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

RENATO DONIZETE PERALTA

Satisfiability-Based Covering of AIGs Using KL-cuts

Thesis presented in partial fulfillment
of the requirements for the degree of
Doctor of Microeletronics

Advisor: Prof. Dr. André Inácio Reis

Porto Alegre
July 2024

CIP — CATALOGING-IN-PUBLICATION

Peralta, Renato Donizete

Satisfiability-Based Covering of AIGs Using KL-cuts / Renato Donizete Peralta. – Porto Alegre: PGMICRO da UFRGS, 2024.

120 f.: il.

Thesis (Ph.D.) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Microeletrônica, Porto Alegre, BR–RS, 2024. Advisor: André Inácio Reis.

1. And-inverter graph. 2. Electronic design automation. 3. Cuts in AIGs. 4. KL-cuts. 5. Logic synthesis. 6. Very-large-scale integration. 7. Satisfiability. I. Reis, André Inácio. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. André Bulhões Mendes
Vice-Reitora: Prof[a]. Patricia Pranke
Pró-Reitor de Pós-Graduação: Prof[a]. Júlio Otávio Jardim Barcellos
Diretora do Instituto de Informática: Prof[a]. Carla Maria Dal Sasso Freitas
Coordenador do PGMICRO: Prof. Cláudio Radtke
Bibliotecário-chefe do Instituto de Informática: Alexsander Borges Ribeiro

ABSTRACT

The rapid advancement of Very-large-scale integration (VLSI) silicon integration has revolutionized the electronics industry, enabling the integration of billions of transistors into a single integrated circuit. This complexity necessitates the use of Electronic Design Automation (EDA) tools for VLSI design. These tools automate various tasks, with the design process typically divided into logic synthesis and physical design stages. Logic synthesis involves capturing the initial logic description and generating a netlist of interconnected cell instances, while physical design focuses on placing and routing these instances to create a circuit layout ready for fabrication. Over the past two decades, logic synthesis has heavily relied on the And-Inverter-Graph (AIG) data structure for its scalability and result quality. This thesis introduces a contribution to logic synthesis based on AIGs, specifically focusing on computing AIG covers with *KL*-cuts, where both the number of inputs (K) and outputs (L) are controlled. While previous work has proposed the use of *KL*-cuts in synthesis flow, efficient algorithms for covering AIGs with *KL*-cuts have been lacking. This thesis addresses this issue by presenting an algorithm to optimize this task. Additionally, a detailed review of AIG cut types is provided. Existing synthesis methodologies often lack efficient algorithms for AIG covering with *KL*-cuts, necessitating the development of optimized solutions. In this thesis are proposed two novel contributions to logic synthesis using AIGs, focusing on *KL*-cut computation to enhance synthesis efficiency. The first contribution introduces a novel method for expanding single-output cuts into *KL*-cuts. The second contribution presents a satisfiability-based approach for generating AIG covers using *KL*-cuts. Experimental results validate the effectiveness of these contributions. The proposed cut expansion method achieved a 1.25x speedup compared to the MFFW method. Furthermore, the proposed approach successfully generated *KL*-cut covers with a reduction: 49.01% compared to K-cuts from ABC and 7.51% compared to multi-output covers derived from post-processing of ABC results.
Keywords: And-inverter graph. electronic design automation. cuts in AIGs. KL-cuts. logic synthesis. very-large-scale integration. satisfiability.

Cobertura de AIGs Baseada em Satisfatibilidade Usando KL-cuts

RESUMO

O rápido avanço da integração de silício em Integração em Larga Escala (VLSI) revolucionou a indústria de eletrônicos, permitindo a integração de bilhões de transistores em um único circuito integrado. Essa complexidade exige o uso de ferramentas de Automação de Projeto Eletrônico (EDA) para o design de VLSI. Essas ferramentas automatizam várias tarefas, com o processo de design tipicamente dividido em etapas de síntese lógica e design físico. A síntese lógica envolve capturar a descrição lógica inicial e gerar uma lista de interconexões entre instâncias de células, enquanto o design físico se concentra na colocação e roteamento dessas instâncias para criar um layout de circuito pronto para a fabricação. Nas últimas duas décadas, a síntese lógica tem dependido fortemente da estrutura de dados And-Inverter-Graph (AIG) devido à sua escalabilidade e qualidade dos resultados. Esta tese apresenta uma contribuição para a síntese lógica baseada em AIGs, especificamente focando no cálculo de coberturas de AIG com *KL*-cuts, onde tanto o número de entradas (K) quanto de saídas (L) são controlados. Embora trabalhos anteriores tenham proposto o uso de *KL*-cuts em fluxos de síntese, algoritmos eficientes para cobrir AIGs com *KL*-cuts têm sido escassos. Esta tese aborda essa questão apresentando um algoritmo para otimizar essa tarefa. Além disso, é forne uma revisão detalhada dos tipos de cortes em AIGs. Metodologias de síntese existentes frequentemente carecem de algoritmos eficientes para cobertura de AIG com *KL*-cuts, exigindo o desenvolvimento de soluções otimizadas. Nesta tese, são propostas duas contribuições inovadoras para a síntese lógica usando AIGs, com foco no cálculo de *KL*-cuts para melhorar a eficiência da síntese. A primeira contribuição introduz um método inovador para expandir cortes de saída única em *KL*-cuts. A segunda contribuição apresenta uma abordagem baseada em satisfatibilidade para gerar coberturas de AIG usando *KL*-cuts. Resultados experimentais validam a eficácia dessas contribuições. O método de expansão de cortes proposto alcançou uma aceleração de 1,25 vezes em comparação com o método MFFW. Além disso, a abordagem proposta gerou com sucesso coberturas *KL*-cut com uma redução de 49,01% em comparação com K-cuts do ABC e 7,51% em comparação com coberturas multi-saídas derivadas do pós-processamento dos resultados do ABC.

Palavras-chave: Grafo and-inverter, automação de projeto eletrônico, cortes em AIGs, KL-cuts, síntese lógica, integração em larga escala, satisfatibilidade.

LIST OF ABBREVIATIONS AND ACRONYMS

AIG	And-Inverter Graph
ASIC	Application-Specific Integrated Circuit
BDD	Binary Decision Diagram
EDA	Electronic Design Automation
FFC	Fanout-Free Cone
FFW	Fanout-Free Window
FPGA	Field-Programmable Gate Array
LUT	Look-Up Table
MFFC	Maximum Fanout-Free Cone
MFFW	Maximum Fanout-Free Window
PI	Primary Input
PO	Primary Output
SAT	Satisfiability
TT	Truth Table
SOP	Sum of Products
TFI	Transitive Fanin
TFO	Transitive Fanout
VLSI	Very-large-scale Integration
IC	Integrated Circuit
MtM	More than Ten Million Benchmark

LIST OF FIGURES

Figure 3.1 ROBDD representing a Full Adder with inputs a, b, and cin........................21
Figure 3.2 Example of an AIG representing a two-input full adder...............................22
Figure 3.3 Transitive fanin and transitive fanout of node 12. 23
Figure 3.4 AIG elements. 23
Figure 3.5 Rewriting method in AIG. 27
Figure 3.6 Example of a K-cut 28
Figure 3.7 Example of a valid cover with non-superposed cuts. 28
Figure 3.8 Example of an invalid cover with non-superposed cuts. 29
Figure 3.9 Example of a valid cover with superposed cuts. 29
Figure 3.10 Example of an invalid cover with superposed cuts. 30
Figure 3.11 Example of a cell-based mapping. 31
Figure 3.12 Example of FPGA-based mapping. 32

Figure 4.1 K-cut example. 36
Figure 4.2 Side-outputs of the cut_1 and cut_3. The blue arrow illustrates the *K*-cut's output and the red arrow illustrates the side edges. 37
Figure 4.3 *KL*-cut in AIG. 38
Figure 4.4 Example of *L*-cut crossing the TFO of K-cuts 39
Figure 4.5 *KL*-cut cover in AIG. The blue arrows represent the *KL*-cuts outputs. 40
Figure 4.6 Example of a not valid *KL*-cut for $L = 2$. 41
Figure 4.7 Example of when a K-cut is a *KL*-cut with $L = 1$. 41
Figure 4.8 Circuit partitioned into *KL*-cuts. 42
Figure 4.9 Points of view of a *1L*-cut. 43
Figure 4.10 1x1 (m x n) Window for node *N*. 44
Figure 4.11 Fanin cone of the node 10. 45
Figure 4.12 Fanout cone of the node 8. 46
Figure 4.13 All Fanout-Free Cones of node 5. 47
Figure 4.14 MFFC. 49
Figure 4.15 MFFW. 50

Figure 5.1 Backcut example. 62
Figure 5.2 Martinello's algorithm. 64
Figure 5.3 Martinello's method result. 65
Figure 5.4 $addInsts()$ function illustration. 67
Figure 5.5 Machado's algorithm. 68
Figure 5.6 Machado's algorithm result. 68
Figure 5.7 MFFW expansion. 70
Figure 5.8 Example of an AIG covered using only K-cut, with K equals 3. 72
Figure 5.9 Another cover using K-cuts with k equals 3. 73
Figure 5.10 Cover example using *KL*-cuts with *K* equals 3 e *L* equals 2. 73

Figure 6.1 K-cut enumeration example. 76
Figure 6.2 K-cut enumeration example. 77
Figure 6.3 Example of cuts *K* generated with K=3 for a given AIG. 79
Figure 6.4 Result of the $markSupport()$ method at a piece of the circuit. 82
Figure 6.5 Intersection of the fanins support list. 83
Figure 6.6 Identifying the outputs of the cut C_1. 85
Figure 6.7 AIG example for SAT formulation. 87

Figure 6.8 CNF encoding for only one cut for node 18. 89
Figure 6.9 BDD for Cuts With Unit Costs. 92
Figure 6.10 BDD for solutions with up to 2 Cuts. 93
Figure 6.11 ROBDD for solutions with up to 2 Cuts. 93
Figure 6.12 BDD for a problem involving four cuts with different costs. Each terminal node shows the total cost of each combination of cuts. 94
Figure 6.13 BDD for a problem involving four cuts, showing combinations of cuts that meet the maximum predefined cost of 2. 95
Figure 6.14 ROBDD for the problem with different costs for the cuts. 95
Figure 6.15 Flowchart of approach to generating minimum cover using SAT. 97

Figure 7.1 Comparison of Runtimes for Machado, Our Method, and MFFW (Log Scale). 101
Figure 7.2 Comparison of Cover Size Using Single-Output Cuts (from ABC) vs. Multi-Output Cuts (with *KL-cuts*) 104
Figure 7.3 Cover Size Comparison of Results of Figure 7.2 after the Post-Process. 105
Figure 7.4 Runtime using only DAG cuts. 107
Figure 7.5 Runtime using DAG and TREE cuts. 109

LIST OF TABLES

Table 3.1 Truth table of a full adder. 18

Table 5.1 All k-cuts for the AIG of Figure 3.2........ 55
Table 5.2 K-cuts and Backcuts of Figure 5.1. 63
Table 5.3 Summary of the coverage results using k-cuts and kl-cuts. 74

Table 6.1 Cortes K de três entradas, e seus custos, do AIG da Figura 6.3. 81
Table 6.2 Result of the Methods mark_support() and forward() in the AIG 6.3. 84
Table 6.3 KL-cuts used for SAT formulation 87

Table 7.1 Runtime comparison in milliseconds. 100
Table 7.2 Cover without optimization for EPFL circuits 111
Table 7.3 Clause size and runtime........ 112

LIST OF ALGORITHMS

6.1 KL-Cuts enumeration Algorithm. 80
6.2 Prepare KL-Cuts Algorithm. 80
6.3 Mark Support Algorithm. 81
6.4 Forward KL-Cuts Algorithm. 83
6.5 Define KL-Cuts outputs Algorithm. 84

A.1 KL-Cuts enumeration Algorithm (Complete). 120

CONTENTS

1 INTRODUCTION **13**
2 MOTIVATION FOR KL CUTS **15**
2.1 About This Chapter **15**
2.2 *KL*-cuts and multi-output LUTs **15**
2.3 *KL*-cuts and multi-output cells in a library **15**
2.4 *KL*-cuts as a partitioning method **16**
2.5 Contributions of this chapter **16**
3 BASIC CONCEPTS **17**
3.1 About this chapter **17**
3.2 Boolean functions and their representation **17**
3.2.1 Historical evolution of Boolean functions representations 17
3.2.2 Truth tables 18
3.2.3 Sum of Products 19
3.2.4 Binary Decision Diagrams 20
3.2.5 And-Inverter Graphs 21
3.3 Physical Implementation Technologies **24**
3.3.1 ASICs 24
3.3.1.1 Cell Library 24
3.3.1.2 Library-based Design specificities 25
3.3.2 FPGAs 25
3.3.2.1 Look-up Tables - LUTs 25
3.3.2.2 LUT-based Design specificities 26
3.4 AIGs-based synthesis **26**
3.4.1 Cuts in AIGs 27
3.4.2 Covers of an AIG 28
3.5 Technology Mapping **30**
3.5.1 Mapping to cell-based ASICs 31
3.5.2 Mapping to FPGAs 31
3.5.3 Cost Functions 32
3.5.3.1 Area 32
3.5.3.2 Delay 33
3.5.3.3 Power 33
3.5.3.4 Design Constraints 33
3.6 The EPFL Benchmarks **33**
3.7 Contributions of this chapter **34**
4 A TAXONOMY OF AIG CUTS **35**
4.1 About this Chapter **35**
4.2 Structural Differences Among Different Types of Cuts **35**
4.2.1 *K*-cut 35
4.2.2 *KL*-cut 37
4.2.2.1 *1L*-cut 42
4.2.3 Windowing 43
4.2.3.1 Fanin Cone 44
4.2.3.2 Fanout Cone 45
4.2.3.3 Fanout-Free Cone 46
4.2.3.4 Maximum Fanout-Free Cone 48
4.2.3.5 Maximum Fanout-Free Window 50
4.2.4 Structural differences between MFFW and *KL*-cuts 51

4.3 Non-structural types of cuts ... **51**
4.3.1 Factor Cuts ... 51
4.3.2 Priority Cuts ... 52
4.4 Windowing vs. Covering ... **52**
4.4.1 Windowing with MFFW ... 52
4.4.2 Covering with *KL*-cuts ... 53
4.4.3 Windowing with *KL*-cuts ... 53
4.4.4 Covering with *KL*-cuts and 1L-cuts ... 53
4.5 Contributions of this chapter ... **53**
5 LITERATURE REVIEW ... **54**
5.1 About this chapter ... **54**
5.2 Cuts in AIGs - a historical overview ... **54**
5.2.1 *K*-cuts ... 54
5.2.1.1 Enumerating all K cuts ... 54
5.2.1.2 Enumerating factor cuts ... 56
5.2.1.2.1 Complete Cut Factorization ... 56
5.2.1.2.1.1 Tree Cuts - (Local Cuts) ... 56
5.2.1.2.1.2 Reduced Cuts - (Global Cuts) ... 57
5.2.1.2.2 Partial Cut Factorization ... 57
5.2.1.2.2.1 Leaf-dag Cuts - (Local Cuts) ... 57
5.2.1.2.2.2 Dag Cuts - (Global Cuts) ... 58
5.2.1.3 Enumerating priority cuts ... 58
5.2.1.4 Cut signature ... 59
5.2.2 *KL*-cuts ... 60
5.2.2.1 Martinello's Enumeration Method ... 60
5.2.2.2 Machado's Enumeration Method ... 65
5.2.2.2.1 IWLS23 Best Paper Method ... 69
5.2.2.3 Optimization Without Coverage ... 71
5.3 Flow Based on Logic Calculation and Signal Distribution ... **71**
5.3.1 *KL*-cuts For Logic Calculation ... 71
5.3.2 1L-cuts For Signal Distribution ... 71
5.4 Comparative Summary Between Covering With *K*-cuts and *KL*-cuts ... **72**
5.5 Contributions of this chapter ... **74**
6 PROPOSED METHOD ... **75**
6.1 About This Chapter ... **75**
6.1.1 *K*-cut Enumeration ... 75
6.1.2 Cut Signature ... 77
6.2 Expanding *K*-cuts to *KL*-cuts ... **78**
6.3 Covering an AIG with *KL*-cuts ... **86**
6.3.1 Satisfiability formulation for a valid cover ... 86
6.3.1.1 Rule 1 ... 88
6.3.1.2 Rule 2 ... 88
6.3.1.3 Rule 3 ... 89
6.3.1.4 Rule 4 ... 90
6.3.2 Satisfiability formulation for a minimum cover ... 90
6.3.3 Approach Using BDDs for Cost Constraint Solutions ... 91
6.4 Our Method Overview ... **96**
6.5 Contributions of This Chapter ... **97**
7 RESULTS AND DISCUSSIONS ... **98**
7.1 About this chapter ... **98**
7.2 *KL*-cut enumeration performance ... **98**

7.3 Covering with *KL*-cuts....102
7.3.1 Exact Cover Size....103
7.3.1.1 Cover Size....103
7.3.1.2 Cover Runtime....106
7.3.2 Cover Without Optimization....110
7.4 Contributions of this chapter....113
8 CONCLUSION....114
REFERENCES....116
APPENDIX A — ALGORITHM WITH OPTMIZATIONS....120

1 INTRODUCTION

The evolution of VLSI silicon integration has led the electronics industry to a present situation where the number of devices in a single integrated circuit can be of the order of billions of transistors. The complexity of large VLSI design must rely then on computer-aided design tools commonly known as Electronic Design Automation (EDA) tools.

Due to the complexity of design flows, the research on EDA focuses generally on small specialized topics. Design flows are generally very complex encompassing a variety of tasks performed by a variety of tools. In a very naive way, the design of integrated circuits can be divided into logic synthesis and physical design. The logic synthesis happens in the front end of the design flow and the goal of logic synthesis can be stated as to capture the initial logic description and generate a netlist of interconnected cell instances that implement the desired logic. The physical design happens in the back end of the design flow and the goal of physical design can be stated as using the netlist generated by the logic synthesis to place and route the cell instances to produce a circuit layout that is ready for fabrication. It must be said that current commercial design flows can estimate placement and floorplanning in the early steps of the design flow, still during logic synthesis, and consider the effect introduced by placement and routing parasitics.

The field of logic synthesis has in the last two decades relied on the AIG data structure. This is done mostly due to scalability reasons while preserving result quality. The logic synthesis using AIGs is based on the concept of AIG cuts. There are several types of cuts in AIGs, so this will be discussed in more detail later in this thesis. The most used type of cut in AIG is the *K*-cut, which can be roughly defined as a sub-graph with K inputs rooted in an output node. *K*-cuts were initially introduced to map Field-Programmable Gate Array (FPGA) circuits composed of K-input look-up tables. However, *K*-cuts later become an important part of AIG-based logic synthesis not necessarily tied to FPGAs.

This thesis presents a contribution to logic synthesis based on AIGs. The contribution is focused on computing AIG covers with *KL*-cuts, where not only the number of inputs K but also the number of outputs L is controlled in the *KL*-cut subgraph. *KL*-cuts were introduced by Martinello (MARTINELLO et al., 2010) and later Prof. André Reis (REIS; MATOS, 2018) proposed an approach to use *KL*-cuts in a synthesis flow based on logic computation and signal distribution. However, the proposal of the logic computation and signal distribution design flow did not provide an efficient algorithm to cover an AIG

with KL cuts. In this thesis, we provide an efficient algorithm for this task, as described in chapter 6. We propose: (1) an efficient method to enumerate *KL*-cuts, (2) a method to generate CNF formulas to obtain the cover of an AIG with *KL*-cuts using satisfiability (SAT), and (3) a BDD-based algorithm to generate maximum cost clauses for problems that will be solved with SAT.

Chapter 3 introduces the basic concepts used in this thesis. For people with familiarity with logic synthesis, this chapter can be partially or completely skipped, with the reader searching only for the necessary concepts from the chapter.

Chapter 4 presents a complete overview of different types of AIG cuts in the literature. This chapter is a contribution in itself, as a comparative review with the level of completeness and detail presented here is not available in previous works.

Chapter 5 discusses prior works in a chronological way. It is a bibliographical review that presents the historical evolution of AIG-based logic synthesis. The focus is to present the evolution of AIG based logic synthesis.

Chapter 6 introduces the contributions of this thesis, where the proposed methods and the steps taken to obtain a cover using *KL*-cuts are presented.

Chapter 7 presents the obtained results and discusses their significance. One point that is important to highlight is that the algorithm for enumerating *KL*-cuts is significantly more efficient in terms of execution time when compared to the method of (MACHADO et al., 2012), and slightly more efficient on average than the method of (TANG et al., 2023). Another point is the reduction in the number of cuts needed to cover circuits when using multi-output cuts, such as *KL*-cuts.

Finally, Chapter 8 concludes this work, discussing the contributions and limitations. Future works are also outlined, in the context of a design flow based on logic computation and signal distribution.

2 MOTIVATION FOR KL CUTS

2.1 About This Chapter

This chapter presents the motivation for *KL*-cuts. *KL*-cuts can be useful on at least three different contexts. Notice that the use in different contexts also happens for the more common *K*-cuts that can be used in independent logic optimization, FPGA mapping, as well as cell-based mapping for ASICs. The different contexts for using *KL*-cuts are discussed in the next subsections.

2.2 *KL*-cuts and multi-output LUTs

The first and more obvious application of *KL*-cuts is for FPGAs with multiple output LUTs. In a very similar way that *K*-cuts can be used to map a circuit for FPGAs based on single output LUTs, the mapping for FPGAs with LUTs with L outputs can be done through a cover based on KL cuts. One interesting research question would be to map a circuit for an FPGA with single output 4-LUTs and compare it with a mapping to an FPGA with 4-2-cuts. In the best case, the number of LUts would be divided by 2. However, not all the pairs of 4-cuts can be packed into 4-2-cuts as the inputs may differ. Additionally, it is possible to compare the dedicated *KL*-cut covering method proposed in this thesis against a method that covers a circuit with *K*-cuts and then packs the compatible cuts into the same KL-Lut when this is possible. This is one type of experiment investigated in this thesis.

2.3 *KL*-cuts and multi-output cells in a library

A second application of *KL*-cuts is for ASIC mapping targeting a library with (some) multiple-output cells. A recent publication on technology mapping using multi-output library cells (CALVINO; MICHELI, 2023) discusses *KL*-cuts as an alternative to the method they introduced. However, they use cut signatures as they target only full-adder cells where the outputs depend exactly on the same set of inputs, which results in equal cut signatures. The use of *KL*-cuts could help to match cells with outputs depending on a slightly different set of inputs. However, we will not investigate this type of approach

in detail, as for ASIC synthesis of ASICs the different outputs will probably require different sizes and the logical match of a cell would not be sufficient to produce a match that is adequate to the timing requirements.

2.4 *KL*-cuts as a partitioning method

A third application of *KL*-cuts is in a method to bring physical awareness starting at technology-independent logic synthesis (REIS; MATOS, 2018). The authors propose a design flow based on (local) logic computation and signal distribution. The local logic computation is done inside *KL*-cuts, meaning that the *KL*-cuts will be used to pack independent portions of the circuit that will be placed locally. In this way, the routing among cells inside a same *KL*-cut is local, using short wires and low metal levels. This means that the *KL*-cuts in (REIS; MATOS, 2018) can be seen as partitions composed of several cells inside the *KL*-cuts as opposed against a *KL*-cut that is a match to a single library cell as in (CALVINO; MICHELI, 2023). The *KL*-cuts can be viewed as physical partitions. As a consequence, the routing among different *KL*-cuts tends to be global, using long wires and high metal levels. The authors (REIS; MATOS, 2018) proposed the design flow with these characteristics, but a scalable algorithm to compute *KL*-cuts and cover a circuit was not provided. The original algorithm for *KL*-cuts (MARTINELLO et al., 2010) also lacks scalability. This way, a scalable *KL*-cut covering algorithm is missing.

2.5 Contributions of this chapter

This chapter presented the motivation for developing a *KL*-cut computation and covering algorithm. The use of *KL*-cuts can be applied in three distinct contexts proposed by different authors. At the same time an efficient scalable algorithm for *KL*-cuts has not yet been proposed.

Finally, we reckon that this chapter presents the motivation in a way that is suitable for those already knowledgeable in logic synthesis. We hope that it motivates those that are not yet fully knowledgeable in logic synthesis to keep reading the thesis so that the motivation is further clarified when the underlying concepts are presented further in the text. For those initiating in logic synthesis, perhaps re-reading this chapter after finishing the thesis will bring a renewed appreciation of the contributions.

3 BASIC CONCEPTS

3.1 About this chapter

This chapter will present the basic concepts essential for understanding this work. This section provides the necessary basis for subsequent discussions and analyses. This chapter can be potentially skipped by those knowledgeable about the field of logic synthesis.

3.2 Boolean functions and their representation

The Boolean domain is defined by the set $B = \{0, 1\}$. In other words, this set contains only the elements 1 or 0; and a variable defined in the Boolean domain can only assume the values 0 or 1. For instance, if a variable x is a Boolean variable, it can only assume the values 0 or 1. The value 1 is sometimes interpreted (named, referred or labeled) as the true value. Similarly, the value 0 is called the false value.

A (single input) Boolean function is a mapping from a Boolean set of size *N*, called a domain, to another set of the size of 1, also a Boolean domain called an image, this is defined as $F : B^n \rightarrow B$. Therefore, a Boolean function associates a combination of input values, of size *N*, with an output of 0 or 1. Normally the term Boolean function refers to a single output Boolean function. However, most functions of practical interest are multiple output functions. A multiple output Boolean function is a mapping from a Boolean set of size *N*, called a domain, to another set of size *M*, also a Boolean domain called an image, this is defined as $F : B^n \rightarrow B^m$.

There are different ways to represent Boolean functions, each with its characteristics and limitations. Next, some of the main representations of Boolean functions that were widely used during the evolution of the circuit design area will be presented.

3.2.1 Historical evolution of Boolean functions representations

As predicted by Gordon Moore in 1965 (MOORE, 1965), the density of transistors in integrated circuits, and consequently, the processing capacity of computers, is expected to double approximately every two years.

As the size of circuits grew over time, it was increasingly necessary to use more optimized ways to represent Boolean functions. This motivated the creation of new data structures to represent Boolean functions in order to keep up with the growth of Boolean functions.

EDA started with the design of the first computers and modern data structures were not yet known, as the first computers did not have the necessary resources to support them. So the evolution of EDA data structures goes in hand with the evolution of computers, as larger computers need and at the same time support smarter and more efficient data structures to design the next generation of computers. In the following sections, we will present some of the most popular ways to represent Boolean functions.

3.2.2 Truth tables

The truth table (TT) was the first way to represent Boolean functions in the very early days of EDA, from the late 50´s to 70´s. In this representation, each combination of input values is associated with a corresponding output value (MICHELI, 1994). Consequently, it is considered canonical, possessing a unique representation under a fixed order of the input variables. Table 3.1 shows a truth table for a full adder, which is a 3-input, 2-output Boolean function denoted as $B^3 \rightarrow B^2$ function.

Table 3.1 – Truth table of a full adder.

A	**B**	$\mathbf{C_{in}}$	**Sum**	$\mathbf{C_{out}}$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

Each row of the truth table represents a combination of input values to the function, which has one or more associated outputs. For example, row 2 of the truth table

3.1 has the input combination (A=0, B=0, C_{in}=1), which produces the outputs $Sum = 1, C_{out} = 0$.

It is important to notice that, as the number of inputs grows, the size of the truth table increases exponentially with the number n of inputs. The number of different input combinations is 2^n, where n is the number of inputs to the function (CRAMA; HAMMER, 2011). For instance, the full adder has a truth table with eight combinations for the three inputs A, B, and C_{in}, as shown in Table 3.1.

3.2.3 Sum of Products

A Sum of Products (SOP) was one of the most popular Boolean representations from the early 70´s to 90´s, is an equation described by the sum of all input combinations (Products) that produce the value 1 as the output of the function. SOP can be considered one of the simplest ways to describe a Boolean function using equations because an SOP is composed only of the sum of the function's minterms, which is a two-level format (WAGNER; REIS; RIBAS, 2006). For example, the function represented by the truth table 3.1 has the following SOPs:

- $Sum = (\overline{A} \cdot \overline{B} \cdot C_{in}) + (\overline{A} \cdot B \cdot \overline{C_{in}}) + (A \cdot \overline{B} \cdot \overline{C_{in}}) + (A \cdot B \cdot C_{in})$;
- $C_{out} = (\overline{A} \cdot B \cdot C_{in}) + (A \cdot B \cdot C_{in}) + (A \cdot \overline{B} \cdot C_{in}) + (A \cdot B \cdot \overline{C_{in}})$;

This form of Boolean function representation has been and continues to be used by tools such as Espresso (BRAYTON et al., 1982), a heuristic two-level logic minimizer. The SOP representation has also been and is still used by tools like MIS (BRAYTON et al., 1987), a multilevel logic synthesis and minimization system responsible for popularizing multilevel synthesis based on the manipulation of factored forms, and SIS (SENTOVICH et al., 1992), a tool for synthesis and optimization of sequential circuits, responsible for adding sequential circuit handling to the MIS algorithms.

Similarly to the TT, the SOP is also considered canonical, thus providing a unique representation of the Boolean function (WAGNER; REIS; RIBAS, 2006). However, as with the TT, the size of the SOP can increase significantly with the number of inputs grows. Due to these limitations, new approaches have been proposed, where methods based on data structures have become prominent, as will be discussed in the following sections.

3.2.4 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) were an important tool in the field of formal verification and optimization of logic circuits, from the late 80´s to the mid 2000´s. The BDD structure is based on the Shannon Expansion (SHANNON, 1949), each node in the BDD has a generic decision variable dv that is used to decide between two other nodes that represent sub-functions where the decision variable dv has a fixed value $dv = 0$ (negative cofactor sub-function) or $dv = 1$ (positive cofactor sub-function). Technically, the two sub-functions that are pointed by a BDD node with decision variable dv do not depend on the variable dv, as the variable dv has already been decided to $dv = 0$ (for the negative cofactor sub-function) or $dv = 1$ (for the positive cofactor sub-function).

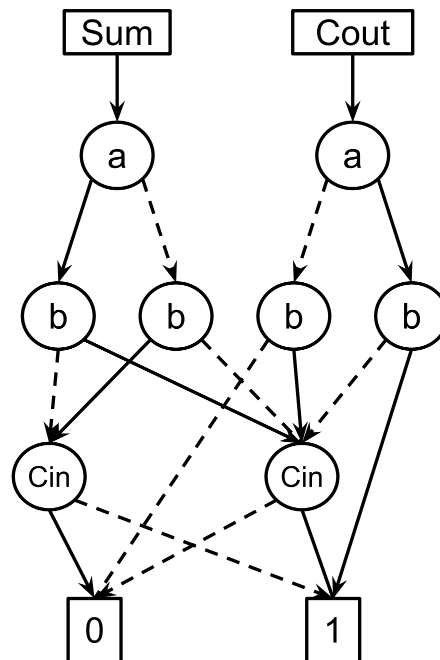
BDDs can be traced to the work by (LEE, 1959) and (AKERS, 1978), but they become widely used after the seminal work of (BRYANT, 1986), which introduced Reduced Ordered BDDs (ROBDDs), their canonical properties and algorithms to manipulate functions. ROBDDs constitute an optimized variant of BDDs, proposed to enhance efficiency in the representation and manipulation of Boolean functions. By eliminating duplicate nodes (i.e. nodes representing the same logic function) and imposing restrictions on the ordering of BDD variables, ROBDDs provide a compact and effective representation.

The defining characteristics of ROBDDs are *i*) variable ordering and *ii*) node irredundancy. Variable ordering means that a fixed order is imposed on the function's input variables, where all paths from the root node to a terminal node follow the same sequence of variables, even when some variables are not present in a specific path. Node irredundancy means that ROBDDs do not have *i*) nodes that do not make decisions and *ii*) redundant sub-graphs representing the same Boolean function. Also, (BRYANT, 1986) has shown that due to these characteristics, ROBDDs are canonical and unique structures for equivalent functions.

Figure 3.1 illustrates a ROBDD example of the Boolean function depicted in Table 3.1. The decision variables for each level are represented on the left, as the ROBDD is ordered and all the nodes in a given level share the same decision variable. The values inside the nodes are keys used in a hash table to guarantee that nodes are not duplicated, as it is usual in the strong canonical form of ROBDDs (BRACE; RUDELL; BRYANT, 1990). The solid lines represent the positive cofactors $dv = 1$, while the dotted lines represent the negative cofactors $dv = 0$ for each node. As shown, every path in the BDD leads to one of the terminal nodes, determining the output value generated by the Boolean

function.

Figure 3.1 – ROBDD representing a Full Adder with inputs a, b, and cin.



The tools SIS (SENTOVICH et al., 1992), VIS (BRAYTON et al., 1996), and MV-SIS (CHAI et al., 2003) use BDDs in their respective methodologies. SIS employs BDDs to represent and manipulate sequential circuits, facilitating optimization and formal property verification. Although VIS does not use BDDs as its primary form of representation, BDDs are employed in processes such as Equivalence checking and Simulation for advanced formal verification. On the other hand, MVSIS deals with multivalued logic and uses methods based on BDDs.

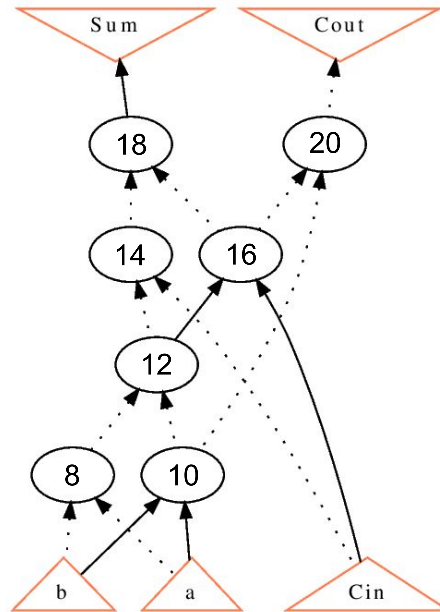
3.2.5 And-Inverter Graphs

In the beginning of the 2000´s, it became clear that ROBDDs would not scale to support EDA aiming large circuits, and AIGs began to gain traction as the main data structure in logic synthesis through the use of AIGs in ABC. This way, AIGs became an essential structural representation in the domain of computer engineering and computer science. Their efficiency and scalability make them a valuable tool for representing complex Boolean functions.

An AIG is a directed acyclic graph composed of primary inputs (PIs), primary outputs (POs), and internal nodes that represent the AND2 function. Variable inversions

are represented in the arcs, that may be labeled as inverters. Figure 3.2 shows the AIG of a full adder. In this AIG example, the PIs are a, b, and Cin, the outputs are Sum and $Cout$, and the nodes numbered six to twelve are AND2 nodes.

Figure 3.2 – Example of an AIG representing a two-input full adder.

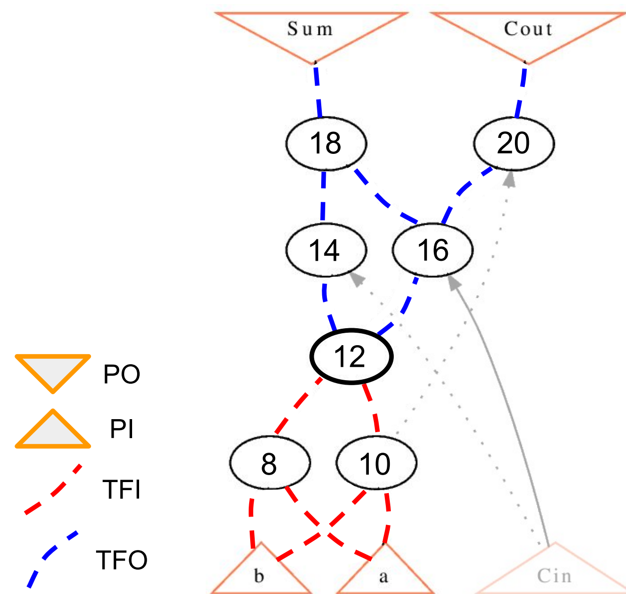


Source: Adapted from (CIESIELSKI et al., 2019).

The dotted and solid lines represent the complemented (inverted) and uncomplemented (non-inverted) edges, respectively, used to describe whether the function accessed by the edge is inverted or not. Internal nodes are 2-input ANDs that have exactly two inputs, called *fanins* of the node. The output of a node nd_i that is input to another node nd_j is called *fanout* of nd_i (MISHCHENKO; BRAYTON, 2005).

An AIG node nd_i is commonly associated with an integer identificator i. Given two nodes nd_i and nd_j, where $i < j$, there is a topological partial order between node nd_i and node nd_j, such that node nd_i has precedence over (i.e. appears before) node nd_j. As a consequence of the partial order, when there is a path between the nodes nd_i and nd_j, the node nd_i is in *fanin* transitive (TFI) of node nd_j. Conversely, node nd_j is in the transitive *fanout* (TFO) of node nd_i (MISHCHENKO; CHATTERJEE; BRAYTON, 2006a). These elements are illustrated in Figure 3.3. For instance, nodes nd_{14}, nd_{16}, nd_{18} and nd_{20} are in the transitive fanout of node nd_{12}. Additionally, nd_8 and nd_{10} are in the transitive fanin of node nd_{12}. Finally, notice that nodes nd_8 and nd_{10} are not in the transitive fanout or transitive fanin of each other because there is no directed path from one to the other, as they are parallel nodes in the graph.

Figure 3.3 – Transitive fanin and transitive fanout of node 12.



Even though the AIG is a graph representation, this data structure can be manipulated as an array. Figure 3.4 illustrates the AIG from Figure 3.2 represented by an array. In this representation approach, each node is indexed by an even integer number, and stored at the position $(Aig_index)/2$. When a node is accessed through an odd Aig index, this means that the inverted function from the node is required instead, and the correspondent node is also accessible through $((Aig_index)/2)$. For example, node 12 is stored at position 6, and its fanins are nodes 8 and 10 (in Figure 3.2), however, both fanins are complemented, thus in the integer array AIG representation these nodes are indexed by the values 9 (inverted 8) and 11 (inverted 10). Finally, position 0 is reserved for the constant zero (and its complement 1), to distinguish it from other PIs, its fanins are marked with "-".

Figure 3.4 – AIG elements.

Array index	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Aig index	**0**	**2**	**4**	**6**	**8**	**10**	**12**	**14**	**16**	**18**	**20**
Fanin_1	-	2	4	6	5	4	11	13	12	17	17
Fanin_2	-	2	4	6	3	2	9	7	6	15	11

In the remainder of this work, some examples of AIGs can have nodes with odd indexes. This is justified because the ABC tool (BRAYTON; MISHCHENKO, 2010)

used to create the examples uses odd indexes for the nodes to generate the figures, even if internally the nodes have even integer identifiers.

Currently, AIGs are one of the main data structures used by the ABC tool (BRAYTON; MISHCHENKO, 2010), one of the most important *open-source* logic synthesis tools. The ABC tool provides state-of-the-art combinational and sequential circuit synthesis algorithms.

3.3 Physical Implementation Technologies

The ultimate goal of EDA tools is to produce a real-world circuit design. To produce such a real-life design, a target implementation technology should be adopted. The main technologies are ASICs and FPGAs, which are discussed in the following subsections.

3.3.1 ASICs

Application-Specific Integrated Circuit (ASIC) is a type of integrated circuit (IC) that is designed for a specific purpose or application, as opposed to general-purpose ICs like microprocessors or memory chips. ASICs are custom-built to perform a particular function or set of functions, making them highly efficient and specialized for the intended task.

The key advantages of ASICs include high performance, low power consumption, and optimized functionality for the targeted application. However, the development of ASICs can be costly and time-consuming, and once manufactured, they are typically not reprogrammable for different tasks. This contrasts with general-purpose processors, which can be reprogrammed to perform a wide range of tasks.

3.3.1.1 Cell Library

A cell library, in the context of integrated circuit (IC) design, is a collection of predefined and characterized building blocks or cells that can be used to create custom integrated circuits. Each cell within the library represents a fundamental functional unit, such as logic gates, flip-flops, multiplexers, and other digital or analog components. These cells are designed, characterized, and tested to meet specific performance and electrical

criteria.

The purpose of a cell library is to provide designers with a set of standardized building blocks that they can use to assemble custom ICs quickly and efficiently. Instead of designing every component from scratch, designers can select cells from the library that match their requirements and integrate them into their overall chip design. This modular approach helps streamline the design process, reduce development time, and ensure consistency in the manufactured ICs.

3.3.1.2 Library-based Design specificities

Each cell from a library implements a function with a given cost. That means that specific sub-functions, implemented by different cells, will have different costs.

3.3.2 FPGAs

A FPGA is a configurable hardware integrated circuit that can be programmed or configured by the user or designer after manufacturing. Unlike the ASICs, which are custom-designed for a specific application and have a fixed functionality, FPGAs offer flexibility and reconfigurability to implement different designs. This makes FPGAs well-suited for prototyping, development, and applications where adaptability to changing requirements is crucial.

FPGAs have gained significant attention in modern computing, owing to their versatility, rapid deployment capabilities, and cost-effectiveness. With short time-to-market and field programmability, FPGAs have become pivotal in diverse computing environments, from data centers to edge computing applications (FAN; WU, 2023; CONG; DING, 1993).

The FPGA architecture comprises programmable logic blocks, interconnections, and I/O pads. The Look-Up Table (LUT)-based architecture dominates the existing programmable chip industry. LUTs are discussed in the next subsection.

3.3.2.1 Look-up Tables - LUTs

The fundamental programmable logic element is the K-input LUT (K-LUT), a versatile element that can implement any combinational logic function up to K inputs. The specific implementation of the LUT architecture is determined by the target size (maxi-

mum number of inputs) to be supported (CHEN; CONG, 2004; CHEN; CONG, 2001). Notice that the LUT architecture does not necessarily need to be based on a table structure, and the specific implementations are generally proprietary. For instance, Vranesic (ZILIC; VRANESIC, 1996) proposes LUT architectures derived from BDDs.

3.3.2.2 LUT-based Design specificities

Each LUT in a given FPGA architecture implements several different functions with the same given cost (the cost of the LUT, not the configured function). That means that different sub-functions, implemented by identical LUTs, will have equal costs.

3.4 AIGs-based synthesis

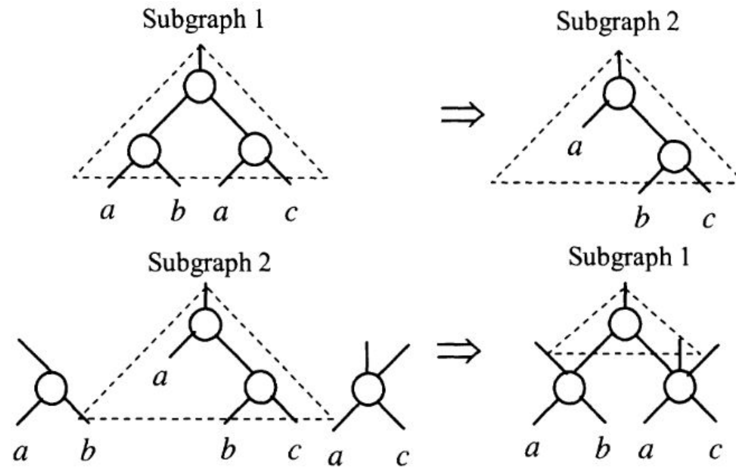
Modern logic synthesis is strongly based on AIG rewriting. The concept of AIG rewriting can be explained with the help of Figure 3.5. The figure presents two examples of AIG rewriting, one in the upper part and one in the bottom part. The AIG rewriting depends on the context of the complete AIG.

In the upper part of the figure, it is illustrated that a subgraph 1 (with equation $(ab) * (ac)$) can be substituted by a subgraph 2 (with equation $(a) * (bc)$) in an AIG such that the number of nodes in the subgraph is reduced from 3 nodes to 2 nodes. This way, there is a reduction of one node in the circuit implementation.

In the bottom part of the figure, the dependence of the complete AIG context is illustrated. In this example, subgraph 2 (with equation $(a) * (bc)$) is substituted by subgraph 1 (with equation $(ab) * (ac)$) in an AIG such that the total number of nodes in the AIG is reduced from 4 nodes to 3 nodes. This way, there is a reduction of one node in the circuit implementation. Notice that this optimization is viable due to the context of the complete AIG, where the nodes with equations ab and ac previously exist in the AIG and do not need to be created.

Several different approaches in the literature are based in AIG rewriting. Examples of AIG based synthesis include (PAN; LIN, 1998), (CONG; DING, 1999), and (NETO et al., 2022).

Figure 3.5 – Rewriting method in AIG.



Source: (MISHCHENKO; CHATTERJEE; BRAYTON, 2006a).

3.4.1 Cuts in AIGs

A cut in a Boolean network, denoted as C, is an important concept utilized in logic synthesis and optimization. It consists of a pair $\{r, \{l_0, \ldots, l_i\}\}$, comprising a root node and a set of nodes called leaves. For each path from a PI to the root of C, it must traverse at least one of its leaves (MISHCHENKO; CHATTERJEE; BRAYTON, 2006a; MISHCHENKO; BRAYTON; CHATTERJEE, 2008).

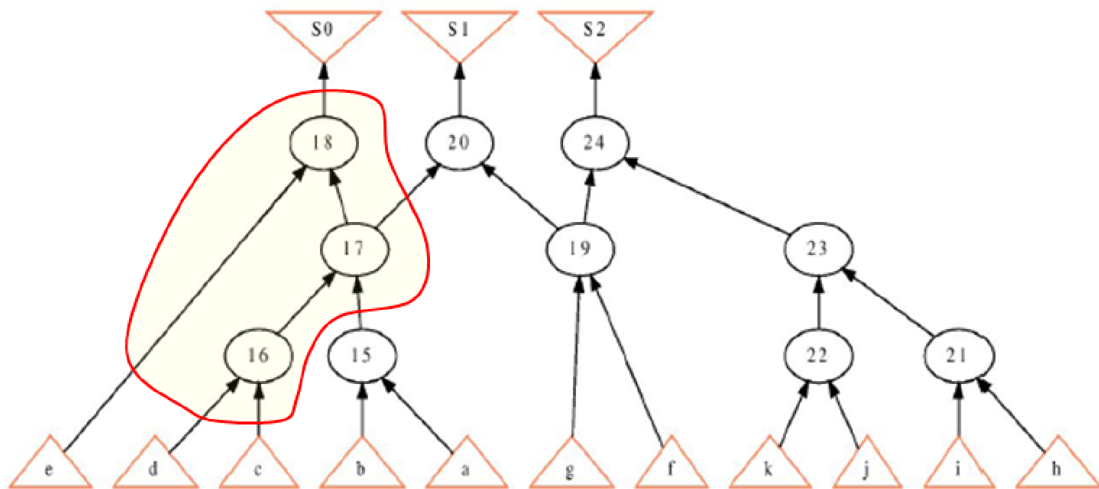
The size of a cut C consists of the number of leaf nodes that compose it and is denoted as $|C|$, and a trivial node cut consists only of the node itself, called self-cut (MISHCHENKO; CHATTERJEE; BRAYTON, 2006a; MISHCHENKO; BRAYTON; CHATTERJEE, 2008). The procedure to enumerate the cuts with K or less inputs will be shown in section 5.2.

Figure 3.6 provides an example of a 4-cut in an AIG. Notice that only one 4-cut is shown for simplicity, but several different cuts may be possible for each node of this AIG. The section 5.2.1.1 will provide a complete explanation of how to generate all *K*-cuts for a given AIG. Figure 3.6, shows a 4-cut with leaves the nodes $\{e, d, c, 15\}$ and root the node 18. The meaning of this 4-cut is that the Boolean function rooted in node 18 can be expressed as a function of the variables representing the Boolean function rooted in nodes $\{e, d, c, 15\}$. Indeed, $n_{18} = e \cdot d \cdot c \cdot n_{15}$.

It is possible to say that the 4-cut $\{e, d, c, 15\}$ rooted on node 18 covers nodes 16, 17 and 18. This has to be understood as the fact that the inputs $\{e, d, c, 15\}$ are sufficient

to compute the function of node 18 and nodes 16 and 17 are not necessary to this end. However, nodes 16 and 17 are still necessary to compute the function in node 20 as node 17 is an input for node 20.

Figure 3.6 – Example of a K-cut



3.4.2 Covers of an AIG

A cover of an AIG is a set of cuts such that the set of Boolean functions corresponding to each cut jointly implements the Boolean function of the complete AIG. In the following, we will see some examples of valid and invalid covers of an AIG.

Figure 3.7 – Example of a valid cover with non-superposed cuts.

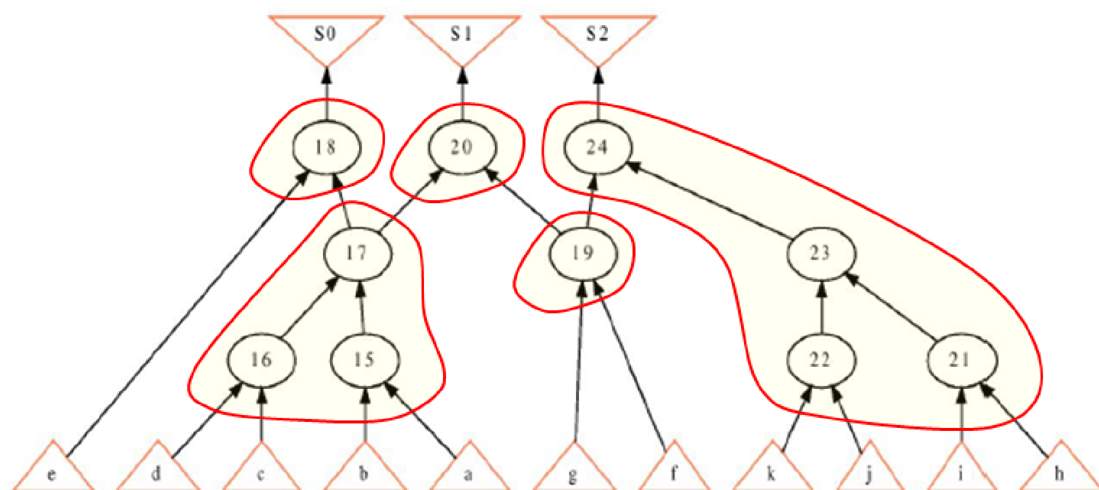


Figure 3.7 presents a valid cover of an AIG with non-superposed cuts. The cover is composed of five distinct cuts and there is no superposition of cuts. That means that each AIG node is covered by only one cut. There is no AIG node belonging to more

than one cut. Notice that the Boolean functions for nodes 15, 16, 21, 22, and 23 were not explicitly implemented as the nodes are internal to cuts. This is not a problem as the nodes are not used as inputs to any other cuts.

Figure 3.8 – Example of an invalid cover with non-superposed cuts.

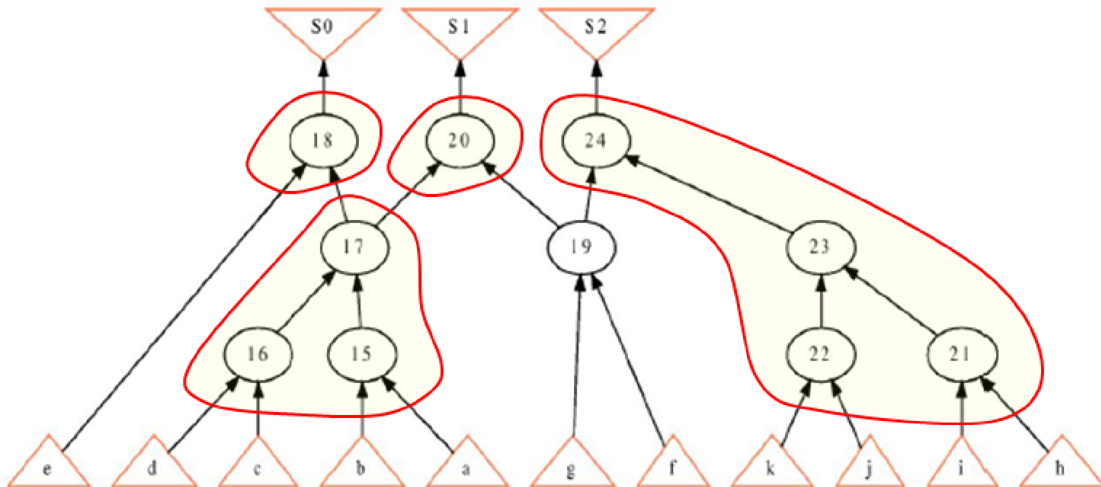


Figure 3.8 illustrates an invalid cover of an AIG with non-superposed cuts. The cover is composed of four distinct cuts and there is no superposition of cuts. That means that each AIG node is covered at most by only one cut. There is no AIG node belonging to more than one cut. However, node 19 is not covered by any cut and it is an input for nodes 20 and 24. This way, this is not a valid cover as the Boolean function corresponding to node 19 is necessary and it was not implemented.

Figure 3.9 – Example of a valid cover with superposed cuts.

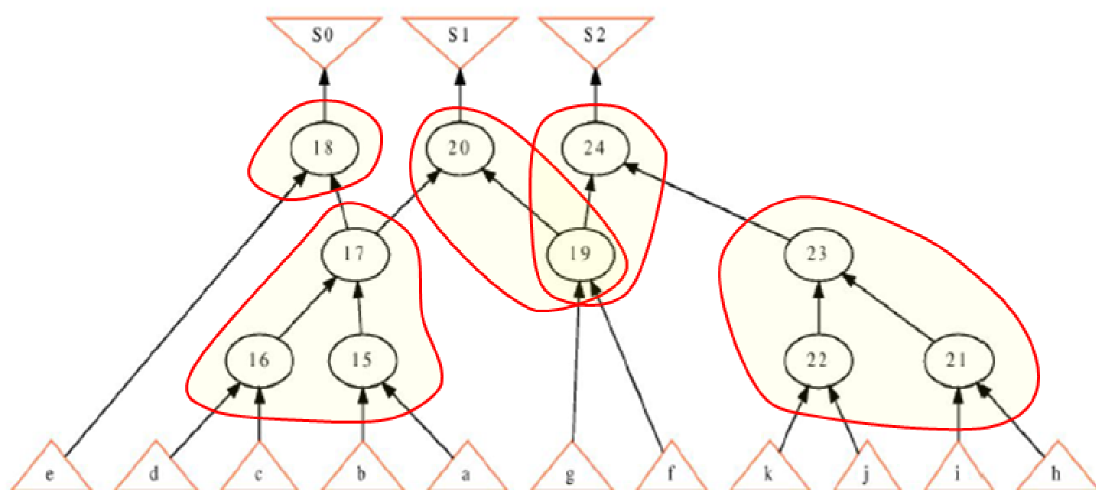


Figure 3.9 shows a valid cover of an AIG with superposed cuts. The cover is composed of five distinct cuts and there is a superposition of cuts between the cuts rooted in nodes 20 and 24. This happens because node 19 is covered by the two cuts. Consequently,

there is an AIG node belonging to more than one cut, resulting in the superposition of cuts. Notice that the Boolean functions for nodes 15, 16, 19, 21 and 22 were not explicitly implemented as the nodes are internal to cuts. This is not a problem as the nodes are not used as inputs to any other cuts.

Figure 3.10 – Example of an invalid cover with superposed cuts.

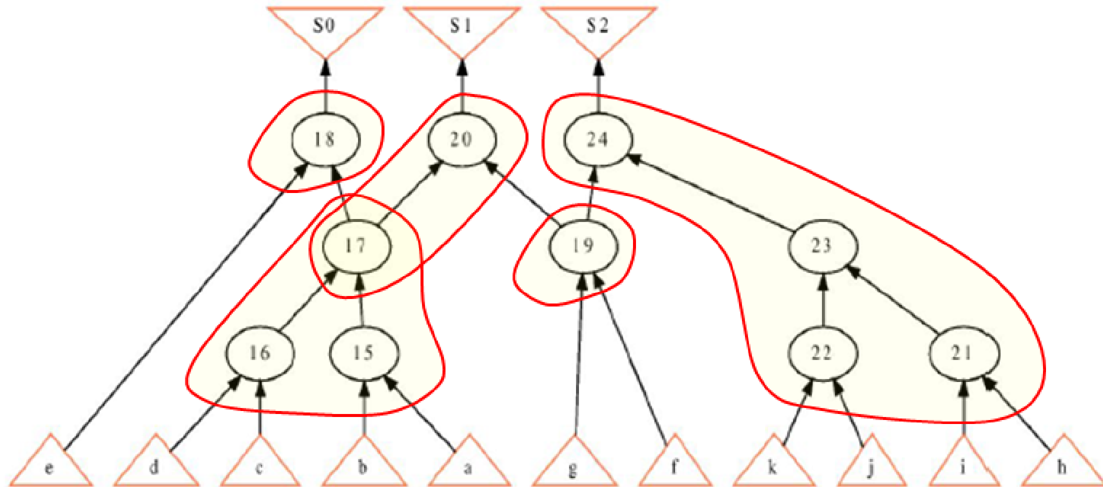


Figure 3.10 presents an invalid cover of an AIG with superposed cuts. The cover is composed of five distinct cuts and there is a superposition of cuts between the cuts rooted in nodes 17 and 20. This happens because node 17 is covered by the two cuts. Consequently, there is an AIG node belonging to more than one cut, resulting in the superposition of cuts. However, the cut rooted in node 20 has the nodes 15 and 16 as inputs and those do not have any cut rooted on them. This way, this is not a valid cover as the Boolean function corresponding to nodes 15 and 16 are necessary and they were not implemented.

3.5 Technology Mapping

Technology mapping is an important step in the digital circuit integrated design flow. This step is responsible for transforming the description of a technology-independent circuit to a set of primitives defined in a given technology. For cell-based designs, the primitives are cells from the target library. In the case of field programmable gate arrays (FPGAs), the primitives are normally K-input LUTs. Before mapping, the Boolean network is represented as a specific type of graph, normally called a subject graph. AIGs are commonly used as subject graphs.

Similarly to a cover of an AIG, a mapping can also be defined as a set of cuts,

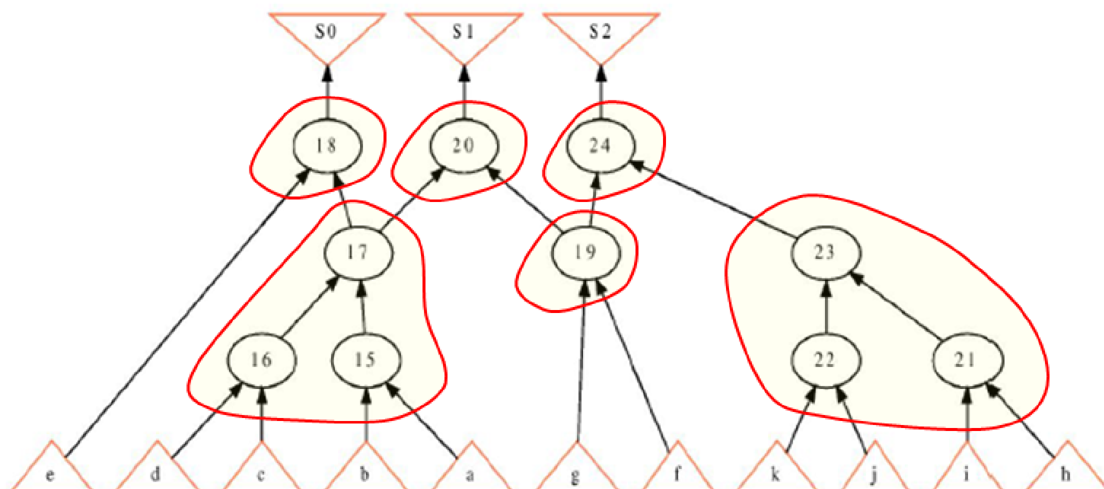
but with more restrictive properties. A mapping of an AIG is a set of cuts such that *i*) the set of Boolean functions corresponding to each cut jointly implements the Boolean function of the complete AIG; *ii*) the Boolean function of each cut can be implemented by a physical element of the target technology; and, *iii*) an overall technology-dependent cost is optimized. In the next sub-sections we will discuss mapping to cell-based designs and FPGAs.

3.5.1 Mapping to cell-based ASICs

Mapping to cell-based ASICs is the process of transforming a high-level electronic design description into a configuration that can be implemented on a specific type of ASIC known as a cell-based ASIC. This process involves selecting appropriate standard cells from the library to implement different parts of the design.

Figure 3.11 illustrates an ASIC mapping considering a cell library with AND cells up to 4 inputs (i.e. a five input AND is not present in the library). Cell-based mapping tends to avoid superposition as the duplication of nodes due to superposition leads to duplication of logic, with an adverse impact on area costs.

Figure 3.11 – Example of a cell-based mapping.



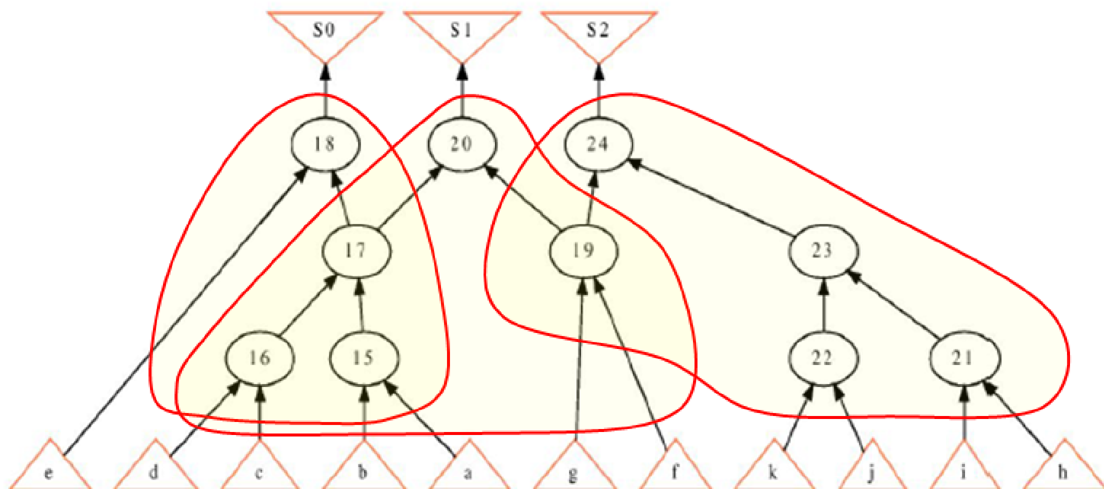
3.5.2 Mapping to FPGAs

In the case of FPGA as target technologies, the physical elements used to implement the circuit are LUTs. Figure 3.12 shows an FPGA mapping considering LUTs up to 6 inputs (i.e. every Boolean function is feasible with a single 6-input LUT). FPGA-based

mapping tends to extensively use superposition as the duplication of nodes due to superposition does not imply in duplication of logic. Indeed, the more logic is packed in a sigle LUT, the better the usage of the LUT.

In the example of Figure 3.12, four nodes are duplicated to allow the implementation of the circuit with only 3 LUTs, one for each output. Three 6-input LUTs are used even if output S0 has only five inputs. The outputs S1 and S2 make full use of a 6-input LUT each.

Figure 3.12 – Example of FPGA-based mapping.



3.5.3 Cost Functions

During technology mapping an overall technology-dependent cost is optimized. In the following, we discuss first-order estimators for cost functions.

3.5.3.1 Area

Area in technology-independent optimization is estimated by counting the number of AIG nodes. The number of AIG nodes, when used for relative comparisons, has good correspondence with the final area of cell-based and FPGA implementations. Area in cell-based implementations is obtained by adding the areas of individual cell instances. Area in FPGA-based implementations is obtained by adding the number of used LUTs.

3.5.3.2 Delay

The first-order estimator for delay in a circuit is the number of elements in the longest path. In the case of FPGA designs, unit delay can be used as all elements are LUTs of the same type. In the case of cell-based designs, the individual delay of cells from the library must be considered, as different types of cells have different associated delays.

3.5.3.3 Power

Power is highly correlated to area, so that area is sometimes used as a first-order estimator. When different Vt options are available for cells, Vt assignment can be made after place and route.

3.5.3.4 Design Constraints

Real-life design optimizations perform a trade-off among different cost functions. To express design intent and direct these trade-offs in the right direction, design constraints are used. Design constraints are commonly expressed in SDC format.

3.6 The EPFL Benchmarks

The EPFL Combinational Benchmark Suite (AMARÚ; GAILLARDON; MICHELI, 2015) is a set of benchmarks proposed to evaluate the performance of academic and commercial logic optimization and synthesis tools. It is composed of 23 combinational circuits divided into three main categories: Arithmetic, Random/Control, and *More than ten Million gates* (MtM). Each circuit is distributed in different formats, including Verilog, VHDL, BLIF, and AIGER.

This set of *benchmarks* has been widely used to validate several works, due to providing a nice variety of circuits. Furthermore, are provided a set of circuits with more than one million AND2 nodes when represented in AIG form. It is a relatively expressive size, making these *benchmarks* a good choice to test the scalability of the proposed methods.

3.7 Contributions of this chapter

In this chapter, we have presented the main definitions necessary for an understanding of this work. The main concepts concerning AIG based logic synthesis were presented.

4 A TAXONOMY OF AIG CUTS

4.1 About this Chapter

This chapter presents a taxonomy for AIG cuts. We discuss the differences among various types of cuts in AIGs, in order to establish a language that allows the reader to understand different types of cuts, their inner workings and the motivation for their use. This discussion is made from two distinct points of view.

Section 4.2 discusses cuts from a structural standpoint. For instance, questions addressed in section 4.2 include the structural differences between *K*-cuts and *KL*-cuts. Are they always structurally different? Can a given cut be classified as a *K*-cut and as a *KL*-cut at the same time? The structural discussion does not consider the algorithms used to generate the cuts.

4.2 Structural Differences Among Different Types of Cuts

Beginning with the structural properties of cuts, with a focus on understanding their definitions and characteristics, this approach enables us to differentiate the various types of cuts and alternative techniques employed for identifying subregions within the graph.

4.2.1 *K*-cut

Let $K \in \mathbb{N}^*$, a cut is considered k-feasible if its size is less than K ($|C| \leq K$), defining what is known as a k-cut. Thererfore, a cut is said to be K-feasible if its number of inputs is up to K inputs (MISHCHENKO; CHATTERJEE; BRAYTON, 2006b; MISHCHENKO; BRAYTON; CHATTERJEE, 2008).

Structurally, a k-cut is the same as the conventional cut, comprising a list of leaf nodes and a single root node, with the only distinction being the imposed limitation on its size. In this category of cuts, only those with a size not exceeding K are retained, while those surpassing K are discarded (CONG; DING, 1996). This approach aims to reduce the number of cuts, thereby enhancing efficiency.

In other words, a *K*-cut defines a region in the graph that represents the logic

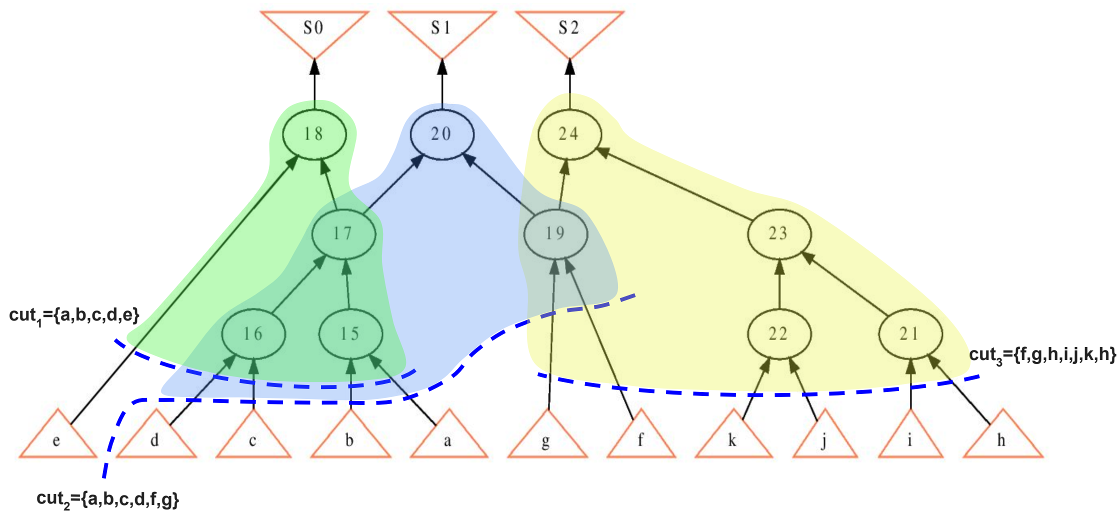
function of n, using at most K variables. It is a useful tool in technology mapping, especially when targeting FPGAs, which are composed of LUTs that can implement any logic function up to a fixed number of inputs.

K-cuts are widely used in logic synthesis, serving to optimize circuits through logic optimization, and also used in technology mapping for both FPGA and ASIC contexts, where the *K*-cuts are replaced by LUTs, for FPGA, or by a standard cell, for ASICs context. The *K*-cuts define regions within the circuit where optimization and mapping techniques are applied.

This type of cut plays an important role in logic synthesis by optimizing circuits through logic optimization such as (MISHCHENKO; CHATTERJEE; BRAYTON, 2006a) where some *K*-cuts are selected and replaced by others with smaller precomputed subgraphs. Moreover, the *K*-cuts are used in technology mapping for both FPGA and ASIC contexts, where the *K*-cuts delineate specific regions within the circuit, which are subsequently replaced by LUTs in FPGA implementations or standard cells in ASICs implementations.

To illustrate the *K*-cut, Figure 4.1 provides an example of an AIG with three cuts, $c_1 = \{a, b, c, d, e\}$, $c_2 = \{a, b, c, d, f, g\}$, and $c_3 = \{f, g, h, i, j, k\}$, considering $K = 5$, we have that the cut c_3 is not a *K*-cut because this cut has as the leaves the set $\{f, g, h, i, j, k\}$ that is composed of six leaves ($|c_3| = 6$), thus this cut is not a k-feasible cut. In summary, a *K*-cut is structurally the same as the conventional cut and only has a restriction on the number of leaves.

Figure 4.1 – *K*-cut example.



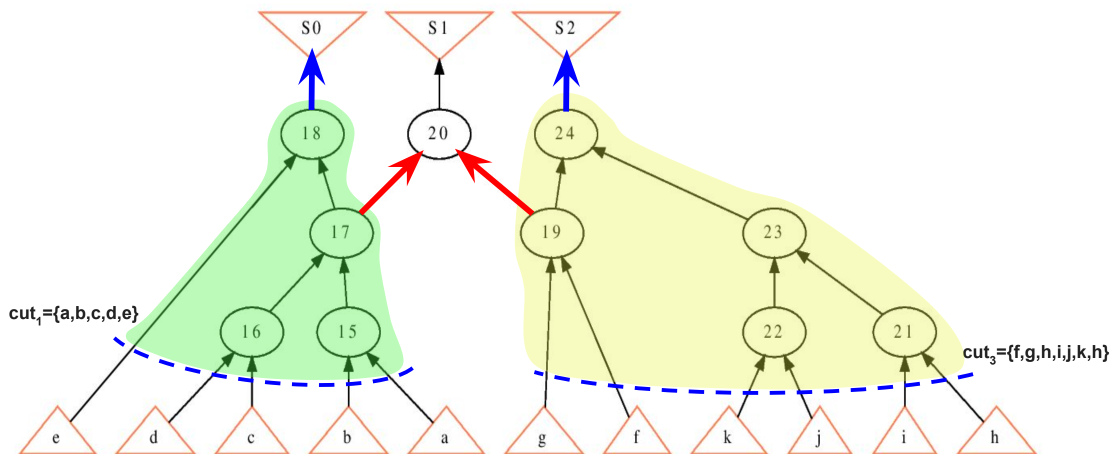
The cuts shown in Figure 4.1 generate the best cover for this AIG, considering a mapping to LUTs using *K*-cuts with $K = 6$, this result was obeyed using the *"if -K 6"*

command in ABC (BRAYTON; MISHCHENKO, 2010). In this example, some nodes belong to more than one *K*-cut, this occurs because the *K*-cut can have only one output, and in this case to obtain the smaller number of LUTs, it is necessary to duplicate some nodes.

A *K*-cut is a type of cut where its structure has only one output. In this type of cut, there may be side-outputs, which are not considered in the cut. Therefore, the subgraph represented by the *K*-cut can have additional interconnection points with the remaining graph (REIS, 2018).

For example, in Figure 4.2, two *K*-cuts, cut_1 and cut_2, are illustrated, which have as outputs the nodes 18 and 24 (blue arrows), respectively. However, as highlighted with the red arrows, cut_1 has the side-output from node 17 to node 20, and cut_3 has the side-output from node 19 to node 20, which are not considered outputs of these *K*-cuts. Consequently, *K*-cuts cannot fully isolate such logic from the rest of the circuit.

Figure 4.2 – Side-outputs of the cut_1 and cut_3. The blue arrow illustrates the *K*-cut's output and the red arrow illustrates the side edges.



The following equations correspond to the *K*-cuts illustrated in Figure 4.2: $18 = a \cdot b \cdot c \cdot d \cdot e$ for cut_1 and $24 = f \cdot g \cdot h \cdot i \cdot j \cdot k$ for cut_2. In these examples, the remaining nodes, including nodes 17 and 19, do not have equations because they are not the outputs of any cut.

4.2.2 *KL*-cut

Since *K*-cut is limited in terms of the number of outputs, which can not have more than one output, the *KL*-cut arises as an alternative to the *K*-cut limitation. Proposed by

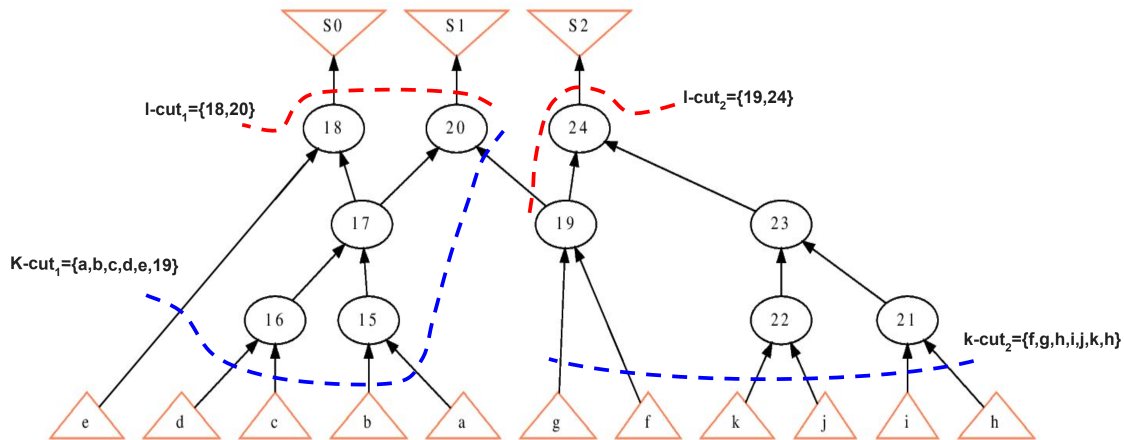
(MARTINELLO et al., 2010), the *KL*-cut is a variation of traditional *K*-cuts which allows the identification of subgraphs with more than only one output.

As the *K*-cuts, the *KL*-cut follows the same definition of a *K*-feasible cut, having up to *K* leaves nodes. However, *KL*-cuts provide more flexibility in terms of the number of outputs, in this structure the number of outputs is not fixed to only one output, the *KL*-cut can have up to L outputs. Thus, the *KL*-cut is also a *L*-feasible cut if the cut has up to *L* outputs.

Structurally, a *KL*-cut is very similar to a *K*-cut, these two types of cuts are exactly equal in terms of the leaf nodes. The main distinction between *K*- and *KL*-cuts is the number of outputs that each type can represent.

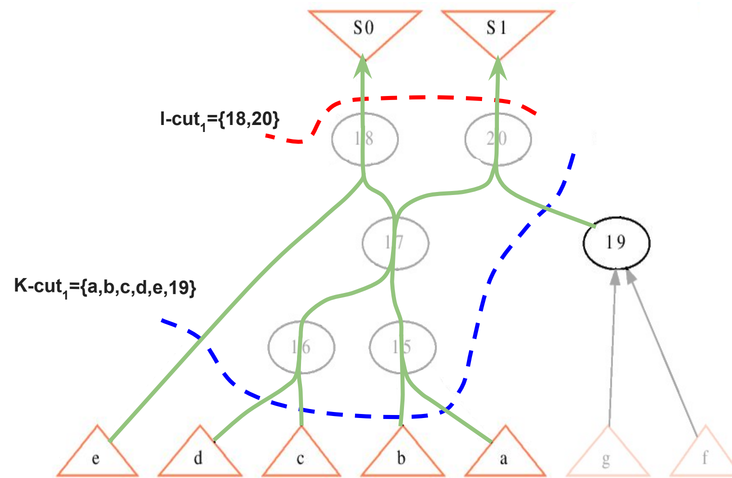
To demonstrate a *KL*-cut, first it is necessary to present the *L*-cut. A *L*-cut is a *backcut*, present in (MARTINELLO et al., 2010), which is quite similar to a cut, that is cut at some edges in the subject graph, however, instead of being a cut at the fanins of a node, which are the inputs of the cut, a *L*-cut is a cut at the fanouts. Figure 4.3 shows an AIG where one can identify two *K*-cuts, described by the blue dashed lines, and two *L*-cuts, described by the red dashed line.

Figure 4.3 – *KL*-cut in AIG.

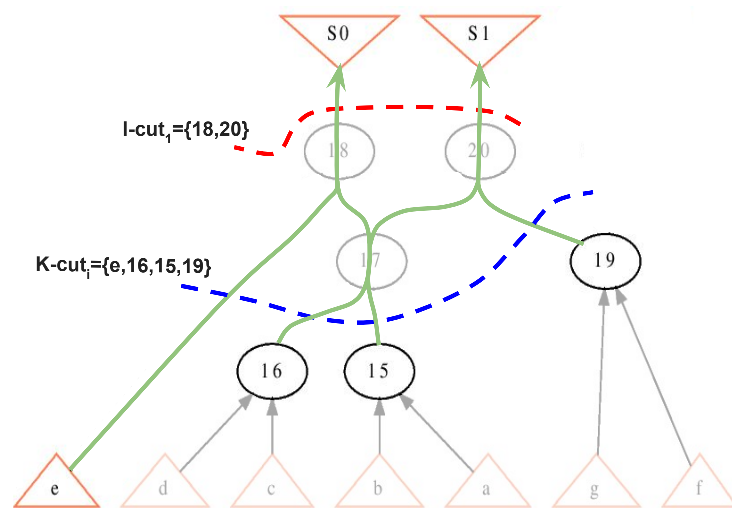


An *L*-cut can be associated with more than one *K*-cut. As an example, Figure 4.4 illustrates two distinct *K*-cuts, $K\text{-}cut_1$ and $K\text{-}cut_2$, associated with the same the $L\text{-}cut_1$. The *backcut* $L\text{-}cut_1$ is associated with both $K\text{-}cut_1$ and $K\text{-}cut_2$ because the $L\text{-}cut_1$ intercepts all TFOs (the solid green arrows) from the leaves of $K\text{-}cut_1$ and $K\text{-}cut_1$, Figure 4.4(a) and Figure 4.4(b) respectively.

Figure 4.4 – Example of L-cut crossing the TFO of K-cuts



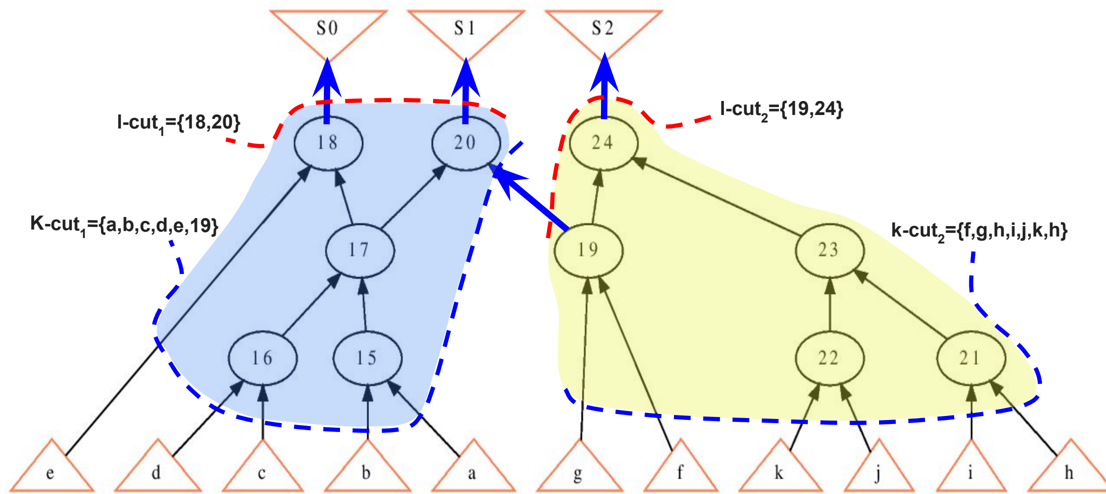
(a) TFO of $k\text{-}cut_1$



(b) TFO of $k\text{-}cut_i$

Like a *K*-cut, a *KL*-cut covers a set of nodes in the AIG, the distinction between these two types of cuts is that *KL*-cut covers a subcircuit with up to *L* outputs. Figure 4.5 illustrates the same AIG example from section 4.2.1 covered with *KL*-cuts. In this example are shown two *KL*-cuts which are $\{\{a, b, c, d, e, 19\}, \{18, 20\}\}$ resultant of $K\text{-}cut_1$ and $L\text{-}cut_1$, and *KL*-cut $\{\{f, g, i, j, k, h\}, \{19, 24\}\}$ resultant of $K\text{-}cut_2$ and $L\text{-}cut_2$,

Figure 4.5 – *KL*-cut cover in AIG. The blue arrows represent the *KL*-cuts outputs.



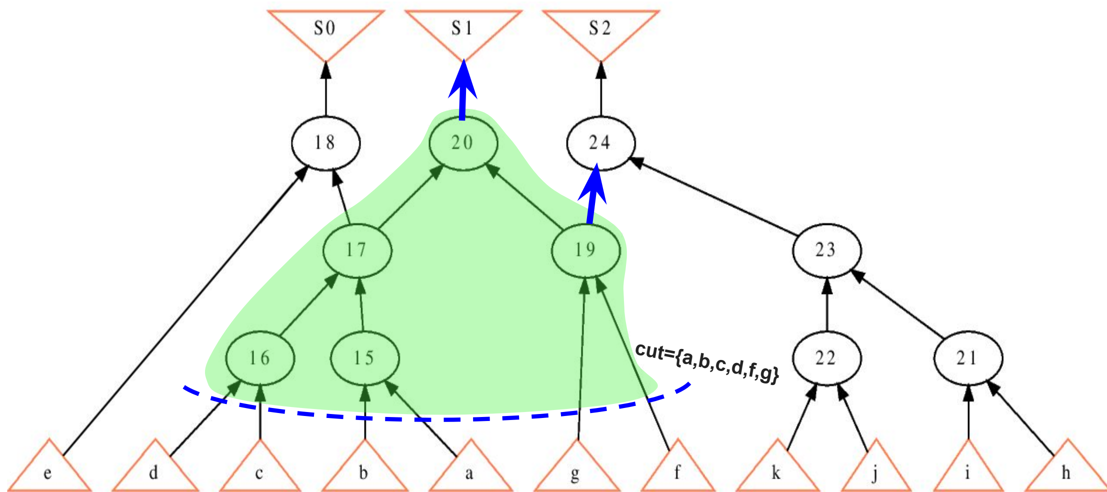
A *KL*-feasible cut is defined as:

$$\{\{l\},\{o\}\} \mid l \in \text{Leaves}, o \in \text{Outputs}, |l| \leq K, |o| \leq L \tag{4.1}$$

Therefore, the *KL*-cuts showed in Figure 4.5 are *KL*-feasible for $K = 6$ and $L = 2$, where both cuts have no more than six inputs and two outputs. Any cut that exceeds these two values, is not a *KL*-feasible cut and, thus, is not a *KL*-cut.

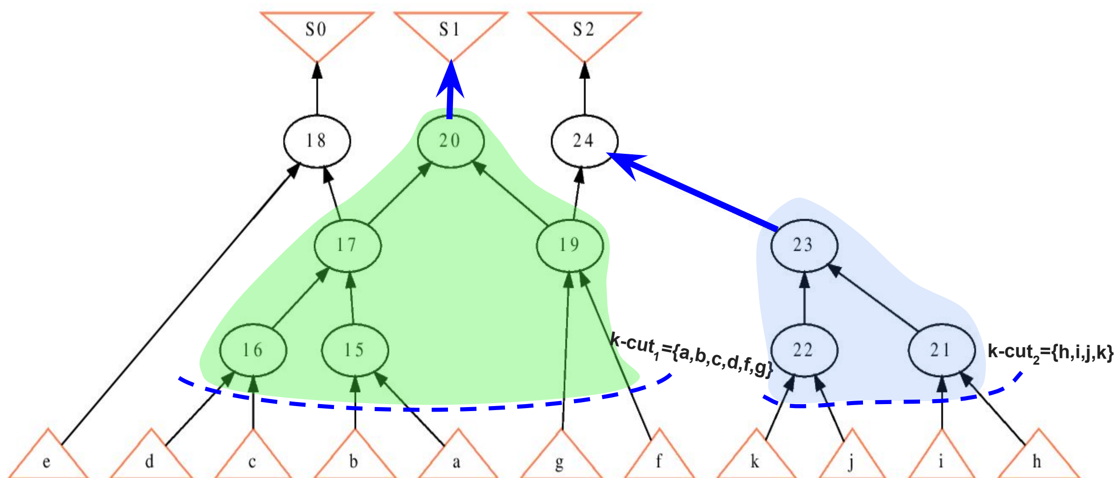
However, not every sub-circuit with more than one output is necessarily a *KL*-cut. For instance, Figure 4.6 depicts a cut that is not a *KL*-cut for $L = 2$, where the blue arrows represent the outputs. This cut might be mistaken for a *KL*-cut with two inputs. However, despite having the correct number of outputs (two in this case), there is a side-output from node 17 to node 18, violating the fundamental property of a *KL*-cut, which dictates that every fanout from the sub-circuit must be an output of the *KL*-cut to qualify as such. Therefore, for a cut to be classified as a *KL*-cut, it must not only be *K*-feasible and *L*-feasible but also can not have side-outputs.

Figure 4.6 – Example of a not valid *KL*-cut for $L = 2$.



Continuing the discussion on the distinctions between *K*-cuts and *KL*-cuts, another question arises: Is a *K*-cut considered a *KL*-cut when $L = 1$? To address this query, Figure 4.7 illustrates two *K*-cuts, $K\text{-}cut_1$ and $K\text{-}cut_2$, with outputs 20 and 23 respectively. With $K = 6$, both cuts qualify as *K*-cuts since they possess only one output, indicated by the blue arrows, and respect the number of inputs limit.

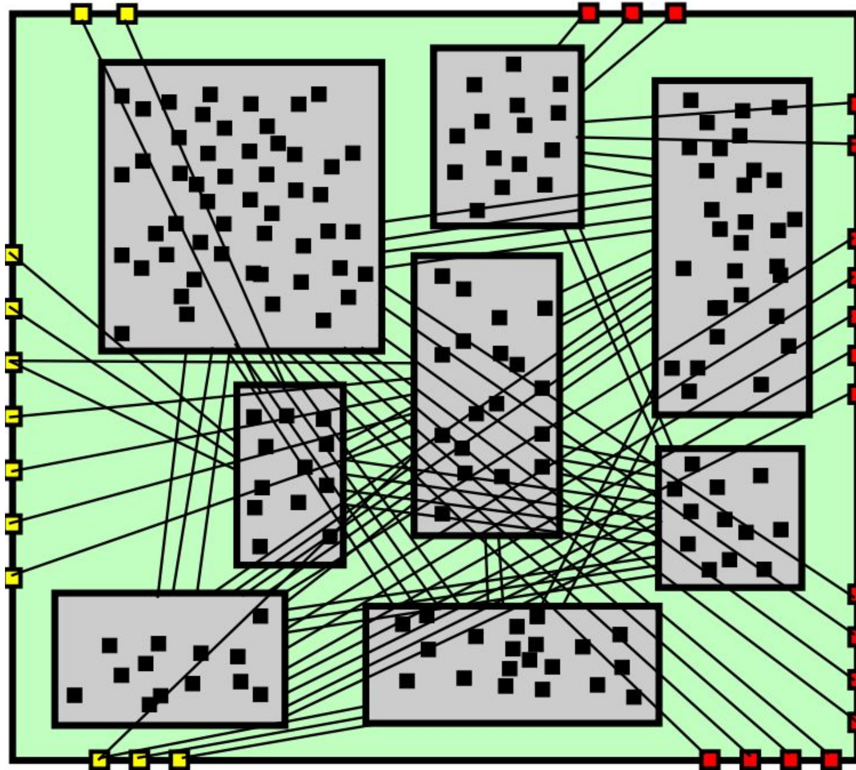
Figure 4.7 – Example of when a *K*-cut is a *KL*-cut with $L = 1$.



However, as depicted in Figure 4.7, only $K\text{-}cut_2$ satisfies the criteria for a *KL*-cut with $L = 1$ since it lacks side-outputs. In contrast, $K\text{-}cut_1$ fails to meet the requirements of a *KL*-cut due to its two side-outputs from nodes 17 and 19. This example highlights that the presence of only one output is insufficient to classify a *K*-cut as a *KL*-cut with $L = 1$; rather, the absence of side-outputs is necessary for a *K*-cut to be also considered a *KL*-cut.

When a circuit is positioned into *KL*-cuts, blocks, as shown in Figure 4.8, the outputs from each block, and the PIs, are connected to other input blocks. This illustrated example is a bad-quality circuit because the blocks' positions imply long wires to connect the blocks and POs (REIS; MATOS, 2018).

Figure 4.8 – Circuit partitioned into *KL*-cuts.



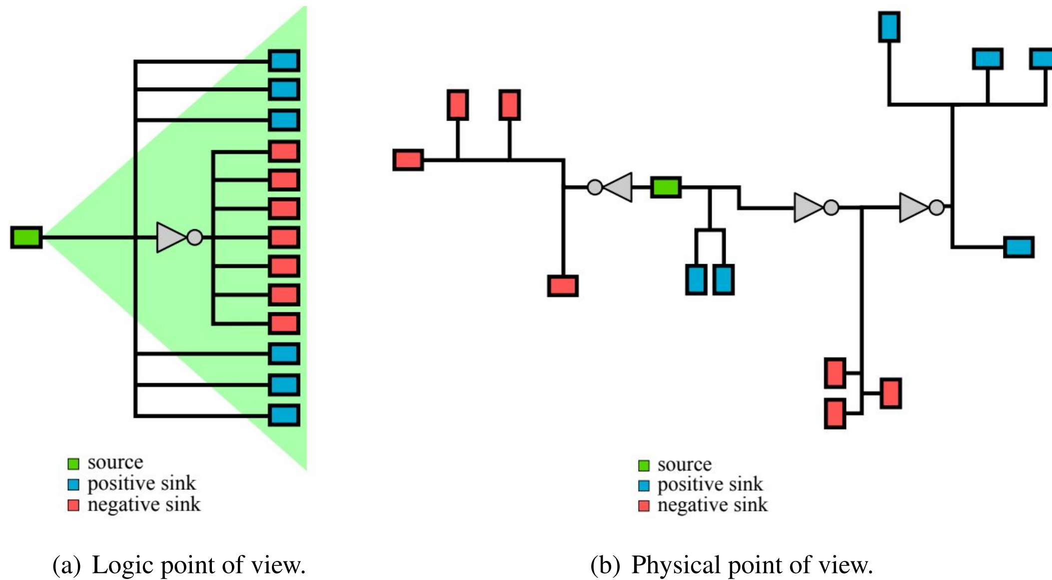
Source: (REIS; MATOS, 2018).

While the *KL*-cuts act as node containers in the AIG, there is another type of cut used to signal distribution called *1L*-cuts, that is a *KL*-cut with the fixed number of inputs $K = 1$. The *1L*-cut is useful to make the placement of the *KL*-cuts in the circuit aiming to improve its quality (REIS; MATOS, 2018).

4.2.2.1 1L-cut

A *1L*-cut is a specialized type of cut that has up to L outputs while being restricted to only one input. Unlike the previously discussed cuts, which are employed in identifying and representing circuit components known as logical blocks, the 1L-cut serves a purpose in routing, where routing involves the distribution of signals between these logical blocks (REIS, 2018). The *1L*-cut structure is shown in Figure 4.9.

Figure 4.9 – Points of view of a *1L*-cut.



(a) Logic point of view. (b) Physical point of view.

Source: (REIS; MATOS, 2018).

4.2.3 Windowing

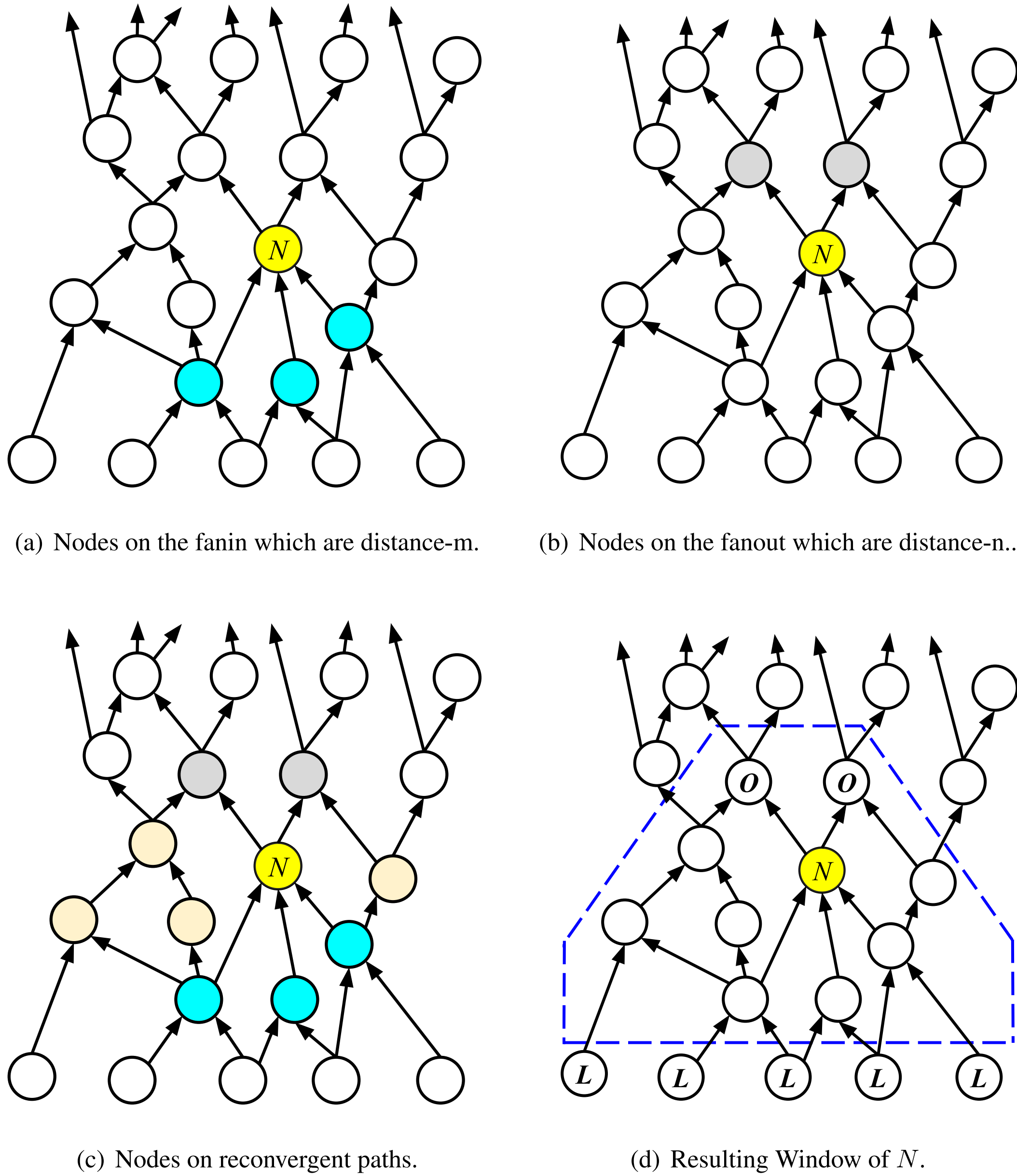
The concept of windowing was introduced in (MISHCHENKO; WANG; KAM, 2003). The technique of defining windows consists of identifying regions around some nodes in the AIG. Delimiting such regions on the circuit, it is possible to apply local optimizations inside each window separately.

A window is constructed around a node N of the graph. To achieve this, a maximum distance limit is defined from the node both in the direction of its fanout and in the direction of its fanin. These values are defined by m and n, respectively.

Figure 4.10 illustrates, at a high level, the identification of nodes in the neighborhood of N. First, are identified the nodes that are at a distance n from the node N (Figure 4.10(a)). Next, are identified the nodes that are at a distance m from the node N (Figure 4.10(b)).

Subsequently, the nodes in reconvergent paths are identified (Figure 4.10(c)). Finally, the set of nodes obtained in the previous steps defines the window around the node N, where the nodes O are the outputs of the window and the nodes L are the inputs of the window (Figure 4.10(d)).

Figure 4.10 – 1x1 (m x n) Window for node N.



(a) Nodes on the fanin which are distance-m. (b) Nodes on the fanout which are distance-n..

(c) Nodes on reconvergent paths. (d) Resulting Window of N.

Source: Adapted from (MISHCHENKO; BRAYTON, 2005).

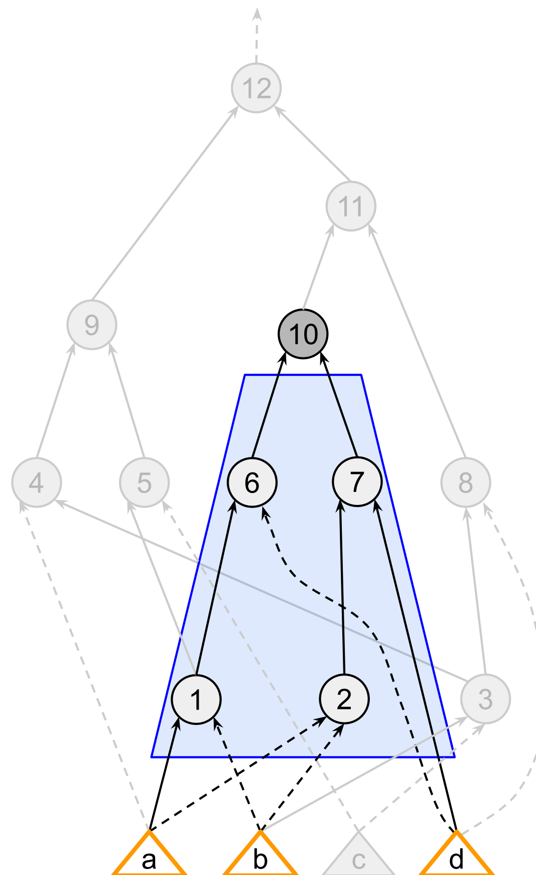
4.2.3.1 Fanin Cone

A node's fanin cone, also known as the TFI cone, refers to the set of nodes accessible through its incoming edges. In this way, the fanin cone of a node n includes the node n itself and all nodes in the TFI of n, including the PIs (MISHCHENKO; CHATTERJEE; BRAYTON, 2006a). In other words, the fanin cone is the set of nodes that produce the logic for the node n.

Figure 4.11 illustrates the fanin cone of the node *10*. As depicted, this fanin cone

is composed of nodes *1*, *2*, *6*, and *7*, which collectively provide the logic utilized by node *10*.

Figure 4.11 – Fanin cone of the node 10.

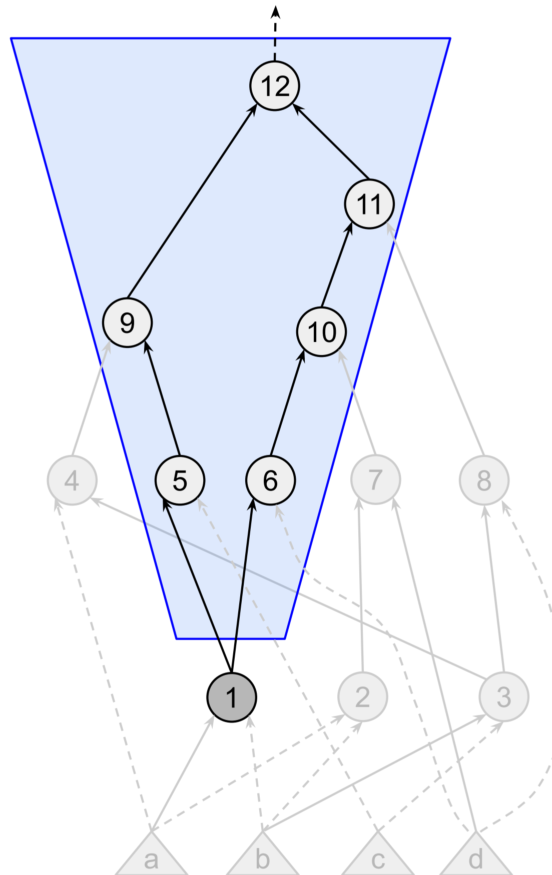


4.2.3.2 Fanout Cone

Similarly to the Fanin Cone, the fanout cone encompasses the nodes accessible through its outgoing edges (MISHCHENKO; CHATTERJEE; BRAYTON, 2006b). The TFO cone of node nd_j includes the node nd_j itself and all nodes that are in the TFO of nd_j, including the POs (MISHCHENKO; CHATTERJEE; BRAYTON, 2006a).

Figure 4.12 illustrates the fanout cone of node *1*, comprised of nodes *5*, *6*, *9*, *10*, *11*, and *12*. These nodes, accessible from the fanouts of node *1*, collectively represent the portion of the circuit influenced by node *1*.

Figure 4.12 – Fanout cone of the node 8.



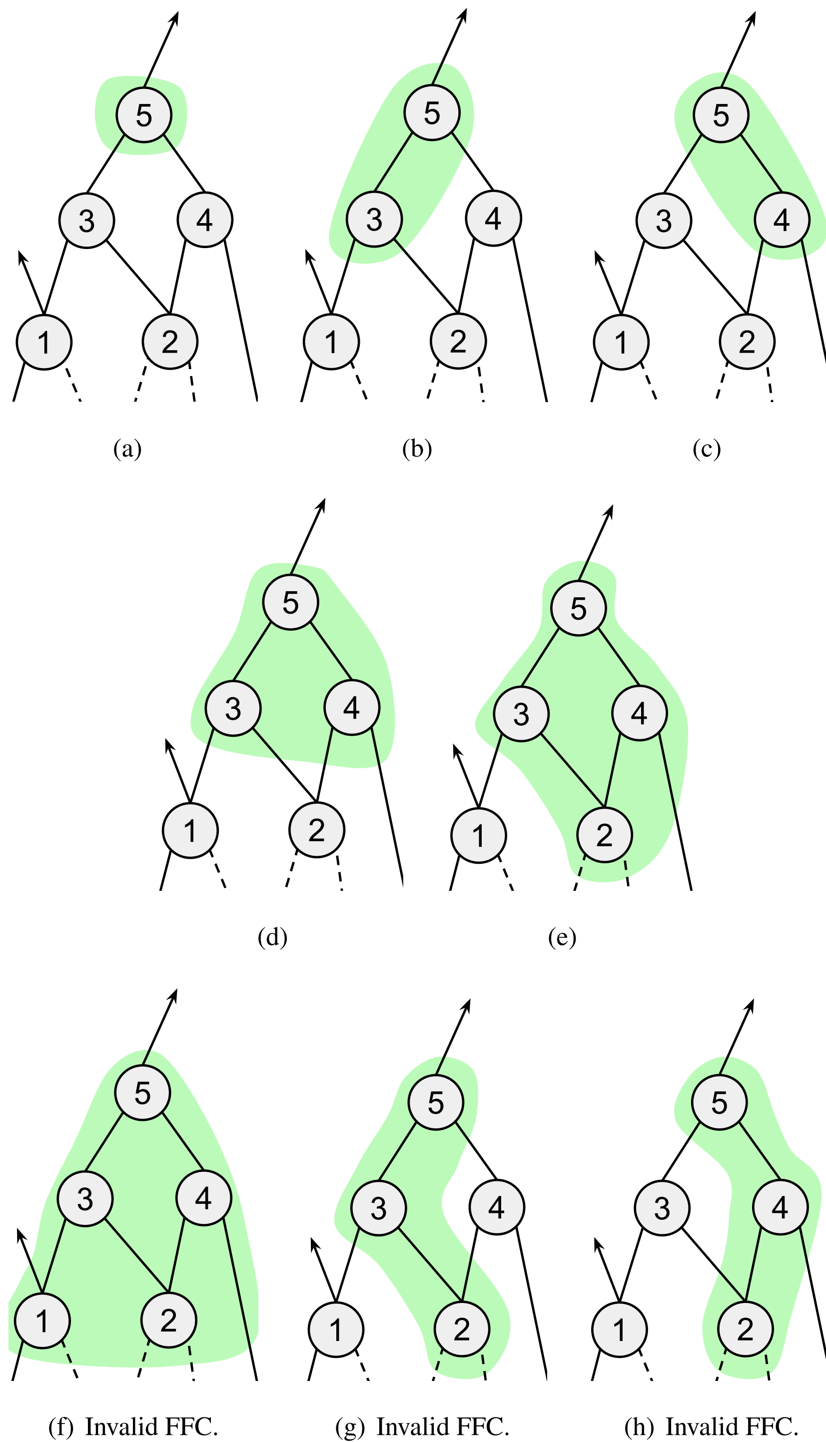
4.2.3.3 Fanout-Free Cone

A Fanout-Free Cone (FFC) centered at node n constitutes a subgraph structure where node n is the root. In the FFC, each node, excluding the root, has outgoing edges directed solely towards the root node, creating a cone-like topology within the graph. In a straight way, each path from an input of the FFC converges to the root node of the FFC (CONG; DING, 1996).

An AIG node can comprise various FFCs, as depicted in Figure 4.13. In this illustration, the FFCs of node 5 are highlighted in green shading. Figures 4.13(a) through 4.13(e) depict five examples of FCCs for node 5, wherein only node 5, the root, has an output outside the FFC, thus illustrating FFCs of node 5.

However, in Figure 4.13(f), node 1 has an outgoing edge outside the shaded area. Since the root node in this example is node 5, this set of nodes does not constitute an FFC for node 5. Similar instances are illustrated in Figures 4.13(g) and 4.13(h), where node 2 has an outgoing edge outside the shaded area. Therefore, these three figures provide examples of node sets that do not form FFCs for node 5.

Figure 4.13 – All Fanout-Free Cones of node 5.



Given that all logic generated within the FFC is solely utilized within the cone itself, the FFC offers the ability to completely isolate the logic contained within the cone from the remainder of the graph. The FFC contains the logic used only by its root node.

4.2.3.4 Maximum Fanout-Free Cone

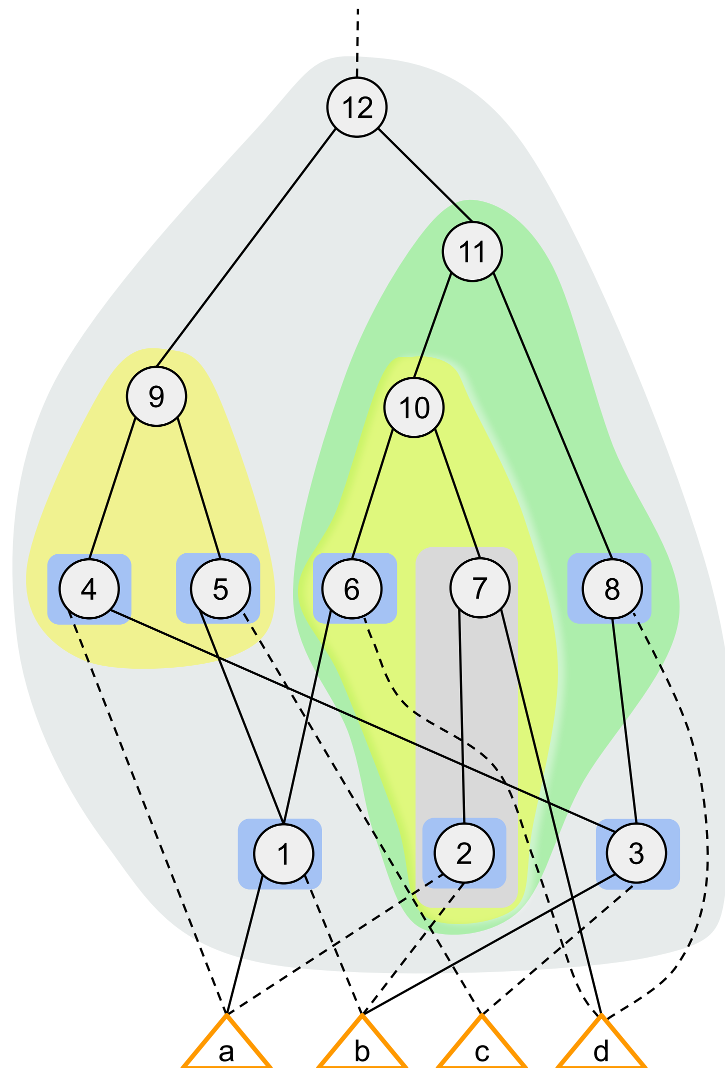
The Maximum Fanout-Free Cone (MFFC) of a node in a circuit refers to a subset of its fanin cone that contains only nodes such that every path from these nodes to the POs passes through the node itself. In simpler terms, the MFFC of a node encompasses all the logic exclusively utilized by that node. When a node is either removed or substituted within the circuit, the logic contained within its MFFC can also be safely removed, as it is no longer necessary for the circuit's functionality (MISHCHENKO et al., 2007; CALVINO et al., 2022).

The MFFC encompasses every FFC rooted at node n. Additionally, an MFFC can be further characterized as follows: for a given node w, if the output of node w is within the MFFC(v), then node w itself is also within the MFFC(v). In other words, *output*(w) $\subseteq$ *MFFC(v)* $\Rightarrow$ $w \in$ *MFFC(v)*. This characterization is based on the work by Cong and Ding (CONG; DING, 1996).

The MFFC is defined as the largest FFC (Fanout-Free Cone) associated with a particular node, it encompasses all logic paths exclusively serving that node's operations (RIENER et al., 2019). Referring back to the example of Figure 4.13, from the section Section 4.2.3.3, the MFFC of node 5 corresponds to the FCC depicted in Figure 4.13(e). This is because the FCC of Figure 4.13(e) encapsulates all logic generated exclusively to node 5.

In an AIG, each node has its unique MFFC, denoted as MFFC(n), where n represents the specific node. Figure 4.14 illustrates the MFFC of each node in a given AIG, with each MFFC's node highlighted in a shaded color.

Figure 4.14 – MFFC.



Notably, an interesting characteristic of MFFCs, as observed in the example of Figure 4.14, is that one MFFC is either completely separate from another (disjoint) or completely encompasses it (contained), as outlined in (CONG; DING, 1993). Hence, there are no instances of partial intersection between two MFFCs. For example, the MFFC of node 10 is fully contained in the MFFC of node 11, while the MFFC of node 9 is completely disjoint, the same can be observed for the other MFFCs in the example.

In summary, the MFFC represents the essential logic utilized by a node within a circuit, offering insights into circuit behavior and facilitating optimization efforts by identifying redundant or unnecessary logic paths, ensuring that some logic optimization can be applied within the MMFC without changing the circuit behavior.

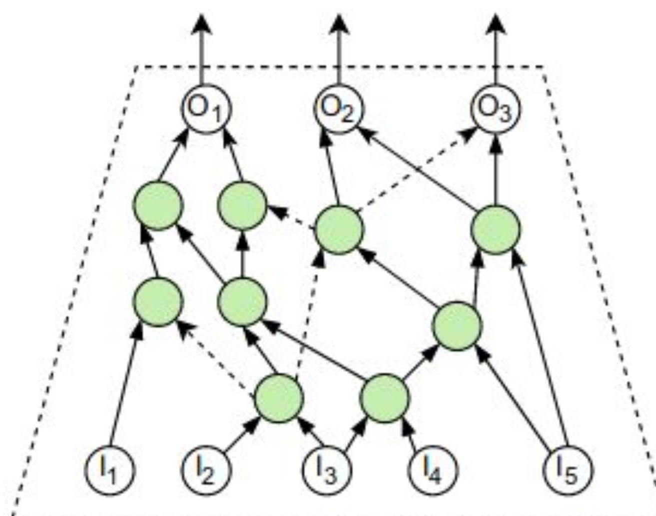
4.2.3.5 Maximum Fanout-Free Window

A recent data structure is the *maximum fanout-free window* (MFFW), proposed by (TANG et al., 2023). The MFFW is an extension of *fanout-free window* (FFW) (ZHU et al., 2023). An MFFW consists of a fully expanded FFW that includes all nodes that can be expressed by the Boolean FFW function.

Let G be an AIG defined by (PI, PO, V, E), where these elements are the primary inputs and outputs, vertices and edges, in that order. An FFW is a subgraph of G defined by (I, V', E') which are the inputs, vertices, and edges, in this order, (ZHU et al., 2023). This is also a technique used to limit a region of the AIG used to allow local optimizations to be applied.

The idea of the MFFW is to group, in the same window, all the Boolean logic of the window's inputs. Figure 4.15 provides an example of an MFFW which has as input the nodes I_1, I_2, I_3, I_4, I_5, and as output the nodes O_1, O_2, O_3. The window retains all nodes that constitute the Boolean logic generated by the window's inputs. This way, optimizations can be applied within the window without affecting other parts of the circuit. This approach allows enhancements made to the logic within the window to remain isolated, preserving the integrity and functionality of the circuit.

Figure 4.15 – MFFW.



Source: (ZHU et al., 2023)

4.2.4 Structural differences between MFFW and *KL*-cuts

MFFW and *KL*-cut are very similar structures; both are *K*-feasible, meaning they have at most *K* inputs and both can encompass the entire logic (subgraph) defined by their set of inputs, particularly when *L* is unbounded in the context of *KL*-cuts.

The main difference between these two structures is that when *KL*-cuts are used with a defined value for *L*, *KL*-cuts with at most *L* outputs are enumerated. This allows for more precise control over the number of outputs. In contrast, MFFW does not provide control over the number of outputs, this data structure always includes the entire subgraph defined by the inputs and, consequently, all the outputs of that subgraph.

4.3 Non-structural types of cuts

In this section, we discuss some types of cuts that are not structurally different. The definition of *factor cuts* and *priority cuts* is related to the way they are computed. The goal of the algorithms to compute factor cuts and priority cuts is simply to compute structural *K*-cuts more efficiently.

4.3.1 Factor Cuts

Proposed by (CHATTERJEE; MISHCHENKO; BRAYTON, 2006), factor cuts is an efficient technique for enumerating k-feasible cuts. Rather than enumerating all k-feasible cuts, this technique involves the enumeration of a collection of cuts, classified as local and global cuts, which collectively form the factor cuts.

The concept behind factor cuts is to construct both local and global sets of cuts, which are smaller than the entire set of k-feasible cuts. As needed, these sets can then be expanded to generate the complete (or nearly complete) set of all k-feasible cuts. The definition of the local and global groups may vary based on the factorization scheme employed.

The primary goal of factor cuts is to enumerate k-feasible cuts without exhaustively enumerating all of them, thereby accelerating the process by reducing the number of cuts. K-feasible cuts for a node can be generated on the fly as needed. A complete explanation of the enumeration process will be presented in Section 5.2.1.2.

4.3.2 Priority Cuts

The Priority Cut technique aims to optimize the efficiency of the cut enumeration algorithm by selectively maintaining a limited set of cuts for each node. Instead of enumerating all possible cuts, this approach focuses on retaining only the best cuts based on specific criteria, such as area and delay (MISHCHENKO et al., 2007). Selecting only the Priority Cuts minimizes the impact on the performance to generate the cuts for huge circuits, which could be impractical.

4.4 Windowing vs. Covering

In the context of Circuit Design, with the use of AIGs, the terms "coverage" and "windowing" refer to two different processes. Both processes involve identifying regions within the circuit, however, each has a specific purpose, in the sequence, the characteristics of each process will be presented and their distinctions will be shown.

Covering an AIG involves identifying specific regions or subgraphs within the AIG that can be used for replacement with a component of the technology aimed at, such as standard cells or LUTs. Also, these regions defined can be used to apply some local logic synthesis within the subregion. This process is often part of local synthesis or technology mapping to improve the design's overall performance, area, or power consumption. The goal is to identify specific subgraphs or regions within the AIG that exhibit opportunities for optimization or replacement.

Windowing, in the context of AIGs, refers to focusing on a specific portion or "window" of the circuit rather than the entire design. It involves selecting a subset of the circuit to optimize or analyze, rather than considering the entire AIG.

In essence, coverage ensures a full representation of the entire AIG, while windowing involves a more targeted and localized approach to synthesis or optimization within specific regions of the AIG.

4.4.1 Windowing with MFFW

MFFWs can be used in the context of windowing, allowing regions of the AIG to be identified and completely isolated. This enables synthesis methods to be performed

within the MFFWs, as presented in (ZHU et al., 2023).

4.4.2 Covering with *KL*-cuts

KL-cuts can be used in the context of AIG covering, as presented in the work of (MARTINELLO et al., 2010). Although their work does not present a method to derive the AIG covering using *KL*-cuts, they generate coverings to validate their proposed method.

4.4.3 Windowing with *KL*-cuts

KL-cuts can also be used in the context of windowing, as in the work proposed by (MACHADO et al., 2012). Their approach utilizes *KL*-cuts in mapped circuits to perform remapping to reduce the area and delay of the circuits.

4.4.4 Covering with *KL*-cuts and 1L-cuts

As discussed in Section 4.2.2, *KL*-cuts are viewed as blocks of local logic, while 1L-cuts are a special type of cut used for signal distribution. The work by (REIS; MATOS, 2018) combines these two types of cuts to derive a cover where the AIG is partitioned into *KL*-cuts, and the signal distribution between the cuts is considered.

4.5 Contributions of this chapter

This chapter presents a review of the different types of cuts, describing the characteristics that differentiate them. In addition to a structural differentiation, some of the main algorithms used in the context of cuts in AIGs are described. Therefore, this chapter provides the reader with a characterization of the fundamental concepts in the context of cuts in AIGs.

5 LITERATURE REVIEW

5.1 About this chapter

Throughout this chapter, we will explore the evolution of cuts in AIGs over time. By examining the existing literature, we will seek to understand how these cuts have evolved, what innovations have been introduced, and how these changes have impacted the field.

This review will allow us to understand the trajectory of cutting variants over time, starting from *K*-cuts to *windowing*, MFFW, and *KL*-cuts techniques. This will allow the identification of possible characteristics that require improvements, such as optimizing the execution time of the cut enumeration algorithms, a task that consumes significant execution time and is crucial for the project.

5.2 Cuts in AIGs - a historical overview

5.2.1 *K*-cuts

Cuts in AIGs can have different applications, such as cuts for mapping and cuts used to perform optimizations, in terms of reduction and balancing, of AIG nodes. The following works are example of employing *K*-cuts (RIENER; MISHCHENKO; SOEKEN, 2020), (RIENER et al., 2019), (YANG; WANG; MISHCHENKO, 2012), (LI; DUBROVA, 2011) and (MISHCHENKO; CHATTERJEE; BRAYTON, 2006a).

5.2.1.1 Enumerating all K cuts

The *K*-cuts enumeration process begins from the PIs to the POs in topological order, and the goal is to compute *K*-cut enumerations for each node, considering that the cuts of a node's fanin nodes must be enumerated before computing the cuts for the current node.

The enumeration process can be expressed by the following equations: first, the *combine* procedure, described in Equation 5.1, creates a new set of *K*-cuts by taking unions of subsets from sets A and B. The condition $|u \cup v| \leq k$ ensures that the resulting *K*-cuts have a limited number of inputs, guaranteeing that the newly generated cuts have

no more than K inputs (MISHCHENKO et al., 2007).

$$A \bowtie B = u \cup v \mid u \in A, v \in B, |u \cup v| \leq k \tag{5.1}$$

The second equation describes the enumeration process. If the node is a PI, it will exclusively have the self-cut, depicted by the node identifier. On another hand, if the node is not a PI, its cuts are computed by combining the cuts of its fanins. The resulting set of cuts is then united with the node's self-cut, as described by Equation 5.2. In this equation, n represents the processed node, while n_1 and n_2 denote its fanins.

$$\mathbf{\Phi}(n) = \begin{cases} \{\{n\}\} & : n \in PI \\ \{\{n\}\} \cup \{\mathbf{\Phi}(n_1) \bowtie \mathbf{\Phi}(n_2)\} & : otherwise \end{cases} \tag{5.2}$$

In Table 5.1, all the k-feasible cuts of the AIG are depicted in Figure 3.2 are enumerated. The first column shows the node's identification, while the second column presents the *K*-cuts generated for each corresponding node. It is notable that, as the number of nodes increases, the number of cuts per node also increases.

Table 5.1 – All k-cuts for the AIG of Figure 3.2.

node	**K-Cuts (k=3)**
a	{a}
b	{b}
C_{in}	{C_{in}}
6	{a,b}, {6}
7	{a,b}, {7}
8	{a,b}, {6,7}, {8}
9	{a, b, C_{in}}, {C_{in}, 8}, {C_{in}, 6, 7}, {9}
10	{a, b, C_{in}}, {C_{in}, 8}, {C_{in}, 6, 7}, {10}
11	{a, b, C_{in}}, {C_{in}, 8}, {C_{in}, 6, 7}, {9, 10}, {11}
12	{a, b, C_{in}}, {a,b,10}, {C_{in}, 7, 8}, {7, 10}, {C_{in}, 6, 7}, {12}

Enumerating all k-feasible cuts can become impractical for very large circuits. According to (CHATTERJEE; MISHCHENKO; BRAYTON, 2006), the number of cuts (K) in a network of size (n) is $O(n^k)$. To address this challenge, various strategies have

been proposed to optimize the enumeration procedure, and some of them will be discussed subsequently.

5.2.1.2 Enumerating factor cuts

As presented in Section 4.3.1, factor cuts (CHATTERJEE; MISHCHENKO; BRAYTON, 2006) is a technique to speed up the cut enumeration process. This method categorizes cuts into two distinct groups: global and local. Furthermore, within this process, there exist different schemes, such as the complete, and partial, factorization. Each scheme is characterized by its unique features, as will be explained below.

Before getting into the procedures for generating global and local cuts, it is important to understand how cuts are generated from the global and local cut sets. The process is called *expansion*, Equation 5.3, and works as follows: Let c be a global cut of node $n \in G$ (where G is the subject graph), and let c_i be a local cut of a node i belonging to c. Define l as $l = \bigcup_i c_i$. l is considered a cut of node n and is a k-feasible cut if $|l| < K$. If l is K-feasible, it represents a 1-step expansion of n. The set of cuts obtained by expanding c is denoted as 1-step(c).

$$\text{1-step}(c) = \{l \mid l \text{ is a 1-step expansion of } c\} \tag{5.3}$$

5.2.1.2.1 Complete Cut Factorization

In AIGs, there are two types of nodes: *dag nodes*, which have more than two outgoing edges, and *tree nodes*, which have only one outgoing edge. The set of dag nodes is denoted by $\mathcal{F}$, and the set of tree nodes is denoted by $\mathcal{T}$.

In the *complete factorization* scheme also referred to as *complete factor cuts*, the local cuts are the *tree cuts*, and the global cuts are the *reduced cuts*. The *complete factorization* scheme provides the capability to generate any k-feasible cut through the 1-step expansion.

5.2.1.2.1.1 Tree Cuts - (Local Cuts)

The set of tree cuts for a node n is denoted by $\mathbf{\Phi}_{\mathcal{T}}(n)$. This set comprises a subset of the k-feasible cuts for node n, which consists only of cuts composed by nodes from the factor tree of n.

The definition of $\Phi_{\mathcal{T}}(n)$ is provided in Equation 5.4. When a node n is not a PI,

its tree cuts include both its self-cut and the k-feasible cuts generated by combining the tree cuts of its fanins.

$$\mathbf{\Phi}_{\mathcal{T}}(n) = \begin{cases} \{\{n\}\} & : n \text{ is a PI} \\ \{\{n\}\} \cup (\Phi^{\dagger}_{\mathcal{T}}(n_1) \bowtie \Phi^{\dagger}_{\mathcal{T}}(n_2)) & : \text{otherwise} \end{cases} \tag{5.4}$$

Only cuts from fanins that are tree nodes are used in this process, as provided by $\Phi^{\dagger}_{\mathcal{T}}(n_i)$, which is described in Equation 5.5.

$$\mathbf{\Phi}^{\dagger}{}_{\mathcal{T}}(n_i) = \begin{cases} \emptyset & : n_i \in \mathcal{F} \\ \Phi_{\mathcal{T}}(n_i) & : otherwise \end{cases} \tag{5.5}$$

5.2.1.2.1.2 Reduced Cuts - (Global Cuts)

The enumeration of reduced cuts for a node n, denoted as $\Phi_{\mathcal{R}}(n)$, is similar to the enumeration of k-feasible cuts ($\Phi(n)$). However, $\Phi_{\mathcal{R}}(n)$ does not include the tree cuts of n ($\Phi_{\mathcal{T}}(n)$); they are removed from this set. This removal results in a significantly smaller size for $\Phi_{\mathcal{R}}(n)$ compared to $\Phi(n)$.

$$\mathbf{\Phi}_{\mathcal{R}}(n) = \begin{cases} \{\{n\}\} & : n \text{ is a PI} \\ \{\{n\}\} \cup ((\Phi_{\mathcal{R}}(n_1) \bowtie \Phi_{\mathcal{R}}(n_2)) \setminus \Phi_{\mathcal{T}}(n)) & : \text{otherwise} \end{cases} \tag{5.6}$$

5.2.1.2.2 Partial Cut Factorization

In the *partial factorization* scheme, the generation of the complete set of K-feasible cuts through a 1-step expansion is not guaranteed, unlike in the complete factorization scheme. However, the *partial factorization* scheme is significantly faster and produces good cuts. In this scheme, local cuts are the *leaf-dag cuts*, and global cuts are the *dag cuts*.

5.2.1.2.2.1 Leaf-dag Cuts - (Local Cuts)

The *factor leaf-DAG* of n is a factor tree that also includes the dag nodes feeding into it. The set of k-feasible *leaf-dag cuts* for node n, denoted by $\Phi_{\mathcal{L}}(n)$ and defined in Equation 5.7, is composed only of cuts derived from its *factor leaf-DAG*.

$$\mathbf{\Phi}_{\mathcal{L}}(n) = \begin{cases} \{\{n\}\} & : n \text{ is a PI} \\ \{\{n\}\} \cup (\Phi^{\dagger}_{\mathcal{L}}(n_1) \bowtie \Phi^{\dagger}_{\mathcal{L}}(n_2)) & : \text{otherwise} \end{cases} \tag{5.7}$$

To compute the *leaf-dag cuts* of n, it is necessary to combine the *leaf-dag cuts* from its fanins, for the non-trivial case. The *leaf-dag cuts* of the fanins used in this process are determined by the function $\Phi^{\dagger}_{\mathcal{L}}(n_i)$, as described in Equation 5.8. Unlike $\Phi^{\dagger}_{\mathcal{T}}(n_i)$, this function also takes into account the self-cut of the dag nodes that feed into the *factor leaf-DAG* of n.

$$\mathbf{\Phi}^{\dagger}{}_{\mathcal{L}}(n_i) = \begin{cases} \{\{n_i\}\} & : n \in \mathcal{F} \\ \Phi_{\mathcal{L}}(n_i) & : otherwise \end{cases} \tag{5.8}$$

5.2.1.2.2.2 Dag Cuts - (Global Cuts)

The set of k-feasible dag cuts for node n is denoted by $\Phi_{\mathcal{D}}(n)$ and defined as described in Equation 5.9. This set exclusively contains dag nodes, tree nodes are not considered in its composition. The idea of using only dag nodes in this set is to capture much of the re-convergent paths in the graph, potentially reducing the number of global cuts.

$$\mathbf{\Phi}_{\mathcal{D}}(n) = \begin{cases} \{\{n\}\} & : n \text{ is a PI} \\ \Phi_{\mathcal{D}}(n_1) \bowtie \Phi_{\mathcal{D}}(n_2) & : n \in \mathcal{T} \\ \{\{n\}\} \cup (\Phi_{\mathcal{D}}(n_1) \bowtie \Phi_{\mathcal{D}}(n_2)) & : \text{otherwise} \end{cases} \tag{5.9}$$

5.2.1.3 Enumerating priority cuts

To avoid enumerating all cuts, it can use the approach described in (MISHCHENKO et al., 2007), in which the number of cuts in each node is limited. To not lose the quality of the final circuit, each cut has an associated score that can be in terms of area and/or time so that each node only keeps its best cuts.

5.2.1.4 Cut signature

During the enumeration of *K*-cuts or other processes involving cuts, the necessity to compare cuts for distinction and check dominance can lead to a time-consuming task. An efficient alternative proposed by (MISHCHENKO; CHATTERJEE; BRAYTON, 2006b) to speed up this process involves the use of signatures for the cuts.

The signature of a cut c, denoted as $sign(c)$, is represented as an M-bit integer, where M corresponds to the number of bits composing a word in the computer's processor, as suggested by the authors (MISHCHENKO; CHATTERJEE; BRAYTON, 2006b). The signature of each node is computed using its leaves, l in c, performing a bitwise OR operation on each leaf, as illustrated in Equation 5.10. The resulting signature will contain 1s at the corresponding leaf positions.

$$sign(c) = \sum_{l \in c} 2^{(l \ mod\, \mathrm{M})} \tag{5.10}$$

Utilizing a cut signature enables efficient processing during cut enumeration or other operations involving cuts. Considering the cuts c_1 and c_2, the following checks can be performed:

1. Compare cuts: If c_1 and c_2 are distinct cuts, then their signatures also are distinct ($sign(c_1) \neq sign(c_2)$). If $c_1 = c_2$, it implies that both cuts have the same signature ($sign(c_1) = sign(c_2)$). However, when both cuts have the same signature, there is no guarantee that $c_1 = c_2$. In this case, it is necessary to compare their leaves.
2. Dominance: Determine whether one cut does not dominate the other by performing a bitwise AND operation on their signatures. If $(sign(c_1)\, \mathrm{AND}\, sign(c_2)) \neq sign(c_1)$, then c_1 does not dominate c_2, indicating that some 1s of c_1 are not included in c_2.
3. K-feasible property of a cut: Before creating a new cut derived from the combination of c_1 and c_2, the size of the potential cut can be checked. If $|sign(c_1) + sign(c_2)| > K$ (where $|sign(c_n)|$ denotes the number of 1s in $sign(c_n)$), then the resulting cut is not k-feasible, and its creation can be avoided.

To exemplify these statements, consider the following example with $M = 8$ and the cuts $c_1 = \{12, 19, 33\}$, which has $sign(c_1) = 00011010$, and $c_2 = \{12, 19, 25\}$, which has $sign(c_2) = 00011010$. Although $c_1 \neq c_2$, these cuts have the same signature

00011010, then their leaves must be tested.

Considering a third cut $c_3 = \{12, 19, 24\}$, which has $sign(c_3) = 00011001$, it is seen that neither c_1 dominates c_3 nor c_3 dominates c_1, this is done without having to compare their leaves. Lastly, using $K = 3$, the $c_1 \bowtie c_3$, Equation 5.1, does not generate a k-feasible cut because $(sign(c_1) + sign(c_3)) = 00011011$, and $|00011011| > K$.

5.2.2 *KL*-cuts

The work of (MARTINELLO et al., 2010) introduces the KL-feasible cuts concept applied in AIGs. *KL*-cut is an extension of traditional *K*-cut, where it does not impose a limit of a single output for cutting. KL cuts can define regions within a subject graph, each with multiple outputs.

This feature makes possible a reduction in the total number of cuts required to cover the graph, as traditional cuts with a single output may necessitate the use of multiple cuts to encompass the same area. KL cuts, by allowing multiple outputs in the cut, offer the advantage of isolating subregions in the graph more efficiently, minimizing the overall number of cuts needed to cover the portions of the graph.

In their work, (MACHADO et al., 2013) introduces an extension of *KL*-cuts aimed at mapped circuits. Compared to *KL*-cuts in AIGs, enumerating *KL*-cuts in mapped circuits requires the consideration of two additional cases: *i*) gates with more than two inputs, and *ii*) gates with only one input (inverter). Furthermore, the enumeration procedure proposed by (MACHADO et al., 2013) represents an improved version, enhancing the efficiency of the process.

In the subsequent sections, we will provide a more detailed discussion of the enumeration procedure employed in these two works.

5.2.2.1 Martinello's Enumeration Method

To enumerate the *KL*-cuts, (MARTINELLO et al., 2010) defines the *l-feasible backcuts* term, which is similar to *K*-cuts. In this context, a *backcut* of a node n is a set of nodes c such that every path between n and a PO contains a node in c. A *backcut* represents nodes that are influenced by the node n.

While the enumeration of K cuts follows the AIG topological order which allows obtaining the set of nodes that can generate n, the enumeration of L cuts follows the topo-

logical reverse order. In the following, the equations that define the backcut enumeration process will be presented.

The first equation, 5.11, describes the *combine* process for backcuts, which makes the union of the cuts from two nodes. This process is closely similar to the *combine* procedure of traditional cuts, as detailed in Equation 5.1. However, in this case, it combines the backcuts from its fanouts, and the resulting cut can not have more than L leaves.

$$A \bowtie B = u \cup v \mid u \in A, v \in B, |u \cup v| \leq L \tag{5.11}$$

The second equation, Equation 5.12, elucidates that the *combine* process is applied to all fanouts of the node. The distinction from traditional cuts, employing an AIG as the subject graph, lies in the fact that the *combine* procedure executes for pairs of nodes (the fanins of the node). In contrast, for backcuts, the procedure is executed for 1 until n, where n is the number of fanouts of the node.

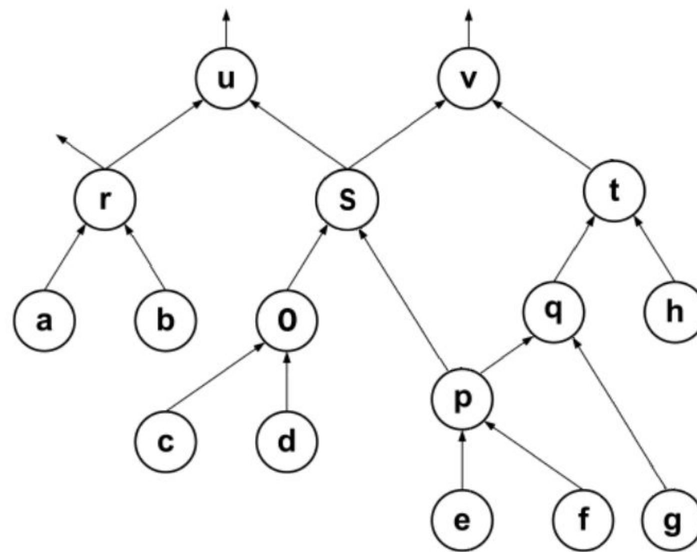
$$\mathop{\bowtie}_{i=m}^{n} x_i = x_m \bowtie \ldots \bowtie x_n \tag{5.12}$$

Lastly, the third equation, Equation 5.13, outlines the *backcut* formulation: If a node is a PO, it possesses only its self-cut. Otherwise, the enumeration process explained in Equation 5.12 is applied to this node.

$$\mathbf{\Phi}_{\mathcal{L}}(n) = \begin{cases} \{\{n\}\} & : \text{n is a PO} \\ \{\{n\}\} \cup (\bowtie_i \mathbf{\Phi}_{\mathcal{L}}(n_i)) & : \text{otherwise} \end{cases} \tag{5.13}$$

To illustrate the backcut enumeration process, Figure 5.1 serves as an example. The backcuts are generated following the rules described in Equation 5.13, and the resulting backcuts for each node are presented in Table 5.2. The highlighted values will be used in the algorithm explanation.

Figure 5.1 – Backcut example.



Source: Adapted from (MARTINELLO et al., 2010).

Table 5.2 – K-cuts and Backcuts of Figure 5.1.

Node	Kcut	Backcut
a	{a}	{a}, {r}
b	{b}	{b}, {r}
c	{c}	{c}, {o}, {s}, {u,v}
d	{d}	{d}, {o}, {s}, {u,v}
e	{e}	{e}, {p}, {q,s}, {s,t}, {s,v}, {q,u,v}, {t,u,v}
f	{f}	{f}, {p}, {q,s}, {s,t}, {s,v}, {q,u,v}, {t,u,v}
g	{g}	{g}, {q}, {t}, {v}
h	{h}	{h}, {t}, {v}
o	{o}, {c,d}	{o}, {s}, {u,v}
p	{p}, {e,f}	{p}, **{q,s}**, {s,t}, {s,v}, {q,u,v}, {t,u,v}
q	{q}, {p,g}, **{e,f,g}**	{q},{t}, {v}
r	{r}, {a,b}	{r}
s	{s}, {o,p}, {e,f,o},{c,d,p}, **{c,d,e,f}**	{s}, **{u,v}**
t	{t}, {h,q}, {g,h,p}, {e,f,g,h}	{t}, {v}
u	{u}, {r,s}, {o,p,r}, {e,f,o,r},{c,d,p,r}, {c,d,e,f,r}, **{a,b,s}**, {a,b,o,p}, {a,b,e,f,o}, {a,b,c,d,p}	{u}
v	{v}, {s,t}, **{h,q,s}**, {o,p,t}, {g,h,p,s}, {h,o,p,q}, {g,h,o,p}, {e,f, o,t}, {c,d,p,t}, {e,f,g,h,s}, {e,f,h,o,q}, {e,f,g,h,o}, {c,d,h,p,q}, {c,d,g,h,p}, {c,d,e,f,t}	{v}

The method proposed by (MARTINELLO et al., 2010), depicted in Figure 5.2, is based on these formulations. To enumerate the *KL*-cuts, it must first enumerate the K-feasible cuts and the L-feasible cuts (backcuts), these cuts are shown in Table 5.2. In this example, are enumerated *KL*-cuts with up to five inputs, and no more than three outputs ($k = 5$ and $l = 3$).

Figure 5.2 – Martinello's algorithm.

```
1. compute_klcuts(k, l, aig) {
2.   kcuts = compute_kcuts(aig, k)
3.   lcuts = compute_lcuts(aig, l)
4.   for (each lcut in lcuts) {
5.     p = combine_kcuts(lcut)
6.     for (each pi in p) {
7.       klcut = create_klcut(pi, lcut)
8.       if (check_and_fix(klcut))
9.         klcuts.add(klcut)
10.    }
11.  }
12.  return klcut
13.}
```

Source: (MARTINELLO et al., 2010).

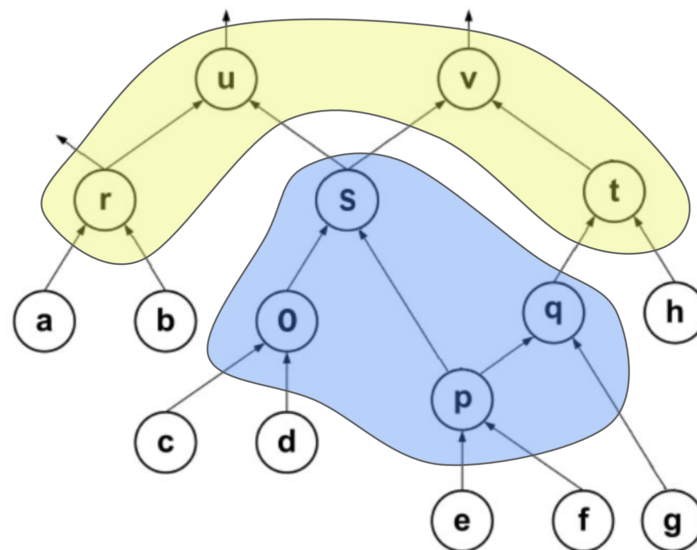
The function *combine_kcuts()*, line 5, is executed for each backcut $lcuts$. Let d be the current processed backcut, and d_i its leaves, the *combine_kcuts()* function computes $p = \bowtie_i \mathbf{\Phi}_{\mathcal{K}}(d_i)$. That is, the *K*-cuts from the nodes that compose its leaves are combined. The objective is to produce new *K*-cuts (p) that will be used to compose the *KL-cuts*.

In the next step, for each generated *K*-cut p_i from p, the *create_klcut()* function is executed, composing the *KL-cuts* based on each *K*-cut p_i and the *lcut* that generated p. Here, p_i and *lcut* are the inputs and outputs of the kl-cut, respectively.

To be a valid *KL-cut*, the generated *KL-cut* is checked, line 8, as follows: Let G_{kl} be a subgraph defined by *KL-cut*, a node belonging to G_{kl} is added to G_l if it has an output node that does not belong to G_{kl}, a node outside of G_{kl} and if G_{kl} has no more than l nodes. Otherwise, the *KL-cut* is discarded.

In Table 5.2, two backcuts, $\{q, s\}$ and $\{u, v\}$, are highlighted. From the backcut $\{q, s\}$, we observe that the *K*-cut $\{e, f, g\}$ from node q can be combined with the *K*-cut c, d, e, f from node s, generating the *K*-cut $\{c, d, e, f, g\}$. The *KL*-cut G_{KL}, described as $\{G_K, G_L\}$, generated by the function is $\{\{c, d, e, f, g\}, \{q, s\}\}$. Applying the same procedure to the backcut $\{u, v\}$, it generates the *K*-cut $\{a, b, h, q, s\}$ from nodes u and v, resulting in the *KL*-cut $\{\{a, b, h, q, s\}, \{u, v\}\}$. In this way, the *KL*-cuts illustrated in Figure 5.3 are obtained.

Figure 5.3 – Martinello's method result.



Source: Adapted from (MARTINELLO et al., 2010).

The approach used by (MARTINELLO et al., 2010) is costly concerning execution time due to having to: 1) enumerate the K-cuts in topological order, 2) enumerate the L-cuts (backcuts), and 3) combine the K-cuts from each l-cut leave.

5.2.2.2 Machado's Enumeration Method

Similar to the approach presented by (MARTINELLO et al., 2010), the method proposed by (MACHADO et al., 2012) relies on K-cuts for enumerating KL-cuts. However, since (MACHADO et al., 2012) is adapted for enumerating KL-cuts in mapping circuits, the method considers additional cases involving nodes with only one or more than two inputs. This is in contrast to K-cuts in AIGs, where each node always has exactly two inputs. The K-cut enumeration employed in this approach is defined by Equation 5.14, which addresses three different types of nodes:

1. When a node is a PI, it has only its self-cut, similar to AIGs;
2. If a node has only one input, its cut set is determined by the cuts of its input, to consider inverters in the circuit;
3. Otherwise, the cuts of a node are the union of its self-cut with the combination of cuts from all its inputs.

$$\Phi_{\mathcal{K}}(n) = \begin{cases} \{\{n\}\} & : \text{n is a PI} \\ \Phi_{\mathcal{K}}(n_1) & : \text{g = 1} \\ \{n\} \cup \{\Phi_{\mathcal{K}}(n_1) \bowtie \ldots \bowtie \Phi_{\mathcal{K}}(n_g)\} & : \text{otherwise} \end{cases} \tag{5.14}$$

This equation can also be applied to AIGs. In the case of a PI in an AIG, the cut computation remains unchanged. For other nodes in the AIG, which always have two inputs, the cuts are computed according to the 'otherwise' case in the equation.

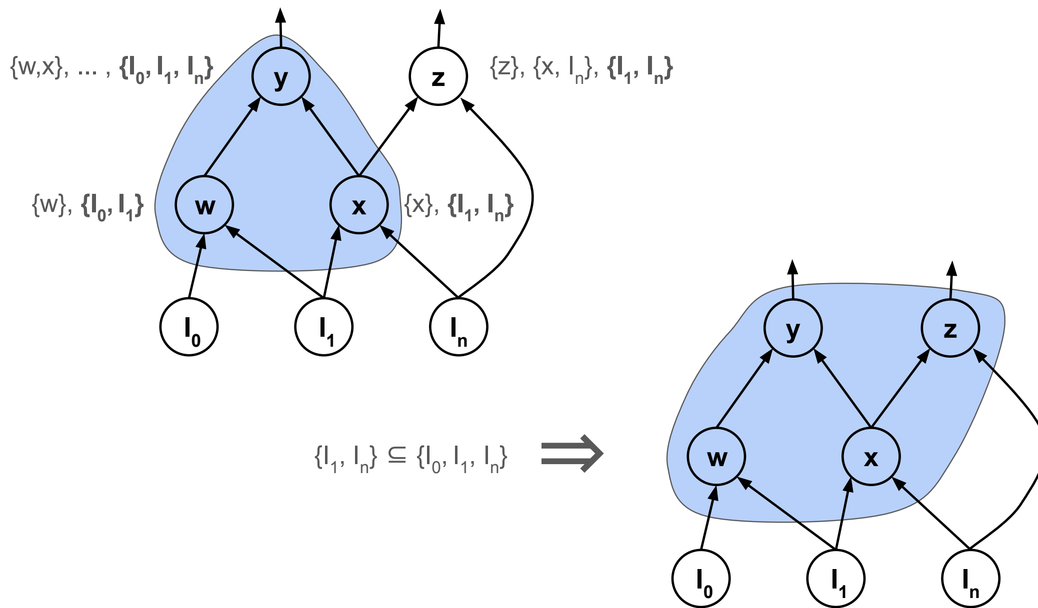
Before going into the explanation of the (MACHADO et al., 2012) method, Figure 5.5, it is important to understand its auxiliary functions. Firstly, the $KCutsOK(node)$ function is responsible for checking whether the node $node$ has at least one *K*-cut formed only by $kcut$ inputs. If the condition is met, this function returns $true$; otherwise, it will return $false$.

The second auxiliary function is $addInsts(node, insts, outputs)$. Here, $node$ represents the node being verified, $insts$ is used to store the internal nodes, and $outputs$ is used to store the outputs of the *KL*-cut, respectively. This function traverses the circuit, starting from each $kcut$ leave, and stops if a PO is reached or the $KCutsOK(node)$ function returns $false$. This function determines whether a node should be classified as an internal or an output node for the *KL*-cut being created.

Figure 5.4 illustrates how the $addInsts()$ function works. The AIG used in this example is only for didactic purposes. Let c_1 be a *K*-cut to be processed, defined as $\{\{I_0, I_1, I_n\}, \mathbf{y}\}$, with inputs $\{I_0, I_1, I_n\}$ and output y. The $addInsts()$ function starts from the inputs of c_1 and traverses the nodes w, y, and x. These three nodes satisfy the $KCutsOK()$ condition with the *K*-cut $\{\{I_0, I_1\}, \{c_1\}, \{I_1, I_n\}\}$, in this order. As a result, these three nodes are stored as instance nodes. Additionally, node y is stored as an output for the *KL*-cut, as its fanout is a PO.

Furthermore, since node x satisfies the $KCutsOK()$ condition, node z must also be verified. Node z has the *K*-cut $\{I_1, I_n\}$, and since $\{I_1, I_n\} \subseteq c_1$, z can be added as an instance node. As it is also a PO, it is included as an output node.

Figure 5.4 – $addInsts()$ function illustration.



Source: Adapted from (MACHADO et al., 2012).

Thus, for each K-cut, the method proposed by (MACHADO et al., 2012) uses the auxiliary function $addInsts()$ to traverse the circuit from its leaves. The idea is to ensure that each cut k is fully expanded in the circuit. Applying the function to the AIG in Figure 5.1, a solution that can be generated is presented in Figure 5.6.

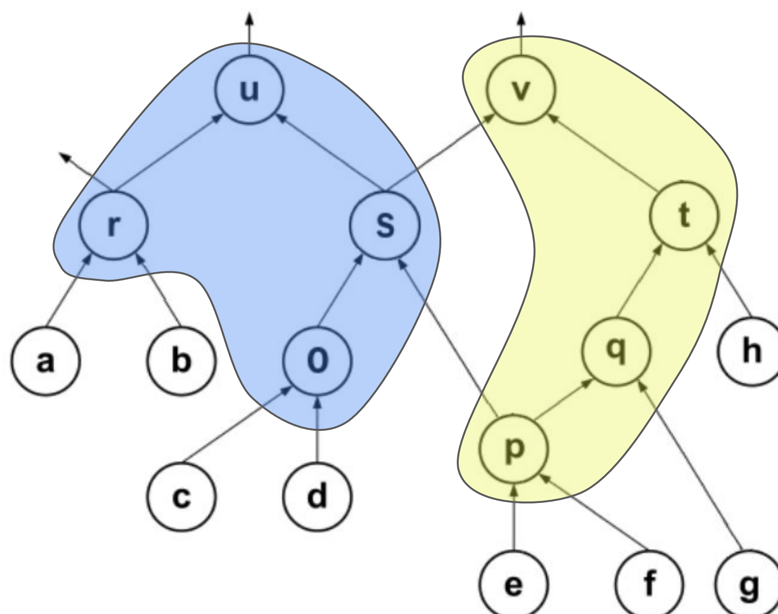
Figure 5.5 – Machado's algorithm.

```
01. compute_klcuts(k, circuit) {
02.   kcuts = compute_kcuts(circuit, k)
03.   for each kcut in kcuts do {
04.     insts <- ø
05.     outputs <- ø
06.     for each node in kcut do {
07.       addInsts(node, insts, outputs)
08.     }
09.     klcut = createKLcut(kcut, insts, outputs)
10.     klcuts.add(klcut)
11.   }
12.   return klcuts
13. }
14. addInsts(node, insts, outputs) {
15.   if KCutsOK(node) and node is not PO then {
16.     insts.add(node.inst)
17.     for each out in node.inst.outputs do {
18.       addInsts(out, insts, outputs)
19.     }
20.   } else {
21.     outputs.add(node)
22.   }
23. }
```

Source: (MACHADO et al., 2012).

In this example, the *K*-cuts $\{a, b, c, d, p\}$ and $\{e, f, g, h, s\}$ have not been expanded in terms of instance nodes, they already contain all instances, but their outputs are expanded. Specifically, the *K*-cut $\{\{a, b, c, d, p\}, u\}$ generates the *KL*-cut $\{\{a, b, c, d, p\}, \{u, r, s\}\}$, and the *K*-cut $\{\{e, f, g, h, s\}, v\}$ generates the *KL*-cut $\{\{e, f, g, h, s\}, \{v, p\}\}$.

Figure 5.6 – Machado's algorithm result.



Source: Adapted from (MACHADO et al., 2012).

Similar to the method of (MARTINELLO et al., 2010), the method of (MACHADO et al., 2012) may also be inefficient in terms of runtime, as the approach involves a massive number of checks for each node processed during the expansion of each cut. Recently, a new work that enumerates mult-output cuts was proposed, which uses a different approach from the previously presented methods and can enumerate the cuts more quickly. In the following, we will present this method in detail.

5.2.2.2.1 IWLS23 Best Paper Method

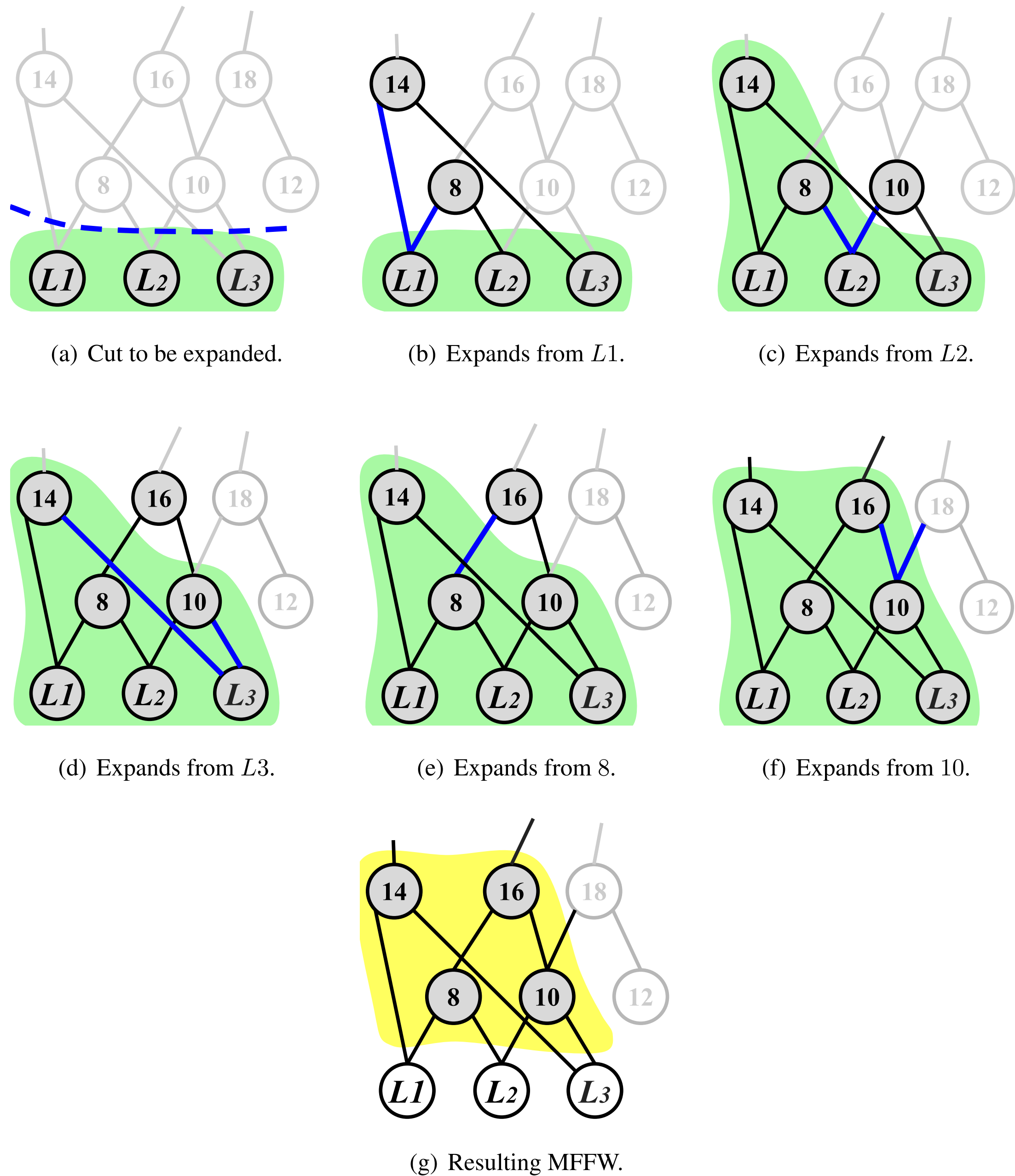
The method proposed by (TANG et al., 2023) expands a K-cut to its MFFW. This means that all the nodes of the circuit that are expressed by a given K-cut, a set of inputs, comprise the MFFW of the cut. The approach proposed by (TANG et al., 2023) begins the expansion from the leafs of the K-cut, expanding the MFFW in topological order by advancing the fanouts of the nodes that compose the MFFW. This process is illustrated in Figure 5.7.

Figure 5.7(a) illustrates the K-cut $\{L1, L2, L3\}$ to be expanded, where L_i refers to the leafs of the cut. The area highlighted in green is the region of the circuit where the nodes to be analyzed must have their fanin within to be part of the MFFW, as will be discussed later.

Next, Figure 5.7(b) illustrates the expansion of the MFFW to the fanouts of node $L1$, where these fanout edges are highlighted in blue. The nodes analyzed in this process are nodes 8 and 14, and as can be seen, both have their other fanin within the expanded region, thus they are added to the MFFW. The same process is applied to nodes $L2$ and $L3$ in Figures 5.7(c) and 5.7(d), respectively. In Figure 5.7(c), node 8 already belongs to the MFFW, and node 10 is added to the MFFW since its other fanin is the node $L3$, which belongs to the MFFW. In Figure 5.7(d), all nodes in the fanout of $L3$ already belong to the MFFW.

Similarly, Figure 5.7(e) illustrates the expansion of the fanout of node 8, which is node 16. Its other fanin is node 10, which belongs to the MFFW, so node 16 is also included in the MFFW.

Figure 5.7 – MFFW expansion.



(a) Cut to be expanded. (b) Expands from $L1$. (c) Expands from $L2$.

(d) Expands from $L3$. (e) Expands from 8. (f) Expands from 10.

(g) Resulting MFFW.

Continuing with the example, Figure 5.7(f) shows the expansion process for node 10, which has fanouts to nodes 16 and 18. Node 16 has already been processed and is part of the MFFW. On the other hand, node 18 has only one of its fanins within the MFFW; its other fanin, node 12, is not part of the MFFW, so node 18 is not included in the MFFW.

Finally, Figure 5.7(g) illustrates the resulting MFFW after the expansion. As discussed in Section 5.2, the leaves do not form part of the cut, and the same applies to the MFFW. Therefore, the resulting MFFW does not include nodes $L1$, $L2$, and $L3$; only nodes 8, 10, 14, and 16 belong to the MFFW in this example. The resulting MFFW is

highlighted in yellow and consists only of its internal nodes.

In their work, (TANG et al., 2023) use an auxiliary method to identify nodes that have both their fanins within the expanding MFFW. This auxiliary method is based on hash tables to optimize the verification process.

5.2.2.3 Optimization Without Coverage

Some approaches work with local optimization without the goal of producing a complete cover of the circuit. In these approaches, a region is isolated and resynthesized, if gains are produced the region is substituted to consolidate the local gains. Examples of this type of approach include (MACHADO et al., 2012) (MACHADO et al., 2013)

5.3 Flow Based on Logic Calculation and Signal Distribution

Recently, an approach for a design flow based on (local) logic computation and signal distribution was proposed (REIS; MATOS, 2018). The goal is to bring physical awareness to the early steps of the design flow, starting with technology-independent logic synthesis. This is done using *KL*-cuts and 1L-cuts for different purposes, as described below.

5.3.1 *KL*-cuts For Logic Calculation

The local logic computation is done inside *KL*-cuts, meaning that the *KL*-cuts will be used to pack independent portions of the circuit that will be placed locally. In this way, the routing among cells inside a same *KL*-cut is local, using short wires and low metal levels. The *KL*-cuts can be viewed as physical partitions.

5.3.2 1L-cuts For Signal Distribution

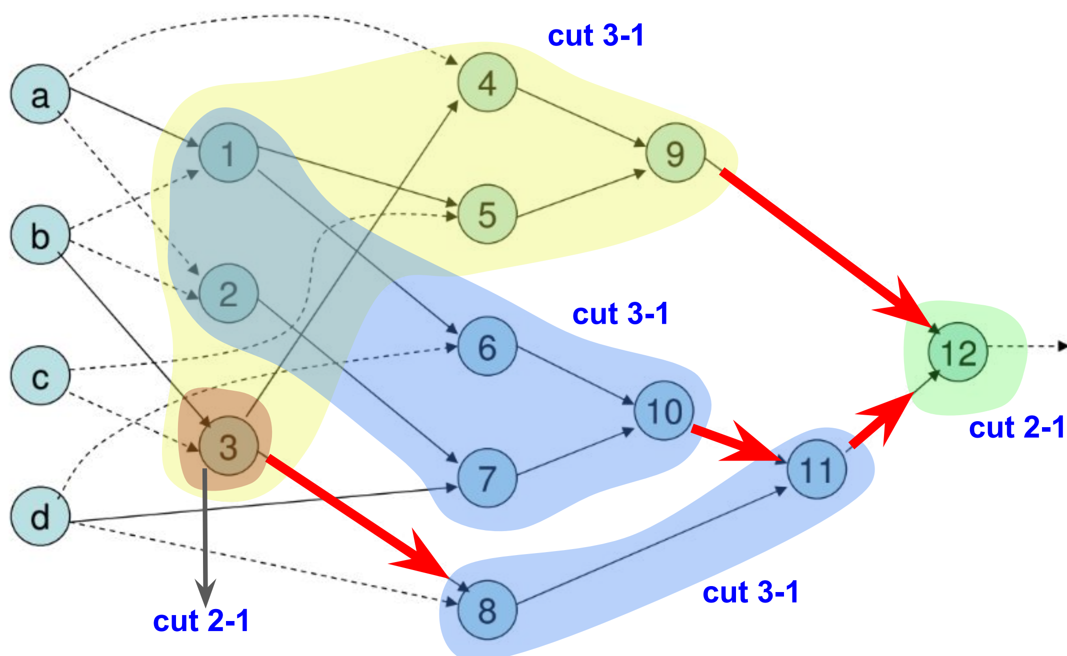
The routing among different *KL*-cuts is done using 1L-cuts to distribute signal. The structure of 1L-cuts tends to be sparse and global, using long wires and high metal levels to connect inverters and buffers to distribute signals from one source to several consumers or sinks.

5.4 Comparative Summary Between Covering With *K*-cuts and *KL*-cuts

Using *KL*-cuts to cover a circuit has some advantages, the first that can be mentioned is the smaller number of cuts needed to cover a circuit when compared to *K*-cuts. To demonstrate this, we will use four different coverage examples for the same AIG example. In the four examples were used cuts with up to three inputs.

Figure 5.8 presents a cover for that AIG, using only *K*-cuts with a single output. To distinguish each cut, we will use the nodes within each cut to describe them. Therefore, in this example, we have the two 2-1 cuts $\{\{b, c\}, 3\}$, and $\{\{9, 11\}, 12\}$, and three 3-1 cuts $\{\{a, b, c\}, 9\}$, $\{\{a, b, d\}, 10\}$, and $\{\{d, 8, 10\}, 11\}$.

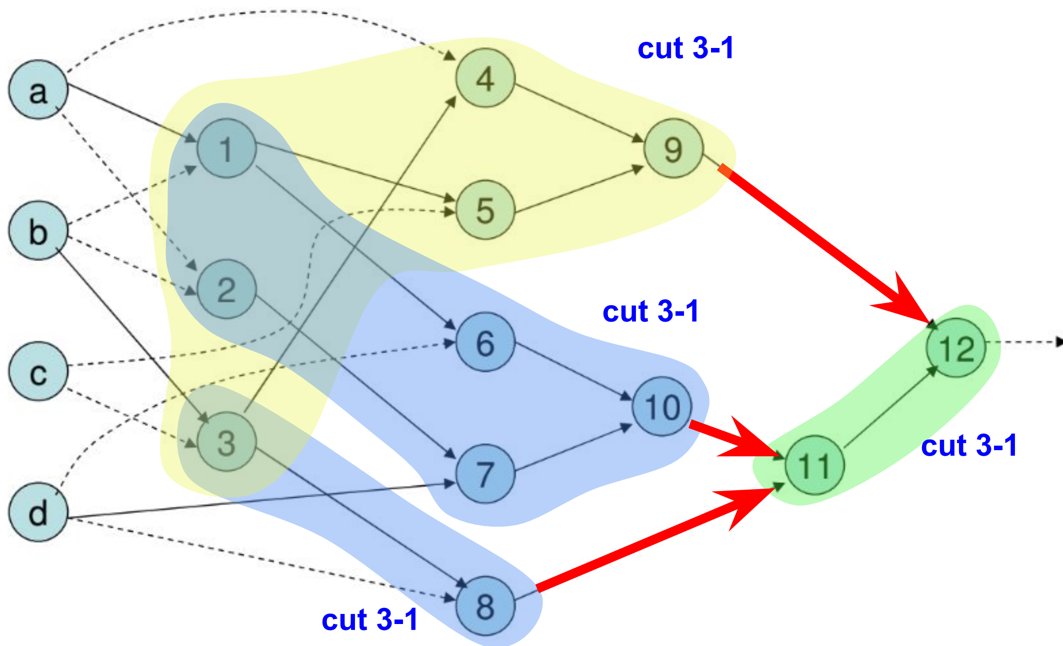
Figure 5.8 – Example of an AIG covered using only *K*-cut, with K equals 3.



As can be seen in Figure 5.8, the fact that the *K*-cuts have the limitation of a single output for each cut, makes it necessary to create a cut that contains only the node 3, which results in a total of five cuts needed to cover the AIG. This cover uses five cuts and has fifteen *AND* nodes since nodes 1, 2, and 3 will be duplicated due to being covered by more than one cut.

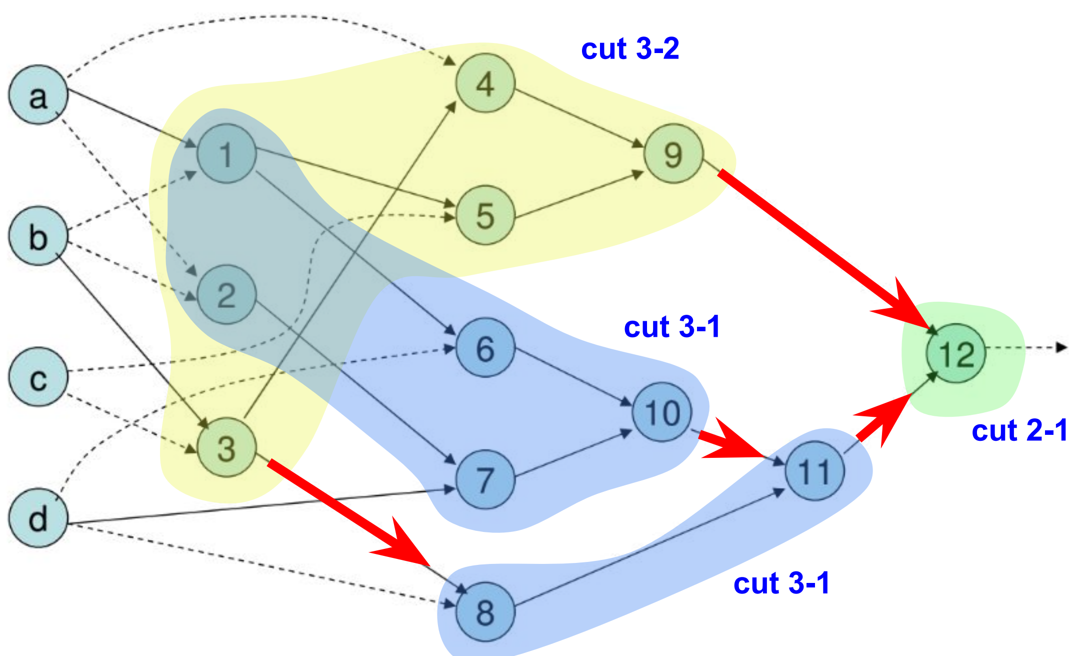
Figure 5.9 shows another possible cover also using *K*-cuts. This coverage uses fewer cuts than the previous cover, only four cuts were used. However, the number of *AND* nodes was not reduced, because node 3 belongs to two different *K*-cuts.

Figure 5.9 – Another cover using K-cuts with k equals 3.



To supply the limitation of only using K-cuts to cover the AIG, which are cuts with a single output, using the KL-cuts, Section 4.2.2, it can obtain the following cover presented in Figure 5.10, where are used KL-cuts with K and L up to 3 and 2, respectively.

Figure 5.10 – Cover example using KL-cuts with K equals 3 e L equals 2.



In this cover, Figure 5.10, only four KL-cuts were used to cover the circuit. It is possible because the KL-cut $\{\{b, c\}, \{3\}\}$ is not needed to be created, as the cut $\{\{a, b, c\}, \{9, 3\}\}$

have the node 3 as an output. Therefore, compared to the covers in Figures 5.8 and 5.9, the cover obtained, Figure 5.10, uses four *KL*-cuts and a total of fourteen *AND* nodes, which can be beneficial for ASICs, for example, as it may reduce the number of cells.

In summary, using *KL*-cuts to cover an AIG allows some improvements. Compared to the solution presented in Figure 5.8, the cover obtained with *KL*-cuts of Figure 5.10 uses fewer cuts, as it is not necessary to create the cut $\{\{b, c\}, \{3\}\}$, just the output of the cut $\{\{a, b, c\}, \{9, 3\}\}$ is used.

And compared to the solution illustrated in Figure 5.9, there are no gains in terms of the number of cuts. However, the cover obtained with *KL*-cuts results in a smaller number of AIG nodes, since node nd_3 is covered by two different *K*-cuts, as illustrated in Figure 5.9. This does not occur in Figure 5.10, due to the *KL*-cuts allowing the creation of multi-output cuts in the circuit.

A summary of the elements of each solution in Figures 5.8, 5.9, and 5.10 is presented in Table 5.3. This table shows a brief comparison of the total elements used in each cover. As you can see, the approach using *KL*-cuts shows to be more efficient in terms of the number of cuts and AND nodes, due to avoid creating a $\{\{b, c\}, \{3\}\}$.

Table 5.3 – Summary of the coverage results using k-cuts and kl-cuts.

Soltion	**Lut Type**				**Elements**			
	2-1	**3-1**	**2-2**	**3-2**	**Cuts**	**In**	**Out**	**Ands**
S1	2	3	0	0	5	13	5	15
S2	0	4	0	0	4	12	4	15
S3	1	2	0	1	4	11	5	14

5.5 Contributions of this chapter

This chapter provides an overview of the different approaches in the literature for enumerating multi-output cuts. The methods of (MARTINELLO et al., 2010) that propose *KL*-cuts are presented, as well as the method of (MACHADO et al., 2012), which is a variation of (MARTINELLO et al., 2010). Additionally, the MFFW method proposed by (TANG et al., 2023), which also deals with multi-output cuts, is also discussed.

6 PROPOSED METHOD

6.1 About This Chapter

This chapter presents the contributions of this work. Essentially, this section is divided into two main parts: the first part focuses on contributions related to cut enumeration, while the second part concentrates on coverage generation for AIG.

In the first part of the chapter, an approach is introduced that not only enumerates *K*-cuts but also identifies trees in the AIG for which special cuts are generated. Following this, the next contribution presented is a novel method to expand *K*-cuts into *KL*-cuts without the need for input-based comparisons.

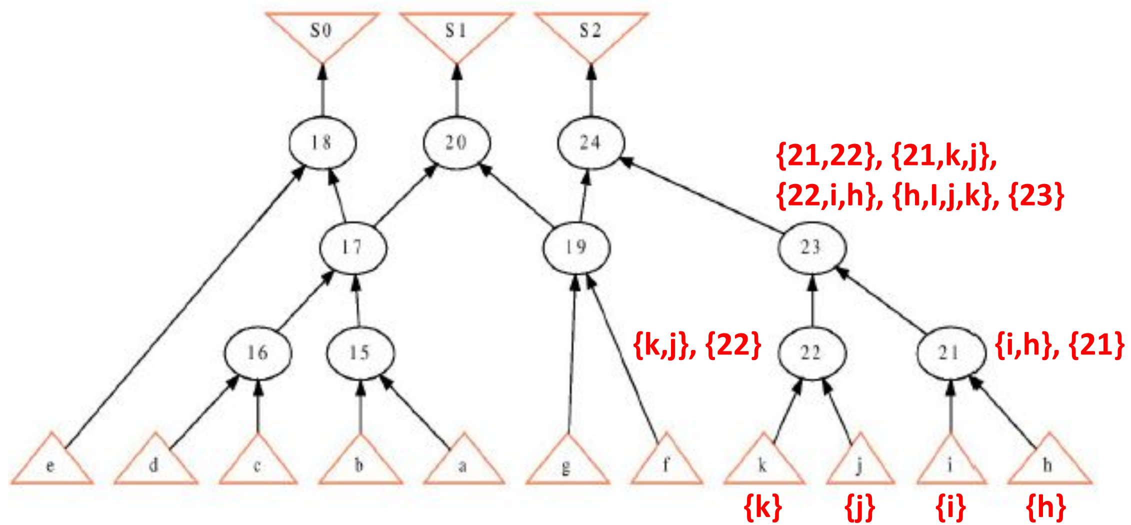
In the second part of the chapter, a method is proposed to achieve coverage of the AIG using *KL*-cuts. This method involves generating a CNF formula that describes the coverage problem constraints and employs SAT solving to obtain the solution.

6.1.1 *K*-cut Enumeration

The *K*-cut enumeration follows the formulation presented in Section 5.2.1.1. The generation of new cuts occurs at the internal nodes of the AIG in topological order; no cuts are generated at the PIs of the AIG. However, to generate the initial cuts and subsequently other cuts, the AIG nodes, including the PIs, must contain their self-cuts. A self-cut is not a cut per se, but it is used to generate other cuts. This process is illustrated in Figure 6.1.

The process is illustrated for only a portion of the AIG. The PIs k, j, i, and h have their self-cuts. Following the topological order, the next nodes to be processed are nodes 21 with fanins h and i, which, by combining the self-cuts of these fanins, obtain the cut $\{i, h\}$, and node 22 with fanins k and j, which, by combining the self-cuts of these fanins, obtain the cut $\{k, j\}$. The self-cuts of these nodes are also listed to generate new cuts with these nodes as inputs.

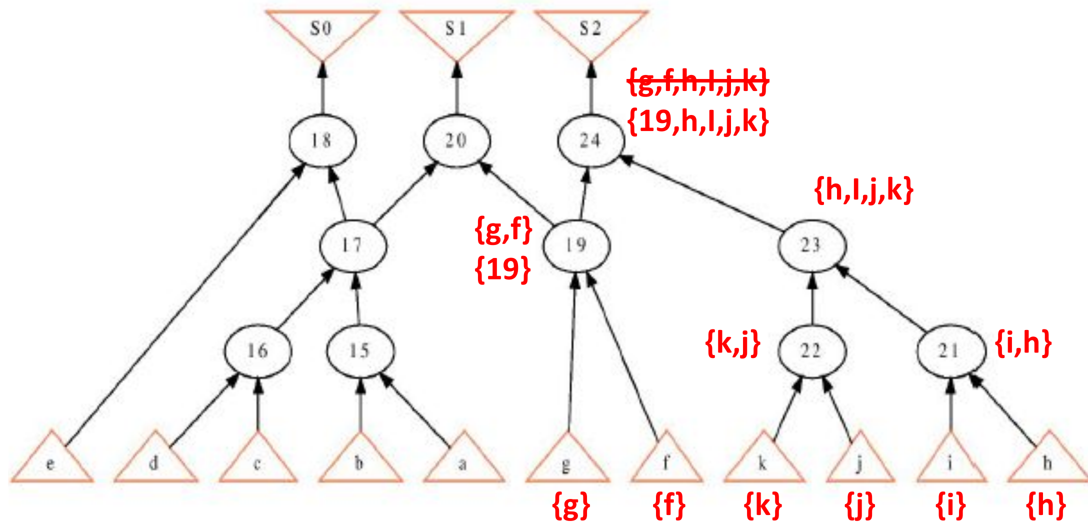
Figure 6.1 – *K*-cut enumeration example.



Continuing the process, node 23 obtains four cuts, derived from the combination of cuts from fanins 21 and 22, in addition to the self-cut of node 23. The process is executed for all nodes in the AIG. In this illustration, we are generating cuts for a portion of the AIG consisting only of nodes with $|fanout| = 1$. This region of the circuit is a tree, meaning other regions of the AIG will not use all logic produced within this region. Therefore, we can use an approach to enumerate AIG cuts capable of identifying trees within the AIG.

The idea behind tree cuts is that self-cuts are not generated within trees, significantly reducing the total number of cuts, as illustrated in Figure 6.2. Here, nodes 21 and 22 are classified as internal to a tree and therefore do not have self-cuts, reducing the total cuts generated at node 23 from four (self-cuts not counted) to just one cut. Consequently, node 23 is also internal to a tree and has no self-cut listed. Figure 6.2 also illustrates the cuts generated for the 19 which has fanout greater than 1, therefore node 19 has its self-cut $\{19\}$.

Figure 6.2 – K-cut enumeration example.



Continuing with the example illustrated in Figure 6.2, considering K-cuts with $K = 3$, node 23 has the cut $\{h, i, j, k\}$, which is not discarded because it is a tree-type cut that can exceed K to obtain the cut of the complete tree region. In turn, at node 24, two cuts are generated: $\{g, f, h, i, j, k\}$ and $\{19, h, i, j, k\}$. However, the cut $\{g, f, h, i, j, k\}$ is discarded because it does not correspond to a tree in the AIG and its size exceeds K. On the other hand, the cut $\{19, h, i, j, k\}$ is maintained as it represents a tree region in the AIG.

With this approach, it is possible to reduce the number of enumerated cuts by avoiding the generation of cuts within the tree regions of the AIG. However, since the trees have more inputs than the value K, this approach cannot be used for mapping cuts to cells with up to K inputs. Nevertheless, for other applications such as synthesis within the cuts, this approach can be beneficial, as trees can be completely isolated.

6.1.2 Cut Signature

The first proposal of this work is the implementation of a method for generating cut signatures for AIG. Each AIG node is identified using a 32-bit unsigned integer value (C++ type uint32_t), so the used AIG is able to index up to a maximum of 2^{32} AIG nodes, which gives a total of 4.294.967.296 nodes.

In the case of cutting signatures, each signature is expected to be unique. That is, for each cut in the AIG it is expected that a different signature will be obtained from the

other cuts, this is the same problem as unique keys in Hash tables, where the cut is the value that will be stored in the hash table and the cut signature is key. However, defining a method capable of generating non-duplicate keys, without compromising performance, is almost impossible.

In this work, it is considered that each cut in the AIG has 6 inputs. An approach capable of generating unique signatures could be to convert each input cut to a string value and concatenate them into single values, as shown in the equation 6.1 where the cut inputs are from in_0 to in_5 which are originally integer values that are converted to a string value to compose the signature.

$$\text{signature} = \text{(in_0)} \,||\, \text{(in_1)} \,||\, \text{(in_2)} \,||\, \text{(in_3)} \,||\, \text{(in_4)} \,||\, \text{(in_5)} \tag{6.1}$$

This approach ensures that each signature is unique. However, considering that an unsigned integer value (uint32_t) has a maximum value of 4.294.967.296, when converted to a string it requires ten characters to be represented, and that each character uses 8 bits, in the worst case each value of input will be an 80-bit value. Furthermore, each signature has 6 entries, which generates a total of 480 bits to store each signature.

Therefore, in this work, a method is proposed to generate the signatures of the cuts using the input values of each cut as integer values. The idea is to apply rotations based on prime numbers to each of the inputs in order to try to get a better dispersion and perform the bitwise XOR operation to compose the final key, as described by the equation 6.2.

$$\begin{aligned} \text{signature} = (\text{in_0} \ll 0) \oplus (\text{in_1} \ll 7) \oplus (\text{in_2} \ll 13) \oplus \\ (\text{in_3} \ll 19) \oplus (\text{in_4} \ll 23) \oplus (\text{in_5} \ll 29) \end{aligned} \tag{6.2}$$

However, as expected, this method does not guarantee collision-free signatures. Therefore, collisions are managed by the C++ hash library.

6.2 Expanding *K*-cuts to *KL*-cuts

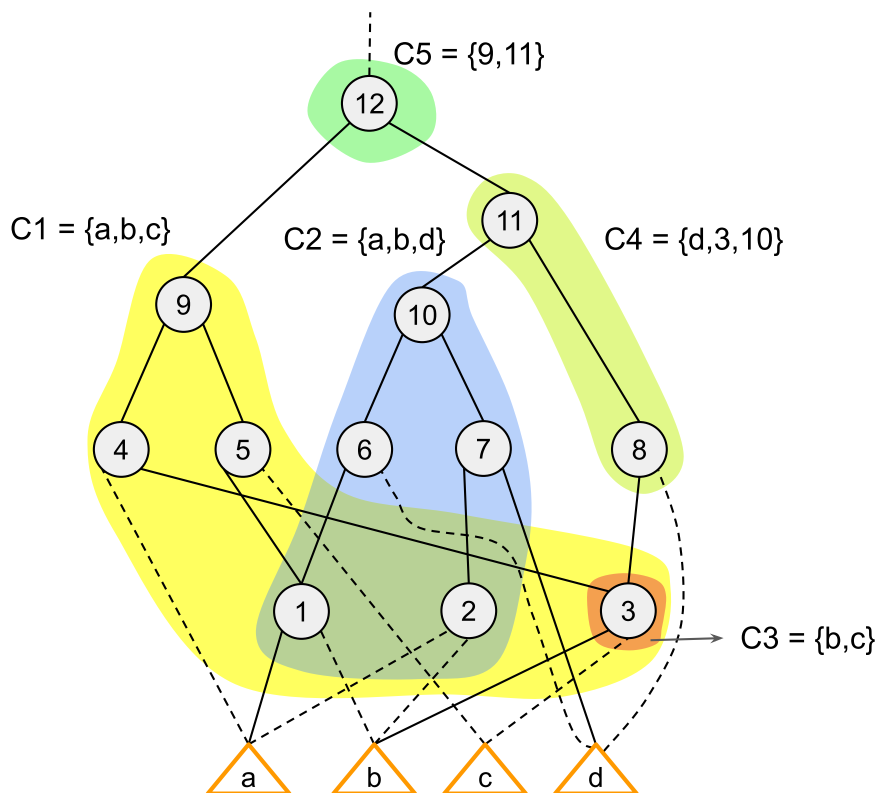
The existent methods to enumerate *KL*-cuts rely on exhaustively comparing the leafs of the cuts. The method of (MARTINELLO et al., 2010) uses the backcuts, discussed in Section 5.2.2.1, to combine *K*-cuts to create new cuts with more than one output. The method of (MACHADO et al., 2012) enumerates the *KL*-cuts by expanding each *K*-cut,

in the subject graph, until it reaches a POs or a node that has no cut dominated by the K-cut is being expanded. To do this, it is necessary to check the leafs of every cut in each node processed.

Given that comparing each leaf in every cut is an expensive task, the approach proposed in this work consists of reducing comparing cut's leafs to enumerate the *KL*-cuts. Instead of comparing the leafs of the cuts to identify the relations between the cuts, we use the Cut signature, which provides a unique identifier for each cut. With this, it is possible to easily determine whether a cut is associated with a node in the subject graph. In our work, the AIG is used as the subject graph.

To illustrate the procedure executed by our method, the AIG presented in Figure 6.3 will be used. In this AIG are shown five K-cuts, C_1, C_2, C_3,C_4, and C_5. In this example, we are using the cut signatures C_1, C_2, C_3, C_4, and C_5 only to provide an easy association of the signature with the cut, however, in the real implementation integer identifiers are used for it.

Figure 6.3 – Example of cuts K generated with K=3 for a given AIG.



The proposed method to enumerate the *KL*-cuts is shown in Algorithm 6.1. This algorithm provides an overview of our method, which is composed of three steps, which are 1) to prepare the *KL*-cuts (line 2), 2) to execute the forward of the cuts (line 3), and

last the outputs are added to each cut (line 4), composing the *KL*-cuts.

Algorithm 6.1: KL-Cuts enumeration Algorithm.

```
1 kl-cuts_list klEnumeration(aig, k, l, n_priority)
2     all_kl_cuts ← prepare_kls(aig, k, n_priority)
3     forward(aig)
4     define_output(aig, l, all_kl_cuts)
5     return all_kl_cuts
```

In the first step of our method, the *K*-cuts are enumerated selecting only the $n_priority$ cuts at each node, as described in Algorithm 6.2.

Algorithm 6.2: Prepare KL-Cuts Algorithm.

```
1 kl-cuts_list prepare_kls(aig, k, n_priority)
2     all_kl_cuts ← ∅;
3     k_cuts ← enumerate_k_cuts(k, n_priority);
4     for ( each k_cut in k_cuts ) do
5         if ( k_cut ≠ self-cut ) then
6             kl_cut ← createKL(k_cut.leaves);
7             all_kl_cuts[k_cut.sign] ← kl_cut; // map structure
8             mark_support(aig, k_cut);
9     return all_kl_cuts
```

The cuts of Figure 6.3 were enumerated with $enumerate_k_cuts()$ method using the area as a criterion in the Priority cuts. The priority cuts of each node are shown in Table 6.1, where the best two cuts are stored at each node. The first column represents the identifier of each node, the second column provides a list of the *K*-cuts stored at that node, and the last column shows the cost of each cut, in the same order that the *K*-cut appears in the list.

Once the *K*-cuts are already enumerated and each one has its signature, the next step consists of creating the basis *KL*-cuts, which is exactly a copy in terms of the leaf nodes (cut inputs) and the signature of each *K*-cut, this procedure is described in line 6 and 7 in Algorithm 6.2. As our method uses cut signature, each cut can be identified with a simple value, and any verification can be easily performed with sets of integer values.

Table 6.1 – Cortes K de três entradas, e seus custos, do AIG da Figura 6.3.

Node	K-cut	Cost
a	{a}	0
b	{b}	0
c	{c}	0
d	{d}	0
1	{a, b}, {1}	0 e 0,5
2	{a, b}, {2}	0 e 1
3	**{b, c}, {3}**	0 e 0,5
4	{a, b, c}, {a, 3}	0 e 0,5
5	{a, b, c}, {c, 1}	0 e 0,5
6	{a, b, d}, {d, 1}	0 e 0,5
7	{a,b,d}, {7}	0 e 1
8	{b, c, d}, {d, 3}	0 e 0,5
9	**{a, b, c}, {9}**	0 e 1
10	**{a, b, d}, {10}**	0 e 1
11	**{d, 3, 10}, {11}**	1,5 e 2,5
12	**{9, 11}, {12}**	3,5 e 4,5

In the following algorithms, it is important to highlight two points: firstly, only the cut signatures are utilized in the procedures, not the cut itself; and secondly, the manipulations executed in the auxiliary methods on the AIG are reflected in the original AIG.

The next step consists of the registry of the cut's support. This procedure is executed with the $mark_support()$ method in line 8 in Algorithm 6.2, and its implementation is shown in Algorithm 6.3.

Algorithm 6.3: Mark Support Algorithm.

```
1 void mark_support (aig, k_cut)
2     for ( each leave of k_cut ) do
3         node ← aig.node[leave]
4         if ( k_cut.sign ∉ node.support ) then
5             node.sup.add(k_cut.sign)
```

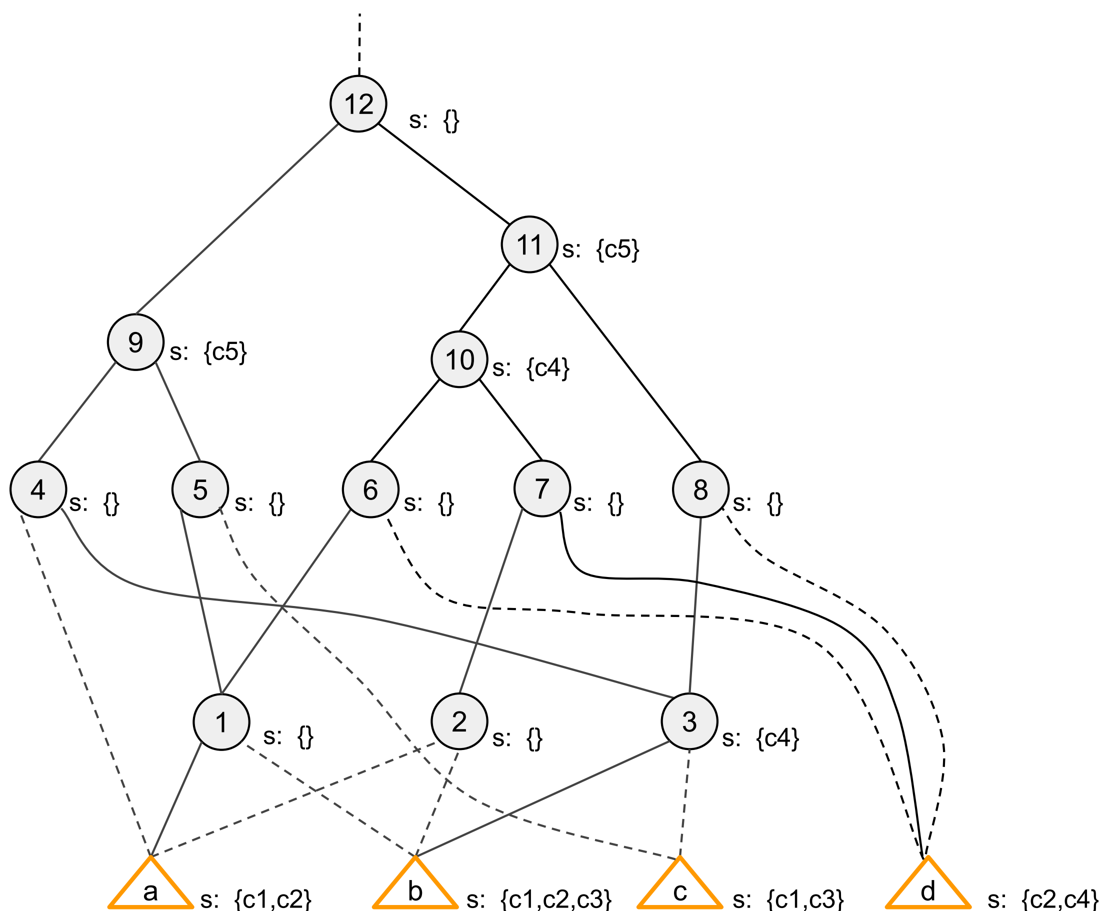
The $mark_support()$ method, Algorithm 6.3, is responsible for traversing the leafs $\{l_1, l_2, ..., l_k\}$ of the cut C_1 and adding C_1 to the "support" list of the nodes $l_1, l_2, ..., l_k$ (*node[l_n].support.add(cut)*). Self-cuts are not considered to be expanded to a *KL*-cut because they are an artifice used to generate cuts in the AIG context because there is no

cut with only one input in the AIG. Thus, the self-cuts are discarded.

For example, in Figure 6.3 there are the cuts C_1, C_2, C_3, C_4, and C_5, and when executing the $mark_support()$ method for each of them, we have for $C_1 = \{a, b, c\}$ that the nodes a, b, and c are leaf of this cut, therefore, C_1 is added to their support list indicating that the nodes a, b, and c are inputs of the cut C_1. The same process is made for the other cuts $C_2 = \{a, b, d\}$, $C_3 = \{b, c\}$, $C_4 = \{d, 3, 10\}$, and $C_5 = \{9, 11\}$.

The result of this procedure is shown in Figure 6.4, where the set s represents the $support$ list. Looking at the *support* list of each node, one can identify which node is an input of each cut. For example, looking only at AND nodes, the nodes 3 and 10 are inputs of the cut $C4$, while the nodes 9 and 11 are inputs of the cut $C5$.

Figure 6.4 – Result of the $markSupport()$ method at a piece of the circuit.



After composing the support of each cut, the propagation of the registered cuts at the support list of the nodes is performed. The cut propagation, Algorithm 6.4, is the core of this work, this consists of, for each AIG node, the intersection of its fanins $support$ list is calculated. The intersection of the fanins $support$ list provides a set of cuts in which

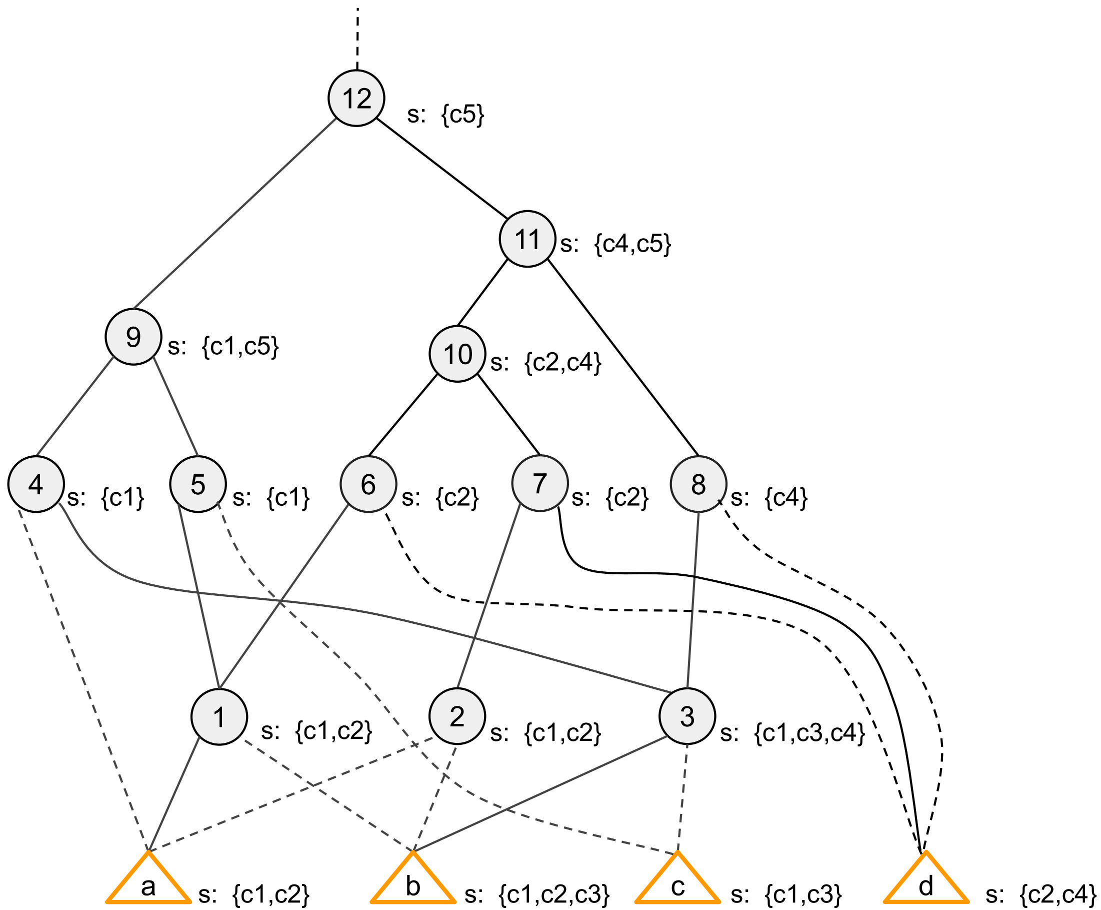
the current node is contained. Thus, the intersection goal is to propagate the cuts for all nodes involved with those cuts.

Algorithm 6.4: Forward KL-Cuts Algorithm.

```
1 void forward(aig)
2     // propagates the kl-cut
3     for ( each aig node ≠ PI ) do
4         node.sup ← node.sup ∪ (node.f1.sup ∩ node.f2.sup);
```

This intersection generates the sets illustrated in Figure 6.5. Also, the lists of cuts generated by this process are shown in Table 6.2, which demonstrates the cuts associated with each node in the AIG.

Figure 6.5 – Intersection of the fanins support list.



In the last step of the method, the outputs of *KL*-cuts are identified and inserted into them. The procedure for determining the output of the cuts is described in Algorithm 6.5. A nd node is just an output of a *KL*-cut C_{kl} in two cases: 1) if a $nd \in POs$, or 2)

Table 6.2 – Result of the Methods mark_support() and forward() in the AIG 6.3.

Node	markSupport()	Fanin1 ∩ Fanin2	∪
a	C1, C2	-	-
b	C1, C2, C3	-	-
c	C1, C3	-	-
d	C2, C4	-	-
1	-	C1, C2	C1, C2
2	-	C1, C2	C1, C2
3	C4	C1, C3	C1, C3, C4
4	-	C1	C1
5	-	C1	C1
6	-	C2	C2
7	-	C2	C2
8	-	C4	C4
9	C5	C1	C1, C5
10	C4	C2	C2, C4
11	C5	C4	C4, C5
12	-	C5	C5

whether at least a fanout node of nd does not have C_{kl} in its "support" list (both in line 4). This procedure is executed for each *KL*-cut in nd, and only those *KL*-cuts that meet this requirement have the node nd added as output.

Algorithm 6.5: Define KL-Cuts outputs Algorithm.

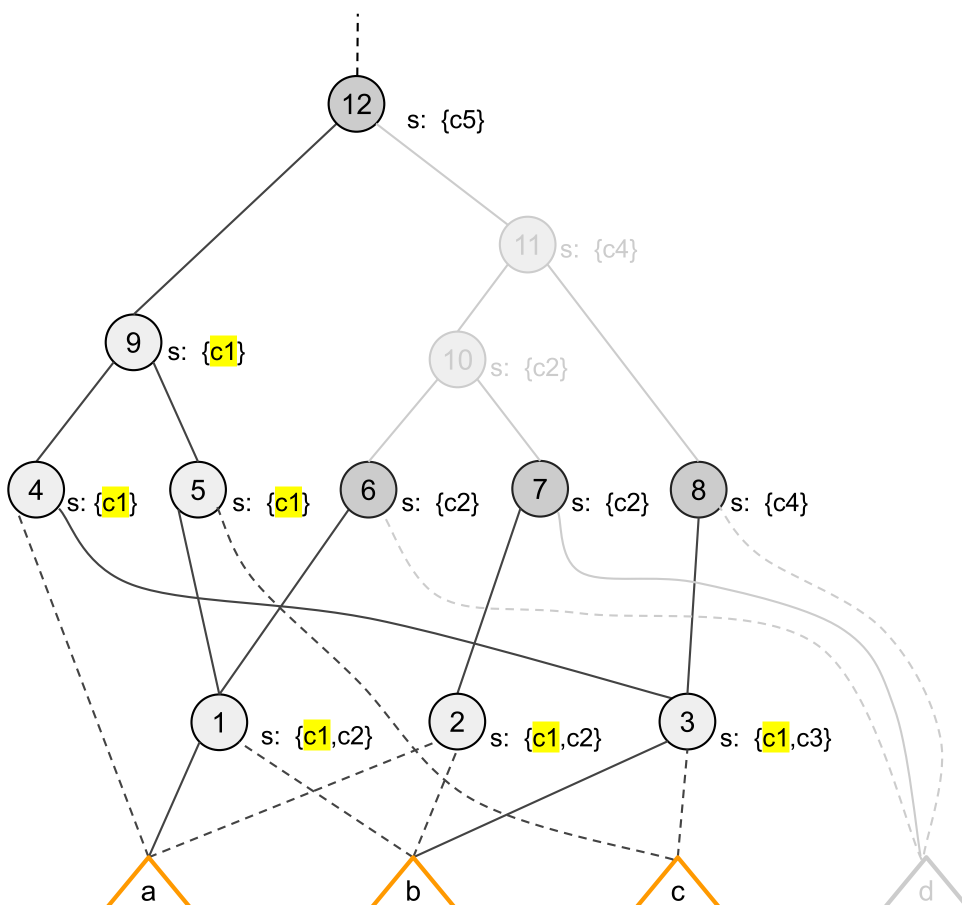
```
1  void define_output (aig, l, all_kl_cuts)
2    for ( each aig node ≠ PI ) do
3      for ( each kl_cut_sign of node.sup ) do
4        if ( (node ∈ POs) or (kl_cut_sign ∉ node.fanouts.sup) ) then
5          kl_cut ← all_kl_cuts[kl_cut_sign]
6          if ( kl_cut.outputs.size() < l ) then
7            all_kl_cuts[kl_cut_sign].outputs.add(node);
8          else
9            // exceeds l outputs
10           all_kl_cuts.remove(kl_cut_sign);
```

In this process, only the *KL*-cuts resulting from the intersection, as listed in the second column of Table 6.2, are considered, because the intersection of cuts from the fanins provides the list of those cuts that the node is an internal node. Therefore, given two nodes nd_i and nd_j, where nd_j is fanout of nd_i if a cut C_n is found in both $support$

lists of nd_i and nd_j, then both nodes are internal to C_n and nd_i is not an output of C_n. However, for a third node nd_y, if C_n is not found in the $support$ list of nd_y, then the node nd_y is not internal to C_n, in this case, nd_i is an output of C_n.

Figure 6.6 illustrates the output identification for the cut C_1. In this example, the white nodes have the cut C_1 in their $support$ list, while the gray nodes do not have C_1. Considering in this example nd_i, nd_j, and nd_y as the nodes 3, 4, and 8 respectively, and $C_n = C_1$, we have that the cut C_1 is found in both $support$ lists of nodes 3 and 4, as highlighted in yellow. However, C_1 is not found in the $support$ list of node 8, therefore node 3 is an output of the cut C_1. The same can seen for the nodes 2 with 7, 1 with 6, and 9 with 12.

Figure 6.6 – Identifying the outputs of the cut C_1.



The Algorithm 6.1 presented in this section is a simplified version of our method, Appendix A shows the complete method considering some optimization during the enumeration of the *KL*-cuts.

6.3 Covering an AIG with *KL*-cuts

The process to generate a valid cover for the AIG must adhere to the requisites described in Section 3.4.2. Essentially, for the cover to be valid, each node in the AIG must be covered by at least one cut, and every cut in the final cover must have its leaf nodes connected to a PI or an output from another cut.

The procedure for obtaining a cover of an AIG, as discussed in (POSSANI, 2019) and (MANOHARARAJAH; BROWN; VRANESIC, 2006), involves selecting cuts from the POs of the AIG and expanding this procedure to the nodes that are input to the selected cuts. This process selects a set of cuts that produce the primary outputs of the AIG, as well as cuts that generate the signals consumed by the selected cuts. While this is an overview, the procedure is more sophisticated and may include refinements to achieve better covers.

Obtaining the cover of an AIG using *KL*-cuts can be more complex than using *K*-cuts. This complexity arises because, when selecting a cut to produce a particular signal (an output of the cut), other cut outputs may or may not be used. Therefore, it is necessary to decide which cut outputs will be used to solve the problem. The cover of a circuit using *KL*-cuts can be obtained with the help of a SAT solver. This requires formulating the problem for which a solution is sought and using the SAT solver to obtain a solution if one exists.

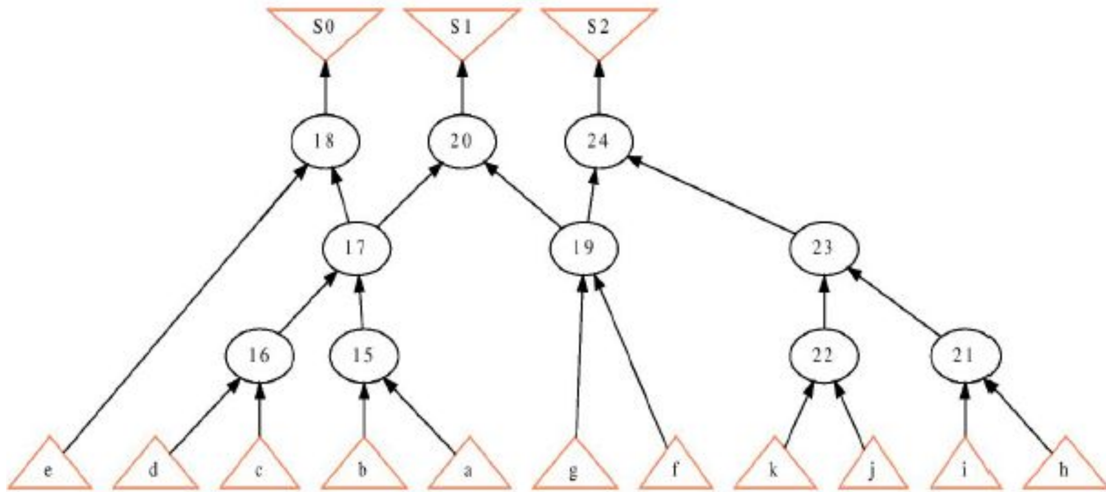
6.3.1 Satisfiability formulation for a valid cover

To obtain cover for an AIG using a SAT solver, it is necessary first to formulate the problem, create a CNF encoding that describes the restrictions to select a set of *KL*-cuts, and then use the SAT solver to determine if there exists a combination of values that satisfies the formula, i.e., if there is a combination of values that makes the problem formulation satisfiable.

To generate a cover for an AIG, it is necessary to meet the requirements described in Section 3.4.2. It is essential to ensure that: 1) all PO nodes are covered by at least one cut, and 2) the inputs to the selected cuts are produced by PIs or by outputs of other cuts. Additionally, other requirements may be considered for obtaining the cover, which will be discussed later in this section.

Following this, Figure 6.7 will be used as an example of an AIG for which we want to obtain a cover using *KL*-cuts, from which the SAT formulation will be derived.

Figure 6.7 – AIG example for SAT formulation.



The *KL*-cuts used to describe the SAT formulation are listed in Table 6.3, notice that not all *KL*-cuts are listed in this table, only a subset of all *KL*-cuts is used here to illustrate the SAT formulation of the problem. As shown, the AIG in Figure 6.7 and the cuts are the same as those used in Section 3.4.2.

Table 6.3 – KL-cuts used for SAT formulation

Cut	**Inputs**	**Outputs**
C_1	{c, d, e, 15}	{17, 18}
C_2	{e, 17}	{18}
C_3	{a, b, c, d}	{17}
C_4	{17, 19}	{20}
C_5	{f, g}	{19}
C_6	{h, i, j, k, 19}	{24}
C_7	{f, g, 17}	{19, 20}
C_8	{f, g, 23}	{19, 24}
C_9	{h, i, j, k}	{23}
C_{10}	{15, 16, 19}	{17, 20}
C_{11}	{19, 23}	{24}
C_{12}	{a, b, c, d, e}	{17, 18}
C_{13}	{a, b, c, d, f, g}	{17, 19, 20}
C_{14}	{f, g, h, i, j, k}	{19, 24}

In the sequence, we present each rule used to generate the CNF encoding to obey an AIG cover using the *KL*-cuts listed in Table 6.3.

6.3.1.1 Rule 1

The first rule of the CNF encoding ensures that each output of the AIG has at least one *KL*-cut that generates this output. To meet this requirement, it is necessary to generate a clause composed of the set of cuts that have these PO nodes as outputs. The resulting clause is an OR of the cuts that have the PO as an output. This ensures that for the clause to be satisfied, at least one cut of the node must be selected to compose the circuit cover.

Therefore, for the example in Figure 6.7, which has three POs: S0, S1, and S2, which are outputs from nodes 18, 20, and 24, respectively. The POs are generated by the following cuts: S0 by C_1, C_2, and C_{12}; S1 by C_4, C_7, C_{10}, and C_{13}; and S2 by C_6, C_8, C_{11}, and C_{14}. This gives us the following clauses:

- PO S0: (C_1 + C_2 + C_{12});
- PO S1: (C_4 + C_7 + C_{10} + C_{13});
- PO S2: (C_6 + C_8 + C_{11} + C_{14}).

In this example, for an assignment to satisfy these clauses, at least one of the cuts in each clause must be true, evaluating that clause as true. If none of the cuts in a clause is selected, assigned a value of 1, the SAT solver will return UNSAT, indicating that there is no assignment capable of satisfying all the clauses.

6.3.1.2 Rule 2

The second rule ensures that at most one *KL*-cut can be selected for each node that is an output of a *KL*-cut. The CNF encoding creates clauses composed of pairs of complemented *KL*-cuts of the node. These are clauses where each *KL*-cut appears in its complemented form with every other *KL*-cut also complemented from the node. Since each cut appears in the complemented form, if the cut is selected and assigned a value of 1, its complement will be 0, forcing the other cut in the clause to be 0 (not selected) for the clause to be satisfied.

For example, for PO S0, which has *KL*-cuts C_1, C_2, and C_{12}, this results in the clauses ($\overline{C}_1 + \overline{C}_2$) * ($\overline{C}_1 + \overline{C}_{12}$) * ($\overline{C}_2 + \overline{C}_{12}$). From the truth table (Figure 6.8), it can be observed that this set of clauses is true only when one cut is set to 1, as highlighted in yellow, or no cut is selected. Therefore, this set of clauses ensures that at most one *KL*-cut is selected for the given node.

Figure 6.8 – CNF encoding for only one cut for node 18.

c1	c2	c12	out
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

For the other POs, we have: S1 with cuts C_4, C_7, C_{10}, and C_{13} resulting in the clauses $(\overline{C}_4 + \overline{C}_7) * (\overline{C}_4 + \overline{C}_{10}) * (\overline{C}_4 + \overline{C}_{13}) * (\overline{C}_7 + \overline{C}_{10}) * (\overline{C}_7 + \overline{C}_{13}) * (\overline{C}_{10} + \overline{C}_{13})$, and S2 with *KL*-cuts C_6, C_8, C_{11}, and C_{14} resulting in the clauses $(\overline{C}_6 + \overline{C}_8) * (\overline{C}_6 + \overline{C}_{11}) * (\overline{C}_6 + \overline{C}_{14}) * (\overline{C}_8 + \overline{C}_{11}) * (\overline{C}_8 + \overline{C}_{14}) * (\overline{C}_{11} + \overline{C}_{14})$.

This rule is applied to each internal node to ensure that no more than one cut is selected for that node. Additionally, this rule can be applied to PO nodes, which in conjunction with the clauses from rule 1 ensure that at least one cut is selected. When used with rule 2, it ensures that exactly one cut is selected for the node in question.

6.3.1.3 Rule 3

The third rule of the CNF encoding ensures that all inputs of each selected *KL*-cut are available. To achieve this, for each cut, it is only selected if all its inputs are being generated. The inputs of the cuts can be PIs, which are always available and therefore do not require clauses when the inputs are PIs. However, when an input originates from a node that is not a PI, it is necessary to ensure that there is at least one cut generating that signal. Thus, for each non-PI input of each cut, a clause is derived, composed of the negation of the *KL*-cut in question along with the *KL*-cuts that generate that input.

To exemplify rule 3 of the CNF encoding, using the cuts from Table 6.3, we have that cut C_1 has inputs {c, d, e, 15}. The inputs c, d, and e are PIs, with only 15 not being a PI and thus needing to be generated by another cut. However, since there are no *KL*-cuts with node 15 as an output, we obtain the clause $(\overline{C}_1)$, which restricts C_1 from being selected.

Continuing with another example, we have cut $C2$ with inputs {e, 17}. The input

e is a PI, and 17 is not a PI. According to Table 6.3, the signal 17 can be generated by the *KL*-cuts C_1, C_3, C_{10}, C_{12}, and C_{13}, resulting in the clause ($\overline{C}_2+C_1+C_3+C_{10}+C_{12}+C_{13}$). In other words, if C_2 is 1, at least one of the cuts in this clause, which have 17 as an output, must be 1 to satisfy the clause.

6.3.1.4 Rule 4

The final rule consists of selecting only the *KL*-cuts whose outputs are used, consumed by other cuts or POs. Therefore, to ensure that all selected *KL*-cuts have their outputs being consumed, a clause is derived for each output of each cut. This clause comprises the complement of the *KL*-cut in question and the *KL*-cuts that consume that output.

For example, we have cut C_1 which has outputs {17, 18}. Among the *KL*-cuts in Table 6.3, we have *KL*-cuts C_2, C_4, and C_7 that have 17 as an input. Therefore, the clause derived for the output 17 of $C1$ is $(\overline{C}_1 + C_2 + C_4 + C_7)$, which imposes the following constraint: for C_1 to be selected (assigned 1), at least one of the cuts C_2, C_4, or C_7 must be selected to consume the output 17 of C_1. No clause needs to be derived for output 18 of C_1 as it is a PO.

The formulations presented in this section allow a cover to be obtained for the AIG; however, there is no guarantee that the result will be the optimal cover. The rules ensure that the cover is valid, but do not optimize the result. When obtaining covers for AIGs, we aim to optimize the result according to some cost function, with the most common objective being to minimize the number of cuts. In the next section, we will discuss a method for adding constraints to our CNF encoding to incorporate costs into the desired solution.

6.3.2 Satisfiability formulation for a minimum cover

As mentioned, the SAT solver aims to find an assignment that makes the CNF evaluate to 1 when a possible assignment exists, this is a decision problem. However, in logic synthesis, the goal is to obtain an optimized result. Therefore, it is necessary to adapt the process to use a SAT solver for optimization. There are some naive ways to do this, but they can result in high computational costs, which we will discuss next.

One of the simplest ways to obtain a minimal solution with a SAT solver could

be to iteratively find solutions with the solver and add new clauses at each iteration that prevent the same result from being obtained again. This process can be repeated until all solutions are found, and the minimal solution is identified. However, this procedure can be extremely slow due to the need to test numerous possible solutions until finding the best one.

Another simplistic approach could be to create additional clauses that impose a maximum number of cuts allowed in the solution. These additional clauses can be derived from the combination of all cuts being evaluated in the process. For example, if there are four cuts C_1, C_2, C_3, and C_4, the following clauses ensure that the selection of three cuts makes the CNF evaluate to 0:

- $(\overline{C}_1 + \overline{C}_2 + \overline{C}_3) \cdot (\overline{C}_1 + \overline{C}_2 + \overline{C}_4) \cdot (\overline{C}_1 + \overline{C}_3 + \overline{C}_4) \cdot (\overline{C}_2 + \overline{C}_3 + \overline{C}_4)$

This can test the maximum number of cuts allowed by varying the number until the minimum possible is found. However, generating a clause for each combination of cuts with at most N cuts is necessary, which results in a massive number of additional clauses.

Instead of exhaustively creating all combinations with the cuts to determine the maximum cost of the solution, we propose a BDD-based approach to determine the costs generated by the combinations of cuts.

6.3.3 Approach Using BDDs for Cost Constraint Solutions

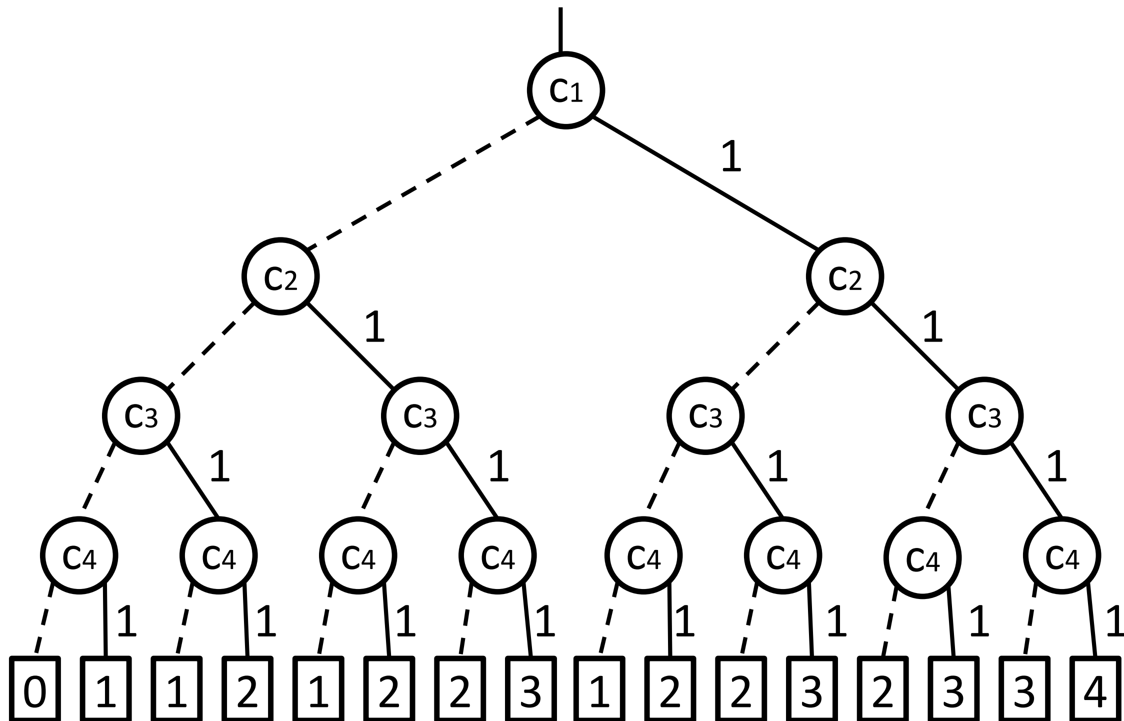
In this section, we present an alternative approach using BDDs to derive clauses that restrict solutions to meet a specified maximum cost. BDDs allow us to represent the costs associated with various sets of solutions to the problem. For instance, Figure 6.9 illustrates a BDD that functions as a cost counter for a problem involving 4 cuts C_1, C_2, C_3, and C_4, where each BDD input variable corresponds to a cut with a unit cost of 1. The number next to each positive cofactor indicates the cost of the cut when that path is taken, on the other hand, negative cofactors have no cost because they are taken when the cut does not compose the solution.

Although the example illustrated in Figure 6.9 resembles an ADD (BAHAR et al., 1997), in the context of this example, the BDD is used only to illustrate all variable combinations and describe the cost (at the terminal node) associated with each combination. This example will be used to illustrate the derivation of the actual BDD utilized to

generate cost clauses.

Each path starting from the node marked with the variable a leads to a terminal node (rectangle) that displays the total cost of that path. For example, following the path $C_1 = 1$, $C_2 = 0$, $C_3 = 1$, and $C_4 = 1$ leads to a terminal node marked with the value 3, representing the total cost of the solution comprising cuts C_1, C_3, and C_4.

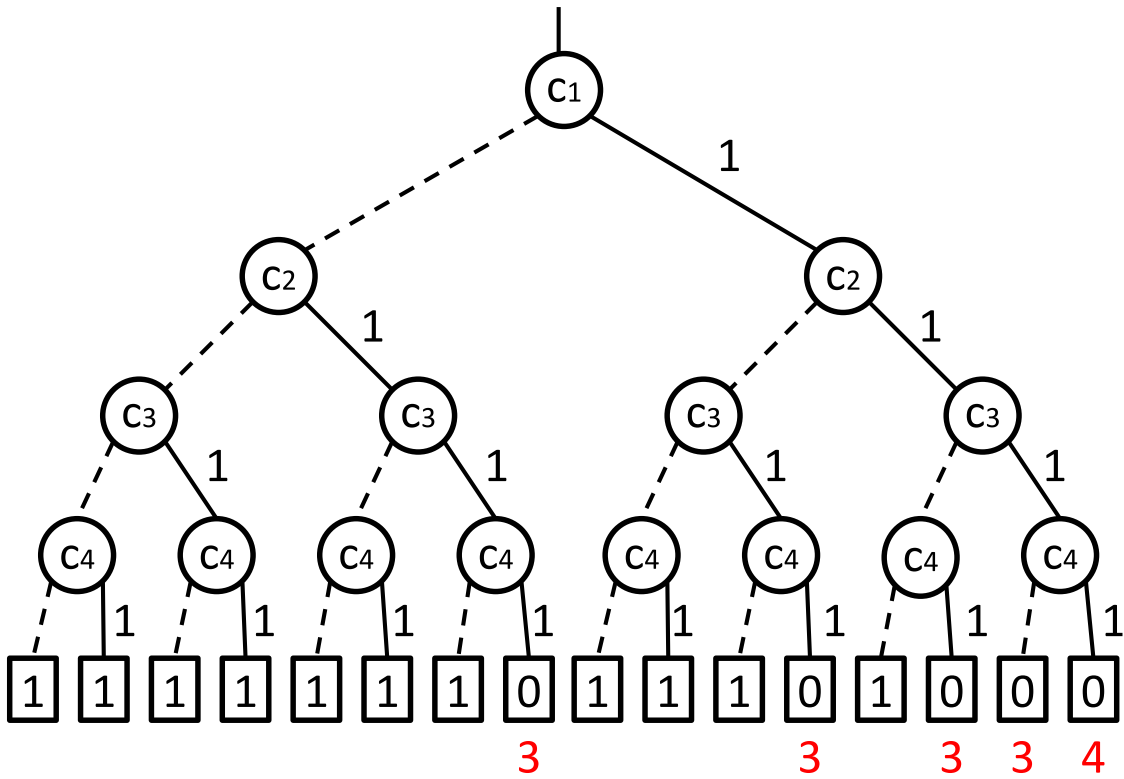
Figure 6.9 – BDD for Cuts With Unit Costs.



The BDD illustrated in Figure 6.9 shows all combinations of cuts and their associated costs, calculated as the sum of each cut's cost within the combination. However, the objective is to identify feasible solutions that do not exceed a predefined maximum cost. Therefore, we construct a BDD where the paths lead to terminal zero when the cost of the path exceeds the maximum cost, indicating that such combinations of cuts cannot be solutions as their total cost exceeds the desired limit, and the path leads to terminal one when the cost of the path does not exceed the maximum cost.

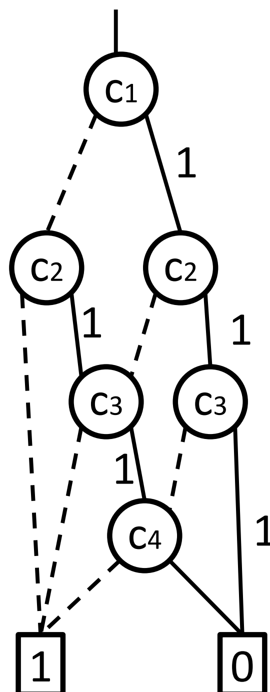
Figure 6.10 illustrates a BDD created for a problem involving four cuts, aiming to restrict solutions to those containing at most two cuts. Paths resulting in a cost greater than two lead to a terminal zero (the path's cost is highlighted below its terminal node), indicating they do not obey the maximum cost pre-defined. Conversely, paths leading to terminals marked one represent combinations of cuts totaling two or fewer, indicating they are solutions meeting the defined maximum cost.

Figure 6.10 – BDD for solutions with up to 2 Cuts.



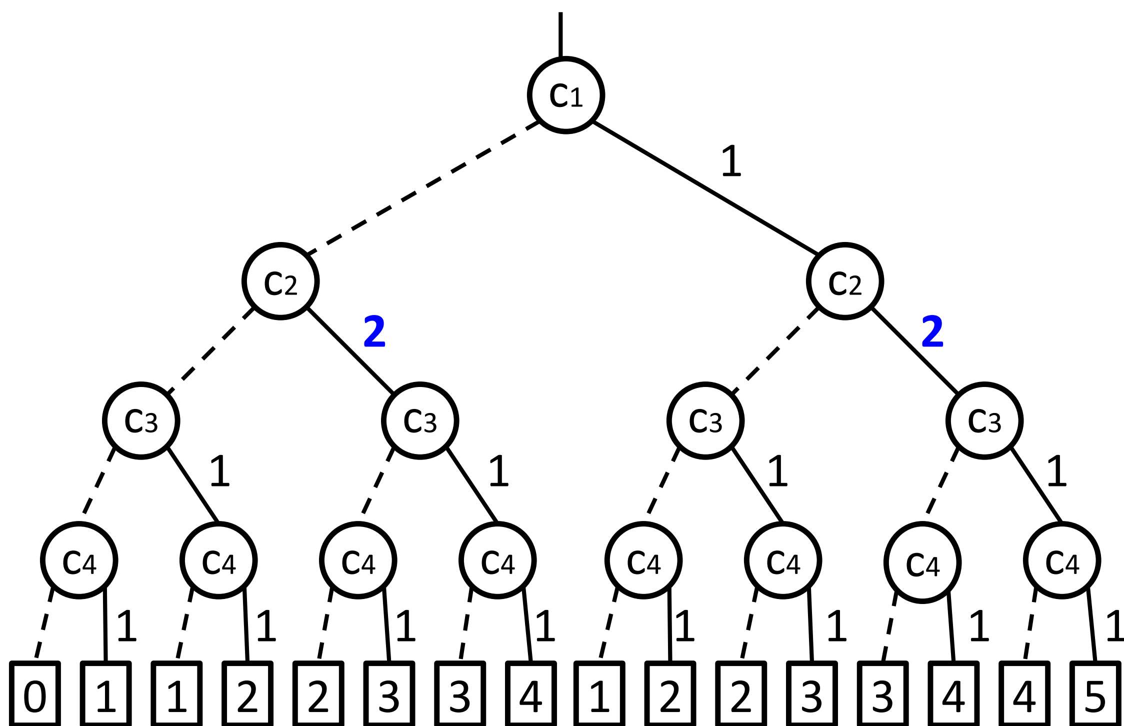
The example is illustrated using a BDD, as shown in Figure 6.10, but the procedure is performed using ROBDDs. In Figure 6.11, the ROBDD of the example problem that involves restricting solutions to at most 2 cuts out of 4 cuts is illustrated.

Figure 6.11 – ROBDD for solutions with up to 2 Cuts.



Until now, we have discussed cuts with a cost of 1, considering the number of cuts as a unit. However, cuts can have different cost metrics, such as the number of inputs, and the number of outputs, among others. With this BDD-based approach, we can consider different values of costs for the cuts, for example, Figure 6.12 illustrates a BDD constructed for four cuts, C_1, C_2, C_3, and C_4. In this example, each cut can have a different cost. For instance, cut C_2 has a cost of 2, while the other cuts have a cost of 1. This figure shows the costs of each combination of cuts, similar to the illustration in Figure 6.9. Therefore, each terminal node displays the total cost of each combination of cuts.

Figure 6.12 – BDD for a problem involving four cuts with different costs. Each terminal node shows the total cost of each combination of cuts.



Source: The Authors.

By restricting solutions to those with a maximum cost of 2, we obtain the BDD illustrated in Figure 6.13. This BDD shows the combinations of cuts that meet the pre-defined maximum cost (leading to terminal 1) and those that do not meet the maximum cost (leading to terminal 0), with the cost of each combination highlighted in red below the terminal node zero.

Figure 6.13 – BDD for a problem involving four cuts, showing combinations of cuts that meet the maximum predefined cost of 2.

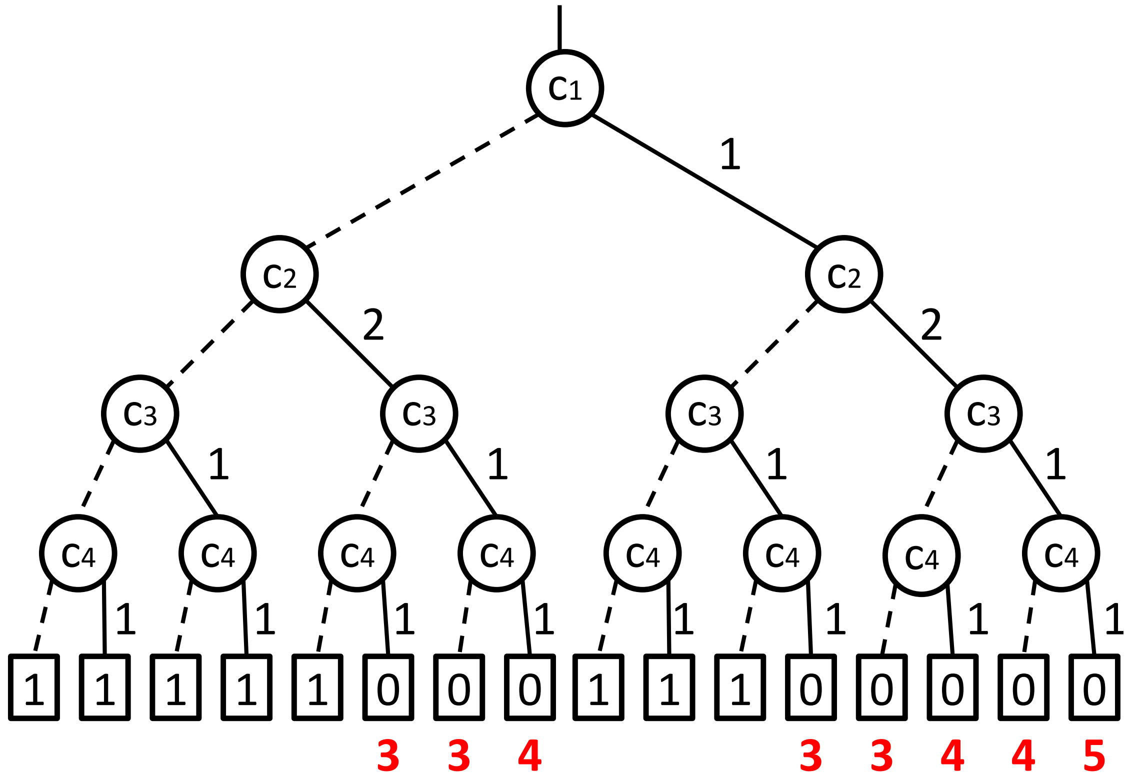
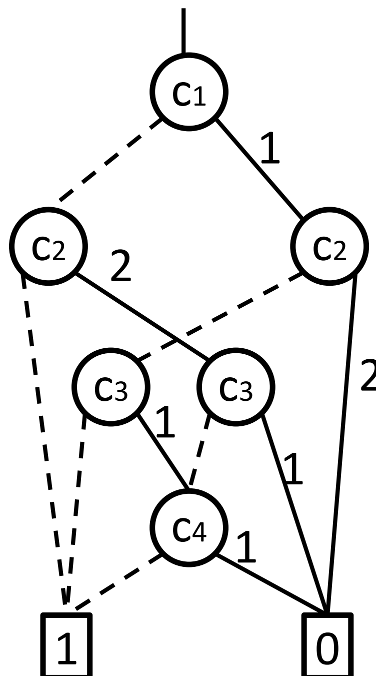


Figure 6.14 illustrates the ROBDD obtained for this problem.

Figure 6.14 – ROBDD for the problem with different costs for the cuts.



The BDD in figure 6.14 is now representing a logic function. In this function,

when the value is equal to one, the cost is acceptable, when the function is equal to zero the cost exceeds the maximum allowed. A common way to generate a satisfiability expression for a circuit is to independently generate the satisfiability expression for each of the logic gates that compose the circuit. This can be done independently for each logic gate in the circuit using the so-called Tseitin transformation (TSEITIN, 1983) for each logic gate. BDDs are not circuits and do not have logic gates. However, each node in a BDD is a Shannon decomposition (SHANNON, 1949), which is equivalent to a multiplexor logic gate. This way, the Tsetsin decomposition for a multiplexor can be used for each node in order to obtain the satisfiability clauses for the complete Boolean function. The value of the output must be set to one with a unit clause (clause with a single literal), so that only functions with an acceptable cost are selected.

6.4 Our Method Overview

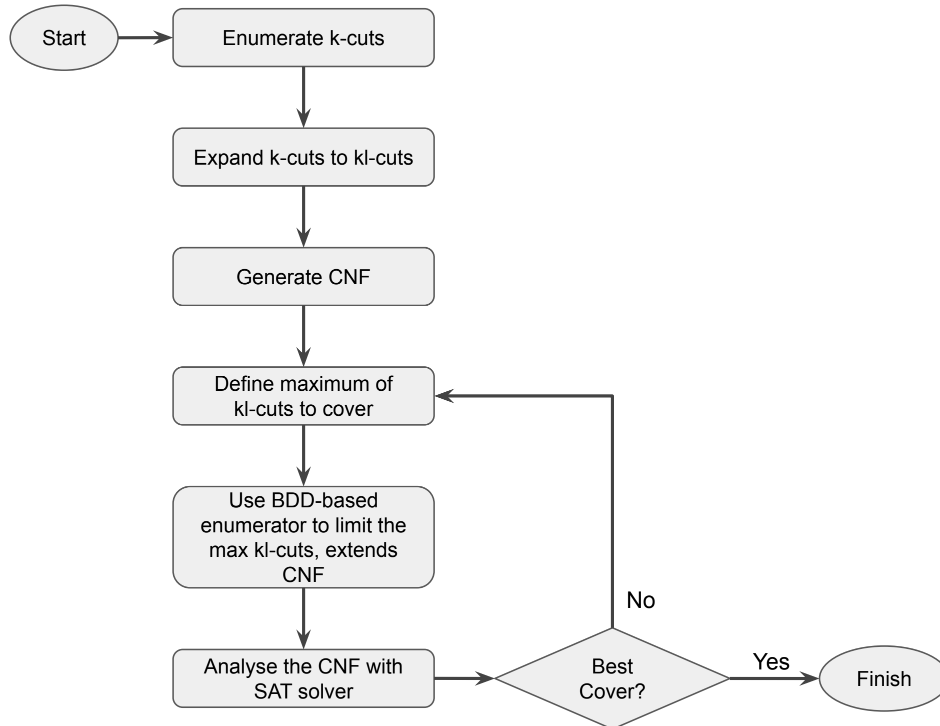
Each stage of the flow was presented separately. Figure 6.15 illustrates a flowchart that describes the organization of stages to generate an optimized cover of an AIG using *KL*-cuts. The first step involves obtaining the set of *K*-cuts, which can be generated traditionally, as described in Section 5.2.1.1, or using the method described in this work (Section 6.1.1).

The obtained *K*-cuts are then expanded into *KL*-cuts (Section 6.2), and subsequently, CNF clauses are derived (Section 6.3.2).In this work, we utilized only rules 1 and 3, as presented in Section 6.3.2. Since the goal is to achieve the best coverage in terms of the number of cuts, in some cases, the overlap between cuts can yield the optimal result.

The following steps are performed iteratively until an optimal solution is obtained. That is, the following steps are repeated until an optimal solution is achieved:

- Define a cost;
- Derive clauses using BDD to constrain the cost (Section 6.3.3), and
- Analyze the clauses using SAT to verify if there exists a solution for the circuit complying with the defined maximum cost.

Figure 6.15 – Flowchart of approach to generating minimum cover using SAT.



6.5 Contributions of This Chapter

This chapter provided a complete explanation of the method proposed in this thesis, offering detailed insights into the algorithms and practical examples of their application.

7 RESULTS AND DISCUSSIONS

7.1 About this chapter

In this chapter, we present and analyze the results obtained from our proposal for *KL*-cuts enumeration and cover generation for circuits using *KL*-cuts. In the first part, we focus on comparing the execution time of our *KL*-cuts enumeration methodology with reference methods, such as the Machado algorithm (MACHADO et al., 2012) and MFFW (ZHU et al., 2023). Time efficiency is a crucial factor to validate the feasibility of our proposal in practical scenarios.

In the second part, we evaluate the cover quality obtained by our method. To do this, we post-process the cover generated by the ABC tool (BRAYTON; MISHCHENKO, 2010). The ABC tool initially generates covers only with single-output cuts. To enable a comparison in the context of multiple-output cuts, we perform post-processing on the cuts obtained from the ABC tool to transform single-output cuts into multiple-output cuts. This post-processing involves grouping cuts whose inputs are dominated.

The cover achieved with our proposed method in this study is validated through combinational equivalence checking using the ABC tool (BRAYTON; MISHCHENKO, 2010), specifically employing its "cec" command (MISHCHENKO et al., 2006). This validation ensures that the circuits generated by our method maintain logical equivalence with the reference circuits, thereby confirming that the obtained cover is valid and that the generated circuits meet the specifications of the original circuits. The SAT solver used to conduct these experiments was the MiniSat (SÖRENSSON, 2010).

7.2 *KL*-cut enumeration performance

To conduct the performance experiment of our proposed method, we used the EPFL benchmark (AMARÚ; GAILLARDON; MICHELI, 2015). This set of benchmarks is widely recognized in the community and provides a solid basis for comparing *KL*-cuts enumeration methods.

All the methods compared, including ours, are based on obtaining the multi-output cuts by expanding *K*-cuts. Therefore, the time required to enumerate the *K*-cuts will not be considered in the performance analysis. Our analysis will focus exclusively on the runtime required for the enumeration of *KL*-cuts, as the number of cuts produced will be

the same for all methods.

To evaluate the performance of our method, we compared it with the *KL*-cuts enumeration method of Machado. Although Machado's work originally focuses on cuts in mapped circuits, it can also be applied to AIGs, allowing for a direct comparison. Additionally, we included the dynamic version of the MFFW method (TANG et al., 2023) in the comparison, a new method that has been gaining attention in the field. To ensure a fair comparison with the MFFW method, we used *KL*-cuts with *L* unbounded.

Our experiment considers runtime as the main comparison metric. This is because, as all methods are based on the expansion of *K*-cuts, the final amount of *KL*-cuts will be the same for all. Therefore, we focus on the runtime of each method, providing a clear and objective measure of performance.

The runtimes of the methods are presented in Table 7.1. The first column of the table contains the names of the benchmarks, while the second column shows the number of ANDs in each circuit. The third column shows the number of *K*-cuts used for expansion. Finally, the fourth, fifth, and sixth columns respectively present the times, in seconds, that the Machado method, our method, and the MFFW method took to expand the *K*-cuts into *KL*-cuts, which is the *KL*-cuts enumeration process.

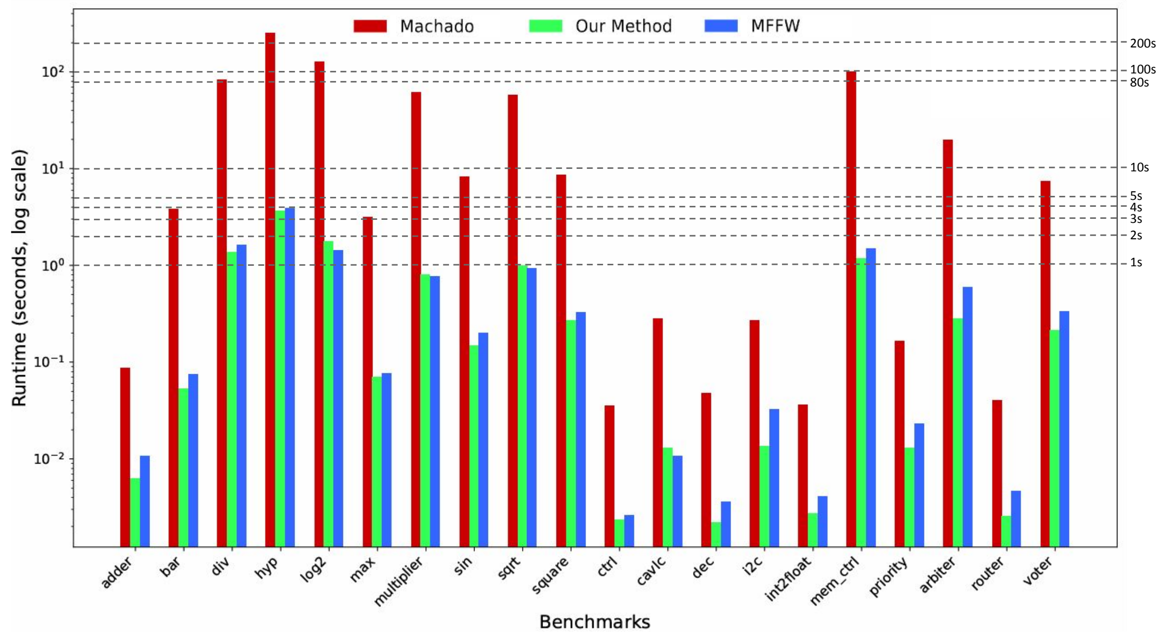
Table 7.1 – Runtime comparison in milliseconds.

Benchmark	Ands	K-cuts	Machado (ms)	Our Method (ms)	MFFW (ms)
adder	1020	5298	86.27	6.20	10.61
bar	3336	25007	3772.25	53.25	74.35
div	44762	724848	83083.27	1358.61	1614.35
hyp	214335	1611185	250120.19	3642.02	3911.15
log2	32060	437138	128035.67	1776.23	1414.56
max	2865	33880	3164.00	70.26	76.14
multiplier	27062	293431	60774.41	801.54	764.27
sin	5416	72026	8261.03	148.29	201.27
sqrt	24618	361355	57880.77	1000.81	922.66
square	18484	154957	8597.41	269.56	326.24
ctrl	11839	972	35.37	2.33	2.58
cavlc	693	5655	282.48	13.05	10.70
dec	174	1718	47.57	2.19	3.63
i2c	304	11403	269.38	13.39	32.31
int2float	1342	1741	36.26	2.71	4.05
mem_ctrl	260	574773	100001.35	1167.75	1481.68
priority	46836	12518	166.35	12.98	22.99
arbiter	978	218166	19921.06	279.89	595.54
router	257	1877	40.27	2.54	4.69
voter	13758	111600	7453.07	214.98	330.87

To conduct this experiment, we used a high value of priority cuts per node, where each node maintained a maximum of 20 K-cuts during the enumeration of K-cuts. This is not a practical value, as according to (MISHCHENKO et al., 2007), only a small number of priority cuts, around 5 to 10, is considered a good number of cuts per node. This value was used solely to allow the generation of a larger number of K-cuts for each circuit, which are later expanded into KL-cuts.

For a clearer analysis of the results, the data from Table 7.1 is organized and presented graphically in Figure 7.1. This graph highlights the magnitude of performance differences among the methods under study. Using a logarithmic scale for execution times, the graph facilitates visual comparison between the methods, enabling a precise analysis of variations in execution times. This approach reveals significant differences that can span multiple orders of magnitude among the evaluated methods.

Figure 7.1 – Comparison of Runtimes for Machado, Our Method, and MFFW (Log Scale).



For instance, as can seen in Figure 7.1, for the "div" benchmark the Machado method has an execution time of approximately 83 seconds, while Our Method takes about 1.36 seconds and the MFFW method takes about 1.61 seconds. This represents a nearly two orders of magnitude difference between Machado and the other two methods.

The Machado method is significantly slower than the other two methods for most benchmarks. For example, in Figure 7.1 for the "hyp" benchmark, the Machado method takes about 250 seconds, while Our Method takes about 3.64 seconds and MFFW takes about 3.91 seconds. This shows that the Machado method can be up to 70 times slower than the other methods. Regarding relative performance, Our Method generally outperforms both the Machado and MFFW methods. On average, our multi-output cut enumeration method is 1.25 times faster than MFFW for this specific benchmark. The difference in the "mem_ctrl" benchmark is even more pronounced: the Machado method takes approximately 100 seconds, while Our Method and MFFW take about 1.17 seconds and 1.48 seconds, respectively. In this case, the Machado method is about 85 times slower than Our Method.

Machado's approach leads to worse runtimes because this expands each cut c_i from each of its input nodes to all nodes containing the cut c_i itself or any other cut whose inputs are dominated by the inputs of c_i. This approach requires numerous dominance checks of cuts, performed by comparing the inputs of the cuts, which has a significant impact on the performance of the Machado method. Overall, Our Method exhibits the best execution times in most benchmarks, closely followed by the MFFW method.

7.3 Covering with *KL*-cuts

The second experiment conducted in this study aims to evaluate the proposed cover method. This experiment is divided into two main parts: 1) Comparing the cover obtained with *KL*-cuts using our enumeration method, CNF encoding of the problem, and SAT solving; and 2) Analyzing the workflow performance.

Since the goal of our work is to obtain solutions with the minimum number of *KL*-cuts, we allow overlap between *KL*-cuts. Therefore, Rule 2 of the formulation, Section 6.3.1, is not used. Additionally, we do not restrict the use of all outputs from the *KL*-cuts; hence, Rule 4 is also not applied. Outputs that are not used in the cover can be removed later.

In this experiment, we compare the covers obtained by our method with those obtained using the ABC tool. However, the ABC tool has no built-in method to generate covers using multi-output cuts. To address this limitation, we implement a post-processing step for the covers generated by the ABC tool.

Initially, ABC generates covers with only single-output cuts. Then, we apply a post-processing method to merge cuts where one cut dominates another. For instance, if we have $c1 = (\{a, b, c, d\}, o1)$ and $c2 = (\{b, c\}, o2)$, $c1$ dominates $c2$ because $c2$'s inputs is a subset of $c1$'s inputs. Thus, we update $c1$ to $c1 = (\{a, b, c, d\}, \{o1, o2\})$.

Additionally, we perform a combination of cuts where the union of inputs from two cuts c_i and c_j has a size smaller than K, thereby forming a *KL*-cut. For example, starting with k-cuts $c_i = (\{a, b, c\}, o_i)$ and $c_j = (\{d, e, f\}, o_j)$, we generate the *KL*-cut $c = (\{a, b, c, d, e, f\}, \{o_i, o_j\})$.

This approach allows us to evaluate our method by comparing our results with those obtained using the ABC tool. Although our results are derived from post-processing the cover obtained with the ABC tool, this comparison provides a good reference for evaluating our results, as the ABC tool is a well-established reference in the field of logical synthesis.

Another point considered in the following experiments is that the number of *KL*-cuts in each node of the AIG can significantly impact the method's performance. Therefore, we decided to limit the number of *KL*-cuts in each node to 10.

7.3.1 Exact Cover Size

In this experiment, we compared the cover sizes of circuits using two different approaches: *K*-cuts, obtained with the ABC tool, and *KL*-cuts, obtained with our method. We used circuits from the MCNC benchmark due to the variety of circuit sizes, making this experiment feasible.

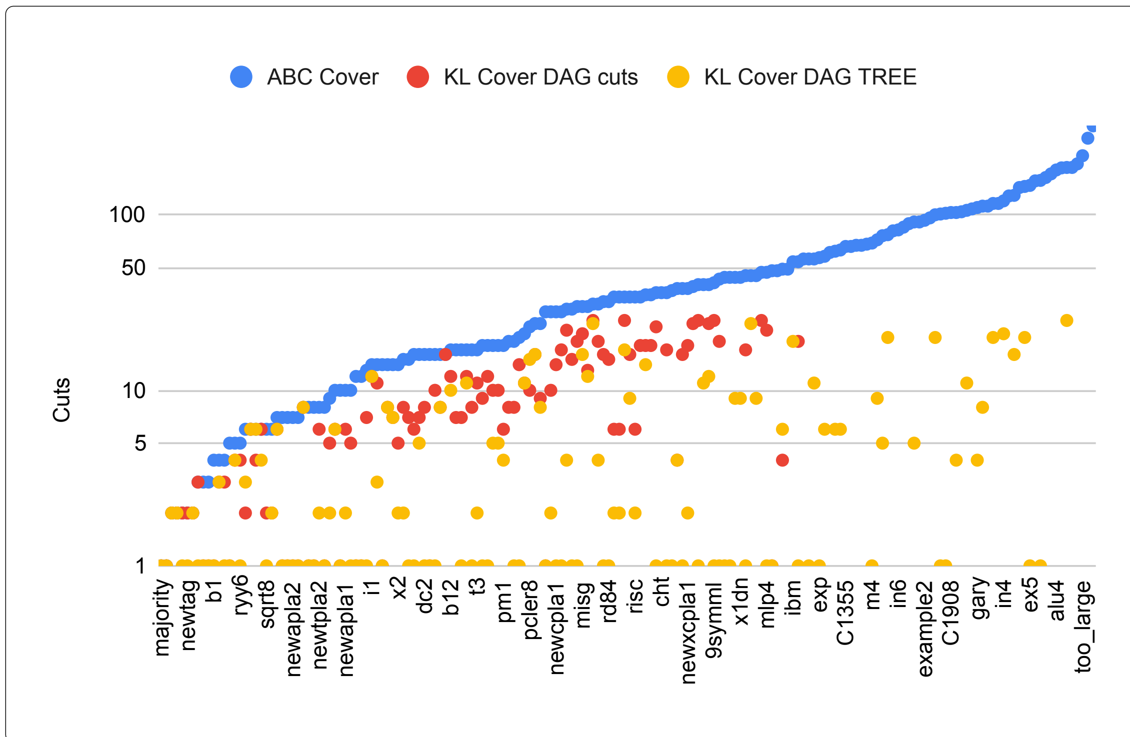
Our method aims to find the minimum cover in terms of the number of cuts for a circuit using *KL*-cuts. To achieve this, we require an intermediate step in the flow that adds new clauses to the CNF to determine the maximum number of cuts in the solution. This approach uses BDDs to create the new clauses, which is more efficient than brute force (generating all possible combinations). However, performance is directly related to the number of cuts in the problem. Therefore, we use only the 10 largest *KL*-cuts for each node in terms of internal nodes.

Therefore, for circuits with a large number of cuts, the performance of our approach is significantly hindered. To conduct this experiment, which aims to evaluate the effectiveness of our method in terms of solutions with the minimum number of cuts, we used circuits with a maximum of 1000 AND gates and generated solutions with a maximum of 25 *KL*-cuts.

7.3.1.1 Cover Size

Figure 7.2 presents a graph illustrating the results obtained with the ABC tool (blue circles), using the command "if -K 6", which generates a cover with *K*-cuts with a maximum of 6 inputs. These values are compared with the results obtained by our method, using *KL*-cuts expanded from DAG cuts only (red circles) and mixed DAG and TREE cuts (orange circles). In our results, DAG cuts also have up to 6 inputs, while TREE cuts can have more than K inputs. The results are ordered according to the values obtained by the ABC tool, from smallest to largest, to provide better organization of the results.

Figure 7.2 – Comparison of Cover Size Using Single-Output Cuts (from ABC) vs. Multi-Output Cuts (with *KL-cuts*)

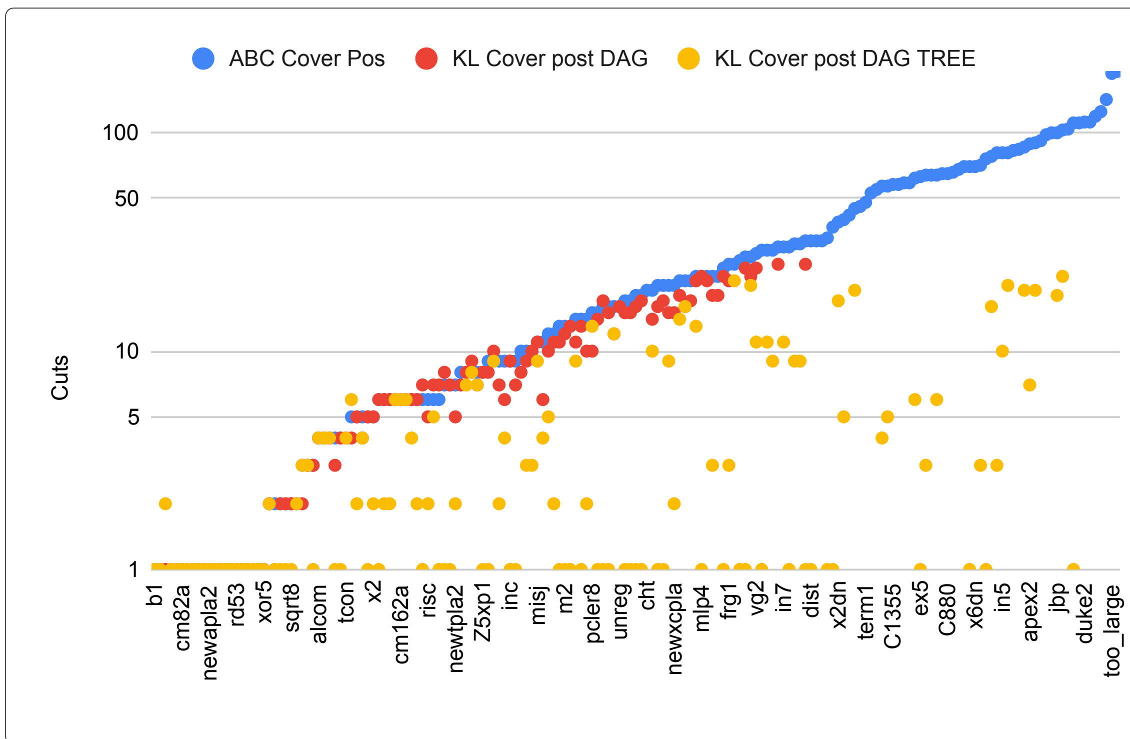


As can be observed, multi-output cuts can significantly reduce the number of cuts required to cover the circuit. This is expected, as a *KL*-cut, with *L* unbounded, encompasses the entire subgraph defined by a *K*-cut. However, our method using only DAG cuts could not obtain solutions for all the tested circuits due to the BDD limitation mentioned earlier.

Despite this limitation, the benefit of using multi-output cuts is evident compared to the approach that uses only single-output cuts. When we compared the results using *K*-cuts from ABC with our method using *KL*-cuts expanded from DAG cuts only, our method achieved cover with an average of 49.01% fewer cuts. Using *KL*-cuts expanded from both DAG and TREE cuts, we achieved cover with an average of 74.85% fewer cuts than when using *K*-cuts. These values were obtained with the successful cover results.

For a closer comparison, we performed post-processing on the cover obtained with ABC, generating multi-output cuts from the *K*-cuts used in the ABC solution. The results of this procedure are presented in Figure 7.3, where we compare these results with those of our method also using post-processing, applied after obtaining the cover from the SAT solver. The results are ordered according to the post-processed values obtained by the ABC tool, from smallest to largest, to provide better organization of the results.

Figure 7.3 – Cover Size Comparison of Results of Figure 7.2 after the Post-Process.



As can be seen, performing post-processing on the results obtained with ABC results in a considerable reduction in the number of cuts. Despite the results being close, our method still shows good results when compared to ABC, where we obtained cover with a reduction of 7.51% using only DAG cuts and 53.65% when using DAG and TREE cuts compared to the ABC results.

It is important to note that ABC also presents the best results for some of the circuits, this occurs because our method does not use all the generated *KL*-cuts to generate the CNF, as explained at the beginning of this section. Consequently, this may exclude *KL*-cuts that could produce the smallest coverage. Lastly, as shown in the graph in Figure 7.2, some circuits did not have a solution due to exceeding the maximum cut threshold set for this experiment.

One point to highlight is that in this experiment, the optimization criterion used was the number of cuts needed to cover the circuit. However, once we have a way to assign costs to the cuts, we can use different cost functions, such as the number of input and output signals of the cuts.

7.3.1.2 Cover Runtime

The next aspect analyzed is the runtime our method takes to generate cover using *KL*-cuts. Thus, in this experiment, we aim to evaluate the performance of our method.

Since ABC involves a comprehensive flow based on *K*-cuts to generate cover, and includes sophisticated techniques, we cannot directly compare the runtime. Therefore, this experiment focuses on expressing the runtime of each stage of our process to generate an AIG cover using *KL*-cuts.

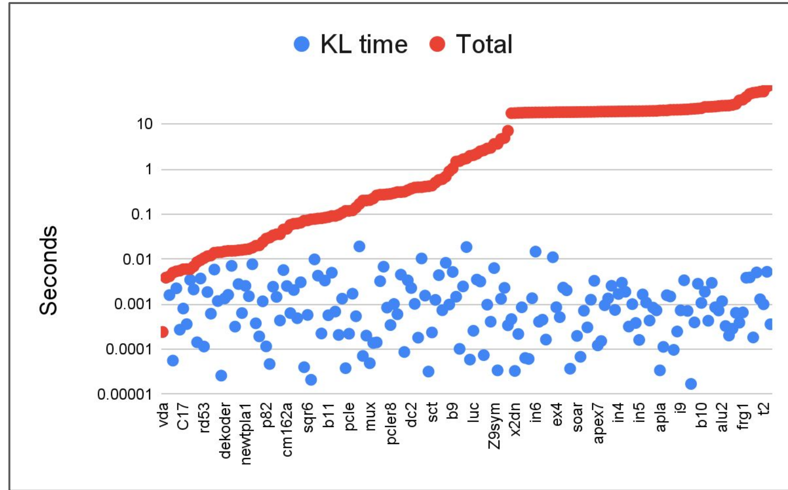
Subsequently, the runtimes of each stage of our procedure are presented, with each stage's runtime compared to the total runtime of the procedure to give a sense of the proportion of each stage's runtime.

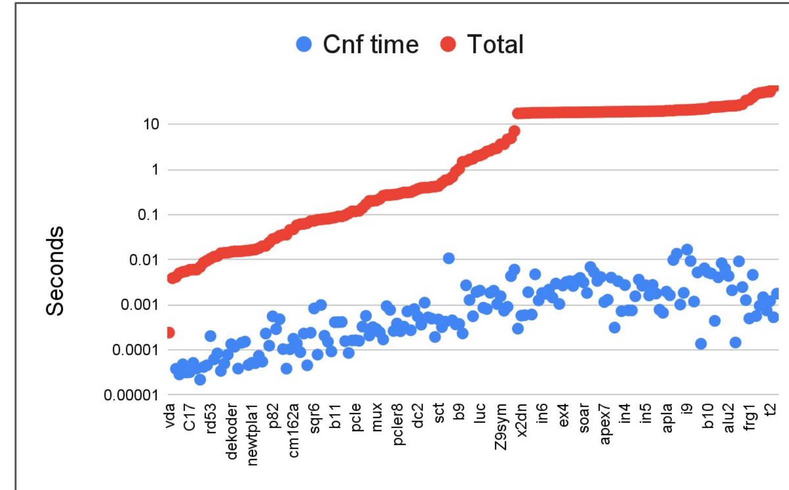
Figure 7.4 presents the runtimes of the stages of our method in dedicated graphs. The data are ordered by total runtime, which is the sum of the four stages. These graphs show the runtimes of the following stages:

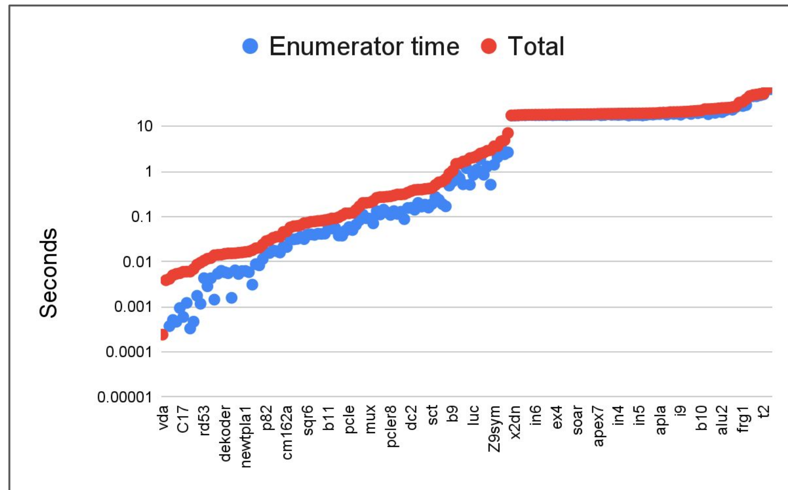
1. Expansion of *KL*-cuts;
2. Generation of CNF;
3. Enumeration of the maximum number of cuts in the solution;
4. SAT solver.

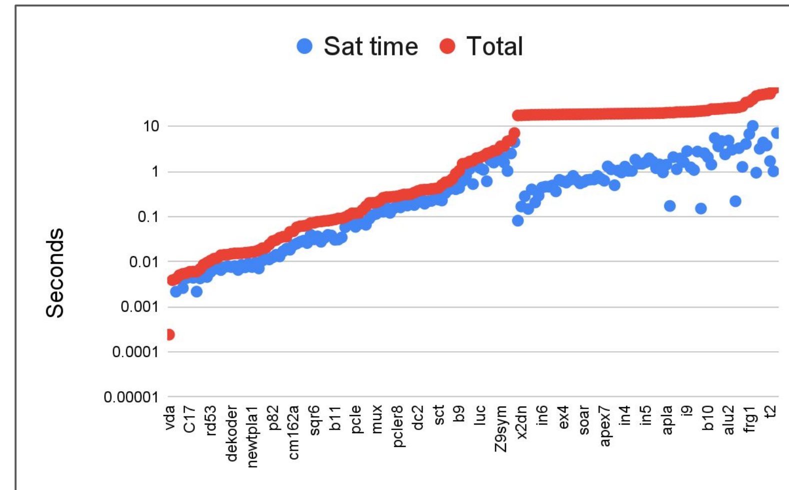
Two considerations should be made at this point: stages 1 and 2 are executed only once for each circuit, while stages 3 and 4 are repeatedly executed until they converge to a solution or fail to find a solution with at most 25 cuts, the same configuration as the previous section.

Figure 7.4 – Runtime using only DAG cuts.









From Figure 7.4, the proportion of the runtime of *KL*-cuts expansion relative to the total runtime of the method is illustrated. It can be observed that expansion is a stage that does not have a significant impact on the total time of the method, given that the graphs are on a logarithmic scale. This stage shows insignificant runtimes for most circuits.

Following in Figure 7.4, similar to *KL*-cuts expansion (stage 1), the generation of CNF (stage 2) does not have a significant impact on the total runtime of the method, with only a few circuits in the test set showing a runtime exceeding 0.01 seconds.

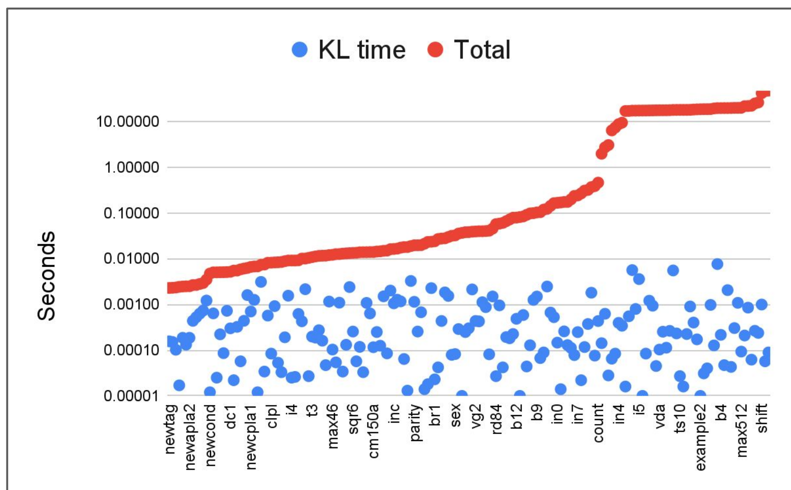
Still in Figure 7.4, the next stage analyzed is the enumeration of the maximum number of solution cuts (stage 3). This stage aims to restrict the maximum threshold of cuts that the solution can contain. As can be seen, it is one of the stages that have the most significant impact on the total runtime of our method. This procedure involves generating additional clauses imposing restrictions on combinations of up to N cuts. This procedure is faster than generating all possible combinations of cuts that do not admit a particular solution with $N + 1$ cuts, but it is still a procedure that becomes more time-consuming as the number of cuts increases.

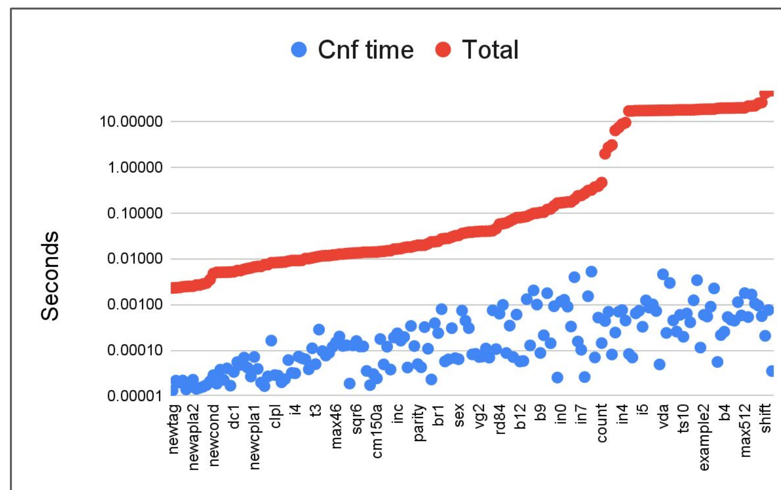
In this experiment, it was observed that solutions with up to 25 cuts could be found in a timely manner; beyond this value, the performance is significantly degraded. As can be observed, the runtime of stage 3 was practically the total runtime of the method for some circuits.

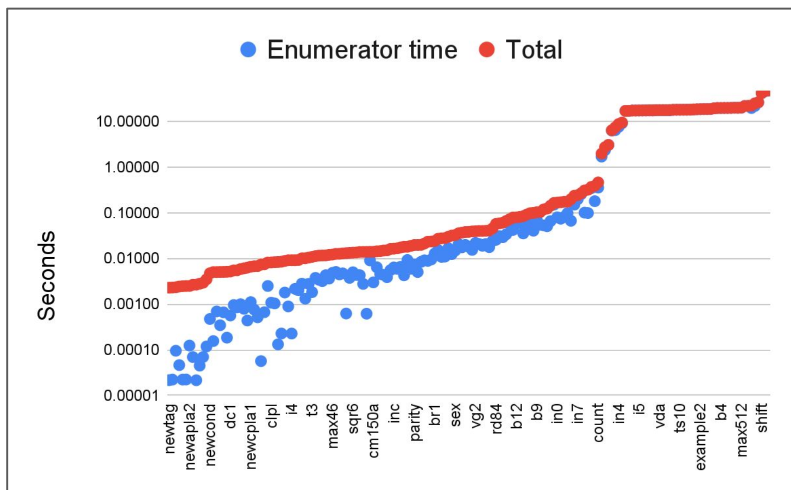
Continuing the analysis, the last stage consists of running the SAT solver to find a solution if one exists (stage 4). It can be observed that this stage also has a significant impact on the total runtime of the method. However, it is noticeable that the runtime of stage 4 is lower than that of stage 3 for the circuits with the highest total runtimes in the graph. In these cases, stage 3 was the bottleneck of the method. For other circuits with a total runtime of less than 1 second, the runtime of stage 4 is considered the bottleneck among the four stages because running the SAT solver requires system calls; it is not being used as a method in the project.

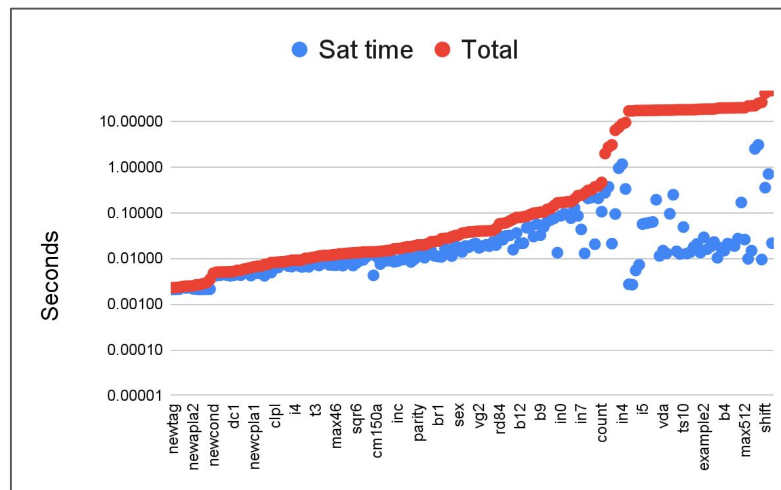
Similarly to the runtime analysis of our method using DAG cuts, Figure 7.5 presents the runtimes of each stage using DAG and TREE cuts. As can be seen, these graphs show behaviors similar to those illustrated in the graphs of Figure 7.4. The difference lies only in the number of *KL*-cuts involved in the process, which, in the case of DAG and TREE cuts, are fewer than with only DAG cuts.

Figure 7.5 – Runtime using DAG and TREE cuts.









7.3.2 Cover Without Optimization

So far, we have analyzed only the results of our method aiming to find the best coverage in terms of the number of cuts for each tested circuit. However, as discussed, there is a bottleneck that prevented larger circuits from being tested.

In this section, our goal is to test our method with larger circuits. To do this, stage 3, responsible for defining the maximum number of solution cuts, is not executed, as it was previously identified as the major bottleneck of our method. Thus, the workflow used in this test does not necessarily find an optimal solution; our goal is solely to analyze the performance of the method on larger circuits.

Table 7.2 presents the results in terms of the number of cuts using the approach of generating coverage with ABC and applying the post-processing described at the beginning of this section. These ABC results are compared with the results obtained with our method using DAG cuts. Since our method uses multiple output cuts, there is a higher chance of obtaining better results using multi-output cuts.

Table 7.2 – Cover without optimization for EPFL circuits

	From ABC		Using KL-cuts (DAG cuts)	
Circuit	Original Cover with K-cuts	Merged Cuts	Cover size	Cover size merged
adder	257	79	64	64
bar	512	448	448	448
div	22253	10719	-	-
hyp	44440	20054	-	-
log2	7931	5370	-	-
max	839	509	601	579
multiplier	5890	3894	-	-
sin	1472	871	1136	1101
sqrt	6409	2501	5434	5328
square	3978	1439	2075	1955
ctrl	29	5	7	6
cavlc	121	70	74	73
dec	287	130	19	11
i2c	363	274	260	241
int2float	49	36	38	37
mem_ctrl	12086	8869	8832	8572
priority	219	119	244	211
arbiter	2722	2466	2404	2404
router	94	40	34	33
voter	2830	873	1404	1190

As seen in Table 7.2, although our approach in this context does not aim to obtain optimal solutions, we used the first solution provided by the SAT solver. In some cases, our method obtained better solutions than the ABC results.

Another analysis that can be conducted is the runtime in this context. For this, Table 7.3 presents data on the size of the CNFs generated and the runtime spent in each stage of our method in the context of this experiment, without using stage 3.

Table 7.3 – Clause size and runtime.

Circuit	KL-cuts	Cnf Clauses	Cnf Vars	Cnf time (ms)	Sat time (ms)
adder	2093	4796	2093	2.61	4.19
bar	7169	19960	7169	22.52	213.81
div	245269	1139142	245269	-	-
hyp	599811	2483613	599811	-	-
log2	159195	713690	159195	-	-
max	15242	44183	15242	53.05	926.71
multiplier	92063	282164	92063	-	-
sin	26292	124257	26292	83.06	445.07
sqrt	116146	597331	116146	910.90	14008.50
square	61607	191054	61607	126.89	470.10
ctrl	281	512	281	0.81	2.87
cavlc	2496	4860	2496	8.47	25.70
dec	353	930	353	1.24	3.71
i2c	6303	11871	6303	10.24	31.04
int2float	1035	1711	1035	1.92	4.87
mem_ctrl	250817	697396	250817	1100.22	13926.00
priority	5050	12927	5050	7.07	11.13
arbiter	108364	364746	108364	248.41	2397.97
router	918	1838	918	1.20	3.48
voter	36570	139435	36570	82.55	100.71

As observed, the size of the CNF is directly related to the number of cuts in the problem, where the number of variables in the CNF equals the number of cuts for which a solution is sought. The number of clauses is also linked to the number of cuts, as the approach used generates clauses based on the list of cuts in the AIG nodes.

The runtime data presented in Table 7.3 show that the method proposed in this work can be feasible, being executable for a considerable set of circuits. Although there is a significant bottleneck in stage 3, the method proves to be viable in the other stages.

However, it is important to note that the circuits "sqrt" and "mem_ctrl" exhibited significantly higher runtimes compared to the other circuits. This is because these circuits have larger CNFs owing to the number of cuts involved in the problem.

7.4 Contributions of this chapter

This chapter presented the methods used to conduct the experiments and the results obtained from them. A detailed discussion of the results was provided to enable the reader to understand the benefits, as well as the drawbacks, of the methods proposed in this thesis.

8 CONCLUSION

This thesis presents a method for deriving cover using *KL*-cuts for AIGs. A *KL*-cut is a multi-output cut that extends *K*-cuts, which are limited to a single output. *KL*-cuts enable regions of the circuit to be more effectively isolated compared to *K*-cuts, as a *KL*-cut can encompass all the logic defined by a cut, something traditional *K*-cuts cannot do due to their single-output limitation.

Three contributions are proposed in this work:

- Expansion of *K*-cuts to *KL*-cuts;
- CNF formulation of the problem to obtain a circuit cover with *KL*-cuts;
- Cost encoding using BDDs for a minimum cover.

The first contribution of this work is a novel approach to enumerating *KL*-cuts in AIGs. Unlike previous methods (MARTINELLO et al., 2010; MACHADO et al., 2012), which rely on comparing the leaf nodes of the cuts to generate *KL*-cuts, an expensive task, our method adopts a different approach. In our method, each *KL*-cut is assigned its own signature, and the *KL*-cuts are composed based on the intersection of the fanins of each node in the AIG. Our approach demonstrates a significant performance gain compared to the method proposed by (MACHADO et al., 2012), which is an enumeration method for *KL*-cuts. Specifically, the method by (MACHADO et al., 2012) is approximately 85 times slower than our method when using the (AMARÚ; GAILLARDON; MICHELI, 2015) benchmark. When compared to the MFFW method of (TANG et al., 2023), our method shows comparable results for most of the tested circuits, with (TANG et al., 2023)'s method performing better on larger circuits. However, the method proposed in this thesis still offers competitive performance for expanding *KL*-cuts.

The second contribution of this work consists of a method, described in Section 6.3.2, to obtain a CNF formulation for the covering problem using multi-output cuts. This method generates a formulation that ensures the requirements presented in Section 3.4.2 are met, thereby producing a valid cover. However, the formulation presented in this work is not yet an optimized version. Section 7.3.2 presents the sizes of the CNFs for the (AMARÚ; GAILLARDON; MICHELI, 2015) benchmark.

Lastly, a BDD-based method was proposed to consider the cost of each *KL*-cut. This method derives CNF clauses that are used to set the desired maximum cost for the solution. These clauses, together with the ones described above, are used to find the

optimal cover for the problem.

Overall, this work presents a flow based on the three steps described above to obtain a minimal cover of an AIG. Since satisfiability problems are decision problems, where the answer provided is either true (SAT) or false (UNSAT), the problem is modeled to use a SAT solver to obtain the best cover. This is done using an iterative approach, where each iteration defines a maximum cost, creates the constraints, and uses a SAT solver to obtain the answer.

The process presents a bottleneck during the stage of creating the maximum cost constraints for the solution, as described in Section 6.3.3. However, since this is an exact synthesis problem, where the goal is to achieve the best result, it is expected that performance might be compromised. Furthermore, results show that using multi-output cuts for the AIG covering problem produces covers with a smaller number of cuts compared to using single-output cuts.

As this is expected behavior, the comparison presented in Section 7 attempts to approximate the comparison by grouping dominated cuts, thus transforming the cuts of the solutions generated by the ABC tool into multi-output cuts. Even so, our approach was able to generate good results compared to those of ABC for a large portion of the tested circuits.

For each *KL*-cut composing the cover generated, a simple equation was generated to validate the result of the method. The results were checked and validated with the ABC tool (BRAYTON; MISHCHENKO, 2010) using the "cec" command (MISHCHENKO et al., 2006) that performs the combination equivalence checking.

REFERENCES

AKERS, S. B. Binary decision diagrams. **IEEE Transactions on computers**, IEEE, n. 6, p. 509–516, 1978.

AMARÚ, L.; GAILLARDON, P.-E.; MICHELI, G. D. The epfl combinational benchmark suite. In: **Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)**. [S.l.: s.n.], 2015.

BAHAR, R. I. et al. Algebric decision diagrams and their applications. **Formal methods in system design**, Springer, v. 10, p. 171–206, 1997.

BRACE, K. S.; RUDELL, R. L.; BRYANT, R. E. Efficient implementation of a BDD package. In: IEEE. **27th ACM/IEEE design automation conference**. [S.l.], 1990. p. 40–45.

BRAYTON, R.; MISHCHENKO, A. Abc: An academic industrial-strength verification tool. In: SPRINGER. **Computer Aided Verification: 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings 22**. [S.l.], 2010. p. 24–40.

BRAYTON, R. K. et al. A comparison of logic minimization strategies using espresso: An apl program package for partitioned logic minimization. In: **Proceedings of the International Symposium on Circuits and Systems**. [S.l.: s.n.], 1982. p. 42–48.

BRAYTON, R. K. et al. Vis: A system for verification and synthesis. In: SPRINGER. **Computer Aided Verification: 8th International Conference, CAV'96 New Brunswick, NJ, USA, July 31–August 3, 1996 Proceedings 8**. [S.l.], 1996. p. 428–432.

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.

CALVINO, A. T.; MICHELI, G. D. Technology mapping using multi-output library cells. In: **IEEE/ACM International Conference on Computer-Aided Design (ICCAD)**. [S.l.: s.n.], 2023.

CALVINO, A. T. et al. A versatile mapping approach for technology mapping and graph optimization. In: IEEE. **2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)**. [S.l.], 2022. p. 410–416.

CHAI, D. et al. Mvsis 2.0 user's manual. **Department of Electrical Engineering and Computer Sciences**, 2003.

CHATTERJEE, S.; MISHCHENKO, A.; BRAYTON, R. Factor cuts. In: **Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design**. [S.l.: s.n.], 2006. p. 143–150.

CHEN, D.; CONG, J. Daomap: A depth-optimal area optimization mapping algorithm for fpga designs. In: IEEE. **IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.** [S.l.], 2004. p. 752–759.

CHEN, G.; CONG, J. Simultaneous logic decomposition with technology mapping in fpga designs. In: **Proceedings of the 2001 ACM/SIGDA ninth international symposium on Field programmable gate arrays**. [S.l.: s.n.], 2001. p. 48–55.

CIESIELSKI, M. et al. Understanding algebraic rewriting for arithmetic circuit verification: a bit-flow model. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 39, n. 6, p. 1346–1357, 2019.

CONG, J.; DING, C. W. Y. Cut ranking and pruning: Enabling a general and efficient fpga mapping solution. p. 29–36, 1999.

CONG, J.; DING, Y. On area/depth trade-off in lut-based fpga technology mapping. In: **Proceedings of the 30th International Design Automation Conference**. [S.l.: s.n.], 1993. p. 213–218.

CONG, J.; DING, Y. Combinational logic synthesis for lut based field programmable gate arrays. **ACM Transactions on Design Automation of Electronic Systems (TODAES)**, ACM New York, NY, USA, v. 1, n. 2, p. 145–204, 1996.

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

FAN, L.; WU, C. Fpga technology mapping with adaptive gate decomposition. In: **Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays**. [S.l.: s.n.], 2023. p. 135–140.

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.

LI, N.; DUBROVA, E. Aig rewriting using 5-input cuts. In: IEEE. **2011 IEEE 29th International Conference on Computer Design (ICCD)**. [S.l.], 2011. p. 429–430.

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

MACHADO, L. et al. Iterative remapping respecting timing constraints. In: IEEE. **2013 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)**. [S.l.], 2013. p. 236–241.

MANOHARARAJAH, V.; BROWN, S. D.; VRANESIC, Z. G. Heuristics for area minimization in lut-based fpga technology mapping. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 25, n. 11, p. 2331–2340, 2006.

MARTINELLO, O. et al. Kl-cuts: a new approach for logic synthesis targeting multiple output blocks. In: IEEE. **2010 Design, Automation & Test in Europe Conference & Exhibition (DATE 2010)**. [S.l.], 2010. p. 777–782.

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

MISHCHENKO, A.; BRAYTON, R.; CHATTERJEE, S. Boolean factoring and decomposition of logic networks. In: IEEE. **2008 IEEE/ACM International Conference on Computer-Aided Design**. [S.l.], 2008. p. 38–44.

MISHCHENKO, A.; BRAYTON, R. K. Sat-based complete don't-care computation for network optimization. In: IEEE. **Design, Automation and Test in Europe**. [S.l.], 2005. p. 412–417.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. Dag-aware aig rewriting a fresh look at combinational logic synthesis. In: **Proceedings of the 43rd annual Design Automation Conference**. [S.l.: s.n.], 2006. p. 532–535.

MISHCHENKO, A.; CHATTERJEE, S.; BRAYTON, R. Improvements to technology mapping for lut-based fpgas. In: **Proceedings of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays**. [S.l.: s.n.], 2006. p. 41–49.

MISHCHENKO, A. et al. Improvements to combinational equivalence checking. In: **Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design**. [S.l.: s.n.], 2006. p. 836–843.

MISHCHENKO, A. et al. Combinational and sequential mapping with priority cuts. In: IEEE. **2007 IEEE/ACM International Conference on Computer-Aided Design**. [S.l.], 2007. p. 354–361.

MISHCHENKO, A.; WANG, X.; KAM, T. A new enhanced constructive decomposition and mapping algorithm. In: **Proceedings of the 40th annual Design Automation Conference**. [S.l.: s.n.], 2003. p. 143–148.

MOORE, G. E. Cramming more components onto integrated circuits. **Electronics**, v. 38, n. 8, 1965.

NETO, W. L. et al. Improving lut-based optimization for asics. In: **Proceedings of the 59th ACM/IEEE Design Automation Conference**. [S.l.: s.n.], 2022. p. 421–426.

PAN, P.; LIN, C.-C. A new retiming-based technology mapping algorithm for lut-based fpgas. In: **Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays**. [S.l.: s.n.], 1998. p. 35–42.

POSSANI, V. N. **Parallel algorithms for scalable logic synthesis & verification**. Thesis (PhD) — Universidade Federal do Rio Grande do Sul, 2019.

REIS, A. Towards a vlsi design flow based on logic computation and signal distribution. In: **Proceedings of the 2018 International Symposium on Physical Design**. [S.l.: s.n.], 2018. p. 58–59.

REIS, A. I.; MATOS, J. M. Physical awareness starting at technology-independent logic synthesis. **Advanced Logic Synthesis**, Springer, p. 69–101, 2018.

RIENER, H. et al. On-the-fly and dag-aware: Rewriting boolean networks with exact synthesis. In: IEEE. **2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2019. p. 1649–1654.

RIENER, H.; MISHCHENKO, A.; SOEKEN, M. Exact dag-aware rewriting. In: IEEE. **2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)**. [S.l.], 2020. p. 732–737.

SENTOVICH, E. et al. **SIS: A System for Sequential Circuit Synthesis**. [S.l.], 1992. Available from Internet: <http://www2.eecs.berkeley.edu/Pubs/TechRpts/1992/2010.html>.

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.

SÖRENSSON, N. Minisat 2.2 and minisat++ 1.1. **A short description in SAT Race**, Citeseer, v. 2010, 2010.

TANG, R. et al. Maximum fanout-free window enumeration: Towards the local multi-output sub-structure synthesis. In: **Proceedings of the 32nd International Workshop on Logic & Synthesis (IWLS)**. EPFL, Lausanne, Switzerland: [s.n.], 2023. Informal Proceedings.

TSEITIN, G. S. On the complexity of derivation in propositional calculus. **Automation of reasoning: 2: Classical papers on computational logic 1967–1970**, Springer, p. 466–483, 1983.

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

YANG, W.; WANG, L.; MISHCHENKO, A. Lazy man's logic synthesis. In: **Proceedings of the International Conference on Computer-Aided Design**. [S.l.: s.n.], 2012. p. 597–604.

ZHU, X. et al. A database dependent framework for k-input maximum fanout-free window rewriting. In: IEEE. **2023 60th ACM/IEEE Design Automation Conference (DAC)**. [S.l.], 2023. p. 1–6.

ZILIC, Z.; VRANESIC, Z. G. Using bdds to design ulms for fpgas. In: **Proceedings of the 1996 ACM fourth international symposium on Field-programmable gate arrays**. [S.l.: s.n.], 1996. p. 24–30.

APPENDIX A — ALGORITHM WITH OPTMIZATIONS

Algorithm A.1: KL-Cuts enumeration Algorithm (Complete).

```
1  kl_cuts klEnumeration (aig, k, n_best)
2      all_kl_cuts ← ∅;
3      k_cuts ← enumerate_k_cuts();
4      for ( each k_cut in k_cuts ) do
5          if ( k_cut ≠ self-cut ) then
6              kl_cut = createKL(k_cut.leaves);
7              all_kl_cuts[k_cut.sign] ← kl_cut; // map structure
8              markSupport(aig, k_cut);
9      // propagates the kl-cut
10     for ( each aig node ≠ PI ) do
11         node.klcuts ← node.f_1.support ∩ node.f_2.support;
12         node.support ← node.support ∪ node.klcuts;
13     // register the outputs and internal node in each kl-cut
14     for ( each aig node ≠ PI ) do
15         if ( node ∈ POs ) then
16             // add this node as output for every KL-cut in node.klcuts
17         else
18             if ( node.support = ∅ ) then
19                 // for each fanout node, remove the kl-cuts which
20                 // have this node as a leave
21             else
22                 if ( node.klcuts.size() ≠ node.support.size() ) then
23                     for ( each kl_cut of node.klcuts ) do
24                         if ( kl_cut ∉ (at least one of node.fannouts.klcuts) )
                             then
25                             all_kl_cuts[kl_cut].outputs.add(node);
26     return all_kl_cuts
```