Minimization of lines in reversible circuits

Law, Jayati J.

Lethbridge, Alta. : University of Lethbridge, Dept. of Mathematics and Computer Science

http://hdl.handle.net/10133/3729

Downloaded from University of Lethbridge Research Repository, OPUS
MINIMIZATION OF LINES IN REVERSIBLE CIRCUITS

JAYATI J LAW
Bachelor of Science, Rajasthan Technical University, 2012

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

© Jayati J Law, 2015
MINIMIZATION OF LINES IN REVERSIBLE CIRCUITS

JAYATI J LAW

Date of Defense: August 6, 2015

Dr. Jackie Rice
Supervisor                Associate Professor    Ph.D.

Dr. Stephen Wismath
Committee Member          Professor             Ph.D.

Dr. Robert Benkoczi
Committee Member          Associate Professor    Ph.D.

Dr. Howard Cheng
Chair, Thesis Examination Committee    Associate Professor    Ph.D.
Dedication

For my mother.
Abstract

Reversible computing has been theoretically shown to be an efficient approach over conventional computing due to the property of virtually zero power dissipation. A major concern in reversible circuits is the number of circuit lines or qubits which are a limited resource. In this thesis we explore the line reduction problem using a decision diagram based synthesis approach and introduce a line reduction algorithm—Minimization of lines using Ordered Kronecker Functional Decision Diagrams (MOKFDD). The algorithm uses a new sub-circuit for a positive Davio node structure in addition to the existing node structures. We also present a shared node ordering for OKFDDs. OKFDDs are a combination of OB-DDs and OFDDs. The experimental results shows that the number of circuit lines and quantum cost can be reduced with our proposed approach.
Acknowledgments

Firstly, I would like to express my sincere gratitude to my supervisor Dr. Jackie Rice for believing in me and guiding me throughout the study. I learned a lot from her professionally and personally. Her expertise, understanding and patience helped me completing my research work and writing this thesis. I would like to thank the members of my committee, Dr. Robert Benkoczi and Dr. Stephen Wismath for the discussions and continuous support they provided at all levels of the research.

I would also like to acknowledge Dr. Daya Gaur and Dr. Saurya Das for helping me understand few important concepts related to my study. Thanks goes to my friends and my lab mates, Ram Dahal, Fariha Naz, Zamilur and Asif for all the discussions and valuable suggestions. It would have been difficult to work without them.

Lastly, I thank my mother and my younger brother for encouraging me with their best wishes.
# Contents

## List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Structure of the Thesis</td>
<td>3</td>
</tr>
<tr>
<td>Background</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Basic Definitions</td>
<td>5</td>
</tr>
<tr>
<td>2.2 Quantum</td>
<td>8</td>
</tr>
<tr>
<td>2.2.1 Quantum States</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2 Quantum Gates</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Cost Metrics</td>
<td>12</td>
</tr>
<tr>
<td>2.4 Reversible Logic Synthesis Techniques</td>
<td>14</td>
</tr>
<tr>
<td>2.4.1 ESOP-based Synthesis</td>
<td>14</td>
</tr>
<tr>
<td>2.4.2 Transformation-based Synthesis</td>
<td>17</td>
</tr>
<tr>
<td>2.4.3 Search-based Synthesis</td>
<td>18</td>
</tr>
<tr>
<td>2.4.4 BDD based Synthesis</td>
<td>20</td>
</tr>
<tr>
<td>Decision Diagrams</td>
<td>25</td>
</tr>
<tr>
<td>3.1 Reduced Ordered BDD (ROBDD)</td>
<td>25</td>
</tr>
<tr>
<td>3.2 Ordered Functional Decision Diagram (OFDD)</td>
<td>27</td>
</tr>
<tr>
<td>3.3 Ordered Kronecker Decision Diagram (OKFDD)</td>
<td>29</td>
</tr>
<tr>
<td>Reducing lines using OKFDDs</td>
<td>32</td>
</tr>
<tr>
<td>4.1 Bounds on reversible circuit lines</td>
<td>32</td>
</tr>
<tr>
<td>4.2 The Algorithm</td>
<td>34</td>
</tr>
<tr>
<td>4.3 Discussion</td>
<td>40</td>
</tr>
<tr>
<td>Experimental Evaluation</td>
<td>43</td>
</tr>
</tbody>
</table>

Bibliography 49
List of Tables

2.1 Quantum cost table .................................................. 13
2.2 Toffoli gates for BDD node types with additional constant line [45] .... 22
3.1 Toffoli gate circuits for node structures of decomposition types ........ 31
4.1 Toffoli gate circuits for node structures of decomposition types .......... 40
5.1 Experimental Results for the Algorithm .................................. 47
5.2 Lower bound Table .................................................... 48
Chapter 1

Introduction

In today’s world power minimization is one of the most concerning issues in electronics. Traditional computing systems use logically irreversible circuits. In irreversible systems once a final state of information is reached, the information cannot be traced back to its initial state. In a logically irreversible system whenever any two inputs have a single output, the two input states transform into one output state as shown in the example given in Figure 1.1. This results in loss of information bits. According to Frank [12] the irreversible information loss is explained in detail by Landauer’s principle. According to Landauer’s principle [18] every logical manipulation of information on an irreversible system results in the increase of entropy. Entropy defines the state of a system in terms of temperature and heat transfer. As given in [18] every time a bit is erased $KT\ln 2$ amount of energy is released, where $K$ is the Boltzmann constant and $T$ is the room temperature (for $T = 300$ Kelvin this energy is about $2.9 \times 10^{21}$ joules). During each operation on an irreversible system this energy is transferred to the environment. This large amount of energy is proportional to the number of transistors being used on a single integrated chip. Moore’s Law says that the number of transistors on an integrated chip will double in approximately every two years [37]. With this increase in the number of transistors, irreversible circuits are becoming highly inefficient. In 1973, Charles Bennett of IBM Research showed that any irreversible computation can be carried out in a reversible manner to avoid energy dissipation. In reversible circuits no information bits are erased, thus there is potentially nearly zero energy dissipation depending on the underlying technology.

Recently reversible circuits have emerged to be useful components of quantum computing. Quantum computations work with unitary matrices [2] imposing a constraint of being
1.1 Motivation

Every resource such as time, energy and distance traveled from a source to destination to perform a job in a working system always has a price depending on its usage and availability. Similarly, in reversible circuits qubits (or quantum bits) which are the information bits used in a quantum circuit are considered to be a very expensive resource. In classical circuits a bit has to be in one of two states, either 0 or 1. As will be described in Chapter 2 qubits may attain any superposition state i.e. 0, 1 or both at the same time. Each qubit is represented by a wire or line in a reversible circuit. The components of a reversible circuit are explained in detail in Chapter 2.

To illustrate the importance of line reduction consider an example where a full adder with three inputs and two outputs is to be realized by a reversible circuit. To implement the truth table from Figure 1.2 as a reversible circuit, extra qubits are required to make the truth table reversible. These extra qubits allow each of the input values of the truth table to be assigned unique output values. The process of converting an irreversible truth table to a reversible truth table is known as ‘Embedding’ [46]. An embedding process for the full adder is shown in Figure 1.3. In the worst case such a function would require four additional qubits.
lines to make a seven-input seven-output circuit. This means seven qubits are required. It becomes difficult and costly to build a reversible circuit if the process of embedding causes the number of qubits to increase drastically. The latest quantum computer built is a 128-qubit computer [52], therefore the necessity to reduce the qubits required in a single computation is vital.

<table>
<thead>
<tr>
<th>cin</th>
<th>x</th>
<th>y</th>
<th>s</th>
<th>g1</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 1.2: Full adder

<table>
<thead>
<tr>
<th>cin</th>
<th>x</th>
<th>y</th>
<th>s</th>
<th>g1</th>
<th>g2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Figure 1.3: Embedding

1.2 Structure of the Thesis

In this thesis we detail our investigation into ways to minimize the number of qubits or circuit lines in a reversible circuit. The remainder of the thesis is structured as follows:

Chapter 2 provides the necessary background to give the basic understanding of reversible logic and circuits. It includes the definitions related to reversible logic and the components of a reversible circuit. It also gives an overview of reversible logic synthesis techniques such as ESOP (Exclusive-Or Sum of Products) -based synthesis.

The reversible circuit line reduction problem has been tackled with some heuristics in
the past. To understand the concept behind the heuristics, we explain related works in Chapter 3.

The Decision-Diagram (DD) -based synthesis approach is described in Chapter 4. This chapter illustrates the construction of decision diagrams as well as different DD-based synthesis algorithms such as BDD (Binary Decision Diagram) and KFDD (Kronecker Functional Decision Diagrams) -based synthesis methods. Here we deliver the primary concept of decision diagrams.

In chapter 5 we discuss the existing bounds on the line reduction problem given in [20]. We introduce our algorithm based on decision diagrams for line reduction with suitable examples. Our approach includes modifications in the KFDD synthesis algorithm. Our results appear in the proceedings of the 2015 IEEE Pacific Rim Conference [15].

Chapter 6 consists of the experimental results obtained by our algorithm. We compare our results with the existing approaches. We also compare our results with the lower bound on the line reduction problem.

The thesis concludes with Chapter 7 summarizing the contributions of this thesis in the field of reversible logic synthesis. The chapter also suggests some possibilities for future work.
Chapter 2

Background

The research on reversible computing began considering the thermodynamic limits of non-reversible computing. The inspiration behind the technology shift is already justified in chapter 1. To understand the significance of the reversible technology, we must comprehend the computational model involved. This chapter provides the necessary background to explore the circuit line minimization problem.

2.1 Basic Definitions

Definition 2.1. A multi-output Boolean Function $f : A^m \rightarrow A^n$ is reversible if it is bijective.

Let $A$ be a finite set and $f : A^m \rightarrow A^n$ be a Boolean function which maps each input vector to a unique output vector (bijection); then this function is reversible. According to [39], a gate is reversible if the function it computes is bijective, and a circuit is reversible if it consists entirely of reversible gates. A cascade of reversible gates implements a reversible function with no fan-out (an output feeding more than one inputs) or feedback [25]. Figures 2.1 and 2.2 show an irreversible and a reversible function truth table.

<table>
<thead>
<tr>
<th>$xy$</th>
<th>$f(xy)$</th>
<th>$xy$</th>
<th>$x'y'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>00</td>
<td>00</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>01</td>
<td>01</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>11</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 2.1: Irreversible function Figure 2.2: Reversible function

Definition 2.2. If a gate computes a (Boolean) function which is bijective then the gate
2.1. BASIC DEFINITIONS

A necessary condition for a gate to be reversible is that it should have same number of input and output wires. A gate is a $k \times k$ gate if it has $k$ wires [39]. Let $X := \{x_1, \ldots, x_n\}$ be the set of Boolean variables. Then, a reversible gate has the form $g(C, T)$, where $C = \{x_{i1}, \ldots, x_{ik}\} \subset X$ is the set of control lines and, $T = \{x_{j1}, \ldots, x_{jl}\} \subset X$ with $C \cap T = \emptyset$ is the set of target lines [47] and $k + l = n$. The following definitions illustrate examples of reversible gates.

**Definition 3.** A $k$-CNOT Gate is a $(k+1) \times (k+1)$ gate which leaves the $k$ inputs unchanged and inverts the $(k+1)^{th}$ input if all the $k$ inputs are 1.

A $0-CNOT$ gate is a simple NOT gate which inverts the input without any controls $(x) \rightarrow (x \oplus 1)$ while a $1-CNOT$ gate is a Controlled NOT Gate which performs $(x, y) \rightarrow (x \oplus y)$ where $\oplus$ is the XOR operation. Figures 2.3 and 2.4 illustrate NOT and CNOT gates.

![Figure 2.3: NOT Gate](image)

![Figure 2.4: CNOT Gate](image)

**Definition 2.4.** A Multiple Control Toffoli Gate (MCT) with target line $x_j$ and control lines $x_{i1}, x_{i2}, \ldots, x_{ik}$ maps $(x_1 x_2 \ldots x_{j} \ldots x_n)$ to $(x_1 x_2 \ldots (x_{i1} x_{i2} \ldots x_{ik}) \oplus x_{j} \ldots x_n)$. All control lines must be 1 in order for the target qubit to be inverted. A MCT gate with no control is a NOT gate. A MCT gate with one control gate is a controlled-NOT gate. A MCT with two control lines is a Toffoli [34] gate.

A MCT gate is identical to the $k$-CNOT gate by definition given that the value of $k > 1$. It is important to note that the MCT can use $k$-CNOT notation and vice-versa. Generally, a $k$-CNOT gate is expressed in terms of Toffoli gates. A $k$-CNOT gate with $k = 0$ is said to be a TOF0 gate or a Toffoli gate with 0 controls (NOT gate), a TOF1 for $k = 1$ and so on. An example of a MCT is given in Figure 2.5:

**Definition 2.5.** A Multiple Control Fredkin Gate (MCF) with target lines $x_p$ and $x_q$ and control lines $x_{i1}, x_{i2}, \ldots, x_{ik}$ maps $(x_1 x_2 \ldots x_p \ldots x_q \ldots x_n)$ to $(x_1 x_2 \ldots x_q \ldots x_p \ldots x_n)$ if all the
2.1. BASIC DEFINITIONS

control lines have value 1. Therefore, it is also called a Swap gate [34]. A MCF gate is shown in Figure 2.6.

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c} \\
\text{d} \\
\text{e} \\
\hline
\text{z} \oplus \text{abcde} \oplus \text{z}
\end{array}
\]

Figure 2.5: MCT Gate

\[
\begin{array}{c}
\text{a} \\
\text{b} \\
\text{c} \\
\hline
\text{a} \times \text{c} \times \text{b}
\end{array}
\]

Figure 2.6: MCF Gate

**Definition 2.6.** A Dual is a gate which reverses the logic function. A gate is Self-reversible if the dual is identical to the gate itself [29].

Every gate has a dual which transforms the output vectors to input vectors. For example the dual of the NOT gate is the NOT gate: \(x' (\text{NOT}) = x\).

The Hamming weight is the number of logical 1s in the set of values.

**Definition 2.7.** A gate is a Conservative gate if the Hamming weight of the set of input values is similar to the set of output values. Similarly, a Non-Conservative gate has unequal Hamming weights for its input and output values [29].

**Definition 2.8.** A gate is Universal if it can implement other basic reversible logic gates independently.

The NAND gate is a universal gate in Boolean logic. Similarly, the Toffoli gate is also an example of a universal gate in reversible logic [29].

**Definition 2.9.** Lines or wires in a reversible circuit represent the variables of a reversible truth table.

**Definition 2.10.** Garbage outputs are additional outputs which do not produce any desired functionality.
In [26] it is shown that at least \( g = \lceil \log_2(\mu) \rceil \) garbage outputs are required for converting an irreversible function to a reversible function, where \( \mu \) is the maximum number of times a single output pattern is repeated in an irreversible truth table. Converting an irreversible function with \( n \) inputs and \( m \) outputs into a reversible function will require \( m + g \) qubits. Since \( m + g > n \), \( c \) number of additional lines with a constant input for each line are added to make a function reversible. Thus it becomes \( n + c = m + g \). In Figure 2.7 the garbage outputs are labeled as ‘g’ and the additional line is labeled as ‘0’.

\[
\begin{align*}
\text{a} & \quad \text{g = a} \\
\text{b} & \quad f = a(a \oplus ab) \oplus b \\
0 & \quad g = a(a \oplus ab)
\end{align*}
\]

Figure 2.7: Reversible circuit with garbage outputs.

**Definition 11.** A **Reversible Circuit** comprises a cascade of reversible gates on circuit lines implementing a reversible function. It may contain garbage outputs.

### 2.2 Quantum Concepts

#### 2.2.1 Quantum States

Qubits exhibit the property of linear superposition of basis states (0,1) as described in Chapter 1. The state of a qubit \( |\psi> \) is defined as [5]

\[
|\psi> = \alpha |0> + \beta |1>
\]

Here we use the Dirac notations of basis state vectors \( |0> = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \) and \( |1> = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \)

whereas the \( \alpha \) and \( \beta \) are complex numbers satisfying the condition \( |\alpha|^2 + |\beta|^2 = 1 \). In the vector form the state of a single qubit is shown by the vector \( \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \). Moreover,
2.2. QUANTUM GATES

the principles of quantum mechanics [28] declare that two quantum states are similar if they differ by a phase factor of $e^{i\theta}$, $\theta \in \mathbb{R}$. Thus, the quantum state $\psi$ is also written as $|\psi\rangle = \cos(\theta/2)|0\rangle + e^{i\phi}\sin(\theta/2)|1\rangle$ where $0 \leq \phi < 2\pi$, $0 \leq \theta < \pi$. $\theta$ and $\phi$ are the angle coordinates indicating a qubit state in the Bloch sphere [4]. The Bloch sphere, as shown in Figure 2.8 provides a geometrical representation of a single-qubit state. The north pole (+Z) represents the state $|0\rangle$ while the south pole (-Z) represents the state $|1\rangle$. The states on the equator are the superpositions of the states $|0\rangle$ and $|1\rangle$ with equal weights $\theta = \pi/2$ and different phases [14]. Further information on these concepts can be found in [28].

![Figure 2.8: Bloch Sphere representation](image)

2.2.2 Quantum Gates

Quantum gates are small circuits operating on qubits. They are reversible by nature and are represented by unitary matrices. A matrix $U$ is a unitary matrix provided that $UU^\dagger = I$ where $I$ is the identity matrix. The $2 \times 2$ identity matrix is given as:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

The unitary operator $U^{-1} = U^\dagger$ ensures the reversible characteristic of a matrix.
2.2. QUANTUM GATES

In reversible logic there exists quantum gate libraries such as NCT (NOT, CNOT, Toffoli), GT (Generalized Toffoli) and NCV (NOT, CNOT, V gates) to design quantum circuits. The use of these libraries affect the cost metrics (explained in 2.3) associated with the circuit design. The NCT library is the most common library used to design quantum circuits. In this thesis we will examine synthesis methods utilizing the NCT library for circuit design. However, the decomposition of the reversible gates to the NCV library provides the exact computation for quantum cost (explained in Section 2.3). The NCV library consists of NOT, CNOT, V and V† operators. The NCV operators are defined by the following quantum gates [28]:

1. NOT gate: The NOT gate simply inverts the \( t \) qubit. The gate is denoted by \( T(\emptyset, t) \), where the control is \( \emptyset \) and target is \( t \). The unitary matrix for the NOT gate is:

\[
\begin{pmatrix}
0 & 1 \\
1 & 0
\end{pmatrix}
\]

2. CNOT gate: The Controlled NOT, \( T(c, t) \), inverts the qubit at target \( t \) only if the control \( c \) is 1. The unitary matrix for CNOT is:

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0
\end{pmatrix}
\]

3. V gate: The controlled V gate, \( V(c, t) \), performs the V operation on the target \( t \) qubit when the control \( c \) is 1. The V operation is equivalent to square root of NOT. Two consecutive V operations result in a NOT operation. This qubit operator causes the half spin of a qubit, as shown in Figure 2.9(b). Thus, two half spins (two V gates) of a qubit concludes in the inversion of a qubit state. The V gate is depicted as \( V \).
The matrix for the $V$ gate is \[
\begin{pmatrix}
\frac{1+i}{2} & \frac{1-i}{2} \\
\frac{1-i}{2} & \frac{1+i}{2}
\end{pmatrix}.
\]

4. **$V^\dagger$ gate:** The controlled $V^\dagger$ gate, $V^\dagger(c,t)$, performs the $V^\dagger$ operation on the target qubit $t$ if the control qubit $c$ is 1. The $V^\dagger$ operation is similar to the inverse of $V$ operation i.e. $V^\dagger = V^{-1}$. Thus, $V$ gate and $V^\dagger$ gate if applied together form an identity gate. This operator also results in the half spin of a qubit but in an opposite direction to the $V$ operator. Similar to the $V$ operation two $V^\dagger$ operations in series produce a NOT operation. The $V^\dagger$ gate is depicted as $\text{\textbullet}$ $V^\dagger$. The unitary matrix for the $V^\dagger$ gate is \[
\begin{pmatrix}
\frac{1-i}{2} & \frac{1+i}{2} \\
\frac{1+i}{2} & \frac{1-i}{2}
\end{pmatrix}.
\]

The $V$ and $V^\dagger$ operations are summarized in Figure 2.9(a). Figure 2.9(b) explains the spin of a qubit induced by each $V$ or $V^\dagger$ gate. The ends of the vertical pole are labeled as 0 and 1 for $|0\rangle$ state and $|1\rangle$ state of a qubit respectively. Similarly, the ends of the horizontal pole are labeled as $+$ and $-$ for any respective positive and negative imaginary state of a qubit. The imaginary state of a qubit can be defined by any complex number.

These basic NCV quantum gates can be used to build other quantum gates such as the Toffoli gate. An example to show the quantum cost calculation of a Toffoli is to decompose a Toffoli gate into NCV quantum gates. Since there are 5 quantum gates in the decomposed Toffoli gate we say that the quantum cost of a TOF3 gate is 5.
2.3 Cost Metrics

Cost metrics allow a circuit designer to compute a measurement of a reversible circuit in terms of some cost. The cost computation of every component of a reversible circuit is calculated according to the standard metrics. The basic parameters for the measurement are as follows:

1. **Quantum Cost:** Quantum cost evaluates the cost of gates in a reversible circuit. For every reversible gate there is an associated cost built on the number of underlying quantum gates. As an illustration, the cost of a Toffoli gate is 5 based on the construction shown in Figure 2.10. Table 2.1 shows the quantum cost of Toffoli gates of size $n$, where $n \in \mathbb{Z}^+$ [19]. Likewise, the cost of a Fredkin gate of size $n$ is the sum of the cost of a $n$-bit Toffoli gate and integer value 2 (for 2 CNOT gates). Therefore, for any two identical circuits with similar line count, the circuit with a lesser quantum cost is considered to be more economical compared to the one with a higher quantum cost.
cost. There has been a lot of study [11] [47] [32] [51] [21] on reducing quantum cost.

Table 2.1: Quantum cost table

<table>
<thead>
<tr>
<th>Size (n)</th>
<th>Garbage</th>
<th>Name</th>
<th>Quantum Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>NOT, t1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>CNOT, t2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>Toffoli, t3</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>Toffoli4, t4</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>t5</td>
<td>29</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>t5</td>
<td>26</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>t6</td>
<td>61</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>t6</td>
<td>52</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>t6</td>
<td>38</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>t7</td>
<td>125</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>t7</td>
<td>80</td>
</tr>
<tr>
<td>7</td>
<td>4</td>
<td>t7</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>t8</td>
<td>253</td>
</tr>
<tr>
<td>8</td>
<td>1</td>
<td>t8</td>
<td>100</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>t8</td>
<td>62</td>
</tr>
<tr>
<td>9</td>
<td>0</td>
<td>t9</td>
<td>509</td>
</tr>
<tr>
<td>9</td>
<td>1</td>
<td>t9</td>
<td>128</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>t9</td>
<td>74</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>t10</td>
<td>1021</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>t10</td>
<td>152</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>t10</td>
<td>86</td>
</tr>
<tr>
<td>n &gt; 10</td>
<td>0</td>
<td>tn</td>
<td>2n - 3</td>
</tr>
<tr>
<td>n &gt; 10</td>
<td>1</td>
<td>tn</td>
<td>24n - 88</td>
</tr>
<tr>
<td>n &gt; 10</td>
<td>n-3</td>
<td>tn</td>
<td>12n - 34</td>
</tr>
</tbody>
</table>

2. **Gate Count:** This parameter refers to the number of gates required for implementing a function. A change in the gate count may increase or decrease the quantum cost subject to the size of gates being used. Thus, the gate count is not directly proportional to the quantum cost. As an illustration in Figure 2.11 the function \((acd \oplus abc)\) is implemented in two ways, first with a quantum cost of 23 with gate count of 3 and secondly with a quantum cost of 26 with gate count of 2 respectively.
3. **Line Count:** The number of lines in a reversible circuit is generally equal to the number of variables in the truth table. In other words, lines represent qubits. Qubits are particles that demand a controlled system to keep them in a stable initial state and change states. Thus, they are expensive to sustain. As a result, minimizing circuit lines is often considered preferable compared to reducing quantum cost [49]. There is usually a trade-off between reducing quantum cost and circuit lines [50]. The problem of reducing circuit lines in reversible circuits is discussed in works such as [48] [20] [9] [49].

### 2.4 Reversible Logic Synthesis Techniques

Conventional logic synthesis approaches use the classical universal gate library of a Boolean function—AND, NOT and OR gates. Reversible logic synthesis, unlike classical logic synthesis techniques, implements a Boolean function using quantum gate libraries (discussed in 2.2) with no fan-out. Since no fan-out is permitted the output of each gate in a reversible circuit is used only once. Reversible logic synthesis produces reversible circuits containing a sequence of gates with no loops [31].

In the literature there are various methods for reversible logic synthesis which are broadly divided into the following categories.

#### 2.4.1 ESOP-based Synthesis

An Exclusive-or Sum Of Products (ESOP) is a variant of the basic Sum Of Products (SOP) representation of a Boolean specification. In an ESOP the product of the literals
i.e. OR of the AND terms are Ex-ORed. For example: \((ab \oplus cd)\). A SOP formulation of a Boolean function is expressed as the sum of the product of the literals. For example: \((a \land b \lor c \land d)\), also written as \((ab + cd)\). The first step of ESOP based synthesis is to obtain an ESOP expression. A given SOP expression \((a + b)\) is formulated as \((a \oplus b \oplus ab)\) in ESOP. An important reason for moving from SOP to ESOP formulation is the efficiency involved. The worst case complexity of SOP formulation considering the size of the truth table is \(O(2^{n-1})\), for a \(n\)-variable Boolean function, which is greater than the complexity of ESOP \((O(3.2^{n-3}))\) [36].

The product of the literals in an ESOP or SOP expression are represented by cubes. These cubes combine to form a cube-list. A major difference between the cubes of ESOP and SOP formulations is that the former includes don’t cares resulting in a smaller set of input values to scan for building a circuit. For both ESOP and SOP representations a lesser number of cubes result in a smaller circuit. Minimizing the SOP expression is a well known DNF (Disjunctive Normal Form) minimization problem [1]. The most commonly used algorithm for ESOP minimization is EXORCISM-4 [27]. The main idea behind both SOP and ESOP minimization is to work with the Karnaugh Map. A Karnaugh Map or K-Map is a method to simplify Boolean expressions. The truth table is transferred onto a two dimensional grid where each cell represents a combination of input conditions while the value of each cell is the corresponding output value for the input conditions. For an SOP formulation the 1 bits are covered by a minimum number of cubes (covers). These cubes are ORed to get the complete SOP expression. However, for an ESOP formulation all the 0 bits are covered by an odd number of cubes and 1 bits are covered by an even number of cubes [8]. The ESOP cubes are then Ex-ORed to get an ESOP expression. Figure 2.12 (a) and (b) show minimized covers with bold lines. Each rectangle identifies a cube for their respective formulations. As illustrated the SOP cubes are covered by 3 covers while in ESOP cubes each 1 bit is covered by a single cover (odd) and 0 bits are covered by two covers (even). The initial Boolean expression in Figure 2.12 (a) covered by dashed squares
in the K-map is \( f = \overline{a}cd + cd + acd \) the minimized expression shown by bold squares is \( f' = ad + ac + cd \). Similarly, in Figure 2.12 (b) the ESOP expression by dashed squares is \( f = \overline{a}cd \oplus abc \oplus abc \overline{d} \oplus \overline{a}bcd \) and the minimized expression is \( f' = ab \oplus cd \). The work on efficient and exact ESOP minimization is discussed in [30] [35].

![Figure 2.12: (a) SOP minimization and (b) ESOP minimization for the given Boolean functions.](image)

The ESOP synthesis algorithm [10] starts with a cube-list, as shown in Figure 2.13. A reversible circuit is formed by starting with an ESOP cubelist representation of a Boolean function. The cubelist consists of inputs and outputs similar to a truth table. Initially, an empty circuit i.e with \( 2n + m \) qubits and 0 gates is created. Here \( n \) is the number of inputs. \( 2n \) covers both positive and negative polarity of the input qubits. \( m \) denotes the number of outputs. Each cube in the cube-list is then mapped on to the circuit. The circuit formation from the cube-list is performed according to these steps:

1. Create an empty circuit of size \( 2n + m \).

2. Insert input qubits \( (x_0, x_1, \ldots, x_{2n}) \) in the circuit, where \( n \in \mathbb{Z}^+ \), for every complemented and uncomplemented literal in an ESOP formulation such as \( a \) and \( a' \). The outputs of these corresponding inputs are labeled as ‘g’ or garbage value.

3. Insert constant qubits \( (k_0, k_1, k_2, \ldots, k_m) \) in the circuit, where \( m \in \mathbb{Z}^+ \). These qubits are initialized to a value of 0 or 1 and remain constant throughout. The outputs for ‘m’ inputs are labeled as \( f1, f2, f3 \ldots, fm \) respectively.
4. Scan the cube-list with the inputs \((x_0, x_1, \ldots, x_i)\) and outputs \((x_0, x_1, \ldots, x_j)\), where \(i, j\) are positive integers. Add a Toffoli gate to the circuit for each cube in the cubelist with \(x_i = 1\) as a positive control and \(x_i = 0\) as a negative control.

An example of a circuit resulting from this process is given in Figure 2.14.

![Figure 2.13: Full adder ESOP cube-list](image)

![Figure 2.14: Reversible circuit](image)

### 2.4.2 Transformation-based Synthesis

Transformation-based synthesis takes an \(n\) variable reversible function specification as input and produces an \(n \times n\) reversible circuit. The approach [24] is more convenient than the other approaches such as proposed in [17] but requires a reversible function as an input. There are two algorithms for transformation based synthesis, the basic algorithm and the bidirectional algorithm. We will only discuss the basic algorithm here. The algorithms derive Toffoli gates by manipulating the input or output bits of a given truth table. The transformation techniques avoid an extensive search of the best collection of gates for the near-optimal reversible circuit.

The basic algorithm is a simple greedy approach which generates Toffoli gates by exploiting the output side of the specification. According to [24] a reversible function can be represented by an ordered set of integers such as \(\{4, 1, 0, 7, 6, 3, 5, 2\}\). Thus, the function over these integers is defined as \(f(0) = 4, f(1) = 1\) and so on. Initially, consider a reversible function as the mapping over \(\{0, 1, \ldots, 2^n - 1\}\) bits. The algorithm iterates over the following steps:
1. Check the integer function \( f(0) \). If \( f(0) \neq 0 \), invert the 1-bits of the corresponding to the \( f(0) \) output bits. For each inversion a single Toffoli gate of size one (TOF1) or a NOT gate is required. Identify the transformed function as \( f^+(0) = 0 \).

2. For every integer \( i = \{0, 1, \ldots, 2^n - 1\} \), if \( f^+(i) \neq i \), a transformation to a new specification \( f^{++}(i) \) is required and the Toffoli gates map the transformation \( f^+(i) \rightarrow i \). Otherwise, if \( f^+(i) = i \).

Figure 2.15 illustrates the manipulation of the bits during each transformation of the specifications in bold. Firstly, a NOT gate is applied to the bit \( a^0 \) of the specification (i) function \( f(0) \) and then, the corresponding gate \( TOF1(a^0) \) is added to the circuit. In the next step, we map \( f^+(5) \rightarrow 5 \) with the application of \( TOF3(\{c^1,b^1\},a^1) \) on (ii) and \( TOF3(\{a^3,c^3\},b^3) \) on (iii). Lastly, to map \( f^+(6) \rightarrow 6 \) we use \( TOF3(\{c^4,b^4\},a^4) \) on (iv). The mapping of these gates to the circuit shown in the steps above in the process of transformation is in reverse order. The resultant mapping of the transformation to a reversible Toffoli gate cascade is shown in Figure 2.16.

The algorithm generates the circuit with at most \((m - 1)2^m + 1\) gates for a \( m \) variable specification with the complexity of \( O(n2^n) \) [24]. Sometimes the final specification does not map the outputs with their correct inputs. In that case an output permutation [24] is applied to the specification. In order to reduce the circuit width template matching [24] is performed on the resultant circuit. The templates are proposed in the paper [24].

### 2.4.3 Search-based Synthesis

Search-based synthesis methods traverse through a search tree built on the factors of the Boolean expression. The search tree is explored for the best set of factors in a path consisting of smallest expressions to build a circuit of minimal quantum cost. Positive Polarity Reed-Muller (PPRM) expansion is the most commonly used method to generate a search tree for a reversible function. The PPRM expansion is obtained only from uncomplemented variables available in ESOP form. The expansion has a canonical form of
2.4. REVERSIBLE LOGIC SYNTHESIS TECHNIQUES

<table>
<thead>
<tr>
<th>cba</th>
<th>(i)</th>
<th>(ii)</th>
<th>(iii)</th>
<th>(iv)</th>
<th>(v)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$c^1b^0d^0$</td>
<td>$c^1b^1d^1$</td>
<td>$c^2b^2d^2$</td>
<td>$c^3b^3d^3$</td>
<td>$c^4b^4d^4$</td>
</tr>
<tr>
<td>000</td>
<td>001</td>
<td>000</td>
<td>000</td>
<td>001</td>
<td>000</td>
</tr>
<tr>
<td>001</td>
<td>000</td>
<td>001</td>
<td>001</td>
<td>000</td>
<td>001</td>
</tr>
<tr>
<td>010</td>
<td>011</td>
<td>010</td>
<td>010</td>
<td>010</td>
<td>010</td>
</tr>
<tr>
<td>011</td>
<td>010</td>
<td>011</td>
<td>011</td>
<td>011</td>
<td>011</td>
</tr>
<tr>
<td>100</td>
<td>101</td>
<td>100</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>101</td>
<td>111</td>
<td>110</td>
<td>111</td>
<td>101</td>
<td>101</td>
</tr>
<tr>
<td>110</td>
<td>110</td>
<td>111</td>
<td>110</td>
<td>111</td>
<td>110</td>
</tr>
<tr>
<td>111</td>
<td>110</td>
<td>111</td>
<td>110</td>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>

Figure 2.15: An example of manipulating the truth table in transformation-based synthesis.

\[
\begin{align*}
  a & \oplus b & d^0 \\
  b & \oplus c & b^0 \\
  c & \oplus c & c^0
\end{align*}
\]

Figure 2.16: Reversible circuit for transformation-based synthesis.

\[
f(x_1, x_2, x_3, \ldots, x_n) = a_0 \oplus a_1 x_1 \oplus a_n x_n \oplus a_1 x_1 x_2 \oplus a_1 x_1 x_3 \ldots \oplus a_n x_{n-1} x_n \ldots \oplus a_1 x_1 x_2 \ldots x_n [13]
\]

Here \( a \in \{0, 1\} \), constant value and \( x_i \) are all uncomplemented variables. Figure 2.18 shows the PPRM expansion of the truth table in Figure 2.17.

<table>
<thead>
<tr>
<th>c b a</th>
<th>c_0</th>
<th>b_0</th>
<th>a_0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0 0 1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0 1 0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0 1 1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1 0 1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1 1 0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1 1 1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
a_0 = a \oplus 1 \\
b_0 = b \oplus c \oplus ac \\
c_0 = b \oplus ab \oplus ac
\]

Figure 2.17: Reversible Function

Here we describe a search-based algorithm from [13] which uses PPRM expansion to derive a reversible circuit from a given Boolean function. The algorithm begins with the PPRM expansion of the function \( f(v_1, v_2, v_3, \ldots, v_n) \). Figure 2.19 shows the expansion of
2.4. REVERSIBLE LOGIC SYNTHESIS TECHNIQUES

an example function. The algorithm works as follows:

1. Create and initialize a root node (Node 0) of the search tree with the PPRM expansion as shown in Figure 2.19(a).

2. Push Node 0 into a priority queue for further exploration. The root node is set to the current best solution.

3. For every output variable in the PPRM expansion of the node being explored, consider all the factors $v_{out,i}$ that do not contain $v_i$. For example for $a_{out} = a \oplus 1 \oplus bc \oplus ac$ the factors are 1 and $bc$, as they do not contain the literal $a$.

4. Substitute $v_i = v_i \oplus factor$ to create child nodes for the further exploration. These factors label the edges of the search tree. Each substitution should reduce the number of terms of the synthesized child node.

5. The child node is explored further if the terms in the child node are less than the parent node. Insert the child node into the priority queue and update the best solution as the child node.

6. The algorithm iterates over these steps until no more nodes in the priority queue are left to explore and the best solution is found.

7. The algorithm returns the path which consists of best factors for a given function. The best factors are recognized if the terms are decreased after the substitution. The path from the root node to the best solution creates the desired reversible circuit. The path guarantees to construct a circuit with minimal number of gates. Each factor on the edge of the path in the tree relates to a Toffoli gate in the circuit.

2.4.4 BDD based Synthesis

Most of the logic synthesis techniques discussed above in this chapter lack efficiency to deal with the large number of variables in a Boolean function. Binary Decision Diagrams
(BDD) and PPRM based synthesis approaches are more efficient in terms of run-time, since they use more compact data structures (tree structure). On the other hand all the other approaches depend on a truth table for the synthesis process.

In BDD a Boolean function $f : \mathbb{B}^n \rightarrow \mathbb{B}$ is represented by a directed acyclic graph $G = (V, E)$ where the Shannon decomposition [45]:

$$f = \bar{x}_i f_{x_i=0} + x_i f_{x_i=1} \quad (1 \leq i \leq n)$$

for any integer $n,$ is carried out on each node $v \in V$ labeled by $x_i$ of a BDD. Here $x_i$ is the variable of a Boolean function and, $(f_{x_i=0})$ is the function $f$ when $x_i = 0$ and $(f_{x_i=1})$ is the function $f$ when $x_i = 1$. The node $v$ has two types of outgoing edges $\{0$-edge, 1-edge$\} \in E$. The 0-edge = low($f$) where low($f$) is $(f_{x_i=0})$ and 1-edge = high($f$) where high($f$) is $(f_{x_i=1})$ in a BDD. The low($f$) and high($f$) can be any internal node marked as a sub-function or a terminal node. The terminal nodes of a BDD have values either 0 or 1. Figure 2.20(a) illustrates a BDD representing the function $x_1 \oplus x_2$.

A network of Toffoli gates is created by adding reversible gates for each node $v$ in a BDD. The reversible gates for each node depend on the type of the node. For a general case, a node has a cascade of Toffoli gates such as shown in Figure 2.20(b) and (c). Other types of nodes are specified in Table 2.2 [45].

The size of a BDD is defined by the number of nodes a BDD consists of. Shared nodes are an important component to significantly reduce the size of a BDD. Any node $v$ with more than one predecessor is identified as a shared node. Figure 2.21 displays an example of a BDD with a shared node and the corresponding reversible circuit. Complementary edges are also considered as a technique to decrease the count of nodes. With complementary edges the function and its negation can be represented by the same node [7].

The algorithm to generate a reversible circuit using BDDs is as follows:

1. Generate a BDD for the Boolean function $f$ to be synthesized.

2. Scan every node in the BDD. If the node $v$ is the identity of the input variable $x_i$
2.4. REVERSIBLE LOGIC SYNTHESIS TECHNIQUES

Table 2.2: Toffoli gates for BDD node types with additional constant line [45].

<table>
<thead>
<tr>
<th>BDD</th>
<th>Toffoli gates</th>
<th>BDD</th>
<th>Toffoli gates</th>
</tr>
</thead>
<tbody>
<tr>
<td>![BDD Diagram]</td>
<td>![Toffoli Gate Diagram]</td>
<td>![BDD Diagram]</td>
<td>![Toffoli Gate Diagram]</td>
</tr>
</tbody>
</table>

$$\text{low}(f) = 0 \text{ or } 1 \text{ and high}(f) = 0 \text{ or } 1$$, then no constant circuit line is added. The node is represented by the input circuit line.

3. Otherwise the Toffoli gates shown in Table 2.2 for each node type are added to the circuit.

4. The successors $\text{low}(f)$ and $\text{high}(f)$ of the node $v$ are preserved using an additional constant line for each if the successors are shared nodes or identity of the input variable. In this case templates shown in Table 2.2 are used. If none of the above cases apply on the nodes, then the template shown in Figure 2.20(c) is used.

The size of a reversible circuit depends on the size of the corresponding BDD. Considering a BDD of size $k$ (K nodes) for a Boolean function $f$ of $n$ variables, a reversible circuit with at most $k + n$ circuit lines is generated. The resultant circuit consists of the maximum of $3 \cdot k$ gates since at most 3 gates are added to the circuit for each node in a BDD. In the worst case scenario a BDD can have $2^n$ nodes for a single output function.
2.4. REVERSIBLE LOGIC SYNTHESIS TECHNIQUES

Node 0

\[ \begin{align*}
    c_{out} &= c \oplus ab \oplus ac \\
    b_{out} &= b \oplus c \oplus ac \\
    a_{out} &= a \oplus 1
\end{align*} \]

Node 1.0

\[ \begin{align*}
    c_{out} &= c \oplus ab \oplus ac \\
    b_{out} &= b \oplus c \oplus ac \\
    a_{out} &= a \oplus 1
\end{align*} \]

Node 1.1

\[ \begin{align*}
    c_{out} &= c \oplus ab \oplus ac \\
    b_{out} &= b \oplus c \\
    a_{out} &= a \oplus 1
\end{align*} \]

Node 1.2

\[ \begin{align*}
    c_{out} &= c \oplus b \oplus ab \\
    b_{out} &= b \oplus ac \\
    a_{out} &= a \oplus 1
\end{align*} \]

Node 2.0

\[ \begin{align*}
    c_{out} &= c \oplus ab \oplus ac \\
    b_{out} &= b \oplus ab \oplus ac \\
    a_{out} &= a
\end{align*} \]

Node 2.1

\[ \begin{align*}
    c_{out} &= c \oplus ab \oplus ac \\
    b_{out} &= b \oplus ac \\
    a_{out} &= a
\end{align*} \]

Node 2.2

\[ \begin{align*}
    c_{out} &= c \oplus b \oplus ab \\
    b_{out} &= b \oplus ac \\
    a_{out} &= a
\end{align*} \]

Figure 2.19: Search tree using PPRM expansion
2.4. REVERSIBLE LOGIC SYNTHESIS TECHNIQUES

Figure 2.20: A BDD and Toffoli gates for a node [45].

Figure 2.21: A BDD with shared node and the equivalent reversible circuit [45].
Chapter 3

Decision Diagrams

There are several means of evaluating an expression to obtain a value of true or false (1 or 0). One popular way is using decision diagrams to test for the value of a Boolean expression. As discussed in Chapter 2 Binary Decision Diagrams are one way to represent a function \( f \in \mathbb{B}_n \), where \( \mathbb{B}_n \) states the set of all \( n \) variable Boolean functions. Each variable \( x_1, x_2, \ldots, x_i \) assigned to a node is tested for an input value defining a path from the root to the leaf node of the tree. The path may consist of a 0-edge or a 1-edge. Leaf nodes give the output value for the sequence of input values assigned to each node variable. We now proceed to discuss different categories of binary decision diagrams.

3.1 Reduced Ordered BDD (ROBDD)

Before understanding the functionalities of a ROBDD it is important to define an OBDD. The authors of [22] define an OBDD as follows:

**Definition 3.1.** Considering the order of the variables \( \rho = (x_1, x_2, \ldots, x_n) \) an Ordered Binary Decision Diagram is a directed acyclic graph with respect to the order \( \rho \). An OBDD satisfies the following properties:

1. There is exactly one root and two nodes labeled by the constants 0 and 1. These two nodes do not have any outgoing edges and are called sinks.

2. Each internal node is marked by a variable \( x_i \) and has two outgoing edges namely its s1-edge and 0-edge. These edges are labeled by 1 and 0 respectively.

3. The sequence in which the variables occur in a path from the root to the sink is the same as the order of the variables defined by \( \rho \). This means if there exists a path from
3.1. REDUCED ORDERED BDD (ROBDD)

The variable ordering in a BDD plays an important role to reduce the size of a BDD. Finding the best variable ordering is a NP-Hard problem [6]. However, the package CUDD [43] has a feature for implementing variable ordering using a sifting algorithm [33].

A matter of concern with OBDDs is the occurrence of redundancies of the following types:

1. The 0-edge and 1-edge of a node $v$ lead to the same successor which means no new information is produced at node $v$.

2. The same information is represented by the OBDD in the form of similar subgraphs.

Therefore, to resolve these issues we define ROBDDs with the reduction rules.

**Definition 3.2.** An OBDD is a ROBDD if the following cases exist [22]

1. There must not exist any node $v$ with $high(v) = low(v)$.

2. No two nodes $u$ and $v$ should exist such that the similar subgraphs are rooted at nodes $u$ and $v$.

Figure 3.1 displays an OBDD and a ROBDD of the function $f = x_2 x_3 + x_1 x_2 \overline{x_3}$ with the variable order of $x_1 < x_2 < x_3$. The 0-edge and 1-edge is displayed by a dashed edge and
3.2. ORDERED FUNCTIONAL DECISION DIAGRAM (OFDD)

regular edge respectively. This definition leads to the reduction rules for OBDDs to form ROBDDs. There are two reduction rules as stated below [22]:

1. **Elimination rule**: For a node \( v \) if the 0-edge and 1-edge have the same successor node \( u \), then eliminate \( v \) and redirect all the incoming edges of node \( v \) to node \( u \).

2. **Merging Rule**: If two internal nodes \( u \) and \( v \) with the same variable name have their 1-edge directed to the same node and 0-edge directed to the same node, then eliminate either \( u \) or \( v \) and redirect all the incoming edges of the removed node to the remaining node.

Figure 3.2 shows the two reduction rules.

![Reduction rules for OBDDs](image)

Figure 3.2: Reduction rules for OBDDs [22].

3.2 Ordered Functional Decision Diagram (OFDD)

Earlier in Chapter 3 in BDD-based synthesis we defined the Shannon’s decomposition type for constructing BDDs. Here we add two other decomposition types, namely the PPRM or Davio decompositions.

Consider a Boolean function \( f \) for \( n \) variables. The functions \( f_0, f_1 \) and \( f_2 \) are defined as:

\[
\begin{align*}
  f_0(x) &= f(x_1, \ldots, x_{n-1}, 0) \\
  f_1(x) &= f(x_1, \ldots, x_{n-1}, 1) \\
  f_2(x) &= f_0(x) \oplus f_1(x)
\end{align*}
\]

Given the above definitions the two decomposition types can be defined by:
3.3. ORDERED KRONECKER DECISION DIAGRAM (OKFDD)

![Diagram](image)

Figure 3.3: Elimination rule for OFDDs

\[
f = f_0 \oplus x_n f_2 \text{ for Positive Davio Decomposition.}
\]

\[
f = f_1 \oplus \overline{x_n} f_2 \text{ for Negative Davio Decomposition.}
\]

**Definition 3.4.** Ordered Functional Decision Diagrams (OFDD) are similar to OBDDs except the function \(f_i\) on node \(i\) is computed by the Reed-Muller decomposition as given below [22]:

1. A node \(v\) labeled as 1 or 0 represents the function \(f_v = 1\) or \(f_v = 0\) respectively.

2. A node \(v\) labeled as \(x_i\) whose low\((v)\) and high\((v)\) denote the functions \(h\) and \(g\) respectively, defines \(f_v = g \oplus x_i h\).

The nodes of the OBDD in Figure 3.1 for the function \(f = x_2 x_3 + x_1 \overline{x_2} \overline{x_3}\) represent an OFDD for the function \(f = x_1 x_2 x_3 \oplus x_1 \oplus x_2 x_3\) provided that the node decomposition is Davio. The nodes of the OFDD represent the following functions:

1. Nodes labeled by \(x_3\): \(f_{x_3,1}(x) = 1 \oplus x_3 \cdot 0 = 1, \quad f_{x_3,2}(x) = 0 \oplus x_3 \cdot 1 = x_3\)

2. Nodes labeled by \(x_2\): \(f_{x_2,1}(x) = 1 \oplus x_2 x_3, \quad f_{x_2,2}(x) = 0 \oplus x_2 x_3 = x_2 x_3\)

3. Node labeled by \(x_1\): \(f(x) = x_2 x_3 \oplus x_1 (1 \oplus x_2 x_3) = x_2 x_3 \oplus x_1 \oplus x_1 x_2 x_3\)

The reduction rules for OFDDs are similar to the rules applied to BDDs except:

1. **Elimination rule:** If the successor of the 1-edge of a node \(v\) is 0, then eliminate \(v\) and direct all the incoming edges to the successor of the 0-edge of node \(v\).

Figure 3.3 shows the elimination rule for OFDDs.
3.3 Ordered Kronecker Decision Diagram (OKFDD)

For some classes of Boolean functions, the OFDD representation is more compact than OBDD. Furthermore, there are some classes of functions that have polynomial size OKFDDs but exponential size OBDDs and OFDDs [3]. OKFDDs are an elegant combination of OFDDs and BDDs which showcase the advantages of each type.

**Definition 3.5.** The Ordered Kronecker Decision Diagram is a representation type where each node \( v \) labeled by \( x_i \) is assigned a decomposition type using Decomposition Type List (DTL) \( d : \{d_1, d_2, \ldots, d_n\} \) where \( d_i \in \{S, pD, nD\} \). Here \( S \) is Shannon decomposition, \( nD \) is negative Davio decomposition and \( pD \) is positive Davio decomposition [22].

In Figure 3.4(a) the OKFDD representing a function \( f = x_1x_2x_4 \oplus x_1x_2 \overline{x_3} \oplus x_1 \overline{x_3} \oplus \overline{x_1}x_2x_4 \) is constructed by the variable ordering \( x_1 < x_2 < x_3 < x_4 \). Each node has a decomposition type given by the DTL \( d : \{S, pD, nD, S\} \). The node \( x_1 \) decomposes \( f \) into \( f_{x_2,1} = x_2x_4 \) and \( f_{x_2,2} = x_2x_4 \oplus x_2 \overline{x_3} \oplus \overline{x_3} \). The node \( x_2,2 \) is further decomposed into \( f_{x_3,1} = \overline{x_3} \) and \( f_{x_3,2} = \overline{x_3} \oplus x_4 \). Lastly, the node \( x_3,2 \) factors into \( f_{x_4} = x_4 \) and 1 while the node \( x_2,1 \) factors into \( f_{x_4} = x_4 \) and 0.

The algorithm to generate the reversible circuit from an OKFDD is similar to the BDD synthesis algorithm discussed in section 2.4 except that an OKFDD uses all the decomposition types for node structures such as shown in Table 3.1. Figure 3.4b shows the equivalent circuit for the OKFDD in Figure 3.4a using the Toffoli gates given in Table 3.1.
3.3. ORDERED KRONECKER DECISION DIAGRAM (OKFDD)

(a) OKFDD for the function \( f = x_1x_2x_4 \oplus x_1x_2x_3 \oplus x_1x_3 \oplus x_1x_2x_4 \).

(b) Equivalent circuit.

Figure 3.4: OKFDD with the specified DTL [3] and its equivalent circuit.
### 3.3. ORDERED KRONECKER DECISION DIAGRAM (OKFDD)

Table 3.1: Toffoli gate circuits for node structures of decomposition types.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Shannon</th>
<th>Positive Davio</th>
<th>Negative Davio</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1" alt="Diagram" /></td>
<td><img src="image2" alt="Diagram" /></td>
<td><img src="image3" alt="Diagram" /></td>
<td><img src="image4" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image5" alt="Diagram" /></td>
<td><img src="image6" alt="Diagram" /></td>
<td><img src="image7" alt="Diagram" /></td>
<td><img src="image8" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image9" alt="Diagram" /></td>
<td><img src="image10" alt="Diagram" /></td>
<td><img src="image11" alt="Diagram" /></td>
<td><img src="image12" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image13" alt="Diagram" /></td>
<td><img src="image14" alt="Diagram" /></td>
<td><img src="image15" alt="Diagram" /></td>
<td><img src="image16" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image17" alt="Diagram" /></td>
<td><img src="image18" alt="Diagram" /></td>
<td><img src="image19" alt="Diagram" /></td>
<td><img src="image20" alt="Diagram" /></td>
</tr>
<tr>
<td><img src="image21" alt="Diagram" /></td>
<td><img src="image22" alt="Diagram" /></td>
<td><img src="image23" alt="Diagram" /></td>
<td><img src="image24" alt="Diagram" /></td>
</tr>
</tbody>
</table>

---

31
Reducing lines using OKFDDs

The demand for an efficient and compact reversible circuit has led to the introduction of various optimization techniques in the past years. Optimization of a reversible circuit is carried out based on the parameters discussed in section 2.3. One of these parameters, which counts the number of lines (qubits) in a reversible circuit is line count. As we know now qubits are a limited resource and thus it is desirable to have reversible circuits generated with minimum line count. In this chapter we discuss the upper and the lower bounds on the number of lines required by a reversible circuit to realize a Boolean function as explained in [20]. We also propose an algorithm to reduce the number of lines generated using OKFDDs for reversible logic synthesis.

4.1 Bounds on reversible circuit lines

**Theorem 1.** For a given function \( f : \mathbb{B}^n \rightarrow \mathbb{B}^m \) the number of garbage bits required is at most \( \log_2 \mu \) where \( \mu \) denotes the maximum repetition of an output pattern [20].

**Proof.** As discussed in Section 1.1 an irreversible function is embedded into a reversible function to be implemented. Since an irreversible function is not bijective, the outputs of an irreversible truth table repeat for some input values. To make the output of the truth table unique additional bits are added. For example if the output pattern \( (o_1,o_2,..o_m) \) occurs most frequently i.e. \( \mu \) times, then \( \lceil \log_2(\mu) \rceil \) new bits are required to make the output unique. Therefore, \( 2\lceil \log_2(\mu) \rceil \) extra output patterns are created.

**Corollary 1.** (Lower Bound) A reversible circuit requires at least \( m + \lceil \log_2(\mu) \rceil \) lines to implement an irreversible Boolean function [20].

Let \( \mu \) be the maximum number of times an output pattern repeats itself in an irreversible
truth table of a Boolean function. Then, approximately $\lceil \log_2(\mu) \rceil$ additional lines (garbage outputs) are required to convert an irreversible function to a reversible function [20]. Therefore, to implement a Boolean function $\mathbb{B}^n \rightarrow \mathbb{B}^m$ at least $m + \lceil \log_2(\mu) \rceil$ lines are required. The final circuit consists of $n$ inputs, $m$ outputs and $\lceil \log_2(\mu) \rceil$ additional constant inputs. If in any case the number of inputs $n$ is greater than $\lceil \log_2(\mu) \rceil$ then the minimum number of lines required is $n$ as the number of lines in a circuit cannot be less than the number of variables present in the function.

The value of $\mu$ is evaluated by scanning the truth table of the function. The computation involves a two-level description [48] of a SOP truth table of a Boolean function. A two-level description represents a SOP truth table with ‘1’ for positive variables, ‘0’ for negative variables and ‘-’ for don’t care values. This representation is similar to ESOP cube-list Figure 2.13. Each row denotes the conjunction of the variables. In a two-level description a function is represented by the disjunction of each row while in ESOP a function is represented by the Ex-OR of each row. The complexity of computing the value of $\mu$ depends on the size of the truth table. Therefore, the complexity is polynomial in the number of rows of the truth table (or other two-level description); however it may be exponential in the number of variables if there are no don’t care conditions in the truth table.

**Example:** Consider the truth table shown in Figure 4.1 in two-description level format. The first row (11 − 01) has two don’t cares so the output 101 is repeated $2^1 = 2$ times. Similarly, for the second row (10 − 01) which represents the conjunction of $x_1, \overline{x_2}, \overline{x_4}$ and $x_5$ the output pattern 110 is repeated $2^1 = 2$ times plus $2^3 = 8$ times in fourth row (00 − − −). Figure 4.1 shows how many times an output pattern is repeated. Since the output 110 is repeated the maximum number of times (10 times), the value of $\mu = 10$ and $\lceil \log_2(\mu) \rceil = \lceil \log_2(10) \rceil = 4$. Thus, the minimum number of lines required to implement the function is $m + \lceil \log_2(\mu) \rceil = 3 + 4 = 7$ where $m = 3$.

**Corollary 2.** (Upper Bound) A given function $f : \mathbb{B}^n \rightarrow \mathbb{B}^m$ requires at most $n + m$ circuit lines to implement [48].
According to the **Theorem 1** the minimal number of lines required to implement a function is \( m + \lceil \log_2(\mu) \rceil \). Considering the worst case the maximum number of times an output pattern can repeat for \( 2^n \) i.e. \( \mu = 2^n \) for an \( n \) variable Boolean function. Therefore, in this case the number of lines required is \( m + \lceil \log_2(2^n) \rceil = m + n \).

### 4.2 The Algorithm

Algorithm MOKFDD is the proposed algorithm for line reduction in OKFDDs. The algorithm explains the process of synthesizing a Boolean function from a given OKFDD. Before discussing the algorithm we introduce the shared node ordering for the algorithm.

**Shared nodes:** Shared node implementation is previously explained in section 2.4 under BDD based synthesis. In OKFDDs the implementation of a shared node structure depends on the decomposition types of the nodes involved. As discussed earlier if two or more nodes have a same successor node \( S \) then \( S \) is a shared node. To obtain an optimized circuit from an OKFDD we introduce the node ordering in case of a shared node. The illustration of the concept is given in Figure 4.2 using Positive Davio (pD) decomposition where the function \( f_3 \) is represented by a shared node.

In Figure 4.2 the node labeled \( x_j \) is a shared node as it is shared by two \( x_i \) nodes. In the case of a shared node, the node \( (v_{f_1}) \) with the 1-edge leading to the \( x_j \) is realized first and then the node \( (v_{f_2}) \) with the 0-edge to \( x_j \). In this case the circuit has no additional constant circuit lines but when synthesizing from a OKFDD the shared node implementation

<table>
<thead>
<tr>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
<th>( x_4 )</th>
<th>( x_5 )</th>
<th>( f_1 )</th>
<th>( f_2 )</th>
<th>( f_3 )</th>
<th>( \text{Freq} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( 2^1 = 2 )</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( 2^1 = 2 )</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>( 2^3 = 8 )</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>( 2^3 = 8 )</td>
</tr>
<tr>
<td>0</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( 2^1 = 2 )</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>( 2^2 = 4 )</td>
</tr>
</tbody>
</table>

Figure 4.1: Calculating \( \mu \) for an irreversible truth table
4.2. THE ALGORITHM

depends on the decomposition type and [45] gives a shared node implementation, but only for Shannon decomposition types. The cost of the circuit depends on the decomposition type of the nodes.

![Diagram](image)

Figure 4.2: Shared node and equivalent circuit

The algorithm MOKFDD is based on synthesizing reversible circuits using the factors of a Boolean function through a decision diagram. Our addition to the work in [42] is a new sub-circuit or template (highlighted in Table 4.1) for positive-Davio decompositions and an ordering in which the variables are to be addressed when a shared node is encountered in the OKFDD. A template in the decision diagram is a representation of a node structure in the form of reversible gates in a reversible circuit according to the given decomposition type. These templates can be computed using the corresponding decomposition type formulas. The algorithm MOKFDD introduces a process of mapping an OKFDD to a reversible circuit in such a way that the circuit lines are minimized. The input for the algorithm MOKFDD is the OKFDD or acyclic directed graph $G(V,E)$ generated by the algorithm in [42]. The output of the algorithm MOKFDD is a reversible circuit termed as $rev\_cascade$. Initially, we take an empty circuit $rev\_cascade$ and add lines or gates when required. In the next three steps we define the depth $d$ of the OKFDD, the number of nodes $k$ at each level and an empty list $L$ respectively. In step 5 we start traversing the graph bottom-up for each level $l$ from the non-terminal nodes (level $d-1$) to the root node. In step 6 each unvisited node $v^l_j$ at level $l$ is scanned for the decomposition type. In Step 7 and 8 we implement each node $v^l_j$ with an update in $rev\_cascade$ and mark it as visited. In the next step we search for
the parent nodes $V_j$ of each node $v^j_l$. Steps 10 to 15 define the cases for a shared node. For a shared node there can be only two parent nodes ($|v_j| = 2$). Step 11 checks the case where (considering the Figure 4.2) 0-edge of $f_2 (v^p_{j1})$ and 1-edge of $f_1 (v^p_{j2})$ leads to the same shared node $x_j$. In this case $f_1 (v^p_{j2})$ is inserted in the $L$ first and then $f_2 (v^p_{j1})$. The second case is the alternate case of the first one. Finally in Steps 17 to 19 the list $L$ is traversed and each element or node in the list $v$ is implemented according to the insertion order before going to the next level. Each of these nodes are marked visited and $rev\_cascade$ is updated.

The procedure $parent(G, v^j_l)$ searches for the parent nodes $v^p_{j1}$ and $v^p_{j2}$ of the input node $v^j_l$ at level $l$ in the graph by tracking the incoming edges to the input node. In the procedure $sub\_circuit(v)$ each node $v$ in an OKFDD is implemented by selecting a sub-circuit from Table 4.1 depending on the node decomposition type and structure. The most frequent sub-circuits for each decomposition type shown in Table 4.1 are given in [42]. We have added a new sub-circuit for Davio decomposition of node structure 3 which requires no additional lines and only one CNOT gate.

**Input:** A KFDD (directed acyclic graph) $G(V, E)$ where $|V| = n$ and $\{0\text{-edge}, 1\text{-edge}\} \in E$.

**Output:** A reversible circuit $rev\_cascade$ with minimal lines.

$update\_cascade(rev\_cascade, sub\_circuit(v))$ procedure evaluates each sub-circuit selected by the procedure $sub\_circuit(v)$ adds to the main circuit $rev\_cascade$. If the $rev\_cascade$ has the input states required by the sub-circuit then they merge otherwise new constant additional lines are added for the required input states.

**Example:** Figure 4.3 shows an OKFDD for the function $x_1(x_2 \oplus x_3) \oplus x_2^3x_3$ where $\rho = x_1 < x_2 < x_3$. Firstly, $f$ decomposes into $f_3 = x_2^3x_3 \oplus x_3$ and $f_2 = x_2 \oplus x_3$ by $nD$ decomposition type. Then, $f_3$ decomposes into $f_1 = x_3$ and terminal node 1 by $pD$ type. Next, $f_2$ into $f_1 = x_3$ and 0 by $pD$ type. Finally, $f_1$ decomposes into terminal nodes 0 and 1 by Shannon type.
4.2. THE ALGORITHM

Algorithm 1 MOKFDD($G$)

1: $\text{rev\_cascade} \leftarrow \emptyset$
2: $d \leftarrow \text{depth of an OKFDD}$.
3: $k \leftarrow \text{no. of nodes in each level of an OKFDD}$.
4: $L \leftarrow \emptyset$
5: for each level $l$; $d - 1 \leq l \leq 0$ do
6:   for each unvisited node $v^l_j \in V$; $0 \leq k$ do
7:     $\text{rev\_cascade} \leftarrow \text{update\_cascade}(\text{rev\_cascade}, \text{sub\_circuit}(v^l_j))$
8:     Mark $v_j$ visited.
9:   $V_j \leftarrow \text{parent}(G, v^l_j)$
10:  if $|V_j|$ equals 2 then
11:     Case 1: 0-edge of $v_j^{p1} \in V_j$ and 1-edge of $v_j^{p2} \in V_j$ share same node then
12:        Insert $v_j^{p2}$ and then $v_j^{p1}$ in $L$.
13:     Case 2: 1-edge of $v_j^{p1} \in V_j$ and 0-edge of $v_j^{p2} \in V_j$ share same node then
14:        Insert $v_j^{p1}$ and then $v_j^{p2}$ in $L$.
15:     end if
16:  end for
17:  for each element $v \in L$ do
18:     $\text{rev\_cascade} \leftarrow \text{update\_cascade}(\text{rev\_cascade}, \text{sub\_circuit}(v))$
19:     Mark $v$ visited.
20:  end for
21: end for

To generate the circuit using MOKFDD, we traverse the OKFDD from node $x_3$ to the root node $x_1$ level wise starting from $l_2$. The suitable sub-circuit is selected from Table 4.1 for each node. Here node $f_1$ at $l_2$ is a shared node and thus, $f_3$ is implemented before $f_2$. For node $f_1$ at $l_2$ since the function is $x_3$ (a single variable) we require only a single line in the circuit. At $l_1$ for node $f_2$ the pD node structure #3 is used from the Table 4.1. Similarly, for the node $f_3$, the pD node structure 5 is used which is similar to a S decomposition. Lastly, for node $f$ at $l_0$, the nD node structure #1 is used. Figures 4.3b and 4.3c show the equivalent circuit implementations using the previous algorithm [42] and our algorithm respectively. As illustrated our approach produces a smaller circuit (quantum cost = 12 and line count = 4) as compared to the previous approach (quantum cost = 13 and line count =
4.2. THE ALGORITHM

Algorithm 2 Procedure \textit{parent}(G, v_j^i)

\begin{verbatim}
V_j ← parent nodes of v_j^i
Return V_j
\end{verbatim}

Algorithm 3 Procedure \textit{sub.circuit}(v)

\begin{verbatim}
if child nodes of v are non-terminal then
    circuit(v) ← template 1 or 2 from Table 4.1
else
    circuit(v) ← other matching template from Table 4.1
end if
Return circuit(v)
\end{verbatim}

![Diagram of OKFDD and its reversible circuit from different algorithms.](image)

(a) OKFDD for the function \(x_1(x_2 \oplus x_3) \oplus x_2 x_3\)

(b) Previous approach

(c) Our approach

Figure 4.3: OKFDD and its reversible circuit from different algorithms.

5) [42].

In order to compare our approach with the existing BDD based synthesis, consider the SOP formulation for the function used in the example above i.e. \(f = x_1 + x_2 + x_3\). The
4.3. DISCUSSION

Algorithm 4 Procedure \textit{update\_cascade}(\textit{rev\_cascade}, \textit{sub\_circuit}(v))

\begin{algorithmic}
\Function{update\_cascade}{\textit{rev\_cascade}, \textit{sub\_circuit}(v)}
\If{\textit{rev\_cascade} has all the required \textit{sub\_circuit}(v) input states}
\State Merge \textit{sub\_circuit}(v) and \textit{rev\_cascade}
\Else
\State Add extra constant lines for the missing input states to \textit{rev\_cascade} and merge \textit{sub\_circuit}(v)
\EndIf
\State Return \textit{rev\_cascade}
\EndFunction
\end{algorithmic}

OBDD for this function using the order of nodes $\rho = x_1 < x_2 < x_3$ is shown in Figure 4.4(a). As illustrated the nodes in an OBDD are more compared to the nodes in an OKFDD for the same function. This shows that the number of nodes in a decision diagram depend on the decomposition type being used. The resultant circuit of the OBDD is shown in Figure 4.4(b). The number of lines in the circuit is 6 and the quantum cost is 29. In this case the cost of the circuit is more than the cost of the circuit synthesized by OKFDD even for a simple function with three variables. The comparison of our approach to OBDD based synthesis for the reversible benchmarks is shown in the results chapter 5. The comparison

![OBDD and Reversible circuit](image)

Figure 4.4: (a) OBDD for the function $f = x_1 + x_2 + x_3$ and (b) its equivalent reversible circuit.

of our approach with other algorithms help to analyze the benefits and limitations of the proposed algorithm. The next section discusses these points along with the contribution.
### 4.3 Discussion

The proposed algorithm MOKFDD uses OKFDD for the synthesis of a reversible circuit. In OBDD synthesis approach the function is decomposed into smaller sub-functions using Shannon decomposition until a constant value is reached. These sub-functions are then mapped on to the reversible circuit. The drawback of using an OBDD based synthesis is that the circuit uses more additional lines to preserve the sub-function for future use compared to OKFDD based synthesis approach for large functions. However, OBDD based synthesis is the first hierarchical synthesis approach which can synthesize Boolean
functions upto 70 variables [42]. The illustration for this is given by an example in the previous section. This reason motivates us to use an OKFDD based synthesis for the algorithm MOKFDD. In an OKFDD approach the functions are decomposed using the Davio decomposition types with Shannon type which allows more compact representations of the sub-functions. Therefore, the advantages of MOKFDD include:

1. The algorithm can take upto 70 variables as input.

2. The resulting reversible circuit is cost effective (reduces line count) compared to other algorithms.

3. The algorithm does not traverse the truth table to generate the reversible circuit.

The algorithm MOKFDD uses the advantages of an OKFDD based synthesis along with the reduction in the circuit lines. It overcomes the limitations introduced by other heuristic methods of synthesis [24], [16], [13] such as limited input variables, scanning of large truth tables and costly circuits. However, there are few limitations associated with the algorithm. They are as follows:

1. Although MOKFDD produces cost effective reversible circuits for large inputs, the circuits are still expensive to implement practically.

2. The size of the decision diagram may increase exponentially with the increase in the number of input variables.

With all these advantages and disadvantages discussed above MOKFDD significantly reduces the number of lines in the reversible benchmarks. The experimental evaluation done in the next chapter shows the reduction table for the benchmarks. The three important contributions from this study are:

1. Introduction of a new template in the pD node structure significantly reduces the number of lines in the circuits.
2. The proposed ordering for the shared nodes allows to implement the template for the line reduction efficiently.

3. The level-wise traversal of the decision diagram checks and implements the shared nodes according to the shared node order in a single pass.

The results form the first contribution depend on the frequency of the presence of the node structure highlighted in Table 4.1 in the decision diagram. Since the node structure is one of the most frequently used structure the line reduction in the circuits is notable. The study can be extended to find more of such node structures with different decomposition types to further reduce the number of circuit lines.
Chapter 5

Experimental Evaluation

The algorithm MOKFDD is implemented in C++ in Revkit [41]. The reversible functions in Table 5.1 are from Revlib [44]. The algorithm to generate an OKFDD is given in Revkit under KFDD-based synthesis algorithms which includes the PUMA package for decompositions and optimizing algorithms. The sifting algorithm [33] is used by PUMA to find a variable ordering and DTL that results in the fewest nodes in the OKFDD. The circuits obtained by our algorithm (Table 5.1) have been verified using Revkit’s equivalence checker. The runtime of the experiment including the verification process for all the benchmarks shown in Table 5.1 is few seconds. The experiment was performed on a 1.9 GB, Intel Core 2 Duo processor Linux machine.

In Table 5.1 the first column shows the functions. The next two columns consist of the number of inputs and outputs for corresponding functions. The results of our algorithm are compared with the results of the previous KFDD approach [42] and BDD approach [45]. The notation ‘L’ denotes the number of lines in the circuit while ‘QC’ and ‘GC’ denote the quantum cost and gate count respectively. The changes in the metrics are shown by ‘ΔL’, ‘Δ QC’ and ‘Δ GC’. The results show a significant decrease in the number of lines as well as in quantum cost and gate count. In the best case (e.g. plus127) the line reduction is 42% compared to the KFDD approach and 29% (e.g. tial) compared to the BDD approach. The average line reduction is approximately 10% in comparison to the KFDD approach. Comparing the quantum cost values the average reduction is around 7% and 23% for the KFDD and the BDD approaches respectively.

Since an OKFDD uses all the decomposition types with the variable ordering, this type of decision diagram is more likely to generate a smaller realization compared to other DD
based algorithms for large functions. Some of the functions show great improvement such as plus127 and tial due to the frequent presence of node structure #3. We can see that if a pD decomposition type #3 is used then only one gate and no additional lines are required. Although there are a few functions that do not show any improvement compared to lines in KFDD approach, they display QC or GC minimization such as sqrt8 and ex2. We hypothesize that when a node is shared with more than two nodes then an additional line is required to preserve the function for future use. This compensates for the previous removal of lines.

A comparison of our experimental results to the lower bound discussed in section 4.1 on the number of lines in a reversible circuit is shown in Table 5.2. Column 1 and 2 show the number of inputs and outputs of a function respectively. Column 3 shows the lower bound ($\lceil \log_2 (\mu) \rceil$) or the least additional lines required for a function. Column 4 shows the total number of lines ($m + \lceil \log_2 (\mu) \rceil$) required. The last column in the table shows the number of lines required by a function using our approach. The difference in the values for some of the functions is small such as ex1 and rd_32, and large in other functions such as plus63 and sqn. The difference in the values may depend on the decomposition of a function. Importantly, the comparison shows that there is scope for more improvement.

The variable ordering of a DD plays an important role in the designing of a reversible circuit. The size of a DD depends on the chosen variable order [33]. The number of gates and lines in a reversible circuit depends on the number of the non-terminal nodes in a DD. If a DD has $k$ nodes and each node needs at least a single extra line then at most $k + n$ lines are required to implement an $n$ input Boolean function [45]. Similarly, $3 \times k$ Toffoli gates are required to realize a DD with $k$ nodes considering that each node can have at most 3 Toffoli gates [45]. In order to minimize the size of DD, the sifting algorithm [33] is used to obtain the best possible variable ordering for a DD. The algorithm tests for all the possible variable orderings and selects the best ordering to minimize the size of a DD. It initially takes a variable and swaps it with other variables until the best position is confirmed. Since
each position is checked for the best results the algorithm is a brute force algorithm. Each variable is swapped until no significant decrease in the size of the DD occurs.

Considering the fact that the variable ordering reduces the size of the DD, we can further investigate the changes in a DD by comparing the variable ordering to favor a particular set of node structures. In Figure 5.1 an OKFDD for a function \( f = x_1(x_3 \oplus x_4) \oplus x_3 x_4 \oplus x_2 x_3 \oplus x_2 x_4 \) is given with the variable order of \( x_1 < x_2 < x_3 < x_4 \) and pD decomposition on all the nodes. This variable ordering for the function is best according to the sifting algorithm due to the minimum number of nodes. Randomly testing the other variable orders an OKFDD with the structure shown in Figure 5.1 is obtained. The quantum cost is 17 while the line count is 4. This is a case where the number of nodes is increased compared to the original OKFDD to favor the pD node structure #3 from Table 5.2. Comparing the resultant circuits from both OKFDDs Figure 5.2 gives a more compact and less expensive circuit with a quantum cost of 12 and line count 4. The reason behind the cost reduction in spite of an increase in the number of nodes is that the increased number of nodes represents the variable itself as a function such as \( x_1 \) and \( x_2 \). Therefore, these nodes require a single input line (no additional line) to represent the function in the circuit. There may be other different cases where an increase in the number of nodes may increase the cost of the circuit.
Figure 5.1: (a) OKFDD with the variable order \( x_1 < x_2 < x_3 < x_4 \) and (b) its equivalent circuit.

Figure 5.2: (a) OKFDD with the variable order \( x_4 < x_3 < x_2 < x_1 \) and (b) its equivalent circuit.
## 5. Experimental Evaluation

### Table 5.1: Experimental Results for the Algorithm

<table>
<thead>
<tr>
<th>Function Name</th>
<th>#in</th>
<th>#out</th>
<th>Our approach</th>
<th>OKFDD approach [42]</th>
<th>BDD approach [45]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>L QC GC</td>
<td>L QC GC</td>
<td>∆L ∆QC ∆GC</td>
</tr>
<tr>
<td>Rd_32</td>
<td>3</td>
<td>2</td>
<td>6 16 8 8 20 12</td>
<td>-2 -4 -4</td>
<td>6 22 10 0 -6 -2</td>
</tr>
<tr>
<td>mod10</td>
<td>4</td>
<td>4</td>
<td>12 60 24 14 64 28</td>
<td>-2 -4</td>
<td>-4 13 80 28 -1 -20 -4</td>
</tr>
<tr>
<td>aos_two_three</td>
<td>3</td>
<td>3</td>
<td>35 15 10 37 17</td>
<td>-1 -2 -2</td>
<td>9 44 16 0 -9 -1</td>
</tr>
<tr>
<td>plus127</td>
<td>13 13</td>
<td>26 85 41 37 97 53</td>
<td>-11 -12 -12</td>
<td>25 98 54 1 -13 -13</td>
<td></td>
</tr>
<tr>
<td>plus63</td>
<td>12 12</td>
<td>23 78 38 34 89 49</td>
<td>-10 -12 -11</td>
<td>23 89 49 1 -11 -11</td>
<td></td>
</tr>
<tr>
<td>radd</td>
<td>5</td>
<td>8</td>
<td>17 59 27 21 68 36</td>
<td>-4 -9</td>
<td>-9 28 217 73 -11 -158 -46</td>
</tr>
<tr>
<td>rd53</td>
<td>5</td>
<td>3</td>
<td>14 64 32 15 69 37</td>
<td>-1 -5 -5</td>
<td>- - - - - - -</td>
</tr>
<tr>
<td>rd73</td>
<td>3</td>
<td>7</td>
<td>23 107 51 25 115 59</td>
<td>-2 -8</td>
<td>8 25 217 73 2 -110 -22</td>
</tr>
<tr>
<td>rd54</td>
<td>4</td>
<td>5</td>
<td>30 153 69 33 161 77</td>
<td>-3 -8</td>
<td>8 34 304 104 -4 -151 -35</td>
</tr>
<tr>
<td>root</td>
<td>3</td>
<td>5</td>
<td>51 413 153 52 415 159</td>
<td>-1 -2</td>
<td>-6 45 444 140 6 -31 13</td>
</tr>
<tr>
<td>sgr6</td>
<td>6</td>
<td>12</td>
<td>48 304 112 50 315 123</td>
<td>-2 -11</td>
<td>-11 49 486 154 -1 -182 -42</td>
</tr>
<tr>
<td>sgr8</td>
<td>8</td>
<td>4</td>
<td>28 166 62 28 170 66</td>
<td>0 -4</td>
<td>- - - - - - -</td>
</tr>
<tr>
<td>zk1</td>
<td>7</td>
<td>4</td>
<td>14 42 22 20 56 32</td>
<td>-6 -10</td>
<td>-10 14 86 30 0 -20 -8</td>
</tr>
<tr>
<td>zk4ml</td>
<td>7</td>
<td>4</td>
<td>14 46 22 20 56 32</td>
<td>-6 -10</td>
<td>-10 14 86 30 0 -20 -8</td>
</tr>
<tr>
<td>add6</td>
<td>12 12</td>
<td>40 183 79 39 184 80</td>
<td>1 -1</td>
<td>-1</td>
<td>54 499 159 -14 -316 -80</td>
</tr>
<tr>
<td>ade4</td>
<td>8</td>
<td>5</td>
<td>16 54 26 23 65 37</td>
<td>-7 -11</td>
<td>-11 16 74 34 0 -20 -8</td>
</tr>
<tr>
<td>bw</td>
<td>5</td>
<td>28</td>
<td>78 581 237 81 593 249</td>
<td>-3 -12</td>
<td>-12 87 943 307 -9 -362 -70</td>
</tr>
<tr>
<td>cm82a</td>
<td>5</td>
<td>3</td>
<td>10 31 15 12 36 20</td>
<td>-2 -5 -5</td>
<td>13 82 30 -3 -51 -15</td>
</tr>
<tr>
<td>con1</td>
<td>7</td>
<td>2</td>
<td>15 94 30 17 100 36</td>
<td>-2 -6</td>
<td>-6 - - - - - -</td>
</tr>
<tr>
<td>cycle</td>
<td>12 12</td>
<td>28 97 49 34 104 56</td>
<td>-6 -7</td>
<td>-7</td>
<td>39 202 78 -11 -105 -29</td>
</tr>
<tr>
<td>dc1</td>
<td>4</td>
<td>7</td>
<td>21 141 45 22 146 50</td>
<td>-1 -5</td>
<td>-5</td>
</tr>
<tr>
<td>inc</td>
<td>7</td>
<td>9</td>
<td>56 442 158 58 447 171</td>
<td>-2 -5</td>
<td>-13</td>
</tr>
<tr>
<td>xor195</td>
<td>5</td>
<td>1</td>
<td>6 6 6 10 10 10</td>
<td>-4 -4</td>
<td>-4</td>
</tr>
<tr>
<td>sym9</td>
<td>9</td>
<td>1</td>
<td>30 150 70 28 154 74</td>
<td>2 -4</td>
<td>-4</td>
</tr>
<tr>
<td>ex1</td>
<td>5</td>
<td>1</td>
<td>6 6 6 10 10 10</td>
<td>-4 -4</td>
<td>-4</td>
</tr>
<tr>
<td>ex2</td>
<td>5</td>
<td>1</td>
<td>11 50 18 11 48 20</td>
<td>0 2</td>
<td>-2</td>
</tr>
<tr>
<td>max46</td>
<td>9</td>
<td>1</td>
<td>62 664 216 67 684 228</td>
<td>-5 -20</td>
<td>-12</td>
</tr>
<tr>
<td>sym10</td>
<td>10 1</td>
<td>1</td>
<td>38 259 103 40 266 110</td>
<td>-2 -7</td>
<td>-7</td>
</tr>
<tr>
<td>life175</td>
<td>9</td>
<td>1</td>
<td>26 159 67 31 168 76</td>
<td>-5 -9</td>
<td>-9</td>
</tr>
<tr>
<td>9x4ml</td>
<td>9</td>
<td>1</td>
<td>30 150 70 28 154 74</td>
<td>2 -4</td>
<td>-4</td>
</tr>
<tr>
<td>sas</td>
<td>4</td>
<td>10</td>
<td>74 562 186 73 568 192</td>
<td>1 -6</td>
<td>-6</td>
</tr>
<tr>
<td>tal</td>
<td>7</td>
<td>14</td>
<td>410 4185 1681 419 4179 1703</td>
<td>-9 6</td>
<td>-22</td>
</tr>
<tr>
<td>url1</td>
<td>9</td>
<td>3</td>
<td>384 4320 1628 390 4372 1672</td>
<td>-6 -32</td>
<td>-44</td>
</tr>
<tr>
<td>url2</td>
<td>8</td>
<td>7</td>
<td>206 2276 920 269 2304 948</td>
<td>-3 -28</td>
<td>-28</td>
</tr>
<tr>
<td>wam</td>
<td>7</td>
<td>4</td>
<td>18 93 37 19 95 39</td>
<td>-1 -2</td>
<td>-2</td>
</tr>
<tr>
<td>majority</td>
<td>5</td>
<td>1</td>
<td>10 37 17 10 38 18</td>
<td>0 -1</td>
<td>-1</td>
</tr>
<tr>
<td>sym6</td>
<td>6</td>
<td>1</td>
<td>16 76 28 16 77 29</td>
<td>0 -1</td>
<td>-1</td>
</tr>
<tr>
<td>condic</td>
<td>23 2</td>
<td>52 264 104 53 264 108</td>
<td>-1 0</td>
<td>-4</td>
<td>- - - - - -</td>
</tr>
<tr>
<td>cm85a</td>
<td>13 1</td>
<td>35 161 61 35 164 64</td>
<td>0 -3</td>
<td>-3</td>
<td>36 275 87 -1 -114 -26</td>
</tr>
<tr>
<td>clip</td>
<td>9</td>
<td>5</td>
<td>66 546 214 69 566 226</td>
<td>-3 -20</td>
<td>-12</td>
</tr>
<tr>
<td>e64</td>
<td>64 64</td>
<td>195 897 385 195 899 387</td>
<td>0 -2</td>
<td>-2</td>
<td>- - - - - -</td>
</tr>
<tr>
<td>cpv</td>
<td>24 109</td>
<td>819 567 2305 623 5668 2332</td>
<td>-6 -9</td>
<td>-27</td>
<td>- - - - - -</td>
</tr>
</tbody>
</table>

---

This table compares the performance of our approach with OKFDD and BDD approaches for various functions, showing differences in performance metrics such as L, QC, GC, ∆L, ∆QC, and ∆GC.
Table 5.2: Lower bound Table

<table>
<thead>
<tr>
<th>Function name</th>
<th>#inputs (n)</th>
<th>#outputs (m)</th>
<th>Lower Bound $LB = \lceil \log_2(\mu) \rceil$</th>
<th>#Total Lines (LB + m)</th>
<th>Our Approach #Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rd_32</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>mod10</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>one_two_three</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>plus127</td>
<td>13</td>
<td>13</td>
<td>0</td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>plus63</td>
<td>12</td>
<td>12</td>
<td>0</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>radd</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>rd53</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>rd73</td>
<td>7</td>
<td>3</td>
<td>9</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>rd84</td>
<td>8</td>
<td>4</td>
<td>7</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>root</td>
<td>8</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>sqn</td>
<td>7</td>
<td>3</td>
<td>7</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>sqrt6</td>
<td>6</td>
<td>12</td>
<td>7</td>
<td>19</td>
<td></td>
</tr>
<tr>
<td>sqrt8</td>
<td>8</td>
<td>4</td>
<td>9</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>z4</td>
<td>7</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>z4ml</td>
<td>7</td>
<td>4</td>
<td>8</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>add6</td>
<td>12</td>
<td>7</td>
<td>13</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>adr4</td>
<td>8</td>
<td>5</td>
<td>9</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>bw</td>
<td>5</td>
<td>28</td>
<td>8</td>
<td>36</td>
<td></td>
</tr>
<tr>
<td>cm82a</td>
<td>5</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>con1</td>
<td>7</td>
<td>2</td>
<td>7</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>cycle</td>
<td>12</td>
<td>12</td>
<td>0</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>dc1</td>
<td>4</td>
<td>7</td>
<td>2</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>inc</td>
<td>7</td>
<td>9</td>
<td>3</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>xor195</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>sym9</td>
<td>9</td>
<td>1</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>ex1</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>ex2</td>
<td>5</td>
<td>1</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>max46</td>
<td>9</td>
<td>1</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>sym10</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>life175</td>
<td>9</td>
<td>1</td>
<td>9</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>9sym1</td>
<td>9</td>
<td>1</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>sao</td>
<td>10</td>
<td>4</td>
<td>10</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>tial</td>
<td>14</td>
<td>8</td>
<td>15</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>urf1</td>
<td>9</td>
<td>9</td>
<td>0</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>urf2</td>
<td>8</td>
<td>8</td>
<td>0</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>wim</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>majority</td>
<td>5</td>
<td>1</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>sym6</td>
<td>6</td>
<td>1</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>cordic</td>
<td>23</td>
<td>2</td>
<td>26</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>cm85a</td>
<td>11</td>
<td>3</td>
<td>11</td>
<td>14</td>
<td></td>
</tr>
<tr>
<td>clip</td>
<td>9</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>e64</td>
<td>64</td>
<td>64</td>
<td>63</td>
<td>128</td>
<td></td>
</tr>
<tr>
<td>cps</td>
<td>24</td>
<td>109</td>
<td>28</td>
<td>137</td>
<td></td>
</tr>
</tbody>
</table>
Bibliography


