МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Києво-Могилянська академія»
Факультет інформатики
Кафедра математики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему: **«Boolean network optimization by stochastic rewiring»**

Виконала: студентка 4-го року навчання
Спеціальності
113 Прикладна математика
Черевко Крістіна Олександрівна

Керівник:
професор, д.ф.-м.н *Олійник Б.В.*

Рецензент _____
(*прізвище та ініціали*)

Кваліфікаційна робота захищена з
оцінкою_____

Секретар ЕК_____
«_____»_____ 2024 р.

Київ – 2024

Abstract

This bachelor thesis introduces novel algorithms for the area minimization of Multi-Input And-Inverter Graphs (MAIGs) within the field of logic synthesis. The primary focus is on the implementation of a new method named "Boolean network optimization by stochastic rewiring." This approach significantly diverges from traditional techniques like algebraic factoring and AIG rewriting by adopting a global strategy to add and remove wires at various locations within the circuit. The method relies on a simple randomization strategy to produce structural variations in the AIGs. Additionally, the thesis formulates and proves a new criterion for acceptable fanins, which ensures valid insertions and deletions without compromising the functionality of the circuit. Experimental results validate the effectiveness of the proposed algorithm in optimizing benchmark circuits.

This work is important as it addresses the growing demand for smaller, more efficient circuits in modern electronics, potentially reducing manufacturing costs and enhancing performance.

Keywords: logic synthesis, Multi-input And-Inverter Graphs, area minimization, Boolean network optimization, stochastic rewiring, redundancy addition and removal.

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра математики

ЗАТВЕРДЖУЮ

Зав. кафедри математики,

доцент, кандидат фіз.-мат. наук

_____ *Чорней Р. К.*

(підпис)

«_____»_____ 2024 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

для кваліфікаційної роботи

студентці 4-го курсу, факультету інформатики

Черевко Крістіні Олександрівні

1. **Тема роботи:** «Boolean network optimization by stochastic rewiring». керівник роботи Олійник Б.В., професор, доктор фіз.-мат. наук затверджені наказом вищого навчального закладу від «17» жовтня 2023 року №_____.

2. Строк подання студентом роботи до 27 травня 2024 р.

3. **План роботи:**

    1. Означення.
    2. Розгляд існуючих алгоритмів оптимізації булевих схем.
    3. Розробка нового алгоритму.
    4. Додавання евристик.
    5. Верифікація оптимізованих результатів.

Тема: Boolean network optimization by stochastic rewiring.

Календарний план виконання роботи:

№ п/п	Назва етапу кваліфікаційної роботи	Термін виконання етапу	Примітка
1.	Отримання теми та завдання на кваліфікаційну роботу.	20.08.2023	
2.	Ознайомлення із завданням кваліфікаційної роботи.	21.09.2023	
3.	Розробка плану та структури роботи.	24.09.2023	
4.	Робота з науковою літературою.	11.10.2023	
5.	Написання теоретичної частини кваліфікаційної роботи.	05.03.2024	
6.	Написання вступу.	07.03.2024	
7.	Побудова та програмування алгоритму. Доведення критерію коректності.	17.04.2024	
8.	Написання практичної частини кваліфікаційної роботи.	25.04.2024	
9.	Попередній аналіз кваліфікаційної роботи. Виправлення помилок та обробка зауважень.	07.05.2024	
10.	Створення презентації.	10.05.2024	
11.	Захист кваліфікаційної роботи.	03.06.2024	

Contents

Introduction

In today's rapidly evolving technological world, the demand for smaller yet more powerful computers is escalating at an unprecedented pace, driving a significant need for research in circuit optimizations.

This surge in demand underscores the critical role of **logic synthesis**, which lies at the heart of digital circuit design. Logic synthesis serves as a pivotal stage in converting a high-level functionality description into an efficient hardware implementation. It is key to meeting the ever-increasing demands for miniaturization and performance enhancement in modern electronics.

At its core, logic synthesis aims to optimize various aspects of circuit performance, including area, power consumption, speed, and delay, while ensuring the correct functionality of the circuit.

Among these optimization goals, **area minimization** stands out as one of the most straightforward and natural criteria. The size of a circuit directly impacts factors such as manufacturing cost, chip area, and overall system complexity. By developing various solutions for area optimization, we address the industry's demand for producing smaller circuits, especially considering the increase in manufacturing costs of silicon chips [7].

Traditionally, a variety of data structures, such as Sum-of-Products (SOPs), Product-of-Sums (POSs), Binary Decision Diagrams (BDDs), and Reduced Ordered Binary Decision Diagrams (ROBDDs) have been used for circuit representation. However, recent years have witnessed a notable shift towards And-Inverter Graphs (AIGs) as the preferred data structure for circuit representation.**And-Inverter Graph**, a compact and canonical representation of Boolean functions, became an essential data structure for circuit representations since they are easy to implement and convenient to manipulate. In this bachelor thesis, the primary focus is on the implementation of an algorithm that uses a multiple-input And-

Inverter graph as its central data structure.

There are many area minimization methods. One of the most widely used logic synthesis methods is called algebraic factoring [3], which looks for larger nodes represented by sum-of-products and tries to break them down (decomposition) while finding shared nodes (factorization). Another popular method is called AIG rewriting [2]. This method reduces AIG size by computing 4-input cuts for a given node, finding the Boolean function of each cut, and re-expressing the node using its Boolean function while possibly reusing other nodes that are already present in the AIG and depend on the same cut variables.

The research presented in this bachelor statement focuses on introducing a **novel algorithm** for area minimization entitled "Boolean network optimization by stochastic rewiring." The new method is conceptually different from algebraic factoring and is more global than AIG rewriting since we add/remove wires in various places.

The **subject of research** in this bachelor's statement is algorithms on logic circuits.

The **object of research** in this bachelor statement is the area optimization of the logic circuit.

The proposed algorithm is a rewiring technique that is based on the notion of redundancy addition and removal. Unlike the traditional work on rewiring [4], [10] our approach uses AIG data structure for representing circuits instead of AND/OR graphs. Also, it relies on a simple randomization strategy to detect possible additions and removals while maintaining the correctness of the circuit instead of using complex rules for computing possibilities of adding and removing fanins. Additionally, our method introduces a new criterion for acceptable fanins, which is based on internal don't-cares computed to check if a fanin can be added or removed. Another difference is that the proposed method targets structural

change rather than immediate improvement. This is particularly important for hard-to-optimize circuits, which can be reduced only after a substantial restructuring.

The outline of this work is as follows. Section 2 introduces the basics of Boolean algebra. Section 3 demonstrates the implementation of the core data structures: truth tables and AIGs. Section 4 elaborates on the algorithmic concept and details the implementation of its main steps. Section 5 introduces the heuristics used in the research. Section 6 delves into the crucial aspect of circuit verification, which is the second most important step in synthesis. Section 7 presents the experimental results obtained from the conducted research. Finally, Section 8 provides a conclusion to the statement, summarizing the key findings and insights.

1 Background

At its core, the proposed algorithm leverages Boolean algebra to minimize the area of digital circuits. This section reviews the key notions, such as Boolean functions, truth tables, logic gates, and AIGs, needed for the understanding of the new algorithm introduced in the subsequent sections. Additionally, the section introduces the concepts of structural and functional hashing.

1.1 Basics of Boolean algebra

This section introduces basic notions of Boolean algebra.

Definition 1.1. A *Boolean variable* is a variable that can take binary values from the set $\mathbb{B} = \{false, true\}$, or alternatively $\mathbb{B} = \{0, 1\}$, depending on the truth assignment.

Let the symbol $^-$ represent an unary operation, while $\wedge$, $\vee$ and $\oplus$ represent binary operation on the $\mathbb{B} = \{0, 1\}$ as follows: For arbitrary x and y in $\mathbb{B}$,

$$\bar{x} = 1 - x$$
$$x \wedge y = x \cdot y$$
$$x \vee y = x + y - x \cdot y$$
$$x \oplus y = x + y \,(\mathrm{mod}\, 2) = x + y - 2xy$$

In the representation above, the symbols $+$, $-$, and $\cdot$ denote ordinary addition, subtraction, and multiplication, respectively.

Definition 1.2. *Boolean function* – is a mapping $f : \mathbb{B}^n \to \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$ – set of Boolean values.

Definition 1.3. *Literal* – is a Boolean variable (x) or its complement ($\bar{x}$).

Definition 1.4. An n-ary *completely specified Boolean function*, denoted as $f : \mathbb{B}^n \to \mathbb{B}$, assigns true or false to every truth assignment on the n input variables.

Definition 1.5. If the output of the function does not change after changing the input, this input is called a *don't-care*.

The don't-care is represented as a symbol "–", "X" or "2". The augmented set of $\mathbb{B} \cup \{-\}$ is denoted as $\mathbb{B}_+$.

Definition 1.6. An n-ary *incompletely specified Boolean function*, denoted as $f : \mathbb{B}^n \to \mathbb{B}_+$, assigns true, false, or don't care to every truth assignment on the n input variables.

Definition 1.7. *Single-output Boolean function* is a boolean function that maps $f : \mathbb{B}^n \to \mathbb{B}$, where $\mathbb{B} = \{0, 1\}$ is a set of Boolean values and n is the number of inputs.

Definition 1.8. *Multiple-output Boolean function* is a boolean function that maps $f : \mathbb{B}^n \to \mathbb{B}^m$, where $\mathbb{B} = \{0, 1\}$ is a set of Boolean values, n is the number of inputs and m is the number of outputs.

1.2 And-inverter graph

In this section, we define the main data structure used in this work.

Definition 1.9. The *undirected graph* is a pair $G = (V, E)$, where

$$V = V(G) \text{ is a set of vertices (nodes),}$$
$$E = E(G) \subset V^{(2)} = \{\{a, b\} \mid a, b \in V\} \text{ is a set of edges.}$$

Definition 1.10. A *directed graph* is a pair $G = (V, E)$, where

$$V = V(G) \text{ is a set of vertices (nodes),}$$
$$E = E(G) \text{ is a set of ordered pairs of vertices, called arcs or directed edges.}$$

Definition 1.11. A *directed acyclic graph* (DAG) is a directed graph with no directed cycles.

Definition 1.12. [5] *Boolean network* is defined as a directed acyclic graph (DAG), where the nodes represent logic gates and the edges represent the connections (wires) between these gates.

Definition 1.13. *And-inverter graph* (AIG) is a Boolean network composed of two-input and-gates and inverters.

Definition 1.14. *Multi-input And-inverter graph* (MAIG) is a Boolean network composed of multi-input and-gates and inverters.

In this work, we use multi-input And-inverter graphs.

Definition 1.15. *Fanins* of the node n are the nodes that are driving n.

Definition 1.16. *Fanouts* of the node n are the nodes driven by n.

Definition 1.17. *Primary inputs* (PIs) are nodes without fanins.

Definition 1.18. *Primary outputs* (POs) are a subset of the network nodes that deliver the functionality of the network to its environment.

Definition 1.19. The *size* (area) of the network is the number of its nodes.

Definition 1.20. *Depth* (delay) is the length (number of nodes) of the longest path from the primary inputs to the primary outputs.

In this research, our focus lies specifically on **area minimization**, as opposed to optimizing for delay or other parameters of the circuit.

Definition 1.21. [9] The set of fanins of node i is denoted as $FI(i)$, the set of fanouts of node i is denoted as $FO(i)$. The *transitive fanins* TFI(i) and *transitive fanouts* TFO(i) of a node i are defined recursively as

$$TFI(i) = \{k \in V \mid k = i, \text{ or } k \in FI(j) \text{ for } j \in TFI(i)\}$$

and

$$TFO(i) = \{k \in V \mid k = i, \text{ or } k \in FO(j) \text{ for } j \in TFO(i)\}$$

respectively.

Figure 1.1 shows primary inputs/outputs and TFO and TFI for the green node.

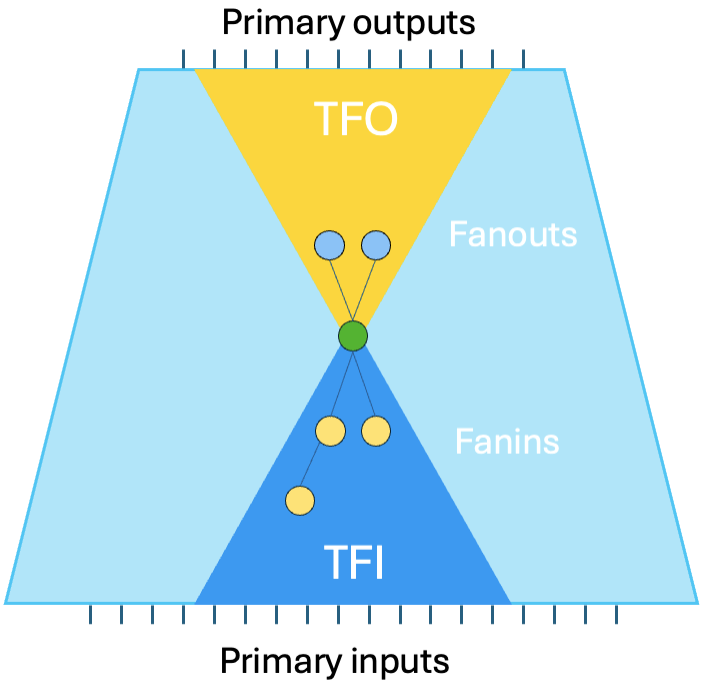

Figure 1.1: TFO and TFI for the green node

Definition 1.22. For a completely specified function f, we define its ***onset*** $f^{\text{on}} = \{a \in \mathbb{B}^n \mid f(a) = 1\}$ and ***offset*** $f^{\text{off}} = \{a \in \mathbb{B}^n \mid f(a) = 0\}$.

Definition 1.23. [11] The ***don't-care set*** of the function f is a set of input combinations for which value of f can be either 0 or 1.

Definition 1.24. The ***care set*** of the function f is the complement of the don't-care set.

Don't-cares in a logic function refer to input patterns where the output can be either 0 or 1. The collection of these don't-care patterns forms the function's don't-care set and all the remaining patterns forming the care set. Given their complementary nature, our focus in the rest of the work is solely on the **care set**.

1.3 Structural and functional hashing

Constant propagation involves identifying constant values within the circuit and propagating these constants throughout the circuit to simplify its structure and improve efficiency. For instance, suppose we have an expression $F = a \wedge 1$. The constant 1 value can be propagated to get $F = a$.

Structural hashing involves propagation of all constants and ensures that for any pair of nodes, there exists a maximum of one two-input and-node with them as fanins (up to a permutation). Structural hashing is applied during AIG construction to reduce the AIG size.

Functional hashing of AIGs guarantees that each internal node possesses a unique Boolean function up to complement. This technique results in a "semi-canonical" AIG representation, where no two nodes within the AIG share the same global function. However, the same global functions across different AIGs can be realized using different circuit structures.

2 Data structures

This section provides an overview of how Boolean functions are represented within the scope of this work. It explores two primary representations: truth tables and multi-input And-Inverter graph (MAIG) representations.

2.1 Boolean function representation

In our optimization package, we work with the Boolean functions represented as truth tables. Each truth table for a function is provided as a binary or hexadecimal string. Table 2.1 shows the truth table for the Boolean function $F = abc$ in the binary notation. The hexadecimal string "80" is also a valid representation of this function.

Since the size of memory used for the truth table representation grows exponentially in the number of variables, we work only with small and medium Boolean functions, that is, functions of 16 or fewer variables.

a	b	c	$f(a, b, c)$
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	0
0	0	1	0
1	0	1	0
0	1	1	0
1	1	1	1

Table 2.1: $F = abc$

In our framework, the truth table is an array of 64-bit words. This data

structure is designed to manage boolean functions truth tables efficiently.

Listing 1 demonstrates a method that initializes a truth table element represented by an array of word values, where the word is an alias for `long long unsigned` data type.

```
void Tt_ElemInit(pTruth: array of word, iVar: number of PIs, nWords: integer) {
  int k;
  if ( iVar < 6 )
    for ( k = 0; k < nWords; k++ )
      // Use pre-defined truth values for iVar less than 6
      pTruth[k] = s_Truths6[iVar];
  else
    for ( k = 0; k < nWords; k++ )
      // Generate truth values dynamically for iVar 6 or greater
      pTruth[k] = (k & (1 << (iVar-6))) ? ~(word)0 : 0;
}
```

Listing 1: Method to initiate truth table

Listing 2 shows how basic Boolean operations, such as AND, OR XOR, NOT, and the check for equality are performed using truth tables. These helper methods are extensively utilized throughout the framework, particularly in the verification process and care set calculations, which will be considered later in this work.

```
static inline void Tt_Not( word * pOut, word * pIn, int nWords ) {
    int w;
    for ( w = 0; w < nWords; w++ ) {
        pOut[w] = ~pIn[w]; // invert bit
    }
}

static inline void Tt_And( word * pOut, word * pIn1, word * pIn2, int nWords ) {
    int w;
    for ( w = 0; w < nWords; w++ ) {
        pOut[w] = pIn1[w] & pIn2[w]; // perform AND operation
    }
}
```

```
static inline void Tt_OrXor( word * pOut, word * pIn1, word * pIn2, int nWords ) {
    int w;
    for ( w = 0; w < nWords; w++ ) {
        // perform bitwise XOR operation, then append
        // current truth table to final result through OR operator
        pOut[w] |= pIn1[w] ^ pIn2[w];
    }
}

static inline int Tt_Equal( word * pIn1, word * pIn2, int nWords ) {
    int w;
    for ( w = 0; w < nWords; w++ )
        if ( pIn1[w] != pIn2[w] ) // check if bits are not equal
            return 0;
    return 1;
}
```

Listing 2: Basic Boolean operations

The following procedure (Listing 3) is an essential step in computing truth tables of the nodes in multi-input AIG. To compute the node's truth table we apply AND operation to all its fanins' truth tables. If the fC parameter is equal to 1 we apply AND operation to complemented truth tables of fanins.

```
static inline void Tt_Sharp( word * pOut, word * pIn, int fC, int nWords ) {
  int w;
  if ( fC )
    for ( w = 0; w < nWords; w++ )
      pOut[w] &= ~pIn[w];
  else
    for ( w = 0; w < nWords; w++ )
      pOut[w] &= pIn[w];
}
```

Listing 3: Node truth table computation

2.2 And-inverter graph representation

In our optimization framework, the input data is a Boolean function represented as two-input And-Inverter Graph (AIG). The multi-input AIG (MAIG) is a network composed of inverters and AND gates, where each gate takes multiple inputs and produces one output. Since each gate is uniquely characterized by its fanins, we can represent a MAIG of any size by an array containing fanin arrays for each node.

All nodes in MAIG belong to one of the three categories:

- If a node doesn't have any fanins, it means it's a **primary input**.
- If a node has one fanin, it's a **primary output**.
- If a node has two or more fanins, it's an **internal node** responsible for some computation.

The code snippet presented in Listing 4 illustrates how the multi-input AIG (MAIG) data structure is implemented. This data structure encompasses the number of inputs and outputs of the Boolean function. The total number of nodes in the MAIG is denoted by the nObjs parameter, with the maximum capacity constrained by the nObjsAlloc parameter.

nWords indicates the size of the truth table, while nTravIds represents the traversal ID used for marking visited nodes. The pTravIds array holds traversal IDs for all nodes. pCopy is utilized for copying purposes throughout the codebase.

For storage-related tasks, pTruths holds truth tables (the first truth table is the main one, the second truth table is used to temporarily compute care sets and the third truth table stores a copy of correct primary output functions), pCare stores care sets of the nodes, and pProd contains functions associated with the nodes.

vOrder denotes the order in which nodes are traversed, while vTfo represents the set of nodes in the transitive fanout cone for each node. Lastly, pvFans is an array of fanins for every node in the structure.

```
typedef struct maig_ {
  int     nIns;          // primary inputs
  int     nOuts;         // primary outputs
  int     nObjs;         // all objects
  int     nObjsAlloc;    // allocated space
  int     nWords;        // the truth table size
  int     nTravIds;      // traversal ID counter
  int *   pTravIds;      // traversal IDs
  int *   pCopy;         // temp copy
  int *   pRefs;         // reference counters
  word *  pTruths[3];    // truth tables
  word *  pCare;         // care set
  word *  pProd;         // product
  vi *    vOrder;        // node order
  vi *    vTfo;          // transitive fanout cone
  vi *    pvFans;        // the array of objects' fanins
} maig;
```

Listing 4: Multi-input AIG data structure

3 Algorithm idea

The main idea of the algorithm is straightforward and can be summarized as randomized redundancy addition and removal (RAR). This algorithm comprises two stages: node expansion and node reduction.

During the first stage, a predefined number of stochastically selected fanins are added to nodes of the MAIG. This number is an input parameter of the program.

Subsequently, in the second stage, fanins of nodes are deleted in a randomizer order, aiming to restructure and simplify the MAIG as much as possible. Following each step, the algorithm verifies that the new circuit remains functionally equivalent to the initial one.

In our approach, we employ a trial-and-error method, supplemented by different heuristics, which is notably simpler than the approaches used in previous work [4], [10].

In the following subsections, we will delve into the specifics of node expansion and reduction. This will include illustrating examples and providing pseudocode for these functions. In the next section, we will provide a comprehensive description of the heuristics utilized and their implementation details.

3.1 Node expansion

During the node expansion stage, the process involves attempting to insert new fanins for all internal nodes in the Multi-valued And-Inverter Graph (MAIG).

Initially, a randomized order of MAIG's nodes is generated. Then, for each node, we compute the transitive fanout (TFO) and define a subset of fanins that are not part of the transitive fanout. It's important to avoid transitive fanouts during node expansion since they might create combinational loops.

Additionally, we exclude transitive fanins to prevent redundant duplication

within the node. To ensure that the insertion of fanins does not compromise the correctness of the Multi-Input And-Inverter graph, the fanins must satisfy the criterion for acceptable fanins outlined in section 5.

Figure 3.1 illustrates how the MAIG would appear after the insertion of a new fanin (picked outside of the transitive fanout) to the node 6 (n6) of the MAIG.

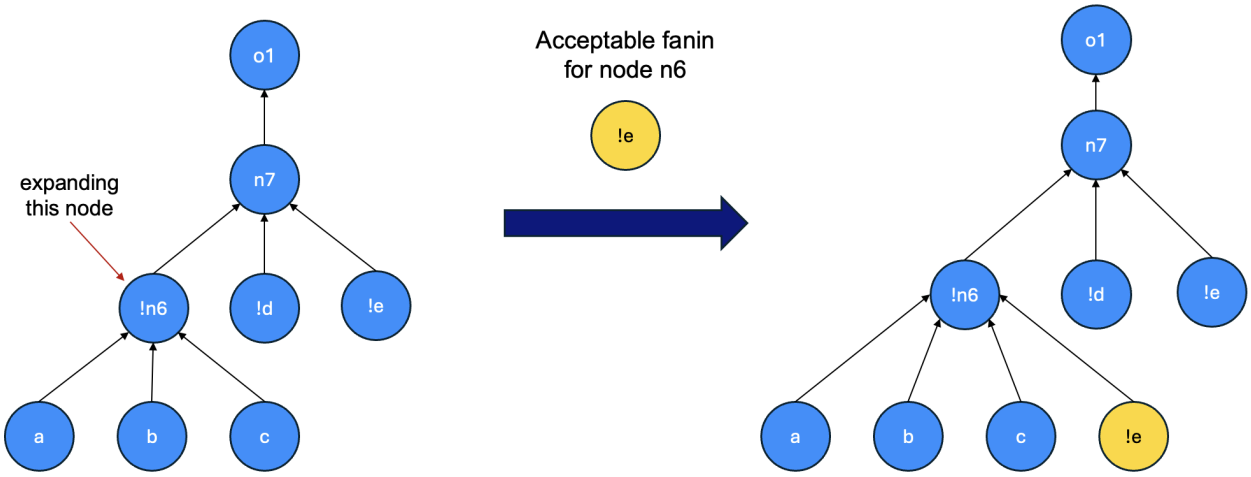

Figure 3.1: Node expansion

3.2 Implementation of node expansion

To understand node expansion better, let's look at the implementation of this procedure at the level of pseudocode.

The following pseudocode (Listing 5) shows how the expansion of one node is performed. First, the care set of the current node iObj is computed. This marks the transitive fanout of the node iObj. The immediate fanins are also marked to make sure they are not added again. Subsequently, the node's onset is computed.

The next step involves iterating through randomly ordered fanins that are not in the TFO of the node, which we aim to expand with new fanins. During each

iteration, the algorithm verifies whether adding the current fanin to the MAIG would affect the graph's functionality by checking if it meets the criteria outlined in section 5. If a new fanin node can be added without compromising the MAIG's functionality, the algorithm invokes the Maig_AppendFanin function.

The procedure continues until a target number of fanins (nAddedMax) has been inserted, at which point the process concludes.

```
static inline int Rw_ExpandOne( maig * p, int iObj, int nAddedMax )
{
  int i, k, n, iLit, nFans = Maig_ObjFaninNum(p, iObj), nAdded = 0;
  word * pCare = Maig_ComputeCareSet( p, iObj ); // compute care set of iObj
  // mark node's fanins
  Maig_ForEachObjFanin( p, iObj, iLit, k )
    p->pTravIds[Lit2Var(iLit)] = p->nTravIds;
  word * pOnset = Maig_ObjTruth( p, iObj, 0 ); // compute the onset of iObj
  Tt_Sharp( pOnset, pCare, 0, p->nWords );
  // create a random order of fanin candidates
  Vi_Shrink( p->vOrderF, 0 );
  Maig_ForEachInputNode( p, i )
    if ( condition for not TFO node i) // check if this node is NOT in the TFO
      Vi_Push( p->vOrderF, i );
  Vi_Randomize(p->vOrderF);
  // iterate through candidate fanins (nodes that are not in the TFO of iObj)
  Vi_ForEachEntry( p->vOrderF, i, k ) {
    // new fanin can be added if its offset does not intersect with the node's onset
    for ( n = 0; n < 2; n++ )
    if ( !Tt_IntersectC(pOnset, Maig_ObjTruth(p, i, 0), !n, p->nWords) ) {
        Maig_AppendFanin(p, iObj, Var2Lit(i, n));
    }
    if ( nAdded == nAddedMax ) break; // condition of completion
  }
  // updating TFO of the node
  Maig_TruthUpdate( p, p->vTfo );
  return nAdded;
}
```

Listing 5: Fanin addition

3.3 Node reduction

After inserting a specified number of fanins, the algorithm proceeds to the second stage, which involves node reduction.

The initial approach to node reduction involved removing nodes by eliminating their edges and then performing constant propagation. However, we later refined our strategy to prioritize the removal of fanins rather than entire nodes. Each node's fanins are evaluated based on their usability, with those referenced more frequently receiving higher priority for removal. The deletion process follows a reverse order, targeting fanins with few or no references first.

During the reduction stage, the algorithm iterates through the ordered fanins of the node being reduced, aiming to remove them. This iterative process aims to optimize circuit efficiency by simplifying and streamlining interconnected nodes.

Figure 3.2 illustrates the reduction process for node n6, providing visual illustration of the presented concepts.

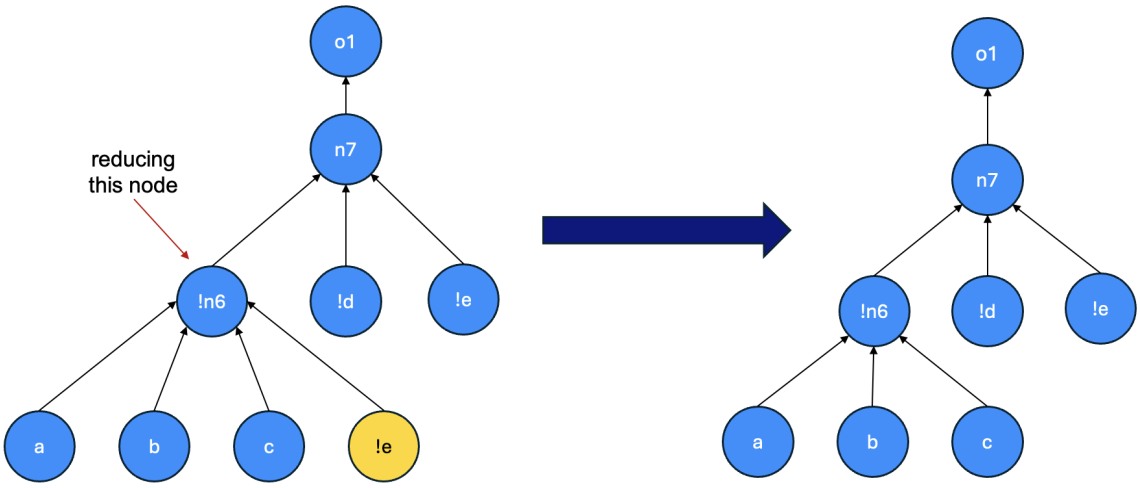

Figure 3.2: Node reduction

3.4 Implementation of node reduction

The following subsection explains the implementation of node reduction using the pseudocode.

For the figure above, the reduction procedure prioritizes all fanins of node n6 and attempts to delete them in reverse order. During deletion, the algorithm verifies if the function of the reduced node matches that of node n6 on a care set. After removing all possible fanins the node is updated. The final step of the reduction algorithm is to verify the correctness of the new circuit. Upon completion of the procedure, a reduced MAIG is obtained.

In our example (Figure 3.2), the verification confirms that the new MAIG is equivalent to the initial one.

Listing 6 provides pseudocode outlining the reduction process.

```
static inline int Rw_ReduceOne( maig * p, int iObj, int fOnlyConst, int fOnlyBuffer )
{
  int i, n, k, iLit, nFans = Maig_ObjFaninNum(p, iObj);
  // compute care set of iObj
  word * pCare = Maig_ComputeCareSet( p, iObj );
  if ( Rw_CheckConst(p, iObj, pCare) )
    return nFans;
  if ( fOnlyConst )
    return 0;
  if ( nFans == 1 )
    return 0;
  // if one fanin can be used, take it
  word * pFunc = Maig_ObjTruth( p, iObj, 0 );
  Maig_ForEachObjFanin( p, iObj, iLit, k ) {
    Tt_DupC( p->pProd, Maig_ObjTruth(p, Lit2Var(iLit), 0), Lit2C(iLit), p->nWords );
    if ( Tt_EqualOnCare(pCare, pFunc, p->pProd, p->nWords) ) {
      Maig_ObjDeref( p, iObj );
      Vi_Fill( p->pvFans+iObj, 1, iLit );
      Maig_ObjRef( p, iObj );
      Maig_TruthUpdate( p, p->vTfo );
      return nFans-1;
```

```
    }
  }
  if ( fOnlyBuffer )
    return 0;
  // create order of fanins with high reference fanins first
  Vi_Shrink( p->vOrderF, 0 );
  Maig_ForEachObjFanin( p, iObj, iLit, k )
    if ( p->pRefs[Lit2Var(iLit)] > 2 )
      Vi_Push( p->vOrderF, iLit );
  Maig_ForEachObjFanin( p, iObj, iLit, k )
    if ( p->pRefs[Lit2Var(iLit)] == 2 )
      Vi_Push( p->vOrderF, iLit );
  Maig_ForEachObjFanin( p, iObj, iLit, k )
    if ( p->pRefs[Lit2Var(iLit)] == 1 )
      Vi_Push( p->vOrderF, iLit );
  assert( Vi_Size(p->vOrderF) == nFans );
  // try to remove fanins starting from the end of the list
  for ( n = Vi_Size(p->vOrderF)-1; n >= 0; n-- ) {
    int iFanin = Vi_Drop(p->vOrderF, n);
    word * pProd = Maig_TruthSimNodeSubset2( p, iObj, p->vOrderF, Vi_Size(p->vOrderF) );
    if ( !Tt_EqualOnCare(pCare, pFunc, pProd, p->nWords) )
      Vi_Push(p->vOrderF, iFanin);
  }
  assert( Vi_Size(p->vOrderF) >= 1 );
  // update the node if it is reduced
  if ( Vi_Size(p->vOrderF) < nFans ) {
    Maig_ObjDeref(p, iObj);
    Vi_Shrink( p->pvFans+iObj, 0 );
    Vi_ForEachEntry( p->vOrderF, iLit, k )
      Vi_PushOrder( p->pvFans+iObj, iLit );
    Maig_ObjRef(p, iObj);
    Maig_TruthUpdate( p, p->vTfo );
    return nFans-Vi_Size(p->vOrderF);
  }
  return 0;
}
```

Listing 6: Node reduction

3.5 Structural hashing and constant propagation

Following the reduction procedure, the algorithm performs constant propagation and structural hashing to remove redundant nodes and ensure optimal structure.

Below are two examples illustrating how constant propagation and structural hashing are implemented by modifying the array of literals.

The Maig_StrashNode (Listing 7) procedure takes an MAIG as input and returns its duplicate with small optimizations, performing both constant propagation and structural hashing.

Example 3.1. In Figure 3.3, the n7 node has a redundant constant 1 value. The rewriting procedure simply deletes redundant fanin.

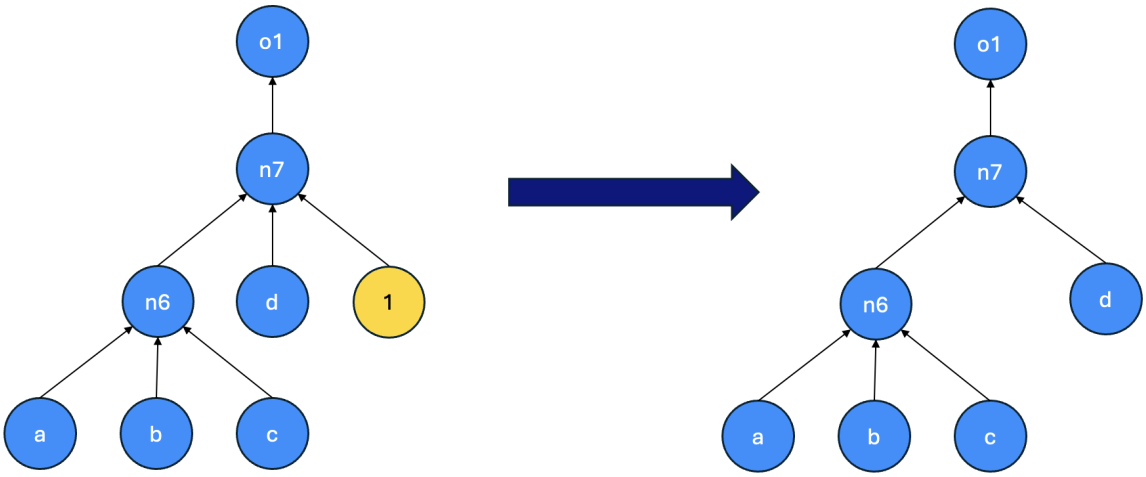

Figure 3.3: Constant propagation

Example 3.2. In Figure 3.4, the first array has two structurally equal nodes (have identical corresponding literals). The structural hashing procedure (Listing 7) iterates through all internal nodes while checking the equality of their fanin literals.

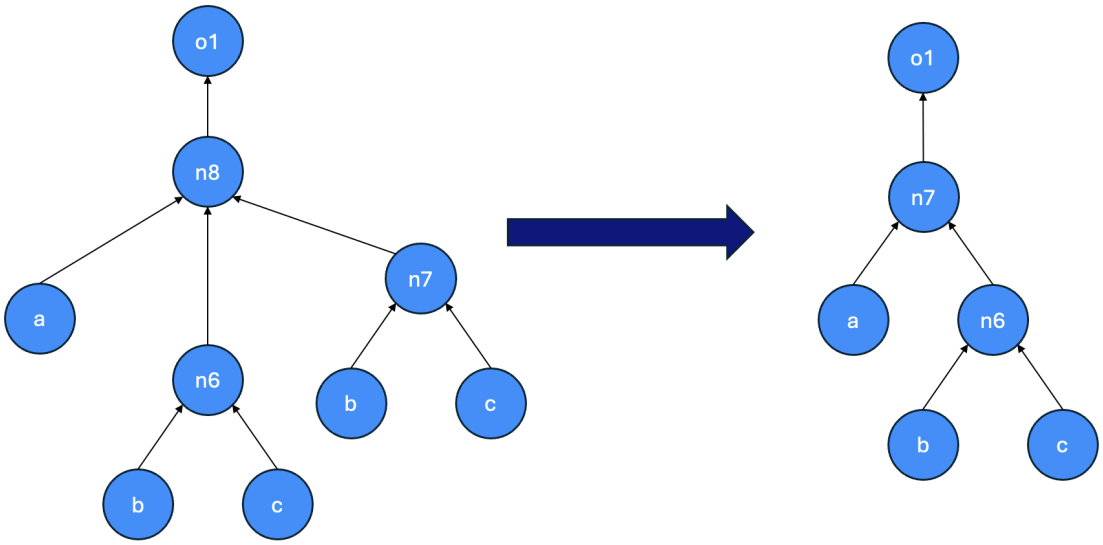

Figure 3.4: Structural hashing

```
// this duplicator creates two-input nodes, propagates constants, and does structural hashing
static inline int Maig_StrashNode( maig * p, int l0, int l1 )
{
  int lit0 = MinInt(l0, l1), lit1 = MaxInt(l0, l1), i;
  Maig_ForEachNode( p, i ) // loop through all nodes of MAIG
    if ( Maig_ObjFanin0(p, i) == lit0 && Maig_ObjFanin1(p, i) == lit1 )
      return Var2Lit(i, 0);
  return -1;
}
```

Listing 7: Structural hashing procedure

4 Heuristics

In the previous section, we explored the main idea of the algorithm and explored implementation of two core stages - node expansion and node reduction. In this section, we describe various heuristics that aim to increase the algorithm's efficiency and reduce its runtime.

4.1 Logic sharing

Our first heuristic is logic sharing, that is, an extraction of shared nodes. The goal of this heuristic is to reduce area of the given multi-input AIG. One difference between structural hashing and logic sharing is that structural hashing is performed on-the-fly in all cases, whereas logic sharing is created intentionally by the algorithm.

Figure 4.1 provide visual illustration of the logic sharing process applied to a node featuring the function $f_n = b \wedge c$.

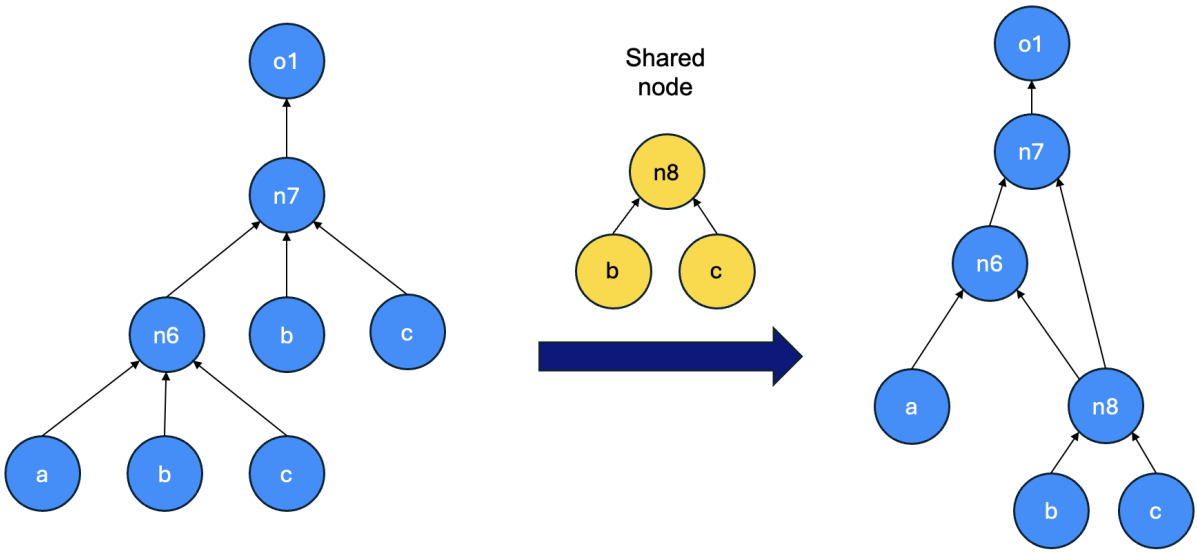

Figure 4.1: Logic sharing

On a high level, the logic sharing works with the following steps:

1. Initially, for a given MAIG, we specify the maximum number of shared nodes that can be extracted (nNewNodesMax). Subsequently, a procedure is invoked to attempt the extraction of the specified number of nodes.

2. The function Rw_FindShared (Listing 9) facilitates the extraction of shared fanin pairs and returns the number of pairs extracted. It makes nNewNodesMax iterations. During each iteration, the function calls Rw_FindPairs (see Listing 8), responsible for identifying sets of fanin pairs occurring more than once. Once the pair search concludes, the algorithm identifies the most frequently occurring pair, deemed the best candidate, and extracts it.

3. The function Rw_FindPairs (Listing 8) traverses all nodes and, for each node, identifies all of its fanin pairs. It then marks these pairs in a bit matrix to denote those previously encountered. The vPairs array is constructed based on a simple principle: each pair consists of three integers, where the first two represent the elements of a fanin pair, and the third integer indicates the number of occurrences of this pair in the MAIG.

```
static inline vi * Rw_FindPairs( maig* p, word* storage, int nWords )
{
  vi * vPairs = Vi_Alloc( 30 );
  int i, f1, f2, iFan1, iFan2;
  // iteration over MAIG nodes
  Maig_ForEachNode( p, i )
  {
    vi * vFans = p->pvFans+i;
    // iteration over node's fanins
    Vi_ForEachEntry( vFans, iFan1, f1 ) {
      word* pRowFan1 = storage + iFan1*nWords;
      Vi_ForEachEntryStart( vFans, iFan2, f2, f1+1 )
        // check if this pair was already seen
        if ( Tt_GetBit(pRowFan1, iFan2) )
          // if this pair exists, we add existing pair to array of pairs
```

```
          Rw_AddPair( vPairs, iFan1, iFan2 );
        else
          // set bit in bit matrix to show that there is such pair
          Tt_SetBit(pRowFan1, iFan2);
    }
  }
  return vPairs;
}
```

Listing 8: Finding those fanin pairs that appear more than once

```
static inline int Rw_FindShared( maig * p, int nNewNodesMax )
{
  // expand the MAIG if number of existing nodes and nodes
  // that need to be extracted exceeds the maximum capacity
  if ( p->nObjs + nNewNodesMax > p->nObjsAlloc )
  {
    p->nObjsAlloc = p->nObjs + nNewNodesMax;
    p->pCopy      = (int *)realloc( (void *)p->pCopy,  sizeof(int)*p->nObjsAlloc );
    p->pvFans     =  (vi *)realloc( (void *)p->pvFans, sizeof(vi) *p->nObjsAlloc );
    memset( p->pCopy+p->nObjs,  0, sizeof(int)*(p->nObjsAlloc-p->nObjs) );
    memset( p->pvFans+p->nObjs, 0, sizeof(vi) *(p->nObjsAlloc-p->nObjs) );
  }
  assert( sizeof(word) == 8 );
  // calculate how many words are needed to have a
  // bitstring with one bit for each literal
  int i, nWords = (2*p->nObjsAlloc + 63)/64;
  int nBytesAll = sizeof(word)*nWords*2*p->nObjsAlloc;
  word * pSto = (word *)malloc(nBytesAll); vi * vPairs; // bit matrix
  // the algorithm extracts only one shared node at a time
  for ( i = 0; i < nNewNodesMax; i++ ) {
    memset( pSto, 0, nBytesAll );
    p->nObjs -= i;
    // finding pairs of shared nodes
    vPairs = Rw_FindPairs( p, pSto, nWords );
    p->nObjs += i;
    // if there are shared nodes, then the best pair
    // (most frequently appeared) is extracted
    if ( Vi_Size(vPairs) > 0 )
```

```
      Rw_ExtractBest( p, vPairs );
    int Size = Vi_Size(vPairs);
    Vi_Stop( vPairs );
    if ( Size == 0 )
      break;
  }
  free( pSto );
  return i;
}
```

Listing 9: Extract shared pairs

Definition 4.1. A *topological order* is a linear ordering of nodes in a directed acyclic graph (DAG). In this order, nodes are visited from primary inputs (PIs) to primary outputs (POs), ensuring that each node is visited after all its fanins have been visited.

To simplify implementation and optimize computation time, all shared nodes are appended to the end of the Multi-Input And-Inverter graph. These newly inserted nodes are called "hidden" since they are not taken into account in the total number of nodes of MAIG. After extracting the target number of nodes, the algorithm invokes the procedure (Listing 10) to restore the topological order of the MAIG.

```
static inline void Maig_DupDfs_rec( maig * pNew, maig * p, int iObj )
{
    int i, iLit;
    // return if current node is already marked
    if ( p->pCopy[iObj] >= 0 )
        return;
    // create fanins for a given node
    Maig_ForEachObjFanin( p, iObj, iLit, i )
        Maig_DupDfs_rec( pNew, p, Lit2Var(iLit) );
    assert( p->pCopy[iObj] < 0 ); // catching the combinational loop
    assert( Maig_ObjFaninNum(p, iObj) > 0 );
```

```
    // create current node
    p->pCopy[iObj] = Maig_AppendObj(pNew);
    // append newly created fanins to the current node
    Maig_ForEachObjFanin( p, iObj, iLit, i )
        Maig_AppendFanin(pNew, p->pCopy[iObj], Lit2LitV(iLit));
}

static inline maig * Maig_DupDfs( maig * p )
{
    maig * pNew = Maig_Alloc( p->nIns, p->nOuts, p->nObjsAlloc );
    // the array is filled with -1 to distinguish visited nodes from unvisited
    memset( p->pCopy, 0xFF, sizeof(int)*p->nObjsAlloc ); int i, k, iLit;
    // for each primary input we mark it with it's index
    Maig_ForEachConstInput( p, i )
        p->pCopy[i] = i;
    // for each primary output we call recursive function for it's fanin
    Maig_ForEachOutput( p, i )
        Maig_DupDfs_rec( pNew, p, Lit2Var(fanin0(p, i)) );
    // for each primary output append it's fanin
    Maig_ForEachOutput( p, i )
        Maig_AppendFanin(pNew, Maig_AppendObj(pNew), Lit2LitV(fanin0(p, i)));
    return pNew;
}
```

Listing 10: Function to restore topological order of the MAIG.

4.2 Criterion for acceptable fanins

The very first implementation of the algorithm used a trial-error approach to insert random fanins into the MAIG. While this method may seem straightforward, its effectiveness is limited. For example, we could have tried to insert 10 different fanins into one node, but only one of them would pass the correctness check. To make the algorithm more practical and save computation time, we introduce a criterion for acceptable fanins. This criterion leverages care set computation to determine which fanins can be added or removed without com-

promising the MAIG's functionality.

To calculate the care set of node n, a four-step process is performed:

1. Invert the truth table of node n.
2. Calculate the truth tables of the nodes in the transitive fanout of node n. Since the And-Inverter Graph (AIG) maintains a topological order, it suffices to compute only these truth tables.
3. Utilize bitwise XOR to compare the truth tables of the old output and the new output. In cases where there are multiple outputs, the comparison is done for each output and the results are aggregated using the OR operator.
4. The resulting bit string is the care set of the node n.

The following pseudocode demonstrates the care set computation described above.

```
static inline int Maig_ComputeTfo_rec( maig * p, int iObj )
{
    int k, iLit, Value = 0;
    if ( p->pTravIds[iObj] == p->nTravIds )
        return 1;
    if ( p->pTravIds[iObj] == p->nTravIds-1 )
        return 0;
    Maig_ForEachObjFanin( p, iObj, iLit, k )
        Value |= Maig_ComputeTfo_rec(p, Lit2Var(iLit));
    p->pTravIds[iObj] = p->nTravIds-1+Value;
    if ( Value )
        Vi_Push( p->vTfo, iObj );
    return Value;
}

static inline vi * Maig_ComputeTfo( maig * p, int iObj )
{
  int i;
```

```
  assert( Maig_ObjIsNode(p, iObj) );
  p->nTravIds += 2;
  p->pTravIds[iObj] = p->nTravIds;
  Vi_Fill( p->vTfo, 1, iObj );
  Maig_ForEachConstInput( p, i )
    p->pTravIds[i] = p->nTravIds-1;
  Maig_ForEachOutput( p, i )
    Maig_ComputeTfo_rec( p, i );
  return p->vTfo;
}

static inline word * Maig_ComputeCareSet( maig * p, int iObj )
{
  vi * vTfo = Maig_ComputeTfo( p, iObj );  int i, iTemp;
  Tt_Not( Maig_ObjTruth(p, iObj, 1), Maig_ObjTruth(p, iObj, 0), p->nWords );
  Tt_Clear( p->pCare, p->nWords );
  Vi_ForEachEntryStart( vTfo, iTemp, i, 1 ) {
    Maig_TruthSimNode( p, iTemp );
    if ( Maig_ObjIsPo(p, iTemp) )
      Tt_OrXor( p->pCare, Maig_ObjTruth(p, iTemp, 0), Maig_ObjTruth(p, iTemp, 1), p->nWords );
  }
  return p->pCare;
}
```

Listing 11: Computing care set

The following theorem establishes a criterion that should be satisfied by a node's fanin in order to maintain the correct functionality of the AIG.

Theorem 4.1. Let f_n denote the function of the node n, f_i denote the function of the fanin, and f^{care} denote the care set of the node n. Let $f_1 = f_n$ represent the function before adding the fanin, and let $f_2 = f_n \wedge f_i$ represent the function after adding the fanin. Define $D = f_1 \oplus f_2$ as the Boolean difference after the change. Then, for the fanin to be inserted, the following property should hold:

$$D \wedge f^{care} = 0 \tag{1}$$

Proof. Suppose the fanin satisfies the condition $D \wedge f^{care} = 0$.

Given that $D = f_1 \oplus f_2$ contains those bits where f_1 and f_2 have different values, if condition (1) holds true, it implies that the bits that differ between f_1 and f_2 are not in the care set f^{care}.

This condition ensures that the change introduced by adding the fanin does not affect any input patterns specified by the care set, meaning it does not change the function of the AIG. Therefore, the fanin satisfies the criterion for valid insertion, as stated in the theorem.

Conversely, if a fanin does not satisfy condition (1), it implies that the change introduced by adding the fanin affects the input patterns specified by the care set. In such cases, adding the fanin would change the function of the AIG.

Thus, condition $D \wedge f^{care} = 0$ is necessary and sufficient for ensuring the validity of fanin insertion.

Therefore, the theorem holds. □

5 Verification

Verification is performed during or after logic synthesis to make sure that the representation of the circuit in the form of the MAIG is functionally correct throughout the computation.

The truth tables are defined for all nodes in the AIG. The truth table for a node can be retrieved using node ID as an index into the vector of truth tables. The truth table for the output is computed after constructing truth tables for the internal nodes by applying logic operations (AND and NOT). To perform verification, we compare the output truth table after optimization with the output truth table before optimization.

If we have multiple outputs, we calculate the truth table for each output and then compare the results against the truth tables before optimization.

The following pseudocode (Listing 12) shows a function that is called every time the fanins of multi-input And-Inverter graph change. For simplicity and optimization purposes, the verification is performed in place.

```
static inline void Maig_TruthUpdate( maig * p, vi * vTfo )
{
  int i, iTemp, nFails = 0;
  p->nTravIds++;
  Vi_ForEachEntry( vTfo, iTemp, i ) {
    // compute truth table of the node
    Maig_TruthSimNode( p, iTemp );
    // check if node is a primary output
    if ( !Maig_ObjIsPo(p, iTemp) )
      continue;
    // check if truth tables are equal, if not - the verification fails
    if ( !Tt_Equal(Maig_ObjTruth(p, iTemp, 2), Maig_ObjTruth(p, iTemp, 0), p->nWords) ) {
      printf( "Verification failed at output %d.\n", iTemp - (p->nObjs - p->nOuts) )
      nFails++;
    }
  }
```

```
  if ( nFails )
    printf( "Verification failed for %d outputs after updating node %d.\n", \
    nFails, Vi_Read(vTfo,0) );
}
```

Listing 12: Pseudocode of an updating function, where the verification is performed

6 Experimental results

The optimization framework implemented in C++ utilizes truth tables and MAIGs for efficient functional and structural representation. Our novel algorithm employs a redundancy addition and removal approach, enhanced by various heuristics, to minimize circuit area. The behavior of the algorithm can be customized using various parameters, such as:

1. *nIters* – the number of iterations.
2. *nExpands* – the number of nodes to expand.
3. *nGrowth* – the maximum number of fanins that can be added to one node.
4. *nDivs* – the number of shared divisors to extract
5. *nFaninMax* – the limit on the fanin count at a node.
6. *nSeed* – the random seed.
7. *fVerbose* – the verbosity level (Boolean variable).

The results of the algorithm are reported iteratively to the console and the best circuit is written to a file with the extension .aig, which can be visualized using ABC [1].

Table 6.1 presents a detailed comparison of synthesis algorithms:

- The column "*Function*" lists the names of the files containing AIGs.
- The column "*Ins*" ("*Outs*") indicates the number of primary inputs and primary outputs, respectively.
- The column "*Factoring truth table into ABC (baseline)*" displays the circuit areas obtained by reading truth tables into ABC and applying the *&strash* command.

- The next three columns, "*Applying dc2 script in ABC to the baseline*", "*Applying resyn2rs script in ABC to the baseline*" and "*Applying compress2rs script in ABC to the baseline*", list the area after applying algorithms dc2, resyn2rs, and compress2rs to the baseline. Script "dc2" performs a sequence of AIG transformations that involve rewriting, refactoring and balancing. The other two scripts, "compress2rs" and "resyn2rs", additionally perform AIG-based resubstitution [13]. The first one (compress2rs) minimized the number of nodes while disregarding the number of levels. The second one (resyn2rs) minimizes the number nodes while making sure that the number of levels does not increase. The AIG rewriting was originally proposed in [2] and developed in [12].

- The last column of the table shows the sizes of circuits that were generated after applying our new algorithm.

The runtime varies from 3 seconds to over 5 hours, depending on a test case. The algorithm is slow when applied to some test cases for several reasons. First of all, runtime depends on the number of iterations (the default number of iterations was 100000), but in those test cases, where the initial area was over 5000 nodes, the number of iterations was reduced to 10000 or 1000. The second reason for slow time was the number of nodes in the circuit, circuits with over 7000 nodes tend to run significantly longer than the smaller ones. Additionally, the value of the nExpands parameter – the greater the number of nodes to insert, the higher runtime we get. One last observation is that node reduction takes the longest time most cases.

The algorithm proposed in this work was tested on 100 test cases with 16 or fewer variables from IWLS 2022 Programming Contest [8]. The correctness of the resulting circuits is checked by comparing the truth tables of the initial MAIGs and the optimized MAIGs. The comparison shows that in all 100 test

cases our algorithm generated circuits with significantly smaller areas (the best improvement was in a circuit that was 17 times smaller than the one from the compress2rs algorithm). In 5 cases our algorithm gave the same area as the best results from IWLS 2022 Programming Contest (test cases ex10, ex16, ex17, ex40, ex56) and in 4 cases our algorithm produced better results than the best competition results (test cases ex51, ex53, ex55, ex97).

Table 6.1: The number of two-input nodes in the minimized circuits produced by different algorithms.

Function	Ins	Outs	Factoring truth table into ABC (baseline)	Applying dc2 script in ABC to the baseline	Applying resyn2rs script in ABC to the baseline	Applying compress2rs script in ABC to the baseline	Applying our algorithm to the baseline
ex00	6	1	57	31	47	34	24
ex01	6	1	60	42	50	48	28
ex02	8	1	219	162	155	153	93
ex03	8	1	132	101	71	52	26
ex04	10	1	904	714	663	663	387
ex05	10	1	396	254	247	249	44
ex06	12	1	3389	2715	2439	2430	1913
ex07	12	1	1822	1383	1266	1261	754
ex08	8	8	1450	1154	1023	1012	746
ex09	8	8	1443	1137	987	988	703
ex10	5	1	24	10	11	11	10
ex11	7	1	69	53	45	44	21
ex12	9	1	143	115	102	108	32
ex13	11	1	302	239	224	220	52
ex14	13	1	638	516	459	455	72
ex15	15	1	1358	1063	921	908	109
ex16	5	5	70	25	22	34	18
ex17	6	6	128	73	65	43	24
ex18	7	7	204	144	107	84	36
ex19	8	8	310	248	187	163	50
ex20	9	9	464	370	312	309	62
ex21	10	10	668	542	463	461	76
ex22	11	11	1006	819	699	710	90
ex23	12	12	1445	1188	1036	1033	135
ex24	13	13	2195	1790	1591	1564	145
ex25	14	14	3160	2554	2307	2301	176
ex26	15	15	4861	3871	3505	3507	224
ex27	16	16	7009	5544	5125	5115	421
ex28	7	10	185	122	115	112	45
ex29	9	1	185	138	143	116	51
ex30	14	8	4099	2816	2377	2338	80
ex31	9	18	3664	2869	2668	2658	1944
ex32	15	9	153	62	57	55	51
ex33	5	28	215	164	128	115	79
ex34	9	5	372	257	222	230	55
ex35	7	2	32	22	18	18	16
ex36	10	10	3328	2697	2521	2513	2249
ex37	8	63	1056	536	506	485	161
ex38	8	7	90	53	41	39	32
ex39	14	14	3898	2507	2037	2022	295
ex40	16	40	1821	1105	873	882	195

Continued on next page

Table 6.1 – continued from previous page

Function	Ins	Outs	Factoring truth table into ABC (baseline)	Applying dc2 script in ABC to the baseline	Applying resyn2rs script in ABC to the baseline	Applying compress2rs script in ABC to the baseline	Applying our algorithm to the baseline
ex41	5	3	59	28	28	30	18
ex42	7	3	143	104	80	81	30
ex43	8	4	230	173	153	145	44
ex44	10	4	332	232	193	204	61
ex45	16	46	2135	1219	1016	1005	214
ex46	5	8	70	44	46	46	34
ex47	16	1	380	110	48	59	27
ex48	14	14	3371	2195	1956	1943	649
ex49	7	10	199	132	117	117	43
ex50	8	2	67	22	35	18	18
ex51	8	2	191	132	124	122	28
ex52	8	2	38	20	26	21	19
ex53	8	2	122	91	87	87	37
ex54	8	2	31	13	13	13	13
ex55	8	7	405	305	281	278	152
ex56	12	3	160	83	99	92	29
ex57	12	3	916	685	582	586	267
ex58	12	3	295	224	210	212	96
ex59	12	3	1435	1065	937	936	416
ex60	12	3	221	176	159	159	73
ex61	12	11	5885	4787	4534	4535	4015
ex62	16	4	273	112	122	110	42
ex63	16	4	8618	7117	6217	6210	5822
ex64	16	4	2050	1585	1450	1449	999
ex65	16	4	15213	11501	11315	11309	7920
ex66	16	4	1485	1164	1067	1067	596
ex67	10	77	13502	10533	10295	10281	10033
ex68	12	3	1347	1098	1009	997	448
ex69	12	3	1176	915	835	836	447
ex70	12	3	1972	1596	1443	1432	712
ex71	12	3	1851	1518	1355	1347	687
ex72	12	3	2839	2265	2029	2023	1264
ex73	12	3	1532	1204	1052	1036	285
ex74	12	3	3888	3144	2916	2908	1765
ex75	12	3	3239	2642	2357	2363	1031
ex76	12	3	1529	1217	1117	1118	392
ex77	12	3	1250	968	862	861	482
ex78	12	3	2047	1610	1438	1432	612
ex79	12	3	1847	1512	1354	1348	623
ex80	12	3	3184	2545	2354	2368	1553
ex81	12	3	2233	1732	1545	1557	529
ex82	12	3	4208	3339	3091	3096	1891
ex83	12	3	4398	3379	3278	3286	2067
ex84	12	3	572	421	368	366	152

Continued on next page

Table 6.1 – continued from previous page

Function	Ins	Outs	Factoring truth table into ABC (baseline)	Applying dc2 script in ABC to the baseline	Applying resyn2rs script in ABC to the baseline	Applying compress2rs script in ABC to the baseline	Applying our algorithm to the baseline
ex85	12	3	818	559	554	559	241
ex86	12	3	745	527	523	517	209
ex87	12	3	1574	1204	1124	1102	638
ex88	12	3	1445	1060	972	973	412
ex89	12	3	1791	1326	1200	1178	306
ex90	12	3	3316	2480	2276	2266	1099
ex91	12	3	1486	1132	1016	1024	480
ex92	10	3	128	51	58	54	31
ex93	12	3	164	82	92	85	43
ex94	11	3	239	148	110	131	37
ex95	11	3	248	159	150	146	67
ex96	12	3	388	290	265	264	80
ex97	12	3	459	298	281	279	67
ex98	12	3	1014	728	688	685	186
ex99	12	3	380	253	235	236	90
Total			**164157**	**125459**	**114972**	**114531**	**62350**

7 Conclusion

In this work, we developed a novel algorithm for area minimization of multi-output completely specified Boolean functions. The algorithm leverages redundancy addition and removal (RAR) and utilizes multi-input And-Inverter Graphs (MAIG) as its primary data structure. We introduced and proved a new criterion for acceptable fanins, which ensures that the addition/removal of new fanins maintains the correct functionality of the MAIG. This criterion is critical for preserving the correctness of the circuit during the optimization.

Our approach also incorporates various heuristics to enhance the performance of the basic algorithm. The effectiveness of the proposed method was demonstrated through the optimization of benchmark circuits from the IWLS 2022 Programming Contest [8], showing significant improvements in area sizes.

The **key contributions** of this thesis are the development of a new algorithm for logic synthesis and the formulation and proof of a criterion for acceptable fanins.

In the future, we may work on implementing the ability to perform multiple randomized optimization runs using multi-core processors and selecting the best result. Another potential research direction is to explore the option of training a neural network to facilitate the process of adding and removing fanins.

References

[1] Berkeley Logic Synthesis and Verification Group. *ABC: A System for Sequential Synthesis and Verification*. URL: http://www-cad.eecs.berkeley.edu/~alanmi/abc.

[2] P. Bjesse and A. Boralv. "DAG-aware circuit compression for formal verification". In: ICCAD '04 (2004), pp. 42–49. DOI: 10.1109/ICCAD.2004.1382541. URL: https://doi.org/10.1109/ICCAD.2004.1382541.

[3] R. K. Brayton. "The decomposition and factorization of Boolean expressions". In: *ISCA-82* (1982), pp. 49–54.

[4] C.-W. Chang and M. Marek-Sadowska. "Single-pass redundancy-addition-and-removal". In: (2001), pp. 606–609. DOI: 10.1109/ICCAD.2001.968723.

[5] K. Cherevko and A. Mishchenko. "Area minimization using decision diagrams without constructing them". In: Proc. RM'23.

[6] K. Cherevko. "Boolean network optimization by stochastic rewiring". In: *Тези доповідей XII Міжнародної наукової конференції*. 2024, p. 112.

[7] D. I. Cutress. *TSMC Financial Year 2022*. 2022. URL: https://morethanmoore.substack.com/p/tsmc-financial-year-2022.

[8] *IWLS 2022 Programming Contest*. 2022. URL: https://github.com/alanminko/iwls2022-ls-contest.

[9] J.-H. R. Jiang and S. Devadas. *Logic Synthesis in a Nutshell*. 2009. URL: https://api.semanticscholar.org/CorpusID:62271846.

[10] T. Lam, W. Tang, X. Wei, Y. Diao, and D. Wu. *Boolean Circuit Rewiring: Bridging Logical and Physical Designs*. 2016. URL: https://books.google.com/books?id=YndjCwAAQBAJ.

[11] C. Meng, W. Qian, and A. Mishchenko. "ALSRAC: Approximate Logic Synthesis by Resubstitution with Approximate Care Set". In: 2020, pp. 1–6. DOI: 10.1109/DAC18072.2020.9218627.

[12] A. Mishchenko, S. Chatterjee, and R. Brayton. "DAG-aware AIG rewriting: a fresh look at combinational logic synthesis". In: (2006), pp. 532–535. DOI: 10.1145/1146909.1147048.

[13] A. Mishchenko and R. K. Brayton. "Scalable Logic Synthesis using a Simple Circuit Structure". In: (2006). URL: https://api.semanticscholar.org/CorpusID:8597391.

[14] Y. Miyasaka. "Transduction Method for AIG Minimization". In: (2024), pp. 398–403. DOI: 10.1109/ASP-DAC58780.2024.10473816.

[15] Y. Miyasaka, A. Mishchenko, J. Wawrzynek, D. Ruic, and X. Xu. "Randomized transduction for high-effort logic synthesis". In: *Submitted to IWLS'24* (2024).

[16] T. Sasao. *Switching Theory for Logic Synthesis*. Kluwer Academic Publishers, 1999. ISBN: 0792384563.

[17] L. Wang, Y. Chang, and K. Cheng. *Electronic Design Automation: Synthesis, Verification, and Test*. ISSN. Elsevier Science, 2009. ISBN: 9780080922003. URL: https://books.google.com/books?id=3XBe7dLb5NEC.