IMPROVED ORDERING OF ESOP CUBES FOR TOFFOLI NETWORKS

by

Zakaria Hamza

Bachelor of Computer Science, UNB, 2009

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Computer Science

Supervisor(s): Dr. Gerhard Dueck, Ph.D., Computer Science
Examin ing Board: Dr. Kenneth Kent, Ph.D., Computer Science, Chair
Dr. David Bremner, Ph.D., Computer Science
External Examiner: Dr. Howard Li, Ph.D., Computer Engineering, UNB

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

May, 2011

©Zakaria Hamza, 2011
Abstract

Logic synthesis deals with the problem of finding a cost-effective realization of a given logic function. This uses several state-of-the-art techniques and involves several tools of mathematical origin. In recent years reversible logic has been suggested to address the power consumption associated with computation. To accomplish such a task, synthesis of reversible logic function is needed. Several new synthesis methods have been developed. In this thesis methods are proposed that improve on a given synthesis method. In particular, interest has been demonstrated in the optimization of this class of circuits which use the particular Exclusive-or Sum of Product (ESOP) terms representation. The advantage this representation format offers is in the ease of mapping the function to a network of Toffoli logic gates. However, this synthesis technique provides non-optimal results which could be improved. This problem has roots in both the representation and mapping processes of synthesis. It is well-known that the order of the terms in the ESOP expression will have a direct effect on the cost of the implementation. The problem of finding the optimal order can be mapped into the Generalized Traveling Salesman Problem. Another route of optimization involves reducing the number of terms used to represent the function. This can be achieved by canonical representation of functions. Both of these have proven to offer enhancements over existing synthesis techniques and have been developed in this thesis. Experimental results show that significant improvements can be achieved with the proposed methods.
# Contents

Abstract

Contents

List of Figures

List of Tables

1 Introduction

2 Background
  2.1 Function Representation in Logic Synthesis .......................... 6
    2.1.1 Truth Tables, Minterms, Maxterms and Operations .............. 6
    2.1.2 Product Terms and Cubes ..................................... 10
    2.1.3 SOP’s, FPRM’s, PPRM’s and ESOP’s ............................ 15
    2.1.4 Cube Properties ............................................. 19
  2.2 Toffoli Gates and Networks ........................................ 26
  2.3 ESOP to Toffoli Network Synthesis ................................ 32
  2.4 Reversible Logic Synthesis ....................................... 34

3 Solution Approaches
  3.1 Problem Description .............................................. 40
  3.2 Changing Function Polarity ....................................... 41
    3.2.1 Assessing Polarity of a Cube ................................. 42
    3.2.2 Rewriting Function Expressions to Alter Polarity .......... 44
  3.3 Cube List Ordering Method ....................................... 52
    3.3.1 Traveling Salesman Problem Approach .......................... 54
    3.3.2 The Generalized Traveling Salesman Problem .................. 56
3.3.3 The Traveling Salesman Problem as an Integer Linear Program ........................................... 60
3.3.4 ESOP Cube Ordering Represented as an Integer Linear Program ........................................... 64

4 Solution Application and Testing
   4.1 Benchmark Testing ................................................................. 76

5 Conclusion & Future Work

Bibliography

Appendices

A Example solution for function misex1

Vita
# List of Figures

2.1 Sample function in PLA format. ........................................ 19
2.2 First 3 gates of the Toffoli series ................................. 27
2.3 Sample function in ESOP PLA format. .............................. 34
2.4 Equation (2.3) represented as a Toffoli network. ............... 34
2.5 PLA definition of 1-bit adder function. ............................ 36

3.1 Function representations of $\overline{a} \oplus \overline{b} \overline{c} \oplus c$. ........................................ 43
3.2 Toffoli network representation of 1-bit adder function. ......... 47
3.3 Polarization of sample function $f$ from Equation (2.3). ...... 51
3.4 Unordered versus ordered ESOP Toffoli network. ................. 53
3.5 Triangle inequality - $\text{cost}(e_{xy}) + \text{cost}(e_{yz}) \geq \text{cost}(e_{xz})$. .......... 57
3.6 GTSP mapping of an ESOP list ................................. 59
3.7 Unordered versus ordered ESOP Toffoli network ................. 63

4.1 Polarized ESOP Toffoli network ................................. 76

A.1 GLPK solver code. ........................................ 94
A.2 Group nodes – 5 main, 2 subgroups. .............................. 95
A.3 Adjacency matrix of all nodes .............................. 95
## List of Tables

<table>
<thead>
<tr>
<th>Table Number</th>
<th>Table Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>A sample truth table.</td>
<td>7</td>
</tr>
<tr>
<td>2.2</td>
<td>Truth table definition of the Exclusive-OR (EXOR), OR, NOT and AND functions.</td>
<td>8</td>
</tr>
<tr>
<td>2.3</td>
<td>Truth table definition of sample function.</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>CHD and HD of a path of cubes.</td>
<td>22</td>
</tr>
<tr>
<td>2.5</td>
<td>Basic cube list properties.</td>
<td>25</td>
</tr>
<tr>
<td>2.6</td>
<td>Re-ordered cube list properties.</td>
<td>25</td>
</tr>
<tr>
<td>2.7</td>
<td>Toffoli gate cost function [1, 29, 40].</td>
<td>31</td>
</tr>
<tr>
<td>2.8</td>
<td>Sample gate cost table.</td>
<td>32</td>
</tr>
<tr>
<td>2.9</td>
<td>Conversion table of 1-bit adder function from SOP to ESOP.</td>
<td>39</td>
</tr>
<tr>
<td>3.1</td>
<td>Truth table definition of 1-bit adder function.</td>
<td>46</td>
</tr>
<tr>
<td>3.2</td>
<td>Sample Simplex tableau.</td>
<td>62</td>
</tr>
<tr>
<td>3.3</td>
<td>Solved sample Simplex tableau.</td>
<td>62</td>
</tr>
<tr>
<td>4.1</td>
<td>Benchmark results: polarization algorithm</td>
<td>78</td>
</tr>
<tr>
<td>4.2</td>
<td>Benchmark results: sorting algorithm</td>
<td>78</td>
</tr>
<tr>
<td>4.3</td>
<td>Benchmark results: both algorithms</td>
<td>79</td>
</tr>
<tr>
<td>4.4</td>
<td>Benchmark results: no improvement found</td>
<td>80</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Reversible logic is an emerging field in the computing sciences as well as other related fields of research \cite{9, 48}. It is used to generate circuit design and realizations from boolean mathematical functions. Synthesis techniques used in reversible logic offer ways of circuit representation, verification and optimization. However, much work and research needs to be done to further the advancements and efficiency in these techniques.

In this thesis, the attempt is to develop better procedures for the synthesis of reversible functions. In particular, reversible functions can be represented using Exclusive-OR Sum Of Products (ESOP) term lists. These can further be mapped into Toffoli networks in an inefficient manner \cite{8}. Further exploration is done for optimizations in such networks with the objective of reducing their costs of realization. The idea behind this reduction is mainly fueled by modern computing demands and information processing techniques that need to become more efficient. Hardware optimization is
critical when it comes to efficiently implementing circuit designs, and with energy conservation becoming a critical issue, this is evolving into a necessity. These networks are used to represent such circuits and logic functions in various applications including low-power Complementary Metallic-Oxide-Semiconductor (CMOS) design, nanotechnology, optical computing [21] and quantum computing [32, 36]. Circuit optimization can also be critical in energy conservation as stated by Landauer [23] since inefficient computation results in information loss which is a key factor in energy loss in the form of heat dissipation. In particular, irreversible circuits suffer from this type of information entropy and can be made reversible to further increase their efficiency. In fact, a proof by Bennet in [3] demonstrates that in order for computations to be completely free of energy loss, they would have to be reversible. Thus computational reversibility is believed to be part of the solution to the synthesis problem. Many attempts have been made to solve this problem but no proven optimal solution exists yet for functions with large numbers of input variables. In this thesis however, an improvement to the algorithm used in [8] was found for the ESOP-based synthesis method and offers improved reversible circuit design.

The problem in this thesis is identified as an optimization problem where the resources required to realize a Toffoli network from an ESOP expression should be minimized. Specifically, the reduction of the distance between individual terms in the list is important in order to avoid inserting
unnecessary gates to realize the function. Variable negation between individual terms is required where variable polarities do not match and so every time this occurs in the representation a NOT gate is inserted. Effectively, the problem can be formulated as a representation and ordering problem where the reduction of inversions of variables is desired.

Two methods have been developed in order to solve this problem. The first method involves analyzing and changing the polarity of the function in order to represent it in a compact fashion and thus reduce the number of gates required to realize the function. The second uses a graph representation of the problem by mapping it into an instance of the Generalized Traveling Salesman Problem (GTSP) [15] – which is a version of the Traveling Salesman Problem (TSP) [18, 25] – and solving it using Integer Linear Programming (ILP) [38]. This particular method aims at sorting terms in order to minimize the overall distance between them as it is a direct factor in reducing the number of NOT gates. The reason behind the required insertion of NOT gates in a given Toffoli network is in the fact that in the function specification the input variable needs to be inverted depending on whether there was a polarity change or not. Each ESOP term maps to a single Toffoli gate and between two gates may be one or more NOT gates which serve the purpose of inversion. Some inversions can in fact been shown as unnecessary. For example, in [8], ESOP synthesis can be further optimized since the $m + n$ lines that are used in networks can be arranged in a minimal-maximal con-
figuration of \( n \) NOT gates. Even with functions that are completely negative in polarity, if the gates are ordered or changed in a consistent manner within the function one can achieve such a configuration.

These methods will be discussed thoroughly in their respective sections. However, a significant background in logic synthesis is required to understand the proposed solutions which will be covered in the coming section. The thesis is organized into four sections each outlining the topic being dealt with. First and foremost, the reader should consult Chapter (2): Background for prior knowledge of the materials which form the backbone of this research. Following this reading, Chapter (3): Solution Approaches will explain the approaches used for the development of the proposed solutions and their methodologies. Finally, results will be given in Chapter (4): Solution Application and Testing where benchmark tests have been recorded and explained with analysis. A short chapter at the end outlines future works that could lead into further improvements of the proposed methods and possibly a final optimal synthesis approach.
Chapter 2

Background

In this thesis, the objective is to reduce as much as possible the number of NOT operations used in a Toffoli network generated from an ESOP expression. In order to clarify the procedure needed to achieve this goal, a few concepts are necessary for the reader to understand. Beginning with basic logic synthesis function representations, this section will elaborate on the terminology used throughout this thesis along with mathematical definitions. Further along, there will be brief explanations regarding data structures, formats and notations used to represent such functions.
2.1 Function Representation in Logic Synthesis

Boolean logic is the backbone used for all function representation in this thesis. All boolean functions have inputs and outputs which are designated as variables, just like in any mathematical function. Some key terminology used to represent a boolean function will be defined below.

2.1.1 Truth Tables, Minterms, Maxterms and Operations

Basic data structures used to represent and store boolean functions range from truth tables to mathematical functions. The truth table offers a particular advantage for visualization over others as it exhausts all possibilities of a function in a table.

Definition 1. A truth table is an exhaustive tabular representation of a function of \( n \) inputs that iterates through all input variable assignments and lists them in a tabular format with their corresponding outputs. Each row represents a variable assignment to an output and there are exactly \( 2^n \) rows.
Example 1. A sample boolean function with 2 inputs ($x_1$ and $x_2$) and single output ($o_1$) is represented as a truth table below.

$$
\begin{array}{ccc}
\hline
x_1 & x_2 & o_1 \\
\hline
0 & 0 & 1 \\
0 & 1 & 0 \\
1 & 0 & 0 \\
1 & 1 & 0 \\
\hline
\end{array}
$$

Table 2.1: A sample truth table.

Transformations in a function need to be represented in order to look at combinations of variables at a time. Thus, the concept of an operation – implemented in a circuit as a logic gate – is introduced which represents transformations of variables in the function. Some basic operations are outlined in Example (2).

Definition 2. A boolean operation or operation is a transformation applied to one or more variables.
Example 2. Some basic operators are outlined below using a truth table.

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>$a \oplus b$ (EXOR)</th>
<th>$a+b$ (OR)</th>
<th>$\bar{a}$ (NOT)</th>
<th>$ab$ (AND)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</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>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2.2: Truth table definition of the Exclusive-OR (EXOR), OR, NOT and AND functions.

Definition 3. A logic gate or gate is the implementation of one or more combined operations in a circuit.

In a truth table, as inputs are exhaustively combined, one can observe specific mappings to output variables. A particular variable is depicted as a literal. Some important representations can be derived from patterns of these mappings.

Definition 4. A literal is either a variable or a negated (complemented) variable.
Definition 5. A **minterm** is defined as a product (AND operation) of distinct input variables that maps to a single positive output literal.

Definition 6. A **maxterm** is defined as a sum (OR operation) of distinct input variables that maps to a single negated output literal.

Example 3. In the truth table defined in Example (1), the only minterm of 2 literals is: \( \overline{x_1}x_2 \). Similarly, the 3 corresponding maxterms of 2 literals for the function are: \( \overline{x_1} + x_2 \), \( x_1 + \overline{x_2} \) and \( x_1 + x_2 \). Note that the complement of any minterm is a maxterm and vice versa.

Maxterms and minterms are used to describe mappings to zeroes and ones in the output respectively. This is useful in attempting a canonical representation of a function. Their use in function representation can be extended in order to represent functions in a more compact algebraic fashion instead of a tabular exhaustive one such as the truth table. This is important when using algorithms on such functions as the runtime or resource requirements is exponential relative to the number of input terms.


2.1.2 Product Terms and Cubes

A more general use expression exists – the product term – and can be derived by taking the product (AND operation) of literals.

**Definition 7.** A product term or term can be expressed using the following formal notation:

\[ P = p_1 p_2 \ldots p_{n-1} p_n \]  

(2.1)

where \( p_i \in \{ \phi, x_i, \bar{x}_i \} \), \( i \in [1, n] \) and \( n \) is a positive integer representing the number of variables. Each \( p_i \) represents a variable.

The product term expression can also be expressed as a cube using elements from the set \( \{0, 1, \bar{\ }\} \) respectively corresponding to whether the literal is negated, positive or absent – referred to as a don’t care and represented as \( \phi \) – in the product term (see Example (5)).

**Definition 8.** A boolean cube or cube is a better representation of a product term for computing and storage. Thus given a product term \( P = \prod_{i=1}^{n} p_i \), the corresponding cube can be repre-
sent as:

\[ C = \prod_{i=1}^{n} c_i \] where \( c_i \in \{1, 0, -\} \) and \( c_i = \begin{cases} 
1 & \text{if } p_i = x_i \\
0 & \text{if } p_i = \overline{x_i} \\
- & \text{if } p_i = \phi
\end{cases} \)

**Example 4.** The sample boolean function \( f \) with 4 inputs, single output \( f(x_1, x_2, x_3, x_4) \) and truth table defined below (see Table (2.3)) can be represented as a sum of product terms as defined in Equation (2.2) using the OR operator. \( f \) can also be represented with the EXOR operator as defined in Equation (2.3). Note: \( f \) can be represented in both ways without modifying product terms as these are already mutually exclusive and due to the operator overlap in definitions for OR and EXOR.
Table 2.3: Truth table definition of sample function.

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$f(x_1, x_2, x_3, x_4)$</th>
</tr>
</thead>
<tbody>
<tr>
<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>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</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>1</td>
</tr>
<tr>
<td>0</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>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The list of 8 maxterms is given below:

1. $x_1 + x_2 + x_3 + \overline{x}_4$

2. $x_1 + x_2 + \overline{x}_3 + x_4$
3. $x_1 + \bar{x}_2 + x_3 + x_4$

4. $x_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4$

5. $\bar{x}_1 + x_2 + x_3 + \bar{x}_4$

6. $\bar{x}_1 + x_2 + \bar{x}_3 + \bar{x}_4$

7. $\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + x_4$

8. $\bar{x}_1 + \bar{x}_2 + x_3 + \bar{x}_4$

Similarly, the list of 8 minterms is given below:

1. $\bar{x}_1\bar{x}_2\bar{x}_3x_4$

2. $\bar{x}_1\bar{x}_2x_3\bar{x}_4$

3. $\bar{x}_1x_2\bar{x}_3\bar{x}_4$

4. $\bar{x}_1x_2x_3x_4$

5. $x_1\bar{x}_2\bar{x}_3x_4$

6. $x_1\bar{x}_2x_3x_4$

7. $x_1x_2x_3\bar{x}_4$

8. $x_1x_2x_3x_4$

Using the product term terminology, the number of minterms can be reduced by taking advantage of the don’t care notation when minterms with only one differing literal are found:

1. $\bar{x}_1\bar{x}_2\bar{x}_3x_4 \rightarrow 1$
2. $\bar{x}_1 \bar{x}_2 x_3 x_4 \rightarrow 1$

3. $\bar{x}_1 x_2 \bar{x}_3 x_4 \rightarrow 1$

4. $\bar{x}_1 x_2 x_3 x_4 \rightarrow 1$

5. $x_1 \bar{x}_2 \bar{x}_3 x_4 + x_1 \bar{x}_2 x_3 x_4 = x_1 \bar{x}_2 x_4 \rightarrow 1$

6. $x_1 x_2 \bar{x}_3 \bar{x}_4 + x_1 x_2 x_3 x_4 = x_1 x_2 x_3 \rightarrow 1$

Below is a generated list of cubes from these product terms:

1. 0001 $\rightarrow 1$

2. 0010 $\rightarrow 1$

3. 0100 $\rightarrow 1$

4. 0111 $\rightarrow 1$

5. 101$-1$ $\rightarrow 1$

6. 111$-1$ $\rightarrow 1$

**Example 5.** Take product term $P = x_0 x_1 x_5 \bar{x}_6$, the corresponding cube is represented as 11---10.

These formalisms are effective for certain types of function representations and although product terms are more human readable, cube terms will be favored throughout this thesis for compactness. In order to represent a function however, multiple cubes or product terms may be required to map
all function outputs. Having defined the truth table, it becomes easier to introduce other expressions that are more compact and use the features of arithmetic and algebraic representation. Some other properties of cubes are also relevant to this application and determine critical roles in ordering them.

2.1.3 SOP’s, FPRM’s, PPRM’s and ESOP’s

Product terms and cubes can be used to define other useful representation structures that are more concise. The most basic of these is the Sum Of Products (SOP) and consists of a sum – equivalent to an OR operation in boolean logic – of product terms or cubes. Any boolean function can be represented as a SOP (see Example (6)).

Definition 9. A Sum Of Products or SOP is an expression consisting of a product term list summed together using the OR operation.

Example 6. Continuing from the previous example (Example (4)) below, is a construct of the SOP from function \( f(x_1, x_2, x_3, x_4) \) defined from its product terms mapping to ones:

1. \( \bar{x}_1 \bar{x}_2 \bar{x}_3 x_4 \rightarrow 1 \)
2. \( \bar{x}_1 x_2 x_3 \bar{x}_4 \rightarrow 1 \)

3. \( x_1 x_2 \bar{x}_3 \bar{x}_4 \rightarrow 1 \)

4. \( \bar{x}_1 x_2 x_3 x_4 \rightarrow 1 \)

5. \( x_1 \bar{x}_2 x_4 \rightarrow 1 \)

6. \( x_1 x_2 x_3 \rightarrow 1 \)

below in Equation (2.2):

\[
f = \bar{x}_1 \bar{x}_2 x_3 x_4 + \bar{x}_1 x_2 \bar{x}_3 \bar{x}_4 + \bar{x}_1 x_2 x_3 x_4 + x_1 \bar{x}_2 x_4 + x_1 x_2 x_3 \quad (2.2)
\]

The representation of boolean expressions using the SOP format can further be expanded into a **Positive Polarity Reed Muller** (PPRM) expansion leaving only positive terms. In this particular definition, **variable polarity** has a different meaning as it refers to the variable itself and not its attribute as is the case in the later definition of term polarity or polarity index (see Defintion (17)) in the coming section. PPRM’s can be generalized into **Fixed Polarity Reed Muller** (FPRM) expressions – this implies that all variables have the same polarity: either all un-complemented (positive) or all complemented (negative) [42].

**Definition 10.** A **Fixed Polarity Reed Muller (FPRM)** expression is a class of product term list summed together using
the EXOR operation where all variables in all product terms have
the same polarity: either all un-complemented (positive) or all
complemented.

Definition 11. A Positive Polarity Reed Muller (PPRM)
expression is a class of product term list summed together using
the EXOR operation where all variables appear positive (un-
complemented). In other words, it is a positive FPRM.

From either of these expressions (usually the FPRM), one can obtain
an EXOR Sum Of Products (ESOP) which is the most general expression.
See Example (7) for an outline of the process.

Definition 12. An Exclusive-Or Sum Of Products or ESOP
is an expression consisting of a product term list summed together
using the EXOR operation.

Example 7. As the original SOP of function $f(x_1, x_2, x_3, x_4)$
given in Example (6) already includes only mutually exclusive
terms (i.e.: no duplicate or overlapping product terms), one sim-
ply needs to change the sum operation from an OR to an EXOR
(see Equation (2.3) below).
$f = \overline{x_1x_2x_3x_4} \oplus \overline{x_1x_2x_3x_4} \oplus \overline{x_1x_2x_3x_4} \oplus \overline{x_1x_2x_3x_4} \oplus x_1x_2x_4 \oplus x_1x_2x_3$ (2.3)

These particular representations of functions are stored for computing in a format referred to as the **Programmable Logic Array** (PLA) format (see Example (8)). Note that changing operations in an expression without changing the product terms or cubes accordingly usually results in altering the function. Algorithms in logic synthesis exist to get from one expression format to the other, but as these are not relevant to our work, they will not be explained.

**Example 8.** This example demonstrates $f$ – defined in Equation (2.3) – represented into the ESOP PLA format. Similarly to the regular SOP PLA format, the only additional specification required is the ‘.type’ attribute which by default is usually assumed to be SOP in most PLA parsers.
Figure 2.1: Sample function in PLA format.

2.1.4 Cube Properties

Another concept in logic synthesis is the concept of Hamming Distance (HD) which is defined as the distance between two cubes. In binary cubes, this is the number of differing digits in the second cube from the first. It is worth to note that HD is unaffected by function representation format – whether the function is in a SOP or an ESOP format will not change the cost provided the product terms remain unmodified.

Definition 13. Hamming Distance (HD) \(d\) between two cubes
or product terms] is defined as the number of corresponding variables with opposing polarity indices. Don’t cares are considered to be wild-cards and thus are not considered in calculations.

Definition 14. Cumulative Hamming Distance (CHD) [for a path of cubes or product terms] is the sum of all the changes incurred in all the literals while traversing the path in the given order. It is not necessarily equal to the sum of Hamming Distances along such a path due to the factoring of don’t cares in calculations (see Example (10) for an explanation) as these simply carry forward the previous value to the next term and do not increase the sum. It is also worth to note that there is a hidden initial term of the all-positive term required for calculation consistency as all literals must start out positive.

Example 9. Given cubes 101111 and 0101-0, one can calculate them to have a HD of 4.

Don’t cares do not contribute to HD calculations as they can be either a 1 or 0. However, they contribute to the overall calculation in cumulative distance. Consequently, Cumulative Hamming Distance (CHD) defines the total distance or cost of ordering a cube list (see Example (10)). One can
note in this example that the sum of HDs is 11 and not equal to the CHD of the path. This is simply because the way HD is defined since it does not weight *don’t cares* in its calculations. Thus, only CHD properly reflects the correct distance in a path of cubes or terms. Other concepts that are relevant to distance calculations are the *path* and the *cycle*. These are both represented as a sequence or list of cubes that are seen in the particular order in which they are specified.

**Definition 15.** A *path* is a traversal from a starting cube or term to an ending one while following a specific order of cubes or terms in between that are all connected to each other.

**Example 10.** Using a path chosen from the cubes mapping to ones defining function \( f(x_1, x_2, x_3, x_4) \) \([111-, 0100, 10-1, 0010, 0001, 0111]\) one can calculate the CHD and individual HDs for pairs of terms to be as follows:
Definition 16. A cycle is a path whose starting point and ending point are the same.

Definition 17. The polarity index of a cube is represented as a positive or negative integer or 0 and is determined by the difference in population of 1’s and 0’s. Thus, the definition of a product term is $t = \prod_{i=1}^{n} l_i$ where $l_i \in \{x_i, \overline{x}_i, \phi\}$. The polarity index of a product term can be calculated according to Equation

<table>
<thead>
<tr>
<th>Path</th>
<th>HD</th>
<th>CHD</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10-1</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>0010</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>0001</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>0111</td>
<td>2</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 2.4: CHD and HD of a path of cubes.
\[ P = \sum_{i=1}^{n} s_i \text{ where } s_i = \begin{cases} 
1 & \text{if } l_i = x_i \\
-1 & \text{if } l_i = \overline{x}_i \\
0 & \text{if } l_i = \phi 
\end{cases} \tag{2.4} \]

**Definition 18.** The cost of a particular order of a list or path is defined as a metric used to measure the number of NOT operations that will be required to travel from the first term in the list or path to the last. This is synonymous to the sum of all individual literal polarities in this path or list and thus equal to the CHD.

**Definition 19.** Cumulative Polarity (CP) [of a list of product terms] is the sum of all polarity indices of each term in the list. Specific Cumulative Polarity (SCP) refers to the sum of all individual polarity indices in such a list for only a specific variable. Some other statistical measures such as average polarity and standard deviation in polarity in the list of terms are also acquired from statistical analysis of these functions which are mathematically defined below:

\[ L = \sum_{j=1}^{m} \sum_{i=1}^{n} s_{ij} \]

\[ L_k = \sum_{i=1}^{m} s_{ki} \]
where \( L \) is the CP, \( L_k \) is the SCP, \( s \) represents the polarity index of each of the \( n \) variables in a list of \( m \) product terms with \( k \) being fixed to a specific variable in the second equation. Note that \( L = \sum_{i=1}^{n} L_i \). A function is said to be polarized when the standard deviation of the cube polarities is small and the absolute value of the average polarity is near 0. When a function is completely polarized, it cannot further be re-ordered in a manner that will decrease its cost.

Example (11) demonstrates the use and application of HD and CHD in the function \( f \) in Equation (2.3). Notice how the list of terms has a total cost of 12. A cost of 9 can be achieved by re-ordering this list and the process will be demonstrated later in this thesis. It is also worth to note that in function representation, the additional complication of starting with all positive variables – in this example, our first term 1111 is not listed but required for calculation consistency – is required in our application. Thus, for all future calculations, this HD cost of an all positive term will be included to the first term as an initial setting.

Example 11. This example demonstrates the various cube properties previously defined and also outlines the reduced cost after re-ordering. The path is essentially a list of terms in a specific
order which was derived from the function in the last example (Example (10)). Below is the original path:

<table>
<thead>
<tr>
<th>Path</th>
<th>Polarity</th>
<th>CP</th>
<th>HD</th>
<th>CHD</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0100</td>
<td>-2</td>
<td>1</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>10-1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>0010</td>
<td>-2</td>
<td>0</td>
<td>2</td>
<td>9</td>
</tr>
<tr>
<td>0001</td>
<td>-2</td>
<td>-2</td>
<td>2</td>
<td>11</td>
</tr>
<tr>
<td>0111</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>13</td>
</tr>
</tbody>
</table>

Table 2.5: Basic cube list properties.

and below is the re-ordered path:

<table>
<thead>
<tr>
<th>Path</th>
<th>Polarity</th>
<th>CP</th>
<th>HD</th>
<th>CHD</th>
</tr>
</thead>
<tbody>
<tr>
<td>111-</td>
<td>3</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>10-1</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0111</td>
<td>2</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>0010</td>
<td>-2</td>
<td>4</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>0001</td>
<td>-2</td>
<td>2</td>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>0100</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 2.6: Re-ordered cube list properties.
2.2 Toffoli Gates and Networks

Some of the most widely used realization systems in reversible logic are Toffoli networks. They provide an important synthesis approach in boolean logic for ESOP’s. The basic unit of a Toffoli network is the Toffoli gate which is a class of reversible operation. Irreversible functions can be modified and then implemented using these networks and thus made reversible.

Definition 20. The Toffoli gate operation can be described as a Controlled-Control NOT (CCNOT) gate – which is a combination of the AND and EXOR operations – as it modifies only the last input by mutually excluding (EXOR) it with the product (AND) of all other input lines. A formal mathematical definition is given below:

\[ T_n(x_0, \ldots, x_i, \ldots, x_n) = \{x_0, \ldots, x_i, \ldots, x_{n-1}, x_n \oplus (x_0 x_1 \ldots x_{n-1})\} \]

\( x_i \) represents the input line variables and \( n \) represents the gate number of the series and consequently also the number of input lines. The output of this function is specified as a set of variables that map to output lines and \( n \) must be greater than 0. As the function has the same number of inputs as outputs and the mapping is one-to-one, this functions series is reversible.
**Definition 21.** The Toffoli network is essentially a cascade of Toffoli gates acting in a specific order on a defined set of boolean variables commonly represented as input lines to produce outputs. A Toffoli network is by definition reversible, thus the number of input variables is always equal to the number of output variables.

**Example 12.** The Toffoli gate series are a special type of reversible operation combining the AND and EXOR functionalities into a single operation. Below are 3 Toffoli gates from the series: the $T_1$ or NOT gate (i.e.: $T_1(a) = a \oplus 1 = \bar{a}$), the $T_2$ or CNOT gate (i.e.: $T_2(a,b) = \{a, b \oplus a\}$) and the $T_3$ gate (i.e.: $T_3(a,b,c) = \{a, b, c \oplus ab\}$) – see Figure (2.2) below. Note that the $T_1$ and $T_2$ gates are special cases.

![Figure 2.2: First 3 gates of the Toffoli series](image-url)

A Toffoli gate is represented as a special type of reversible operation in a circuit involving a combination of basic gates – see Example (12) for
examples of these gates. A cascade of such gates is defined as a Toffoli network which can be visually depicted as a series of input and output lines that are passed through Toffoli gates according to the function definition.

**Definition 22.** A line is defined as a synchronized and atomic visual representation of a variable or constant in a circuit. All lines begin in a circuit as input and end as output.

**Definition 23.** A constant line is a class of line that has a fixed value such as 0 or 1 at the input.

**Definition 24.** A variable line is a class of line that does not have a fixed value at the input.

**Definition 25.** The garbage line is a special type of output line that is discarded since it is not used.

To graphically construct a Toffoli network, one creates a separate parallel line for each input labeling them with their respective variable names or with 1 or 0 if they are constants. Note that the function should have as
many inputs as outputs, if not, the function is thus irreversible and these outputs will be created. The beginning of a line is an input, its end is its output. Even if the output is not needed, such a line is necessary to preserve reversibility in some functions – in particular, modified irreversible functions. Toffoli gates are applied to the lines depending on their variable mapping in the specified function (see Example (12)).

In order to measure Toffoli network performance, metrics are required in order to properly contrast and compare networks with respect to each other. This leads us to the definition of the concepts of Quantum Cost (QC), Gate Count (GC), and transistor cost. Another criteria that can be analyzed is the number of lines used in the network.

**Definition 26.** The Gate Count (GC) [of a Toffoli network] can be defined as the number of gates required to realize the function.

**Definition 27.** The Quantum Cost (QC) [of a Toffoli network] can be defined as the number of elementary quantum operations used to realize the function [1, 29, 40]. Used primarily to determine theoretical costs of implemented circuits with Toffoli
networks, it is described as the main criteria for evaluating implementation costs. This is an approximation to the real transistor cost of a circuit which can only be determined by implementing the circuit. The QC function is described in Table (2.7) below.
<table>
<thead>
<tr>
<th>Gate ((n))</th>
<th>Garbage ((o))</th>
<th>QC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>= 0</td>
<td>29</td>
</tr>
<tr>
<td></td>
<td>(\geq 1)</td>
<td>26</td>
</tr>
<tr>
<td>6</td>
<td>= 0</td>
<td>61</td>
</tr>
<tr>
<td></td>
<td>= 1</td>
<td>52</td>
</tr>
<tr>
<td></td>
<td>(\geq 2)</td>
<td>38</td>
</tr>
<tr>
<td>7</td>
<td>= 0</td>
<td>125</td>
</tr>
<tr>
<td></td>
<td>= 1</td>
<td>80</td>
</tr>
<tr>
<td></td>
<td>(\geq 2)</td>
<td>50</td>
</tr>
<tr>
<td>8</td>
<td>= 0</td>
<td>253</td>
</tr>
<tr>
<td></td>
<td>= 1</td>
<td>100</td>
</tr>
<tr>
<td></td>
<td>(\geq 2)</td>
<td>62</td>
</tr>
<tr>
<td>9</td>
<td>= 0</td>
<td>509</td>
</tr>
<tr>
<td></td>
<td>= 1</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>(\geq 2)</td>
<td>74</td>
</tr>
<tr>
<td>10</td>
<td>= 0</td>
<td>1021</td>
</tr>
<tr>
<td></td>
<td>= 1</td>
<td>152</td>
</tr>
<tr>
<td></td>
<td>(\geq 2)</td>
<td>86</td>
</tr>
<tr>
<td>(\geq 11)</td>
<td>= 0</td>
<td>(2^n - 3)</td>
</tr>
<tr>
<td></td>
<td>(\geq 1)</td>
<td>(24n - 88)</td>
</tr>
<tr>
<td></td>
<td>(\geq 2)</td>
<td>(12n - 34)</td>
</tr>
</tbody>
</table>

Table 2.7: Toffoli gate cost function \([1, 29, 40]\).
Example 13. The table below outlines 5 gates along with their costs calculated according to Table (2.7):

<table>
<thead>
<tr>
<th>Gate</th>
<th>Garbage ((o))</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>CNOT</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>(T_{10})</td>
<td>0</td>
<td>1,021</td>
</tr>
<tr>
<td>(T_{100})</td>
<td>99</td>
<td>1,166</td>
</tr>
<tr>
<td>(T_{1000})</td>
<td>1</td>
<td>23,912</td>
</tr>
</tbody>
</table>

Table 2.8: Sample gate cost table.

2.3 ESOP to Toffoli Network Synthesis

As mentioned previously, all boolean logic functions can be represented as SOPs and ESOPs. A Toffoli network can thus be used to represent and visualize either of these. In particular, the ESOP format is well designed for this mapping. In this section, one can find a description to the mapping process used to generate a Toffoli network from an ESOP and vice versa.

Given an ESOP one can easily generate a Toffoli network. This may require one or more gates of different sizes for each term in the list and the number and type of gates used will allow us to estimate implementation costs.
of the circuit. The network is realized one product term or cube at a time in a sequential manner. Example (14) demonstrates the basic idea behind this mapping.

**Example 14.** In this example, the sample function $f$ defined from Example (4) will be used to construct a Toffoli network. The constants – such as the 0 line at the bottom of the network – and variables (a, b, c and d) are depicted as input lines which are used to map the function onto said lines. Some output lines are not required to define the output function, and so are labeled g as “garbage”. In Figure (2.4) the first transformation specified from the PLA definition (111→1) applied to the network is an AND operation on the first second and third variables followed by an EXOR operation with the output thus describing a $T_4$ gate. NOT operations on the first, third and fourth variables are required in order to apply a $T_5$ gate to the network according to the function definition. Input lines at the output end are discarded as garbage (labeled g) as they are not necessary in the function definition.
# Function: Sample1
.i 4
.o 1
.type esop
111- 1
0100 1
10-1 1
0010 1
0001 1
0111 1
.e

Figure 2.3: Sample function in ESOP PLA format.

Figure 2.4: Equation (2.3) represented as a Toffoli network.

2.4 Reversible Logic Synthesis

Function representation in logic synthesis is the key in the realization process of circuits. Algebra can be used to get from one representation to another, however, some formats can be more useful for particular applications
than others. Several representation formats exist but only those previously defined will be discussed in this thesis for relevance purposes.

An irreversible function can be made reversible using synthesis techniques [8]. ESOP based functions are not necessarily reversible, however, when synthesized using a Toffoli network they are made reversible due to the fact that only Toffoli gates are used and as these are reversible, the circuit consequently becomes reversible as a result [48]. In order to create a proper mapping to a Toffoli network representation, it is required to convert the SOP to an ESOP. The difference between the two formats – as mentioned in the previous section – is in the operation applied to terms in the list: one uses an OR operation while the other uses an EXOR operation. When evaluating terms in both cases, only those that are mutually exclusive will occur in an ESOP, while all terms will occur in the SOP. Using set theory terminology, the SOP can be described as a union of all terms (i.e.: \( \cup(P_1, P_2, \ldots, P_{m-2}, P_{m-1}, P_m) \) – see Equation (2.1)) in the list and the ESOP can be described as a union of all terms while eliminating the intersection of a term with the list (those that occur in pairs: \( \cup(P_1, P_2, \ldots, P_{m-2}, P_{m-1}, P_m) - \cap(P_1, P_2, \ldots, P_{m-2}, P_{m-1}, P_m) \) – see Example (15).
Example 15. Systematically, one can convert the 1-bit adder function defined in the PLA below (see Figure (2.5)) as a SOP to an ESOP:

```
# Function:
# 1-bit adder/rd32
# This file has
# been taken
# from RevLib
# (www.revlib.org)
.i 3
.o 2
.ilb a b c
.ob o r
000 00
001 01
010 01
011 10
100 01
101 10
110 10
111 11
.e
```

Figure 2.5: PLA definition of 1-bit adder function.
In Table (2.9) is a breakdown of the completely specified function into its mappings. For compactness, it is not necessary to remember which values map to 0 since those that map to 1 are known and vice versa. As expressed below, one may convert cubes into ESOP product terms in order to evaluate them. In order to do this, one method involves simply converting the SOP to a PPRM. One way to do this is by complementing pairs of terms with negative variable polarities:

\[
\begin{align*}
o &= 011 \oplus 101 \oplus 110 \oplus 111 \\
&= \bar{a}bc \oplus abc \oplus ab\bar{c} \oplus abc \\
&= \bar{a}bc \oplus ac \oplus ab\bar{c} \\
&= (\bar{a}bc \oplus 1) \oplus ac \oplus (ab\bar{c} \oplus 1) \\
&= (\bar{a}bc \oplus \bar{a}b \oplus a) \oplus ac \oplus (abc \oplus \bar{a}b \oplus \bar{a}) \\
&= (\bar{a}b \oplus \bar{a}b \oplus \bar{a}) \oplus (a\bar{b} \oplus a) \oplus ac \oplus \bar{a}bc \oplus abc \\
&= \bar{a}b \oplus ab \oplus ac \oplus \bar{a}bc \oplus abc \\
&= (\bar{a}b \oplus \bar{a}bc \oplus abc) \oplus ab \oplus ac \\
&= (\bar{abc} \oplus abc) \oplus ab \oplus ac \\
&= bc \oplus ab \oplus ac \\
&= -11 \oplus 11 \oplus 1 - 1
\end{align*}
\]

\[
r = 001 \oplus 010 \oplus 100 \oplus 111 \\
= \bar{a}bc \oplus ab\bar{c} \oplus a\bar{b}c \oplus abc
\]
\[
= (\overline{abc} \oplus 1) \oplus (\overline{abc} \oplus 1) \oplus \overline{abc} \oplus abc
\]
\[
= (\overline{abc} \oplus \overline{ab} \oplus a) \oplus (\overline{abc} \oplus \overline{ab} \oplus a) \oplus \overline{abc} \oplus abc
\]
\[
= (a \oplus a) \oplus (\overline{abc} \oplus \overline{ab}c) \oplus (\overline{abc} \oplus abc) \oplus \overline{ab} \oplus \overline{ab}
\]
\[
= b\overline{c} \oplus bc \oplus \overline{ab} \oplus \overline{a}b
\]
\[
= (\overline{b} \oplus 1) \oplus bc \oplus \overline{ab} \oplus (\overline{ab} \oplus 1)
\]
\[
= (\overline{ab} \oplus a) \oplus bc \oplus \overline{ab} \oplus (\overline{bc} \oplus b)
\]
\[
= (\overline{ab} \oplus \overline{ab}) \oplus (bc \oplus \overline{bc}) \oplus a \oplus b
\]
\[
= c \oplus a \oplus b
\]
\[
= \overline{1} \oplus 1 \oplus \overline{1} -
\]

Note that the selection of terms or order in which they are simplified is not quite important as the polarity distribution of the equation which is being changed gradually. For example, if the literal \(\overline{abc}\) was chosen instead of \(abc\) in line 2 of the equation for \(o\) the result would be the same in list of literals.
<table>
<thead>
<tr>
<th>Mappings to 0</th>
<th>Mappings to 1</th>
<th>PPRM</th>
<th>ESOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>r</td>
<td>o</td>
<td>r</td>
</tr>
<tr>
<td>000</td>
<td>011</td>
<td>011</td>
<td>001</td>
</tr>
<tr>
<td>001</td>
<td>101</td>
<td>101</td>
<td>010</td>
</tr>
<tr>
<td>010</td>
<td>110</td>
<td>110</td>
<td>100</td>
</tr>
<tr>
<td>100</td>
<td>111</td>
<td>111</td>
<td>111</td>
</tr>
</tbody>
</table>

Table 2.9: Conversion table of 1-bit adder function from SOP to ESOP.

\[ r = a \oplus b \oplus c \quad (2.5) \]

\[ o = ab \oplus ac \oplus bc \quad (2.6) \]
Chapter 3

Solution Approaches

3.1 Problem Description

The problem to be solved in this thesis is to optimize Toffoli networks specified as ESOP cube lists. The particular objective in this optimization attempt is in the reduction of NOT operations required to implement such networks. Every time a variable changes polarity a NOT operation is required which results in an increase in the overall network cost if terms in the list are not properly ordered. Two major approaches have been developed and will be explained in the coming sections. In this thesis, the algorithm concept provided should offer some critical optimizations for any ESOP specified Toffoli network and a means to determine if certain specifications are optimal or not. In this thesis, the goal is to reduce NOT operations as much as possible as these are usually not all required to realize the function. This is
actually done by minimizing the CHD of the given ESOP list. The approach attempts to modify the function representation in order to alter its polarity of the ESOP list. By representing a function in a different way, it is also occasionally possible to reduce the number of Toffoli gates required to represent it. However, this is not a common process as there is no discernible gate pattern that an algorithm could optimize – only certain template identity patterns can be identified and eliminated [28].

3.2 Changing Function Polarity

In any function, the overall function polarity is determined by individual variables in each product term or cube representing it. This polarity can be changed by using basic logic rules and identities (i.e.: Modus Tollens, Modus Ponens, distributivity, associativity, etc.) on such expressions. In particular, the focus is on a direct method for modifying polarity involving complementing terms in a way that preserves the function. Thus, function polarity is modified pairs of terms at a time until the ESOP list becomes more compact or has a more balanced polarity. Our main task consists of reducing NOT operations and the size of Toffoli gates needed. A fully worked out example will be given in Subsection (3.2.2). In general, individual variable polarity should be optimized, however this is a different dedicated problem on its own.
3.2.1 Assessing Polarity of a Cube

The work done in [8] mainly pertains to the synthesis of Toffoli networks using ESOP lists. However, this approach is not optimal and can be improved greatly by focusing on polarity indices and variable order in expressions. When assessing the overall polarity in terms of an ESOP, terms with like variable polarity can be put in sequence to avoid having to complement an input line – or variable – multiple times. In Example (11), this was demonstrated by grouping cubes with like polarities in variables and modifying pairs of cubes at a time to minimize the overall list polarity. This is done by complementing both cubes selected with similar polarity which does not change the result of the function and may simplify it further to reduce $T_1$ gates.

Example 16. In this example, a simple expression is taken in the attempt to reduce its polarity by complementing terms. The circuits are also shown below.

\[
\begin{align*}
\bar{a} \oplus \bar{b} \bar{c} \oplus c &= (\bar{a} \oplus 1) \oplus (\bar{b} \bar{c} \oplus 1) \oplus c \\
&= (a) \oplus (\bar{bc} \oplus b) \oplus c \\
&= a \oplus (\bar{bc} \oplus c) \oplus b \\
&= a \oplus bc \oplus b
\end{align*}
\]
At this point, our result is a PPRM with a reduction of three $T_1$ gates. However, this circuit can be further optimized as there are dependent terms in the list:

$$= a \oplus (bc \oplus b)$$

$$= a \oplus \overline{b} \overline{c}$$

Note that the overall reduction is of two $T_1$ gates and one $T_2$ gate through further polarization.

Figure 3.1: Function representations of $\bar{a} \oplus \bar{b} \overline{c} \oplus c$.

As one may notice in Example (16) shown above, the use of three $T_1$ gates can be avoided and thus reduce the overall quantum cost of the circuit. The degree of polarization depends on the number of iterations and speed of convergence to a constant value for average polarity – some functions are better optimized towards negative polarity for example. Ideally, the goal is to organize cube lists with individual cube polarities very close to the average CP – such representations of the functions are likely to be optimal due to the
fact that the list will be easier to group and sort thus resulting in improved costs.

3.2.2 Rewriting Function Expressions to Alter Polarity

Critical factors in altering function polarity rely on attempting or achieving canonical representation and grouping. The process of grouping refers to reordering terms with like variables consecutively in the list. The process of canonical representation on the other hand aims at reducing the number of terms used to represent the function altogether. In this process, similar terms in the specification can sometimes be avoided altogether and replaced by fewer terms. Once any process iteration is done – see Example (15) as the process is outlined – the resulting list will always be in an ESOP format. It is useful to group terms together prior to completing this conversion as it will help greatly in specifying the function. This is due to the fact that overlapping terms will be consecutive to each other and can offer better reduction patterns. Reversing the process is a little more complicated as it requires completely eliminating only pairs of duplicate terms. This implies that if the term appears an odd number of times it will still be in the SOP, otherwise it will not. It is interesting to note that if all terms are mutually exclusive in the list, it is both representable as a SOP and an ESOP by simply changing the OR operator into an EXOR as the resulting
function description will be the same. This is because of the overlap in the EXOR and OR operation definitions: the OR function and EXOR function have a common operation with the addition of an intersecting operation included in the EXOR (see the truth table definition in Table (2.2)). In [8], the EXORCISM-4 algorithm automatically converts from SOP to ESOP, however, the process described here outlines part of our algorithm developed to further optimize reversible functions.

**Example 17.** The previously shown example (Example (15)) defining the 1-bit adder function is mapped to a truth table below (see Table (3.1)). Its corresponding ESOP was generated in order to map it to a Toffoli network. Below are shown 3 figures using various methods contrasting the utility of polarity in optimization in function representation.
Table 3.1: Truth table definition of 1-bit adder function.

Table (3.1), which is also used in [8] to demonstrate the effect of polarity on ordering ESOP terms, is also used here to demonstrate how to further optimize the circuit specification. As one may notice from example (15), terms with similar polarity tend to expand with similar intersecting and mutually exclusive terms, thus canceling each other. The algorithm used will be further described in this section. In this example, the requirement of all NOT operations was eliminated successfully. In [8], the relative improvement is by 3 of these operations and a 38% lower cost in the network produced (see Figure (3.2)).
In order to change a term however, a few criteria need to be determined. First and foremost, the list has to have an overall unbalance in polarity thus meaning that there are more negative terms than positive ones or vice versa. These terms have to also exist in pairs in order to use the algebraic simplification demonstrated above in Section (2.3). Finally, only terms that will improve the overall CP balance of the list should be kept. If no measure is taken to do this, one may end up with more terms in the list than initially. A key step also involves removing terms that appear in pairs and merging terms together when they are not mutually exclusive with close enough polarity. The simplification step consists of picking two terms of like polarities and complementing them. Pairs of cubes are chosen based on the standard deviation from the CP of the cube list. When complementing a term, there will be as many new terms as there are fixed variables (i.e.: vari-
ables that are un-complemented or complemented). The steps are outlined below:

1: Take cube input $P$ with $n$ variables denoted as $p_i$

2: Set output cube list $O \leftarrow \phi$

3: Set index variable $i \leftarrow 0$

4: While $i < n$

4.1: If $p_i = 0$ or $p_i = 1$

4.1.1: Set $p_i \leftarrow \overline{p}_i$

4.1.2: Add $p$ to $O$

4.1.3: Set $p_i \leftarrow -$

4.2: Set $i \leftarrow i + 1$

5: Return $O$

Pairs of cubes are chosen based on their standard deviation in polarity from the average polarity. It is also necessary to calculate the direction of polarity optimization as mentioned before since some functions are more compactly represented when represented as negative FPRM’s instead of positive ones. If a pair of cubes are found with similar and large enough standard deviation values from the average CP, they are passed into the algorithm above using a negative or positive mask depending on whether the overall CP is negative or positive. The results are then intersected with the list and iterations of this procedure can be continued until the polarities stop con-
verging quickly enough or begin to diverge. An iteration instance is denoted below using the following algorithm:

1: Take PLA input file as list of cubes $L$

2: Set output cube list $O \leftarrow \phi$

3: For each cube $L_i \in L$:

   3.1: Evaluate statistics for $L_i$ (i.e.: CP, HD, etc.)

   3.2: If $L_i$ is not within 2 standard deviations from the mean polarity, remove $L_i$ from $L$ and add to $O$ with a pairing flag

   3.3: If $L_i$ has a pair, complement pair and add output to $O$ without pairing flags

4: Return $O$ as ESOP PLA file

Note: This algorithm is implemented using dynamic programming in order to efficiently find terms without exhausting every possible combination of cube pairs. An exhaustive version of this algorithm was not tested but would provide optimal results for function representation compactness. This algorithm will always return an ESOP formatted PLA due to the intersection function used during the complementation of terms.

Example 18. Take as an example product term $101\overline{0}1$ and complement it:
\[
101 - 01 = (101 - 00) \oplus 101 - 01
\]
\[
= 101 - 00 \oplus (101 - 1) \oplus 101 - 01
\]
\[
= 101 - 00 \oplus 101 - 1 - \oplus (100 - 0) \oplus 100 - 0
\]
\[
10 - 01 - 0 - 0
\]
\[
= 101 - 00 \oplus 101 - 1 - \oplus 100 - 0 - 0
\]
\[
(11 - 0 - 0 - 0) \oplus 1 - 0 - 0 - 0 - 0
\]
\[
= 101 - 00 \oplus 101 - 1 - \oplus 100 - 0 - 0
\]
\[
11 - 0 - 0 - 0 - 0
\]

Similarly, using product term notation:
\[
x_1\bar{x}_2x_3\bar{x}_6x_7 = (x_1\bar{x}_2x_3\bar{x}_6\bar{x}_7) \oplus x_1x_2x_3\bar{x}_6
\]
\[
= x_1\bar{x}_2x_3\bar{x}_6\bar{x}_7 \oplus (x_1\bar{x}_2x_3\bar{x}_6) \oplus \bar{x}_1x_2x_3
\]
\[
= x_1\bar{x}_2x_3\bar{x}_6\bar{x}_7 \oplus x_1\bar{x}_2x_3\bar{x}_6 \oplus (x_1\bar{x}_2\bar{x}_3) \oplus \bar{x}_1\bar{x}_2
\]
\[
= x_1\bar{x}_2x_3\bar{x}_6\bar{x}_7 \oplus x_1\bar{x}_2x_3\bar{x}_6 \oplus x_1\bar{x}_2\bar{x}_3 \oplus (x_1x_2) \oplus \bar{x}_1
\]
\[
= x_1\bar{x}_2x_3\bar{x}_6\bar{x}_7 \oplus x_1\bar{x}_2x_3\bar{x}_6 \oplus x_1\bar{x}_2\bar{x}_3 \oplus x_1x_2 \oplus \bar{x}_1
\]

**Example 19.** When \( f \) (see Equation (2.3)) is polarized note that the HD between terms is smaller (see Equation (3.1)). In Figure (3.3), by taking the complements of certain terms large Toffoli gates can be avoided in this example. Below is a demonstration of the overall polarization algorithm and its effect resulting in a 62% reduction from 13 gates to 5 in Example (15) – see Figure 50.
(2.4):

\[ f = x_1\bar{x}_2x_4 \oplus \bar{x}_1\bar{x}_2 \oplus \bar{x}_1x_2 \oplus x_2x_3 \oplus \bar{x}_1x_4 \]  

(3.1)

Figure 3.3: Polarization of sample function \( f \) from Equation (2.3).

Changing polarity in this particular example allows us to reduce the number of NOT gates even further as shown in Figure (3.3). Note that the QC has been reduced from 155 to 51. In the coming section, a combination using both of these techniques will be demonstrated as it is possible to improve over the significant reduction of network costs in this particular example and many others.
3.3 Cube List Ordering Method

The main metrics used when ordering cubes are HD and CHD as explained previously. In essence, the largest obstacle in ordering cubes is the 
don’t care 
values in variables which are essentially wild-cards in determining distance values between cubes. HD alone is insufficient to provide previous HD information from other cubes. Since a 
don’t care 
could be either a 1 or a 0, it preserves the value for that digit/variable based on the previous cube in the ordering procedure. Thus, the solution is to track the path taken using the CHD which actually does take into account these values.

A solution for cube ordering [30] with 
don’t care 
values omitted is relatively straightforward to order as the relative distances between cubes does not change dynamically when calculated. The introduction of variable literals makes it very difficult to fix costs and also requires a complex and dynamic solver for optimal results. Although the solutions provided in most of the benchmarks are an improvement over work done in [8], some are also optimal. It is difficult to show and demonstrate that a solution is in fact optimal. This is due to the fact that costs between cubes vary as they are visited, thus not allowing us to use certain related graph theorems. The algorithm allows us to check for optimality if there is a distinct path passing through all cubes and going from the first and returning to itself. Only small functions can actually be truly verified as optimal since this requires drawing
out a truth table and checking all the terms through a process which will not be discussed in this thesis.

**Example 20.** When taking function $f$ (see Equation (2.3)) and attempting to order the terms using the constraint of minimizing $HD$, one can notice an improvement in circuit costs again:

\[
f = x_1x_2x_3 \oplus x_1 \overline{x_2}x_4 \oplus \overline{x_1}x_2x_3x_4 \oplus x_1 \overline{x_2}x_3 \overline{x_4} \oplus \overline{x_1}x_2 \overline{x_3} \overline{x_4} \oplus \overline{x_1}x_2 \overline{x_3} \overline{x_4} \oplus \overline{x_1} \overline{x_2} \overline{x_3} \overline{x_4} \oplus \overline{x_1} \overline{x_2} \overline{x_3} \overline{x_4}
\]

(a) Original

(b) Ordered

Figure 3.4: Unordered versus ordered ESOP Toffoli network.

Note an improvement of 21% in the number of gates, where the QC was reduced from 155 to 151 as 4 $T_1$ gates were removed.
In Figure (3.4), the effect of cube re-ordering is demonstrated using example (4) (see Table (2.3)). In this particular example, it does not offer a better solution than polarization simply because it cannot change the way the function is represented. As mentioned earlier in Section (2.1), HD and CHD are the main ordering criteria for this method of optimization. The model however will be translated and abstracted into the Traveling Salesman Problem (TSP).

### 3.3.1 Traveling Salesman Problem Approach

The Traveling Salesman Problem (TSP) is formulated in having a network of cities and a starting point for a salesman who is required to visit all cities while minimizing the distance traveled. In a more concrete sense, the input for a TSP instance is given as a graph $G$ consisting of a set of nodes $V$ and edges $E$ with weights $w$. A solution consists of a set of cities in the order they must be traveled by the salesman. An optimal solution holds the additional constraint that this path or order in which cities are visited has to yield the minimum distance traveled as is possible. It forms the rigid backbone of this thesis and is deemed a necessary asset to solve the cube ordering problem since no other mapping to related problems was found so far. Essentially an ancient problem formulated by many mathematicians under different names [18], it is used to reduce more complex problems relevant to many graph algorithms, and in graph theory. Even though it does not have a generalized solution and is a well-known NP-complete problem
some solutions for fixed graph type configurations have been found. For specific known large city networks, there are fixed optimal solutions found however, these use the enumeration of all possible paths between cities including those that may be irrelevant to the solution. There is no discernible pattern in finding unnecessary paths in solutions as there is no predictable means to avoid calculating unnecessary paths: it is sometimes obligatory to exhaust all possible paths to prove optimality. Thus, only functions with small numbers of input variables can be proven optimal at this point in time and any proofs have to be done on a case by case basis. One method of proof involves using Gray code to generate a solution and compare the ordering found to it. As all the Hamming distances between product terms in the solution are 1 and the order is optimal by definition one can verify whether a solution is optimal or not. The drawback with this method is in the fact that there is a need to generate an exponential number of nodes relative to input size and this is unfeasible for larger input sizes.

An instance of the ESOP cube list can be translated into an instance of the TSP (more specifically into the GTSP which uses clusters of cities instead of single cities and will be defined in the coming subsection) by creating a city for each cube and a cost matrix for the costs between cubes. As there are don’t cares this will be done as a pre-processing element of the algorithm when assigning groups and choosing a starting node – one that has the least HD to the all-positive starting node and belongs to the largest number
of groups. When posed as a TSP, one can group cubes together based on whether they intersect in order to avoid visiting all of them thus eliminating redundant cubes and edge cost calculations that are not necessary in the list. In order to formulate a proper instance of the problem, an explanation will be provided on how to reduce the ordering problem into an instance of the Generalized Traveling Salesman Problem (GTSP).

### 3.3.2 The Generalized Traveling Salesman Problem

The criteria used in ordering product terms or cubes is known as the Hamming distance (see Definition (13)). Example (20) demonstrates the advantages of ordering according to this criteria. In [8], a standard ESOP cube list is sorted according to a divide-and-conquer heuristic algorithm. It has been demonstrated that this sorting algorithm for cubes is not optimal (see Example (17)) and also propose a better alternative to ordering cube lists.

**Definition 28.** The **triangle inequality** is defined as a property of a weighted graph that guarantees that there is no intermediary shortcut possible between any 2 nodes by using a third node. In other words, this states that the edge weights are consistent in the fact that no edge weight is greater than the sum of two others in a cycle involving 3 edges connecting any 3 nodes in the graph. The
constraint is as follows:

\[
\text{cost}(e_{xy}) + \text{cost}(e_{yz}) \geq \text{cost}(e_{xz})
\] (3.2)
ear variables. There is a one-to-one mapping and a certificate guaranteeing
that a solution to the Integer Linear Program will map to a GTSP solution
through TSP [25, 38] and it is deemed the most effective and exact way of
solving the latter [15]. This is due to shortcomings in using regular LP which
will be discussed further in the coming sections. ILP however does not of-
er a solution in polynomial time. Although all cubes are connected, many
edges are unnecessary in calculations as they are irrelevant to the solution
(see example in Figure (3.7(a))). This approach uses the original solution for
the simpler theoretical outlook on the problem using the Generalized Trav-
eling Salesman Problem (GTSP) previously mentioned in the background
research section. The attempt to minimize the overall distance between con-
secutive cubes is done in an ILP which is used to approximate a solution
to the mapped TSP instance. Due to the fact that the distance definition
(see Definition (13)) satisfies the triangle inequality (see Equation (3.2))
when applied to a graph, the required mapping to a TSP is possible in a
given undirected weighted graph $G = (V, E, w)$ with $V$ the set of nodes,
$E$ the set of edges and $w$ the set of edge weights – see Figure (3.5). This
distance corresponds to the total Hamming distance between each variable
in the cube relative to the previous one. A GTSP instance can be converted
to a TSP instance easily by inserting a key node in each group – this concept
will be explained later – which is used to control access to other nodes in
the group. The solution then can be calculated by optimizing group con-
nections to each other, then within the individual groups. A complication
arises however if a variable in a cube has a *don’t care* value: the value of the immediate predecessor (if the value is not a *don’t care* as well) is taken by the current variable for dynamic distance calculations. This extra complication creates a cumulative issue in cube reordering as it will be necessary to recalculate the Hamming distance of each cube after each iteration of the proposed algorithm if the value is a *don’t care*. This in fact enforces dynamic recalculation of Cumulative Hamming Distance (CHD) every time a *don’t care* is encountered.

**Example 21.** *In this example, Equation (2.3) is used to demonstrate how the ordering process is done when mapping the cube list to a GTSP instance. The solved cycle can then be mapped back to an ESOP list order that is better than the original order.*

![GTSP instance](image1)

(a) GTSP instance.  

![Solved GTSP instance](image2)

(b) Solved GTSP instance.

Figure 3.6: GTSP mapping of an ESOP list
3.3.3 The Traveling Salesman Problem as an Integer Linear Program

An equally fundamental part of this thesis is in Linear Programming (LP) as it is the quickest known method for solving the TSP [5]. A linear program instance consists of a set of constraints and an optimization direction which are used to find a solution – preferably an optimal one if any. Using the simplex method, these are transformed into a linear algebra problem using matrices and linear multivariable equations with pivot variables in order to systematically determine a solution. The use of pivot rows representing variables in a matrix allows for the recalculation of linearly independent variables and iterates towards a feasible solution (see Example (22)). As this problem requires an integer solution specifically since it is not feasible to think of node fractions in the solution, an integer constraint sophisticates the LP aspect as the feasible region becomes a discrete set of points instead of a contiguous area. Although solution optimality is not necessary, it can be demonstrated that in general this approach is by far better than the proposed greedy algorithm in [8].

Example 22. The following problem is given to demonstrate the functionality of an Linear Program (LP) solver using the Simplex methodology. Given 3 numbers \(x_1, x_2\) and \(x_3\), the objective is to
maximize their sum \(5x_1 + 2x_2 + x_3\) such that:

\[
\begin{align*}
  x_1 + 3x_2 - x_3 & \leq 6 \quad (3.3) \\
  x_2 + x_3 & \leq 4 \quad (3.4) \\
  3x_1 + x_2 & \leq 7 \quad (3.5)
\end{align*}
\]

The sum \(5x_1 + 2x_2 + x_3\) is called the objective function. The constraints above will be used to construct a Simplex tableau (see Table (3.2)). This is done by taking the coefficients of each variable in each equation and adding slack variables to absorb the constants so that the inequality constants are all greater or equal to 0.

\[
A = \begin{pmatrix}
1 & 3 & -1 \\
0 & 1 & 1 \\
3 & 1 & 0
\end{pmatrix}
\]

The vector \(c = (6, 4, 7)^T\) contains the right hand side constants. Thus, our new inequalities are:

\[
\begin{align*}
  x_1 + 3x_2 - x_3 - 6 & \geq 0 \quad (3.6) \\
  x_2 + x_3 - 4 & \geq 0 \quad (3.7) \\
  3x_1 + x_2 - 7 & \geq 0 \quad (3.8)
\end{align*}
\]

Let \(b = (5, 2, 1)^T\) represent the objective function determinants.
The solution vector \( x \) can be described as \( x = (x_1, x_2, x_3)^T \). Thus, the resulting inequality region is described as \( xA \geq c \) which can be restated as \( xA - c \geq 0 \). Denote the solution vector \( y = xA - c \) as an equality since the logical lower bound of the problem is the equality region. The tableau can be drawn as follows:

<table>
<thead>
<tr>
<th></th>
<th>( x_1 )</th>
<th>( x_2 )</th>
<th>( x_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y_1 )</td>
<td>1</td>
<td>3</td>
<td>-1</td>
</tr>
<tr>
<td>( y_2 )</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( y_3 )</td>
<td>3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>-5</td>
<td>-2</td>
<td>-1</td>
</tr>
</tbody>
</table>

Table 3.2: Sample Simplex tableau.

By pivoting the highlighted rows through algebraic interchanging of \( x_2 \) with \( y_3 \) and \( x_3 \) with \( y_1 \), the tableau can be redrawn as follows:

<table>
<thead>
<tr>
<th>( y_3 )</th>
<th>( x_2 )</th>
<th>( y_2 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( y_1 )</td>
<td></td>
<td>23/3</td>
</tr>
<tr>
<td>( x_3 )</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>( x_1 )</td>
<td></td>
<td>7/3</td>
</tr>
<tr>
<td>1</td>
<td>5/3</td>
<td>2/3</td>
</tr>
</tbody>
</table>

Table 3.3: Solved sample Simplex tableau.
The resulting table has no negative values in the last column and row, thus providing a solution \(x_1 = 7/3, \ x_2 = 0\) and \(x_3 = 4\). Pivoting is systematically decided based on negativity.

**Example 23.** In this example, a graph is constructed from an ESOP list mapping to ones \{1100, 1101, 1111, -111, 0000, 1110, 11--, 0111\}. The procedure to find the shortest path to connect all groups of nodes in the graph will also be demonstrated. In this current order, the CHD (including the all-positive node ‘1111’) is 11. Each node is a group of its own, but in a preprocessing portion of the algorithm such groups are merged to avoid unnecessary edge calculations.

![Figure 3.7: Unordered versus ordered ESOP Toffoli network](image)

(a) GTSP instance  
(b) Solved GTSP instance

When solved, a CHD of 8 is found with list order:
The complication in the task of ordering arises due to the fact that
\textit{don’t care} digits in variables need to be fixed in order to determine the next
in sequence for ordering. In particular, the ESOP cube is defined as an
exclusive-or sum of products which are mutually exclusive – thus making
it easier to represent the function in several ways – and so polarizing these
expressions can be the key in minimizing the number of gates necessary to
implement the function as a circuit. The algorithm changes dramatically in
this approach of solving the problem as it involves grouping cubes before
ordering them. The solver is then relieved from calculating permutations
of all possible paths between cubes in the graph of the GTSP generated
instance.

3.3.4 ESOP Cube Ordering Represented as an Integer
Linear Program

This is deemed the ideal method in this thesis for sorting cubes as a reduc-
tion of the problem maps directly onto the Generalized Traveling Salesman
Problem (GTSP) instance. A pre-processing algorithm is required to organize
the ESOP cube list provided as input in order to generate the ILP (Integer
Linear Program) instance directly. In this particular application, the prob-
lem design involves defining clusters of cities from which the salesman needs
only to visit one of each while maintaining a minimum distance traveled in
order to complete the cycle. In the original TSP, the salesman is required to visit all the cities while maintaining a minimum distance travelled. An optimization rule using the triangle inequality which states that the shortest path must always be taken in any group of three cubes in the graph (see Figure (3.5) and Equation (3.2)) guarantees near-optimality in our solution as it can be proven that it is within a factor of an optimal solution for the problem [24]. Along with an additional constraint, a slight modification of the problem is necessary in order to create a one-to-one mapping of the cube list ordering problem to the GTSP. In order to properly understand the mapping, for each problem, a node is a cube and nodes are connected by edges with weights corresponding to the Hamming distances between cubes. Another addition involves labeling groups by a key node that is contained by all the other cubes in the group (see Figure (3.7(a))). This key node is defined based on its intersection with other cubes in the group: a key node must intersect each other cube in the group by at least one term (see examples below).

\[
11 \quad \text{and} \quad 10011 = \emptyset
\]
\[
11 \quad \text{and} \quad 1100- = 1100-
\]
\[
11 \quad \text{and} \quad -1 - 00 = 11 - 00
\]

Cubes with many intersections automatically become key nodes. A key node acts in a way as a gateway to other non-key nodes for the solver and helps
in finding an optimal path by avoiding unnecessary edges with other key nodes. These cubes are determined during the preprocessing portion of the algorithm as follows:

1: Take 2 cubes input as $P$ and $Q$

2: Set output cube $o \leftarrow \phi^n$

3: For $i \leftarrow 0$ increment by 1 loop condition $i < n$

   4.1: If $p_i = -$ 

       4.1.1: Set $o_i \leftarrow q_i$

   4.2: Else if $q_i = -$ 

       4.2.1: Set $o_i \leftarrow p_i$

   4.3: Else if $p_i \neq q_i$

       4.3.1: Return $\phi^n$

   4.4: Else

       4.4.1: Set $o_i \leftarrow p_i$

4: Return $o$

A linear program solver is then used to calculate the shortest Hamiltonian path satisfying the group constraints for the solution (see example function $misex$ in Appendix (A)). Therefore, in order to solve an instance of the problem, three major steps are required: the first one involves creating the groups and calculating the cost matrix, the second part generates an integer linear program with the calculated constraints and solves it, then finally parses the
solution and orders the cubes back into a PLA format. The resulting path found from the graph, or GTSP solution determines the order in which the cubes must be ordered. This approach represented as an integer linear program is capable of finding a solution to a problem in polynomial time [6] even though it may not be optimal. The drawback with this method is there is no guarantee the approximation will offer an integer solution. Also, generating the constraints can be done in polynomial time. Although an optimal solution cannot be guaranteed, the solution offers improvements to alternative synthesis algorithms [8] (see Tables (4.1-4.4)).

Given a function based on binary cubes – the base set only contains \{0,1\} and does not include don’t cares – a TSP approach would be optimal and sufficient as a GTSP instance would not be required. This is due to the fact that the distances do not vary when travelling from one cube to another. However, other approaches [25, 31, 5] may prove to be more resource efficient as the TSP and ILP (Integer Linear Programming) are proven NP-hard problem. A graph is generated where nodes represent cubes and edges with weights represent the total Hamming distances between connected expressions. This is a possible modification of the TSP approach including groups which enclose all cubes generated from all particular don’t care variables. Essentially, there is no need to generate all possible nodes of the cube with the largest number of digits since the expressions can be dynamically generated as they are traversed and some cubes are also completely ignored.
Using a linear program based on the GTSP, the proposed method will solve for groups/clusters of nodes instead of individual ones and will then be tested for efficiency with a large number of inputs. The inputs are given through a PLA file that defines the cube list which then needs to be processed by an independent algorithm from the LP solver. This algorithm is roughly outlined below:

1: Process PLA input and group cubes into nodes and groups.
2: Translate the group information into a GTSP instance.
3: Attempt to solve the Integer Linear Program (ILP) and gather results for further processing while constraining resources.
4: Determine if the solution is optimal and feasible, then parse it and order cubes.

A solution is unfeasible if it contains more than one cycle. The reason behind this is that they disrupt the ordering of the whole list by splitting the list into segments. This may only occur in the efficient version of the algorithm given below as it uses an approximative method. When multiple cycles occur, the algorithm breaks all of them at the most expensive edge and reruns with the additional constraints of the removed edges. An alternative to solving the TSP would typically involve the exponential enumeration of required cube traversals. Obviously, this approach is only useful for cube lists with less than approximately 20 don’t cares in cubes as this may involve analyzing
roughly more than $2^n$ permutations (which would be pushing the limits of an array data structure with over 1 million indices), where $n$ represents the largest number of don’t care digits in the cubes visited. In fact, this is done for small functions to check for optimality and is the simplest way known to verify that a solution is optimal algorithmically. This approach does not impose a specific data structure and could be improved drastically as not all permutations of don’t cares are relevant to finding a solution. Don’t care values fork calculations exponentially and could potentially be fixed for certain instances, thus eliminating spent time and resources where they are not needed. Although it does guarantee an optimal solution every time, the resources used make it an unappealing method.

The approach in this thesis is based on the TSP, thus involving the goal of minimizing travel distance as much possible when travelling to a set of points or regions. The original context of the problem refers to cities that a salesman has to visit. This is relevant to the original problem in the scope that Hamming distances and cubes properly represent the layout of the problem to be solved. Two algorithms have been devised to solve this problem: the first attempts an exact TSP mapping of the problem from the ILP perspective [35] and thus an optimal solution, while the second attempts to find a feasible solution that is non-optimal and resource efficient. Both algorithms are given below:
Optimal Version:

1: Load PLA file.

2: Create groups and exhaust *don’t cares* from expressions:

2.1: Create a group for each expression.

2.2: If an expression contains *don’t cares*, exhaust elements and add them to the original expressions group. An element may be part of more than one group.

3: Write LP file:

3.1: Calculate edge costs between groups and store into an array as an adjacency matrix. This is referred to as the *cost matrix*.

3.2: Force a cycle by constraining a path from groups to have at least 2 edges (one in, one out).

3.3: Write GTSP problem reduction constraints:

3.3.1: Create single cycle traversal constraint.

3.3.2: Create group edge constraints.

3.3.3: Create node loop constraints.

3.3.4: Create optimization constraint.

3.4: Create minimization and optimization constraints for minimal cost.
3.5: Run GLPSol and output solution into file.

4: Analyze LP solution and check order and consistency:

4.1: If an integer relaxation of the problem is not possible, then rerun using the Efficient Version.

4.2: Otherwise the optimal solution was found and the PLA file is rewritten in the order found starting with the node in the cycle that is nearest to the all-positive node and belongs to the largest number of groups.
Efficient Version:

1: Load PLA file.

2: Create groups and fix don’t care distances in expressions:
   2.1: Create a group for each expression.
   2.2: If an expression contains don’t cares, flag the group. An element may be part of more than one group.

3: Write LP file:
   3.1: Calculate edge costs between groups and store into an array as an adjacency matrix, fixing don’t cares in flagged groups as calculations progress. This is referred to as the cost matrix.
   3.2: Force cycles by constraining a path from groups to have at least 2 edges (one in, one out).
   3.3: Write GTSP problem reduction constraints:
      3.3.1: Create loop constraints.
      3.3.2: Create optimization constraint.
   3.4: Create minimization and optimization constraints for minimal cost.
   3.5: Run GLPSol and output solution into file.

4: Analyze LP solution and check order and consistency:

72
4.1: If a solution was found, but multiple cycles exist, break each individual cycle at the most expensive edge and reconnect them with other groups in the same way until there is only a single cycle.

4.2: If the solver runs out of resources, or the solution is non-integer then there is no feasible solution and the program ends.

4.3: Otherwise an approximate solution was found.

4.4: A PLA file is rewritten in the order found starting with the node in the cycle that is nearest to the all-positive node and belongs to the largest number of groups.

The main difference between the algorithms is the optimal version of the algorithm generates nodes from terms that contain *don't cares* and expands groups as much as possible in order to avoid approximative calculations in distances. This algorithm is exact but uses an exponential amount of resources. The efficient version of the algorithm however does not avoid such approximations and simply searches for a solution satisfying the given constraints. This greatly reduces the load on the ILP portion of the solver as it does not involve massive recalculations on key nodes and does not reroute a path in the case of an edge cost change when coming from a non-key node. Although the triangle inequality holds, the cost approximations may invalidate this inequality and the LP also cannot guarantee this consistency.
Having an optimal result however does consequently result in an optimal path for the solution.
Chapter 4

Solution Application and Testing

The overall improvements will be demonstrated by applying the overall algorithm to sample PLA files. Most of these are also used in [8] and the algorithm proposed has achieved better or similar configurations. These benchmarks were acquired by running the solver on a combination of two networked machines with a power grid configured to optimize task parallelism. One being an Intel-based iMac running Mac OS X and the other a Lenovo running Linux CentOS. The algorithm was developed in GLPK [11] and Java [47] and uses GraphViz [7] to display graphs generated by the algorithm. The resulting PLA files are first converted to circuit realizations using an external algorithm, then visualized with Quiver [50] in order to display the final Toffoli networks. Costs are then calculated and confirmed within each stage.
(preprocessing, ILP and ILP rerun) of the algorithm. All original results were optimized and converted to ESOP PLA format using EXORCISM-4 [31] – all improvements in the benchmarks are relative to their algorithm results.

![Figure 4.1: Polarized ESOP Toffoli network](image)

As in example (19) (see Figure (4.1)), sometimes only one or no optimization technique is needed. Some functions may already be optimal [50] but not yet proven so. In this example, further ordering the polarized list is useless as there is no means to avoid the variable polarity changes; thus requirement of the 6 NOT gates used - the order is already optimal. A list of all the PLA files tested with the algorithm and the methods used to optimize them is given in Tables (4.1-4.4). If the solver found an optimal order, it will be marked in the optimal column.

### 4.1 Benchmark Testing

Results as one can see below are an improvement from the original algorithm in [8]. Some networks are unchanged and this is likely because it is not very plausible to easily find a solution to optimize them any further. Also,
most are non-optimal simply because there is no way to certainly prove that such functions are represented optimally. As mentioned in previous sections, it is only possible to prove optimality for functions with small numbers of input variables due to the exponential amount of testing required to achieve this. The time limit for the polarization technique was set to 5 minutes, although leaving it to run longer – and this has been attempted with some cases – may actually find better results occasionally. In particular, one can notice major improvements using the polarization technique specifically. This is due to the factor that this technique does in fact not only reduce NOT gates but whole terms in the cube lists in exceptional cases. This explains the improvements seen with misex3 from Table (4.1) and b12, bw and misex2 from Table (4.3). The sorting algorithm on the other hand appears to offer little improvement, however, if one looks closely at apex3 from Table (4.2) the critical improvement is in the CHD which is the actual optimization criteria of the algorithm. These tables all demonstrate critical statistics of the algorithm performance on various selected logic function PLA definitions after using the [8] algorithm. The overall analysis of this benchmark test demonstrates that there is still room for improvement upon existing state-of-the-art algorithms in logic synthesis. CHD, CHD’, QC, QC’ and % are abbreviations for initial Cumulative Hamming Distance, final Cumulative Hamming Distance, initial Quantum Cost, final Quantum Cost and cost improvement (in percent) respectively.
<table>
<thead>
<tr>
<th>PLA</th>
<th>Method</th>
<th>Time (s)</th>
<th>CHD</th>
<th>CHD'</th>
<th>QC</th>
<th>QC'</th>
<th>Optimal</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>cordic</td>
<td>Polar</td>
<td>0.409</td>
<td>3067</td>
<td>3022</td>
<td>197014</td>
<td>195600</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>cps</td>
<td>Polar</td>
<td>0.589</td>
<td>9666</td>
<td>3525</td>
<td>465444</td>
<td>437244</td>
<td>No</td>
<td>6</td>
</tr>
<tr>
<td>misex3</td>
<td>Polar</td>
<td>0.684</td>
<td>2906</td>
<td>2867</td>
<td>178820</td>
<td>161408</td>
<td>No</td>
<td>10</td>
</tr>
<tr>
<td>rf_func</td>
<td>Polar</td>
<td>0.010</td>
<td>12</td>
<td>3</td>
<td>143</td>
<td>51</td>
<td>Yes</td>
<td>64</td>
</tr>
</tbody>
</table>

Table 4.1: Benchmark results: polarization algorithm

<table>
<thead>
<tr>
<th>PLA</th>
<th>Method</th>
<th>Time (s)</th>
<th>CHD</th>
<th>CHD'</th>
<th>QC</th>
<th>QC'</th>
<th>Optimal</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>apex1</td>
<td>Sort</td>
<td>71.039</td>
<td>1873</td>
<td>313</td>
<td>27832</td>
<td>27560</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>apex3</td>
<td>Sort</td>
<td>100.306</td>
<td>2354</td>
<td>684</td>
<td>31028</td>
<td>30798</td>
<td>No</td>
<td>2</td>
</tr>
<tr>
<td>duke2</td>
<td>Sort</td>
<td>0.746</td>
<td>178</td>
<td>121</td>
<td>9304</td>
<td>9249</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>sao2</td>
<td>Sort</td>
<td>60.368</td>
<td>118</td>
<td>83</td>
<td>4169</td>
<td>4159</td>
<td>No</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.2: Benchmark results: sorting algorithm
<table>
<thead>
<tr>
<th>PLA</th>
<th>Method</th>
<th>Time (s)</th>
<th>CHD</th>
<th>CHD'</th>
<th>QC</th>
<th>QC'</th>
<th>Optimal</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>5xp1</td>
<td>Both</td>
<td>0.384</td>
<td>136</td>
<td>73</td>
<td>2058</td>
<td>1975</td>
<td>No</td>
<td>4</td>
</tr>
<tr>
<td>b12</td>
<td>Both</td>
<td>0.204</td>
<td>320</td>
<td>36</td>
<td>13143</td>
<td>8712</td>
<td>Yes</td>
<td>34</td>
</tr>
<tr>
<td>bw</td>
<td>Both</td>
<td>2.322</td>
<td>154</td>
<td>66</td>
<td>2774</td>
<td>1156</td>
<td>No</td>
<td>58</td>
</tr>
<tr>
<td>con1</td>
<td>Both</td>
<td>0.066</td>
<td>8</td>
<td>2</td>
<td>99</td>
<td>89</td>
<td>No</td>
<td>10</td>
</tr>
<tr>
<td>inc</td>
<td>Both</td>
<td>0.425</td>
<td>104</td>
<td>76</td>
<td>2361</td>
<td>1981</td>
<td>No</td>
<td>16</td>
</tr>
<tr>
<td>misex1</td>
<td>Both</td>
<td>0.029</td>
<td>61</td>
<td>21</td>
<td>817</td>
<td>471</td>
<td>No</td>
<td>42</td>
</tr>
<tr>
<td>misex2</td>
<td>Both</td>
<td>0.046</td>
<td>76</td>
<td>27</td>
<td>1704</td>
<td>1684</td>
<td>No</td>
<td>1</td>
</tr>
<tr>
<td>rd53</td>
<td>Both</td>
<td>0.272</td>
<td>64</td>
<td>32</td>
<td>1094</td>
<td>958</td>
<td>No</td>
<td>12</td>
</tr>
<tr>
<td>rd73</td>
<td>Both</td>
<td>1.373</td>
<td>326</td>
<td>281</td>
<td>7320</td>
<td>7143</td>
<td>No</td>
<td>2</td>
</tr>
<tr>
<td>sqrt8</td>
<td>Both</td>
<td>0.137</td>
<td>52</td>
<td>21</td>
<td>1079</td>
<td>857</td>
<td>No</td>
<td>21</td>
</tr>
<tr>
<td>squar5</td>
<td>Both</td>
<td>0.640</td>
<td>147</td>
<td>83</td>
<td>1914</td>
<td>1491</td>
<td>No</td>
<td>22</td>
</tr>
<tr>
<td>table5</td>
<td>Both</td>
<td>240.588</td>
<td>1315</td>
<td>494</td>
<td>25357</td>
<td>23118</td>
<td>No</td>
<td>13</td>
</tr>
<tr>
<td>vg2</td>
<td>Both</td>
<td>2.265</td>
<td>261</td>
<td>259</td>
<td>18508</td>
<td>17922</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>xor5</td>
<td>Both</td>
<td>0.092</td>
<td>36</td>
<td>22</td>
<td>644</td>
<td>574</td>
<td>No</td>
<td>11</td>
</tr>
</tbody>
</table>

Table 4.3: Benchmark results: both algorithms
<table>
<thead>
<tr>
<th>PLA</th>
<th>Method</th>
<th>Time (s)</th>
<th>CHD</th>
<th>CHD'</th>
<th>QC</th>
<th>QC'</th>
<th>Optimal</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>9sym</td>
<td>None</td>
<td>60.337</td>
<td>122</td>
<td>122</td>
<td>3563</td>
<td>3563</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>alu4</td>
<td>None</td>
<td>1.800</td>
<td>1196</td>
<td>1196</td>
<td>73051</td>
<td>73051</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>apex2</td>
<td>None</td>
<td>300.000</td>
<td>2089</td>
<td>2089</td>
<td>152372</td>
<td>152372</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>apex4</td>
<td>None</td>
<td>300.000</td>
<td>4190</td>
<td>4190</td>
<td>52157</td>
<td>52157</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>apex5</td>
<td>None</td>
<td>60.988</td>
<td>2254</td>
<td>2254</td>
<td>60523</td>
<td>60523</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>clip</td>
<td>None</td>
<td>1.868</td>
<td>164</td>
<td>164</td>
<td>3671</td>
<td>3671</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>e64</td>
<td>None</td>
<td>60.241</td>
<td>241</td>
<td>241</td>
<td>24498</td>
<td>24498</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>ex1010</td>
<td>None</td>
<td>300.000</td>
<td>2480</td>
<td>2480</td>
<td>64845</td>
<td>64845</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>ex4p</td>
<td>None</td>
<td>60.272</td>
<td>450</td>
<td>450</td>
<td>39576</td>
<td>39576</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>ex5p</td>
<td>None</td>
<td>300.000</td>
<td>1168</td>
<td>1168</td>
<td>67117622</td>
<td>67117622</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>misex3c</td>
<td>None</td>
<td>300.000</td>
<td>603</td>
<td>603</td>
<td>57871</td>
<td>57871</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>o64</td>
<td>None</td>
<td>0.162</td>
<td>0</td>
<td>0</td>
<td>325</td>
<td>325</td>
<td>Yes</td>
<td>0</td>
</tr>
<tr>
<td>pdc</td>
<td>None</td>
<td>300.000</td>
<td>11648</td>
<td>11648</td>
<td>37079</td>
<td>37079</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>rd84</td>
<td>None</td>
<td>67.143</td>
<td>845</td>
<td>845</td>
<td>21314</td>
<td>21314</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>seq</td>
<td>None</td>
<td>300.000</td>
<td>2507</td>
<td>2507</td>
<td>185082</td>
<td>185082</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>spla</td>
<td>None</td>
<td>300.000</td>
<td>12144</td>
<td>12144</td>
<td>517947</td>
<td>517947</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>t481</td>
<td>None</td>
<td>300.000</td>
<td>382</td>
<td>382</td>
<td>47145</td>
<td>47145</td>
<td>No</td>
<td>0</td>
</tr>
<tr>
<td>table3</td>
<td>None</td>
<td>300.000</td>
<td>907</td>
<td>907</td>
<td>26613</td>
<td>26613</td>
<td>No</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 4.4: Benchmark results: no improvement found
Chapter 5

Conclusion & Future Work

Previous synthesis approaches look at reversible gate systems involving only Toffoli and NOT gates and consist of analyzing the polarity of cubes in order to weight and order them. These approaches have been improved upon in this thesis by changing the cost metric definition which did not properly account for overall cube order as well as don’t cares. Although the previous synthesis approaches in general are not time consuming and heuristic algorithms perform quickly, the solutions they offer often overlook the ordering and likely will only roughly sort the list. Cubes may be interdependent in the lists and thus cubes weighted in different ways should have been ignored as they do not benefit the overall ordering process. In fact, they may prove to create a disorder in the list by considering terms that otherwise would not be necessary to sort. In this thesis, it has been demonstrated that by sorting the cube list of the function properly, one can avoid the need of certain NOT
gates. Also, the idea of a polarity weighting heuristic can be ineffective at cascading terms simply because the literals are not sorted according to the actual distances between terms. Furthermore, the new technique of polarization has been introduced which further optimizes terms as a preprocessing technique to avoid representations that require more terms than are needed to represent a function. In this thesis, the goal was to create the best algorithm possible to sort and polarize cubes in order to optimize circuit design for ESOP lists defined in PLA files.

This thesis offers a stepping stone for further investigation into improving logic synthesis techniques and methods. In particular, this has opened a path into looking at related problems such as gate and variable reordering in a gate network. The polarization technique also seems to promise further improvements when applied as a metric to assess optimality in function representation. There may be better ways to implement this technique that still need to be discovered. For example, take the 5 minute limit imposed on the runtime of the polarization algorithm. Improving the data structures used in the algorithm could also greatly enhance the solution technique. These need to be further investigated in order to determine an optimization pattern in this type of solving methodology.
Bibliography


[30] V. Melkonian. LP-based solution methods for the asymmetric TSP. Department of Mathematics, Ohio University, Athens, Ohio 45701, USA.


Appendices
Appendix A

Example solution for function $misex1$

The linear program code (see Figure (A.1)) generated by the efficient version of our algorithm as an optimal integer solution cannot be found with the optimal version.

The solver resolved the edges between the 5 main groups and realized 2 of the groups to be redundant to the solution as they were internally connected (see Figure (A.2)). The full list of connections between all nodes can be found in the adjacency matrix shown in Figure (A.3). The improvement was registered from an 817 to a 471 quantum cost as recorded in the benchmark Table (4.3). The ordering edge cost between groups was recorded to be 2 and the final Hamming cost was reduced from 76 to 21. This is an
improvement relative to the EXORCISM-4 generated solution in [8].
set n := 1..18;
param (n,n):  
for (i in n) {print "n", i;
}
solve;
for (i in n) {print "\nX", i;
}
solve
\n\n\n\n\nsolve;
for (i in n) {print "a", i;
}
\n\n\n\n
\n\n
Figure A.1: GLPK solver code.

94
Figure A.2: Group nodes – 5 main, 2 subgroups.

Figure A.3: Adjacency matrix of all nodes
Vita

Full name: Zakaria Hamza

University: University of New Brunswick (2003-2009)

Degree: Bachelors of Computer Science

Publications: