ABSTRACT

Evolving Quantum Algorithms with Genetic Programming

Duncan MacKinnon
University of Guelph, 2017

Advisors:
Dr. Daniel Ashlock and Dr. David Kribs

In this thesis we present a genetic programming scheme for evolving quantum programs in the form of arbitrary unitary operators. Expanding on the work done by Hutsell and Greenwood, and later by Krawec in developing unitary operator evolution schemes, we present an alternative representation for unitary operators which has a better search topology than the representations used in these previous implementations. This alternative representation is tested on a general unitary evolution problem, and displays better convergence than previously implemented representations.
Contents

List of Figures v

1 Introduction 1

2 Background 3
   2.1 Quantum Computing .............................................. 3
      2.1.1 Dirac Notation ............................................ 4
      2.1.2 Quantum Information Basics: Quantum States, Qubits, and Superposition .............................................. 4
      2.1.3 Postulates of Quantum Mechanics .............................................. 7
      2.1.4 The Circuit-Gate Picture of Quantum Computation .............................................. 8
      2.1.5 Universal Gate Sets .............................................. 11
      2.1.6 Entanglement .............................................. 12
      2.1.7 Quantum Programs .............................................. 13
   2.2 Evolutionary Computation and Genetic Programming .............................................. 15
      2.2.1 Evolutionary Computation .............................................. 16
      2.2.2 Traditional Genetic Programming .............................................. 18
      2.2.3 Symbolic Regression .............................................. 19
      2.2.4 Symbolic Regression Example .............................................. 20

3 Genetic Programming for Quantum Computers 23
   3.1 Developing Quantum Algorithms .............................................. 23
   3.2 Previous Work and Opportunities .............................................. 24
      3.2.1 Quantum Circuit Synthesis .............................................. 24
      3.2.2 Genetic Programming for Quantum Computers .............................................. 25
      3.2.3 Quantum Genetic Programming as Unitary Evolution .............................................. 30
   3.3 Our Approach .............................................. 37
      3.3.1 Representation .............................................. 37
      3.3.2 The GP System .............................................. 41
      3.3.3 Results .............................................. 42
List of Figures

2.1 The Bloch Sphere (By Glosser.ca - Own work, CC BY-SA 3.0, https://commons.wikimedia.org /w/index.php?curid=23263326) .................................................. 7
2.2 An example of a simple two qubit quantum circuit. Here, the input is the state $|00\rangle$, and the output state after the quantum program has operated on the input is $\frac{|00\rangle+|11\rangle}{\sqrt{2}}$. ........ 9
2.3 Circuit representation of Deutsch’s algorithm .................................................. 13
2.4 Quantum Teleportation Circuit ......................................................................... 15
2.5 In this crossover operation, the last 6 characters of two binary strings are swapped. .... 16
2.6 In this unary mutation, the 7th, 8th, and 9th characters in the string are replaced with the randomly generated string “100” .................................................................................. 17
2.7 A parse tree representation of a simple program ............................................... 18
3.1 Mean number of mating events required to generate a solution for varying problem dimensions and representation lengths. .................................................. 43
Chapter 1

Introduction

In the past few decades quantum computers have progressed from being mere mathematical entities, confined to the world of thought experiments and speculation, to being physically implementable machines with which we can conduct actual experiments, computations, and information processing tasks. Vast amounts of research have been done to develop physical implementations of quantum computers, and to build up a sophisticated theory of quantum information and quantum information processing. As of yet, however, much is still unknown regarding the precise advantages quantum computers offer over their classical counterparts. Several quantum algorithms have been discovered which can perform information processing tasks far more efficiently than is possible on a classical computer. Recently, however, progress in the field of quantum algorithm development seems to have stalled, with few quantum algorithms of importance having been discovered since the wave of discoveries led by David Deutsch, Peter Shor, and Lov Grover in the 1990s. There are several possible reasons for this lack of progress, not the least of which is the possibility that there simply are no more useful quantum algorithms waiting to be discovered. If this is not the case, however, the reason might actually be that the development of quantum algorithms is a very difficult problem for humans to solve.

It is this second possibility which has led some researchers to look for new ways to approach the problem of quantum algorithm development - ways which make use of tools like machine learning, artificial intelligence, and automatic search based problem solving and optimization. To this end, there has been a growing body of research in applying the techniques of evolutionary computation to the problem of developing quantum algorithms and quantum circuits. Evolutionary computation can be well suited for solving certain problems which are difficult for mere human intelligence to approach, and so its application
As part of this growing body of research, various evolutionary computation-based development schemes have been created with the goal of generating quantum algorithm circuits as solutions for particular problems. The complexity of the search space of quantum algorithms has, however, been a major limitation on the kinds of problems for which these development schemes have been implemented to generate solutions. For more complex problems, evolutionary search algorithms can take exponentially longer to find solutions than with simpler problem instances, and so presently, evolutionary techniques have only been successfully implemented to discover (or rediscover) very simple quantum algorithms.

Precisely because of this limitation, it is crucial that the evolutionary techniques being applied to quantum algorithm generation are implemented in a way that maximizes search efficiency, so as to reduce the constraints on complexity as much as possible and maximize the potential for interesting results and new discoveries. If the particular evolutionary approaches which are being applied can be improved upon, then it is hoped that these approaches will gain new and broader applicability within the field of quantum algorithm development, and elsewhere. In this thesis, we take stock of evolutionary approaches which have been applied to the problem of quantum algorithm generation, in order to find ways in which these implementations of evolutionary computation can be improved upon. We then present our own evolutionary program for evolving quantum algorithms, which, due to an alternative program representation, demonstrates better convergence properties when compared with previous implementations.
Chapter 2

Background

Before we can address questions specifically related to our work, it will be useful to provide a brief summary of some relevant concepts from the two fields which are essential for understanding our project. For the remainder of this chapter, that is just what we will do.

2.1 Quantum Computing

The strange behavior of very small particles which is described by the relatively new science of quantum mechanics has, in recent times, opened the door to radical new possibilities in computation and information processing. In 1982, physicist Richard Feynman conceived the idea of a computational device based on the principles of quantum mechanics, and thus, the idea of the quantum computer was born [16]. In the decades since this conception, great strides have been made with respect to the implementation of quantum computing systems. No longer confined to the world of abstract mathematics, actual small scale quantum computers are presently being studied and experimented with in laboratories around the world.

Much progress has also been made in the theory of quantum computer programming. David Deutsch, in the research stemming from Feynman’s original conception of a quantum mechanical computer, discovered a quantum algorithm which, by exploiting quantum mechanical phenomena, could in theory be run on a quantum computer to solve a particular problem (known as Deutsch’s Problem) in less operations than is possible on a classical computer. (This algorithm will be examined in more detail below.)

This discovery generated some interest in the study of quantum algorithms, which culminated in 1994 with Peter Shor discovering quantum algorithms that could be used to
efficiently solve two very important problems; the problem of finding the prime factors of an integer, and the *discrete logarithm* problem [16]. The discovery of these algorithms was very significant for the field of quantum computing: it is still widely believed that these problems cannot be efficiently solved on a classical computer, and so these algorithms have been seen as evidence that quantum computers have unique advantages and are suitable for solving problems which classical computers are incapable of handling efficiently [16]. Further important discoveries of better-than-classical quantum algorithms include Lov Grover’s database search algorithm for searching through an unstructured search space, which was discovered in 1995 [9].

What is it, then, that makes quantum computers able to solve these problems differently from classical computers? How does quantum computation work? The remainder of this section gives a brief introduction to the theory of quantum computation in order to address these questions.

### 2.1.1 Dirac Notation

It is standard in quantum information research to use the Paul Dirac’s notation to represent the states of quantum systems. In Dirac notation, vectors are either *bra* vectors or *ket* vectors. A ket vector is just a typical column vector, so for a vector \( q = [q_1, ..., q_n] \), \( |q \rangle \) (the ket of \( q \)) is simply a column vector with the entries \( q_1, ..., q_n \). A bra vector, on the other hand, is the conjugate transpose (or *dual*) of the corresponding ket vector, so the bra vector of \( q \) would be the row vector with entries \( q_1^*, ..., q_n^* \) [8].

This notation makes it easy to work with inner and outer products of our vectors. For example, the inner product of vectors \( q \) and \( s \) is simply given by the vector product \( \langle q|s \rangle \), and their outer product is simply given by the product \( |q\rangle\langle s| \).

### 2.1.2 Quantum Information Basics: Quantum States, Qubits, and Superposition

In the classical model, bits are the most basic units of information. A string of \( n \) bits is in one of a finite number of states (\( 2^n \) states to be exact, since each bit in the string is one of two states, 0 or 1). The basic units of quantum information, on the other hand, are *quantum bits* or *qubits*. Since their properties are modeled after the predictions of quantum mechanics, qubits aren’t limited to being in one of the two classical bit states (i.e. 0 or 1), but can rather exist in a *superposition* of these two states. In order to represent this, we complicate
our classical picture a bit. The classical states 0 and 1 are, for qubits, are represented by
the following complex unit vectors:

\[ |0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \]  

(2.1)

where \(|0\rangle, |1\rangle \in \mathbb{C}^2\). A qubit in a superposition state is represented by a complex vector
\(|\psi\rangle = \alpha |0\rangle + \beta |1\rangle\), where \(|\alpha|^2 + |\beta|^2 = 1\) (i.e. the complex vector in superposition still has
a Euclidean norm of 1, like the classical state vectors of which it is a linear combination).\(^1\)

Thus, generally speaking, the state of a qubit is represented by a vector on the unit sphere
in \(\mathbb{C}^2\). This means that, in contrast to classical bits which can only be in one of two states, a
qubit actually has an infinite number of possible states, corresponding to infinite points on
the continuous surface of the unit sphere in \(\mathbb{C}^2\). When we perform a measurement on a qubit,
it collapses from its superposition state into one of its classical states. This measurement
is probabilistic, and the result is determined by the probability amplitudes \(\alpha\) and \(\beta\). If we
measure the state of the qubit \(|\psi\rangle\), we will observe the classical state \(|0\rangle\) with probability
\(|\alpha|^2\), or the state \(|1\rangle\) with probability \(|\beta|^2\).

Since qubits are complex vectors rather than simple Boolean variables, concatenating
multiple qubits into an \(n\)-qubit system is not as simple as just grouping them together into
a string. An \(n\)-qubit system is represented in the Hilbert space \(\mathcal{H}\) obtained by taking the
tensor product of the complex vector spaces of each of the qubits in the system:

\[ \mathcal{H} = (\mathbb{C}^2)^\otimes n = \mathbb{C}^N, N = 2^n \]  

(2.2)

With the standard orthonormal basis for this state space being made up of the following computational basis states:

\[ \{ |i_1...i_n\rangle := |i_1\rangle \otimes ... \otimes |i_n\rangle |i_j \in \{0,1\} \}. \]  

(2.3)

This Hilbert space is the state space for the system of qubits, and a complex vector in
this state space is the quantum computing equivalent of a Boolean string of bits in classical
computing. A system of qubits is situated as a state vector in this state space by taking
the tensor product of all of the qubits, e.g. the system made up of \(n\) qubits \(|\psi_1\rangle, ..., |\psi_n\rangle\)

\(^1\)Notational note: two superposition states which come up often in quantum information are the states
\(\frac{1}{\sqrt{2}} (|0\rangle + |1\rangle)\) and \(\frac{1}{\sqrt{2}} (|0\rangle - |1\rangle)\), which are denoted by \(|+\rangle\) and \(|-\rangle\) respectively.
would have its state represented by the following \( N \)-dimensional complex vector: 
\[
\psi = |\psi_1\rangle \otimes ... \otimes |\psi_n\rangle,
\]
with \( \psi \in \mathbb{C}^N \). Note that the resultant state vector will, as above, be located on the unit sphere in the state space.

Another common way to represent a quantum state is by using a *density operator* or *density matrix*. The density matrix representation of a quantum state is mathematically equivalent to the state vector representation, but in certain contexts it can provide a more convenient way of looking at certain properties of quantum information. In fact, the density operator representation of a quantum state is generally seen as the most complete way of portraying it, with the vector representation being a convenient shorthand. For a finite dimensional state, the density operator \( \rho \) is a linear operator that is positive semidefinite and has a trace of 1, where the magnitudes of the values along the main diagonal correspond to the probabilities of measurement outcomes in the computational basis [16]. The density matrix \( \rho \) corresponding to a state with vector representation \( |\psi\rangle \) is given by \( \rho = |\psi\rangle \langle \psi| \).

For a *pure* state, that is the state of a quantum system for which we know precisely its quantum state, and which can subsequently be represented simply as the complex state vector \( |\psi\rangle \) corresponding to that quantum state, the density matrix operator will simply be the matrix obtained by taking the outer product of the state vector with itself (as above). Density operators are also, however, capable of representing so called *mixed* states, meaning the states of systems which may be in state \( |\psi_1\rangle \) with probability \( p_1 \), or might be in some other states \( |\psi_2\rangle, ..., |\psi_n\rangle \) with respective probabilities \( p_2, ..., p_n \). In such a case, we say that \( \rho \) represents a *mixture* of the states \( |\psi_1\rangle, ..., |\psi_n\rangle \), and the corresponding density matrix is defined as follows [16]:

\[
\rho = \sum_{i=1}^{n} p_i |\psi_i\rangle \langle \psi_i|
\]

In the case of a single qubit, a useful way to visualize the qubit’s quantum state is by using the *Bloch Sphere* representation. To do this, we write the qubit’s state vector using the following parameterization:

\[
|\psi\rangle = e^{i\gamma} (\cos (\theta/2) |0\rangle + e^{i\phi} \sin (\theta/2) |1\rangle).
\]

Then interpret the angles \( \theta \) and \( \phi \) as spherical coordinate parameters to get the picture in Figure 2.1.

When we are working with a higher dimensional quantum system (i.e. multiple qubits),
picturing the state vector becomes very difficult, but for the development of basic intuitions about quantum information, reflecting on this Bloch sphere representation of a qubit state in 3-d Euclidean space can be quite helpful.

2.1.3 Postulates of Quantum Mechanics

Now that we’ve established some of the basic differences between quantum information and classical information, it will be expedient to turn to the more general theory of quantum mechanics, and outline some postulates of this theory which are relevant to understanding the potentialities and constraints of quantum computers. These postulates describe the behaviour of quantum systems, which a register of qubits just one example of [16].

(i) **Quantum State Postulate:** Every isolated physical system has an associated Hilbert space known as the state space of the system. The quantum state of this system is completely described by its state vector, a unit vector in this associated state space.

(ii) **Evolution Postulate:** A closed quantum system evolves over time according to a unitary transformation; the state $|\psi_1\rangle$ of a quantum system at time $t_1$ is related to the state
\( |\psi_2\rangle \) at time \( t_2 \) by a unitary operator \( U \) which depends only on the times \( t_1 \) and \( t_2 \):

\[
|\psi_2\rangle = U |\psi_1\rangle
\] (2.6)

Note: when applied to a complex vector, unitary operators preserve the length of that vector - this means that if a unitary is applied to a state vector which is on the unit sphere in our complex state space, it will map it to another vector on this unit sphere.

(iii) Composite System Postulate: The state space of a composite system, i.e. a system containing several smaller quantum systems is the tensor product of the state spaces of the component quantum systems. Likewise, the overall state of this composite system is the tensor product of the state vectors of the component systems.

(iv) Measurement Postulate: Measurements on a quantum system can be described by a collection \( \{M_m\} \) of so called measurement operators, which act on the state space of that quantum system. The index \( m \) refers to possible outcomes that could result from the measurement. If the quantum system is in state \( |\psi\rangle \) before a measurement is performed, the the probability that the outcome \( m \) occurs is given by:

\[
p(m) = \langle \psi | M_m^* M_m |\psi\rangle
\] (2.7)

Once this result has been observed, the probabilistic superposition of this outcome collapses to align with the actual observed result. The resultant state is given by:

\[
\frac{M_m |\psi\rangle}{\sqrt{\langle \psi | M_m^* M_m |\psi\rangle}}
\] (2.8)

2.1.4 The Circuit-Gate Picture of Quantum Computation

The above four postulates give a sense of the rules which we are operating under in the field of quantum computing. A register of qubits is a composite quantum system, the tensor product of the subsystem corresponding to each qubit. This register of qubits is the input for a quantum computation. The system of qubits evolves according to a series of unitary operators (known in aggregate as a quantum algorithm or quantum program - these two terms will be used interchangeably) which are implemented to transform the input into a desired output state. Finally, this output state is generally measured in order to extract desired information from the computation. This entire process can be nicely represented symbolically by using something called a circuit-gate diagram.
A quantum circuit-gate diagram is in some ways similar to the Boolean logic circuit diagrams which are used to represent simple classical computer programs. In the classical picture, we have a register of input bits on the left with a wire leading from each of them. These wires then pass through logic gates which implement changes to the state of the effected bits corresponding to logic operations. Finally, the output of the circuit, a string of bits, is found at the right end of the diagram.

In the quantum circuit-gate picture, by contrast, the input is the quantum state of a system of qubits (i.e. a complex unit vector in the qubit system’s state space). Each of these qubits has corresponding wire that stretches to the right side of the diagram. Quantum gates (i.e. unitary operations), or other special operations such as measurements, are represented pictorially by boxes or other signs which are overlayed on the wires corresponding to qubits which their action targets. These operations collectively compose the quantum algorithm which is being represented. Finally, as with the classical picture, the output is found on the right side of the diagram, however in this case instead of a string of bits, our output is the new quantum state of the system of qubits which was produced by our quantum program (again a complex unit vector in the state space).

Some of the most common single qubit gates in quantum circuits are the so called Pauli operators, $X, Y,$ and $Z$ (also commonly denoted $\sigma_1, \sigma_2,$ and $\sigma_3$); the identity operator, $I$; the Hadamard operator, $H$; and the $\frac{\pi}{8}$ phase gate, $T$. When acting on a single qubit (i.e. on $\mathcal{H} = \mathbb{C}^2$, in the standard computational basis), these operators correspond to the following
unitary matrices:

\[ X = \sigma_1 = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \quad Y = \sigma_2 = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad Z = \sigma_3 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \]

\[ I = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad H = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix}, \quad T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix} \]

In general, the unitary operations that we apply to a system of qubits must match the dimension of that system - obviously we can’t multiply a 4 dimensional state vector by a \(2 \times 2\) matrix. Thus, if we want to implement these single qubit gates in a circuit with multiple qubits, we must expand them so that they match the dimension of the qubit system’s state space. We do this by taking the tensor of the gate we wish to implement with identity operators. For example, if we want to target just the second qubit in a register of 3 qubits with an \(X\) gate, the action of this gate on the composite system would be given by the matrix \(I \otimes X \otimes I\). Note that the resultant operator will be unitary on the state space of the 3 qubit state.

There are also gates which necessarily act on more than one qubit. The most common of these multi-qubit gates is the controlled-not or \(CNOT\) gate. The \(CNOT\) gate can act on any number of qubits in the system, taking one of them as its control qubit, and any number of others as its targets. The \(CNOT_{1,2}\) gate (with qubit 1 as the control and qubit 2 as the target) in a 2-qubit system is represented by the following matrix:

\[ CNOT_{12} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \]

The action of this gate amounts to flipping the target qubit if the control qubit is in state \(|1\rangle\), and doing nothing if the control qubit is in state \(|0\rangle\). Note that other controlled gates can be implemented in a similar manner. A controlled \(Z\) gate acting on the same qubits could be formulated by replacing the bottom right quadrant of the above matrix with the entries from a \(Z\) matrix instead of the \(X\) entries that are currently there. Clearly, if the first qubit is in state \(|0\rangle\), the gate changes nothing, and if the first qubit is in state \(|1\rangle\), the \(Z\) gate will be applied to the second qubit.
When dealing with a control qubit in a state of superposition, however, forming intuitions about the behaviour of this gate can be challenging. Essentially, the controlled gate acts only on the states in which the control qubit would actually be observed in state $|1\rangle$ when measured. In particular, it acts on the subspace of the state in which the control qubit is in state $|1\rangle$. Interpreting this and grasping it with clear and distinct intuitions is one of the challenges of studying quantum systems.\textsuperscript{2} Fortunately, where intuition may fall short, the mathematics are still reliable. For calculating the resultant state after the application of the above controlled not gate, we need only apply the gate’s corresponding unitary matrix to the state vector, and examine the result.

One more important class of gates is the class of so called black box gates or oracles. An oracle gate can be thought of as a subroutine in a quantum algorithm, which, when called on appropriately in a quantum program, can be used to do useful computations. [16] Oracles will be addressed in more depth as needed, but for now it is sufficient to say that these gates take the form of unitary operators which appeal to some external information to determine exactly how they act on the state.

There are, of course, many other quantum gates which are used to fill out quantum circuits, but those detailed above are sufficiently illustrative for our purposes. What all these gates have in common is that they are unitary operators on the complex vector space that they act on. Thus they map one quantum state to another by mapping a state’s corresponding complex unit vector to another complex unit vector (i.e. another quantum state) in the state space. Furthermore, the composition of a unitary operator with a unitary operator yields another unitary operator. So, the evolution of a system of qubits from its input state to its output state in a quantum algorithm can be captured in a single unitary, the composition of all the individual unitaries which make up the algorithm. A quantum circuit essentially illustrates a particular unitary as a composition of other unitary operators.

### 2.1.5 Universal Gate Sets

In classical computation, a small set of gates (such as \{AND, OR, NOT\}) can be sufficient to compute an arbitrary classical function. Such a set is called a universal set of gates for classical computation. Similarly, in quantum computation if a set of gates can be used to construct a quantum circuit to approximate any unitary operation to arbitrary accuracy,\footnote{In fact, intuitions about the puzzling behaviour of quantum systems are so elusive that even the greatest scientific minds have struggled with it. Richard Feynman, for example, famously claimed “I think I can safely say that nobody understands quantum mechanics”.[7]}
we call it *universal for quantum computation*. One example of a universal set of gates for quantum computation is the set \{H, CNOT, T\} \[16\].

### 2.1.6 Entanglement

One fascinating property which can emerge from quantum systems is the property of *entanglement*. Entangled states are essentially states in composite quantum systems in which the state of one part of the system is not independent from the state of another part of the system. An example of an entangled state in a 2-qubit system is the following state; one of the maximally entangled so called *Bell states* or *EPR pairs*:

\[
|\psi\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 1 \end{bmatrix} \tag{2.11}
\]

In this particular state, it is evident that if the first qubit is found to be in state $|0\rangle$, the second qubit must then also be in the state $|0\rangle$. Likewise, if one of the qubits is found to be in the state $|1\rangle$, it is necessarily the case that the other qubit is in the state $|1\rangle$.

At first glance, this may seem not all that surprising. Of significance, however, is the fact that a qubit system could be prepared in such an entangled state, and then the qubits could be physically separated and brought to different geographic locations, and this property of entanglement will still hold. Hypothetically, if the above EPR pair of qubits was prepared in Guelph, and then one of the qubits were transported to Hong Kong (with great care not to interfere with the quantum state), then a measurement of the qubit in Guelph would completely determine the state of the qubit in Hong Kong as well, even though until the moment the measurement was made, the theory of quantum mechanics tells us that both qubits existed in a state of probabilistic superposition. It thus seems, strangely, that a measurement that is done to a qubit in Guelph can instantaneously effect the state of a qubit in Hong Kong, collapsing it from a state of superposition to a classical state. The reality of this seemingly paradoxical “spooky action at a distance” is supported by numerous experimental verifications \[5\].

Quantum entanglement challenges and stretches our intuitions about quantum states and the meaning of superposition, but it can also be viewed as a vital resource in quantum computation. In fact, entanglement and the use of entangled states as a resource is at the
heart of important quantum algorithms such as the quantum teleportation algorithm, which will be described in more detail below.

2.1.7 Quantum Programs

In this section, we will detail two examples of quantum programs, in order to give the reader a sense of exactly what a quantum program looks like, and of the kinds of programs which have been discovered so far. This section largely draws from the presentation of these algorithms in [16] by Nielson and Chuang.

David Deutsch, who first conceived of the idea of quantum computers, is also the inventor of the first better-than-classical quantum algorithm. Deutsch’s Algorithm can be run on a quantum computer to determine whether a given Boolean function is constant or balanced in with less calls to the function than is possible on a classical computer. For example, imagine we have the function \( f \) defined by \( f : \{0, 1\} \rightarrow \{0, 1\} \). We don’t explicitly know what \( f \) maps each input to, but have a black box that can be fed an input and return \( f \) of that input. Classically, if we want to determine if this function is constant or balanced (i.e. if the function maps both inputs to the same output, or each to a different output), we must call on the black box for each input (0 and 1), and then compare the outputs. Deutsch showed, however, that on a quantum computer, we can determine with certainty whether this particular function is constant or balanced by calling the black box only one time.

Deutsch’s algorithm can be represented easily as a quantum circuit, using only Hadamard gates and an oracle gate, a unitary \( U_f \) which transforms the input state \( |x\rangle \otimes |y\rangle \) to the state \( |x\rangle \otimes |y \oplus f(x)\rangle \), where \( \oplus \) represents addition modulo 2. The algorithm is implemented by the circuit in Figure 2.3.

\[
\begin{array}{c}
|0\rangle & \xrightarrow{H} & |\psi\rangle \\
|1\rangle & \xrightarrow{H} & U_f \xrightarrow{H} |\psi\rangle \\
\end{array}
\]

Figure 2.3: Circuit representation of Deutsch’s algorithm

The first qubit of the output state \( |\psi\rangle \) is then measured in the computational basis, a measurement of the state \( |0\rangle \) meaning that the function is balanced, and a measurement of the state \( |1\rangle \) meaning that the function is constant. An algebraic representation of this algorithm is given below in Equation 2.12. Since the possible output states for the cases
where the function is constant are located in the subspace of the state space where the first qubit is in state \( |0\rangle \), and the possible output states for the cases where the function is balanced are located in the subspace of the state space where the first qubit is in state \( |1\rangle \), a measurement of the first qubit is sufficient to tell us which of these cases we are in.

\[
\psi = (H \otimes I)(U_f)(H \otimes H)(|0\rangle \otimes |1\rangle)
\]

\[
= (H \otimes I)(U_f)(\frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}})
\]

\[
= \begin{cases} 
(H \otimes I)(\pm \frac{|0\rangle + |1\rangle}{\sqrt{2}}) \otimes (\frac{|0\rangle - |1\rangle}{\sqrt{2}}) & \text{if } f(0) = f(1) \\
(H \otimes I)(\pm \frac{|0\rangle - |1\rangle}{\sqrt{2}}) \otimes (\frac{|0\rangle - |1\rangle}{\sqrt{2}}) & \text{if } f(0) \neq f(1) 
\end{cases}
\]

\[
= \begin{cases} 
\pm |0\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} & \text{if } f(0) = f(1) \\
\pm |1\rangle \otimes \frac{|0\rangle - |1\rangle}{\sqrt{2}} & \text{if } f(0) \neq f(1) 
\end{cases}
\]

The algorithm described above solves the 2 input case, but a generalized algorithm that, given a function \( f : \{0, 1, ..., 2k\} \rightarrow \{0, 1\}, k \in \mathbb{N} \) that we know must be either constant or balanced can determine which of these cases is true, was subsequently developed by Deutsch and Jozsa. This more generalized algorithm, known as the Deutsch-Jozsa algorithm, remarkably still only requires one call to the oracle to solve the problem with certainty.

Another interesting and relatively simple quantum program is the quantum teleportation algorithm. Essentially, quantum teleportation is a way to send a quantum state over a large distance by using only a shared EPR state, local quantum operations, and classical communication. To illustrate this, imagine that Alice and Bob prepare an EPR pair of qubits in Guelph, and then Bob goes to Hong Kong, bringing one of the qubits from this EPR state with him. Now, suppose that Alice has some other qubit in the quantum state \( |\phi\rangle \) which she wishes to send to Bob in Hong Kong without having to physically transport it. Alice can remotely transfer the state \( |\phi\rangle \) to Bob if the two of them follow the quantum teleportation algorithm, displayed in Figure 2.4.

In the above circuit, the first two qubits represent qubits in Alice’s possession, and the third qubit represents the qubit from the EPR pair that Bob has possession of. \( M_1 \) and \( M_2 \) are not unitaries, but measurements in the computational basis performed by Alice on her qubits. Once these measurements have been made, those qubits collapse into classical states, indicated by the double wires. Finally, the \( X(M_2) \) and \( Z(M_1) \) gates are conditionally applied by Bob to decode his qubit, depending on what results Alice tells him she observed.
from the measurements $M_1$ and $M_2$. If the measurement result was 1, then Bob should apply the corresponding Pauli gate to the qubit. If the measurement result was 0, then Bob should simply apply the identity instead. The result is that Bob’s qubit will now be in the state $|\phi\rangle$, even though the only communication that was required between Alice and Bob was the transmission of two classical bits worth of information (namely, the measurement results of $M_1$ and $M_2$). This algorithm shows that it is possible to use a shared EPR state to transmit a qubit’s worth of information using only local operations and minimal classical communication, a result that is quite remarkable.

The two algorithms detailed above are just a sample of the quantum programs that have been developed since the conception of quantum computing. Their example illustrates just what, generally speaking, a quantum program actually is. Having touched on this, and the fundamental ideas from quantum computation that have been outlined thus far, we can now move on to describing relevant concepts from the other field which is central to our project: genetic programming.

### 2.2 Evolutionary Computation and Genetic Programming

In order to properly describe what genetic programming is, it is prerequisite to describe the broader field of evolutionary computation, of which the field of genetic programming is a subcategory. Thus we will begin this section with a description of evolutionary computation, which is largely drawn from [1].
2.2.1 Evolutionary Computation

Evolutionary computation can be simply described as the use of the theory of evolution as an algorithm. Essentially, evolutionary algorithms encapsulate any program which starts by generating a population of structures, and then repeats the following process until satisfied:

- Test the structures for quality
- Select structures to reproduce
- Produce new variations of selected structures
- Replace old structures with new ones

This fundamental approach can be applied to a wide variety of structures to solve an even wider variety of problems. Evolutionary algorithms can be used to generate mathematical models to fit unwieldy data sets, to discover designs for application in mechanical engineering problems, and to generate computer programs.

Two important issues to consider when implementing an evolutionary algorithm are the choice of a representation, and of a fitness function.

The issue of choosing a representation refers to the choice of what kind of data structures are being manipulated by the evolutionary algorithm, and what variation operators are being used to generate new structures. These choices are not independent of each other. For example, if we are developing an evolutionary algorithm that evolves binary strings, we need to choose mutation operators which produce appropriately varying strings in the “produce new variations” step of the algorithm. One such mutation operator is the crossover operator, which exchanges material between two data structures, which in the case of binary strings would simply swap portions of the strings from two elements.

\[
\begin{array}{c}
\text{Parent one:} & 010101010101 \\
\text{Parent two:} & 000000111111 \\
\text{Offspring one:} & 010101111111 \\
\text{Offspring two:} & 000000010101
\end{array}
\]

Figure 2.5: In this crossover operation, the last 6 characters of two binary strings are swapped.

Another common kind of mutation operator simply rewrites part of a data structure. In the binary string example, such a unary mutation would entail replacing part of a string with a randomly generated string of the same length.
The issue of choosing a fitness function refers to the problem of creating a function which heuristically estimates the quality of members of the evolving population by mapping them to a so called fitness value (or in some cases simply by comparing pairs of members of the population and deciding which of the two is better without assigning a numerical fitness value).

The representation and fitness function that are chosen can ultimately have an enormous effect on the performance of an evolutionary algorithm. As such, the study of different representations and fitness functions in the context of different applications of evolutionary algorithms can be quite fruitful: a novel approach to representation or to estimating fitness can dramatically increase the power and decrease the run time of a given evolutionary algorithm, yielding results which were impossible with less sophisticated implementations. An example of an evolutionary algorithm in which a different choice of representation led to an increase in the algorithm’s speed and power can be found in [3], wherein a representation for walks on a 2-d grid is developed for solving the so called self avoiding walk problem. This representation, by design, excludes parts of the search space (i.e. the space of walks on a particular grid) which can be deductively ruled out as solutions for this problem, and thus the search through the remainder of the space can be conducted with far greater efficiency. For certain problem instances, the performance increase from this representation translated to a search efficiency that was several orders of magnitude greater than searches which used less sophisticated representations. Again in [2] we can see an illustrative example of how the right choice of representation can actually allow a genetic algorithm to find solutions which would not have been discoverable using other representations. Here, Ashlock et al. show that by using a sparse initialization as part of their representation in a maze generating algorithm, interesting maps can be generated which would not show up with any realistic probability without such initialization.

In this paper, we are interested in a particular subset of the applications of evolutionary algorithms, namely the field of genetic programming, which we will describe below.
2.2.2 Traditional Genetic Programming

Genetic programming (abbreviated GP) is characterized by the use of an evolutionary algorithm to evolve computer programs. The data structures that the evolutionary algorithm manipulates are themselves programs, which are traditionally represented and stored as parse trees. These parse trees are made up of interior nodes which represent operations and terminal nodes which represent so called terminals. The so called operation nodes, as one might expect, describe operations performed in the program. The terminals represent inputs or parameters for these operations, containing either values passed to the program from an external source, constants, or input/output devices. Figure 2.7 below is an example of a parse tree representation of a program:

```
* + x y /
 \\ \\
\pi SQRT
```

Figure 2.7: A parse tree representation of a simple program

The program tree in Figure 2.7 can also be represented in a functional notation as is used in the computer language LISP (from here on referred to as LISP-like notation) as follows:

\[(\ast (\div (\ast x y) (\sqrt (\pi ))) (\ast z 3)))\]  \hspace{1cm} (2.13)

Terminology note: The root of a parse tree is the first operation when written in this LISP-like form, and the arguments of an operation are the subtrees that stem from that operation’s node.

This is the kind of data structure that is manipulated in traditional GP. The genetic operations which are performed on these structures can take several forms. One possible
genetic operation is *subtree crossover*, which consists in exchanging two subtrees from different members of the program population. Other possible genetic operations include *subtree mutation* and *chopping*: subtree mutation entails taking a randomly chosen subtree, deleting it, and replacing it with another randomly generated subtree; while chopping is an operation which reduces the size of a parse tree by replacing the entire tree with just one of the arguments of its root operation.

With the elementary data structures and genetic operations defined thus, the traditional GP algorithm follows the same essential structure defined above for evolutionary algorithms. First, we generate a population of structures, typically parse trees represented in a LISP-like notation. Then these parse tree programs are tested for performance based on a chosen fitness function. The fittest ones are reproduced and mutated using our genetic operators, and the subsequently generated programs then replace the worst performing programs from the old population. This selection and mutation process is repeated until we have obtained a sufficiently fit program for our needs.

### 2.2.3 Symbolic Regression

One typical use for GP algorithms is to evolve formulas to fit a data set, i.e to do *symbolic regression*. To do this, we simply run a GP algorithm using the minimization of the squared error summed over all the data points as our fitness function. The GP algorithm generates random parse trees, and evolves them to minimize the squared error. The subsequent trees define functions which accurately model the data.

The algorithm is effectively searching the space of parse trees, an enormous and complex search space. Unlike simpler evolutionary algorithms which are often used to perform a simple unimodal optimization over the real numbers, symbolic regression through GP entails a mixed search of a discrete space of formulas (parse trees with a particular set of operations) with real parameters. The particular set of operations that are defined for our parse trees is very important here, as it has a huge impact on the topology of the search space. Differences in the choice for the set of operators will have a significant impact on the performance of the algorithm. For example, if we are trying to create a model for data which is consistent with the exponential function $e^x$ but only have operations defined for addition, subtraction, multiplication, and division, the functions we evolve will look and perform a lot differently than if we have an explicitly defined exponential function as one of our possible operators.

Other important aspects to consider when applying a GP algorithm are the set of muta-
tion operators which are used. In addition to the ones described above (i.e. subtree crossover, subtree mutation, and chopping), common mutation operators used for symbolic regression include constant mutation, terminal mutation, and operation mutation.

A constant mutation locates a constant in the parse tree and mutates it, depending on its data type. If the constant is a real number, then we add a number from some predetermined distribution to it; if it’s an integer or character, we replace it with some appropriate new integer or character.

A terminal mutation, similarly to the constant mutation, locates a terminal in the parse tree and replaces it with a new terminal. The key difference, however, is that this mutation acts not only on constants, but can change terminals containing variables as well: a variable or constant in the designated terminal is replaced by either a variable or a constant.

An operation mutation, locates an operation and replaces it with another operation from the operation set that takes the same number of arguments.

The inclusion of the appropriate mutation operators, as detailed above when it came to the inclusion of the appropriate operations, can have a significant impact on the performance of the algorithm by influencing the topology of the search space. With the right mutation operators defined, the GP algorithm can converge to good solutions much quicker and with a greater rate of success than otherwise.

### 2.2.4 Symbolic Regression Example

To illustrate how symbolic regression works, we will here detail an example where symbolic regression is used to evolve a function to model data points which are taken from the function $f(x) = \frac{1}{x-1}$. In particular, we use symbolic regression to evolve a function that models the following set of points:

$$\left\{ \left( \frac{i}{40} - 1, \frac{1}{(i/40 - 1)^2 + 1} \right) : i = 0, \ldots, 80 \right\}.$$  

The GP scheme to carry out this symbolic regression uses as its fitness function the sum of the squared error (SSE) over all the data points. In theory, when this SSE is minimized, the evolved function should resemble the original $f(x)$ that generated these data points.

So, a GP algorithm was run with functions represented by parse trees with an initial population of trees of six nodes, and a chop operation to limit the size to no more than 12 nodes; with defined binary operations including plus (+), minus (-), times (*), and divide.
(\sqrt{\cdot}); unary operations including square (sqr), square root (sqt), sine (sin), cosine (cos), arctangent (atan), and minus\(^3\) (-), and terminals including real constants \(\in [-1, 1]\), and \(x\) (the independent variable). The fitness function used is the minimization of SSE as described above. Variation operators in this implementation include crossover, and unary mutations (constant mutation, terminal mutation, operation mutation).

This scheme was applied to 30 randomly initialized populations of 400 parse trees, and for each population the algorithm was run until 500 generations had passed. Optimal evolved solutions included:

- \(\text{sqr}(\cos(\tan(x)))\) with \(\text{SSE} = 5.582\,96 \times 10^{-12}\)
- \(\cos(\tan(x)) \ast \cos(\tan(x))\) with \(\text{SSE} = 5.582\,96 \times 10^{-12}\)
- \(\cos(\sqrt{x \ast \tan(x/0.469912)})\) with \(\text{SSE} = 8.529\,68 \times 10^{-6}\)

At first glance, it may seem that our symbolic regression failed to match the function which generated the data points. This is, however, not the case. With the application of a trigonometric identity (namely that \(\cos(\tan(x)) = \frac{1}{\sqrt{x^2 + 1}}\)) we can see that these evolved functions are in fact identical (in the case of the first two) or similar on the relevant interval (in the case of the third one) to the data generating function. We have:

\[
\text{sqr}(\cos(\tan(x))) = (\cos(\tan(x)))^2
= \left(\frac{1}{\sqrt{x^2 + 1}}\right)^2
= \frac{1}{x^2 + 1}
\]  

(2.15)

So, although no solutions of the precise form \(\frac{1}{x^2 + 1}\) were evolved, equivalent functions were. These functions provided perfect models for the data that came from the original function. One might ask why no functions of this original form were evolved when equivalent trigonometric representations of the same function were found. The reason for this is that the simple function \(f(x) = \cos(x)\), which occurs with a relatively high likelihood, already gets a relatively good score from our SSE fitness function. Thus, individuals which take this form early on in the evolution process will be kept around, and new mutations will be applied to them, which can eventually result in these individuals converging to the exact solutions given above. By contrast, the functions \(f(x) = \frac{1}{x}\) or \(f(x) = \frac{1}{x^2}\), which are natural intermediary

\(^3\)The unary minus operation simply flips the sign of its argument.
steps on the way to evolving the function $\frac{1}{x^2+1}$, will receive very bad fitness scores due to the vertical asymptote at $x = 0$, and so these intermediary functions which connect the population of randomly generated functions with the non-trigonometric representation of our generative function are likely to be quickly eliminated from the population whenever they occur in the evolution process. Thus, the only way for this non-trigonometric representation to occur in this scheme would be if the entire function (i.e. both the division by $x^2$ and the addition of a positive number close to 1 in the denominator of the resultant function) were randomly generated all at once, which could happen, for instance, if the entire function were to appear in the initialized population. This is evidently an unlikely occurrence.
Chapter 3

Genetic Programming for Quantum Computers

In this thesis, we are concerned with the intersection of the two exciting research areas outlined above. In particular, we inquire: how are evolutionary algorithms being applied to further our understanding of quantum computation and information, and how might these applications be expanded on? How can traditional GP techniques be adapted in order to effectively develop quantum programs? These questions motivated the work laid out in the subsequent sections.

3.1 Developing Quantum Algorithms

Discoveries of novel quantum algorithms have been limited, with several significant algorithms being discovered relatively soon after the inception of quantum computing (e.g. Deutsch’s algorithm, Shor’s algorithm, the Grover database search algorithm, etc.), and little notable progress on this front being made in the decades since then. According to leading researchers in the field, this is likely because for several reasons the problem of developing new quantum algorithms is quite difficult [16, 19, 8]. Firstly, quantum algorithms are notoriously difficult to understand and non-intuitive. It is a common view that our classically-rooted intuitions tend to fall short when trying to clearly and distinctly understand the inner workings of a quantum program [16, 8]. Furthermore, when a quantum algorithm is developed which is capable of solving some problem, we are only really interested if the quantum program can outperform known classical algorithms for solving this problem [16]. This limitation further complicates the problem of discovering a truly useful
new quantum algorithm.

In light of these difficulties, however, applying GP to this problem is a natural choice. GP doesn’t rely on human intuitions to discover programs. Rather, by mechanically searching the space of possible programs, GP approaches are capable of leveraging computational power to generate algorithms which elude easy intuitive understanding while delivering desirable results. GP approaches can be well suited for searching large complex spaces like the space of quantum programs, so it is hoped that their application therein might lead to the discovery of novel and/or better than previously known quantum algorithms [8].

3.2 Previous Work and Opportunities

Considering how well suited GP techniques seem to be for approaching the problem of quantum algorithm development, there has to date been a fairly small (but growing) amount of research done regarding their application in this area. This section will give an overview of some of the important studies and developments in GP for quantum computers so far. The subsection on Quantum Circuit Synthesis is based on A. Gepp and P. Stocks’ excellent 2009 review of the field [8]. The following subsection about Unitary Evolution refers to some subsequent developments in the field. This section also describes some of the opportunities for improvement and further work in these studies which motivated the work done in this thesis.

3.2.1 Quantum Circuit Synthesis

GP techniques were first applied to the evolution of quantum algorithms by C. Williams and A. Gray as a way to synthesize quantum circuits to implement a known unitary. [8, 23, 22] Williams and Gray designed a GP scheme which would, given a target unitary, search the space of quantum circuits for ones which efficiently implemented that unitary. This scheme did not use traditional parse trees to represent the quantum programs in its population, but rather a linear structure consisting of a sequence of instructions, with each instruction represented by a three-tuple of the following form: [8]

\[
\text{\{quantum gate, [parameters for the gate], set of qubits acted upon\}.}
\]

By drawing from an approximately universal set of gates (the CNOT gate and two other single qubit gates), a sequence of these three-tuple instructions can approximately represent
any unitary. A fitness function was defined based on the similarity of the aggregate action of the evolved circuit to the target unitary corresponding to the program which was trying to be evolved. In their study, Williams and Gray successfully evolved alternative circuits for the quantum teleportation algorithm by implementing their GP scheme on a population of 100 programs, requiring an average of 26.4 generations to find a solution. [23] Among the circuits that were evolved in this study was one which was able to perform the teleportation algorithm using less gates than the previously known algorithm, meaning Williams and Gray’s GP scheme successfully discovered a more efficient quantum teleportation circuit than was previously known [8].

Williams and Gray’s approach entails a particular kind of application of GP to quantum programs which we will refer to as quantum circuit synthesis, which is characterized by the use of a GP scheme to decompose a quantum algorithm with a known unitary representation into a circuit composed of elementary quantum gates, generally with the goal of discovering simpler and more efficient circuits to implement the given quantum algorithm.

3.2.2 Genetic Programming for Quantum Computers

Around the same time that Williams and Gray were developing their quantum circuit synthesis GP scheme, L. Spector et al. were researching applications of GP to automatically generate quantum programs [19, 20, 21]. In 2004, Spector published a book [18] which summarized his extensive work on GP for quantum computers, and is considered to have provided the main foundation for the field. [8] Spector et al. also developed a software suite known as QGAME for implementing his GP scheme to evolve quantum circuits.

Throughout Spector et al.’s work in this field, three different GP schemes were designed and applied to the problem of automatically generating quantum computer programs. Here we will outline some things which these schemes have in common, before detailing some key differences between them.

First, Spector et al. take a so-called developmental approach to genetic programming, in which the data structures which are being operated in the GP algorithm are classical programs which, when executed, construct quantum gate arrays, which are in turn evaluated for fitness [18]. The developmental programs themselves are essentially a sequence of instructions for assembling a quantum circuit (i.e. for constructing a series of unitary matrices).

Spector et al.’s work is also consistently characterized by a standardized fitness function
incorporating the following three fitness parameters which are to be minimized:

- **Hits**: The total number of fitness cases (denoted $\text{numcases}$) minus the number of cases for which the program gives a correct answer with probability $p > 0.52$.

- **Correctness**: $\sum_{i=1}^{\text{numcases}} \max(0, \text{error}_i - 0.48) \max(\text{hits}, 1)$. This formula is designed to capture the error probabilities that are greater than 0.48, weighted by the total number of hits.

- **Efficiency**: The number of quantum gates in the circuit.

These components are combined and weighted based on their relative importance in a particular implementation. For example, the correctness of circuits will only be compared if they already have the same hits value, and their efficiency will only be compared if the other two parameters are the same.

Finally, all of Spector’s GP for quantum programming schemes use a tournament selection method with a tournament size of five.

There are several important ways in which Spector et al.’s three quantum GP models are different from each other. Some key differences, as well as some results from implementations are outlined below:

**Spector’s first quantum GP scheme:**

The first model, detailed in [19, 21], uses a standard LISP-based parse tree structure for representing the programs, with nodes calling gate building functions, iteration structures, or arithmetic operators. This model was applied to evolve solution circuits for Deutsch’s problem and for a problem known as the *scaling majority-on problem*. With a population size of 10,000, this approach successfully evolved a better-than-classical solution circuit for Deutsch’s problem, however it failed to produce a better-than-classical solution to the scaling majority-on problem [8].

**Spector’s second quantum GP scheme:**

Spector et al.’s second model represents its programs using a stack-based linear structure. This model stores arguments for its functions externally in a global stack which the functions point to, and represents programs as linear sequences of operations, rather than as parse trees. Using this model, Spector et al. successfully evolved an algorithm that solved the four-item database search problem faster than any classical algorithm. Notably, the solution algorithm that was evolved for this problem has the same efficiency and a very similar structure to the 4 item version of Grover’s search algorithm [8].

**Spector’s third quantum GP scheme:**
The final model bears a close resemblance to the second one, but with a few key differences. Firstly, this modified scheme uses a program structure that is still linear, but no longer stack-based: the functions do not call on arguments that are stored externally in a stack. Rather, parameters which determine the arguments of the functions are included in the definition of the function itself. This stackless linear GP model was applied to successfully discover a previously unknown better-than-classical solution to the And-Or Query problem. Unlike the other two, this third GP model supported the use of variable length program representations and variable length crossover operators. Additionally, this third model featured support for single qubit partial measurements that are taken in partway through the algorithms execution rather than at the end of it. Spector et al. found that this third scheme yielded better results when applied to the And-Or Query problem when compared to the previous stack-based model, and thus claim that this stackless linear model is probably the best of their quantum GP schemes.

Spector’s discovery of a better-than-classical solution for the And-Or Query problem is a concrete example of how GP can successfully find new quantum algorithms which approach existing computing problems in novel ways. This result and the others described above are evidence that GP has real potential for application in the development of quantum algorithms.

In 2001, B. Rubenstein published a study in which a stackless linear GP approach is applied to the problem of generating a maximally entangled quantum state. Rubenstein’s approach, like Williams’, entailed a three-tuple representation of gate operators, except the three-tuples were encoded together as a single bit string rather than as a set of three parameters. The three-tuples employ gates from an unspecified set which includes CNOT, Hadamard, and partial measurement gates. Rubenstein’s approach uses the error of the evolved circuit as its sole fitness function, which is calculated using the formula

$$error = \sum_{i=0}^{k-1} \sum_{j=0}^{2n-1} (o_{ij} - d_{ij}),$$

where $i$ indexes the input cases out of $k$ total cases, $j$ indexes the computational basis states in a quantum system of $n$ bits, and $o_{ij}$ and $d_{ij}$ are the observed and desired amplitudes respectively, corresponding to the $i^{th}$ input case and $j^{th}$ basis state. As mutation operators, the scheme employed unary mutations with low probability, and more often crossover opera-

---

1 The And-Or Query problem is to determine if the Boolean function $(f(0) \lor f(1)) \land (f(2) \lor f(3))$ is true for a given function $f : \{00, 01, 10, 11\} \rightarrow \{true, false\}$. 27
tions on the different parts of the bit strings (i.e. the parts specifying which gate, parameters for that gate, and target qubits).

Using a population of 5000, Rubenstein successfully evolved circuits for the most efficient known quantum algorithms for producing maximally entangled states on two, three, four, and five qubit systems. From these results, deductions were made about arbitrary sized maximal entanglement producing circuits [8].

Lukac and Perkowski have also presented studies in applying genetic algorithms to quantum circuits [14, 15]. In their approach, quantum circuits were represented by ordered arrays of strings of quantum gates, where each element in the array corresponds to a specific point in time in the execution of the quantum algorithm when the gates from that string would act on the qubits. Each string was made up of an ordered list of gates, where the \( i^{th} \) entry in the list indicated a gate acting on the \( i^{th} \) qubit in the register. The gates were randomly chosen from a large set of various one, two, and three qubit gates, which included a so called no-op or wire gate which simply applied the identity operator to that qubit. The fitness function used in this implementation was similar to the one used in both Rubenstein and Williams’ schemes, and variation operators included both unary mutation and crossover operators. The populations of data structures were small compared to the studies described above, with runs using population sizes of 50 and 100 individuals.

Lucak and Perkowski applied their scheme to evolve various circuits, including some single gate circuits, the quantum teleportation circuit (as was evolved by Williams in [23]), and circuits for generating maximally entangled states in a three and four qubit system (as was evolved by Rubenstein in [17]). For each of these target algorithms, the desired quantum circuit was evolved in similar or less time than previously published. Interestingly, a higher probability of mutation was found to decrease the number of generations and the real-time required for the successful evolution of a solution circuit [8].

Leier and Banzhaf presented two GP schemes for evolving quantum algorithms, one of which used a linear structure, while the other used a linear-tree structure. Both GP schemes were applied to the Deutsch-Josza problem (a scalable version of Deutsch’s problem where the function has \( > 2 \) outputs) and the 1-SAT problem.\(^2\)

The linear-tree based GP algorithm successfully generated circuits that were essentially the same as the known solutions for both problems, although not every run found a solution for the Deutsch-Josza problem. One interesting finding in this study was that the inclusion of partial measurement gates did not cause the algorithm to converge to solutions any quicker.

\(^2\)For a detailed description of the 1-SAT problem, see [12].
Furthermore, it was concluded that the added flexibility of the linear-tree representation did nothing to increase the effectiveness of the algorithm for solving these problems when compared to the strictly linear representation [8].

Leier and Banzhaf also present a study of the search space of quantum algorithms from an evolutionary algorithms perspective [13]. With the Deutsch-Josza problem as a case study, they conclude that evolutionary searches for larger quantum algorithms are very difficult search problems because of the formidable complexity of the search space. In this study, they also point out that the choice of gate set and of variation operators (i.e. choice of representation) is very important, and significantly impacts the efficiency of the search3 [8].

Massey, Clark, and Stepney also published extensively in the field of evolving quantum algorithms. Massey et al., like Spector et al., developed a software suite (called QPACE) specifically designed for evolving quantum algorithms. Four different implementations of QPACE have been released, with each subsequent release containing updates and improvements to the previous version. All of the implementations use a tree-based representation for the data structures, although the precise specifications of this representation change from implementation to implementation: QPACE I uses a simple parse tree representation, QPACE II represents its programs as trees of gates, QPACE III represents its data structures as trees of functions which generate gates and terminals which pass parameters to these functions, and QPACE IV uses the same representation as QPACE III except with additional support for arithmetic and iterative functions, and variables (such as iteration counters) as variables. These features were included in QPACE IV with the specific objective of generating scalable quantum algorithms: the code which it generates, it was hoped, could be executed to produce a set of quantum algorithms which solve different instantiations of a scalable problem [8].

Besides these differences in representation, the implementations all included a gate set of various one and two qubit gates. Also, in each implementation QPACE uses a fitness function that resembled the one used by Spector et al. for the generation of probabilistic solutions (i.e. circuits which give the right answer more than 50% of the time), and a fitness function that resembled the one used by Rubenstein for finding deterministic solutions. As for variation operators, QPACE employs a crossover operator that swaps subtrees between data structures, as well as various unary mutation operators.

Massey et al.’s most significant result was that they were able to use QPACE IV to evolve

---

3This is a well known principle of evolutionary computation in general, as was described in the previous chapter.
a Quantum Fourier Transform algorithm for any number of qubits (i.e. a scalable algorithm) which was as efficient as the best known algorithms for performing this transform. However, this was only achieved by adding some problem specific assistance into the GP system: the fitness function included a penalty for individuals which had a number of swap gates that didn’t match the already known correct amount [8].

3.2.3 Quantum Genetic Programming as Unitary Evolution

While the studies detailed above approached quantum algorithm evolution primarily as a question of generating quantum circuits (i.e. sequences of quantum gates), there are in fact other ways to approach this question. In algebraic terms, a quantum circuit is simply a series of unitary matrices which act on the input state vector and bring it to a desired output state. Since the composition of two unitary operators is itself unitary, the combined action of the series of unitary operators is itself simply a unitary operator. So, the problem of evolving quantum algorithms for a certain application can be thought of as the problem of finding the appropriate unitary operator which brings the input state to the desired output state. This more general way of looking at quantum circuits yields a less constrained way of approaching the problem of evolving quantum programs: rather than having our evolutionary approaches construct sequences of gates which in aggregate act on the input state to map it to a desired output state, we can simply have it construct a unitary operator which encompasses this aggregate action of the circuit. Once a suitable unitary operator is discovered which has the desired aggregate action, it could in theory then be decomposed into a quantum circuit if that is required. This decomposition of the unitary into a sequence of gates is however, crucially, not an integral part of the problem of finding new quantum programs. Thus the problem of quantum algorithm evolution can be reformulated as a problem of evolving unitary operators instead of as a problem of evolving quantum circuits.

Hutsell and Greenwood were the first to take advantage of this more abstract way of looking at quantum algorithm evolution by, in a 2007 paper, presenting an evolutionary algorithm scheme for evolving arbitrary unitary matrices which constituted the aggregate action of quantum circuits [10]. In constructing their evolutionary representation, Hutsell and Greenwood draw from work done by K. Zyczkowski and M. Kus developing a method for generating random unitary matrices [24]. This method in turn takes advantage of A. Hurwitz’ parameterization of the unitary group $U(N)$ using compositions of elementary unitaries. In an 1897 paper, Hurwitz proved that any $N \times N$ unitary matrix $U$ can be decomposed as
a product of $N(N - 1)/2$ elementary special unitary matrices which apply Euler rotations to 2-dimensional subspaces of the matrix space. These nonzero entries of these elementary unitary matrices are given by the following rules:

$$
E_{ll}^{(j,k)}(\phi, \psi, \chi) = 1, \text{ for } l = 1, ..., N, \ l \neq j, k \\
E_{jj}^{(j,k)}(\phi, \psi, \chi) = \cos(\phi)e^{i\psi} \\
E_{jk}^{(j,k)}(\phi, \psi, \chi) = \sin(\phi)e^{i\chi} \\
E_{kj}^{(j,k)}(\phi, \psi, \chi) = -\sin(\phi)e^{-i\chi} \\
E_{kk}^{(j,k)}(\phi, \psi, \chi) = \cos(\phi)e^{-i\psi}
$$

(3.2)

Where $\phi \in [0, \pi/2]$ and $\psi, \chi \in [0, 2\pi)$. In Hurwitz’ parameterization, a total of $(N - 1)N/2$ elementary operators are grouped together into $N - 1$ “composite rotations”, denoted $E_1, E_2, ...E_{N-1}$, where:

$$
E_1 = E^{(1,2)}(\phi_{1,2}, \psi_{1,2}, \chi_{1,2}), \\
E_2 = E^{(2,3)}(\phi_{2,3}, \psi_{2,3}, 0) \times E^{(1,3)}(\phi_{1,3}, \psi_{1,3}, \chi_{1,3}), \\
: \\
E_{N-1} = E^{(N-1,N)}(\phi_{N-1,N}, \psi_{N-1,N}, 0) \times E^{(N-2,N)}(\phi_{N-2,N}, \psi_{N-2,N}, 0) \\
\times (\ldots) \times E^{(1,N)}(\phi_{1,N}, \psi_{1,N}, \chi_{1,N})
$$

(3.3)

The decomposed representation of unitary operator $U \in U(N)$ can then be given as the product of all of these composite rotations with a complex phase coefficient$^4$:

$$
U = e^{i\chi_{1,1}}E_1E_2E_3...E_{N-1}.
$$

(3.4)

Because any unitary operator in $U(N)$ can be decomposed as such, and since the matrices in this decomposition are fully defined by their respective Euler angles, it follows that any $N \times N$ unitary matrix can be parameterized by $(N - 1)N/2$ real values for $\phi$, $(N - 1)N/2$ real values for $\psi$, and $N$ real values for $\chi$ (i.e. a total of $N^2$ real parameters). Thus, since we have that any $N \times N$ unitary matrix can be generated by the appropriate choice of values for the $N^2$ real parameters (from parameters’ respective real intervals), then a $N^2$-tuple specifying parameters for $\phi$, $\psi$, and $\chi$ is a complete representation for the set of $N \times N$ unitary matrices (i.e. the function defined by plugging in the values from the $N^2$-tuple into the elementary unitary matrices).

$^4$For a more detailed account of Hurwitz’ unitary parameterization, see [6].
matrices in the parameterization and then composing the resultant matrices together is an onto mapping from the set of such $N^2$-tuples to the set of $N \times N$ unitary matrices).

Below, in Equation 3.5, an example is given of how the $CNOT_{12}$ gate can be represented using Hurwitz’ parameterization:

$$CNOT_{12} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} = E^{(3,4)}(\pi/2, 0, 0)$$

$$= e^{0i\pi} \times E^{(1,2)}(0, 0, 0) \times E^{(2,3)}(0, 0, 0) \times E^{(1,3)}(0, 0, 0) \times E^{(3,4)}(\pi/2, 0, 0)$$

$$\times E^{(2,4)}(0, 0, 0) \times E^{(1,4)}(0, 0, 0)$$

$$= e^{ix_{1,1}} E_1 E_2 E_3$$

The $CNOT_{12}$ gate can thus be parametrized by a 16-tuple of euler angles where all of the angles are 0 except for $\phi_{3,4}$, which is equal to $\pi/2$.

Hutsell and Greenwood’s GP scheme thus uses this Hurwitz’ parameterization as a representation for the quantum algorithms in its population, and Zyczkowski and Kus’ random unitary generation method, which involves randomly sampling these parameters from their respective intervals to construct a random $N \times N$ unitary matrix, is used to initialize this population [24].

The fitness of the evolved unitaries in this scheme is defined by the following function:

$$fitness(U) = \sum_{i=0}^{N-1} \frac{1}{(|c_i|^2 - |c_i^*|^2)^2 + \epsilon}$$

(3.6)

where $c_i$ is the $i^{th}$ entry in the output vector $|\psi\rangle$ yielded by

$$U |\psi_0\rangle = |\psi\rangle , \text{ where } |\psi_0\rangle = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

(3.7)

and $c_i^*$ is the $i^{th}$ entry of a target output vector, which is chosen depending on the specific
Hutsell and Greenwood tested their system on a so called \textit{general problem instance} where a unitary is evolved to map the input state to some particular output state. The target output state was set as $|\psi^*\rangle = \begin{bmatrix} 1 & 0 & 0 & \ldots & 0 \end{bmatrix}^T$, i.e. the state $|0\rangle$ (wherein each qubit in the system is in state $|0\rangle$). The stated goal in the general problem instance is to generate an output state that has a probability of at least 0.7 of yielding the state $|0\rangle$ when measured, meaning that $|c_0|^2 \geq 0.7$. The unitary evolution scheme was run with a population size of 100 until a unitary was discovered that gave an output state that satisfied this condition for the general problem instances on 1, 2, 3, and 4 qubit systems. For the 1, 2, and 3 qubit cases, a satisfactory unitary operator was easily found, however with the 4 qubit case, the evolutionary algorithm began to require very long runtimes to generate a solution, and it was concluded that for the 5 qubit instance of this problem, days of runtime would be required to generate a satisfactory solution, with exponentially increasing runtimes for even larger systems.

Hutsell and Greenwood also applied their scheme to two other problems: evolving a Hadamard gate, and evolving a 2-qubit oracle for Deutsch’s problem.

In the Hadamard evolution problem, a pseudo-Hadamard gate$^6$ was successfully evolved in 100 generations, however the actual Hadamard gate was not evolved.

\textbf{Remark:} Hutsell and Greenwood claim in their paper that the actual Hadamard gate cannot be evolved using Zyczkowski and Kus’ random unitary matrix method since when $N = 2$ the method can only generate anti-symmetric matrices. [10] This is, however, demonstrably not the case: the misunderstanding comes from the fact that in Hutsell and Greenwoods version of the parameterization, they left out the complex phase coefficient which multiplies the elementary rotation matrices in Zyczkowski and Kus’ random matrix generation scheme [24], and so had $U$ parameterized simple by $U = E_1E_2...E_{N-1}$. Using this incorrect form of the parameterization, it is true that for $N=2$, only anti-symmetric $2 \times 2$ matrices can be generated. However, if they had included the complex phase factor in the parameterization, as we see in the correct version of the unitary decomposition in equation 3.4, this would not have been the case. The correct decomposition of a $2 \times 2$ unitary matrix $U$, as given by equation

\begin{equation}
\begin{bmatrix}
1 & 1 \\
-1 & 1 \\
\end{bmatrix}
\end{equation}

\begin{equation}
\begin{bmatrix}
1 & \sqrt{2} \\
\sqrt{2} & 1 \\
\end{bmatrix}
\end{equation}

Note that this fitness function is designed to be maximized rather than minimized.

\textsuperscript{5}Note that this fitness function is designed to be maximized rather than minimized.

\textsuperscript{6}That is, a gate of the form $\frac{1}{\sqrt{2}} \begin{bmatrix}
1 & 1 \\
-1 & 1 \\
\end{bmatrix}$
3.4 is:

\[ U = e^{i\chi_{1,1}} E_1 = e^{i\chi_{1,1}} \begin{bmatrix} \cos(\phi)e^{i\psi} & \sin(\phi)e^{i\chi} \\ -\sin(\phi)e^{-i\chi} & \cos(\phi)e^{-i\psi} \end{bmatrix}, \tag{3.8} \]

and if we let \( \chi_{1,1} = 3\pi/2, \phi_{1,2} = \pi/4, \psi_{1,2} = \pi/2, \) and \( \chi_{1,2} = \pi/2, \) we get

\[ U = e^{3i\pi/2} \begin{bmatrix} \cos(\pi/4)e^{i\pi/2} & \sin(\pi/4)e^{i\pi/2} \\ -\sin(\pi/4)e^{-i\pi/2} & \cos(\pi/4)e^{-i\pi/2} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = H. \tag{3.9} \]

Thus, evidently the Hadamard gate can in fact be generated by appropriate parameters in Hurwitz’ parameterization/Zyczkowski and Kus’ random unitary matrix generation scheme, but couldn’t be in this experiment because of a flaw in Hutsell and Greenwood’s application of this parameterization. This result is to be expected, since we claimed above that Hurwitz’ parameterization is complete for all unitary matrices, and if the simple \( 2 \times 2 \) Hadamard unitary were not representable through this parameterization, we would evidently have a contradiction. (end of Remark)

Finally, when approaching the evolution of a 2-qubit oracle gate for Deutsch’s problem, Hutsell and Greenwood look at the specific case where the function being evaluated is the function \( f(x) = \text{NOT}(x) \). The desired output state for this problem is set as \( |\psi^*\rangle = \begin{bmatrix} 0 & 0 & 0 & -1 \end{bmatrix}^T \), presumably since this is the state outputted by other implementations of Deutsch’s algorithm when evaluated on the function \( f(x) = \text{NOT}(x) \). In 100 generations, an oracle matrix is successfully evolved which, when inserted into Deutsch’s algorithm in the spot where the oracle gate is known to go, completes a circuit which brings the input state \( |0\rangle \otimes |1\rangle \) to \( |\psi^*\rangle \) [10].

Ultimately, in the work of Hutsell and Greenwood, potential can be seen for approaching problems of quantum circuit evolution as a problem of evolving matrices.

In a 2014 paper [11], W. Krawec expanded on Hutsell and Greenwood’s GP scheme for evolving arbitrary unitaries and used it to evolve some quantum unitary operators which represent quantum programs. Krawec, like Hutsell and Greenwood, suggests that there are advantages to looking at evolving quantum operators (i.e. unitaries) instead of quantum circuits, particularly for more abstract applications. Krawec designed a genetic algorithm that could evolve unitaries conforming to a user specified rules list. The rules list can be used to describe different types of problems for the genetic algorithm to generate solutions to in the form of unitary operators.
To use this approach, the user first initializes the system by inputting the dimension of the space which is being operated on, and specifying a set of problem specific operators \( \{ V_i \} \) (for example, oracle calls). A so called **RulesList** is then constructed, in which each **Rule** defines constructions of the operators which are to be tested, in addition to a fitness function for testing test these constructions. The constructions of the operators, which can be defined differently for each rule, are given by an ordered list of what I will refer to as **suboperators**, which are composed together to generate the operator which will actually be tested. These suboperators are chosen from the set containing an \( l \)-tuple of evolvable unitaries, denoted \( \{ U_{j,k} \} \) for \( k = 1, ..., l \), unioned with whatever other problem specific operators have been defined, denoted \( \{ V_i \} \), i.e. the set \( \{ U_{j,1}, U_{j,2}, ..., U_{j,l}, V_1, V_2, ..., V_n \} \). For simple problems which don’t require oracle calls or the like, the constructed operator will simply consist of one evolvable unitary \( \text{Op}_j = (U_{j,1}) \). If we wanted to generate more complex quantum operators which, say, call the oracle \( V_1 \) in the middle, however, the construction would be specified as \( (U_{j,1}, V_1, U_{j,2}) \), and the resultant constructed operator would be \( \text{Op}_j = U_{j,1}V_1U_{j,2} \).

The individuals in the evolving population would, in this case, consist of a double \( \{ U_{j,1}, U_{j,2} \} \).

Once the rule has constructed \( \text{Op}_j \), we calculate its fitness by applying it to an initial state \( \rho_j \) to obtain an output state \( \rho_j' \). This output state is then evaluated based on the fitness criteria embedded in the **Rule**, which take the form of an array, as follows:

\[
\text{Rule}_j = \{ p_1 : [space_{1,1} : basis_{1,1}, space_{1,2} : basis_{1,2}, ...], \quad p_2 : [space_{2,1} : basis_{2,1}, space_{2,2} : basis_{2,2}, ...], ... \}
\]  

with the elements in this array being considered sequentially. Each entry in the **Rule** represents the requirement that if the supspaces \( space_{m,1}, space_{m,2}, \) etc. are measured, we should receive outcome \( |basis_{m,1} \rangle, |basis_{m,2} \rangle \), etc. with probability \( p_m \), where measurements and possible outcomes are in the computational basis. The fitness of the operator under each **Rule** is obtained by taking the sum squared error of the actual probabilities \( q_m \) of these measurement outcomes compared to the desired probabilities \( p_m \) for each entry in the rule, and then averaging the SSE for each entry. Finally, the overall fitness of the operator is given by the average of the fitness scores over all the **Rules** in the **RulesList**.

A variation operator is defined which mutates the parameters which define the unitary by a *step size* which is randomly selected from the interval \( \in [0, \pi/10] \), adding or subtracting this step size from the selected parameters. A crossover operator is also used which swaps parameters from different individuals.
Krawec first evaluated his GP system on the same general problem instance that Hutsell and Greenwood tested their system on, with the slightly stricter stopping condition of bringing the chance of an incorrect measurement result below 10%. On a 16-dimensional Hilbert space (i.e. the 4-qubit problem instance), this stopping condition was achieved in an average of 32 generations, which constitutes a substantial improvement over Hutsell and Greenwood’s implementation which required 460 generations to meet a less stringent stopping condition (i.e. incorrect measurement probability below 30%) for this same problem instance with the same population size. It is uncertain, however, to what degree this speedup is due to structural differences in Krawec’s GP implementation, and to what degree it is a result of him having fine tuned his genetic parameters and mutation operators better. [11]

In particular, Krawec points out that the choice of step size for the mutation operator has a large impact on the convergence of the algorithm, with larger step sizes resulting in serious difficulties with convergence. This can’t be ruled out as a possible cause of the difference in run times since Hutsell and Greenwood do not specify the nature of mutation operators they are using in their scheme.

Krawec also tested his GP system on generating a solution for Deutsch’s problem. For this problem, the oracle operators are encoded as problem specific $V_i$ operators, with $V_0$ and $V_1$ representing the two constant oracles and $V_3$ and $V_4$ representing the two balanced oracles. As in the “more complex” example above, each individual in the population then consists of a double of unitary operators \{$U_{j,1}, U_{j,2}$\}, with four individual rules defined, one to test the fitness of each of the constructions $U_{j,1}V_1U_{j,2}, U_{j,1}V_2U_{j,2}, U_{j,1}V_3U_{j,2}$, and $U_{j,1}V_4U_{j,2}$.

Overall, Krawec’s GP approach builds on the ideas put forward by Hutsell and Greenwood, and refines them into a more sophisticated and powerful scheme. Of particular interest is the added ability to test different constructed operators with multiple evolvable components and problem specific components.

Finally, Bang, Yoo, et. al published a study in 2014 in which a differential evolution algorithm, rather than a GP approach, was used to evolve quantum algorithms as arbitrary unitary operations [4]. The structure of their scheme mirrors Krawec’s in that it evolves a double of unitaries, one of which is applied before a problem specific oracle unitary, and the other of which is applied after. In applying their scheme to the Deutsch-Jozsa problem with a population size of 10, they were on average able to obtain solutions with probability of an incorrect measurement less than 10% within 50 generations for the 3-qubit (8-dimensional) case, 180 generations for the 4-qubit (16-dimensional) case, and 400 generations for the 5-qubit (32-dimensional) case. While this scheme require more iterations to obtain solutions
when compared Krawec’s scheme, and is thus seems to be a more computationally expensive approach to evolving quantum operators, this study provides some insight into the applicability of alternative evolutionary techniques, in particular differential evolution, to this problem. It should also be kept in mind that Krawec’s scheme was acting on a population of 100, and so differences in the speed of the two approaches in finding a solution could be in part attributable to the difference in population size.

3.3 Our Approach

We wanted to create a new GP system that builds on the previous work done in the field of GP for quantum computers and takes advantages of opportunities for improvement and refinement. In this section, we present the system that we created for doing GP for quantum computer programs.

3.3.1 Representation

Like Hutsell and Greenwood, and subsequently Krawec, we take advantage of the fact that quantum algorithms are essentially unitary operators, and so we approached our quantum GP system as essentially a system for evolving unitary operators. We thus represent the quantum algorithms in our genetic population as parameterized \( N \times N \) unitary matrices. Our parameterization is, like the one described above, inspired by Hurwitz’ parameterization of the unitary group, however it differs from that one in some important ways.

Hurwitz showed that any \( N \times N \) unitary matrix can be decomposed as the product of \( N(N−1)/2 \) elementary unitary matrices which act as the identity except for on 2-dimensional subspaces. The action of the elementary unitaries amounts two a \( 2 \times 2 \) special unitary submatrix acting on that 2-d subspace. Thus, the action of an \( N \times N \) unitary matrix which is decomposed as such on a \( N \)-dimensional state vector can be efficiently simulated as follows: we sequentially simulate the action of each elementary unitary operator, where the action of an elementary operator on the \( i-j \) subspace is simulated by taking the \( i^{th} \) and \( j^{th} \) elements from the input vector, combining these two elements into a 2-dimensional vector, left-multiplying this vector by the elementary unitary’s corresponding \( 2 \times 2 \) special unitary submatrix, and then reinserting the updated elements into their respective positions in the input vector. Once this has been done for all of the \( N(N−1)/2 \) elementary unitary operators, then the entire vector is multiplied by the complex phase factor from the parameterization.
Once this is complete, we have approximated the action of the \( N \times N \) unitary matrix that was decomposed into Hurwitz’ parameterization on our input vector with relative efficiency.

Our representation differs from the one directly constructed from Hurwitz’ parameterization in that it uses a more flexible structure for composing the elementary unitary operators together. Rather than using the strictly ordered structure from Hurwitz’ decomposition where \( U = e^{i\chi}E_1E_2E_3...E_{N-1} \), and thus where the first submatrix would act on the 1-2 subspace, the second and third would act on the 2-3 and 1-3 subspaces respectively, and so on, we chose to let each elementary operator act on any \( j \)-\( k \) subspace, with the indices \( j \) and \( k \) then being included as parameters for each submatrix. Adding this flexibility doesn’t undermine the completeness of the representation:

**Proposition:** The more flexible representation still constitutes an onto map from the set of representation parameter combinations to the set of \( N \times N \) unitaries.

**Proof:** Let \( U = e^{i\chi}E^{(j_1,k_1)}E^{(j_2,k_2)}...E^{(j_{N(N-1)/2},k_{N(N-1)/2})} \), where the action of \( E^{(j_l,k_l)} \) for \( l = 1,...,N(N-1)/2 \) is equivalent to a \( 2 \times 2 \) special unitary acting on the \( j \)-\( k \) subspace, like in equation 3.2. Then, let \( j_1 = 1 \) and \( k_1 = 2 \); \( j_2 = 2 \) and \( k_2 = 3 \); \( j_3 = 1 \) and \( k_3 = 2 \); \( ...; j_{N(N-1)/2-1} = 2 \) and \( k_{N(N-1)/2-1} = 3 \); \( j_{N(N-1)/2} = 1 \) and \( k_{N(N-1)/2} = 2 \). Then we have \( U = e^{i\chi}E^{(1,2)}E^{(2,3)}E^{(1,3)}...E^{(3,N)}E^{(2,N)}E^{(1,N)} \), which, by equations 3.3 and 3.4 is equivalent to \( U = e^{i\chi}E_1E_2...E_{N-1} \), which is itself just Hurwitz’ decomposition. Therefore, since the parameters in this more flexible representation can be chosen so that a representation that is equivalent to Hurwitz’ complete representation for \( N \times N \) unitary matrices is constructed, we have that the flexible representation itself is complete for \( N \times N \) unitary matrices. \( \square \)

Furthermore, since this more flexible representation allows for alternative orderings of elementary matrices, and these alternative orderings will still yield unitary matrices since they are still simply products of unitary matrices, we now will have multiple parameter combinations for getting the same aggregate unitary operators. That is to say, this representation is also a many-to-one mapping from the set of representation parameter combinations to the set of unitary matrices. By contrast, since the Hurwitz parameterizations are unique (except for the arbitrariness of the \( \chi \) and \( \psi \) angles when \( \phi = 0 \) or \( \phi = \frac{\pi}{2} \) respectively), the Hurwitz representation for a given unitary is almost a one-to-one mapping. This means that our more flexible representation should have a more searchable topology with less isolated local optima compared to the more rigid Hurwitz representation.

This is because in general, when there are unique representations for each solution, the
search space tends to be more rugose and rich in local optima (i.e. evolutionary dead ends) which can slow down or even trap an evolving population. The many-to-one character of our alternative representation results in a search space with greater dimensionality, and thus a more connected topology. While this more connected topology comes at a price - namely we now need to search a space that is larger as a result of its greater dimensionality - it also can result in the merging of local optima into global optima in the search topology, creating new evolutionary paths out of what were formerly local optima in the search space.

Another benefit to this more flexible representation is that the length of the representation can be tweaked and experimented with. While we need \( N(N - 1)/2 \) elementary operations (plus the complex phase operation) if we wish to guarantee completeness, for certain problems less or more elementary operations may lead to better convergence. Since each elementary operator acts on an arbitrary 2-dimensional subspace instead of a fixed one, elementary operators can be removed without disrupting the representation's ability to act in any particular subspace. Using less than \( N(N - 1)/2 \) elementary operators will not be guaranteed to be able to represent any arbitrary \( N \times N \) unitary matrix, but for many problems the particular unitary we are searching for may not require all of these elementary operations to represent it, and so less elementary operators may be sufficient. Changes in the length of the representation can have a significant impact on the search efficiency and rate of convergence of solutions, and since the length of our more flexible representation can be easily scaled and experimented with, this representation is better equipped to exploit this potential source of speed-up.

Our representation also differs from the Hurwitz representation in its treatment of the elementary unitary operators, or more specifically, the special unitary submatrices from these elementary operators. In Hurwitz' parameterization, each of the \( N(N - 1)/2 \) special unitary submatrices is determined by 2 or 3 real parameters which yield a \( 2 \times 2 \) Euler rotation matrix [10]. A third parameter \( \chi \) is only required for a total of \( N - 1 \) of the elementary unitaries; for the remaining ones we take \( \chi \) to be zero, and thus only require two parameters to define them.

Our representation uses different parameterization for these special unitary submatrices:

**Proposition:** Any \( 2 \times 2 \) special unitary matrix can be written in the form \( U = a_0I + i(a_1X + a_2Y + a_3Z) \) where \( (a_0, a_1, a_2, a_3) \) is a real valued vector of norm 1, \( I \) is the identity matrix, and \( X, Y, \) and \( Z \) are the Pauli matrices.

**Proof:** Let \( U \) be a \( 2 \times 2 \) special unitary matrix. Then we have that \( U \) preserves the length
of vectors in $\mathbb{C}^2$, and has determinant 1. Thus, $U$ can be written in the following form:

$$U = \begin{bmatrix} \alpha & \beta \\ -\beta^* & \alpha^* \end{bmatrix}, \text{ where } \alpha, \beta \in \mathbb{C}, \text{ and } |\alpha|^2 + |\beta|^2 = 1$$

$$= \begin{bmatrix} \alpha_r + i\alpha_i & \beta_r + i\beta_i \\ -\beta_r + i\beta_i & \alpha_r - i\alpha_i \end{bmatrix}, \text{ where } \alpha_r, \alpha_i, \beta_r, \beta_i \in \mathbb{R}$$

$$= \alpha_r \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + i\alpha_i \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} + i\beta_r \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} + i\beta_i \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$$

$$= \alpha_r \mathbf{I} + i(\beta_r X + \beta_i Y + \alpha_i Z) \tag{3.11}$$

Where $(\sqrt{\alpha_r^2 + \alpha_i^2})^2 + (\sqrt{\beta_r^2 + \beta_i^2})^2 = 1$. Thus if we let $a_0 = \alpha_r$, $a_1 = \beta_i$, $a_2 = \beta_r$, and $a_3 = \alpha_i$, we have $U = a_0 \mathbf{I} + i(a_1 X + a_2 Y + a_3 Z)$ where $(a_0, a_1, a_2, a_3)$ is a real valued vector of norm 1. □

Using this alternative parameterization, each elementary unitary can be defined by the 4 parameters of a unit vector in $\mathbb{R}^4$. Since this parameterization, like the Euler angle parameterization of special unitaries used for the elementary operators in Hurwitz’ unitary decomposition scheme, is complete for any $2 \times 2$ special unitary matrix, we can swap it in to the unitary decomposition scheme in the place of the Euler rotation submatrices without effecting its completeness: we can still use the same argument from Hurwitz’ parameterization to prove that any $N \times N$ unitary matrix can be decomposed by $N(N - 1)/2$ such elementary special unitary matrices along with a complex phase factor. This alternative parameterization has the small disadvantage that it requires 4 parameters to define each elementary unitary instead of just 2 or 3, however it has the significant advantage that the only elementary matrices required to implement it are the Pauli matrices along with the identity. In our scheme, we take these 4 parameters from a discretized subset of the real numbers on $[0, 1]$, namely the diadic rationals (up to a chosen depth) on this interval.

So, we can represent an arbitrary unitary matrix by real parameters from $N(N - 1)/2$ norm 1 vectors in $\mathbb{R}^4$ which specify special unitary submatrices, with each vector coupled with indices $(i_k, j_k)$ where $i, j \in \{1, 2, ..., N\}$, $i \neq j$ to specify which 2-d subspace the corresponding special unitary submatrix is acting on, and finally a real number $\chi \in [0, 2\pi)$ to give the complex phase factor out front. Our chromosome for an evolved unitary operator thus contains $4N(N - 1)/2 = 2N(N - 1)$ real vector parameters $\in [0, 1]$, $2N(N - 1)/2 = 40$
\(N(N-1)\) integer parameters \(\in\{1, 2, ..., N\}\), and one real parameter \(\in [0, 2\pi]\) for the complex phase coefficient, meaning a total of \(2N^2 - 2N + 1\) real and \(N^2 - N\) integer parameters. This chromosomal representation for evolvable unitaries requires more parameters than the \(N^2\) real parameters used in Hurwitz’ decomposition, however the amount of parameters needed is only greater by a linear factor, so it is believed that the advantages of this representation with regards to its more connected search topology will outweigh this relatively insignificant increase in memory requirements.

### 3.3.2 The GP System

We designed a GP system to evolve \(N \times N\) unitary operators using the chromosomal representation detailed above, with the code for the GP system written in C. In the evolution model implemented in the system, selection and replacement are performed with size four single tournament selection. This model of evolution updates the population in the following fashion: first, four members of the population are selected uniformly at random. Then, they are sorted based on their fitness scores and then the two best are copied over the two worse. The new copies are subjected to two point crossover, which entails swapping some of the elementary operators which compose them. The results of this crossover are then mutated, making one or two unary mutations, with this number of mutations selected uniformly at random. The choice of one or two mutations is controlled by the MNM parameter, specified subsequently. Mutation has a 50% change of changing a submatrix in some operator while leaving the indices for the subspace it modifies the same, and a 50% of preserving the submatrix but changing which 2-dimensional subspace it is applied to. In order to run the program, values are chosen for:

- **Population size**: the number of individual chromosomes.
- **Decomposition length**: the number of elementary unitary operations simulated in each chromosome.
- **Dimension**: the dimension \(N\) of the Hilbert space \(\mathcal{H}\) that the evolved unitary will be operating in.
- **MNM**: the minimum number of unary mutations in each “produce new variations” step of the algorithm.
- **Runs**: the number times the evolutionary algorithm is to be run before stopping.
In addition to choosing the appropriate values for these settings, a problem specific fitness function must be defined and included.

### 3.3.3 Results

We developed a fitness function for the general problem instance of mapping a given input state $|\psi_0\rangle$ to an output state that, when measured, will give a desired measurement outcome with high probability. The fitness of the unitary is simply given by the probability of observing this desired measurement outcome if, after the unitary is applied, all the qubits are measured in the computational basis. Using this fitness function, we tested our GP system on the general problem instance in order to compare the convergence (and thus search efficiency) under our alternative unitary representation with the convergence of previous implementations for this problem. Here, we let $|\psi_0\rangle = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & \ldots & 1 \end{bmatrix}^T$ and let the desired measurement outcome be the state $|0\rangle = \begin{bmatrix} 1 & 0 & \ldots & 0 \end{bmatrix}^T$ (i.e. the first computational basis state), in order to match the test problem that was attacked by previous unitary evolving schemes. The probability of measuring this outcome is precisely given by the modulus squared of the first entry in the output vector $|\psi'\rangle = U_i |\psi_0\rangle$, and so fitness is evaluated by the following formula:

$$
\text{fitness}(U_i) = |c_1|^2, \quad \text{where} \quad \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_N \end{bmatrix} = |\psi'\rangle = U_i |\psi_0\rangle = U_i \frac{1}{\sqrt{N}} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}
$$

(3.12)

We applied our GP scheme to several variations of this general problem instance. Figure 3.1 details some variations of the problem which we applied our GP scheme to, and gives, for each problem, the average number of mating events\(^7\) which were required to generate a solution unitary which met the stopping condition, i.e. which gives at least a 90% chance of measuring the state $|0\rangle$ when a measurement is made in the computational basis. For each experiment, a population size of 10, and MNM value of 2 were used, and 30 runs were executed.

The rate of convergence of solutions is here measured by the mean number of mating events required to generate an acceptable solution, instead of by the number of generations

\(^7\)Or, equivalently, the average number of calls to the fitness evaluation function, since each mating event involves a fitness evaluation.
<table>
<thead>
<tr>
<th>Dimension</th>
<th>Decomposition length</th>
<th>Mean ME to solution (with 95% CI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>N=16</td>
<td>48</td>
<td>6700 ± 1564</td>
</tr>
<tr>
<td>N=16</td>
<td>120</td>
<td>3767 ± 844</td>
</tr>
<tr>
<td>N=16</td>
<td>192</td>
<td>2033 ± 393</td>
</tr>
<tr>
<td>N=16</td>
<td>240</td>
<td>2200 ± 357</td>
</tr>
<tr>
<td>N=24</td>
<td>112</td>
<td>8733 ± 1753</td>
</tr>
<tr>
<td>N=24</td>
<td>280</td>
<td>4033 ± 970</td>
</tr>
<tr>
<td>N=24</td>
<td>448</td>
<td>2667 ± 424</td>
</tr>
<tr>
<td>N=24</td>
<td>560</td>
<td>1133 ± 124</td>
</tr>
<tr>
<td>N=32</td>
<td>200</td>
<td>12267 ± 2396</td>
</tr>
<tr>
<td>N=32</td>
<td>500</td>
<td>5000 ± 1051</td>
</tr>
<tr>
<td>N=32</td>
<td>800</td>
<td>2500 ± 262</td>
</tr>
<tr>
<td>N=32</td>
<td>1000</td>
<td>2200 ± 256</td>
</tr>
<tr>
<td>N=64</td>
<td>800</td>
<td>22267 ± 5314</td>
</tr>
<tr>
<td>N=64</td>
<td>2000</td>
<td>4900 ± 978</td>
</tr>
<tr>
<td>N=64</td>
<td>3200</td>
<td>3100 ± 172</td>
</tr>
</tbody>
</table>

Figure 3.1: Mean number of mating events required to generate a solution for varying problem dimensions and representation lengths.

(as it is in Krawec’s and Hutsell and Greenwood’s results). This is because our GP scheme does not have generations, but rather a steadily evolving population. These two metrics, however, are commensurable: since the other schemes were acting on populations of 100 individuals, and each generation entails a mutation for each individual in the population, the number of mutations required in these other schemes is given by the number of generations required times 100. Thus, for the 4 qubit (N = 16) case, Hutsell and Greenwood’s scheme required $460 \times 100 = 46000$ fitness evaluations to get a solution that would yield the desired measurement 70% of the time. Krawec’s scheme was much quicker for this case, requiring an average of $32 \times 100 = 3200$ fitness evaluations to get a solution that would result in the desired measurement at least 90% of the time. The largest problem instance which Krawec tackled with his scheme was the $N = 24$ case, a slightly smaller problem than the 5 qubit case (where $N = 32$). For this $N = 24$ general problem, Krawec’s scheme generated a solution in an average of 60 generations, meaning $60 \times 100 = 6000$ mating events.

Comparing the results of our GP experiments with these previously published results, we see that in many cases our scheme is capable of converging to solutions more quickly (i.e. after less fitness evaluations). This decrease in the required number of fitness evaluations to generate an acceptable solution is likely a result of the altered search topology under the
more flexible representation which we developed.

Notably, the number of fitness evaluations required to get an acceptable solution is dependent on the length of the representation that was used. Representations which contained more elementary operations than the $N(N - 1)/2$ that are required to guarantee completeness tended to converge to solutions much quicker than representations with less elementary operations. In fact, for the $N = 16$ case, when our representation length was equal to the 120 elementary operations which were required for completeness, our scheme actually required more mating events than Krawec’s did to find a solution. When the representation length was increased our scheme became more efficient. In the larger problem instances, however, our scheme outperformed Krawec’s even when the representation length was close to the $N(N - 1)/2$ minimum completeness requirement. For the $N = 24$ case, with a representation length of $280 \approx 276 = N(N - 1)/2$, our scheme outperformed Krawecs. An increase in representation length then led to a further increase in efficiency.

Our scheme managed to evolve a solution to the $N = 24$ case after less than one fifth of the fitness evaluations that were required in Krawec’s implementation. Furthermore, solutions were successfully evolved for the 5 qubit ($N = 32$) and 6 qubit ($N = 64$) cases in reasonably short runtimes. This far exceeds the scale of the problems which were able to be handled by the other unitary evolution schemes, which is likely at least in part a result of the better search topology under our representation, although this increased capability may also be partially due to computational efficiencies elsewhere in our algorithm.

The representation used in our GP system, which entails a decomposition of a unitary operator as a user specified number of $2 \times 2$ submatrices, is a sparse approach which enables the evolution of leaner quantum programs. While our approach is complete when the representation length is sufficient (i.e. when there are at least $N(N - 1)/2$ elementary operators and a complex phase coefficient), it is not obvious what representation length will yield the most efficient search topology. This means that the increased speed of our approach is purchased at the cost of the need for some experimentation or, possibly, the development of a more complete theoretical basis for the approach.
Chapter 4

Conclusions And Further Work

4.1 Conclusions

In this thesis we have presented a genetic programming scheme for evolving quantum programs in the form of arbitrary unitary operators. Expanding on the work done by Hutsell and Greenwood, and later by Krawec in developing a unitary operator evolution scheme, we developed an alternative representation for unitary operators which seems to display a better search topology than the representations used in these previous implementations, as evidenced by the quicker rate of convergence in our experiments on the general problem instance.

4.2 Further Research

The unitary evolution scheme described in this thesis has better convergence properties on the problems it was tested on than any other unitary evolving scheme that has been published before. It, however, has presently only been tested on the general problem instance. In future work, additional fitness evaluation functions could be written to test this scheme on other problems, in order to better understand how it performs compared to other schemes and what kinds of problems it is capable of finding solutions for. Some potential fitness functions for future implementations are included in appendix A.

Our quantum GP scheme could benefit from further development in order to expand the kinds of quantum computation problems it can be applied to. Much can be learned from Krawec’s scheme in this regard: his system is equipped to evolve multiple unitaries that can
then be combined with an oracle gate or some other problem specific gate to make complex quantum operators with problem specific unitaries implemented mid circuit. This mirrors the form of many known quantum algorithms, and so if we wish to apply our alternative GP scheme to such problems, it would be necessary to add a similar functionality.

In particular, our implementation could be expanded so that each individual chromosome contains enough parameters to construct two unitary operators, one which is applied before the problem specific oracle operator, and one which is applied after. Taken together, these two unitaries interjected by the oracle gate represent a more complex evolvable structure, which is able to represent solution circuits to a wider range of problems, such as the general version of Deutsch’s problem and the related general scalable Deutsch-Jozsa problem. This would make the GP scheme more well suited to evolving solutions for problems for which details about the solution circuit (such as what the proper input state is) are not known in advance.

The experimental implementations described in the previous chapter also all took standard parameter values that were located during the debugging of the evolution code. Therefore, while the performance of the system is already acceptable, it may be further enhanced by performing a careful parameter study, looking at the impacts of tweaking factors like population size, mutation rate, representation length, and tournament size.

It may also be rewarding to do more quantitative analysis about the relative convergence of different machine learning approaches for generating quantum operators. The work done by Bang et. al shows that differential evolution can be a useful tool for evolving quantum unitary operators, although at present the relative advantages and disadvantages of taking such an approach instead to a GP approach for evolving unitaries remain underexplored.

Finally, our matrix evolving scheme may also have applicability outside of the field of quantum computing. The evolvable matrix representation described in this paper is very flexible, and could potentially be adapted for use in research areas in which matrices with particular properties are needed.
Bibliography


Appendix A

Potential Fitness Functions for Future Implementations

A simple fitness function for the problem of evolving a unitary to map a given input state to an arbitrary output state (including superposition states) could employ the following formula:

\[
fitness(U_i) = \frac{1}{N} \sum_{i=1}^{N} (|\alpha_i|^2 - |\alpha'_i|^2)^2
\]

(A.1)

where \(\alpha_i\) is the \(i^{th}\) entry in the output vector \(|\psi\rangle\) yielded by \(U_i|\psi_0\rangle = |\psi\rangle\), and \(\alpha'_i\) is the \(i^{th}\) entry of the target output vector \(|\psi'\rangle\).

Another potential fitness function is described below which could be applied to evolving solutions to a particular subproblem of Deutsch’s problem, in which it is assumed that the input is prepared in the state \(|\psi_0\rangle = \frac{1}{\sqrt{2}}(|+\rangle)(|-\rangle) = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & -1 & 1 & -1 \end{bmatrix}^T\), before the oracle \(U_f\) is applied.\(^1\) The evolved unitary is applied to the state after one of the four oracle matrices has been applied, to decode the resultant state and predict whether the oracle gate corresponded to a function that was constant or balanced. In this problem, we measure the first qubit to get the prediction of whether the function is constant or balanced: if the state \(|0\rangle\) is measured, then this is interpreted as a prediction that the oracle function is constant, and if the state \(|1\rangle\) is measured, this is interpreted as a prediction that the function is balanced. Therefore the fitness function has to evaluate 4 possible cases, corresponding to the 4 possible oracle functions on two bits:

\[
\begin{align*}
    f_1(0) = 0 &= f_1(1), \\
    f_2(0) = 0 &\neq 1 = f_2(1), \\
    f_3(0) = 1 &\neq 0 = f_3(1), \text{ and } f_4(0) = 1 = f_4(1).
\end{align*}
\]

The fitness function evaluates each of these cases individually for each \(U_i\) in the population:

\(^1\)This preparation is considered to be part of the solution algorithm when we are looking at the standard version of Deutsch’s problem.
for each case (i.e. for \( j = 1, 2, 3, 4 \), \( U_{f_j} \) is applied to the input state \( |\psi_0\rangle \), then \( U_i \) is applied, then finally the probability of measuring the first qubit to be in state \( |0\rangle \) is extracted from the resultant state vector, and the squared difference is taken with the \textit{desired} probability for this measurement outcome, denoted \( p'_j \) (\( p'_j = 1 \) for the constant oracles \( U_{f_j} \) where \( j = 1 \) or \( j = 4 \), and \( p'_j = 0 \) for the balanced oracles where \( j = 2 \) or \( j = 3 \)). In particular:

\[
fitness(U_i) = \sum_{j=1}^{4} |p_{i,j} - p'_j|^2 \tag{A.2}
\]

where \( p_{i,j} \), i.e. the probability of measuring the first qubit of our output state to be in the state \( |0\rangle \), is given by:

\[
p_{i,j} = |c_1^{(i,j)}|^2 + |c_2^{(i,j)}|^2, \tag{A.3}
\]

and where \( c_1^{(i,j)} \) and \( c_1^{(i,j)} \) are the first two entries of the output state \( |\psi_{(i,j)}\rangle \), given by:

\[
\begin{bmatrix}
    c_1^{(i,j)} \\
    c_2^{(i,j)} \\
    c_3^{(i,j)} \\
    c_4^{(i,j)}
\end{bmatrix} = |\psi_{(i,j)}\rangle = U_i U_{f_j} |\psi_0\rangle. \tag{A.4}
\]