Learning Programs For The Quantum Computer

by

Olivia–Linda Enciu

Bachelor of Science (Honours), University of Toronto, 2010

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the Program of

Computer Science

Toronto, Ontario, Canada, 2014

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public.

Learning Programs For The Quantum Computer

Master of Science 2014

Olivia–Linda Enciu

Computer Science

Ryerson University

Abstract

Manual quantum programming is generally difficult for humans, due to the often hard-to-grasp properties of quantum mechanics and quantum computers. By outlining the target (or desired) behaviour of a particular quantum program, the task of programming can be turned into a search and optimization problem. A flexible evolutionary technique known as genetic programming may then be used as an aid in the search for quantum programs. In this work a genetic programming approach uses an estimation of distribution algorithm (EDA) to learn the probability distribution of optimal solution(s), given some target behaviour of a quantum program.

Acknowledgements

FINALLY!

I would like to thank my advisor, Dr. Marcus dos Santos. Many months ago when I first mentioned my quantum interests during one of our meetings I really didn't even hope you would take it too seriously. I'm glad you did. Thank you for all your help. Thanks for the patience, guidance and _many_ awesome conversations!

I would also like to thank my committee members: Dr. McInerney, Dr. Ding and Dr. Harley, for their genuine interest in my thesis and for their time and valuable questions and comments.

The authors of QCircuit really deserve a big thank you, for without this brilliant package the best parts of this thesis would have looked a whole lot worse.

Thanks to Ryerson University for allowing me this opportunity to study what I love. Thanks to the friends I made at Ryerson, especially those in ENG251. It's been great sharing this grad school experience with you all. I will look back fondly on my time spent here.

A sincere thank you to Iron Maiden for inspiration and the soundtrack to this thesis.

Thanks to my sister Teona for always being positive when I needed encouragement and to little Joe, Felix (a.k.a. Schrödi) for company during long coding nights. To my wonderful parents, I'm not sure how you put up with me, but know that it's _highly_ appreciated and I could not have got this far without your support! :). I'm afraid I'm still not done with school, so I look forward (as I'm sure you do) to your continuing support during whatever comes next!

Up the Irons!

Linda Enciu
(On a GO bus, somewhere between Markham and Toronto)
Friday, Jun. 13th, 2014

Dedication

To Rex

Contents

List of Tables

List of Figures

List of Appendices

Chapter 1

Introduction

Can we automatically program quantum computers using machine learning ?

1.1 Thesis statement

Quantum computer programming involves the manipulation of information on a machine called a quantum computer, whose foundation is quantum mechanical. The bit is replaced by the qubit as the smallest unit of information. A qubit's capabilities and limitations are closely related to its physical realization, which is directly subject to the laws of physics; thus, the role physics has in computation cannot be abstracted for quantum computation as is usually done for classical computation.

Strange behaviours arise in the context of quantum mechanics. A qubit may exist in a superposition of two mutually exclusive states, which equips the qubit with an inherent ability for parallelization. This could lead to a computational advantage for certain problems. Two or more qubits may become entangled, which means that mutually-dependent relationships are forged among the qubits, such that the information they convey as a system can no longer be broken up and attributed to individual qubits. Entanglement can be turned into a computational resource. Quantum physics grants us new abilities; however, it also takes away some powers we take for granted in classical computing, such as the freedom to observe a qubit without loss of information and the ability to copy an arbitrary qubit.

Due to superposition and entanglement a quantum system can only be classically simulated with exponential costs in terms of time and memory. A true quantum computer would render the study of quantum systems possible. Such a computer would require effective quantum software.

Programming can refer to something as trivial as a NOT gate, whose sole purpose is

to flip a bit. When we have various inputs and multiple gates acting on the inputs, we can form full circuits, such as a half-adder for arithmetic, or a flip-flop, which gives us memory. Given our more complex circuits we can arrange them in ways which produce actual programs. If we come up with rules for the expected behaviour of a program, independent of the size of the input and the available hardware, then we have defined an algorithm.

Quantum programming differs from classical programming in that we must take into account and understand quantum phenomena, ideally learn to use these to some sort of an advantage and beware of easily abounding errors that might disturb our computations. Experience over the last three decades shows that quantum programming is not trivial for someone who is conditioned to the logic of classical physics, which is something most of us humans will have trouble disclaiming. The purpose of this work is to explore the possibility of quantum programming for quantum computers with the aid of classical computers and classical machine learning.

Automatic generation of programs is the specialty of a metaheuristic known as *genetic programming* (GP). A genetic program starts off by initializing (usually randomly) a set of potential solutions to a given problem. Problem solutions are usually described by a collection of features, which can be represented by a tree, a linear linked-list, an acyclic graph, or some other useful data-structure. A GP then attempts to improve the solution set in an iterative process in which it identifies the better solutions in the set and encourages the propagation of features that make up these solutions by preferentially applying transformations (or search operators) to the current set. To compare solutions, a fitness function is used, which maps each candidate to a fitness value, indicating how closely the candidate approximates an ideal solution.

GP has successfully generated quantum programs in the past, but previous work clearly suggests that the task is difficult. Estimation of distribution algorithms (EDA) allow a GP to model the probability distribution of a set of potential solutions and using machine learning gradually approach the ideal distribution, from which a solution might be sampled. An EDA-based GP differs from normal GP in that it does not use explicit search operators. Programs often exhibit patterns and repetition in their code. Through an EDA-based GP, patterns might be identified and exploited to improve the search for quantum programs. The work of this thesis uses EDA-based GP to learn quantum programs, starting with simple quantum circuits and building up to more complex programs. Our hypothesis for this thesis is thus made of two parts:

1. Quantum programs exhibit sequential patterns and relationships between their functions and inputs which can be learned by an EDA-based GP to help automatically generate programs.

2. A stochastically-driven GP engine with an underlying learner to guide perturbation of features could have an advantage over one without the learner.

1.2 Contributions

For the purpose of this thesis a quantum program is represented as a linked-list of function nodes, where a function is a *quantum gate* and each node encodes the information of where and how it should be applied. As opposed to previous work, we decided to use EDA-based GP to help automate the search for quantum programs. The contributions of this thesis are as follows:

- We have designed a framework for studying and generating quantum programs using a learner.
- Three EDA-based GP variants were developed and employed to learn the optimal solutions to a variety of problems:
  1. **EDA-QP** (EDA quantum program evolver) tries to find relationships between the function nodes and their inputs.
  2. **ngram-QP** (N-gram quantum program evolver) looks at sequences of quantum gates and tries to generate programs through 2 and 3-tuple combinations of gates.
  3. **HQP** (Hybrid-EDA quantum program evolver) uses a guided stochastic search operator enhanced by an underlying EDA for a greater capacity to explore.
- We have developed a quantum computer simulator named `smallqc` useful for evolving, testing and running quantum programs.
- We have incorporated an additional fitness component which attempts to promote function sequences that could potentially result in entanglement.
- Our experimental results support the following two ideas.
  1. Structures in (quantum) programming are difficult to capture in circuits at the low-level.
  2. Fully-automatic quantum programming is difficult for GP in general and more so for EDA-QP than for ngram-QP and HQP.

1.3 Roadmap

Chapter 2 presents background information on the fields of quantum computation and evolutionary computation. Chapter 3 is a survey of related work. Chapter 4 introduces the methodology and algorithm employed and breaks down the implementation of our approach. Chapter 5 consists of experiments, results, an analysis of the results and an evaluation of the approach used. Conclusions and suggestions for future work are given in Chapter 6, which then ends with a short summary of the entire work completed.

Chapter 2

Background

This chapter provides all the necessary background material relevant to the rest of this thesis. It begins with an introduction to *quantum computation*, from a computer scientist's point of view. This should be accessible to someone without a physics background. Section 2 gives an introduction to *evolutionary computation*, with an emphasis on *genetic programming*.

2.1 Quantum computing

Quantum computing is an eclectic field at the intersection of quantum physics, computer science and information theory. Quantum computing is based on *quantum mechanics*, which is the branch of physics describing the world at the atomic level. It is an alternative to classical computing, where the underlying models of computation are enhanced to support quantum phenomena and the theoretical and practical aspects have a much closer relationship. Quantum computing encompasses all aspects of theoretical quantum computer science, quantum information processing and quantum technology.

2.1.1 History

Quantum mechanics arose from a struggle to explain certain peculiar behaviours in nature (especially those of light), which failed to make sense in the context of classical physics. At the beginning of the 20^{th} century a number of experiments[1] baffled physicists, as unexpected results suggested that a revision of classical Newtonian dynamics was in order. Empirical evidence led to the realization that light exhibits both wave-like and particle-like behaviours[26]. Thus, light can be infinitely spread out, but it can also be deterministically localized. This is in fact true of any matter. This paradox is what

[1] The interested reader may consult a physics textbook, such as *Principles of Physics* by Serway and Jewett [61] and find out more about the *Open-Slit Experiment* and the *Compton Experiment*, which revealed light in the act of behaving both as a wave and a particle (called a photon).

gives quantum computation its quirk and its power. Quantum mechanics, though non-intuitive, has nonetheless been experimentally verified and (at least) at the time of this thesis forms our best approximation to the dynamics of the universe [45]. Quantum computing was initiated with the goal of building a machine capable of *using* quantum mechanics rather than have to *simulate* it, as simulations of quantum systems turned out to be prohibitive, in terms of time and space.[2]

2.1.2 The quantum computer

A *quantum computer* is much like a classical computer, except that its foundation is quantum mechanical and it is able to use and exploit quantum phenomena directly, in order to compute [45]. In all other respects a quantum computer is just like a classical computer; there is nothing that a quantum computer can or cannot do which a classical computer cannot or can do, but a quantum computer is theoretically believed to be superior in space and time efficiency [45, 74].

2.1.3 Quantum computation

Quantum computation is a branch of quantum computing that studies both the practical and theoretical computational capabilities of a quantum computer [54].

Quantum notation

Quantum mechanics and quantum computation use a special notation called *bra-ket notation* or *Dirac notation*[3], which is worth an introductory discussion. In quantum mechanics, a *ket* is a column vector of complex numbers[4] in a *Hilbert* space[5] and is usually denoted by $|x\rangle$, where x is simply a label for the column vector. A ket has a corresponding *dual* vector called a *bra*, denoted by $\langle x|$, which is simply the transpose of the *complex conjugate* of the ket[6].

For example, a ket of dimension n can be written as:

$$|x\rangle = \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_{n-1} \end{pmatrix}, \tag{2.1}$$

[2] Richard Feynmann first suggested the quantum machine in 1982 [45, 12].

[3] The terminology and notation for the mathematics of quantum mechanics is due to Paul Dirac. [45]

[4] For a review of complex numbers and some important related results that will be used throughout this thesis, consult Appendix A.

[5] A Hilbert space [41, 45] is an infinite complex vector space, in which quantum mechanics roams, but for the purposes of quantum *computation* we can always regard it as a finite dimensional vector space.

[6] Thus, since a ket is a column vector, its corresponding bra will always be a row vector.

where all $z_i \in \mathbb{C}$. The bra of this ket is:

$$\langle x| = |x\rangle^\dagger = \begin{pmatrix} \bar{z_0} & \bar{z}_1 & \dots & \bar{z}_{n-1} \end{pmatrix}, \tag{2.2}$$

where $\bar{z_i}$ denotes the complex conjugate of z_i and $(\cdot)^\dagger$ denotes the *conjugate transpose*. The terms ket and vector will be used interchangeably, throughout this thesis.

A normalized ket is a unit vector, or a vector whose *norm* or *modulus* is equal to unity. The modulus is induced by an *inner product* on the vector space. An inner product in the quantum case is a transformation or a many-to-one function that takes two kets of the same dimension and maps them to a complex number:

$$\langle \cdot | \cdot \rangle : \{|\psi\rangle, |\phi\rangle\} \mapsto \langle \phi | \psi \rangle \in \mathbb{C}. \tag{2.3}$$

The inner product on kets is also called the *braket*, since it is the product of a bra and its ket. The modulus of a ket is analogous to the modulus of a complex number. For example, the modulus of a ket $|x\rangle$ of dimension n is:

$$\begin{aligned} \|x\| &= \sqrt{\langle x|x\rangle} \\ &= \sqrt{\begin{pmatrix} \bar{z}_0 & \bar{z}_1 & \dots & \bar{z}_{n-1} \end{pmatrix} \begin{pmatrix} z_0 \\ z_1 \\ \vdots \\ z_{n-1} \end{pmatrix}} \\ &= \sqrt{\bar{z}_0 z_0 + \bar{z}_1 z_1 + \dots + \bar{z}_{n-1} z_{n-1}} \\ &= \sqrt{\|z_0\|^2 + \|z_1\|^2 + \dots + \|z_{n-1}\|^2} \end{aligned} \tag{2.4}$$

If the inner product of two kets, $\langle x|y\rangle$, is 0, then we say that $|x\rangle$ and $|y\rangle$ are *orthogonal*. If the inner product of a ket with itself, $\langle x|x\rangle$, is 1, then we say that $|x\rangle$ is *normal*. A set of kets $|x_1\rangle, |x_2\rangle, ...|x_n\rangle$ is *orthonormal* iff:

$$\langle x_i | x_{j \neq i} \rangle = 0$$

and

$$\langle x_i | x_i \rangle = 1$$

for all $1 \leq i, j \leq n$.

Composition

Given two arbitrary kets $|x\rangle = \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix}$ and $|y\rangle = \begin{pmatrix} y_0 \\ \vdots \\ y_{m-1} \end{pmatrix}$ of arbitrary lengths n and m, respectively, a composite ket $|xy\rangle$ of length $n \times m$ may be formed through a *tensor product* of the two kets, denoted by $\otimes$:

$$\begin{aligned} |xy\rangle = |x\rangle \otimes |y\rangle &= \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix} \otimes \begin{pmatrix} y_0 \\ \vdots \\ y_{m-1} \end{pmatrix} \\ &= \begin{pmatrix} x_0 \\ \vdots \\ x_{n-1} \end{pmatrix} \otimes |y\rangle = \begin{pmatrix} x_0|y\rangle \\ x_1|y\rangle \\ \vdots \\ x_{n-1}|y\rangle \end{pmatrix} = \begin{pmatrix} x_0y_0 \\ x_0y_1 \\ \vdots \\ x_0y_{m-1} \\ x_1y_0 \\ \vdots \\ x_{n-1}y_{m-2} \\ x_{n-1}y_{m-1} \end{pmatrix} \end{aligned} \tag{2.5}$$

$|xy\rangle$ is thus a ket that contains every possible product of an element of $|x\rangle$ and an element of $|y\rangle$. As there are $n \times m$ ways to pair an element of $|x\rangle$ with an element of $|y\rangle$, the length of $|xy\rangle$ is thus the product of the lengths of $|x\rangle$ and $|y\rangle$. The tensor product may be written as $|xy\rangle$, $|x\rangle|y\rangle$ or $|x\rangle \otimes |y\rangle$. We will often prefer the common notation $|xy\rangle$, due to the reduced clutter.

2.1.4 Information and qubits

Quantum computation is done on *information* stored in **qu**antum **bit**s (or *qubit*s for short) [45, 54, 74]. As the quantum analogue to the classical bit, the qubit is the smallest unit of information (or memory) in a quantum computer. A qubit has two mutually exclusive *computational basis states*, often labelled $|0\rangle$ and $|1\rangle$, just as a classical bit has the mutually exclusive states 0 and 1 (or *off* and *on*), but a qubit is endowed with an additional power, which is unknown to a classical bit; a qubit has the ability to exist in a *superposition* of its basis states [45, 54, 74]:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle, \tag{2.6}$$

where $\alpha, \beta \in \mathbb{C}$.

Since a single qubit is a combination of bipolar states, any two orthogonal vectors in two dimensions can form a computational basis and any qubit can be written as a linear expansion of such a basis. The labels $|0\rangle$ and $|1\rangle$ are a useful allusion to binary states 0 and 1, but any other labels can be used and any (necessarily orthogonal) computational basis can be used, as long as the use is consistent. The *standard computational basis*[7] is:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{2.7}$$

which can easily be verified to be orthonormal:

$$\langle 0|0\rangle = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 1 \cdot 1 + 0 \cdot 0 = 1, \tag{2.8}$$

$$\langle 1|1\rangle = \begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 0 \cdot 0 + 1 \cdot 1 = 1, \tag{2.9}$$

$$\langle 0|1\rangle = \begin{pmatrix} 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = 1 \cdot 0 + 0 \cdot 1 = 0, \tag{2.10}$$

$$\langle 1|0\rangle = \begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 0 \cdot 1 + 1 \cdot 0 = 0. \tag{2.11}$$

Our qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ from above can thus be equivalently rewritten in vector form in the standard computational basis as:

$$|\psi\rangle = \alpha \begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}. \tag{2.12}$$

The complex numbers α and β are called the *probability amplitudes* of the qubit and the squares of their moduli, $\|\alpha\|^2$ and $\|\beta\|^2$, represent the chance of finding the qubit in state $|0\rangle$ and the chance of finding the qubit in state $|1\rangle$, respectively, upon measurement[8]. A qubit superposition is thus loosely defined by a probability distribution over the computational basis states, which means that a qubit is valid only when *normalized*,

[7] For the rest of this thesis, unless specifically mentioned otherwise, all operators and vectors will be defined with respect to the standard computational basis.

[8] For a simple qubit a computer scientist might envision measurement as simply an attempt to read out the value of the qubit. When this happens, the superposed quantum state is projected onto one of its computational basis states. Measurement in quantum mechanics is a complex topic. For more information, see Chapter 2 of *Quantum Computation and Quantum Information* [45] by Nielsen and Chuang.

which is equivalent to the sum of the squares of the moduli of the probability amplitudes being equal to 1, since $\sqrt{\langle x|x\rangle} = 1$ implies $\langle x|x\rangle = 1$.

Measurement

While a qubit sits in a superposition it does not have a definite state, just as light, while acting as a wave, does not have a definite position. Upon measurement of the qubit, the qubit probabilistically chooses one of its computational basis states and the qubit's superposition collapses to this particular definite state. This is analogous to light observed as a particle, which collapses its wavelike form to a definite position. Since measurement of a qubit will force it to one of its orthogonal states, a subsequent measurement will yield the same result with 100% probability, which makes it impossible to determine the probability amplitudes of an unknown qubit [45, 41]. For example, suppose we have a qubit $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$ and a measurement reveals that the qubit is in state $|1\rangle$. All this means is that the post-measurement state is $|1\rangle$, but we do not know whether the state before was a superposition which collapsed to $|1\rangle$ or whether it was simply the computational basis state $|1\rangle$. An estimate of the true quantum state can be determined by preparing a large number of qubits identically (an *ensemble* [41]), performing the same measurement on all qubits, and using the actual frequencies of each resulting state to *approximate* the true probability amplitudes[9]. As an example, suppose we have some way of preparing a large ensemble of identical $|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$. Additionally, suppose $\alpha = \frac{1}{2}$ and $\beta = \frac{\sqrt{3}}{2}$, but this information is not known to us. After measuring 10 of the qubits in our ensemble we might have observed $|0\rangle$ as a result 2 times and $|1\rangle$ 8 times, suggesting that the amplitude of $|0\rangle$ might be such that its modulus squared is equal to 0.20 and that of $|1\rangle$ such that its modulus squared is equal to 0.80. After measuring 32 qubits we might have observed $|0\rangle$ 9 times and $|1\rangle$ 23 times, and we would update our estimates of the probabilities to 0.28 and 0.72. The more qubits we measure the more precise our estimates become and the closer we are able to approximate the true probabilities of 0.25 ($\frac{1}{4}$) and 0.75 ($\frac{3}{4}$), for $|0\rangle$ and $|1\rangle$, respectively.

2.1.5 Quantum operations and operator properties

To effect changes in a quantum state, a quantum operation (or *quantum gate*) is applied to the quantum state [45]. A transformation which takes a vector $|x\rangle$ of a vectorspace V and maps it to a vector $|x\prime\rangle$ of the *same* vectorspace V is called an *operator* [10]. In order to apply individually to *each* element of a superposition, a quantum operation

[9] It is important to note, however, that even though we might approximate the *probabilities* to some high precision, information on the actual probability *amplitudes* can not be discovered through measurements, since the amplitudes are complex numbers and we are using *real* numbers to approximate the probabilities of measuring $|0\rangle$ and $|1\rangle$. The modulus of a complex number is a many-to-one function, which means we cannot recover a 2-dimensional complex number from its 1-dimensional modulus.

[10] It follows that an arbitrary matrix representation of an operator must be square.

must necessarily be a *linear* operator[11] and can thus be represented by a transformation matrix. A vector (or ket) may then be transformed into another vector by application of such an operator, through matrix multiplication.

Hermitian adjoint of an operator

An operator A has a *Hermitian adjoint* (or *conjugate transpose*) $A^\dagger$, just as a ket has a bra. Such an adjoint must satisfy the following for two vectors $|x\rangle$ and $|y\rangle$ in the vectorspace V:

$$\langle x|A|y\rangle = \langle y|A^\dagger|x\rangle$$

This means that the inner product of $|x\rangle$ and $A|y\rangle$ must be the same as the inner product of $|y\rangle$ and $A^\dagger|x\rangle$.

To construct the Hermitian adjoint of an operator [45, 41]:

1. Take the transpose of the operator (matrix)

2. Take the complex conjugate of each entry in the transpose

An operator H is called *Hermitian* if it satisfies $H = H^\dagger$. An operator U is called *unitary* if it satisfies $UU^\dagger = U^\dagger U = I_n$. Here I_n denotes the identity operator for the vectorspace of dimension n. This means that the adjoint of a unitary operator is its inverse, since $XX^{-1} = X^{-1}X = I_n$ is necessary and sufficient for an inverse and a unitary operator satisfies this relation. This means $UU^\dagger = UU^{-1} = U^{-1}U = U^\dagger U = I_n$. Unitary operators are essential to quantum computation, since a valid operator must preserve the unitarity of a state vector.

Pauli operators and their matrices

Quantum mechanics has four basic operators which are known together as the *Pauli operators* [75] and are denoted by $I = \sigma_0,\ X = \sigma_1 = \sigma_x,\ Y = \sigma_2 = \sigma_y$ and $Z = \sigma_3 = \sigma_z$. All these operators are unitary and Hermitian and defined on a Hilbert space of dimension 2, but may be used to form more complex operators on higher-dimensional Hilbert spaces. In fact, any single-qubit operator may be used to build multi-qubit operators.

[11] An operator O is *linear* if it satisfies the following two conditions:

$$O(|x\rangle + |y\rangle) = O(|x\rangle) + O(|y\rangle),\ \ \forall |x\rangle, |y\rangle \in V \tag{2.13}$$

$$O(c|x\rangle) = cO(|x\rangle),\ \forall |x\rangle \in V,\ \ \forall c \in \mathbb{C} \tag{2.14}$$

The Pauli I: The first Pauli operator is simply the identity operator on vectorspace of dimension 2. In the standard basis this is:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{2.15}$$

The Pauli X: The second Pauli operator (also known as the *quantum NOT gate*) in standard basis is:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{2.16}$$

The Pauli Y: The third Pauli operator in standard basis is:

$$Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \tag{2.17}$$

The Pauli Z: Finally, the Pauli Z operator (also known as the *phase flip operator*) in standard basis is:

$$Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{2.18}$$

Action of Pauli operators on standard basis

From the above matrix representations of the Pauli operators, the following actions on the standard basis vectors $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ can be computed:

$$\sigma_0|0\rangle = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle \tag{2.19}$$

$$\sigma_0|1\rangle = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle \tag{2.20}$$

$$\sigma_x|0\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \end{pmatrix} = |1\rangle \tag{2.21}$$

$$\sigma_x|1\rangle = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle \tag{2.22}$$

$$\sigma_y|0\rangle = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ i \end{pmatrix} = i|1\rangle \tag{2.23}$$

$$\sigma_y|1\rangle = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} -i \\ 0 \end{pmatrix} = -i|0\rangle \tag{2.24}$$

$$\sigma_z|0\rangle = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} = |0\rangle \tag{2.25}$$

$$\sigma_z|1\rangle = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{pmatrix} 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ -1 \end{pmatrix} = -|1\rangle \tag{2.26}$$

To recapitulate, the following mappings occur:

$$\sigma_0|0\rangle \mapsto |0\rangle \quad ; \quad \sigma_0|1\rangle \mapsto |1\rangle \tag{2.27}$$
$$\sigma_x|0\rangle \mapsto |1\rangle \quad ; \quad \sigma_x|1\rangle \mapsto |0\rangle \tag{2.28}$$
$$\sigma_y|0\rangle \mapsto i|1\rangle \quad ; \quad \sigma_y|1\rangle \mapsto -i|0\rangle \tag{2.29}$$
$$\sigma_z|0\rangle \mapsto |0\rangle \quad ; \quad \sigma_z|1\rangle \mapsto -|1\rangle \tag{2.30}$$

Hadamard gate

The *Hadamard gate* is an operator which takes a qubit sitting in either of its two computational basis states and puts it in a superposition of those two states:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{2.31}$$

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{2.32}$$
$$H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{2.33}$$

The Hadamard gate is both unitary and Hermitian. An implication of this is that two consecutive applications of a Hadamard cancel each other out. The Hadamard gate is essential to quantum computation for creating superpositions.

2.1.6 Qubit registers

In classical computing 8 bits can be stringed together to form one byte and multiple bytes can form words. Qubits can *also* be combined to form quantum registers of memory. Due to the potential for superposition, the way in which a quantum register is composed is very different from the way in which a classical register might be formed.

Imagine a classical system of n bits. Each bit can be either *on* or *off*. Thus, if we combine n bits into an array, we will get n different slots with each slot either in *on* or *off* state (2 states), which means we simply need to know what each individual bit is, in order to know the full state of the n-bit system. There are $2 \times 2 \times ... \times 2 = 2^n$ possible states. Thus the composite system (or n-bit array) can be in exactly *one* of 2^n states:

$$\begin{aligned}
0: &\quad 0_{n-1}0_{n-2}...0_20_10_0 \\
&\quad \text{or} \\
1: &\quad 0_{n-1}0_{n-2}...0_20_11_0 \\
&\quad \text{or} \\
&\vdots \\
&\quad \text{or} \\
2^n-2: &\quad 1_{n-1}1_{n-2}...1_21_10_0 \\
&\quad \text{or} \\
2^n-1: &\quad 1_{n-1}1_{n-2}...1_21_11_0,
\end{aligned}$$

where the subscripts denote the indices of the bits in the register. This means we need to keep track of n values (which may each be *on* or *off*). We can thus describe the full system (array) in terms of the individual small systems (bits).

Qubits combine through tensor products to produce (normalized) quantum registers. For example, we can tensor the two qubits $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ to produce a necessarily normalized 2-qubit quantum register[12]:

$$|+-\rangle = \tfrac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle). \tag{2.34}$$

For a system of n qubits, each qubit can be in a superposition of its computational basis states, and it follows that when combined into a composite quantum system[13], again the qubits will be in a superposition of all possible states. In our example above, there are

[12] Here $|+\rangle$ and $|-\rangle$ are simply labels that are commonly used to denote the *Hadamard basis states* given by the qubits in our example.

[13] In this thesis, in the context of quantum computation, when we say quantum system we usually refer to a register made up of 1 or more qubits.

now $2 \times 2 = 4$ computational basis states[14]. Thus, for our system of n qubits we need to keep track of 2^n values (or coefficients given by the probability amplitudes) for each of the possible computational states of the new system. The composite qubit system will be in a superposition that includes *all* of the 2^n states:[15]

$$\begin{aligned} 0: &\quad |0_{n-1}0_{n-2}...0_20_10_0\rangle \\ &\quad \text{and} \\ 1: &\quad |0_{n-1}0_{n-2}...0_20_11_0\rangle \\ &\quad \text{and} \\ &\vdots \\ &\quad \text{and} \\ 2^n-2: &\quad |1_{n-1}1_{n-2}...1_21_10_0\rangle \\ &\quad \text{and} \\ 2^n-1: &\quad |1_{n-1}1_{n-2}...1_21_11_0\rangle \end{aligned}$$

The amazing thing about quantum mechanics is that when a quantum operation is applied to a quantum system, the operation is applied to all the individual components of the superposition at the same time, which is referred to as *quantum parallelism* [45, 74, 54]. Quantum parallelism can be a great computational resource, if an algorithm is designed to handle it effectively [45]. Unfortunately, quantum parallelism does not imply an ability to extract exponential amounts of data from the system, since, as mentioned earlier, a quantum system always collapses to one of its basis states upon measurement. This means that even if we apply an operation to all the states, we are only able to retrieve the result of that operation on *one* of the states [45].

Controlled operations

Given a single-qubit operator U, a *controlled* version of the operator can be built, which acts on two qubits and applies the operator U to one of the qubits, which is designated the *target*, if and only if the *control* (the other qubit) is 1. If the control qubit is 0, nothing happens. A very important two-qubit gate is the *controlled-NOT* or *CNOT* gate, in which the target is inverted (the NOT gate is applied to the target) when the control is 1.

[14] Recall that since computational basis states are *not* themselves superpositions, they can simply be ordered and related to the binary representation of their respective indices. As such, it is common to use the binary index of a computational basis state as the label for the state. For example, $|000\rangle$ simply refers to binary value 000, or the 0^{th} state in a 0-indexed ordering.

[15] Of course, if a particular probability amplitude is 0, not all of the basis states might be present in a given superposition. For example, given the kets $|x\rangle = |1\rangle$ and $|y\rangle = \sqrt{\frac{3}{4}}|0\rangle + \sqrt{\frac{1}{4}}|1\rangle$, the tensor product of the two kets gives $|x\rangle \otimes |y\rangle = \sqrt{\frac{3}{4}}|10\rangle + \sqrt{\frac{1}{4}}|11\rangle$ and so the two states $|00\rangle$ and $|01\rangle$ are not present in the composite, as their probability amplitudes are 0.

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{2.35}$$

Since CNOT is a 2-qubit gate, it takes as input kets which are linear combinations of four computational basis states: $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$. The action of the CNOT gate on the individual states is:

$$CNOT\ |00\rangle = |00\rangle \leftarrow \text{control: 0; target: 0; no action} \tag{2.36}$$

$$CNOT\ |01\rangle = |01\rangle \leftarrow \text{control: 0, target: 1; no action} \tag{2.37}$$

$$CNOT\ |10\rangle = |11\rangle \leftarrow \text{control: 1, target: 0; flip target} \tag{2.38}$$

$$CNOT\ |11\rangle = |10\rangle \leftarrow \text{control: 1, target: 1; flip target} \tag{2.39}$$

Operators on multi-qubit systems

As mentioned previously, a single-qubit operator can be used to create larger multi-qubit operators. To create such operators, we use an analogue of the tensor product for matrices, called a *Kronecker product*. As an example, suppose we have the following:

$$Z_{2\times 2} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \quad \text{and} \quad CX_{4\times 4} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \tag{2.40}$$

where the subscripts show the sizes of the operators and implicitly, of the Hilbert

spaces on which these operate. The Kronecker product of Z and CX is:

$$Z \otimes CX = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \otimes CX = \begin{pmatrix} 1 \cdot CX & 0 \cdot CX \\ 0 \cdot CX & -1 \cdot CX \end{pmatrix}$$

$$= \begin{pmatrix} 1 & 0 & 0 & 0 & & \\ 0 & 1 & 0 & 0 & & 0 \cdot CX \\ 0 & 0 & 0 & 1 & & \\ 0 & 0 & 1 & 0 & & \\ & 0 \cdot CX & & & & -1 \cdot CX \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \end{pmatrix} \tag{2.41}$$

$Z \otimes CX$ is a 3-qubit operator of size 8×8, which applies the CX operator to qubits 0 and 1 and the Z operator to qubit 2.

Recall that $I = \sigma_0$ is the identity operator for one qubit. Given that I leaves the qubit unchanged, this operator can be used to compose operators that act on only certain qubits of a multi-qubit system, while leaving other qubits undisturbed. For example, imagine we have a qubit register of arbitrary size n and we want to create an operator that performs a certain action given by O on the i^{th} qubit only. To do this we can create composite operator M as follows:

$$M = I_{n-1} \otimes I_{n-2} \otimes \cdots \otimes I_{i+1} \otimes O_i \otimes I_{i-1} \otimes \cdots \otimes I_0 \tag{2.42}$$

where the subscripts specify the qubits.

2.1.7 Spooky action at a distance

Probability and uncertainty are at the core of quantum mechanics and by extension, quantum computation. A state in a superposition has certain probabilities of ending up in each of its computational basis states and these probabilities evolve in time according to unitary transformations, but there is no way to predict with accuracy the result of a measurement, so long as a state remains in a superposition.

In quantum systems of more than one qubit it becomes possible to witness *quantum entanglement*, which can be explained [45, 51] as a phenomenon that binds together quantum systems such that we cannot describe the full system by its individual parts. A good example of quantum entanglement is seen in the first of four states known as the

Bell states[16]:

$$|\beta_{00}\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \tag{2.43}$$

It can easily be shown[17] that there are no two kets that combine through the tensor product to create this state, such that we could describe the composite state $|\beta_{00}\rangle$ by individual descriptions of these separate states. Two qubits that are entangled retain their entanglement even when physically separated and moved far apart. Because of their interrelationship, measurements on one qubit will affect the system as a whole and so the full result will be seen even at the distant partner qubit, despite its being incapable of directly communicating or interacting with the first qubit. Einstein referred to this occurrence as "spooky action at a distance" and such strange behaviour came to be known as the *EPR paradox* [45, 54], which Einstein, Rosen and Podolsky (for whom it is named) tried to explain classically in terms of what they called *hidden variables*. Einstein, Rosen and Podolsky believed these hidden variables were present in the system and predetermined the outcomes; however, real experiments contradicted their hypothesis [45, 54].

Quantum teleportation

A perfect example of the power of entanglement can be seen in the *quantum teleportation protocol*, in which the goal is to send an unknown quantum state $|\psi\rangle$ between two physically separated endpoints, with a traversal of the space between only by *classical* information[18] [45]. Quantum teleportation allows for this to happen by entangling $|\psi\rangle$ with a qubit at one endpoint, call it $|A\rangle$, which had already been pre-entangled with another qubit, call it $|B\rangle$. At the time of their entanglement $|A\rangle$ and $|B\rangle$ were close together; however, at present, even while the two retain their entanglement, $|B\rangle$ is far away from $|A\rangle$, at the opposite endpoint. To make the teleportation example more dramatic, the distance between the two endpoints can be thought to be on the order of several light years. Measurements are made on the two qubits $|\psi\rangle|A\rangle$, which are physically together (and entangled) and the measurement results (two classical bits) are then sent

[16] To create $|\beta_{00}\rangle$ start with the state $|00\rangle = |0\rangle|0\rangle$, apply the Hadamard gate to the first qubit from the left, transforming the state to a superposition in the first qubit, $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|0\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |10\rangle)$ and then apply a CNOT gate targeting the rightmost qubit, with the qubit on the left as the control qubit. This yields the final state is $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$.

[17] Suppose there exist $|v\rangle = v_1|0\rangle + v_2|1\rangle$ and $|w\rangle = w_1|0\rangle + w_2|1\rangle$ such that $|v\rangle \otimes |w\rangle = |\beta_{00}\rangle$. Then it would have to be that $|v\rangle \otimes |w\rangle = v_1w_1|00\rangle + v_1w_2|01\rangle + v_2w_1|10\rangle + v_2w_2|11\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$. Equating like states gives $v_1w_1 = \frac{1}{\sqrt{2}}$; $v_1w_2 = v_2w_1 = 0$; $v_2w_2 = \frac{1}{\sqrt{2}}$. Since $v_1w_2 = 0 \rightarrow$ (1) $v_1 = 0$ or (2) $w_2 = 0$. If (1) is true, then it would follow that $v_1 = 0 \rightarrow v_1w_1 = 0 \neq \frac{1}{\sqrt{2}}$ and so (1) cannot be true. Then it must be that (2) is true; however, this similarly implies a contradiction and so $|\beta_{00}\rangle$ cannot be decomposed into two factor kets.

[18] Binary information can be represented by bits. Since a qubit upon measurement assumes one of its two computational basis states, the result from such a measurement can be contained by a normal bit.

to the other end (through a telephone wire, communicated by a physical messenger, in a letter on a spaceship, or through some other means), where $|B\rangle$ exists. Based on the measurements, simple corrective transformations may be applied to the lone qubit $|B\rangle$ which effectively reconstructs the state of $|\psi\rangle$ at $|B\rangle$. The *no-cloning theorem* [45, 54] states that it is impossible to perfectly copy an arbitrary, unknown quantum state, which means that the original qubit $|\psi\rangle$ will not retain its state (this will have been destroyed in the transition to $|B\rangle$; otherwise, there are now two copies of the original $|\psi\rangle$, which contradicts the no-cloning theorem). Additionally due to the non-unitary property of quantum measurement, the state of $|B\rangle$ cannot be determined, but the objective of quantum teleportation is met by this protocol, as the unknown quantum state $|\psi\rangle$ is passed to $|B\rangle$, with only two classical bits travelling the physical space from $|A\rangle$ to $|B\rangle$.

2.1.8 Quantum programs

A quantum program is simply a sequence of quantum operations (quantum gates) and possible measurements, applied to some input quantum register. A striking difference between classical and quantum programming is that quantum operations must all be *reversible* [75]. This is a direct implication of the unitarity requirement of a quantum state. An operator is reversible if the original state can be uniquely determined from the post-operator state. For example, the Z gate is unitary, because a result of $|0\rangle$ tells us with certainty that the input was $|0\rangle$ and a result of $-|1\rangle$ tells us with certainty that the input was $|1\rangle$. Measurements are non-unitary operators which collapse a quantum register's state to a classical equivalent (one of the computational basis states) [45].

There is an infinity of valid single-qubit gates, as the only requirement is that the gate be unitary. It has been shown that all single-qubit gates together with the CNOT gate form a universal set[19] for quantum computation; while in practice, a properly chosen finite set could approximate any computation sufficiently well [45].

2.1.9 Quantum algorithms

There are few *significant* distinctly-quantum algorithms which have been discovered [62], but the ones that have been developed are impressive in their elegance and use of non-classical logic. The first and the most famous algorithm was discovered by Peter Shor in 1994 [63] and is known as *Shor's quantum factoring algorithm*. The most important part of this algorithm is the use of the quantum Fourier transform (QFT) to find the period of a function[20], which allows the algorithm to determine the prime factorization of any composite number with an exponential speedup over the best (as currently known) classical algorithm for the same problem. Running on a real quantum computer, Shor's

[19] A set of gates forms a universal set if the gates are sufficient to construct any possible program.
[20] The period of a function $f(x)$ is a number k such that $f(x+k) = f(x)$.

algorithm would render obsolete[21] certain security and cryptographic protocols (such as RSA[22]), which rely on our current inability to efficiently factor large numbers. Shor's factoring algorithm is also important, as, due to its intriguing implications, it spurred a lot of research into quantum computation for the very first time.

The ability of quantum states to exist in superpositions allows for *interference* of probability amplitudes [45, 54], in which probability amplitudes of like signs add up and probability amplitudes of opposite signs cancel each other out. This is once again analogous to the interference effects in waves, where deconstructive interference leads to two out-of-phase waves cancelling each other out and constructive interference reinforces two waves that are in phase [61]. *Amplitude amplification* is a technique [45] used in quantum programming by which the probability amplitudes of desired states are sought to be increased, whilst reducing the rest of the amplitudes, such that the desired state stands out and its likelihood of being measured is increased. An important group of algorithms is based on Lov Grover's *quantum search algorithm*, which makes use of amplitude amplification to efficiently find the index of a marked item in an unstructured database [54]. This group of inspirational algorithms does not offer as great a speedup as the group based on the QFT, but is likely to become of more practical use once quantum computers come to be, as our data needs and thus, database application requirements are always becoming more significant [45].

2.1.10 Why quantum computation ?

Quantum computers are not currently on the market and will most likely not make their debut for many years to come[23]. There are many problems that quantum engineers face. The greatest problem is that of *decoherence*, which limits the time a quantum state can spend in a superposition. In a theoretical, perfectly-closed system, decoherence would not pose an issue, but in the real world, there is no perfectly-closed system and every system interacts with its environment. This interaction causes loss of information in a quantum state, which collapses the superposition of the state, or aggregates a non-negligible error in time. Thus, there is a limit to the number and duration of computations that may be done on a particular quantum state, before decoherence causes erroneous results. Without the quantum effect of superposition, a quantum computer has no advantage

[21] Post-quantum cryptography [5] is an area of quantum computation that focuses on cryptographic methods that will be suitable once quantum computers have been built. Another (and at the time of this thesis, more expansive) area is that of quantum cryptography [46, 7], which deals mostly with secure cryptographic key distributions, based on quantum physics, but for which a real quantum computer is not required.

[22] RSA, named after its developers Rivest, Shamir and Adleman is a very popular public-key cryptosystem [73], in which very large prime numbers are used to establish a public-private key pair, such that the private key cannot tractably be discovered from possession of only the public key.

[23] At the time of writing, the surprising *D-Wave*[55] quantum computer is causing some excitement as a possible real quantum computer capable of handling 512 qubits (a huge number by quantum standards), but it has not yet been validated as a true quantum computer and many researchers retain their reservations.

over a classical computer; thus, decoherence is a real problem blocking progress on quantum technology.

Quantum *simulations* on a classical machine suffer from different problems, such as enormous memory need. To simulate a quantum computer, memory need would rise exponentially with a linear increase in quantum memory. A quantum register of only 64 qubits would require unprecedented amounts of memory[24], which is completely unattainable at present and possibly in the future too. One thing a quantum simulator cannot simulate is true randomness [73]. True randomness is easily achieved in a quantum system[25], but is impossible to simulate on a classical computer alone[26].

Given the problems in constructing quantum computers, one might question the need for such machines. Engineers, computer scientists and physicists have different answers to such a question [45, 54, 51, 75, 41]. An engineer might point out that Moore's Law [75, 45] is predicting a stasis point in computer technology in the early 21st century, when we will reach a physical limit in miniaturizing electrical components. Physicists are more concerned with the potential of simulating quantum systems [51, 45] on a quantum computer, to enable study of quantum mechanics and the physical properties of our universe. Such a simulation is difficult to do efficiently on a classical computer, as mentioned earlier. From a computer science point of view, we might simply be interested in the theoretical limits of computation and curious about whether quantum computers would allow us to tackle new problems [45], which are in complexity classes out of reach for classical computers. While a quantum computer cannot do more than a classical computer, it most likely can do some things more efficiently, which might even render feasible some useful NP problems.

2.1.11 Summary

Quantum computing is an alternative to classical computing. The main concept behind quantum computing is that information is physical [30]; thus, manipulating information, which is the essence of a computation, is bound by the physical laws describing our world. These strange laws allow data to exist in a superposition of values. On a quantum computer, real quantum programming would have to make ingenious use of quantum properties, such as superposition and entanglement in an effort to compete with corresponding classical solutions.

[24] A quantum register 64 qubits in length has $2^{64} \approx 20$ million trillion amplitudes. Recall that due to the possibility of entanglement each amplitude needs to be kept track of, as an entangled system may not be decomposed into smaller subsystems that may be individually tracked.

[25] For example, placing a state $|\psi\rangle$ in a perfect superposition $|\psi\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and then measuring this state in the standard basis would return 0 with 50% probability and 1 with 50% probability.

[26] Although it would be possible to perhaps hook up a classical computer to some outside source of true randomness [73].

2.2 Evolutionary computation

Inspiration from nature and the evolutionary process in particular have engendered a class of metaheuristics[27] collectively known as *evolutionary algorithms*. The methods are multifarious, yet all methods have in common their reliance on the concept of natural selection [9], in which the fittest candidate is more likely to survive. To this end, a *fitness* (quality) measure is used to differentiate one candidate solution from another and the better solution will generally have an advantage to persist [21, 49, 2]. Evolutionary computation was first outlined by Alan Turing in his seminal paper on artificial intelligence, *Computing Machinery and Intelligence* [76], first published as early as 1950.

Evolutionary algorithms can tackle diverse tasks, but they really excel at problems which require only approximate solutions, or for which a solution is not known to exist in advance, or for which the conditions for a solution are difficult to nail down with the allowable representations of traditional search and optimization techniques.

At a high-level, *machine learning* [42] concerns the construction of computer systems which use experience and seek patterns in their inputs in order to adapt (i.e. *learn*) and improve themselves with respect to some objective [57, 6, 80]. A machine learning method can thus be described by its combination of a solution representation, an evaluation method to determine the quality of a solution and an optimization procedure which decides how to move through the space of solutions [15].

Due to the learning component of evolutionary computation, it can be related to the broad field of machine learning [2], but its methods have a much more empirical approach than most machine learning techniques, which rely more heavily on mathematical bounds, guarantees and theory. Randomization plays a crucial role in most evolutionary search methods and the algorithms are characterized by their search through a pool of candidate solutions, which is in contrast to other methods which tend to a solution by continuous improvements of a single candidate [2, 57]. By favouring the better candidates, a solution pool[28] is expected to inherit improved features over time and in the best-case scenario, eventually converge upon an acceptable solution. The general process of an evolutionary method is shown in Algorithm 1.

The most popular of the evolutionary methods is the *genetic algorithm* [21], which has enjoyed a relatively long history. Another form of evolutionary method is a *genetic program* [2], which is normally used to evolve programs. Various other methods, including cellular automata and particle swarm optimization [49] fall under evolutionary computation as well, but are more heavily influenced by metaphors inspired from biological processes.

[27] Sorensen defines a metaheuristic [65] as a framework which guides the development of heuristic optimization algorithms.

[28] In fact, evolutionary computation methods can be viewed as part of a set of *stochastic beam search*[57] methods, which make use of multiple candidate solutions and attempt to test random moves in the search space and proceed to keep those which yield an improvement over the current situation.

```
find_solution( problem_specifics, N, G, stopping_criteria ):
begin
    create fitness function based on problem_specifics and stopping_criteria
    randomly initialize population of N candidate solutions
    set generation = 0
    repeat
        calculate fitness of each candidate solution
        apply genetic (search) operators to candidate pool to create new pool
        generation = generation + 1
    until generation has exceeded G or stopping_criteria have been met;
    return the candidate solution with best fitness value
end
```

Algorithm 1: Evolutionary algorithm

Evolutionary methods have a panoply of applications [49, 2] in fields such as data mining, computer vision, robotics, engineering, art, astronomy and others. In many instances these methods are very competitive with other well-established machine learning methods. Additionally, there are cases where evolutionary methods are the best or only way to approach a problem; for example, when a problem cannot be adequately defined in mathematical terms, or not all aspects of the problem are known or defined. In other words, evolutionary methods are very helpful when the idea of what constitutes a target solution is rather vague. If one more or less knows what they are looking for, evolutionary methods are just alternative search methods, but if exploration is necessary, an evolutionary method allows for great search capacity.

2.2.1 Genetic programming

As a machine learning metaheuristic, the genetic algorithm, which is limited by a set solution size, gives rise to a closely-related extension, known as the *genetic program*, which allows for variable-sized solutions and thus aims to *model* a solution, rather than *optimize* a solution for which the structure is already known [71]. While a genetic program can be applied to any problem a genetic algorithm might be applied to, the usual solutions of a genetic program are executable programs. The conventional genetic algorithm uses a binary string representation, called a *chromosome* [21]. In contrast, a genetic program can define its solution more flexibly and adopt the use of more complex data structures [49] whose properties could potentially optimize a search.

Genetic programming ingredients

Genetic programs allow for high flexibility and customization for the various problems they can solve. The following components together make up a genetic program.

- **Solution Representation (*Genotype*)**
  The representation of a solution is very important to a genetic program. The

problem and solution must be formulated in such a way so as to allow for easy manipulation, adjustment and querying of all details that make up a possible solution. Important features of a potential candidate solution are called *genes* and the entire representation is called the *genotype*. Often the result of a genetic program can be directly executed. Depending on the problem, it is possible to have a *second-order encoding*, which first decodes the final genotype into an executable program. In such a case, the genotype is distinct from the *phenotype*, which is the final output, after decoding.

- **Primitive set**
  A solution candidate of a genetic program can be expressed as a structure of relationships between elements of a primitive set. This set comprises two distinct sets:

  1. **Terminal set**: elements (often integers, real numbers, or characters) called terminals, which can be manipulated as the *alleles* (or options) for the different genes (attributes or features) of the genotype.
  2. **Function set**: functions or behaviours which are mostly used to manipulate the elements of the terminal set, but can also just have side-effects on a solution.

- **Fitness evaluation**
  Paralleling Darwin's theory of natural selection[29] [9], the evolutionary process of a genetic program is a competition in which potential solutions compete against one another to have their features propagated to later generations. A solution which is a better approximation to the ideal (or exact) solution is a better candidate and is favoured to win in the competition. A *fitness function* is defined in some suitable way to correctly identify the better candidates in a set. While it might not be able to tell us exactly what a solution looks like, the fitness function must be able to recognize a solution. Upon evaluation, an individual is associated with a *fitness value* which defines its relative quality in the set. Somewhat counterintuitively, a *standardized fitness function* [2, 49] defines 0 as corresponding to the highest fitness and increasing fitness values correspond to decreasing quality.

- **Stopping criteria**
  A genetic program cannot run forever. It thus requires a set of stopping criteria, which can either limit the evolution to a preset number of iterations, or ensure

[29] Darwin introduced the concept of natural selection in 1859 [9]. His idea was that evolution is a competition for survival. The term *competition* implies a *population* and this population naturally has variations which render certain individuals more fit for survival, giving them a greater chance to reproduce. Whatever good traits these individuals possess are inherited by their offspring, so that in a way these individuals are preserved through their traits.

that evolution stops when the solution pool has converged upon a good enough solution, or some other criteria have been met.

- **Search (*genetic*) operators**
  A genetic operator is little more than a *search* operator. Genetic operators are functions applied to the candidate solution pool to navigate through the search space and identify new candidate solutions. The most common genetic operators used are:

  1. **Crossover**: usually applied to 2 (but possibly more) candidate solutions called the parents, crossover [49] borrows features from both (all) parents to create a new candidate solution. The ways in which the parents can be combined are endless, but one popular way is to define a random crossover point in each of two parents to split the parents in two parts, and create the offspring by recombining the split parts of opposite parents.

  2. **Mutation**: some feature of a candidate solution is randomly changed. This could be a terminal in a leaf node of a tree, or perhaps a function node.

  3. **Reproduction**: reproduction is a direct copy of an individual to the next generation. This is not always used, but could ensure that the best candidates at any generation are not lost before the next.

- **Selection**
  At every generation, the genetic operators are applied to a selected percentage of the candidate pool. Selection methods must try to balance the trade-offs [49] between greedy *exploitation* of the best solutions and *exploration* of the seemingly weaker solutions, which might in time lead to improved discoveries. A popular selection method is called *tournament selection* and it works by running a set of tournaments, in each of which a number (often 2 or 3) of randomly chosen candidates are compared against each other based on fitness and the candidate with best fitness is designated the winner of the tournament and is thus selected. Another selection method might simply rank all candidates by fitness and select some percentage of the top-ranked.

- **Parameters**
  A genetic program will use a number of adjustable parameters which determine things such as: the rates of application for each genetic operator employed, the maximum number of generations, and how biased the selection routine is towards the better solutions (*selection pressure*).

2.2.2 EDA-GP variant

In the traditional genetic program a population of candidate solutions is improved at each iteration with the aim of approximating the ideal solution. *Estimation of distribution algorithms* (EDA) instead use probability distributions to model the solutions [25, 48, 35]. Attributes are each associated with a random variable[30], where the random variable can take on any of the appropriate alleles for that feature. EDAs attempt to gradually *learn* the probability distribution of the ideal solution. At each iteration a candidate set is sampled from the probability distribution model and each candidate in that set is tested for fitness. A percentage of the best part of the set is selected and the distribution is updated from this subset. The next generation of candidates is sampled from the new (and hopefully improved) model and the process is repeated until convergence of the distribution, or until a maximum number of iterations have been run.

A genetic program employing an EDA gives up its explicit use of search operators, but it still attempts to improve the new candidate solution sets through inheritance of good features, by implicitly incorporating the more successful features in its probability distribution model.

EDAs benefit from the vast availability of machine learning techniques which can be used to properly learn a probability distribution. In the simplest case an assumption of independence can be made for all the attributes of a candidate solution. The *Bayesian optimization algorithm* (BOA) [47, 34] uses Bayesian networks to infer relationships, such as conditional dependence among different attributes of the solutions.

While EDAs can be very successful, they can also be very expensive to use in genetic programming, where the data structure and primitive set used for representing a solution might spawn an enormous number of dimensions and random variables [50].

Just as with other evolutionary methods, an EDA must be cautious when exploiting the better solutions, as a high selection pressure might lead to premature convergence to a local optimum.

2.2.3 Summary

This chapter served as an introduction to quantum computation and genetic programming. We covered the basics of quantum computation and its mathematics and pointed out a few reasons why quantum computation is a significant field. We also saw that genetic programming is a highly-customizable metaheuristic, which generally does not discriminate against its problems.

[30] A random variable is a variable that may take on different values with various probabilities, which must sum up to 1.0.

Chapter 3

Literature review

The work of this thesis falls on a boundary connecting two distinct areas of computer science: quantum computation and evolutionary computation. This intersection is made up of two[1] main categories:

- Evolutionary quantum computation
- Quantum evolutionary computation

While these names are confusingly similar, they embrace very different ideas and approaches to combining quantum computation and evolutionary computation[2] . The first topic involves the use of evolutionary methods, including, but not limited to genetic algorithms and genetic programming, to gain insight into quantum computation and quantum information, by aiding the development of quantum programs, algorithms and protocols and discovering intricacies of quantum effects. In other words, the subarea concerns the use of evolutionary methods for the advancement of quantum computation.

The goal of quantum evolutionary computation is to use real quantum hardware, or more feasibly at the moment, the theoretical model of a quantum computer as the basis to devise new evolutionary methods which can directly make use of the underlying quantum effects to boost search and optimization. These methods are expected to run

[1] It should be noted that there exists another branch of evolutionary computation, namely, *quantum-inspired evolutionary computation*, which comprises evolutionary algorithms that run on classical computers and which only take *inspiration* from quantum mechanics or quantum computation for representation of a candidate solution, or for development of new search operators. Often the quantum connection is merely superficial (or even faulty) and for this reason, the view in this thesis is that the body of quantum-inspired search methods does not have a direct connection to quantum computation and thus should not be a separate category in this intersection, but should be considered instead under evolutionary computation. Donald Sofge also suggests [64] that quantum-inspired work should fall under the emerging field of *quantum interaction* [1], which uses ideas from quantum theory in computer science, in general.

[2] Yet these areas are not mutually exclusive, as one would hope that with the advent of quantum computers the broader fields would become mutually beneficial.

on real quantum computers, if and when such machines will exist.[3]

This thesis itself falls into the first category in the intersection; that is, it concerns the use of genetic programming for the advancement of quantum computation and quantum information. Evolutionary quantum computation is thus the sole category whose works are elaborated upon in what follows.

3.1 Evolutionary QC

Peter Shor's *quantum factoring algorithm* in 1994 generated a lot of interest in quantum algorithms, but no associated productivity in this area followed and so, at the time of this writing, twenty years from the QFA, there are still very few *significant* quantum algorithms. Spector suggests [66] that design of quantum algorithms at present is no better suited for humans than for machines, since our grasp of quantum concepts is still fragile and it is not trivial to decouple our programming from our classical thinking. With this view it is natural to seek help from stochastic optimization methods, such as evolutionary computation, to attempt discovery of new quantum algorithms. Such work has already been attempted, as reviewed in the following sections.

3.1.1 Decomposition of quantum targets

In the late 1990s researchers began using genetic algorithms and genetic programming for evolving quantum circuits. The founding work [79] in this area came from Williams and Gray, who used genetic programming to find a gate decomposition for a known quantum circuit, namely, the quantum teleportation circuit.[4] The authors were interested in how one could efficiently decompose an end result (in this case, a target unitary transformation) into a sequence of gates drawn from some specified quantum gate set (not necessarily universal). An efficient decomposition uses as few gates as possible, while an efficient search considers as few solutions as possible. The results of the work by Williams and Gray demonstrated that a deterministic circuit as efficient as the most efficient hand-crafted circuit could be found about 10 times faster than by exhaustive enumeration.

[3] An obvious obstacle for work in this subarea is the current lack of quantum hardware. An excellent critique and evaluation of the relevant efforts towards better and scalable search methods is provided by Sofge [64]. There are two main ideas for creating quantum evolutionary algorithms: quantum parallelism is either used to access a greater portion of the search space than is possible with a classical algorithm [58], or quantum parallelism is used to distribute fitness evaluations in such a way so as to speed up the evolutionary process. All the algorithms semi-proposed so far leave out important (essential) implementation details on how one would circumvent the *no-cloning theorem* (see Chapter 2) and the collapse of the quantum wave function (superposition) when observations (which are necessary for fitness evaluation and crossover) are made.

[4] Quantum teleportation is a communication circuit which makes use of quantum entanglement and classical communication to allow a qubit whose quantum state is not known, to be *teleported* across an arbitrary distance (say, from an endpoint Alice to an endpoint represented by her friend Bob), without ever physically crossing the space.

3.1.2 Scalable quantum programs

Without a working quantum computer, some form of a quantum simulator must be used for evaluation of the evolved quantum programs. Quantum simulators limit the resources (memory and speed) available for evolutionary methods. Given simulators that work on few qubits (perhaps between 1 and 5), Spector et al. were quick to notice and emphasize [67] the need for evolution of *scalable* quantum programs. In 1998 Spector et al. [67] used traditional tree-based GP with a second-order encoding to evolve scalable solutions for oracle[5] problems, including the *majority-on problem*[6], the *Deutsch-Jozsa problem*[7] and *Grover's database search*[8]. The function set incorporated a control structure for looping a body of code (which is essential for scaling). With the second-order encoding, a result of the genetic programming run was a program itself, which upon execution could generate quantum gate arrays (or circuits) for different sizes of a given problem.

The goal of scalability permeates the work in the field, as reiterated most strongly by Massey et al. [38, 40], who also made use of second-order encodings and control structures, in order to produce a parameterizable program for the quantum Fourier transform.

3.1.3 Probabilistic vs. deterministic quantum circuits

Quantum circuits can either be *deterministic*, in which case they give the right result for all inputs (or the *wrong* result for all inputs[9]), or they can be *probabilistic*, in which case the only guarantee is that they are more reliable than random chance, or a pre-specified threshold (e.g. 30%). Since quantum computation is naturally probabilistic and some of the most famous hand-crafted quantum algorithms are themselves probabilistic (e.g. QFA), it is perhaps not surprising that Spector et al., in a great portion of their work [67, 69, 68, 3, 4], thought to use a fitness function which guided the genetic program to probabilistic solutions. Massey [40] et al. [37] also adopted a probabilistic approach to evolve, among others, a probabilistic quantum-half adder, which used only the Hadamard gate and a ZERO gate[10]. Massey et al. remark [37] that it is not yet clear whether (and

[5] A blackbox function, or oracle function is a function which can be queried, but whose internals are not known.

[6] Given a blackbox which computes an unknown function $f(x)$, the objective is to determine with as few calls to the blackbox as possible, whether $f(x)$ returns mostly 1s (majority on) or not.

[7] Given a blackbox which implements an unknown binary function $f(x)$, the goal is to determine with as few as possible calls to the blackbox, whether $f(x)$ is *uniform*, that is, it returns the same value for all inputs, or it is *balanced*, that is, it returns 0 half the time and 1 the other half. The blackbox is guaranteed to have one of these two properties, but nothing else is known about it.

[8] Given a database and a function $f(x)$, which takes an address x (or an index into the database) and returns *true* if $f(x)$ contains a particular desired item, the goal is to find x with as few calls to $f(x)$ as possible.

[9] For the rest of this thesis a deterministic quantum circuit will mean a quantum circuit that is correct; that is, it gives the correct result for all possible inputs.

[10] The ZERO gate forces a quantum bit to the computational state $|0\rangle$. Massey defines it as a gate by noting that a swap of an arbitrary qubit with an *ancilla* (or work) qubit, which is known to be in the $|0\rangle$ state, satisfies the unitarity condition of a quantum gate.

how) probabilistic quantum circuits might be entirely useful.

3.1.4 Solution representations

One of the strong points of evolutionary methods is the flexibility for designing solution representations. As evolutionary quantum programming is a young field, research is still needed to determine what genotype/chromosome encoding is suitable or even optimal, for a given problem. Spector et al. [69] have reasoned that a linear encoding, rather than the traditional tree encoding which is widely used in genetic programming, is better-suited for evolving quantum circuits, since the result of a quantum circuit is made up of a linear sequence of gate applications and the gates' side effects on a given quantum state are more important than some immediate return value, due to the entanglement capabilities of quantum bits. Their experimental results also supported the use of the linear genotype. In the same paper [69], Spector et al. noted that a tree representation offers better scalability, which might suggest that the linear representation is at odds with the goal of scalability.

In 2000, Yabuki and Iba tackled the evolution of a quantum teleportation circuit [81]. Pointing out that the work done previously by Williams and Gray [79] allowed the evolution of circuits which might violate the quantum teleportation protocol,[11] the authors split up the quantum teleportation circuit into three parts: 1) entanglement production, 2) Alice's part (in which she interacts the unknown qubit with her own and performs a measurement) and 3) Bob's part (in which he performs transformations on his own qubit). They developed a genetic algorithm which used codes (made up of 3 letters) translating to specific functions, depending on which stage of the circuit they were found in. Special codes were used to delimit the stages. The remaining codes mapped to one of three different translation tables, one for each stage of the protocol, such that known restrictions (for example, Bob being allowed to apply transformations *only* to the qubit he owned) could be enforced. In this way, the circuits which evolved did not break the protocol. The method required specific knowledge about the problem and might be useful for similar protocols, but could be difficult to generalize to other problems.

Most of the work in automated quantum programming has either used a tree representation [37, 67], or a linear representation as either a chromosome in a genetic algorithm [81], or some sort of a list in genetic programming [69, 79, 56]. Leier and Banzhaf [31] have opted to use a linear-tree encoding[12], alluding to the closed-system evolution of a

[11] Specifically, the method might evolve a quantum circuit in which Alice is able to perform quantum transformations on Bob's qubit, which of course is impossible, because Bob's qubit is assumed to be a great distance away from Alice. For more information, see Chapter 2.

[12] A linear-tree in GP is essentially a linked-list nesting functions, where functions of arity greater than 1 cause the data structure to branch off, as would an internal node of a normal tree.[29]

quantum system, in which the system evolves in a linear fashion, until a measurement causes a probabilistic branching.

No study has been done to compare the different representations for quantum circuit evolution, so at present it is not known whether one might truly be better than another. This remains an open research question.

3.1.5 Fitness function and evaluation

A variety of fitness functions have been used in this field. Williams and Gray used a simple sum of absolute differences which compared every single element of the known target matrix to the corresponding element in the matrix representation of the evolved decomposition [79]. Given that this work made the unique assumption that the target unitary would be given, this choice of fitness function did its intended job. Williams and Gray suggested that approximate circuits could also be evolved with this method, although they did not experiment with such.

For the more common case, in which a target unitary matrix was not known in advance, a number of fitness cases were defined, such as randomly generated qubits [81], or vectors spanning the entire quantum space [37] and a target vector was computed for each fitness case. The target vectors were then compared to the vector that resulted from applying the evolved program to the target's respective input. This comparison was usually done as a sum of absolute differences of the probability amplitudes [37]. Rubinstein was able to use a *single* fitness case [56] to evolve his circuits, since he sought an exact circuit for producing a *maximally entangled quantum state*[13], which he knew in advance.

The most significant fitness function contribution can be attributed to Spector et al. [67] who introduced a fitness function made up of 3 components, prioritized in lexicographical order:

- MISSES[14] : the total number of fitness cases for which the solution produced an incorrect result according to some threshold
- CORRECTNESS : the average error for all fitness cases which missed the threshold
- EFFICIENCY : $\frac{\text{Total number of gates in program}}{100000}$

This fitness function was used for probabilistic circuits and the idea was to use the components in the order given and only use the EFFICIENCY component as a way to compare quantum circuits that already gave the correct results (scored 0 on MISSES and CORRECTNESS), to look for most efficient (in terms of number of gates) program

[13] A multi-partite quantum state is maximally entangled if any measurement results in a completely random outcome [51].

[14] Originally this component was named HITS [67], but given its definition it was appropriate to rename it, which indeed they did [69].

solutions. It should be noted that MISSES (and thus CORRECTNESS) only considered misses for which the probability of error was 48% or more. This was so for two reasons: 1) to ensure that round-off error did not contribute to success and 2) to focus the search towards circuits that had 0 misses, instead of towards circuits that had high probability of success, but lots of misses. In later work by Spector et al. [69, 68], some components were altered, exchanged, or added. For example, in a paper which studied the evolution of quantum circuits for solving the *AND-OR problem*[15], a component titled EXPECTED-QUERIES was added as a way to gauge the circuit complexity [68] in terms of the number of calls made to a blackbox function. Massey et al. similarly used a lexicographic fitness function in their work on probabilistic circuits [37, 39]. Following suit, Stadelhofer et al. made use of a similar lexicographic fitness function in their studies [72] of quantum circuits for determining properties of blackbox functions.

3.1.6 Search operators and selection

Crossover and mutation have been employed as search operators in most of the work reviewed. Mutation has been applied to the parameters of gates, or to the gates themselves, with various mutation rates, ranging from the very small, 0.001 as used in a paper by Rubinstein which studied circuits for entanglement production [56], to exclusive use as in a work by Leier and Banzhaf [33], which compared selection strategies. Leier and Banzhaf studied the evolution of a quantum circuit for the Deutsch-Jozsa problem in terms of the fitness landscape[16] induced by a mutation operator [32]. They looked at 100 different random walks for 100000 time steps and calculated the auto-correlation[17] of the series, as well as the information content and partial information content of a transformation of the same time series[18]. Their results showed almost no correlation beyond the second time step, which came as a disappointing confirmation that quantum program search spaces are highly irregular and difficult to search. In their follow-up work [33] Leier and Banzhaf stated that a study of the fitness landscape induced by crossover suggested crossover was an even weaker operator for the Deutsch-Jozsa problem than was mutation; hence, they dropped the crossover operator from later work.

[15] A binary function $f(x)$ is given. An AND/OR tree of size n is a full binary tree with a Boolean AND at the root and n alternating layers of Boolean OR and AND nodes. Each leaf holds the result of function $f(x)$, where x is the index of the respective leaf and the leaves are numbered from 0 to $2^n - 1$. The goal is to determine whether the AND/OR tree evaluates to *true* for the given function $f(x)$.

[16] The fitness landscape is a graph showing how the fitness changes in the neighbourhood of a candidate solution, where the neighbourhood is made up of solutions transformed from the candidate through a particular search operator. For this reason, we say the fitness landscape is induced by a search operator.

[17] The auto-correlation function shows how points on the landscape separated by a number of time steps (iterations or transformations) correlate.

[18] The measures of information indicate how random the fitness transitions are, as induced by the search operator.

3.1.7 Interesting findings

As mentioned at the beginning of the chapter, work in evolutionary quantum programming does not involve just evolution of quantum circuits and quantum programs, but also any other searches or optimizations that might advance the field of quantum computation and quantum information. Spector and Bernstein studied the communication capacities of 2-qubit gates which are used for entanglement production [70]. For a 2-qubit system, where Alice has access to one qubit and Bob has access to the other, communication of one classical bit is achieved if there is a sequence of operations Alice can make on her qubit, a sequence of operations Bob can make on his qubit, and a 2-qubit gate is applied at some point in the process, to both qubits (thus, not by Alice or Bob, but perhaps by an intermediary), such that when Bob measures his qubit it will reveal a bit value (0 or 1), consistent with what Alice wished to communicate. In other words, if a circuit made up of single-qubit gates and a lone 2-qubit gate transforms the state such that upon measurement Bob's qubit reveals the original state of Alice's qubit with some non-zero probability, then the 2-qubit gate is said to have a communication capacity. Similarly, if the circuit ends up entangling the qubits which did not start out entangled, then the 2-qubit gate is said to have entanglement production capability. Bennett conjectured that a 2-qubit gate could both entangle to produce an e-bit[19], and communicate one c-bit (or classical bit) from one end to another (though not at the same time). Smolin, through private communication with Spector and Bernstein [70] suggested a 2-qubit gate known as the Smolin gate, as a counterexample to Bennett's conjecture. Spector and Bernstein used genetic programming to discover that the Smolin gate was in fact not a valid counterexample, as it *did* have the power to both entangle *and* communicate. Spector and Bernstein further discovered a 2-qubit gate which could entangle, but had no power to communicate; thus, they discovered a counterexample to Bennett's conjecture!

3.1.8 Summary

So far no novel quantum algorithm has been evolved, but some of the most famous quantum algorithms have been reproduced by GP [66, 40, 20], for small sets of qubits. Despite these successes, automatically programming a quantum computer is extremely difficult to do when the hardware has to be simulated with an exponential slowdown and when little is known about what makes quantum programming effective. If crossover and mutation are both equally weak search operators, perhaps there are better and more efficient ways to navigate a quantum program search space. Many research questions remain open, concerning the better-suited solution representation, search operators and fitness functions. Although in recent years work in this field has slowed down, the numerous works reviewed in this chapter show that the evolutionary framework is capable

[19] An e-bit [54] is basically a quantum resource which is made up of 2 entangled qubits.

of generating quantum circuits, programs, algorithms [40] and in one case [70] has even managed to produce an artifact that invalidated a conjecture. These results are hopefully all previews of more good results to follow.

Chapter 4

Methodology

The objective of this thesis was to test our hypotheses from Chapter 1:

1. *Quantum programs exhibit sequential patterns and relationships between their functions and inputs which can be learned by an EDA-based GP to help automatically generate programs.*

2. *A stochastically-driven GP engine with an underlying learner to guide perturbation of features could have an advantage over one without the learner.*

To this end we have developed various software:

1. `smallqc`: a quantum computer simulator for testing and executing quantum circuits and programs

2. GP quantum program evolver: a GP suite comprising four GP variants to learn and generate quantum circuits and programs

The chapter begins with an overview of the quantum simulator `smallqc` and then describes in detail each instance of the GP quantum program evolver suite.

4.1 Quantum simulations

In the absence of real quantum hardware on which to test quantum programs, we have to make do with simulations of quantum programs on classical machines. As previously mentioned in Chapter 2, this is no easy feat: when simulating a quantum system or a quantum computer, memory requirements grow exponentially with a linear growth in quantum memory. Moreover, calculations that can be done directly on superpositions in one single step on a real quantum computer need to be performed separately in a classical simulation of a quantum execution. These computations could potentially

be programmatically parallelized in certain cases; however, we will not be concerned with such things in this thesis. This section describes the quantum simulator we have implemented for testing our programs.

4.1.1 `smallqc`: Quantum simulator

`smallqc` is a simple quantum computer simulator. It can create single-qubit and multi-qubit (directly or through tensor product composition) quantum states as qubit registers of various sizes, manipulate them through unitary operators, perform measurements on the states, read in and run programs written in simple `smallqc` syntax and output results.

Quantum operations

Most of the quantum operations used by `smallqc` are 1-qubit and 2-qubit gates. The CCNOT[1] (controlled-controlled-NOT)[2] gate and CSWAP (controlled-swap) gate are the only 3-qubit gates that are also explicitly implemented. The simulator allows for easy (though, programmatical) introduction of any 2-qubit gate and controlled 2-qubit gate, as well as for the introduction of controlled gates with more than 2 control qubits; however, no versions of the latter have been used throughout the experiments. See Table 4.1 for a full listing of the currently (explicitly) implemented gates.

Quantum system representation

A quantum system is represented as an array of complex numbers in `smallqc`. Each slot in the array holds the amplitude of the computational basis state associated with the corresponding index. The standard computational basis in standard ordering is assumed. For example, a random 3-qubit unit vector might look as follows:

0.45	$0.31i$	$0.32i$	-0.43	0.29	$-0.01i$	0.47	0.33
$\|000\rangle$	$\|001\rangle$	$\|010\rangle$	$\|011\rangle$	$\|100\rangle$	$\|101\rangle$	$\|110\rangle$	$\|111\rangle$

The ordering of the qubits goes from the right to the left and is 0-indexed[3], such that in the 6^{th} basis state of a 3-qubit system ($|101\rangle$) only the 0^{th} and 2^{nd} qubits are set and the 1^{st} qubit is 0.

Quantum gate representation

An explicit matrix representation is not used for the quantum gates in `smallqc`; instead, `smallqc` defines a gate by a set of input-output action pairs, where each input is a com-

[1] The CCNOT gate is also known as the *Toffoli gate*.

[2] Given a single-qubit unitary operator U, a doubly-controlled gate takes two qubits as controls and a third as a target to which it applies U if and only if the controls are *both* 1.

[3] The basis states are 1-indexed, however.

Gate	Arity	Description	Example
`NOT`	1	inverts the qubit specified by the argument	`NOT(2)`
`CNOT`	2	controlled-NOT applies `NOT` to the qubit specified by the second argument (target), if and only if the qubit specified by the first (control) is 1	`CNOT(2,1)` has target 1 and control 2
`CCNOT`	3	applies `NOT` to the qubit specified by the third argument (target), if and only if both qubits specified by the first and the second arguments (controls) are 1	`CCNOT(2,1,0)` has target 0 and controls 2 and 1
`Y`	1	applies the Pauli Y operation to the qubit specified by the argument	`Y(3)`
`Z`	1	applies the Pauli Z operation to the qubit specified by the argument	`Z(2)`
`Hadamard`	1	applies the Hadamard operation to the qubit specified by the argument, which puts the system in a superposition of said qubit	`H(0)`
`WHT`	0	applies the Walsh-Hadamard operation to all qubits in the system	`WH()`
`Rx(/y/z)`	2	applies the X(/Y/Z) rotation operation to the qubit specified by the first argument, where the second argument is an index into a restricted table containing valid rotation angles[a]	`Ry(0,5)`
`PS`	2	applies a phase shift gate to the qubit specified, where the second argument is an index into a restricted table containing valid phase angles	`PS(2,0)`
`SWAP`	2	swaps the amplitudes of the two qubits specified by the first and second arguments	`SWAP(0,2)`
`CSWAP`	3	the controlled-swap applies `SWAP` to the qubits specified by the second and third arguments iff the qubit specified by the first argument is 1	`CSWAP(1,0,2)`
`Oracle`	0	queries an oracle[b]	`ORACLE()`
`T`	0	applies a special phase shift gate called the pi over 8 gate, with angle equal to $\frac{\pi}{4}$	`T(0)`
`W`	0	applies a special phase shift gate with angle equal to $\frac{3\pi}{4}$	`W(1)`
`S`	0	applies a special phase shift gate with angle equal to $\frac{\pi}{2}$	S(2)
`CZ`	2	controlled-Z	`CZ(0,1)`
`CH`	2	controlled-Hadamard	`CH(1,0)`
`CT`	2	controlled-T	`CT(3,5)`
`CS`	2	controlled-S	`CS(1,3)`
`CW`	2	controlled-W	`CW(2,6)`

Table 4.1: `smallqc` quantum operations

[a] `smallqc` also supports arbitrary angles for Rx, Ry, Rz and the PS gates; however, only discrete and prespecified angles are used by the GP evolvers.

[b] The implementation of the oracle gate depends on the context of the problem being solved.

putational basis state and the output is a state (perhaps a superposition) that would result from an application of the given gate to the input state. These actions are derived from the matrix representation of the gates, however. For example, the Hadamard gate is defined as follows:

$\|0\rangle$	$\|0\rangle$	$\frac{1}{\sqrt{2}}$
	$\|1\rangle$	$\frac{1}{\sqrt{2}}$
$\|1\rangle$	$\|0\rangle$	$\frac{1}{\sqrt{2}}$
	$\|1\rangle$	$-\frac{1}{\sqrt{2}}$

In this way, `smallqc` makes direct use of the intended effects of some gate to compute an application of the gate to an arbitrary system. When applying an arbitrary gate to an arbitrary qubit register, each basis state in the register is examined in order, the corresponding action is retrieved for the given gate and basis state, a result is computed and temporarily saved[4] and the previous amplitude is discarded. Once all basis states have been taken care of, all the results that were saved are merged into the register. The use of vectors as opposed to matrices allows `smallqc` to handle larger quantum systems, of up to 16 qubits, although experiments have not been done on quantum systems exceeding 9 qubits and even at 8 qubits the computational time was uncomfortably slow. There were a few reasons for opting for a non-matrix representation. The main issue was that of applying 2-qubit gates to qubits that were not adjacent. If matrices were employed this would be difficult to do for any two non-adjacent qubits and the only efficient way to have done so would have been to apply a sequence of SWAP operations to one of the qubits, in order to bring it close to the first qubit, apply the 2-qubit gate to the two qubits and finally apply SWAP operations again to move the qubit back to its original position. To illustrate the problem, imagine we have a three qubit system in a perfect superposition as follows:

$$|x\rangle = \frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle)$$

and we wish to apply the CNOT gate to qubits 2 and 0, where the control is qubit 2 and the target is qubit 0. A CNOT gate is a matrix of size 4×4. If the two qubits were adjacent (for example, if we used qubits 2 and 1) the CNOT gate could be tensored with the identity gate to create a large 8×8 matrix $CNOT \otimes I$ to operate on the entire system. In our case, however, the qubits are non-adjacent, which would require the creation of a 8×8 CNOT gate in order to link the two non-adjacent qubits across the middle qubit. This would be extremely cumbersome to do each time, so instead a solution would be

[4] A temporary working quantum system is used for intermediary storage and transfer of components for gates such as the 2-qubit ones.

to swap qubits 0 and 1, perform the normal $CNOT \otimes I$ and then swap qubits 1 and 0 again. This, however requires additional complications as well.

`smallqc` instead makes use of the binary form of each basis state to index into the vector of amplitudes and directly access those elements which require a change. In this example it picks out the amplitudes associated with the highlighted indices

$$|x\rangle = \frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle),$$

calculates the binary form of the *new* indices (i.e. $100 \mapsto 101$, $101 \mapsto 100$, $110 \mapsto 111$ and $111 \mapsto 110$) and then re-associates the amplitudes with their respective new indices. In this case it is trivial, since all amplitudes are equal and the final state is the same as the original.

Modes of operation

`smallqc` has two modes of operation: *interactive* and *automatic*. In interactive mode it allows a user to create different quantum states and apply all sorts of operations to them. The user may also delete some states and print states. This mode is currently useful for quickly testing parts of algorithms, but is not extensively used for the experiments in this thesis. In automatic mode the quantum simulator may read in a text-file containing a quantum program written in the language of `smallqc`. The programs are designated by a `.qcx` extension. The .qcx programs have two parts:

- **init**: multiple quantum registers can be created and tensored, input is pre-processed and prepared and eventually one resulting quantum system of a particular size is designated as the single *input* system on which the rest of the program runs

- **code**: a number of operations are listed, which by default are to be run on the input resulting from the init part

Not all programs require a customized input, in which case the init section can be left blank and the input will be a ZEROed quantum system of default system size (3 qubits).

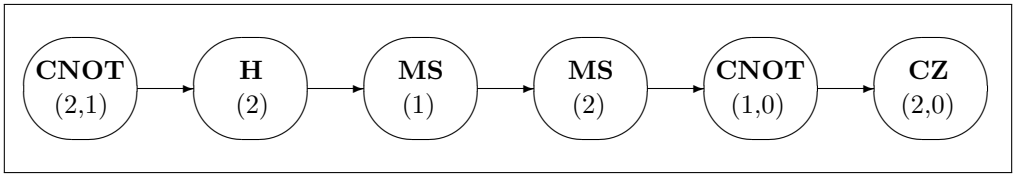
Example program for quantum teleportation

For an example of a .qcx file see Program 1, which is an implementation of quantum teleportation [45] for `smallqc`. The code section of the .qcx file translates to the linked-list representation of a program as shown in Program 2.

```
# quantumTeleportation.qcx:
#    teleports an unknown qubit from Alice to Bob
# init phase:
#    all code in here will be used to set up the system.
#    we can tensor, we can define variables and do all sorts of
#    preprocessing. eventually the last thing we do is set up the qsystem
# code phase:
#   all this code works on the qsystem tensor product
#   there are no more variable names
#   no use of the memory table
begin_init
   new_system( psi, 1 ) ;
   # initialize a random psi -- this is our unknown state
   set_amp( psi, 0, 0.132960386522622, 0.492926311010698 ) ;
   set_amp( psi, 1, 0.852892235498773, 0.109178853973861 ) ;
   # initialize alice and bob's EPR pair
   new_system( ab, 2 ) ;
   op_ZERO( ab ) ;
   # entangle alice and bob
   # hadamard on 1 (alice)
   op_H( ab, 1 ) ;
   # cnot with control: alice (1) and target: bob (0)
   op_CNOT( ab, 1, 0 ) ;
   # entangle alice's qubit with psi
   op_COMPOSE( psi, ab, phi ) ;
   rem( psi ) ;
   rem( ab ) ;
   # this final one, phi will become the qsystem (input to code phase)
   set_prog_input( phi ) ;
   print ;
   # this concludes the init phase of the program
end_init

# what follows is the code phase of the program
# the code part can be read in as an individual program to be run on
# variable input. it can be executed directly, or stored as a linked-list.
# notice that all operations by default operate on the input qsystem
# which was set at the end of the previous phase
# (hence NULL as the first argument)
begin_code
   # entangle control: psi (2), target: alice (1)
   op_CNOT( NULL, 2, 1 ) ;
   # hadamard on psi (2)
   op_H( NULL, 2 ) ;
   # measure alice's qubit (1) and then psi (2)
   measure( NULL, 1 ) ;
   measure( NULL, 2 ) ;
   # finally, correct the qubit that bob is left with
   # apply X to bob's qubit if alice's measured qubit was 1
   op_CNOT( NULL, 1, 0 ) ;
   # apply Z to bob's qubit if psi's measured qubit was 1
   op_CZ( NULL, 2, 0 ) ;
   # end of program
end_code
```


Program 1: `smallqc` program implementing quantum teleportation



Program 2: Quantum teleportation as a `smallqc` program

4.2 Genetic programming variants

Four slightly distinct GP variants were implemented and used for this study: *NQP* (Normal quantum program evolver), *EDA-QP* (EDA quantum program evolver), *ngram-QP* (N-gram quantum program evolver) and *HQP* (Hybrid-EDA quantum program evolver). They all have an evolutionary procedure in common and follow the algorithm shown in Algorithm 1 of Chapter 2 and illustrated in Figure 4.1. The common aspects of the implementations are described below. Common aspects include the solution representation, the function and terminal sets, the quality assessment (or fitness evaluation), as well as most of the parameters guiding the evolutionary process. A brief discussion follows, highlighting the distinctive search features of each GP variant.

4.2.1 Pseudo-random number generator

Random numbers play an important role in our GP runs. Random numbers are used in the random initialization of the candidate solution set, as well as in the mutation functions and all sampling functions for the EDA-based approaches. To create random[5] numbers we access the operating system's pseudo-random number generator. To ensure that no two consecutive runs of a GP are similar, we initialize the pseudo-random number generator using a seed based on the system time (in milliseconds) at the moment a GP run is begun.

4.2.2 Solution representation

A solution is implemented as a doubly-linked list of nodes, each of which encodes a certain function.[6] The order of the nodes determines a time-order of execution. See Program 2 for a rough[7] example of a program that implements quantum teleportation. Each node contains an index into a function table, as well as values for any parameters it might take. For example, the first function node in Program 2 encodes the CNOT function on control qubit 2 and target qubit 1.

[5] In truth, these are just *pseudo*-random.

[6] When talking about a node in the solution, a *function* refers to an actual quantum gate, or quantum operator. The terms operator and function will be used interchangeably in this and the following chapter.

[7] Program 2 is depicted as a *singly*-linked list to emphasize the time order from left to right. Instead of indices, the corresponding functions are shown directly.

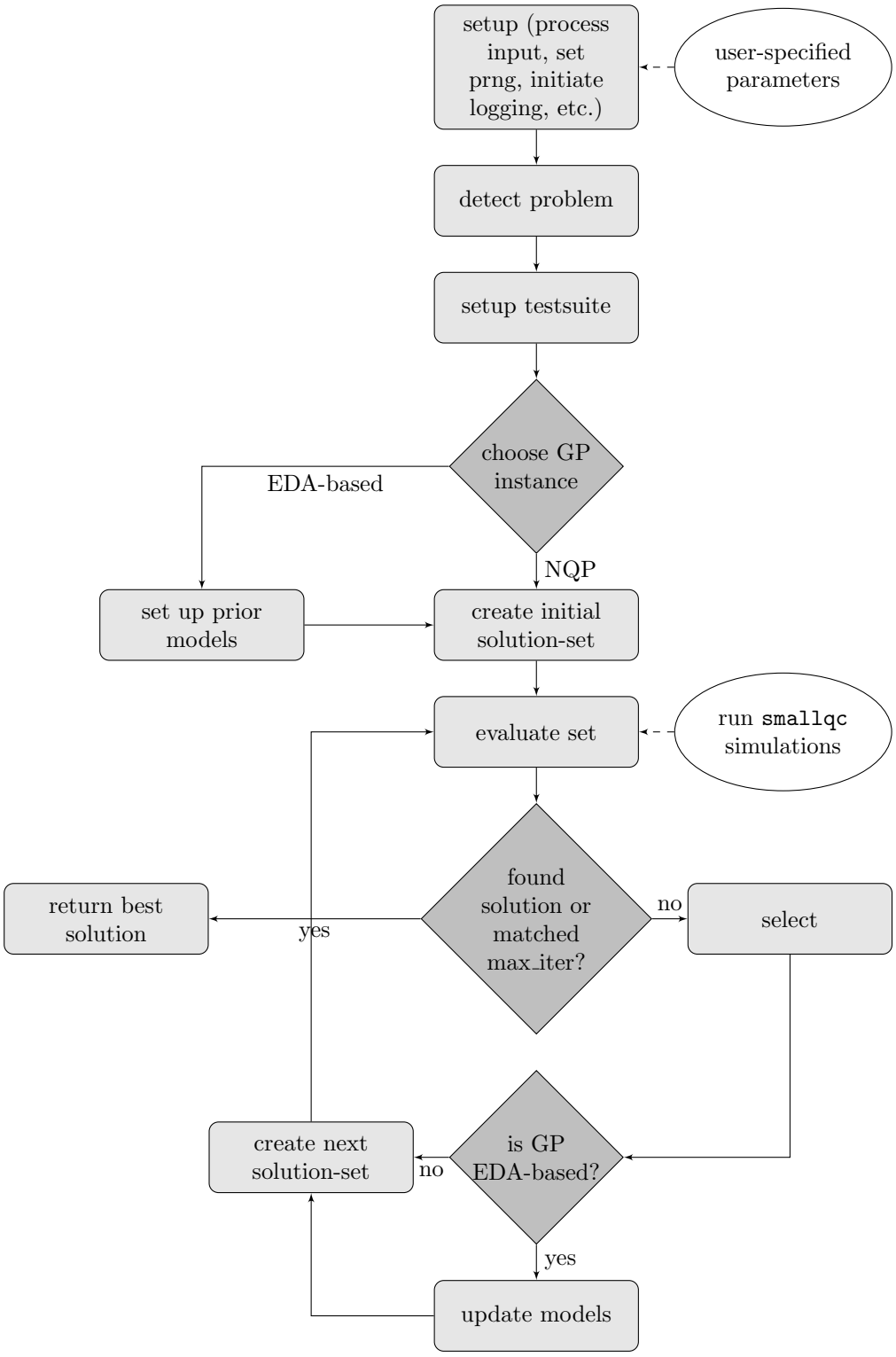


Figure 4.1: Flow of main program

The smallest program is made up of a single node. The GP evolver does not allow for empty programs. The linked-list representation encourages programs of variable size to be generated, up to and including a maximum size, which can be user-specified at the beginning of a run. This is significant, since a solution to a given problem is not always known in advance. This may not be true for some of the small test problems explored in this thesis, which have been previously explored in similar work, but for future problems this will most likely always be the case, so a GP needs the flexibility of variable-sized solutions to explore and discover for itself what works best.

Solutions are identified by two integers: *GID*, the iteration (or generation) into which the solution was introduced and *UID*, a unique identifier assigned upon creation. The GID is useful to track a solution as it persists and adapts through the iterations.[8]

4.2.3 Function set

As alluded to throughout this thesis, there is an infinity of potential single-qubit operators. With no regard to resources such as time and memory, in order to ensure that the GP scheme is capable of generating a solution to an arbitrary problem, the function set should be universal; however, as this is impossible due to the infinite size of the quantum operator set, a function set that approximates any quantum operator to some precision that is deemed to be good enough, is sufficient. As such, instead of allowing arbitrary angles for the rotation gates (which are the only gates that can produce infinite versions), the angles are restricted to integer multiples of $\frac{\pi}{4}$ and $\frac{\pi}{3}$.

Any function that is implemented in `smallqc` is also a valid function in the genetic program variants of this section. See Table 4.1 for a list of the available functions. In general, the function set may vary from run to run, as knowledge about an existing solution or the desire to experiment might influence the inclusion or exclusion of certain functions. In Chapter 5 we will see the specific function sets employed for each experiment.

4.2.4 Terminal set

The terminal set for all GP runs is made up of positive integral numbers and 0, which are either indices into the qubit register, or indices into the function set, or the pre-defined array of rotation angles.

4.2.5 GP general parameters

There are various parameters that guide the GP run. Some of these are dependent on the search operators used, which differentiate the four variants and will be described

[8] This is only useful for NQP and HQP, as the other two implementations fully replace their populations each iteration, so no solution will ever make it to the following iteration.

Parameter	Default	Description
`lgp_max_solution_size`	30	Sets the maximum solution size
`max_iter`	100	Determines the maximum number of iterations the evolutionary loop can run for before the best-so-far solution is returned
`solution_set_size`	50	Determines the number of solutions in the pool of candidates at each iteration. The pool size is maintained throughout the iterations.
`top_best`	0.25	Determines the percentage of the fitness-ranked candidate solution pool that is either used as training (for the EDA-based variants), or preferred for direct reproduction (in the normal implementation).

Table 4.2: GP general parameters

separately, later. Table 4.2 contains descriptions of the general parameters, which are *common* to all implementations.

4.2.6 Fitness evaluation and training

At each iteration of the program, the solutions are evaluated and a fitness value is associated with each solution, indicating its quality, relative to an ideal target solution.

Fitness function

The choice of fitness function determines the type of quantum algorithm generated: *exact* or *probabilistic*. The former will always return the correct result corresponding to a given input, while the latter will only return the correct result with high probability, where the probability can be adjusted. Spector et al. [67] and later Massey et al. [38] experimented with probabilistic algorithms, claiming that in some cases these were good enough solutions and their generation seemed to be much more amenable to GP techniques. The fitness function used in this study is based on that introduced by Spector et al. [67] in which three components, in order of importance, make up the overall fitness of an individual. The fitness for an individual solution is calculated as follows:

$$fitness = numFailedTests + avgErr + eff. \tag{4.1}$$

$numFailedTests$ represents the number of testcases for which the program fails to give the correct (expected) result with a probability of 0.48 or more; $avgErr$ is the average error over the $numFailedTests$ mentioned above (if the probability of success is 0.52 or more for a given testcase, its error and the testcase itself do not count towards the average); eff for *efficiency* is calculated as the size of the solution (number of function nodes, which is equivalent to the number of operations, since each function node encodes

exactly one operation) over a large constant (100,000) and it is only used as a component of the fitness when the first two components are 0 (the solution gets all testcases correct with certainty). In this way the GP focuses on finding correct solutions first and then attempts to optimize them in terms of size. Spector and Massey explain that $avgErr$ is not going to be larger than $numFailedTests$, so that a GP will always give priority to finding solutions that pass all testcases, even if this is done with slightly over 0.52 chance, rather than solutions that pass only a number of the testcases, but pass those with high likelihood.

Entanglement promotion The fitness function in Equation 4.1 has been slightly adapted with the inclusion of a fourth component called *entanglementPromotion* which is a very primitive attempt at identifying solutions which might have the *potential* for entanglement, by looking at sequences of nodes and noting whether a Hadamard gate is followed by a CNOT or a CZ gate, where the control qubit of the latter is equal to the target qubit of the Hadamard. When no promising sequence is found a solution incurs a tiny penalty, which is related to the size of the solution, similar to the eff component. It is important to note that although a sequence of such operations when applied to a zeroed state will produce entanglement, none of the nodes before or after such a sequence are checked and there is no way to know for sure that entanglement took place.[9] Furthermore, there are of course other sequences that could lead to entanglement and we do not attempt to find those. For these reasons we need to stress the fact that this component is very simple and does not measure entanglement, or even identify it. Entanglement is a complex topic and an improved component will be discussed for future work, in Chapter 6. Together with this component the fitness function used in all our GP implementations is:

$$fitness = numFailedTests + avgErr + eff + entanglementPromotion \tag{4.2}$$

Furthermore, any component may be turned off by a user option (see `-noeff, -noerr, -nomisses, -noent` in the help screen of the program) and the threshold of success determining the *numFailedTests* can be adjusted as well.[10] If `-nomisses` is used, then a threshold is not used and the average error is calculated over *all* the testcases.

Fitness testcase

A testcase is a pair of input and output structures. The input structure contains a qubit register (complex vector), but depending on the problem (see Chapter 5) might

[9] For example, while $CNOT(0,1)H(0)|00\rangle$ will create entanglement, $CNOT(0,1)H(0)H(0)|00\rangle$ will *not*, since the two Hadamard gates will cancel each other out and there will be no superposition and so the CNOT gate will not have any effect; thus, no entanglement will be produced.

[10] Of course some combinations, such as `-nomisses -noerr` do not make sense and no evolution will take place. The program will complain if all components are turned off.

additionally contain an identifier for an *oracle* function.

The output structure also contains a qubit register and a number *num_result_bits* which determines how many qubits are either *ancilla qubits*[11] or qubits whose values we do not care to know and indicates that a register should be measured starting just past these qubits, so at index *num_result_bits*. Measurement always goes from right to left (from smallest qubit to largest), so, for example, if *num_result_bits* were equal to 0, we would measure all qubits starting from the 0^{th} and if *num_result_bits* were 2 we would start measuring to the left starting at the 2^{nd} qubit.

To evaluate a candidate solution it is simulated on the input of each testcase and its output is then compared to the target output of the same testcase. In general a testcase *set* is formed by preparing all computational basis states as input and pre-computing the true (or pre-setting the desired) respective outputs; thus, a testcase set is dependent on the problem we are trying to solve.

4.2.7 Normal quantum program evolver (NQP)

The simplest quantum program evolver is the NQP which makes use of traditional mutation functions to navigate the search-space of programs[12]. There are six different mutation functions, employed with varying mutation rates. The mutation functions are described below. For default rates of application for each operator see Table 4.3.

- MUT_TYPE_0_FUNC : given a node, randomly alters its function and adjusts the parameters if the function arity has changed or expected parameter type has changed; otherwise, the parameters are left intact.
- MUT_TYPE_1_TRANSPOSITION : picks two random nodes in the linked list and swaps their positions.
- MUT_TYPE_2_PHASE_PAR : for a function node with a non-zero arity, one of the parameters is randomly chosen and incremented mod upper-range-bound.
- MUT_TYPE_3_PAR_SWAP : if the function is a controlled (or doubly-controlled) function, the (or a) control qubit is swapped with the target qubit.
- MUT_TYPE_4_RANDOM_INSERT : a new node is generated (randomly for the traditional NQP, or sampled from the underlying EDA for HQP) and inserted in a randomly chosen location in the solution.
- MUT_TYPE_5_RANDOM_DELETE : a randomly chosen node is deleted from the solution.

[11] An *ancilla* qubit is usually just a work qubit that is used in the computational work in a problem [45], but is not considered as part of the result register. It may also be presupposed to be in some specific state, such as for example, $|0\rangle$.

[12] A very simple 1-point crossover function was at first implemented, but showed no real advantage or difference and so it was left out entirely, similar to previous work by Leier and Banzhaf [33].

Operator type	Frequency
0	0.30
1	0.05
2	0.30
3	0.05
4	0.15
5	0.15

Table 4.3: Default rates of application for each search operator

There are two different mutation rates. One, *m_top* is used to mutate solutions that are in the top percentage, as determined by the general parameter *top_best* (see Table 4.2) and the second, *m_rest* is usually much larger and is used to mutate the rest of the solutions. Among the top candidates, tournament selection with a tournament size of 2 (by default) is used to select a winning candidate, who is replicated and together with the replica is then mutated into the next generation, while the loser is removed from the pool of solutions.

4.2.8 EDA data structures and sampling

The various EDA-based GP approaches all attempt to model different probability distributions. A probability distribution in our evolver is defined as an array (of dimensions 1 to 4, inclusive) of real numbers. Each distribution is associated with an independent variable, such as *length* for our 1-dimensional length model, or *node index* for a number of our function models. In the 1-dimensional case each value of the independent variable has a particular probability mass. The rest of the models attempt to relate the independent variable to one or more dependent variables, where each possible combination of values is associated with a probability mass. The sum of these masses must equal 1.0 for each model. We describe each distribution below as modelling an independent variable versus one or more dependent variables.

EDA sampling

In order to sample a function, a length, an input or a full node, a random number $x \in [0, 1]$ is first generated. A sum is calculated by traversing the respective probability distribution array, in order, and summing its entries until the sum has either met or exceeded x, at which point the current index of the array is returned. This index corresponds to a particular value (or in the multi-dimensional case, a set of values) for the dependent variable(s). An example is shown in Figure 4.2.

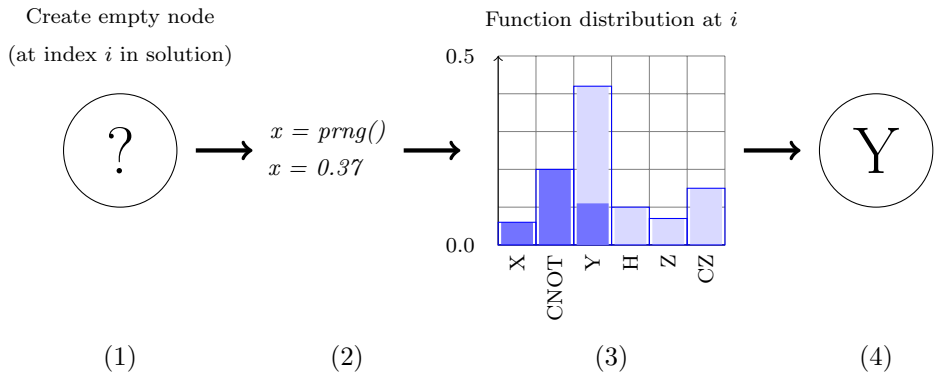


Figure 4.2: Sampling a function for a new node at some index i. (1) A new empty node is created. (2) A random number x in the range $[0, 1]$ is generated. (3) Given x, a function is sampled from the probability distribution of node i. The shaded regions in the bars for `X`, `CNOT` and `Y` represent a total probability mass equal to x and thus, we reach our sum in the region of `Y`. (4) `Y` is returned as the function for our new node.

4.2.9 EDA quantum program evolver (EDA-QP)

An EDA-GP attempts to improve at each iteration the probability distribution model describing the optimal solution(s). It does so by learning from the better solutions that were evolved up to that iteration. This means that at each iteration a part of the current solution set becomes the training set for the next round. There are no more explicit search operators. The length of the solution is also learned, similarly to the work done by Poli and McPhee [50]. For the EDA-QP, two different learning options are used:

- EDA-QP-I (separate) learns three separate distribution models:
  1. nodes vs. functions
  2. nodes vs. inputs (by arity)
  3. program length
- EDA-QP-II (mixed) learns two distribution models:
  1. nodes vs. functions vs. inputs
  2. program length

In the case of EDA-QP-I the models are smaller, but there is no dependency between the first two distribution models; essentially, this causes relationships between functions and inputs to be completely missed (in so far as arity is not concerned).[13]

[13] An early attempt of EDA-QP-I modelled inputs as versus functions, instead of versus nodes, but this was abandoned quickly, as it would be useless when a function was repeated but with different inputs each time. For example, a function that comes up very often is the Hadamard. To put a system of 2 qubits in a perfect superposition, Hadamard is applied to both qubits 0 and 1, with these initially in state $|00\rangle$.

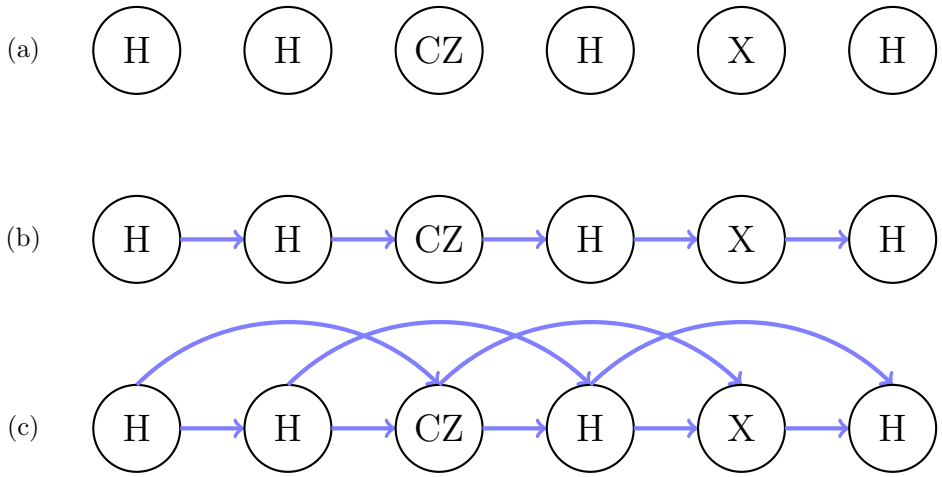


Figure 4.3: Markov chains showing dependencies among events. (a) In the uni-gram model, or 0^{th}-order Markov chain the occurrence of a particular event (presence of function `H`, for instance) is independent of all other events. (b) In the bi-gram model, or 1^{st}-order Markov chain, the occurrence of a particular event is dependent on the previous event. (c) In the tri-gram model, or 2^{nd}-order Markov chain, the occurrence of a particular event is dependent on the two most recent events.

Parameter	Value
`Unigram component`	0.3
`Bigram component`	0.3
`Trigram component`	0.4
`Learning rate (for all)`	0.01

Table 4.4: N-gram

In the case of EDA-QP-II there is a greater dependency between the inputs and functions, but the EDA model is much larger and heavier, making it more difficult to scale up to larger problems.

4.2.10 N-gram quantum program evolver (ngram-QP)

The N-gram model is inspired by previous work done by Poli and McPhee [50]. In this work a full EDA is not employed and an assumption is made that the target solutions exhibit repetitive patterns in the function sequence. At each iteration the GP attempts to learn the unigram, bigram and trigram distributions of functions and creates a new population of solutions by sampling the function sequences from these distributions, with default proportions given in Table 4.4. The unigram is equivalent to a simple probability distribution over the functions, independent of everything else. The bigram is a first-order Markov chain [6, 14], where the next function is predicted by the previous function. The trigram, a second-order Markov chain uses the two previous functions to predict the

next one. For an illustration of the dependencies in the various N-gram models, see Figure 4.3. Like EDA-QP, ngram-QP also has two different learning options:

- NGRAM-QP-I (separate) learns three separate distribution models:
  1. trigram for functions
  2. nodes vs. inputs (same as in EDA-QP-I)
  3. program length
- NGRAM-QP-II (mixed) learns three separate models:
  1. bigram for {function, target}
  2. target qubits vs. rest of inputs (by arity)
  3. program length

NGRAM-QP-I uses separate models for the inputs and functions, where the inputs are location-dependent as with EDA-QP-I and are once again independent of the functions. To sample a new solution with NGRAM-QP-I, a size is randomly generated, after which the first node in the sequence is picked using the unigram, the second node using the bigram (depending on the first node) and starting with the third node a Markov process is generated using the trigram, bigram and unigram models, with the different probabilities given in Table 4.4. The inputs to the functions are then sampled separately of this process, although, the arity of the function dictates how many inputs are sampled.

NGRAM-QP-II is the favoured learning model. Out of all the probability models, the ones learnt by NGRAM-QP-II are the only ones to both not be location-dependent and to attempt to link the functions to their target inputs.

4.2.11 Hybrid-EDA quantum program evolver (HQP)

HQP aims to combine the best of both the traditional GP and the EDA-GP. A distribution model is learnt and updated at each iteration, just as in a usual EDA-GP. HQP uses an explicit solution set at each iteration and navigates the search space much like the NQP, with the same six operators. The difference is that a function mutation[14] is applied

[14] Currently inputs are mutated as in NQP, without the use of the probability distribution models, except when:

1. The arity of the mutated function is different from the previous function, or the type (rotation, non-rotation) function is different from the previous function, in which case inputs are also sampled
2. MUT_TYPE_4_RANDOM_INSERT is used, in which case the entire node, including inputs, is sampled from the probability distribution model

The reason inputs are generally not sampled in HQP is because we felt inputs are difficult to model probabilistically, as they are either node (location) -dependent, which we wanted to avoid in HQP, or they are function-dependent, which limits the usefulness of a model, since a function will most likely appear multiple times with different inputs.

more *intelligently*, as instead of a random change, the EDA is used to predict a more suitable transformation. Additionally, when a solution candidate loses a tournament to another solution candidate, HQP has the choice (with some user-defined probability) to sample an entire solution completely from the models instead of copying the winner directly into the next iteration, as NQP does. HQP may choose any of the previously-discussed EDA-based learning models as its underlying learner. We expected the HQP to perform best out of the EDA-based instances as well as better than the NQP, because it combines the ability to do high rate mutations, that is, it can jump across the rugged fitness landscape, but it still allows us to garner knowledge from previous solutions in order to help determine exactly which jumps might be better.

4.2.12 On the learning of models

All probability distribution models are initialized uniformly, such that sampling a solution results in a completely random product. At each iteration a sample set of candidate solutions is created from the models and each individual is evaluated and ranked. The top ranked candidate solutions (see Table 4.2) are used to update the probability distribution models for the next iteration. Although the models vary somewhat by the GP variant in use, the update rule is similar. Based on the original PIPE (Probabilistic incremental program evolution) algorithm [60, 59], the update rule works to gradually increase the probabilities of events that are observed more frequently, relative to those that appear less often. The idea is that events observed in a top candidate solution are quite probably contributing positively to the overall fitness of that candidate and are thus good features to incorporate in the probability model and keep around by promoting them through the model which is gradually becoming partial to them. An update increment is very small (and can be adjusted by the user) and is calculated as:

$$Prob[eventObserved] = Prob[eventObserved] + \lambda * \mu \tag{4.3}$$

Here μ is the update unit, which is a small probability mass calculated as $\frac{1.0}{TE}$, where TE represents the total number of possible distinct and valid [15] events associated with the distribution. The learning rate λ is usually set somewhere between 0.05 and 0.1. What constitutes an event being observed depends on the distribution in question; an event might be a specific function type for a distribution modelling functions, or it might be an input for a distribution modelling inputs. An event might also be a pair of a function and an input or something more complex, as we saw in the case of ngram-QP.

The batch of top solutions are all examined in turn and each observed event's probability is raised. Once all solutions have been examined, all models who have seen an

[15] We say *valid* events instead of *total* events, as some events are invalid and completely ignored from the model; for example seeing a CNOT gate with inputs 0 and 0 for control and target, respectively, would have a probability of 0.0 to start out with and would never have a chance of seeing an increase.

update are re-normalized. The general EDA-based evolutionary learning process follows an algorithm as seen in Algorithm 2.

```
set iteration = 0
set solutionFound = false
set solutionSet = NULL
repeat
    solutionSet = sample set of candidate solutions from probability models
    foreach candidate in solutionSet do
        evaluate candidate and save fitness
    end
    sort candidates according to fitness
    if solution in solutionSet then
        solutionFound = true
    end
    learningSet = select top eda_top of best candidates from solutionSet
    foreach candidate in learningSet do
        foreach event from model seen in candidate do
            update probability of event
        end
    end
    re-normalize model
    iteration = iteration + 1
until while iteration has not exceeded max_iter and solutionFound != true;
return top candidate in solutionSet
```

Algorithm 2: Evolutionary loop and learning in the EDA-based GP variants

Sub-optimal convergence

Even though the update increment is slight, there is a chance a distribution model will face premature and sub-optimal convergence. In order to ensure that the GP does not lose its ability to explore, the user can set an option to force a learner to use a randomly-generated set of solutions every N number of iterations (where N is also user-specified), to perturb its distribution models.

4.3 An EDA example

For an example, we run through one iteration of NGRAM-QP-II, where we have set the learning rate λ fairly high for effect.
Recall from Section 4.2.8 that NGRAM-QP-II learns three separate models:

1. bigram for {function, target}

2. target qubits vs. rest of inputs (by arity)

Length	Probability
1	0.33
2	0.33
3	0.33

(a) Length model (in number of nodes)

Function	Target	Probability
X	0	0.25
X	1	0.25
CNOT	0	0.25
CNOT	1	0.25

(b) Prior model (functions vs. target qubits)

Function1	Target1	Function2	Target2	Probability
X	0	X	0	0.0625
X	0	X	1	0.0625
X	0	CNOT	0	0.0625
X	0	CNOT	1	0.0625
X	1	X	0	0.0625
X	1	X	1	0.0625
X	1	CNOT	0	0.0625
X	1	CNOT	1	0.0625
CNOT	0	X	0	0.0625
CNOT	0	X	1	0.0625
CNOT	0	CNOT	0	0.0625
CNOT	0	CNOT	1	0.0625
CNOT	1	X	0	0.0625
CNOT	1	X	1	0.0625
CNOT	1	CNOT	0	0.0625
CNOT	1	CNOT	1	0.0625

(c) Transition model ({function, target} pairs)

Target	Control	Probability
0	0	0.0
0	1	0.5
1	0	0.5
1	1	0.0

(d) Input model (target vs. others)

Table 4.5: Distribution models at iteration 0

3. program length

Assume the evolver is run with the following parameters:

- *GPType* : NGRAM-QP-II
- *functionSet* : {CNOT, X}
- *systemSize* : 2
- λ : 0.1
- *solutionSetSize* : 3

- *maxSolutionLength* : 3

At iteration 0, we thus have the (uniform) distributions seen in Table 4.5. The four different update units for each model are as follows: 0.33 (Length), 0.25 (Unigram), 0.0625 (Bigram), 0.5 (Input). Hence, the update increments for each, calculated as $(\lambda * \mu)$ are: 0.033 (Length), 0.025 (Unigram), 0.00625 (Bigram), 0.05 (Input).

Next, 3 random solutions are sampled from these models:

1. `X(0) - CNOT(0,1) - X(0)`
2. `CNOT(0,1) - CNOT(0,1)`
3. `X(1) - CNOT(0,1)`

For the Length model, 3 events are observed: 3, 2, 2. Thus, the probabilities for 3 and 2 are increased.

For the Prior model, 3 events are observed: {X, 0}, {CNOT, 1} and {X, 1}.

For the Transition model, 4 events are observed: {X, 0, CNOT, 1}, {CNOT, 1, X, 0}, {CNOT, 1, CNOT, 1}, {X, 1, CNOT, 1}.

For the Input model, 4 events are observed (where we have two inputs, as the rest are single input functions): {1, 0}, {1, 0}, {1, 0}.

The models are updated to reflect these observations. Finally, the new models at iteration 1 can be seen in Table 4.6.

Length	Probability
1	0.30
2	0.36
3	0.33

(a) Length model (in number of nodes)

Function	Target	Probability
X	0	0.26
X	1	0.26
CNOT	0	0.23
CNOT	1	0.26

(b) Prior model (functions vs. target qubits)

Function1	Target1	Function2	Target2	Probability
X	0	X	0	0.061
X	0	X	1	0.061
X	0	CNOT	0	0.061
X	0	CNOT	1	0.067
X	1	X	0	0.061
X	1	X	1	0.061
X	1	CNOT	0	0.061
X	1	CNOT	1	0.067
CNOT	0	X	0	0.061
CNOT	0	X	1	0.061
CNOT	0	CNOT	0	0.061
CNOT	0	CNOT	1	0.061
CNOT	1	X	0	0.067
CNOT	1	X	1	0.061
CNOT	1	CNOT	0	0.061
CNOT	1	CNOT	1	0.067

(c) Transition model ({function, target} pairs)

Target	Control	Probability
0	0	0.0
0	1	0.43
1	0	0.57
1	1	0.0

(d) Input model (target vs. others)

Table 4.6: Distribution models at iteration 1

Chapter 5

Experiments and results

Quantum algorithms are evaluated based on their time and space complexity relative to their most-effective classical counterparts. This motivates the search for super quantum algorithms, but we think it is not entirely fair to dismiss the search for quantum algorithms that are *as good* as their classical counterparts. However, perhaps the motivation is required since quantum technology and hardware are not easy to come by and some may feel that there need to be clear benefits before resources are expended in the creation of quantum computers. We take the view that quantum computation is interesting in its own right for what it might potentially teach us about nature, information and limits on computation [10] and for this reason lesser quantum algorithms should also be explored.

In an effort to determine whether the EDA-based GPs presented in the previous chapter are indeed capable of detecting useful patterns in quantum programs and helping with the automatic programming of quantum computers, we tested all GP variants on six different problems. We begin this chapter by outlining each problem and discussing each problem's individual experimental setup. The rest of the chapter demonstrates and analyzes the results obtained for each problem. We also discuss difficulties encountered and summarize all experiments.

5.1 Experimental set-up

To evaluate a quantum program as produced by one of our GP variants, with the goal of solving a particular problem, it is necessary to set up a suite of testcases which only a good solution would pass completely.

Each of the six problems we tackled requires a unique set-up and distinct set of testcases. We introduce each problem in the following sections and give a detailed account of the problem set-up.

5.1.1 On blackbox functions

A large proportion of extant quantum algorithms can be categorized as *blackbox optimization* algorithms [27]. To remind the reader, a *blackbox function* is a function which can be queried, but whose internals are not known.

Calls to blackbox functions are always expensive to make and for this reason all blackbox optimization problems aim to reduce the number of necessary calls to the given blackbox function.

5.1.2 Problem descriptions

We chose six problems, some of which were expected to be more suitable to the EDA-based genetic programming instances. We felt the need to have a decent balance of problems and so we have introduced a pair of arithmetic problems into our repertoire, as well as a stranger sort of problem: the imperfect copy machine. We have categorized two problems as *quantum arithmetic* problems, two as *blackbox optimization* problems and two as *miscellaneous*, since they are neither blackbox-based nor quantum arithmetic problems. Short descriptions of all problems follow here for easy later reference.

Problem #1: Deutsch-Jozsa

The problem [12, 45, 54] is given in terms of a binary blackbox function $f(x)$ which takes as input a quantum register x and outputs either 1 or 0. Nothing is known about the function except that it must either be *balanced*, in which case it will return 0 for half of its domain and 1 for the other half of the domain, or *uniform*, in which case it will either always return 0 or always return 1 for all values of its domain. Calling this blackbox function is said to be expensive. As such, the objective is to determine the property of the blackbox (i.e. balanced or uniform) with as few calls to it as possible. Deutsch-Jozsa is a famous problem and has been solved with GP previously [66].

Category: blackbox optimization
Cases: 1-qubit, 2-qubit
Required system sizes: 2 qubits, 3 qubits

Problem #2: Imperfect copy machine

Due to the no-cloning theorem we know that it is impossible to perfectly copy an arbitrary and unknown quantum state. This follows directly from the linearity of quantum me-

chanics.[1] The theorem does not preclude a copy circuit for computational basis states.[2] Thus, it should be possible to copy a state, as long as that state is *not* in a superposition. We have come up with the problem of the *imperfect copy machine*, whose goal is to devise a quantum algorithm that produces a copy of a computational basis state.

Category: miscellaneous
Cases: 1-qubit, 2-qubit
Required system sizes: 3 qubits, 6 qubits

Problem #3: Quantum addition

This is the first of two quantum arithmetic problems we expected to be quite suitable for the N-gram approach. Given two input quantum registers of identical size, the problem is to find the binary addition of the two computational basis states.

Category: quantum arithmetic
Cases: 1-qubit
Required system size: 3 qubits, 4 qubits

Problem #4: Quantum multiplication

Given two input quantum registers of identical size, the problem is to find the binary multiplication of the two computational basis states. Quantum multiplication is the

[1] A simple proof of the no-cloning theorem, as given by Nielsen and Chuang [45] is as follows: Let $U : |x\rangle|z\rangle \mapsto |x\rangle|x\rangle$ be a unitary operator capable of copying an unknown arbitrary state $|x\rangle$. Let $|x\rangle$ and $|y\rangle$ be two non-orthogonal states. Then

$$U(|x\rangle \otimes |z\rangle) = |x\rangle \otimes |x\rangle \tag{5.1}$$
$$U(|y\rangle \otimes |z\rangle) = |y\rangle \otimes |y\rangle \tag{5.2}$$

Taking the inner product of (5.1) and (5.2) gives the following on the left side:

$$\begin{aligned}(\langle x| \otimes \langle z|)U^\dagger U(|y\rangle \otimes |z\rangle) &= (\langle x| \otimes \langle z|)U^\dagger U(|y\rangle \otimes |z\rangle) \\ &= (\langle x| \otimes \langle z|)(|y\rangle \otimes |z\rangle) \leftarrow \text{by the unitarity of } U \\ &= \langle x|y\rangle\langle z|z\rangle \leftarrow \text{by linearity of a tensor product} \\ &= \langle x|y\rangle \cdot 1 \leftarrow \text{by unitary requirement of ket } z \\ &= \langle x|y\rangle \end{aligned} \tag{5.3}$$

Similarly, on the right side we have:

$$\begin{aligned}(\langle x| \otimes \langle x|)(|y\rangle \otimes |y\rangle) &= \langle x|y\rangle\langle x|y\rangle \\ &= (\langle x|y\rangle)^2 \end{aligned} \tag{5.4}$$

Equating the left side (5.3) with the right side (5.4)

$$\langle x|y\rangle = (\langle x|y\rangle)^2 \tag{5.5}$$

forces either (1) x or y to be 0 or (2) x and y to be orthogonal. The second option contradicts our original assumption that x and y are non-orthogonal and the first option contradicts our assumption that x and y are arbitrary.

[2] Recall that a computational basis state is *not* in a superposition, and can be likened to a normal binary register.

basis for quantum exponentiation, which is extremely important, especially as part of the quantum Fourier transform [77].

Category: quantum arithmetic
Cases: 1-qubit, 2-qubit
Required system sizes: 3 qubits, 8 qubits

Problem #5: Finding the minimum

Given a bijective function $f(x) : [0, N] \rightarrow [0, N]$, the goal is to find the value for x such that $f(x)$ returns the minimum value of its range (i.e. 0). This problem is similar to Massey's PFMAX problem [40], in which he finds the *maximum* of the function instead. Since the functions are all one-to-one, we can view them as permutations. For example, the identity function for $N = 5$ would simply be $f(x) = \{0, 1, 2, 3, 4, 5\}$. That is, $f(0) = 0; f(1) = 1; f(2) = 2; f(3) = 3; f(4) = 4; f(5) = 5$.

Category: miscellaneous
Cases: 1-qubit, 2-qubit
Required system sizes: 2 qubits, 4 qubits

Problem #6: 2-element quantum sorting

For the general sorting problem, a list of numbers (not necessarily distinct) is given and the goal is to find a new listing of the numbers in non-decreasing order. The idea is to first find a quantum sort program for a particular pre-specified list of numbers, while the ultimate goal is to find a quantum sort algorithm for an arbitrary list of numbers. This general problem is more difficult to solve, as it would involve large qubit registers (to hold the list of numbers) and it would necessitate conditional logic and processing (such as swaps) of numbers. The plan was to start off with a simpler sorting problem, where only two numbers are considered and their positions are switched whenever the first is larger than the second. In solving this simpler problem it is hoped that an actual *algorithm* will easily be generalized to larger problems.

Category: blackbox optimization
Cases: 1-qubit, 2-qubit
Required system sizes: 3 qubits, 5 qubits

5.1.3 Implementation and test environment

All the code was written in C, on a Linux computer running Debian Wheezy (kernel 3.2.0-2-amd64). Code was compiled with the GNU compiler gcc (version Debian 4.7.2-5). All experiments were run on the same system.

The computer's specifications are:
Model name : Intel(R) Core(TM) i5-2520M CPU @ 2.50GHz
CPU MHz : 2501.000
Cores : 4
Cache (level 3) size : 3072 KB
Architecture : x86_64

5.1.4 Parameter settings

Before we describe each experiment's individual set-up we start by explaining the different parameters available, together with their settings. As there are three GP variants we are testing, plus the non-EDA GP variant (NQP) there are many different combinations of parameters.

Name	*Default*	*Parameter*
GP type	EDA-QP-I	`-g [012345]`
GP sub-type	EDA-QP-I	`-sub [0123]`
System size	3	*NA*[a]
Solution set size	100	`-n <size>`
Max iterations	100	`-i <max>`
Max sol. length	10	`-len <length>`
Selection %	0.20	`-top <per>`
Fitness	all components	`-noerr, noeff, -noent, -nomisses`
Function set	all gates	*NA*
Mut. rate top	0.05	`-mrt <prob>`
Mut. rate rest	0.25	`-mrr <prob>`
Mut. weights	0.3/0.05/0.3/0.05/0.15/0.15	`-mut <mt0> ... <mt5>`
EDA learning rate	0.05	`-lre <lambda>`
N-gram learning rate	0.01	`-lrg <lambda>`
Length learning rate	0.01	`-lrl <lambda>`
1/2 -gram	0.05/0.95	`-n2 <uni> <bi>`
1/2/3 -gram	0.03/0.12/0.85	`-n3 <uni> <bi> <tri>`
Perturb. freq.	50	`-pfreq <iters>`
Perturb. set size	50	`-pfrac <frac>`
Learn len. model	Yes	`-nolen`

Table 5.1: GP parameters and their default values

[a] The *system size*, *function set* and *rotation angle set* are all adjusted programmatically. Thus, recompilation of the program is required for any changes and there are currently no available arguments to manipulate these at runtime.

The most important parameters are the *solution set size*, the *maximum number of iterations*, the *maximum solution length* and the various learning rates. Table 5.1 lists all the different parameters, together with their default values and, where applicable, their corresponding arguments recognized by our main GP evolver program. The top of the

table contains parameters common to all GP variants, as well as the different mutation rates for the six mutation functions (employed by HQP and NQP) discussed in Chapter 4. The bottom part of the table contains EDA-specific learning rates and weights for the n-gram models. *Perturb. freq* determines how often (if at all) the EDA-based GPs introduce random samples to perturb their models and the *size* of the set of these random samples is controlled as a fraction of the solution set size by the parameter *perturb. set size*.

Unless explicitly stated, the mutation rates for the search operators described in Section 4.2.7 for a particular run of NQP or HQP are just the defaults given in Table 4.3.

5.1.5 Problem #1: Deutsch-Jozsa

The Deutsch-Jozsa problem was a good starting point for our experiments, as it has been solved through GP in the past [33, 66], rendering it a good choice for validating our code.

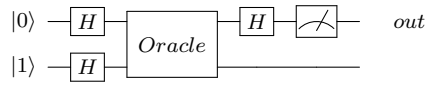
It is also one of the well-known quantum problems which has an established algorithm. This algorithm is remarkable, as it makes use of quantum superposition and parallelism to solve the problem with as little as one oracle call! Classically this is impossible to do[3], as in the worst case we would have to go through at least $N/2 + 1$ calls to the oracle (where N is the total number of distinct inputs) before we could conclude with certainty that the function is balanced or uniform.[4] The Deutsch-Jozsa algorithm assumes an input of $|01\rangle$. While the Deutsch-Jozsa algorithm is short and elegant, in an effort to not restrict the evolution to algorithms mimicking Deutsch-Jozsa, the more general input state of $|00\rangle$ is assumed, same as in previous work by Spector et al.[67], which naturally leads to an inescapable increase in the length of potential solutions, in order to compensate for the less specific input.

Problem setup for the 1-qubit case

For the 1-qubit problem there are only 4 possible functions $f(x) : [0, 1] \rightarrow [0, 1]$:

[3] It is impossible to do *deterministically*.

[4] The Deutsch-Jozsa algorithm generalizes to multiple qubits. The simplest case, for the 1-qubit problem (also known as the *Early Promise Problem*[45] is illustrated by the following circuit:



where —[H] represents the Hadamard gate, the upper wire represents the 1^{st} qubit, the lower wire represents the 0^{th} qubit (from the right, in ket notation), —[meter] represents a measurement (in this case on qubit 1) and time flows from left to right. The oracle is a multi-qubit gate that acts on the entire system (of two qubits), whose input is given by the qubits $|1\rangle$ (zeroth) and $|0\rangle$ (first), equivalent to the composite state $|01\rangle$.

1. $f_0(x) : f_0(0) = 0; f_0(1) = 0$ (uniform)
2. $f_1(x) : f_1(0) = 1; f_1(1) = 1$ (uniform)
3. $f_2(x) : f_2(0) = 0; f_2(1) = 1$ (balanced)
4. $f_3(x) : f_3(0) = 1; f_3(1) = 0$ (balanced)

To make use of function $f_i(x)$ it has to be wrapped in a unitary transformation, U_{f_i}, such that a call to the oracle is reversible. The action of U_{f_i} on an input register $|xy\rangle$ is $U_{f_i}|x\rangle|y\rangle \rightarrow |x\rangle|f_i(x) \oplus y\rangle$.

Each oracle function U_{f_i} is considered a separate testcase; thus, there are only four distinct fitness testcases for this instance of the Deutsch-Jozsa problem. A program which is a potential solution is simulated on an input of $|00\rangle$ for each different oracle and in the end the leftmost qubit is measured to produce the result. If the result is 0, then the function should be uniform (constant); else, the function should be balanced. The result is then verified against the expected output.

Penalty functions

As mentioned previously, oracle calls are deemed expensive, so as in any blackbox optimization problem, the goal is to reduce the overall number of oracle calls. A potential solution which is found to have more than one oracle call is given a small fitness increase[5] proportional to the number of extra oracle calls. Similarly, a potential solution which is found to have no oracle calls is given an extremely large fitness increase, as it cannot gain access to the results of the oracle and thus to the implicit definition of $f_i(x)$.

Problem setup for the 2-qubit case

For the 2-qubit problem there are now 8 possible functions, of which again 2 are uniform and the last 6 are balanced: $f(x) : [0, 1] \rightarrow [0, 1]$:

1. $f_0(x) : f_0(00) = 0; f_0(01) = 0; f_0(10) = 0; f_0(11) = 0$ (uniform)
2. $f_1(x) : f_1(00) = 1; f_1(01) = 1; f_1(10) = 1; f_1(11) = 1$ (uniform)
3. $f_2(x) : f_2(00) = 0; f_2(01) = 0; f_2(10) = 1; f_2(11) = 1$ (balanced)
4. $f_3(x) : f_3(00) = 0; f_3(01) = 1; f_3(10) = 0; f_3(11) = 1$ (balanced)
5. $f_4(x) : f_4(00) = 0; f_4(01) = 1; f_4(10) = 1; f_4(11) = 0$ (balanced)
6. $f_5(x) : f_5(00) = 1; f_5(01) = 0; f_5(10) = 0; f_5(11) = 1$ (balanced)
7. $f_6(x) : f_6(00) = 1; f_6(01) = 0; f_6(10) = 1; f_6(11) = 0$ (balanced)

[5] A standardized fitness measure is used (for all experiments), in which a fitness value of 0 is best, while a fitness value becomes worse as it increases from 0.

8. $f_7(x) : f_7(00) = 1; f_7(01) = 1; f_7(10) = 0; f_7(11) = 0$ (balanced)

To verify a solution, it is simulated on the input state of $|000\rangle$ and the leftmost 2 qubits of the final state are measured. If this measurement yields 00 with 100% certainty, then the function is said to be constant. If the measurement has no chance (probability amplitude is zero) of yielding 00, then the function is balanced[6]. Again, the actual result is then compared against the known result (uniform or balanced) for each testcase.

5.1.6 Problem #2: Imperfect copy machine

The imperfect copy machine problem is interesting for two reasons. First of all, as mentioned earlier, due to the no-cloning theorem we know it is impossible to copy an unknown arbitrary quantum state, but we are interested in how we could copy a computational basis state. Second, with a little thought we could come up with a very simple algorithm for copying a computational basis state and this algorithm would have a lot of repetition, so we hoped our GP could replicate our algorithm or come up with something more interesting, both of which it did.

Problem setup for the 1-qubit imperfect copier

For the 1-qubit problem we need to have 3 qubits in total. The leftmost qubit (qubit 2) is the input we wish to copy, the middle qubit (qubit 1) is the output qubit, where we wish to replicate qubit 2 and the rightmost qubit (qubit 0) is simply an ancilla bit, or work bit. We do not measure the work bit and we assume it is always set to 0 upon input. This means that there are only 4 unique combinations of our inputs and thus, for the 1-qubit problem there are only 4 possible fitness testcases:

1. **in**: 000 ; **out**: 00*
2. **in**: 010 ; **out**: 00*
3. **in**: 100 ; **out**: 11*
4. **in**: 110 ; **out**: 11*

The '*' is a wildcard, as we do not care about the final value of the ancilla bit.

Problem setup for the 2-qubit imperfect copier

For the 2-qubit case we use two ancilla bits, but this time our copy register is 2 qubits long and so the result register must also be 2 qubits long, which brings the total to 6 qubits. Since we only measure the leftmost 4 qubits there are now 16 possible combinations of inputs for our tests:

[6] To distinguish a balanced function, the amplitude must be zero for 00, otherwise we could at best achieve a probabilistic solution.

1. **in**: 000000 ; **out**: 0000*
2. **in**: 000100 ; **out**: 0000*
3. **in**: 001000 ; **out**: 0000*
4. **in**: 001100 ; **out**: 0000*
5. **in**: 010000 ; **out**: 0101*
6. **in**: 010100 ; **out**: 0101*
7. **in**: 011000 ; **out**: 0101*
8. **in**: 011100 ; **out**: 0101*
9. **in**: 100000 ; **out**: 1010*
10. **in**: 100100 ; **out**: 1010*
11. **in**: 101000 ; **out**: 1010*
12. **in**: 101100 ; **out**: 1010*
13. **in**: 110000 ; **out**: 1111*
14. **in**: 110100 ; **out**: 1111*
15. **in**: 111000 ; **out**: 1111*
16. **in**: 111100 ; **out**: 1111*

5.1.7 Problem #3: Adder

Addition is an essential operation in classical computing and it is no less important in quantum computing. For this experiment we attempted to generate a 1-qubit *half-adder* and a 1-qubit *full-adder*.

Problem setup for the 1-qubit half-adder

Given two qubits $|x\rangle$ and $|y\rangle$ in basis states, a quantum half-adder returns their sum in two output registers, where one holds the XOR of x and y and the other holds a possible carry-bit.

We used a format as given in the work by Gossett [22] to define the input-output pairings. For the 1-qubit half-adder we are using 3 qubits and define addition as a unitary operator

$$U_{add}|x\rangle|y\rangle|z\rangle \mapsto |x\rangle|x \oplus y\rangle|(x \wedge y) \oplus z\rangle.$$

Since we do not clear the output register $|z\rangle$ beforehand and do not pre-specify its value, we have 8 different inputs spanning the 3-qubit space and hence 8 testcases:

1. **in**: 000 ; **out**: 000 $(0 + 0 = 0)$
2. **in**: 001 ; **out**: 001
3. **in**: 010 ; **out**: 010 $(0 + 1 = 1)$
4. **in**: 011 ; **out**: 011
5. **in**: 100 ; **out**: 110 $(1 + 0 = 1)$
6. **in**: 101 ; **out**: 111
7. **in**: 110 ; **out**: 101[7] $(1 + 1 = 2)$
8. **in**: 111 ; **out**: 100

Problem setup for the 1-qubit full-adder

The full-adder is just a little more complicated, as it takes *three* inputs, where in addition to x and y, it accepts a *carry-in* bit, c. The full-adder then computes the sum of x, y and c.

The 1-qubit full-adder is very similar to the half-adder, except it uses 4 qubits, where the 1^{st} (from the right) holds a carry-in bit c. The output now keeps both x and y, but overwrites c and the arbitrary output register z and so the unitary addition operator we are looking for should act as follows:

$$U_{add}|x\rangle|y\rangle|c\rangle|z\rangle \mapsto |x\rangle|y\rangle|x \oplus y \oplus c\rangle|(x \wedge y) \vee (x \wedge c) \vee (y \wedge c)\rangle.$$

There are now 16 testcases spanning the 4-qubit space and again the output should be read in reverse, since the significant bit (carry-out bit) is always the rightmost of the system:

1. **in**: 0000 ; **out**: 0000 $(0 + 0 + 0 = 0)$
2. **in**: 0001 ; **out**: 0001
3. **in**: 0010 ; **out**: 0010 $(0 + 0 + 1 = 1)$
4. **in**: 0011 ; **out**: 0011
5. **in**: 0100 ; **out**: 0110 $(0 + 1 + 0 = 1)$
6. **in**: 0101 ; **out**: 0111
7. **in**: 0110 ; **out**: 0101 $(0 + 1 + 1 = 2)$

[7] Remember that the 0^{th} qubit here holds the carry-bit, which means that the output is reversed; that is, 01 should read 2.

8. **in**: 0111 ; **out**: 0100
9. **in**: 1000 ; **out**: 1010 $(1 + 0 + 0 = 1)$
10. **in**: 1001 ; **out**: 1011
11. **in**: 1010 ; **out**: 1001 $(1 + 0 + 1 = 2)$
12. **in**: 1011 ; **out**: 1000
13. **in**: 1100 ; **out**: 1101 $(1 + 1 + 0 = 2)$
14. **in**: 1101 ; **out**: 1100
15. **in**: 1110 ; **out**: 1111 $(1 + 1 + 1 = 3)$
16. **in**: 1111 ; **out**: 1110

5.1.8 Problem #4: Multiplier

The multiplier problem was chosen for its importance in mathematical operations as well as the suspicion that it might be a good problem to mine for patterns and repetitive sequences. In the general multiplication problem we are given two registers x and y (of equal size) and an output register z, whose contents we make no assumptions about and the goal is to output the multiplication of x and y in z. For z to hold all possible results of the multiplication of two registers of equal size, z must (in general) be twice the size of the input registers. Of course, we need a unitary operator and so we are looking for a quantum circuit that implements the following unitary operator for multiplication:

$$U_{mult}|x\rangle|y\rangle|z\rangle \mapsto |x\rangle|y\rangle|mult(x, y) \oplus z\rangle.$$

For our purposes here we *do* make an assumption about the output register z; that is, we expect it to start out in a cleared state, 0.

Problem setup for the 1-qubit multiplier

In order to make the problem reversible, even for the 1-qubit operand case we need to have 3 qubits in total: 1 qubit for each operand and 1 qubit to hold the answer. Since the maximum digit a single qubit can hold is 1, the greatest product we can produce with two qubits is 1, so only one qubit is needed for the answer. We cannot overwrite the second operand's register with the answer, since unlike in addition, we cannot always use a single operand together with the product to determine the other operand. For example, if the result is 0 and we know one of the factors is 0, then at best we can only *guess* what the other factor is (1 or also 0). Again, our last qubit is the answer qubit and we assume that it starts off as a 0 (or that a zeroing operation can easily be applied

to it before the multiplication). When measuring the answer we must measure all the qubits to ensure that the answer is reversible (we can determine both operands from the result). The product of the multiplication is the rightmost qubit.

1. **in**: 000 ; **out**: 000 $(0 * 0 = 0)$
2. **in**: 010 ; **out**: 010 $(0 * 1 = 0)$
3. **in**: 100 ; **out**: 100 $(1 * 0 = 0)$
4. **in**: 110 ; **out**: 111 $(1 * 1 = 1)$

Problem setup for the 2-qubit multiplier

The more interesting problem is a 2-qubit operand multiplier. A 2-qubit register can hold a range of values: $[0, 3]$. That means that the maximum product we can form from two 2-qubit registers is $3 * 3 = 9$, which in binary form is 1001 and requires a 4-qubit register. Requiring that the problem be reversible, we need two 2-qubit registers for the two factors and a 4-qubit register for the answer, for a total of 8 qubits. We make the problem a little easier by assuming that the answer register always starts off in state $|0000\rangle$, which could be achieved by swapping the respective qubits with specially-prepared $|0\rangle$ qubits before the multiplication. Since we have only 2 registers to multiply and each 2-qubit register can hold 4 distinct values, there are $4 * 4 = 16$ combinations and so there are 16 testcases:

1. **in**: 00|00|0000 ; **out**: 00|00|0000 $(0 * 0 = 0)$
2. **in**: 00|01|0000 ; **out**: 00|01|0000 $(0 * 1 = 0)$
3. **in**: 00|10|0000 ; **out**: 00|10|0000 $(0 * 2 = 0)$
4. **in**: 00|11|0000 ; **out**: 00|11|0000 $(0 * 3 = 0)$
5. **in**: 01|00|0000 ; **out**: 01|00|0000 $(1 * 0 = 0)$
6. **in**: 01|01|0000 ; **out**: 01|01|0001 $(1 * 1 = 1)$
7. **in**: 01|10|0000 ; **out**: 01|10|0010 $(1 * 2 = 2)$
8. **in**: 01|11|0000 ; **out**: 01|11|0011 $(1 * 3 = 3)$
9. **in**: 10|00|0000 ; **out**: 10|00|0000 $(2 * 0 = 0)$
10. **in**: 10|01|0000 ; **out**: 10|01|0010 $(2 * 1 = 2)$
11. **in**: 10|10|0000 ; **out**: 10|10|0100 $(2 * 2 = 4)$
12. **in**: 10|11|0000 ; **out**: 10|11|0110 $(2 * 3 = 6)$

13. **in**: 11|00|0000 ; **out**: 11|00|0000 $(3 * 0 = 0)$
14. **in**: 11|01|0000 ; **out**: 11|01|0011 $(3 * 1 = 3)$
15. **in**: 11|10|0000 ; **out**: 11|10|0110 $(3 * 2 = 6)$
16. **in**: 11|11|0000 ; **out**: 11|11|1001 $(3 * 3 = 9)$

5.1.9 Problem #5: Minimum finder

The problem can make use of an oracle on three registers, which is capable of comparing two of the registers and returning a value of 0 or 1 in the third register, indicating whether or not the value in the first register is smaller than the value held by the second.

In another way, one can simply encode an entire permutation function in the input as Massey has done for his PFMAX problem [40], in which he finds the *maximum* instead. For this study we have opted for Massey's encoding.

Problem setup for the 1-qubit minimum-finder

We are only interested in bijective functions in this case, so functions that are one-to-one, or in our case here, permutations.

In the 1-qubit case, there are only two possible permutation functions:

1. $f_0(x) = \{0, 1\} \rightarrow \{0, 1\}$
2. $f_1(x) = \{0, 1\} \rightarrow \{1, 0\}$

Using Massey's encoding we have 2 qubits, where the first qubit specifies a value for x and the last qubit specifies a value for $f(x)$. Using an even superposition we are able to encode an entire permutation function as a 2-qubit state. The idea is to compute on this state such that in the end when we measure the first two qubits, we get the index of x for which $f(x)$ returns the minimum element of its range; that is, 0. Each permutation function serves as a testcase; thus, we have two testcases for the 1-qubit instance of this problem:

1. **in**: $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$; **out**: 0
2. **in**: $\frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$; **out**: 1

Again, let us explain the encoding. For the first testcase this corresponds to the identity function f_0 from above, in which case $f(0) = 0$, which is the expected output. The first qubit represents x, while the second qubit represents $f(x)$. All the valid transformations are encoded in the input state, such that for the first testcase we can see only $f(0) \rightarrow 0$ and $f(1) \rightarrow 1$ appear, which is consistent with the identity transformations, while in the second testcase (for f_1 above) the only valid transformations are $f(0) \rightarrow 1$ and $f(1) \rightarrow 0$.

Problem setup for the 2-qubit minimum-finder

For the 2-qubit case we have 24 different permutations, since there are now 4 possible values in the domain and range of $f(x)$:

1. $f_0(x) = \{0,1,2,3\} \rightarrow \{0,1,2,3\}$
2. $f_1(x) = \{0,1,2,3\} \rightarrow \{0,1,3,2\}$
3. $f_2(x) = \{0,1,2,3\} \rightarrow \{0,2,1,3\}$
4. $f_3(x) = \{0,1,2,3\} \rightarrow \{0,2,3,1\}$
5. $f_4(x) = \{0,1,2,3\} \rightarrow \{0,3,1,2\}$
6. $f_5(x) = \{0,1,2,3\} \rightarrow \{0,3,2,1\}$
7. $f_6(x) = \{0,1,2,3\} \rightarrow \{1,0,2,3\}$
8. $f_7(x) = \{0,1,2,3\} \rightarrow \{1,0,3,2\}$
9. $f_8(x) = \{0,1,2,3\} \rightarrow \{1,2,0,3\}$
10. $f_9(x) = \{0,1,2,3\} \rightarrow \{1,2,3,0\}$
11. $f_{10}(x) = \{0,1,2,3\} \rightarrow \{1,3,0,2\}$
12. $f_{11}(x) = \{0,1,2,3\} \rightarrow \{1,3,2,0\}$
13. $f_{12}(x) = \{0,1,2,3\} \rightarrow \{2,0,1,3\}$
14. $f_{13}(x) = \{0,1,2,3\} \rightarrow \{2,0,3,1\}$
15. $f_{14}(x) = \{0,1,2,3\} \rightarrow \{2,1,0,3\}$
16. $f_{15}(x) = \{0,1,2,3\} \rightarrow \{2,1,3,0\}$
17. $f_{16}(x) = \{0,1,2,3\} \rightarrow \{2,3,0,1\}$
18. $f_{17}(x) = \{0,1,2,3\} \rightarrow \{2,3,1,0\}$
19. $f_{18}(x) = \{0,1,2,3\} \rightarrow \{3,0,1,2\}$
20. $f_{19}(x) = \{0,1,2,3\} \rightarrow \{3,0,2,1\}$
21. $f_{20}(x) = \{0,1,2,3\} \rightarrow \{3,1,0,2\}$
22. $f_{21}(x) = \{0,1,2,3\} \rightarrow \{3,1,2,0\}$
23. $f_{22}(x) = \{0,1,2,3\} \rightarrow \{3,2,0,1\}$
24. $f_{23}(x) = \{0,1,2,3\} \rightarrow \{3,2,1,0\}$

This gives us 24 testcases:

1. **in**: $\frac{1}{2}(|0000\rangle + |0101\rangle + |1010\rangle + |1111\rangle)$; **out**: 0
2. **in**: $\frac{1}{2}(|0000\rangle + |0101\rangle + |1011\rangle + |1110\rangle)$; **out**: 0
3. **in**: $\frac{1}{2}(|0000\rangle + |0110\rangle + |1001\rangle + |1111\rangle)$; **out**: 0
4. **in**: $\frac{1}{2}(|0000\rangle + |0110\rangle + |1011\rangle + |1101\rangle)$; **out**: 0
5. **in**: $\frac{1}{2}(|0000\rangle + |0111\rangle + |1001\rangle + |1110\rangle)$; **out**: 0
6. **in**: $\frac{1}{2}(|0000\rangle + |0111\rangle + |1010\rangle + |1101\rangle)$; **out**: 0
7. **in**: $\frac{1}{2}(|0001\rangle + |0100\rangle + |1010\rangle + |1111\rangle)$; **out**: 1
8. **in**: $\frac{1}{2}(|0001\rangle + |0100\rangle + |1011\rangle + |1110\rangle)$; **out**: 1
9. **in**: $\frac{1}{2}(|0001\rangle + |0110\rangle + |1000\rangle + |1111\rangle)$; **out**: 2
10. **in**: $\frac{1}{2}(|0001\rangle + |0110\rangle + |1011\rangle + |1100\rangle)$; **out**: 3
11. **in**: $\frac{1}{2}(|0001\rangle + |0111\rangle + |1000\rangle + |1110\rangle)$; **out**: 2
12. **in**: $\frac{1}{2}(|0001\rangle + |0111\rangle + |1010\rangle + |1100\rangle)$; **out**: 3
13. **in**: $\frac{1}{2}(|0010\rangle + |0100\rangle + |1001\rangle + |1111\rangle)$; **out**: 1
14. **in**: $\frac{1}{2}(|0010\rangle + |0100\rangle + |1011\rangle + |1101\rangle)$; **out**: 1
15. **in**: $\frac{1}{2}(|0010\rangle + |0101\rangle + |1000\rangle + |1111\rangle)$; **out**: 2
16. **in**: $\frac{1}{2}(|0010\rangle + |0101\rangle + |1011\rangle + |1100\rangle)$; **out**: 3
17. **in**: $\frac{1}{2}(|0010\rangle + |0111\rangle + |1000\rangle + |1101\rangle)$; **out**: 2
18. **in**: $\frac{1}{2}(|0010\rangle + |0111\rangle + |1001\rangle + |1100\rangle)$; **out**: 3
19. **in**: $\frac{1}{2}(|0011\rangle + |0100\rangle + |1001\rangle + |1110\rangle)$; **out**: 1
20. **in**: $\frac{1}{2}(|0011\rangle + |0100\rangle + |1010\rangle + |1101\rangle)$; **out**: 1
21. **in**: $\frac{1}{2}(|0011\rangle + |0101\rangle + |1000\rangle + |1110\rangle)$; **out**: 2
22. **in**: $\frac{1}{2}(|0011\rangle + |0101\rangle + |1010\rangle + |1100\rangle)$; **out**: 3
23. **in**: $\frac{1}{2}(|0011\rangle + |0110\rangle + |1000\rangle + |1101\rangle)$; **out**: 2
24. **in**: $\frac{1}{2}(|0011\rangle + |0110\rangle + |1001\rangle + |1100\rangle)$; **out**: 3

5.1.10 Problem #6: 2-element (ascending-order) sorter

Finally we have the 2-element quantum sorter problem, in which we are given two registers containing (not necessarily distinct) integers in a range bound by the size of the problem and the idea is to re-order these registers if and only if the first is larger than the second, such that the final ordering is non-decreasing. This problem, like the minimum-finder problem could also be posed and encoded in several ways. We experimented with two such ways.

Sorter as an oracle problem

We assumed an oracle existed which upon input consisting of 2 registers would place the value 1 in an output register if and only if the first register contained a value larger than the second. Then we let a GP find a solution in which an oracle could be consulted. A really simple (though, classical in its logic) algorithm could be created given such an oracle. First an oracle is called for the two registers. If the oracle returns 1 then we swap all corresponding qubits of the two registers. To encourage the creation of similar programs, which could easily be generalized to size-independent algorithms, we included the controlled-SWAP gate (CSWAP) into our function set, for the first time.

Oracle for 2-element sorter for arbitrary element size An oracle for the 2-element sorter problem would have to of course be reversible. Its underlying (non-reversible) function would need to do the mappings:

$$f(x, y) \mapsto 0, \text{if } x \leq y$$
$$f(x, y) \mapsto 1, \text{if } x > y$$

An oracle for the 2-element sorter problem might thus operate as follows:

$$O|x\rangle|y\rangle|0\rangle \mapsto |x\rangle|y\rangle|f(x, y) \oplus 0\rangle$$

Basically this oracle flips the 0^{th} qubit iff the value x is greater than the value y. A program calling this oracle can read the 0^{th} qubit to determine whether the two registers x and y are out of order.

Sorter as a state-encoding

A non-oracle version would simply encode the list of numbers (in this case the pair of numbers) into a qubit register, which is then tensored with some work qubits, perform computations onto that composite state and then finally read out the final state as the (hopefully) ordered list of numbers.

Problem setup for the 1-qubit 2-element sorter

In order to create a reversible solution we require an extra work bit, so that we might recreate the original state from the output state. As usual, we suppose this work bit starts out as 0. The 1-qubit case is not overly difficult, as there are only 4 testcases:

1. **in**: 000 ; **out**: 000
2. **in**: 010 ; **out**: 010
3. **in**: 100 ; **out**: 011
4. **in**: 110 ; **out**: 110

For the 1-qubit 2-element sorter we used the oracle encoding. An oracle in this case was easily implemented as a reversible function. For the 2-qubit problem an oracle cannot easily be implemented using our function set, which essentially makes it a *real* oracle, as we implement it manually simply by the actions $U_f|x\rangle|y\rangle \mapsto |x\rangle|f(x) \oplus y\rangle$ and do not worry about how its internals would work in practice.

Problem setup for the 2-qubit 2-element sorter

The 2-qubit problem requires a total of 5 qubits: 2 qubits each for the registers and 1 work qubit, whose final value is not measured.

1. **in**: 00000 ; **out**: 0000*
2. **in**: 00010 ; **out**: 0001*
3. **in**: 00100 ; **out**: 0010*
4. **in**: 00110 ; **out**: 0011*
5. **in**: 01000 ; **out**: 0001*
6. **in**: 01010 ; **out**: 0101*
7. **in**: 01100 ; **out**: 0110*
8. **in**: 01110 ; **out**: 0111*
9. **in**: 10000 ; **out**: 0010*
10. **in**: 10010 ; **out**: 0110*
11. **in**: 10100 ; **out**: 1010*
12. **in**: 10110 ; **out**: 1011*
13. **in**: 11000 ; **out**: 0011*

14. **in**: 11010 ; **out**: 0111*

15. **in**: 11100 ; **out**: 1011*

16. **in**: 11110 ; **out**: 1111*

5.1.11 Summary

We have introduced six different problems which were used to test our methodology and have thoroughly described how we have set up each experiment. We are now ready to discuss the results of the tests.

5.2 Results and discussion

Some of our problems were easily solved by the GPs introduced in the previous chapter, while others were fairly difficult. For the more difficult problems we were not always able to find exact (deterministic) solutions, but were often able to still find probabilistic solutions. This section discusses each of the experiments, and shows the more interesting results obtained for each problem.

5.2.1 Problem #1: Deutsch-Jozsa

Expectations

The Deutsch-Jozsa problem has a very *quantum* solution as given by the Deutsch-Jozsa algorithm and we expected the evolver to come up with something very similar to the famous algorithm. Since we are only dealing with at most 3 qubits for this problem we did not expect to see a particular GP perform better than any other. We hoped each GP would find a solution.

Deutsch-Jozsa 1-qubit problem

A result was first found when the controlled- phase flip (CZ) gate was added to the function set. The program is shown in Program 3 in `smallqc` syntax and in Figure 5.1 as a quantum circuit.[8] Its run details are also shown in Table 5.2.

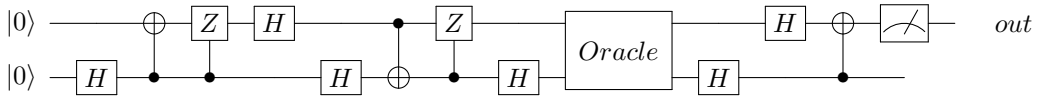


Figure 5.1: Solution #1 for 1-qubit Deutsch-Jozsa

[8] A *CNOT* gate is denoted by the bullet connected vertically to the crossed-circle. The bullet is the control qubit, while the circle is the target qubit. The bullet as a control qubit generalizes to other controlled gates as well.

```
# id: sol[387:23349]
# fitness: -0.000000
# solution size: 12
begin_init
end_init
begin_code
   op_H( NULL, 0 )
   op_CNOT( NULL, 0, 1 )
   op_CZ( NULL, 0, 1 )
   op_H( NULL, 1 )
   op_H( NULL, 0 )
   op_CNOT( NULL, 1, 0 )
   op_CZ( NULL, 0, 1 )
   op_H( NULL, 0 )
   oracle( NULL )
   op_H( NULL, 0 )
   op_H( NULL, 1 )
   op_CNOT( NULL, 0, 1 )
end_code
```

Program 3: Solution #1 for 1-qubit Deutsch-Jozsa

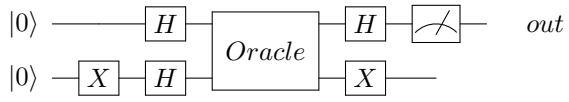


Figure 5.2: Solution #2 for 1-qubit Deutsch-Jozsa

Significance of result

This result was pleasantly surprising for two reasons. First, the system is entangled at multiple points. Second, there are clear patterns in the sequence of functions used. For example, there are two (2) occurrences of $H, CNOT, CZ$, two (2) occurrences of H, H and three (3) occurrences of $H, CNOT$.

This result, however, is not the most efficient. In later runs the efficiency component of the fitness function[9] was activated and HQP running with EDA-QP-I found a shorter solution which closely resembles the Deutsch-Jozsa algorithm. Details for this solution can be found in Table 5.3 and the actual program is shown in Program 4 and Figure 5.2.

The last X gate on qubit 0 in Program 4 can be ignored, as we no longer use qubit 0 past the point of this gate. The first X however serves to change the input $|0\rangle|0\rangle$ to $|0\rangle|1\rangle$, which, as mentioned previously, is the input for the normal Deutsch-Jozsa algorithm for the problem of size 1. From there on (and given the removal of the last X gate) the program is an exact implementation of the Deutsch-Jozsa algorithm.

[9] Recall from Chapter 4 that the fitness function uses four different components: 1) average error (avgerr), 2) misses, 3) eff (efficiency) and 4) ent (entanglement).

Parameter	*Setting*
GP type	NQP
Solution set size	150
Max iterations	1000
Max sol. length	12
Selection %	0.20
Mut. rate top	0.05
Mut. rate rest	0.10
Fitness	avgerr
Function set	ORACLE, CNOT, H, CZ

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	387
Solution size	12

(b) Solution details

Table 5.2: Details for 1-qubit Deutsch-Jozsa solution #1

Parameter	*Setting*
GP type	HQP
Sub EDA type	EDA-QP-I
Solution set size	1500
Max iterations	5000
Max sol. length	15
Selection %	0.20
Learning rate	0.05
Perturb. freq.	50
Perturb. set size	750
Learn len. model	No
Fitness	avgerr, misses, eff
Function set	ORACLE, CNOT, X, H

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	27
Solution size	6

(b) Solution details

Table 5.3: Details for 1-qubit Deutsch-Jozsa solution #2

Deutsch-Jozsa 2-qubit problem

For the 2-qubit instance of the Deutsch-Jozsa problem we have an input register of 3 qubits, where the leftmost 2 qubits make up the input query register for an oracle call and the rightmost qubit makes up the answer register for the oracle call. In the end we measure only the 2 qubits on the left. If these result in $|00\rangle$ with 100% probability, then the function should be constant; else the function should be balanced. Again, we compare the result with the expected result.

We found multiple solutions for the 2-qubit problem and show two of them here, which were both found by HQP with sub-EDA option EDA-QP-I. The details of each run are in Tables 5.4 and 5.5 and the solutions themselves are shown in Programs 5 and 6 and as quantum circuits in Figures 5.3 and 5.4.

```
# id: sol[9:6541]
# fitness: 0.000600
# solution size: 6
begin_init
end_init
begin_code
op_X( NULL, 0 )
op_H( NULL, 0 )
op_H( NULL, 1 )
oracle( NULL )
op_X( NULL, 0 )
op_H( NULL, 1 )
end_code
```

Program 4: Solution #2 for 1-qubit Deutsch-Jozsa

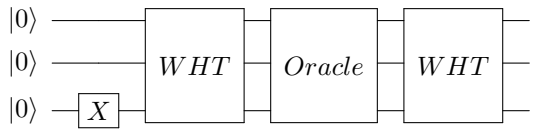


Figure 5.3: Solution #1 for 2-qubit Deutsch-Jozsa

Recall that the Deutsch-Jozsa algorithm begins by initializing an input query register as $|00\rangle$ and an input answer register of one qubit as $|1\rangle$. Then it applies a Walsh-Hadamard transform to the composite of these two kets. These steps have the effect of creating the following superposition before a call is made to the oracle:

$$\begin{aligned}|\psi\rangle &= WHT|001\rangle \\ &= H|0\rangle \otimes H|0\rangle \otimes H|1\rangle \\ &= \tfrac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\tfrac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\tfrac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \\ &= \tfrac{1}{\sqrt{8}}(|000\rangle - |001\rangle + |010\rangle - |011\rangle + |100\rangle - |101\rangle + |110\rangle - |111\rangle) \end{aligned} \tag{5.6}$$

It is very easy to see that the first two actions in solution #2 (Figure 5.4) take the

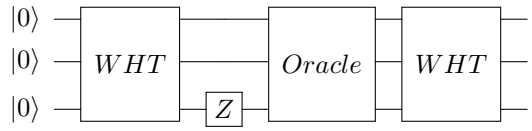


Figure 5.4: Solution #2 for 2-qubit Deutsch-Jozsa

```
# id: sol[79:48397]
# fitness: 0.000400
# solution size: 4
begin_init
end_init
begin_code
op_X( NULL, 0 )
op_WHT( NULL )
oracle( NULL )
op_WHT( NULL )
end_code
```

Program 5: Solution #1 for 2-qubit Deutsch-Jozsa

```
# id: sol[64:39577]
# fitness: 0.000400
# solution size: 4
begin_init
end_init
begin_code
op_WHT( NULL )
op_Z( NULL, 0 )
oracle( NULL )
op_WHT( NULL )
end_code
```

Program 6: Solution #2 for 2-qubit Deutsch-Jozsa

Parameter	*Setting*
GP type	HQP
Sub EDA type	EDA-QP-I
Solution set size	1500
Max iterations	5000
Max sol. length	15
Selection %	0.20
Learning rate	0.05
Perturb. freq.	50
Perturb. set size	750
Learn len. model	No
Fitness	avgerr, misses, eff
Function set	ORACLE, WHT, CNOT, CCNOT, X, H, Z, S, W, T

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	83
Solution size	4

(b) Solution details

Table 5.4: Details for 2-qubit Deutsch-Jozsa solution #1

state $|000\rangle$ to the same superposition. First WHT creates an even superposition:

$$\begin{aligned} |\phi_1\rangle &= WHT|000\rangle \\ &= \tfrac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle) \end{aligned} \tag{5.7}$$

And then the Z gate on qubit 0 flips the sign of the probability amplitude wherever qubit 0 is 1:

$$\begin{aligned} Z_0|\phi_1\rangle &= I \otimes I \otimes Z|\phi_1\rangle \\ &= \tfrac{1}{\sqrt{8}}(|000\rangle - |001\rangle + |010\rangle - |011\rangle + |100\rangle - |101\rangle + |110\rangle - |111\rangle) \end{aligned} \tag{5.8}$$

Since Equation (5.8) is the same as Equation (5.6), it is clear that our second solution to the Deutsch-Jozsa 2-qubit problem is also just an implementation of the Deutsch-Jozsa algorithm.

One reason these solutions are so compact is obviously the use of WHT as an operator in our function set. Given that the Hadamard gate and the more general Walsh-Hadamard[10] transform are essential for quantum phenomena to manifest[11] we thought it would be a good idea to include the Walsh-Hadamard transform in our function set and the GP system seems to have made good use of it.

[10] Recall that the Walsh-Hadamard transform applies Hadamard gates to multiple qubits. In our case we define the WHT as a gate that applies Hadamard to *all* qubits in the system.

[11] That is, in the absence of the general rotation gates, which we have not always included in our runs.

Parameter	*Setting*
GP type	HQP
Sub EDA type	EDA-QP-I
Solution set size	1500
Max iterations	5000
Max sol. length	15
Selection %	0.20
Learning rate	0.05
Perturb. freq.	50
Perturb. set size	750
Learn len. model	No
Fitness	avgerr, misses, eff
Function set	ORACLE, WHT, CNOT, CCNOT, X, H, Z, S, W, T

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	65
Solution size	4

(b) Solution details

Table 5.5: Details for 2-qubit Deutsch-Jozsa solution #2

Conclusions

As we had hypothesized, every GP variant was able to handle this problem and we were happy to see that the pure learners were able to keep up with NQP and HQP. The inclusion of the WHT gate in our function set definitely helped keep solutions short and lead the evolver to a quicker find.

5.2.2 Problem #2: Imperfect copy machine

Expectations

The imperfect copier was one of the first problems we thought might be easier to handle by the learners, as we expected there to be a lot of controlled operations anchoring on the ancilla qubits and hoped that good sequences might be discerned by ngram-QP or EDA-QP.

Imperfect copier 1-qubit problem

The 1-qubit copier is a fairly straightforward problem for the program evolver, so it is not too surprising that it quickly came up with a solution in multiple runs and using all the different GP variants. Here we discuss three of them.

The first solution, shown in Program 7 and as a quantum circuit in Figure 5.5[12] was found by NQP. The second was found by EDA-QP-I and is shown in Program 8, with its corresponding circuit Figure 5.6. Finally, the third solution shown in Program 9 and

[12] A SWAP gate is represented by the two connected × symbols.

Figure 5.7 was found by EDA-QP-II. Details for all the runs are shown in Tables 5.6, 5.7 and 5.8.

Parameter	*Setting*
GP type	NQP
Solution set size	1500
Max iterations	10000
Max sol. length	10
Mut. rate top	0.25
Mut. rate rest	1.0
Selection %	0.10
Fitness	avgerr, misses
Function set	H, CNOT, X, CZ, CCNOT, SWAP

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	0
Solution size	3

(b) Solution details

Table 5.6: Details for 1-qubit imperfect copier solution #1

Parameter	*Setting*
GP type	EDA-QP-I
Solution set size	1500
Max iterations	10000
Max sol. length	10
Selection %	0.10
Learning rate	0.05
Perturb. freq.	50
Perturb. set size	750
Learn len. model	Yes
Fitness	avgerr, misses, eff, ent
Function set	H, CNOT, X, CZ, CCNOT

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	14
Solution size	5

(b) Solution details

Table 5.7: Details for 1-qubit imperfect copier solution #2

While all solutions are exact, the second one is interesting because it makes use of a superposition and controlled-Z gate to compute its result. The first and third solutions are more efficient as they only use 3 gates, however, they are strictly reversible logic circuits, while the second actually exhibits quantum properties. Without the sequence of H(0), CZ(0,2), H(0) the last two testcases would fail. The first H(0) puts the 0^{th} qubit in a superposition, while the second H(0) contracts the state to a computational basis state; were it not for the CZ(0,2) gate in the middle, the two Hadamards would simply cancel each other out. As it is, though, the CZ(0,2) sandwiched between two Hadamard gates has the effect of flipping the 0^{th} qubit.

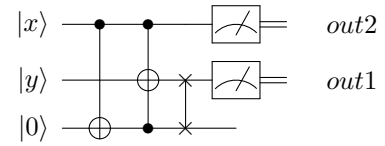


Figure 5.5: Solution #1 for 1-qubit imperfect copier

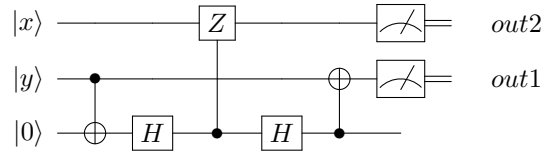


Figure 5.6: Solution #2 for 1-qubit imperfect copier

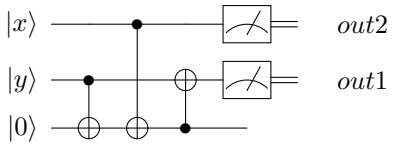


Figure 5.7: Solution #3 for 1-qubit imperfect copier

```
# id: sol[0:14]
# fitness: 0.000000
# solution size: 3
begin_init
end_init
begin_code
op_CNOT( NULL, 2, 0 )
op_CCNOT( NULL, 2, 0, 1 )
op_SWAP( NULL, 1, 0 )
end_code
```

Program 7: Solution #1 for 1-qubit imperfect copier

```
# id: sol[14:21273]
# fitness: 0.000500
# solution size: 5
begin_init
end_init
begin_code
op_CNOT( NULL, 1, 0 )
op_H( NULL, 0 )
op_CZ( NULL, 0, 2 )
op_H( NULL, 0 )
op_CNOT( NULL, 0, 1 )
end_code
```

Program 8: Solution #2 for 1-qubit imperfect copier

Parameter	*Setting*
GP type	EDA-QP-II
Solution set size	1500
Max iterations	2000
Max sol. length	10
Selection %	0.20
Learning rate	0.05
Perturb. freq.	50
Perturb. set size	750
Learn len. model	No
Fitness	avgerr, misses, eff
Function set	CNOT, X, H, CCNOT, CZ, Z

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	98
Solution size	3

(b) Solution details

Table 5.8: Details for 1-qubit imperfect copier solution #3

```
# id: sol[98:148487]
# fitness: 0.000300
# solution size: 3
begin_init
end_init
begin_code
op_CNOT( NULL, 1, 0 )
op_CNOT( NULL, 2, 0 )
op_CNOT( NULL, 0, 1 )
end_code
```

Program 9: Solution #3 for 1-qubit imperfect copier

Imperfect copier 2-qubit problem

Parameter	*Setting*
GP type	NGRAM-QP-II
Solution set size	2500
Max iterations	2500
Max sol. length	25
Selection %	0.20
Learning rate	0.01
1/2 -gram	0.05/0.95
Perturb. freq.	50
Perturb. set size	1250
Learn len. model	No
Fitness	misses
Threshold	0.50
Function set	CNOT, CCNOT, X, CSWAP, SWAP, H

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	50
Solution size	11
Optimized size	4

(b) Solution details

Table 5.9: Details for 2-qubit imperfect copier solution #1

The 2-qubit problem was definitely more difficult, but we managed to find a number of solutions. Two solutions are shown here in their original forms and in their hand-optimized forms. The first exact solution was found by NGRAM-QP-II and consisted of 11 gates. The original is shown in Program 10.

Parameter	*Setting*
GP type	NGRAM-QP-II
Solution set size	2500
Max iterations	2500
Max sol. length	25
Selection %	0.20
Learning rate	0.01
1/2 -gram	0.05/0.95
Perturb. freq.	50
Perturb. set size	1250
Learn len. model	No
Fitness	misses
Threshold	0.75
Function set	CNOT, CCNOT, X, CSWAP, SWAP, H

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	2402
Solution size	14
Optimized size	4

(b) Solution details

Table 5.10: Details for 2-qubit imperfect copier solution #2

After some inspection it is easy to see that several of the gates can be removed:

- CCNOT(1,5,3) has no effect, because work qubit 1 is always going to start out as 0
- X(3) becomes lost, since right after this, qubits 1 and 3 are swapped and we never again see the original 3
- CNOT(3,1) (the 5^{th} gate) never gets activated because prior to this, qubit 3 is swapped with qubit 1, which means that the control qubit for this gate is always going to be 0 (since qubit 1 starts as 0)
- H(1) is useless because it puts qubit 1 in a superposition, but qubit 1 is not used after this gate and since it is a work qubit it is not measured either
- CCNOT(2,1,5) is never activated, because the two controls will never both be 1; qubit 2 cannot be 1 because the very early swap gate swaps qubits 0 and 2 and qubit 0, a work qubit, is initially 0, which is now the value at qubit 2. No other actions on qubit 2 change it before the CCNOT gate, so it never has a chance to acquire a value other than 0.

```
# id: sol[50:125856]
# fitness: 0.000000
# solution size: 11
begin_init
end_init
begin_code
op_CCNOT( NULL, 1, 5, 3 )
op_SWAP( NULL, 0, 2 )
op_X( NULL, 3 )
op_SWAP( NULL, 1, 3 )
op_CNOT( NULL, 3, 1 )
op_CNOT( NULL, 5, 3 )
op_H( NULL, 1 )
op_CCNOT( NULL, 2, 1, 5 )
op_CNOT( NULL, 4, 2 )
op_CNOT( NULL, 3, 1 )
op_CCNOT( NULL, 0, 5, 1 )
end_code
```

Program 10: Solution #1 for 2-qubit imperfect copier

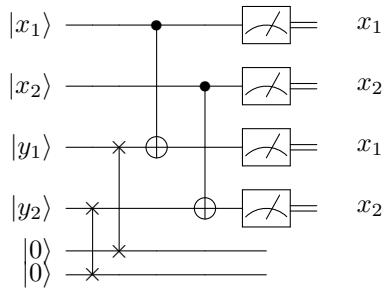


Figure 5.8: Optimized solution #1 for 2-qubit imperfect copier

- CNOT(3,1) and CCNOT(0,5,1) at the very end do not matter, because they both target qubit 1, which is a work qubit that is not used after these two gates, and whose final value is not measured

After removing all the gates which have no effect on the solution we are left with a very short solution as shown in Program 11 and as a quantum circuit in Figure 5.8. This solution is very intuitive, as it can be generalized to an actual algorithm. It basically says to throw away the second register (with the first two SWAP operations), which is made up of qubits 1 and 2 and then just copy the second register over the first (with the CNOT operations).

The second exact solution was also found by NGRAM-QP-II and consisted of 14 gates. The original is shown in Program 12. Again we were able to optimize this solution and reduce it to 4 gates, similar to the one before, by noting the following:

```
# id: EDITED sol[50:125856]
# fitness: 0.000000
# solution size: 4
begin_init
end_init
begin_code
op_SWAP( NULL, 0, 2 )
op_SWAP( NULL, 1, 3 )
op_CNOT( NULL, 5, 3 )
op_CNOT( NULL, 4, 2 )
end_code
```

Program 11: Optimized solution #1 for 2-qubit imperfect copier

- SWAP(0,1) (two near the beginning) are both useless because both qubits 0 and 1 are work qubits and no operation changes either 0 or 1 between these two SWAP gates
- CNOT(5,3) CNOT(5,3) (at the beginning) cancel each other out
- CSWAP(0,2,4) has no effect because the control qubit is 0
- SWAP(0,1) (last one) is useless because we use neither work bit after this point
- CNOT(5,3) CNOT(5,3) (at the end) cancel each other out
- SWAP(3,1) (the last two) cancel each other out, because nothing is done with either qubit 3 or qubit 1 between these two

Having edited the original solution we obtain Program 13 and the quantum circuit in Figure 5.9. This solution is pretty much the same as the one before, except here we can think of it as throwing away the first qubit of the destination register (qubit 3), copying the first qubit of the source (qubit 5) over it, making a copy of the second qubit (qubit 4) of the source register in one of the work bits (which we know started off cleared), and finally swapping this work bit with the second qubit of the destination register (qubit 2).

Conclusions

Some of these solutions are clearly longer than they have to be, which makes sense since we did not always use the efficiency component of the fitness function. A reason for this was that we wanted to allow the evolver more room to encourage interesting patterns.

The 1-qubit case was solved by all GPs equally well. For the 2-qubit case, however, we did have more success in general with ngram-QP than with the others; however, given the random aspect of an evolutionary run we do not have enough data to support an actual claim that ngram-QP is better for this problem.

```
# id: sol[2402:6005607]
# fitness: 0.000000
# solution size: 14
begin_init
end_init
begin_code
op_SWAP( NULL, 0, 1 )
op_CNOT( NULL, 5, 3 )
op_SWAP( NULL, 0, 1 )
op_CNOT( NULL, 5, 3 )
op_SWAP( NULL, 3, 1 )
op_CNOT( NULL, 5, 3 )
op_SWAP( NULL, 3, 1 )
op_CSWAP( NULL, 0, 2, 4 )
op_CNOT( NULL, 4, 0 )
op_SWAP( NULL, 0, 2 )
op_SWAP( NULL, 3, 1 )
op_CNOT( NULL, 5, 3 )
op_SWAP( NULL, 0, 1 )
op_CNOT( NULL, 5, 3 )
end_code
```

Program 12: Solution #2 for 2-qubit imperfect copier

```
# id: EDITED sol[2402:6005607]
# fitness: 0.000000
# solution size: 4
begin_init
op_SWAP( NULL, 3, 1 )
op_CNOT( NULL, 5, 3 )
op_CNOT( NULL, 4, 0 )
op_SWAP( NULL, 0, 2 )
end_init
begin_code
end_code
```

Program 13: Optimized solution #2 for 2-qubit imperfect copier

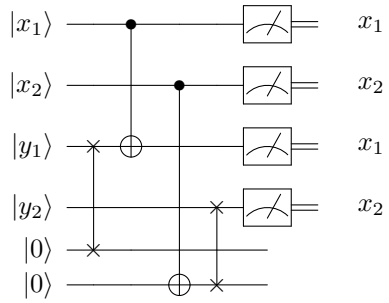


Figure 5.9: Optimized solution #2 for 2-qubit imperfect copier

We also experimented for the first time with the entanglement promotion component of the fitness function, for the 1-qubit case. This led to the interesting second solution (Program 8 and Figure 5.6), which as a result actually made use of a superposition to compute the final answer.

Parameter	*Setting*
GP type	NGRAM-QP-I
Solution set size	50
Max iterations	1000
Max sol. length	10
Selection %	0.20
Learning rate	0.075
1/2/3 -gram	0.02/0.13/0.85
Perturb. freq.	50
Perturb. set size	25
Learn len. model	No
Fitness	misses, avgerr
Function set	CNOT, CCNOT, CZ, H, CH, Z, X

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	2
Solution size	2

(b) Solution details

Table 5.11: Details for 1-qubit half-adder

Parameter	*Setting*
GP type	NGRAM-QP-II
Solution set size	500
Max iterations	2000
Max sol. length	10
Selection %	0.20
Learning rate	0.075
1/2 -gram	0.35/0.65
Perturb. freq.	50
Perturb. set size	250
Learn len. model	No
Fitness	avgerr, eff
Function set	CNOT, CCNOT, CZ, Z, X

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	55
Solution size	4

(b) Solution details

Table 5.12: Details for 1-qubit full-adder

5.2.3 Problem #3: Adder

To remind the reader, we have two types of adders: the half-adder and the full-adder. The former takes two (binary) numbers and computes the sum as the XOR of the two

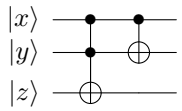


Figure 5.10: Solution for 1-qubit half-adder

numbers, plus a possible carry-bit. The latter may accept a carry-in bit in addition to the two numbers and it computes the sum of all three numbers.

Expectations

Given efficient networks for the half-adder and full-adder problem made of CNOT and CCNOT gates, as shown in multiple works [77, 22], we thought the sequences would easily be picked up by ngram-QP and that EDA-QP would also be able to learn the circuits. We expected the EDA-based GPs to have an advantage over NQP here, because NQP relies on mutation only and might waste more time on functions that contributed nothing to the fitness, while the learners should be able to pick out CNOT and CCNOT as the better gates in the function set.

1-qubit half-adder

The 1-qubit half-adder has a very simple solution which was easily found by all the GPs. It is shown in Program 14, with its circuit in Figure 5.10 and the details of this particular run in Table 5.11.

1-qubit full-adder

The 1-qubit full-adder is slightly more complicated, but again the GPs were able to find multiple solutions fairly quickly. We show here one minimal solution which was found by including the efficiency component of the fitness function. The solution can be seen in Program 15, with its circuit in Figure 5.11 and all details of the run in Table 5.12.

```
# id: sol[2:100]
# fitness: 0.000000
# solution size: 2
begin_init
end_init
begin_code
op_CCNOT( NULL, 1, 2, 0 )
op_CNOT( NULL, 2, 1 )
end_code
```

Program 14: Solution for 1-qubit half-adder

Both the half-adder and the full-adder are optimal circuits as given in literature [22].

```
# id: sol[55:27616]
# fitness: 0.000400
# solution size: 4
begin_init
end_init
begin_code
op_CCNOT( NULL, 1, 2, 0 )
op_CNOT( NULL, 2, 1 )
op_CCNOT( NULL, 1, 3, 0 )
op_CNOT( NULL, 3, 1 )
end_code
```

Program 15: Solution for 1-qubit full-adder

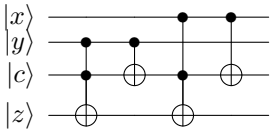


Figure 5.11: Solution for 1-qubit full-adder

Experiments with 2-qubit adders

While the 1-qubit adders are simple for all GPs, things get a lot more difficult when we increase the operand size to 2 qubits. Having had success with the 1-qubit adders, we attempted to implement a 2-qubit adder, expecting the patterns of CNOT and CCNOT gates to be easily picked up by the ngram-QP; however, we failed to get an exact solution. For a 2-qubit full-adder we require 7 qubits in order to make the adder reversible, which means there are at least 64 testcases. The processing was thus considerably slow (for max solution size of 100 and population size of 2500 it ran for several hours) and we only ran a few trials with population sizes up to 2500, which appear to not have been large enough sizes. The 2-qubit *half*-adder, however, requires only 32 testcases and 5 qubits (with an encoding similar to the one we used for the 1-qubit case). After a few runs with EDA-QP, NQP and ngram-QP we were not able to find a result and so we decided to introduce the solution from the 1-qubit half-adder as an operator in our function set to see if this might help things along.

We thus created the new operator:

$$PROC1(x, y, z) = CCNOT(z, y, x), CNOT(z, y) \tag{5.9}$$

Basically `PROC1` is an operator that takes three qubits as input and applies a CCNOT gate to the first argument, with arguments two and three as controls, followed by a CNOT gate to the second argument, with control the third. In this way we turned the

Parameter	*Setting*
GP type	HQP
Sub EDA type	EDA-QP-I
Solution set size	2750
Max iterations	2000
Max sol. length	22
Selection %	0.20
Mut. rate top	0.1
Mut. rate rest	1.00
Learning rate	0.05
Perturb. freq.	50
Perturb. set size	1350
Learn len. model	No
Fitness	misses
Function set	PROC1, CNOT, CCNOT, H, T, S, W, X, Z, CZ, SWAP

(a) Run settings

Found solution	Yes
Solution type	Probabilistic
Correctness	50.00% (all testcases)
Found in iteration	54
Solution size	3 (5 with PROC1 uncompressed)

(b) Solution details

Table 5.13: Details for 2-qubit half-adder

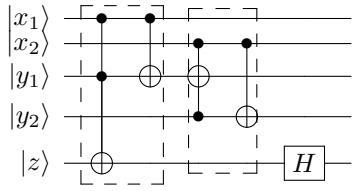


Figure 5.12: Probabilistic solution for 2-qubit half-adder

1-qubit half-adder into a procedure which we hoped would become a building block for the 2-qubit half-adder.

With this new operator, the GPs were able to find multiple *probabilistic* solutions, though still no exact solution. The best probabilistic solution (correctness of 50% for every testcase) was discovered by HQP. This is shown in Program 16 and Figure 5.12 and the details of the run are shown in 5.13. The dashed boxes in the circuit of Figure 5.12 group the operations of `PROC1`.

Conclusions

The 1-qubit problem was not as difficult as we might have expected and again all GPs were able to solve this. We were particularly happy to see that efficient solutions were found by using the efficiency component of the fitness function.

The more interesting problem was definitely the 2-qubit half-adder. Our expectations were not met, as neither pure learner was able to find a good solution to this problem. The best, though probabilistic, solution, was found by HQP, which would suggest that

```
# id: sol[54:61688]
# fitness: 0.000000
# solution size: 3
begin_init
end_init
begin_code
PROC1( NULL, 0, 2, 4 )
PROC1( NULL, 2, 1, 3 )
op_H( NULL, 0 )
end_code
```

Program 16: Probabilistic solution for 2-qubit half-adder

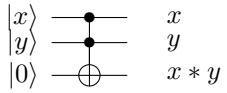


Figure 5.13: Solution #1 for 1-qubit multiplier (just a CCNOT gate)

HQP's capacity for exploration using the mutation functions which it shares with NQP gave it an advantage over the pure learners for this particular problem.

As for the Deutsch-Jozsa problem, where we introduced WHT as a building block, we saw improved performance when we gave the evolver a building block in the form of a 1-qubit half-adder (PROC1).

5.2.4 Problem #4: Multiplier

Expectations

As with the adder, we thought this particular problem would be better-suited for the learners, as we again expected a lot of CNOT and CCNOT gates in the solutions and hoped the learners might pick out some patterns.

1-qubit multiplier

The 1-qubit multiplier is a trivial problem to solve using a GP, as it is implemented by a single gate, the CCNOT gate, as seen in Figure 5.13.

2-qubit multiplier

NGRAM-QP-II was able to find a solution for the 2-qubit multiplier problem, albeit a long one. The solution is given in Program 17. This solution is long, but upon inspection it is clear that it can be drastically reduced:

- CCNOT(5,6,1) and CCNOT(6,5,1) cancel each other out
- CZ(3,2) and CZ(2,3) cancel each other out

Parameter	*Setting*
GP type	NGRAM-QP-II
Solution set size	2500
Max iterations	2500
Max sol. length	25
Selection %	0.20
Learning rate	0.01
1/2 -gram	0.05/0.95
Perturb. freq.	50
Perturb. set size	1250
Learn len. model	No
Fitness	misses
Threshold	0.40
Function set	CNOT, CCNOT, H, Z, S, T, CZ

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	216
Solution size	25
Optimized size	6

(b) Solution details

Table 5.14: Details for 2-qubit multiplier

- CCNOT(4,6,0) and CCNOT(6,4,0) cancel each other out
- CCNOT(5,7,2) and CCNOT(5,7,2) also cancel each other out, since in between no gate operates on qubits 5, 7 or 2
- CCNOT(6,4,0) and CCNOT(4,6,0) (in the middle) cancel each other out
- S(4), CZ(3,7), CZ(3,2), CZ(2,3), Z(3), CZ(3,0), CZ(3,7), T(2), T(5) and Z(6) have no effect, since all they do is (*potentially*, in the case of CZ) add a phase to the state. Amplitude interference cannot occur in this program, since we do not have any uses of a Hadamard gate and no superpositions.
- CNOT(3,4) is useless, because qubit 3 is never altered prior to this gate's being called and since qubit 3 is one of the answer qubits, the assumption is that it starts out as $|0\rangle$.

After making all these changes we are left with a much shorter deterministic solution as shown in Program 18 and illustrated as a quantum circuit in Figure 5.14.

Conclusions

We had originally hypothesized that this problem would be easier for the learners (ngram-QP and EDA-QP) than for HQP or NQP; however, given the difficulty we had with producing a 2-qubit half-adder in the previous experiment, a deterministic 2-qubit multiplier from NGRAM-QP-II came as a slight (pleasant) surprise. The solution was hand-optimized and reduced to less than a $\frac{1}{4}$ of its original size. This huge difference suggests that ngram-QP is especially susceptible to bloat. This might also have been caused by a

```
# id: sol[216:541355]
# fitness: 0.000000
# solution size: 25
begin_init
end_init
begin_code
op_CCNOT( NULL, 6, 4, 0 )
op_CCNOT( NULL, 6, 5, 1 )
op_S( NULL, 4 )
op_CZ( NULL, 3, 7 )
op_CCNOT( NULL, 5, 7, 2 )
op_CCNOT( NULL, 6, 4, 0 )
op_CCNOT( NULL, 5, 6, 1 )
op_CCNOT( NULL, 5, 7, 2 )
op_CCNOT( NULL, 6, 5, 1 )
op_CZ( NULL, 3, 2 )
op_CZ( NULL, 2, 3 )
op_Z( NULL, 3 )
op_CCNOT( NULL, 4, 6, 0 )
op_CZ( NULL, 3, 0 )
op_CZ( NULL, 3, 7 )
op_CCNOT( NULL, 4, 6, 0 )
op_CCNOT( NULL, 6, 4, 0 )
op_CCNOT( NULL, 4, 7, 1 )
op_CCNOT( NULL, 7, 5, 2 )
op_CNOT( NULL, 3, 4 )
op_T( NULL, 2 )
op_T( NULL, 5 )
op_CCNOT( NULL, 2, 0, 3 )
op_Z( NULL, 6 )
op_CNOT( NULL, 3, 2 )
end_code
```

Program 17: Solution for 2-qubit multiplier

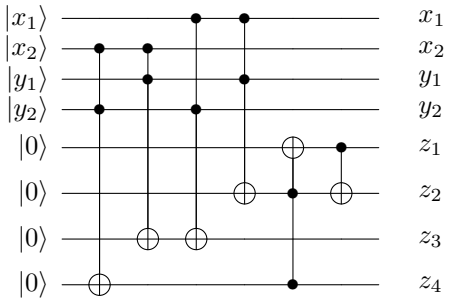


Figure 5.14: Optimized solution for 2-qubit multiplier

```
# id: EDITED sol[216:541355]
# fitness: 0.000000
# solution size: 6
begin_init
end_init
begin_code
op_CCNOT( NULL, 6, 4, 0 )
op_CCNOT( NULL, 6, 5, 1 )
op_CCNOT( NULL, 7, 4, 1 )
op_CCNOT( NULL, 7, 5, 2 )
op_CCNOT( NULL, 2, 0, 3 )
op_CNOT( NULL, 3, 2 )
end_code
```

Program 18: Optimized solution for 2-qubit multiplier

very low learning rate, which effectively slows down the learning process and allows for much greater variety in the solutions, which could lead to long sequences that are useless (but also to greater exploration). Looking at the original solution in Program 17 it is obvious that the model this solution was drawn from is not optimal as it produces long sequences that have no effect. As a result this solution was found on very precarious grounds, as a replacement of a single one of its many useless gates might easily disturb the sequence of 6 transformations which make up the real solution.

5.2.5 Problem #5: Minimum finder

The minimum-finder problem is interesting because it tries to determine a global property of a function. A procedure to find a minimum of a function might also be very useful as a building-block for more complex programs.

Expectations

This particular problem is the single one of our set where we were expecting a guarantee of visible quantum effects, given that the input for each testcase is a superposition of values representing a permutation function and the whole idea was to find some optimal way to detect a global property of each function, namely, the minimum. We did not know in advance what a solution might look like; however, we did expect multiple uses of Hadamard gates and controlled operations. Our hypothesis for this particular experiment was that HQP would be the best of the EDA-based variants, as it has a greater ability to explore the search space.

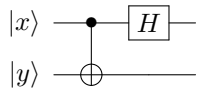


Figure 5.15: Solution for 1-qubit minimum-finder

1-qubit minimum-finder

A short deterministic solution was found for this problem as seen in Program 19 and Figure 5.15. It is interesting to note that the input states are two of the Bell states. Recall that the Bell states are entangled. The program essentially *undoes* the entanglement.

Parameter	*Setting*
GP type	NGRAM-QP-I
Solution set size	1500
Max iterations	2000
Max sol. length	2
Selection %	0.20
Learning rate	0.01
1/2/3 -gram	0.02/0.13/0.85
Perturb. freq.	50
Perturb. set size	750
Learn len. model	Yes
Fitness	avgerr, misses
Function set	CNOT, CZ, S, T, CS, H, SWAP, X

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	0
Solution size	2

(b) Solution details

Table 5.15: Details for 1-qubit minimum-finder

```
# id: sol[0:444]
# fitness: 0.000000
# solution size: 2
begin_init
end_init
begin_code
op_CNOT( NULL, 1, 0 )
op_H( NULL, 1 )
end_code
```

Program 19: Solution for 1-qubit minimum-finder

2-qubit minimum-finder

A probabilistic solution for this problem was found by EDA-QP-II as shown in Program 20 and its slightly optimized circuit in Figure 5.16. By hand we found that two of the

Parameter	*Setting*
GP type	EDA-QP-II
Solution set size	1500
Max iterations	2000
Max sol. length	25
Selection %	0.20
Learning rate	0.05
Perturb. freq.	50
Perturb. set size	1875
Learn len. model	Yes
Fitness	avgerr, misses
Threshold	0.52
Function set	CNOT, H, SWAP, X, CCNOT, Z

(a) Run settings

Found solution	Yes
Solution type	Probabilistic
Correctness	56.25% (all testcases)
Found in iteration	153
Solution size	10
Optimized size	7

(b) Solution details

Table 5.16: Details for 2-qubit minimum-finder

gates could be removed; namely, the CNOT(1, 3) gates, as they cancelled each other out, since no operator works on qubits 1 or 3 between these two. The last gate in the solution, CCNOT(2, 3, 1) turned out to also have no effect on the final result, since qubit 1 is not read out.

```
# id: sol[153:230778]
# fitness: 0.000000
# solution size: 10
begin_init
end_init
begin_code
op_CCNOT( NULL, 0, 1, 2 )
op_CNOT( NULL, 1, 3 )
op_CNOT( NULL, 1, 2 )
op_CNOT( NULL, 0, 2 )
op_CNOT( NULL, 1, 3 )
op_H( NULL, 1 )
op_CNOT( NULL, 1, 0 )
op_H( NULL, 1 )
op_CCNOT( NULL, 0, 1, 3 )
op_CCNOT( NULL, 2, 3, 1 )
end_code
```

Program 20: Solution for 2-qubit minimum-finder

While the solution is only probabilistic, it achieves a correctness of 56.25% for every testcase in the suite.

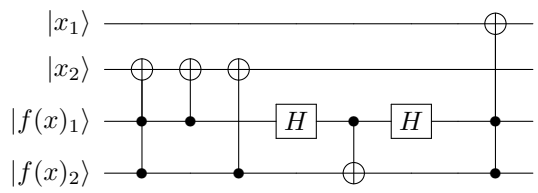


Figure 5.16: Optimized solution for 2-qubit minimum-finder

Conclusions

Our hypothesis for this experiment was proved wrong, as all solutions we found came from EDA-QP and not from HQP. We were not able to find a deterministic solution for the 2-qubit case, but did not spend a lot of time looking for one and believe it might in fact be possible to improve the performance with longer running times (more iterations) and larger population sizes.

What we thought was interesting was that the EDA-QP found a solution that only saw a 30% reduction in size when we optimized it by hand. This was one of the few runs when we actually activated the length learner as well and it appears that this model played a big role in focusing the search. By iteration 153 in which our solution was found, the length model had most of its probability mass in the range $[5, 12]$. EDA-QP also exhibits high-location dependence for its functions and inputs, since as was described in Chapter 4, the models use node index in a solution as the independent variable. This prompted us to try an experiment with NGRAM-QP-I (where inputs are modelled as location-dependent) and an activated length model to see if we could do better. We did in fact not do better, but we were able to find a probabilistic solution of 11 gates. Further experiments were not attempted; however, we have mentioned previously that the length model tends to keep sizes to a minimum and we think it might be worthwhile to use it together with ngram-QP to see whether it could help minimize the bloat.

5.2.6 Problem #6: 2-element (ascending-order) sorter

Expectations

As our final experiment, the 2-element sorter was expected to also be a very good problem for the learners.

1-qubit case

The 1-qubit problem has a very short solution using the oracle encoding. The 2 most popular solutions that were found multiple times by the EDA-based GPs are shown in Programs 21 and 22 with their respective circuits in Figures 5.17 and 5.18. The details of these runs are found in Tables 5.17 and 5.18.

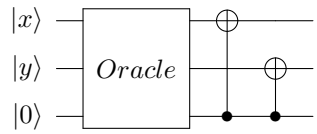


Figure 5.17: Solution #1 for 1-qubit 2-element sorter (oracle-based)

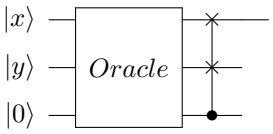


Figure 5.18: Solution #2 for 1-qubit 2-element sorter (oracle-based)

```
# id: sol[1:2107]
# fitness: 0.000000
# solution size: 3
begin_init
end_init
begin_code
oracle( NULL )
op_CNOT( NULL, 0, 2 )
op_CNOT( NULL, 0, 1 )
end_code
```

Program 21: Solution #1 for 1-qubit 2-element sorter (oracle-based)

```
# id: sol[30:307]
# fitness: 0.000000
# solution size: 2
begin_init
end_init
begin_code
oracle( NULL )
op_CSWAP( NULL, 0, 1, 2 )
end_code
```

Program 22: Solution #2 for 1-qubit 2-element sorter (oracle-based)

Parameter	*Setting*
GP type	NGRAM-QP-II
Solution set size	1500
Max iterations	2000
Max sol. length	3
Selection %	0.20
Learning rate	0.01
Perturb. freq.	50
Perturb. set size	750
Learn len. model	Yes
Fitness	avgerr, misses
Function set	CNOT, H, SWAP, X, CCNOT, Z, ORACLE, S, CS

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	1
Solution size	3

(b) Solution details

Table 5.17: Details for 1-qubit 2-element sorter #1 (oracle-based)

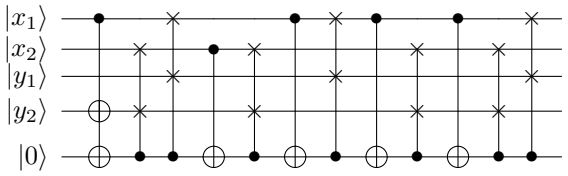


Figure 5.19: Optimized solution for 2-qubit 2-element sorter (non-oracle-based)

2-qubit case (non-oracle)

As mentioned in the experimental set-up we looked at two different ways of encoding the 2-qubit sorter. One used an oracle and the other did not. For our first experiment we tried to sort without an oracle. A solution was found by NGRAM-QP-II and is shown in Program 23 and as a circuit in Figure 5.19. An interesting thing to note is that this solution was found at iteration 1701, shortly after we had perturbed the distribution models with some random samples. The fitness had hovered around 2 (missed testcases) for several iterations leading up to that point, when it suddenly found a solution. This might be an indication that the random perturbation option (see Chapter 4 for more details) is indeed working to keep the models exploring. Details of this run can be found in Table 5.19.

Finally, we attempted to find a sorter using the oracle encoding described earlier. We had a result using NGRAM-QP-II, which is shown in Program 24 in `smallqc` syntax, as well as a quantum circuit in Figure 5.20. Details of this run can be found in Table 5.20. It appears that the GP did what we expected and came up with a very simple algorithm that swaps the given registers conditional on the green light of the oracle.

```
# id: sol[1701:4252811]
# fitness: 0.000000
# solution size: 23
begin_init
end_init
begin_code
op_CSWAP( NULL, 0, 4, 2 )
op_CNOT( NULL, 4, 0 )
op_CSWAP( NULL, 0, 3, 1 )
op_CSWAP( NULL, 0, 4, 2 )
op_CNOT( NULL, 3, 0 )
op_CSWAP( NULL, 0, 2, 4 )
op_CSWAP( NULL, 0, 2, 4 )
op_CSWAP( NULL, 0, 1, 3 )
op_CNOT( NULL, 4, 0 )
op_CNOT( NULL, 4, 0 )
op_CSWAP( NULL, 0, 2, 4 )
op_CSWAP( NULL, 0, 1, 3 )
op_CSWAP( NULL, 0, 1, 3 )
op_CSWAP( NULL, 0, 4, 2 )
op_CNOT( NULL, 4, 0 )
op_CSWAP( NULL, 0, 2, 4 )
op_CNOT( NULL, 4, 0 )
op_CSWAP( NULL, 0, 1, 3 )
op_CNOT( NULL, 4, 0 )
op_CSWAP( NULL, 0, 1, 3 )
op_CNOT( NULL, 4, 0 )
op_CNOT( NULL, 4, 0 )
op_CSWAP( NULL, 0, 4, 2 )
end_code
```

Program 23: Solution for 2-qubit 2-element sorter (non-oracle-based)

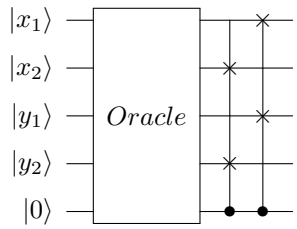


Figure 5.20: Solution for 2-qubit 2-element sorter (oracle-based)

Parameter	*Setting*
GP type	EDA-QP-II
Solution set size	10
Max iterations	2000
Max sol. length	2
Selection %	0.20
Learning rate	0.05
Perturb. freq.	50
Perturb. set size	5
Learn len. model	Yes
Fitness	avgerr, misses
Function set	CNOT, H, SWAP, X, CCNOT, Z, ORACLE, S, CS, CSWAP

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	30
Solution size	2

(b) Solution details

Table 5.18: Details for 1-qubit 2-element sorter #2 (oracle-based)

```
# id: sol[192:481615]
# fitness: 0.000300
# solution size: 3
begin_init
end_init
begin_code
oracle( NULL )
op_CSWAP( NULL, 0, 1, 3 )
op_CSWAP( NULL, 0, 4, 2 )
end_code
```

Program 24: Solution for 2-qubit 2-element sorter (oracle-based)

Conclusions

We saw most solutions to this problem from the pure learners, EDA-QP and ngram-QP, but the set-up based on the idea of an oracle was conducive to good performance by the learners. The more interesting case was the non-oracle 2-qubit sorter. For this particular case we did not find any other solution than the one returned by ngram-QP using NGRAM-QP-II. Recall that NGRAM-QP-II models sequences of function-target pairs, where a relationship is thus established between functions and their target qubits. It appears as though the solution (Program 24 and Figure 5.20) sees a lot of repetition of CNOT gates with target qubit 0 and CSWAP gates with target qubits 1, 2, 3 and 4. NGRAM-QP-II was the only one of our GP variants that was capable of learning the features of a solution that would lead to this particular one.

Parameter	*Setting*
GP type	NGRAM-QP-II
Solution set size	2500
Max iterations	2500
Max sol. length	25
Selection %	0.20
Learning rate	0.01
1/2 -gram	0.05/0.95
Perturb. freq.	50
Perturb. set size	1250
Learn len. model	No
Fitness	misses
Function set	CNOT, H, SWAP, X, CCNOT, Z

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	1701
Solution size	23
Optimized size	12

(b) Solution details

Table 5.19: Details for 2-qubit 2-element sorter (non-oracle-based)

5.2.7 Summary

Clearly not all of our experiments were successful. By examining the EDA models throughout the runs it is easy to see that even though sometimes the models converge, the convergence is at times sub-optimal, as with the length model which was often biased towards shorter sizes. During other runs, the distribution models would stay fairly uniform, which effectively made the learner GP act similar to NQP. Some problems, like the 2-qubit non-oracle-based sorter seem to be fair candidates for the learner GPs, but little can be said about the general performance of the learner GPs on arbitrary problems.

Numerous experiments point to ngram-QP having a higher probability of generating a lot of bloat, which makes it more difficult to identify the good functions within a sequence littered with useless functions. ngram-QP with the NGRAM-QP-II option was possibly our most successful GP variant throughout all the runs, so even though it generates a lot of by-products, it might be possible that it uses this to its advantage. NGRAM-QP-II is the only one of the variants which models neither inputs nor functions as location-dependent and instead tries to approximate global properties.

The fact that HQP dealt better with one of the harder problems (the 2-qubit half-adder) than did the pure learner GPs, suggests, as does the literature reviewed in Chapter 3, that quantum programming is a difficult task for a GP and a better ability to mutate results in more successful jumps across a fitness landscape that is rugged, unwelcoming and difficult to navigate.

The few problems we have tried are small in size and the circuit-level programming is extremely low-level, rendering a sequence of operations immensely sensitive to perturbations. For this reason we think that the results in this chapter might not be the best

Parameter	*Setting*
GP type	NGRAM-QP-II
Solution set size	2500
Max iterations	2000
Max sol. length	15
Selection %	0.20
Learning rate	0.075
1/2 -gram	0.05/0.95
Perturb. freq.	50
Perturb. set size	1250
Learn len. model	No
Fitness	misses, err, eff
Function set	CNOT, CCNOT, CZ, H, WHT, CH, ORACLE, Z, X, SWAP, CSWAP

(a) Run settings

Found solution	Yes
Solution type	Deterministic
Found in iteration	192
Solution size	4

(b) Solution details

Table 5.20: Details for 2-qubit 2-element sorter (oracle-based)

indicators of future performance on more complex, but higher-level problems. Unfortunately, at present such experiments are not feasible, due to the difficult task of simulating quantum hardware and most likely will not be possible before we have working quantum computers.

The inclusion of WHT for the Deutsch-Jozsa experiments, CSWAP for the sorter and and the multi-gate PROC1 operator for the adder experiments all improved performance and helped guide the evolver to better and more efficient solutions. This is to be expected, as these operators can be thought of as building blocks that give extra power and cohesion to the evolver. To conclude this chapter on a positive note, perhaps more of these building blocks can be identified and added to the function set, as more problems are explored, in the future.

Chapter 6

Conclusions and future work

"Problems worthy of attack prove their worth by hitting back"[1]

The process of this thesis has spawned many new ideas that unfortunately were not possible to attempt within the time constraints. Several ideas were narrowed down to the topic presented in this thesis; that is, the evolution of quantum programs through EDA-based GP. But several more ideas that we believe are more interesting and have potential for future research are more than just slight extensions of the current topic and so this chapter begins with conclusions and potential improvements to the current EDA-based GP framework for evolution of quantum programs and then proceeds into a lengthier discussion on how we might branch for future research.

6.1 Conclusions

The original hypothesis of this work was that *a) quantum programs exhibit sequential patterns and relationships between their functions and inputs that can be mined to help automatically generate programs* and *b) a stochastically-driven GP engine with an additional learner to intelligently perturb features could have an advantage over one without the learner.*

The results presented in the previous chapter were too ambiguous to either prove or disprove the hypothesis. The four instances of GP that were tested against each other seemed to perform equally well on the problems that were able to be solved and equally poorly on the problems that were not solved. In some cases the EDA-based GPs did perform better; however, given that each run took a few hours to complete, not enough experiments were completed to make an actual claim that the these approaches were in any way better.

[1] Anonymous quote, taken from *Contemporary Abstract Algebra* [19].

As for the second point, we found no evidence that either NQP which was fully driven by mutation, or the HQP, which used intelligent mutation performed better. The test problems were small, however, and we feel it is too early to generalize the results to larger-scale problems.

6.1.1 Potential flaws

While some of the results we achieved were interesting, we realize the methodology has a few shortcomings (both at the implementation level and the experimentation level), which we shall try to address here.

Lack of data

Machine learning requires data [6, 14] on which to operate and from which to learn; hence the three EDA-based GP instances all require data to learn from. During a GP run this data comes from candidate solutions which have a better relative fitness; however, the knowledge an EDA model acquires is recycled at each iteration and fed back into a new sample population which then proceeds to reinforce or improve the model. It is difficult to realize great amounts of data from which to learn, as the limitations imposed on a quantum simulation force us to keep solution set size small. The fewer solutions a model has to work with at a particular iteration, the less it can learn.

For the N-gram approaches it is also very important to have *longer* programs to learn from, such that more sequences may be observed. It is difficult for the N-gram distributions to be learnt properly, especially when our function set is large and not enough good solutions appear at each iteration. In many of our solutions which we had to optimize by hand, *introns*[2] accounted for a large number of the gates; that is, useless code took up a *lot* of space (seemingly) unnecessarily. While the GPs all allowed for variable-sized solutions to be generated, we did use an upper bound on the size, so as to avoid high computation times and to focus the search on smaller, more efficient solutions. Despite this and despite the occasional use of the efficiency component (see Chapter 4) in the fitness evaluation, a lot of introns still made their way into our solutions. If it were not for the space they take up in the candidate solutions and the misleading (useless) information they feed to the models, introns in a final solution would not be a big deal, as we can always edit a solution by hand, as we did multiple times in Chapter 5.

One of our distribution models attempted to learn the length of a solution. We found early on that this did not always lead to good results. The length model had a tendency for premature convergence, which in retrospect makes sense when we recognize that there is even less data at each iteration for the length model to work with. An N-gram, for

[2] An intron in GP [2] is a small piece of code which has no effect on the overall fitness of a solution; that is, the code is useless, for example, such as a no-op function, or a sequence of CNOT(0,1) - CNOT(0,1) gates, which cancel each other out.

example is able to split up one solution into multiple segments to learn from, while the length model only gets a single length value from each solution in the set. We added an option to turn off the learning of the solution size and found that our results improved considerably.

6.2 Improvements

Several improvements could be made to the methodology of Chapter 4. Here we discuss four main ideas: *quantification of entanglement*, *self-adaptive parameters*, *variable-length ancilla register* and *data mining of known algorithms*.

6.2.1 Entanglement

There is at least one feature that was intended to be implemented as part of the EDA-based approaches of Chapter 4. Unfortunately, this feature has been left out for now and is considered for future work. Entanglement is an essential quantum property which allows for interesting behaviours, such as the teleportation circuit discussed in Chapter 2. On the convoluted road to understanding entanglement, we thought it would be especially interesting to see what a GP would do with the ability to *measure* entanglement and *induce* entanglement into its solutions through positive reinforcement from a fitness function. Quantifying entanglement is not an easy process [24, 78], but for small systems of 2 and 3 qubits it could be possible to track entanglement and its effects on a quantum state, throughout the entire solution. We have only introduced a very simple component to the fitness function in Chapter 4, which favours solutions that might have the potential for entanglement production, as suggested by specific sequences of gates. As discussed in Chapter 4, this component does *not* guarantee that a solution which produces and/or uses entanglement will be recognized. As such, an interesting improvement to the fitness function might be a full-fledged entanglement-quantifier.

While our methodology was able to generate reversible circuits made of quantum gates, a lot of the results were not strictly-quantum circuits, in that they made no special use of entanglement or superposition. We believe the fitness function would be a great way to promote quantum properties in our solutions.

6.2.2 Self-adaptive parameters

GP often makes use of self-adaptive parameters [49], such as crossover rates which are lowered as a solution set starts to converge, or mutation rates which are increased as a solution set starts to stagnate.

The N-gram GP and HQP with an N-gram learner used different rates to combine the use of a unigram, a bigram and a trigram. These rates were set by the user before a run

and never changed throughout a run. It might be instructive to allow self-adaptation of these parameters, as a run proceeds. For example, at the beginning of the evolutionary process it makes little sense to heavily rely on the bigram and trigram models, as these have not had a chance to properly learn anything yet. So at the beginning of a run, the parameters could perhaps be 0.75 0.10 0.05, to encourage the use of the unigram. As the models gather more data these parameters could change to reflect the new confidence in the bigram and trigram models, by increasing their rates and decreasing that of the unigram.

Other parameters that can be self-adapted are the learning rates for all the models. We currently use a constant learning rate, which is also set by the user at run-time. A larger learning rate allows for faster learning, but can lead to overshooting and sub-optimal convergence. A learning rate that is too low would slow down the process substantially. Allowing the learning rate to increase or decrease according to the quality of the current iteration's candidate set, or some indicator that the evolution is either not going well, or indeed going well, might help to speed up the process, or avoid sub-optimal convergence.

6.2.3 Training data and bias

One of the main motivators for an EDA-based approach for generation of quantum programs was the relative ease with which one could *bias* the original solution set to knowledge extracted from previous programs, algorithms and even concepts. As more quantum algorithms are developed these can be gathered as training data and mined in order to create the prior models for a GP run. Earlier it was mentioned that a major reason the N-gram approach might not have done as well as hoped was the lack of data. Not only *more* programs, but *longer* programs would be especially helpful to the N-gram approach, as longer programs would have more potential for exhibiting sequential patterns. Observing longer sequences might even allow the modelling of a 4-gram or greater.[3] As we gather more data from known successful programs and algorithms, we can build much better prior models for an EDA-based approach[4] which might lead to an improved performance of our method.

For a simple example of how biasing the priors might help, we can think back to the *imperfect copy machine* of Chapter 5, in which our input contained two registers `src` and `dst` (not in a superposition) and we wished to replicate `src`'s contents in `dst` (essentially copying a computational basis state). We do not suppose that `dst` is clear (zeroed) and instead use a number of pre-zeroed work qubits which we can swap with the qubits of

[3] Actually, we expect, that as in natural language computing, a tri-gram is probably no worse than anything larger than that and probably the better choice [36], due to its lesser dimensionality, but even so it might still be worthwhile to study longer sequences for quantum code, given that, for example, the amplitude amplification step in Grover's search is longer than 3 gates.

[4] Currently our prior models are uniform.

`dst` to ensure that the destination register may *become* clear. With such knowledge we could bias the priors to include SWAP operations on the work qubits and qubits of `dst`. For EDA-QP-II, where operators and their inputs are modelled versus node locations we can ensure that for example, the first nodes are such SWAP operations to clear out `dst` before we attempt to copy `src` into it.

As another example for the same problem, we could restrict 2-qubit operations between qubits of `src` and qubits of `dst` to only go one way; that is, control qubits would always be those of `src` and target qubits could be those of `dst`, but not the other way around.

As a third example, recall the quantum teleportation protocol of Chapter 2, whose GP evolution in previous work was discussed in Chapter 3. In the protocol two parties own different qubits and they may not perform operations on each other's qubits. Such restrictions could easily be incorporated by the prior models.

As a final example, it would be possible to bias the length model to longer solutions, if we know for sure that a solution cannot possibly be made up of 1 or 2 nodes, or if we have some specific idea about how large a solution might be.

6.2.4 Variable-length ancilla registers

Ancilla qubits are really important to a computation, as they are able to hold temporary values, control operations on other qubits and swap out garbage data from qubits with unknown starting values. Our experiments all used fixed-length ancilla registers, for obvious reasons which included the need to keep the overall tensor length down for feasible simulations and the need to restrict the number of possible combinations of functions and operands, for smaller search spaces and higher probability of success.

Allowing the GP to use variable-length ancilla registers might lead to more flexible evolution. Of course this adds a burden to the simulator and to the entire evolutionary process as a result, but multiple (temporary) work qubits might allow a problem to complete quicker, or better. This is especially true in the absence of a ZERO gate (as with our GPs), as the only way to ensure a qubit is 0 is by swapping it with an ancilla qubit which we assume starts off in a state of 0.

To keep the computational costs down we could use hybrid computations, where classical and quantum bits combine. For example, we might use classical work bits instead of qubits, simply to control other operations. Obviously such bits could not exist in superpositions and would not form a valid part of the quantum tensor, but they might be useful to control operations. We could limit the actions of and on the classical bits through the biased priors.

Finally, with variable-length ancilla registers we might also use an additional distribution model to learn the optimal length for the register for a given problem.

6.3 Future research

The ideas for future research are best introduced as the following two questions:

1. *Can we automatically generate* coherent *quantum programs using GP?*

2. *Might GP be more successful for a different model of quantum computation?*

Let us revisit each question in turn.

6.3.1 Automatic fault-tolerant quantum programming

To the knowledge of the author, all the quantum programs and algorithms studied from a GP point of view thus far have been generated under the implicit assumption that the final result (the program) would run in an ideal environment[5]. In reality, as mentioned in Chapter 2, a quantum computer will never be able to run in a completely isolated environment; it will always be susceptible to random changes triggered by its interactions (even though they may be slight) with the larger system in which it is contained. These interactions might cause unpredictable decoherence and partial collapse of quantum superpositions. This effectively renders an idealist quantum algorithm unpredictable, in practice, even though in theory the algorithm might be deterministic.

Quantum error correction [13, 23, 52] deals with correcting decoherence errors in quantum computations, much like *classical error correction* attempts to correct bit flip errors in classical communications. Unlike classical calculations, where the only possible error is a bit flip, in quantum computing there are multiple types of errors: the bit flip (think gate X or NOT), the phase flip (think gate Z) and random perturbations to amplitudes – something that cannot be observed in a classical bit. Current quantum technology is ever so sensitive to decoherence and it is unrealistic to expect future technology to handle coupling with the environment with significant improvements in precision. Thus, quantum error correction will continue to be essential to quantum computation and we believe that GP might be a very useful tool to learn *coherent* and *fault-tolerant* quantum circuits and programs. Coherence here refers to the ability of the quantum circuits (and programs) to deal with and correct errors. This is something that can be built into a circuit itself and thus could also be encouraged or promoted by a suitably-defined fitness function.

[5] This point can be debated, since we have discussed work by Spector and Massey and others [40, 66, 20] which generated probabilistic quantum programs and the simple fact that these were probabilistic solutions in a sense allowed for unexpected quantum state changes, as long as we view a non-deterministic choice as a random collapse of the quantum state and assume it can be deffered as a normal measurement would.

6.3.2 GP for alternate models of quantum computation

Just as in classical computing the digital circuit model is not the only model of computation, in quantum computing there are other models of computation besides the quantum circuit model, although, the quantum circuit model is possibly the easiest model to comprehend, as it is most alike the standard classical model.

Three main models of quantum computation are 1) the *quantum circuit*, 2) the *quantum Turing machine* [11, 18] and 3) the *quantum cluster-state* [54, 41]. These three are equivalent, in that either of them can simulate the other two, in polynomial time. The quantum Turing machine, just like its classical counterpart is very useful as a theoretical construct, but not very practical. The current standard model of quantum computation is based on the classical circuit model. As such, the standard model is a quantum circuit model, in which a state in memory (a register of qubits) evolves through time as a result of interacting with unitary operations (quantum gates), similar to the way a classical bit register evolves through time as a result of interactions with classical gates. Another model for quantum computation is measurement-based quantum computation [28], in which changes in the quantum system are not effected by unitary transformations, but instead by measurements on subsets of the qubits in the system [28, 8, 53, 44]. All known models of quantum computation are equivalent, in that one can be used to efficiently simulate another; however, studying each model in its own right offers new insights into quantum algorithms and the quantum effects at the heart of these algorithms.

GP for the cluster-state model

There are a few different approaches to measurement-based quantum computation, such as the *cluster-state model* [53], and the *teleportation quantum model* [28], which is based on the quantum teleportation protocol (see Chapter 2). In the cluster-state model a graph (or lattice) of qubits (for real applications this would most likely be a really large number of qubits) is initially prepared in an entangled state. The graph's vertices represent qubits of the system, while any edges in the graph represent an entanglement between the connected qubits. The qubits can normally be prepared in the superposition state $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. A CZ gate (controlled-Z gate) can then be applied to each edge of the graph to entangle qubits which are neighbours. This entanglement is independent of the problem being solved and is purely a *resource* used for computation [53]. Information is processed by measuring certain qubits, in a specific order, where measurements on subsequent qubits are done in bases which are sometimes dependent on measurement results of prior qubits. As such, the measurement bases are "adaptive" and a cluster-state quantum algorithm is defined by the temporal ordering of the measurements. The entanglement in the cluster state allows information to flow, which is somewhat analogous to the information flow in quantum teleportation. It has been shown that the cluster-state model is equivalent to the quantum circuit model, in that the

cluster-state model can be used to simulate any algorithm in the quantum circuit model and vice versa [53]. Raussendorf and Briegel [53] describe the computational model of cluster-state computing as distinct from the network model, with focus on what they call the information flow vector, which propagates measurement results throughout the cluster and in a sense replaces the idea of the qubit as the quantum unit of information.

GP could be useful in a number of ways for evolution of cluster-state quantum programs and algorithms. Firstly, a unitary transformation in the cluster-state model happens as the result of a measurement; however, this measurement has to be in a specific basis (not always the standard basis). A GP could be used to decompose different transformations into a sequence of measurements in various bases, in an efficient manner. Such a decomposition might be useful in practice when actually building devices that operate on the cluster-state model.

A GP might also be used to evolve cluster-state programs (and algorithms) by modelling a solution as a list of nodes, where each node encodes a measurement on a particular qubit in the lattice. The lattice restricts the length of the programs, as each qubit can be measured exactly once and so an EDA-based approach where measurements are location-specific might be useful.

Finally, GP might be used to exploit the entanglement resource and find optimal ways of using entanglement in the lattice, for a particular problem.

As mentioned, the cluster-state model is not the only model of quantum computation and it is not even the only in the class of measurement-based models, but it is an interesting model as it emphasizes the importance of entanglement for computation. It could be that the cluster-state model might be more amenable to a GP and a study of the fitness landscape of a cluster-state model GP versus a circuit model GP might show promise.

6.4 Summary

The automatic generation of quantum programs is a difficult task, but there remain many as yet unexplored paths. Several of the results we have achieved would indicate that it might be possible to use EDA-based GP as an aid in the generation process of quantum circuits and programs, while our probabilistic results might encourage further improvements to the subarea of automatic quantum programming. Not all of our experiments were successful, however, even after multiple attempts and for this reason, we would like to emphasize once again the difficulty of automatic low-level quantum programming. We have discussed how it might be possible to improve the current methodology and have suggested several paths for further research, which we believe could hold promise for the future of automatic quantum programming.

Appendix A

Complex math basics

Complex numbers play an important role in quantum mechanics. This section reviews the essentials, as relevant to this thesis. A complex number z is a 2-dimensional mathematical entity, of the form:

$$z = a + bi;$$

where $a, b \in \mathbb{R}$ and i satisfies $i^2 = -1$.

A complex number can thus be described by its *real* part,

$$\mathbf{Re}(z) = a$$

and its *imaginary* part,

$$\mathbf{Im}(z) = b.$$

The *complex conjugate* of a complex number $z = a + bi$ is defined as:

$$\bar{z} = a - bi$$

Given two complex numbers $v = a + bi$ and $w = c + di$, addition and subtraction are defined as:

$$z = v \pm w = (a \pm c) + (b \pm d)i$$

multiplication is defined as:

$$z = vw = ac + adi + bci + bdi^2 = ac + (ad + bc)i + bd(-1) = (ac - bd) + (ad + bc)i$$

and division is defined as:

$$z = \frac{v}{w} = \frac{v\bar{w}}{w\bar{w}} = \frac{(a+bi)(c-di)}{(c+di)(c-di)} = \frac{(ac+bd)+(bc-ad)i}{c^2+d^2}. \tag{A.1}$$

The *norm* or *modulus* of a complex number z is a scalar defined[1] by the inner product of z and its complex conjugate $\bar{z}$:

$$\begin{aligned} \|z\| &= \sqrt{z\bar{z}} \\ &= \sqrt{(a+bi)(a-bi)} \\ &= \sqrt{a^2 - abi + abi - b^2 i^2} \\ &= \sqrt{a^2 - b^2(-1)} \\ &= \sqrt{a^2 + b^2} \\ &\geq 0 \end{aligned}$$

(with equality only when $z = 0$).

A complex number z can also be represented by a 2-dimensional vector:

$$\begin{pmatrix} a \\ b \end{pmatrix}.$$

We can now think of a complex number as a point in a plane (for example, the X-Y plane). This gives a complex number $z = a + bi$ a non-unique *polar representation*:

$$a = \|z\| \cos\theta \quad \text{and} \quad b = \|z\| \sin\theta$$

$$\text{where} \quad \theta = \tan^{-1}\left(\tfrac{a}{b}\right) \quad \text{and} \quad z = \|z\|(\cos\theta + i\sin\theta).$$

Euler's formula

A very important relation is Euler's formula which relates the exponential function to complex numbers as follows:

$$e^z = e^{a+bi} = e^a(\cos b + i \sin b)$$

[1] We say that the modulus is *induced* by a particular inner product, since it is defined relative to the given inner product.

From this we can easily derive two important properties:

$$\begin{aligned}
\|e^z\| &= \|e^a(\cos b + i \sin b)\| \\
&= \sqrt{e^a(\cos b - i \sin b)(e^a \cos b + i \sin b)} \\
&= \sqrt{e^{a+a}(\cos b \cos b + i \cos b \sin b - i \sin b \cos b - i^2 \sin b \sin b)} \\
&= \sqrt{e^{2a}(\cos^2 b + \sin^2 b)} \\
&= \sqrt{e^{2a}(1)} \\
&= \sqrt{e^{a+a}} \\
&= \sqrt{e^a e^a} \\
&= e^a,
\end{aligned}$$

and consequently,

$$\|e^{it}\| = e^a = e^0 = 1, \forall t \in \mathbb{R}.$$

Appendix B

Quantum gates used in `smallqc`

The matrix representations in standard computational basis of most of the quantum gates implemented in `smallqc` and discussed in Chapter 4 are listed below. Not listed are some of the controlled versions.

B.1 Quantum gates

$$NOT = X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix},$$

$$H = \tfrac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad PS(\theta) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{pmatrix}, \quad S = PS(\tfrac{\pi}{2}) \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix},$$

$$T = PS(\tfrac{\pi}{4}) \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{pmatrix}, \quad W = PS(\tfrac{3\pi}{2}) \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix},$$

$$R_X(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -i\sin\frac{\theta}{2} \\ -i\sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix}, \quad R_Y(\theta) = \begin{pmatrix} \cos\frac{\theta}{2} & -\sin\frac{\theta}{2} \\ \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{pmatrix},$$

$$R_Z(\theta) = \begin{pmatrix} e^{-\frac{i\theta}{2}} & 0 \\ 0 & e^{\frac{i\theta}{2}} \end{pmatrix}, \quad CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix},$$

$$CZ = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{pmatrix}, \quad CS = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{pmatrix},$$

$$SWAP = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad CCNOT = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix},$$

$$CSWAP = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Bibliography

[1] Quantum interaction. `http://www.quantuminteraction.org/`.

[2] Banzhaf, W., Nordin, P., Keller, R. E., and Francone, F. D. *Genetic Programming. An Introduction.* Morgan Kaufmann Publishers, Inc., San Francisco, California, 1998.

[3] Barnum, H., Bernstein, H. J., and Spector, L. A quantum circuit for OR. `http://arXiv.org/abs/quant-ph/9907056`, Oct. 08 1999.

[4] Barnum, H., Bernstein, H. J., and Spector, L. Quantum circuits for OR and AND of ORs. *Journal of Physics A: Mathematical and General 33*, 45 (17 Nov. 2000), 8047–8057.

[5] Bernstein, D. J. Introduction to post-quantum cryptography. In *Post-quantum cryptography: [First International Workshop on Post-Quantum Cryptography ... at the Katholieke Universiteit Leuven in 2006]*, D. J. D. J. Bernstein, J. Buchmann, and E. Dahmén, Eds. Springer-Verlag, pub-SV:adr, 2009, pp. 1–14.

[6] Bishop, C. M. *Pattern Recognition and Machine Learning.* Springer, 2006.

[7] Brassard. Cryptology column – quantum computing: The end of classical cryptography? *SIGACTN: SIGACT News (ACM Special Interest Group on Automata and Computability Theory) 25* (1994), 15–21.

[8] Browne, D., and Briegel, H. One-way quantum computation, 2006. `arxiv.org/abs/quant-ph/0603226v2`.

[9] Darwin, C. *The Origin of Species.* Penguin Group, Inc., New York, New York, 2003.

[10] Denning, P. J. Computing is a natural science. *Commun. ACM 50*, 7 (2007), 13–18.

[11] Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. *PRSLA: Proc. R. Soc. Lond. A 400* (1985), 97–117.

[12] Deutsch, D., and Jozsa, R. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London Ser. A A439* (1992), 553–558.

[13] Devitt, S. J., Nemoto, K., and Munro, W. J. Quantum error correction for beginners, June 20 2009. `http://arxiv.org/abs/0905.2794`.

[14] Dietterich, T. G. Machine learning for sequential data: A review. In *SSPR: SSPR, International Workshop on Advances in Structural and Syntactical Pattern Recognition* (2002), LNCS, pp. 15–30.

[15] Domingos, P. A few useful things to know about machine learning. *Commun. ACM 55*, 10 (2012), 78–87.

[16] Eastin, B., and Flammia, S. T. Q-circuit tutorial, Aug. 24 2004. `http://arxiv.org/abs/quant-ph/0406003`.

[17] Fisher, S. D. *Complex Variables*, 2nd ed. Dover Publications, 1999.

[18] Fouché, W., Heidema, J., Jones, G., and Potgieter, P. H. Deutsch's universal quantum turing machine (revisited), Jan. 16 2007. `http://arxiv.org/abs/quant-ph/0701108`.

[19] Gallian, J. A. *Contemporary Abstract Algebra*, 7th ed. Brooks/Cole Cengage Learning, 2009.

[20] Gepp, A., and Stocks, P. A review of procedures to evolve quantum algorithms. *Genetic Programming and Evolvable Machines 10*, 2 (June 2009), 181–228.

[21] Goldberg, D. E. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, Reading, Mass., 1989.

[22] Gossett, P. Quantum carry-save arithmetic, Aug. 29 1998. `http://arxiv.org/abs/quant-ph/9808061`.

[23] Gottesman, D. An introduction to quantum error correction and fault-tolerant quantum computation, Apr. 16 2009. `http://arxiv.org/abs/0904.2557`.

[24] Gühne, O., and Toth, G. Entanglement detection. Tech. rep., Feb. 27 2008.

[25] Hauschild, M., and Pelikan, M. An introduction and survey of estimation of distribution algorithms. *Swarm and Evolutionary Computation 1*, 3 (2011), 111–128.

[26] Holzner, S. *Quantum Physics for Dummies*. Wiley Publishing, Inc., Hoboken, New Jersey, 2009.

[27] Hoyer. Introduction to recent quantum algorithms. In *MFCS: Symposium on Mathematical Foundations of Computer Science* (2001).

[28] Jozsa, R. An introduction to measurement based quantum computation, 2005. `arxiv.org/abs/quant-ph/0508124`.

[29] Kantschik, W., and Banzhaf, W. Linear-tree GP and its comparison with other GP structures. In *Genetic Programming, Proceedings of EuroGP'2001* (Lake Como, Italy, 18-20 Apr. 2001), J. F. Miller, M. Tomassini, P. L. Lanzi, C. Ryan, A. G. B. Tettamanzi, and W. B. Langdon, Eds., vol. 2038 of *LNCS*, Springer-Verlag, pp. 302–312.

[30] Landauer, R. The physical nature of information. *Physical Letters A 217 (1996)* (May 9 1996), 188–193.

[31] Leier, A., and Banzhaf, W. Evolving Hogg's quantum algorithm using linear-tree GP. In *Genetic and Evolutionary Computation – GECCO-2003* (12-16 July 2003), vol. 2723, Springer-Verlag, pp. 390–400.

[32] Leier, A., and Banzhaf, W. Exploring the search space of quantum programs. In *Proceedings of the 2003 Congress on Evolutionary Computation CEC2003* (8-12 Dec. 2003), IEEE Press, pp. 170–177.

[33] Leier, A., and Banzhaf, W. Comparison of selection strategies for evolutionary quantum circuit design. In *Genetic and Evolutionary Computation Conference - GECCO 2004, Proceedings, Part II* (26-30 June 2004), vol. 3103 of *Lecture Notes in Computer Science*, Springer, pp. 557–568.

[34] Looks, M., Goertzel, B., and Pennachin, C. Learning computer programs with the bayesian optimization algorithm. In *GECCO* (2005), H.-G. Beyer and U.-M. O'Reilly, Eds., ACM, pp. 747–748.

[35] Lozano, J. A., naga, P. L., and Inza, I. *Towards a New Evolutionary Computation : Advances in the Estimation of Distribution Algorithms*. Springer, Berlin, 2006.

[36] Manning, C., and Schütze, H. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, 1999.

[37] Massey, P., Clark, J. A., and Stepney, S. Evolving quantum circuits and programs through genetic programming. In *Genetic and Evolutionary Computation – GECCO-2004, Part II* (26-30 June 2004), vol. 3103 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 569–580.

[38] Massey, P., Clark, J. A., and Stepney, S. Evolution of a human-competitive quantum fourier transform algorithm using genetic programming. In *Proceedings of the 2005 conference on Genetic and evolutionary computation* (New York, NY, USA, 2005), ACM, pp. 1657–1663.

[39] Massey, P., Clark, J. A., and Stepney, S. Human-competitive evolution of quantum computing artefacts by genetic programming. *Evolutionary Computation 14*, 1 (Spring 2006), 21–40.

[40] Massey, P. S. *Searching for Quantum Software*. PhD thesis, Department of Computer Science, University of York, UK, Aug. 2006.

[41] McMahon, D. *Quantum Computing Explained*, 1st ed. John Wiley & Sons, Inc., Hoboken, New Jersey, 2008.

[42] Mitchell, T. M. The discipline of machine learning. Unpublished manuscript, Carnegie Mellon University., 2006.

[43] Nicholson, W. K. *Linear Algebra with Applications*, 4th ed. McGraw-Hill Ryerson Limited, Toronto, 2003.

[44] Nielsen, M. A. Cluster-state quantum computation, 2005. `arxiv.org/abs/quant-ph/0504097`.

[45] Nielsen, M. A., and Chuang, I. L. *Quantum Computation and Quantum Information*, 10th anniversary ed. Cambridge University Press, Cambridge, UK, 2010.

[46] Papanikolaou, N. An introduction to quantum cryptography. *ACM Crossroads 11*, 3 (2005), 3.

[47] Pelikan, M. *Bayesian optimization algorithm: from single level to hierarchy*. PhD thesis, University of Illinois at Urbana-Champaign.

[48] Pelikan, M., Goldberg, D. E., and Lobo, F. G. A survey of optimization by building and using probabilistic models. *Comp. Opt. and Appl 21*, 1 (2002), 5–20.

[49] Poli, R., Langdon, W. B., and McPhee, N. F. *A field guide to genetic programming*. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk`, 2008.

[50] Poli, R., and McPhee, N. F. A linear estimation-of-distribution GP system. Tech. Rep. CES-479, Department of Computing and Electronic Systems, University of Essex, UK, 2008. Retrieved via: `http://www.essex.ac.uk/dces/research/publications/technicalreports/2007/ces-479.pdf`, May 2013.

[51] Preskill, J. Quantum computation ph219/cs219 lecture notes. Retrieved from `http://www.theory.caltech.edu/people/preskill/ph229` on Friday, August 2nd, 2013.

[52] Preskill, J. Fault-tolerant quantum computation, Dec. 19 1997. http://arxiv.org/abs/quant-ph/9712048.

[53] Raussendorf, R., and Briegel, H. Computational model underlying the one-way quantum computer, 2002. arxiv.org/abs/quant-ph/0108067v2.

[54] Rieffel, E., and Polak, W. *Quantum Computing A Gentle Introduction*. Massachusetts Institute of Technology, 2011.

[55] Rincon, P. D-wave: Is $15m machine a glimpse of future computing?, May 2014. BBC News website. Retrieved via: arxiv.org/abs/quant-ph/0603226v2.

[56] Rubinstein, B. I. P. Evolving quantum circuits using genetic programming. In *Genetic Algorithms and Genetic Programming at Stanford 2000*. Stanford Bookstore, June 2000, pp. 325–334.

[57] Russell, S., and Norvig, P. *Artificial Intelligence: A Modern Approach*, 2nd ed. Prentice Hall, 2003.

[58] Rylander, B., Soule, T., and Forster, J. Quantum evolutionary programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2001)* (7-11 July 2001), Morgan Kaufmann, pp. 1005–1011.

[59] Salustowicz, R. *Probabilistic Incremental Program Evolution*. PhD thesis, Technischen Universitaet Berlin.

[60] Salustowicz, R. P., and Schmidhuber, J. Probabilistic incremental program evolution. *Evolutionary Computation 5*, 2 (1997), 123–141.

[61] Serway, R. A., and Jewett, J. W. J. *Principles of physics: A calculus-based text.*, 3rd ed., vol. 1. Thomson Learning, Inc., 2002.

[62] Shor. Why haven't more quantum algorithms been found? *JACM: Journal of the ACM 50* (2003).

[63] Shor, P. W. Ploynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer, Nov. 18 2011. The Pennsylvania State University CiteSeerX Archives, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.205.4779.

[64] Sofge, D. A. Prospective algorithms for quantum evolutionary computation. *CoRR abs/0804.1133* (2008). http://arxiv.org/abs/0804.1133.

[65] Sorensen, K., and Glover, F. Metaheuristics. *Encyclopedia of Operations Research and Management*. Retrieved via: www.opttek.com/sites/default/files/Metaheuristics.pdf, Apr. 2014.

[66] Spector, L. *Automatic Quantum Computer Programming A Genetic Programming Approach*. Springer Science+Business Media, LLC, New York, 2007.

[67] Spector, L., Barnum, H., and Bernstein, H. J. Genetic programming for quantum computers. In *Genetic Programming 1998: Proceedings of the Third Annual Conference* (Jan. 1998), Morgan Kaufmann, pp. 365–374.

[68] Spector, L., Barnum, H., Bernstein, H. J., and Swamy, N. Finding a better-than-classical quantum and/or algorithm using genetic programming. In *Proceedings of the Congress of Evolutionary Computation* (6-9 July 1999), vol. 3, IEEE Press, pp. 2239–2246.

[69] Spector, L., Barnum, H., Bernstein, H. J., and Swamy, N. Quantum computing applications of genetic programming. In *Advances in Genetic Programming 3*. MIT Press, Cambridge, MA, USA, June 1999, ch. 7, pp. 135–160.

[70] Spector, L., and Bernstein, H. J. Communication capacities of some quantum gates, discovered in part through genetic programming. In *Proceedings of the Sixth International Conference on Quantum Communication, Measurement, and Computing (QCMC)* (2003), Rinton Press, pp. 500–503.

[71] Stadelhofer, R. *Evolving blackbox quantum algorithms using genetic programming*. PhD thesis, Department of Physics, University of Dortmund, Dortmund, Germany, May 2006.

[72] Stadelhofer, R., Banzhaf, W., and Suter, D. Evolving blackbox quantum algorithms using genetic programming. *AI EDAM 22*, 3 (2008), 285–297.

[73] Stallings, W. *Cryptography and network security: Principles and practice*, 5th ed. Prentice Hall, 2011.

[74] Steane, A. Quantum computing, 1997. `http://arxiv.org/abs/quant-ph/9708022`.

[75] Stolze, J., and Suter, D. *Quantum Computing: A Short Course from Theory to Experiment*. WILEY-VCH GmbH & Co. KGaA, Weinheim, 2004.

[76] Turing, A. M. Computing machinery and intelligence. *Mind 59*, 236 (Oct. 1950), 433–60.

[77] Vedral, V., Barenco, A., and Ekert, A. Quantum networks for elementary arithmetic operations, Nov. 16 1995. `http://arxiv.org/abs/quant-ph/9511018`.

[78] Vedral, V., Plenio, M. B., Rippin, M. A., and Knight, P. L. Quantifying entanglement, Feb. 11 1997. `http://arxiv.org/abs/quant-ph/9702027`.

[79] Williams, C. P., and Gray, A. G. Automated design of quantum circuits. In *QCQS: NASA International Conference on Quantum Computing and Quantum Communications, QCQS* (1998).

[80] Witten, I. H., Frank, E., and Hall, M. A. *Data Mining: Practical machine learning tools and techniques*, 3rd ed. Morgan Kaufmann, 2011.

[81] Yabuki, T., and Iba, H. Genetic algorithms for quantum circuit design - evolving a simpler teleportation circuit. In *Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference* (8 July 2000), pp. 425–430.

Index