UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

RENATO DONIZETE PERALTA

Geração de Circuitos a Partir de BDDs: Junção de Funções Booleanas Incompletamente Especificadas

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Orientador: Profª. Drª. Mariana Luderitz Kolberg
Co-orientador: Prof. Dr. André Reis

Porto Alegre
2022

CIP — CATALOGAÇÃO NA PUBLICAÇÃO

Peralta, Renato Donizete

Geração de Circuitos a Partir de BDDs: Junção de Funções Booleanas Incompletamente Especificadas / Renato Donizete Peralta. – Porto Alegre: PPGC da UFRGS, 2022.

94 f.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2022. Orientador: Mariana Luderitz Kolberg; Co-orientador: André Reis.

1. Funções Booleanas Incompletamente Especificadas. 2. Diagrama de Decisão Binária. 3. Relações Booleanas. I. Kolberg, Mariana Luderitz. II. Reis, André. III. Título.

RESUMO

Funções booleanas especificadas incompletamente (também conhecidas como relações booleanas) são definidas por seu On-set, Off-set e DC-set (Don't care set). Como o conjunto DC-set pode ser atribuído ao On-set ou ao Off-set, uma relação booleana pode conter várias funções completamente especificadas que satisfaçam a relação booleana original. Este trabalho propõe um algoritmo para a atribuição de don't cares para relações booleanas representadas como um par de Binary Decision Diagrams (BDDs). O algoritmo consiste em explorar o On-set e Off-set da relação booleana de entrada representada como BDDs e juntá-los em um terceiro BDD, que terá um número menor de nodos. Em média, nosso algoritmo foi capaz de gerar coberturas com 23,53% do número de nodos comparado aos BDDs dos on-sets das relações booleanas originais. Enquanto que métodos como restrict, constrain e *leaf-identifying compaction* geraram coberturas com tamanho em torno de 30%. Além disso, o algoritmo mostrou-se eficaz na redução do número de variáveis de entrada, onde apresentou uma redução média de 15,67%.

Palavras-chave: Funções Booleanas Incompletamente Especificadas. Diagrama de Decisão Binária. Relações Booleanas.

Generating Circuits from BDDs: Joining Incompletely Specified Boolean Functions

ABSTRACT

Incompletely specified Boolean functions (a.k.a. Boolean Relations) are defined by their On-set, Off-set, and dc-set (don't care set). As the dc-set can be assigned either to the On-set or to the Off-set, a Boolean relation can contain several completely specified functions that satisfy the original Boolean Relation. This work proposes an algorithm for the assignment of don't cares for Boolean relations represented as a couple of Binary Decision Diagrams (BDDs). The algorithm consists of exploring the On-set and Off-set of the input Boolean relation represented as BDDs and joining them into a third BDD, which will have a lower number of nodes. On average, our algorithm was able to generate coverage with 23.53% of the number of nodes compared to the BDDs of the on-sets of the original Boolean relations. While methods such as restrict, constrain and *leaf-identifying compaction* generated coverage with size around 30%. In addition, the algorithm proved to be effective in reducing the number of input variables, where it presented an average reduction of 15.67%.

Keywords: Incompletely Specified Boolean Functions, Binary Decision Diagrams, Boolean Relations.

LISTA DE ABREVIATURAS E SIGLAS

AIG	And-Inverter Graph
BDD	Binary Decision Diagram
BDD_for_CF	BDD for Characteristic Function
BDT	Binary Decision Tree
DAG	Directed Acyclic Graph
ITE	If-Then-Else
MIG	Majority-Inverter Graph
PI	Primary Input
PLA	Programmable Logic Array
PO	Primary Output
ROBDD	Reduced Ordered Binary Decision Diagram
SCF	Strong Canonical Form
SOP	Sum-Of-Products
ZDD	Zero-suppressed BDD

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ALGORITMOS

SUMÁRIO

1 INTRODUÇÃO

Um diagrama de decisão binária (BDD) é uma das estruturas de dados mais utilizadas para representar e manipular funções Booleanas. BDDs ordenados e reduzidos (ROBDDs) podem ser construídos para funções completamente, ou incompletamente, especificadas. Para funções Booleanas completamente especificadas, a representação é única, dada uma ordem das variáveis de entrada. No caso de funções Booleanas incompletamente especificadas (também conhecidas como relações Booleanas, no caso de funções de apenas uma saída), várias funções Booleanas distintas completamente especificadas podem satisfazer a relação Booleana original (HONG; BEEREL; BURCH; MCMILLAN, 1997).

Funções do tipo *index generation function* (SASAO, 2014) tendem a ser incompletamente especificadas para grande parte do espaço Booleano. Em outro exemplo de ocorrência de funções incompletamente especificadas, recentemente o concurso de programação da conferência IWLS 2020 abordou o problema de aprendizado de máquina no contexto de funções Booleanas incompletamente especificadas (RAI et al., 2020).

É de grande importância ter métodos que descubram funções completamente especificadas que produzem BDDs com menos nodos, enquanto satisfazem a relação Booleana inicial. Desta forma, representar e manipular funções Booleanas incompletamente especificadas de forma otimizada tem sido o foco de vários pesquisadores.

Um algoritmo heurístico capaz de reduzir o tamanho dos BDDs atribuindo valores binários aos *don't cares* foi apresentado em (SHIPLE; HOJATI; SANGIOVANNI-VINCENTELLI; BRAYTON, 1994), usando a forma fortemente canônica (BRACE; RUDELL; BRYANT, 1990) dos ROBDDs. A abordagem proposta por (HONG; BEEREL; BURCH; MCMILLAN, 1997) foi motivada para gerar BDDs menores comparados aos produzidos por (SHIPLE; HOJATI; SANGIOVANNI-VINCENTELLI; BRAYTON, 1994). No entanto, seu algoritmo não foi baseado na forma canônica forte de BDDs.

A abordagem proposta em (CHANG; CHENG; MAREK-SADOWSKA, 1994) também usa um BDD para representar uma função incompletamente especificada incluindo uma variável de controle extra (denominada Z), para diferenciar o *On-set* e o *Off-set* da função. Também é proposto um algoritmo heurístico para reduzir o tamanho do BDD que represente a função completamente especificada resultante

quando a variável Z é removida (assinalando os don´t cares). Esse algoritmo gera uma função completamente especificada a partir da atribuição do *Dc-set* para o *On-set* e *Off-set*. Esta abordagem tem como pontos fracos a inserção da variável Z (que pode aumentar a complexidade do BDD) e o fato de não usar a forma fortemente canônica de BDDs.

Em (MATSUURA; SASAO, 2007), funções incompletamente especificadas de múltiplas saídas são representadas através de um BDD para funções características, que é um tipo de BDD onde tanto as entradas como as saídas da função são variáveis de controle dos nodos. Além disto BDDs de função característica tem uma só raiz mesmo representando várias funções. Isto será visto em mais detalhe no capítulo 2. O método proposto aqui tem como vantagem usar BDDs onde somente variáveis de entrada controlam nodos do BDD.

O método proposto por (SASAO, 2014) tem como objetivo reduzir o número de variáveis necessárias para representar *index generation functions*. Neste sentido, os métodos descritos nesta dissertação podem ter aplicações em *index generation functions*. O trabalho aqui proposto é uma tentativa de contribuição para a síntese de funções incompletamente especificadas, usando BDDs.

Neste trabalho, apresentamos o método JOIN, um algoritmo baseado na forma fortemente canônica do operador *if-then-else* (ITE) (BRACE; RUDELL; BRYANT, 1990). Seu funcionamento consiste em construir o BDD do On-set e o BDD do Off-set de uma relação Booleana e prossegue percorrendo os dois BDDs simultaneamente enquanto cria um novo BDD usando a forma fortemente canônica. Testes realizados no *benchmark* do concurso de programação IWLS 2020 indicam uma grande capacidade de redução do tamanho do BDD das funções representadas.

A primeira versão do método Join gerava resultados idênticos, em termos de qualidade, ao método constrain (COUDERT; MADRE, 1990) previamente existente. A vantagem era aceitar diretamente o formato dos benchmarks do IWLS2020. Baseado na análise feita foi proposta uma segunda versão melhorada, que resulta em circuitos com menor número de nodos (maior qualidade). Considerando o quão raro é encontrar um algoritmo melhorado para BDDs, área extensivamente estudada por quatro décadas, acreditamos que esta dissertação apresente uma boa contribuição ao campo de síntese lógica.

Esta dissertação está organizada da seguinte forma. O capítulo 2 apresenta os conceitos básicos sobre funções Booleanas e como podemos representá-las. O

capítulo 3 apresenta uma revisão sobre BDDS, estrutura de dados utilizada neste trabalho. No capítulo 4 é apresenta uma revisão sobre trabalhos que objetivam representar e minimizar funções Booleanas incompletamente especificadas. A proposta deste trabalho é apresentada no capítulo 5. Os experimentos e resultados são apresentados no capítulo 6. Por último, o capítulo 7 conclui este trabalho e discute as possibilidade de trabalhos futuros.

2 CONCEITOS BÁSICOS

2.1 Sobre este capítulo

Este capítulo apresenta uma revisão sobre os conceitos básicos fundamentais para este trabalho. São apresentadas as definições de funções Booleanas e algumas das principais maneiras de se representar funções Booleanas e suas características. São apresentadas as funções Booleanas completamente especificadas e detalhados os seus conjuntos On-set e Off-set, e na sequência é apresentado o conjunto de funções incompletamente especificadas, foco deste trabalho.

2.2 Contexto Geral

Esta seção descreve o contexto mais amplo da dissertação. Vamos descrever este contexto geral a partir dos trabalhos do grupo LogiCS. A dissertação foca no tópico de síntese lógica (REIS; DRECHSLER, 2018) de circuitos digitais (WAGNER; REIS; RIBAS, 2006). A síntese lógica transforma uma descrição lógica inicial do circuito em uma rede de primitivas lógicas tais como células de bibliotecas (TOGNI; SCHNEIDER; CORREIA; RIBAS; REIS, 2002). A etapa de síntese lógica que cria uma rede de células em uma determinada tecnologia é conhecida como mapeamento tecnológico (REIS, 1999). A síntese lógica é uma etapa importante no fluxo de projeto de circuitos digitais tais como microprocessadores (ROSA; WAGNER; CARRO; CARISSIMI; REIS, 2003). As células da biblioteca são feitas de transistores, e a síntese lógica pode ser usada para sintetizar redes de transistores para as células da biblioteca (JUNIOR et al., 2006; SILVA; REIS; RIBAS, 2009; ROSA; REIS; RIBAS; MARQUES; SCHNEIDER, 2007; BUTZEN; BEM; REIS; RIBAS, 2010; BUTZEN; BEM; REIS; RIBAS, 2012; ROSA; SCHNEIDER; RIBAS; REIS, 2009).

A síntese eficiente de redes de transistores pode ser importante para otimizar custos relacionados a área (POLI; SCHNEIDER; RIBAS; REIS, 2003), variabilidade (SILVA; REIS; RIBAS, 2009; BUTZEN; BEM; REIS; RIBAS, 2010; BUTZEN; BEM; REIS; RIBAS, 2012) e potência (BUTZEN; JR; FILHO; REIS; RIBAS, 2010). Métodos propostos no grupo LogiCS, tais como síntese baseada em cortes KL (MACHADO; MARTINS; CALLEGARO; RIBAS; REIS, 2012) e síntese baseada em composição funcional (MARTINS; RIBAS; REIS, 2012) são usados em novas

tecnologias (NEUTZLING; MARTINS; RIBAS; REIS, 2013; MARRANGHELLO; CALLEGARO; MARTINS; REIS; RIBAS, 2015; NEUTZLING; MATOS; REIS; RIBAS; MISHCHENKO, 2015), circuitos robustos (GOMES et al., 2014; GOMES; MARTINS; REIS; KASTENSMIDT et al., 2015) e circuito assíncronos (MOREIRA et al., 2014).

2.3 Funções Booleanas

Funções Booleanas são utilizadas para especificar/descrever o comportamento de circuitos lógicos. Onde, para cada possível entrada, a função Booleana descreve o comportamento que o circuito deve possuir na prática.

De forma sucinta, uma função Booleana consiste em um mapeamento de um domínio (entradas) à uma imagem (saída) em que ambos pertencem ao conjunto de valores binários $\{0, 1\}$. Dado isto, uma função F de n entradas e uma saída é definida como $F : B^n \rightarrow B$, F mapeia uma relação das variáveis de entrada, denominado vetor de entrada, para um valor binário de saída.

Além disso, algumas funções Booleanas podem ter a sua imagem completamente especificada, enquanto outras funções Booleanas possuem saídas que não são fixadas com um valor. Quando dizemos funções Booleanas que possuem saídas não especificadas, nos referimos a relações Booleanas, uma vez que uma função Booleana deve obrigatoriamente gerar um valor bem definido de saída.

2.3.1 Representação de Funções Booleanas

A forma que uma função é representada depende fortemente da aplicação, pois cada maneira de se representar uma função possui características específicas que podem ser benéficas a um determinada contexto, mas podem ser inapropriadas para outros contextos. Na sequência são apresentadas algumas formas de representar funções Booleanas, bem como suas características.

2.3.1.1 Tabela-verdade

A tabela verdade é uma das formas mais utilizadas para representação de funções Booleanas. Uma tabela verdade descreve o comportamento de uma função

Booleana e a sua principal característica é que todas as possibilidades, de combinações de valores para as variáveis de entrada, são listadas (MICHELI, 1994; JÚNIOR, 2021). Desta forma, a tabela verdade possui a propriedade de representar uma função na forma canônica, ou seja, a função é sempre representada de forma única.

Na Figura 2.1 é apresentado um exemplo de função Booleana. Neste exemplo a função é representada através de uma tabela verdade que descreve o comportamento lógico da função para todas as possíveis entradas onde são associados valores de saída correspondentes às combinações de entradas.

Figura 2.1 – Exemplo de função Booleana.

Índice	x_0	x_1	x_2	x_3	F
0	0	0	0	0	0
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	1
14	1	1	1	0	0
15	1	1	1	1	1

Fonte: Extraído de (KATZ; BORRIELLO, 2005).

Neste exemplo de tabela verdade, Figura 2.1, trata-se de uma função de três variáveis. A primeira coluna apenas apresenta os índices de cada linha, as três seguintes colunas correspondem às variáveis de entrada que compõem a função, a quarta coluna corresponde aos valores de saída que a função pode assumir. Desta forma, a tabela verdade associa combinação de valores de entradas a um valor de saída, por exemplo, $F(0,0,0) \rightarrow 0$.

Uma característica da tabela verdade que em muitas aplicações torna-se empecilho é que o seu tamanho é exponencial em relação ao número de variáveis de entrada. Uma tabela verdade contém 2^n linhas, onde n é o número de variáveis de entrada (CRAMA; HAMMER, 2011).

2.3.1.2 Equações Booleanas

Uma equação Booleana é uma representação algébrica de uma função Booleana. Equações Booleanas consistem de combinações de operações (+ e ×) entre literais das varáveis de entrada, em que uma variável x_i pode aparecer na forma de literal positivo (x_i) ou literal negativo ($\overline{x}_i$). A equação 2.1 apresenta um exemplo de equação Booleana que descreve uma função com variáveis de entrada x_0, x_1 e x_2.

$$F(x_0, x_1, x_2) = x_0 \cdot x_1 + x_1 \cdot \overline{x}_2 \tag{2.1}$$

Uma mesma função Booleana pode ser descrita de diferentes formas, por exemplo as equações $\overline{(x_0 \cdot x_1)}$ e $(\overline{x}_0 + \overline{x}_1)$ possuem estruturas diferentes, porém descrevem a mesma função Booleana. Desta forma, as equações alguns tipos de equações podem não descrever a função na forma canônica.

No entanto, algumas maneiras de descrever uma função por meio de equação podem ser classificadas como canônica. Para isto, é necessário que exista uma correspondência de um-para-um com a tabela verdade da função. Uma forma de representar uma função através de uma equação na forma canônica é por meio de uma soma de produtos (SOP - *Sum of Products*, em inglês).

2.3.1.3 Soma-de-produtos

Um produto de variáveis consiste na operação AND entre as variáveis de entrada. Uma SOP é uma equação na qual são realizadas a operação lógica OR entre produtos de variáveis. O comportamento da função Booleana da tabela verdade apresentada na Figura 2.1, pode ser descrito através da seguinte SOP: $(\overline{x}_0 \cdot x_1 \cdot \overline{x}_2)$ + $(x_0 \cdot x_1 \cdot \overline{x}_2)$ + $(x_0 \cdot x_1 \cdot x_2)$.

SOPs descreve a função no formato de dois níveis. SOP é considerada uma das maneiras mais simples de se descrever uma função Booleana através de equação. A SOP possui a propriedade de representação canônica da função, porém é necessário que cada produto contenha todas as variáveis, na forma de literal direto ou negado, de que a função depende. Isto permite que a SOP atenda o requisito de correspondência um-para-um com a tabela verdade (WAGNER; REIS; RIBAS, 2006).

Um produto de variáveis que contém todas as variáveis de entrada é dito ser um *mintermo*. Por outro lado, um produto que não contém todas as variáveis é chamado de cubo. Cada *mintermo* possui uma correspondência única com uma determinada linha da tabela e, os *mintermos* associados a saída 1 da tabela verdade são chamados de *mintermos* implicantes. Desta forma, é possível derivar uma equação canônica, a partir de uma tabela verdade, através de uma SOP dos *mintermos* implicantes da tabela.

2.3.1.4 Forma Fatorada

Na prática, a grande maioria dos circuitos não utilizam o formato em dois níveis pois torna-se inviável, pois esta forma de representação requer um número de transistores maior do que a forma multi-nível. A forma fatorada utiliza o formato multi-nível que diferente do formato de dois níveis não restringe a profundidade do circuito a apenas dois níveis. A forma fatorada pode ser tanto um literal, uma soma, ou um produto de formas fatoradas (BRAYTON, 1987; CALEGARO, 2016).

São exemplos de formas fatoradas os seguintes itens: (x_0), $(\overline{x}_0)$, $(x_0 * \overline{x}_1)$, $(x_0(x_1(x_2 + x_3) + x1(x_2x_3 + x_4)))$.

Algoritmos de fatoração possuem como objetivo a redução de número de literais (CALEGARO, 2016). Comparada a forma de dois níveis, a forma multi-nível provê a redução do número de literais, que por sua vez implica na redução do número de transistores necessário para construir o circuito. Por outro lado, o formato multinível pode apresentar maior atraso de propagação de sinal, comparado ao formato de dois níveis. No entanto, ainda se mostra mais viável na prática.

Além da forma fatorada, outras técnicas multi-níveis são amplamente utilizadas para o propósito. Como exemplo, são as técnicas baseadas em grafos acíclicos dirigidos (DAGs - *Directed Acyclic Graphs*, em inglês) que fazem o uso dessa estrutura de dados para manipular funções Booleanas.

2.3.1.5 AIGs

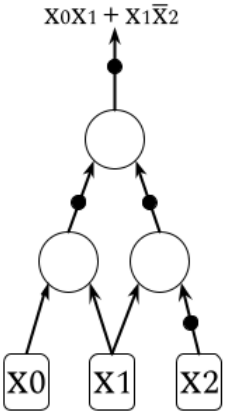
Grafos de And e Inversão (AIGs - And-Inverter-Graphs, em inglês) são DAGs compostos por portas AND de duas entradas e inversores. Cada nodo de AIG possui nenhuma ou duas arestas incidentes. Um nodo que não possui arestas incidentes é rotulado como entrada primária (PI - *Primary Inputs*, em inglês). Os demais

nodos, que possuem duas arestas incidentes, representam uma porta AND de duas entradas (MISHCHENKO; CHATTERJEE; BRAYTON, 2006; YU; CIESIELSKI; MISHCHENKO, 2017; POSSANI; MISHCHENKO; RIBAS; REIS, 2019).

Qualquer nodo pode ser classificado como saída primária (PO - *Primary Output*, em inglês). Em um AIG, as arestas podem ser complementadas ou não. Uma aresta que aparece complementada indica a inversão do sinal ao qual ela representa (MISHCHENKO; CHATTERJEE; BRAYTON, 2006; YU; CIESIELSKI; MISHCHENKO, 2017).

A Figura 2.2 apresenta um exemplo de um AIG. Os retângulos representam as PIs, os círculos não preenchidos representam as portas AND. Por último, os pontos pretos sólidos acima das arestas representam a negação da aresta em questão.

Figura 2.2 – Exemplo de um AIG para a expresão $x_0.x_1 + x_1.\overline{x}_2$.



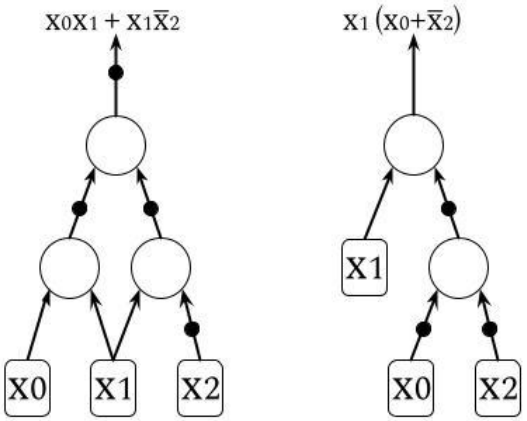
Fonte: Autor.

De acordo com (MATOS, 2018) um AIG, como estrutura de dados, apresenta como vantagens para síntese lógica as seguintes características:

- A manipulação lógica é mais fácil devido a homogeneidade dos nodos do AIG;
- Facilidade de armazenamento em memória;
- Correlação justa entre contagem de nodos de AIG e área do circuito. Além disso, a profundidade lógica e o atraso do circuito podem ser correlacionados.

Apesar de apresentar essas características, AIGs não representam a função na forma canônica (CALEGARO, 2016). A Figura 2.3 apresenta dois BDDs que representam a mesma função, porém são estruturalmente distintos.

Figura 2.3 – Exemplo de AIG.



Fonte: Autor.

2.3.1.6 BDDs

BDD é um DAG originalmente desenvolvido para manipular funções Booleanas de forma eficiente, requerido em projeto lógico VLSI (*Very Large Scale Integration*) (BERNASCONI; CIRIANI, 2016; MINATO, 2017). Cada nodo não terminal de um BDD é marcado com uma variável Booleana de entrada e possui exatamente duas arestas de saída. Desta forma, cada nodo não terminal representa um multiplexador 2:1 (AMARU; GAILLARDON; MICHELI, 2015).

Diversas aplicações práticas podem ser decompostas em estruturas discretas. Representar estruturas discretas, de forma compacta, é muito importante, pois isto permite que tarefas como otimização e verificação de validade/equivalência sejam executada eficientemente. BDD é um dos modelos de estruturas discretas mais básico de manipulação de funções Booleanas (ANDERSEN, 1997; MINATO, 2013). Além de tarefas de validação e equivalência, BDD também pode ser utilizado para geração de redes de transistores, como em (POLI; SCHNEIDER; RIBAS; REIS, 2003) e (JUNIOR et al., 2006).

Uma importante característica é que BDDs permitem a representação canônica e canônica forte da função. No capítulo 3 será apresentada uma revisão mais detalhada sobre BDDs.

2.3.1.7 Netlist

O nome netlist, no contexto de descrições VLSI, geralmente se refere a descrições representadas através de um conjunto de primitivas lógicas específicas que

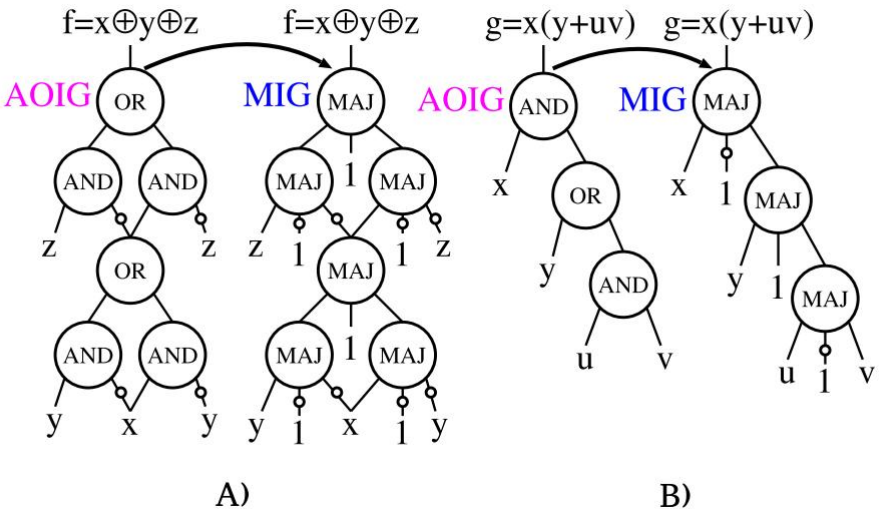
são instanciadas e interligadas para formar o circuito. Não existe definição única de netlist, portanto é importante verificar o contexto para compreender o significado em que a palavra está sendo empregada. Podem existir netlists de transistores, de portas lógicas, de células de biblioteca, etc.

2.3.1.8 MIGs

Assim como BDDs e AIGs, Majority-Inverter Graphs (MIGs) (AMARÚ; GAILLARDON; MICHELI, 2014), (AMARU; GAILLARDON; MICHELI, 2015) e (AMARÚ; GAILLARDON; MICHELI, 2015) também são uma representações lógicas baseadas em grafos. Além disso, MIGs também são DAGs homogêneos em que cada nodo representa uma função majoritária de três entradas.

O operador majoritário $M(a, b, c)$ possui o comportamento de um operador $OR(a, b)$ quando $c = 1$ e, o comportamento de um operador $AND(a, b)$ quando $c = 0$ (AMARÚ; GAILLARDON; MICHELI, 2014). Portanto, cada nodo de um MIG representa um operador majoritário MAJ3. Além disso as arestas podem ser negadas, assim como em AIGs e BDDs. A Figura 2.4 apresenta dois exemplos de MIGs convertidos a partir de AND/OR/Inverter Graphs (AOIGs).

Figura 2.4 – Conversão de AND/OR/Inverter Graphs para MIG.



Fonte: Extrído de (AMARÚ; GAILLARDON; MICHELI, 2014).

2.3.2 Funções Booleanas Completamente Especificadas

O conjunto de valores binários {0, 1} possui tamanho igual a 2, o que implica que uma função com n entradas contém 2^n possíveis vetores de entrada. Uma função Booleana é completamente especificada quando o seu domínio é mapeado por completo para uma imagem. Portanto, para uma função Booleana ser completamente especificada é necessário que todas as combinações das variáveis de entrada tenham associado a si um valor de saída, como pode-se observar no exemplos da Figura 2.1 que possui quatro variáveis de entrada e 16 possíveis combinações.

Funções Booleanas completamente especificadas são compostas por dois conjuntos, um que engloba os vetores de entradas que geram as saídas 1 e, o conjunto que engloba os vetores de entradas que geram as saídas 0, são eles o On-set e Off-set, denotados respectivamente como f_{on} e f_{off}. O On-set é composto pelos vetores de entradas que levam a saída da função ao valor 1. Por outro lado, o Off-set consiste nos vetores de entradas que resultam no valor de saída 0 (FROEHLICH; GROSSE; DRECHSLER, 2017).

2.3.2.1 Exemplo de On-set e Off-set

A função Booleana da Figura 2.1 tem o seu On-set especificado com os vetores de entradas correspondentes aos índices 4-10, 13 e 15. Enquanto que o Off-set é composto pelos vetores de entradas 0-3, 11, 12 e 14. No exemplo em questão, o On-set e Off-set cobrem todas as 16 possíveis combinações de entradas, desta forma a função em questão trata-se de uma função Booelana que tem o seu domínio mapeado por completo.

2.3.3 Funções Booleanas Incompletamente Especificadas

Em um projeto de circuito podem haver determinadas combinações da variáveis de entrada que não afetam o comportamento da saída do circuito. Desta forma, tais combinações de variáveis não necessitam ser especificadas.

Diferente das funções Booleanas completamente especificadas em que todos os vetores de entradas possuem um valor binário de saída associado, as funções Booleanas incompletamente especificadas possuem vetores de entradas que não têm

um valor de saída definido, devido esses vetores de entradas não influenciarem no comportamento da função (REIS; DRECHSLER, 2018).

Uma função Booleana incompletamente especificada possui o seguinte mapeamento: $F : B^n \rightarrow \{0, 1, X\}$, onde o simbolo $''X''$ representa o valor *don't care*. Com este mapeamento, vetor de entradas da função pode ter especificado como saída um dos seguintes valores: zero, um, ou *don't care* (REIS; DRECHSLER, 2018). Desta forma, as combinações das variáveis de entrada que levam ao valor $''X''$ compõem o conjunto *don't care* (DC-set). Portanto, o domínio de uma função incompletamente especificada ff pode ser dividido em três subconjuntos, ff_{on} (On-set), ff_{off} (Off-set) e ff_{dc} (DC-set) (HONG; BEEREL; BURCH; MCMILLAN, 1997).

O que difere este dois tipos de funções Booleanas é devido ao fato das funções Booleanas completamente especificadas possuírem o conjunto DC-set vazio, enquanto que funções Booleanas incompletamente especificadas possuem o conjunto DC-set $\neq \emptyset$ (HONG; BEEREL; BURCH; MCMILLAN, 1997) (MARRANGHELLO; CALLEGARO; REIS; RIBAS, 2015). Um exemplo de função que possui saídas *don't care* é apresentado na Figura 2.5.

Figura 2.5 – Exemplo de função Booleana com saídas *don't care*.

Índice	x_0	x_1	x_2	x_3	*ff*
0	0	0	0	0	X
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	X
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	1
14	1	1	1	0	0
15	1	1	1	1	X

Fonte: Extraído de (KATZ; BORRIELLO, 2005).

No exemplo da Figura 2.5, os índices 0, 7 e 15 da tabela verdade têm associados a si como saída o valor $''x''$. Ou seja, esta é uma função incompletamente especificada, onde sua saída não depende dos vetores de entrada: $x_0 = 0, x_1 = 0$, $x_2 = 0$ e $x_3 = 0$ (índice 0), $x_0 = 0, x_1 = 1$, $x_2 = 1$ e $x_3 = 1$ (índice 7) e $x_0 = 1, x_1 = 1$, $x_2 = 1$ e $x_3 = 1$ (índice 15).

2.3.3.1 Exemplo de Dont-care set

Em funções Booleanas incompletamente especificadas, um vetor de variáveis de entradas que não tenha sua saída especificada pode ser atribuído tanto ao On-set quanto ao Off-set. As saídas associadas a *don't care* é uma maneira explícita de falar que o projetista pode escolher livremente o valor a ser atribuído aqueles vetores de entrada (KATZ; BORRIELLO, 2005). Este é um tópico de pesquisa, em que procura-se uma maneira para realizar o assinalamento de *don't care* de maneira eficiente com o objetivo de gerar circuitos otimizados.

O assinalamento de *don't cares* de uma função Booleana incompletamente especificada pode ser realizado de diferentes maneiras, podendo ser triviais ou não triviais. Pode-se assinalar todas as saídas *don't cares* a um mesmo valor (0 ou 1), esta é forma trivial pelo fato de não ter a necessidade de buscar um assinalamento eficiente. Outra maneira de assinalar *don't cares* consiste em realizar assinalamentos mistos, onde uma parte dos *don't cares* são assinalados com o valor 0 e outra parte assinalados com o valor 1.

A Figura 2.6 apresenta quatro diferentes escolhas de assinalamento de *don't cares*, os valores encontram-se destacado em negrito com cor vermelha em cada tabela. A Figura 2.6(a) apresenta o assinalamento trivial, onde todos os *don't cares* são assinalados com o valor 0. Por sua vez, a Figura 2.6(b) apresenta o mesmo procedimento, porém assinalando os *don't cares* com o valor 1.

As Figuras 2.6(c) e 2.6(d) apresentam exemplos de assinalamento misto de 1 e 0 aos *don't cares*. Os impactos resultantes de cada assinalamento realizado na Figura 2.6 são apresentados na Figura 2.7.

Para evidenciar o impacto da escolha no assinalamento de *don't cares*, a Figura 2.7 apresenta os circuitos obtidos a partir de cada assinalamento apresentado na Figura 2.6. O circuito apresentado na Figura 2.7(a) é derivado da função Booleana obtida a partir do assinalamento do valor 0 a todos os *don't cares*, Figura 2.6(a), desta forma todos os vetores de entradas que inicialmente eram *don't cares* passam a compor o Off-set da função.

Semelhante ao exemplo anterior, o circuito apresentado na Figura 2.7(b) implementa a função Booleana obtida a partir do assinalamento do valor 1 a todos os *don't cares*, Figura 2.6(b). Deste modo, todos os vetores de entradas inicialmente *don't cares* passam a integrar o On-set da função.

O circuito apresentado na Figura 2.7(c) implementa a função Booleana obtida

Figura 2.6 – Exemplos de assinalamentos de *don't care*.

Índice	x_0	x_1	x_2	x_3	f
0	0	0	0	0	**0**
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	**0**
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	1
14	1	1	1	0	0
15	1	1	1	1	**0**

(a) Tudo para 0.

Índice	x_0	x_1	x_2	x_3	f
0	0	0	0	0	**1**
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	**1**
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	1
14	1	1	1	0	0
15	1	1	1	1	**1**

(b) Tudo para 1.

Índice	x_0	x_1	x_2	x_3	f
0	0	0	0	0	**1**
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	**0**
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	1
14	1	1	1	0	0
15	1	1	1	1	**1**

(c) Misto ineficiente.

Índice	x_0	x_1	x_2	x_3	f
0	0	0	0	0	**0**
1	0	0	0	1	0
2	0	0	1	0	0
3	0	0	1	1	0
4	0	1	0	0	1
5	0	1	0	1	1
6	0	1	1	0	1
7	0	1	1	1	**1**
8	1	0	0	0	1
9	1	0	0	1	1
10	1	0	1	0	1
11	1	0	1	1	0
12	1	1	0	0	0
13	1	1	0	1	1
14	1	1	1	0	0
15	1	1	1	1	**0**

(d) Misto eficiente.

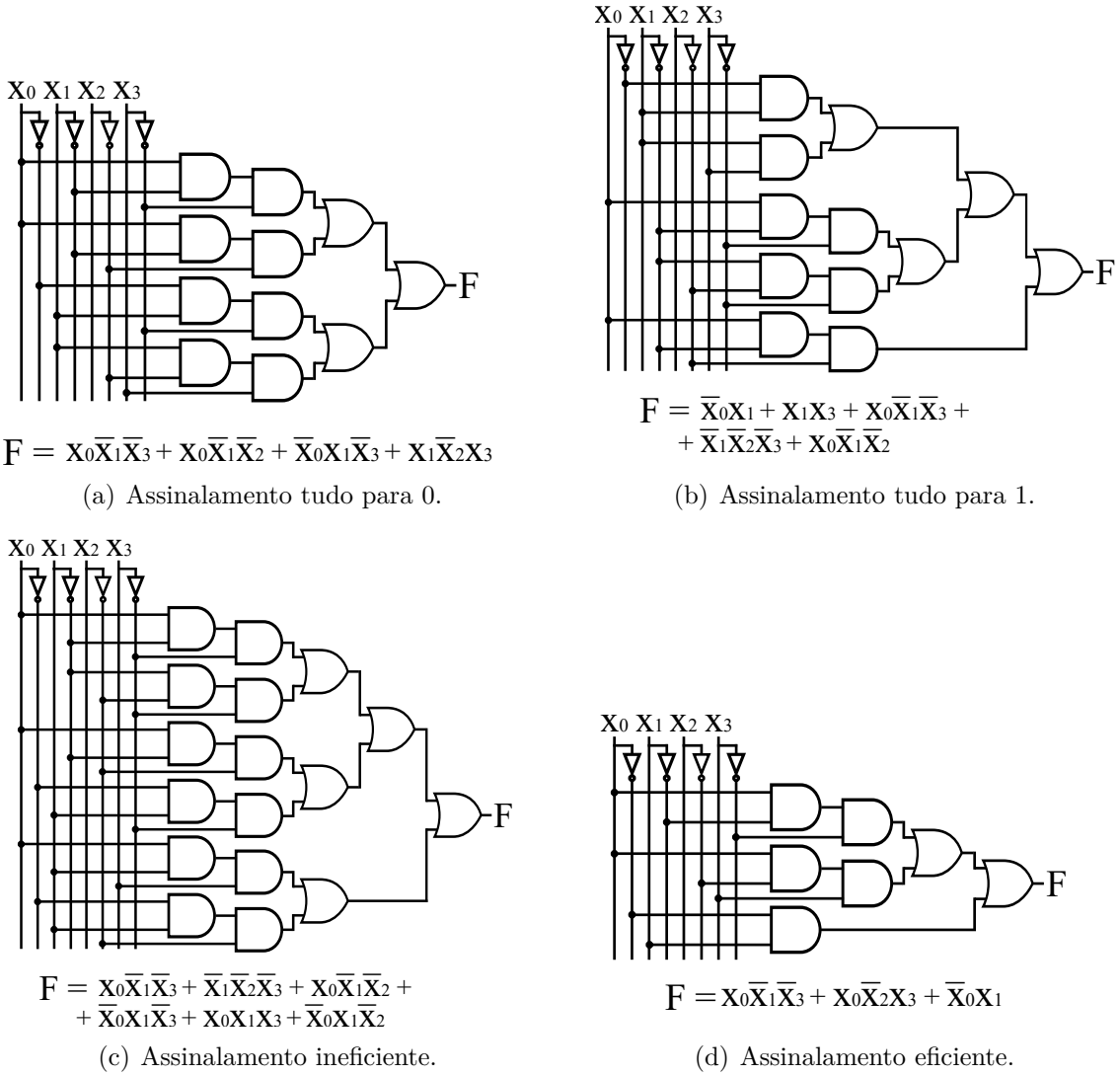
Fonte: Autor.

a partir do assinalamento misto demonstrado na Figura 2.6(c). Desta forma, uma parte dos vetores de entradas passam a integrar o On-set e outra parte passa a compor o Off-set da função.

Por último, o circuito apresentado na Figura 2.7(d) é derivado da função Booleana obtida a partir do assinalamento do valor 0 ou 1 aos *don't cares*, Figura

2.6(d). Da mesma forma descrita no exemplo anterior, uma parcela dos vetores de entradas passam a compor o On-set e outra o Off-set da função.

Figura 2.7 – Circuitos, compostos por portas AND2 e OR2, obtidos com diferentes escolhas de assinalamento de *don't care*.



(a) Assinalamento tudo para 0.

(b) Assinalamento tudo para 1.

(c) Assinalamento ineficiente.

(d) Assinalamento eficiente.

Fonte: Autor.

Como pode-se observar a partir da Figura 2.7, a decisão tomada em cada exemplo (Figuras 2.6) pode impactar de forma positiva, ou negativa, no tamanho do circuito resultante. Para a função Booleana utilizada de exemplo, Figura 2.5, assinalar todos os *don't cares* com o valor 0 resultou em um circuito composto por 11 portas lógicas (entre AND2 e OR2) Figura 2.7(a). Ao assinalar todos os *don't cares* com o valor 1, o circuito resultante possui 12 portas lógicas (entre AND2 e OR2), como apresentado na Figura 2.7(b). Ao optar por um assinalamento misto qualquer, pode-se obter um circuito ainda maior que os dois anteriores, como é apresentado na

Figura 2.6(c) em que o circuito contém 17 portas lógicas. Por último, ao assinalar os *don't cares* cuidadosamente, pode-se obter um resultado melhor, como é apresentado na Figura 2.6(d), onde o circuito resultante possui apenas 7 portas lógicas.

A tarefa de assinalar valores aos *don't cares* de uma função Booleanas incompletamente especificada é conhecido como o problema de minimização cujo objetivo é encontrar uma cobertura, completamente especificada, para uma função Booleana incompletamente especificada (SHIPLE; HOJATI; SANGIOVANNI-VINCENTELLI; BRAYTON, 1994). Na sequência é apresentado o conceito de cobertura de função e também é discutido o problema de encontrar uma cobertura.

2.3.4 Cobertura de funções Booleanas incompletamente especificadas

Uma função Booleana incompletamente especificada ff pode ser expressa através de um par de funções completamente especificadas $[f, c]$, onde f representa a função, para o espaço Booleano especificado, e c representa o seu Care-set que é domínio especificado da função. Portanto, deve-se ter que $f_{on} \supseteq ff_{on}$, $f_{off} \supseteq ff_{off}$ e por sua vez, $c = ff_{on} \cup ff_{off}$ (HONG; BEEREL; BURCH; MCMILLAN, 1997). Como o Care-set é espaço do domínio especificado, por isso consiste da união dos conjuntos ff_{on} e f_{off}. Esses termos são apresentados na seção 2.3.3.

Além disso, como apresentado no trabalho de (SHIPLE; HOJATI; SANGIOVANNI-VINCENTELLI; BRAYTON, 1994), uma outra relação que se obtém a partir de f e c é que $onset = f \cdot c$ e $offset = \overline{f} \cdot c$. Esta relação será importante para exemplificar, na seção 5.7, o funcionamento do método proposto nesta dissertação.

Dadas essas definições, o problema de encontrar uma cobertura f' para uma função Booleana incompletamente especificada ff consiste em encontrar uma função Booleana completamente especificada que atenda aos seguintes requisitos: $i)$ $f'_{on} \supseteq ff_{on}$, $ii)$ $f'_{off} \supseteq ff_{off}$, $iii)$ $f'_{on} \cup f'_{off} = 1$ e $iv)$ $f'_{on} \cap f'_{off} = \emptyset$ (CHANG; CHENG; MAREK-SADOWSKA, 1994; HONG; BEEREL; BURCH; MCMILLAN, 1997; YANG; CIESIELSKI, 2002). Ou seja, de acordo com os itens (i) e (ii) a cobertura f' deve conter o On-set e Off-set de ff, o item (iii) diz que a cobertura f' deve ser completamente especificada e por último, item (iv) um elemento do domínio pode apenas pertencer a um dos conjuntos (On-set ou Off-set, devido ser f' completamente especificada).

A Figura 2.8 apresenta um exemplo em que pode-se observar essas afirmações.

A tabela é divididas em três partes, a primeira parte (Entradas) consiste nas variáveis de entrada x_0 x_1 *e* x_2, a segunda parte (ISF) fornece um exemplo de função Booleana incompletamente especificada ff, onde são descritos seus conjuntos ff_{on} (possui valor 1 quando ff é 1), ff_{off} (possui valor 1 quando ff é 0) e c que é o Care-set (possui valor 1 quando alguns dos conjuntos anteriores são especificados). Por último, a terceira parte da tabela (Uma cobertura para ff) apresenta uma cobertura f' qualquer para ff, onde são descritos seus conjuntos f'_{on}, f'_{off}, a união e intersecção de f'_{on}, f'_{off}, nesta ordem.

Figura 2.8 – Exemplo das regras para f' ser uma cobertura de ff.

Entradas			ISF				Uma cobertura para *ff*			
x_0	x_1	x_2	ff	ff_{on}	ff_{off}	c	f'_{on}	f'_{off}	$f'_{on} \cup f'_{off}$	$f'_{on} \cap f'_{off}$
0	0	0	1	1	0	1	1	0	1	0
0	0	1	X	0	0	0	1	0	1	0
0	1	0	X	0	0	0	0	1	1	0
0	1	1	0	0	1	1	0	1	1	0
1	0	0	0	0	1	1	0	1	1	0
1	0	1	X	0	0	0	0	1	1	0
1	1	0	X	0	0	0	1	0	1	0
1	1	1	1	1	0	1	1	0	1	0

Fonte: Autor.

Como pode-se observar na Figura 2.8, todas as saídas 1 de ff_{on} estão contidas em f'_{on} (item i). Da mesma forma, todas as saídas 1 de ff_{off} estão contidas em f'_{off} (item ii). O resultado de $f'_{on} \cup f'_{off}$ é 1 para todos os vetores de entradas, ou seja f' é completamente especificada. Por último, $f'_{on} \cap f'_{off}$ é 0 para todas os vetores de entradas, portanto f'_{on} e f'_{off} não se sobrepõem.

O problema de encontrar uma cobertura mínima para uma função Booleana incompletamente especificadas é uma tarefa de grande importância, pois isto pode lidar em ganho de performance (SHIPLE; HOJATI; SANGIOVANNI-VINCENTELLI; BRAYTON, 1994), devido prover uma representação otimizada, em termo de nodos, para a aplicação que utilizará a função minimizada. Diante disto, é de grande importância o desenvolvimento de algoritmos capazes de forma otimizada representar, manipular e minimizar funções Booleanas incompletamente especificadas, foco deste trabalho.

2.3.5 Diferentes usos da palavra cobertura

A palavra cobertura pode ser usada com diferentes sentidos em diferentes contextos de síntese lógica. No contexto desta dissertação uma cobertura é uma função Booleana completamente especificada que atende os requisitos da especificação de uma função incompletamente especificada. No contexto de mapeamento tecnológico uma cobertura é uma rede de células interconectadas que implementa a função Booleana desejada.

2.4 Funções Booleanas de Múltiplas Saídas

Funções Booleanas de múltiplas saídas são semelhantes as funções Booleanas descritas na seção 2.3. O que difere estes dois tipos de funções é que uma função de múltiplas saídas mapeia uma relação Booleana para mais de uma saída, este mapeamento é dado da seguinte forma: seja $B \in \{0, 1\}$, então $F : B^{\,n} \rightarrow B^{\,m}$, onde n é número de variáveis de entrada e m é o número de saídas que a função possui. Ou seja, essa classe de funções mapeia um *array* de valores binários de tamanho n para outro *array* de valores binários de tamanho m (EBENDT; FEY; DRECHSLER, 2005).

Pode-se utilizar o somador completo como um exemplo de função Booleana de múltiplas saídas. A Figura 2.9 apresenta a tabela verdade de um somador completo que recebe como entrada C_{in} (*Carry In*) e dois bits de entrada (x_1 e x_2) e gera as saídas C_o e S que são, respectivamente, o *Carry Out* e resultado da soma.

2.5 Ferramentes para otimização de funções Booleanas

2.5.1 ESPRESSO

Desenvolvida na IBM e depois popularizada ainda mais pela Universidade de Berkeley, a ferramenta ESPRESSO (BRAYTON; HACHTEL; MCMULLEN; SANGIOVANNI-VINCENTELLI, 1984) é um minimizador lógico dois níveis heurístico. O ESPRESSO foi a referência de síntese heurística em dois níveis para uso na síntese com matriz lógica programável (PLA - *Programmable Logic Array*, em inglês)

Figura 2.9 – Tabela verdade de um meio somador.

Entradas			Saídas	
C_{in}	X_1	X_2	C_o	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Fonte: Autor.

como a forma de implementação e depois como forma de gerar uma expressão em dois níveis que seria fatorada para gerar uma expressão multinível com o SIS (SENTOVICH et al., 1992). O formato de descrição *.pla* foi introduzido e popularizado no contexto do ESPRESSO.

2.5.2 MIS, SIS e ABC

A primeira ferramenta a difundir a síntese multinível baseada na manipulação de formas fatoradas foi o MIS (BRAYTON; RUDELL; SANGIOVANNI-VINCENTELLI; WANG, 1987), da Universidade de Berkeley. O MIS foi substituído pela ferramenta SIS (SENTOVICH et al., 1992), que adicionou tratamento a circuitos sequenciais aos algoritmos do MIS, que continuaram disponíveis. A estrutura interna do MIS e do SIS era baseada na manipulação de formas fatoradas, o que causa alguns problemas de escalabilidade para circuitos maiores. Por esta razão foi introduzido o ABC (BRAYTON; MISHCHENKO, 2010) que usa AIGs como estrutura de dados interna para permitir maior escalabilidade. AIGs são discutidos na próxima seção.

2.6 Concurso IWLS 2020

O concurso IWLS 2020 trouxe como desafio o uso de técnicas de aprendizado de máquina aplicadas a Síntese Lógica. O desafio proposto inclui, entre outros passos, a tarefa de obter uma representação de uma função a partir de uma descrição incompletamente especificada.

O concurso IWLS 2020 dispôs um conjunto de *benchmarks* compostos de arquivos de treinamento de teste. Os arquivos de treinamento contém as especificações das funções incompletamente especificadas e, os arquivos de treinos são utilizados para avaliar a acurácia do método desenvolvido pela equipe. Todos os *benchmarks* foram descritos através de arquivos no formato PLA, que descreve uma função Booleana como soma de produtos (para síntese usando PLAs).

2.6.1 *PLAs e Formato PLA*

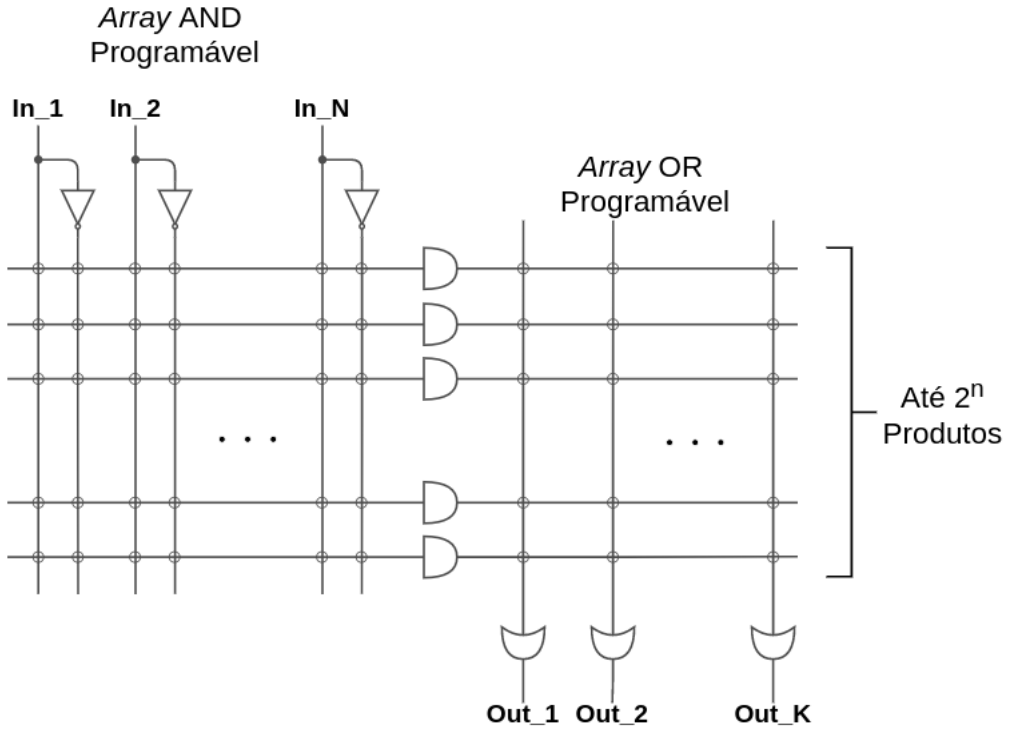
Uma Programmable Logic Array (PLA) consiste na implementação na forma de SOP de uma lógica combinacional (MANO; CILETTI, 2013). O conceito de um PLA é semelhante ao de uma memória programável somente de leitura (PROM - *Programmable Read-Only Memory*, em inglês) com a exceção de que o PLA não gera todos o mintermos e não oferece uma completa decodificação da variáveis. No PLA o decodificador, utilizado no PROM, dá lugar a uma matriz de portas AND que permitem a programação das variáveis de entrada para gerar qualquer termo produto (MANO; CILETTI, 2013).

A Figura 2.10 apresenta a estrutura de um PLA. No exemplo ilustrado, o PLA pode conter N entradas e, consequentemente, pode gerar no máximo 2^n produtos através das entradas. Além disso, possui K saídas.

Como apresentado na Figura 2.10, a estrutura de um PLA é composta por um *Array* AND Programável e um *Array* OR Programável. O *Array* AND contém as especificações, na forma de produto, das variáveis de entrada da função, ou seja, é realizada a operação lógica AND entre as variáveis. Cada variável pode estar na forma de literal negado ou não negado.

Já o *Array* OR Programável consiste na operação Lógica OR entre o produtos das variáveis de entrada. Para cada saída da função o *array* OR especifica quais produtos integram a soma. Desta forma, um PLA possui a estrutura de Soma-de-

Figura 2.10 – Estrutura de um PLA.



Fonte: Autor.

Produtos.

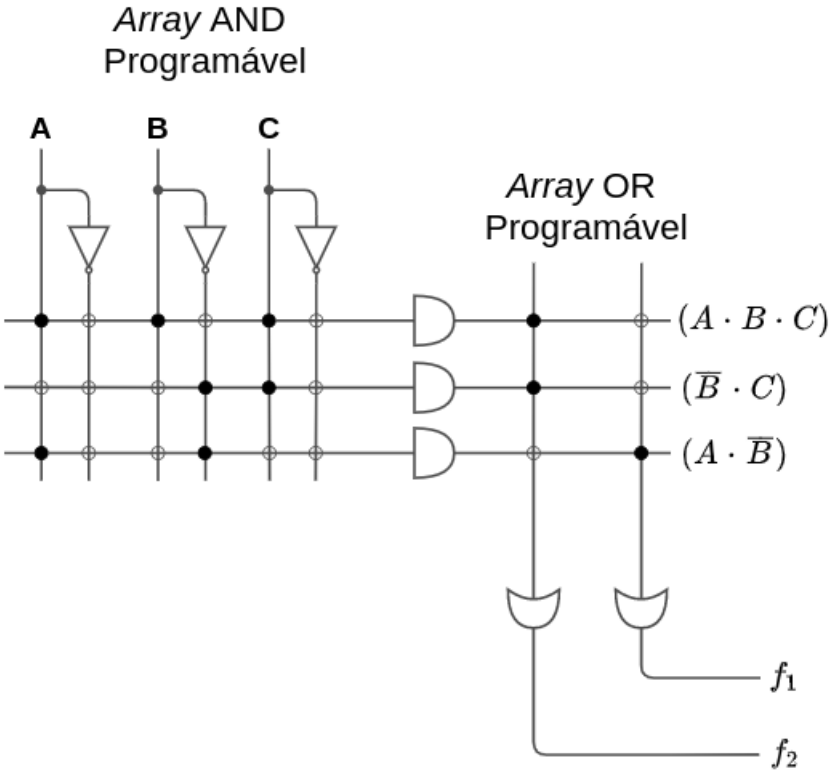
Na Figura 2.11 é apresentado um exemplo de uma especificação Booleana através de PLA. A especificação é composta por três variáveis de entrada, A, B e C, e possui duas saídas f_1 e f_2.

A Listagem 2.1 apresenta a representação textual do PLA apresentado na Figura 2.11. As quatro primeiras linhas do arquivo contêm o cabeçalho do PLA. A primeira linha especifica do número de variáveis de entrada, a segunda linha especifica o número de saídas da função descrita pelo PLA, a terceira linha especifica número de cubos especificados e, a quarta linha descreve o tipo de arquivo PLA. A última linha é marcada com *.e* que indica o final do arquivo.

De acordo (YANG, 1991), o arquivo PLA é logicamente tipado com **f**, **fd**, **fr**, ou **fdr**. Para cada saída, os valores '1', '0', '-' ou '~ ' definem as seguintes propriedades:

- **f**: 1 significa que o termo de produto pertence ao onset, 0 ou - não possui significado para a função;
- **fd**: 1 significa que o termo de produto pertence ao onset, 0 não tem significado para a função e '-' implica que o termo pertence ao DC;
- **fr**: 1 significa que o termo de produto pertence ao onset, 0 significa que o

Figura 2.11 – Exemplo de um PLA.



Fonte: Autor.

termo de produto pertence ao OFFset e '-' não tem significado para a função;

- **fdr**: Por último, 1 significa que o termo de produto pertence ao onset, 0 significa que o termo de produto pertence ao OFFset, '-' significa que o termo de produto pertence ao DC e '˜ ' implica que este termo não tem significado para o valor da função.

As demais linhas de uma descrição textual de um PLA descreve os *arrays* programáveis AND e OR. Portanto, nesta seção do arquivo contém a especificação lógica da função. Na seção *Array Programável AND* os arranjos das variáveis de entrada são especificadas com os valores: 1, 0 ou -. O valor 1 indica que a variável em questão aparece na forma de seu literal não negado, o valor 0 indica que a variável aparece na forma de literal negado e, - indica que a variável não aparece no cubo.

Listing 2.1 – Descrição textual do PLA da Figura 2.11.

```
.i 3
.o 2
.p 3
.type fd
111 10
```

−01 10
10− 01
. e

De forma semelhante, no *Array Programável OR* as saídas da função podem ser especificadas por: 1, 0 ou -. O valor 1 indica que o produto das variáveis de entrada correspondente está associado ao On-set da função, o valor 0 indica que o produto está associado ao Off-set, enquanto que, o valor - indica que o produto está associado ao *don't care* da função especificada pelo PLA.

2.7 Contribuições deste capítulo

Este capítulo apresentou uma revisão sobre os conceitos básicos que são fundamentais para este trabalho. A revisão apresentada, neste capítulo, sobre funções Booleanas discute o seu papel em circuitos digitais, algumas das principais maneiras de representá-las e as principais características resultantes de cada de representação empregada.

3 BDDS - DIAGRAMAS DE DECISÃO BINÁRIOS

3.1 Sobre este capítulo

Neste capítulo é feita uma revisão sobre diagrama de decisão binária (BDD - *Binary Decision Diagram*, em inglês). São apresentados conceitos fundamentais, como a representação canônica de uma função Booleana por meio de um BDD, a influência da ordem no tamanho do BDD resultante e algumas modificações de BDDs que objetivam aprimorá-lo para um determinado contexto.

3.2 Um breve histórico de BDDs

Proposto por (LEE, 1959) e posteriormente aprimorado por (AKERS, 1978), BDD é uma representação baseada em grafo de uma função lógica. O BDD é um DAG composto por dois nodos terminais que representam os valores 0 e 1 e nodos não-terminais que são marcados com as variáveis que compõem a função.

A estrutura de um BDD é baseada na Expansão de Shannon (SHANNON, 1949). Desta forma, cada nodo não terminal no BDD representa uma função Booleana e são chamados de nodos de decisão. Cada nodo não terminal em um BDD é marcado com uma variável Booleana e possui exatamente dois nodos filhos que correspondem aos cofatores positivo e negativo da função que o nodo representa (AKERS, 1978).

Dado um vetor de valores de entradas, ao atribuir estes valores ao nodos do BDD deve-se avaliar a variável, que marca cada nodo não terminal, com o valor correspondente no vetor de entrada. Em cada nodo não terminal, a variável v_i a qual ele é marcado pode ser avaliada com $v_i = 0$ ou $v_i = 1$ que correspondem, respectivamente, às arestas de saída $low(v_i)$ e $high(v_i)$ do nodo (AKERS, 1978). Assim, atribuindo valores às variáveis, é percorrido um caminho no BDD e ao final de um caminho percorrido, pode-se atingir um nodo terminal 0 ou 1. Isto significa que a sequência de valores atribuídas às variáveis de entrada resultam no determinado valor de saída para a função (BRYANT, 1986).

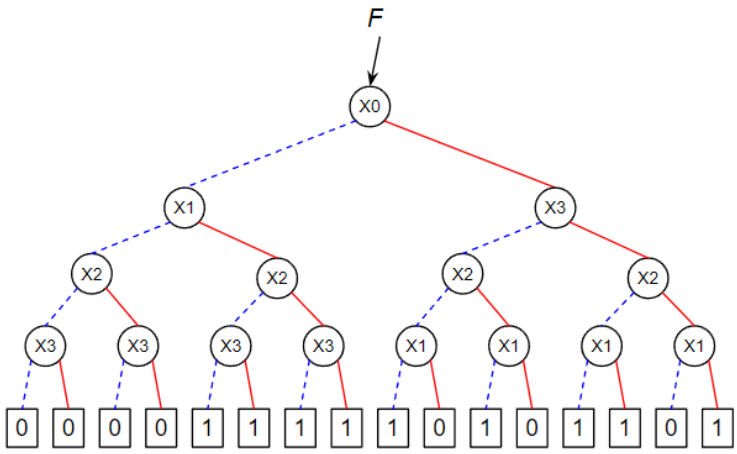
Na Figura 3.1(b) é apresentado o BDD obtido a partir da função descrita pela tabela verdade da Figura 3.1(a). Em BDDs os círculos representam os nodos não terminais, por sua vez, os retângulos representam os nodos terminais (1 e 0). O

nodo raiz do BDD, deste exemplo, é marcado com a variável x_0, este é o nodo pelo qual se tem acesso ao BDD. As linhas sólidas na cor vermelha representam os filhos F1 (cofator positivo). Por sua vez, as linhas tracejadas na cor azul representam os filhos F0 (cofator negativo).

Figura 3.1 – Exemplo de BDD.

x_0	x_1	x_2	x_3	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	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1
1	0	0	0	1
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	1
1	1	1	0	0
1	1	1	1	1

(a) Especificação, por tabela verdade, de uma função Booleana (mesma da seção 2.3).



(b) BDD obtido a partir da tabela verdade.

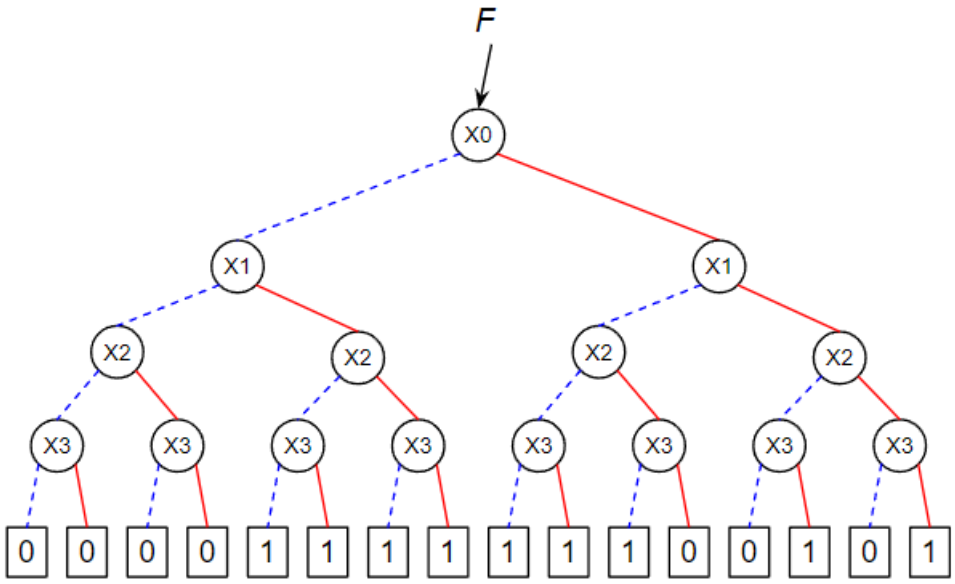
Fonte: Autor.

Como pode-se observar no BDD da Figura 3.1(b), trata-se de um BDD não ordenado, pois a sequência das variáveis não é a mesma para todos os caminhos da raiz até os nodos terminais. Por exemplo, o caminho $x_0 = x_1 = x_2 = x_3 = 0$ segue a ordem $x_0 < x_1 < x_2 < x_3$, enquanto que, $x_0 = x_1 = x_2 = x_3 = 1$ segue a ordem $x_0 < x_3 < x_2 < x_1$.

Uma BDD ordenado (OBDD - *Ordered Binary Decision Diagram*, em inglês) possui como restrição que todos os caminhos no BDD devem obrigatoriamente seguir a mesma ordem das variáveis. A Figura 3.2 apresenta um OBDD, com ordem $x_0 < x_1 < x_2 < x_3$, para a mesma função Booleana da Figura 3.1(b). Como pode-se observar, a restrição de ordenamento é satisfeita, dado que todos os caminhos da raiz até os nodos terminais seguem o mesmo caminho no BDD.

Como pode-se notar na Figura 3.2, o OBDD possui redundâncias, onde existem nodos repetidos que representam a mesma função Booleana e/ou nodos que não tomam decisões, onde seus dois filhos f_0 e f_1 apontam para um mesmo nodo.

Figura 3.2 – OBDD da função da Figura 3.1(b).



Fonte: Autor.

Segundo (LIAW; LIN, 1992) e (BRYANT, 1986), um OBDD "completamente expandido", como o utilizado de exemplo na Figura 3.2, possui $2^n - 1$ nodos não terminais e dentre eles, muitos são redundantes.

3.3 ROBDDs

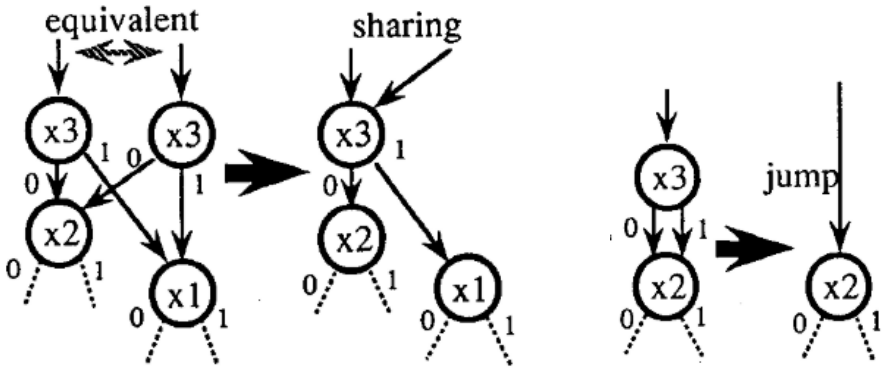
Como comentado anteriormente, o OBDD da Figura 3.2 não encontra-se representado com tamanho mínimo. Desta forma, o OBDD pode ser reduzido, sem alterar a função a qual ele representa, com a remoção das redundâncias de acordo com as seguintes regras de (BRYANT, 1986):

1. Compartilhamento de nodos - Consiste em fundir dois, ou mais, nodos que representam as mesmas funções;
2. Eliminação de nodos - Um nodo é dito redundante se os seus filhos F0 e F1 apontam para o mesmo nodo. Então ele é removido e as arestas que apontam para si são redirecionadas para o nodo apontando por F0 e F1.

As regras listadas são ilustradas na Figura 3.3. A Figura 3.3(a) apresenta uma ilustração da aplicação da Regra 1. A ilustração da regra 2 é apresentada na Figura 3.3(b).

As regras 1 e 2 são aplicadas, em um OBDD previamente construído, até que não exista nodos duplicados e verificações redundantes no OBDD (LIAW; LIN,

Figura 3.3 – Regras de redução de ROBDD.

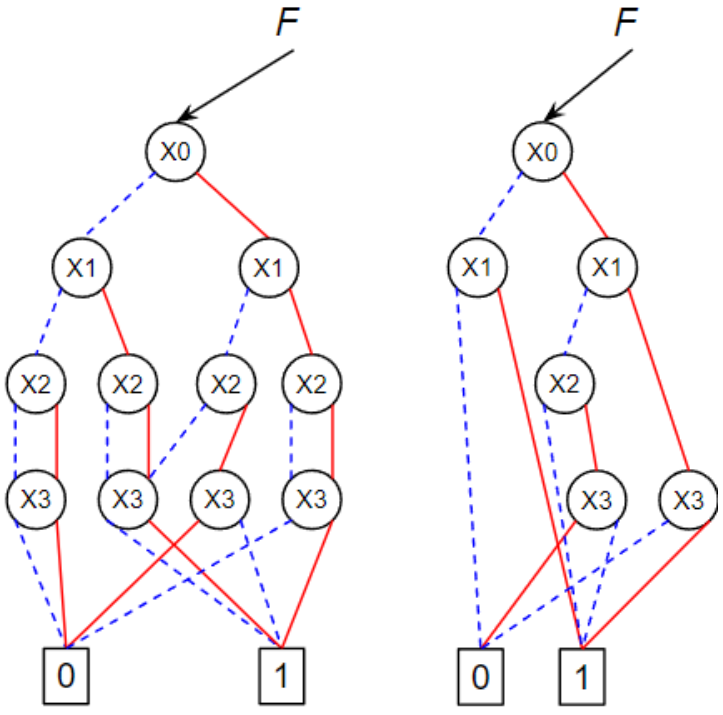


(a) (b)

Fonte: Extraído de (MINATO, 1993).

1992). Com isto, obtêm-se o OBDD reduzido (ROBDD - *Reduced Ordered BDD*, em inglês), que possui a característica de representar a função na forma canônica, uma representação mínima e única da função (para uma certa ordem). Os resultados da aplicação das regras são apresentados na Figura 3.4. A Figura 3.4(a) apresenta o OBDD após a aplicação da regra 1 e, a Figura 3.4(b) apresenta o OBDD gerada após a aplicação das regras 1 e 2.

Figura 3.4 – Resultado das regras de redução.



(a) OBDD após a aplicação da regra 1.

(b) ROBDD após a aplicação das regras 1 e 2.

Fonte: Autor.

Segundo (BRYANT, 1986), funções não representadas na forma canônicas podem ter distintas representações, o que pode tornar mais complicado testes de equivalência ou satisfatibilidade. No entanto, reduzir um OBDD para o ROBDD correspondente, capaz de representar a função na forma canônica, aplicando repetidamente as regras de redução é uma tarefa custosa, uma vez que as regras de redução devem ser aplicadas até que não exista redundâncias no OBDD.

Diante disto, (BRACE; RUDELL; BRYANT, 1990) propõem uma maneira de se obter o ROBDD sem a necessidade de realizar re-computações para construir o ROBDD. Esta maneira consiste em realizar as verificações de redundâncias durante a construção do ROBDD. Para garantir que não existam nodos repetidos, faz-se o uso de uma tabela *hash* para manter um registro dos nodos criados durante a construção do ROBDD. Esta tabela *hash* recebe o nome de Tabela Única, pois ela mantém uma correspondência única para cada nodo do ROBDD, onde cada nodo possui uma chave composta pela tripla $(v_i, f_{vi}(0), f_{vi}(1))$.

3.4 Forma Canônica Forte

Toda função Booleana representada por um ROBDD, que utiliza a Tabela Única, encontra-se na sua forma canônica forte (SCF - Strong Canonical Form, em inglês) (BRYANT, 1986). Antes de criar um novo nodo, é verificado na Tabela Única se já existe uma instância para o nodo, caso o nodo já exista na Tabela Única, então é retornada a referência dele. Entretanto, se o nodo não está contido na Tabela Única um novo nodo é criado e adicionado à tabela. Desta forma é garantido que cada nodo representa uma única função (BRACE; RUDELL; BRYANT, 1990).

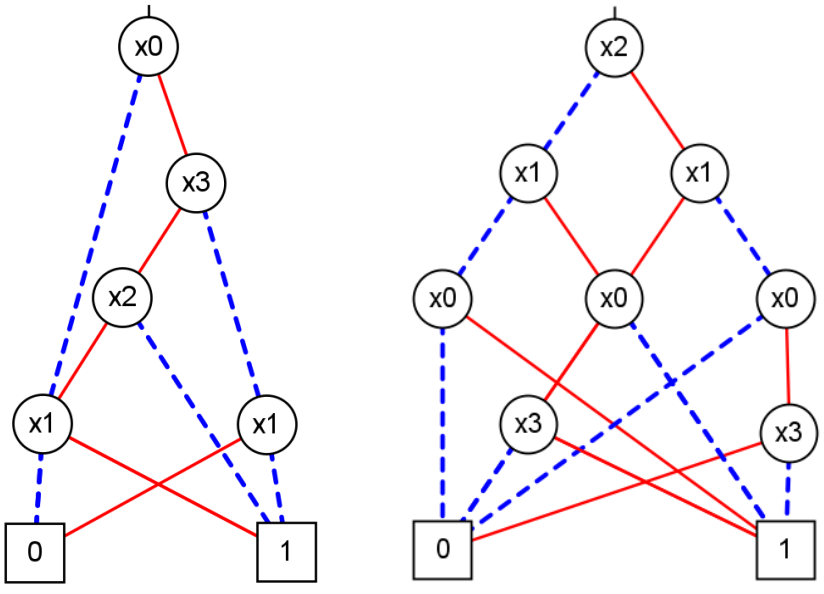
Além de evitar a re-computação, criação de nodos já existentes, a canonicidade é uma propriedade relevante por permitir que testes de equivalência ou satisfatibilidade possam ser realizados de forma eficaz. Como uma representação canônica é uma representação única, o teste por equivalência pode ser realizado apenas comparando se dois gráficos são correspondentes, já o teste por satisfatibilidade consiste em comparar o gráfico da função com o valor 0 (constante) (BRYANT, 1986).

3.5 Influência da Ordem em ROBDDs

O tamanho de um ROBDD é sensitivo a ordem das variáveis de entrada. A escolha da ordem pode gerar um significante impacto no número de nodos do ROBDD resultante, podendo variar de linear a exponencial, de acordo com o número de variáveis de entrada (BRYANT, 1986).

Na Figura 3.5 é demonstrado o efeito da escolha da ordem das variáveis em um ROBDD. O exemplo utiliza a mesma função Booleana utilizada nos exemplos anteriores, Figura 3.4, apenas são utilizadas ordens diferentes para as variáveis de entrada para evidenciar o impacto no tamanho final do ROBDD. O ROBDD da Figura 3.5(a) foi construído seguindo a ordem $x_0 < x_3 < x_2 < x_1$ que resultou em ROBDD com cinco nodos não terminais. Por outro lado, o ROBDD da Figura 3.5(b) foi construído de acordo com a ordem $x_2 < x_1 < x_0 < x_3$ e resultou em um ROBDD com oito nodos.

Figura 3.5 – Influência da ordem em um ROBDD para a função da Figura 3.1(a); a) ROBDD com a ordem $x_0 < x_3 < x_2 < x_1$, b) ROBDD com a ordem $x_2 < x_1 < x_0 < x_3$.



(a) (b)

Fonte: Autor.

3.6 BDDs baseados em Inteiros

Nos pacotes clássicos de BDDs, como em (BRACE; RUDELL; BRYANT, 1990), os nodos são representados por estruturas de dados que mantém um identificador para a variável e ponteiros para os nodos F0 e F1. Além disso, utiliza um ponteiro adicional chamado de *NEXT* para tratamento de colisão na Tabela Única (JANSSEN, 2001).

No entanto, o uso de ponteiros em BDDs pode ocasionar um *overhead* de memória devido ao espaço de armazenamento requerido para ponteiros. Diante disto, em 2001 (JANSSEN, 2001) introduziu um pacote de BDDs, inspirado em (LONG, 1998), que utiliza índices inteiros ao invés de ponteiros para referenciar nodos no BDD.

A Figura 3.6 apresenta a estrutura genérica, baseada na estrutura proposta por (JANSSEN, 2001), de um BDD baseado em inteiros. A Figura 3.6(a) apresenta a estrutura de um nodo que armazena a variável e os filhos F0 e F1. A Figura 3.6(b) apresenta uma esquematização de um *array* de nodos. Por último, a Figura 3.6(c) apresenta a BDD correspondente representado através de um DAG. Vale ressaltar que o exemplo dado na Figura 3.6(a) consiste em um exemplo simplificado, (JANSSEN, 2001) utiliza uma estrutura que possui outros campos.

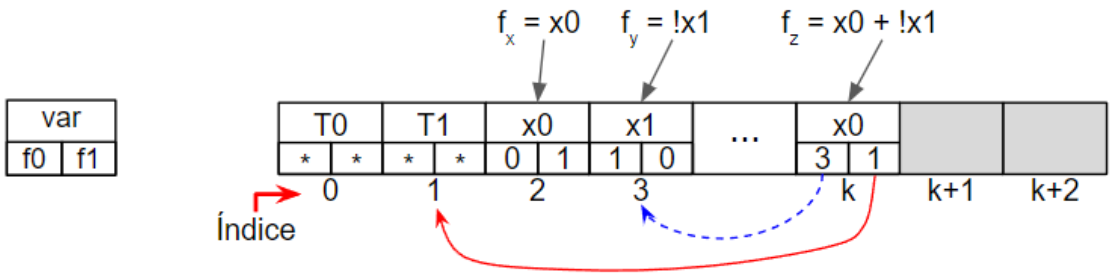
Na Figura 3.6(a), o número inteiro logo abaixo de cada nodo é o seu índice no *array*. Os nodos que se encontram nas posições 0 e 1 são os nodos terminais do BDD. As duas posições $k+1$ e $k+2$ são utilizadas para demonstrar as próximas posições livre no *array*, pois um novo nodo é inserido na primeira posição vaga no BDD que se encontra imediatamente após o último nodo inserido.

Ainda de acordo com a Figura 3.6(a), os filhos F0 e F1 mantém o índice ao qual o nodo "apontado" se encontra no *array*. Como pode ser visto, o nodo na posição k representa a função $x_0 + \overline{x}_1$ que é equivalente ao BDD da Figura 3.6 (c) na forma de grafo.

3.7 BDDs com arcos negados

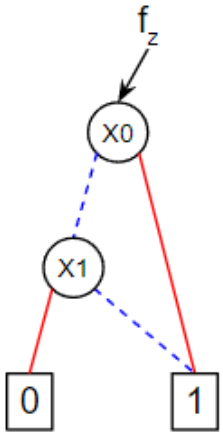
Uma forma de reduzir o número de nodos de um ROBDD é através da utilização de Arestas Complementares (CE - Complemented Edges, em inglês). Em um ROBDD com CEs, pode-se utilizar um mesmo subgrafo para representar uma

Figura 3.6 – BDD com índices inteiros.



(a) Estrutura de um nodo.

(b) *Array* que representa o BDD.



(c) Representação através de DAG

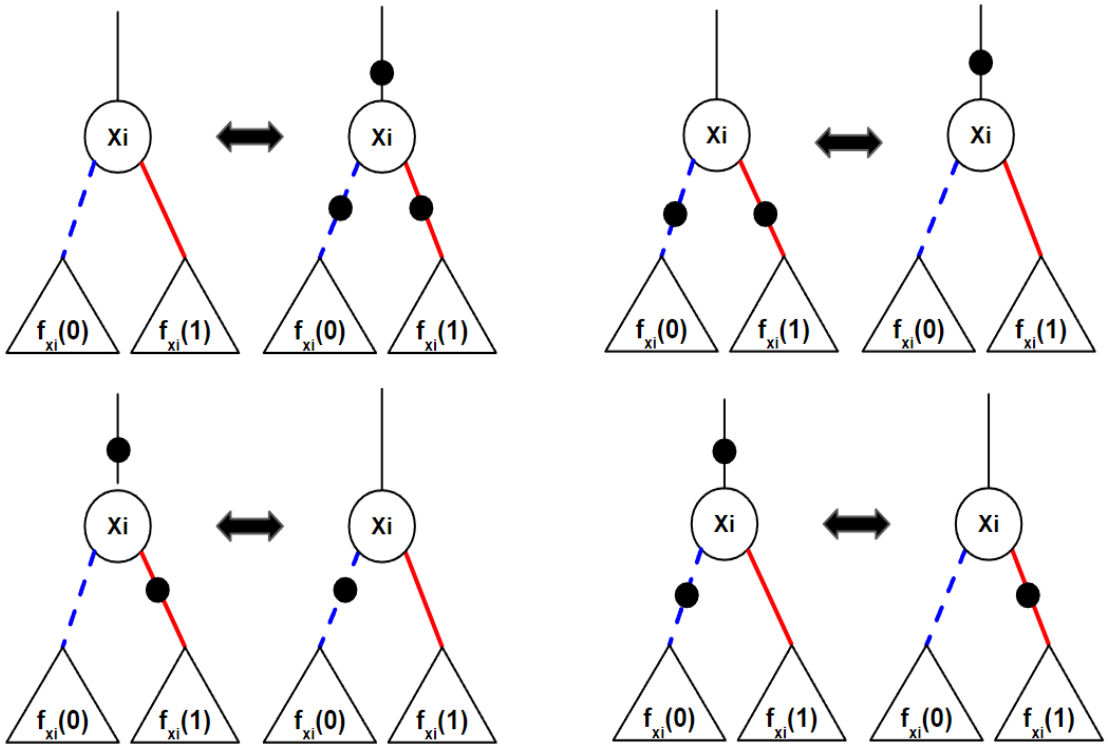
Fonte: Autor.

função F e também o seu complemento $\overline{F}$. As arestas são marcadas com um bit que sinaliza se o nodo apontado, por tal aresta, deve ser avaliado como a sua função original ou negada. A Figura 3.7 apresenta exemplos de BDDs, que representam um dada função, e seu respectivo BDD complementar. As equivalências apresentadas na figura são descritas por (BRACE; RUDELL; BRYANT, 1990). Os círculos preto indicam a negação da linha a qual ele sobrepõe. Para se manter a canonicidade, pode-se apenas complementar a aresta F_0 do nodo, a aresta F_1 de cada nodo deve ser regular.

Como pode-se observar na Figura 3.7, com apenas o uso de um bit extra em cada aresta é possível reaproveitar um subgrafo completo sem ser necessário criar novos nodos. Em seu trabalho, com o uso de arestas complementares (BRACE; RUDELL; BRYANT, 1990) diz obter uma redução de 7% do número de nodos de BDD para um conjunto de 12 exemplos.

Devido a grande popularidade dos BDDs, muitas modificações baseadas em

Figura 3.7 – Arcos Negados em BDDs.



Fonte: Adaptada de (BRACE; RUDELL; BRYANT, 1990).

BDDs foram propostas na literatura. Dentre elas, podemos citar uma das mais populares que são os BDDs com zero suprido (ZDD - Zero-suppressed BDDs, em inglês).

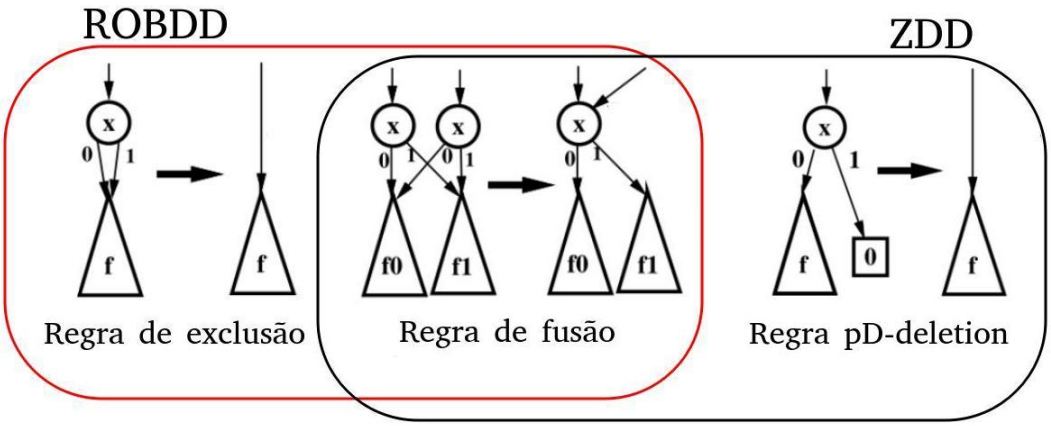
3.7.1 ZBDDs

ZDDs foram introduzidos por (MINATO, 1993). Esse diagrama é semelhante a um BDD, entretanto se mostra mais eficiente, em relação a tamanho, quando manipulando conjuntos esparsos (MISHCHENKO, 2001). Comparados a BDDs, os ZDDs são mais apropriados para representar conjuntos de combinação que é um vetor (x_n, x_{n-1}, ..., x_1, x_0) de binários que expressa se um objeto está incluso, ou não, na combinação (MINATO, 1993).

Ambos, ROBDDs e ZDDs, utilizam a regra de fusão em que subgrafos que representam a mesma função são compartilhados. A diferença entre ROBDDs e ZDDs é que ao invés de remover nodos com decisões redundantes (Regra de exclusão emprega em ROBDDs), em ZDDs são removidos os nodos que possuem a sua aresta 1 apontando diretamente para o nodo terminal 0 (Regra pD-deletion) (MINATO,

2001). A Figura 3.8 apresenta as regras de redução empregadas em ROBDDs e ZDDs.

Figura 3.8 – Regras de redução empregadas por ZDDs.



Fonte: Adaptado de (MINATO, 2013).

De acordo com (MINATO, 2001) é assumido para conjuntos de combinações que itens irrelevantes nunca aparecem em qualquer combinação do conjunto, desta forma, essas variáveis devem ser zero para satisfazer as funções características. A regra de fusão utilizada em ROBDDs mostra-se ineficaz neste tipo de problema, entretanto, ZDDs mostram-se mais apropriados.

Em um ZDD, os caminhos a partir da raiz que atingem o nodo terminal 1 representam as combinações válida dentro de um conjunto. Este caminhos são chamados de 1-path. Além disso, quando uma variável x_i não está presente no 1-path, significa que ao atribuir o valor 1 para x_i a saída da função é 0. Caso contrário, se atribuir o valor 0 para x_i é necessário avaliar o restante do caminho para se obter o valor de saída da função (BERNASCONI; CIRIANI, 2016).

De modo geral, ZDDs são mais eficientes para manipular conjuntos de combinações esparsos (combinações com literais ausentes). Enquanto que BDDs são melhores do ZDDs quando manipulando funções Booleanas comuns (MINATO, 2001).

3.8 Contribuições deste capítulo

Este capítulo provê uma revisão sobre BDDs, estrutura de dados utilizada neste trabalho. BDDs se tornaram popular para representar e manipular funções Booleanas devido a sua capacidade de representar a função Booleana de forma mais

reduzida, comparado à representação através de tabela verdade, além de ser capaz de representar a função na forma canônica. Este capítulo também apresenta algumas variações de implementações que tornam os BDDs mais otimizados, em relação a memória, como é o caso de arestas complementares e utilização de índice inteiros. Por último, são apresentadas algumas variações de BDDs, mas com foco no ZDD pois trata-se de uma das variações mais popular.

4 TRABALHOS ANTERIORES

4.1 Sobre este capítulo

Neste capítulo são apresentados quatro trabalhos que utilizam BDDs para representar, manipular e otimizar funções Booleanas incompletamente especificadas. Para isto, utilizaremos uma função Booleana incompletamente especificada para exemplificar cada método discutido.

A função utilizada é apresentada na Figura 4.1, esta se trata da função de múltiplas saídas que implementam um somador completo (colunas C_o e S) apresentado na seção 2.4, onde alguns valores das saídas são marcados com *Don't care* (colunas C_{ox} e S_x). A função Booleana incompletamente especificada de múltiplas saídas (Exemplo DC) trata-se apenas de um exemplo prático que não possui significado lógico.

Figura 4.1 – Função exemplo para exemplificar os métodos.

Entradas			Saídas		Exemplo DC	
X0	X1	X2	Co	S	Cox	Sx
0	0	0	0	0	0	0
0	0	1	0	1	x	1
0	1	0	0	1	0	x
0	1	1	1	0	1	0
1	0	0	0	1	0	1
1	0	1	1	0	x	0
1	1	0	1	0	1	x
1	1	1	1	1	1	1

Fonte: Autor.

4.2 BDDs e funções incompletamente especificadas

Como apresentado na seção 2.3, uma função Booleana incompletamente especificada é representa através de seu On-set, Off-set e DC-set. E também, a tarefa de minimizar o BDD que representa essas funções é de grande importância, pois isto define a qualidade do circuito resultante. Desta forma, trabalhos como (CHANG;

CHENG; MAREK-SADOWSKA, 1994), (SHIPLE; HOJATI; SANGIOVANNI-VINCENTELLI; BRAYTON, 1994), (HONG; BEEREL; BURCH; MCMILLAN, 1997) e (MATSUURA; SASAO, 2007) apresentam diferentes maneiras de representar e minimizar funções Booleanas incompletamente especificadas através de BDDs.

O problema de minimização de BDD que representa uma função Booleana incompletamente especificada $[f, c]$ consiste em encontrar uma cobertura completamente especificada g de modo que $|g|$ seja a menor dentre todas as coberturas de $[f, c]$ (SHIPLE; HOJATI; SANGIOVANNI-VINCENTELLI; BRAYTON, 1994). Métodos de minimização que encontram o menor BDD que representam funções Booleanas incompletamente especificadas são classificados como NP-completo (SAUERHOFF; WEGENER, 1996). Com isto, algoritmos heurísticos foram propostos para este problema.

São conhecidas três maneiras para se representar, através de BDDs, funções Booleanas incompletamente especificadas, sendo elas: *i*) Função ternária, que possui como saída os valores 0, 1 e *don't care*; *ii*) Par de BDDs, que representa três valores e, *iii*) Utilizar uma variável extra, que permite representar os 0, 1 e *don't care* tudo junto em um mesmo BDD (MATSUURA; SASAO, 2007).

Na sequência são apresentados quatro trabalhos que abordam o problema de minimização de nodos de BDDs de funções Booleanas incompletamente especificas. E também são discutidas as formas como as funções são representadas por cada trabalho.

4.3 Trabalho 1 - Chang

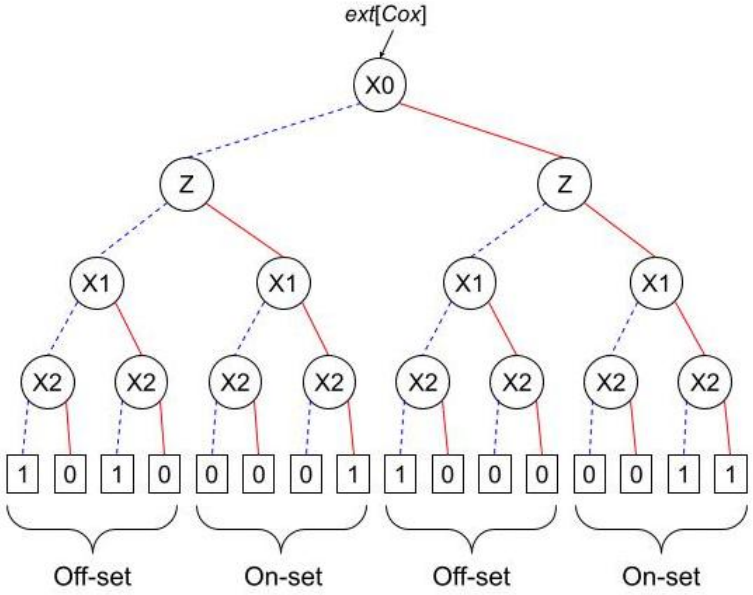
O primeiro trabalho a ser apresentado é o de (CHANG; CHENG; MAREK-SADOWSKA, 1994). Os autores apresentam uma técnica heurística que minimiza um ROBDD que representa uma função incompletamente especificada de múltiplas saídas. Basicamente, sua técnica consiste em atribuir os *don't cares* da função tanto para o On-set quanto ao Off-set de modo a gerar uma função completamente especificada que possui o menor tamanho de ROBDD. Mas para isso, utiliza uma representação que pode se tornar inadequada devido utilizar varáveis extra.

4.3.1 Representação da função

Em seu trabalho, (CHANG; CHENG; MAREK-SADOWSKA, 1994) representa uma função incompletamente especificada ff através de uma função estendida apelidada de $ext[ff]$, em que $ext[ff] = Z \cdot ff_{on} + \bar{Z} \cdot ff_{off}$. Desta forma, a variável extra Z é utilizada para carregar as informações do on-set, off-set e dos *don't cares* dentro de uma única função.

A Figura 4.2 apresenta o BDD da função estendida de Cox ($ext[Cox]$) da Figura 4.1. Como pode-se observar, em uma função estendida a variável extra têm como cofatores negativo e positivo o Off-set e o On-set da função, respectivamente. O On-set e Off-set de uma função são obtidos a partir das relações descritas na seção 2.3.4.

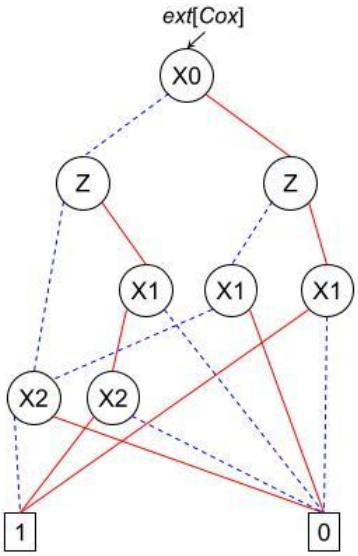
Figura 4.2 – Representação em BDD, de Chang, da função da Figura 4.1.



Fonte: Autor.

A Figura 4.3 apresenta o ROBDD obtido através do BDD da Figura 4.2. A partir deste ROBDD, (CHANG; CHENG; MAREK-SADOWSKA, 1994) executa o algoritmo chamado de *remove_Z* que é responsável pela remoção dos nodos que representam a variável Z no ROBDD.

Figura 4.3 – Representação em ROBDD, de Chang, da função da Figura 4.1.



Fonte: Autor, sugestão André Reis.

4.3.2 Objetivo do trabalho

Em seu trabalho, Chang propõe uma interpretação de uma função Booleana incompletamente especificada através de função estendida, que consiste em utilizar uma variável extra. O trabalho propõe um algoritmo heurístico cuja finalidade é minimizar do número de nodos de BDD.

Devido a representação adotada por (CHANG; CHENG; MAREK-SADOWSKA, 1994) incluir variáveis adicionais, diferente do padrão do CUDD (SOMENZI, 2018), não entraremos em detalhes neste método. Pois a inclusão de variáveis extras na representação intermediaria, antes de realizar a minimização, de Chang pode representar um aumento de consumo de memória para realizar a execução do método.

4.3.3 Contribuições do Trabalho

De acordo com o autor, o algoritmo é capaz de gerar uma cobertura completamente especificada mínima, em número de nodos de BDD, para a função. Porém, faz o uso de variável extra para representar a função a ser minimizada, o que pode ocasionar o aumento do número de nodos do BDD a ser minimizado.

4.4 Trabalho 2 - Sasao

Em seu trabalho, (MATSUURA; SASAO, 2007) utiliza BDD para função característica (BDD_for_CF, BDD for *Characteristic Function*, em inglês) para representar funções Booleanas incompletamente especificadas. Desta forma, o BDD utilizado possui nodos não terminais controlado por variáveis de entrada e de saída.

Uma função característica, introduzida por (CERNY; MARIN, 1977), envolve as variáveis de entrada e de saída para representar uma função Booleana. A saída de uma função característica representa as combinações possíveis do circuito (ASHAR; MALIK, 1995; SASAO; MATSUURA, 2004). Com isto, têm-se que dado I e O sendo os números de variáveis de entrada e saídas, respectivamente, de uma função. O número de combinações possíveis de vetores de entradas e valores de saídas para o circuito de uma função característica é 2^{I+O} (ASHAR; MALIK, 1995).

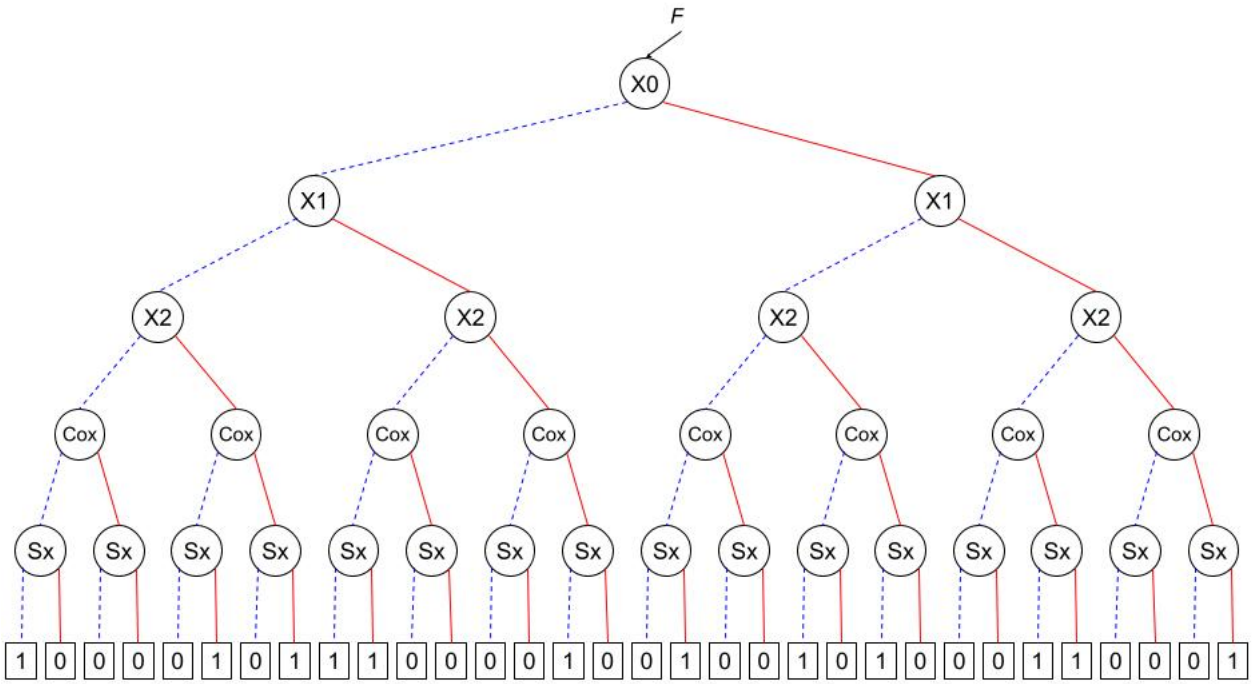
4.4.1 Representação da função

Para ilustrar a representação utilizada no trabalho de (MATSUURA; SASAO, 2007), a Figura 4.4 apresenta o BDD_for_CF que representa a função característica derivada da função Booleana descrita na Figura 4.1. Como pode ser visto na Figura 4.4, a função característica encontra-se representada através de um BDD totalmente expandido (árvore de decisão binária), portanto todo o espaço de possibilidades de combinações de entradas e saídas (2^{I+O}) da função estão representados. Os caminhos que levam o nodo terminal 1 e 0 são combinações válidas e não válidas, respectivamente.

Como pode se observar na Figura 4.4, todas as combinações, de valores de entradas e saídas, que são válidas geram a saída 1 no BDD_for_CF, como por exemplo a combinação $x_0 = 0$, $x_1 = 0$, $x2 = 0$, $C_{ox} = 0$ e $S_x = 0$. Enquanto que, as combinação não válidas geram o valor 0 no BDD_for_CF, como por exemplo a combinação $x_0 = 0$, $x_1 = 0$, $x2 = 0$, $C_{ox} = 0$ e $S_x = 1$.

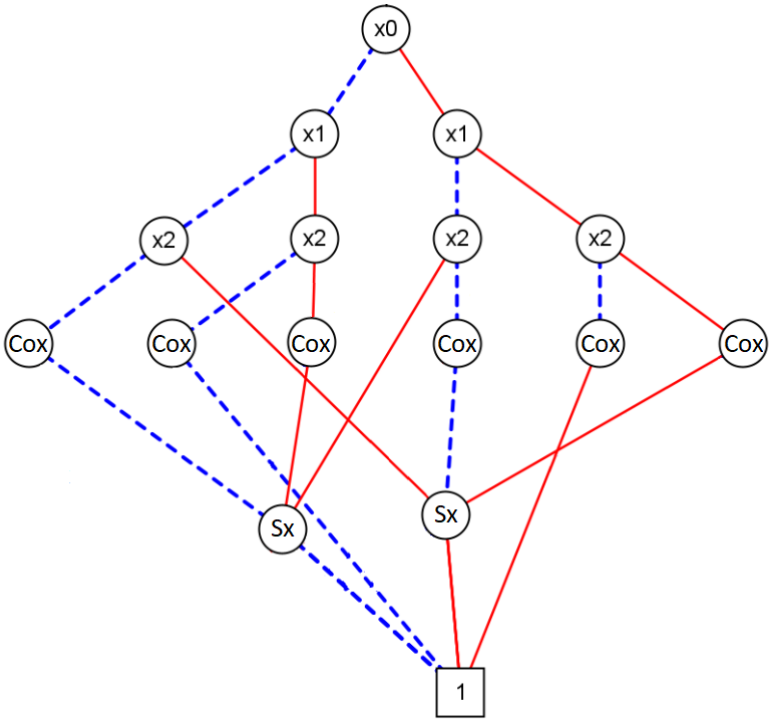
No exemplo da figura 4.4, o BDD_for_CF é apresentado completamente expandido para facilitar a visualização de todas as combinações. Em seu trabalho, (MATSUURA; SASAO, 2007) utiliza BDD_for_CF reduzido, como apresentado na Figura 4.5. Neste exemplo reduzido, as arestas que levam ao nodo terminal 0 são omitidas para simplificar a ilustração.

Figura 4.4 – BDD_for_CF não reduzido que representa a função característica derivada da função Booleana apresentada na 4.1.



Fonte: Autor.

Figura 4.5 – BDD_for_CF, reduzido, para a função 4.1.



Fonte: Autor.

4.4.2 Objetivo do trabalho

Em seu trabalho, (MATSUURA; SASAO, 2007) tem como objetivo reduzir a largura de BDD_for_CF que representam funções característica incompletamente especificadas de múltiplas saídas. No entanto, o BDD tem como entradas as saídas que são consideradas como variáveis de entrada, isto serve para agrupar várias funções (saídas) em um BDD de uma única saída. Para a obtenção de uma função Booleana ou de um circuito para cada função de saída é necessário eliminar as referências das variáveis de saída que aparecem como entrada, separando o BDD em vários.

Por exemplo, para obter o BDD de C_{ox} é feito $C_{ox} = 1$ e $S_x = 0$, obtendo-se o BDD de C_{ox}. Os nodos que só tem um filho (o outro está faltando) tem o filho faltante ligado a zero.

4.4.3 Contribuições do Trabalho

O trabalho do Prof. Sasao contribui mais para a representação de funções Booleanas do que para gerar uma representação diretamente sintetizável. O objetivo inicial desta dissertação era obter rapidamente um formado tipo ROBDD com o menor tamanho possível para uma única saída. A proposta de Sasao é representar várias funções Booleanas em um BDD com uma única saída (nodo raiz) de modo a guardar as informações de cada saída na estrutura da dados sem necessariamente produzir uma estrutura com múltiplas saídas (nodos raiz) como seria necessário para a síntese do hardware. Obviamente algoritmos poderiam ser criados para tentar extrair cada uma das funções da estrutura proposta por Sasao e extrair o hardware para cada uma delas. Mas este é um trabalho extenso e independente por si só e não havia tempo hábil para fazer os estudos e implementações necessárias dentro do escopo de uma dissertação.

4.5 Trabalho 3 - Shiple

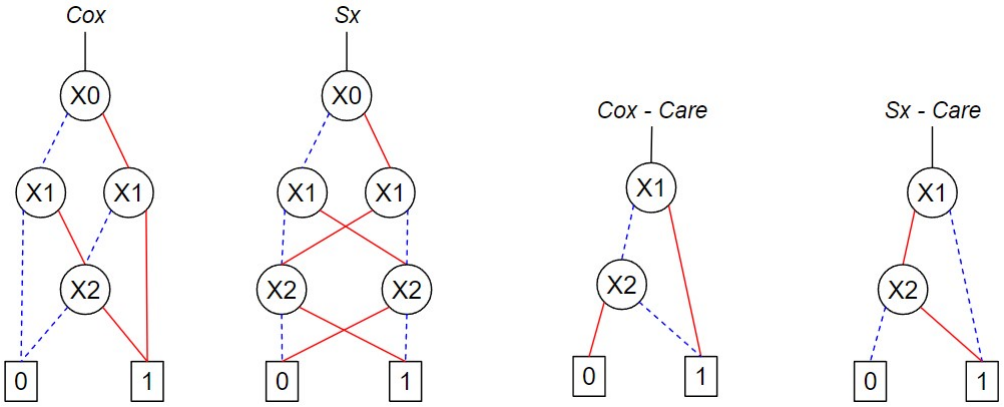
O trabalho de (SHIPLE; HOJATI; SANGIOVANNI-VINCENTELLI; BRAYTON, 1994) propõe um *framework* que explora variações para os operadores *restrict*

(COUDERT; BERTHET; MADRE, 1989) e *constrain* (COUDERT; MADRE, 1990). As heurísticas propostas em seu trabalho são baseadas no conceito atribuir valores à alguns dos *don't cares* e, desta forma tornar dois nodos de BDD iguais. Os autores chamam esta operação de *matching*.

4.5.1 Representação da função

Neste trabalho, a função Booleana é representada através de dois BDDs. O primeiro BDD representa uma função f completamente especificada a ser minimizada, enquanto que o segundo BDD representa o *Care-set* de f. Como apresentado na Figura 4.6.

Figura 4.6 – Representação, de Shiple, da função da Figura 4.1.



Fonte: Autor.

4.5.2 Objetivo do trabalho

O trabalho de (SHIPLE; HOJATI; SANGIOVANNI-VINCENTELLI; BRAYTON, 1994) propõe um *framework* para avaliar quais critérios de *matching* apresentam melhores resultados para a minimização de nodos de BDD que representam funções Booleanas incompletamente especificadas.

4.5.3 Contribuições do Trabalho

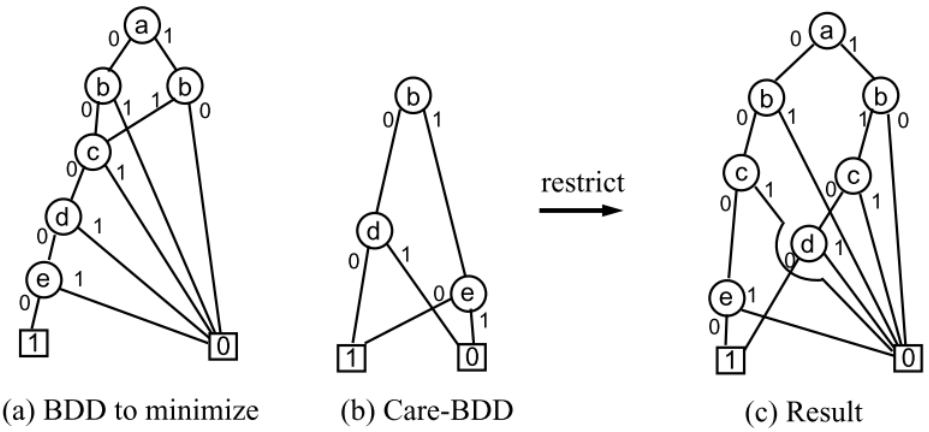
Os resultados experimentais de (SHIPLE; HOJATI; SANGIOVANNI-VINCENTELLI; BRAYTON, 1994) indicam que suas variações associadas ao operador *restrict* são

eficazes em redução de nodos de BDD e tempo de execução. No entanto, este tipo de técnica apresenta o problema de aumento do número de nodos do BDD resultante (HONG; BEEREL; BURCH; MCMILLAN, 1997).

4.6 Trabalho 4 - Hong

De acordo com (HONG; BEEREL; BURCH; MCMILLAN, 1997), o algoritmo de minimização *restric*, em alguns casos, gera um BDD maior do que o BDD original, como demonstrado na Figura 4.7, em que o nodo d do BDD da Figura 4.7(a) pode ser alcançado através de dois caminho a partir da raiz. Ao executar o método *restrict* seguindo as regras do *Care-BDD* Figura 4.7(b), identifica-se que para $b = 0$ e $d = 1$ leva um *don't care* e desta forma o nodo d (Figura 4.7(a)) pode ser substituído pelo nodo e.

Figura 4.7 – Exemplo de aumento do número de nodos de BDD do método *restrict*.



Fonte: Extraído de (HONG; BEEREL; BURCH; MCMILLAN, 1997).

No entanto, ao eliminar o nodo d a subárvore rooteada pelo nodo c, que inicialmente era compartilhado pelos nodos b, torna-se não compartilhável, como pode ser visto na Figura 4.7(c). Desta forma, houve a necessidade de criar novos nodos para o BDD resultante.

Dado o problema do aumento do número de nodos, (HONG; BEEREL; BURCH; MCMILLAN, 1997) propõe uma melhoria baseado no trabalho de (SHIPLE; HOJATI; SANGIOVANNI-VINCENTELLI; BRAYTON, 1994) que garante que o processo de minimização nunca gera um BDD maior do que o original. Sua técnica consiste em dividir o processo de minimização em duas etapas.

Na primeira etapa, chamada de *mark-edges*, é realizada uma avaliação e mar-

cação de arestas que devem ser mantidas após a minimização. Na segunda etapa, chamada de *build-result*, se um nodo v que possui uma aresta de saída não marcada que leve a um outro nodo u, então o nodo v pode ser substituído pelo nodo irmão de u.

4.6.1 Representação da função

Em seu trabalho, (HONG; BEEREL; BURCH; MCMILLAN, 1997) utiliza a mesma representação de função que (SHIPLE; HOJATI; SANGIOVANNI-VINCENTELLI; BRAYTON, 1994), apresentada na Figura 4.6. A função Booleana incompletamente especificada é representada através de dois BDDs, um para a função a ser minimizado e outro que representa o *care-set* da função, onde ambos BDDs representam funções Booleanas completamente especificadas.

4.6.2 Objetivo do trabalho

Diante do problema de aumento do número de nodos de BDD resultante do algoritmo *restrict*, (HONG; BEEREL; BURCH; MCMILLAN, 1997) propõe o algoritmo heurístico *leaf-identifying compaction* para minimização de nodos de BDD que garante o não aumento do tamanho do BDD. Para isto, propõe um método que, antes de realizar a minimização do BDD, realiza a marcação de arestas que caso sejam removidas podem resultar no aumento do BDD resultante. Após a etapa de marcação, executa a etapa de minimização através de combinação de nodos irmãos.

4.6.3 Contribuições do Trabalho

O trabalho de (HONG; BEEREL; BURCH; MCMILLAN, 1997) apresenta dois métodos de minimização, utilizando *don't cares*, de funções Booleanas incompletamente especificadas. O métodos são *leaf-identifying compaction* e *basic compaction*. Ambos métodos utilizam a técnica marcação de arestas para garantir o não aumento do tamanho do BDD cobertura gerado. De acordo com os autores, o método *leaf-identifying compaction* apresenta melhores resultados, pois trata-se de uma melhoria aplicada ao seu próprio método *basic compaction*.

4.7 Contribuições deste capítulo

Neste capítulo foi feita uma revisão dos métodos que utilizam BDDs para representar funções Booleanas incompletamente especificadas e que, além disso, apresentam como proposta minimizar o número de nodos de BDD. Apesar de um dos métodos listados, (MATSUURA; SASAO, 2007), não ter como objetivo a minimização de nodos de BDDs, trata-se de um método que trabalha com representação de funções Booleanas incompletamente especificadas, por este motivo foi apresentado nesta seção.

A Tabela 4.1 apresenta, de forma resumida, as principais característica que cada trabalho emprega em seus métodos. Cada linha da tabela corresponde a um dos métodos descritos neste capítulo, a última linha trata-se do métodos que é proposto neste trabalho. Na segunda coluna (SCF) é descrito se o método representa a função na forma canônica forte, na terceira coluna (Safe) é descrito se o método garante que o resultado gerado não seja pior do que a função de entrada, na quarta coluna é descrita a maneira como a função incompletamente especificada é representada, na quinta coluna é descrita a quantidade de BDDs utilizados para representar a função, a sexta coluna descreve como são representadas funções de múltiplas saídas e, por último a sétima coluna descreve as variáveis que compõem a representação do função através do(s) BDD(s).

Tabela 4.1 – Resumo dos métodos.

Artigo	FCF	Safe	BDDs	Quantidade de BDDs	Múltiplas saídas	BDD variáveis
Chang 94	Não	Não	{Função incompletamente especificada}	Um BDD por função	Representação separada	Entradas + Z
Sasao 07	Sim	Não	{Função característica}	Apenas um BDD para todas funções	Representação junta, resultados separados	Entradas e saídas
Shiple 94	Sim	Não	{Funçao; Care-set}	Dois BDDs por função	Representação separada	Entradas
Hong 97	Não	Sim	{Função; Care-set}	Dois BDDs por função	Representação separada	Entradas
Nosso	Sim	Em progresso	{On-set; Off-set}	Dois BDDs por função	Representação separada	Entradas

5 MÉTODO PROPOSTO

5.1 Sobre este capítulo

Neste capitulo será apresentada a metodologia empregada neste trabalho. Em um primeiro momento, são apresentadas as etapas que precedem a execução do método de minimização de funções Booleanas incompletamente especificadas, proposta deste trabalho, e na sequência o método em questão é apresentado.

5.2 Visão Geral da Metodologia

Nesta dissertação, trabalharemos com a classe de funções Booleanas incompletamente especificadas de saída única. Nos referimos aos nodos terminais 1 e 0 do BDD como T1 e T0, respectivamente. O On-set é representado por um primeiro BDD construído de forma que os valores de entrada associados ao valor de saída 1 pertençam ao On-set. Da mesma forma, o Off-set é representado por um segundo BDD construído de forma que os valores de entrada associados ao valor de saída 1 pertençam ao Off-set da relação Booleana original. Dessa forma, temos duas representações em BDD parciais para o On-set e o Off-set da função.

O algoritmo JOIN recebe um primeiro BDD para o On-set e um segundo BDD para o Off-set e recursivamente encontra um BDD mínimo representando uma função completamente especificada que satisfaça a relação Booleana para a ordem dos BDDs iniciais. Isso é feito aplicando recursivamente a decomposição de Shannon na forma canônica forte de BDDs (BRACE; RUDELL; BRYANT, 1990).

5.2.1 Função usada como exemplo

Para demonstrar o funcionamento do nosso algoritmo, utilizaremos o seguinte exemplo. Considere a função Booleana mostrada na tabela verdade na Figura 5.1(a), onde as saídas com valor 1 correspondem a On-set e, as saídas com valor 0 correspondem a Off-set, e as saídas com o valor X representam o don´t care set. A Figura 5.1(b) apresenta a descrição PLA correspondente, que é o formato de entrada do método Join. Neste trabalho é utilizado o formato PLA, pois o concurso IWLS 2020

(RAI et al., 2020) utiliza esse formato para especificar os seus *benchmarks*.

Figura 5.1 – Função Booleana incompletamente especificada utilizada para exemplificar o funcionamento do método Join.

x_0	x_1	x_2	x_3	F
0	0	0	0	x
0	0	0	1	x
0	0	1	0	x
0	0	1	1	0
0	1	0	0	0
0	1	0	1	x
0	1	1	0	x
0	1	1	1	x
1	0	0	0	x
1	0	0	1	0
1	0	1	0	x
1	0	1	1	x
1	1	0	0	x
1	1	0	1	x
1	1	1	0	x
1	1	1	1	1

(a) Especificação através de tabela verdade.

```
.i 4
.o 1
.p 4
.type fr
0011 0
0100 0
1001 0
1111 1
.e
```

(b) Especificação através de PLA.

Fonte: Autor.

5.3 Representação usada para On-set e Off-set

Este trabalho tem como alvo as funções Booleanas incompletamente especificadas, exclusivamente aquelas representadas apenas por seu on-set e off-set. Desta forma, se faz necessário representar os conjuntos on-set e off-set, da função original, através de uma estrutura de dados para posteriormente realizar manipulações. Na sequência é descrita a construção dos BDDs On-set e Off-set.

5.3.1 Construção do On-set

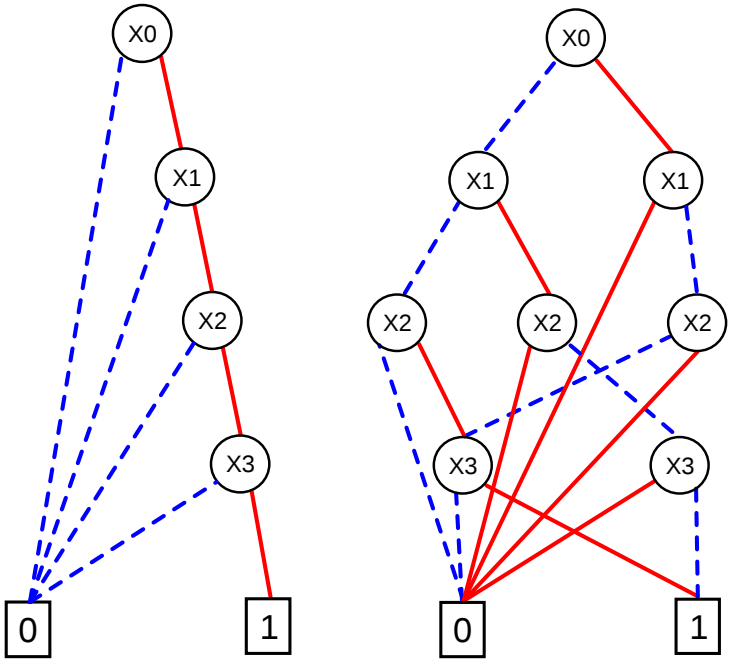
Os cubos/mintermos com valor de saída 1 são usados para construir o BDD On-set. Desta forma, os arranjos de variáveis que levam a saída T1 no On-set representam os 1's (on-set) da função original, enquanto a saída T0 do BDD On-set

representa uma mistura dos don't cares ou 0's (off-set) da função original. A Figura 5.2(a) apresenta o BDD On-set obtido da função da Figura 5.1.

5.3.2 Construção do Off-set

Os cubos/mintermos com valor 0 são utilizados para construir o BDD Off-set. Desta forma, no BDD Off-set as saídas T1 representam os 0's (off-set) da função especificada de forma incompleta. O terminal T0 representa uma mistura de *don't cares* e 1's (on-set). O BDD Off-set obtido, da função da Figura 5.1, é apresentado na Figura 5.2(b).

Figura 5.2 – Exemplo de BDD.



(a) On-set. (b) Off-set.

Fonte: Autor.

5.4 Rotina ITE Recursiva

Essa rotina não faz parte da proposta. Ela poderia ser descrita no capítulo de conceitos básicos. Porém optamos por descrevê-la aqui por facilidade de leitura, já que nossa proposta é baseada em uma modificação dessa rotina. Assim fica mais fácil de entender nossa proposta se ela for lida em sequência da descrição da rotina.

As operações/manipulações realizas em um ROBDD podem utilizar como base o operador ternário *If-Then-Else* (ITE). O operador ITE pode ser utilizado para implementar qualquer função Booleana de duas variáveis, como pode ser visto na Tabela 5.1. Dadas as entradas F, G e H, o operador ITE computa: Se F então G senão H (BRACE; RUDELL; BRYANT, 1990). Ou seja, se F é verdadeiro, então o operador retorna G, caso contrário retorna H. O ITE é definido pela seguinte expressão:

$$ite(F, G, H) = F \cdot G + \overline{F} \cdot H \tag{5.1}$$

Tabela 5.1 – Funcões Booleanas implementadas pelo operador ITE.

Nome	Expressão	Forma ITE
0	0	0
1	1	1
f	f	$ite(f, 1, 0)$
g	g	$ite(g, 1, 0)$
$f > g$	$f \cdot \overline{g}$	$ite(f, \overline{g}, 0)$
$f < g$	$\overline{f} \cdot g$	$ite(f, 0, g)$
$f \geq g$	$f + \overline{g}$	$ite(f, 1, \overline{g})$
$f \leq g$	$\overline{f} + g$	$ite(f, g, 1)$
$NOT(f)$	$\overline{f}$	$ite(f, 0, 1)$
$NOT(g)$	$\overline{g}$	$ite(g, 0, 1)$
$OR(f, g)$	$f + g$	$ite(f, 1, g)$
$AND(f, g)$	$f \cdot g$	$ite(f, g, 0)$
$NOR(f, g)$	$\overline{f + g}$	$ite(f, 0, \overline{g})$
$NAND(f, g)$	$\overline{f \cdot g}$	$ite(f, \overline{g}, 1)$
$XOR(f, g)$	$f \oplus g$	$ite(f, \overline{g}, g)$
$XNOR(f, g)$	$\overline{f \oplus g}$	$ite(f, g, \overline{g})$

Fonte: Adaptada de (BRACE; RUDELL; BRYANT, 1990).

Além das funções Booleanas primitivas apresentadas na Tabela 5.1, o operador ITE pode ser estendido para reescrever recursivamente qualquer função Booleana. Para isto, a decomposição de Shannon (SHANNON, 1949) é utilizada para decompor recursivamente a função em rezão de seus cofatores. A decomposição de Shannon é dada por:

$$F = v_i \cdot F_{v_i} + \overline{v}_i \cdot F_{\overline{v}_i} \tag{5.2}$$

onde F é uma função, v_i a variável de topo da i-ésima recursão, e F_{v_i} e $F_{\overline{v}_i}$ são os cofatores de F em relação a v_i.

5.5 Construção do Join

Neste trabalho, o algoritmo proposto é baseado no funcionamento do operador ITE. O algoritmo Join, Algoritmo 5.1, recursivamente compara os conjuntos On-set e Off-set ao mesmo nível, nó por nó, em uma abordagem *top-down*, e gera um terceiro BDD reduzido. Os pontos de terminação do algoritmo estão entre as linhas 2 e 7. O algoritmo verifica quando os nodos On-set e Off-set são T0, retornando *don't care*, uma vez que nenhum dos BDDs apresenta o valor de saída T1. Quando On-set é T0 e Off-set é T1, ele retorna T0 porque a atribuição de entrada particular está contida no Off-set. No último caso, quando On-set é T1 e Off-set é T0, o algoritmo retorna T1, indicando que a atribuição de entrada particular está contida no On-set.

Entre as linhas 9 e 13 do Algoritmo 5.1, seguindo o princípio da decomposição de Shannon, os próximos nodos são selecionados para fazer comparações em On-set e Off-set. Como o algoritmo atua em um ROBDD, a expansão segue a ordem das variáveis do ROBDD, então *idxTmp* mantém o índice da variável atual, que é encontrada primeiro na ordem. Quando os dois nodos processados não estão no mesmo nível no BDD (por exemplo, variáveis B e C), o nó inferior não expande a recursão para seus filhos. Os nodos de ambos os BDDs são processados considerando o nível de variável superior entre eles e, o alinhamento de topo é obtido nas linhas 10 e 11 através do retorno do Algoritmo topAlignment 5.2.

O Algoritmo topAlignment 5.2 é responsável pelo alinhamento de topo que verifica se existe uma diferença de nível do nodo *curtSet* com o índice da variável de topo *idxTmp*. Se *curtSet* se encontra abaixo (no BDD) de *idxTmp*, a recursão para o nodo verificado não é expandida (linha 2), caso contrário a seus filhos são selecionados para expandir a recursão (linha 5). Por fim o algoritmo retorna os nodos que irão compor as próximas chamadas do algoritmo Join.

Algoritmo 5.1: Join.

```
1  bdd_node Join(onset, offset)
2      if (onset = T0 and offset = T0)then
3          return TDC
4      else if (onset = T0 and offset = T1)then
5          return T0
6      else if (onset = T1 and offset = T0)then
7          return T1
8      else
9          idxTmp ← min(onset.var, offset.var)
10         {onset_cf0, onset_cf1} ← topAlignment(idxTmp, onset)
11         {offset_cf0, offset_cf1} ← topAlignment(idxTmp, offset)
12         n0 ← Join(onset_cf0, offset_cf0)
13         n1 ← Join(onset_cf1, offset_cf1)
14         if (n0 = TDC and n1 = TDC)then
15             nd_return ← TDC
16         else if (n0 = TDC)then
17             nd_return ← n1
18         else if (n1 = TDC)then
19             nd_return ← n0
20         else if (n0 = n1)then
21             nd_return ← n0
22         else
23             nd_return ← getOrCreateNd(idxTmp, n0, n1)
24         return nd_return
```

Algoritmo 5.2: Alinhamento de topo.

```
1  {curSet_cf0, curSet_cf1} topAlignment(idxTmp, curtSet)
2      if (idxTmp < curtSet.var or isTerminal(curtSet))then
3          curtSet_cf0 ← curtSet
4          curtSet_cf1 ← curtSet
5      else
6          curtSet_cf0 ← curtSet.f0
7          curtSet_cf1 ← curtSet.f1
8      return {curtSet_cf0, curtSet_cf1}
```

Após o retorno das chamadas recursivas, as variáveis *n0* e *n1* contêm os resultados retornados e são então consideradas para determinar o valor final de retorno. Se ambas as chamadas geraram *TDC* (*don't care*), então isso é retornado imediatamente (linha 15). Quando apenas um dos dois valores de retorno é *TDC*, o

valor que é diferente de *TDC* é selecionado como o valor de retorno final (linhas 17 ou 19). Quando *n0* é igual a *n1*, o valor final de retorno é qualquer um deles (linha 21). Quando *n0* e *n1* são diferentes, a função *getOrCreateNode()* é chamada na linha 23 para produzir o valor de retorno final, o que garante que um nodo repetido não seja criado.

O algoritmo JOIN considera o fato de que o On-set é construído a partir dos mintermos associados à saída 1. Enquanto que, o Off-set, está associado à saída 0. Assumindo a notação explicada anteriormente nesta seção. Para uma melhor compreensão do funcionamento do algoritmo JOIN 5.1, apresentaremos um passo a passo da construção do BDD com o JOIN.

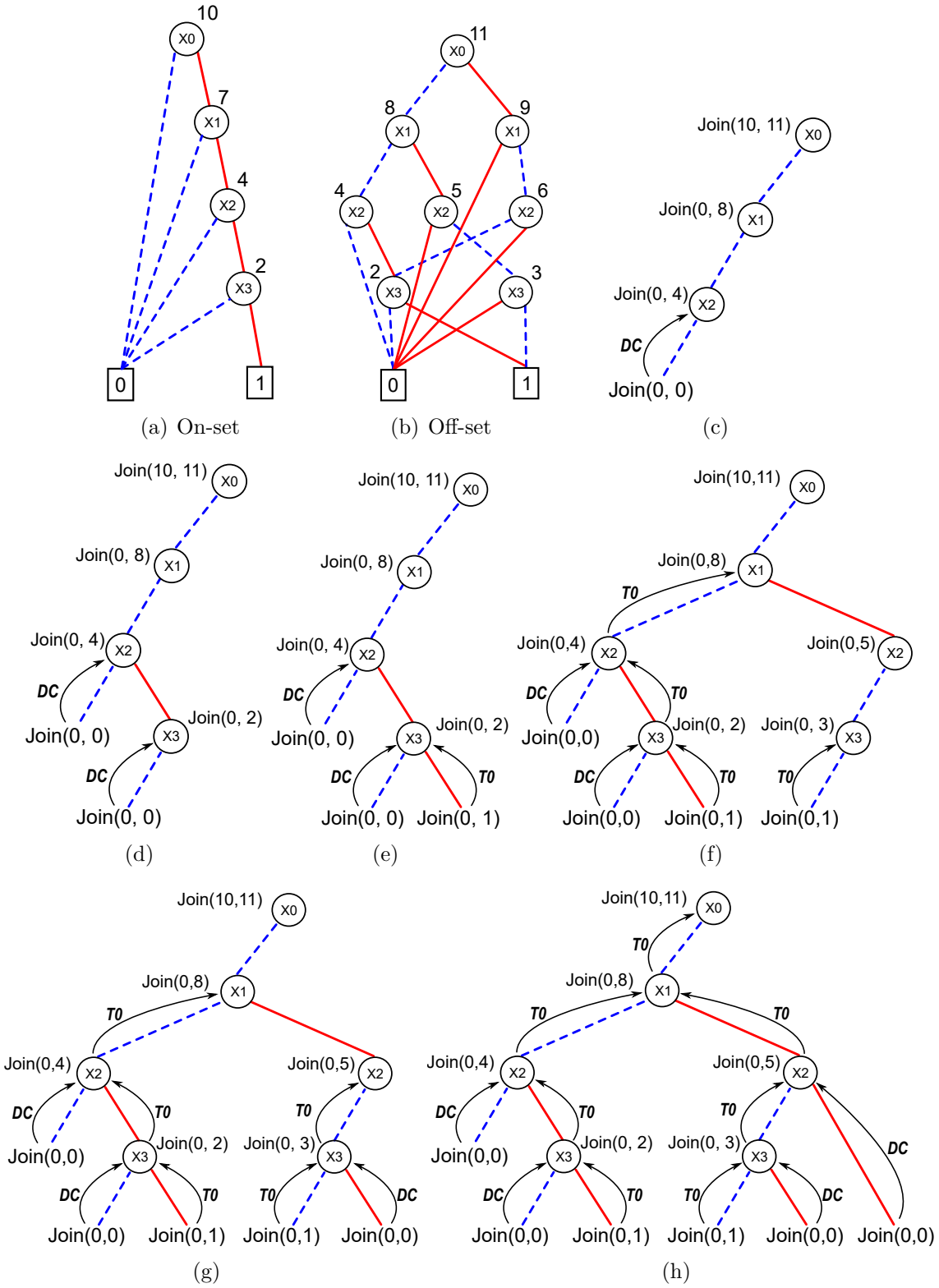
5.6 Execução do JOIN

Em nosso algoritmo os BDDs On-set e Off-set compartilham nodos entre si, como em (MINATO; ISHIURA; YAJIMA, 1990). Os BDDs On-set e Off-set, Figuras 5.3(a) e 5.3(b) respectivamente, são ilustrados separados para facilitar a visualização da execução do algoritmo JOIN. O número inteiro localizado próximo a cada nodo no BDD representa o seu índice identificador.

Como descrito no capítulo 3, a linha azul tracejada representa o filho F0 e a linha vermelha contínua representa o filho F1 do nodo. Na Figura 5.3 é ilustrado um exemplo de execução do Join, onde as flechas representam os valores de retorno das chamadas do algoritmo JOIN. Os BDDs On-set (Figura 5.3(a)) e Off-set (Figura 5.3(b)) são utilizados neste exemplo. Como os nodos 10 e 11 são, respectivamente, os nodos raízes dos BDDs On-set e Off-set, a execução do Join inicia com a chamada *Join(10,11)* e percorre os dois BDDs, como em uma busca em profundidade, expandindo a recursão primeiro para os filhos F0 de cada BDD. Com isso, o primeiro caminho percorrido neste exemplo é $x_0 = 0, x_1 = 0, x_2 = 0$, como apresentado na Figura 5.3(c). Este caminho leva ao valor de saída 0 por ambos os BDDs On-set e Off-set, então se trata de uma combinação associada à saída *don't care* e o algoritmo retorna o valor DC.

Seguindo com a execução, ao receber o retorno de F0 a chamada Join(0,4) expande a recursão para os filhos F1 de cada BDD (On-set e Off-set), Figura 5.3(d), assim é criada a chamada Join(0, 2) correspondente ao caminho $x_0 = 0, x_1 = 0, x_2 = 1$. Desta vez a variável x_3 aparece no caminho, onde são criadas as recursões para

Figura 5.3 – Demonstração da execução do método JOIN.



(a) On-set (b) Off-set (c)

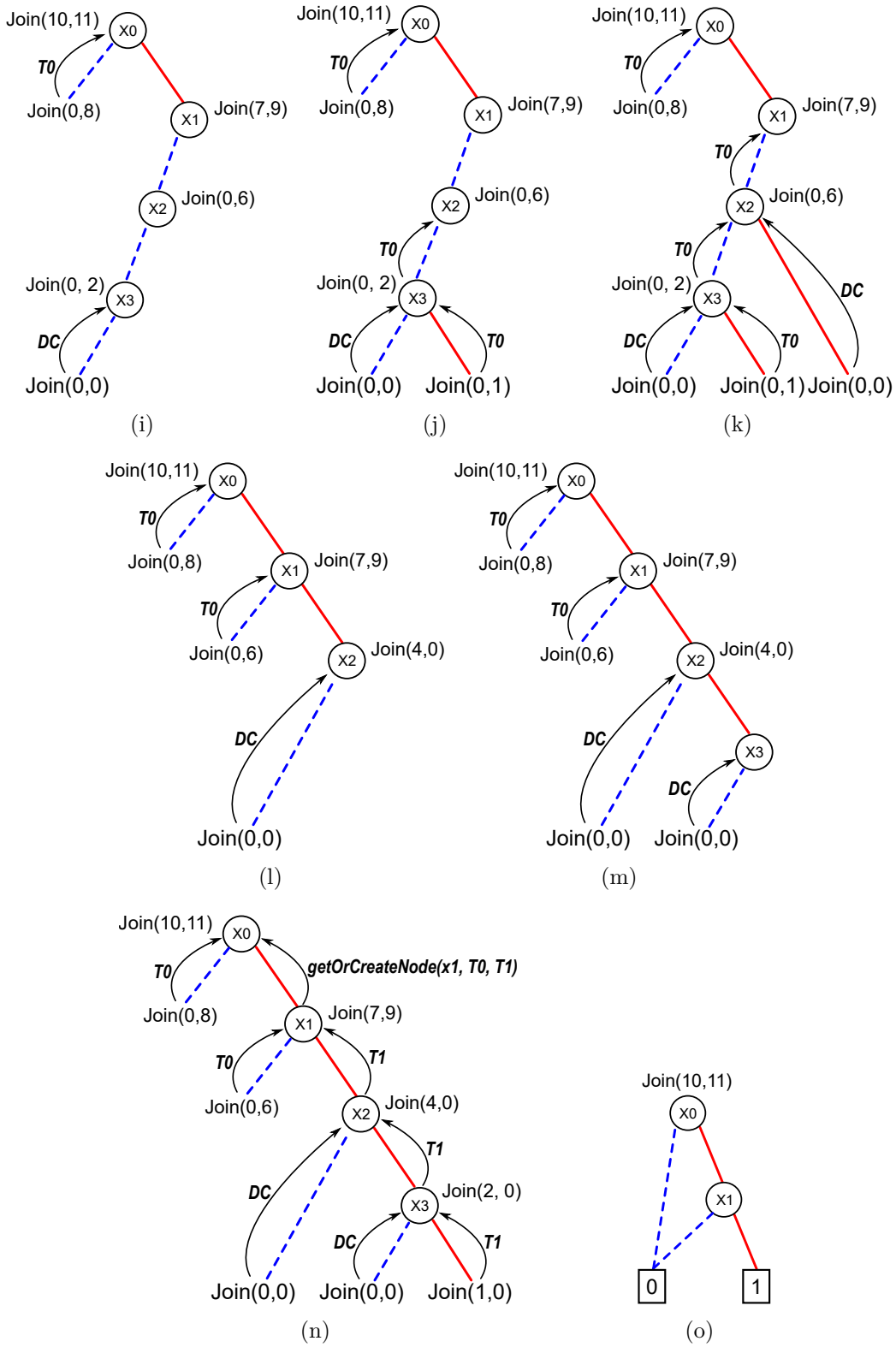
(d) (e) (f)

(g) (h)

Fonte: Autor.

$x_3 = 0$ (Figura 5.3(d)) e $x_3 = 1$ (Figura 5.3(e)) que geram os valores DC e 0, respectivamente. Como uma das chamadas retorna DC o resultado da outra chamada é retornado, pois DC indica que tal caminho nem o BDD do On-set nem do Off-set

Figura 5.3 – Continuação da figura: Demonstração da execução do método JOIN.



Fonte: Autor.

geram uma saída especificada. Portanto, a chamada Join(0,2) retorna o valor 0 indicando que o através do caminho $x_0 = 0, x_1 = 0, x_2 = 1$ a função original gera a saída 0, isso pode ser observado na parte esquerda da Figura 5.3(f).

Entre as Figuras 5.3(c) e 5.3(h) é ilustrado o processo de exploração dos caminhos com $x_0 = 0$ e todas as possíveis combinações para as variáveis $, x_1, x_2, x_3$ através dos BDDs do On-set e Off-set. Nas figuras podemos observar a sequência gerada de retornos de valores 0 e *don't care*, que de acordo com as regras descritas no Algoritmo 5.1 resulta no valor 0 a partir do caminho setando a variável x_0 com o valor 0.

Após receber o retorno do filho F0, a chamada Join(10, 11) expande recursão através do On-set e Off-set para os filhos F1, com isso é criada a chamada Join(7,9). Como dito anteriormente, seguindo a sequência de uma busca em profundidade, a chamada Join(7,9) cria a chamada Join(0,6) que é a expansão do filho F0 do nodo 7 (nodo 0) e do filho F0 do nodo 9 (nodo 6), que por sua vez cria a chamada Join(0,2), como pode ser visto na Figura 5.3(i).

Entre as Figuras 5.3(j) e 5.3(n) são ilustradas chamadas recursivas restantes executas pelo método Join nesse exemplo. A última recursão, Figura 5.3(n), é expandida para os filhos F1 da chamada Join(2,0) que resulta no valor de retorno T1, sinalizando que o caminho percorrido $x_0 = 1, x_1 = 1, x_2 = 1, x_3 = 1$ gera uma saída 1 na especificação original. A chamada Join(2,0) recebe os retornos DC e T1 de F0 e F1, respectivamente. De acordo com a linha 16 do Algoritmo 5.1, o valor de retorno computado é T1. Da mesma forma, a chamada Join(4,0) recebe os mesmos valores de retorno da chamada Join(2,0), então computa seu valor de retorno que é T1.

Por último, ainda na Figura 5.3(n), a chamada Join(7, 9) recebe como retorno de F0 e F1 os valores T0 e T1, respectivamente. Com isso é invocado o método *getOrCreateNode* que cria uma novo nodo marcado com a variável *X1* e que possui como filhos F0 e F1 os valores 0 e 1, nesta ordem. Desta forma, a chamada Join(10,11) recebe o valor 0 pelo filho F0 e o nodo x_1 através do filho F1. Assim é gerado o BDD apresentado na Figura 5.3(o). Ou seja, o BDD gerado pelo Algoritmo JOIN é uma redução do BDD do On-set.

Os principais pontos desse exemplo podem ser observados nos itens a e b da Figura 5.3. Ao se atribuir o valor 1 às variáveis x_0 e x_1 o BDD do Off-set gera a saída 0, ou seja, a partir desse caminho não é possível gerar o valor 0 na saída da função original. Por outro lado, existe um caminho no BDD do On-set a partir de $x_0 = 1$ e $x_1 = 1$ que gera a saída 1 na função original. Portanto, independentemente dos valores das variáveis x_2 e x_3 o valor de saída da função original será 1. Então

os nodos 2 e 4 podem ser removidos, o que resulta no BDD apresentado na Figura 5.3(g).

A Figura 5.3 mostra o BDD representando uma função completamente especificada que satisfaz a relação Booleana original que é gerada pelo o JOIN partir do On-set unido ao Off-set. O BDD resultante tem um número significativamente menor de nodos em comparação com os dois BDDs iniciais. O algoritmo JOIN recebe o On-set e o Off-set como parâmetros e os combina para gerar um BDD minimizado que satisfaça a relação Booleana original expressa pelo par de BDDs iniciais.

Através da Figura 5.4 é possível acompanhar a sequência em que as chamadas recursivas do algoritmo JOIN são executadas. Pode-se observar o comportamento do JOIN que explora primeiro o caminho atribuindo o valor 0 à variável e desta forma expande a recursão para o filho F0 do BDD do On-set e do Off-set e, após atingir um nodo terminal, explora o filho F1 (caminho atribuindo o valor 1 à variável).

Figura 5.4 – Chamadas recursivas da execução do exemplo da Figura 5.3.

```
1  Join(10, 11)
2  |  --x0=0--> Join(0, 8)
3  |  |  --x1=0--> Join(0, 4)
4  |  |  |  --x2=0--> Join(0, 0)
5  |  |  |  |  return TDC
6  |  |  |  --x2=1--> Join(0, 2)
7  |  |  |  |  --x3=0--> Join(0, 0)
8  |  |  |  |  |  return TDC
9  |  |  |  |  --x3=1--> Join(0, 1)
10 |  |  |  |  |  return T0 //(0, 0, 1, 1) atingiu a saída 1 pelo Off-set
11 |  |  |  |  n0 = TDC, n1 = T0, return T0
12 |  |  |  n0 = TDC, n1 = T0, return T0
13 |  |  |  --x1=1--> Join(0, 5)
14 |  |  |  |  --x2=0--> Join(0, 3)
15 |  |  |  |  |  --x3=0--> Join(0, 1)
16 |  |  |  |  |  |  return T0 //(0, 1, 0, 0) atingiu a saída 1 pelo Off-set
17 |  |  |  |  |  --x3=1--> Join(0, 0)
18 |  |  |  |  |  |  return TDC
19 |  |  |  |  n0 = T0, n1 = TDC, return T0
20 |  |  |  |  --x2=1--> Join(0, 0)
21 |  |  |  |  |  return TDC
22 |  |  |  n0 = T0, n1 = TDC, return T0
23 |  |  n0 = T0, n1 = T0, return T0
24 |  |  --x0=1--> Join(7, 9)
25 |  |  |  --x1=0--> Join(0, 6)
26 |  |  |  |  --x2=0--> Join(0, 2)
27 |  |  |  |  |  --x3=0--> Join(0, 0)
28 |  |  |  |  |  |  return TDC
29 |  |  |  |  |  --x3=1--> Join(0, 1)
30 |  |  |  |  |  |  return T0 //(1, 0, 0, 1) atingiu a saída 1 pelo Off-set
31 |  |  |  |  n0 = TDC, n1 = T0, return T0
32 |  |  |  |  --x2=1--> Join(0, 0)
33 |  |  |  |  |  return TDC
34 |  |  |  n0 = T0, n1 = TDC, return T0
35 |  |  |  --x1=1--> Join(4, 0)
36 |  |  |  |  --x2=0--> Join(0, 0)
37 |  |  |  |  |  return TDC
38 |  |  |  |  --x2=1--> Join(2, 0)
39 |  |  |  |  |  --x3=0--> Join(0, 0)
40 |  |  |  |  |  |  return TDC
41 |  |  |  |  |  --x3=1--> Join(1, 0)
42 |  |  |  |  |  |  return T1 //(1, 1, 1, 1) atingiu a saída 1 pelo On-set
43 |  |  |  |  n0 = TDC, n1 = T1, return T1
44 |  |  |  n0 = TDC, n1 = T1 return T1
45 |  |  n0 = T0, n1 = T1, return getOrCreateNode(x1, T0, T1) // create nd12
46 |  n0 = T0, n1 = nd12, return getOrCreateNode(x0, T0, nd12)
```

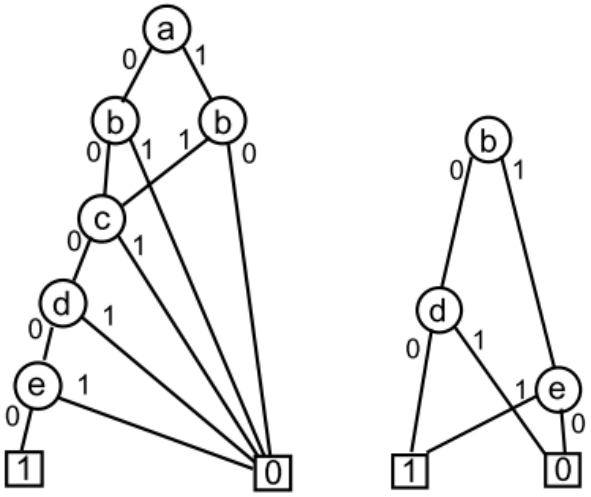
Fonte: Autor.

5.7 Join2: uma versão aprimorada

Nesta seção será discutida uma melhoria aplicada ao algoritmo 5.1, proposto neste trabalho. Para melhor compreender a rotina que torna o algoritmo 5.1 mais eficaz em termo de redução de nodos de BDD, primeiro será utilizado o exemplo do BDD abordado no trabalho de (HONG; BEEREL; BURCH; MCMILLAN, 1997) e, na sequência é apresentado a melhoria aplicada ao nosso método Join.

Como discutido na seção 4.6, (HONG; BEEREL; BURCH; MCMILLAN, 1997) representa uma função Booelana incompletamente especificada através de duas função completamente especificadas (On-set e DC-set). O exemplo que será utilizado é apresentado na Figura 5.5.

Figura 5.5 – Exemplo Hong.



(a) BDD a ser minimizado (b) Care-set

Fonte: Extraído de (HONG; BEEREL; BURCH; MCMILLAN, 1997).

Para uma melhor compreensão do exemplo, na Figura 5.6 apresenta a tabela verdade do exemplo dado em (HONG; BEEREL; BURCH; MCMILLAN, 1997). As colunas F e *care* correspondem, respectivamente, ao BDD a ser minimizado (Figura 5.5(a)) e ao Care-set (Figura 5.5(b)). As colunas *On-set* e *Off-set* são obtidas a partir das relações discutidas na seção 2.3.4.

Os BDDs correspondentes ao onset e offset da função são apresentados na Figura 5.7. Eles serão utilizados para demonstrar a melhoria proposta para o Join. Como foi apresentado na seção anterior, o algoritmo Join 5.1 realiza o procedimento de assinalamento de *Don't cares* com base nas informações contidas nos BDDs On-set e Off-set. A melhoria segue o princípio de exploração do Join, entretanto reforça

Figura 5.6 – Tabela verdade da função de Hong.

a	b	c	d	e	F	Care-set	On-set	Off-set
0	0	0	0	0	**1**	**1**	**1**	0
0	0	0	0	1	0	**1**	0	**1**
0	0	0	1	0	0	0	0	0
0	0	0	1	1	0	0	0	0
0	0	1	0	0	0	**1**	0	**1**
0	0	1	0	1	0	**1**	0	**1**
0	0	1	1	0	0	0	0	0
0	0	1	1	1	0	0	0	0
0	1	0	0	0	0	0	0	0
0	1	0	0	1	0	**1**	0	**1**
0	1	0	1	0	0	0	0	0
0	1	0	1	1	0	**1**	0	**1**
0	1	1	0	0	0	0	0	0
0	1	1	0	1	0	**1**	0	**1**
0	1	1	1	0	0	0	0	0
0	1	1	1	1	0	**1**	0	**1**
1	0	0	0	0	0	**1**	0	**1**
1	0	0	0	1	0	**1**	0	**1**
1	0	0	1	0	0	0	0	0
1	0	0	1	1	0	0	0	0
1	0	1	0	0	0	**1**	0	**1**
1	0	1	0	1	0	**1**	0	**1**
1	0	1	1	0	0	0	0	0
1	0	1	1	1	0	0	0	0
1	1	0	0	0	**1**	0	0	0
1	1	0	0	1	0	**1**	0	**1**
1	1	0	1	0	0	0	0	0
1	1	0	1	1	0	**1**	0	**1**
1	1	1	0	0	0	0	0	0
1	1	1	0	1	0	**1**	0	**1**
1	1	1	1	0	0	0	0	0
1	1	1	1	1	0	**1**	0	**1**

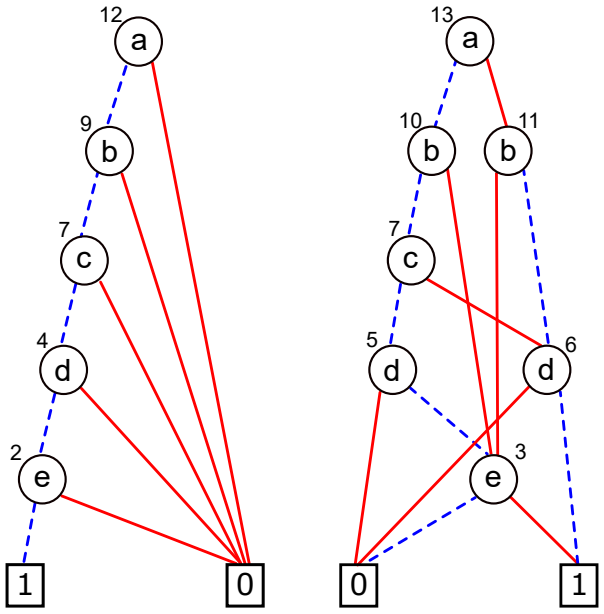
Fonte: Autor.

a minimização durante o retorno das chamadas recursivas do Join 5.1.

A Figura 5.7 será útil para auxiliar na compreensão da melhoria. A Figura 5.8(a) demonstra resumidamente a execução do algoritmo Join2 5.3, as chamadas recursiva sinalizadas com três pontos (...) indicam que a execução teve o comportamento normal do Join 5.1, onde não foi possível realizar otimizações. O retorno do Join é uma cobertura parcial (para a subárvore atual), ilustrado com uma tripla (variável, F_0, F_1), para facilitar o entendimento sem se preocupar com índices de novos nodos criados.

Ainda na Figura 5.8(a), a chamada recursiva *Join(9, 10)* recebe como retorno (c, 2, T0) e T0. O retorno (c, 2, T0) é ilustrado na Figura 5.8(b). Por último, a tabela apresentada na Figura 5.8(c) descreve o comportamento de três BDDs, cujas

Figura 5.7 – BDDs On-set e Off-set do exemplo do Hong.



(a) On-set (b) Off-set

Fonte: Autor.

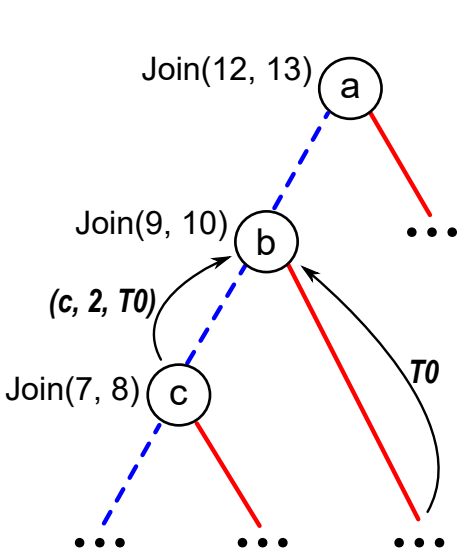
raízes são os nodos 9 (pertencente ao onset), 10 (pertencente ao offset) e (c, 2, T0) (retorno do método Join2), a variável A não compõem a tabela pois BDDs avaliados não dependem dela.

Na Figura 5.8(c), é demonstrado pela tabela verdade que o BDD (c, 2, T0), gerado através do assinalamento de *don't cares* pelo Join2 5.3, é uma cobertura para a função incompletamente especificada representada pelo par de BDDs nodos 9 e 10 (On-set e Off-set, respectivamente). Na tabela é demonstrado que o BDD (c, 2, T0) é uma cobertura válida para os nodos 9 e 10, onde os valores de saída do On-set são destacados na cor vermelha com o BDD (c, 2, T0) e, os valores de saída do Off-set são destacados na cor azul com o BDD (c, 2, T0). Para (c, 2, T0) ser uma cobertura válida, deve ter como saída os valores 1 quando On-set for 1, 0 quando o Off-set for 1 e, pode assumir 0 ou 1 quando On-set e Off-set forem 0.

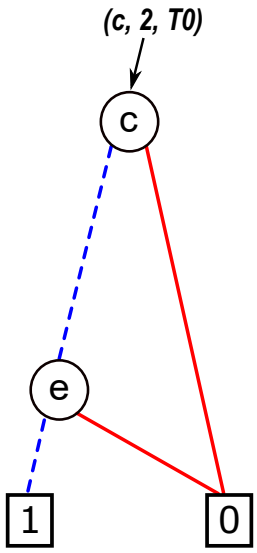
Desta forma, a cobertura (c, 2, T0) pode ser utilizada no lugar do nodo 9 e, como (c, 2, T0) não depende da variável B, pode resultar em uma redução do número de nodos do BDD. É proposto o algoritmo *isItPossible* 5.4 para realizar esta análise. O *isItPossible* compõem o algoritmo Join 5.1 o que resulta no algoritmo Join2 5.3. Portanto, os detalhes de casos terminais, alinhamento de topo do Join se mantém para o Join2.

Dado um nodo Nd_{ith}, o algoritmo *isItPossible* 5.4 verifica se algumas das

Figura 5.8 – Minimização executada pelo método Join2.



(a) Execução resumida Join.



(b) Retorno da chamada recursiva Join(7, 8).

Entradas				On-set	Off-set	Retorno Join2
b	c	d	e	Nodo 9	Nodo 10	(c, 2, T0)
0	0	0	0	1	0	1
0	0	0	1	0	1	0
0	0	1	0	0	0	1
0	0	1	1	0	0	0
0	1	0	0	0	1	0
0	1	0	1	0	1	0
0	1	1	0	0	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	1
1	0	0	1	0	1	0
1	0	1	0	0	0	1
1	0	1	1	0	1	0
1	1	0	0	0	0	0
1	1	0	1	0	1	0
1	1	1	0	0	0	0
1	1	1	1	0	1	0

(c) Tabela verdade para a funções representadas pelos nodos 9, 10 e (c,2,0).

Fonte: Autor.

coberturas obtidas para seus filhos ($F_{Nd_{ith}}(0)$ e $F_{Nd_{ith}}(1)$) também é capaz de cobrir Nd_{ith}. O algoritmo recebe como entrada três nodos de BDD, sendo eles as raízes da

cobertura parcial, do On-set e do Off-set.

A partir disto realiza as atribuição de valores, percorre, simultaneamente os três BDDs e analisa os valores de saída obtidos. Se On-set e Off-set são *T0* (linha 2), então o caminho leva a *don't care* e portanto a cobertura é válida para aquela combinação de entrada. As demais verificações consistem em avaliar se a cobertura, apelidada de *cand* é compatível com o On-set e Off-set (linhas 4 e 9). Se algum dos caminhos avaliados não for compatível entre a cobertura e On-set/Off-set, o algoritmo sinaliza com *IMPS*.

5.8 Contribuições deste capítulo

Este capítulo apresenta o método Join, a proposta deste trabalho. O Join é um método para gerar um BDD cobertura completamente especificado para uma função Booleana incompletamente especificada representada por um par de BDDs completamente especificados. O método Join realiza assinalamento de *don't cares* com base no On-set e Off-set da função de entrada. Além disso, é apresentada uma melhoria aplicada ao próprio método Join, chamada de Join2, que reforça a busca por coberturas menores durante o retorno das chamadas do método Join.

Os algoritmos dos métodos são apresentados e detalhados e, também são apresentados exemplos do funcionamento do método proposto para facilitar a compreensão para o leitor.

Algoritmo 5.3: Join2, versão *safe*.

```
1  bdd_node Join2(onset, offset)
2      if (onset = T0 and offset = T0)then
3          return TDC
4      else if (onset = T0 and offset = T1)then
5          return T0
6      else if (onset = T1 and offset = T0)then
7          return T1
8      else
9          idxTmp ← min(onset.var, offset.var)
10         {onset_cf0, onset_cf1} ← topAlignment(idxTmp, onset)
11         {offset_cf0, offset_cf1} ← topAlignment(idxTmp, offset)
12         n0 ← Join2(onset_cf0, offset_cf0)
13         n1 ← Join2(onset_cf1, offset_cf1)
14         if (n0 = TDC and n1 = TDC)then
15             nd_return ← TDC
16         else if (n0 = TDC)then
17             nd_return ← n1
18         else if (n1 = TDC)then
19             nd_return ← n0
20         else if (n0 = n1)then
21             nd_return ← n0
22         else
23             forced_n0 ← isItPossible(n1, onset, offset)
24             forced_n1 ← isItPossible(n0, onset, offset)
25             if (forced_n0=IMPS and forced_n1=IMPS)then
26                 nd_return ← getOrCreateNd(idxTmp, n0, n1)
27             else if (forced_n0 = n1)then
28                 nd_return ← n1
29             else if (forced_n1 = n0)then
30                 nd_return ← n0
31         return nd_return
```

Algoritmo 5.4: isItPossible.

```
1  bdd_node isItPossible(cand, onset, offset)
2      if (onset = T0 and offset = T0)then
3          return cand
4      else if (onset = T0 and offset = T1)then
5          if (cand = T0)then
6              return T0
7          else
8              return IMPS
9      else if (onset = T1 and offset = T0)then
10         if (cand = T1)then
11             return T1
12         else
13             return IMPS
14     else
15         idxTmp ← min(cand.var, onset.var, offset.var)
16         (onset_cf0, onset_cf1) ← topAlignment(idxTmp, onset)
17         (offset_cf0, offset_cf1) ← topAlignment(idxTmp, offset)
18         (cand_cf0, cand_cf1) ← topAlignment(idxTmp, cand)
19         n0 ← isItPossible(cand_cf0, onset_cf0, offset_cf0)
20         n1 ← isItPossible(cand_cf1, onset_cf1, offset_cf1)
21         if (n0 = IMPS or n1 = IMPS)then
22             nd_return ← IMPS
23         else
24             nd_return ← cand
25         return nd_return
```

6 EXPERIMENTOS E RESULTADOS

6.1 Sobre este capítulo

Neste capítulo são apresentados os experimentos conduzidos para avaliar o eficácia do método Join proposto neste trabalho. Os experimentos são divididos em duas etapas, sendo elas 1) o método Join é comparado com a minimização executada pelo ESPRESSO e, 2) o método Join é comparados com os métodos *leaf-identifying compaction* (HONG; BEEREL; BURCH; MCMILLAN, 1997), *restric* (COUDERT; BERTHET; MADRE, 1989), e *constrain* (COUDERT; MADRE, 1990).

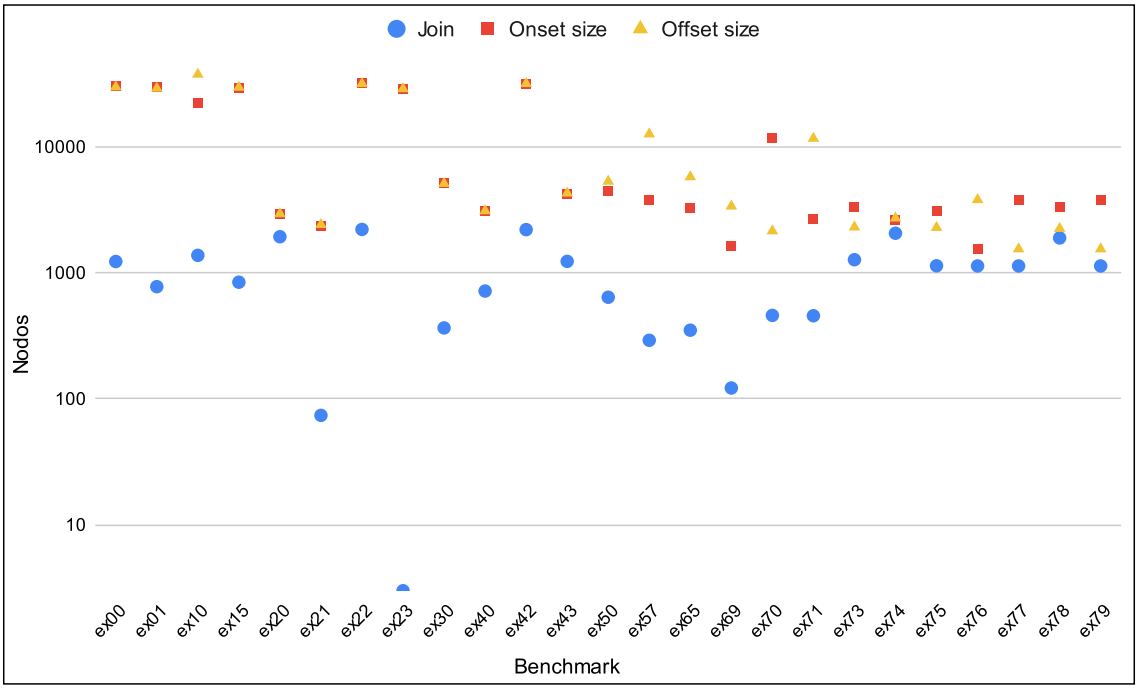
6.2 Resultados do método Join

Para avaliar o método proposto, foram utilizados os *Benchmarks* do concurso de programação IWLS 2020. Esses benchmarks foram propostos em outro contexto, para explorar técnicas de aprendizado de máquina aplicadas a desafios de Síntese Lógica. No entanto, eles contêm grandes funções incompletamente especificadas, definidas por seu On-set e Off-set. Em nossos experimentos, os BDDs para o On-set e para o Off-set de cada circuito de benchmark foram construídos e, em seguida, o algoritmo JOIN proposto foi aplicado para juntar os dois conjuntos em um BDD completamente especificado.

6.2.1 Avaliação da Redução de Nodos

Considerando uma avaliação independente do JOIN, o algoritmo apresentou uma redução média de 73,88% de redução na contagem de nodos em relação à contagem de nodos do BDD para o On-set. A Figura 6.1 mostra os resultados em termos de contagem de nodos BDD obtidos pelo algoritmo JOIN para cada benchmark em comparação com os tamanhos de On-set e Off-set usados como entrada. Os valores no eixo Y correspondem à contagem de nodos dos BDDs, em escala logarítmica. Observe que os resultados dependem da ordem da variável, portanto, para cada circuito mostrado no eixo X, os resultados relatados são para a ordem da variável específica que resultou na melhor contagem de nodos para o método JOIN.

Figura 6.1 – Melhores resultados do algoritmo JOIN com os *benchmarks* do concurso IWLS 2020.



Fonte: Autor.

6.2.2 Efeito do Ordenamento

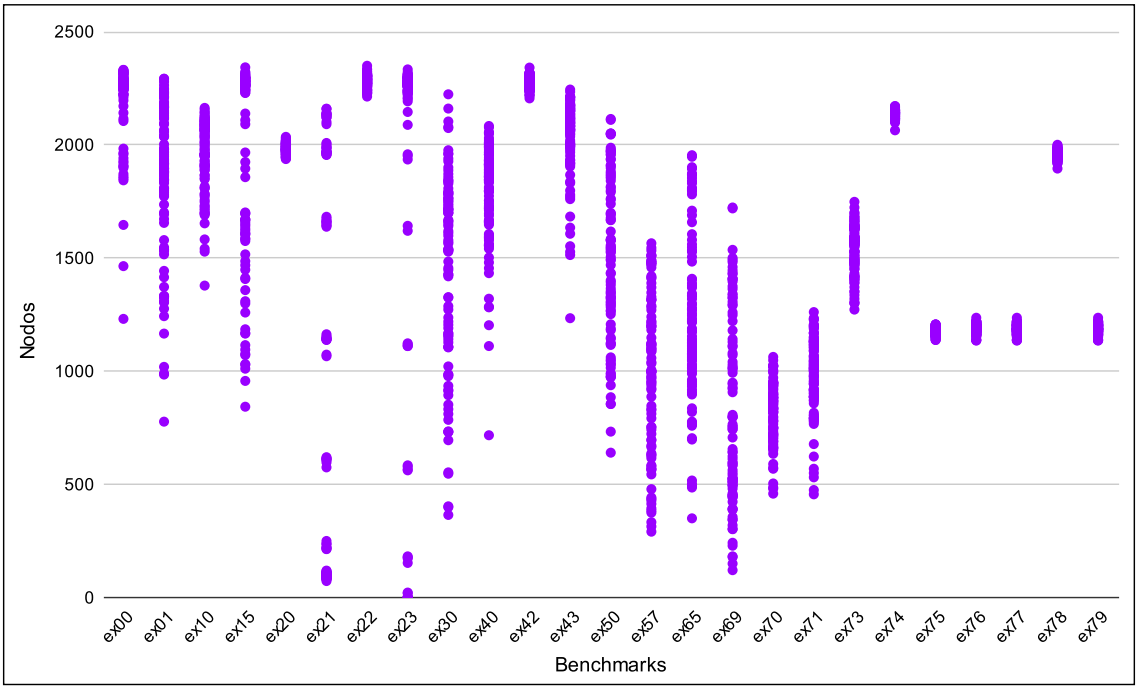
Para ilustrar o efeito da ordenação das variáveis de um BDD, que é conhecido por ter um grande impacto em aplicativos BDD, geramos 100 ordens aleatórias diferentes para cada circuito e calculamos o tamanho dos BDDs resultantes. As contagens de nodos para os BDDs resultantes são mostradas na Figura 6.2. É possível perceber que a ordem BDD impacta fortemente nos resultados conforme o esperado.

6.2.3 Comparação com ESPRESSO

Nosso método foi comparado ao ESPRESSO, que também realiza a síntese de funções incompletamente especificadas. Os resultados são apresentados em duas tabelas distintas, que são descritas a seguir.

A tabela 6.1 apresenta a comparação entre JOIN e ESPRESSO em termos de número de nodos de BDD, Cubos e, Literais, assim como o tempo de execução apenas para síntese (tempo para Join e ESPRESSO completar a execução), denotados na tabela como: N, C, L e T, nesta ordem.

Figura 6.2 – Influência da ordem das variáveis no tamanho do BDD.



Fonte: Autor.

Reconhecemos que Join e ESPRESSO fazem coisas diferentes enquanto produzem uma função completamente especificada que satisfaz a relação booleana. O JOIN minimiza nodos BDD, enquanto o ESPRESSO minimiza o número de literais em uma soma de produtos (SOP). Isso é mostrado na Tabela 6.1, com algumas exceções. Na terceira coluna da tabela, os valores marcados em negrito foram aqueles em que o algoritmo Join possuía menos nodos de BDD do que ESPRESSO. Em termos de tempo de execução, a sexta coluna mostra o tempo gasto pelo algoritmo JOIN para construir o BDD final, incluindo a construção prévia do On-set e do Off-set; enquanto a décima coluna mostra o tempo gasto pelo ESPRESSO para minimizar cada *benchmark*.

6.2.3.1 Efeito da Ordem na Comparação com ESPRESSO

Como o algoritmo usado para ordenação de variáveis dos BDDs é heurístico, não há garantia de que a melhor ordem seja encontrada. Além disso, não há garantia de que as ordens do BDD para a solução JOIN e a solução ESPRESSO sejam as mesmas. Além disso, a grande diferença em termos de contagem de nodos de BDD para ex10 e ex22 na Tabela 6.1 era estranha. Estávamos mais dispostos a investigar se a ordem variável estava desempenhando um papel. Para atestar que a escolha das ordens das variáveis não favoreceu a nossa abordagem, fizemos uma

Tabela 6.1 – Comparação entre o JOIN e ESPRESSO.

Bench-mark	***Entra-das***	**JOIN**				**ESPRESSO**			
		N	***C***	***L***	***T (s)***	***N***	***C***	***L***	***T (s)***
ex00	32	**1196**	756	7550	1,79	63669	123	1026	123,66
ex01	32	**848**	416	3687	1,79	4052	57	389	58,17
ex10	32	**1389**	862	9277	1,79	785222	174	1601	88,42
ex15	32	**978**	596	5921	1,60	586495	144	1273	76,80
ex20	16	**1932**	1436	15563	0,37	6670	553	5328	1,70
ex21	16	82	40	176	0,36	17	14	42	0,57
ex22	32	**2230**	1627	17887	1,61	8848750	384	3678	36,99
ex23	32	**3**	2	4	1,53	3	2	4	4,10
ex30	20	**263**	165	1226	0,55	582	57	360	4,98
ex40	16	**866**	541	5139	0,31	1483	246	2106	1,33
ex41	10	61	63	364	0,10	60	63	364	0,03
ex42	32	**2217**	1617	17814	1,79	7992139	374	3544	57,19
ex43	18	**1452**	966	9910	0,38	9747	367	3348	7,80
ex50	19	643	349	3264	0,58	108	28	178	2,53
ex57	24	254	125	1308	0,84	62	9	64	2,70
ex65	19	587	307	2785	0,52	96	29	150	2,11
ex69	16	240	130	1218	0,32	36	16	70	0,52
ex70	23	**345**	143	765	0,91	1871	49	220	13,69
ex71	23	**331**	108	1128	0,89	374	16	148	1,60
ex73	16	**1332**	851	8392	0,37	1936	160	1241	1,47
ex74	16	**2087**	1621	17960	0,35	6920	633	6219	1,96
ex75	16	**1136**	755	6362	0,40	3025	241	1626	2,11
ex76	16	**1147**	666	7227	0,38	2876	190	1662	0,94
ex77	16	**1147**	703	6040	0,38	4469	304	2342	3,14
ex78	16	**1911**	1423	14549	0,32	6367	569	5228	2,00
ex79	16	**1147**	703	6040	0,38	4577	308	2387	2,51

Fonte: Autor.

comparação computando os BDDs com as melhores ordens para JOIN e ESPRESSO. Neste experimento, as melhores ordens do JOIN foram usados para criar o BDD dos arquivos otimizados pelo ESPRESSO e vice-versa. Os resultados são apresentados na Tabela 6.2.

Na Tabela 6.2, as colunas 2 e 5, denotadas como *VR*, apresentam a redução

do número de variáveis de entrada. As colunas 3 e 6 (*Ordem Join*) correspondem ao número de nodos obtidos por cada método utilizando a melhor ordem encontrada (dentre as 100 aleatória discutida na seção 6.2.2) para o Join. Por último, as colunas 4 e 7 (Ordem Espresso) listam os resultados para o mesmo procedimento descrito anteriormente, porém utilizando a melhor ordem obtida para o ESPRESSO.

Tabela 6.2 – Comparação com as ordens trocadas.

Benchmark	**Join**			**ESPRESSO**		
	VR	***Ordem Join***	***Ordem Espresso***	***VR***	***Ordem Join***	***Ordem Espresso***
ex00	**11**	1196	1167	0	94459	63669
ex01	**12**	848	944	0	4695	4052
ex10	**10**	1389	1480	0	945490	785222
ex15	**11**	978	2300	0	784186	586495
ex20	0	1932	1976	0	7018	6670
ex21	**1**	82	85	0	23	17
ex22	**10**	2230	2241	0	10554376	8848750
ex23	**30**	3	2286	**30**	3	3
ex30	**2**	263	1756	0	608	582
ex40	0	866	580	0	1830	1483
ex41	0	61	60	0	61	60
ex42	**9**	2217	2310	0	10223966	7992139
ex43	**1**	1452	1749	0	9385	9747
ex50	**1**	643	1792	0	268	108
ex57	**6**	254	683	0	147	62
ex65	0	587	1660	**1**	133	96
ex69	**2**	240	373	**2**	40	36
ex70	**6**	345	559	0	1976	1871
ex71	**6**	331	1261	0	555	374
ex73	0	1332	1286	0	2032	1936
ex74	0	2087	2138	0	7130	6920
ex75	0	1136	1151	0	3320	3025
ex76	0	1147	1184	0	3090	2876
ex77	0	1147	1193	0	4681	4469
ex78	0	1911	1948	0	6527	6367
ex79	0	1147	1208	0	4681	4577

Fonte: Autor.

Como pode ser visto na Tabela 6.2, o único *benchmark* do ESPRESSO que se beneficiou do uso da ordem JOIN foi o benchmark ex43, mas o impacto é mínimo. No entanto, isso não foi suficiente para atingir o número de nodos obtidos pela abordagem JOIN. Quatro benchmarks do JOIN se beneficiaram do uso das ordens do ESPRESSO (ex00, ex40, ex41 e ex73). A conclusão deste experimento é que a grande diferença entre a contagem de nodos do BDD para este experimento não estava na ordem do BDD, mas em um motivo diferente. Ainda de acordo com a Tabela 6.2, pode-se verificar que o algoritmo JOIN é bastante eficaz na eliminação de entradas redundantes de funções incompletamente especificadas. Observando a coluna *Variáveis reduzidas*, é possível ver que o algoritmo JOIN elimina mais variáveis redundantes que ESPRESSO, o que explica a grande diferença em termos de contagem de nodos. O número máximo de nodos de um BDD é exponencial com o número de entradas, dado pela expressão (NEWTON; VERNA, 2019), portanto, a eliminação de variável redundante realizada pelo algoritmo JOIN tem um grande impacto na redução do tamanho do BDD. Por exemplo, um BDD com 10 variáveis a mais pode ser 1024 vezes maior, o que é consistente com nossos resultados.

6.3 Resultados Join2

A versão aprimorada Join2 é comparada com a sua versão inicial, Join1, e com os métodos *leaf-identifying compaction* , *restrict* e *constrain*. Onde são comparados o número total de nodos dos BDDs resultantes de cada método.

As ordens utilizadas nesta comparação foram obtidas através do CUDD (SOMENZI, 2018). Onde, para cada *benchmark*, foi construído o BDD com o reordenamento *sifting* (RUDELL, 1993) das variáveis, habilitado através comando **Cudd_AutodynEnable**(manager, CUDD_REORDER_SIFT). O CUDD foi utilizado apenas para o fim de gerar a ordem para cada *benchmark*, o método proposto nesta dissertação não faz o uso do CUDD.

A Tabela 6.3 apresenta os resultados obtidos por cada método, onde |F| é o tamanho do BDD que representa a função original e, |F'| é o tamanho do BDD cobertura obtido por cada método, ambos medidos por número de nodos. O método join é denotado como join1, o método *leaf-identifying compaction* é denotado como *LIC*, os demais métodos são denotados pelo seu nome original. As colunas 3, 4, 5, 6 e 7 apresentam os resultados, dos métodos avaliados, dados pela proporção de

|F'| em relação à |F|. Desta maneira, os valores tabelas descrevem o tamanho |F'| em relação a |F|. As células marcadas em negrito destacam os melhores resultados, menores coberturas, para cada *benchmark*.

Tabela 6.3 – Comparação Join2 com outros métodos.

Benchmark	**|F|**	**|F'| em relação a |F| (em %)**				
		Join2	**LIC**	**Restrict**	**Constrain**	
ex00	32260	7,06	**5,77**	7,12	7,06	7,06
ex01	31194	6,90	**5,57**	6,99	6,90	6,90
ex10	23489	8,74	**7,06**	8,81	8,74	8,74
ex15	30594	7,51	**6,08**	7,58	7,51	7,51
ex20	2953	67,19	**54,35**	76,16	66,14	67,19
ex21	2624	3,81	**2,93**	4,15	3,73	3,81
ex22	31905	7,09	**5,75**	7,17	7,09	7,09
ex23	29603	0,58	**0,01**	0,64	0,58	0,58
ex30	5171	39,74	**30,88**	46,28	39,74	39,74
ex40	2733	**14,38**	**14,38**	15,33	**14,38**	**14,38**
ex41	48	100,00	100,00	100,00	100,00	100,00
ex42	31690	7,05	**5,82**	7,15	7,05	7,05
ex43	4314	38,04	**27,91**	46,11	38,02	38,04
ex50	4555	36,27	**24,76**	47,03	36,14	36,27
ex57	4428	31,17	**23,62**	35,14	31,17	31,17
ex65	3419	54,23	**37,76**	63,94	53,90	54,23
ex69	1747	4,06	**2,23**	4,41	4,06	4,06
ex70	11936	5,80	**4,42**	5,97	5,80	5,80
ex71	2709	42,08	**34,07**	50,68	42,05	42,08
ex73	3292	47,66	**38,67**	50,64	47,17	47,66
ex74	2641	80,61	**67,74**	84,85	80,61	80,61
ex75	3123	37,69	**29,30**	42,56	37,08	37,69
ex76	1541	80,14	**61,52**	76,90	79,17	80,14
ex77	3833	32,22	**24,73**	39,60	31,88	32,22
ex78	3344	58,70	**48,80**	66,51	58,37	58,70
ex79	3833	32,22	**24,73**	39,60	31,88	32,22
Média	-	30,04	**23,56**	33,65	29,85	30,04

Fonte: Autor.

Pode-se observar que: 1) a melhoria Join2, discutida na seção 5.7, apresenta

uma significante melhoria ao método Join1, 2) o método Join2 apresentou os melhores resultados diante dos demais métodos, apenas para o *benchmark ex40* todos os método, exceto LIC, obtiveram o mesmo resultado. Por último, o *benchmark ex41* não apresentou redução do número de nodos, pois trata-se de uma função completamente especificada e desta forma a F' é igual à F.

A partir da Tabela 6.3, nota-se que o método Join1 gerou resultados iguais aos do método constrain. No entanto, este dois métodos utilizam abordagem diferentes, em que a função Booleana incompletamente especificada é descrita pelo seu On-set e Off-set para o método Join1, enquanto que para o método constrain função é descrita através do seu On-set e DC-set. Por outro lado, a melhoria Join2 apresenta melhores resultados, portanto é mais eficaz do que o método constrain.

A última linha Tabela 6.3 apresenta as médias dos resultados obtido por cada método. Nessa tabela, os valores tabelados correspondem a uma relação do do número de nodos da cobertura obtida, por cada método, em relação a número de nodos da função original. Desta forma, o melhor resultado é aquele que gera o menor valor diante os demais métodos. O método Join2 apresenta uma média de 23,56%, valor consideravelmente inferior comparado aos valores obtidos pelos métodos LIC, restrict, constrain que são em torno de 30%. Portanto, o método Join2 se mostrou mais eficiente, em termos de redução de nodos de BDD, em relação aos demais métodos. Os resultados para o *bechmark* ex48 não compõem as médias, pois não houve minimização para esse *bechmark*.

6.4 Contribuições deste capítulo

Neste capítulo foram apresentados os resultados comparativos do método proposto nesta dissertação com outros métodos presentes na literatura. Foram conduzidos dois principais experimentos, um comparando o método Join com o ESPRESSO em relação à redução da variáveis de suporte da função e, um segundo experimento que compara a melhoria (Join2) com outros métodos presentes na literatura para minimização de funções Booleanas incompletamente especificas.

O método Join2, proposta deste trabalho, apresentou resultados satisfatórios quando comparado aos demais métodos. Além disso, mostra-se uma boa alternativa para *benchmarks* descritos no formato adotado pelo concurso do IWLS 2020, onde as funções são descritas pelos seus On-set e Off-set, pois os demais métodos comparados

neste capítulo utilizam o padrão par de funções F e $F_{care-set}$ para representar uma função Booleana incompletamente especificada.

7 CONCLUSÃO E TRABALHOS FUTUROS

Neste trabalho foi apresentado um método para minimizar funções Booleanas incompletamente especificadas descritas pelo seu On-set e Off-set. Nossa abordagem de representação da função Booleana incompletamente especificada difere das abordagens dos métodos tradicionais (restric, constrain e *leaf-identifying compaction*), onde estes métodos fazem o uso de uma função F (o On-set) e o seu Care-set. Enquanto que representamos a função através do seu On-set e Off-set, padrão utilizado pelo concurso do IWLS de 2020.

Os resultados mostram um significativo ganho do Join2 diante os demais métodos comparados. O Join2 gerou, em média, coberturas com 23,56% do tamanho da funções originais (On-set), enquanto que os demais métodos geraram coberturas com tamanho em torno de 30% da mesma referência. Além disso, a abordagem de representar uma função Booleana incompletamente especificada através do par de BDDs On-set e Off-set mostrou-se mais eficaz, comparada a abordagem clássica (On-set e Care-set), considerando o número de nodos gerados e a memória utilizada. Isso é demonstrado na Tabela A.1, em apêndice.

Uma importante função de custo para *Index Generation Functions* é eliminar variáveis redundantes (NAGAYAMA; SASAO; BUTLER, 2020). Como o algoritmo JOIN tem o efeito colateral de eliminar variáveis, ele poderia ser mais sistematicamente adaptado para essa função de custo. Nesse sentido, (FEY; DRECHSLER, 2005) apresentou um algoritmo para encontrar ordens de BDD que minimizam o número de caminhos em um BDD, ao invés de encontrar uma ordem que minimiza o número de nodos.

Em termos de desempenho, acreditamos que a eficiência do nosso pacote de BDD atual pode ser melhorada, especialmente em termos de encontrar uma melhor ordem de variável para esta aplicação específica. Um aspecto que pode ser mais investigado é de certa forma sugerido por (FEY; DRECHSLER, 2005). O número de cubos em um SOP está de alguma forma relacionado aos caminhos que levam ao nodos terminal 1 em BDDs. Nesse sentido, minimizar uma expressão SOP como ESPRESSO tende a minimizar caminhos no BDD. Conforme demonstrado por (FEY; DRECHSLER, 2005), minimizar o número de caminhos pode vir à custa do aumento do número de nodos. Por esse motivo, acreditamos que a eliminação de variáveis redundantes em funções incompletamente especificadas como um objetivo

de projeto pode potencialmente levar a algoritmos de ordenação específicos para BDDs.

A abordagem vencedora (MIYASAKA; ZHANG; YU; YI; FUJITA, 2021) do concurso IWLS 2020 (RAI et al., 2020) utilizou BDDs como parte da abordagem, usando uma versão do algoritmo apresentado em (SHIPLE; HOJATI; SANGIOVANNI-VINCENTELLI; BRAYTON, 1994). Acreditamos que o algoritmo aqui apresentado possa gerar melhorias, mas isso ainda precisa ser testado e comprovado como trabalho futuro. Por último, este trabalho de mestrado gerou a publicação (PERALTA; NESPOLO; BUTZEN; KOLBERG; REIS, 2021).

7.1 Trabalhos Futuros

Acreditamos que este trabalho apresenta relevantes contribuições à área de síntese lógica. Com isto, alguns trabalhos futuros podem aprimorar nosso método para torná-lo uma ferramente ainda mais relevante.

Os resultados obtidos indicam que o método Join2 tem grande potencial de ser uma versão *safe*. Entretanto, ainda não temos uma comprovação matemática de tal afirmação. Devido isso, um trabalho futuro consiste em definir provas matemáticas que sustentem a afirmação do método Join2 ser *safe*.

Outro ponto queremos aprimorar o nosso método é sobre o ordenamento das variáveis de entrada, dado que a ordem pode impactar na qualidade do BDD, em termo de tamanho. A implementação do nosso método não faz uso de algum pacote de BDDs, como o CUDD, que provê funcionalidades de reordenamento. Portanto, um dos trabalhos futuros consiste em integrar o reordenamento ao nosso método, ou implementá-lo com o pacote CUDD.

REFERÊNCIAS

AKERS, S. B. Binary decision diagrams. **IEEE Computer Architecture Letters**, IEEE Computer Society, v. 27, n. 06, p. 509–516, 1978.

AMARÚ, L.; GAILLARDON, P.-E.; MICHELI, G. D. Majority-inverter graph: A novel data-structure and algorithms for efficient logic optimization. In: IEEE. 2014 51ST ACM/EDAC/IEEE DESIGN AUTOMATION CONFERENCE (DAC). **Proceedings...** [S.l.], 2014. p. 1–6.

AMARÚ, L.; GAILLARDON, P.-E.; MICHELI, G. D. Boolean logic optimization in majority-inverter graphs. In: PROCEEDINGS OF THE 52ND ANNUAL DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.: s.n.], 2015. p. 1–6.

AMARU, L.; GAILLARDON, P.-E.; MICHELI, G. D. Majority-inverter graph: A new paradigm for logic optimization. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 35, n. 5, p. 806–819, 2015.

ANDERSEN, H. R. An introduction to binary decision diagrams. **Lecture notes, available online, IT University of Copenhagen**, p. 5, 1997.

ASHAR, P.; MALIK, S. Fast functional simulation using branching programs. In: IEEE. PROCEEDINGS OF IEEE INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN (ICCAD). **Proceedings...** [S.l.], 1995. p. 408–412.

BERNASCONI, A.; CIRIANI, V. Index-resilient zero-suppressed bdds: Definition and operations. **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, ACM New York, NY, USA, v. 21, n. 4, p. 1–27, 2016.

BRACE, K. S.; RUDELL, R. L.; BRYANT, R. E. Efficient implementation of a bdd package. In: IEEE. 27TH ACM/IEEE DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 1990. p. 40–45.

BRAYTON, R.; MISHCHENKO, A. Abc: An academic industrial-strength verification tool. In: SPRINGER. INTERNATIONAL CONFERENCE ON COMPUTER AIDED VERIFICATION. **Proceedings...** [S.l.], 2010. p. 24–40.

BRAYTON, R. K. Factoring logic functions. **IBM Journal of research and development**, IBM, v. 31, n. 2, p. 187–198, 1987.

BRAYTON, R. K. et al. **Logic minimization algorithms for VLSI synthesis**. [S.l.]: Springer Science & Business Media, 1984.

BRAYTON, R. K. et al. Mis: A multiple-level logic optimization system. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 6, n. 6, p. 1062–1081, 1987.

BRYANT, R. E. Graph-based algorithms for boolean function manipulation. **Computers, IEEE Transactions on**, IEEE, v. 100, n. 8, p. 677–691, 1986.

BUTZEN, P. F. et al. Transistor network restructuring against nbti degradation. **Microelectronics Reliability**, Pergamon, v. 50, n. 9, p. 1298–1303, 2010.

BUTZEN, P. F. et al. Design of cmos logic gates with enhanced robustness against aging degradation. **Microelectronics Reliability**, Pergamon, v. 52, n. 9-10, p. 1822–1826, 2012.

BUTZEN, P. F. et al. Standby power consumption estimation by interacting leakage current mechanisms in nanoscaled cmos digital circuits. **Microelectronics Journal**, Elsevier, v. 41, n. 4, p. 247–255, 2010.

CALEGARO, V. **On the optimal minimization of special classes of Boolean functions**. Thesis (PhD) — Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil, 2016.

CERNY, E.; MARIN, M. A. An approach to unified methodology of combinational switching circuits. **IEEE Transactions on Computers**, IEEE Computer Society, v. 26, n. 08, p. 745–756, 1977.

CHANG, S.-C.; CHENG, D. I.; MAREK-SADOWSKA, M. Minimizing robdd size of incompletely specified multiple output functions. In: IEEE. PROCEEDINGS OF EUROPEAN DESIGN AND TEST CONFERENCE EDAC-ETC-EUROASIC. **Proceedings...** [S.l.], 1994. p. 620–624.

COUDERT, O.; BERTHET, C.; MADRE, J. C. Verification of synchronous sequential machines based on symbolic execution. In: SPRINGER. INTERNATIONAL CONFERENCE ON COMPUTER AIDED VERIFICATION. **Proceedings...** [S.l.], 1989. p. 365–373.

COUDERT, O.; MADRE, J. C. A unified framework for the formal verification of sequential circuits. In: SPRINGER. INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings...** [S.l.], 1990. p. 126–129.

CRAMA, Y.; HAMMER, P. L. **Boolean functions: Theory, algorithms, and applications**. [S.l.]: Cambridge University Press, 2011.

EBENDT, R.; FEY, G.; DRECHSLER, R. **Advanced BDD optimization**. [S.l.]: Springer Science & Business Media, 2005.

FEY, G.; DRECHSLER, R. Minimizing the number of paths in bdds: Theory and algorithm. **IEEE Transactions on Computer-Aided Design of Integrated circuits and systems**, IEEE, v. 25, n. 1, p. 4–11, 2005.

FROEHLICH, S.; GROSSE, D.; DRECHSLER, R. Error bounded exact bdd minimization in approximate computing. In: IEEE. 2017 IEEE 47TH INTERNATIONAL SYMPOSIUM ON MULTIPLE-VALUED LOGIC (ISMVL). **Proceedings...** [S.l.], 2017. p. 254–259.

GOMES, I. et al. Using only redundant modules with approximate logic to reduce drastically area overhead in tmr. In: IEEE. TEST SYMPOSIUM (LATS), 2015 16TH LATIN-AMERICAN. **Proceedings...** [S.l.], 2015. p. 1–6.

GOMES, I. A. et al. Methodology for achieving best trade-off of area and fault masking coverage in atmr. In: IEEE. TEST WORKSHOP-LATW, 2014 15TH LATIN AMERICAN. **Proceedings...** [S.l.], 2014. p. 1–6.

HONG, Y. et al. Safe bdd minimization using don't cares. In: PROCEEDINGS OF THE 34TH ANNUAL DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.: s.n.], 1997. p. 208–213.

JANSSEN, G. Design of a pointerless bdd package. In: INTERNATIONAL WORKSHOP ON LOGIC SYNTHESIS (IWLS). **Proceedings...** [S.l.: s.n.], 2001.

JÚNIOR, J. V. d. O. **Otimização de funções lógicas majoritárias utilizando programação linear inteira binária e quantificação de primitivas**. 87 p. Dissertation (Master) — Universidade Estadual Paulista (UNESP), Ilha Solteira, 2021.

JUNIOR, L. S. da R. et al. Fast disjoint transistor networks from bdds. In: ACM. PROCEEDINGS OF THE 19TH ANNUAL SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN. **Proceedings...** [S.l.], 2006. p. 137–142.

KATZ, R. H.; BORRIELLO, G. Contemporary logic design. Pearson Prentice Hall, 2005.

LEE, C.-Y. Representation of switching circuits by binary-decision programs. **The Bell System Technical Journal**, Nokia Bell Labs, v. 38, n. 4, p. 985–999, 1959.

LIAW, H.-T.; LIN, C.-S. On the obdd-representation of general boolean functions. **IEEE Transactions on computers**, IEEE Computer Society, v. 41, n. 06, p. 661–664, 1992.

LONG, D. E. The design of a cache-friendly bdd library. In: PROCEEDINGS OF THE 1998 IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN. **Proceedings...** [S.l.: s.n.], 1998. p. 639–645.

MACHADO, L. et al. Kl-cut based digital circuit remapping. In: IEEE. NORCHIP, 2012. **Proceedings...** [S.l.], 2012. p. 1–4.

MANO, M. M.; CILETTI, M. **Digital design: with an introduction to the Verilog HDL**. [S.l.]: Pearson, 2013.

MARRANGHELLO, F. S. et al. Factored forms for memristive material implication stateful logic. **IEEE Journal on Emerging and Selected Topics in Circuits and Systems**, IEEE, v. 5, n. 2, p. 267–278, 2015.

MARRANGHELLO, F. S. et al. Sop based logic synthesis for memristive imply stateful logic. In: IEEE. 2015 33RD IEEE INTERNATIONAL CONFERENCE ON COMPUTER DESIGN (ICCD). **Proceedings...** [S.l.], 2015. p. 228–235.

MARTINS, M. G.; RIBAS, R. P.; REIS, A. I. Functional composition: A new paradigm for performing logic synthesis. In: IEEE. QUALITY ELECTRONIC DESIGN (ISQED), 2012 13TH INTERNATIONAL SYMPOSIUM ON. **Proceedings...** [S.l.], 2012. p. 236–242.

MATOS, J. M. A. d. **Graph based algorithms to efficiently map VLSI circuits with simple cells**. Thesis (PhD) — Universidade Federal do Rio Grande do Sul, Porto Alegre, Brazil, 2018.

MATSUURA, M.; SASAO, T. Bdd representation for incompletely specified multiple-output logic functions and its applications to the design of lut cascades. **IEICE transactions on fundamentals of electronics, communications and computer sciences**, The Institute of Electronics, Information and Communication Engineers, v. 90, n. 12, p. 2762–2769, 2007.

MICHELI, G. D. **Synthesis and optimization of digital circuits**. [S.l.]: McGraw Hill, 1994.

MINATO, S. Zero-suppressed bdds for set manipulation in combinatorial problems. **30th ACM/IEEE Design Automation Conference**, p. 272–277, 1993.

MINATO, S.-i. Zero-suppressed bdds and their applications. **International Journal on Software Tools for Technology Transfer**, Springer, v. 3, n. 2, p. 156–170, 2001.

MINATO, S.-i. Techniques of bdd/zdd: brief history and recent activity. **IEICE TRANSACTIONS on Information and Systems**, The Institute of Electronics, Information and Communication Engineers, v. 96, n. 7, p. 1419–1429, 2013.

MINATO, S.-i. Power of enumeration—recent topics on bdd/zdd-based techniques for discrete structure manipulation. **IEICE transactions on information and systems**, The Institute of Electronics, Information and Communication Engineers, v. 100, n. 8, p. 1556–1562, 2017.

MINATO, S.-i.; ISHIURA, N.; YAJIMA, S. Shared binary decision diagram with attributed edges for efficient boolean function manipulation. In: IEEE. 27TH ACM/IEEE DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 1990. p. 52–57.

MISHCHENKO, A. An introduction to zero-suppressed binary decision diagrams. In: CITESEER. PROCEEDINGS OF THE 12TH SYMPOSIUM ON THE INTEGRATION OF SYMBOLIC COMPUTATION AND MECHANIZED REASONING. **Proceedings...** [S.l.], 2001. v. 8, p. 1–15.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. Dag-aware aig rewriting: A fresh look at combinational logic synthesis. In: IEEE. 2006 43RD ACM/IEEE DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 2006. p. 532–535.

MIYASAKA, Y. et al. "logic synthesis for generalization and learning addition. **In Proceedings of Design Automation and Test in Europe (DATE)**, 2021.

MOREIRA, M. et al. Semi-custom ncl design with commercial eda frameworks: Is it possible? In: INTERNATIONAL SYMPOSIUM ON ASYNCHRONOUS CIRCUITS AND SYSTEMS (ASYNC), POTSDAM. **Proceedings...** [S.l.: s.n.], 2014.

NAGAYAMA, S.; SASAO, T.; BUTLER, J. T. On optimum linear decomposition of symmetric index generation functions. In: IEEE. 2020 IEEE 50TH INTERNATIONAL SYMPOSIUM ON MULTIPLE-VALUED LOGIC (ISMVL). **Proceedings...** [S.l.], 2020. p. 130–136.

NEUTZLING, A. et al. Synthesis of threshold logic gates to nanoelectronics. In: IEEE. 2013 26TH SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). **Proceedings...** [S.l.], 2013. p. 1–6.

NEUTZLING, A. et al. Threshold logic synthesis based on cut pruning. In: IEEE. 2015 IEEE/ACM INTERNATIONAL CONFERENCE ON COMPUTER-AIDED DESIGN (ICCAD). **Proceedings...** [S.l.], 2015. p. 494–499.

NEWTON, J.; VERNA, D. A theoretical and numerical analysis of the worst-case size of reduced ordered binary decision diagrams. **ACM Transactions on Computational Logic (TOCL)**, ACM New York, NY, USA, v. 20, n. 1, p. 1–36, 2019.

PERALTA, R. D. et al. A method to join the on-set and off-set of an incompletely boolean function into a single bdd. In: IEEE. 2021 34TH SBC/SBMICRO/IEEE/ACM SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (SBCCI). **Proceedings...** [S.l.], 2021. p. 1–6.

POLI, R. E. et al. Unified theory to build cell-level transistor networks from bdds [logic synthesis]. In: IEEE. 16TH SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2003. SBCCI 2003. PROCEEDINGS.. **Proceedings...** [S.l.], 2003. p. 199–204.

POSSANI, V. N. et al. Parallel combinational equivalence checking. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 39, n. 10, p. 3081–3092, 2019.

RAI, S. et al. Logic synthesis meets machine learning: Trading exactness for generalization. **arXiv preprint arXiv:2012.02530**, 2020.

REIS, A. I. Covering strategies for library free technology mapping. In: IEEE. PROCEEDINGS. XII SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN (CAT. NO. PR00387). **Proceedings...** [S.l.], 1999. p. 180–183.

REIS, A. I.; DRECHSLER, R. **Advanced logic synthesis**. [S.l.]: Springer, 2018.

ROSA, L. et al. Scheduling policy costs on a java microcontroller. In: SPRINGER. ON THE MOVE TO MEANINGFUL INTERNET SYSTEMS 2003: OTM 2003 WORKSHOPS. **Proceedings...** [S.l.], 2003. p. 520–533.

ROSA, L. S. da et al. A comparative study of cmos gates with minimum transistor stacks. In: PROCEEDINGS OF THE 20TH ANNUAL CONFERENCE ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN. **Proceedings...** [S.l.: s.n.], 2007. p. 93–98.

ROSA, L. S. da et al. Switch level optimization of digital cmos gate networks. In: IEEE. 2009 10TH INTERNATIONAL SYMPOSIUM ON QUALITY ELECTRONIC DESIGN. **Proceedings...** [S.l.], 2009. p. 324–329.

RUDELL, R. Dynamic variable ordering for ordered binary decision diagrams. In: IEEE. PROCEEDINGS OF 1993 INTERNATIONAL CONFERENCE ON COMPUTER AIDED DESIGN (ICCAD). **Proceedings...** [S.l.], 1993. p. 42–47.

SASAO, T. Index generation functions: Tutorial. **Journal of Multiple-Valued Logic & Soft Computing**, Citeseer, v. 23, 2014.

SASAO, T.; MATSUURA, M. A method to decompose multiple-output logic functions. In: PROCEEDINGS OF THE 41ST ANNUAL DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.: s.n.], 2004. p. 428–433.

SAUERHOFF, M.; WEGENER, I. On the complexity of minimizing the obdd size for incompletely specified functions. **IEEE transactions on computer-aided design of integrated circuits and systems**, IEEE, v. 15, n. 11, p. 1435–1437, 1996.

SENTOVICH, E. M. et al. Sis: A system for sequential circuit synthesis. Citeseer, 1992.

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

SHIPLE, T. R. et al. Heuristic minimization of bdds using don't cares. In: IEEE. 31ST DESIGN AUTOMATION CONFERENCE. **Proceedings...** [S.l.], 1994. p. 225–231.

SILVA, D. N. da; REIS, A. I.; RIBAS, R. P. Cmos logic gate performance variability related to transistor network arrangements. **Microelectronics Reliability**, Pergamon, v. 49, n. 9-11, p. 977–981, 2009.

SOMENZI, F. CUDD: CU decision diagram package release 2.7.0. **University of Colorado at Boulder**, 2018.

TOGNI, J. et al. Automatic generation of digital cell libraries. In: IEEE. PROCEEDINGS. 15TH SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN. **Proceedings...** [S.l.], 2002. p. 265–270.

WAGNER, F.; REIS, A.; RIBAS, R. Fundamentos de circuitos digitais. **Sagra Luzzatto, Porto Alegre**, 2006.

YANG, C.; CIESIELSKI, M. Bds: A bdd-based logic optimization system. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 21, n. 7, p. 866–876, 2002.

YANG, S. **Logic synthesis and optimization benchmarks user guide: version 3.0**. [S.l.]: Citeseer, 1991.

YU, C.; CIESIELSKI, M.; MISHCHENKO, A. Fast algebraic rewriting based on and-inverter graphs. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 37, n. 9, p. 1907–1911, 2017.

APÊNDICE A — COMPARATIVO DE NÚMERO DE NODOS E MEMÓRIA

A Tabela A.1 apresenta uma comparação, número de nodos gerados e memória utilizada, para as abordagens de representar uma função Booleana incompletamente especificada através dos pares de BDDs On-set e Care-set (abordagem clásica) e, On-set e Off-set (abordagem proposta neste trabalho).

Na Tabela A.1, On, Off e Care se referem a On-set, Off-set e Care-set, nesta ordem. As colunas 2 e 3, marcadas com On & Off, apresentam os valores de número de nodos e memória utilizada pela abordagem par de BDDs On-set e Off-set. Por sua vez, as colunas 4 e 5, marcadas com On & Care, apresentam os valores de número de nodos e memória utilizada pela abordagem par de BDDs On-set e Care-set. As colunas 6 e 7 apresentam a diferença das duas abordagens. Esses valores foram obtidos com o CUDD.

Tabela A.1 – Comparação de número de nodos e uso de memória (em bytes) das abordagens.

File	**On & Off**		**On & Care**		**(On & Care) - (On & Off)**	
	Nodes	Memory	Nodes	Memory	Nodes	Memory
ex00	140201	49404008	143431	50779048	3230	1375040
ex01	140266	49404008	143438	50779048	3172	1375040
ex10	140305	49436744	142748	50386216	2443	949472
ex15	140269	49436744	142311	50680840	2042	1244096
ex20	34042	16097640	35277	17272104	1235	1174464
ex21	33492	16064904	35226	17272104	1734	1207200
ex22	140147	49404008	143444	50746312	3297	1342304
ex23	140195	49404008	143459	50746312	3264	1342304
ex30	63100	22641224	65098	23840200	1998	1198976
ex40	34041	16097640	35296	17272104	1255	1174464
ex41	2095	10864776	2095	10971176	0	106400
ex42	140098	49404008	143426	50746312	3328	1342304
ex43	48945	18992968	50705	20282024	1760	1289056
ex50	55909	20824264	56761	21691784	852	867520
ex57	89191	41179752	89384	41441640	193	261888
ex65	55971	20715816	57678	21593576	1707	877760
ex69	33931	16142664	34640	16797416	709	654752
ex70	82667	38405160	82918	39616520	251	1211360
ex71	82671	38405160	82708	38536104	37	130944
ex73	33718	16130376	35048	17615944	1330	1485568
ex74	33548	16064904	35258	17239368	1710	1174464
ex75	33781	16175400	34623	17403080	842	1227680
ex76	33706	16175400	34176	16650088	470	474688
ex77	33706	16175400	34771	17795976	1065	1620576
ex78	33864	16097640	35404	17648680	1540	1551040
ex79	33706	16175400	34771	17795976	1065	1620576

Fonte: Autor.