CPSC 4210 - Final Paper

R. Lowry, C. Rabl, R. Rana

Abstract

his paper explores the use of genetic programming in conjunction with a shared cube representation of reversible cascades to minimize the gate count and quantum cost of arbitrary reversible circuits. Rather than optimizing for either quantum cost or cascade length, we utilize a feature vector approach to minimize both aspects. A recent approach by Nayeem and Rice provides a foundation for the shared-cube approach to reversible logic synthesis. Due to the complexity of these operations, we expand on this foundation, combining it with a genetic algorithm to yield a solution. This approach allows us to achieve a solution more quickly than by performing a brute-force search over the problem space. We demonstrate the use of a shared cube representation from Nayeem and Rice [2011] and apply an evolutionary algorithm to find optimized representations. We consider an implementation that uses a subset of the ordering rules in Rice and Nayeem [2011] as a guideline for implementing our mutation function. We present an implementation of this approach and selected examples using the Python programming language, and compare our results to those achieved in the literature as applied to Revlib benchmarks Wille et al. [2008].

1 Introduction

Reversible computing is a field that is rapidly gaining interest among researchers due to the potential to circumvent the fundamental limitations in energy efficiency and heat loss that are being faced by traditional irreversible computing designs. It has been suggested that within 20 years we will no longer be able to achieve further increases in performance or efficiency at a chip level in irreversible circuits (`frank 2005`) It has also been long known that reversible logic can lead to circuits with much lower power dissipation (`R. Landauer, 1961`) and in theory it is possible to design a reversible circuit capable of dissipating zero energy (`Bennett 1973`). Interest in reversible logic is also growing because it has been shown to be applicable to other fields such as optical computing (`P.Picton, 1991`), low power CMOS design (`W.C. Athas & L.J. Svensson, 1994`), nanotechnology (`R.C. Merkle, 1993`), and quantum computing (`A.N Al-Rabadi, 2004`).

2 Background

2.1 Unitary Matrices

Every quantum gate can be represented by a unitary transformation (in the form of a unitary matrix) whose entries are complex variables corresponding to the complex coefficients of a given particle's wave function. Unitary transformations allow us to perform actual computations with qubits since they can be realized using technologies like NMR, for instance: a qubit in an NMR machine undergoes state changes due to a changing magnetic field. These magnetic field changes are, in turn, represented by unitary matrices (Lukac et al. [2003]). Constructing useful quantum circuits in this way is analogous to early computer programmers who "programmed" massive machines like ENIAC by physically connecting relays and vacuum tubes with wires. How far we have come since then...

Definition 1. *A* **unitary matrix** *is an $n \times n$ matrix of complex coefficients which, when multiplied by its Hermitian, gives the identity matrix. Thus, for a unitary matrix U, it is true that $(U^T)^* = U^{-1}$ where $(\cdot)^*$ denotes complex conjugation.*

In addition, every 2×2 unitary matrix can be expressed using the following product (Barenco et al. [1995]). This representation is particularly useful since it corresponds to the exact rotation matrix which can be applied to the Bloch sphere representation of a qubit:

$$\begin{bmatrix} e^{i\delta} & 0 \\ 0 & e^{i\delta} \end{bmatrix} * \begin{bmatrix} e^{\frac{i\alpha}{2}} & 0 \\ 0 & e^{\frac{-i\alpha}{2}} \end{bmatrix} * \begin{bmatrix} \cos\theta/2 & \sin\theta/2 \\ -\sin\theta/2 & \cos\theta/2 \end{bmatrix} \times \begin{bmatrix} e^{i\beta/2} & 0 \\ 0 & e^{-i\beta/2} \end{bmatrix}$$

In order to create useful operations out of "quantum primitives", we can compose unitary transformations in order to come up with a permutation representation of a gate or cascade of gates. We can use the "Square-Root-of-NOT" gate to construct a NOT gate, for instance:

$$\sqrt{\text{NOT}} = \frac{1+i}{2}\begin{bmatrix} 1 & -i \\ -i & 1 \end{bmatrix} \Rightarrow \sqrt{\text{NOT}} * \sqrt{\text{NOT}} = \left(\frac{1+i}{2}\right)^2 \begin{bmatrix} 1 & -i \\ -i & 1 \end{bmatrix} * \begin{bmatrix} 1 & -i \\ -i & 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

There are many other unitary matrices: more examples may be found in Lukac et al. [2003]. Gates such as the Pauli-X, Pauli-Y, Pauli-Z, are interesting examples to play around with.

2.2 Permutation Matrices

Definition 2. *A* **permutation matrix** *P is an $n \times n$ matrix created by permuting the rows of the identity matrix I_n. It is the case that $P * P^T = P^T * P = I$ and that $det(P) = 1$.*

Rather than deal with extremely large and unwieldy unitary transformations all the time, we can use permutation matrices, as described by Williams [1999]. These are a powerful tool, since an $n \times n$ permutation matrix can be used to represent a $2^n \times 2^n$ unitary operation. Additionally, since permutation matrices are sparse, they can be computed with and stored

more efficiently than full matrices. As an aside, some unitary matrices are also permutation matrices (such as the NOT gate), but the gates which are "true quantum primitives", as described in Lukac et al. [2003] are only unitary.

In brief, permutation matrices encode the rows of a circuit or gate's truth table. Given the truth table for a CNOT gate, for instance, it is quite simple to construct its permutation matrix: we begin by encoding the inputs and outputs of the gate as decimal numbers, and create a mapping between them. Then, we use this mapping to construct the permutation matrix, using the following rule:

$$P = [p_{ij}] \text{ where } p_{ij} = \begin{cases} 1 & \text{if } i = n \text{ and } j = M(n) \quad \forall n \in \mathbb{Z}_k \\ 0 & \text{otherwise} \end{cases}$$

In this case, $k = 2^w$ where w is the "width" of the gate, or the number of inputs. Since CNOT has a width of 2, that means $k = 2^2 = 4$, in this case.

a	b	a'	b'
0	0	0	0
0	1	0	1
1	0	1	1
1	1	1	0

$\rightarrow$

n	$M(n)$
0	0
1	1
2	3
3	2

$\rightarrow \quad P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$

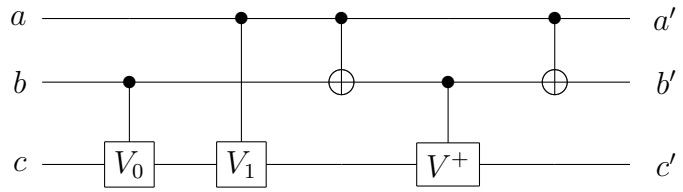
A useful property of permutation matrices is that they allow us to "compose" permutations. In order to do this, we use the following identity: $P_{\sigma\circ\pi} = P_\pi * P_\sigma$. Note that the order of the matrix multiplication matters, as matrices do not typically commute under multiplication. Having this composition operator makes it easy to represent cascades in a unique way. We can check that two cascades realize the same function if their output permutation matrices are identical. This provides circuit designers with an efficient way to "equate" cascades and determine which is "better".

2.3 Quantum Cost

Since we can represent operations on qubits using unitary transformations (which conveniently correspond to exactly one quantum operation each), we can devise a metric called "quantum cost" in order to determine whether the transformations we perform constitute an efficient synthesis of a given operation. In an NMR system, each electromagnetic pulse to which we subject a qubit has a cost: whether it is the amount of energy required to create the pulse, or the risk of the qubit decohering into a useless state (through vibrations, or other environmental perturbations), these factors may be treated as unitless "cost" variables which must be taken into account.

As quantum cost is a unitless quantity which corresponds directly to the number of unitary operations in a quantum circuit, it is a very useful metric for calculating the efficiency of an

implementation of a circuit. In order to determine the quantum cost of a gate or cascade, we need to break it down into "quantum primitives" (unitary transformations). For instance, we can break down a 3-input Toffoli gate like so:



Of course, it is not immediately obvious why this construction gives us a Toffoli gate. Note that the $\sqrt{\text{NOT}}$ gates (and their Hermitian friend) do not get activated unless their control lines are 1.

So, if we pass $a = 0$ and $b = 0$ through our gate, c remains unchanged, as do a and b. If $a = 0$ and $b = 1$, then the gate that gets applied to c will be $V_0 * V^+ = I$, which is the identity, so c will be unchanged. If $a = 1$ and $b = 0$, then the gate that gets applied to c will be $V_1 * V^+ = I$, so c will be unchanged, and finally, if $a = 1$ and $b = 1$, c will be inverted because the gate that gets applied will be $V_0 * V_1 = \text{NOT}$. And thus, we have shown that a 3-input Toffoli gate may be simulated by at least five quantum primitives, and so it has a quantum cost of 5. This result is due to DiVincenzo and Smolin [1994].

2.4 ESOP Cube-list Representations

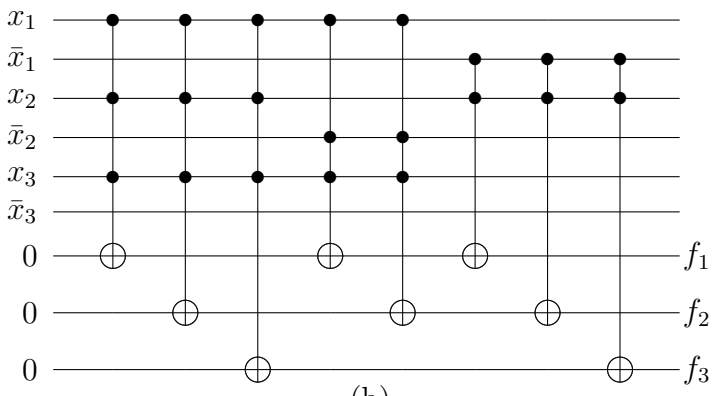
Any boolean function can be represented by an exclusive-or sum-of-products (ESOP) expression.This is particularly useful for reversible logic synthesis since there are existing algorithms for converting any ESOP expression into a cascade of reversible toffoli gates, thus generating a reversible circuit to implement your original function.

In reversible circuit design ESOP expressions are often written as a cube-list. A cube list is an $n \times m$ matrix with m, the number of product terms in the ESOP expression, and $n = i + j$ where i is the number of input variables and j is the number output variables. Each of the m rows in the matrix are the individual cubes that make up the cube list and represent one of the products from the ESOP expression.

Each cube in the list takes the general form: $x_1x_2...x_if_1f_2...f_j$, where each of the elements $x_1..x_i$ represent an input variables and each element $f_1..f_j$ represents an output variable. For each cube in the cube list a 1 is written in cube position x_k , $k \in \{1, 2, ..., i\}$ if the variable x_k is in the ESOP product for that row, a 0 is written if the negation $\bar{x}_k$ is present, and a '$-$' nis written if x_k is not present in that product term for that cube. For each element, f_p, $p \in 1, 2, ...j$, a 1 is written if that output variable contains the product represented by the input portion of the list and a 0 otherwise. See figure *draw figure 1* for an example.

x_1	x_2	x_3	f_1	f_2	f_3
1	1	1	1	1	1
1	0	1	1	1	0
0	1	–	1	1	1

(a)



(b)

Figure 1: A (a) the cube-list and (b) resulting circuit.

Given an ESOP represented as a cube list in this way Fazel et al. [2007] proposed a method of reversible logic synthesis that implements the function as a reversible circuit using only Toffoli gates. In this method an empty cascade with $2i + j$ lines is created. Two input lines are given for every input variable where one line corresponds to the variable x_k, the 2^{nd} line corresponds to its negation, $\bar{x}_k$, and the remaining j lines correspond to the output variables. For every output line f_p, a Toffoli gate is placed with its target on line f_p and a control for that target is placed on the input line x_k if there was a 1 in the cube for the corresponding variable, or on the negation line of x_k if there was a 0 in the cube for x_k, see figure (draw figure 2). This method allows a cube-list to be quickly and efficiently transformed into a reversible cascade and allows for the synthesis of rather large functions.

2.5 ESOP Cube-list Ordering Rules

Modifications to the above method with the use of inverters instead of a negation line, careful ordering of cubes to reduce the number of not gates and and sharing cubes between output lines have both reduced the number of lines and the number of gates required to implement these circuits see *[To add citations for these papers]*. In Nayeem and Rice [2011] a number of rules were proposed that manipulate the cube-list representation to create a new cube-list with the same output as the first one, depending on the rule and the state of the cube that the rule is being applied to these rules may increase or decrease the number of cubes in the list.

[To add brief description of ordering rules as these are to be used in our mutation function ...]

2.6 Genetic Algorithm

The Genetic Algorithm is a search heuristic that was introduced and investigated by John Holland (1975) and his students (DeJong, 1975). The algorithm attempts to mimic the

evolutionary process of natural selection (Mitchell, 1996) by modelling the concepts of individuals in a population, fitness and selection, crossover, and mutation that are found in the biological reproduction of organismsMitchell [1996].

The basic idea is that over a number of generations an individual that is the solution to a problem can evolve out of the latest generation of the individuals in the population. While there are a many variations on this theme, the basic algorithm is as follows:

1. Generate an initial population of 'individuals' that represent potential solutions to the problem.
2. Evaluate each individual's Fitness
3. Until a solution is found or a maximum number of generations have passed, repeat the following:
   (a) Select individuals to reproduce.
   (b) Apply crossover
   (c) Apply mutation
   (d) Evaluate each individual's fitness.

2.6.1 Representation

Representation of the individuals in a population is critical to the successful application of this algorithm and is perhaps the most challenging aspect of implementing it. Since the model is based on biology, each individual is often encoded as some type of string which the mutation and crossover functions can be applied to.

To show how a the genetic algorithm works we will use a simple example of adding two numbers to reach a specified sum. Each individual in our algorithm will be represented as bit strings consisting of 6 'genes' with each gene composed of 4 bits each. For example, an individual in our example may be represented as: 0001 1110 1001 1111 0110 0000. We choose an encoding scheme where each gene in the individual encodes for either a number or an operator. We'll let the genes 0000 to 1101 represent the numbers 0 through 13 and the gene 1110 and 1111 can represent addition and subtraction respectively. Using this encoding scheme, our individual would represent the arithmetic expression 1 + 9 - 70.

2.6.2 Fitness and Selection

The fitness function allows us to measure how close the solution represented but any specific individual is to the desired output solution. Each individual is evaluated and ranked according to its fitness, and then according to the selection criteria some are chosen for crossover and mutation. To measure the fitness we need to decode the representation of each individual and compar it against our goal result. From the example individual above, if our goal was to find an expression whose result is equal to the absolute value of the number 30, our fitness function would decode the individual and find that it has a value of -60 and then compare

it to the goal and give it a fitness, say 0.5. Once every individual is ranked according to its fitness, a number of individuals are selected for crossover and mutation according to the selection criteria that we specify.

2.6.3 Crossover

Crossover tries to model the process of sexual reproduction in nature. After parent individuals are selected from the population of the previous generation, the 'genes' of these individuals are combined to make one or more child individuals. To combine the genes a crossover point is specified in each parent individual, it is important that the crossover point occur between genes and not inside of a gene or our encoding would be broken. Consider the following two binary strings, A: 1010 1100 0001 1000 1111 0000 and B: 1011 0011 1010 1111 0000 1110. The offspring individuals will be a combinateion of the first part of A with the second part of B, or the first part of A with the second part of B. Here is an illustration of crossover using randomly chosen crossover points X:

A: 1010 1100 0001 X 1000 1111 0000
B: 1011 0011 1010 X 1111 0000 1110

We now have fragments
A1: 1010 1100 0001
A2: 1000 1111 0000
B1: 1011 0011 1010
B2: 1111 0000 1110

Combining the fragments of A and B we would produce two offspring A1B2 and B1A2:

A1B2: 1010 1100 0001 1111 0000 1110
B1A2: 1011 0011 1010 1000 1111 0000

2.6.4 Mutation

The mutation operation models mutation of the genes in organisms. For selected individuals individual 'genes' are randomly changed in some way (depending on your algorithm and encoding this mutation may be through addition, deletion, swapping of bits, or switching one gene another type of gene). For example given the individual A: 1010 1100 0001 1000 1111 0000 mutation could occur through swapping a bit to resulting individual: A_{sw}: 1010 1000 0001 1000 1111 0000, or by replacing a gene with another gene with the resulting individual: A_{sw}: 1010 1100 1110 1000 1111 0000.

2.6.5 Initial Parameter Variations

There are a number of other factors in the design of genetic algorithm that can impact the chances of finding a useful result. These factors include:

- The initial population size.
  If you have too few individuals in your initial popluation you may not be able to search enough of the search space to find an acceptable solution, conversly, if you have too many you may be searchhing more of the search space than you need to.

- The maximum number of generations.
  Specifying a maximum number of generations ensures that your search will not continue indefinitely, however if this number is too small it limits the possibility for your algorithm to find a good solution.

- The fitness threshold that you are willing to accept.
  Depending on the application you may be able to accept a solution that is close-to but does not exacly provide the desired result, whereas in other situations you can only accept a solutiong that is guaranteed to give you the desired result. A fitness threshold allows you to specify how exact of a solution you need.

2.6.6 Applications of the Genetic Algorithm to Reversible Logic Synthesis

There have been a number of applications og the genetic algorithm to logic synthesis in general, and reversible logic synthesis in particular. Lukac et al. [2003] Lukac and Perkowski [2008] Khan and Perkowski [2004] Aguirre and Coello [2003]

2.7 Parallelization

Parallelization is an approach to computational problem solving where the computation is divided into smaller subproblems and each sub-problem is computed simultaneously. The results of each sub-problem are then combined to get the final result of the whole computation.

The process of parallelizing a computation can be taken at different levels, from the bit level on a single machine to distributed computing over multiple machines (such as cluster or grid computing).

multiple threading and processes A common approach is to split a computation into into multiple threads or processes where it can be concurrently worked on by different processors on the same cpu (references)

Grid Computing add basic definition: which is each node is generally loosely coupled, generally heterogeneous.

3 Our Approach

This section will detail our approach to the genetic algorithm, including the planned use of the cube reordering rules in our mutation function and a general description of our sofware and the hardware it runs on

3.1 Revsim: Reversible Logic Simulator

3.1.1 Overview

Representing reversible circuits in a way that we could both easily simulate them and process them through a genetic algorithm was one of the first challenges we faced. There have been a number of reversible approaches to circuit synthesis using the genetic algorithm *(*references lukac, etc) but after conducting a preliminary review, none of them offered the flexibility and extensibility that we were seeking. We set out to develop a software suite that could would allow us to simulate any number of reversible circuits and let us manipulate those circuits at the gate level. We will provide a brief overview and description of the development of our software below.

3.1.2 Explanation of Simulator Design

We first developed an initial version of our circuit simulator using a functional but once we had conducted some initial experimentation and testing we decided to refactor our code using an object oriented approach.

Fig (7.1) shows the basic class structure of our software. Conceptually a gate has a number of input lines, output lines, controls and targets. The abstract Gate class formalizes this basic representation of a gate and is subclassed to create the various types of gates and their implementations.

As you can see from Fig (7.2) There are three main types of gates that we have represented in our simulator: Single Target gates, such as the the Toffoli, Multiple Target gates, like the Fredkin or Swap, and Same Target gates such as the inverter where the target and the control are on the same line.

The Cascade class is used to represent a reversible cascade of gates. It provides the primary functionality for modelling and modifying our reversible circuits, and is used other classes such as the Truth Table and Genetic Algorithm classes as described below.

The Truth Table class allows us to generate the truth tables of our reversible cascade and compare all or part of the truth table from one circuit with another. Since we have to propagate values across each gate in the cascade to generate the output values and need to generate all 2^n entries in the truth table, the process of generating the truth table is linear in the number of gates of the circuit is exponential in the number of variables

Our simulation software also has a number of classes that provide additional extensibility. For example there are input/output classes that allow us to directly read from and write to files in the revlib.org's .real file format. Our initial GA implementation required us to manually enter each goal cascade that we wanted to test against, but using these helper classes we were able to implement the ability to parse any .real file and use its target output function in the fitness calculations for our Genetic Algorithm. (more below) We also developed an experimental Shared Cube class that allows us to generate and use the shared cube representation that we were initially going to use to represent individuals in our genetic algorithm so that we could apply a number of the rule based transformations from

(ref: jackies paper on rule transformations) but after implementing it we decided on using the representation described below.

3.1.3 Description of Our Genetic Algorithm

Representation. We initially explored using a Shared Cubes representation for the individuals in used in our algorithm however while we initially thought that the transformational rules in [reference] would be useful in implementing mutation we found the difficult to implement in the context of our genetic algorithm and we not able to discover an appropriate method of implementing crossover using the shared cube representation. Instead we settled on using the cascade representation from our Cascade class which stores the list of gates used in the circuit for the individuals in our populations.

Our Algorithm initially reads in a circuit that specifies the desired output behaviour and then creates the initial population as copies of the initial circuit that have been mutated from the initial circuit up to a maximum number of mutations specified in the initial conditions.once the initial population is generated, the cycle of fitness evaluations, selection mutation and crossover repeat until one of two terminal conditions are reached, either a fitness of 1.0 or a maximum number of generations.

Fitness and Selection. Our fitness metric was a weighted measure of how closely the truth table of the current individual matched the truth table of the of the desired output behaviour. It basically performs an exhaustive comparison and so the number of comparisons needed are approximately exponential in the number of variables (linewidth) of the circuit.

Crossover. Our implementation of crossover selects the best two individuals as parents and creates two child individuals. The first child has the first half of the gates from parent 1 and the second half from parent 2 while the second child has the first half of the gates from parent 2 and the second half from parent 1, The children are then added to the population for the next iteration of the algorithm.

Mutation. The mutation function randomly selects a certain number of gates and will either replace or remove them from the cascade of the individuals it it being applied to.

Parameter Variations. Our initial parameters

3.1.4 Parallelization

We found that with larger circuits processing time became significantly longer so we looked at a few ways to parallelize our simulations in order to gather a larger dataset of results in a shorter period of time.

Grid Computing. We first spent some time adapting our software so that we could run it across the university of Lethbridge's HTcondor computing grid. Because we needed to test a number of circuits across a set of variable initial parameters one of the key challenges we faced was automating the task of creating the HTcondor job submissions. We were able to script a solution that allows us to automatically take a batch of any number reversible circuits in .real format and output a set of submission-ready executables that we could run across the HTcondor grid. This provided a significant increase in the speed we were able to generate tests and obtain results.

Parallelization of large circuits. Even using condor we still faced the problem that that the processing time for runs of our genetic algorithm with large circuits was significantly longer. In order to mitigate this we began experimenting with spitting the processing of each individual circuit into multiple processes that we can run concurrently on each of the processors of the machine running the job. Preliminary tests have been promising. *DOUBLE CHECK THIS STATEMENT!!! is is really promising??*

4 Experimental Results

This section will detail our results

5 Conclusion

This section will detail our conclusion

References

A. H. Aguirre and C. A. Coello Coello. Evolutionary synthesis of logic circuits using information theory. *Artificial Intelligence Review*, 20(3–4), December 2003.

Adriano Barenco, Charles H. Bennett, Richard Cleve, David P. DiVincenzo, Norman Margolus, Peter Shor, Tycho Sleator, John A. Smolin, and Harald Weinfurter. Elementary gates for quantum computation. *Phys. Rev. A*, 52:3457–3467, Nov 1995. doi: 10.1103/PhysRevA.52.3457. URL `http://link.aps.org/doi/10.1103/PhysRevA.52.3457`.

David P. DiVincenzo and J. Smolin. Results on two-bit gate design for quantum computers. In *Physics and Computation, 1994. PhysComp '94, Proceedings., Workshop on*, pages 14–23, Nov 1994. doi: 10.1109/PHYCMP.1994.363704.

K. Fazel, M. Thornton, and J. E. Rice. ESOP-based Toffoli gate cascade generation. In *Proceedings of the IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, pages 206–209, 2007. Aug. 22–24 2007, Victoria, BC, Canada, IEEE Press.

M. H. A. Khan and M. Perkowski. Genetic algorithm based synthesis of multi-output ternary functions using quantum cascade of generalized ternary gates. In *Proceedings of the Congress on Evolutionary Computation (CEC)*, volume 2, pages 2194–2201, 2004.

M. Lukac and M. Perkowski. Evolutionary approach to quantum symbolic logic synthesis. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, pages 3374–3380, 2008. doi: 10.1109/CEC.2008.4631254.

Martin Lukac, Marek Perkowski, Hilton Goi, Mikhail Pivtoraiko, Chung Hyo Yu, Kyusik Chung, Hyunkoo Jeech, Byung-Guk Kim, and Yong-Duk Kim. Evolutionary approach to quantum and reversible circuits synthesis. *Artif. Intell. Rev.*, 20(3-4):361–417, December 2003. ISSN 0269-2821. doi: 10.1023/B:AIRE.0000006605.86111.79. URL `http://dx.doi.org/10.1023/B:AIRE.0000006605.86111.79`.

Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press., Cambridge, MA, 1996. ISBN 9780585030944.

N. M. Nayeem and J. E. Rice. A shared-cube approach to esop-based synthesis of reversible logic. *Facta Universitatis Series: Electronics and Energetics*, 24:385–403, 2011. ISSN 0353-3670.

J.E. Rice and N. M. Nayeem. Ordering techniques for esop-based toffoli cascade generation. In *Communications, Computers and Signal Processing (PacRim), 2011 IEEE Pacific Rim Conference on*, pages 274–279, Aug 2011. doi: 10.1109/PACRIM.2011.6032905.

R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler. RevLib: An online resource for reversible functions and reversible circuits. In *Int'l Symp. on Multi-Valued Logic*, pages 220–225, 2008. RevLib is available at http://www.revlib.org.

C.P. Williams. *Quantum Computing and Quantum Communications: First NASA International Conference, QCQC '98, Palm Springs, California, USA, February 17-20, 1998, Selected Papers*. Lecture Notes in Computer Science. Springer, 1999. ISBN 9783540655145. URL `http://books.google.ca/books?id=4QuAhlwj2icC`.