdade. Portanto, é preciso reajustar os pesos. Em situações como esta, três passos são seguidos:

1. Incrementar o valor da variável e das variáveis maiores do que ela, em uma unidade. $incremento = \frac{-delta}{delta - delta'}$
2. Computar o novo delta, chamado de delta'.
3. Se delta = delta', então o incremento é desfeito, e a inequação é mantida como inconsistente, e o algoritmo continua o procedimento para as próximas inequações falsas.

O delta é a diferença entre o greater side o lesser side. No exemplo original, em que $A > C + C$ ficou $3 > 1 + 1$, o delta $= 3 - 2 = 1 > 0$, portanto, não é preciso de ajuste. No segundo exemplo proposto, em que $B + C > A$ ficou $2 + 1 > 3$, delta $= 3 - 3 = 0$. Quando delta é menor ou igual a zero, significa que a inequação precisa de reajuste.

Repete-se este processo até ter todas as inequações consistentes.

É importante ressaltar que, exceto a atribuição inicial, este procedimento também não foi implementado, uma vez que o passo a passo de (LIU et al., 2019) é bem diferente. É neste ponto em que há a grande contribuição do atual estado-da-arte em relação ao trabalho de (NEUTZLING et al., 2018).

Por fim, já conhecendo os valores de cada peso, falta apenas determinar o valor threshold T. No exemplo, $A = 3, B = 2, C = 1$ ($w_1 = 3, w_2 = 2, w_3 = w_4 = 1$). Ao substituir estes valores de pesos nas inequações originais da figura 4.4, se obtém a figura

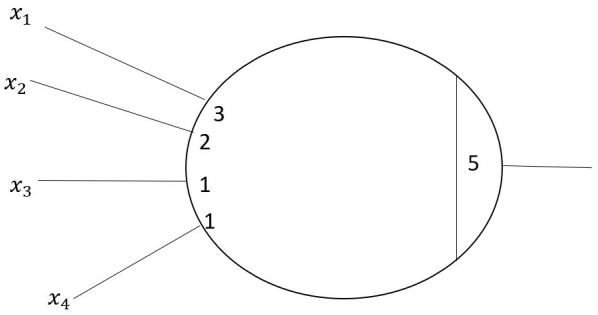
4.7. O valor threshold precisa ser maior do que qualquer soma do *lesser side*. Portanto, o menor valor threshold possível é aquele em que ele é uma unidade maior do que a maior soma do lesser side. Portanto, para este exemplo, $T = 4 + 1 = 5$. Com isso, a TLG que representa esta TF é mostrada na figura 4.8.

Figura 4.7: Inequações com os valores numéricos dos pesos

#	Inequality		
1	(3+2)	>	(3+1)
2	(3+2)	>	(3+1)
3	(3+2)	>	(2+1+1)
4	(3+1+1)	>	(3+1)
5	(3+1+1)	>	(3+1)
6	(3+1+1)	>	(2+1+1)

Fonte: (NEUTZLING et al., 2018)

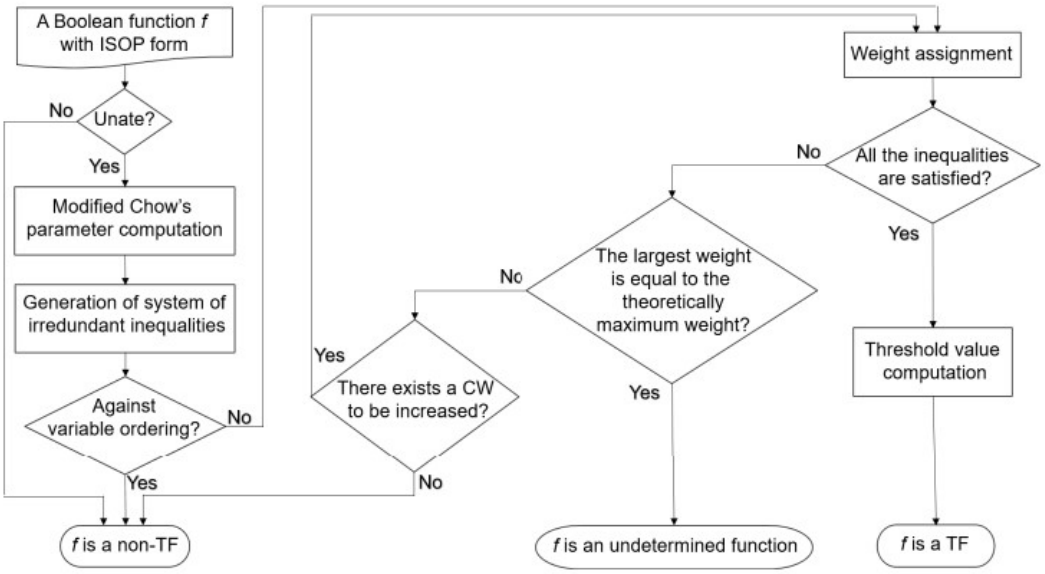
Figura 4.8: TLG da função $f = x_1 \cdot x_2 + x_1 \cdot x_3 \cdot x_4$



4.2 Liu

A figura 4.9 resume o fluxo de identificação de TF feito por (LIU et al., 2019). Como é possível ver, a entrada é sempre uma função boolena no seu formato ISOP, ou seja, o passo do Diagrama de Hasse feito por (NEUTZLING et al., 2018) não é necessário aqui. Os passos de checar a monotonicidade, identificar VWO e gerar as inequações seguem o mesmo raciocínio do trabalho de (NEUTZLING et al., 2018) (exceto pelo fato de usar parâmetros de Chow para determinar o VWO). As diferenças aparecem nos passos para simplificar as inequações e na maneira de atribuir os valores para os pesos das entradas.

Figura 4.9: Fluxograma de identificação de TF feita por (LIU et al., 2019)



Fonte: (LIU et al., 2019)

4.2.1 Simplificação das Inequações

Recapitulando, a geração de inequações simplificadas de (NEUTZLING et al., 2018) é dividida em três etapas:

1. Geração das inequações a partir da ISOP.
2. Construção do sistema de inequações (permutar greater side e lesser side)
3. Simplificação das inequações.

(LIU et al., 2019) também segue estes três passos, mas entre eles é adicionado dois para eliminar eventuais redundâncias. Por mais que (NEUTZLING et al., 2018) diminui significativamente o tamanho do sistema de inequações, ainda sim podem haver algumas redundâncias, que são eliminadas por (LIU et al., 2019). Entre as etapas 1 e 2, é adicionado um novo passo para retirar algumas destas redundâncias, chamado de "Remoção de Somas Redundantes de Pesos". Para exemplificar, supõe-se que o VWO seja $w_1 > w_2 > w_3 = w_4 > w_5 = w_6$ e que, dentre outras, se tenha as seguintes somas no *greater side*: $w_1 + w_2$ e $w_2 + w_3$. No entanto, pelo VWO, é sabido que $w_1 + w_2 > w_2 + w_3$, umas vez que $w_1 > w_3$. Portanto, é possível eliminar a soma $w_1 + w_2$, uma vez que, se uma expressão é menor do que $w_2 + w_3$, por transitividade obrigatoriamente será menor do que $w_1 + w_2$ também.

Entre as etapas 2 e 3, um novo passo é adicionado para retirar outras redundâncias, chamado de "Remoção das Inequações Redundantes". Vamos supor que o VWO seja

$w_1 > w_2 = w_3 > w_4$ e que uma das inequações construídas na etapa 2 seja $w_1 + w_2 > w_3 + w_4$ ($w_1 + w_2$ estava no greater side, enquanto $w_3 + w_4$ estava no lesser side). Pelo VWO, obrigatoriamente w_1+w_2 é menor do que w_3+w_4, pois ambos w_3 e w_4 são menores (ou iguais) a w_1 e w_2, ou seja, não há necessidade de ter esta inequação no sistema. É diferente de, por exemplo, caso houvesse, a inequação $w_1 + w_4 > w_2 + w_3$. Ambos w_2 e w_3 são maiores do que w_4. mas ambos também são menores do que w_1.Neste caso, a inequação não é redundante (e, portanto, não poderia ser descartada), pois não sabemos o quão w_1 é maior em relação à dupla e quão w_4 é menor.

4.2.2 Atribuição dos Valores dos Pesos

É neste ponto que está a grande diferença entre os dois trabalhos, e que torna (LIU et al., 2019) mais eficiente, conseguindo identificar todas as funções threshold de até oito variáveis. O procedimento de atribuição inicial dos pesos é exatamente o mesmo de (NEUTZLING et al., 2018). A seguir, verifica-se, substituindo as variáveis por seus valores numéricos, se todas as inequações são consistentes. Caso positivo, então já são encontrados os menores valores de pesos. Basta fazer a atribuição do valor threshold (cujo processo também é igual a (NEUTZLING et al., 2018)) que os parâmetros da TLG correspondente estão determinados. Caso negativo, é preciso fazer uma nova retribuição de valores para os pesos.

Pensando-se apenas em ILP, podem ser infinitas as possibilidades de atribuição de valores para os pesos. No entanto, (MUROGA, 1971) mostra que existe um limite máximo para atribuição dos pesos, por motivos de engenharia para a construção da TLG. Caso um dos pesos seja superior a este limite, é impossível implementar a TLG. Este limite é dado pela expressão abaixo: $0 < w_i \leq 2 \cdot \frac{n+1}{4}^{\frac{n+1}{2}}, i = 1, 2, ..., n$

Para funções de 2 a 9 variáveis, por exemplo, estes limites superiores são, respectivamente: 1, 2, 3, 6, 14, 32, 76 e 195.

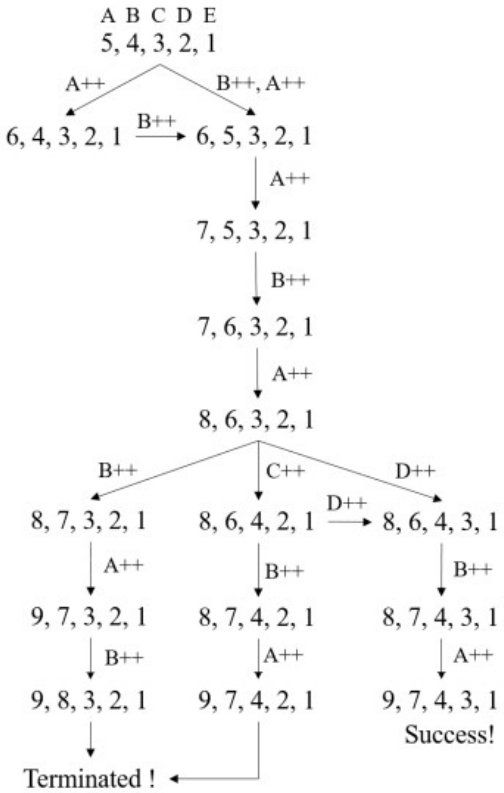
Por este motivo, antes de fazer a retribuição, (LIU et al., 2019) checa se o maior peso até o momento já é igual a este peso máximo. Caso isso tenha acontecido, então a função é classificada como indeterminada, pois não é mais possível continuar com o algortimo e determinar se a função é threshold ou não. Caso negativo, o procedimento continua.

Após esta checagem do limite máximo, o programa verifica se há mais pesos críticos (CWs) a serem incrementados. Os possíveis pesos w_i a serem incrementados são

aqueles em que $d(w_i) > 0$, conforme explicado no capítulo de Fundamentação Teórica. Caso todos os $d(w_i)$ sejam menores ou iguais a zero, então não há mais o que incrementar e, portanto, não há valores que satisfaçam o conjunto de inequações. Com isso, a função booleana é classificada como não-threshold. Caso haja $d(w_i) > 0$, então o processo de rebalenceamento dos pesos começa. Para cada peso que $d(w_i) > 0$, cria-se um cenário novo em que se incrementa em 1 unidade o valor do peso. Para cada cenário, o procedimento é repetido.

Para ficar mais claro, a figura 4.10 traz um exemplo. Vamos começar do cenário em que os pesos são $A = 8, B = 6, C = 3, D = 2, E = 1$. Para este exemplo, foi calculado que os CWs são B, C e D. Para cada um destes cenários, repetimos o procedimento até que chegamos no estado de sucesso (é threshold) ou de término (a função não é threshold ou é indeterminada). Continuando o caminho em que incrementamos B ou C, chegaremos em término. No entanto, se incrementarmos D, chegaremos, depois de repetir o procedimento mais duas vezes, em sucesso, concluindo que os pesos desta TF são $A = 9, B = 7, C = 4, D = 3, E = 1$.

Figura 4.10: Exemplo de uma retribuição de pesos

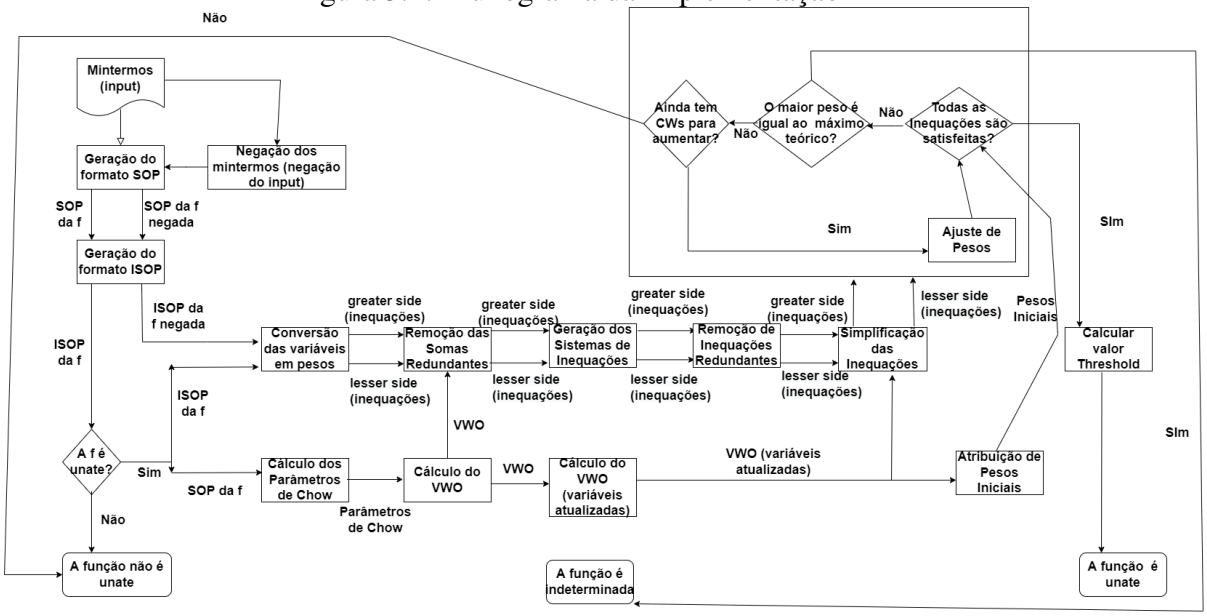


Fonte: (LIU et al., 2019)

5 IMPLEMENTAÇÃO

Para implementar o trabalho de (LIU et al., 2019), foi usada a linguagem C++. Todo o programa foi testado e executado no Microsoft Visual Studio 2022. Diversas bibliotecas foram utilizadas, sendo a mais importante a <*list*>, pois toda a estratégia foi baseada em listas como estrutura de dados. Além disso, é importante esclarecer que inicialmente o autor até cogitou usar princípios de orientação a objetos, mas percebeu que não fazia muito sentido, pois o fluxo se resume em receber uma sequência de zeros e uns (os mintermos da função booleana de entrada), fazer determinadas operações e ao final dizer se aquela sequência é uma função threshold ou não. Por este motivo, o autor optou por programação imperativa, e não por por programação orientada a objetos. O fluxograma da implementação é mostrada na figura 5.1.

Figura 5.1: Fluxograma da Implementação



A grande maioria das variáveis usadas são listas. Algumas são listas de inteiros, e outras listas destas listas de inteiros. Como é possível ver na figura 5.1, a entrada do programa é uma lista de inteiros, onde cada inteiro representa um mintermo m_i.

O algoritmo 1 mostra como gerar o formato SOP. Em resumo, se determinado mintermo é 1, então a linha da tabela-verdade correspondente a este mintermo é inserido na SOP. Por exemplo, supondo que a lista (de inteiros) de mintermos seja [1, 0, 1, 0]. Então a lista (de lista de inteiros) de linhas tabela-verdade é [00,01,10,11]. A primeira lista de inteiros (a linha) da tabela-verdade corresponde ao primeiro termo, e assim sucessivamente, até o final de ambas as listas. Vale destacar que, como a quantidade de

mintermos é justamente a quantidade de linhas da tabela verdade, ambas as listas têm o mesmo tamanho. Neste exemplo, apenas o primeiro e o terceiro mintermos são 1, então a primeira e terceira linhas da tabela-verdade são inseridas na SOP. Com isso, a lista (de lista de inteiros) da SOP é [00,10].

Algorithm 1 Geração do Formato SOP

```
    entrada list <int> minTermos
    Output list <list <int> > sop
  criar list <list <int> > tabelaVerdade(lista das linhas da tabela-verdade)
  Inicializar ponteiros de minTermos e de tabelaVerdade
  while não chegar ao fim de tabelaVerdade e de minTermos do
      if valor do minTermo == 1 then
          Inserir linha da tabela-verdade na sop
      end if
      Incrementar ponteiros
  end while
```

Como é possível ver na figura 5.1, para algumas etapas do fluxograma será necessária a ISOP da função negada. Por este motivo, é preciso que os mintermos da função negada também passem pela geração do formato SOP, descrito no algoritmo 1. A geração dos mintermos negados é mostrado no algoritmo 2 e é bem simples: para todos os mintermos da entrada, basta inverter o valor booleano.

Algorithm 2 Geração dos mintermos da função negada

```
    entrada list <int> minTermos
    Output list <int> minTermosNegados
  Inicializar ponteiro de minTermos
  while não chegar ao fim de tabelaVerdade e de minTermos do
      if valor do minTermo == 1 then
          Adicionar 0 em minTermosNegados
      else
          Adicionar 1 em minTermosNegados
      end if
      Incrementar ponteiro
  end while
```

Para a geração da ISOP, foi usado o algoritmo de Quine McCluskey. Em resumo, o Quine McCluskey é dividido em duas etapas. Na primeira, encontra-se os primos implicantes. Na segunda etapa, chamada de tabela cobertura, encontra-se quais, entre os primos implicantes, são os essenciais, e define-se quais termos, de fato, vão fazer parte da ISOP.

Na primeira etapa, cada termo da SOP é comparado com os demais. Se uma dupla tem distância de Hamming igual 1, ou seja, eles são diferentes em apenas um bit. Então

é feito um *merge* entre estes dois termos, onde o termo diferente se torna um termo *don't care*. Neste implementação, foi definido que o inteiro 9 representaria os *don't cares*. Por exemplo, se dentro da lista SOP há as listas [101] e [100], então é feito um *merge* entre as duas, cujo resultado é [109]. A lista de lista de inteiros com os termos "mergeados"servirá como entrada de uma chamada recursiva do Quine McCluskey. A recursão acaba quando esta lista é vazia. Os termos da entrada que não se combinaram com nenhum outro são inseridos automaticamente na lista chamada primos (referente aos primos implicantes). Executa-se todas as chamadas recursivas, até que se encontre todos os primos implicantes.

No segundo momento, analisa-se os primos e se busca por aqueles que são essenciais. Um primo essencial é aquele que tem um mintermo correspondente que mais nenhum outro primo é correspondido. Para exemplificar, supõe-se que a lista de pimos é [1110,0901,9101.0099].Primeiro, calcula-se quais são os mintermos referentes a cada primo:

- $1110 = 2^3 + 2^2 + 2^1 = 14$.
- 0901
  - $0001 = 2^0 = 1$
  - $0101 = 2^2 + 2^0 = 5$
- 9101
  - $0101 = 2^2 + 2^0 = 5$
  - $1101 = 2^3 + 2^2 + 2^0 = 13$
- 0099
  - $0000 = 0$
  - $0001 = 2^0 = 1$
  - $0010 = 2^1 = 2$
  - $0011 = 2^1 + 2^0 = 3$

Observa-se que o primeiro, terceiro e quarto são os primos essenciais, pois tem pelo menos um mintermo que só eles cobrem. O segundo mintermo é desnecessário, pois ele cobre os mintermos 1 (que é coberto pelo quarto também) e 5 (que é coberto pelo terceiro). Portanto, a lista de lista de inteiros que representa a ISOP nesta situação é [1110, 9101.0099].

Encontrada a ISOP, agora de fato começa a implementação o fluxograma de (LIU et al., 2019).

O próximo passo é checar a monotonicidade (unateness) da função, mostrado no algoritmo 3. Para isso, o formato ISOP será o entrada da função. A estratégia utilizada é checar se, para cada variável, há a presença das duas polaridades (0 e 1). Se todas as variáveis têm polaridade única, então a função é unate. Caso pelo menos uma variável seja binate (apresenta as duas polaridades), então a função não é unate.

Por exemplo, seja a isop [100,191]. Observando os primeiros inteiros de cada termo, referentes a x_1, ambos são 1, ou seja, x_1 é unate porque aparece em apenas na polaridade direta. Analisando os segundos inteiros de cada termo, referentes a x_2, há um 0 e um 9. Como o 9 é um *don't care*, isso significa que x_2 não aparece no segundo termo e, portanto, pode ser ignorado. Portanto, x_2 é unate porque aparece apenas na polaridade negada. Por fim, a variável x_3 aparece em ambas as polaridades (0 no primeiro termo, 1 no segundo). Para armazenas as polaridades, duas listas (de inteiros) são criadas: countOnes e countZeros. Neste exemplo, CountOnes = [2,0,1] e CountZeros = [0,1,1]. Percorre-se paralelamente ambas as listas. Se, em determinado, ambos valores são diferentes de zero, então significa que a variável não é unate. Por exemplo, na primeira posição, em CountOnes há um 2, mas um 0 no CountZeros, o que implica que a variável corresponde só tem polaridade direta. Na segunda posição, em CountOnes há um 0, mas um 1 no CountZeros, o que implica que a variável corresponde só tem polaridade inversa. No entanto, aparece na última posição, nenhuma das listas tem 0, o que significa que esta variável aparece em ambas as polaridades. Quando isso acontece, a função não é unate, pois ao menos uma de suas variáveis é binate.

Já sabendo que a função é unate, é possível calcular os parâmetros de Chow (antes seria desnecessário, caso a função não fosse unate). Como já foi discutido, os termos m e n são ambos referentes às linhas da tabela-verdade em que $f = 1$, que são justamente as linhas da tabela-verdade pertencentes à sop. Por isso, basta ver a quantidade de 1s e 0s para cada variável dentro da sop. Por exemplo, se a SOP for [1011,1100,1101,1110,1111]. Analisa-se a primeira variável, isto é, a primeira posição de cada um dos cincos elementos da lista. Todos são 5, nenhum é 0. Portanto, adiciona-se, à lista (de inteiros) do Parâmetro de Chow o valor $2 \cdot 5 - 2 \cdot 0 = 10$. Então se analisa a segunda variável (segunda posição de cada um dos cinco elementos da lista). O número 1 aparece quatro vezes, enquanto o 0 aparece uma. Portanto, adiciona-se à lista do Parâmetro de Chow o valor $2 \cdot 4 - 2 \cdot 1 = 6$. Repetindo o processo para as demais variáveis, a lista (de inteiros) dos Parâmetros de Chow fica [10,6,2,2]. A SOP é a entrada da função que se calcula os Parâmetros de Chow, e a lista dos parâmetros é o output.

Algorithm 3 Unateness

```
    entrada list list «int» isop
    Output bool isUnate
  bool isUnate = true
  list <int> countOnes
  list <int> countZeros
  while não chegar ao fim da isop (lista de lista de inteiro) do
      inicializa ponteiro do termo da isop que está sendo inicializado (i)
      while não chegar ao fim do termo da isop (lista de inteiro) do
          inicializa ponteiro de CountOnes (j)
          inicializa ponteiro de CountZeros (z)
          if valor de i == 1 then
              *j ++
          else
              if valor de i == 0 then
                  *z++
              end if
          end if
      end while
  end while
  while não chegar ao fim de countOnes e CountZeros do
      if valor da posição em CountOnes != 0 AND valor da posição em CountZeros != 0
  then isUnate = false
      end if
  end while
```

A seguir, já é possível fazer o cálculo do VWO. A lista de inteiros dos Parâmetros de Chow é a entrada desta função, e a lista de inteiros com o VWO é a saída desta função. Em resumo, a função busca o maior elemento da lista dos Parâmetros de Chow, para a posição que possui o maior elemento, na mesma posição, na lista de VWO, é atribuído o valor 0. De modo análogo, atribui-se 1 para o segundo maior, e assim sucessivamente. Para o exemplo anterior de [10,6,2,2], a lista de VWO é [0,1,2,2].

É possível,também, calcular o VWO das variáveis "atualizadas". As variáveis "atualizadas"são aqueles em que já há um *merge* entre os pesos que possuem o mesmo Parâmetro de Chow. Neste caso, o VWO das variáveis atualizadas seria [0,1,2]. Em resumo, exclui-se elementos repetidos do VWO.

Algorithm 4 Cálculo dos Parâmetros de Chow

```
    entrada list <list <int> sop
    Output list <int> parametrosChow
  Inicializar ponteiros para o início de cada elemento da lista da sop
  repeat
      m recebe a quantidade de 1
      n recebe a quantidade de 0
      p ←2 ·m − 2 · n
      Inserir p em parametrosChow
      Incrementar ponteiros
  until ponteiros para os elementos chegarem ao final da lista
```

Algorithm 5 Cálculo do VWO

```
    entrada list <int> parametrosChow
    Output list <int> vwo
  list <int> copiaParametrosChow
  int valorOrdem = 0
  itcopiaParametrosChow = copiaParametrosChow.begin
  while itcopiaParametrosChow != copiaParametrosChow.end do
      encontrarPosicaoMaiorValor
      inserirValorOrdemNaPosicaoEquivalenteNoVWO
      valorOrdem++
      itcopiaParametrosChow++
  end while
```

A partir deste estágio, começa o processo de geração e simplificação de inequações. O primeiro passo é transformar listas das ISOPs (de entrada e a negada) em listas de pesos. O processo é bastante simples: zerar os don't cares. Por exemplo, se um termo da ISOP é $x_1 \cdot x_3$, esta lista, dentro da ISOP, é [1,9,1]. Transforma-se em [1,0,1] para indicar que a soma (de pesos) é $w_1 + w_3$. A partir da ISOP da entrada, gera-se a lista

(de lista de inteiros) da soma de pesos maiores do que threshold, que será chamada de *greaterThanThreshold*. A partir da ISOP da negada, gera-se a lista (de lista de inteiros) da soma de pesos menores do que threshold, que será chamada de *lesserThanThreshold*.

Os demais passos seguem rigorosamente a descrição feita no capítulo anterior, nas seções 4.1.2 e 4.2.1. Cada função implementa um dos cinco passos, e cada uma é executada duas vezes: uma para tratar a lista que armazena o somatório dos pesos que são o lado maior das inequações, e outra para tratar a lista do lado menor das inequações.

A penúltima fase do programa é fazer a atribuição dos pesos (algoritmo 6), que engloba as três condicionais e o processo de ajuste de peso que estão dentro do retângulo da figura 5.1. O primeiro passo é rodar a função *allTheInequalitiesAreSatisfied*, que recebe como parâmetro a lista de pesos e as listas referentes aos lados maior e menor das inequações. Esta função pode ser ilustrada com um pequeno exemplo. Supõe-se que *weights* = [4,3,2], *greaterSide*=[[1,0,1],[1,0,0]] e *lesserSide*=[[0,1,0],[0,1,1]]. Para cada elemento do greaterSide e do lesserSide, faz-se uma soma ponderada dos pesos. Portanto, para o *greaterSide*, a lista de soma ponderada é [6,4], e para o lesserSide, a lista de soma ponderada é [3,5]. Isso significa que, para os lados maiores das inequações, a primeira delas a soma dos pesos é 6, enquanto a segunda é 4. De maneira análoga, a soma dos pesos do lado menor da primeira e segunda inequações são, respectivamente, 3 e 5. Para conferir a consistência das inequações, basta comparar, par a par, os valores das listas de somas ponderadas. Para este exemplo, ao comparar os primeiros elementos, se tem 6>3, o que é consistente. No entanto, ao comparar os segundos elementos, se tem 4 > 5, ou seja a soma ponderada do lado menor é maior do que a soma ponderada do lado maior, o que torna a inequação falsa. Quando isso acontece, significa que ao menos uma das inequações está inconsistente e, portanto, é preciso reajustar os pesos e continuar com o procedimento.

O próximo passo é checar se o maior peso até agora atribuído é igual ao peso máximo teórico (cuja fórmula está descrita na subseção 4.2.2). Para isso, existe a função theLargestWeightIsEqualToTheTheoreticallyMaximumWeight que checa se o maior valor de list <int> *weights* é maior do que o limite teórico, calculado pela fórmula apresentada na subseção 4.2.2. Caso seja igual, então a função é dita indefinida. Caso não seja igual, o procedimento continua.

O passo seguinte é checar se existem CWs a serem aumentados. Para cada elemento da lista *weights*, é calculado os "d", conforme descrito na parte de CW da Fundamentação Teórica, e armazenado na lista dCw (função *'calculateCWs*). Se algum ele-

mento de dCw é maior do que 0, então existe CW a ser aumentado (*existsCWToBeIncreased*). Caso esta função retorne falso, então a função é definida como não-threshold. Caso a função retorne verdadeiro, então, para cada variável que é CW (dCW >0), incrementar 1 em seu peso (em list <int> weights) e chamar recursivamente o Weight Assignment com o weight novo.

Algorithm 6 Weight Assignment

entrada list <int> weights, int maxWeight, list <list <int> > greaterSide, list <list <int> > lesserSide
Output list <int> weights
if allTheInequalitiesAreSatisfied(weights,greaterSide,lesserSide) == false **then**
if theLargestWeightIsEqualToTheTheoreticallyMaximumWeight == false **then** calculateCWs
if existsCWToBeIncreased = true **then**
Para cada variável que é CW, incrementar 1 em seu peso e chamar recursivamente o Weight Assignment com o weight novo
end if
end if
end if

O último passo é a atribuição do valor threshold (algoritmo 7). Como já foi visto, o valor threshold precisa ser maior do que o lado menor de qualquer inequação. Por isso, basta incrementar uma unidade à maior soma do lado menor e atribuir este valor para threshold.

Algorithm 7 Valor Threshold

entrada list <int> weights list <list <int> > lesserSide
Output int T
Para cada elemento de lesserSide, fazer soma ponderada com weights
Escolher o maior valor da soma ponderada (highValue)
T $\leftarrow highValue + 1$

O código do trabalho pode ser encontrado em https://github.com/lucashagemeister/tcc.

6 RESULTADOS EXPERIMENTAIS (VALIDAÇÃO)

Após testar o código para algumas funções booleanas simples, o autor testou para as funções booleanas de exemplo dos trabalhos de (NEUTZLING et al., 2018) de (LIU et al., 2019). Para cada uma destas funções, o autor inseriu a expressão no programa Karma e converteu em tabela-verdade. Os mintermos gerados foram salvos em um arquivo .txt. Foi criada uma função provisória, em que se abria os arquivos txt, lia a sequência de bits, os convertia no formato SOP e seguia o passo a passo do algoritmo, como descrito nos capítulos anteriores. As funções booleanas testadas, na ordem em que aparencem na figura 6.1, foram:

1. $f = x_1 \cdot x_2 \cdot x_3 + x_1 \cdot x_2 \cdot x_4 + x_1 \cdot x_2 \cdot x_5 \cdot x_6 + x_1 \cdot x_2 \cdot x_5 \cdot x_7 + x_1 \cdot x_3 \cdot x_4 + x_1 \cdot x_3 \cdot x_5 \cdot x_6 + x_1 \cdot x_4 \cdot x_5 \cdot x_6 + x_2 \cdot x_3 \cdot x_4 + x_2 \cdot x_3 \cdot x_5 \cdot x_6 + x_2 \cdot x_4 \cdot x_5 \cdot x_6$
2. $f = x_1 \cdot x_2 \cdot x_3 + x_1 \cdot x_2 \cdot x_4 + x_1 \cdot x_2 \cdot x_5 + x_1 \cdot x_2 \cdot x_6 + x_1 \cdot x_2 \cdot x_7 + x_1 \cdot x_3 \cdot x_4 + x_1 \cdot x_3 \cdot x_5 + x_1 \cdot x_3 \cdot x_6 + x_1 \cdot x_4 \cdot x_5 + x_1 \cdot x_4 \cdot x_6 + x_1 \cdot x_5 \cdot x_6 \cdot x_7 + x_2 \cdot x_3 \cdot x_4 + x_2 \cdot x_3 \cdot x_5 + x_2 \cdot x_3 \cdot x_6 + x_2 \cdot x_4 \cdot x_5 + x_2 \cdot x_4 \cdot x_6 + x_2 \cdot x_5 \cdot x_6 \cdot x_7 + x_3 \cdot x_4 \cdot x_5$
3. $f = x_1 \cdot x_2 + x_1 \cdot x_3 \cdot x_4$

Todos os testes tiveram êxitos, pois os resultados obtidos para TLGs são os mesmos descritos nos papers.

Figura 6.1: Exemplos de Outputs

```
Funcao booleana a ser testada:
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 0 1 1 0 0 0 0 0 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 1 1 1
1 1 1 1 1 1 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ]
Threshold Gate:
[ 9 9 7 7 4 3 1 23 ]

Funcao booleana a ser testada:
[ 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 1 0 0 1 1 1
 1 1 1 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 1 0 0 1 1 1 1 1 1 0 0 1 1 1 1 1 1 1
1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ]
We don't know if this boolean function is a Threhold Function or not (undefined)!

Funcao booleana a ser testada:
[ 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 ]
Threshold Gate:
[ 3 2 1 1 5 ]
```

Conforme visto na parte de Fundamentação Teórica, não há a necessidade de testar o programa para todas as funções booleanas possíveis. Se for comprovado que uma função booleana de uma classe NP-equivalente é threshold, então todas as funções desta classe também são (MUROGA, 1971). Foi assim que (PALANISWAMY; GOPARAJU; TRAGOUDAS, 2012), (NEUTZLING et al., 2018) e (LIU et al., 2019) testaram seus trabalhos. Conforme é possível ver na tabela 6.1, o número de execuções de classes NPs

seria bem menor do que testando todas as funções. Por exemplo, há mais de 17 trilhões de funções threshold de 8 variáveis, que estão agrupadas em um pouco mais de 2,7 milhões de classes NP-equivalentes. Encontrar milhões de resultados tem um custo computacional muito menor do que encontrar trilhões deles.

Entretanto, o autor não teve tempo hábil de estudar como gerar classes NP equivalentes e, por isso, recorreu apenas à geração de funções threshold. O problema desta abordagem é que não foi possível gerar todas as funções de 5 a 8 variáveis. A figura 6.2 mostra um pedaço de código de teste (que não entra na versão final) apenas para ver o tempo de execução para gerar todas as funções booleanas. O resultado é mostrado na figura 6.3. Como é possível ver na figura 6.3, o programa só imprime até 4 variáveis. A partir de 5, o programa simplesmente não termina de executar. Em determinado momento, o programa encerra sozinho, sem mostrar o total de funções booleanas de até 5 variáveis.

Figura 6.2: Código para testar tempo de execução de geração de funções booleanas

```
list <list <int>> f;

for (auto i = 1; i <= 8; i++) {
    auto inicio = std::chrono::high_resolution_clock::now();

    f = gerarFuncoesBooleanas(i);
    auto resultado = std::chrono::high_resolution_clock::now() - inicio;
    long long microseconds = std::chrono::duration_cast<std::chrono::microseconds>(resultado).count();
    cout << "Total de Funcoes Booleanadas de ate " << i << " variaveis: " << f.size() << endl;
    cout << "tempo de execucao: " << microseconds << "ms" << endl;
    cout << endl;
}
```

Figura 6.3: Resultado da execução do código da figura 6.2

```
Total de Funcoes Booleanadas de ate 1 variaveis: 4
tempo de execucao: 266ms

Total de Funcoes Booleanadas de ate 2 variaveis: 16
tempo de execucao: 412ms

Total de Funcoes Booleanadas de ate 3 variaveis: 256
tempo de execucao: 5952ms

Total de Funcoes Booleanadas de ate 4 variaveis: 65536
tempo de execucao: 996590ms
```

Por este motivo, o autor buscou encontrar o número de funções threshold testando todas as funções booleanas de até 4 variáveis, cujos valores são encontrados na tabela 6.1.

A tabela 6.2 mostra os valores que desejamos encontrar com os nossos testes. Os resultados são mostrados nas figuras 6.5 a 6.9.

Tabela 6.1: Número de Classes Equivalentes de Até 8 Variáveis

n	1	2	3	4	5	6	7	8
Nº Funções Booleanas	2	10	218	64594	$4.3 \cdot 10^9$	$1.8 \cdot 10^{19}$	$3.4 \cdot 10^{38}$	$1.16 \cdot 10^{77}$
Nº Funções Threshold	2	8	72	1536	86080	$1.4 \cdot 10^7$	$8.2 \cdot 10^9$	$1.7 \cdot 10^{13}$
Nº Classes NPs	1	3	16	380	1227756	$4 \cdot 10^{14}$	-	-
Nº Classes NPs de TFs	1	2	5	17	92	994	28262	2700791

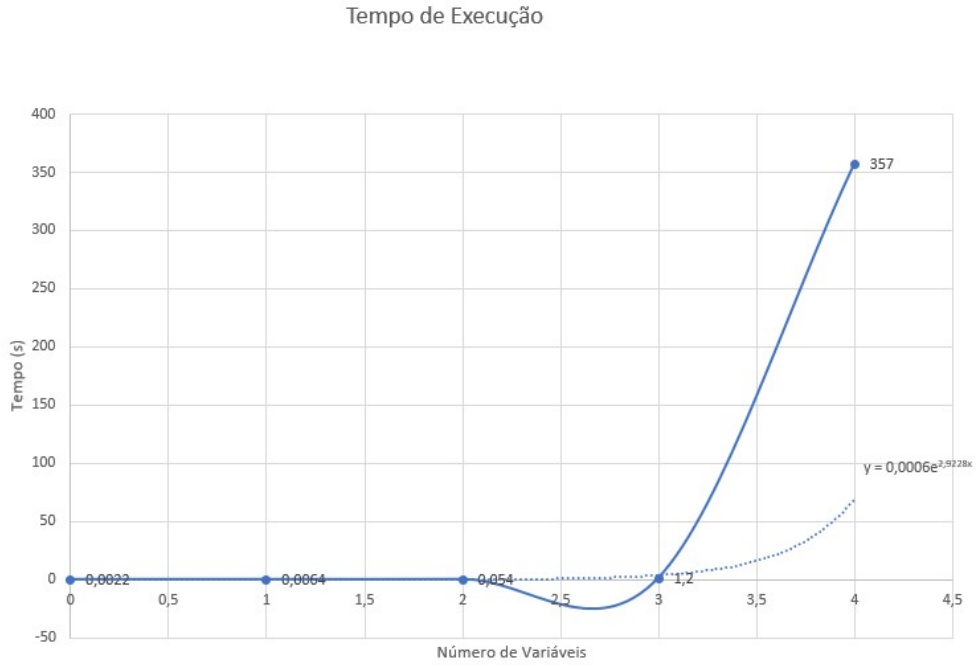
Fonte: (MUROGA, 1971)

Os tempos de execução foram de:

- 0 variável (2 funções booleanas): 2,2ms (1,1ms/função)
- 1 variável (4 funções booleanas): 6,4ms (1,6ms/função)
- 2 variáveis (16 funções booleanas): 54ms (3,4ms/função)
- 3 variáveis (256 funções booleanas): 1,2s (4,6ms/função)
- 4 variáveis (65.536 funções booleanas): 357s = 6min9s (5447ms/função)

Com estas informações, geramos a linha de tendência no gráfico da figura 6.4.

Figura 6.4: Linha de tendência para tempo de execução da geração das funções booleanas



Com a equação da linha de tendência, estima-se que os tempos de execução para 5 a 8 variáveis seriam:

- 5 variáveis: 23 minutos (o que é possível de testar, se o Microsoft Visual Studio não tivesse travado a execução).

- 6 variáveis: 7 horas
- 7 variáveis: 5,5 dias
- 8 variáveis: 105 dias

Isso prova, então, que , para um n a partir de 7, é impraticável testar cada função booleana e checar se é threshold ou não. Por isso a importância de gerar classes NPs (ou NPNs) equivalentes para simplificar a execução.

Os resultados obtidos são mostrados nas figuras 6.4 a 6.8. As figuras 6.5 e 6.6 mostram todas as TFs identificadas e suas TLGs correspondentes. As demais figuras (6.7 a 6.9) mostram apenas as últimas TFs identificadas. Além disso, ao final de todas as execuções, é possível ver o número total de funções que foram geradas, o número de TFs identificadas e o tempo de execução (em microssegundos). É possível ver que todos os valores da tabela 6.2 foram obtidos.

Tabela 6.2: Número de Funções de até 4 Variáveis

n	0	1	2	3	4
Nº funções booleanas de até n variáveis	2	4	16	256	65536
Nº funções threshold de até n variáveis	2	4	14	104	1882

Fonte: (MUROGA, 1971)

Figura 6.5: Execução para funções booleanas de 0 variável

```
Funcao booleana a ser testada:
[ 0 ]
Threshold Gate:
[ ]

Funcao booleana a ser testada:
[ 1 ]
Threshold Gate:
[ 1 ]

Calculando TFs de ate 0 variaveis
Numero de TFs identificadas: 2
total de Funcoes = 2   executado em 2232 us
```

Figura 6.6: Execução para funções booleanas de 1 variável

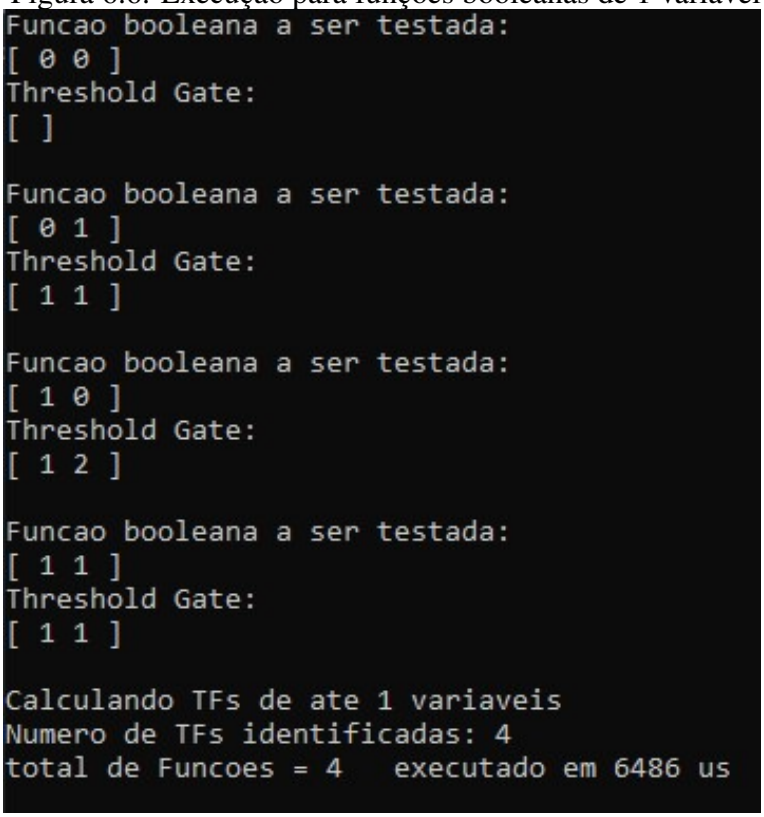


Figura 6.7: Execução para funções booleanas de 2 variáveis

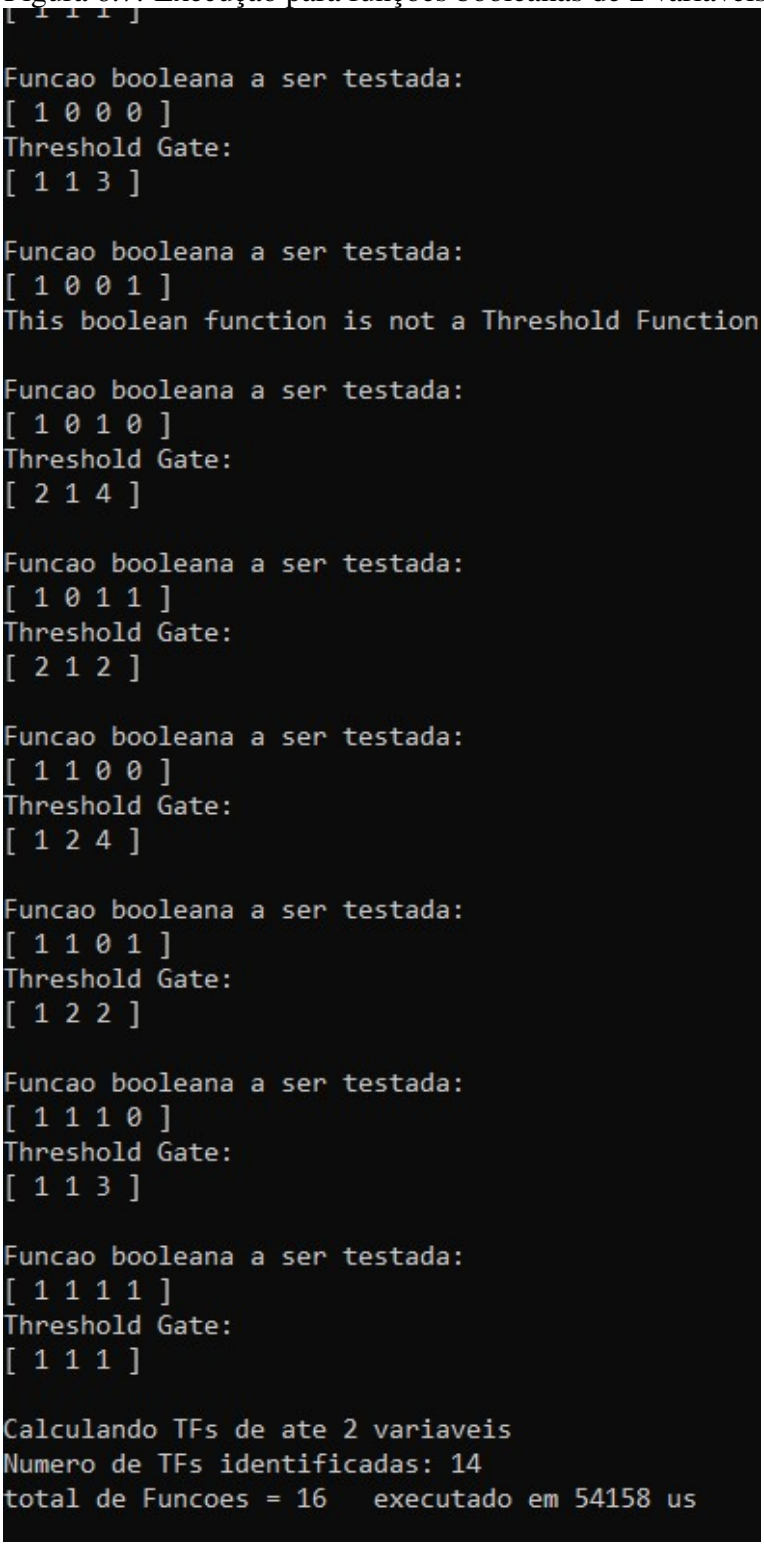


Figura 6.8: Execução para funções booleanas de 3 variáveis

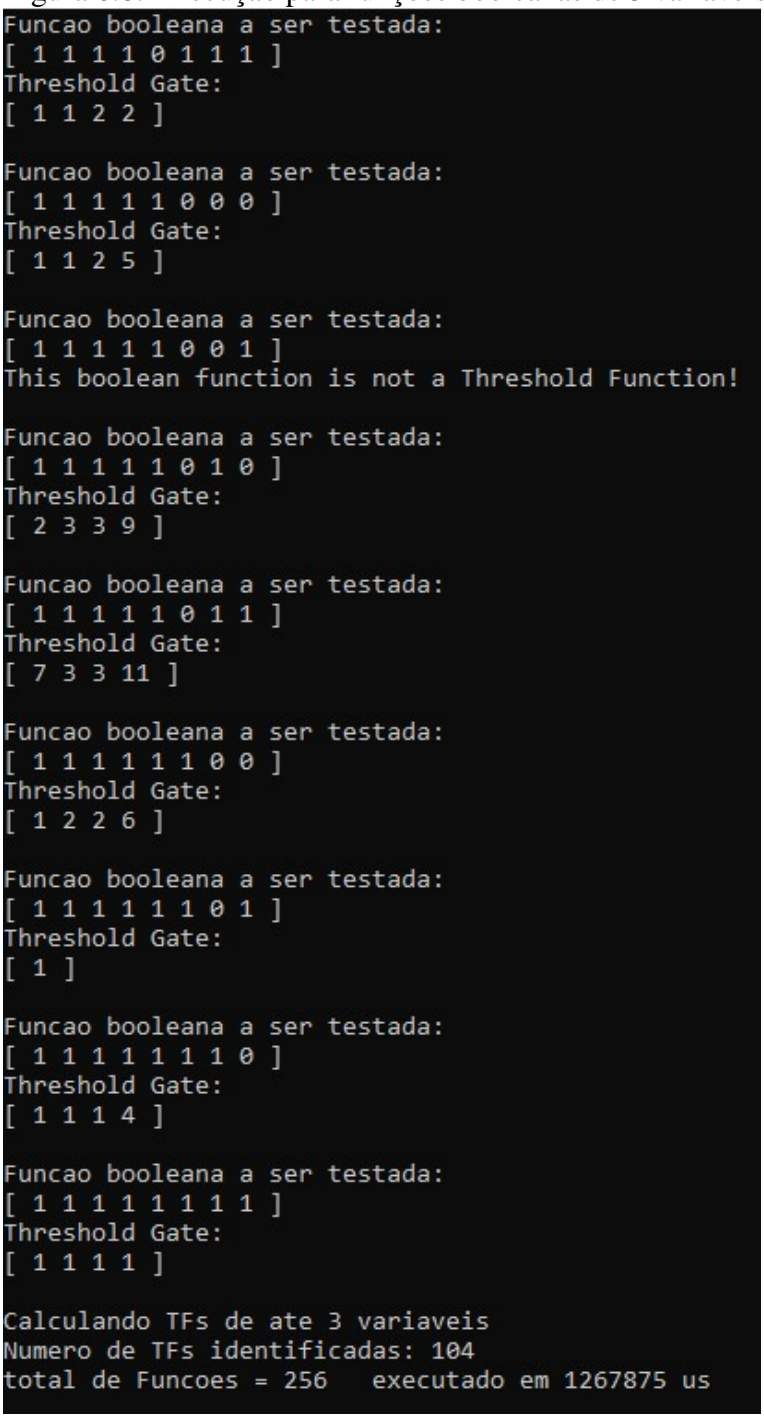
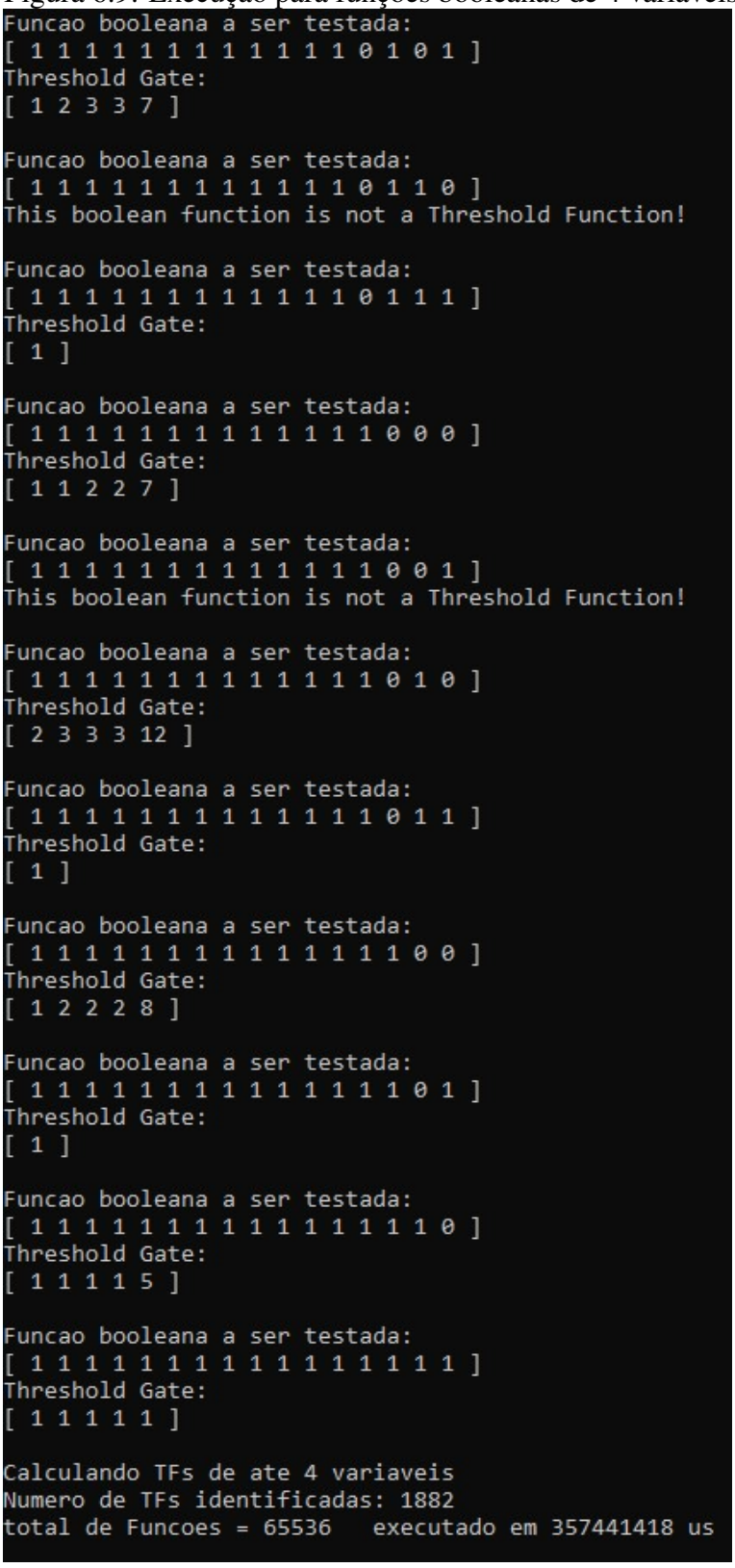


Figura 6.9: Execução para funções booleanas de 4 variáveis



7 CONCLUSÃO

Pelo fato de os resultados experimentais do presente trabalho terem sido mais incompletos do que o realizado por (LIU et al., 2019), em caso de continuidade, é interessante que o autor se aprofunde no estudo de geração das classes NP e/ou NPN (como fizeram os trabalhos de (PALANISWAMY; GOPARAJU; TRAGOUDAS, 2012), (NEUTZLING et al., 2018) e (LIU et al., 2019)), para que futuramente possa testar de maneira completa as TFs de 8 variáveis.

O trabalho de (LIU et al., 2019) identifica todas as funções threshold de até 8 variáveis e identificou 100.000 TFs aleatoriamente geradas de 9 a 15 variáveis (cada). Mesmo que tenha sido provado que tenha identificado uma quantidade grande de TFs de 9 a 15 variáveis, não se pode dizer que se identifica todas. Caso o outro do presente trabalho continue os estudos em uma pós-graduação, o primeiro passo seria continuar o trabalho de (MUROGA, 1971), que calculou o número de TFs e de classes NPs e NPs de TFs para até 8 variáveis. O interessante seria calcular e catalogar estes valores para até 15 variáveis, pois assim será possível verificar se o trabalho de (LIU et al., 2019) falha ou não em alguma identificação.

Caso se prove que o algoritmo de (LIU et al., 2019) funciona até 15 variáveis, é bem possível que o algoritmo não irá mais falhar. Neste caso, um trabalho de pós-graduação seria provar matematicamente que o algoritmo de (LIU et al., 2019) é completo, que pode identificar qualquer TFs de qualquer quantidade de variáveis. Caso se prove que (LIU et al., 2019) falhe, seria interessante compreender o porquê da falha e propor uma nova heurística, com resultados melhores.

REFERÊNCIAS

AVEDILLO, M. J.; QUINTANA, J. M. A threshold logic synthesis tool for rtd circuits. **Proc. Euromicro Symp. Digit. Syst. Design**, p. 624–627, 2004.

BLAIR, E. P.; LENT, C. S. Quantum-dot cellular automata: An architecture for molecular computing. **17in Proc. Int. Conf. Simul. Semiconductor Processes Devices**, p. 14–18, 2003.

GOWDA, T. et al. Identification of threshold functions and synthesis of threshold networks. **IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.**, v. 30, n. 5, p. 665–677, 2011.

LIKHAREV, K. K. Single-electron devices and their applications. **Proc. IEEE**, v. 87, n. 4, p. 606–632, 1999.

LIU, C. H. et al. Threshold function identification by redundancy removal and comprehensive weight assignments. **IEEE Trans. on Computer-Aided Design**, v. 38, n. 12, p. 2284–2297, 2019.

MUROGA, S. **Threshold Logic and its Applications**. [S.l.: s.n.], 1971.

NEUTZLING, A. et al. A simple and effective heuristic method for threshold logic identification. **IEEE Trans. on Computer-Aided Design**, v. 37, n. 5, p. 1023–1036, 2018.

PACHA, C. **Resonant Tunneling Device Logic Circuits**. [S.l.]: Univ. Dortmund/ Gerhard-Mercator Univ. Duisburg, Germany, Tech. Repr, 1999.

PALANISWAMY, A. K.; GOPARAJU, M. K.; TRAGOUDAS, S. An efficient heuristic to identify threshold logic functions. **ACM Journal on Emerging Technologies in Computing Systems**, v. 8, n. 3, p. 1–17, 2012.

SUBIRATS, J. L.; JEREZ, J. M.; FRANCO, L. A new decomposition algorithm for threshold synthesis and generalization of boolean functions. **IEEE Trans. Circuits Syst. I, Reg. Papers**, v. 55, n. 10, p. 3188–3196, 2008.

WINDER, R. O. **Threshold logic**. Thesis (PhD) — Princeton Univ., Princenton, 1962.

WINSTON, W. L. **Operations Research – Applications and Algorithms**. [S.l.]: Duxbury Press: Belmont (CA), 1994.

ZHANG, R. et al. Threshold network synthesis and optimization and its application to nanotechnologies. **IEEE Trans. on Computer-Aided Design**, v. 24, n. 1, p. 107–118, 2005.