A Genetic Approach to Reversible Logic Synthesis

Rio Lowry, Christopher Rabl, Hardipsihn Rana

Abstract

This paper explores the use of genetic programming applied to a representation of reversible cascades in order to reduce the gate count and quantum cost of arbitrary reversible circuits. We present a new software suite for simulating reversible circuits implemented using the Python programming language and discuss methods of parallelization. We present an implementation of this approach, selected results, and a comparison of our results against those achieved in the literature as well as the RevLib benchmarks of Wille et al. [2008].

1 Introduction

Reversible computing is a field that is rapidly gaining interest among researchers due to its potential to circumvent the fundamental limitations in energy efficiency and heat loss that are 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 using irreversible circuits (Frank [2005]). It has also been long known that reversible logic can lead to circuits with much lower power dissipation (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 (Picton [1991]), low power CMOS design (Athas and Svensson [1994]), nanotechnology (Merkle [1993]), and quantum computing (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 quantum state changes due to a changing magnetic field. These magnetic field changes are, in turn, represented by unitary matrices (Lukac et al. [2003]).

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 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].

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 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, the unitary matrices of some gates (such as NOT) are also permutation matrices, but the gates which are "true quantum primitives", as described in Lukac et al. [2003] have only unitary representations.

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 $M : \mathbb{Z}_k \rightarrow \mathbb{Z}_k$ between them, where $k = 2^w$ where w is the "width" of the gate, or the number of inputs. Then, we use this mapping to construct a 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, 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

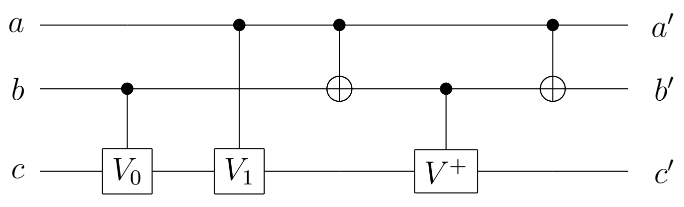
$$\rightarrow \quad P = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

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 a way to “equate” cascades.

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 analogs) 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 $V_0 * V_1 =$ NOT. Thus, a 3-input Toffoli gate may be simulated by at least 5 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

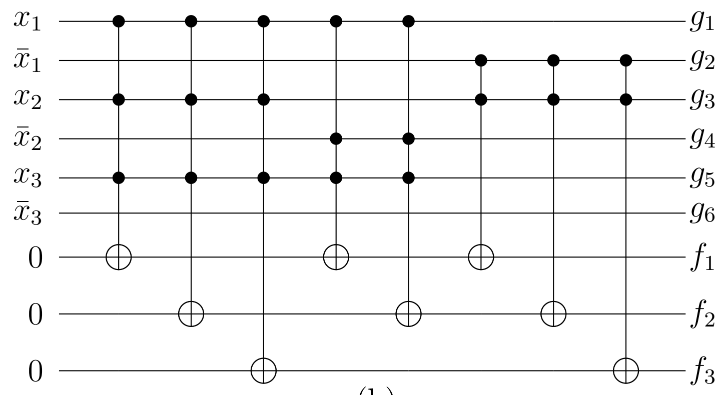
Any Boolean function can be represented by an exclusive-or sum-of-products (ESOP) expression. This is particularly useful for reversible logic synthesis since there are existing algorithms for converting any ESOP expression into a cascade of Toffoli gates, thus allowing us to generate a reversible circuit from arbitrary Boolean functions.

In reversible circuit design, ESOP expressions are often written as a cube list. A cube list is an $n \times m$ matrix, where m is 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 of output variables in the expression. Each of the m rows in the matrix are the "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 variable 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 '$-$' is written if x_k is not present in the 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 is written otherwise. See Figure 1a 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) The cube list and (b) resulting circuit.

Given an ESOP expression encoded as a cube list, Fazel et al. [2007] proposed a method for 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 next line corresponds to its negation, $\bar{x}_k$. 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 is placed on the input line x_k if there is a 1 in the cube for the corresponding variable, or if we encounter a 0 in the cube for x_k, we place the control on the negation line of x_k. see Figure 1b. This method allows a cube list to be efficiently transformed into a reversible cascade and allows for the synthesis of large functions.

2.5 ESOP Cube List Ordering Rules

Modifications to the above method have both reduced the number of lines and the number of gates required to implement these circuits. In Nayeem and Rice [2011], for example, 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.

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

Over a number of generations, fit individuals (effective solutions to a problem) evolve out of the previous generations of individuals in a population. While there exist 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, typically in a randomized fashion.
2. Evaluate the "fitness" of each individual in the population according to some metric (a fitness function).
3. Until a solution is found or a maximum number of generations have elapsed, repeat the following process:
    (a) Select individuals from the population to reproduce. This is typically done according to the heuristic provided by the fitness function.
    (b) Perform crossover between individuals in order to create a new population.

(c) Apply mutations to certain members of the population in order to expand the search space.

(d) Evaluate each individual's fitness and keep track of the top individuals since they may be candidates for the next selection.

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 a string to which the mutation and crossover functions are applied.

To show how 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. Thus, 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 either a number or an operator. We will let the genes 0000 to 1101 represent the numbers 0 through 13 and the genes 1110 and 1111 represent addition and subtraction operators, 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 by any individual is to the desired output solution. Each individual is evaluated and ranked according to its fitness, and then according to some selection criteria where candidates are chosen for crossover and mutation. To measure the fitness, we need to decode the representation of each individual and compare it against our goal result. From the example individual above, if our goal is to find an expression whose result is equal to the absolute value of the number 30, our fitness function would decode the individual's genetic representation and find that it has a value of -60. The individual would then be compared to the goal and its fitness would be evaluated, yielding a value between 0.0 and 1.0, inclusive. Once every individual is ranked according to its fitness, candidates are selected for crossover and mutation according to the selection criteria specified.

2.6.3 Crossover

Crossover attempts to model the process of sexual reproduction found in nature. After "parent" individuals are selected from the population of a generation, the genes of these individuals are combined in order to produce one or more child individuals. To combine their genes, a crossover point is specified in each parent. Consider the following two binary strings, $A = 1010\ 1100\ 0001\ 1000\ 1111\ 0000$ and $B = 1011\ 0011\ 1010\ 1111\ 0000\ 1110$. Their offspring individuals might be a combination of the first part of A with the second part of B, or the first part of A combined with the second part of B. Here is an illustration of crossover using a randomly chosen crossover point, X:

$$A = 1010\ 1100\ 0001\ \mathrm{X}\ 1000\ 1111\ 0000$$
$$B = 1011\ 0011\ 1010\ \mathrm{X}\ 1111\ 0000\ 1110$$

We now have the following fragments:

$$A_1 = 1010\ 1100\ 0001$$
$$A_2 = 1000\ 1111\ 0000$$
$$B_1 = 1011\ 0011\ 1010$$
$$B_2 = 1111\ 0000\ 1110$$

Combining the fragments of A and B we would yield two offspring, A_1B_2 and B_1A_2:

$$A_1B_2 = 1010\ 1100\ 0001\ 1111\ 0000\ 1110$$
$$B_1A_2 = 1011\ 0011\ 1010\ 1000\ 1111\ 0000$$

2.6.4 Mutation

The mutation operation models the natural mutation of genes in living organisms. For selected individuals, individual genes may be randomly changed in some way. Depending on algorithm and encoding, this mutation is usually achived through addition, deletion, or swapping of bits. For example given the individual A = 1010 1100 0001 1000 1111 0000, mutation could occur by swapping a bit. An individual resulting from this operation might be:

$$A_{sw} = 1010\ 1\underline{0}00\ 0001\ 1000\ 1111\ 0000$$

If instead we replace a gene with another gene within the individual, we might obtain:

$$A_{re} = 1010\ \underline{1100}\ 1110\ 1000\ 1111\ 0000$$

2.6.5 Initial Parameter Variations

There are a number of other factors in the design of genetic algorithms that can impact the likelihood of finding a useful (or convergent, in some cases) result. These factors include:

- **Initial population size:**
  Having less individuals in the initial popluation may not allow us to search enough of the search space to find an acceptable solution. Conversely, having too many may result in the algorithm searching more of the search space than necessary.

- **Maximum number of generations:**
  Specifying a maximum number of generations ensures that search will not continue indefinitely. However, if this number is too small it may limit the algorithm's ability to find a reasonable solution.

- **Fitness threshold:**
  Implementing a fitness threshold allows us to specify a desired "precision" for the solution. Depending on the application, an acceptable solution could be one that is very close to the ideal solution but does not exactly provide the desired result, whereas in other situations, an acceptable solution could be the unique solution that provides a desired result.

2.6.6 Applications of the Genetic Algorithm to Reversible Logic Synthesis

There have been a number of applications of the genetic algorithm to logic synthesis in general, and reversible logic synthesis in particular. Lukac et al. [2003] used genetic algorithm for reversible circuits to generate near-optimal circuits, to which they subsequently applied optimization transformations to generate optimal circuits. Lukac and Perkowski [2008] implemented symbolic synthesis within the genetic algorithm to reduce the complexity of the resulting circuits, Khan and Perkowski [2004] described a new genetic algorithm-based synthesis method for ternary quantum circuits which reduced gate count in some instances, and Aguirre and Coello [2003] proposed the use of information theory as the basis for designing a fitness function for Boolean circuit design using genetic programming.

2.7 Parallelization

Parallelization is an approach to computational problem-solving where the computation is divided into smaller sub-problems and the solution to each sub-problem is computed simultaneously. The solutions to each sub-problem are then combined to get the final result of the whole computation. The process of parallelizing a computation can be analyzed at different levels, from the bit level on a single machine to the distributed computing level over multiple machines (using cluster or grid computing).

Multiple Threading and Processes Independent computation tasks may be delegated across separate processor cores using threads or processes. When processing large cascades, we can make use of these techniques in order to reduce computation time and take full advantage of the host system's processing capabilities.

Grid Computing In cases where instances of a problem are independent of each other (such as the block computations described under "Parallelization in Revsim" in the following section), parallelization is a useful method for reducing the amount of time required to compute a solution. Not only is it possible to distribute jobs (problem instances) across the processing units of a single machine, but using distributed computing systems, jobs may be delegated across an entire network of computers which work as distinct processing units. Such computing grids are powered by distributed software such as HTCondor (Thain et al. [2005]). In typical grid scenarios, machine configurations (both in terms of hardware and software) are heterogeneous, thereby alleviating configuration-dependent hardware and software bugs.

3 Our Approach

3.1 Revsim: A Reversible Logic Simulator

3.1.1 Overview

Representing reversible circuits in a way that we could both easily simulate and process them through a genetic algorithm was one of the first challenges we faced. While there have been a number of approaches to reversible circuit synthesis using the genetic algorithm, we found that 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 capable of simulating any number of reversible circuits while allowing us to manipulate the same circuits at the gate level. In this section, we provide a brief overview and description of the development of our software.

3.1.2 Explanation of Simulator Design

We developed an initial version of our circuit simulator using a functional programming approach but once we had conducted some initial experimentation and testing, it was decided that the code should be refactored to use an object-oriented approach instead.

Figure 2 shows the basic class structure of the software. Conceptually, a reversible 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 in order to implement arbitrary reversible gates.

Figure 3 shows that there are three main types of gates that we are capable of representing in our framework: “Single Target” gates such as Toffoli, “Multiple Target” gates, like Fredkin and Swap, and “Same Target” gates such as inverters, where the target and the control exist 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 circuit designs, and is used by other classes such as the `TruthTable` and `GeneticAlgorithm` classes, as described below.

The `TruthTable` class allows us to generate the truth tables of any `Cascade` and compare all or part of the truth table from one circuit with another. Since we must propagate values across each gate in the cascade to generate the output values (and since we must do this for 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 and exponential in the number of variables ($\mathcal{O}(2^{mn})$, where m is the number of gates in the cascade, and n is the number of variable lines).

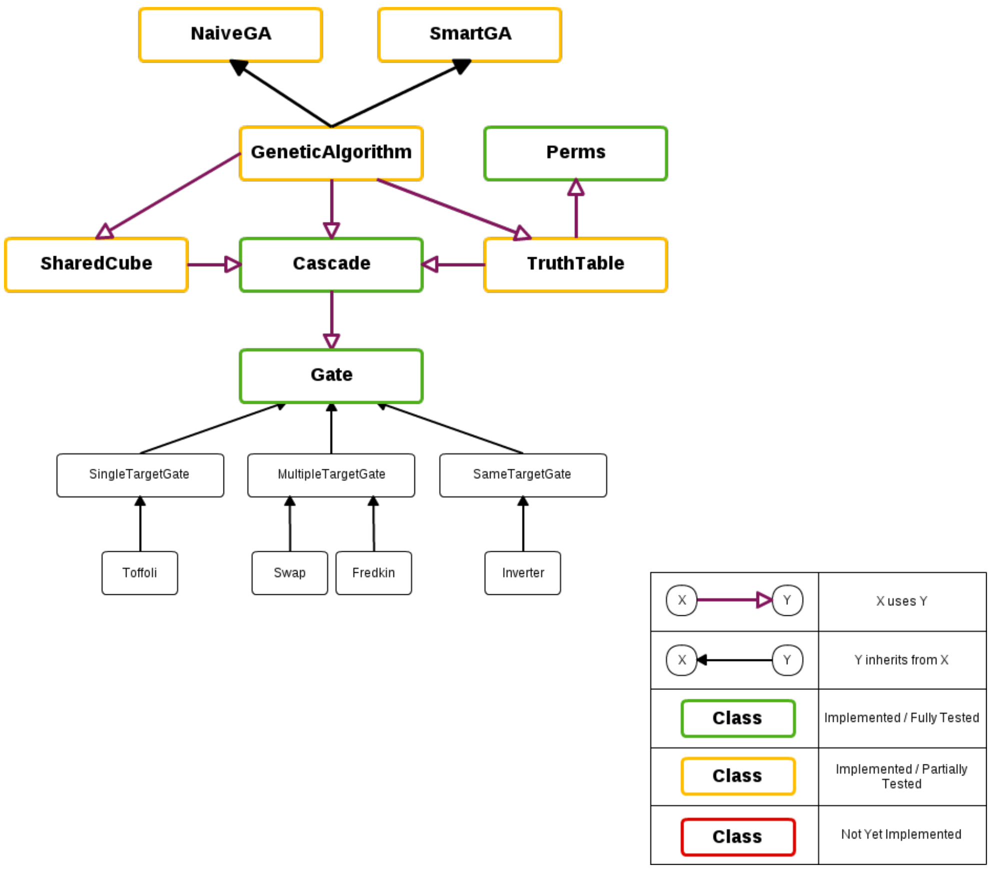


Figure 2: Class structure of Revsim

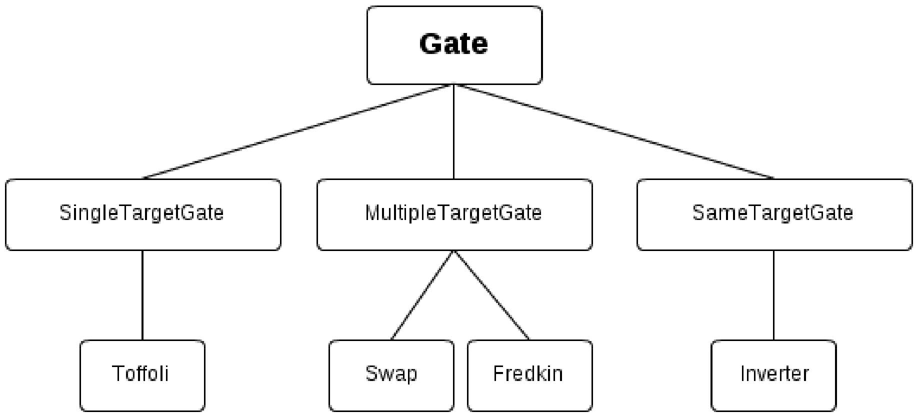


Figure 3: Gate inheritence in Revsim

Revsim also contains numerous classes designed to provide additional extensibility. For instance, the `RealIO` class allows us to read from and write to files in `www.revlib.org`'s `*.real` file format. Our initial genetic algorithm 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. We also developed an experimental `SharedCube` class that allows users to generate and use the shared cube representation that we originally intended to use for representing individuals in our genetic algorithm. The goal was to create a system that would apply a number of the rule based transformations from Nayeem and Rice [2011], but after implementing it, we decided on using the representation described below. Due to time constraints, this was not implemented in the Release Master of the software.

3.1.3 Description of Our Genetic Algorithm

Representation We initially explored using a shared cube representation for the individuals in used in our algorithm however while we thought that the transformational rules in Fazel et al. [2007] would be useful in implementing mutation, we found it difficult to implement in the context of our genetic algorithm. Additionally, we were not able to discover an appropriate method of implementing crossover using the shared cube representation. Instead, we settled on using the cascade-based representation from our `Cascade` class which stores the list of gates used in the circuits representing the individuals in our populations.

Our algorithm starts by reading 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 repeats 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 measured how closely the truth table of the current individual matched the truth table of the of the desired output behaviour. It performs an exhaustive comparison between the lines of the truth tables of the current individual versus those of the "goal" cascade. Therefore, the number of comparisons needed are approximately exponential in the number of variables (logical width) 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. This procedure is nearly identical to the method described in Section 2.6.3.

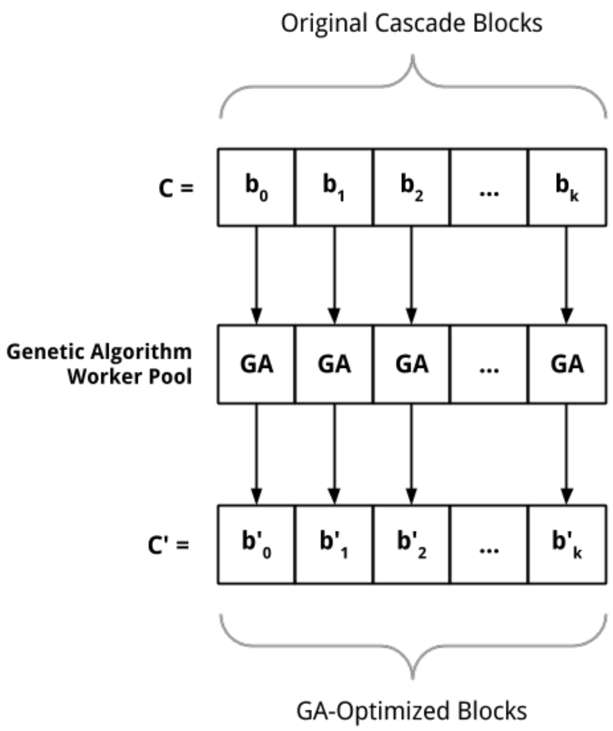


Figure 4: Parallelization in Revsim.

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

Parallelization in Revsim In order to efficiently optimize large cascades on multi-processing systems, Revsim uses a “splitting” approach, wherein cascades are partitioned into sub-cascade “blocks”, which are then optimized independently. Each block is passed through a corresponding instance of the Genetic Algorithm class, which runs in a Python sub-process. The host operating system is then free to delegate the sub-process to a particular processing unit, which allows the system to process many blocks in parallel. This process is illustrated in Figure 4.

Once a genetic algorithm sub-process completes, the resulting (optimized) cascade is returned to the main Python thread which delegates further blocks to genetic algorithm sub-processes. These steps are continued until there are no unoptimized blocks remaining in the cascade.

Since Revsim’s genetic algorithm class preserves all function outputs when performing optimizations, we can show that each input block is logically equivalent to output blocks. If a particular block cannot be optimized (in the case when the maximum generation count is exceeded), then the unoptimized block is returned.

Avoiding Global Interpreter Lock When using an interpreted programming language such as Python, it is important to keep in mind that if each thread is running in the same interpreter instance, it is possible that one thread may "lock" the interpreter, preventing the execution of other threads. Thus, rather than using threads, Revsim uses *sub-processes* in order to delegate tasks to separate processing units. Each sub-process runs its own interpreter instance, thus sidestepping Global Interpreter Lock. This advantage comes at the cost of increased interpreter overhead, but this cost is negligible when the benefits of sub-processing are considered.

Grid Computing We first spent some time adapting our software so that it could be run 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 scripted 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 may run across the HTCondor grid. This provided a significant increase in speed when it came to generating tests and obtaiing results.

4 Experimental Results

We ran our genetic algorithm against 16 circuits found in the `revlib.org` circuit library. Table 1 shows the results of the trials for all 16 circuits. From these trials, we were able to generate results from 14 of the circuits. 9 of those trials also generated at least one circuit with an improved quantum cost when compared to the initialgit circuit's `.real` file which was downloaded from RevLib. The failure to generate any results for the circuits `seq_314` and `hwb9_121` is due to the time complexity of our fitness function and the extreme logical width of those circuits.

There were a number of circuits generated which showed a significant relative improvement in quantum cost over the original comparison circuit downloaded from RevLib (see Table 2). Upon doing a detailed comparison of our best circuits with their corresponding original circuit obtained from RevLib, we noticed a discrepency between our calculation of quantum cost for gates with more than four input lines and the RevLib calculations. We realized that our quantum cost calculation (which uses the quantum cost table found at `http://webhome.cs.uvic.ca/~dmaslov/definitions.html`) did not account for the garbage lines. This discrepency only exists in circuits with gates larger than 3 inputs, and all circuits that only consisted of gates that were 3 input or smaller were accurately reported. We did not have the time to implement a revised quantum cost function which accounted for the garbage lines, however our results were self-consistent accross runs and it should only be a matter of revising our cost function and re-running our output circuits against the new cost function to compare the circuits with larger gates directly to the RevLib circuit.

Circuit	Attempted Runs	Data Generating Runs	Number of Improved Circuits	% of Runs Generating an Improved circuit	Best Reported Quantum Cost Improvement
`Max46_240`	5000	5000	5000	100.00	14038
`life_238`	5000	5000	4999	100.00	11084
`f51m_233`	5000	60	60	100.00	4472
`alu-v0_27`	1020	1020	904	88.63	7
`decod24-v3_45`	1020	1020	838	82.16	31
`gsym9`	1040	1040	665	63.94	52
`4gt11_83`	1020	1020	576	56.47	9
`gsym6`	1020	1020	536	52.55	6
`rd32_273`	1020	1020	526	51.57	92
`mod5adder_129`	5000	1277	0	0.00	0
`hwb7_62`	5000	60	0	0.00	0
`urf4_187`	1020	8	0	0.00	N/A
`hwb8_116`	5000	14	0	0.00	0
`seq_314`	3060	0	0	N/A	N/A
`hwb9_121`	1020	0	0	N/A	N/A

Table 1: Circuit Improvement Results

5 Conclusion

We have presented a method to optimize reversible logic synthesis using genetic algorithms. Additionally, we have detailed the implementation of a reversible logic synthesis framework which allowed us to implement the genetic algorithm method to a large extent.

Our algorithm shows promise, but it is by no means optimal: further investigation is required in order to ascertain its effectiveness due to discrepancies in quantum cost calculation between our framework and those presented in RevLib.

Future research in this area should address topics such as function-preserving mutation and crossover functions, as these were points of difficulty in our implementation. As our research focused more on the parallel implementation of a genetic algorithm, some more time should be spent on finding optimal cascade representations as well as on overall scalability. Currently, due to the nature of our approach, we require that a cascade's truth table be calculated before any quantum cost evaluation can take place. In practice, it would be far more efficient to utilize function-preserving transformations which would eliminate this requirement.

In conclusion, even though our algorithm delivered sub-optimal results, we were able to develop an effective and solid framework for future research on the optimization of large reversible cascades.

Circuit	Listed Cost on RevLib	Our Calculated Cost	Best Generated Circuit Cost
`Max46_240`	5444	15319	1281
`life_238`	6766	11275	191
`f51m_233`	37400	28303	32831
`alu-v0_27`	14	14	7
`decod24-v3_45`	35	35	4
`gsym9`	206	206	154
`4gt11_83`	12	12	3
`gsym6`	72	72	66
`rd32_273`	116	116	92
`mod5adder_129`	77	77	77
`hwb7_62`	2611	2839	0
`urf4_187`	160020	160020	N/A
`hwb8_116`	7015	8341	0
`seq_314`	19362	N/A	N/A
`hwb9_121`	44665	N/A	N/A

Table 2: Circuit Cost Comparisons

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.

A. N. Al-Rabadi. *Reversible Logic Synthesis: From Fundamentals to Quantum Computing.* Springer-Verlag, 2004.

W. C. Athas and L. J. Svensson. Reversible logic issues in adiabatic CMOS. In *Proceedings of the Workshop on Physics and Computation (PhysComp)*, pages 111–118, Dallas, TX, 1994.

C. H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 6:525–532, 1973.

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. P. Frank. Introduction to reversible computing: Motivation, progress, and challenges. In *Proceedings of the 2nd Conference on Computing Frontiers*, pages 385–390, 2005. May 4–6, Ischia, Italy, ACM 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.

R. Landauer. Irreversibility and heat generation in the computing process. *IBM Journal of Research and Development*, 5:183–191, 1961.

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`.

R C Merkle. Reversible electronic logic using switches. *Nanotechnology*, 4(1):21, 1993. URL `http://stacks.iop.org/0957-4484/4/i=1/a=002`.

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.

P. Picton. Optoelectronic, multivalued, conservative logic. *International Journal of Optical Computing*, 2:19–29, 1991.

Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

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`.