

An Algorithm For Quantum Circuit Optimization

Raymond Garwei Wong

Computer Science Department
California Polytechnic State University
San Luis Obispo, CA

March 2012

Completed under the supervision of Brian Granger, Ph.D., of the Physics Department
and Franz Kurfess, Ph.D., of the Computer Science Department at California Polytechnic
State University - San Luis Obispo.

© 2012

Raymond Garwei Wong
ALL RIGHTS RESERVED

Abstract

In the past 20 years, many researchers shifted their focus to developing computers based on quantum mechanical phenomenon as current computers started to plateau in performance. Some problems such as integer factorization have been shown to perform much more efficiently on a quantum computer than on its classical counterpart. However, quantum computers will continue to remain the object of theoretical research unless it can be physically manifested, and quantum circuit optimization hopes to be a useful aid in turning the theory into a reality. My project looks at a possible approach to solving the issue of circuit optimization by incorporating classical algorithms as well as touching on another possible application of machine learning. The research and implementation is incomplete, but a glance at the material shows it has potential to be fairly effective.

Acknowledgements

The foundations of SymPy's quantum physics module are given to Dr. Brian Granger and two former Cal Poly students, Addison Cugini and Matthew Curry, both of whom I had the privilege of working alongside with towards the end of their time at Cal Poly. In addition, a lot of the high level ideas dealing with circuit optimization arose out of discussions between Dr. Granger and Matt. Without their work, my project would not exist. Furthermore, Dr. Franz Kurfess provided some much needed input on the implementation of machine learning algorithms. As a student of his for three quarters, I was able to create a knowledgebase of AI material that proved useful in my efforts. I humbly thank each individual for the direct and indirect contributions they have made.

Contents

1	Introduction	1
1.1	A Very, Very Brief History	1
2	Basics of Quantum Computing	1
3	Quantum Circuit Optimization	3
3.1	Why optimize circuits?	3
3.2	Related Work	3
4	Initial Solution Proposal	3
5	Incorporating Genetic Algorithms	5
6	Revised Solution Proposal	6
6.1	Details of the Revised Solution	6
7	Implementation	8
7.1	Sympy Overview	8
7.2	Development in SymPy	8
8	Initial Results	11
9	Conclusions and Future Work	11
A	identitysearch.py	13
B	test_identitysearch.py	31
C	qcevolve.py	37
D	test_qcevolve.py	46

List of Figures

1	Quantum Circuit Example	2
2	Locally Optimal Quantum Circuit	4
3	Globally Optimal Quantum Circuit	5
4	Identity Search Example 1	7
5	Identity Search Example 2	7
6	Gate Rules	9
7	Gate Rule Graph	10
8	Identity Search Example	12

List of Tables

1	Summary of Gate Rule Operators	9
---	------------------------------------------	---

1 Introduction

Quantum computing is a relatively new field that intersects ideas from both physics and computer science. Its major feature is in the form that it performs its computations. That is, it uses quantum phenomena to store and manipulate information in place of the conventional integrated circuit. As the transistors used to construct integrated circuits reach a physical limit, quantum computers represent a viable alternative to traditional computing devices.

1.1 A Very, Very Brief History

In 1982, Richard Feynman published a paper demonstrating how classical computers inefficiently simulated quantum mechanical processes.[7] However, he hypothesized that his universal quantum simulator would not suffer the same fate.[7] A few years later, David Deutsch extended Feynman's idea of a universal quantum simulator to that of a universal quantum Turing Machine.[6] When the 1990's rolled around, the research community saw two major breakthroughs when Peter Shor and Lov Grover introduced Shor's integer factorization algorithm and Grover's database search algorithm, respectively.[11] Both algorithms easily outperform similar algorithms running on classical computers.[11] As a result, major research institutions have shifted their focus on developing this new model for information processing.

2 Basics of Quantum Computing

In quantum computing, the basic unit of information is referred to as a qubit, the quantum analog of the classical computer's bit, or cbit as it is known in the literature. Much like the cbit, a qubit $|\phi\rangle$ may assume the binary values $|1\rangle$ or $|0\rangle$. But unlike classical computing, a qubit may also appear in a superposition of those two states, as in the following:

$$|\phi\rangle = \alpha |0\rangle + \beta |1\rangle$$

In the example above, α and β represent the probability amplitudes of each value, while α^2 and β^2 represent the probabilities of measuring the state $|\phi\rangle$ in either $|0\rangle$ or $|1\rangle$. Since they represent probabilities, one condition they must meet is that they must sum to unity, which is known as the normalization condition.

One peculiarity of quantum computing, and of quantum mechanics in general, is the effect that a measurement would have on a qubit. An unobserved quantum state is said to have "collapsed" when a measurement is made on the system. The qubit is no longer in a superposition of states, and rather, it takes on one of either values $|0\rangle$ or $|1\rangle$. Furthermore, subsequent measurements on the qubit would measure the same value as when the state first "collapsed." To illustrate this concept, consider a qubit in the state $|\phi\rangle = \alpha_1 |0\rangle + \beta_1 |1\rangle$. A measurement occurs and yields $|1\rangle$. The qubit now is not simultaneously in the states $|0\rangle$

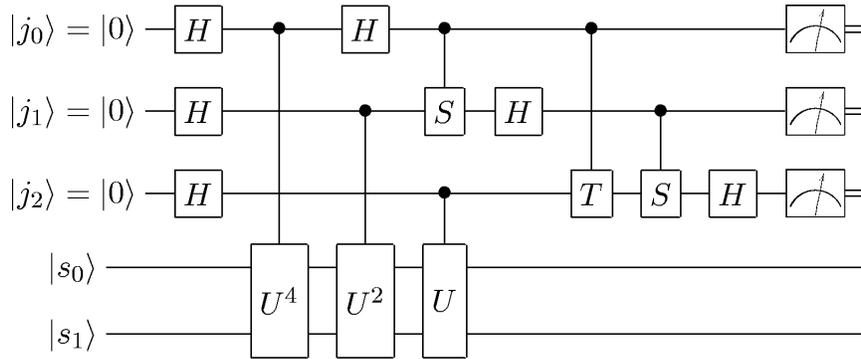


Figure 1: Example of a quantum circuit.[1]

and $|1\rangle$ anymore, and it can be more formally described by the expression $|\phi\rangle = \alpha_2 |1\rangle = |1\rangle$, where $\alpha_2 = 1$. Because of this behavior, one should restrain from measuring the state unless all quantum operations have been applied to the system; otherwise, all other information contained in the qubits will be lost.

In an n -qubit system, the total number of allowable states is 2^n , and the values range from 0 to $2^n - 1$. These allowable states are known as the computational basis states, and they form an orthogonal basis in a 2^n dimensional vector space. Like the one qubit example above, an n -qubit system may also be described as being in a superposition of one or more of its basis states, as illustrated below with a 2 qubit example:

$$\begin{aligned} \text{Superposition of 4 basis states: } |\phi\rangle &= \alpha |00\rangle + \beta |01\rangle + \gamma |10\rangle + \lambda |11\rangle \\ \text{Superposition of 2 basis states: } |\phi\rangle &= \alpha |01\rangle + \beta |10\rangle \end{aligned}$$

The most popular model for quantum computation is the quantum circuit. Like the boolean circuit found in classical computing, a quantum circuit is also defined in terms of a configuration of wires and gates. However, the gates making up a quantum circuit, called quantum gates, are distinctly different from boolean logic gates. A quantum gate acting on a system must preserve the normalization condition, and this is only achievable with the use of unitary gates. In the matrix formulation, a gate U is said to be unitary when $U^+U = U^{-1}U = I$, where U^+ is the Hermitian Adjoint of U and I is the identity matrix. The Hermitian Adjoint is obtained by taking the transpose of U and complex conjugating the values.

Some well known gates that operate on one qubit are X, Y, Z , known as the Pauli gates, H , which is the Hadamard gate, and the Phase Shift gate. Moreover, there exists gates that depend on more than one qubit. Examples of multiple qubit gates are the Swap gate, the Toffoli gate, and Controlled gates. For more detailed information about the basics of quantum computing, please see [11].

3 Quantum Circuit Optimization

In quantum computing, an algorithm may be represented by several equivalent quantum circuits that perform the same job functions as one another. However, only one or a handful of the circuits in the equivalence class contains the fewest number of gates. The goal of quantum circuit optimization, therefore, is to minimize the number of elements a quantum circuit must contain to perform the same operations.

3.1 Why optimize circuits?

During algorithmic analysis, big-O notation provides the luxury of ignoring constant values added or multiplied to a function describing the algorithm's behavior.[5] In reality, quantum gates must use a nonzero amount of time or resource to make a change on a quantum system. Given the infancy of the field, the ability to remove even a couple of gates can have substantial savings, which makes circuit optimization a highly impactful area.

3.2 Related Work

In 2008, Michal Sedlak and Martin Plesch proposed an optimization algorithm based on the properties of generalized Toffoli gates.[12] Their algorithm uses the fact that a circuit with arbitrarily defined gates can be reconstructed out of a set of universal quantum gates, which was first proven by Barenco et al.[4] The reconstruction process proposed by Barenco et al. starts by decomposing a quantum circuit into generalized Toffoli and basic gates, then the generalized Toffoli gates are further decomposed into basic quantum gates.[12]

According to Sedlak and Plesch, generalized Toffoli gates have 4 main properties, and each property either involves a merging or complex exchanging of 2 gates, depending on the neighboring conditions. Their proposed optimization takes place after the initial decomposition, when the original circuit is transformed into a circuit with generalized Toffoli and basic gates. Afterwards, Toffoli gates are shifted around repeatedly as long as the gates continue to merge. Once the reduction is complete, the second stage of Barenco's decomposition continues. Since the number of Toffoli gates is reduced before the second decomposition, the final circuit should contain fewer gates than Barenco's original method.[12]

They ran their algorithm on a couple of cases and compared their output to some known optimized quantum circuits. Although they achieved significantly better results than Barenco's approach, they were unable to obtain the fully optimized versions.[12]

4 Initial Solution Proposal

The solution outlined in this paper takes a somewhat different perspective to tackling this problem. It operates under the notion that a given circuit has many equivalent configura-

tions that span a space, and each configuration is reachable from one another through a series of gate manipulations. To demonstrate, imagine the following circuit:

$$X_0Y_0H_0H_0Z_0$$

Each of the gates X_0, Y_0, H_0 , and Z_0 carry a unique function and they all operate on the least significant qubit, the 0^{th} qubit (indicated by the subscript). It turns out the Hadamard gate, H , also happens to be Hermitian, which means the Hermitian Adjoint of the gate is equal to the gate itself, i.e. $H^+ = H$. Since all quantum gates are unitary, $H^+H = H^{-1}H = HH = I$. Therefore, the subcircuit H_0H_0 can be removed:

$$X_0Y_0H_0H_0Z_0 \rightarrow X_0Y_0Z_0$$

Circuits such as H_0H_0 are known as gate identities. For my purposes, the result of a gate identity is not limited to identity matrices. I also include circuits whose resulting matrix is the identity matrix times a scalar value, e.g. $2.718I$. The reasoning for the inclusion is simple: circuits can be renormalized such that the probability densities in the output state remain consistent relative to one another.

In an arbitrary quantum circuit, several gate identities may exist, and the simple optimization procedure is to remove each one that is found. However, a couple of issues arise with this approach. First, identities in a circuit may overlap with one another, portrayed in the following situation:

$$\begin{aligned} \text{circuit:} & \quad X_0H_0H_0CNOT_{1,0}H_0C_0(Z_1)Z_0 \\ \text{identities:} & \quad H_0CNOT_{1,0}H_0C_0(Z_1) \\ & \quad H_0H_0 \end{aligned}$$

The circuit contains two identities, and either one can be removed. An argument can be made to remove the larger of the two identities, yielding $X_0H_0Z_0$, but the situation becomes even trickier if three identities overlap with one another:

$$\begin{aligned} \text{circuit:} & \quad CNOT_{1,0}H_0R_{\pi/2}X_0X_0Y_0Z_0Z_0 \\ \text{identities:} & \quad X_0Y_0Z_0 \\ & \quad X_0X_0 \\ & \quad Z_0Z_0 \end{aligned}$$

Although the examples are contrived, the given gate identities are legitimate, and overlapping identities are entirely possible.

The second issue poses an even larger obstacle to circuit optimization. Even if none of the identities overlapped, removing them on the onset may not yield the globally optimally

$$X_0H_0Y_0H_0Z_0X_0Y_0 \rightarrow X_0H_0Y_0H_0$$

Figure 2: Although the original circuit is shortened, immediate removal of an identity may not lead to the globally optimal circuit.

$$\begin{aligned}
X_0H_0Y_0H_0Z_0X_0Y_0 &\rightarrow X_0Y_0Y_0H_0Y_0H_0Z_0X_0Y_0 \\
&\rightarrow X_0Y_0Z_0X_0Y_0 \\
&\rightarrow X_0Y_0
\end{aligned}$$

Figure 3: The globally optimal circuit. Extra steps could be taken in order to reach this solution.

solution, perhaps only a locally optimal one. For example, the circuit $X_0H_0Y_0H_0Z_0X_0Y_0$ contains the identity $Z_0X_0Y_0$, and removing the identity gives $X_0H_0Y_0H_0$. But the identity Y_0Y_0 may be inserted into the first slot between X_0 and H_0 to produce $X_0Y_0Y_0H_0Y_0H_0Z_0X_0Y_0$. The new circuit now contains a larger identity, $Y_0H_0Y_0H_0$, that may be removed, leading to the circuit $X_0Y_0Z_0X_0Y_0$. Additionally, the newer circuit has another identity, $X_0Y_0Z_0$, that may be removed to yield the globally optimal circuit, X_0Y_0 .

In light of these conundrums, a much more sophisticated path needs to be considered.

5 Incorporating Genetic Algorithms

Evolutionary approaches to optimization problems have been the subject of intense research since the beginnings of computer science. Inspired by evolutionary biology, genetic algorithms were developed as an alternative to other learning algorithms by conducting “a randomized, parallel, hill-climbing search for hypotheses that optimize a predefined fitness function.”[10] They can be applied to a variety of optimization problems in which the hypotheses are complex. One such case is genetic programming, in which the hypotheses under concern represent computer programs.[10]

Recently, genetic algorithms were employed by several research groups around the world to evolve quantum algorithms.[8] In genetic programming, the choice of program representation has a significant impact on the results.[10] Among the groups involved, the three most common representations were tree, linear, and linear-tree, a hybrid of tree and linear.[8] The tree representation was traditionally used to represent classical algorithms, but early proponents of the linear structure argue quantum algorithms are linear by nature. Thus far, the studies point in favor of linear and linear variant structures.

With this in mind, it seems genetic programming can provide a new means to optimize quantum circuits. If it wasn’t obvious from the previous examples, quantum circuits easily comply with a linear program structure. Most importantly, the genetic operators introduce a random element that can help address the two issues mentioned above. Circuits with overlapping identities may be selected for mutation or recombination, thereby removing the overlaps, and a well defined fitness function would help identify the globally optimal arrangements. The difficult now lies in defining the fitness function.

6 Revised Solution Proposal

The revised procedure can be divided into two main steps:

1. **Compile a “complete” set of gate identities.** Arbitrary quantum circuits may contain several gate identities, each of which may be removed to reduce the circuit’s gate count. To find gate identities in a circuit, a repository containing the identities must exist. The set is “complete” in the sense that the repository doesn’t contain composite identities, e.g. $X_0X_0Y_0Y_0$.
2. **Given a circuit to optimize, run the genetic programming algorithm over the circuit.** Once an initial population is created, new generations are spawned by carrying out a series of predetermined genetic operations. The goal of the fitness function is to identify circuits equivalent to the original but with the lowest gate count.

This is simply a high level overview of the proposed algorithm. A couple strategies exist for accomplishing the first, but methods for achieving the second are more or less similar to one another. For those who wish to expand upon this work, the details are left up to the individual, but I shall present my ideas in the upcoming sections for those who are interested.

6.1 Details of the Revised Solution

For the first step, I could have easily used the list created by Chris Lomont to initiate my repository, but a closer look showed that his list was quite lacking.[9] It turns out his set contained only 1 qubit gates while my set required the presence of multiple qubit gates, and so his set was insufficient for my purposes.[9] I decided to turn towards an exhaustive search approach, and found breadth first search to be particularly useful because it would find the shortest identities first before diving deeper into the search.

The search space can be represented as a tree built from a predefined set of quantum gates, with the root of the search tree being I , the Identity gate (or Identity matrix). Each unique path in the tree represents a circuit, and each identity represents a path with a finite depth. From a theoretical standpoint, the depth of the tree can go on to infinite, requiring me to include a constraint on the maximum search depth for each run.

With the genetic programming step, I can define a variety of genetic operators that manipulate the circuits in any manner I desire, but they generally fall under the categories of mutation and crossover operators. Mutations in this case can either be a gate identity removal or insertion, while crossover operations choose a point in both circuits to swap genomes (which are just subcircuits).

For the fitness function, I can define something simple that depends on two parameters. The first is the difference in gate count between the new and original circuit. The second is a summation of the errors between the matrix representations of the new and original

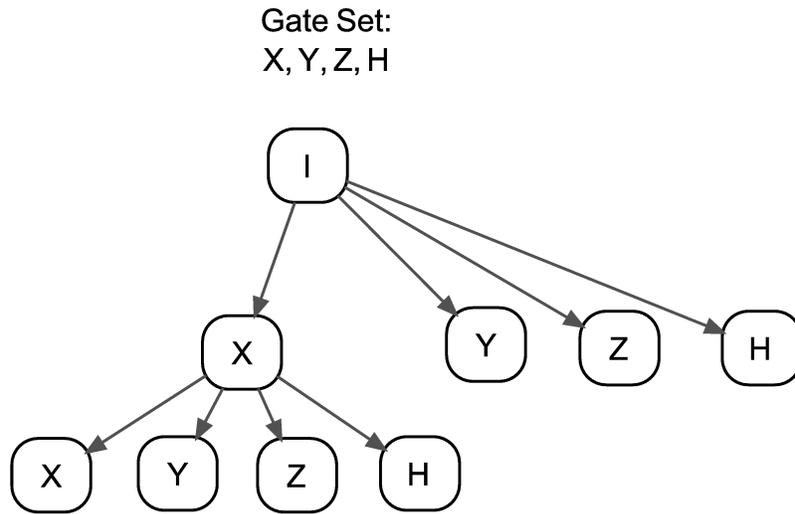


Figure 4: Expanding on the X gate with breadth first search. The expansion leads to the discovery of the identity XX .

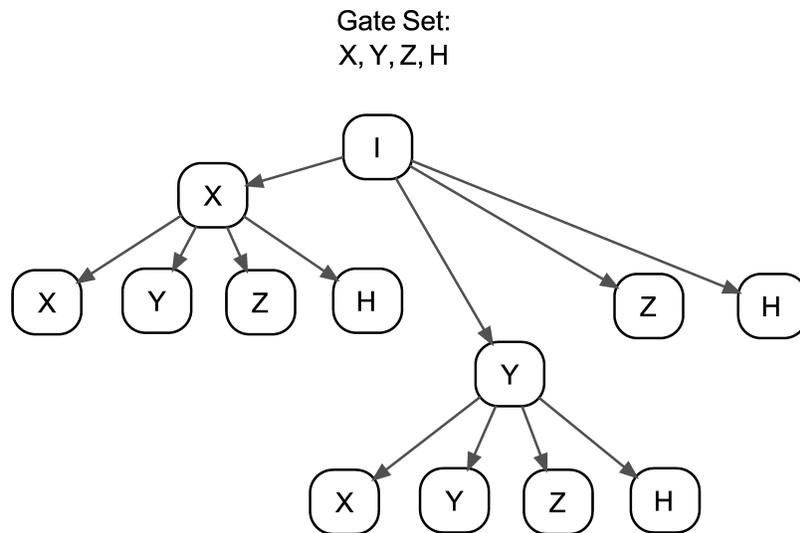


Figure 5: After expanding on X , the next step is to expand on the neighboring node, the Y gate. The expansion leads to the discovery of the identity YY .

circuit. The second parameter is necessary since crossovers may create circuits outside the equivalence class. Then the weightings on each parameter can be determined empirically through a series of experiments.

However, this function will not be suitable if I include identity insertions as a genetic operator. Given my earlier examples, insertions are definitely a valuable technique for bypassing local optimality. I would require a more sophisticated fitness function to deal with the insertions, but I have yet to come up with such a solution.

7 Implementation

For my implementation, I decided to add my work to the quantum physics module of SymPy, an open source Python library for scientific computing.

7.1 Sympy Overview

SymPy “is an open source Python library for symbolic mathematics” with the goal of becoming a “full-featured computer algebra system.” [2] Its features include algebraic manipulation, integration, differentiation, series, and a variety of other mathematical tools commonly found in commercial packages. [2] The quantum physics module of SymPy provides the capability for describing general quantum systems - especially those frequently found in textbooks - and makes it suitable for simulating quantum computations.

The most appealing aspect of SymPy is that it evaluates its expressions symbolically rather than numerically, and it proves to be a huge plus for evaluating quantum systems. N-qubit quantum systems were traditionally represented as $2^n \times 2^n$ matrices, with the size of the matrices growing exponentially with the number of qubits. As a result, simulations become a computationally intensive process. With SymPy, a complicated quantum expression could be manipulated algebraically and reduced to something much simpler, at which point a numerical answer can be computed. In other words, symbolic computation is a much less intensive process than straightforward matrix multiplication.

7.2 Development in SymPy

I wrote two major modules to include in the SymPy library: `identitysearch` and `qcevolve`. The `identitysearch` module contains an API for conducting identity searches. Among them is a function that performs a breadth first search over a predefined gate set. In addition, it contains a class definition for gate identities - conveniently called `GateIdentity` - that essentially acts as a wrapper over a Python tuple object.

An interesting property of `GateIdentity` is that it also generates a set of equivalent identities by rotating elements around the circuit. As a consequence, some permutations of an identity will be explicitly excluded from the final gate identity set. To generate the

equivalent circuits, a search of some type could be performed over the set of permutations. In general, every gate identity can be classified as an equality statement called a gate rule.

$$X_0Y_0Z_0 = I$$

$$CNOT_{1,0}H_0C_0(Z_1) = H_0$$

$$CNOT_{1,0}H_0 = H_0C_0(Z_1)$$

Figure 6: Examples of gate rules.

Starting from the original identity, I can apply four reversible gate rule operations to create a new rule. In this manner, I effectively conduct a search about a graph, where each gate rule represents a node and each edge represents an operation. The four reversible operations are LL, LR, RL, and RR. The first letter in the operator’s name indicates which side of the equality to concentrate on, while the second letter determines which gate to focus on. For example, LL means look at the left circuit’s leftmost gate. The meaning behind each of the operator’s name is summarized in Table 1.

Operator	Target Gate	Inverse	Example
LL	Left circuit, leftmost gate	RL	$\boxed{A}BCD = EFG$
LR	Left circuit, rightmost gate	RR	$ABC\boxed{D} = EFG$
RL	Right circuit, leftmost gate	LL	$ABCD = \boxed{E}FG$
RR	Right circuit, rightmost gate	LR	$ABCD = EF\boxed{G}$

Table 1: Summary of Gate Rule Operators

Once the target gate is known, the inverse of the target gate is multiplied on both circuits, thereby removing the gate on one circuit and adding it to the other. As a note, not all permutations of an identity constitute an equivalent circuit because the order of the gate arrangements matter, and not all quantum gates are necessarily Hermitian, i.e. $U^+ = U^{-1} \neq U$. It should also become apparent that not all permutations of the original identity are reachable given these four operations. Therefore, the breadth first search may locate permutations of previously found identities, but the use of gate rules help keep the set of gate identities to a much more manageable size. The search for equivalent circuits continue until I appears on either side of a gate rule, or at most k operations have been applied, where k is the number of gates in the circuit. Applying more than k operations would simply land on previously visited nodes.

The qcevolve module is intended to contain a set of genetic operators to use with PyEvolve, a “genetic algorithm framework” for Python.[3] At the present moment, the module is incomplete as it only contains a few basic mutation operators. On the other hand, it does include a couple of utility functions that aid the genetic operators in manipulating a

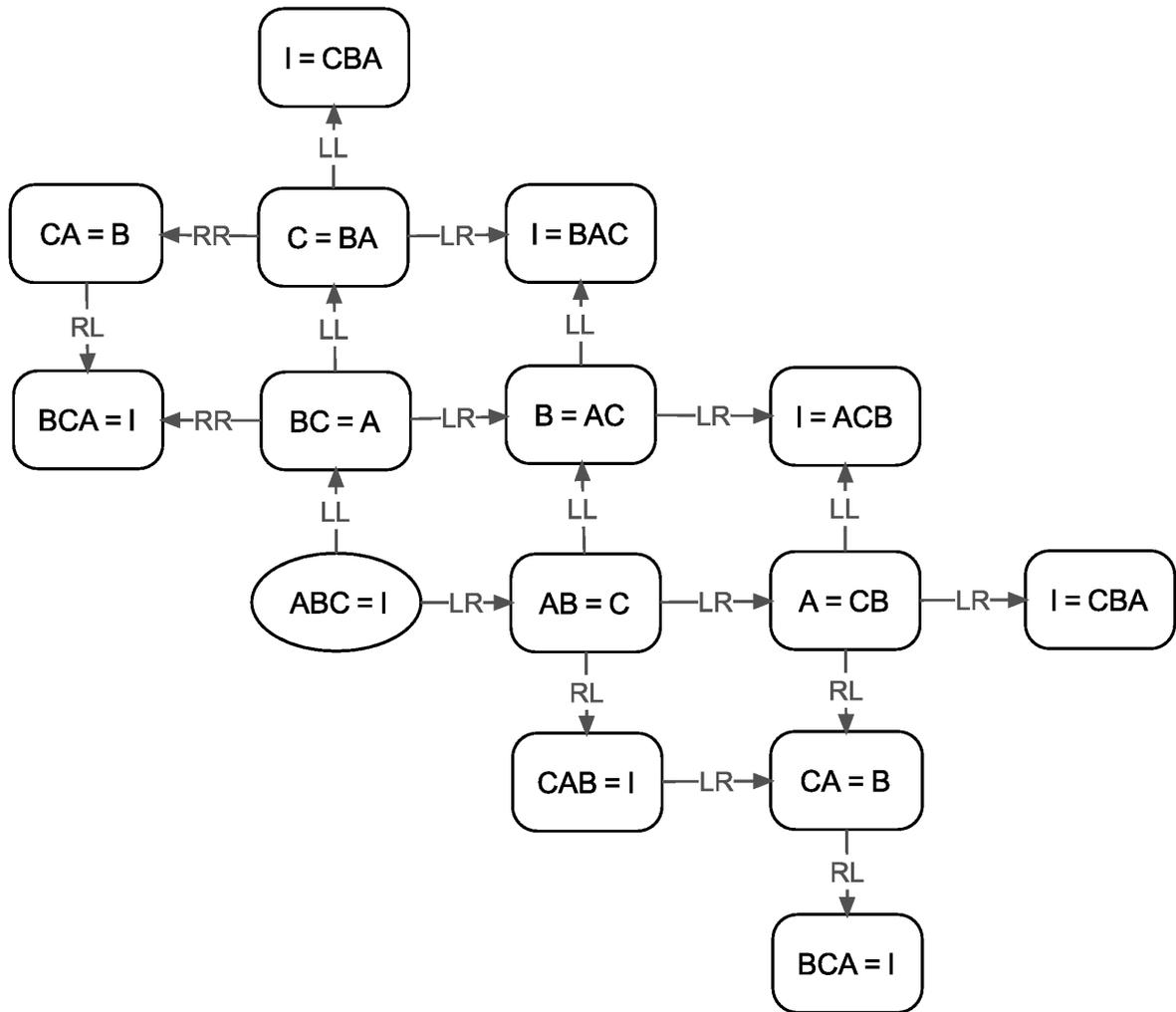


Figure 7: The four gate rule operations allow one to find permutations of the original identity (indicated in an ellipse).

circuit. One is an implementation of the Knuth-Morris-Pratt algorithm, an efficient string searching algorithm. With the KMP algorithm, I can find gate identities in an arbitrary circuit much faster than with a naive string searching approach.

As I mentioned before, I currently have two mutation operators in mind: identity removals and identity insertions. The KMP algorithm is used with the removal mutator to eliminate an identity from the circuit. The removal starts by determining which identities exist, then randomly choosing an identity to remove. The insertion operator is more trivial. It randomly selects an identity to insert and then randomly chooses a location to insert the identity.

To test my code, SymPy provides a test script that executes a file with test functions in it. In testing my mutators, I supplied a seed such that I could predict the “randomly” generated values. The testing of the rest of my code was pretty standard. Each function in a module was associated with its own test function, and each test function checked the function’s behavior under normal and edge cases.

The source code and test modules are included in the Appendix as reference.

8 Initial Results

Since the qcevolve module is incomplete, I did not have the opportunity to run my algorithm from start to finish and determine if it was effective for optimizing general quantum circuits. Initial runs of the identity search were encouraging though, as it returned several identities that were consistent with Chris Lomont’s results.[9] An example of a search is illustrated in Figure 8.

9 Conclusions and Future Work

Despite an incomplete implementation, the search by itself is quite motivating. It is too early to tell if genetic algorithms are the best approach, but any attempts to shed new light on circuit optimization would be valuable information to the community.

To gather data and analyze its results, the next step would be to complete the qcevolve module and create a fitness function that somehow takes into account insertions. A simple fitness function that only looks at the gate count would always rule against this mutation. Another feature I wish to see is the ability to generalize the qubits a circuit operates on. It basically means being able to convert integer indices into symbolic indices, i.e. $X_0 \rightarrow X_i$. By doing this, subcircuit searches will gain flexibility. A gate on one circuit won’t also have to match on the exact qubit of the other circuit’s corresponding gate; they only have to match on a qubit relative to some reference gate. Once these major tasks have been completed, it can be reviewed and integrated into the SymPy library for the rest of the world to use.

```
In [1]: from sympy.physics.quantum.gate import *
        from sympy.physics.quantum.identitysearch import *
        x = X(0); y = Y(0); z = Z(0); h = H(0);
        cnot = CNOT(1,0); cgate_z = CGate(0,Z(1));
        gate_list = [x,y,z,h,cnot,cgate_z]
```

```
In [2]: bfs_identity_search(gate_list, 2, max_depth=4)
```

```
Out[2]: set([GateIdentity(Y(0), H(0), Y(0), H(0)),
             GateIdentity(H(0), CNOT(1,0), H(0), C((0),Z(1))),
             GateIdentity(X(0), Y(0), X(0), Y(0)),
             GateIdentity(X(0), Y(0), Z(0)),
             GateIdentity(Y(0), Y(0)),
             GateIdentity(Z(0), Z(0)),
             GateIdentity(Z(0), C((0),Z(1)), Z(0), C((0),Z(1))),
             GateIdentity(X(0), X(0)),
             GateIdentity(X(0), H(0), Z(0), H(0)),
             GateIdentity(X(0), CNOT(1,0), X(0), CNOT(1,0)),
             GateIdentity(C((0),Z(1)), C((0),Z(1))),
             GateIdentity(X(0), Z(0), X(0), Z(0)),
             GateIdentity(H(0), H(0)),
             GateIdentity(Y(0), Z(0), Y(0), Z(0)),
             GateIdentity(CNOT(1,0), CNOT(1,0))])
```

Figure 8: Example of an identity search run.

A identitysearch.py

```
from collections import deque
from random import randint
import numpy

from sympy import Mul, Basic, Number
from sympy.matrices import Matrix, eye
from sympy.physics.quantum.gate import (X, Y, Z, H, S, T, CNOT,
    IdentityGate, gate_simp)
from sympy.physics.quantum.represent import represent
from sympy.physics.quantum.operator import (UnitaryOperator,
    HermitianOperator)
from sympy.physics.quantum.dagger import Dagger

__all__ = [
    'll_op',
    'lr_op',
    'rl_op',
    'rr_op',
    'generate_equivalent_ids',
    'GateIdentity',
    'is_scalar_sparse_matrix',
    'is_scalar_matrix',
    'is_degenerate',
    'is_reducible',
    'bfs_identity_search',
    'random_identity_search'
]

def ll_op(left, rite):
    """Perform a LL operation.  If LL is possible, it
    returns a 2-tuple representing the new gate rule;
    otherwise None is returned.

    Parameters
    =====
    left : tuple, Gate
        The left circuit of a gate rule expression.
    rite : tuple, Gate
```

The right circuit of a gate rule expression.

Examples

=====

```
>>> from sympy.physics.quantum.identitysearch import ll_op
>>> from sympy.physics.quantum.gate import X, Y, Z
>>> x = X(0); y = Y(0); z = Z(0)
>>> ll_op((x, y, z), ())
((Y(0), Z(0)), (X(0),))
>>> ll_op((x, y), (z,))
((Y(0),), (X(0), Z(0)))
"""
if (len(left) > 0):
    ll_gate = left[0]
    ll_gate_is_unitary = is_scalar_sparse_matrix(
        (Dagger(ll_gate), ll_gate),
        ll_gate.min_qubits,
        True)

if (len(left) > 0 and ll_gate_is_unitary):
    # Get the new left side w/o the leftmost gate
    new_left = left[1:len(left)]
    # Add the leftmost gate to the left position on the right side
    new_rite = (Dagger(ll_gate),) + rite
    # Return the new gate rule
    return (new_left, new_rite)

return None

def lr_op(left, rite):
    """Perform a LR operation.  If LR is possible, it
    returns a 2-tuple representing the new gate rule;
    otherwise None is returned.

    Parameters
    =====
    left : tuple, Gate
        The left circuit of a gate rule expression.
    rite : tuple, Gate
        The right circuit of a gate rule expression.
```

Examples

=====

```
>>> from sympy.physics.quantum.identitysearch import lr_op
>>> from sympy.physics.quantum.gate import X, Y, Z
>>> x = X(0); y = Y(0); z = Z(0)
>>> lr_op((x, y, z), ())
((X(0), Y(0)), (Z(0),))
>>> lr_op((x, y), (z,))
((X(0),), (Z(0), Y(0)))
"""
if (len(left) > 0):
    lr_gate = left[len(left)-1]
    lr_gate_is_unitary = is_scalar_sparse_matrix(
        Dagger(lr_gate), lr_gate),
        lr_gate.min_qubits,
        True)

if (len(left) > 0 and lr_gate_is_unitary):
    # Get the new left side w/o the rightmost gate
    new_left = left[0:len(left)-1]
    # Add the rightmost gate to the right position on the right side
    new_rite = rite + (Dagger(lr_gate),)
    # Return the new gate rule
    return (new_left, new_rite)

return None

def rl_op(left, rite):
    """Perform a RL operation.  If RL is possible, it
    returns a 2-tuple representing the new gate rule;
    otherwise None is returned.

    Parameters
    =====
    left : tuple, Gate
        The left circuit of a gate rule expression.
    rite : tuple, Gate
        The right circuit of a gate rule expression.
```

Examples

=====

```
>>> from sympy.physics.quantum.identitysearch import rl_op
>>> from sympy.physics.quantum.gate import X, Y, Z
>>> x = X(0); y = Y(0); z = Z(0)
>>> rl_op((x,), (y, z))
((Y(0), X(0)), (Z(0),))
>>> rl_op((x, y), (z,))
((Z(0), X(0), Y(0)), ())
"""
if (len(rite) > 0):
    rl_gate = rite[0]
    rl_gate_is_unitary = is_scalar_sparse_matrix(
        (Dagger(rl_gate), rl_gate),
        rl_gate.min_qubits,
        True)

    if (len(rite) > 0 and rl_gate_is_unitary):
        # Get the new right side w/o the leftmost gate
        new_rite = rite[1:len(rite)]
        # Add the leftmost gate to the left position on the left side
        new_left = (Dagger(rl_gate),) + left
        # Return the new gate rule
        return (new_left, new_rite)

    return None

def rr_op(left, rite):
    """Perform a RR operation.  If RR is possible, it
    returns a 2-tuple representing the new gate rule;
    otherwise None is returned.

    Parameters
    =====
    left : tuple, Gate
        The left circuit of a gate rule expression.
    rite : tuple, Gate
        The right circuit of a gate rule expression.

    Examples
```

```

=====

>>> from sympy.physics.quantum.identitysearch import rr_op
>>> from sympy.physics.quantum.gate import X, Y, Z
>>> x = X(0); y = Y(0); z = Z(0)
>>> rr_op((x, y), (z,))
((X(0), Y(0), Z(0)), ())
>>> rr_op((x,), (y, z))
((X(0), Z(0)), (Y(0),))
"""
if (len(rite) > 0):
    rr_gate = rite[len(rite)-1]
    rr_gate_is_unitary = is_scalar_sparse_matrix(
        (Dagger(rr_gate), rr_gate),
        rr_gate.min_qubits,
        True)

if (len(rite) > 0 and rr_gate_is_unitary):
    # Get the new right side w/o the rightmost gate
    new_rite = rite[0:len(rite)-1]
    # Add the rightmost gate to the right position on the right side
    new_left = left + (Dagger(rr_gate),)
    # Return the new gate rule
    return (new_left, new_rite)

return None

def generate_equivalent_ids(*gate_seq):
    """Returns a set of equivalent gate identities.

    A gate identity is a quantum circuit such that the product
    of the gates in the circuit is equal to a scalar value.
    For example, XYZ = i, where X, Y, Z are the Pauli gates and
    i is the imaginary value, is considered a gate identity.

    This function uses the four operations (LL, LR, RL, RR)
    to generate gate rules and, subsequently, to locate equivalent
    gate identities.

    A gate rule is an expression such as ABC = D or AB = CD, where
    A, B, C, and D are gates. Each value on either side of the

```

equal sign represents a circuit. The four operations allow one to find a set of equivalent circuits from a gate identity. The letters denoting the operation tell the user what activities to perform on each expression. The first letter indicates which side of the equal sign to focus on. The second letter indicates which gate to focus on given the side. Once this information is determined, the inverse of the gate is multiplied on both circuits to create a new gate rule.

For example, given the identity, $ABCD = 1$, a LL operation means look at the left value and multiply both left sides by the inverse of the leftmost gate A. If A is Hermitian, the inverse of A is still A. The resulting new rule is $BCD = A$.

The following is a summary of the four operations. Assume that in the examples, all gates are Hermitian.

```

LL : left circuit, left multiply
    ABCD = E -> AABCD = AE -> BCD = AE
LR : left circuit, right multiply
    ABCD = E -> ABCDD = ED -> ABC = ED
RL : right circuit, left multiply
    ABC = ED -> EABC = EED -> EABC = D
RR : right circuit, right multiply
    AB = CD -> ABD = CDD -> ABD = C

```

Note that all equivalent identities are reachable in n operations from the starting gate identity, where n is the number of gates in the sequence.

Parameters

=====

gate_seq : tuple, Gate

A variable length tuple of Gates whose product is equal to a scalar matrix.

Examples

=====

Find equivalent gate identities from the current circuit:

```

>>> from sympy.physics.quantum.identitysearch import \
        generate_equivalent_ids
>>> from sympy.physics.quantum.gate import X, Y, Z
>>> x = X(0); y = Y(0); z = Z(0)
>>> generate_equivalent_ids(x, x)
set([(X(0), X(0))])

>>> generate_equivalent_ids(x, y, z)
set([(X(0), Y(0), Z(0)), (X(0), Z(0), Y(0)), (Y(0), X(0), Z(0)),
      (Y(0), Z(0), X(0)), (Z(0), X(0), Y(0)), (Z(0), Y(0), X(0))])
"""

# Each item in queue is a 3-tuple:
#   i) first item is the left side of an equality
#   ii) second item is the right side of an equality
#   iii) third item is the number of operations performed
# The argument, gate_seq, will start on the left side, and
# the right side will be empty, implying the presence of an
# identity.
queue = deque()
# visited is a list of equalities that's been visited
vis = []
# A set of equivalent gate identities
eq_ids = set()
# Maximum number of operations to perform
max_ops = len(gate_seq)

queue.append((gate_seq, (), 0))
vis.append((gate_seq, ()))
eq_ids.add(gate_seq)

while (len(queue) > 0):
    rule = queue.popleft()
    left = rule[0]
    rite = rule[1]
    ops = rule[2]

    # Do a LL, if possible
    new_rule = ll_op(left, rite)
    if (new_rule is not None):

```

```

(new_left, new_rite) = new_rule

# If the left side is empty (left side is scalar)
if (len(new_left) == 0 and new_rite not in eq_ids):
    eq_ids.add(new_rite)
# If the equality has not been seen and has not reached the
# max limit on operations
elif (new_rule not in vis and ops + 1 < max_ops):
    queue.append(new_rule + (ops + 1,))

vis.append(new_rule)

# Do a LR, if possible
new_rule = lr_op(left, rite)
if (new_rule is not None):
    (new_left, new_rite) = new_rule

    if (len(new_left) == 0 and new_rite not in eq_ids):
        eq_ids.add(new_rite)
    elif (new_rule not in vis and ops + 1 < max_ops):
        queue.append(new_rule + (ops + 1,))

vis.append(new_rule)

# Do a RL, if possible
new_rule = rl_op(left, rite)
if (new_rule is not None):
    (new_left, new_rite) = new_rule

    if (len(new_rite) == 0 and new_left not in eq_ids):
        eq_ids.add(new_left)
    elif (new_rule not in vis and ops + 1 < max_ops):
        queue.append(new_rule + (ops + 1,))

vis.append(new_rule)

# Do a RR, if possible
new_rule = rr_op(left, rite)
if (new_rule is not None):
    (new_left, new_rite) = new_rule

```

```

        if (len(new_rule) == 0 and new_left not in eq_ids):
            eq_ids.add(new_left)
        elif (new_rule not in vis and ops + 1 < max_ops):
            queue.append(new_rule + (ops + 1,))

    vis.append(new_rule)

return eq_ids

class GateIdentity(Basic):
    """Wrapper class for circuits that reduce to a scalar value.

    A gate identity is a quantum circuit such that the product
    of the gates in the circuit is equal to a scalar value.
    For example, XYZ = i, where X, Y, Z are the Pauli gates and
    i is the imaginary value, is considered a gate identity.

    Parameters
    =====
    args : tuple, Gate
        A variable length tuple of Gates that form an identity.

    Examples
    =====

    Create a GateIdentity and look at its attributes:

    >>> from sympy.physics.quantum.identitysearch import \
        GateIdentity
    >>> from sympy.physics.quantum.gate import X, Y, Z
    >>> x = X(0); y = Y(0); z = Z(0)
    >>> an_identity = GateIdentity(x, y, z)
    >>> an_identity.circuit
    (X(0), Y(0), Z(0))

    >>> an_identity.eq_identities
    set([(X(0), Y(0), Z(0)), (X(0), Z(0), Y(0)), (Y(0), X(0), Z(0)),
        (Y(0), Z(0), X(0)), (Z(0), X(0), Y(0)), (Z(0), Y(0), X(0))])

    """

```

```

def __new__(cls, *args):
    # args should be a tuple - a variable length argument list
    obj = Basic.__new__(cls, *args)
    obj._eq_ids = generate_equivalent_ids(*args)

    return obj

@property
def circuit(self):
    return self.args

@property
def eq_identities(self):
    return self._eq_ids

def __str__(self):
    """Returns the string of gates in a tuple."""
    return str(self.circuit)

def is_scalar_sparse_matrix(circuit, nqubits, identity_only, eps=1e-11):
    """Checks if a given scipy.sparse matrix is a scalar matrix.

    A scalar matrix is such that  $B = bI$ , where  $B$  is the scalar
    matrix,  $b$  is some scalar multiple, and  $I$  is the identity
    matrix. A scalar matrix would have only the element  $b$  along
    it's main diagonal and zeroes elsewhere.

    Parameters
    =====
    circuit : tuple, Gate
        Sequence of quantum gates representing a quantum circuit
    nqubits : int
        Number of qubits in the circuit
    identity_only : bool
        Check for only identity matrices
    eps : number
        The tolerance value for zeroing out elements in the matrix.
        Values in the range  $[-eps, +eps]$  will be changed to a zero.

    Examples
    =====

```

```

>>> from sympy.physics.quantum.identitysearch import \
        is_scalar_sparse_matrix
>>> from sympy.physics.quantum.gate import X, Y, Z
>>> x = X(0); y = Y(0); z = Z(0)
>>> nqubits = 2
>>> circuit = (x, y, z)
>>> is_scalar_sparse_matrix(circuit, nqubits, False)
True
>>> is_scalar_sparse_matrix(circuit, nqubits, True)
False
"""

matrix = represent(Mul(*circuit), nqubits=nqubits,
                  format='scipy.sparse')

# In some cases, represent returns a 1D scalar value in place
# of a multi-dimensional scalar matrix
if (isinstance(matrix, int)):
    return matrix == 1 if identity_only else True

# If represent returns a matrix, check if the matrix is diagonal
# and if every item along the diagonal is the same
else:
    # Due to floating pointing operations, must zero out
    # elements that are "very" small in the dense matrix
    # See parameter for default value.

    # Get the ndarray version of the dense matrix
    dense_matrix = matrix.todense().getA()
    # Since complex values can't be compared, must split
    # the matrix into real and imaginary components
    # Find the real values in between -eps and eps
    bool_real = numpy.logical_and(dense_matrix.real > -eps,
                                  dense_matrix.real < eps)
    # Find the imaginary values between -eps and eps
    bool_imag = numpy.logical_and(dense_matrix.imag > -eps,
                                  dense_matrix.imag < eps)
    # Replaces values between -eps and eps with 0
    corrected_real = numpy.where(bool_real, 0.0, dense_matrix.real)
    corrected_imag = numpy.where(bool_imag, 0.0, dense_matrix.imag)

```

```

# Convert the matrix with real values into imaginary values
corrected_imag = corrected_imag * numpy.complex(1j)
# Recombine the real and imaginary components
corrected_dense = corrected_real + corrected_imag

# Check if it's diagonal
row_indices = corrected_dense.nonzero()[0]
col_indices = corrected_dense.nonzero()[1]
# Check if the rows indices and columns indices are the same
# If they match, then matrix only contains elements along diagonal
bool_indices = row_indices == col_indices
is_diagonal = bool_indices.all()

first_element = corrected_dense[0][0]
# If the first element is a zero, then can't rescale matrix
# and definitely not diagonal
if (first_element == 0.0+0.0j):
    return False

# The dimensions of the dense matrix should still
# be 2^nqubits if there are elements all along the
# the main diagonal
trace_of_corrected = (corrected_dense/first_element).trace()
expected_trace = pow(2, nqubits)
has_correct_trace = trace_of_corrected == expected_trace

# If only looking for identity matrices
# first element must be a 1
real_is_one = abs(first_element.real - 1.0) < eps
imag_is_zero = abs(first_element.imag) < eps
is_one = real_is_one and imag_is_zero
is_identity = is_one if identity_only else True
return is_diagonal and has_correct_trace and is_identity

def is_scalar_matrix(circuit, nqubits, identity_only):
    """Checks if a given circuit, in matrix form, is equivalent to
    a scalar value.

    Parameters
    =====
    circuit : tuple, Gate

```

```

    Sequence of quantum gates representing a quantum circuit
nqubits : int
    Number of qubits in the circuit
identity_only : bool
    Check for only identity matrices

Note: Used in situations when is_scalar_sparse_matrix has bugs
"""

matrix = represent(Mul(*circuit), nqubits=nqubits)

# In some cases, represent returns a 1D scalar value in place
# of a multi-dimensional scalar matrix
if (isinstance(matrix, Number)):
    return matrix == 1 if identity_only else True

# If represent returns a matrix, check if the matrix is diagonal
# and if every item along the diagonal is the same
else:
    # Added up the diagonal elements
    matrix_trace = matrix.trace()
    # Divide the trace by the first element in the matrix
    # if matrix is not required to be the identity matrix
    adjusted_matrix_trace = (matrix_trace/matrix[0]
                             if not identity_only
                             else matrix_trace)

    is_identity = matrix[0] == 1.0 if identity_only else True

    has_correct_trace = adjusted_matrix_trace == pow(2, nqubits)

    # The matrix is scalar if it's diagonal and the adjusted trace
    # value is equal to 2^nqubits
    return (matrix.is_diagonal() and
            has_correct_trace and
            is_identity)

def is_degenerate(identity_set, gate_identity):
    """Checks if a gate identity is a permutation of another identity.

Parameters

```

```

=====
identity_set : set
    A Python set with GateIdentity objects.
gate_identity : GateIdentity
    The GateIdentity to check for existence in the set.

```

Examples

```
=====
```

Check if the identity is a permutation of another identity:

```

>>> from sympy.physics.quantum.identitysearch import \
        GateIdentity, is_degenerate
>>> from sympy.physics.quantum.gate import X, Y, Z
>>> x = X(0); y = Y(0); z = Z(0)
>>> an_identity = GateIdentity(x, y, z)
>>> id_set = set([an_identity])
>>> another_id = (y, z, x)
>>> is_degenerate(id_set, another_id)
True

```

```

>>> another_id = (x, x)
>>> is_degenerate(id_set, another_id)
False

```

```
"""
```

```

# For now, just iteratively go through the set and check if the current
# gate_identity is a permutation of an identity in the set
for an_id in identity_set:
    if (gate_identity in an_id.eq_identities):
        return True
return False

```

```

def is_reducible(circuit, nqubits, begin, end):
    """Determines if a subcircuit in circuit is reducible to a scalar value.

```

Parameters

```
=====
```

```

circuit : tuple, Gate
    A tuple of Gates representing a circuit. The circuit to check
    if a gate identity is contained in a subcircuit.

```

```

nqubits : int
    The number of qubits the circuit operates on.
begin : int
    The leftmost gate in the circuit to include in a subcircuit.
end : int
    The rightmost gate in the circuit to include in a subcircuit.

```

Examples

=====

```

>>> from sympy.physics.quantum.identitysearch import \
        GateIdentity, is_reducible
>>> from sympy.physics.quantum.gate import X, Y, Z
>>> x = X(0); y = Y(0); z = Z(0)
>>> is_reducible((x, y, z), 1, 0, 3)
True

>>> is_reducible((x, y, z), 1, 1, 3)
False

>>> is_reducible((x, y, y), 1, 1, 3)
True
"""

current_circuit = ()
# Start from the gate at "end" and go down to almost the gate at "begin"
for ndx in reversed(range(begin, end)):
    next_gate = circuit[ndx]
    current_circuit = (next_gate,) + current_circuit

# If a circuit as a matrix is equivalent to a scalar value
if (is_scalar_sparse_matrix(current_circuit, nqubits, False)):
    return True

return False

def bfs_identity_search(gate_list, nqubits, **kwargs):
    """Constructs a set of gate identities from the list of possible gates.

    Performs a breadth first search over the space of gate identities.
    This allows the finding of the shortest gate identities first.

```

Parameters

=====

gate_list : list, Gate

A list of Gates from which to search for gate identities.

nqubits : int

The number of qubits the quantum circuit operates on.

max_depth : int

The longest quantum circuit to construct from gate_list.

identity_only : bool

True to search for gate identities that reduce to identity;

False to search for gate identities that reduce to a scalar.

Examples

=====

Find a list of gate identities:

```
>>> from sympy.physics.quantum.identitysearch import \
      bfs_identity_search
```

```
>>> from sympy.physics.quantum.gate import X, Y, Z, H
```

```
>>> x = X(0); y = Y(0); z = Z(0)
```

```
>>> bfs_identity_search([x], 1, max_depth=2)
```

```
set([GateIdentity(X(0), X(0))])
```

```
>>> bfs_identity_search([x, y, z], 1)
```

```
set([GateIdentity(X(0), X(0)), GateIdentity(Y(0), Y(0)),
      GateIdentity(Z(0), Z(0)), GateIdentity(X(0), Y(0), Z(0))])
```

```
>>> bfs_identity_search([x, y, z], 1, identity_only=True)
```

```
set([GateIdentity(X(0), X(0)), GateIdentity(Y(0), Y(0)),
      GateIdentity(Z(0), Z(0))])
```

```
"""
```

```
# If max depth of a path isn't given, use the length of the gate_list
```

```
if ("max_depth" in kwargs and kwargs["max_depth"] > 0):
```

```
    max_depth = kwargs["max_depth"]
```

```
else:
```

```
    max_depth = len(gate_list)
```

```
eye_only = (kwargs["identity_only"] if "identity_only" in kwargs else
            False)
```

```

# Start with an empty sequence (implicitly contains an IdentityGate)
queue = deque([()])

# Create an empty set of gate identities
ids = set()

# Begin searching for gate identities in given space.
while (len(queue) > 0):
    current_circuit = queue.popleft()

    for next_gate in gate_list:
        new_circuit = current_circuit + (next_gate,)
        #matrix_version = represent(Mul(*new_circuit), nqubits=nqubits,
        #                            format='scipy.sparse')

        # Determines if a (strict) subcircuit is a scalar matrix
        circuit_reducible = is_reducible(new_circuit, nqubits,
                                         1, len(new_circuit))

        # In many cases when the matrix is a scalar value,
        # the evaluated matrix will actually be an integer
        if (is_scalar_sparse_matrix(new_circuit, nqubits, eye_only) and
            not is_degenerate(ids, new_circuit) and
            not circuit_reducible):
            ids.add(GateIdentity(*new_circuit))

        elif (len(new_circuit) < max_depth and
              not circuit_reducible):
            queue.append(new_circuit)

return ids

def random_identity_search(gate_list, numgates, nqubits):
    """Randomly selects numgates from gate_list and checks if it is
    a gate identity.

    If the circuit is a gate identity, the circuit is returned;
    Otherwise, None is returned.
    """

```

```
gate_size = len(gate_list)
circuit = IdentityGate(0)

for i in range(numgates):
    next_gate = gate_list[randint(0, gate_size - 1)]
    circuit = initial_circuit*next_gate

matrix_version = represent(circuit, nqubits=nqubits,
                           format='scipy.sparse')

return (circuit
        if is_scalar_sparse_matrix(matrix_version, nqubits, False)
        else None)
```

B test_identitysearch.py

```
from sympy.core.symbol import Wild
from sympy.physics.quantum.gate import (X, Y, Z, H, S, T, CNOT,
    IdentityGate, CGate, Phase, gate_simp)
from sympy.physics.quantum.identitysearch import *
from sympy.physics.quantum.dagger import Dagger

def create_gate_sequence(qubit=0):
    gates = (X(qubit), Y(qubit), Z(qubit), H(qubit))
    return gates

def test_generate_equivalent_ids():
    (x, y, z, h) = create_gate_sequence()

    assert generate_equivalent_ids(x) == set([(x,)])
    assert generate_equivalent_ids(x, y) == set([(x, y), (y, x)])

    gate_seq = (x, y, z)
    gate_rules = set([(x, y, z), (y, z, x), (z, x, y), (z, y, x),
        (y, x, z), (x, z, y)])
    assert generate_equivalent_ids(*gate_seq) == gate_rules

    gate_seq = (x, y, z, h)
    gate_rules = set([(x, y, z, h), (y, z, h, x),
        (h, x, y, z), (h, z, y, x),
        (z, y, x, h), (y, x, h, z),
        (z, h, x, y), (x, h, z, y)])
    assert generate_equivalent_ids(*gate_seq) == gate_rules

    gate_seq = (x, y, x, y)
    gate_rules = set([(x, y, x, y), (y, x, y, x)])
    assert generate_equivalent_ids(*gate_seq) == gate_rules

    cgate_y = CGate((1,), y)
    gate_seq = (y, cgate_y, y, cgate_y)
    gate_rules = set([(y, cgate_y, y, cgate_y), (cgate_y, y, cgate_y, y)])
    assert generate_equivalent_ids(*gate_seq) == gate_rules

    cnot = CNOT(1,0)
```

```

cgate_z = CGate((0,), Z(1))
gate_seq = (cnot, h, cgate_z, h)
gate_rules = set([(cnot, h, cgate_z, h), (h, cgate_z, h, cnot),
                  (h, cnot, h, cgate_z), (cgate_z, h, cnot, h)])
assert generate_equivalent_ids(*gate_seq) == gate_rules

def test_is_scalar_matrix():
    numqubits = 2
    id_only = False

    id_gate = (IdentityGate(1),)
    assert is_scalar_matrix(id_gate, numqubits, id_only) == True
    assert is_scalar_sparse_matrix(id_gate, numqubits, id_only) == True

    x0 = X(0)
    xx_circuit = (x0, x0)
    assert is_scalar_matrix(xx_circuit, numqubits, id_only) == True
    assert is_scalar_sparse_matrix(xx_circuit, numqubits, id_only) == True

    x1 = X(1)
    y1 = Y(1)
    xy_circuit = (x1, y1)
    assert is_scalar_matrix(xy_circuit, numqubits, id_only) == False
    assert is_scalar_sparse_matrix(xy_circuit, numqubits, id_only) == False

    z1 = Z(1)
    xyz_circuit = (x1, y1, z1)
    assert is_scalar_matrix(xyz_circuit, numqubits, id_only) == True
    assert is_scalar_sparse_matrix(xyz_circuit, numqubits, id_only) == True

    cnot = CNOT(1,0)
    cnot_circuit = (cnot, cnot)
    assert is_scalar_matrix(cnot_circuit, numqubits, id_only) == True
    assert is_scalar_sparse_matrix(cnot_circuit, numqubits, id_only) == True

    h = H(0)
    hh_circuit = (h, h)
    assert is_scalar_matrix(hh_circuit, numqubits, id_only) == True
    assert is_scalar_sparse_matrix(hh_circuit, numqubits, id_only) == True

# NOTE:

```

```

# The elements of the sparse matrix for the following circuit
# is actually 1.0000000000000002+0.0j.
h1 = H(1)
xhzh_circuit = (x1, h1, z1, h1)
assert is_scalar_matrix(xhzh_circuit, numqubits, id_only) == True
assert is_scalar_sparse_matrix(xhzh_circuit, numqubits, id_only) == True

id_only = True
assert is_scalar_matrix(xhzh_circuit, numqubits, id_only) == True
assert is_scalar_matrix(xyz_circuit, numqubits, id_only) == False
assert is_scalar_matrix(cnot_circuit, numqubits, id_only) == True
assert is_scalar_matrix(hh_circuit, numqubits, id_only) == True

assert is_scalar_sparse_matrix(xhzh_circuit, numqubits, id_only) == True
assert is_scalar_sparse_matrix(xyz_circuit, numqubits, id_only) == False
assert is_scalar_sparse_matrix(cnot_circuit, numqubits, id_only) == True
assert is_scalar_sparse_matrix(hh_circuit, numqubits, id_only) == True

def test_is_degenerate():
    (x, y, z, h) = create_gate_sequence()

    gate_id = GateIdentity(x, y, z)
    ids = set([gate_id])

    another_id = (z, y, x)
    assert is_degenerate(ids, another_id) == True

def test_is_reducible():
    nqubits = 2
    (x, y, z, h) = create_gate_sequence()

    circuit = (x, y, y)
    assert is_reducible(circuit, nqubits, 1, 3) == True

    circuit = (x, y, x)
    assert is_reducible(circuit, nqubits, 1, 3) == False

    circuit = (x, y, y, x)
    assert is_reducible(circuit, nqubits, 0, 4) == True

    circuit = (x, y, y, x)

```

```

assert is_reducible(circuit, nqubits, 1, 3) == True

circuit = (x, y, z, y, y)
assert is_reducible(circuit, nqubits, 1, 5) == True

def test_bfs_identity_search():
    assert bfs_identity_search([], 1) == set()

    (x, y, z, h) = create_gate_sequence()

    gate_list = [x]
    id_set = set([GateIdentity(x, x)])
    assert bfs_identity_search(gate_list, 1, max_depth=2) == id_set

    # Set should not contain degenerate quantum circuits
    gate_list = [x, y, z]
    id_set = set([GateIdentity(x, x),
                  GateIdentity(y, y),
                  GateIdentity(z, z),
                  GateIdentity(x, y, z)])
    assert bfs_identity_search(gate_list, 1) == id_set

    id_set = set([GateIdentity(x, x),
                  GateIdentity(y, y),
                  GateIdentity(z, z),
                  GateIdentity(x, y, z),
                  GateIdentity(x, y, x, y),
                  GateIdentity(x, z, x, z),
                  GateIdentity(y, z, y, z)])
    assert bfs_identity_search(gate_list, 1, max_depth=4) == id_set
    assert bfs_identity_search(gate_list, 1, max_depth=5) == id_set

    gate_list = [x, y, z, h]
    id_set = set([GateIdentity(x, x),
                  GateIdentity(y, y),
                  GateIdentity(z, z),
                  GateIdentity(h, h),
                  GateIdentity(x, y, z),
                  GateIdentity(x, y, x, y),
                  GateIdentity(x, z, x, z),
                  GateIdentity(x, h, z, h),

```

```

        GateIdentity(y, z, y, z),
        GateIdentity(y, h, y, h)])
assert bfs_identity_search(gate_list, 1) == id_set

id_set = set([GateIdentity(x, x),
              GateIdentity(y, y),
              GateIdentity(z, z),
              GateIdentity(h, h)])
assert id_set == bfs_identity_search(gate_list, 1, max_depth=3,
                                     identity_only=True)

id_set = set([GateIdentity(x, x),
              GateIdentity(y, y),
              GateIdentity(z, z),
              GateIdentity(h, h),
              GateIdentity(x, y, z),
              GateIdentity(x, y, x, y),
              GateIdentity(x, z, x, z),
              GateIdentity(x, h, z, h),
              GateIdentity(y, z, y, z),
              GateIdentity(y, h, y, h),
              GateIdentity(x, y, h, x, h),
              GateIdentity(x, z, h, y, h),
              GateIdentity(y, z, h, z, h)])
assert bfs_identity_search(gate_list, 1, max_depth=5) == id_set

id_set = set([GateIdentity(x, x),
              GateIdentity(y, y),
              GateIdentity(z, z),
              GateIdentity(h, h),
              GateIdentity(x, h, z, h)])
assert id_set == bfs_identity_search(gate_list, 1, max_depth=4,
                                     identity_only=True)

cnot = CNOT(1,0)
gate_list = [x, cnot]
id_set = set([GateIdentity(x, x),
              GateIdentity(cnot, cnot),
              GateIdentity(x, cnot, x, cnot)])
assert bfs_identity_search(gate_list, 2, max_depth=4) == id_set

```

```

cgate_x = CGate((1,), x)
gate_list = [x, cgate_x]
id_set = set([GateIdentity(x, x),
              GateIdentity(cgate_x, cgate_x),
              GateIdentity(x, cgate_x, x, cgate_x)])
assert bfs_identity_search(gate_list, 2, max_depth=4) == id_set

cgate_z = CGate((0,), Z(1))
gate_list = [cnot, cgate_z, h]
id_set = set([GateIdentity(h, h),
              GateIdentity(cgate_z, cgate_z),
              GateIdentity(cnot, cnot),
              GateIdentity(cnot, h, cgate_z, h)])
assert bfs_identity_search(gate_list, 2, max_depth=4) == id_set

s = Phase(0)
t = T(0)
gate_list = [s, t]
id_set = set([])
#assert bfs_identity_search(gate_list, 1, max_depth=4) == id_set
# Fails because Phase(0) is not Hermitian, so Dagger(Phase(0))
# returns S(0)**(-1), which is Pow object. Throws exception
# at line 136ish.

```

C qcevolve.py

```
"""Evolution of quantum circuits.
```

```
This module provides utilities to evolve quantum circuits in SymPy.  
It uses Pyevolve, a Python library for developing genetic algorithms.  
More information about Pyevolve available at:
```

```
http://pyevolve.sourceforge.net/index.html
```

```
"""
```

```
from random import Random  
from sympy import Basic  
from pyevolve.GenomeBase import GenomeBase
```

```
__all__ = [  
    'GQCBase',  
    'GQCLinear',  
    'kmp_table',  
    'find_subcircuit',  
    'qc_remove_subcircuit',  
    'qc_random_reduce',  
    'qc_random_insert'  
]
```

```
class GQCBase(Basic, GenomeBase):
```

```
    """A base representation of quantum circuits for genetic algorithms.
```

```
    Specifically, GQCBase is used for genetic programming,  
    a specialization of genetic algorithms that evolve computer  
    programs, or generally, algorithms. Each quantum circuit  
    represents a quantum algorithm.
```

```
    The base class provides no default behavior other than those  
    provided by Basic and GenomeBase; all subclasses are expected  
    to provide class specific implementations.
```

```
    """
```

```
    def __new__(cls, *args):
```

```

    # args is the quantum circuit representing a genome
    obj = Basic.__new__(cls, *args)
    return obj

@property
def circuit(self):
    return self.args

def __repr__(self):
    """Return a string representation of GQCBBase"""
    rep = GenomeBase.__repr__(self)
    rep += "- GQCBBase\n"
    rep += "\tCircuit:\t\t %s\n\n" % (self.circuit,)
    return rep

class GQCLinear(GQCBBase):
    """A linear program representation of quantum circuits.

    This class also does not provide behavior for the following:

        * Initializer operator
        * Mutator operator
        * Crossover operator
        * Evaluation function

    These functions must be set before an instance of the
    class can be used.

    GQCLinear was created to provide a meaningful name
    for a representation of quantum circuits.
    """

    def __new__(cls, *args, **kwargs):
        # args is the quantum circuit representing a genome
        # kwargs should be a variable length dictionary
        obj = GQCBBase.__new__(cls, *args)

        obj._genome_circuit = args
        obj._gate_identities = False
        obj._insert_choices = []

```

```

# Doing this is a questionable design
if 'GateIdentity' in kargs.keys():
    obj._gate_identities = kargs['GateIdentity']

if 'choices' in kargs.keys():
    obj._insert_choices = kargs['choices']

if obj._gate_identities:
    collapse_func = lambda acc, an_id: acc + list(an_id.eq_identities)
    ids_flat = reduce(collapse_func, obj._insert_choices, [])
    obj._insert_choices = ids_flat

return obj

@property
def genome_circuit(self):
    return self._genome_circuit

@genome_circuit.setter
def genome_circuit(self, new_circuit):
    self._genome_circuit = new_circuit

@property
def insert_choices(self):
    # List of circuits that could be inserted into another circuit
    return self._insert_choices

def __repr__(self):
    """Return a string representation of GQCBase"""
    rep = GQCBase.__repr__(self)
    rep += "- GQCLinear\n"
    rep += "\tIdentities:\t\t %s\n\n" % (self.identities,)
    return rep

def kmp_table(word):
    """Build the 'partial match' table of the
    Knuth-Morris-Pratt algorithm.

    Note: This is applicable to strings or quantum circuits.
    """

```

```

# Current position in subcircuit
pos = 2
# Beginning position of candidate substring that
# may reappear later in word
cnd = 0
# The 'partial match' table that helps one determine
# the next location to start substring search
table = list()
table.append(-1)
table.append(0)

while pos < len(word):
    if word[pos-1] == word[cnd]:
        cnd = cnd + 1
        table.append(cnd)
        pos = pos + 1
    elif cnd > 0:
        cnd = table[cnd]
    else:
        table.append(0)
        pos = pos + 1

return table

def find_subcircuit(circuit, subcircuit, start=0, end=0):
    """Finds the subcircuit in circuit, if it exists.

    If the subcircuit exists, the index of the start of
    the subcircuit in circuit is returned; otherwise,
    -1 is returned. The algorithm that is implemented
    is the Knuth-Morris-Pratt algorithm.

    Parameters
    =====
    circuit : tuple, Gate
        A tuple of Gates representing a quantum circuit
    subcircuit : tuple, Gate
        A tuple of Gates to check if circuit contains
    start : int
        The location to start looking for subcircuit.
        If start is the same or past end, -1 is returned.

```

```

end : int
    The last place to look for a subcircuit.  If end
    is less than 1 (one), then the length of circuit
    is taken to be end.
"""

if len(subcircuit) == 0 or len(subcircuit) > len(circuit):
    return -1

if end < 1:
    end = len(circuit)

# Location in circuit
pos = start
# Location in the subcircuit
index = 0
# 'Partial match' table
table = kmp_table(subcircuit)

while (pos + index) < end:
    if subcircuit[index] == circuit[pos + index]:
        index = index + 1
    else:
        pos = pos + index - table[index]
        index = table[index] if table[index] > -1 else 0

    if index == len(subcircuit):
        return pos

return -1

def qc_remove_subcircuit(circuit, subcircuit, pos=0):
    """Removes subcircuit from circuit, if it exists.

    If multiple instances of subcircuit exists, the
    first instance is removed.  A location to check may
    be optionally given.  If subcircuit can't be found,
    circuit is returned.

    Parameters
    =====

```

```

circuit : tuple, Gate
    A quantum circuit represented by a tuple of Gates
subcircuit : tuple, Gate
    A quantum circuit to remove from circuit
pos : int
    The location to remove subcircuit, if it exists.
    This may be used if it is known beforehand that
    multiple instances exist, and it is desirable
    to remove a specific instance.  If a negative number
    is given, pos will be defaulted to 0.
"""

if pos < 0:
    pos = 0

# Look for the subcircuit starting at pos
loc = find_subcircuit(circuit, subcircuit, start=pos)

# If subcircuit was found
if loc > -1:
    # Get the gates to the left of subcircuit
    left = circuit[0:loc]
    # Get the gates to the right of subcircuit
    right = circuit[loc + len(subcircuit):len(circuit)]
    # Recombine the left and right side gates into a circuit
    circuit = left + right

return circuit

def qc_random_reduce(circuit, gate_ids, seed=None):
    """Shorten the length of a quantum circuit.

    qc_random_reduce looks for circuit identities
    in circuit, randomly chooses one to remove,
    and returns a shorter yet equivalent circuit.
    If no identities are found, the same circuit
    is returned.

    Parameters
    =====
    circuit : tuple, Gate

```

```

    A tuple of Gates representing a quantum circuit
gate_ids : set, GateIdentity
    Set of gate identities to find in circuit
seed : int
    Seed value for the random number generator
"""

if len(gate_ids) < 1:
    return circuit

# Create the random integer generator with the seed
int_gen = Random()
int_gen.seed(seed)

# Flatten the GateIdentity objects (with gate rules)
# into one single list
collapse_func = lambda acc, an_id: acc + list(an_id.eq_identities)
ids = reduce(collapse_func, gate_ids, [])

# List of identities found in circuit
ids_found = []

# Look for identities in circuit
for an_id in ids:
    if find_subcircuit(circuit, an_id) > -1:
        ids_found.append(an_id)

if len(ids_found) < 1:
    return circuit

# Randomly choose an identity to remove
remove_id = int_gen.randint(0, len(ids_found)-1)

# Remove the identity
new_circuit = qc_remove_subcircuit(circuit, ids_found[remove_id])

return new_circuit

def qc_random_insert(circuit, choices, seed=None):
    """Insert a circuit into another quantum circuit.

```

qc_random_insert randomly selects a circuit from choices and randomly chooses a location to insert into circuit.

Parameters

=====

```
circuit : tuple, Gate
    A tuple of Gates representing a quantum circuit
choices : list
    Set of circuit choices
seed : int
    Seed value for the random number generator
"""
```

```
if len(choices) < 1:
    return circuit
```

```
# Create the random integer generator with the seed
int_gen = Random()
int_gen.seed(seed)
```

```
insert_loc = int_gen.randint(0, len(circuit))
insert_circuit_loc = int_gen.randint(0, len(choices)-1)
insert_circuit = choices[insert_circuit_loc]
```

```
left = circuit[0:insert_loc]
right = circuit[insert_loc:len(circuit)]
```

```
return left + insert_circuit + right
```

```
def qc_opt_linear_init(genome, **args):
    """Initialization function for optimizing quantum circuits
    problem using an linear circuit representataion.
```

Parameters

=====

```
genome : GQCLinear
    The genome in the population
args : any (variable length)
    In many cases, will include an instance of GSimpleGA
"""
```

```

new_circuit = qc_random_insert(
    genome.genome_circuit,
    genome.insert_choices
)

genome.genome_circuit = new_circuit

def qcopt_linear_mutator(genome, **args):
    """Mutator function for optimizing quantum circuits
    problem using an linear circuit representataion.

    Parameters
    =====
    genome : GQCLinear
        The genome in the population
    args : any (variable length)
        In many cases, will include an instance of GSimpleGA
    """

    # The mutator may either look for identities in
    # the circuit and make a reduction or it may
    # insert new identities into the circuit.

    # Return the number of mutations that occurred
    # with this genome (following convention)
    pass

def qcopt_linear_eval(chromosome):
    """Evaluation function for optimizing quantum circuits
    problem using an linear circuit representataion."""
    pass

```

D test_qcevolve.py

```
from sympy.physics.quantum.qcevolve import *
from sympy.physics.quantum.gate import (X, Y, Z, H, S, T, CNOT,
    CGate)
from sympy.physics.quantum.identitysearch import (
    GateIdentity, bfs_identity_search)

def test_kmp_table():
    word = ('a', 'b', 'c', 'd', 'a', 'b', 'd')
    expected_table = [-1, 0, 0, 0, 0, 1, 2]
    assert expected_table == kmp_table(word)

    word = ('P', 'A', 'R', 'T', 'I', 'C', 'I', 'P', 'A', 'T', 'E', ' ',
        'I', 'N', ' ', 'P', 'A', 'R', 'A', 'C', 'H', 'U', 'T', 'E')
    expected_table = [-1, 0, 0, 0, 0, 0, 0, 0, 1, 2, 0, 0,
        0, 0, 0, 0, 1, 2, 3, 0, 0, 0, 0, 0]
    assert expected_table == kmp_table(word)

    x = X(0)
    y = Y(0)
    z = Z(0)
    h = H(0)
    word = (x, y, y, x, z)
    expected_table = [-1, 0, 0, 0, 1]
    assert expected_table == kmp_table(word)

    word = (x, x, y, h, z)
    expected_table = [-1, 0, 1, 0, 0]
    assert expected_table == kmp_table(word)

def test_find_subcircuit():
    x = X(0)
    y = Y(0)
    z = Z(0)
    h = H(0)
    x1 = X(1)
    y1 = Y(1)

    circuit = (x, y, z)
```

```

assert find_subcircuit(circuit, (x,)) == 0
assert find_subcircuit(circuit, (x1,)) == -1
assert find_subcircuit(circuit, (y,)) == 1
assert find_subcircuit(circuit, (h,)) == -1
assert find_subcircuit(circuit, (x, h)) == -1
assert find_subcircuit(circuit, (x, y, z)) == 0
assert find_subcircuit(circuit, (y, z)) == 1
assert find_subcircuit(circuit, (x, y, z, h)) == -1
assert find_subcircuit(circuit, (z, y, x)) == -1
assert find_subcircuit(circuit, (x,), start=2, end=1) == -1

circuit = (x, y, x, y, z)
assert find_subcircuit(circuit, (x, y, z)) == 2
assert find_subcircuit(circuit, (x,), start=1) == 2
assert find_subcircuit(circuit, (x, y), start=1, end=2) == -1
assert find_subcircuit(circuit, (x, y), start=1, end=3) == -1
assert find_subcircuit(circuit, (x, y), start=1, end=4) == 2
assert find_subcircuit(circuit, (x, y), start=2, end=4) == 2

circuit = (x, y, z, x1, x, y, z, h, x, y, x1,
           x, y, z, h, y1, h)
assert find_subcircuit(circuit, (x, y, z, h, y1)) == 11

def test_qc_remove_subcircuit():
    x = X(0)
    y = Y(0)
    z = Z(0)
    h = H(0)
    cnot = CNOT(1,0)
    cgate_z = CGate((0,), Z(1))

    # Standard cases
    circuit = (z, y, x, x)
    remove = (z, y, x)
    assert qc_remove_subcircuit(circuit, remove) == (x,)
    assert qc_remove_subcircuit(circuit, remove + (x,)) == ()
    assert qc_remove_subcircuit(circuit, remove, pos=1) == circuit
    assert qc_remove_subcircuit(circuit, remove, pos=0) == (x,)
    assert qc_remove_subcircuit(circuit, (x, x), pos=2) == (z, y)
    assert qc_remove_subcircuit(circuit, (h,)) == circuit

```

```

circuit = (x, y, x, y, z)
remove = (x, y, z)
assert qc_remove_subcircuit(circuit, remove) == (x, y)
remove = (x, y, x, y)
assert qc_remove_subcircuit(circuit, remove) == (z,)

circuit = (x, h, cgate_z, h, cnot)
remove = (x, h, cgate_z)
assert qc_remove_subcircuit(circuit, remove, pos=-1) == (h, cnot)
assert qc_remove_subcircuit(circuit, remove, pos=1) == circuit
remove = (h, h)
assert qc_remove_subcircuit(circuit, remove) == circuit
remove = (h, cgate_z, h, cnot)
assert qc_remove_subcircuit(circuit, remove) == (x,)

def test_qc_random_reduce():
    x = X(0)
    y = Y(0)
    z = Z(0)
    h = H(0)
    cnot = CNOT(1,0)
    cgate_z = CGate((0,), Z(1))

    seed = 1
    gate_list = [x, y, z]
    ids = list(bfs_identity_search(gate_list, 1, max_depth=4))
    ids.sort()

    circuit = (x, y, h, z, cnot)
    assert qc_random_reduce(circuit, []) == circuit
    assert qc_random_reduce(circuit, ids) == circuit

    circuit = (x, y, z, x, y, h)
    # With seed = 1, randint(0, 0) = 0
    assert qc_random_reduce(circuit, ids, seed=seed) == (x, y, h)

    circuit = (x, x, y, y, z, z)
    # randint(0, 2) = 0
    assert qc_random_reduce(circuit, ids, seed=seed) == (y, y, z, z)

```

```

seed = 2
# randint(0, 2) = 2
assert qc_random_reduce(circuit, ids, seed=seed) == (x, x, y, y)

gate_list = [x, y, z, h, cnot, cgate_z]
ids = list(bfs_identity_search(gate_list, 2, max_depth=4))
ids.sort()

circuit = (x, y, z, y, h, y, h, cgate_z, h, cnot)
expected = (x, y, z, y, h, y)
# randint(0, 2) == 2
assert qc_random_reduce(circuit, ids, seed=seed) == expected

seed = 5
expected = (x, y, z, cgate_z, h, cnot)
# randint(0, 1) == 1
assert qc_random_reduce(circuit, ids, seed=seed) == expected

def test_qc_random_insert():
    x = X(0)
    y = Y(0)
    z = Z(0)
    h = H(0)
    cnot = CNOT(1,0)
    cgate_z = CGate((0,), Z(1))

    seed = 1
    choices = [(x, x)]
    circuit = (y, y)
    # insert location: 0;
    assert qc_random_insert(circuit, choices, seed=seed) == (x, x, y, y)

    seed = 8
    circuit = (x, y, z, h)
    choices = [(h, h), (x, y, z)]
    expected = (x, x, y, z, y, z, h)
    # insert location: 1; circuit choice: 1
    assert qc_random_insert(circuit, choices, seed=seed) == expected

    gate_list = [x, y, z, h, cnot, cgate_z]
    ids = list(bfs_identity_search(gate_list, 2, max_depth=4))

```

```
ids.sort()

collapse_func = lambda acc, an_id: acc + list(an_id.eq_identities)
ids = reduce(collapse_func, ids, [])

circuit = (x, y, h, cnot, cgate_z)
expected = (x, cnot, h, cgate_z, h, y, h, cnot, cgate_z)
# insert location: 1; circuit choice: 31
actual = qc_random_insert(circuit, ids, seed=seed)
assert actual == expected
```

References

- [1] [Online]. Available: <http://www.media.mit.edu/quanta/qasm2circ/test13.png>
- [2] [Online]. Available: <http://code.google.com/p/sympy/>
- [3] [Online]. Available: <http://pyevolve.sourceforge.net/>
- [4] A. Barenco, C. H. Bennett, R. Cleve, D. P. DiVincenzo, N. Margolus, P. Shor, T. Sleator, J. A. Smolin, and H. Weinfurter, “Elementary gates for quantum computation,” *Phys. Rev. A*, vol. 52, pp. 3457–3467, Nov 1995. [Online]. Available: <http://link.aps.org/doi/10.1103/PhysRevA.52.3457>
- [5] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*, 2nd ed. McGraw-Hill Higher Education, 2001.
- [6] D. Deutsch, “Quantum theory, the church-turing principle and the universal quantum computer,” *Proceedings of the Royal Society of London*, vol. 400, pp. 97–117, 1985, 10.1098/rspa.1985.0070. [Online]. Available: <http://dx.doi.org/10.1098/rspa.1985.0070>
- [7] R. Feynman, “Simulating physics with computers,” *International Journal of Theoretical Physics*, vol. 21, pp. 467–488, 1982, 10.1007/BF02650179. [Online]. Available: <http://dx.doi.org/10.1007/BF02650179>
- [8] A. Gepp and P. Stocks, “A review of procedures to evolve quantum algorithms,” *Genetic Programming and Evolvable Machines*, vol. 10, pp. 181–228, 2009, 10.1007/s10710-009-9080-7. [Online]. Available: <http://dx.doi.org/10.1007/s10710-009-9080-7>
- [9] C. Lomont, “Quantum circuit identities,” presented as a computational effort to list all identities on one qubit to assist computer simplification of quantum circuits. [Online]. Available: <http://arxiv.org/abs/quant-ph/0307111>
- [10] T. M. Mitchell, *Machine Learning*. McGraw Hill, 1997.
- [11] M. Nielsen and I. Chuang, *Quantum computation and quantum information*, ser. Cambridge Series on Information and the Natural Sciences. Cambridge University Press, 2000. [Online]. Available: <http://books.google.com/books?id=65FqEKQOfP8C>
- [12] M. Sedlak and M. Plesch, “Towards optimization of quantum circuits,” *Central European Journal of Physics*, vol. 6, pp. 128–134, 2008. [Online]. Available: <http://arxiv.org/abs/quant-ph/0607123v2>