Synthesis of Irreversible Incompletely Specified Multi-Output Functions to Reversible EOSOPS Circuits with PSE Gates

Robert Adrian Fiszer
Portland State University
Synthesis of Irreversible Incompletely Specified Multi-Output Functions to Reversible EOSOPS Circuits with PSE Gates

by

Robert Adrian Fiszer

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science
in
Electrical and Computer Engineering

Thesis Committee:
Marek Perkowski, Chair
Małgorzata Chrzanowska-Jeske
Donald Duncan

Portland State University
2014
Abstract

As quantum computers edge closer to viability, it becomes necessary to create logic synthesis and minimization algorithms that take into account the particular aspects of quantum computers that differentiate them from classical computers. Since quantum computers can be functionally described as reversible computers with superposition and entanglement, both advances in reversible synthesis and increased utilization of superposition and entanglement in quantum algorithms will increase the power of quantum computing.

One necessary component of any practical quantum computer is the computation of irreversible functions. However, very little work has been done on algorithms that synthesize and minimize irreversible functions into a reversible form. In this thesis, we present and implement a pair of algorithms that extend the best published solution to these problems by taking advantage of Product-Sum EXOR (PSE) gates, the reversible generalization of inhibition gates, which we have introduced in previous work [1,2].

We show that these gates, combined with our novel synthesis algorithms, result in much lower quantum costs over a wide variety of functions as compared to our competitors, especially on incompletely specified functions. Furthermore, this solution has applications for multi-valued and multi-output functions.
Dedication

To the unmoved mover and uncaused cause.
Acknowledgments

I’d like to thank the following people: my friends and family for supporting me during my academic endeavors, Marek Perkowski for introducing me to the amazing world of quantum computing, the members of my thesis committee, Marek Perkowski, Malgorzata Chrzanowska-Jeske, and Donald Duncan for their immeasurable patience with me as I wrote this thesis, Addy Gronquist for making a cost calculator which saved me countless hours of computation, and finally Alan Mishchenko, for making EXORCISM v4, without which the ideas in this thesis could probably never have been realized.
# Table of Contents

Abstract .................................................................................................................. i
Dedication .............................................................................................................. ii
Acknowledgments ................................................................................................. iii
List of Tables .......................................................................................................... vi
List of Figures ........................................................................................................ vii

1. Introduction ........................................................................................................ 1

2. Background ......................................................................................................... 11
   2.1 Overview ......................................................................................................... 11
   2.2 Boolean Logic ................................................................................................ 11
      2.2.1 AND-OR Logic ...................................................................................... 11
      2.2.2 AND-EXOR Logic ............................................................................... 13
   2.3 Boolean Ring .................................................................................................. 15

3. Current Synthesis Methods .................................................................................. 17
   3.1 EXORCISM .................................................................................................. 17
   3.2 Non-Competing Methods ............................................................................ 20
      3.2.1 MMD ................................................................................................... 20
      3.2.2 DCARL ............................................................................................... 21
      3.2.3 Other MMD-like Algorithms ............................................................. 21
      3.2.4 Cycle Based Algorithms ................................................................... 21

4. PSE Gates and PSEycic ...................................................................................... 23
   4.1 PSE Gate ....................................................................................................... 23
   4.2 Synthesis with PSE gates.............................................................................. 25
      4.2.1 Synthesis with PSE gates by hand. .................................................... 25
      4.2.2 Synthesis with the PSEycic Tool Suite ................................................. 28
         4.2.2.1 PSEycic Postprocessor ................................................................. 28
         4.2.2.2 PSEycic Pre-processor ................................................................. 29

5. The Muller Transform ......................................................................................... 33

6. Alhagi Method .................................................................................................... 36
   6.2 Improvements ............................................................................................... 38
      6.2.1 Delta Management ............................................................................... 38
      6.2.2 Folded K-Maps ................................................................................... 39
   6.2.3 Improvements with PSE synthesis ......................................................... 41
   6.3 Tree Search ................................................................................................... 42
      6.3.1 Variant One .......................................................................................... 43
      6.3.2 Variant Two ......................................................................................... 43

7. Numerical Results ............................................................................................... 45

8. Multivalued Circuits ........................................................................................... 48

9. Conclusion ........................................................................................................... 51

10. Future Work ........................................................................................................ 52

References............................................................................................................... 53
Appendix A: Detailed Muller Transform ......................................................... 55
Appendix B: Detailed Modified Alhagi Method .............................................. 56
Appendix C: Iterated Runs of Preprocessor ..................................................... 58
Appendix D: Source Code ............................................................................. 61
List of Tables

Table 1: Results for completely specified functions.................................45
Table 2: Results for incompletely specified functions...............................47
Table 3: 30 runs on sym10_d_75.................................................................58
Table 4: 30 runs on max46_d_50 ................................................................59
Table 5: 30 runs on newtag_d_75 .................................................................60
Table 6: 30 runs on sao2f2_75 .................................................................60
List of Figures

Figure 1: A PSE Gate ........................................................................................................... 24
Figure 2: Function to be synthesized .................................................................................. 26
Figure 3: Synthesis with PS Implicants .............................................................................. 26
Figure 4: Same function, different realization .................................................................... 27
Figure 5: PSEycic Preprocessor .......................................................................................... 32
Figure 6: Circuit realization ................................................................................................. 34
Figure 7: Inverse Muller Transform .................................................................................... 35
Figure 8: Alhagi's Reversible Synthesis Method ................................................................. 40
Figure 9: Synthesis with Folded K-maps .......................................................................... 41
Figure 10: A realization with ternary PSE gates ................................................................. 50
Figure 11: A ternary function ............................................................................................ 50
Figure 12: 4x4 Truth Table ................................................................................................. 55
Figure 13: 8x1 K-map ........................................................................................................... 55
Figure 14: Function after first pass ...................................................................................... 56
Figure 15: Folded K-map for line B ...................................................................................... 56
Figure 16: After reducing line b again .................................................................................. 57
1. Introduction

Advances in the computing industry have allowed for smaller and smaller transistors, making it practical to put more transistors on an integrated circuit. This has been codified as Moore’s law, that the number of transistors on an integrated circuit will double every two years. Although there have been multiple times in the past decades that Moore’s law has been falsely declared dead, the fact remains that sooner or later the ability to shrink transistors must come to an end. In fact, in 2012, Fuechsle et al. [4] demonstrated a single atom transistor. Of course, this transistor is not commercially practical, but it does demonstrate a hard limit on the minimal size that a transistor can take.

Improvements in classical computing have been met with the quantum problem, as devices get smaller and are placed closer together, quantum fluctuations can no longer be ignored, and must be mitigated or accounted for, or so has been the paradigm for the vast majority of people facing this problem. However, there is a diametrically opposing view, that is, to harness the power of quantum mechanics and create a quantum computer more capable than a classical one.

Such a quantum computer would be able to take advantage of the power of quantum superposition and entanglement and have higher performance than a classical computer for certain classes of problems, as demonstrated by Grover’s and Shor’s algorithms [5,6]. Superposition is the quantum phenomena which allows particles to in two states at once. So, a qubit could be in both state 0 and 1 with a certain amplitude probability attached to each basis state. Entanglement is the quantum phenomena which
allows for two or more seemingly independent particles to instantaneously affect one another. The measurement of one particle can lead to an entangled particle also taking on a specific value. This is where quantum systems greatly differ from analog systems. It is possible to make two qubits take on an entangled superposition in which the qubits have a 50% chance of being measured as 01, a 50% chance of being measured as 10, and no chance of being measured 00 or 11. This cannot be replicated with any classical technology with two independent bits.

Despite controversy, D-Wave Systems produced a functional quantum computer, and this past year, 2013, a team of scientists proved that it is in fact taking advantage of quantum phenomena, making it a true quantum computer [28]. In fact, the past few years have been very good for the field of quantum computing. D-Wave has produced its second generation quantum computer with more qubits that the first generations, Muhonen et al. [8] found a way to extend the decoherence time of a single qubit to thirty seconds, and Everitt et al. have entangled electron and nuclear spins in a Nitrogen vacancy [7].

D-Wave System’s quantum computer is known as an adiabatic quantum computer, this means that the internal computation is performed not by electrical circuits, but by tiny pipes filled with superfluid helium. This quantum computer is very specialized, and is only able to solve one type of problem, quantum annealing.

Quantum annealing is similar to simulated annealing, but there is a very big difference. Classical simulated annealing tries to find a minimum in a multi-variable cost function. However, this function is very complicated, and riddled with local minima (if it
wasn’t, a simpler, deterministic method would be more appropriate). Local optimization is unable to find a global minimum in such a case. This requires the annealer to travel over higher cost regions in order to potentially find different valleys with lower local minima.

A Quantum annealer is capable of tunneling through higher cost regions and jumping directly from one local minima to another with a lower cost. This will allow for faster and more efficient simulated annealing.

However, D-Wave System’s quantum computers are special purpose machines, not general purpose quantum computers. General purpose quantum computers are still not commercially viable. Nonetheless, we feel that these general purpose quantum computers will eventually come to fruition, and CAD synthesis tools will be needed.

In prior work, our group has shown that ESOP is a good form for quantum logic because AND-XOR logic naturally maps to one of the universal quantum logic gates, the Toffoli Gate. In this thesis, we will focus on an extension to ESOP called EOSOPS (Exclusive-OR Sum Of Product-Sums).

Our synthesis algorithm and implementation take advantage of PSE gates introduced by Perkowski et al. [1] Similarly to the Toffoli gate, the PSE gate forwards all of its inputs to its outputs unchanged, except for one (x); the last output (X) is expressed as follows: \( X = x \oplus P \cdot \overline{Q} \), where \( P \) and \( Q \) are arbitrary products of literals. The second part of \( X \) is the **PS term** (Product-Sum term) which is the product of a product term and the complement of a product term, \( (PS = P \cdot \overline{Q}) \). Compared to traditional Toffoli-based reversible circuits, this gate requires one more ancilla qubit that can be reused for all PSE
gates in the same circuit because we use mirror gates for each Q term. A lower cost version of this synthesis can be reached by using more garbage lines, lines that are initialized to 0, but are allowed to take any value at the end of the circuit. We do not utilize garbage lines, because they make a line, or qubit, unusable for the rest of the circuit. If many garbage lines are used throughout a circuit, easily possible if there are many subfunctions synthesized and each of them creates a few garbage lines, the total number of qubits required to synthesize the circuit can expand greatly. Furthermore, we know that adding more qubits reduces the decoherence time. There is a tradeoff between reducing the computation time by using garbage lines with the corresponding reduction in decoherence time. Unfortunately, we do not know that tradeoff, nor are we aware that anyone else does. We assume that the cost is greater than the reward, so we do not utilize garbage lines.

The cost metric for a quantum circuit is the sum of the quantum costs of the gates in that circuit. This cost is technology dependent. For a linear ion trap quantum computer, this cost is the number of pulses needed to perform the operation represented by a quantum gate. The currently used cost function for quantum circuits is Maslov’s cost function, also known as Maslov Cost [9]. Maslov Cost assigns the inverter and all two input gates (single control, single target) a cost of one. For quantum gates with more inputs, a cost is given which is equivalent to the number of two input gates required to realize it. For example, the Toffoli gate can be decomposed to five two input gates, so it has a cost of five. Quantum cost grows exponentially with the number of inputs to a quantum gate.
In this thesis, we will exclusively use a slight variation of Maslov’s cost function. Specifically, we will treat inverters as being free. The reason for this change is twofold. First, we find that an inverter should have a quantum cost of $1/5^{th}$ of that of a CNOT, and since the CNOT is the unit cost, it is simpler, and slightly more accurate to use a cost of 0, than 1. Although this slightly underestimates the cost, it does so uniformly, so all comparisons to other methods are still valid. Secondly, if inverters are not free, then there is a possible post processing stage where the order in which a function is realized will affect the cost. By giving inverters a cost of 0, we reduce the variability in cost for a certain realization. This allows us to make more accurate comparisons. For the purposes of this thesis, Maslov Cost and quantum cost will be used interchangeably.

We will be focusing on a specific class of functions, that is, functions which are inherently irreversible. In other words, we will focus exclusively on functions which are not bijective because they don’t have a 1:1 mapping of inputs and outputs. These functions usually have the property that there are fewer output bits than input bits and, by definition, they always have multiple input values mapping to the same output value.

We focus on irreversible functions because we know from the state and history of computer science and engineering, that the vast majority of interesting functions are not reversible. One need only look at the operations available in a modern microprocessor to see the lack of inherently reversible functions and operations. The reversible operations are limited to negations, increments, and XORs where one of the inputs is preserved. Several complicated operations that are reversible are transformations. These include several operations in DSP and graphics processing. Another group of transformations
include code transformations, for example converting between binary order and Gray
code, encryption/decryption, and compression/decompression.

A Quantum computer can be simplistically described as a classical reversible
computer, with the quantum principles of superposition and entanglement providing a
level of massive parallelism. A reversible computer can be viewed as a classical
computer with certain restrictions, namely that only reversible operations can be
performed. By analogy, this can be imagined as being similar to a computer with a single
register of arbitrary length. At each time slice, any bijective (i.e. reversible) operation can
be performed on the data stored in this register. Because the composition of two bijective
functions is itself a bijective function, any number of these operations can be applied and
no information will be lost.

Under these restrictions, it is not possible to calculate irreversible functions
because there are no permitted operations that will lose information. Stated another way,
it is not possible to arrive at a unique input from every possible output of an irreversible
function. However, by adding enough output lines to make each output value unique, any
irreversible function can be transformed into a more complication reversible function
which, under some subset of inputs and outputs, realizes the original irreversible
function. For example, we might take a five input, single output oracle and transform it
into a six input, six output, reversible function. If we only look at five of the inputs and
one of the outputs (the ones that correspond to our original oracle), we will have the same
function realized as the original oracle.
This means that if we wish to perform an irreversible calculation using a quantum computer, we must first transform the function to be calculated, or operation to be applied, into a reversible function or operation. There are two main ways of transforming irreversibly specified functions of m inputs and n outputs into reversible functions. The two most common approaches are the minimum bit approach to minimization and the \(m+n\) approach [26]. In the minimum bit approach, the minimum number of additional output lines are added to make each output unique. In the \(m+n\) approach, inputs and outputs are separated to different lines. Input lines pass their values through unchanged. Output lines are initialized to 0 and take the calculated value on their output.

In the minimum bit approach, the circuit requires the greater of the following: the number of inputs or the number of outputs plus sufficient ancilla lines to make each output unique. If we take a full-adder for example, there are three inputs and two outputs. However, the total number of required lines is four because the output “01” is repeated three times, and two ancilla lines must be added to the output.

As is apparent from the name, the \(m+n\) approach results in a circuit that has \(m+n\) lines as a minimum. Optionally, any number of ancilla lines can be added. With the exception of several trivial classes of functions, the minimum bit approach results in a circuit that has fewer bits than the \(m+n\) approach.

We will only focus on the \(m+n\) approach in this thesis for the following two reasons. First, this approach makes transformation from an irreversible ESOP specification [19], as well as any other similar specification that has a global XOR on the final stage, to a reversible specification trivially easy. Furthermore, and more
importantly, the two most important algorithms that allow for a quantum speedup are Shor’s algorithm and Grover’s algorithm which provide exponential and quadratic improvements in speed over classical computers respectively. These two algorithms require that the inputs be passed through the function unchanged. The reason is that the measurement is performed, not on the output qubits, but on the input qubits.

A measurement is a type of classical interaction with a qubit that returns one of the basis values of the qubit (1 or 0). This type of interaction collapses the wave function of the qubit, and is innately irreversible if the qubit is in a superposition state. This is the only potentially irreversible operation that a quantum computer performs. In practical terms and assuming a linear ion trap quantum computer, this would mean using a laser to check if an ion is in the ground state or in an excited state.

Quantum computers have their computational power restricted by the decoherence time of their qubits. Furthermore, a greater number of qubits in a quantum computer reduces the decoherence time, because it is more likely that two qubits will interact in an undesired fashion. It is a well-known property of reversible and quantum computing that adding additional ancilla and garbage lines reduces the number of gates needed to realize a function, however, the total number of bits in a quantum computer will necessarily be limited.

Quantum computers can provide exponential or quadratic improvements in performance over classical computers on certain problems using Shor’s algorithm and Grover’s algorithm respectively. However, a problem that takes long enough to solve that
the qubits decohere out of their quantum states cannot be solved on that quantum computer.

There are four ways to move a problem from the unsolvable category, to the solvable. The first is to increase the decoherence time of the qubits, so the same number of qubits can maintain their quantum state for a longer period of time. The second is to improve the containment around these qubits, allowing for more qubits in a quantum computer without negatively affecting the decoherence time. The third is decreasing the cycle time of the computer. Similar to increasing the clock speed in a classical computer, this means that each operation (a series of laser pulses) takes less time to perform, which allows for more operations to be performed in a constant amount of time. These problems are all highly related physical challenges and fall outside the scope of this thesis.

The fourth way is to improve reversible synthesis techniques, which will allow for functions to be synthesized into more efficient circuits, thereby allowing functions that previously could not be implemented reliably on a particular quantum computer to now be implemented. The best part of this approach is that it is independent of the physical advances and synergizes with them. In other words, if the physical limit of a quantum computer is 1000 operations, and an improved synthesis algorithm results in 20% savings over an older algorithm, this is equivalent in power as if the physical limit of the computer was 25% higher, i.e. 1250 operations. This advantage grows with any advancement in the isolation, containment, or targeting of a quantum computer. If the decoherence time is increased or cycle time decreased so that there is sufficient time for 2000 operations, a 20% improvement in the results of the synthesis algorithm will have
the same improvement in computational power as if there was sufficient time for 2500 operations using the old synthesis algorithm, a net gain of 500 operations. Because these advances have the property that their effects are multiplied, it is important that both the physical and synthesis components are improved for overall performance improvement.

This should provide sufficient motivation that tools for the synthesis of irreversible functions into reversible circuits are not merely warranted, but needed.

The thesis is outlined as follows. First we will reintroduce background information necessary to understand the topics in this thesis. Then we will introduce other synthesis algorithms and explain whether or not it is possible to make a fair comparison between each existing algorithm and our new algorithm. Starting in section 4, we will introduce our new contributions, namely the PSE gate and our synthesis algorithm. Following that, are sections with the numerical results that we have achieved, and then how this algorithm and tool can be combined with the Muller transform to work on multi-output functions, how it can be extended to multi-valued logic, and how it can improve the performance of Alhagi’s synthesis method. Finally, we will conclude our work, and mention ways in which this work can be further expanded.
2. Background

2.1 Overview

This section will introduce the concepts necessary to understand this thesis: inherently reversible and irreversible functions, ESOP synthesis, reversible synthesis, the reversibilization process, and reversible circuits and the Toffoli gate. Later sections will reintroduce PSE gates, the concept of inhibition of logic functions, our new algorithms for synthesizing with PSE gates, as well as the results of these algorithms as applied to some well-known benchmarks.

It is assumed the reader has working knowledge of the following concepts: Karnaugh-maps (K-maps), minterms, cubes, basic Boolean algebra, and the two input logic gates. These topics will not be explained in this thesis.

2.2 Boolean Logic

Before proceeding any further, we will quickly review a few basic precepts of logic synthesis, namely, AND-OR logic, AND-EXOR logic, and how they differ. This will serve as an important foundation for this work.

2.2.1 AND-OR Logic

AND-OR logic simply means that the logic operations used to describe a function and therefore the logic gates used to realize a function are limited to the AND and OR gates and operations. There are two commonly used regular structures and an innumerable number of irregular structures.

The two two-level regular structures are Sum of Products (SOP) and Product of Sums (POS). SOP logic results in logical expressions composed of a logical sum (OR) of
multiple products of literals (AND). For example, the expression $A + \overline{B} + \overline{C} + D$ is in SOP form. We will call variables and their negations, “literals.” In terms of circuits, the product clauses are realized with AND gates and the output of the AND gates are all fed into a global OR gate.

POS, as can be gathered from the name is very similar to SOP, but the order is reversed. POS expressions have OR clauses that are ANDed together. For example, the expression $(\overline{A} + B) \cdot (C + D)$ is in POS form. Likewise, a POS circuit has multiple sum clauses realized with OR gates feeding into a global AND gate.

Furthermore, AND-OR expressions and circuits do not need to be two levels deep nor do they need to have the same types of gates at each level. However, the regular structure of SOP has made it possible for researchers to create many algorithms and programs for the synthesis of SOP circuits. The most famous of these synthesis programs is ESSPRESSO.

Conceptually speaking, a single output function can be completely specified by providing the following information: the set of all input values which must result in a high output, the set of all input values which must result in a low output, and the set of all input values in which the output may be permitted to take any value. We call these the ON-set, OFF-set and Don’t-Care-set (DC-set), respectively. Furthermore, because there are a finite number of input values and these three sets together contain every possible input value without any repetitions, knowing any two of these three sets is sufficient to describe the circuit because the third will be known implicitly.
ESPRESSO, uses the ON-set and DC-set during its synthesis. SOP synthesis, like in ESPRESSO, must follow certain basic rules. Minterms corresponding to the ON-set must be covered with a cube at least once. Minterms corresponding to the OFF-set cannot be covered by any cube. Minterms corresponding to the DC-set can be covered by any number of cubes, including 0.

The ideal way to find a high quality cover is to use essential prime implicants. An implicant is a subfunction that covers only asserted minterms and possibly Don’t Cares. A product implicant is an implicant which is a product of literals. A prime implicant is a product implicant that cannot be made any larger by removing a literal from the product and still covering only asserted minterms and possibly Don’t Cares. An essential prime implicant is a prime implicant that covers a minterm which is not covered by any other prime implicant. Since larger cubes are cheaper to implement than smaller cubes and implementing a fewer number cubes is cheaper than implementing a greater number of equally sized cubes, essential prime implicants are useful for efficiently synthesizing SOP.

2.2.2 AND-EXOR Logic

Like AND-OR logic, AND-EXOR logic refers to a logic system where the only permitted operations on literals are AND and XOR. The primary regular structure within this paradigm is Exclusive Sum of Products (ESOP), wherein multiple product terms are then summed by a global XOR. An example of an ESOP function is $\bar{A}B \oplus C\bar{D}$, where the circumscribed plus sign is the symbol for the XOR operation. In binary and quantum binary logic, XOR can be viewed as modulo-2 addition.
The rules of ESOP and ESOP synthesis are simultaneously less restrictive, but also more complicated. Each minterm in the ON-set must be covered an odd number of times. Each minterm of the OFF-set must be covered an even (including zero) number of times. Like in SOP, the minterms in the DC-set can be covered any number of times.

The ability to select a minterm in the OFF-set an even number of times allows us to select cubes that are larger, hence cheaper, than the largest prime implicant. These large cubes by definition contain OFF-set minterms. As long as these minterms are later covered again so that they are selected an even number of times before synthesis is complete, there are no violations of the synthesis rules. This causes ESOP synthesis to be much more complicated than SOP synthesis. SOP synthesis is asymptotic, each selected cube brings the synthesis closer to completion; one or more minterms are satisfied, and no minterms require a second covering. In contrast, in ESOP synthesis, it is possible that a selected cube satisfies certain minterms, but then also requires other minterms that were already satisfied to require more attention at a later step.

While there are many cases where a larger cube and a smaller cancelling cube are cheaper than two smaller cubes, the real power of this approach arises when two neighboring terms can be expanded and cancel out each other’s negated minterms. The simplest possible example is the XOR function itself. One valid representation of this function is: $\overline{a \overline{b} + \overline{a} \overline{b}}$. However, an equivalent and cheaper representation is: $a \oplus b$. In the second equation, the cube $a$ and the cube $b$ both cover the negated minterm $ab$. Because it is covered by two cubes, the minterm correctly evaluates to 0.
2.3 Boolean Ring

As mentioned earlier, we assume the reader is familiar with the basic parts of Boolean algebra, but not necessarily with Boolean Rings.

Boolean rings are based on the following rules. All variables correspond to literals or products of literals.

\[
\begin{align*}
X \oplus 0 &= X \\
X \oplus \overline{X} &= 1 \\
X \oplus X &= 0 \\
1 \oplus X &= \overline{X} \\
1 \oplus \overline{X} &= X \\
X \oplus Y &= Y \oplus X \\
X \oplus (Y \oplus Z) &= (X \oplus Y) \oplus Z \\
X (Y \oplus Z) &= XY \oplus XZ \\
X \oplus XY &= X \overline{Y} \\
\overline{X} &= X \\
X + Y &= X \oplus Y \oplus XY = X \overline{Y} \oplus Y = \overline{X} Y \oplus X = 1 \oplus (X \ast \overline{Y}) \\
A \overline{B} &= A \ast (AB)
\end{align*}
\]

Here we see the power of EXOR logic. As mentioned earlier, the property of EXOR logic that allows us to negate a minterm after having asserted it gives us more freedom of action compared to AND-OR logic. This freedom comes with a corresponding cost. When synthesizing for a SOP representation, there is a hard ceiling on the number of product terms that can be summed without repetition that is smaller than the set of all possible cubes for that many variables, because the selected cubes cannot intersect with the OFF-set. Although this number can be innumerably large, these restrictions guide synthesis in such a way that makes efficient SOP synthesis feasible.

Conversely, when synthesizing for ESOP, we are able to select any possible cube for the
number of variables that we are working with because any false minterms that are covered can later be cancelled.
3. Current Synthesis Methods

To the best of our knowledge, the only published paper on the topic of synthesizing irreversible functions onto a reversible circuit is by Kumar et al. [13]. However, as mentioned previously, a function in ESOP form can be trivially converted to one type of reversible circuit. This means that our algorithm directly competes with all ESOP synthesizers. The accepted gold standard in this field is Alan Mishchenko’s EXORCISM v4, an ESOP minimizer.

Additionally, we will also go over several other reversible and quantum synthesis methods and explain how they differ sufficiently to make a comparison impractical. These methods include: MMD, and DCARL, and we will also briefly mention MMD-like algorithms and cycle-based algorithms as a group without delving into individual algorithms.

3.1 EXORCISM

EXORCISM is the current best ESOP synthesizer. The original program was written by Martin Helliwell [20]. The next two versions were written by Ning Song [21,22]. The current version, version 4, was written by Alan Mishchenko [19], currently a senior scientist at the University of California at Berkeley.

EXORCISM v4 works by using a single cube operation called exorlink. It takes two cubes in the cube list and performs the exorlink operation on them. This operation does different things to the cubes depending on what Hamming Distance separates the cubes.
A Hamming Distance of 0 means that the cubes are identical. Per the third equation previously posted, we can see that these two cubes cancel out and are therefore both removed from the cube list.

A Hamming Distance of 1 means that a single literal differentiates the two cubes. These two cubes will be combined into a single cube of smaller cost. For example, $abc \oplus ab\bar{c} = ab$.

A Hamming Distance of 2 means that two literals differentiate the two cubes. HD-2 cubes cannot be combined into a single cube; however they can be replaced with a pair of cubes of potentially smaller cost. For example, $abc\bar{d} \oplus ab\bar{c}d = abc \oplus abd$.

A Hamming Distance of 3 means that three literals differentiate the two cubes. This operation will replace the two cubes with three cubes of possibly, but not necessarily, smaller cost than the original cubes. This operation will increase the total cost because EXORCISM uses “Cubes.Literals” as its cost function. That is, more cubes are always treated as more expensive, regardless of their size. This is because EXORCISM was originally written for classical, not quantum, ESOP synthesis.

A Hamming Distance of 4 means that four terms differ between the two cubes. This operation will always increase the total number of cubes and also result in more expensive cubes.

It may seem counterintuitive that this algorithm uses operations that increase the cost of the circuit. However, these operations are necessary in order to find lower cost ESOP realizations of a function. One cannot reach the minimal ESOP circuit simply by continuously minimizing. If one were to visualize the cost function for an ESOP circuit,
one would see an uneven landscape filled with many local minima. These high Hamming distance operations allow EXORCISM to escape from a local minimum into another valley and attempt to find a possibly lower minimum there.

Greatly simplified, from an input cube list, EXORCISM will first use HD-0, 1, and 2 operations to find the local minimum. Once this is found, EXORCISM will use HD-3, and 4 operations to escape from that local minimum into another valley, at which point HD-0, 1, and 2 operations are yet again used to find the local minimum.

This process repeats until a certain number of new valleys have failed to produce a better minimum. This result is then returned as the solution. Although no guarantees are provided that this is the minimal solution, this algorithm is currently acknowledged as the best ESOP minimizer available for large variable counts.

However, EXORCISM is not without a glaring fault. Prior versions of EXORCISM (v1-3) all took into account the DC-set and tried to synthesize in such a way as to properly utilize the freedom and power of Don’t Care terms. EXORCISM v4 does not utilize the DC-set; it simply treats everything that is not part of the ON-set as being part of the OFF-set. This leads to paradoxical situations where simplifying the function by replacing positive minterms with Don’t Cares results in a more expensive circuit than the original.

EXORCISM-4 is more streamlined than the previous version, able to deal with larger functions, and much faster than its previous, Don’t Care aware, versions. However, the streamlining came at the cost of correctly dealing with Don’t Cares. We will address this shortcoming with our new algorithm later in the thesis.
3.2 Non-Competing Methods

In this section we will introduce several other synthesis methods, which cannot be fairly compared with our method. The reason why these algorithms cannot be compared with ours is that these work on problems that are inherently reversible, a completely different class of problems. Although the end result is similar, these methods as well as our method all produce reversible circuits from a problem specification. It would be trivial to produce problems or functions which synthesize well (that is with a low quantum cost) with our algorithm and poorly with the algorithms below. However, it would be just as easy to produce problems for which the reverse is true.

3.2.1 MMD

First and foremost among the methods that deal with inherently reversible functions is MMD, named after the creators of this algorithm, Miller, Maslov, and Dueck. MMD begins with a truth table of the function to be synthesized. Proceeding in binary order, it adds gates from output to input. The order ensures that synthesized minterms are never affected again by future operations. MMD is by no means the best synthesis tool of its kind. However, there is an entire branch of tools that have been based on MMD and improved it in various ways.

MMD synthesizes functions that are inherently reversible, and hence have the same number of inputs and outputs. The function must also be fully specified. If the problem is not inherently reversible, it must be modified so that the number of outputs and input is the same, all outputs are fully specified, and all values map to unique output
values. Our team has done a few comparisons, and we have found that using MMD on these classes of problems results in solutions with very large quantum cost [27].

3.2.2 DCARL

DCARL by Majith Kumar et al. [13] stands for Don’t Care Algorithm for Reversible Logic. It addresses one of MMD’s shortcomings, namely that MMD does not handle Don’t Cares. DCARL serves as a preprocessor to MMD, exhaustively filling in all Don’t Care values with 1s and 0s. The program ensures that the values filled in for the Don’t Cares do not violate the tenets of reversibility. Furthermore, it also iterates over all valid replacements for the Don’t Cares. It gives all of these problems to MMD and selects the one that results in the lowest quantum cost.

Since this uses MMD at its core, all but one of the reasons against comparing MMD and our algorithm still stands. This algorithm deals with incompletely specified functions like our algorithm, but it still works on functions that have the same number of inputs and outputs.

3.2.3 Other MMD-like Algorithms.

As mentioned previously, there are many more algorithms that are based on MMD. [15,16,17,18]. All of these algorithms require inherently reversible functions. They do not work with the class of functions that we are interested in, inherently irreversible functions.

3.2.4 Cycle Based Algorithms

Another group of algorithms which need to be mentioned are the cycle based algorithms [23]. Whereas the MMD-like algorithms focus on how each individual input
value changes to reach the correct output value, cycle-based algorithms focus on cycles that occur when the output is passed back into the input. For example, the Toffoli gate can be viewed as a gate that passes all of its input to outputs except the last one, where it realizes $C = c \oplus ab$. However, when looking at it from the point of view of the Permutation Group Theory based on cycles, the Toffoli gate realizes the cycle, or transposition, (6,7). That is, when the input has a value of 110 (or 6 in decimal), then the output will be 111 (or 7 in decimal). Likewise, if the input is 7, then output will be 6.

Just like the MMD-like algorithms, these algorithms cannot be fairly compared with our algorithm because they require the problem to be inherently reversible. This is because the two requirements to decompose a function into cycles are another way of stating that the function is reversible. First off, the number of inputs and outputs must be equal. If the number of outputs was greater than the number of inputs, the output would not be able to be fed back to the input. If the number of outputs was smaller than the number of inputs, then it would be impossible for every input value to map to a unique output value, which is the second requirement. Each input value must map to a unique output value, otherwise the function would have merging paths when one attempted to decompose the function to its cycles.

These two requirements, number of inputs equals number of outputs and all input values map to unique output value, mean that only bijective functions will satisfy the requirement. Since all bijective functions are reversible, we reach the conclusion that this method only works with inherently reversible functions. Furthermore, we are not aware of any algorithm which, like DCARL, can loosen these restrictions.
4. PSE Gates and PSEycic

PSE stands for Product-Sum-EXOR. It is a three level circuit. At the highest level, a global EXOR is performed on a number of PS implicants. A PS implicant is the product of several literals and a single negated product of additional literals. In other words, a PS implicant takes the form $P \cdot \bar{Q}$ where $P$ and $Q$ are both arbitrary products of literals. An objection may be raised at this point that the above form does not represent a Product-Sum implicant, but rather a Product-NAND implicant. By DeMorgan’s rule, we know that the negation of a product is the same as the sum of the negation of the literals in the product. In other words: $\overline{(abc)} = \bar{a} + \bar{b} + \bar{c}$.

If we were to represent these PS implicants on a Karnaugh Map, they would take the form of a normal cube (product of literals), with a secondary, smaller, cube subtracted from it. We call this a “cube difference” as the result is the difference between a larger cube and a smaller cube.

Drawing on a concept from classical logic, this structure is the reversible analogue of classical inhibition. However, inhibition is not a reversible operation. Much like how an irreversible function can be made reversible by adding a sufficient number of ancilla bits, the reversible realization of inhibition is the PSE gate, which requires the addition of an extra ancilla bit.

4.1 PSE Gate

Although we have stated the logical form of a PSE circuit, we have not illustrated how to build this gate. As we stated, this is not a reversible operation, therefore we require an extra ancilla line. However, the operations targeting this ancilla can be
mirrored. This means that only a single extra ancilla line needs to be added for an arbitrary number of PSE gates.

Starting from a simple example $F = ab\overline{cd} \oplus f$, we begin by realizing the most deeply nested part of the equation, namely: $\overline{cd}$. This is a simple operation, a Toffoli gate is placed with the controls going to $c$ and $d$, and the output targeting a cleared ancilla line, $T1 = 0$. This makes $T1 = cd$. Then we simply invert the line and $T1 = \overline{cd}$.

Already, there is an optimization that should be applied. Rather than targeting a cleared ancilla line, we will target a preset ancilla line $T1 = 1$. By doing this we can reduce the cost by a single inverter. With this change, the value of T1 after the Toffoli gate is $T = 1 \oplus cd = \overline{cd}$.

Now that we have $T1$ set to $\overline{cd}$, the next step is simple, we use a 4x4 Toffoli gate that is controlled by $T1$, $a$, and $b$, and targets $f$. The value on line $f$ is now: $f \oplus ab\overline{cd} = F$, which is what we want our output to be.

With the output at the correct value, all that remains is to restore $T1$ to its original value. To do this, we simply repeat the first Toffoli gate. This makes $T1 = 1 \oplus cd \oplus \overline{cd} = 1$, which is the initial input to $T1$.

From Figure 1, we can see that the PSE gate will forward all but one of its inputs unchanged (including the ancilla bit), and the final bit will be EXORed with a PS implicant. One stylistic note, the outputs are written in capital letters, even if they are unchanged by any operation inside of the circuit. The reason for this is

Figure 1: A PSE Gate
that it avoids confusion. Lower case variables refer to the value on a line before the circuit and capital variables refer to the value on a line after the circuit. This is important because “input” and “output” are ambiguous in these types of circuits.

From the point of view of the circuit, everything on the left side is an input, and everything on the right is an output. However, when viewed from the perspective of the function that we are synthesizing, the top four lines are input lines, the fifth is an ancilla line (used to temporarily hold a value, and then returned to its original value), and the bottom line is the output line. We will attempt to clearly distinguish these two different meanings by using the unambiguous terms “input to the circuit” and “input line,” likewise for outputs.

4.2 Synthesis with PSE gates

As of the writing of this thesis, there are only two ways of synthesizing circuits with PSE gates. The first is of course by hand, and the second is using the synthesis suite that is presented as part of this thesis.

4.2.1 Synthesis with PSE gates by hand.

To understand the power and the usefulness of PSE gates, we will first go through the process of synthesizing a multi-input, single output function by hand. This is a non-algorithmic presentation of the thought process that should go into synthesizing circuits with PSE gates.

Much like synthesis of classical ESOP circuits, we begin with a Karnaugh Map like in Figure 2, to take full advantage of the human ability to see patterns in visual media. Our first goal is to identify any shapes inside of the Karnaugh map that
correspond to a cube difference. As presented previously, a cube difference on a Karnaugh map looks like a normal cube with a subcube cancelled out. On a small Karnaugh map, this subset will almost always be a corner or a side of the larger cube.

In Figure 3, the function can be solved with two PSE implicants. This is cheaper than the ESOP solution.

Care must be taken when considering whether it is more appropriate to select cubes or cube differences. A cube difference realized by a PSE gate will be cheaper than reconstructing the shape from multiple disjoint cubes, or from selecting a large cube and cancelling a subcube from it. However, there are cases where partial cancellation of cubes as in regular ESOP will give results that are superior to selecting multiple cube differences (PS implicants). For example, the function $F_1 = ab \oplus cd$ can be realized with two cubes, or with two PS implicants. The two cube realization, like in regular ESOP, will be cheaper than the two-PS implicant solution. This is because the two cubes already perform the same cancellation for each other that the PS implicants require additional logic to perform individually.
It must be emphasized that the PSE gate and cube difference synthesis are not a replacement for Toffoli gates and the regular Odd-Even covering synthesis approaches. An individual PSE gate is more expensive than an individual Toffoli gate, but represents more complicated logic structure than a simple cube, and thus covers more asserted and fewer negated minterms. Using a PSE gate where a Toffoli gate will suffice is a guaranteed way to increase, rather than decrease the quantum cost. Judicious use of these tools can, and does, result in circuits with lower quantum costs compared to prior synthesis methods. Reckless use of these tools can easily lead to much larger quantum costs and unnecessarily complicated circuits, see Figure 4a, which shows the minimal realization of this function and Figure 4b, which shows a bad PSE realization.

Figure 4: Same function, different realization

Continuing with the synthesis, once the large cube and cube differences have been identified, we select the true minterms that have not yet been selected (or selected an even number of times). We also identify all false minterms that have been selected an odd number of times and select them once more so they are covered an even number of times. These remaining minterms that have not been satisfied form a remainder function. This remainder function, usually of high cost, can then be synthesized using methods from standard ESOP synthesis.
4.2.2 Synthesis with the PSEycic Tool Suite

The PSEycic tool suite, developed by me for this thesis, consists of three parts: The PSEycic preprocessor, the PSEycic postprocessor, and Alan Mishchenko’s EXORCISM v4. EXORCISM has been introduced earlier in this thesis; the preprocessor and postprocessor are part of the contribution of this thesis.

One of EXORCISM’s well known problems is that its performance suffers on incompletely specified functions. This is due to the fact that it treats Don’t Cares as being part of the OFF-set. This means that EXORCISM gives worse results when synthesizing an incompletely specified function as compared to its completely specified counterpart. The preprocessor takes multiple input, single output, incompletely specified functions where the ON sets and OFF sets are specified in PLA format [29].

This algorithm uses a heuristic to choose values for the DC set so EXORCISM can work on a fully specified function. Once EXORCISM minimizes the fully specified function to the best of its ability, the results are passed to the postprocessor.

The postprocessor identifies cube patterns of Toffoli gates that can be replaced with PSE gates of lower cost. The final output is printed in a format that is a modification of the original ESOP format used by EXORCISM.

4.2.2.1 PSEycic Postprocessor

The PSEycic postprocessor is a simple cost reducer that works on any ESOP file, whether produced by EXORCISM or any other source. From the ESOP specification it reads and stores every cube that is used to represent the function in ESOP form. It identifies any cube that is completely contained within another cube and replaces the two
cubes with a PS implicant. Once it has compared every possible pair of cubes, an O(n^2) operation, these cube pairs are replaced with PS implicants using the final equation from the background. Every literal that is shared between the two cubes is removed from the smaller cube, and then this new cube is negated. The product of this new negated cube and the original containing cube is a PS implicant.

For example, if we have a cube list containing the following three cubes, “111X1, 1XX11, and 10011” The program will go through all permutations of two cubes, searching for one which is contained in another. In this example, cube “10011” is completely contained by cube “1XX11.” The program then removes all literals that are shared between the two cubes from the contained cube. This means that “10011” becomes “X00XX.” This cube now becomes the inhibition term for the original containing cube “1XX11.” In the end, we are left with cube “111X1,” which is realized by a single generalized Toffoli gate, and “1XX11 inhibited with X00XX,” which is realized by a single PSE gate.

Once this processing has been completed, we write the unchanged ESOP terms and the additional PS terms into a new .eospos file. The format of this file is an extension of EXORCISM’s ESOP file format.

4.2.2.2 PSEycic Pre-processor

The PSEycic preprocessor addresses EXORCISM’s previously stated problem with Don’t Cares. The preprocessor accepts an incompletely specified function as its input. Using a refined version of the conditional coloring algorithm that we have
presented in prior work [1], this algorithm attempts to find large cubes and cube differences so as to intelligently assign all Don’t Care values to either high or low.

The preprocessor begins with a complete list of the ON-set. This list is shuffled at the beginning of the preprocessor, which allows us to achieve different results on subsequent iterations of the preprocessor. The preprocessor takes the first minterm in the ON-list as a core. The preprocessor then works through the entire ON-list in order trying to expand this core. All subsequent minterms are tested for compatibility. They can either be compatible, incompatible, or conditionally compatible. We explain these terms shortly.

If the second minterm is compatible, it is added to the core to create a larger cube. All minterms covered by this new core are moved from the ON-list to the OFF-set and the next minterm in the list is selected for the next step. If the second minterm is incompatible with the core, it remains in the ON-list and the next minterm is selected for the next step. If a minterm is found to be conditionally compatible with the core, it is added to the core along with the condition, and all covered minterms are moved from the ON-list to the OFF-set.

This process is repeated until the entire ON-list is traversed. The resultant cube described by the core, along with the condition if there is one, are stored as ESOP cubes. The process is repeated with the first remaining entry in the ON-list forming the new core until the entire list has been exhausted.

Because this is a preprocessor and the results will be used in subsequent steps, finding an exact minimum is not of the greatest concern. Therefore, this algorithm is a
greedy algorithm with no backtracking. As stated earlier, there is a randomization step, so the preprocessor can be run multiple times in order to find potentially superior results.

Now we will describe what we mean by compatibility, incompatibility, and conditional compatibility, first in plain English, later with cube and set formulas. The core is compatible with a minterm if the supercube of the core and the minterm do not cover any elements of the OFF-set. The core is conditionally compatible if the supercube of the core and the minterm intersect part of the OFF-set, but the supercube of all covered false minterms does not in turn cover any element of the ON-set. The core is incompatible with the minterm if the supercube of the core and the minterm intersects part of the OFF-set, and the supercube of all false minterms covered by that cube in turn intersects with an element of the ON-set.

Now, mathematically: we will call the core “C,” the minterm that we are trying to add “m,” the OFF-set as “F,” and the ON-set as “T.” The core is compatible with the minterm when \((C \cup m) \cap F = \emptyset\). The core is conditionally compatible when \(\cup [(C \cup m) \cap F] \cap T = \emptyset\). All remaining cases are incompatible. Please note, “\(\cup\)” is being used to represent the supercube operation, while “\(\cap\)” is being used to denote the set intersect operation. These are the two main operations of “Cube Calculus,” a notation and algebra used to implement a subset of certain logic synthesis algorithms. Also, “\(\cup\)” is being used to represent the iterated supercube operation.

As an example, let us take \(ab\oplus cd\) with some minterms replaced with don’t cares as in Figure 5a. One run of the preprocessor might look like this (it is not guaranteed to perform these steps in this order because the algorithm is randomized). First a minterm is
selected to be the core, in this case 1100. 1100 is incompatible with 0111, because there is no cube or cube difference that can cover both those minterms without covering false minterms as well. However, 1100 is compatible with 1110, since both minterms can be covered by 11X0. 11X0 now becomes the core as in Figure 5b. 11X0 is incompatible with 1011. Since there are no more minterms left, all positive minterms covered by the core are turned to 0s, like in odd-even covering, and the core is added to a cube list. Next, 0111 is selected. It is conditionally compatible with 1011. In other words, these two minterms cannot be covered by a cube, but they can be covered by a cube difference as in Figure 5c. Now that all minterms are covered, the cube difference is decomposed to its two constituent cubes, XX11 and 1111, and added to the cube list. EXORCISM requires a cube list as its input, so we must use a pair of cubes to represent the cube difference rather than utilizing a PS-implicant.

Figure 5: PSEycic Preprocessor
5. The Muller Transform

The PSE minimizer is only defined for the single output Boolean function \( f_i \), specified by the ON-set \( \text{ON}(f_i) \) and the OFF-set \( \text{OFF}(f_i) \). To use it for multiple-output functions, a multi-output to single-output transformation must be applied. R.E. Miller presented the so-called \textit{Muller transformation} (attributed to Muller) in [24] that transforms a multi-output function with \( m \) input and \( n \) output variables to a single-output function with \( m+n \) input variables. The method is not practically applicable for multi-output functions that are represented by the ON and DC sets (such as in Espresso), because it creates \( m \ast (2^n - n) \) additional DC-minterms. This disadvantage is not relevant in our PS minimization, because the DC-set is absent from our representation as we store only ON and OFF sets. We are using the principle of “don’t care about Don’t Cares” to avoid this expensive penalty.

On a cube representation, the transformation is incredibly simple. We prepend the one-hot representation of each output line to the cubes specified for that output. This means that if output C has a cube \( a\bar{b} \) selected, the algorithm will transform that cube into \( a\bar{b}\bar{A}\bar{B}\bar{C} \), where the lower case letters are input lines, and the capital letters are output lines. A cube with zero or more than one output asserted does not correspond to any physical reality, and cannot be obtained directly from the transformation. However, if this transformed function is then optimized by tool that properly handles Don’t Cares, such as the PSEycic tool suite presented in this thesis, it is not merely possible, but expected that the synthesis will result in cubes that include these non-physical cubes. Although they do
not correspond to any physical reality, logically these cubes represent cubes that are shared between two or more outputs of the circuit.

As an example, Figure 6 is a multi-output function which has the following entry in its cube list “1X0 110.” That is, the cube 1X0, or $A\overline{X}\overline{C}$ should be applied to output R and S, but not T. Depending on the implementation, this entry would be transformed into either “1X0XX0” or “1X0100 and 1X0010.” Both the single cube and double cube representation mean the same thing, but we use the double cube option, because it produces more don’t care minterms in this transformed six input, one output function. A more detailed example is presented in Appendix A: Detailed Muller Transform.

Figure 6: Circuit realization

The reverse Muller Transform takes the synthesized m+n input, single output function and transforms it into a realization of the original m input, n output function. If the function was properly minimized after the Muller transform, the reverse transform will provide us with a solution that optimally reuses subfunctions from the various outputs. However, even if the synthesis was not minimal, the reverse Muller Transform will still give us a sub-function sharing answer, although it will not necessarily be the best possible result.
An additional point, it is possible that PSEycic will produce a result which involves inhibition by both input and output lines. Again, this seemingly nonsensical result means that some outputs are given the uninhibited arguments, and the inhibited outputs are given the full inhibition.

As an example, a PS implicant on the transformed function of $ab\bar{S}(\overline{cdR})$ means that all output lines except $S$ and $R$ have the cube “abxx” applied to them, output line $R$ has the PS implicant $\overline{abcd}$ applied to it, and output line $S$ is unaffected. This is demonstrated in Figure 7.

As can be seen, the circuit cheaply realizes both a cube shared between two outputs, and also a PS implicant which uses that cube as well.

This circuit will be one element in a cascade, where each element or subfunction corresponds to one entry in the cubelist. In the event that the subcircuit in Figure 7 is the first element of the cascade, all the output lines ($r$, $s$, $t$, and $u$) will be initialized to 0. If this subcircuit is any of the later elements in the cascade, the output lines will begin with the output value from the previous element in the cascade.

We have not implemented this transform for our algorithm, but we have shown how the regular transform and our special inverse transform work.
6. Alhagi Method

The ideas presented in our thesis can be used to improve the performance of Alhagi's synthesis method for truly reversible functions [12]. Furthermore, we will present several concepts that help justify the decisions made by Alhagi regarding his synthesis method and also adjust his algorithm to make it clearer and more usable.

6.1 Background

Alhagi's method applies to MxM reversible circuits. The fundamental approach is that he divides the synthesis in two stages. The first stage tries to reduce the Hamming Distance of each line independently, starting from the line that has the greatest number of differences, or deltas, between the input and output and then moving on to each line in order of decreasing deltas. After each line has been locally minimized once, the addition of a single garbage line allows for all remaining deltas to be removed, each with a high order Toffoli gate.

The procedure for this algorithm is as follows. One must first have a truth table for the function to be synthesized. Using this information, deltas, or differences, are calculated for the output columns, which correspond to each line of the circuit. Secondly, the line, or column, with the greatest number of deltas is selected for synthesis. An M input, single output Karnaugh map is constructed. The outputs of the Karnaugh map are the outputs specified in the truth table for that line and the inputs are the values in the truth table.

Unfortunately, this Karnaugh map cannot be solved as an ESOP of arbitrary cubes, which is explained below. First, the cube corresponding to the line being
synthesized must be selected (if line A is being synthesized, then cube 1XXX, corresponding to variable “a” must be selected). The reason is that the output passes the input through, along with some changes, which we call deltas. An equivalent statement is that these deltas are the Boolean difference of the input and output of the function to be synthesized. Furthermore, the deltas can be separated into two categories, those that don't depend on the target line, and those that do. As an equation,
\[
\text{Deltas} = F(\text{all lines except the current one}) \oplus G(\text{all lines}).
\]
From this equation, we identify the goal of the first portion of the algorithm, reducing the number of deltas by removing all of the deltas that do not depend on the line on which they exist. Because a line cannot be both a control and the target of a gate, selection of cubes whose specification includes the variable corresponding to the line being targeted is prohibited (continuing with the above example, any cube that includes literal \textit{a} or \textit{a} cannot be selected). The minterms corresponding to these prohibited cubes will be satisfied during the second stage of the synthesis.

The partially realized line is given a name, in this case A1, and replaces the input \textit{a} on the truth table. The above steps are repeated for each of the remaining lines in the circuit, in order of decreasing deltas. Each time, the partially realized line is given an intermediate name. In this example, the names are B2, C3, and D4. Once the function has had each line has been partially synthesized once, the first part of this synthesis method is complete.

We will not go in depth on the second portion of this synthesis method. Suffice it to say, the remaining deltas are resolved individually using two high-order Toffoli gates.
for each delta and a single garbage line that is shared for the entire circuit. The single
shared garbage line adds a reasonable, but hard to quantify, cost to the circuit (unless this
method is used to synthesize multiple sub-circuits, in which case the number of garbage
lines is equivalent to the number of sub-circuits to be synthesized and can result in an
unreasonable number of total lines). The two high order gates per unrealized minterm can
result in an incredibly expensive circuit, especially if there are a large number of inputs
because then each high order gate will be very expensive.

6.2 Improvements

In this thesis we propose several improvements to the Alhagi method. The first is
to use the deltas from the original algorithm to more clearly establish the minimization
problem. Secondly we will introduce the concept of a folded k-map, which will allow us
to easily select a good covering. Third, we will show how these improvements can be
further compounded by using PSE gates and PSE synthesis during the sub-circuit
synthesis stages.

6.2.1 Delta Management

The original algorithm requires the calculation of deltas for the purposes of
deciding which order to synthesize the lines in, but then discards this information and
works only on the actual functions that need to be synthesized. As explained in the
previous section, the deltas are computed by finding the Boolean difference of the input
and output. The deltas are important because they are the minterms of the remainder
functions. With this change, the algorithm will no longer seek to synthesize the output,
but rather reduce the remainder function as greatly as possible.
Although this seems like a minor change, this has several positive impacts and no negative ones. First, the lack of negative impacts: this calculation is already being performed, this means that no additional computation needs to be performed. Secondly, we will be able to discard the output data and work entirely with the remainder function, which will either save memory or processing power depending on whether one implemented the original algorithm with the deltas being stored or dynamically generated.

Furthermore, this change provides us with a pedagogical improvement, the function that needs to be synthesized is much clearer. In the original algorithm, one had to realize that by having the inputs and outputs on a shared line, the output had to take the form of the input variable exored with some other function that does not depend on the input variable. By implicitly removing the input variable from the output function, we remove what seems like an arbitrary restriction on the output, allowing us to use experience from ESOP synthesis more closely. Finally, and perhaps most importantly, this change allows us to implement the next improvement to this method, the folded K-map.

6.2.2 Folded K-Maps

The second algorithmic improvement that we propose here is the folded K-map. We have previously explained that when synthesizing each line in the circuit, we are only allowed to find cubes that do not depend on the variable corresponding to the line being synthesized. That is, if we are synthesizing line A, we cannot use any cube that uses variable a for the reason that a line cannot be both the target and the control of a gate.
The folded K-map transforms this restriction into visual form. This visual form behaves just like a K-map and allows designers to use prior ESOP synthesis experience to synthesize the function. The restriction that only cubes without the variable corresponding to the line being selected is implicitly enforced by the folded K-map. The way the folded K-Map works is that, as the name implies, the original K-map is folded along the axis of the variable (line) being synthesized. For example, if line A is being synthesized, then the K-map will be folded along variable a. This means that both minterms in the cube X111 will be stored in the minterm corresponding to 111 on the smaller folded K-map. This results in the effect that our function becomes four-valued. The minterms can take the value 00, 01, 10, or 11. However, we will treat both 01 and 10 as Don’t Cares because if they are selected, there will be no net change in the number of unresolved minterms in the remainder function.

Exor logic on this folded K-map behaves just like one would expect. Selecting a cell changes the contents according to the following rules. 00 becomes 11, 01 becomes 10, 10 becomes 01, and 11 becomes 00.

As an example of a folded K-map, we will use a figure from the paper by Alhagi et al. [12]. In Figure 8, it is not clear why the chosen covering is in fact the best possible covering for this subproblem (covering line A). By using a folded K-Map we will show that this is the best covering, and that it becomes intuitively obvious that this is the best covering.

Figure 8: Alhagi's Reversible Synthesis Method
First, we will use deltas, rather than the raw output to fill in the K-Map like in Figure 9. Then, we will fold the K-map along the \( a \) variable. Once the K-map is folded, we will replace minterms that contain both a 0 and a 1 with a “-”. After doing this, the K-map can now be solved using regular ESOP and PSE synthesis. In this form, we can tell from simple observation that the best covering is the cube \( d \).

![Figure 9: Synthesis with Folded K-maps](image)

During our experiments, we noticed that this formulation provided an additional advantage that we did not expect. Selecting a Don’t Care minterm (10 or 01), shuffles the remainder function in the other lines. This means that phase one of the Alhagi method can be repeated multiple times until the remainder functions can no longer be partially resolved any further. This will be further expanded on in a later section.

6.2.3 Improvements with PSE synthesis

In prior sections, we demonstrated that PSE synthesis is especially useful for functions with Don’t Cares. In the previous section, we demonstrated that the restrictions imposed by the Alhagi method can be implicitly removed by changing the representation. This representation results in a function with don’t cares, even though the original function was fully specified.
We did not perform any numerical calculations for this improvement, because the Alhagi method primarily deals with a class of functions (naturally reversible functions) that we are not concerned with. However, we have shown in this thesis that our PSE synthesis algorithm works well on incompletely specified functions, compared to other ESOP synthesizers. Since the folded K-map representations of the remainder functions behave like incompletely specified functions, our tool should produce a cheaper circuit compared to one realized with EXORCISM.

6.3 Tree Search

As stated in a previous section, we discovered an interesting property while applying the folded K-map to a problem. Selecting a cube in one line that includes a Don’t Care term will shuffle the values in other lines. This means that it is possible to reduce the remainder as much as possible in the first iteration, but changes from later minimizations to other lines can unblock previous lines. That is, changes to the remainder functions introduced by operations on other lines allow us to further remove deltas using the cheap phase one method, rather than the expensive phase two method on lines that were already minimized as much as possible with the phase one method.

However, this shuffling is not guaranteed to unblock an earlier remainder function, and it is also capable of blocking a later remainder function. For this reason we propose that a tree search should be used to determine the order that the lines should be minimized in.

There are two variants, but the rules are shared between them. First, the algorithm picks a line to partially minimize using the method from phase 1. The algorithm cannot
choose a line that has been used immediately before (cannot select the same line two times in a row) and still has remaining deltas. If the line cannot be minimized any further, another line is selected. If all lines cannot be selected, the algorithm reaches the end of one run. If the tree search implementation is depth first, there will then be a backtracking step to return to a node where another selection could have been made, continuing until all possibilities have been explored. If the implementation is breadth first, each valid selection will be taken at each node, and added to open list, following the principles of breadth first search. A detailed example is shown in Appendix B: Detailed Modified Alhagi Method

6.3.1 Variant One

The first variant of the tree search is faster, but is not a complete search. In the first variant of the search, the lines of the function are synthesized in the order specified by the original algorithm, in decreasing order of number of deltas. Once this first pass is complete, an exhaustive tree search is performed on the remainder function. This phase is then treated like phase one and a half, occurring between phase one and two of the original algorithm.

6.3.2 Variant Two

The second variant completely replaces the first phase of the original algorithm with a tree search. This is obviously much more powerful than a fixed execution order, but it is also more time consuming. Nonetheless, we have shown that phase two results in expensive gates and finding several additional reductions to the remainder functions during the first phase can provide a dramatic decrease in the final cost of the circuit.
Again, we did not perform any numeric calculations as this is outside the primary focus of this thesis.
7. Numerical Results

For the numeric results, we must first divide the results in two categories. As mentioned earlier, the PSEycic preprocessor does not provide any gains when the input function is fully specified.

Table 1: Results for completely specified functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Inputs</th>
<th>EXORCISM</th>
<th>PSEycic</th>
<th>Cost savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>9sym_d</td>
<td>9</td>
<td>20839</td>
<td>14153</td>
<td>32.1%</td>
</tr>
<tr>
<td>conf1f1</td>
<td>7</td>
<td>129</td>
<td>118</td>
<td>8.5%</td>
</tr>
<tr>
<td>conf2f2</td>
<td>7</td>
<td>56</td>
<td>56</td>
<td>0.0%</td>
</tr>
<tr>
<td>eosops1</td>
<td>5</td>
<td>68</td>
<td>46</td>
<td>32.4%</td>
</tr>
<tr>
<td>exam1_d</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>0.0%</td>
</tr>
<tr>
<td>exam3_d</td>
<td>4</td>
<td>31</td>
<td>31</td>
<td>0.0%</td>
</tr>
<tr>
<td>life_d_75</td>
<td>9</td>
<td>12682</td>
<td>8662</td>
<td>31.7%</td>
</tr>
<tr>
<td>max46_d</td>
<td>9</td>
<td>14082</td>
<td>12598</td>
<td>10.5%</td>
</tr>
<tr>
<td>newill_d</td>
<td>8</td>
<td>1199</td>
<td>1199</td>
<td>0.0%</td>
</tr>
<tr>
<td>newtag_d</td>
<td>8</td>
<td>660</td>
<td>660</td>
<td>0.0%</td>
</tr>
<tr>
<td>rd53f1</td>
<td>5</td>
<td>177</td>
<td>107</td>
<td>39.5%</td>
</tr>
<tr>
<td>rd53f2</td>
<td>5</td>
<td>72</td>
<td>72</td>
<td>0.0%</td>
</tr>
<tr>
<td>rd53f3</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0.0%</td>
</tr>
<tr>
<td>rd73f1</td>
<td>7</td>
<td>171</td>
<td>171</td>
<td>0.0%</td>
</tr>
<tr>
<td>rd73f2</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>0.0%</td>
</tr>
<tr>
<td>rd84f1</td>
<td>8</td>
<td>217</td>
<td>217</td>
<td>0.0%</td>
</tr>
<tr>
<td>rd84f2</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>0.0%</td>
</tr>
<tr>
<td>rd84f3</td>
<td>8</td>
<td>509</td>
<td>509</td>
<td>0.0%</td>
</tr>
<tr>
<td>rd84f4</td>
<td>8</td>
<td>4439</td>
<td>3531</td>
<td>20.5%</td>
</tr>
<tr>
<td>sao2f1</td>
<td>10</td>
<td>5090</td>
<td>5090</td>
<td>0.0%</td>
</tr>
<tr>
<td>sao2f2</td>
<td>10</td>
<td>8156</td>
<td>8156</td>
<td>0.0%</td>
</tr>
<tr>
<td>sao2f3</td>
<td>10</td>
<td>8285</td>
<td>4132</td>
<td>50.1%</td>
</tr>
<tr>
<td>sao2f4</td>
<td>10</td>
<td>10727</td>
<td>7958</td>
<td>25.8%</td>
</tr>
<tr>
<td>sym10_d</td>
<td>10</td>
<td>50378</td>
<td>38355</td>
<td>23.9%</td>
</tr>
<tr>
<td>t481_d</td>
<td>16</td>
<td>252</td>
<td>208</td>
<td>17.5%</td>
</tr>
<tr>
<td>xor5_d</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

For the fully specified functions, we took 26 well known PLA benchmarks plus one more function which we knew would synthesize well with PSE gates [25] and synthesized them with EXORCISM and our PSEycic postprocessor. These benchmarks are functions that have similar properties as practical functions used in the real world. Our gains were noteworthy. Several functions (especially those with few inputs), saw minimal gains. Our cherry-picked function, eosops1, saw a savings of 32.4% in Maslov cost. On average, a fully specified PLA function of at least five inputs saw an 11.7% reduction in its Maslov
cost. If we look only at the twelve functions that had savings, the savings amounted to 24.4%.

For incompletely specified functions, we created incompletely specified benchmarks. We used the same 27 benchmarks mentioned above, but created 50% and 75% specified versions of those functions. We accomplished this by decomposing the functions to their asserted and negated minterms and randomly removing 50% and 25% of the minterms respectively. These functions are all available online [25].

Our results are even more impressive for the incompletely specified functions than they are for completely specified functions, as seen in Table 2. On functions of at least 5 inputs, the variation in savings was much greater. In the worst case, we realized a small increase in cost compared to EXORCISM. However, in the best case, we found savings of 99.3%; Newtag_d_75 was synthesized by EXORCISM with a Maslov cost of 4334. Using the PSEycic tool suite, the same function was synthesized with a Maslov cost of 29. On average, we found a savings of 33% (excluding exam1_d_75, which was an extreme outlier) on incompletely specified functions over EXORCISM’s results.

Because the algorithm implemented in the preprocessor is randomized, we have run the program a number of times on the same functions to see how the results vary over various iterations. These results are available in Appendix C: Iterated Runs of Preprocessor.
<table>
<thead>
<tr>
<th>Function</th>
<th>inputs</th>
<th>EXORCISM</th>
<th>PSEycic</th>
<th>Cost savings</th>
<th>EXORCISM</th>
<th>PSEycic</th>
<th>Cost savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>9sym_d</td>
<td>9</td>
<td>12276</td>
<td>9914</td>
<td>19.2%</td>
<td>13178</td>
<td>8685</td>
<td>34.1%</td>
</tr>
<tr>
<td>conf1f1</td>
<td>7</td>
<td>1169</td>
<td>329</td>
<td>71.9%</td>
<td>1142</td>
<td>169</td>
<td>85.2%</td>
</tr>
<tr>
<td>conf2f2</td>
<td>7</td>
<td>1089</td>
<td>56</td>
<td>94.9%</td>
<td>1213</td>
<td>56</td>
<td>95.4%</td>
</tr>
<tr>
<td>eosops1</td>
<td>5</td>
<td>100</td>
<td>87</td>
<td>13.0%</td>
<td>148</td>
<td>135</td>
<td>8.8%</td>
</tr>
<tr>
<td>exam1_d</td>
<td>3</td>
<td>3</td>
<td>10</td>
<td>-233.3%</td>
<td>10</td>
<td>10</td>
<td>0.0%</td>
</tr>
<tr>
<td>exam3_d</td>
<td>4</td>
<td>31</td>
<td>31</td>
<td>0.0%</td>
<td>31</td>
<td>28</td>
<td>9.7%</td>
</tr>
<tr>
<td>life_d_75</td>
<td>9</td>
<td>8924</td>
<td>8251</td>
<td>7.5%</td>
<td>10027</td>
<td>7151</td>
<td>28.7%</td>
</tr>
<tr>
<td>max46_d</td>
<td>9</td>
<td>14734</td>
<td>11966</td>
<td>18.8%</td>
<td>15285</td>
<td>10948</td>
<td>28.4%</td>
</tr>
<tr>
<td>newill_d</td>
<td>8</td>
<td>3842</td>
<td>1276</td>
<td>66.8%</td>
<td>2763</td>
<td>632</td>
<td>77.1%</td>
</tr>
<tr>
<td>newtag_d</td>
<td>8</td>
<td>4334</td>
<td>29</td>
<td>99.3%</td>
<td>4489</td>
<td>665</td>
<td>85.2%</td>
</tr>
<tr>
<td>rd53f1</td>
<td>5</td>
<td>132</td>
<td>71</td>
<td>46.2%</td>
<td>58</td>
<td>13</td>
<td>77.6%</td>
</tr>
<tr>
<td>rd53f2</td>
<td>5</td>
<td>118</td>
<td>107</td>
<td>9.3%</td>
<td>89</td>
<td>84</td>
<td>5.6%</td>
</tr>
<tr>
<td>rd53f3</td>
<td>5</td>
<td>131</td>
<td>79</td>
<td>39.7%</td>
<td>145</td>
<td>145</td>
<td>0.0%</td>
</tr>
<tr>
<td>rd73f1</td>
<td>7</td>
<td>1289</td>
<td>1041</td>
<td>19.2%</td>
<td>1124</td>
<td>916</td>
<td>18.5%</td>
</tr>
<tr>
<td>rd73f2</td>
<td>7</td>
<td>1155</td>
<td>746</td>
<td>35.4%</td>
<td>1237</td>
<td>839</td>
<td>32.2%</td>
</tr>
<tr>
<td>rd73f3</td>
<td>7</td>
<td>1517</td>
<td>1053</td>
<td>30.6%</td>
<td>854</td>
<td>865</td>
<td>-1.3%</td>
</tr>
<tr>
<td>rd84f1</td>
<td>8</td>
<td>3756</td>
<td>2734</td>
<td>27.2%</td>
<td>4963</td>
<td>3388</td>
<td>31.7%</td>
</tr>
<tr>
<td>rd84f2</td>
<td>8</td>
<td>4112</td>
<td>2298</td>
<td>44.1%</td>
<td>3126</td>
<td>3012</td>
<td>3.6%</td>
</tr>
<tr>
<td>rd84f3</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0.0%</td>
<td>509</td>
<td>509</td>
<td>0.0%</td>
</tr>
<tr>
<td>rd84f4</td>
<td>8</td>
<td>3692</td>
<td>3772</td>
<td>-2.2%</td>
<td>3597</td>
<td>2535</td>
<td>29.5%</td>
</tr>
<tr>
<td>sao2f1</td>
<td>10</td>
<td>6632</td>
<td>7659</td>
<td>-15.5%</td>
<td>8171</td>
<td>4081</td>
<td>50.1%</td>
</tr>
<tr>
<td>sao2f2</td>
<td>10</td>
<td>10457</td>
<td>7929</td>
<td>24.2%</td>
<td>7150</td>
<td>7150</td>
<td>0.0%</td>
</tr>
<tr>
<td>sao2f3</td>
<td>10</td>
<td>23405</td>
<td>17730</td>
<td>24.2%</td>
<td>24616</td>
<td>8249</td>
<td>66.5%</td>
</tr>
<tr>
<td>sao2f4</td>
<td>10</td>
<td>17218</td>
<td>8211</td>
<td>52.3%</td>
<td>18167</td>
<td>7869</td>
<td>56.7%</td>
</tr>
<tr>
<td>sym10_d</td>
<td>10</td>
<td>40988</td>
<td>34235</td>
<td>16.5%</td>
<td>40694</td>
<td>31323</td>
<td>23.0%</td>
</tr>
<tr>
<td>t481_d</td>
<td>16</td>
<td>103279993</td>
<td>30860437</td>
<td>70.1%</td>
<td>95757499</td>
<td>58083556</td>
<td>39.3%</td>
</tr>
<tr>
<td>xor5_d</td>
<td>5</td>
<td>111</td>
<td>74</td>
<td>33.3%</td>
<td>94</td>
<td>76</td>
<td>19.1%</td>
</tr>
</tbody>
</table>
8. Multivalued Circuits

The method of synthesis with PSE gates can also be adapted for multi-valued logic. Multi-valued logic has been a subject of research for many years, but practical, commercially-viable, multiple-valued technology has not been discovered. However, quantum computers will allow us to emulate multi-valued logic [10]. Moreover, multi-valued logic is a fundamental part of machine learning because most machine learning databases contain data in multi-valued form, like the University of California in Irvine machine learning database taken from the University of Wisconsin breast cancer database [11].

Depending on whether one is working on ESOP-like or SOP-like multivalued synthesis, the approach will be different. In SOP-like multi-valued synthesis, each non-zero value is synthesized, starting from the largest and all values larger than the currently synthesized value are treated as Don’t Cares. This means that in the synthesis of ternary logic circuits, first the minterms with a value of 2 are realized. Following that, the remainder function has all the 2s replaced with Don’t Cares and the 1s are synthesized. In this style, PSE gates do not provide too much of an advantage because multi-valued PSE gates behave in an ESOP-like manner. SOP-like multivalued circuits can utilize the inhibition portion of PSE gates, but in this case, this is no different than classical inhibition.

Multivalued PSE gates are of much more importance to ESOP-like multivalued circuits. To clarify the distinction, SOP-like ternary has “+” defined as “maximum” and “*” defined as “minimum.” Therefore, 1+1=1, 1+2=2, 2+2=2, 2*0=0, 2*1=1, and 2*2=2.
On the other hand, ESOP-like multivalued logic is defined by operations that resemble XOR. The reversible ternary operations are (0-1), (0-2), (1-2), +1mod3, and -1mod3, as well as the controlled versions of those gates. +1mod3 and -1mod3, just like their names imply, increase the value on the target line by one (with 2 going to 0) and decrease the value on the target line (with 0 going to 2) respectively. The uncontrolled versions of these gates always perform the operation on the target, and the controlled version of these gates only does so if all the controls are asserted.

The operation (0-1) behaves just like an inverter in reversible logic; 1 becomes 0 and 0 becomes 1. The difference is that in ternary logic, there is also the value 2. This operation simply does not change it. If the value is 2, it remains 2 after the operation. By adding controls, we can make the ternary analog of Feynman, Toffoli, and generalized Toffoli gates. Following this pattern, (0-2) and (1-2) swap the value 0 and 2, and 1 and 2 respectively, while leaving the third value the same.

The way that PSE gates can improve Multi-valued synthesis is twofold. First, the PSEycic postprocessor can replace cancelling cubes with inhibiting cubes in the same way that it does in the binary case. This provides the same improvement that is achieved in the fully specified binary case.

Secondly, because the “(0-1)” operation doesn’t affect a value that is neither 1 nor 0, all the minterms that have been set to 2 in the previous phase will become Don’t Cares in the second phase. This means that in the second phase we must solve an incompletely specified function. In this second phase, we are able to use both the PSEycic preprocessor and the postprocessor. The preprocessor will decide which of the Don’t Care terms
should be selected and the postprocessor will ensure that all cancelling cubes are replaced with inhibiting cubes.

However, there is another type of multi-output circuit which is also important. That is one which has multi-valued input, but binary output. PSE synthesis can also be used to improve results. If we were to realize the multi-valued input, binary output function in Figure 11 in an ESOP-like manner, the opportunity to use inhibition presents itself just like in the binary case. In this particular case, we can reduce the cost of the function by using a ternary PSE gate to realize the PS implicant in $^2b$. The resulting formula for this function is: $1a^1c \oplus 2b^0a^2c$.

In all of these cases, we are showing that our PSE gates and PSEyclic synthesis algorithm can be used to improve multi-valued synthesis without any major changes. The program itself will have to be rewritten so as to work properly with multi-valued inputs and possibly outputs, but the high level concepts remain unchanged and can also be applied to this topic.
9. Conclusion

In conclusion, we have created an algorithm which greatly reduces the cost of inherently irreversible functions for reversible and quantum computing. We have found savings on average of 24.4% for fully specified functions, and 33% for incompletely specified functions when compared to EXORCISM. Also of note, the output of this program is compatible with Grover’s algorithm.

Additionally, we have shown how this algorithm can be expanded to multi-value and multi-output functions.
10. Future Work

The work from this thesis can be expanded upon in several ways. First and foremost, the algorithms presented in this thesis and implemented in the PSEyclic postprocessor and preprocessor, can be modified and integrated with EXORCISM to create a single tool. This tool will be capable of naturally handling Don’t Cares from start to finish and should produce better results than EXORCISM v4 for incompletely specified functions. Furthermore, EXORCISM can be modified to use a quantum cost function such as Maslov cost, instead of Cubes.Literals, so that its results are optimized for quantum circuits rather than classical ESOP circuits.

Additionally, the improvements to Alhagi’s method can be implemented and quantified. Lastly, PSE gates can be generalized to include more than one inhibiting term, and the postprocessor can be modified to account for this change.
References

1. Marek Perkowski, Robert Fiszer, Pawel Kerntopf, Martin Lukac “Minimization of Multi-Level Multi-Output Incompletely Specified Reversible Functions” ULSI 2012


Appendix A: Detailed Muller Transform

We present a detailed example of how the Muller transform works. We will begin with the truth table for a four input, four output functions as in Figure 12. However, this method is not limited to functions with the same number of inputs as outputs. We could of course realize this function by separating it into four K-maps and solving each of them individually. However, we would not necessarily find a solution that allows for good cube sharing coverage. The Muller transform modifies this unspecified (has many don’t cares) function that is seen in Figure 13.

*Figure 12: 4x4 Truth Table*

*Figure 13: 8x1 K-map*
Appendix B: Detailed Modified Alhagi Method

We will present an example of variant one of our modification to Alhagi’s synthesis algorithm. We will follow his algorithm in the same way until we reach the end of phase 1. That is, each line (A, B, C, and D) has had the phase one algorithm applied. We now begin the so called phase 1.5 with the following truth table in Figure 14.

![Figure 14: Function after first pass](image)

The phase 2 algorithm will result in 22 four-controlled Toffoli gates, so even relatively small additional reductions in the Hamming distance of this function can cause a large reduction in phase 2 cost.

<table>
<thead>
<tr>
<th>(a)</th>
<th>00</th>
<th>01</th>
<th>11</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

![Figure 15: Folded K-map for line B](image)

We cannot select line D again, because that was the most recent line to have been synthesized. We are free to choose between lines A, B, or C. Line A cannot be further reduced, so we move on to line B. We follow the method outlined in this thesis and find the folded K-map for line B. There is a positive minterm on this K-map, so we select it and add a Toffoli gate controlled by \(\overline{acd}\) and targeting line B. Then, we update the truth table to reflect the changes, as seen in Figure 16.
Once the next line has been reduced, we try to reduce the every other line. However, lines A, C, and D, cannot be reduced any further. It is important to note, that we must try to synthesize line A again, because it is possible that the changes than we made to line B, could have unblocked line A and allowed another reduction. Because no other line can be reduced, this is a terminal node in our tree search. From here, we can backtrack and try changing the order of minimization in the hopes that we the circuit can be reduced even more.

If we were to terminate phase 1.5 here, we would have added one additional 3-control Toffoli gate (of Maslov cost 13), and removed three 4-control Toffoli gates (totaling a Maslov cost of 87).

![Figure 16: After reducing line b again](image-url)
Appendix C: Iterated Runs of Preprocessor

Table 3 shows 30 runs of the PSEycic tool suite on function sym10_d_75 to show the variance that exists in the preprocessor. The EXORCISM solution of this problem resulted in a quantum cost of 40988. The results are sorted by savings from least savings (the only run which resulted in a cost increase) to greatest savings.

<table>
<thead>
<tr>
<th>Run</th>
<th>Cost</th>
<th>Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>41775</td>
<td>-1.9%</td>
</tr>
<tr>
<td>3</td>
<td>40338</td>
<td>1.6%</td>
</tr>
<tr>
<td>26</td>
<td>39339</td>
<td>4.0%</td>
</tr>
<tr>
<td>24</td>
<td>38896</td>
<td>5.1%</td>
</tr>
<tr>
<td>28</td>
<td>38188</td>
<td>6.8%</td>
</tr>
<tr>
<td>11</td>
<td>36940</td>
<td>9.9%</td>
</tr>
<tr>
<td>15</td>
<td>36354</td>
<td>11.3%</td>
</tr>
<tr>
<td>18</td>
<td>35989</td>
<td>12.2%</td>
</tr>
<tr>
<td>8</td>
<td>35829</td>
<td>12.6%</td>
</tr>
<tr>
<td>5</td>
<td>35588</td>
<td>13.2%</td>
</tr>
<tr>
<td>4</td>
<td>35521</td>
<td>13.3%</td>
</tr>
<tr>
<td>19</td>
<td>35447</td>
<td>13.5%</td>
</tr>
<tr>
<td>22</td>
<td>35422</td>
<td>13.6%</td>
</tr>
<tr>
<td>13</td>
<td>35173</td>
<td>14.2%</td>
</tr>
<tr>
<td>20</td>
<td>35116</td>
<td>14.3%</td>
</tr>
<tr>
<td>7</td>
<td>34783</td>
<td>15.1%</td>
</tr>
<tr>
<td>21</td>
<td>33691</td>
<td>17.8%</td>
</tr>
<tr>
<td>25</td>
<td>33344</td>
<td>18.6%</td>
</tr>
<tr>
<td>9</td>
<td>33140</td>
<td>19.1%</td>
</tr>
<tr>
<td>30</td>
<td>32992</td>
<td>19.5%</td>
</tr>
<tr>
<td>27</td>
<td>32985</td>
<td>19.5%</td>
</tr>
<tr>
<td>14</td>
<td>32821</td>
<td>19.9%</td>
</tr>
<tr>
<td>12</td>
<td>32293</td>
<td>21.2%</td>
</tr>
<tr>
<td>1</td>
<td>32253</td>
<td>21.3%</td>
</tr>
<tr>
<td>17</td>
<td>31806</td>
<td>22.4%</td>
</tr>
<tr>
<td>6</td>
<td>31240</td>
<td>23.8%</td>
</tr>
<tr>
<td>23</td>
<td>30649</td>
<td>25.2%</td>
</tr>
<tr>
<td>16</td>
<td>30186</td>
<td>26.4%</td>
</tr>
<tr>
<td>29</td>
<td>27469</td>
<td>33.0%</td>
</tr>
<tr>
<td>10</td>
<td>26075</td>
<td>36.4%</td>
</tr>
</tbody>
</table>
Table 4 shows 30 runs of the PSEycic tool suite on function max46_d_50. The EXORCISM solution of this problem resulted in a quantum cost of 15285. The results are sorted by savings from least savings to greatest savings.

<table>
<thead>
<tr>
<th>Run</th>
<th>Cost</th>
<th>Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>13508</td>
<td>11.6%</td>
</tr>
<tr>
<td>23</td>
<td>12225</td>
<td>20.0%</td>
</tr>
<tr>
<td>4</td>
<td>11972</td>
<td>21.7%</td>
</tr>
<tr>
<td>5</td>
<td>11719</td>
<td>23.3%</td>
</tr>
<tr>
<td>12</td>
<td>11719</td>
<td>23.3%</td>
</tr>
<tr>
<td>13</td>
<td>11719</td>
<td>23.3%</td>
</tr>
<tr>
<td>14</td>
<td>11719</td>
<td>23.3%</td>
</tr>
<tr>
<td>15</td>
<td>11719</td>
<td>23.3%</td>
</tr>
<tr>
<td>18</td>
<td>11719</td>
<td>23.3%</td>
</tr>
<tr>
<td>22</td>
<td>11719</td>
<td>23.3%</td>
</tr>
<tr>
<td>27</td>
<td>11719</td>
<td>23.3%</td>
</tr>
<tr>
<td>28</td>
<td>11719</td>
<td>23.3%</td>
</tr>
<tr>
<td>29</td>
<td>11719</td>
<td>23.3%</td>
</tr>
<tr>
<td>30</td>
<td>11719</td>
<td>23.3%</td>
</tr>
<tr>
<td>1</td>
<td>11716</td>
<td>23.3%</td>
</tr>
<tr>
<td>8</td>
<td>11716</td>
<td>23.3%</td>
</tr>
<tr>
<td>20</td>
<td>11716</td>
<td>23.3%</td>
</tr>
<tr>
<td>25</td>
<td>11716</td>
<td>23.3%</td>
</tr>
<tr>
<td>6</td>
<td>11591</td>
<td>24.2%</td>
</tr>
<tr>
<td>7</td>
<td>11591</td>
<td>24.2%</td>
</tr>
<tr>
<td>9</td>
<td>11591</td>
<td>24.2%</td>
</tr>
<tr>
<td>3</td>
<td>11463</td>
<td>25.0%</td>
</tr>
<tr>
<td>11</td>
<td>11463</td>
<td>25.0%</td>
</tr>
<tr>
<td>17</td>
<td>11463</td>
<td>25.0%</td>
</tr>
<tr>
<td>21</td>
<td>11463</td>
<td>25.0%</td>
</tr>
<tr>
<td>24</td>
<td>11463</td>
<td>25.0%</td>
</tr>
<tr>
<td>10</td>
<td>11204</td>
<td>26.7%</td>
</tr>
<tr>
<td>16</td>
<td>11204</td>
<td>26.7%</td>
</tr>
<tr>
<td>19</td>
<td>10948</td>
<td>28.4%</td>
</tr>
<tr>
<td>26</td>
<td>10564</td>
<td>30.9%</td>
</tr>
</tbody>
</table>
On some runs, the PSEycic preprocessor will find little to no variation. The functions newtag_d_75 and sao2f2_75, when synthesized with EXORCISM, had Maslov costs of 4334 and 10457 respectively.

Clearly, in some cases the randomness in the preprocessor creates large variability in the savings, and in some cases it produces little to no variability.

Table 5: 30 runs on newtag_d_75

<table>
<thead>
<tr>
<th>Run</th>
<th>Cost</th>
<th>Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>2</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>3</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>4</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>5</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>6</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>7</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>8</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>9</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>10</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>11</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>12</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>13</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>14</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>15</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>16</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>17</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>18</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>19</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>20</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>21</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>22</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>23</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>24</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>25</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>26</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>27</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>28</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>29</td>
<td>29</td>
<td>99.3%</td>
</tr>
<tr>
<td>30</td>
<td>29</td>
<td>99.3%</td>
</tr>
</tbody>
</table>

Table 6: 30 runs on sao2f2_75

<table>
<thead>
<tr>
<th>Run</th>
<th>Cost</th>
<th>Savings</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6632</td>
<td>36.6%</td>
</tr>
<tr>
<td>2</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>3</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>4</td>
<td>6632</td>
<td>36.6%</td>
</tr>
<tr>
<td>5</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>6</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>7</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>8</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>9</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>10</td>
<td>6632</td>
<td>36.6%</td>
</tr>
<tr>
<td>11</td>
<td>6632</td>
<td>36.6%</td>
</tr>
<tr>
<td>12</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>13</td>
<td>6632</td>
<td>36.6%</td>
</tr>
<tr>
<td>14</td>
<td>6632</td>
<td>36.6%</td>
</tr>
<tr>
<td>15</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>16</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>17</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>18</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>19</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>20</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>21</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>22</td>
<td>6632</td>
<td>36.6%</td>
</tr>
<tr>
<td>23</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>24</td>
<td>6632</td>
<td>36.6%</td>
</tr>
<tr>
<td>25</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>26</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>27</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>28</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>29</td>
<td>7659</td>
<td>26.8%</td>
</tr>
<tr>
<td>30</td>
<td>7659</td>
<td>26.8%</td>
</tr>
</tbody>
</table>
Appendix D: Source Code

```java
package eosops;

import java.io.File;
import java.io.FileNotFoundException;
import java.io.IOException;
import java.io.PrintStream;
import java.util.ArrayList;
import java.util.Collections;
import java.util.LinkedList;
import java.util.List;
import java.util.Scanner;
import java.util.SortedSet;
import java.util.TreeSet;

public class PreV2 {
    public static final String LOCATION = "DEFINE_LOCATION_HERE";

    public static void main(String[] args) throws Exception {
        Scanner scan = null;
        String nextFile = "#% EOSOPS Pre-Minimizer v2.0 %#
        String str;
        SortedSet<Cube> asserted = new TreeSet<Cube>();
        SortedSet<Cube> negated = new TreeSet<Cube>();
        int inputs;
        List<Cube> results = new LinkedList<Cube>();
        try {
            scan = new Scanner(new File(LOCATION));
            nextFile = "#% EOSOPS Pre-Minimizer v2.0 %#
            str = scan.nextLine();
            String[] tokens = str.split(" ");
            if (tokens[0].equals(".i")) {
                System.out.println("First line must be: .i #of_inputs");
                scan.close();
                return;
            } else if (tokens[0].equals(".o")) {
                System.out.println("Second line must be: .o #of_outputs");
                scan.close();
                return;
            }
            inputs = Integer.parseInt(tokens[1]);
            if (inputs > 31) {
                System.out.println("Program currently limited to 31 inputs");
            } else {
                str = scan.nextLine();
                String[] tokens = str.split(" ");
                if (tokens[0].equals(".i")) {
                    System.out.println("First line must be: .i #of_inputs");
                    scan.close();
                    return;
                }
                inputs = Integer.parseInt(tokens[1]);
                if (inputs > 31) {
                    System.out.println("Program currently limited to 31 inputs");
                } else {
                    str = scan.nextLine();
                    String[] tokens = str.split(" ");
                    if (tokens[0].equals(".o")) {
                        System.out.println("Second line must be: .o #of_outputs");
                        scan.close();
                        return;
                    }
                }
            }
        } catch (FileNotFoundException e) {
            e.printStackTrace();
            return;
        }
    }
}
```
if (Integer.parseInt(str.substring(3)) != 1) {
    System.out.println("Program only works for single output functions");
    scan.close();
    return;
}

str = scan.nextLine();
while (str.substring(0, 1).equals(".")) {
    str = scan.nextLine();
}
while (!str.substring(0, 1).equals(".")) {
    if (Integer.parseInt(str.substring(inputs + 1)) == 1) {
        asserted.addAll(Cube.cubeSet(str.substring(0, inputs)));
    } else {
        negated.add(new Cube(str.substring(0, inputs), inputs));
    }
    str = scan.nextLine();
}  
if (!str.equals(".e")) {
    System.out.println("Input file has additional information. Please inspect manually");
    scan.close();
    return;
}  
scan.close();
// End of parsing.

List<Cube> temp = new ArrayList<Cube>();
temp.addAll(asserted);
Collections.shuffle(temp);
List<Cube> unselected = new LinkedList<Cube>();
unselected.addAll(temp);

Cube core = null;
Cube intersect = null;
int compatible = 0;

while (!unselected.isEmpty()) {
    core = unselected.get(0);
    int i = 1;
    while (i < unselected.size()) {
        Cube superCube = core.superCube(unselected.get(i));
        compatible = checkCompatibility(superCube, unselected, negated);
        if (compatible >= 0) {
            if (compatible == 1) {
                \n                // Further processing...
            } else {
                \n                // Further processing...
            }
        } else {
            \n            // Further processing...
        }
        \n        i++;
    }
    \n    // Further processing...
}

// Additional code...

\n\n62
if (checkCompatibility(core, unselected, negated) == 0) {
    Cube condition = null;
    for (Cube c: negated) {
        intersect = c.intersect(core);
        if (intersect != null) {
            condition = intersect.superCube(condition); // needs to be in this order to prevent null calls.
        }
    }
    results.add(condition);
}
results.add(core);
// move the selected cube to the results list
SortedSet<Cube> retain = new TreeSet<Cube>);
retain.addAll(unselected);
// deep copy of the unselected cubes.
SortedSet<Cube> decomposed = Cube.cubeSet(core.string());
// the minterms covered by the core.
retain.retainAll(decomposed);
// the unselected minterms covered by the core.
egated.addAll(retain);
// the minterms that are now covered now are 0s
unselected.removeAll(decomposed);
// the minterms that are covered are no longer unselected.
}
nextFile += ".i "+inputs+"\n";
nextFile += ".o 1\n";
nextFile += ".p " + results.size() +"\n";
nextFile += ".type esop\n";
for(Cube c: results) {
    nextFile+= c + "\n";
}
nextFile+= ".e\n";
File out = new File(LOCATION.substring(0, LOCATION.length()-3)+"esop");
if(out.exists()) {
    out.delete();
}
try {
    out.createNewFile();
} catch (IOException e) {
    e.printStackTrace();
}
PrintStream print = null;
```java
try {
    print = new PrintStream(out);
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
print.print(nextFile);
}

private static int checkCompatibility(Cube cube, List<Cube> unselected, SortedSet<Cube> negated) {
    SortedSet<Cube> negatedIntersection = new TreeSet<Cube>();
    for (Cube c: negated) {
        Cube intersect = cube.intersect(c);
        if (intersect != null) {
            negatedIntersection.add(intersect);
        }
    }
    if (negatedIntersection.isEmpty()) {
        return 1;
    }
    Cube negatedSuper = null;
    for (Cube c: negatedIntersection) {
        negatedSuper = c.superCube(negatedSuper);
    }
    for (Cube c: unselected) {
        Cube intersect = cube.intersect(c);
        if (intersect != null) {
            return -1;
        }
    }
    return 0;
}
```
package eosops;

import java.io.*;
import java.util.*;

public class Post {
    public static final String LOCATION = "DEFINE_LOCATION_HERE";

    public static void main(String[] args) {
        long startTime = System.currentTimeMillis();
        Scanner scan = null;
        String nextFile = "";
        String str;
        List<Cube> source = new ArrayList<Cube>();
        try {
            scan = new Scanner(new File(LOCATION));
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        }
        str = scan.nextLine();
        while (scan.hasNext() && !str.substring(0, 2).equals(".")) {
            nextFile += str + 
                    
        } 
        nextFile += str + 
                    
        if (!scan.hasNext()) {
            System.out.println("Bad Input Specification");
            return;
        }
        int inputs = Integer.parseInt(str.substring(3));
        str = scan.nextLine();
        nextFile += str + 
                    
        if (Integer.parseInt(str.substring(3)) != 1) {
            System.out.println("Program only works on single output functions at the moment.");
            return;
        }

        nextFile += scan.nextLine()+"\n"

        //check for esop
        str = scan.nextLine();
        if (!str.equals(".type esop")) {
            System.out.println("Input must be of type ESOP");
            return;
        }
        nextFile += ".type eosops\n"

        //read cubes
str = scan.nextLine();
while(!str.equals(".e")) {
    source.add(new Cube(str.substring(0, str.indexOf(" ")),
                        inputs));
    str = scan.nextLine();
}
scan.close();
Collections.shuffle(source);
List<PSImplicant> eosops = new LinkedList<PSImplicant>();
for (int i=0; i < source.size(); i++) {
    for (int j=0; j < source.size(); j++) {
        if(i==j) {
            continue;
        }
        if(source.get(i).contains(source.get(j))) {
            eosops.add(new PSImplicant(source.get(i),
                                        source.get(j), true));
            if(i > j) {
                source.remove(i);
                i -=2;
                source.remove(j);
                j--;
            } else {
                source.remove(j);
                j-=2;
                source.remove(i);
                i--;
            }
        }
    }
}
for(Cube c: source) {
    nextfile += c+"\n";
}
for (PSImplicant ba: eosops) {
    nextFile += ba+"\n";
}
nextFile += ".e\n";
nextFile += System.currentTimeMillis()-startTime+"ms\n";

File out = new File(LOCATION.substring(0, LOCATION.length()-
                        4)+"eosops");
if(out.exists()) {
    out.delete();
try {
    out.createNewFile();
} catch (IOException e) {
    e.printStackTrace();
}

PrintStream print = null;
try {
    print = new PrintStream(out);
} catch (FileNotFoundException e) {
    e.printStackTrace();
}

print.print(nextFile);
package eosops;

import java.util.SortedSet;
import java.util.TreeSet;

public class Cube implements Comparable<Cube> {
    public long value;
    public int size;

    public Cube(long value, int size) {
        this.value = value;
        this.size = size;
    }

    public Cube(String cube, int size) {
        value = 0;
        this.size = size;
        char[] ca = cube.toCharArray();
        for (int i = 0; i < size; i++) {
            if (ca[i] == '-') {
                value |= (0b11 << 2*i);
            } else if (ca[i] == '1') {
                value |= (0b10 << 2*i);
            } else if (ca[i] == '0') {
                value |= (0b01 << 2*i);
            } else {
                throw new IllegalArgumentException();
            }
        }
    }

    public boolean contains(Cube other) {
        return (~value & other.value) == 0;
    }

    public String toString() {
        return Cube.string(value, size) + " 1";
    }

    public String string() {
        return Cube.string(value, size);
    }

    public static String string(long cube, int size) {
        String result = "";
        for (int i = 0; i < 2*size; i+=2) {
            long masked = (cube >> i) & 0b11;
            if (masked == 3) {
                result += "-";
            }
        }
        return result;
    }
}
```java
if (masked == 2) {
    result += "1";
} else if (masked == 1) {
    result += "0";
} else {
    throw new IllegalArgumentException();
}
return result;
}

public static SortedSet<Cube> cubeSet(String inputs) {
    SortedSet<Cube> result = new TreeSet<Cube>();
    int index = inputs.indexOf("-");
    if (index == -1) {
        result.add(new Cube(inputs, inputs.length()));
    } else {
        result.addAll(Cube.cubeSet(inputs.substring(0, index) + "0" + inputs.substring(index + 1)));
        result.addAll(Cube.cubeSet(inputs.substring(0, index) + "1" + inputs.substring(index + 1)));
    }
    return result;
}

public Cube superCube(Cube other) {
    if (other == null) {
        return this;
    }
    return new Cube(other.value | value, size);
}

public Cube intersect(Cube other) {
    Cube result = new Cube(other.value & value, size);
    if (result.isValid()) {
        return result;
    }
    return null;
}

public int compareTo(Cube other) {
    if (value > other.value) {
        return 1;
    } else if (value == other.value) {
        return 0;
    } else {
        return -1;
    }
}

private boolean isValid() {
```
for (int i = 0; i < size; i++) {
    if(((value >> (2*i)) & 3) == 0) {
        return false;
    }
}
return true;
package eosops;

public class PSImplicant {
    public long implicant;
    public long inhibition;
    public int size;

    public PSImplicant(Cube big, Cube small, boolean reduce) {
        implicant = big.value;
        inhibition = small.value;
        size = big.size;
        if(reduce) {
            for (int i = 0; i<size; i++) {
                if (((implicant >> 2*i) &0b1) != ((implicant >> 2*i+1) & 0b1)) {
                    inhibition |= 0b11 << 2*i;
                }
            }
        }

        return Cube.string(implicant, size)+" "+
               Cube.string(inhibition, size)+" 1";
    }
}