

Submitted in part fulfilment for the degree of MEng.

# **A Research Tool for the Automated Synthesis of Quantum Algorithms**

Sam Ratcliff

10th May 2011

Supervisor: John A Clark

Number of words = 32359, as counted by texcount.  
This includes the body of the report, and Abstract, but not any Appendices.

Number of pages = 69.  
This includes the body of the report, title page, abstract page, Contents, List of Figures and List of Tables but not any Appendices or blank sides.



### **Abstract**

Quantum computation provides massively parallel processing power. However, artefacts that can exploit this power are hard to design. Heuristic search has been employed to assist with research into the discovery of these artefacts. Various search techniques are used. Common to most of this research is the phase of bespoke development to implement peripheral utilities such as simulation and visualisation. This software is rarely released to the wider community. This seems an inefficient use of resources. The toolkit developed in this project provides implementations of a comprehensive set of such peripheral utilities. Using the toolkit, researchers can focus on developing new search technique or discovering new artefacts without the large preparatory development process previously required. The toolkit includes an implemented search technique and several suitability measures based on previous research or otherwise developed as part of this project. The toolkit is used to search for solutions to several identified quantum computation problems. Some problems are solved by heuristic search for the very first time.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Statement of Ethical Implications . . . . .	1
1.2	Report Structure . . . . .	1
<b>2</b>	<b>Literature Review</b>	<b>3</b>
2.1	Introduction to Quantum Computation . . . . .	3
2.2	An Introduction to Quantum Circuits and Algorithms . . . . .	7
2.3	The Use of Evolutionary Computation in the Synthesis of Quantum Algorithms . . . . .	12
2.4	The Focus of this Project . . . . .	17
<b>3</b>	<b>Requirements</b>	<b>18</b>
3.1	Definition, Acronyms, and Abbreviations . . . . .	18
3.2	Requirements Summaries . . . . .	18
<b>4</b>	<b>Design and Implementation</b>	<b>23</b>
4.1	Framework . . . . .	23
4.2	Provided Tools . . . . .	36
4.3	Search Engine Implementations . . . . .	37
4.4	Suitability Measure and Circuit Evaluator Implementations . . . . .	37
4.5	Provided Search Problems . . . . .	39
4.6	Provided GUI . . . . .	40
4.7	Summary . . . . .	43
<b>5</b>	<b>Testing</b>	<b>46</b>
5.1	Unit Tests . . . . .	46
5.2	Integration Tests . . . . .	48
5.3	Client GUI Testing . . . . .	49
5.4	Summary . . . . .	49
5.5	Tracability . . . . .	50
<b>6</b>	<b>Experimentation</b>	<b>53</b>
6.1	Deutsch Algorithm . . . . .	53
6.2	Deutsch-Jozsa Problem . . . . .	54
6.3	o..3 Permutation Max Problem . . . . .	56
6.4	Quantum Fourier Transform . . . . .	57
6.5	Unintended Functionality . . . . .	57
6.6	Summary . . . . .	58
<b>7</b>	<b>Evaluation and Future Work</b>	<b>59</b>
7.1	Have the Requirements Been Met? . . . . .	59
7.2	Strengths of the Toolkit . . . . .	59
7.3	Areas of Improvement and Future Work . . . . .	59
7.4	Experimentation Summary . . . . .	60
7.5	Project Evaluation . . . . .	61
<b>A</b>	<b>User Guide</b>	<b>66</b>
A.1	Creating the Search Problem . . . . .	66
A.2	Loading a Predefined Search Problem . . . . .	67
A.3	Editing an Existing Search Problem . . . . .	68
A.4	Carrying Out The Search . . . . .	69
A.5	Analysing the Search Results . . . . .	69
A.6	Analysing the Search Results . . . . .	70
A.7	Referenced Figures . . . . .	71

<b>B</b>	<b>Suitability Measure Definitions</b>	<b>78</b>
B.1	Simple Suitability Measure . . . . .	78
B.2	Phase Aware Suitability Measure . . . . .	78
<b>C</b>	<b>Experiment Data</b>	<b>79</b>
C.1	Deutsch Experiment . . . . .	79
C.2	Deutsch Jozsa Experiment . . . . .	80
C.3	0..3 Max Permutation Problem . . . . .	83
C.4	Quantum Fourier Transform Experiment . . . . .	86
<b>D</b>	<b>Libraries Used</b>	<b>91</b>
D.1	Software . . . . .	91
D.2	Report . . . . .	91
<b>E</b>	<b>Architechture Diagram</b>	<b>93</b>
<b>F</b>	<b>XML Outlines</b>	<b>95</b>
F.1	Search Engine XML Outline . . . . .	95
F.2	Search Problem Definition XML Outline . . . . .	95
F.3	Suitability Measure Definition XML Outline . . . . .	95
F.4	Matrix Definition XML Outline . . . . .	95
F.5	Test Suit Definition XML Outline . . . . .	95
<b>G</b>	<b>Available Algorithm Instructions</b>	<b>96</b>

## List of Figures

2.1	The 1-Qubit Bloch Sphere [1]	3
2.2	Controlled Not Gate	6
2.3	Single Qubit Gate, 2 Qubit Circuit	6
2.4	Computational Artefacts	8
2.5	Deutsch Circuit	9
2.6	Deutsch-Jozsa Circuit	9
2.7	Grover's Search Algorithm	10
2.8	The Components of Shor's algorithm	11
2.9	The Quantum Teleportation Circuit[2]	12
2.10	Q-Pace III	15
4.1	Pauli-I Gate	24
4.2	Test Suite Structures	25
4.3	XML for Suitability Measures Manager Configuration	26
4.4	Manager - Tag Class Diagram	26
4.5	XML for Problem Manager Configuration	27
4.6	Quantum Instruction Structure	28
4.7	Visual Representation of Direct Vector Manipulation Equivalent of Pauli-X Operation	30
4.8	Main User Interface - After Search	40
4.9	Main User Interface - Before Search	42
4.10	Accurate State Readout	43
4.11	Step-By-Step Evaluation Dialog	44
5.1	Expnode Test Expressions	47
6.1	Matrices of Custom Gates Implementing the four Possible Functions	53
6.2	Matrix Circuit	54
6.3	Evolved Deutsch Solution	54
6.4	Deutsch-Jozsa Solutions	55
6.5	o..3 Max Permutation Solution	56
6.6	Evolved Quantum Fourier Transform Solution	57
A.1	Max Problem Test Cases	67
A.2	Initial State of the Client GUI	71
A.3	Right hand menu	72
A.4	Create Problem and Test Suite dialog box	72
A.5	Full Create Problem and Test Suite dialog box	73
A.6	Load Test Suite to Create Problem dialog box	73
A.7	Edit Current Problem and Test Suite dialog box	74
A.8	Warning Produced by Pressing Okay	74
A.9	Search Progress Statistics	75
A.10	Client GUI after Search is Complete - Result 0	75
A.11	Client GUI after Search is Complete - Result 1	76
A.12	Client GUI after Search is Complete - Result 2	76
A.13	Client GUI Showing the Search Statistics	77
A.14	Client GUI Showing the Search Statistics of Entanglement Search	77
C.1	Matricies of Custom Gates Implementing the Eight Possible Functions	80
C.2	Search Parameters used for Deutsch Jozsa Search	81
C.3	Deutsch Jozsa Solution - Two Qubits	82
C.4	Deutsch Jozsa Solution - Three Qubits	82
C.5	Deutsch Jozsa Solution - Four Qubits	82
C.6	Search Parameters Used for o..3 Max Permutation Problem	83
C.7	Test Cases for Two Qubits	86

C.8 Test Cases for Two Qubits . . . . . 86

C.9 Algorithm Instructions used for Quantum Fourier Transform Search . . . . . 87

C.10 Evolved Quantum Fourier Transform Solution . . . . . 90



## List of Tables

2.1	Classical CNOT Truth Table . . . . .	6
2.2	Quantum Teleportation Protocol State Evolution . . . . .	12
3.2	Supported Gates and Definitions . . . . .	20
4.1	Expnode Context Free Grammar . . . . .	28
4.2	Gate Implementation . . . . .	31



# 1 Introduction

In 1980, Richard Feynman noted “it is impossible to represent the results of quantum mechanics with a classical universal device”[3]. This statement was a seed for interest in the field of Quantum Computation, which uses properties of quantum mechanics to perform computation. The true power of quantum computation was not initially realised. The discovery of a quantum artefact by David Deutsch[4] in 1985 that would perform better than an algorithm running on a classical computer was the first glimpse of the potential power provided by harnessing quantum mechanics. The algorithm was able to distinguish between balanced and constant binary functions. A balanced binary function evaluates to 0 for the same number of input values as it evaluates to 1. The algorithm categorised simple functions with inputs limited to 0 and 1 and the advantage over its classical equivalent was the number of times the function needed to be evaluated. The classical equivalent requires two evaluations, once for both possible inputs, yet the quantum algorithm proposed by Deutsch performs a single evaluation. Deutsch’s algorithm is explained in detail in Section 2.2.1.

However, the following years saw the development of only a few new quantum algorithms. The excitement and energy of initial research appeared to be faltering. It would take a discovery by Peter Shor[5] to reignite the excitement surrounding the subject. Shor’s discovery challenged one of the foundations of many encryption techniques currently used. Many cryptographic techniques are based on the belief that factorisation of a large number into its constituent primes would take such a long time that the encrypted information would be useless by the time the factorisation was complete. Shor’s algorithm challenged that belief and provided a polynomial time solution to the factorisation problem using quantum computation. Shor’s algorithm is explained in Section 2.2.4.

The discovery of new quantum algorithms seems difficult. In many ways we may not currently have the intuition needed to easily identify the opportunities presented by quantum mechanics. Some have speculated that evolutionary inspired search techniques might provide a means of discovery that would overcome the lack of human intuition. These search techniques take inspiration from biological understanding. These techniques are not limited to research into quantum algorithms. A variety of problems have been attacked by evolutionary approaches and they have been shown, in some instances, to be human competitive. These range from optimising the set up of a simulated Formula 1 car[6] to the design of a communications antenna[7] by NASA.

This project provides a usable toolkit for researchers using heuristic search techniques to design quantum artefacts. This report records the software engineering process undertaken from requirements through to testing. The report also contains experimental details and results for a number of preliminary experiments that were undertaken after the toolkit was implemented. The software artefact produced by the project has been made publicly available.

## 1.1 Statement of Ethical Implications

This project is not seen to directly result in any significant ethical implications. It is however possible that a result of a successful search could produce an artefact that could have ethical implications. The exact nature of such an artefact cannot be predicted. The onus is on the researchers to act responsibly with any solution found using this toolkit as it is if that researcher were to use any other tool.

The publication of any artefact found by this toolkit must also be produced with professional ethics. This includes correct citation to this toolkit. Any publication must not imply the artefact was manually conceived but must outline the use of this toolkit to maintain academic integrity.

## 1.2 Report Structure

This report has the following structure:

- **Literature Review - Section 2** presents an introduction to quantum computing and previous research using heuristic search to find quantum algorithms. The introduction to quantum computing contains all the information required to understand the remainder of this report. It also introduces several well known quantum algorithms. The heuristic search techniques used in the research presented are evolutionary approaches. A brief overview of these techniques is also provided.
- **Requirements - Section 3** outlines the requirements of the toolkit.
- **Design and Implementation - Section 4** details design and implementation decisions made.

- **Testing - Section 5** provides information on the techniques used during the testing phase. Individual test cases are not provided but higher level test aims are given.
- **Experimentation - Section 6** outlines a number of experiments performed using the implemented toolkit. The experiments performed were chosen to allow comparison with the research reviewed in Section 2.
- **Evaluation and Future Work - Section 7** concludes this report with an evaluation of both the software artefact produced and the results of the experimentation phase. As part of this evaluation functionality that could be added to the toolkit and areas that could be improved are discussed.

## 2 Literature Review

This chapter provides a brief introduction to quantum computation. This is followed by explanations of several quantum artefacts and a summary of previous research into the discovery of quantum artefacts by heuristic search. The chapter concludes with a statement of intent for this project.

### 2.1 Introduction to Quantum Computation

In classical computers, computation is performed using the discrete values of 0 and 1. These values are typically indicated by +5V and 0V signals propagating round circuits. A signal can only be interpreted as 0 or 1, there is no in-between value. Each signal can indicate the value of a single bit of data. A combination of  $n$  bits can be used to represent a number from 0 to  $2^n - 1$ , an  $n$  bit number. Classical computation works through the manipulation of these  $n$  bits.

Qubits are the quantum equivalent of the classical bit. Like the classical bit, a qubit can be in one of two states,  $|0\rangle$  or  $|1\rangle$ . The notation used to specify quantum states is the Dirac[8] “Bra-Ket” notation.

A Ket is mathematical notation,  $|a\rangle$ , that represents a basis of an  $n$  dimensional vector space. It can be interpreted, as a column vector as shown in Equation 2.1.

$$|a\rangle = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{pmatrix} \quad (2.1)$$

A quantum state can be visualised as a point of the Bloch sphere, shown in Figure 2.1, that represents the quantum state space. The shown Bloch sphere is for a single qubit, the principle can be extended to  $n$  qubits however the visualisation breaks down. All quantum states can be described using the Bloch sphere. The states that this project is concerned with are those that lie on the surface of the respective Bloch sphere, known as “pure” states.

In quantum mechanics, a Ket is used to indicate a state, for example  $|0\rangle$  is the state equivalent to logical 0 whereas  $|1\rangle$  is the state equivalent to logical 1.

A dual to the Ket notation is the Bra notation,  $\langle a|$ . This notation is used to denote the “dual vector” of the corresponding Ket. For a state vector represented by Equation 2.2 there is a dual vector representing its Hermitian Conjugate, Equation 2.3. The Hermitian Conjugate of a vector is the transpose of the vector

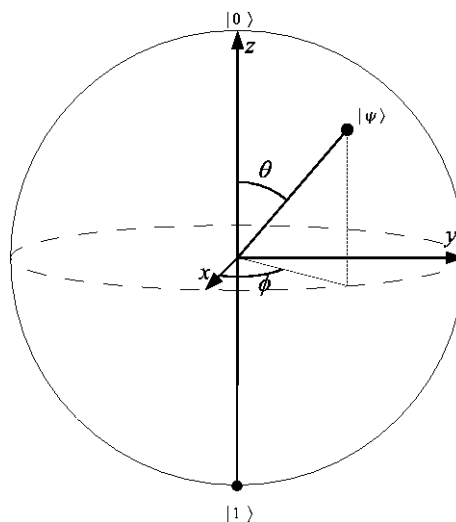


Figure 2.1: The 1-Qubit Bloch Sphere [1]

after each entry is replaced by its complex conjugate.

$$|a\rangle = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \\ \vdots \\ a_n \end{pmatrix} \quad (2.2)$$

$$\langle a| = (a_1^* \ a_2^* \ a_3^* \ \cdots \ a_n^*) \quad (2.3)$$

Combining the two vectors  $\langle a|$  and  $|b\rangle$ , written  $\langle a|b\rangle$ , represents the inner product of the two vectors,  $|a\rangle$  and  $|b\rangle$ . If  $a$  and  $b$  are unit vectors and  $a = b$ ,  $\langle a|b\rangle = 1$ . If  $a$  and  $b$  are orthogonal,  $\langle a|b\rangle = 0$ .

The outer product of two vectors,  $|a\rangle$  and  $|b\rangle$ , can be represented by  $|a\rangle\langle b|$ . This represents the transformation from  $|a\rangle$  to  $|b\rangle$ . It can also be represented in matrix form.

With  $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$  and  $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$  it is possible to represent single qubit operations in the “Bra-Ket” notation. For example, the NOT gate performs a simple negation of a qubit’s value. This can be written as  $|0\rangle\langle 1| + |1\rangle\langle 0|$ . Substituting in the vector values we have:

$$\begin{aligned} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \end{pmatrix} &= \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \end{aligned} \quad (2.4)$$

This matrix can be seen as a transformation matrix for the NOT operation. In quantum computation, the NOT gate is one of the 4 gates known as the Pauli gates, more specifically it is the Pauli-X gate. It is called the Pauli-X gate as it can be seen as a rotation of  $\pi$  radians about the X axis of the Bloch sphere, Figure 2.1. The Pauli gates are named after Wolfgang Pauli who published the matrices that define them.

The matrices representing the Pauli-X gate and all other quantum logic gates are *unitary*. A unitary matrix,  $U$ , is one which adheres to Equation 2.5 where  $I_n$  is the identity matrix in  $n$  dimensions and  $U^\dagger$  is the Hermitian Conjugate of  $U$ .

$$U^\dagger U = UU^\dagger = I_n \quad (2.5)$$

Unitary transformations are physically reversible. Some classical gates are not. For example, the AND operation loses information about the inputs and can be represented as the function  $AND(a, b) \rightarrow a \wedge b$ . However, for practical purposes it is possible to produce physically reversible versions by incorporating supplementary information in the output. For the AND operation both inputs have to be provided in the output so the function becomes  $AND(a, b) \rightarrow (a, b, a \wedge b)$ .

All the details of quantum computation introduced so far do not offer anything more than classical computation. The power of quantum computation comes from the use of superposition. Superposition is the ability of a qubit to be in both the  $|0\rangle$  and  $|1\rangle$  states simultaneously with certain probabilities. It is not possible to observe the superposition of a qubit. When observed the superposition “collapses” to either  $|0\rangle$  or  $|1\rangle$ , the computational basis states. The probability of the superposition collapsing to a specific state is determined by the superposition’s complex amplitudes.

$$\alpha|0\rangle + \beta|1\rangle \quad (2.6)$$

The notation for the superposition of states that a qubit is currently in is shown in Equation 2.6.  $\alpha$  and  $\beta$  are the probability amplitudes for the respective states and are expressed as complex numbers. The complex probability amplitudes have phases that can interfere with each other through computation. However, only relative phase factors can be observed. The probability of the state in Equation 2.6 collapsing to the basis state  $|0\rangle$  is given by  $|\alpha|^2$ . Similarly, the probability of this state collapsing to the basis state  $|1\rangle$  is  $|\beta|^2$ . The overall probability of a superposition collapsing to any of the states it contains must equal 1. For the basis state  $|0\rangle$   $\alpha = 1$  and  $\beta = 0$  whereas for the basis state  $|1\rangle$   $\alpha = 0$  and  $\beta = 1$ . Any state where  $\alpha$  and  $\beta$  do not equal 1 and 0 is in superposition.

The state  $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$  is an equal superposition where the collapse under observation to  $|0\rangle$  is just as likely as collapsing to  $|1\rangle$ , probabilities of  $|\frac{1}{\sqrt{2}}|^2 = \frac{1}{2}$ . With  $n$  qubits in equal superposition, there are  $n$  binary values that have an equal probability of taking the value 0 as the value 1. With an ordering decided of these qubits, collapsing the superposition of each qubit will result in a binary value of length  $n$ . With the probabilities of each qubit being  $\frac{1}{2}$  this binary value takes a truly random value between 0 and  $2^n - 1$ . It is not possible to produce a truly random number using a classical computer. Equation 2.7 is a generalisation for the equal superposition state over  $n$  qubits.

$$\frac{1}{\sqrt{2}} \sum_{i=0}^N |x_i\rangle \quad (2.7)$$

In 1935, Erwin Schrödinger[9] proposed a thought experiment to explain the idea of superposition. Imagine a cat in a fully opaque box with a vial of poison. The vial may break at any time, a truly random variable. After sealing the box the liveness state of the cat is not known. The cat could be alive if the vial has not broken but could just as likely be dead. Only by looking inside the box will the state of the cat be known. Until this time the cat could be thought of as both alive and dead at the same time. If we assign “dead” to the state  $|0\rangle$  and “alive” to the state  $|1\rangle$  the situation looks very similar to the state in Equation 2.6 where both  $\alpha$  and  $\beta$  equal  $\frac{1}{\sqrt{2}}$ . Therefore, just as the cat can be thought of as both dead and alive at the same time, a qubit in the superposition  $\frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$  can be thought of both in the  $|0\rangle$  and  $|1\rangle$  states at the same time. This leads to a very powerful property of quantum computers. With  $n$  classical bits, a single number in the range 0 to  $2^n - 1$  can be expressed at any one time. However with  $n$  quantum qubits, every number in the range 0 to  $2^n - 1$  can be expressed at the same time.

Quantum mechanics is said to be linear and unitary operations can distribute over linear sums. As a quantum state can be expressed as the linear sum in Equation 2.7 the computation over all  $2^n$  inputs to be performed in parallel. This is represented by Equation 2.8.

$$U \sum_{x=0}^{2^n-1} |x\rangle \equiv \sum_{x=0}^{2^n-1} U |x\rangle \quad (2.8)$$

This parallelism is very powerful and has been shown to enable the computation of problems classified as NP to be performed in polynomial time. For example the factorisation algorithm discovered by Peter Shor[5]. This does however have a caveat. As mentioned previously the superposition cannot itself be observed. When observation is attempted, the superposition collapses to a basis state with respect to the superposition probability amplitudes. This means that even though  $2^n$  calculations can be performed in parallel, only a single answer can be observed.

Non-measurement operations that can be performed on quantum states are unitary and the Pauli-X operation is just one such operation. Along with the Pauli-X gate, there are an additional 3 Pauli gates. The Pauli-I gate is the simplest of all quantum gates. It is the identity gate, the output is identical to the input. In Dirac notation this is  $|0\rangle\langle 0| + |1\rangle\langle 1|$ , and Equation 2.9 shows the matrix form.

$$\begin{aligned} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \end{pmatrix} &= \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \end{aligned} \quad (2.9)$$

The Pauli-Z gate rotates the quantum state by  $\pi$  radians about the Z axis. This represents a phase flip of the quantum state. The phase of a state is important when interference is used in computation. In Dirac notation this is  $|0\rangle\langle 0| + |1\rangle\langle -1|$ , and Equation 2.10 shows the matrix form.

$$\begin{aligned} \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 1 & 0 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} 0 & -1 \end{pmatrix} &= \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ 0 & -1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \end{aligned} \quad (2.10)$$

The Pauli-Y gate is similar to both the Pauli-X and Pauli-Z gates, but differs in the axis about which it performs the rotation. The Pauli-Y gate rotates the quantum state by  $\pi$  radians about the Y axis. In Dirac

Control	Target Before	Target After
$ 0\rangle$	$ 0\rangle$	$ 0\rangle$
$ 0\rangle$	$ 1\rangle$	$ 1\rangle$
$ 1\rangle$	$ 0\rangle$	$ 1\rangle$
$ 1\rangle$	$ 1\rangle$	$ 0\rangle$

Table 2.1: Classical CNOT Truth Table



Figure 2.2: Controlled Not Gate

notation this is  $|0\rangle (i \langle 1|) + |1\rangle (-i \langle 0|)$ , and Equation 2.11 shows the matrix form.

$$\begin{aligned}
 \begin{pmatrix} 1 \\ 0 \end{pmatrix} \begin{pmatrix} 0 & -i \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \end{pmatrix} \begin{pmatrix} i & 0 \end{pmatrix} &= \begin{pmatrix} 0 & -i \\ 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 0 \\ i & 0 \end{pmatrix} \\
 &= \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}
 \end{aligned} \tag{2.11}$$

Along with the single qubit operations, like those above, there are operations which can act over  $n$  qubits. A simple example of a 2 qubit operation is the Controlled-NOT(CNOT) operator. This is a simple extension of the Pauli-X gate.

The CNOT gate has a control qubit which is required to be in the state  $|1\rangle$  for the NOT operation on the target qubit to be carried out. In classical logic this would extend the truth table to be as shown in Table 2.1. The truth table of the CNOT gate is the same as that of the classical XOR gate. The Dirac notation of the CNOT gate is  $|00\rangle \langle 00| + |01\rangle \langle 01| + |10\rangle \langle 11| + |11\rangle \langle 10|$ .

Circuit representations of the CNOT gate are shown in Figure 2.2. Both the circuit representations are equivalent with the upper qubit as the control qubit with the lower qubit as the target. The circuit in Figure 2.2(b) shows the widely used shorthand for the CNOT gate with  $\text{---}\oplus\text{---}$  representing the Pauli-X operation.

For a 2 qubit circuit, the state has 4 elements in the state vector. This means that to apply a single qubit gate to this state, multiply the state by the respective  $2 \times 2$  unitary matrix, there will be a problem due to the matrix dimensions not matching. The matrix of the single qubit gate has to be expanded to a  $4 \times 4$  matrix so that it can be applied to the state vector. Taking the circuit shown in Figure 2.3(a) as an example we can see that it is a simple Pauli-X gate, bit-flip operator, acting on the higher order qubit while nothing occurs to the lower order qubit. Nothing happening to the lower order qubit is the same as applying the Pauli-I gate, identity operator, to this qubit. The circuit in Figure 2.3(b) shows the resulting circuit with identical effect on all possible input states. This interpretation of the circuit still doesn't provide a solution as there are separate  $2 \times 2$  matrices for each of the two gates. These can't be applied to the state individually for that same reason as before.

To produce a  $4 \times 4$  matrix for the circuit, the two  $2 \times 2$  matrices have to be combined. The matrix operation that is used for this is the Tensor Product. The tensor product is a simple matrix operation that is denoted using the  $\otimes$  symbol. The result of the tensor product operation is shown in Equation 2.12.

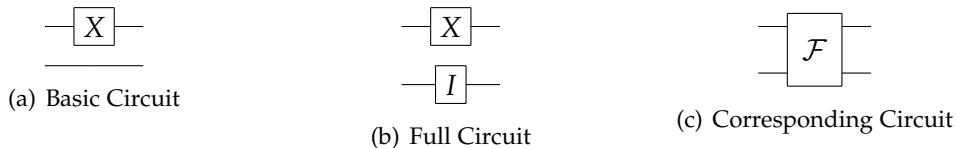


Figure 2.3: Single Qubit Gate, 2 Qubit Circuit



$$\begin{aligned}
A \otimes B &= \begin{pmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{pmatrix} \otimes \begin{pmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{pmatrix} = \begin{pmatrix} a_{00}B & a_{01}B \\ a_{10}B & a_{11}B \end{pmatrix} \\
&= \begin{pmatrix} a_{00}b_{00} & a_{00}b_{01} & a_{01}b_{00} & a_{01}b_{01} \\ a_{00}b_{10} & a_{00}b_{11} & a_{01}b_{10} & a_{01}b_{11} \\ a_{10}b_{00} & a_{10}b_{01} & a_{11}b_{00} & a_{11}b_{01} \\ a_{10}b_{10} & a_{10}b_{11} & a_{11}b_{10} & a_{11}b_{11} \end{pmatrix} \quad (2.12)
\end{aligned}$$

So when applying this to the matrices of Pauli-X as A and the Pauli-I as B we get the unitary matrix for the corresponding  $\mathcal{F}$  gate, shown in Figure 2.3(c). This can be seen in Equation 2.13.

$$\begin{aligned}
\mathcal{F} = X \otimes I &= \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \otimes \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 0I & 1I \\ 1I & 0I \end{pmatrix} \\
&= \begin{pmatrix} 0 \times 1 & 0 \times 0 & 1 \times 1 & 1 \times 0 \\ 0 \times 0 & 0 \times 1 & 1 \times 0 & 1 \times 1 \\ 1 \times 1 & 1 \times 0 & 0 \times 1 & 0 \times 0 \\ 1 \times 0 & 1 \times 1 & 0 \times 0 & 0 \times 1 \end{pmatrix} \\
&= \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} \quad (2.13)
\end{aligned}$$

Tensor product scales to any number of qubits and can therefore produce  $2^n \times 2^n$  matrices for systems involving  $n$  qubits. The tensor product cannot however be used in this same way to produce unitary matrices for controlled gates. It can only be used when the action on each qubit is independent.

More than two matrices be combined by the tensor product operation,  $A \otimes B \otimes C$ . The operation is broken down from right to left into a number of two qubit operations,  $A \otimes (B \otimes C)$ . The ordering of the matrices is important. Assuming  $A$ ,  $B$  and  $C$  are single qubit quantum operations the operation encoded by  $A$  would operate on the highest significant qubit while the operation encoded by  $C$  would operate on the lowest significant qubit. The operation encoded by  $B$  would operate on the middle qubit.

Not all multiple qubit states can be expressed as the tensor products of individual states. Such states are said to be entangled. An example with two qubits is the state  $\frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ . This state cannot be factored into two quantum states, one for each qubit. This is also a superposition of two states and under observation the superposition will collapse to one of  $|00\rangle$  or  $|11\rangle$ . Thus when one of the qubits is observed and is seen to be in state  $|0\rangle$  we can deduce that that the other qubit is also in the state  $|0\rangle$ . Similarly, if the qubits were observed to be in the state  $|1\rangle$  we can deduce that that the other qubit is also in the state  $|1\rangle$ . It is important to note that this property is maintained regardless of spacial separation.

## 2.2 An Introduction to Quantum Circuits and Algorithms

Quantum computational artefacts can be considered at various levels of abstraction. At the implementation level computation is the application of a sequence of specific operations. This sequence of operations shall be referred to as a circuit. A circuit generally operates over a quantum system of a fixed size, referred to as the system size. For example a circuit could invert, by applying the Pauli-X operation, each qubit of a three qubit state. A program is an artefact that when evaluated produces a circuit. An algorithm is an artefact that when instantiated with the system size produces a program. This relationship can be seen in Figure 2.4.

This gives three artefacts, each at a different level of abstraction that can be designed to specify a solution to a problem.

- A circuit
- A program
- An algorithm

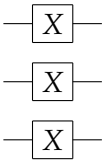
Circuit Artefact	Algorithm 1 Program Artefact	Algorithm 2 Algorithm Artefact
	<pre> <b>for</b> <math>i : 1 \rightarrow 3 : i++</math> <b>do</b>   <i>Pauli</i> – <math>X(i)</math> <b>end for</b> </pre>	<pre> <b>for</b> <math>i : 1 \rightarrow System\_Size : i++</math> <b>do</b>   <i>Pauli</i> – <math>X(i)</math> <b>end for</b> </pre>

Figure 2.4: Computational Artefacts

The same is true for quantum artefact design. The design can either be done at the circuit level, program level or parametrisable algorithm level.

Currently there are very few problems for which a quantum solution is known. Peter Shor published a discussion on the progress made “in discovering algorithms for computation on a quantum computer”[10]. Shor suggests two possible reasons for the lack of quantum algorithms. The first was “that there might really be only a few problems for which quantum computers can offer a substantial speed-up over classical computers”[10]. This would indeed make the discovery of useful quantum algorithms difficult. However, I feel this is rather pessimistic. The main focus of the paper published by Feynman[3] was the problem of simulating the physics of quantum mechanics on a classical computer. This suggests there the potential of many applications for quantum computers, even if they aren’t analogous to classical computational applications.

The second reason was “that quantum computers operate in a manner so non-intuitive, and so different from classical computers”[10] that our current algorithm knowledge is close to useless. This, in my opinion, is a much more significant obstacle. Quantum mechanics is seen by many as a confusing and mystical subject. Prize winning mathematician and physicist Roger Penrose is attributed to the remark “Quantum mechanics makes absolutely no sense”. Statements such as this do nothing to promote the universal study of quantum mechanics. If quantum mechanics and other subjects, such as general relativity, were, even in part, to filter into curricula I feel general understanding would improve, leading to greater intuition. Without this it may simply be unreasonable to expect discoveries that exploit the finer details of these complex and subtle theories to become frequent occurrences. I think this applies to many areas of research including quantum computation.

Research into the use of heuristic search techniques to assist in the design of quantum algorithms has shown promise. Several different techniques have been used to varying success. Not only do the techniques differ, but so do the target artefacts. Research has been presented to design at each of the three levels of abstraction outlined above.

In the following sections several quantum algorithms shall be explained in detail. This is followed by an overview of the research that has been carried out using heuristic search to perform quantum algorithm synthesis.

### 2.2.1 Deutsch Algorithms

In 1985, Deutsch[4] proposed a probabilistic solution to a simple promise problem. The solution was improved to a deterministic solution by Cleve et al.[11]. Despite the improvement being produced by Cleve et al. the algorithm is popularly known as the Deutsch algorithm.

The Deutsch algorithm is intended to solve the following problem:

Given a function  $f : \{0,1\} \rightarrow \{0,1\}$ , state whether it is either a balanced, or constant function.

The “promise” is that the function  $f$  is guaranteed to be either balanced or constant.

The algorithm’s major breakthrough was that it only required a single invocation of the function  $f$  to decide in which category the function belongs. The best classical approach requires two invocations. This was the first algorithm to exploit quantum mechanics to compute the solution to a problem provably more efficiently, with efficiency defined as the number of evaluations of  $f$ , than a classical solution.

The circuit for this algorithm is presented in Figure 2.5. Despite this being known as the Deutsch Algorithm, the solution is a circuit rather than a parametrisable list of instructions. The mathematical

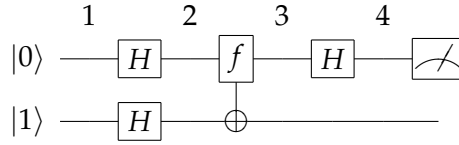


Figure 2.5: Deutsch Circuit

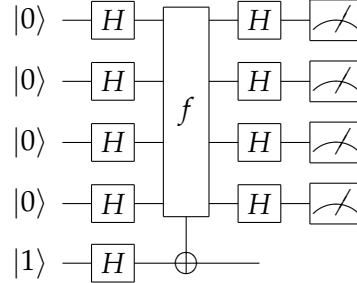


Figure 2.6: Deutsch-Jozsa Circuit

reasoning does not directly scale to functions with greater than a single input qubit. Between points 2 and 3 in Figure 2.5 is a modified Controlled-Not gate. The standard Controlled-Not gate uses the value of the control qubit to decide whether to perform the Pauli-X operation. This modified gate does not use the control qubit value,  $x$ , but the result of the  $f(x)$  to decide whether to perform the Pauli-X operation. However, as with the standard Controlled-Not gate, this modified version does not affect the value of the qubits used by the control function. The circuit contains the symbol  $\text{---} \boxed{\text{meter}} \text{---}$  which represents an observation.

It can be shown that for a balanced function the observed state is  $|1\rangle$  and that for a constant function the observed state is  $|0\rangle$ . This works as the modified Controlled-Not gate acts on the lower qubit in a superposition with non-zero relative phases which results in the action of the modified Controlled-Not gate affecting the upper qubit. For the mathematics please refer to Lectures 11 and 12 of QIP[12].

This algorithm reinforces the statement made by Shor[10] regarding the lack of background knowledge that can be drawn upon to create these algorithms. The construction of such an algorithm cannot be produced simply by understanding the concepts of classical algorithm design.

A notable feature of this algorithm is that the observation is made on the control qubit of the modified Controlled-Not gate. If the circuit between points 3 and 4 in Figure 2.5 were analysed by hand without the upper qubit in superposition the upper qubit would not be affected. However, when the upper qubit is placed into superposition it is affected by the modified Controlled-Not gate. This is a prime example where the solution requires the understanding of the subtle implications of complex probability amplitudes and superposition, and the vision to exploit it.

### 2.2.2 Deutsch-Jozsa Algorithm

The Deutsch-Jozsa algorithm[13] is a generalisation and improvement over the earlier Deutsch algorithm[4]. The algorithm was improved by Cleve et al.[11]. The improved algorithm is still generally referred to as the Deutsch-Jozsa algorithm.

In 1992 David Deutsch and Richard Jozsa[13] presented an extension to the Deutsch algorithm, Section 2.2.1, that allowed for functions  $f : \{0,1\}^{2^n} \rightarrow \{0,1\}$  to be categorised as constant or balanced. The algorithm was probabilistic and required 2 invocations to the function  $f$ . This means that in the worst case, with  $n = 1$ , the algorithm will perform worse than both the original Deutsch algorithm[4] and the classical algorithm due to the probabilistic nature of its result. This is a very limited case and performance improves as the value of  $n$  increases. The improvement by Cleve et al. reduces this to a single invocation of the function  $f$  and is deterministic. As with the original Deutsch algorithm, the number of invocations of  $f$  is constant, truly independent of both  $n$  and  $f$ . This is again an improvement over the classical algorithm which requires in the worst case  $2^{n-1} + 1$  to be certain of the function's classification.

The circuit that is produced by this algorithm is shown in Figure 2.6. The algorithm requires  $n$  input

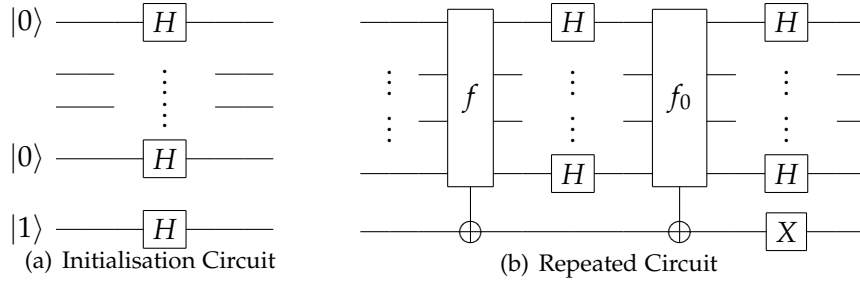


Figure 2.7: Grover's Search Algorithm

qubits and a single ancillary qubit. The gate labelled  $f$  is analogous to the modified Controlled-Not gate in the Deutsch algorithm.

The algorithm effectively works in the same way as the Deutsch algorithm. However, due to the increase in  $n$  some of the mathematical reasoning has a different result. The only observable difference is that if  $f$  is balanced functions the observation will not always result in the same state. Just as with the Deutsch algorithm it can be shown that if the function  $f$  is constant the observed state will be  $|0\rangle^{\otimes n}$ . However, if the function  $f$  is balanced the observed state will be any state other than  $|0\rangle^{\otimes n}$ , i.e. any state orthogonal to  $|0\rangle^{\otimes n}$ . For the mathematics please refer to Lectures 11 and 12 of QIP[12] or Section B.1 in [14].

The similarities between the circuits of the Deutsch and Deutsch-Jozsa algorithms are quite apparent. As with the Deutsch algorithm, the Deutsch-Jozsa algorithm uses a superposition containing non-zero relative phases to cause an affect on the qubits providing the input to the control function  $f$  despite the modified Controlled-Not gate physically acting on the ancillary qubit.

### 2.2.3 Grover's Search Algorithm

The Grover Search algorithm[15] is a search problem to find a solution  $x$  such that  $f(x) = \text{True}$  for some predicate  $f$ . The algorithm does not exploit or assume any structure, such as sorting, in the data set over which it is searching.

It has previously been proven[16] that the lower complexity limit for any algorithm identifying an element without knowledge of underlying structure in the data is  $\Omega(\sqrt{N})$ .  $N$  represents the number of elements in the search space. It is assumed  $N = 2^n$ . The complexity is measured by the number of elements which need to be queried in order to find the desired element. The Grover Search algorithm has the complexity  $O(\sqrt{N})$  and so "is within a constant factor of the fastest possible quantum mechanical algorithm"[15].

The algorithm works for predicates with one or more solutions, referred to as the correct states. The number of solutions shall be represented by  $M$ . Any state which under observation will with high probability provide one of the  $M$  solutions solves the search problem. The remaining  $N - M$  states are referred to as the incorrect states.

The mechanisms used within the algorithm to produce a solution to the problem are more subtle than those used but the Deutsch-Jozsa algorithm. The algorithm does not perform the computation in a single step. After the initialisation circuit has been applied, see Figure 2.7(a), the circuit shown in Figure 2.7(b) is repeated in  $O(\sqrt{N})$  times. The actual number of times depends on  $N$  and the number of solutions,  $M$ .

What is important about this algorithm is that it works on amplification. Each time the circuit is applied the probability amplitude of the desired solution is increased.

The application of Figure 2.7(b) can be seen as rotating by  $\sqrt{\frac{M}{N}}$  Radians around a Bloch Sphere of the 1 qubit system, see Figure 2.1. The state  $|1\rangle$  of the Bloch Sphere represents the desired states in which only the  $M$  correct states have non-zero probability amplitudes. I.e. an observation made on the  $n$  input qubits would always result in one of the  $M$  correct states and would never result in one of the  $N - M$  incorrect states. The state  $|0\rangle$  of the Bloch Sphere represents the state in which only the  $N - M$  incorrect states have non-zero probability amplitudes. I.e. an observation, made on the  $n$  input qubits would always result in one of the  $N - M$  incorrect states and would never result in one of the  $M$  correct states. After  $\frac{\pi\sqrt{N}}{4\sqrt{M}}$  repetitions the probability of observing the a correct state is  $\sqrt{\frac{N-M}{N}}$ , the same as the probability of

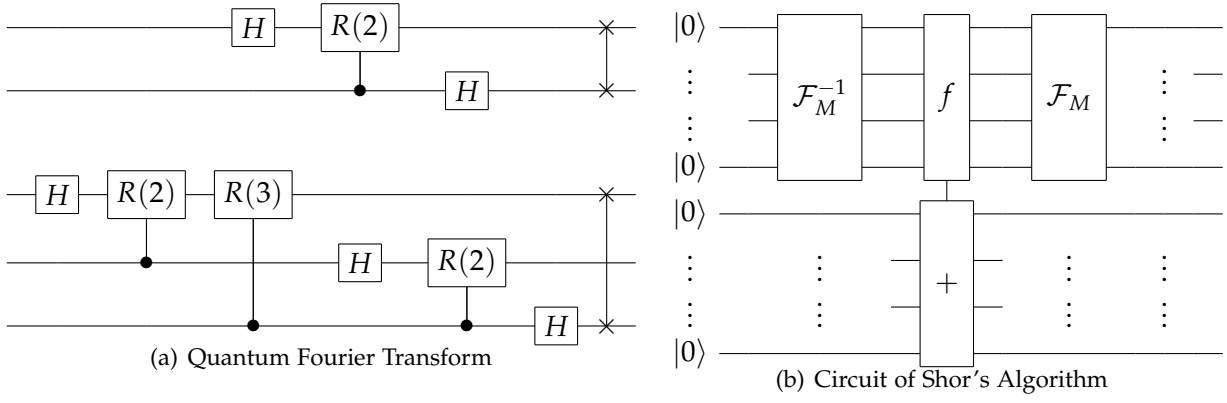


Figure 2.8: The Components of Shor's algorithm

observing an incorrect state after the circuit in Figure 2.7(a) was initially applied. This state is very close to that represented by the  $|1\rangle$  on the Bloch Sphere. The value of  $M$  is required to calculate the number of repetitions to perform. As the Bloch Sphere is spherical the repeated application of a rotation about a fixed axis is cyclical. This means if Figure 2.7(b) is repeated more than  $\frac{\pi\sqrt{N}}{4\sqrt{M}}$  times the state starts rotating away from the state  $|1\rangle$  of the Bloch Sphere and back towards state  $|0\rangle$  of the Bloch Sphere. Therefore, the  $n$  qubit state of Grover's algorithm is periodic.

For the mathematics please refer to Lectures 16, 17 and 18 of QIP[12] or Section 7.1 of [17].

#### 2.2.4 Shor's Factorisation Algorithm

In 1994, Peter Shor astonished, and worried, the computer science community with a quantum factorisation algorithm[5]. This allowed the factorisation of integers into their constituent primes in polynomial time. This algorithm is not a pure quantum algorithm, but a hybrid algorithm. It has both a quantum and classical portion.

The algorithm exploits the phase components of the complex probability amplitudes. The quantum portion of the algorithm, shown in Figure 2.8(b), is more general than is often explained.

The quantum portion is simply a periodicity finding algorithm. The period of certain functions, such as  $a^x \pmod{N}$ , can be used with elements of number theory to find the factors of a number. However, the function  $f$  in the circuit shown in Figure 2.8(b) could be replaced by any periodic function in order to find its period.

For example, the Grovers search algorithm in Section 2.2.3 requires the number of solutions,  $M$ , to be known. If  $M$  is initially unknown, Shor's algorithm can be used to find it. As is explained in Section 2.2.3, Grovers algorithm is periodic. With this period known for a certain instance of Grovers algorithm the value of  $M$  can be calculated as each rotation is  $\sqrt{\frac{M}{N}}$  Radians. This is known as Quantum Counting[18].

Shor's algorithm includes the Quantum Fourier Transform and its inverse,  $\mathcal{F}$  and  $\mathcal{F}^{-1}$ . Circuits that implement the Quantum Fourier Transform for 2 and 3 qubits can be seen in Figure 2.8(a). A pattern can be seen in the two circuits that scales up to  $n$  qubits. The action of applying the Quantum Fourier Transform and the inverse can be seen in Equations 2.14 and 2.15 respectively.

$$\mathcal{F}_N |x\rangle = \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} e^{\frac{2\pi ixy}{N}} |y\rangle \quad (2.14)$$

$$\mathcal{F}_N^{-1} |y\rangle = \frac{1}{\sqrt{N}} \sum_{z=0}^{N-1} e^{-\frac{2\pi ixy}{N}} |z\rangle \quad (2.15)$$

The use of the QFT is important as it exploits the complex nature or the probability amplitudes. This allows algorithms to harness the affects of constructive and destructive interference.

For the mathematics please refer to Lectures 14 and 15 of QIP[12] or Section 6 of [17].

#### 2.2.5 Quantum Teleportation Protocol

The quantum teleportation protocol provides a solution to the problem

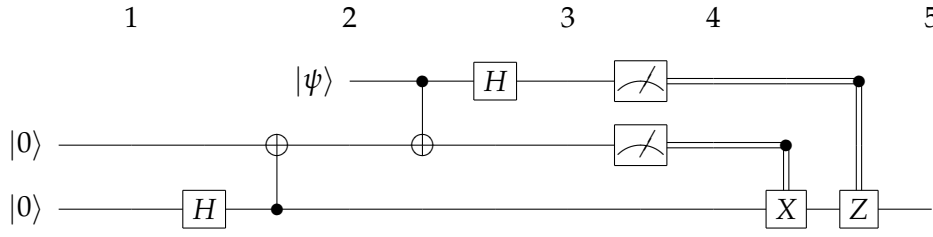


Figure 2.9: The Quantum Teleportation Circuit[2]

Alice's Observations		Bob's Qubit		
Upper Qubit	Lower Qubit	At Label 4	Applied operations	At Label 5
$ 0\rangle$	$ 0\rangle$	$\alpha  0\rangle + \beta  1\rangle$	None	$\alpha  0\rangle + \beta  1\rangle$
$ 1\rangle$	$ 0\rangle$	$\alpha  1\rangle + \beta  0\rangle$	Bit Flip	$\alpha  0\rangle + \beta  1\rangle$
$ 0\rangle$	$ 1\rangle$	$\alpha  0\rangle - \beta  1\rangle$	Phase Flip	$\alpha  0\rangle + \beta  1\rangle$
$ 1\rangle$	$ 1\rangle$	$\alpha  1\rangle - \beta  0\rangle$	Bit Flip followed by a Phase Flip	$\alpha  0\rangle + \beta  1\rangle$

Table 2.2: Quantum Teleportation Protocol State Evolution

Alice has a quantum state,  $|\Psi\rangle$ , to send to Bob. However, Alice and Bob can only communicate using a classical communication channel.

There is no assumption that Alice actually “knows” the quantum state  $|\Psi\rangle = \alpha |0\rangle + \beta |1\rangle$ , that is she may possess the physical qubit but be unaware of the actual values of  $\alpha$  and  $\beta$ . Furthermore, measurement would cause the qubit to collapse to  $|0\rangle$  or  $|1\rangle$ . We cannot “look up” and send the value of the state as we might do for a classically stored bit of information. However, quantum mechanics provides a solution.

The circuit implementing the solution can be seen in Figure 2.9. The numbers above the circuit will be used in the following explanation.

For Alice to send the state  $|\Psi\rangle = \alpha |0\rangle + \beta |1\rangle$  to Bob they must share the entangled state  $|\Phi\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$ . The state  $|\Phi\rangle$  is created between labels 1 and 2. This requires a two qubit operation, which requires physical proximity of the qubits. Once the qubits are entangled, the lowest significant, bottom, qubit is transported to Bob. Now the teleportation part of the algorithm is carried out.  $|\Psi\rangle = \alpha |0\rangle + \beta |1\rangle$  may now be created or may have existed prior to the entanglement process. The state at label 2 is shown in Equation 2.16. The subscript letters show the qubits are held by Alice and the qubit held by Bob.

$$\alpha |0\rangle + \beta |1\rangle \left( \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle) \right) = \frac{1}{\sqrt{2}}(\alpha |00\rangle_A |0\rangle_B + \beta |10\rangle_A |0\rangle_B + \alpha |01\rangle_A |1\rangle_B + \beta |11\rangle_A |1\rangle_B) \quad (2.16)$$

The operations between labels 2 and 3 produce the state in Equation 2.17.

$$\begin{aligned} & \frac{1}{2}(\alpha |00\rangle_A |0\rangle_B + \alpha |10\rangle_A |0\rangle_B + \beta |01\rangle_A |0\rangle_B - \beta |11\rangle_A |0\rangle_B + \\ & \alpha |01\rangle_A |1\rangle_B + \alpha |11\rangle_A |1\rangle_B + \beta |00\rangle_A |1\rangle_B - \beta |10\rangle_A |1\rangle_B) \end{aligned} \quad (2.17)$$

Between labels 3 and 4, Alice will observe one of four possible states  $|00\rangle$ ,  $|01\rangle$ ,  $|10\rangle$  or  $|11\rangle$ . Alice can communicate this observed state to Bob using the classical communication channel. The observed state indicates which variant of  $|\Psi\rangle$  is the state of Bob's qubit. Table 2.2 shows the relationship between the observation made by Alice and the state of Bob's qubit. With the state observed by Alice, Bob can use Table 2.2 to find out what operations need to be applied to recover the original state  $|\Psi\rangle$ . Table 2.2 shows how the state observed by Alice affects Bob's qubit and how the state  $|\Psi\rangle$  can be recovered.

At label 5 Bob is in possession of the original state  $|\Psi\rangle$ .

### 2.3 The Use of Evolutionary Computation in the Synthesis of Quantum Algorithms

Nature inspired computation is a highly active research area. Taking inspiration from nature and biological theories, search techniques such as Genetic Algorithms and Genetic Programming are being

employed to a wide range of industrial problems. What makes these approaches different is that they are based on a population, or “generation”, of individuals. The basic principle is to evaluate the “suitability” of all individuals in the generation, select the individuals with probabilities based on their suitability. Mutate each selected individual with a fixed probability and add it to the next ‘generation’ of individuals. Alongside mutation, a computational analogy to biology’s reproduction, called crossover, can be used. Crossover takes two, or potentially more, selected individuals and combines them to produce other individuals which are then added to the next ‘generation’. The each cycle of evaluate, selection and mutation and/or crossover produces a ‘generation’ of individuals. The process repeats until a required suitability is found or a resource limit is reached, time or number of generations produced for example. As the process progresses, with the a reasonable representation and suitability measure, the average suitability of each generation should improve.

The use of evolutionary techniques to try synthesize quantum algorithms is not new. There are many examples of successes in producing solutions to problems already solved by a manual approach and some producing novel solutions to problems without previously known quantum solutions. The techniques used vary from Genetic Algorithms to Genetic Programming with varying success.

Not only are the techniques varied, the representation of the desired solutions are also varied. Some research focuses on the evolution of quantum circuits or programs, whereas some focus on more general quantum algorithms. As the system size increases the resources required for simulation increase exponentially. As a result quantum algorithms are not usually tested on large system sizes.

### 2.3.1 Q-PACE I

Massey[19, 20] explores both Genetic Algorithms and Genetic Programming as search techniques. The software suites presented, Q-PACE I - IV, have varying success rate and each version provides additional expressive power over its predecessors. Q-PACE II[20] is described as solving “a number of basic proof of concept problems”[19] and “proves the concept that evolutionary search techniques can be used to evolve quantum software”[19]. Q-PACE I uses a fixed length array of quantum gates and is based on the simple Genetic Algorithm found in [21].

This representation is fairly limiting. If a solution would require  $x$  gates but the fixed length array is  $< x$  the solution cannot be found. Therefore the array either has to be very large or the number of gates required needs to be known. This is clearly not desirable as increasing the array length increases the search space.

### 2.3.2 Q-PACE II

Q-PACE II[19] is a suite based on Q-PACE I but uses Genetic Programming instead. This allows Q-PACE II to handle variable length solutions. Individuals are represented as lists of quantum gates parametrized with the label representing the type of gate, target and control qubits and phase factor.

Unitary operators, gate implementations, are just bijective functions acting on state vectors. Matrix multiplication is a simple representation of unitary application, however it is very computationally expensive. Some operations can be implemented more efficiently as simple sets of instructions, referred to in this report as direct vector manipulation, that when followed produce exactly the same final state as the matrix multiplication. Algorithm 3 shows the direct vector manipulation of the Pauli-X gate of quantum state  $S$ . Q-PACE II includes the use of direct vector manipulation rather than matrix manipulation to improve efficiency.

---

#### Algorithm 3 Pauli-X Gate Direct Vector Manipulation

---

```

for  $k : k \rightarrow 2^n : k + 2^m$  do
  for  $l : l \rightarrow 2^{m-1} : l + 1$  do
     $temp \leftarrow S[k + l]$ 
     $S[k + l] \leftarrow S[k + l + 2^{m-1}]$ 
     $S[k + l + 2^{m-1}] \leftarrow temp$ 
  end for
end for

```

---

The representation used in Q-PACE II cannot express a Toffoli, Controlled-Controlled-Not, gate as a single gate. This made evolving a half-adder circuit more than a trivial test. When Q-PACE II was asked to produce

a circuit with the specification  $|x, y, z\rangle \rightarrow |x, x \oplus y, x \wedge y\rangle$  it was able to produce several exact solutions. One of these was claimed, at the time, to be the “best known solution to the problem”[19] with the restricted gate set. Q-PACE II was also asked to produce a circuit to implement  $|c, a, b, z\rangle \rightarrow |c, a, (a + b)_0, (a + b)_1\rangle$  and again produced “the most efficient solution to this particular problem”[19].

All tests of Q-PACE outlined above were focussed on producing deterministic solutions. During experiments focussed on more complicated problems deterministic solutions were not found. Following on from the work carried out by Spector et al[22–24], Massey changed to a probabilistic approach. The definition, referring to the target state  $a|000\rangle + b|001\rangle$ , of a probabilistic solution with an acceptance level of 0.5 is:

The probability of measuring  $|000\rangle$  is at least  $0.5 \times a\bar{a}$ , and the probability of measuring  $|001\rangle$  is at least  $0.5 \times b\bar{b}$ . [19]

With this focus on probabilistic correctness, the Q-PACE II software was used to evolve a quantum circuit to implement the specification in Equation 2.18.

$$|a_1, a_0, b_1, b_0, z_1, z_0\rangle \rightarrow |a_1, a_0, (a + b)_2, (a + b)_1, (a + b)_0, z_0\rangle \quad (2.18)$$

The result was, despite aiming for probabilistic correctness, a deterministic solution[19].

### 2.3.3 Spector *et al*, Deutsch’s Early Promise Problem

Lee Spector *et al*. [22–24] published a paper that used Genetic Programming to produce a better than classical solution to “Deutsch’s Early Promise”[22–24] problem. This was not an unsolved problem as explained in Sections 2.2.1 and 2.2.2, however it was an example of where quantum computation was known to be more efficient than any possible classical approaches. The specific problem presented was where the number of input qubits for  $f$  was two. This makes it more comparable to the Deutsch-Jozsa algorithm than the original Deutsch algorithm.

The genetic programming approach taken was the traditional tree based representation. The level at which the solution was represented was higher than that of both Q-PACE I and Q-PACE II. Both Q-PACE and Q-PACE II evolved quantum circuits whereas Spector represent solutions as “classical programs which, when executed, construct quantum gate arrays”. This is equivalent to the quantum program definition given earlier.

The best solution produced was a circuit that was different to the Deutsch-Jozsa circuit, Figure 2.6 and was a probabilistic answer to the problem rather than deterministic.

As well as fixed size solutions, the function set, non-terminal set for the Genetic Programming tree, included the control structures required to produce parameterisable quantum algorithms. These were not used to attempt a parameterisable solution to the Deutsch Jozsa problem, which would have seemed the more obvious progression. Instead Spetor et. al. tried to find a solution to the majority-on problem. The majority-on problem is to decide whether the output of a function,  $f$ , has a majority of outputs being 1.

The best solution found was a parametrisable algorithm, which for functions with a large variation from  $2^{n-1}$  of 1’s the solution performed well. However, when tested on functions with  $2^{n-1}$  inputs evaluating to  $f(x) = 1$  the solution ends up with an output error of 0.5.

Both the solution for the Deutsch problem and the *majority-on* problem were searched for with a probabilistic approach, looking for solutions with an acceptance level of 0.48 compared to the 0.5 by Massey.

### 2.3.4 Q-PACE III

Massey’s third generation, Q-PACE III, evolved quantum programs, inspired by the Spector[22–24] results and taking on one of their suggested “Future Work” topics. Just as with Q-PACE II, Q-PACE III was a Genetic Programming suite. A major difference between the two suites is the representation. Q-PACE III represents programs as trees, rather than lists. The execution of the solutions is performed by a pre-order traversal of the solutions tree representation. The tree in 2.10(a) produces the circuit in 2.10(b) when executed.

Due to the change in representation, the evolutionary operations, mutation and crossover, occur at the second order level. As shown in 2.10(a), the different non-terminal nodes were of different arity, allowing for more expressive trees. Whereas in Q-PACE I and II the suitability of an individual could be calculated



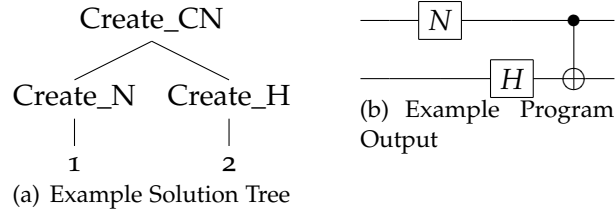


Figure 2.10: Q-Pace III

directly from the individual, in Q-PACE III the individuals have to be executed to produce the quantum circuit before the suitability can be evaluated. This is known as a second order representation. With this additional step, the suitability evaluation requires more computational resources.

Massey defines the PF Max problem as:

You are given a permutation function  $f(x)$  which operates over the integer range  $[0..3]$ . Using a suitable encoding, evolve a quantum program  $U$  which returns the value of  $x$  that gives the maximum value of  $f(x)$ . [19]

Q-PACE III was used to try find a probabilistic solution to the PF Max problem.

It was found that if the probabilistic requirement was reduced to 0.4, from 0.5, a quantum program was evolved which returns the correct value for all 24 possible permutation exactly 50% of the time. The value of 0.5 only produced solutions that could perform correctly on 20 out of the 24 functions. This result was quite remarkable, this probability is twice that of the best classical approach, guessing.

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{\frac{2\pi i j k}{N}} \quad (2.19)$$

Q-PACE III was also used to evolve an solution which, when run, produced the circuit for the Quantum Fourier Transform on 3 qubits. The Quantum Fourier Transform is an operation defined by equation 2.19 where  $N = 2^n$ ,  $x = (x_0, x_1, \dots, x_{N-1})$  is the input state and  $y = (y_0, y_1, \dots, y_{N-1})$  is the resulting state. As mentioned in Section 2.2.4 it is fundamental for Shor's factorisation algorithm. The problem was approached both deterministically and probabilistically, both were successful. Whereas for the PF Max problem the correct answer was a single value, the correct answer for the Quantum Fourier Transform is a state vector. The definition of the acceptance requirement had to be generalised. An acceptance level of  $x$  is define as when each state is has a probability of being observed of at least  $x \times p$  where  $p$  is the ideal probability for the respective state. It was found that for acceptance levels of 0.75, 0.5 and even 0.25, the evolved circuits often had an acceptance value in excess of 0.99.

### 2.3.5 Q-PACE IV

With Q-PACE IV, Massey once again raised the level at which the solutions were represented. Q-PACE IV was a Genetic Programming suite to evolve quantum algorithms, parametrisable with the system size. To reduce the complexity of the representation, all non-terminals were made to be the same arity, 3. This was to remove the restrictions on the mutation operators while ensuring only syntactically correct algorithms were developed. As not all gates require 3 parameters, excess parameters were ignored during evaluation.

The desire to produce quantum algorithms required the inclusion of an iteration construct, numerical arithmetic and a store of variables so loop variables can be used. Several issues were encountered. An issue with the numerical arithmetic inclusion was with the possibility to specify a qubit which does not exist. In a system of 3 qubits, there is no sixth qubit so the instruction "*Create\_H(MULTIPLY(3, 2, X), X, X)*" [19], where  $X$  is a don't care symbol, although syntactically correct is semantically undefined. It was decided that any number above the system size would be interpreted as the system size. A comment in [14] outlines a similar problem when representing gates types using three bits. With only 5 gates types there are two extra values that the three bits can express. Using modulo 5 would be one approach however it leads to two gates represented a single value but three represented by two values, potentially leading to bias towards the over-represented gates. The approach taken by Massey has the same issue. The system

size is massively over-represented in the search space as it is represented by all integers greater than the system size.

The major test for Q-PACE IV was to try and evolve an algorithm to produce an  $n$  qubit Quantum Fourier Transform with 100% fidelity. There is a known algorithm to produce these circuits, provided as Figure 32 in [19], so the test was quantifiable. It was also shown that the instruction set of Q-PACE IV was able to express the algorithm. Q-PACE IV was unsuccessful using the same suitability measure as used by Q-PACE III in its evolution of the 3-qubit Quantum Fourier Transform. The suitability measure was subsequently changed so that it used the polar form of the complex probability amplitudes of each state rather than their Cartesian form. This was more successful and managed to produce an algorithm capable of producing a circuit with 100% fidelity for 1, 2 and 3 qubits.

This algorithm was not entirely system-size independent. The problem was due to the requirement of Quantum Fourier Transform to reverse the order of the qubits. This requires the use of swap gates. The suitability measure was once again altered, however this alteration guided the search in the direction of using swap gates. The alteration provided the ideal number of swap gates, if the circuit does not contain this number it is punished. A solution was found that was system size independent and produced 100% fidelity.

Both of these Quantum Fourier Transform examples show the importance of the suitability measure.

### 2.3.6 Williams and Gray

None of the approaches presented so far have been used to produce the circuit for the quantum teleportation protocol or any other distributed protocol. There is essentially no difference between the circuits required for the quantum teleportation protocol than to produce the Quantum Fourier Transform. The main difference is a restriction on communication.

With the quantum teleportation protocol, only classical communication can be used as control when acting on the qubit held by Bob. In terms of automatically producing the circuits the restriction can be characterised as *only single qubit gates, controlled or uncontrolled, can be applied to the qubit held by Bob*. The use of controlled gates can allow the search to loosely represent measurement and communication of the measured value.

Williams and Gray[25] use a genetic programming approach to tackle the quantum teleportation protocol. The result required fewer gates than the smallest previously known circuit.

The Williams and Gray approach uses a comparison between the target unitary matrix and the unitary matrices that the produced circuits represent. The approach also tackled the quantum teleportation protocol in two parts, based on the two parts of the circuit introduced by Brassard[26].

The use of the target unitary matrix is the major disadvantage of this approach. The construction of the target unitary matrix is also usually a significantly difficult task. Therefore the approach can only really be used to try reduce the circuit size of problems that already have known solutions or for problems with target matrices that are simple to produce. If a solution is known, the target unitary matrix can be easily produced by the multiplication of the unitary operations in the known solution. If the target matrix is easy to produce it is also likely that a correct circuit will be easy to produce.

The other major disadvantage of the use of matrices is that the search is for circuits rather than algorithms. A target matrix is specific for a certain number of qubits in a circuit, it is not scalable, requiring a new matrix for 1, 2, 3, ... qubits involved in the circuit. For problems where the matrix is hard to produce this a much less viable approach.

### 2.3.7 Yabuki

A genetic algorithm approach to produce the quantum teleportation protocol is presented by Yabuki[27]. The approach is again a circuit, rather than algorithm, generator.

Compared to the genetic programming approach taken by Williams and Gray, the use of genetic algorithms has a dramatic impact on the scaling of the search. With a typical genetic algorithm the length of the chromosome is fixed. As a result the maximum number of gates that can be encoded in the chromosome is fixed. However, this requires some indication as to how many gates will be required. The success of a search is likely to be relatively sensitive to the amount of "excess" chromosome space. This means that for problems where the size of the solution circuit is unknown, setting an arbitrary chromosome size could hinder the search. Another crucial point is that the structure of the system is assumed. The assumed structure does not, like in Williams and Gray[25], allow separate searches for the structural

sections. It does however change the interpretation of the genes. Each gene contains three values between 0 and 3. The semantic meaning of a gene is looked up in a table using the three values. Each structural section has a different table. If the first value is 3 it indicates the end of a structural section.

The approach found a quantum teleportation circuit that was “simpler than ever known”[27]. However, this result has to be viewed with respect of the restrictions placed on the search and a known solution being able to provide the interpretation tables for each section. The assumed structure and changing interpretation between sections allows the system to accurately enforce the result of measurement.

For unsolved problems this reliance on structure could limit the success of the approach. Defining the interpretation tables will also be difficult for unsolved problems.

This results in a very specialised search technique. It cannot easily be generalised to be applicable to other problems.

## **2.4 The Focus of this Project**

The sections above summarise current research into the use of evolutionary computation to design quantum computational artefacts such as circuits, programs, protocols and algorithms. All reviewed research appears to have required the bespoke development of software to carry out peripheral tasks such as the circuit simulation and state representation. This software is generally not publicly available. This clearly impedes progress in this area.

Tools are available to visualise circuits. Also simulators are available that given a circuit evaluate it on a state. However, there are no tools available to provide assistance during the design.

The aim of this project is to provide tool support for the automated synthesis of quantum artefacts via heuristic search. Typically these artefacts will be circuits, programs, algorithms and protocols. The support provided will allow users to rapidly configure and run searches for these artefacts, and also allow them to visualise simulations of the artefacts produced.

The aim is to free the user from having to develop a great deal of bespoke software. Elements such as a heuristic search algorithm and several suitability measures will be provided. If these do not suffice for the user’s needs then other components should easily be assembled into the framework in a plug-and-play style.

The creation of such a toolkit is, to the best of my knowledge, unprecedented. All previous work in the area has focussed purely on the discovery of quantum artefacts with each researcher working in isolation. The adoption by researchers of the toolkit could improve the productivity of research in this area.

As a demonstration of the toolkit a number of experiments searching for quantum algorithm will be performed as a final part of the project.

### 3 Requirements

This project aims to provide software to facilitate research into quantum computation through heuristic search. As an initial architectural decision the software is broken down into three sections.

The first section is a plug-and-play research framework that will enable the search for quantum algorithms by providing developed peripheral utilities such as circuit construction and simulation. The framework will allow researchers to perform research into search techniques and suitability measures without reimplementing peripheral, non-research, tasks. The framework is not intended as a standalone application but as a library that can be incorporated into other software.

The second section is a prototype implementation of at least one search technique and suitability measure. The inclusion of these modules shall be used to indicate how the framework is intended to be used by researchers and demonstrate the plug-and-play ability of the framework.

The third section is a client with a graphical user interface for the framework. A client is necessary to allow the toolkit to be used as a standalone application. This may seem as though it should be included as part of the framework, however the distinction is made to highlight the different aims. The framework is intended to provide the structure and utilities, such as circuit simulation, that can be included as a single module in any application, for example a general search application or a quantum IDE. Some researchers may not wish to develop an application in which to embed the framework. The client provides a basic user interface to the framework for such researchers.

#### 3.1 Definition, Acronyms, and Abbreviations

The definitions given here are consistent with those used in the rest of the document but are summarised here for convenience.

**System Size** - The number of qubits in the system. For example the quantum teleportation protocol has a fixed system size of 3 whereas the Quantum Fourier Transform can scale to any system size.

**Quantum State** - A column vector of  $2^n$  complex numbers representing the probability amplitudes of the  $2^n$  states  $|0\rangle \dots |2^n - 1\rangle$  for a system size  $n$ .

**Quantum Gate** - A unitary operation performed on a quantum state.

**Quantum Circuit** - An ordered list of quantum gates to be applied to the quantum state.

**Quantum Program** - An ordered list of instructions used to construct a quantum circuit.

**Quantum Algorithm** - An ordered list of instructions including variables such as the system size. A instantiation of a quantum algorithm with the system size specified becomes a quantum program.

**Suitability Measure** - A function that assigns a quantitative measure of how well a proffered solution performs. A value of 0 indicates ideal functionality with increasingly higher values of the function indicating ever poorer performance.

**Search Target** - Every search has a target which is a representation of the area of the search space that contains solutions to the search problem. For example, in a search for  $x$  in a list the search target could be the indices of all occurrences of  $x$  in the list and in a search for  $x = \sqrt{4}$  the search target would be  $x = \pm 2$ .

#### 3.2 Requirements Summaries

This section contains a summary of the requirements of each of the separate phases of the project.

##### 3.2.1 Framework

###### Additional Search Engines - Req:ASE

The framework shall allow researchers to provide search engines for the system to use. Different search techniques are likely to perform better for different problems. The way in which the framework provides this shall not favour any search technique.

###### Additional Suitability Measures - Req:ASM

The framework shall allow researchers to provide suitability measures for the system to use. In heuristic search, the terms fitness function and cost functions are typically used to indicate suitability measures. In general we would seek to maximise "fitness" and minimise "cost" of any proffered solution.

It is well known that the suitability measure may have significant impact on the success of a search. A suitability measure that works well for one problem may not work for another. Therefore, the generality of the framework relies heavily on being able to add suitability measures.

### **Quantum Algorithm Output - Req:QAO**

The solution of a search, a quantum algorithm, shall be presented to the user as a list of instructions. An algorithm is a list of instructions parametrised by the system size. When an instantiation with a specified system size is executed a circuit for that system size is produced. The solution of a search using the framework is an algorithm as defined in Section 3.1. This solution shall be provided to the user as a list of instructions in a consistent format.

### **Circuit Visualisation - Req:CV**

The system shall provide visualisation of the circuit produced by the solution of the search for a user specified system size. To ensure that the output of the search is useful the framework shall provide a representation of the resulting circuit that can be rendered into a circuit diagram.

The circuit visualisations produced use standard depictions of gates and circuits.

### **Third Party Software - Req:TPS**

The framework shall be able to be embedded in third party software. The framework is intended for use by the research community and it is not intended to limit the ways that it can be used. As a result it is not only important that the framework be able to use third party software, search engines and suitability measures, but is also important for the framework to be available for inclusion in third party software.

### **Definition of Search Target - Req:DST**

The framework shall provide a standardised format for users to define the target of the search. The framework needs to provide a standard way of defining the details of a search target. The standard shall be formalised so it is able to be used and generated by third party software.

### **Use of Configuration Files - Req:UCF**

The customisation of the framework shall be provided through a series of configuration files. All third party additions to the framework, search engines and suitability measures, shall be specified using a series of configuration files. These configuration files shall be well defined and able to be used and produced by third party software. Files are specified for persistence.

Another option was for configuration to be manually performed each time the framework is used. This was rejected as the configuration of available search engines and suitability measures are unlikely to change very often. Maintaining a persistent configuration will also reduce the opportunity for human error during configuration as it will be performed much less frequently.

### **Provided Gates and Algorithm Instructions - Req:PGAI**

The framework shall provide implementations of all gates specified in Table 3.2. The framework shall provide algorithm instructions for each of these gates and for the instantiation of the Controlled-U gate with all single qubit gates. Table 3.2 defines most well known quantum gates and indicates the visual representations used in the project.

### **Algorithm Control Structures - Req:ACS**

The system shall provide an iterate control structure and support nested iterate structures. Many algorithms require nested loop structures. However, naïve nesting is not sufficient. Each loop construct shall provide a loop variable that must be accessible within the loop including inside any nested loops. The algorithm produced by Massey[19] producing the Quantum Fourier Transform circuit is an algorithm that requires such a loop structure.

### **Producing Circuits from Algorithms - Req:PCA**

The framework shall be able to produce a circuit, for any system size, from a quantum algorithm.

### **Circuit Simulation - Req:CS**

The framework shall provide the simulation of a circuit given an initial state. Using the gate definitions given in Table 3.2, a circuit constructed of the supported gates shall be able to be accurately simulated. Given an initial state the framework shall be able to give the final state up to the accuracy of floating point arithmetic.

### **Step-by-Step State Evolution - Req:SBSSE**

The framework shall provide a way to perform step-by-step evaluation of a circuit given an initial state. To aid researchers in understanding the circuits produced by the preferred algorithms a step-by-step














 $\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$	 $\begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}$	 $\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$
 $\begin{pmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{pmatrix}$	 $\begin{pmatrix} \cos \frac{\theta}{2} & -i \sin \frac{\theta}{2} \\ -i \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{pmatrix}$	 $\begin{pmatrix} \cos \frac{\theta}{2} & -\sin \frac{\theta}{2} \\ \sin \frac{\theta}{2} & \cos \frac{\theta}{2} \end{pmatrix}$
 $\begin{pmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{pmatrix}$	 $\begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}$	 $\begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}$
 $\begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}$	 $\begin{pmatrix} a \\ b \\ c \\ d \end{pmatrix} \rightarrow \begin{pmatrix} a \\ c \\ b \\ d \end{pmatrix}$	 $\begin{pmatrix} I & 0 \\ 0 & U \end{pmatrix}$
	 $\begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{2\pi i}{2^k}} \end{pmatrix}$	

Table 3.2: Supported Gates and Definitions

evaluation shall be provided. Given an initial state and a circuit, the state after the application of each unitary operation, gate, shall be reported so the state evolution can be traced.

This will also provide a debugging mechanism to ensure that all unitary operations are performing the expected operation on state.

### 3.2.2 Prototype Implementation

#### Sample Search Engine - Req:SSE

**The tool shall provide at least one implemented search engine.** The tool shall provide a basic search engine that will allow researchers interested in the quantum algorithms, rather than the search techniques, to use the tool “out of the box”. If time permits more may be provided.

#### Sample Suitability Measure - Req:SSM

**The tool shall provide at least one implemented suitability measure.** The tool shall provide a basic suitability measure that will allow researchers interested in the quantum algorithms, rather than the suitability measures, to use the tool “out of the box”. The suitability measure provided must allow basic circuits to be produced but is otherwise unspecified.

#### Sample Search Targets - Req:SST

**The tool shall provide a number of search targets with known outputs.** To allow search engine and suitability measure researchers to perform simple experiments the tool shall provide a selection of basic search targets. The search targets included are not specified.

### 3.2.3 Client

#### Search Engine Selection - Req:SES

**The GUI shall provide a user with a selection of search engines to use in a search.** The GUI shall provide a selection between all search engines registered in the framework. The most recently selected search engine shall be used by subsequent searches.

### **Suitability Measure Selection - Req:SMS**

The GUI shall provide a user with a selection of suitability measures to use in a search. The GUI shall provide a selection between all suitability measures registered in the framework. The most recently selected suitability measure shall be used by subsequent searches.

### **Search Target Selection - Req:STS**

The GUI shall provide a user with a selection of search targets to be used as the search goal. The GUI shall provide a selection between all search targets registered in the framework. The most recently selected search target shall be set for subsequent searches.

### **Search Target Creation - Req:STC**

The GUI shall provide a way for users to create a new search target without to explicitly writing the configuration file. Writing configuration files is quite monotonous and highly error prone. The GUI shall provide a way to create these configuration files that reduces the error rate. The way that the GUI provides this is not expected to dramatically decrease the monotony due to the nature of the amount of information required for problems with high numbers of qubits. The inclusion of such a feature is very important to improve the usability of the system and improve the potential level of use in the research community.

### **Search Target Editing - Req:STE**

The GUI shall provide a way for users to edit the contents of a previously created search target without manual editing the configuration file. The size of the data per test case increases exponentially as the number of qubits increases. The size of the configuration file required to specify a search target will increase linearly on top of this with respect to the number of test cases. With the size of configuration file increasing in such a dramatic way the risk of error when directly editing the configuration files increases in a similar fashion. To decrease the risk of errors the GUI shall provide a way to graphically edit the test cases.

### **Loading a Search Target From a Previously Defined Configuration File - Req:LSTPDC**

The GUI shall provide a way to import a predefined search target from a configuration file. One of the intended uses of the GUI is for research into producing quantum algorithms. It is possible that researchers will want to distribute the search target definitions they create. This distribution may be to colleagues or simply to other computers for them to continue work. Either way once a search target is defined and distributed, the use of received search target configuration files should be supported by the GUI. The GUI shall provide a way for users to import search targets using the respective search target configuration file as long as the configuration file is of the correct format.

### **State Visualisation - Req:SV**

The GUI shall provide a way to visualise any quantum state. A quantum state is defined as a vector of complex numbers. Depending on size, comparing two or more states can become monotonous. If the comparison of the two states does not need to be exact, a visual representation of the two states can provide a simpler, and quicker, method for comparison. To provide such comparison the GUI shall provide a way to visualise a quantum state.

### **Reporting the Search Result - Req:RSR**

The GUI shall provide a way to report the search result, a quantum algorithm, to the user. The GUI would be of no use to any quantum algorithm researcher if it did not provide the results of a search. The GUI shall provide the quantum algorithm meeting requirement Req:QAO.

### **Graphical Circuit Visualisation - Req:GCV**

Given a quantum algorithm and a system size, the GUI shall produce a visualisation of the resulting circuit. Some quantum algorithms produced using the search are likely to be hard to understand in pure algorithm form. Understanding a circuit is likely to be easier. To save researcher time in drawing the circuits by hand, the GUI shall provide a visualisation of the circuit for a specified system size.

### **Graphical Step-by-Step State Evolution - Req:GSBSSE**

The GUI shall provide a way to perform, control and visualise the step-by-step state evolution for an initial state and circuit. The framework provides the ability to analyse the evolution of a state with respect to an initial state and a circuit. The GUI shall provide a way of controlling and reporting this step

by step evaluation to the user.

#### **Tooltips - Req:TT**

**The GUI shall provide user help through the use of tooltips.** All elements of the GUI shall be explained through the use of tooltips. A separate help system would be another option. Tooltips were chosen over a separate help system as they provide less intrusive help and improved context. A second consideration is the provision of GUI components by third party. With a separate help system these GUI components would either have to provide another separate help system or integrate into the main help system. With tooltips, all help is locally defined and therefore does not require integration making it much simpler when considering potential third party components.

#### **3.2.4 General Requirements**

##### **Portability - Req:POR**

**The framework, fully implemented tool and the GUI shall be able to be used on a range of Operating Systems.** The produced software shall be able to be run on:

- Windows 7
- Linux



## 4 Design and Implementation

This section includes both design decisions and implementation details of the software produced in this project. Following good software engineering practice, version control over all code and documentation was performed using SVN provided by Google[28].

One of the basic implementation choices was the programming language used to implement the software. Several properties had to be considered. The framework is intended to be used by researchers, if it is implemented in a language that is scarcely used the adoption of the framework would be affected. Tool support was also a significant consideration. The availability of tools such as development environments, debuggers and profilers can vastly improve the quality of software. With a motivation of the project being to reduce bespoke research software, the libraries that are available must be considered.

After consideration of all three factors Java was selected as the programming language for the implementation. Java has good tool support, a high number of available libraries and is used sufficiently in the research community. An additional benefit of Java is that requirement Req:POR is made considerably simpler by the Java virtual machine.

### 4.1 Framework

In this section I will outline the design decisions that directly effect the framework produced and how it was implemented. The architecture of the framework can be seen in the class diagram in Appendix E.

#### 4.1.1 Complex Numbers

Complex numbers are central to quantum computing and so are included in the framework. There are two ways to encode complex numbers. One represents complex numbers explicitly as a pair of floating point numbers the other encapsulates the real and imaginary components inside a complex number data structure.

The framework implements the second of these options. This was chosen for several reasons. The primary reason was to reduce the risk of programming errors. The replication of complex number arithmetic throughout the framework, any implemented search engines and suitability measures would increase the probability of implementation errors. It is also good software engineering practice to encapsulate the constituent elements and relevant operations.

Secondly, brief research online confirmed that complex number libraries are already available for Java. This reuse of previously written software can reduce the likelihood of errors and is good software practice.

The third reason is that one of the principles behind producing the framework is the attempt to try standardise the research from different researchers. Without the provision of this “Complex” class researchers could use both Cartesian form and the Polar form without clear distinction as both would be encoded as a pair of floating point values. The use of a common Complex class throughout the framework removes this ambiguity.

The implemented Complex class is based on an implementation provided as part of a “Complex Function Grapher”[29]. The Complex class provides both Cartesian form  $real + imaginary$ , through `real()` and `imag()` returning the real and imaginary components respectively, and Polar form  $re^{i\theta}$ , through `mod()` and `arg()` returning  $r$  and  $\theta$  respectively. The implementation has been adapted to better suit this application such as the addition of the euclidean distance between two complex numbers. A second addition is the ability create a Complex object from a string, such as  $3 + 4i$ , using `parseComplex` just as `parseInt` and `parseDouble` are used to create integers and doubles from strings respectively.

#### 4.1.2 Matrices

As explained in Section 2 the application of a quantum gate is simply the application of a unitary operation, represented as a matrix, to a quantum state. The framework has to implement matrices in some way.

There are several possible representations. The framework could use either an explicit two-dimensional array representation or could provide an encapsulated matrix data structure.

The framework uses the data structure encapsulation. The justification for this decision is identical to that discussed above for complex numbers. If the framework were to simply represent matrices as two dimensional arrays, two researchers could order the dimensions differently. This ambiguity is removed when the matrices are encapsulated in a matrix data structure.

The implementation of the “Matrix” class is based on the JAMA Matrix package[30]. The JAMA Matrix package provides matrices of double values. This was updated to provide matrices of Complex objects

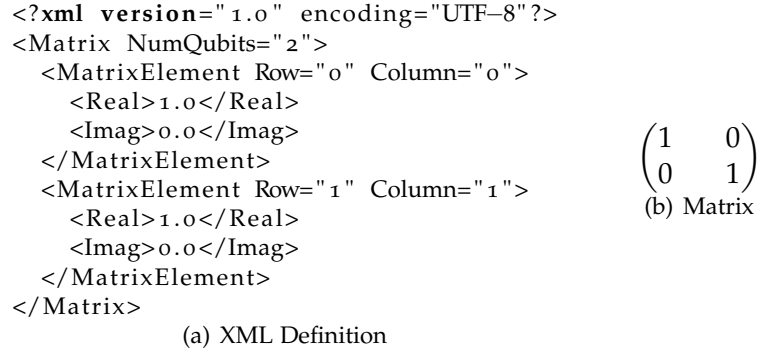


Figure 4.1: Pauli-I Gate

with all the required operations on these matrices. The JAMA Matrix package provided functionality that is not required for the framework and was therefore removed.

The JAMA Matrix package does not include the tensor product operation. This operation is used heavily when applying  $n$  qubit gates to  $n < \text{qubit}$  systems and so was specifically developed.

Matrices are used to specify arbitrary gates, such as the modified Controlled-Not in the Deutsch algorithm. To enable researchers to distribute search problems including any corresponding arbitrary gates these unitary matrices need to be encoded in a form that can be distributed between researchers. All elements of this framework that needs to be distributed use XML. An example of the XML encoding and the matrix it defines can be seen in Figure 4.1. The XML representation only stores non-zero elements so as not to store unnecessary information. XML was used rather than the Java Serializable interface to ensure that 3rd party applications can be used to modify the stored matrices if required.

#### 4.1.3 State

Quantum states are naturally represented as  $2^n \times 1$  matrices, generally referred to as a vector. All unitary operators are implemented simply as the application of  $2^n \times 2^n$  matrices to these quantum state vectors.

##### 4.1.4 Test Suite Structures

With most heuristic search problems there are a series of test points that are used to measure the suitability of any proffered solution.

For quantum algorithms the expected results are the state vectors that would be produced by an ideal solution. It was decided that a test case would be represented as a pair of state vectors, the starting state and the expected state.

For some problems, such as the Deutsch and Deutsch-Jozsa problems, the test cases requires more than the starting and final states. As is shown in Section 4.1.11 the gate  $f$  used in the Deutsch and Deutsch-Jozsa algorithms is not provided by the framework. This may initially seem a rather strange omission. However,  $f$  in the Deutsch and Deutsch-Jozsa algorithms is not a fixed gate, it is an arbitrary binary function that is guaranteed to be either constant or balanced. Therefore it is not sufficient for the test cases to be simply the starting and final state. The test cases includes the custom unitary matrices. With the matrices for the modified Controlled-Not gates held in the test cases the Deutsch and Deutsch-Jozsa problems can be defined.

Each circuit produced by a quantum algorithm has  $n$  qubits and can only be evaluated using test cases for  $n$  qubits. Simulating test cases for any other number of qubits is incomputable due to incorrect matrix dimensions. The notion of a test set was introduced to contain all test cases for a specific  $n$  for a given search problem.

However, the power of a quantum algorithm compared to a quantum circuit is in the generality of the algorithm for any system size,  $n$ . This means that a single test set is not suitable as it would only evaluate the algorithm for test cases for a single  $n$ . The notion was introduced of a test suite to contain all the test sets produced for the same search problem. There is only one test set within a test suite for each distinct value of  $n$ . The implemented structure can be seen in Figure 4.2(b).

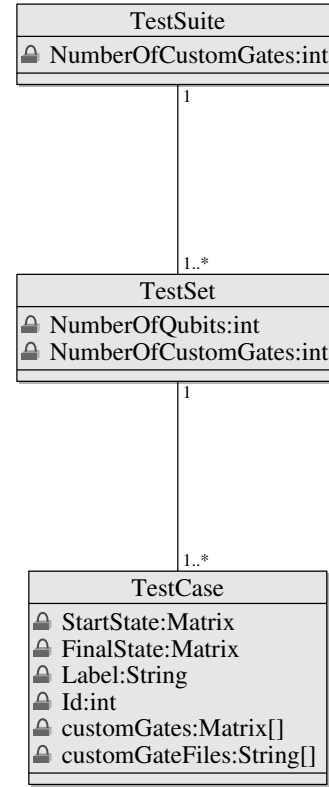
The number of custom gates that are available to use are constant for all the test cases in the test suite. This number cannot vary as the algorithms produced must be able to contain only the gates available. If

```

<testsuite>
  <testset NumQubits="1">
    <testcase><!-- 0 -->
      <starting_state>
        <matrix_element><!-- 0 -->
          <Real>1.0</Real>
          <Imag>0.0</Imag>
        </matrix_element>
        <matrix_element><!-- 1 -->
          <Real>0.0</Real>
          <Imag>0.0</Imag>
        </matrix_element>
      </starting_state>
      <final_state>
        <matrix_element><!-- 0 -->
          <Real>0.0</Real>
          <Imag>0.0</Imag>
        </matrix_element>
        <matrix_element><!-- 1 -->
          <Real>1.0</Real>
          <Imag>0.0</Imag>
        </matrix_element>
      </final_state>
    </testcase>
  </testset>
</testsuite>

```

(a) Partial Test Set for Pauli X Gate



(b) Test Suite Class Diagram

Figure 4.2: Test Suite Structures

test cases were able to contain a different number of custom gates an algorithm produced could contain the instruction to construct “Custom Gate 4” but a test case might only provide two custom gates.

The test suite is fully defined in a single XML file. The XML in Figure 4.2(a) is a sample of such a file. It is easy to see how the file structure reflects the internal structure of test suites just described.

Separate files are used to encapsulate the specification of different test suites. This separation follows the software engineering principle of modularity to ensure that the files remain readable and maintainable. The framework is intended to improve the ability of researchers to reuse development of other researchers including defined test suites. With this modularity the distribution and reuse of defined test suites will be much simpler than if the test suites were all defined in a single file

The implementation of the test suite structure can be seen in Figure 4.2(b). The custom matrices are held both as an array of Strings and of Matrix objects. This is because when creating a test suite definition XML file the file names of the custom gate XML files are required but creating matrices from file every evaluation is slow. If there were just an array of Matrix objects these file names would be lost. The inclusion of a second array was initially discounted as it would introduce an unnecessary source of potential bugs concerning the two arrays being “out of sync”; meaning the Matrix objects may not represent the matrices encoded in the respective files. To remove the potential for the two arrays to be out of sync the matrix array is created by the constructors while the matrices are only externally accessibly using their index index. When the custom matrices are retrieved, the matrices are copied to ensure the matrices held in the test case cannot be altered.

Alongside the framework, an independent test suite graphical editor is provided. A user can use this to produce the XML file and therefore does not need to explicitly create the XML file. The design and implementation of the graphical editor can be found in Section 4.2.1.

The contents of a test suite are not fixed. If a user finds an error or wishes to add test cases, either a third party application or the provided test suite graphical editor can be used to update the respective test suite elements. These changes need to be reflected in the XML file to become persistent so to ensure that the updated XML file is well-formed the framework provides a class to output the XML of a given test suite. This class is used in the independent test suite graphical editor and is recommended for use by any

```

<FitnessFunc>
  <FitnessFunctionTag>
    <Name>SUITABILITY MEASURE NAME</Name>
    <Class>IMPLEMENTING FULLY QUALIFIED CLASS NAME</Class>
    <Desc>SUITABILITY MEASURE DESCRIPTION</Desc>
  </FitnessFunctionTag>
</FitnessFunc>

```

Figure 4.3: XML for Suitability Measures Manager Configuration

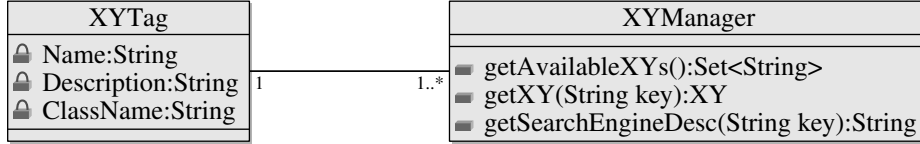


Figure 4.4: Manager - Tag Class Diagram

third party applications.

When editing a test suite, if a test set for  $n$  qubits is added to the test suite that already contains a test set for  $n$  qubits, the two test sets are merged. The test cases of the test set being added have their ID and labels modified so that they continue the sequence of the existing test cases already in the test suite. During this merge process the test cases are cloned. This is in case the added test set is used in a different test suite. If this were not done, when the ID and label were modified they would also be updated in the second test suite. This would cause unpredictable behaviour.

#### 4.1.5 Manager Classes

As can be seen in the architecture diagram of the framework, Section E, there are several classes with names suffixed with “Manager”. These classes provide access to the extendible areas of the framework. There is a Manager class for the suitability measures, the search engines and the search problems. Each of these are a specific site of expansion.

Each Manager is configured using an XML file specifying all options for the specific Manager. The suitability measure Manager will be configured for all the available suitability measures, the search engine Manager for all the available search engines, and the problem Manager for all the available search problems.

The three managers are configured at runtime, not at compile time. This allows researchers to add suitability measures, search engines and search problems without recompiling the framework. This runtime configuration provides a better plug-and-play design, a key foundation of the framework, compared to compile time configuration.

Figure 4.3 shows an outline of the XML file used to specify the available suitability measures. The XML files specifying the available search engines and the available search problems can be found in Appendix F.1 and F.5.

These XML files are used to register the available implementations with the respective Managers. The Manager classes use these registrations to provide the choice of available instantiations.

The search problem Manager class is outlined in detail in Section 4.1.7. Both the search engine and suitability measure Manager classes are implemented in the same way. The class diagram in Figure 4.4 shows the Manager design. A mapping between the Name of the registered options and Tag objects is held in each Manager. A Tag contains the information within the `<XYTag></XYTag>` element of the XML file. The `getAvailableXYs()` method returns a set of the registered names rather than the XY Tags. This ensures the Manager classes have full control over the creation of the managed objects. To produce an XY object the Manager classes use reflection to instantiate an object of the class indicated by the respective `ClassName` variable.

#### 4.1.6 Multiple Search Engines and Suitability Measures

It is hoped the framework will be widely adopted by quantum algorithm researchers, who may use different search techniques and suitability measures. A simple interface that allows researchers to easily select a different search technique or suitability measure. The simple interface for the Manager classes of

```

<Problems>
  <prob>
    <Name>Final Pauli X</Name>
    <DefFile>config/finalpaulix.xml</DefFile>
    <Desc>A Pauli X gate on the final Qubit</Desc>
  </prob>
</Problems>

```

Figure 4.5: XML for Problem Manager Configuration

both the search engine and suitability measure allow a user to:

- retrieve the names, used as the identifier, of all registered implementations
- retrieve the instantiation of the specified implementation
- retrieve the description of the specified implementation

All registered search engines and suitability measures must implement a respective interface. The internal workings of the different implementations are unrestricted.

The alternative approach would have been to implement a series of search engines based on several different techniques alongside a collection of suitability measures and provide researchers this choice. This was not accepted as it moved the tool away from the plug-and-play framework intended.

#### 4.1.7 Multiple Problems and Problem Specification

Section 4.1.4 describes how test suites are encoded in XML files. The problem manager uses a single XML file to specify all available search problems while retaining separate files for each search problem's test suite. An example of this XML files can be seen in Figure 4.5. A search problem has a name, description and file name for the respective test suite XML file. The name and description are used to provide a human readable explanation of the problem represented by the test suite XML file. The use of a separate XML file to collate all defined problems makes maintenance much simpler. Without this each test suite would have to be individually registered with the framework at runtime or all search problems including the test suites would have to be specified in a single file. This would result is a very hard to maintain and would make the distribution of test suites much harder.

Providing a problem manager allows the framework to be used for different problems without having to restart the system or the third party software in which the framework is embedded directly providing different test suites.

The search problem Manager works in a similar way to the search engine and suitability measure Managers detailed in Section 4.1.5. However, the difference between two registered search problems is the test suite XML file rather than the class of the instantiated object. A second difference is that the registered options are can change, a third party application can interact with the Manager to create new search problems and update the already registered search problems. Therefore, the search problem Manager has to be able to produce an updated XML file so the changes made are persistent. The changes managed by the search problem Manager are updates to the values in the Manager XML file. Any changes made to the test suite are not maintained by the search problem Manager. This ensures that the different concerns are separated. The update of test suite XML file is detailed in Section 4.1.4.

#### 4.1.8 Quantum Algorithms

The results of the searches are quantum algorithms. To maintain the "Plug and Play" nature of the framework, the representation of these algorithms needed to be specified and standardised. However, the representation also had to allow easy manipulation by a variety of search techniques.

To provide a standardised and non-limiting representation the framework provides an internal quantum algorithm structure that can be built by any search engine. This allows the search engines to have a different internal representation that is then used to build the standardised algorithm. Using this there are no limitations on the structures used internal to the search engines.

The use of the standardised quantum algorithm also ensures that the reporting of an algorithm to the user is consistent.

Instruction QuantumInstruction(Enumeration)	Gate1 expnode	Gate2 expnode	Phase expnode	Sub-Algorithms QuantumAlgorithm[]
--	------------------	------------------	------------------	--------------------------------------

Figure 4.6: Quantum Instruction Structure

$exp \rightarrow e + e$	$e \rightarrow exp$
$exp \rightarrow e - e$	$e \rightarrow SystemSize$
$exp \rightarrow e * e$	$e \rightarrow Value[1..10]$
$exp \rightarrow \frac{e}{e}$	
$exp \rightarrow e^e$	
$exp \rightarrow \frac{\pi}{2^{e(mod10)}}$	
$exp \rightarrow LoopVars[e]$	

Table 4.1: Expnode Context Free Grammar

An algorithm is a list of instructions. The instructions that are used in the standardised quantum algorithms take the form shown in Figure 4.6. The list of values that the Instruction element can take can be found in Appendix G. When the algorithm is reported to the user, it is reported as a list of instructions. To improve readability, for instructions that do not use the Gate2, Phase or sub algorithm elements they are omitted and the { and } characters are used to denote the body of control structure instructions. The  $\frac{\pi}{2^{e \% 10}}$  rule has the exponent modulo 10. This modulo term was added for pragmatic reasons. During initial experimentation it was found that without this term the search techniques could produce large exponents, requiring high levels of resources but did not provide any interesting results

*Gate1*, *Gate2* and *Phase* are all listed as being of type *expnode*. The type after evaluation for *Gate1* and *Gate2* is integer and for *Phase* is double. The *expnode* type is required for these values to depend on value variables such as the system size and also loop variables. The grammar that defines the *expnode* type can be seen in Table 4.1. The *expnode* type is needed so that the algorithms can react to the parametrisation that provides the increased power when compared to quantum circuits. A constant cannot be used for *Gate1*, *Gate2* or *Phase* as the value may depend on the system size and have to be calculated when a circuit is being built using the algorithm.

---

#### Algorithm 4 Nested Loop Variable Access

---

```

for  $i : 0 \rightarrow n : i++$  do
  {You can use i here}
  for  $j : 0 \rightarrow m : j++$  do
    {You can use i and j here}
  end for
end for

```

---

With the inclusion of a loop control construct in the form of the *Iterate* and *RevIterate* instructions additional parameters are introduced. In Java and many programming languages when inside nested loops all loop variables are accessible. Such a structure is shown in Algorithm 4. This means after the first **for** statement the loop variable *i* is able to be used but after the second **for** statement both loop variable *i* and *j* can be used. This nesting is quite a powerful language feature. With the inclusion of the *iterate* instructions, and therefore the possibility of nested iterations, this loop variable access could be implemented in one of two ways. The most naïve implementation would be to just allow access to the “closest” loop variable. This would mean that the code at after the first **for** statement wouldn’t be effected but the code at after the second **for** statement would no longer be able to use the *i* variable. The more sophisticated option is implemented to provide access to all loop variables. This is a second set of variables, alongside the system side, that necessitated the use of the *expnode* type. All current iteration variables are provided in a *LoopVars* array. As can be seen in Table 4.1 the array is indexed by the value of a sub-expression. To ensure that all indices requested are valid the value of the sub-expression is calculated using modulus *LoopVars.length()*. If the *LoopVars* array has a of length 0 the result is 0 irrespective of the index requested.

The difference between the *Iterate* and *RevIterate* instructions is that *Iterate* counts from 1 to *Gate1* and

*RevIterate* counts from *Gate1* to 1. As is explained in Section 4.1.9, the qubits are numbered from 1 to  $n$  which is why the iteration instructions count to and from 1 rather than 0 as is normal in many parts of Computer Science. The two different iterate instructions are needed as they can express some looping constructs in a much simpler form than would be possible with just one of them.

For all *Create\_\** instructions, *Gate1* is used to index the target qubit the gate. *Gate2* is only used by *Create\_C\** and *Create\_SWAP* instructions to index the control qubit and second qubit respectively. For the *Iterate* and *RevIterate* instructions, *Gate1* is used to provide number of iterations,  $n$ . For the *Create\_R\**, *Create\_CR\**, *Create\_R* and *Create\_CR* instructions, the Phase element is used to specify the value of  $\theta$  used by the gate.

#### 4.1.9 Qubit Numbering

One of the major decisions made was way in which the qubits should be numbered.

The chosen approach was that the identifiers increase with significance. This means that for state  $|zy\dots ba\rangle$  the qubit represented by  $a$  would always have the identifier 1 and the qubit represented by  $z$  will always have the identifier equal to the system size. For example, if the system size is three the identifier of the qubit represented by  $z$  would be 3. This was chosen to ensure that an identifier always represents the same qubit, irrespective of the number of qubits in the system.

The discarded option was for the identifiers to increase as significance decreases. This would mean that the qubit represented by  $z$  would always be 1 whereas the qubit represented by  $a$  will always have the identifier equal to the system size. Therefore with different system sizes the same identifier actually correspond to qubits of different significance.

The justification for this choice is to make the algorithm simpler. If the identifiers were dependant on the number of qubits it would make the understanding algorithm slightly more involved.

The use of this numbering is also much more natural as the identifier,  $x$ , of a qubit,  $a$ , is related to the value of the qubit when read in binary. The value of the qubit  $a$  is  $2^{x-1}$ . This makes the optimisation of gate application, see Section 4.1.11, much simpler.

The identifiers start at 1 rather than 0. This allows the value 0 to be used as the flag within the framework to indicate that the system size should be used. For example the instruction *Create\_H 0* would create a Hadamard gate on the qubit with the identifier equal to the system size, highest significant qubit. The use of 0 allows the integer type to be used without additional flag types to complicate the framework and algorithms.

#### 4.1.10 Quantum Circuits

To perform the evaluation of an algorithm the circuits for the test sets need to be produced. Both the representation of the circuit and the mechanism to construct the circuit from the algorithm needed to be standardised to ensure the “Plug and Play” nature of the framework.

The framework provides a default circuit builder and allows a separate circuit builder to be provided as long as it conforms to the interface and the circuits it produces also adheres to the respective interface. There is no manager class provided for circuit builders. This was due to an assessment of the intended uses of the framework. It is intended that the framework would be used primarily to perform the following:

- Perform research into the affect of different suitability measures on the search for quantum algorithms
- Perform research into different search techniques that could be used to produce quantum algorithms
- Perform research to produce new quantum algorithms for a specific problem

It is not seen as a priority of the system to provide the same level of flexibility to the circuit building as the search engines and suitability measures.

The provided circuit builder has several rules. Firstly, for *Gate1* and *Gate2* the absolute value of the *expnode* expression is used as there are no qubits with negative indices. Secondly, For *Gate1* and *Gate2* the *expnode* expressions, see Section 4.1.8, can evaluate to 0, however due to the way the qubits are numbered, see Section 4.1.9, there is not a qubit with identifier 0. As is mentioned in Section 4.1.9 the number 0 is used as the *System\_Size* flag so if *Gate1* or *Gate2* were to evaluate to 0 it would be replaced by the value of *System\_Size*. Thirdly, in contrary with the decision taken by Massey[19], if the *expnode* expression evaluates to more than *System\_Size* that value won't be coerced to *System\_Size* but the instruction shall just be skipped. The result of these three tackles the problem outlined by Clark and Stepney[14] of unequal

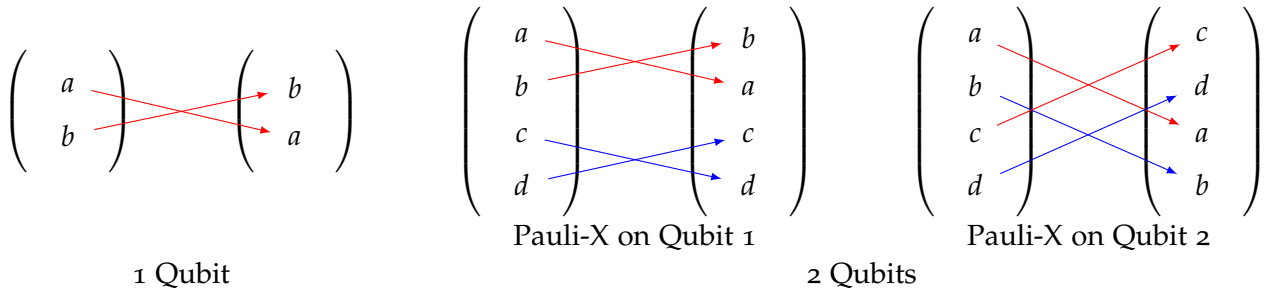


Figure 4.7: Visual Representation of Direct Vector Manipulation Equivalent of Pauli-X Operation

identifier representation. With these three simple rules each qubit identifier and the *System\_Size* flag are represented by two elements. Each qubit identifier is only represented by the identifier and the negative value of the identifier. The *System\_Size* flag is represented by the dedicated *System\_Size* value and if an *expnode* expression evaluates to 0.

A simple circuit optimiser is built into the provided circuit builder. Using static knowledge about which of the gates shown in Table 3.2 are self reversing the circuit builder builds higher efficiency circuits. This is performed by removing elements such as  $\boxed{H} \boxed{H}$  that do not have any affect on the state. However it cannot perform these improvements if there are any gates, even on different qubits, between the removable gates.

The circuits that are produced by a circuit builder are hidden behind an interface. This is to allow third party circuit builders to use their own internal circuit representation and also to allow any optimisations made in future work on this framework to be made without impacting the work of researchers.

The circuit implementations produced provide the quantum circuit as an ordered iterator of quantum gates. The use of an ordered iterator rather than a specific data structure is to ensure that any future optimisation or third party circuit representation is not limited. It also reduced the potential errors involving the interpretation of a more complex data structure.

The circuits also provide a  $\text{\LaTeX}$  representation to allow the circuit to be visualised. The  $\text{\LaTeX}$  representation uses the *QCircuit* package which can be freely obtained at [31].

The circuit implementation, *basiccircuit*, provided in the framework is implemented as a simple ordered list. It uses the Java provided *LinkedList* implementation. The use of a list implementation ensures that the number of gates is almost unlimited until the heap is exhausted. With this implementation there is no difference in the representation between a circuit for a system size of 1 or a system size of 1000. The only affect the system size has is during unitary application to a quantum state, see Section 4.1.11.

#### 4.1.11 Quantum Gates

Any quantum circuit will be a series of quantum gates on specified qubits. The quantum gates provided by the system are hidden behind an interface. This is to ensure that any future optimisation of any gate's implementation cannot interfere with the implementation of any other component of the system.

Each quantum gate is required to provide a unitary matrix but it is not required that the matrix must be used in the application of the gate. For quantum circuits with a high number of qubits, the cost of simulation increases rapidly. This is mainly due to the increase in the size of state vectors and unitary matrices. Matrix multiplication is usually used to apply a unitary operation to a state vector, yet it is an expensive operation.

To improve the performance optimisations can be applied for several gate types. This is most obvious when analysing the operation of the Pauli-X gate. Figure 4.7 shows, with the help of coloured arrows, that the application of a Pauli-X gate on Qubit 1 is essentially a flip of “neighbouring” values. This is also true for a Pauli-X gate on any other qubit, just the definition of a state's “neighbour” is modified with respect to the identifier of the qubit on which the gate is applied.

The use of direct vector manipulation is not required but the interface has been designed to ensure that the gate implementations can use direct vector manipulation or matrix multiplication interchangeably. The interface includes an *apply* method that takes an initial state and returns the state after the application of the gate. This ensures that only the gate implementation needs to understand how the circuit should be applied.



Matrix Manipulation		Direct Vector Manipulation
Phase Gate	W Gate	Hadamard
RX Gate	Custom U Gate	Pauli-X
RY Gate	Controlled U Gate	Pauli-Y
RZ Gate	Swap	Pauli-Z
V Gate		

Table 4.2: Gate Implementation

Each gate must also provide a QCircuit [31] representation of itself in order for the QCircuit representation of the entire circuit to be produced.

The implementations for gates that affect two qubits are hidden by an extended interface to provide access to the identifier of the second qubit but ensures that all standard gate operations are also available.

As has been mentioned, a number of the gates shown in Table 3.2 can be implemented either as a matrix multiplication or a direct vector manipulation. Several of the gate implementations in the framework use the direct vector manipulation and others use the matrix multiplication. Table 4.2 shows which of the provided gates are implemented using the matrix manipulation and which use the direct vector manipulation. Some of the algorithms used by to perform the direct vector manipulation were based on those used in jQuantum[32]. For all gates that use matrix multiplication their equivalent unitary is based on those listed in Table 3.2 but has to adapt to the qubit they are applied to and the system size. The equivalent matrix is calculated in two sections. The initial section is the construction of the matrix listed in Table 3.2. This initial section is calculated in the constructor of each gate. The gate on which the gate is applied is provided as an argument to the constructor of the gate. Unfortunately a Java interface cannot specify the signature of the constructor of implementing classes and abstract classes require the constructor to be fully defined which is not possible when the constructor needs to perform this first section. As a result this cannot be ensured but is expected as part of implementing the interface. The second section is the tensor product of the matrix constructed in the initial phase and identity matrices for all qubits not effected by the gate. This second section is performed as part of the *apply* method and does not impose any restriction on the system size.

To produce the unitary matrices shown in Table 3.2 for the RX, RY and RZ gates the matrices of other gates are used. Equations 4.1 - 4.3 shows the details of these matrix calculations. These equations are taken directly from Lecture 9 of QIP[12].

$$R_x(\theta) = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} X \quad (4.1)$$

$$R_y(\theta) = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} Y \quad (4.2)$$

$$R_z(\theta) = \cos \frac{\theta}{2} I - i \sin \frac{\theta}{2} Z \quad (4.3)$$

The second gate implementation that requires a more detailed summary is the Controlled U Gate. The implementation of the gates cannot assume that the circuit that is produced by an algorithm is “correct”. Placing all the restrictions on the search process that need to be placed on the circuit would have a serious impact on certain search processes. Therefore all the restrictions need to be handled by elements of the framework. If the U gate affects more that a single qubit, possible with custom matrices, it would be possible for the U operation to affect the qubit specified as the control qubit. It was decided that a restriction for Controlled U Gates would be added. The restriction added was that the control qubit cannot be affected by the U operation. If the specified qubit would be affected by the U operation the implementation acts as though it was an uncontrolled U gate. A second consideration for the design and implementation is if U is a custom gate the U matrix can change. This means that the calculation of the matrix that represents the controlled version of U needs to be able to update depending on the test case.

A major implementation detail is how the matrix that represents a controlled version of unitary U is calculated. The calculation changes depending on the relative position of control qubit. If the control qubit is a higher significant qubit the calculation is different than if the control bit is a lower significant qubit.

Equation 4.4 is the calculation if the control qubit is a lower significant qubit. Equation 4.5 is the calculation if the control qubit is a higher significant qubit.

$S1$  is an identity matrix of the size  $2^a$  where  $a$  is the number of qubits that are between the control qubit and effected qubits.  $S2$  is an identity matrix of the size  $2^b$  where  $b$  is the number of qubits with lower significance than the target and control qubits.  $S3$  is an identity matrix of the size  $2^c$  where  $c$  is the number of qubits with higher significance than the target and control qubits.  $U$  is the matrix that defines the operation of the gate under control. This unitary matrix is retrieved using the *getUnitary* operation on an object of the correct class. The *getUnitary* operation returns the matrix created by the first section explained previously in this section.

$$CU = (S3 \otimes ((I \otimes S1 \otimes \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}) + (U \otimes S1 \otimes \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix})) \otimes S2) \quad (4.4)$$

$$CU = (S3 \otimes ((\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} \otimes S1 \otimes I) + (\begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} \otimes S1 \otimes U)) \otimes S2) \quad (4.5)$$

Equations 4.4 and 4.5 are generalisations of the Controlled-U gate that have not previously been published.

With the calculation being flexible for any gate  $U$  there is an additional scenario that must be considered. When the control qubit would be affected by the application of the gate, due to the gates size being greater than 1 or the control and target qubit identifiers being equal, the control is removed. This results in a standard  $U$  gate without any controlling qubits.

As with standard gates, the calculation of the full matrix is performed in two stages. For non-custom gates the matrix calculations in Equations 4.4 and 4.5 can be performed by the constructor up until the matrix  $S3$  which is then performed when the operation is applied to a state. If  $U$  is a custom gate then the full matrix is performed in a single stage as described in Section 4.1.12.

#### 4.1.12 Custom Gates

As has been explained in Section 4.1.4, the framework has a mechanism to specify custom gates. Each custom gate has an identifier between 0 and  $n - 1$  where  $n$  is the number of custom gates defined for the respective test suite, see Section 4.1.4.

To use these custom unitary matrices, the framework provides a gate implementation called *Custom\_Gate*. This is significantly different from all other gates in one particular feature. All gates listed in Table 3.2 have a fixed unitary matrix and only have to adapt for the number of higher significant qubits that are not effected by the gate. This is not so for custom gates, their operation change on a test case by test case basis and therefore both the unitary construction sections mentioned in Section 4.1.11 are performed by the *apply* method.

The *apply* method in the Gate interface has two parameters, one is the quantum state on which the custom operation is applied and one that is the current test case. The test case is provided to extract the custom matrix for the custom gate's identifier set by the constructor. If the custom gate identifier is invalid,  $< 0$  or  $n \geq$ , an array out of bounds exception is thrown. If this exception is thrown, both the *apply* and *getUnitary* methods catch it and behave as though the matrix returned were the identity matrix.

The full unitary matrix, including identity padding for unaffected qubits, is produced using the matrix retrieved from the test case. The state is multiplied by this unitary matrix to produce the return state.

The *getUnitary* method also uses the test case provided to retrieve the matrix from the test case using the custom gate identifier.

The unitary matrices that each test case provides are not guaranteed to affect the same number of qubits. As a result the production of the `QCircuit[31]` representation cannot be guaranteed to represent the exact affect of a custom gate. The framework therefore treats all custom gates as equal. The lowest number of qubits they can affect is 1. As a result the framework shows all custom gates as single qubit gates on the lowest significant qubit that they affect.

#### 4.1.13 Search Engine Parameters

With many search techniques there are a number of parameters that can be configured and altered, sometimes having dramatic effects on the search results. To enable the configuration of these parameters through the framework a suitable, and flexible method had to be introduced into the design. Not all search techniques have the same number of parameters to configure and due to search engines being developed

potentially by different researchers, even if the parameters are the same, their internal representation may be different.

The framework requires a search engine to provide two *search* methods that start the search process. The first takes two arguments and is intended for use within third party applications that want full control over GUI components or do not provide a GUI. The first argument is a boolean array. This is an array with a length equal to the number of gates in the *QuantumInstructionEnum* enumeration. The value at each index indicates whether the element of the *QuantumInstructionEnum* enumeration with the same index should be included in the search and therefore allowed in the result. The second is an Object array. This second array is provided to allow search engine researchers to have search engine parameters that are configured by a third party application. The array cannot be more specific than an Object array. If the array were more specific, a String array for example, all parameters would have to be expressible in the chosen type. This could restrict the parameters available for the researchers to provide. An Object array is so general that any parameters can be expressed. However, it places responsibility on the researcher to provide adequate documentation and explicit error messages with regards to the ordering and the type of the parameters required by the individual search engines.

The second *search* method takes no arguments. It is intended to either launch a search parameter dialog to modify the search parameters. The dialog should start the search process after all parameters have been specified.

This second method ensures the parameter dialog is entirely in under the control of the search engine implementer. It is assumed that the person with best understanding of the parameters provided by a search engine is the implementer. This aims to improve the user interface with respect to the structure principle, see Section 4.6, as the search engine implementer can structure the dialog to group similar parameters. The ordering of parameters in the Object array of the first *search* method cannot be assumed to be a good indication of grouping and there is no explicit separation of groups. As a result if the framework, or a third party client, were to provide an the automated dialog generator the similar parameters may not be correctly grouped.

A second justification is the aim of a “Plug and Play” framework. If the dialog was included as part of third party software, or even by the framework, it would have to be adapted for each search engine implementation or be generated automatically. Automatic generation is unlikely to produce a “principled” user interface, see Section 4.6. The modification of the framework and third party clients for each search engine is not acceptable and is totally incompatible the principle of “Plug and Play”.

#### 4.1.14 Design as a Black Box

The framework is designed to be used as a black box. All external code is allowed access only through the interfaces and final or abstract class types. The implementation of the internal framework functionality is hidden and not required by third party software. This design ensures that third party software shall be loosely coupled with the framework.

Likewise, all internal modules (algorithms, circuit builders, circuits, gates, circuit evaluations, etc) are also designed to view each other as black box modules. This improves the modularity and reduces the coupling between these internal modules.

#### 4.1.15 Algorithm and Circuit Evaluation

When an algorithm is produced by a search engine, the circuit is built for each test set. This circuit then needs to be evaluated against all test cases held in the respective test set. As mentioned in Section 4.1.4 the framework uses a start and final state pair to define the desired operation of the algorithm. Therefore the framework must provide the simulation of an arbitrary circuit on an arbitrary start state.

Due to the design decisions taken previously all the simulation functionality is provided and just needs orchestrating, this is provided by a class implementing the *circuitevaluator* interface. A basic implementation is provided and third party implementations can also be used.

With the circuit interface providing an ordered iterator over the gates in the circuit, and each gate providing an *apply* method the circuit evaluator just needs to work through all the gates in the iterator and calling the *apply* method with the returned state of the previous call. To evaluate the circuit against a test case, the process simply needs to use the start state provided by the test case as the argument of the *apply* method of the first gate. The state returned by the final call to the *apply* method is a state equivalent to the quantum state that would exist if the circuit were to be produced and provided with the same initial state.

The suitability measures have a very simple interface. They only provide a method to retrieve the name of the suitability measure and a method to produce a numerical measure of similarity between two quantum states.

Using the selected suitability measure the final state produced by the circuit is compared to the expected final state defined in the test case. Repeating this process for all available test cases in the current test suite, a numerical and therefore comparable suitability value can be assigned to any algorithm produced by the search engine.

This is one example where the design of the framework ensures that when combining all the elements of the framework (the gate, the circuit, the test cases etc) the logic required is simple. This improves the readability of the code and therefore is likely to improve the quality of the code. With higher readability the number of software bugs are likely to be reduced due to higher levels of understanding.

It is assumed that when a researcher creates a test suite, the test cases that are included are those that they have a higher interest in. As a results it can also be assumed that the researcher would be interested in the final state produced by the best solution found by the search engine for all the start states provided by the test suite. The circuit evaluator interface provides a simple method call to produce these. The *getResults* method returns a test suite data structure that is almost identical to that provided by the search problem used in the search. The only difference is that the final state of each test case in the returned structure is not the desired theoretical state but the actual final quantum state produced by the circuit simulation.

#### 4.1.16 Step-By-Step Evaluation

The framework is required to provide a step-by-step evaluation facility. There are two abstract ways this can be provided. The framework can provide an interactive process that only applies a gate when it receives the command to do so and reporting the “current state”. The second option is to record the “current state” at each point in the circuit.

The main advantage of the first method is that it doesn’t require processing of gate applications if the interactive evaluation does not reach them. A second advantage is that the memory requirement is very small as only the “current state” and the position in the circuit the evaluation is up to. A disadvantage is that for complex gates the application of the unitary operation could take longer than it is acceptable for a user to wait. A second disadvantage is that if the memory required is kept to the minimum, moving backward will also require the application of a gate. However, the gate that needs to be applied is not necessarily the same gate that is applied when working forward. This could make the step-by-step evaluation quite complex to implement. The time required for the reversed gate to be calculated and applied could also be an issue for responsiveness.

The main advantage of the second alternative is that to move forward, and also to move backward, is a simple loading of the respective “current state” from a data structure. This results in the requests from the user to move forward and backward in the circuit taking approximately constant time irrespective of the stepped over gate’s complexity. A second advantage is that taking a step backward in the evaluation never requires the inverse of a gate to be calculated. For gates such as the Pauli gates this is not an issue as they are their own inverse. However this is not the case for arbitrary gates which would require the inverse of arbitrary matrices to be calculated. The main disadvantage of the trace method is that the memory requirements are not constant, they are linear with respect to the number of gates in the circuit but exponentially with respect to the number of qubits in the system. The second disadvantage is that although stepping through a circuit does not require the application of gates to the “current state”, the full trace has to be produced before the step-by-step evaluation can be performed. This means that if the circuit produced is large and complex, the start up and initialisation time could be significant. The requirement for the full circuit to be evaluated irrespective of how many steps are taken in the step-by-step evaluation could lead to trace elements never being reached.

The way the framework has been designed is to include the second of these two options. The additional memory requirements are not likely to be excessive when viewed in respect to the amount of RAM available in the average PC. The main reason for the choice was that the calculation of each “current state” is done once and can be accessed as many times without extra computation required. This should make the responsiveness of the framework much better.

There is also a second major design decision to be made with respect to the step-by-step evaluation, what the initial state should be to produce the trace. There are really two alternatives. The initial state

could either be provided by the user explicitly or the test cases of the search problem could be used as the source of the initial states.

The decision was made to combine the two options. The framework accepts as an input to the step-by-step tracer a test suite data structure. The test suite can either be the same that was provided by the search problem and used by the suitability measure or can be a new test suite that has been created. This allows the client or third party application to provide either or both of the options.

The circuit evaluator interface provides the *getTrace* method that provides a list of test sets. The step-by-step evaluation is performed on a circuit rather than an algorithm. This means that only test cases for a specific number of qubits can be stepped through together, this is why the list is of test sets rather than test suites. It would have been possible to produce the traces for all test sets in the test suite and then returned a list of test suites. This was not implemented as it is likely that a researcher would try and understand each circuit in turn and then look for how they related as a second phase. The step-by-step evaluation is expected to aid in the semantic understanding of the circuits rather than the syntactic comparison. Using the Qcircuit representation would be easier to perform syntactic comparisons. Therefore if a researcher had produced a large test suite, and is only concerned with the understanding of a particular circuit, the wait for the trace to be produced for the full test suite is likely to be significantly longer than for just the relevant test set.

The implementation of the circuit evaluator that provides this trace runs through each test case in turn. After each gate has been applied, the “current state” is set as the final state attribute of a clone of the current test case and is added to the returned data structure. The implementation is purely concerned with providing the trace, no suitability evaluation is performed.

As mentioned in Section 4.1.15, the design of the framework makes the creation of this trace much simpler. The separation of concerns makes the tracing logic much simpler and much less obscured than it would have been if design decisions regarding the gate and circuit interfaces had not been made.

#### 4.1.17 Batch and Distributed Processing

The framework is intended to assist research focussed on the discovery of quantum algorithms through heuristic search. Additionally, the framework intends to assist research focussed on producing heuristic search techniques that can be used to discover quantum algorithms.

Evaluating these search techniques usually requires analysis of multiple runs of the same experiment. The framework does not provide any explicit management for batch processing. However, it does allow for multiple results to be returned by a search. Additionally the contents of these results are not constrained to an algorithm. The framework provides the *SearchResult* class which acts as a container for an algorithm and it’s calculated suitability. An object of any type that extends this class can be returned by the search engine allowing additional statistics to be collected.

The alternative would be for framework to provide a batch processing manager that runs a search engine  $x$  times, collects the search results and statistics, and returns them to the user. Without the batch processing being managed by the framework it is much simpler to collect additional statistics exactly as the researcher requires for their experiment without modifying the framework.

A second consideration concerning batch operations is distribution. Each run of the search is totally independent from any other run. This makes it ideal for distribution as there is no inter-run communication. Providing a batch processing manager would have required specifying how all batches should be processed, in parallel or sequentially. If the chosen approach were that each job would be executed sequentially it could cause frustration for researchers with access to clusters. If the chosen approach were that each job would be run in parallel each researcher would have to have a very powerful PC or access to a cluster. Even if there was an option to select which the batch processing manager would do there are additional considerations. Certain researchers may wish to only use specific distribution methods due to more sophisticated load balancing algorithms. Other researchers may wish to be able to use any available idle PC at their institution which could be difficult using some methods. As a result, specifying the distribution method could lead to modification of the framework for individual researchers which is not in line with the project aims.

This means search engine implementations need to provide any distribution management the developer wishes to use. If search engines were only provided with their selected, or even bespoke, distribution management the reuse of search engines by different researchers could be affected. To reduce this affect

it is recommended that each search engine implements a thread based version as well as any additional versions that exploit distribution.

The decision not to include batch management or statistic collection appears to reduce the assistance the framework gives researchers. It is true that if statistics were collected and batch distribution were managed by the framework it is likely to reduce the development required by some researchers. However, for the remaining researchers it could cause significant problems and potentially increase the required development. The possibility of providing utilities that some researchers would have to update within the source code of the framework was not acceptable.

## 4.2 Provided Tools

The research framework is accompanied by two independent tools. All of the configurations for manager classes and encodings of stored test suites and matrices are stored in XML. The configuration files for the manager classes are relatively simple and are unlikely to be hard to produce manually. However, the encoded test suites and matrices are likely to require longer XML files. To improve the efficiency of the researcher and to reduce the probability of simple, but hard to spot, errors being introduced graphical editors are provided alongside the framework.

### 4.2.1 Graphical Test Suite Editor

A meaningful test set is likely to contain at least 4 test cases, but usually many more. Say for example that the 4 test cases are all for the system size of 2. This results in 4 test cases, each with 2 matrices each containing 4 complex numbers. This would require an XML file to contain 64 floating point numbers, as complex numbers are stored in the XML file as two floating point values. Remember that this is a test suite of just 4 test cases acting on 2 qubits, the number quickly increases with the system size and number of test cases. Writing this XML file by hand in a text editor is not only tedious, but error prone.

To reduce the tedium of the process and the number of errors a graphical user interface is provided. The interface is simple and clear, providing only essential functionality. The interface allows the user to either edit a test suite that is already encoded in a file, or to create a new test suite from scratch for up to 10 qubits. The limit is not imposed by the framework but by the editor. Any test suite containing test cases of greater than ten qubits, the final state is likely to have been calculated by a third party application and manual input of  $2^{11}$  complex numbers for both a starting and final state is likely to introduce errors. Therefore it is recommended that time is spent integrating the third party software and the framework so the test suite can be automatically generated.

A user can add and remove test sets and test set is presented on a separate tab. The state is represented by a  $2^n \times 1$  matrix. This makes the simple tabular visualisation used to edit the starting and final states the natural choice. Two tables are used, one to edit the starting state and one to edit the final state. The complex numbers are not listed as two separate floating point values but in Cartesian form. When a user has created or made the changes, the test suite can be saved to an XML file. The XML file matches the format defined in Section 4.1.7 and also contains comments to improve understanding if it were read using a simple text editor.

The design of the editor follows the user interface design principles, see Section 4.6. The editor is very simple with no extraneous functionality. The use of tabs ensures that the editor design is structured, all related test cases are shown together and all user controls are also grouped whether they act on the test suite as a whole, the current test set, or a test case. The two tables use the same class and each tab is also of the same class ensuring consistent visual representation and component reuse.

### 4.2.2 Graphical Matrix Editor

As with the number of floating point values required to define a test suite, the number required to define a matrix rapidly grows with the number of qubits. This again makes the manual production of the defining XML files again tedious and error prone. The matrix encoding in the file is optimised so that only non-zero values are stored in the file. This improves the process slightly but is can still be seen as rather tedious and error prone.

The framework provides a simple editor that can be used to create the XML files in a much more familiar way. The editor allows users to edit a matrix currently encoded in a file or to create a matrix of up to four qubits from scratch. This is not a limit imposed by the framework. The limit aims to encourage integration with third party applications and therefore to reduce human error. For gates acting on four qubits the matrix contains 256 complex numbers, for gates acting on five qubits this increases to 1024. Manually

inputting 1024 complex numbers into a table with 32 columns and 32 rows makes it very easy to make a mistake. For operations of this size it is likely the matrix would have been calculated by a third party application.

### 4.3 Search Engine Implementations

#### 4.3.1 Q-Pace Inspired Search Engine

Alongside the framework, a search engine inspired by the Q-Pace IV search engine featured in [19] has been developed. This is a Genetic Programming based search engine using the Java-based Evolutionary Computation Research System ECJ[33].

The search engine is provided in both local and distributed versions. The framework requires any distribution to be implemented by the individual search engines. The local version uses separate threads to serially perform jobs. The distributed version uses the JPPF Framework[34] to perform jobs in parallel using all resources available. To ensure that the function of both the local and distributed versions are the same, the two versions are based on the same code parametrised by the container class for the processing, Thread or JPPFJob. The processing unit is implemented as a search engine core that implements the JPPFTask interface, which inherits the Runnable interface. This allows the processing unit to be used both for the local and distributed versions and so ensures consistency.

The JPPF Framework was chosen for the distribution as it is very simple to configure and provides the functionality required in a very easy to understand way ensuring code readability is maintained. Other third-party distribution frameworks were considered as well as the option to produce a bespoke solution. The bespoke solution was rejected as it is completely opposed to the philosophy behind the framework. Other open source frameworks, including Apache Hadoop[35], were not chosen as they provided much more functionality than is required and as a result were more complex. They also required significantly more additional statements within the functional source code to orchestrate the distribution and therefore are more intrusive.

The batch processing element of the framework requires any statistical information to be collected by the search engine. The search engine must also provide a dialog box in which to display this information.

A user of this search engine may often be interested in the likelihood of searches finding optimal solutions. A search terminates when an ideal solution is found or else when the maximum number of generations is reached. The search engine maintains data indicating when searches terminate. For a batch of runs this is rendered in the dialog box with generation number along the  $x$  axis and the percentage of runs terminating along the  $y$  axis. This can be seen in Figure A.14.

#### 4.4 Suitability Measure and Circuit Evaluator Implementations

There are several suitability measures provided with the framework. The two main measures are outlined in this section. Formal definitions can be found in Section B.

It can be seen that neither of the suitability measures introduce parsimony. Parsimony is an important goal. Parsimony is best measured as some function of the size of the algorithm itself and the sizes of the circuits produced by it.

Both a parsimonious and non-parsimonious circuit evaluator implementations are provided. The parsimonious implementation follows Equation 4.6 where  $M$  is the number of test sets and  $N_i$  is the number of test cases in test set  $i$ . The non-parsimonious implementation follows Equation 4.7 where  $M$  is the number of test sets and  $N_i$  is the number of test cases in test set  $i$ .

$$totalsuit = \left( \sum_{i=0}^{M-1} \left( \sum_{j=0}^{N_M-1} fit_{ij} \right) + circuitsize_i \right) + algorithmsize \quad (4.6)$$

$$totalsuit = \sum_{i=0}^{M-1} \sum_{j=0}^{N_M-1} fit_{ij} \quad (4.7)$$

Both circuit evaluators provide scaling with respect to the number of “hits”. A “hit” is a heuristic search term that refers to when the expected value equals the value produced by the search result. In terms of quantum algorithms this could refer to the final state vector equalling the expected state vector provided in a test case. It could be at a lower level and compare the individual vector elements. The framework does not specify how the “hits” value should be calculated by the individual suitability measures. The “hits”

for each test case is returned by the suitability measure in a *Fitness* object, the return type of the *evaluate* method. This scaling works as shown in Equation 4.8.

$$algsuit = \begin{cases} totalsuit, & \text{if } hits = 0 \\ \frac{totalsuit}{hit}, & \text{otherwise} \end{cases} \quad (4.8)$$

Another consideration is the weighting of each test set. Test sets for different system sizes may naturally have different numbers of test cases. For example, if we restrict ourselves to test cases comprising of simple basis states (i.e. without any superposition) a full test set for a system of size  $n$  will contain  $2^n$  test cases. This raises the question as to whether test sets or test cases should be weighted equally. If test cases of different test sets are weighted equally no additional scaling needs to be performed by the circuit evaluator. However, if test sets are weighted equally the circuit evaluator needs to scale the suitability values produced for each test case by the total number of test cases in the respective test set.

The circuit evaluator implementations provided by the framework weight the test cases equally rather than the test sets therefore no additional scaling is performed.

In the remainder of this section the final state vector defined in the test case currently being used to evaluate a proffered solution is referred to as the expected state. The state that is produced by the circuit, constructed using the proffered solution, from the initial state defined in the same test case is referred to as the final state.

#### 4.4.1 Simple Suitability Measure

The simple suitability considers only the probabilities for each state. The phase,  $\theta$  in polar form  $re^{i\theta}$ , is ignored entirely. This emulates measurement where only the probability, the absolute value of the complex number, is taken into account. As the phase, both global and relative, cannot be directly observed it could be seen as irrelevant for measurement based problems. A good example of this is the Deutsch problem. The output of both the original Deutsch, see Section 2.2.1, and the Deutsch-Jozsa, see Section 2.2.2, algorithms are the result of measuring a number of the qubits. Each possible state has a specific probability of being observed. These probabilities are not affected by the phase and therefore it can be proposed that the phase of the state is irrelevant to solving the problem. The simple suitability measure uses the “hits” value to indicate how many elements of the final and expected state vectors are equal, up to an accuracy of 0.00000001.

There are several variants of this simple suitability measure. The first variant is labelled “Non Zero Count”. This variant only increments the “hits” value when the element of the expected state is not 0 and is equal to respective element of the final state vectors, up to an accuracy of 0.00000001. This is for problems where the state vector is sparse, mostly 0. With the standard version, the fitness could appear very high simply due to the scaling performed by the circuit evaluators. This is because an incorrect algorithm that produces many of the 0’s would have a very high hit count even if the probability amplitudes of the non-zero elements were totally incorrect. Through experimentation, it has been found that reducing the scaling effect by only using the hit count for correct non-zero elements to be highly beneficial.

The second variant is almost the compliment to the first variant. This variant is labelled “Zero Focused”. To the best of my knowledge this suitability measure is novel and previously unseen. This variant focuses purely on the elements of the expected state that are 0. All elements not 0 in the expected state are ignored by the suitability measure. This may appear slightly unusual. However, with problems such as the Max Problem, as used in the User Guide in Section A, it makes much more sense than more traditional, all encompassing, suitability measures.

The Max problem is a permutation problem of [0..3] that returns the index of the maximum value, 3. There are four possible output states that would be correct. This is because the input is a four qubit state but the output is only a two qubit state but the circuit must obviously contain four qubits. Using Massey’s[19] representation this is done using “don’t care” values, in state  $|ab**\rangle$  the only values that are interesting are  $a$  and  $b$ , the two  $*$  values are ignored. The problem is that in this representation there are four states that would be seen as correct  $|ab00\rangle$ ,  $|ab01\rangle$ ,  $|ab10\rangle$  and  $|ab11\rangle$ . This raises the question “What probability amplitudes should be assigned to these four output states?” This is an impossible question to answer without knowing the solution circuit. A guess could be made and specify them as an equal superposition, or pick one at random and set it to a probability amplitude of 1. However, if the output state is incorrectly specified it could significantly increase the complexity of the search.



The question effectively being asked to the search engines with the more traditional, all encompassing, suitability measures is “how can these input states be transformed into these output states”. However, if the question to the search engine is reversed it becomes “how can this input be transformed into any state except these output states, the specific output state is not important as long as it isn’t one of these states”. This question makes specifying the Max problem significantly easier. It is known which four states are correct as an answer, and therefore by implication it is known that the other twelve possible state that would be incorrect. Therefore if the four correct states were ignored, and only the twelve incorrect states were used the problem becomes much easier to specify. The “Zero Focused” variant ignores all elements except those with probability amplitudes of 0. The search problem can be defined so that the twelve incorrect states have probability amplitudes of 0 and the four correct states have probability amplitudes of any other value between the natural limits,  $-1$  to  $1$ , as long as they are not 0.

#### 4.4.2 Phase Aware Suitability Measure

This suitability measure includes both the probabilities and phase for each state. Some search problems are not based on the result of measurement. For example, the Quantum Teleportation protocol, see Section 2.2.5, is not based on the probability of observation. The expected result of the protocol is the teleportation of a quantum state, both probability amplitudes and the phases. The same is true for the Quantum Fourier Transform, see Section 2.2.4.

There are several variants of this suitability measure. The first variant is labelled “Non Zero Count” and is analogous to the respective variant of the simple suitability measure, see Section 4.4.1.

The second set of variants are probability scaled variants. Using the polar form of complex numbers,  $re^{i\theta}$ , these variants put emphasis on correct  $r$  values over correct  $\theta$  values. This emphasis was initially proposed by Massey[19]. The underlying principle is to try split the search into two sections, the probability section and the phase section. With additional emphasis on correct  $r$  values, this variant promotes searching for the correct probability before the correct phase.

#### 4.5 Provided Search Problems

As proof of concept the framework provides simple test problems together with a more sophisticated problem, the Quantum Fourier Transform. These provide ready-made case studies for researchers who wish to carry out comparative studies on search techniques but also serve as exemplars for those who wish to create their own. They also serve as test cases for framework verification purposes.

- Pauli X gate on qubit 1
- Pauli Z gate on qubit 1
- Hadamard gate on qubit 1
- Pauli X gate on final, highest significant qubit
- Pauli Z gate on final, highest significant qubit
- Hadamard gate on final, highest significant qubit
- Controlled Pauli X gate with qubit 1 as the target and qubit 2 as the control
- Quantum Fourier Transform on system size of 2 qubits
- Quantum Fourier Transform on system size of 3 qubits
- Quantum Fourier Transform on system sizes of 2 and 3 qubits

The expected final states were calculated using Octave[36] and Quantum Computing Functions(QCF) for Matlab[37]. It would be simple to replace some of these with the problems used during experimentation in Section 6.

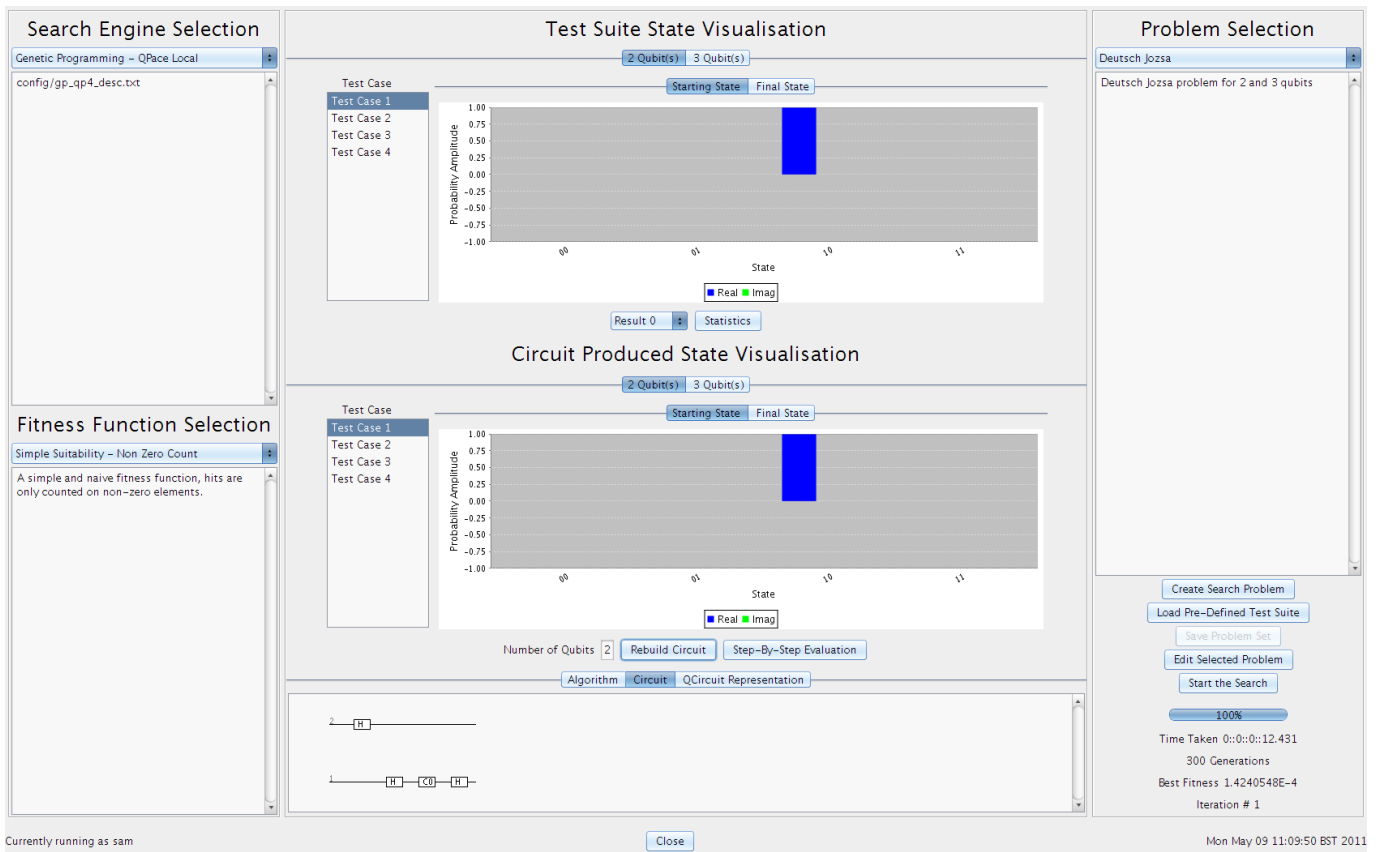


Figure 4.8: Main User Interface - After Search

#### 4.6 Provided GUI

To design the provided GUI I followed the following principles:

- **The structure principle:** The design should ensure that similar components visually resemble each other, related components are grouped and the appearance is consistent to the user throughout.
- **The simplicity principle:** The design should assist the user to understand information presented to them and ensuring common tasks are simple to perform.
- **The visibility principle:** The design should ensure that at any stage all necessary information is provided to to the user.
- **The feedback principle:** The design should reflect the user's previous actions and that all feedback is useful to correct a mistake or make a decision. Any information presented to the user should be in natural language or using predefined conventions.
- **The tolerance principle:** The design should expect incorrect values to be entered and mitigate against their impact. Any update that would change the contents of a configuration file should request confirmation before the file is altered.
- **The reuse principle:** Components should be reused where possible to maintain consistent presentation to the user. Any output provided by the design should be in a reusable format.

These are based on those outlined in [38] and adapted for this project. Where additional principles are used they are explained alongside the design element they apply to.

The design of the main screen of the GUI can be seen in Figure 4.8. The figure shows the main screen after a search has been completed.

Each section of the GUI is explained separately with reference to the principles listed.

#### 4.6.1 Main Window Layout

As can be seen in Figure 4.8 the layout of the main window separates the “dissimilar things” with the use of visible but subtle borders. This meets the *structure* principle. The interface is structured so that the centre of the display contains the information of the highest importance, framed by two control menus. This central panel collates all of the main results of the latest search.

The layout is also intended to take into account the recent move towards wide screen monitors. Wide screen monitors provide a new problem in GUI design. If a GUI fills the screen area and fills it fully with the display of information, it can appear stretched and distorted. If the GUI were to have a single menu along one side it also doesn’t “look right”, it looks excessively heavy on the non-menu side. This is a issue with standard monitors also but in my opinion is exacerbated by wide screen ratios. Although it is not directly related to any of the design principles it is an important property of a GUI to appear well balanced across the available screen area.

With the two menu panels the separation of the configuration options allows for a simple layout of configuration. On the main window the only configuration elements that are available is the selection of the search engine, suitability measure and search problem. The configuration of the search engine and the definition of the search problem is not handled by the main window. This is to ensure that the display does not become overly cluttered, it maintains a simple and clean appearance. This is a result of both the *simplicity* and the *visibility* principles.

#### 4.6.2 Search Engine, Suitability Measure and Search Problem Selection

Before a search can be started, selections have to be made for the search engine, suitability measure and search problem. The selection of these are provided by three drop down lists. The available options for a user to select are limited to the search engines and suitability measures that are registered in the respective managers in the framework. For a user to add a new search engine or suitability measure the respective XML configuration file needs to be manually updated. This was done so as to follow the *simplicity* and *tolerance* principles. It ensures that any selection made by the user is a valid selection.

The selection of the search problem is again provided by a drop down list, following the *simplicity* and *tolerance* principles. The difference is that the creation of a new problem from scratch and using a predefined test suite can be performed within the GUI. Despite this difference, the use of the drop down list still ensures that any selection made by the user is a valid selection. The validity of each individual search problem is checked by the editor dialog boxes described next.

All three of these selections are performed in the same way and ensure that the GUI maintains a level of consistency, meeting the *reuse* principle. When a selection is made, it is shown as selected and its description is displayed in the text area below. The description is provided in the relevant configuration XML file. The description areas are included to meet the *feedback* principles and allow developers of search engines, suitability measures and indeed search problems to provide full text descriptions to the users from within the GUI.

#### 4.6.3 Search Problem Creator and Editor

As mentioned above, the creation of search problems is provided by an on-screen dialog box. The creation and editing of search problems is also provided by a standalone application, see Section 4.2.1. The interface includes an *import from file* function. This allows the user to register a new search problem using a predefined the test suite. This allows researchers to easily register search problems created and distributed by other researchers.

The integrated editor and the standalone application use the same components and overall design. The same components are also used when creating a new search problem as when editing an existing search problem.

Not only are the individual components reused but each dialog box follows the same design layout. This promotes the *reuse* and *structure* principles.

The dialog boxes ensure that the user has entered values for the required fields and that each entry is valid. This implicitly ensures that each selection available to the user in the Search Problem drop down list is a valid option. This follows the *tolerance* principle.

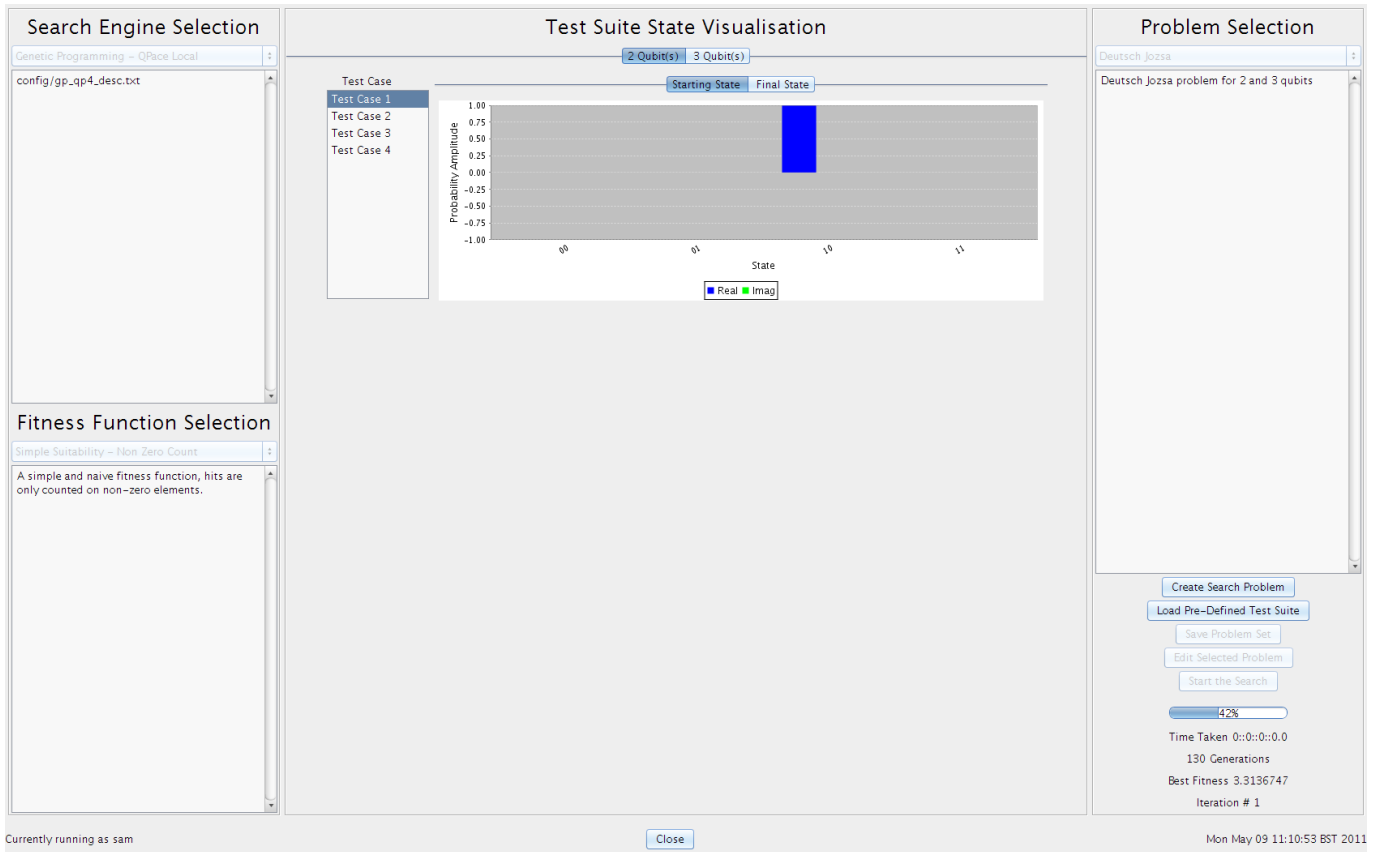


Figure 4.9: Main User Interface - Before Search

#### 4.6.4 Reporting Results

When a search problem is selected, in the central area a visual representation of the test suite is produced. This representation can be seen in Figure 4.9.

After a search has been completed, the final states produced by the realised algorithm are shown using the same representation. This can be seen in Figure 4.8. This provides users with both a quick, simple and visual way to compare states. It is true that small differences may not be noticed using the visualisation. To counter this problem the visualisation allows the user to hover the mouse over each column to get an actual value. This can be seen in Figure 4.10. Another option would have been to provide a table, similar to that used when specifying test cases, containing the probability amplitudes in the selected state. This was rejected as the visual read out provides the same information and is simpler to comprehend than a long list of floating point complex numbers, and therefore follows the *simplicity* principle.

The result of a search is a quantum algorithm rather than the final states. The GUI provides the user three different ways to see the search result. A simple textual listing of the algorithm is provided in the same format as the framework produces on its own. To help with the user's understanding of what the algorithm means a circuit diagram is produced for a user controlled number of qubits. The diagram is produced using the symbols that are shown in Table 3.2. The circuits produced are not just provided as a rendered circuit diagram but also in QCircuit[31] representation so that the circuits can be placed into any publication produced in L<sup>A</sup>T<sub>E</sub>X simply with the use of the QCircuit package.

Providing the three representations meets the *simplicity*, *feedback* and *reuse* principles. The search result is simplified to a simple instruction list that is easily readable. Previous research such as published by Massey[39] present algorithms in a tree structure similar to how they are represented in the search. This resulted in algorithms that are very hard to read. Simplified algorithms alongside automated circuit construction and rendering assist the user to understand how the algorithm works, meeting the *simplicity* principle. The *feedback* principle is met as the circuit diagrams that are produced are done so using widely accepted symbols and conventions for quantum circuit drawing. The *reuse* principle is met with the use of QCircuit to produce circuits in a form that can be included in publications. An alternative would have

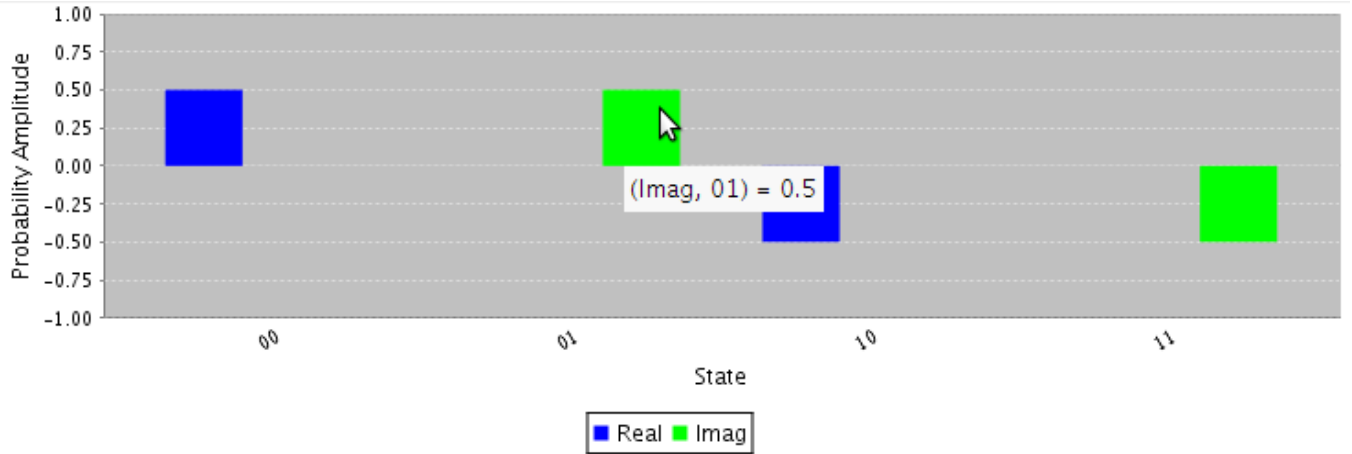


Figure 4.10: Accurate State Readout

been to output the circuit diagram that is drawn by the GUI as an image that could have been included in any, not just  $\text{\LaTeX}$ , publication. I feel the use of QCircuit is a better choice as the user then has control and is able to carry out, if necessary, manual circuit optimisation.

#### 4.6.5 Step-By-Step Evaluator

The way in which quantum algorithms and the circuits they produce work is usually subtle and can be hard to understand by simply looking at the circuit. The framework produces a step-by-step evaluation trace of the current solution on a test suite provided to it, see Section 4.1.16. When implementing the visual component to display the results of the step-by-step evaluation, the origin of the test suite that would be provided to the framework had to be decided. The two options considered were to require the user to specify the test suite from a list of the existing search problems or to automatically use the test suite from the currently selected search problem. The first option was rejected as it seemed unlikely that a user would want to see the trace of any other test suite other than that used during the search. As a result the implementation provides to the framework the test suite of the currently selected search problem.

The step-by-step evaluator is provided in a dialog box rather than integrated in the main frame. This dialog box can be seen in Figure 4.11. This was done so as to focus the user's attention and to ensure that the addition of the functionality did not result in a cluttered GUI. This follows the *structure* and *visibility* principles.

The step-by-step evaluation is performed with respect to a produced circuit. This requires the user to select the number of qubits the circuit should be produced for and the step-by-step evaluation is provided for all test cases of that number of qubits. The framework provide full traces rather than interactive evaluation, and the test cases can be switched between at any step without returning to the start of the circuit.

The dialog box provides a circuit diagram, an initial state selector and a visual representation of the state at the "current step" in the evaluation for the selected initial state. The circuit diagram is produced by reusing the circuit diagram drawn in the results pane of the main window. This ensures that the circuit is represented using the same standards as that shown in the results pane. The only difference is that the "current step" is indicated using a vertical line on the circuit diagram, this can be seen in Figure 4.11. This follows the *reuse*, *feedback*, *visibility* and *simplicity* principles.

The visual representation and initial state selector are the same as those used to report the final states produced for the test suite in the main window. The only difference is that only the test cases for the current number of qubits is shown. The use of the same visual representation ensure the user does not have to understand anything extra to use this functionality and follows the *reuse* principle.

#### 4.7 Summary

The design and implementation details outlined in the sections above cover all the functionality provided by the toolkit. It is intended that the toolkit be released to the research community. As a result the design and implementation was performed with a high level of rigour. Throughout the project third party libraries

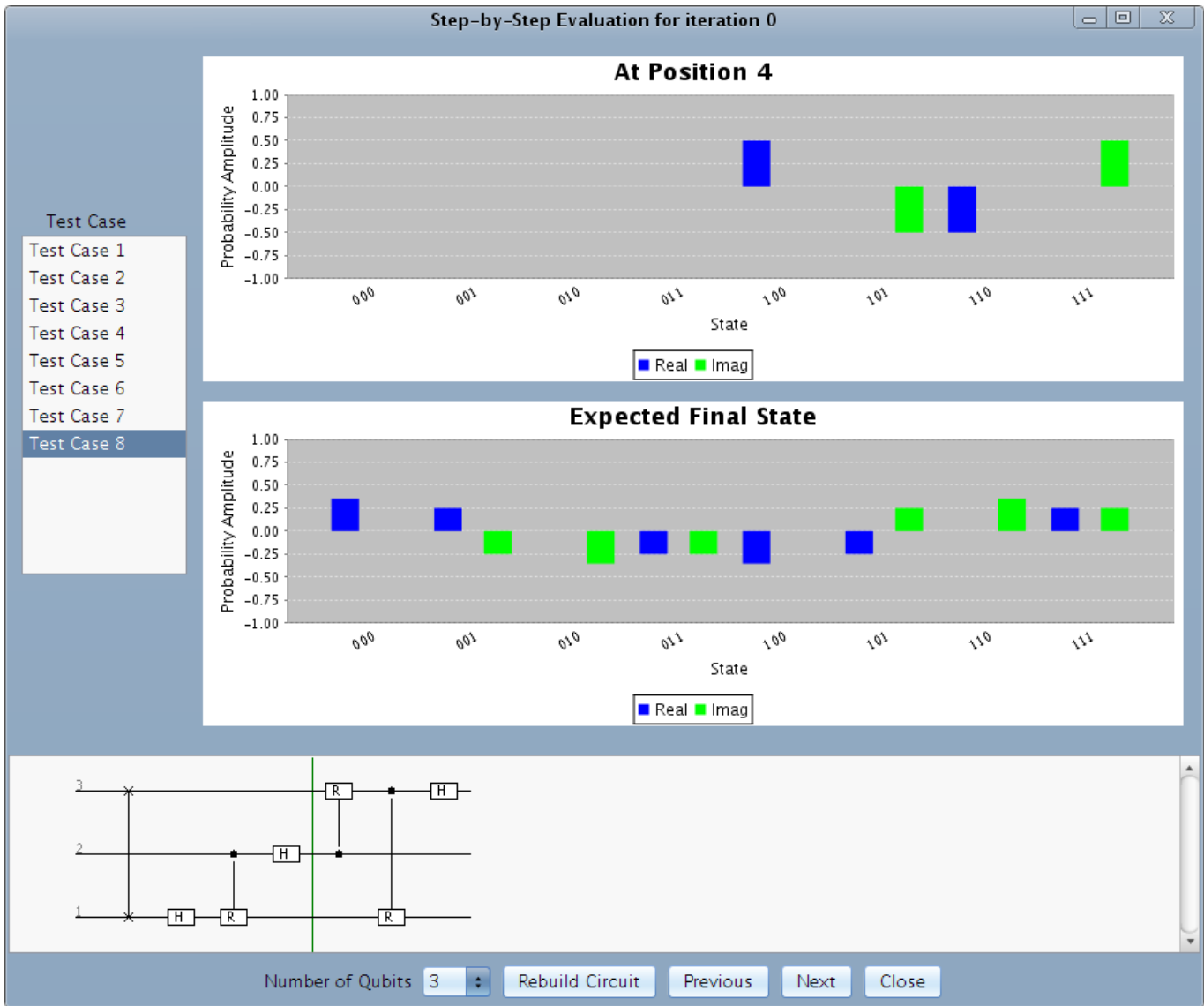


Figure 4.11: Step-By-Step Evaluation Dialog

were employed to carry out specific tasks within the toolkit. A full list of the libraries used can be found in Section D.1. Libraries were used to reduce development risk.

Both the design and implementation were produced following sound software engineering principles. The extensibility of the toolkit has been kept central to the design. The toolkit also ensures that its implementation can scale with respect to the number of instructions in a proffered algorithm and system size, i.e. number of gates in a constructed circuit and the dimensions of the matrices defining the operations of the gates. With the representation of the circuit and algorithm using *List* implementations the toolkit does not restrict the size of the algorithms or circuits that can be produced by the framework. Additionally, as the circuit evaluator progresses through circuits gate by gate rather than computing a matrix to represent the whole circuit this also is not adversely affected by the circuit size. As indicated in the sections above the GUI of the client application was also designed in a principled manner.

The aims for the project remained constant throughout the process. However, the ambition of the project increased quite frequently. During weekly meetings additional functionality was frequently discussed and subsequently included in the toolkit. To accommodate the changing ambitions and increasing functionality an agile approach was taken. This proved very successful. The implementation includes just under 13000 lines of code implementing 751 methods. This is a large code base providing a high level of functionality.

Google CodePro[40] was used to maintain a high level of quality over this large code base. CodePro identifies violations to coding conventions and good practices. Using the core settings the code base

was analysed and updated to address any violations found. For example, exceptions were not originally logged by the framework when they were caught. Subsequently Log4J[41] has been used to implement this logging. Thus an opportunity was taken to embrace good industrial software engineering practice. The envisaged public release of the software fully justifies the level of rigour embraced.

## 5 Testing

### 5.1 Unit Tests

Many of the components created for the framework were tested by unit testing. With the framework created in Java, JUnit 4[42] was used to produce and run the unit tests. Not all components were suitable for unit testing, e.g. those classes providing higher level control rather than simple encapsulated functionality.

#### 5.1.1 Complex Number Implementation Unit Tests

The unit tests were simple with a range of positive, negative, integer and decimal values used in the construction of the complex classes.

The tests of the *toString* and *parseComplex* methods were combined to ensure that the *toString* method produces strings that can be used by the *parseComplex* method.

For the numeric methods, Octave[36] was used to create test oracles.

#### 5.1.2 Matrix Implementation Unit Tests

The Matrix class is based on a third party implementation but due to both its importance to the correct operation of the framework and the extent of the modification made to it, extensive unit testing was performed. Even though the framework only requires the correct operation of unitary matrices the testing was not restricted to unitary matrices. Test matrices were created as real only, imaginary only and real and imaginary complex values with both positive and negative values.

The Matrix unit tests include testing of the MatrixUtil class that provides the encoding and decoding of matrices to XML files and tensor product of two matrices.

For the arithmetic methods, Octave[36] was used to create test oracles.

#### 5.1.3 Gate Implementation Unit Tests

Unit testing the gate implementations is a relatively simple process. The correct operation of the all gates, excluding the custom gates, are statically defined as in Table 3.2. The test oracles are simply expected quantum states that the gates should be produced. To reduce the probability of errors being introduced into the test oracles used by the unit tests, the test oracles were calculated by Octave[36] using Quantum Computing Functions(QCF) for Matlab[37].

QCF does not provide all of the gates defined in Table 3.2. For the missing gates, Matlab functions were created using the definitions in Table 3.2.

To test the custom gate implementation XML files were created with the definitions of some single qubit gates. Custom gates were created using these XML files and all the unit tests for the respective gate were applied. This ensured that the operation of the custom gate is functionally equivalent to the non-custom gates.

Alongside the *apply* operation performed on quantum states, all other methods are also tested by the unit tests. The test oracles are much simpler to define. The *getTarget* method is required to return the integer value set as the target qubit ID in the constructor. Only positive values should be accepted by the constructor as there are no qubits with a negative or zero ID. The returned value of each of the other operations are known either implicitly or explicitly by Table 3.2 and the QCircuit[31] documentation.

#### 5.1.4 Circuit Implementation Unit Tests

The unit tests for the circuit implementation followed the following scenarios:

- Add a gate
- Add a subcircuit
- Add a gate and ensure the Latex representation is correct
- Add a subcircuit and ensure the Latex representation is correct
- Add a gate and ensure the circuit size is correct
- Add a subcircuit and ensure the circuit size is correct
- Add a gate and ensure the gate in the iterator is correct
- Add a subcircuit and ensure the gates in the iterator are correct and in the correct order



0	$2 + \text{SystemSize}$	$\text{LoopVars}[0]$	$2 + \text{LoopVars}[2 + \text{SystemSize}]$
2	$2 - \text{SystemSize}$	$\text{LoopVars}[1]$	$2 - \text{LoopVars}[2 + \text{SystemSize}]$
-2	$2 * \text{SystemSize}$	$\text{LoopVars}[-1]$	$2 * \text{LoopVars}[2 + \text{SystemSize}]$
	$2 / \text{SystemSize}$	$\text{LoopVars}[\text{SystemSize}]$	$2 / \text{LoopVars}[2 + \text{SystemSize}]$
	$2^{\text{SystemSize}}$	$\text{LoopVars}[2 + \text{SystemSize}]$	
	$\frac{\pi}{2^{\text{SystemSize}}}$		

Figure 5.1: Expnode Test Expressions

### 5.1.5 Expnode Implementation Unit Tests

These unit tests concern the implementation of the Expnode context free grammar shown in Table 4.1. The unit tests were based on the expressions that can be seen in Figure 5.1. A collection of values are used as SystemSize including positive, negative, integer and decimal values. LoopVars shall be set as one of a collection of arrays with lengths 0, 1, 2 and 3.

The 0 length array is important to test as search engines are not required to ensure that a loop variable is not used in an algorithm instruction that is not inside the respective loop. I.e. a loop variable can be referenced in the algorithms produced by search by instructions outside of the loop, however it is not legally accessible so this must be handled. If the array is of length 0, or the requested loop variable identifier is greater than the length of LoopVars, the returned value is 0. If the index requested is greater than the length of the array, modulus is used with the array length to produce a valid index. This is defined in Section 4.1.8.

The test oracles are calculated in the test cases to ensure that precision rounding is handled by the test cases.

### 5.1.6 Algorithm Implementation Unit Tests

The unit tests for the algorithm implementation followed the following scenarios:

- Add an instruction
- Add four different instructions
- Add a instruction and ensure the algorithm size is correct
- Add four different instructions and ensure the algorithm size is correct
- Add a instruction and ensure the instruction in the iterator is correct
- Add four different instructions and ensure the instructions in the iterator are correct and in the correct order
- Add a instruction and ensure the printed algorithm is correct
- Add four different instructions and ensure the printed algorithm contains the correct instructions and expressions, and they are printed in the correct order

### 5.1.7 Suitability Measure Unit Tests

There are three suitability measures provided with the framework, see Section 4.4. Each of these needed to be tested to ensure that the implementation was consistent with their definitions.

A series of  $2^n \times 1$  matrices were created and provided to the suitability measure. The theoretical value, the test oracle, was calculated using Octave[36]. The suitability measure tests were not restricted to “correct” quantum states. This is important as suitability measures are general  $\mathbb{C} \rightarrow \mathbb{C}$  functions not just limited to quantum state vectors.

### 5.1.8 Test Suite Unit Tests

The test suite data structure is a combination of three classes not covered by other unit tests. Although the test will cover the three classes used in the test suite data structure, see Section 4.1.4, this is still being classed as a unit test as it is a test of the data structure unit rather than the individual classes.

The tests covered the following scenarios:

- (Three tests) Create a new test suite and insert a single test case for  $1/2/3$  qubits. Check test suite data structure against the oracle.
- Create a new test suite and insert a single test case for 1 qubit and another test case for 2 qubits. Check test suite data structure against the oracle.
- Create two new test suite (A and B), insert a single test case for 1 qubit in test suite A and another for 1 qubit in test suite B, add the test set from test suite B to test suite A. Check test suite A and B data structures against the oracles.
- Create a new test suite and insert a single test case for 1 qubit and another test case for 2 qubits, encode the test suite as an XML file, check the test suite XML file against the oracle.
- Decode a predefined test suite XML file, check the test suite data structure against the oracle.

The scenario consisting of two test suites checks both test suites to ensure that the merge of test sets, and therefore the modification of labels and IDs, does not effect the test suite that is not being modified.

### 5.1.9 Manager Classes Unit Tests

For each of the manager classes, a series of XML configuration files were created for testing purposes only. For the Search Engine and Suitability Measure Manager classes the tests contained several checks:

- Check the list of available implementations against the oracle
- Select each implementation in turn and check against the oracle the class of the object provided by the manager

For the Problem Manager the checks were slightly more in depth due to the returned object containing a test suite data structure. The test suite implementation includes an *equal* method which provides a “deep equality” check. This is used by the unit test to ensure that the object created by the manager against an oracle.

## 5.2 Integration Tests

As mentioned in Section 5.1, the framework contains classes that coordinate the interaction between the functional classes. The integration tests were designed to test these classes. These classes in particular were:

1. The circuit builder implementation - *basiccircuitbuilder*
2. The circuit evaluator implementation - *basiccircuitevaluator*

The approach that was taken was a bottom-up approach. The tests were also performed in the order they are listed. This is because the circuit builder is used by the circuit evaluator and therefore it imposes a dependency.

### 5.2.1 Circuit Builder Integration Tests

The circuit builder integration test combined:

- Gate implementations
- Circuit implementation
- Algorithm implementation

Due to the simplicity of the circuit builder interface, the tests that were carried out were also simple. The circuit builder interface provides two methods. Both methods are intended to perform the same action; to take a quantum algorithm and to return the circuit for the specified system size. The difference between the two methods is that one allows an integer array to be passed as an argument. This integer array is the *LoopVars* array used in the Exprnode grammar, see Section 4.1.8. The second method acts as if an empty array were provided to the first.

To test the two methods, a collection of simple quantum algorithms were produced to include each gate instruction at least once and each of the iterate control instructions at least five times. These algorithms are

passed to the circuit builder and the returned circuits are checked against the test oracles. The test oracles are circuits that represent the circuit that would be created by the algorithm.

The tests are run over system sizes of 1, 2 and 3. Using a code review, the method with the additional integer array parameter will be tested by the other method when the algorithm includes iterate control instructions. This is the reason which each gate was included at a minimum of once but the iterate control instruction were included at least five times.

### 5.2.2 Circuit Evaluator Integration Tests

The circuit evaluator integration test combined:

- Circuit builder implementation
- Suitability measure implementation
- Test suite implementation

The suitability measure that was used for the tests was the Simple Suitability Measure, see Section 4.4.1.

The tests for the circuit evaluator were very similar to those used to test the individual suitability measures, see Section 5.1.7. A series of algorithms were produced alongside a collection of small test suites, containing less than three test cases. The theoretical suitability value of each algorithm was determined and then compared with the value produced by the circuit evaluator.

Two additional tests were produced to test the *getResults* and *getTrace* methods. The test oracles were test suites and arrays of test suites respectively.

## 5.3 Client GUI Testing

The client GUI that is provided alongside the framework was tested using scenario based testing. A scenario was created to verify each of the client requirements. These scenarios were carried out on both Linux and Windows.

## 5.4 Summary

The above summarises the levels of testing carried out. Although most the software did indeed work as intended testing did reveal errors. For example, originally the Controlled-U gate was implemented using only Equation 4.4 and not Equation 4.5, see Section 4.1.11. This was found to be a bug as the Controlled-U gate only worked when the control bit was of a lower significant qubit than the target qubit. Other tests found a bug where with the QCircuit representation when a Controlled-U gate has at least one qubit in-between the control and target qubits. The additional vertical lines to bridge the addition qubits were not originally included.

One error has not been fixed. The first search run after the client is launch does not use the fixed seeds even if time option is disabled. To work around this a search has to be run with the time disabled, once completed enable and then disable the time option again and start the search. 95% subsequent searches using the fixed seeds work as expected until the client is closed and reopened. Without the work around performed only 4 out of 10 repetitions of the Deutsch experiment, see Section 6.1, produced the expected result. With this work around performed 19 out of 20 subsequent iterations of the same Deutsch experiment produced the expected result.

The cause of this error is unknown. The ECJ configuration files automatically produced for both the pre and post work around searches are identical. As only these files should affect the value of the seeds used by the random number generators the cause is unknown and could possibly be an error within ECJ[33].

With almost 13000 lines of code it is reasonable to expect that some bugs remain undiscovered. Additional testing should be carried out, especially focussing on non-functional aspects that were not tested in the process explained above.

It is intended that the toolkit shall be released to the research community. The effort invested in testing would appear to have been worthwhile. The level of testing is consistent with a severe attempt to use a software engineering processes to develop the software artefact. As part of this a traceability matrix was produced and is presented in Section 5.5.

## 5.5 Tracability

Requirement ID	Requirement Title	Full Requirement	Addressed by Design	Addressed by Test
Req:ASE	The framework shall allow researchers to provide search engines for the system to use.	Page 18	Section 4.1.6	Section 5.1.9
Req:ASM	The framework shall allow researchers to provide suitability measures for the system to use.	Page 18	Section 4.1.6	Section 5.1.9
Req:QAO	The solution of a search, a quantum algorithm, shall be presented to the user as a list of instructions.	Page 19	Section 4.1.8	Section 5.1.6
Req:CV	The system shall provide visualisation of the circuit produced by the solution of the search for a system of a user specified number of qubits.	Page 19	Section 4.1.10	Section 5.1.4
Req:TPS	The framework shall be able to be embedded in third party software.	Page 19	Section 4.1.14	Code Review of Client to ensure only interface knowledge is required
Req:DST	The framework shall provide a standardised definition format for users to specify the target of the search.	Page 19	Section 4.1.4	Section 5.1.8
Req:UCF	The customisation of the framework shall be provided through a series of configuration files.	Page 19	Section 4.1.5	Section 5.1.9
Req:PGAI	The framework shall provide implementations of all gates specified in Table 3.2. The framework shall provide algorithm instructions for each of these gates and for the instantiation of the Controlled-U gate with all single qubit gates.	Page 19	Sections 4.1.8, 4.1.11	Sections 5.1.3, 5.1.6
Req:ACS	The system shall provide the iterate control structure and support nested iterate instructions.	Page 19	Section 4.1.8	Sections 5.1.3, 5.1.6

Req:PCA	The framework shall be able to produce a circuit, for any given number of qubits, from a quantum algorithm.	Page 19	Sections 4.1.10, 4.1.11	Code review of both the circuit implementation and all of the gate implementations to ensure the logic does not rely on upper bound to the system size. The only restriction that is present is where $2^{SystemSize} > Integer.MAX\_VALUE$ .
Req:CS	The framework shall provide the simulation of a circuit given an initial state.	Page 19	Section 4.1.11	Sections 5.1.3, 5.2.2
Req:SBSSE	The framework shall provide a way to perform step-by-step evaluation of a circuit given an initial state.	Page 19	Section 4.1.16	Section 5.2.2
Req:SSE	The tool shall provide at least one implemented search engine.	Page 20	Section 4.3	Section 5.1.9
Req:SSM	The tool shall provide at least one implemented suitability measure.	Page 20	Section 4.4	Section 5.1.9
Req:SST	The tool shall provide a number of search targets with known outputs.	Page 20	Section 4.5	Section 5.1.9
Req:SES	The GUI shall provide a user with a selection of search engines to use in a search.	Page 20	Section 4.6.2	Section 5.3
Req:SMS	The GUI shall provide a user with a selection of suitability measures to use in a search.	Page 21	Section 4.6.2	Section 5.3
Req:STS	The GUI shall provide a user with a selection of search targets to be used as the search goal.	Page 21	Section 4.6.2	Section 5.3
Req:STC	The GUI shall provide a way for users to create a new search target without needing to explicitly write a configuration file.	Page 21	Sections 4.2.1, 4.6.3	Section 5.3

Req:STE	The GUI shall provide a way for users to edit the contents of a previously created search target without manual editing of the configuration file.	Page 21	Sections 4.2.1, 4.6.3	Section 5.3
Req:LSTPDC	The GUI shall provide a way to import a predefined search target from a configuration file.	Page 21	Section 4.6.3	Section 5.3
Req:SV	The GUI shall provide a way to visualise any quantum state.	Page 21	Section 4.6.4	Section 5.3
Req:RSR	The GUI shall provide a way to report the search result, a quantum algorithm, to the user.	Page 21	Section 4.6.4	Section 5.3
Req:GCV	Given a quantum algorithm and a system size, the GUI shall produce a visualisation of the resulting circuit.	Page 21	Section 4.6.4	Section 5.3
Req:GSBSSE	The GUI shall provide a way to perform, control and visualise the step-by-step state evolution for an initial state and circuit.	Page 21	Section 4.6.4	Section 5.3
Req:TT	The GUI shall provide user help through the use of tooltips.	Page 22	Implemented for all buttons	Witnessed during Section 5.3
Req:POR	The framework, fully implemented tool and the GUI shall be able to be used on a range of Operating Systems.	Page 22	Property of Java, Section 3	Section 5.3

## 6 Experimentation

The majority of this report concentrates on the development of the toolkit and so far no experimentation has been performed using the implemented toolkit. This section is solely focused on using the toolkit in a series of experiments aimed at solving a selection of previously solved quantum problems. The results reported here are the final results.

The experiments were carried out on a Linux platform and all search parameters were found through the experimentation. The work around detailed in Section 5.4 was performed before the experiments were started. For genetic programming there is not a widely accepted “rule of thumb” on which to base the mutation rate. However for genetic algorithms there is “rule of thumb” which is  $\frac{1}{\text{genomelength}}$  [43]. Values around 0.003846154 are used for the mutation rate during some experiments. This is  $\frac{0.1}{26}$  which is roughly  $\frac{0.1}{\text{numberofinstructions}}$ . This was initially experimented with as an analogy to the “rule of thumb” for genetic algorithms. For some problems it worked particularly well even if the  $\frac{0.1}{26}$  value was only used as an initial value that was altered slightly through experimentation.

### 6.1 Deutsch Algorithm

The details of the Deutsch algorithm is presented in Section 2.2.1. The problem is to determine whether the binary function  $f$  is balanced or constant. The Deutsch algorithm only applies to functions  $f$  with a single input qubit. Each of the four possible binary functions are either balanced or constant.

This search problem uses the toolkit’s custom gate mechanism to define the four test cases, one for each of the possible binary functions. The problem is provided with a single ancillary qubit resulting in a system size of 2 qubits. The input state for each test case is  $|10\rangle$ .

Using this input state could be seen as cheating slightly as the solution is hinted at by the input state. However, by analysing the solutions to similar quantum problems, such as the Deutsch-Jozsa or Grover’s algorithm, it can be argued that  $|10\rangle$  is a sensible suggestion for the input state.

Custom gates are used to provide the oracle functions. The matrices implementing the oracles can be seen in Figure 6.1. The matrices are the two qubit unitary operations of a controlled not acting on the more significant qubit. However, the control is not provided by the value of the lower significant qubit,  $x$ , but the value  $f(x)$ . They effectively represent the circuit shown in Figure 6.2. It can be seen that the circuit of the Deutsch Algorithm, Figure 2.5, includes the circuit represented by the matrices. The circuit in Figure 6.2 is inverted compared to Figure 2.5 due to the numbering of the qubit identifiers and significance in the framework. The matrices have been created taking into account the different qubit numbering so the semantics of the circuit in Figure 6.2 and the respective section of the Deutsch Algorithm circuit in Figure 2.5 are the same.

The start state for each test case is  $|10\rangle$ . For functions 1 and 2, which are constant, the expected final state is  $\frac{1}{\sqrt{2}}(|00\rangle - |10\rangle)$ , equivalent to  $\frac{1}{\sqrt{2}}((|0\rangle - |1\rangle)|0\rangle)$ . For functions 3 and 4, which are balanced, the expected final state is  $\frac{1}{\sqrt{2}}(|01\rangle - |11\rangle)$ , equivalent to  $\frac{1}{\sqrt{2}}((|0\rangle - |1\rangle)|1\rangle)$ . It is simple to see that, taking into account the different qubit numbering, these are the same output states that are produced by the Deutsch Algorithm in Section 2.2.1.

After creating the custom matrices using the provided matrix editor, see Section 4.2.2, and the test suite using the integrated search problem creator, all elements required for the search were complete. The search engine that was used was the Q-Pace inspired search engine, using local threads rather than the JPPF distribution, provided by the prototype implementation, see Section 4.3. The suitability measure that was used was the Simple suitability measure variant that only reports a “hit” for non-zero expected values, see Section 4.4.1.

All instructions were included in the search except Create\_SWAP and Create\_Zero. The default search

$$\begin{array}{cccc}
 \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{pmatrix} & \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
 \text{(a) } f_1 & \text{(b) } f_2 & \text{(c) } f_3 & \text{(d) } f_4
 \end{array}$$

Figure 6.1: Matrices of Custom Gates Implementing the four Possible Functions



Figure 6.2: Matrix Circuit

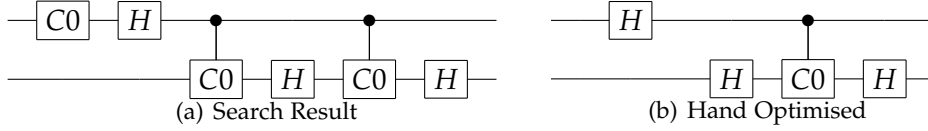


Figure 6.3: Evolved Deutsch Solution

parameters except time was not used to produce the seeds to ensure repeatability and generations were increased to 130.

The full algorithm, Algorithm 8, that was proffered as the final solution is listed in Section C.1.1. Algorithm 5 is a manually evaluated program produced by Algorithm 8 when *System\_Size* is 2. Instructions marked by an asterisk refer to instructions that ignored when creating the two qubit circuit shown in Figure 6.3(b).

---

**Algorithm 5** Evolved Solution for Deutsch Problem

---

Create\_Custom1 on gate 0\*  
 Create\_H on gate 0  
 Create\_CCCustom1 on gate 1.0 and gate 2\*  
 Create\_H on gate 1.0  
 Create\_CCCustom1 on gate 1.0 and gate 2  
 Create\_H on gate 1.0

---

The first instruction is to put a Hadamard gate onto gate 0. Gate zero doesn't exist as the qubit numbering is from 1 upwards. This is an example of when the genetic programming has taken advantage of the environment. 0 is used as the value for the "System\_Size" flag within the system. When an expression tree evaluates to 0 it is effectively instructing that it should be replaced by the system's size.

The circuit produced by this algorithm is shown in Figure 6.3(a). The gate C0 is "Custom Gate 1" provided by the test cases, i.e. the custom unitary operations defined in Figure 6.1.

It is obvious that the search process has found, shown in Figure 6.3(a), is very similar to the circuit that was published by Cleve et al[11] and is a deterministic solution. However, through hand manual optimisation the circuit is transformed into Figure 6.3(b). This can be done as the C0 gate on qubit 2 is evaluated as a Pauli-I gate as there are not enough qubits with higher significance for the 2 qubit custom gates to be applied. It can be shown that the first Controlled-C0 can be removed as qubit 1 is not in superposition. This is an unprecedented result. To the best of my knowledge, this circuit has not been produced using heuristic search by previous research. This is also, to the best of my knowledge, the first time a deterministic algorithm has been found, using heuristic search, for the original Deutsch, single input qubit oracles, problem.

## 6.2 Deutsch-Jozsa Problem

After success with the original Deutsch problem the most natural progression is to multiple qubit Deutsch-Jozsa problem. The basic aim is the same, to decide if a function is balanced or constant. However, unlike when the input cardinality is 1 there exist functions with input cardinality  $n$  that are neither balanced or constant. The Deutsch-Jozsa algorithm only applies when  $f$  is guaranteed to be either balanced or constant.

In this experiment, the test suite used includes the test cases with oracle functions on 1 and 2 qubits. There are twelve test cases for the Deutsch-Jozsa algorithm; 6 balanced two qubit function, 2 constant two qubit functions, 2 balanced single qubit functions and 2 constant single qubit functions. After conducting



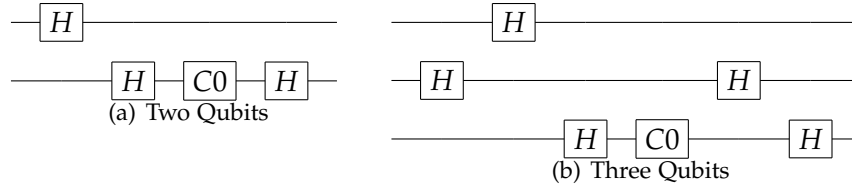


Figure 6.4: Deutsch-Jozsa Solutions

the previous experiment, genetic programming is known to produce a solution to the single qubit problem and it is expected that including these test cases will assist the search for a more general solution.

Custom matrices were made to represent the oracle functions. The matrices for the 2 qubit oracles are shown in Figure C.1 of Section C.2.

The same test cases as used in the previous experiment is for the single qubit test set. For the two qubit oracles, the test cases that are used all have a starting state of  $|100\rangle$ . For  $f_1$  and  $f_2$  the expected final state is set to  $\frac{1}{\sqrt{2}}(|000\rangle - |100\rangle)$ . For  $f_3, f_4, f_5, f_6, f_7$  and  $f_8$  the expected final state is set to  $\frac{1}{\sqrt{6}}(|001\rangle + |010\rangle + |011\rangle - |101\rangle - |110\rangle - |111\rangle)$ .

The search engine that was used was the Q-Pace inspired search engine provided by the prototype implementation, see Section 4.3. The suitability measure that was used was the Simple suitability measure variant that only reports a “hit” for non-zero expected values.

Initial experimentation showed that this search problem was significantly harder than the previous experiment. The final search parameters, including the enabled instructions, used are shown in Figure C.2 in Section C.2.1. The time was not used as the seeds so this experiment is reproducible.

The full algorithm, Algorithm 9, that was found is listed in Section C.2.2. Algorithm 6 is the manually evaluated program produced by Algorithm 9 when *System\_Size* is 2. Instructions marked by an asterisk refer to gates that have been removed from the two qubit circuit shown in Figure 6.4(a) by the circuit builder, not manually, to improve the circuits efficiency.

---

**Algorithm 6** Program to Produce the Solution for the Two Qubit Deutsch-Jozsa Problem

---

```
Create_H on gate 2.0
Create_H on gate 3.0
Create_H on gate 1.0
Create_Custom1 on gate 1.0
Create_H on gate 2.0
Create_H on gate 3.0*
Create_H on gate 0.0*
Create_H on gate 1.0
```

---

Figures 6.4(a) and 6.4(b) show the circuits produced by the Algorithm 9 for two and three qubits respectively. The full circuits that have not been optimised are shown in Section C.2.3. The gate C0 is “Custom Gate 1” provided by the test cases, i.e. the custom unitary operations defined in Figures 6.1 and C.1. The C0 gate rendering has also been modified to take into account the knowledge of gate size that the framework cannot assume when producing the QCircuit representation. The full circuits in Section C.2.3 are exactly as the framework produced them and do not show the actual gate size. The controls shown in the circuits of Section C.2.3 are also removed as they cannot be on a qubit affected by C0, see Section 4.1.11. The algorithm is almost scalable. The circuit produced for a system size of 4 is shown in Figure C.5 in Section C.2.3. The only problem is that the Hadamard gate on the most significant qubit is the wrong side of C0.

The solution found to this problem is quite remarkable. The circuits produced are exactly the same as those presented by Cleve et al.[11]. This is also unprecedented. To the best of my knowledge this solution has not been found through heuristic search. The previous attempt made by Spector *et al.*[22–24] also only used a single call to the oracle. However, Spector *et al.* placed that restriction on the search and were only searching for the circuit for the 3 qubit problem and the result was only probabilistic. No such restriction was placed during the experiment to produce Algorithm 9.



Figure 6.5: 0..3 Max Permutation Solution

### 6.3 0..3 Permutation Max Problem

The 0..3 permutation problem is used by Massey[19] to evaluate Q-PACE III. The problem is quite simple, given a permutation of the numbers 0..3 give the index of the maximum value, 3. Due to the simplicity of the problem it was chosen for the user guide produced for the client provided with the framework, see Section A.

Interestingly a deterministic solution was not found by Massey for all 24 possible permutations. Only a probabilistic solution was discovered. This is the first experiment where a probabilistic solution is expected. Massey specifically searched for a probabilistic solution with a fitness function designed as such.

The search engine that was used was the Q-Pace inspired search engine provided by the prototype implementation, see Section 4.3. Initially, the suitability measure that was used was the basic Simple suitability measure variant. The search problem was specified with all 24 test cases defined in Figure A.1.

Initial results were disappointing and provided little useful information. The solution that is presented by Massey includes the use of the Controlled-Controlled-Not(CCN) gate that is not included in the requirements and therefore was not implemented. The decision was made to extend the framework past the original requirements and include the CCN gate. To reduce the impact on the existing framework, the gate was implemented as a Controlled-U gate where U was a Controlled Pauli-X gate. The CCN gate obviously requires 3 qubits identifiers, the third qubit identifier was provided by casting the phase value to an integer. The addition of the CCN gate required additional logic to ensure the control qubits were set with the control qubit closest to the target qubit set as the control qubit of the inner Controlled Pauli-X gate. This is necessary for the matrix calculation. If this logic were not performed the matrix calculated could be incorrect as the control of the outer Controlled-U gate would have to be inserted into the matrix of the inner Controlled Pauli-X gate rather than the other way round.

Even with the addition of the CCN gate the results were not that impressive. A different approach was taken to try and improve the results. Reflecting on the difficulties encountered when specifying the desired output state the Zero Focussed variant of the Simple suitability measure was developed. The difficulties encountered and the Zero Focussed variant are described fully in Section 4.4.1. With the new suitability measure implemented the results were astounding.

The final search parameters used are shown in Figure C.6 in Section C.3.1.

Algorithm 7 was proffered as the solution. The circuit produced when *System\_Size* is 4 is shown in Figure 6.5(a). This circuit contains the gate  $\text{—}\boxed{\text{XT4}}\text{—}$  which represents the CCN gate. The symbols inside this gate indicate certain properties of the CCN gate. The X symbol indicates that the operation under control is the Pauli-X gate. The T4 shows that the target of the CCN is qubit 4 that means that qubit 2, the qubit on which the gate is shown, is a control qubit. If this were C4 instead it would indicate that the second control qubit would be qubit 4 and qubit 2 would be the target of the CCN gate. The circuit shown in Figure 6.5(b) is manually redrawn to show the circuit with both control qubits and the target qubit shown.

---

#### Algorithm 7 Program to Produce the Solution for the Max Permutation Problem

---

Create\_X on gate 0

Create\_CCX on gate 4.0 and gate 1.0 with phase 2.0

---

The solution in Figure 6.5 is identical to that found by Massey[19]. However, the solution found by Massey has been hand optimised whereas the solution in Figure 6.5 is how the framework found it without any manual optimisation required. Figure 6.5(b) has been redrawn but no optimisation has been performed.

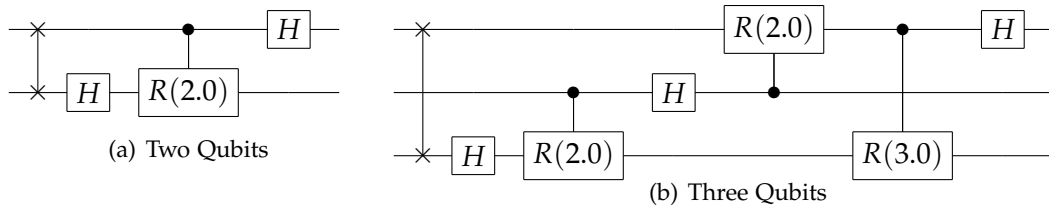


Figure 6.6: Evolved Quantum Fourier Transform Solution

The measurement probabilities for each index can be found in Section C.3.3.

## 6.4 Quantum Fourier Transform

The most impressive result presented in Massey's thesis[19] and subject of the subsequent paper[39] was the evolution of a human competitive algorithm to produce the Quantum Fourier Transform(QFT). With the QFT acting as such a central part of Shor's Algorithm, see Section 2.2.4, it is a very important element for quantum algorithms.

To try reproduce the results found by Massey, the test cases for QFT acting on two and three qubits were initially used. These were created using Octave[36] using Quantum Computing Functions(QCF) for Matlab[37]. The input states that were used were all twelve possible non-superpositional states. All test states are provided in Section C.4.

The search engine that was used was the Q-Pace inspired search engine provided with the framework, see Section 4.3. The suitability measure that was used was the "Phase Sensitive" suitability measure, see Section 4.4.2. The final search parameters used are shown in Figure C.9 in Section C.4.1.

As can be seen in Figure C.9 the search only included a few of the instructions. However, the only instructions that are not included in this experiment but were in the experiment carried out by Massey are the Pauli-X instructions. However, the suitability measure used by Massey was specifically for the QFT and included a term that punished a solution with an incorrect number of swap gates. The suitability measure used in this experiment was not modified for the QFT in any way.

The algorithm produced as the result of the search can be found in Algorithm 11 in Section C.4.2. This algorithm has been slightly optimised by removing control structures with empty bodies so it will fit on a single page. The circuits produced by this algorithm for system sizes of 2 and 3 qubits can be seen in Figure 6.6. These circuits have not been manually optimised and are exactly as the algorithm produced them.

These circuits reproduce the action of the QFT correctly for system sizes of 2 and 3 qubits. However, it does not correctly scale to a system size of 4. The circuit can be seen in Figure C.10(c) in Section C.4.3 and it shows similarity to the correct circuit.

In a final attempt to produce a fully scalable solution an additional test set with four tests cases for a system size of 4 qubits was added to the search problem. The search settings and enabled instructions were kept the same. This was unsuccessful and further experimentation is required.

## 6.5 Unintended Functionality

The experimentation also showed how easy the framework makes creating and editing search problems both within in the client and using the standalone editor. During the experimentation each method provided by the toolkit to create a new search problem and the respective test suite were used as part of this evaluation. Some test sets were only partially specified so the test suite editing facility could also be used. Each of these methods were very simple to use showing how easy the toolkit makes configuring search problems.

During the experimentation the Step-By-Step Evaluation facility, see Section 4.1.16, was found to provide additional unintended functionality. A piece of functionality that was going be put as a potential area of future work was post-search evaluation of additional test cases not included in the test suite used during the search. This could be useful as simulation is quite expensive so if large test suites are used during the search it can have a significant impact on the speed of the search. Whereas a larger test suite can be evaluated post-search with a smaller impact. During the search the test suite is evaluated against many potential individuals but post-search it would only have to be evaluated against the proffered solution.

The functionality is provided by the framework and during the experimentation it was found that it is accessible through the client.

Within the client this can be performed by creating two search problems. The first would include only those test cases that should be used during the search. The section would include these test cases and additional test cases. After selecting the first search problem and performing the search, the second search problem is then selected and the Step-By-Step dialog is launched. This evaluates the newly selected test suite against the algorithm proposed as the result of the search. Admittedly this is not convenient but it was not developed to provide this functionality.

This functionality was found to be quite useful when evaluating the generality of the algorithms produced. A larger test suite, with additional test sets for larger system sizes, can be used post-search to analyse whether the algorithms produced have the ability to scale.

## **6.6 Summary**

In the sections above the results of several experiments are presented. All these results are positive despite the solutions produced not necessarily scaling to larger system sizes. The similarity between the produced solutions and the best known solutions is rather astounding. Evolutionary approaches are usually associated with solutions that are not particularly easy to understand. Additionally genetic programming normally produces massively bloated solutions. Admittedly the algorithms can be seen to fit the bloated description but the circuits are relatively minimal. This is quite surprising with the simplicity of the optimisation performed by the circuit builder.

The results of the experimentation include several solutions (Deutsch and Deutsch-Jozsa) that, to the best of my knowledge, have not previously been produced by heuristic search. These are very impressive results.

Not only are these results impressive, the experimentation process has highlighted several successes of the toolkit. The toolkit was intended to be easily extendible and for configuration to be simple. Through experimentation these intentions were seen to be realised by the toolkit. The addition of the zero focused suitability measure and even the addition of the Controlled-Controlled-Not gate during the 0..3 Permutation Max experiment was very simple. To reach the configurations presented for each experiment in this report required additional experimentation with different configurations. The on-screen dialog made this very simple, much easier than directly editing configuration files.

## 7 Evaluation and Future Work

### 7.1 Have the Requirements Been Met?

The aim of this project was to provide an easily extendible toolkit to assist research into the discovery of quantum artefacts through heuristic search. Previous research was deemed inefficient with bespoke development of implementations to perform peripheral tasks, such as artefact and state representation, and circuit simulation, carried out by each individual researcher. The toolkit was intended to provide a more efficient environment for such research by providing implementations of these peripheral tasks.

As described in Section 3, the toolkit produced by this project is split into three distinct components. The first component is a framework that aims to provide the core research environment. It is in the form of a library that can be embedded into any third party application. The framework provides the orchestration between implementations of the peripheral tasks and the user provided search engine, suitability measure and search problem. The framework provides two standalone graphical applications to create and edit test suites, and to create and edit matrices used to define custom unitary operations.

The second component is a prototype that provides implementations of a genetic programming based search engine, multiple suitability measures and multiple search problems. The search engine is inspired by Q-Pace IV[19] and Implemented using the Java-based Evolutionary Computation Research System ECJ[33]. A total of 10 suitability measures are provided. The suitability measures are from two main categories. The first category contains all the Simple suitability measures, see Section 4.4.1 for more details. These consider only the probabilities for each state being compared. The second category contains all the Phase Aware suitability measures, see Section 4.4.2 for more details. These include both the probabilities and phase for each state the state vectors being compared.

The third component is a client application that provides the framework with a standalone graphical user interface. The framework is not a standalone application and needs to be embedded within another application. Without providing this client a third party application that embeds the framework would have to be produced before any improved efficiency could be realised. This client provides a number of research aids including rendered circuit visualisation, quantum state visualisation and graphical step-by-step evaluation of quantum circuits.

In Section 3, 27 requirements for the toolkit were specified. Section 5.5 provides a method to trace the requirements into the design and implementation with reference to the tests verifying the requirements. The traceability matrix shows that each of the 27 requirements has been met by the design and implementation and has been verified by the testing process.

In a formal sense we should conclude that the requirements agreed have been delivered. Now it is appropriate to take a step back and carry out a more holistic evaluation, assessing the strengths and weaknesses of the work done.

### 7.2 Strengths of the Toolkit

There are several strengths of the toolkit produced. One of the main goals of the toolkit is extendibility; the design of the framework is heavily focused on this.

During the experimentation summarised in Section 6 this flexibility was tested. The Zero-Focused Simple suitability measure, explained in Section 4.4.1, was not originally developed. During the experimentation of the Max Permutation problem, see Section 6.3, the basic variants of both the Simple and Phase Sensitive suitability measures did not produce any useful results. As a consequence the Zero-Focused Simple suitability measure was developed and added to the prototype.

The addition of this suitability measure was straight forward. The class implementing the suitability measure was developed and the process required to add the new suitability measure to the configuration file took roughly 2 minutes.

The experimentation also showed how easy it was to create and edit search problems. During the experimentation process, both the client and standalone editors were used to perform these tasks. Both methods were clear to use and the reuse of components made the transition between the two effortless.

### 7.3 Areas of Improvement and Future Work

There are areas of the framework that could be the focus of future work.

Some of the circuits produced during the initial experimentation phase contained sequences of gates that didn't result in any change to the state. This is not ideal and would require manual optimisation to produce a circuit with only functional gates. There are a few rules that could be included in an higher

efficiency circuit builder to reduce the redundancy of circuits built. The provided circuit builder can perform simple improvements by removing sequences such as  $\text{---}\boxed{H}\text{---}\boxed{H}\text{---}$ . However, the current rules are very simple and the circuit builder only has a memory of a single gate, i.e. the last added gate, so the sequence  $\text{---}\boxed{X}\text{---}\boxed{H}\text{---}\boxed{H}\text{---}\boxed{X}\text{---}$  can currently only be improved to  $\text{---}\boxed{X}\text{---}\boxed{X}\text{---}$  which obviously isn't optimal as this sequence can also be removed.

The second area would assist with understanding. When constructing a circuit from an algorithm some of the instructions are ignored, see Section 4.1.10, with the result that the algorithm contains instructions that did not contribute to the specific circuit. I refer to these as "inactive" instructions and to the instructions that are not ignored as "active" instructions. I use the term "Active Algorithm" to refer to an algorithm that only contains "active" instructions. As the "Active Algorithm" produces the evaluated circuits only "active" instructions contribute to the proffered solution. For a researcher, understanding the "Active Algorithm" is more important than the full algorithm. It is important to note that the "Active Algorithm" is likely be different for each system size. The framework could easily be extended to provide this as part of the result.

A third area of future development would be an extension of the circuit optimisation rules mentioned as the first area of improvement. Various problems have certain properties that cannot currently be expressed to the framework. One such search problem is the quantum teleportation protocol, see Section 2.2.5. Due to the spacial separation of the qubits held by Bob and Alice certain operations cannot be performed by the solution. For example a controlled gate acting on Bob's qubit controlled by one of Alice's qubits has to be modelled to represent the required measurement and classical communication to communicate to Bob whether the operation should be applied. As a result any superposition that contained the controlling qubit is effectively collapsed due to the measurement. To model this without explicit measurement would require rules to ensure further operations cannot be applied to but can be controlled by Alice's qubit used to control the operation on Bob's qubit.

It is important to note that any rules to improve efficiency or enforce properties of individual problems should only be applied at the circuit construction phase. Enforcing these rules at the search phase, for example through strongly typed genetic programming, could prune useful areas of the search space. Although certain algorithms would not meet the restrictions they could form part of the search path to a desired solution. When enforced by the circuit builder all algorithms are valid and the rules are applied to ensure all circuits produced are also valid.

The fourth area of improvement would be the customisation of suitability measures. Currently the search engine is able to provide customisation via either an on-screen dialog or parametrised *search* method. Through experimentation it was found that this ability to customise the search engine parameters was very useful. However, being able to configure the suitability measures, such as whether hits were counted or component weighting, only at the source code level was quite inconvenient. It is proposed that the suitability measures provide on-screen dialog and dedicated customisation methods.

The final improvement proposed is potentially quite complex. The idea of saving algorithms and providing algorithm editing facilities sounds quite simple. This could be provided by simply using the serialisation provided by Java however this would not match the use of XML throughout the rest of the framework and is potentially less compatible with third party applications. Therefore an XML file based solution would be preferable. However, this requires a parser to be able to interpret the specification of an arbitrary *expnode* expression. Additionally, to provide editing facilities an algorithm editor would have to be designed. This again sounds quite simple. However due to the potentially bloated algorithms produced by search techniques the editor assist the user with understanding an algorithm in a similar way the Step-By-Step evaluator assists the user to understand a circuit. Step-By-Step algorithm evaluation is a possible techniques that could be incorporated. It also could include functionality usually found in modern Integrated Development Environments like structural folding, content assistance such as bracket matching, and a dedicated *expnode* expression editor.

## 7.4 Experimentation Summary

An advantage of producing a usable toolkit before the end of the project was that time could be dedicated to using the toolkit to solve a selection quantum problems. The results were very encouraging.

The Deutsch and Deutsch-Jozsa experiments produced results that were particularly impressive, see Sections 6.1 and 6.2. The algorithms that were produced had not previously been produced by heuristic search. Both resulted in deterministic results, the best previous heuristic search result for the Deutsch-Jozsa

problem was a probabilistic circuit and the single qubit Deutsch problem had never previously been tackled by heuristic search. These results are human competitive and previously unseen as the result of heuristic search. The Quantum Fourier Transform and Max Permutation experiments were impressive but not unprecedented.

All the experiments showed one property that was particularly impressive yet a second that is quite disappointing. The impressive property was how small the solutions produced were. The circuit builder is only implemented with a very simple optimisation rule yet still the circuits produced were still very small. However, the lack of scalability is relatively disappointing. The main advantage of an algorithm compared to a program is the application over an arbitrary system size.

It may be that the search problems were defined with too many test cases concentrated in too few test sets. Further experimentation would have to be performed to verify this theory.

Future work on the search engine provided in the toolkit could be undertaken to potentially address this issue. I suggest the introduction of multiple populations. Each population would concentrate on forming a solution for system sizes up to a value  $x$ . The value  $x$  would be different subpopulation. Migration would be allowed of highly suitable individuals from a subpopulation to a second subpopulation with a higher value of  $x$ . This is a similar idea to coevolution where the problem evolves alongside the original population with effectively an “inverse” suitability measure. This is based on the idea that a solution for smaller system sizes can provide useful stepping stones when searching for a solution to cover larger system sizes.

In my opinion this process of effectively increasing the difficulty of the problem would, in the many cases, be more effective than attempting to solve the full problem in a single population. This opinion is based on my theory that this subpopulation idea would effectively use the “lower” subpopulations to prune the search space for the “higher” subpopulations. With the search space increases each time a qubit is added it would be easier to perform this pruning on the search spaces of lower system sizes. Despite the search space increasing with system size it would already be partially pruned due to the results of “lower” subpopulations.

## 7.5 Project Evaluation

In conclusion the project as a whole has been successful. The high level project aim was to provide a toolkit to the research community that provided implementations of tasks peripheral to research into quantum algorithms. The toolkit produced does this. With a code base of just under 13000 lines of code it shows the level of development that is no longer required by each researcher. Not only does the toolkit provide the implementations of peripheral tasks but also implements features that are likely to be useful to users that they may not have developed themselves as part of their own bespoke software.

The literature survey highlighted several issues with previous research such as assumed knowledge and over representation of values in the search space. Throughout the design and implementation of the toolkit these issues were addressed where possible. The toolkit was implemented so that irrespective of the system size each qubit is equally represent in the search.

Viewed as a software engineering project the success is quite evident with a fully implemented toolkit developed on time that meets all the requirements and provides a high level of functionality much greater than envisaged initially. The software engineering process was rigorous and as a result the software produced is of a high standard. Despite the continually increasing ambition throughout this project the fundamental design of the toolkit was largely unaltered from the start of development. Where alterations were made they were not due to functional incompatibility with the original design but to improve the software engineering qualities such as modularity and reuse with respect to the additional functionality.

Viewed as a research project the success is also evident from the results of the experiments carried out in Section 6. These experiments were carried out in a manner that is reproducible and used only generic suitability measures provided with the toolkit. As part of this experimentation the 0..3 permutation experiment provided the inspiration for a suitability measure which is, to the best of my knowledge, previously unseen in heuristic search, the Zero Focused suitability measure, see Section 4.4.1. This suitability measure is now included as part of the toolkit and it is thought it will be useful for other quantum search problems beside the 0..3 permutation experiment. Additionally, several of the results presented are, to the best of my knowledge, unprecedented for heuristic search. Not only are the individual results unprecedented but the collection of results produced using the same software is, to the best of

my knowledge, unprecedented. The different experiments may have used different suitability measures but they were all be specified, configured, run and analysed one after another all from within the same application.

The toolkit has been made freely available at <http://code.google.com/p/mengquantumalgorithm/>. I hope it facilitates further research in the area of quantum algorithm synthesis via heuristic search.



## Bibliography

- [1] P. Gawron, "File:bloch.png - quantiki | quantum information wiki and portal," <http://www.quantiki.org/wiki/File:Bloch.png>.
- [2] Qcircuit tutorial. [Online]. Available: <http://www.cquic.org/Qcircuit/Qtutorial.pdf>
- [3] R. Feynman and P. W. Shor, "Simulating physics with computers," *SIAM Journal on Computing*, vol. 26, pp. 1484 – 1509, 1982.
- [4] D. Deutsch, "Quantum theory, the church-turing principle and the universal quantum computer," *Proceedings of the Royal Society of London*, vol. 400, pp. 97–117, 1985.
- [5] P. W. Shor, "Polynomial time algorithms for prime factorization and discrete logarithms on a quantum computer," *SIAM J. Sci. Statist. Comput.*, vol. 26, p. 1484, 1997.
- [6] K. Wloch and P. J. Bentley, "Optimising the performance of a formula one car using a genetic algorithm," in *In Proceedings of Eighth International Conference on Parallel Problem Solving From Nature*, 2004, pp. 702–711.
- [7] Nasa - antenna design. [Online]. Available: <http://ti.arc.nasa.gov/projects/esg/research/antenna.htm>
- [8] P. Dirac, *The principles of quantum mechanics*. Oxford University Press, 1958.
- [9] E. Schröginger, "[A translation by John D. Trimmer] 'The Present Situation in Quantum Mechanics'," <http://www.tu-harburg.de/rzt/rzt/it/QM/cat.html>.
- [10] P. W. Shor, "Progress in quantum algorithms," *Quantum Information Processing*, vol. 3, pp. 5–13, October 2004. [Online]. Available: <http://portal.acm.org/citation.cfm?id=1032132.1032149>
- [11] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca, "Quantum algorithms revisited," in *Proceedings of the Royal Society of London A*, 1998, pp. 339–354.
- [12] Qip module page. [Online]. Available: <http://www-course.cs.york.ac.uk/qip>
- [13] D. Deutsch and R. Jozsa, "Rapid solution of problems by quantum computation," *Proc Roy Soc Lond A*, vol. 439, pp. 553–558, October 1992.
- [14] S. Stepney and J. A. Clark, "Searching for quantum programs and quantum protocols: a review," 2007.
- [15] L. K. Grover, "A fast quantum mechanical algorithm for database search," 1996.
- [16] C. H. Bennett, E. Bernstein, G. Brassard, and U. Vazirani, "Strengths and Weaknesses of Quantum Computing," 1996.
- [17] E. Rieffel and W. Polak, "An Introduction to Quantum Computing for Non-Physicists," *ACM Comput. Surv.*, vol. 32, pp. 300–335, September 2000. [Online]. Available: <http://doi.acm.org/10.1145/367701.367709>
- [18] G. Brassard, P. Hoyer, and A. Tapp, "Quantum Counting," 1998.
- [19] P. Massey, *Searching for Quantum Software*. University of York, 2006.
- [20] P. Massey, *Evolving Quantum Programs and Circuits*. University of York, 2000.
- [21] D. E. Goldberg, *Genetic algorithms in search, optimization and machine learning*, Goldberg, D. E., Ed., 1989.
- [22] L. Spector, H. Barnum, H. Bernstein, and NewAuthor4, "Genetic programming for quantum computing," in *Genetic Programming 1998 - Preceedings of the Third Annual Conference*, 1988, pp. 365–374.
- [23] L. Spector, H. Barnum, H. Bernstein, and N. Swamy, "Finding a better-than-classical quantum and/or algorithm using genetic programming," in *Evolutionary Computation, 1999. CEC 99. Proceedings of the 1999 Congress on*, 1999.

- [24] L. Spector, H. Barnum, H. J. Bernstein, and N. Swamy, *Quantum computing applications of genetic programming*. Cambridge, MA, USA: MIT Press, 1999, pp. 135–160. [Online]. Available: <http://portal.acm.org/citation.cfm?id=316573.317112>
- [25] C. P. Williams and A. G. Gray, “Automated design of quantum circuits,” in *Selected papers from the First NASA International Conference on Quantum Computing and Quantum Communications*, ser. QCQC '98. London, UK: Springer-Verlag, 1998, pp. 113–125. [Online]. Available: <http://portal.acm.org/citation.cfm?id=645812.670824>
- [26] G. Brassard, S. L. Braunstein, and R. Cleve, “Teleportation as a quantum computation,” *Physica D Nonlinear Phenomena*, vol. 120, pp. 43–47, Sep. 1998.
- [27] T. Y. Yabuki, “Genetic algorithms for quantum circuit design –evolving a simpler teleportation circuit–,” in *In Late Breaking Papers at the 2000 Genetic and Evolutionary Computation Conference*. Morgan Kaufman Publishers, 2000, pp. 421–425.
- [28] Google project hosting. [Online]. Available: <http://code.google.com/hosting/>
- [29] Complex number implementation. [Online]. Available: <http://www.math.ksu.edu/~bennett/jomacg/>
- [30] Jama matrix library. [Online]. Available: <http://math.nist.gov/javanumerics/jama/>
- [31] Latex package - qcircuit. [Online]. Available: <http://www.cquic.org/Qcircuit/>
- [32] jqquantum - quantum computer simulator. [Online]. Available: <http://jqquantum.sourceforge.net/>
- [33] Ecj. [Online]. Available: <http://www.cs.gmu.edu/~eclab/projects/ecj/>
- [34] Jppf framework. [Online]. Available: <http://www.jppf.org/>
- [35] Apache hadoop. [Online]. Available: <http://hadoop.apache.org/>
- [36] Octave gnu. [Online]. Available: <http://www.gnu.org/software/octave/>
- [37] Quantum computing functions for matlab and octave. [Online]. Available: <http://sourceforge.net/projects/qcf/>
- [38] L. Constantine and L. Lockwood, *Software for use: a practical guide to the models and methods of usage-centered design*, ser. Acm Press Series. Addison Wesley, 1999. [Online]. Available: <http://books.google.com/books?id=WKRQAAAAMAAJ>
- [39] P. Massey, J. A. Clark, and S. Stepney, “Evolution of a human-competitive quantum fourier transform algorithm using genetic programming,” in *Proceedings of the 2005 conference on Genetic and evolutionary computation*, ser. GECCO '05. New York, NY, USA: ACM, 2005, pp. 1657–1663. [Online]. Available: <http://doi.acm.org/10.1145/1068009.1068288>
- [40] Google codepro. [Online]. Available: <http://code.google.com/javadevtools/codepro/doc/index.html>
- [41] Apache log4j. [Online]. Available: <http://logging.apache.org/log4j/1.2/>
- [42] Junit. [Online]. Available: <http://www.junit.org/>
- [43] G. Ochoa, “Setting the mutation rate: Scope and limitations of the 1/1 heuristic,” in *Proceedings of the Genetic and Evolutionary Computation Conference*, ser. GECCO '02. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, pp. 495–502. [Online]. Available: <http://portal.acm.org/citation.cfm?id=646205.683104>
- [44] Apache commons logging. [Online]. Available: <http://commons.apache.org/logging/>
- [45] Apache xerces-j. [Online]. Available: <http://xerces.apache.org/xerces-j/>
- [46] Jfreechart. [Online]. Available: <http://www.jfree.org/jfreechart/>

- [47] Oracle javamail. [Online]. Available: <http://www.oracle.com/technetwork/java/javamail/index.html>
- [48] Seaglass - java cross-platform look and feel. [Online]. Available: <http://code.google.com/p/seaglass/>
- [49] Latex package - algorithm. [Online]. Available: <http://mirror.ox.ac.uk/sites/ctan.org/macros/latex/contrib/oberdiek/pdflscape.dtx>
- [50] Latex package - appendix. [Online]. Available: <http://www.ctan.org/pkg/appendix>
- [51] Latex package - array. [Online]. Available: <http://www.ctan.org/pkg/array>
- [52] Latex package - color. [Online]. Available: <http://www.ctan.org/pkg/color>
- [53] Latex package - emp. [Online]. Available: <http://www.ctan.org/pkg/emp>
- [54] Latex package - geometry. [Online]. Available: <http://www.ctan.org/pkg/geometry>
- [55] Latex package - graphicx. [Online]. Available: <http://www.ctan.org/pkg/graphicx>
- [56] Latex package - inputenc. [Online]. Available: <http://www.ctan.org/pkg/inputenc>
- [57] Latex package - listings. [Online]. Available: <http://www.ctan.org/pkg/listings>
- [58] Latex package - longtable. [Online]. Available: <http://www.ctan.org/pkg/longtable>
- [59] Latex package - metapost. [Online]. Available: <http://www.ctan.org/pkg/metapost>
- [60] Latex package - pdflscape. [Online]. Available: <http://www.ctan.org/pkg/pdflscape>
- [61] Latex package - qtree. [Online]. Available: <http://www.ctan.org/pkg/qtree>
- [62] Latex package - subfigure. [Online]. Available: <http://www.ctan.org/pkg/subfigure>
- [63] Latex package - tikz. [Online]. Available: <http://sourceforge.net/projects/pgf/>
- [64] Latex package - xypiclatex. [Online]. Available: <http://www.ctan.org/pkg/xypiclatex>

## A User Guide

This appendix is a walk through of how to use the framework and the provided client to search for a quantum algorithm to solve the Max Problem. This is a simple permutation problem. The goal is to take a permutation of the values 0..3 and return the index of the maximum value, 3.

This is used as an experiment in Massey's thesis[19] for the Q-Pace III system. The Q-Pace III was able to produce a probabilistic solution to the problem.

There are  $24, 4!$ , permutation functions possible. To encode the four values requires four qubits. This is done using superposition.

The permutation  $(0, 1, 2, 3) \rightarrow (w, x, y, z)$  is encoded as the superposition of  $\frac{1}{2}(|00w_1w_0\rangle + |01x_1x_0\rangle + |10y_1y_0\rangle + |11z_1z_0\rangle)$ . There are several options when selecting the encoding of the expected output of the system. This is due to the output being two qubits, as it is an index between 0 and 3, but the input requires four qubits. Three of the encoding choices are  $|**a_1a_0\rangle$ ,  $|a_1a_0**\rangle$  and  $|a_1a_0a_1a_0\rangle$ , where  $*$  denotes that we don't care what the value is. The encoding that we shall use in the walk through is  $|a_1a_0**\rangle$  so that it is only the highest significant qubits that we are actually interested in.

Figure A.1 shows the list of all 24 test cases.

### A.1 Creating the Search Problem

Using the test cases listed in Figure A.1 this section will go through how to create a search problem for the Max problem. The walk through will use the provided client rather than the standalone editor but as the two use the same components using the guide with the standalone editor should not cause any problems. The walk through shall only include the input of two randomly chosen test cases, 1 and 10.

1. Launch the client with the command below

```
java -jar MengQuantum.jar config/SearchEngine.xml config/FitnessFunction.xml config/Problems.xml false
```

The client shall launch and you shall be provided with the Graphical User Interface as shown in Figure A.2.

2. Using the mouse, left click on the button labelled "Create", as shown in Figure A.3. When prompted, select 0 custom gates as none are required for the Max problem. The "Create Problem and Test Suite" dialog box shall be displayed, as shown in Figure A.4.
3. Using the mouse, left click in the text area to the right of the "Name" and enter "Max Problem".
4. Using the mouse, left click on the button labelled "Select Destination File". A new dialog box shall open to select the location and file name for the test suite definition file. The dialog box is a standard "file chooser" as used in many applications so the operation of it should be familiar to all users.
5. Using the file chooser, navigate to the "Config" directory provided in the distribution. A number of XML files should be listed, including SearchEngine.xml and FitnessFunction.xml. Using the mouse, left click in the text area next to "File Name:" and enter "maxproblem". The ".xml" is added automatically by the software. Using the mouse, left click on the button labelled "Open" to close the dialog box.
6. Now we can start entering the test cases. Using the mouse, left click on the button labelled "Add Test Set" and, when prompted, enter 4 as the number of qubits. This shall change the appearance of the dialog box to match that shown in Figure A.5.
7. We shall enter test case 1.

The input state is shown in the left hand table. Put the value 0.5 in the second column of the rows labelled with the states  $|0000\rangle$ ,  $|0101\rangle$ ,  $|1010\rangle$  and  $|1111\rangle$ .

The output state is shown in the right hand table. Put the value 0.5 in the second column of the rows labelled with the states  $|1100\rangle$ ,  $|1101\rangle$ ,  $|1110\rangle$  and  $|1111\rangle$ . The value 0.5 is used as it means that the probability of any of them is  $\frac{1}{4}$  as no specific output is desired as long as the two most significant qubits match. This is just one way this could be encoded, the state  $|1100\rangle$  could be given the value 1 while all the others are given 0. This difficulty with representing the Max Permutation problem is explained in more detail in Section 4.4.1.

Test Case ID	Input State	→	Output State
1	$\frac{1}{2}( 0000\rangle +  0101\rangle +  1010\rangle +  1111\rangle)$	→	$\frac{1}{2}( 1100\rangle +  1101\rangle +  1110\rangle +  1111\rangle)$
2	$\frac{1}{2}( 0000\rangle +  0101\rangle +  1011\rangle +  1110\rangle)$	→	$\frac{1}{2}( 1000\rangle +  1001\rangle +  1010\rangle +  1011\rangle)$
3	$\frac{1}{2}( 0000\rangle +  0110\rangle +  1001\rangle +  1111\rangle)$	→	$\frac{1}{2}( 1100\rangle +  1101\rangle +  1110\rangle +  1111\rangle)$
4	$\frac{1}{2}( 0000\rangle +  0110\rangle +  1011\rangle +  1101\rangle)$	→	$\frac{1}{2}( 1000\rangle +  1001\rangle +  1010\rangle +  1011\rangle)$
5	$\frac{1}{2}( 0000\rangle +  0111\rangle +  1001\rangle +  1110\rangle)$	→	$\frac{1}{2}( 0100\rangle +  0101\rangle +  0110\rangle +  0111\rangle)$
6	$\frac{1}{2}( 0000\rangle +  0111\rangle +  1010\rangle +  1101\rangle)$	→	$\frac{1}{2}( 0100\rangle +  0101\rangle +  0110\rangle +  0111\rangle)$
7	$\frac{1}{2}( 0001\rangle +  0100\rangle +  1010\rangle +  1111\rangle)$	→	$\frac{1}{2}( 1100\rangle +  1101\rangle +  1110\rangle +  1111\rangle)$
8	$\frac{1}{2}( 0001\rangle +  0100\rangle +  1011\rangle +  1110\rangle)$	→	$\frac{1}{2}( 1000\rangle +  1001\rangle +  1010\rangle +  1011\rangle)$
9	$\frac{1}{2}( 0001\rangle +  0110\rangle +  1000\rangle +  1111\rangle)$	→	$\frac{1}{2}( 1100\rangle +  1101\rangle +  1101\rangle +  1111\rangle)$
10	$\frac{1}{2}( 0001\rangle +  0110\rangle +  1011\rangle +  1100\rangle)$	→	$\frac{1}{2}( 1000\rangle +  1001\rangle +  1010\rangle +  1011\rangle)$
11	$\frac{1}{2}( 0001\rangle +  0111\rangle +  1000\rangle +  1110\rangle)$	→	$\frac{1}{2}( 0100\rangle +  0101\rangle +  0110\rangle +  0111\rangle)$
12	$\frac{1}{2}( 0001\rangle +  0111\rangle +  1010\rangle +  1100\rangle)$	→	$\frac{1}{2}( 0100\rangle +  0101\rangle +  0110\rangle +  0111\rangle)$
13	$\frac{1}{2}( 0010\rangle +  0100\rangle +  1001\rangle +  1111\rangle)$	→	$\frac{1}{2}( 1100\rangle +  1101\rangle +  1101\rangle +  1111\rangle)$
14	$\frac{1}{2}( 0010\rangle +  0100\rangle +  1011\rangle +  1101\rangle)$	→	$\frac{1}{2}( 1000\rangle +  1001\rangle +  1010\rangle +  1011\rangle)$
15	$\frac{1}{2}( 0010\rangle +  0101\rangle +  1000\rangle +  1111\rangle)$	→	$\frac{1}{2}( 1100\rangle +  1101\rangle +  1110\rangle +  1111\rangle)$
16	$\frac{1}{2}( 0010\rangle +  0101\rangle +  1011\rangle +  1100\rangle)$	→	$\frac{1}{2}( 1000\rangle +  1001\rangle +  1010\rangle +  1011\rangle)$
17	$\frac{1}{2}( 0010\rangle +  0111\rangle +  1000\rangle +  1101\rangle)$	→	$\frac{1}{2}( 0100\rangle +  0101\rangle +  0110\rangle +  0111\rangle)$
18	$\frac{1}{2}( 0010\rangle +  0111\rangle +  1001\rangle +  1100\rangle)$	→	$\frac{1}{2}( 0100\rangle +  0101\rangle +  0110\rangle +  0111\rangle)$
19	$\frac{1}{2}( 0011\rangle +  0100\rangle +  1001\rangle +  1110\rangle)$	→	$\frac{1}{2}( 0000\rangle +  0001\rangle +  0010\rangle +  0011\rangle)$
20	$\frac{1}{2}( 0011\rangle +  0100\rangle +  1010\rangle +  1101\rangle)$	→	$\frac{1}{2}( 0000\rangle +  0001\rangle +  0010\rangle +  0011\rangle)$
21	$\frac{1}{2}( 0011\rangle +  0101\rangle +  1000\rangle +  1110\rangle)$	→	$\frac{1}{2}( 0000\rangle +  0001\rangle +  0010\rangle +  0011\rangle)$
22	$\frac{1}{2}( 0011\rangle +  0101\rangle +  1010\rangle +  1100\rangle)$	→	$\frac{1}{2}( 0000\rangle +  0001\rangle +  0010\rangle +  0011\rangle)$
23	$\frac{1}{2}( 0011\rangle +  0110\rangle +  1000\rangle +  1101\rangle)$	→	$\frac{1}{2}( 0000\rangle +  0001\rangle +  0010\rangle +  0011\rangle)$
24	$\frac{1}{2}( 0011\rangle +  0110\rangle +  1001\rangle +  1100\rangle)$	→	$\frac{1}{2}( 0000\rangle +  0001\rangle +  0010\rangle +  0011\rangle)$

Figure A.1: Max Problem Test Cases

8. Using the mouse, left click on the button labelled “Add Test Case”. We shall now enter test case 10. The input state is shown in the left hand table. Put the value 0.5 in the second column of the rows labelled with the states  $|0001\rangle$ ,  $|0100\rangle$ ,  $|1010\rangle$  and  $|1111\rangle$ . The output state is shown in the right hand table. Put the value 0.5 in the second column of the rows labelled with the states  $|1000\rangle$ ,  $|1001\rangle$ ,  $|1010\rangle$  and  $|1011\rangle$ .
9. Using the mouse, left click in the text area to the right of the “Description” label and enter “Max Problem as described by Massey”.
10. Using the mouse, left click on the button labelled “Okay” to save the test suite to file and close the dialog box.
11. Optional: To ensure that the “Max Problem” search problem is registered and loaded when the client is launched in the future left click on the button labelled “Save”. This “Save” button does not save the test suite, this is done automatically by the “Create Problem” dialog box. It saves the changes to the “config/Problems.xml” file containing the problem definitions, see Section 4.1.7.

The steps outlined produce a very simple test suite for the search to use. Providing the search with more of the 24 test cases that are available than the two created by the steps above is likely to improve the performance of the search. To create additional test cases repeat step 8 for each additional test case desired.

## A.2 Loading a Predefined Search Problem

This section shall provide a guide to creating a search problem by using an existing test suite definition XML file. The example that shall be used is if the user followed the walk through in Section A.1 by did not perform Step 11. This would result in the “Max Problem” created not to be registered, and therefore

not listed in the search problem drop down list, once the software is restarted. This is the process that would have to be performed if a researcher were to try use a test suite created and distribute by another researcher.

1. Launch the client with the command below

```
java -jar MengQuantum.jar config/SearchEngine.xml config/FitnessFunction.xml config/Problems.xml false
```

The client shall launch and you shall be provided with the Graphical User Interface as shown in Figure A.2.

2. Using the mouse, left click on the button labelled Load", as shown in Figure A.3. The "Load Test Suite to Create Problem" dialog box shall be displayed, as shown in Figure A.6.
3. Using the mouse, left click in the text area to the right of the "Name" and enter "Max Problem".
4. Using the mouse, left click on the button labelled "Select Definition File". A new dialog box shall open to select the location and file name for the test suite definition file. The dialog box is a standard "file chooser" as used in many applications so the operation of it should be familiar to all users.
5. Using the file chooser, navigate to the "Config" directory provided in the distribution. A number of XML files should be listed, including SearchEngine.xml and FitnessFunction.xml. Using the mouse, select the "maxproblem.xml" file and left click on the button labelled "Open" to close the dialog box.
6. Using the mouse, left click in the text area to the right of the "Description" label and enter "Max Problem as described by Massey".
7. Using the mouse, left click on the button labelled "Okay" to save the test suite to file and close the dialog box.
8. Optional: To ensure that the "Max Problem" search problem is registered and loaded when the client is launched in the future left click on the button labelled "Save". It saves the changes to the "config/Problems.xml" file containing the problem definitions, see Section 4.1.7.

### A.3 Editing an Existing Search Problem

Using the test cases listed in Figure A.1 this section will go through how to edit the search problem for the Max problem created in Section A.1. The walk through will use the provided client rather than the standalone editor but as the two use the same components using the guide with the standalone editor should not cause any problems. With the standalone editor the test suite definition XML file needs to be manually loaded using the "Open File" button. The walk through shall add an additional test case, test case 2.

1. Launch the client with the command below

```
java -jar MengQuantum.jar config/SearchEngine.xml config/FitnessFunction.xml config/Problems.xml false
```

The client shall launch and you shall be provided with the Graphical User Interface as shown in Figure A.2.

2. Using the problem selection drop down list, select "Max Problem".
3. Using the mouse, left click on the button labelled "Edit Selected Problem", as shown in Figure A.3. This shall launch the "Edit Current Problem and Test Suite" dialog box, as shown in Figure A.7. This dialog box is almost identical to the "Create Problem and Test Suite" used in Section A.1.
4. Using the mouse, left click on the button labelled "Add Test Case". We shall now enter test case 2. The input state is shown in the left hand table. Put the value 0.5 in the second column of the rows labelled with the states  $|0000\rangle$ ,  $|0101\rangle$ ,  $|1011\rangle$  and  $|1110\rangle$ . The output state is shown in the right hand table. Put the value 0.5 in the second column of the rows labelled with the states  $|0010\rangle$ ,  $|0110\rangle$ ,  $|1010\rangle$  and  $|1110\rangle$ .

5. Using the mouse, left click on the button labelled “Okay”. A warning dialog box is produced, see Figure A.8 . This shows the difference between the “Edit Current Problem and Test Suite” dialog box and the “Create Problem and Test Suite” used in Section A.1. When the “Okay” button is pressed on the “Create Problem and Test Suite” used in Section A.1 the test suite definition file is automatically updated. This is because the “Create Problem and Test Suite” creates an entirely new search problem and test suite. The “Edit Current Problem and Test Suite” dialog box is used to edit an existing search problem with a pre-existing test suite XML definition file. A researcher may want to make changes to a search problem for experimentation but not necessarily effect the contents of the test suite XML definition file.

If the researcher does want the test suite XML definition file to be updated to reflect the changes made to the test suite, the button labelled “Save and Close” should be used instead of the button labelled “Okay”.

## A.4 Carrying Out The Search

This section shall use the search problem created in Section A.1 to perform a search. The search engine that shall be used is the local version of the “Q-Pace IV Based Search Engine”, see Section 4.3 with the “Simple - Zero Focussed” suitability measure, see Section 4.4.1.

1. Launch the client with the command below

```
java -jar MengQuantum.jar config/SearchEngine.xml config/FitnessFunction.xml config/Problems.xml false
```

The client shall launch and you shall be provided with the Graphical User Interface as shown in Figure A.2.

2. Using the search engine selection drop down list, select “Genetic Programming - QPace Local”.
3. Using the suitability selection drop down list, select “Simple Suitability - Zero Focussed”.
4. Using the problem selection drop down list, select “Max Problem”.
5. Using the mouse, left click on the button labelled “Evolve”, as shown in Figure A.3. This shall open up a new dialog box specific for the search engine selected.
6. Most default settings shall be maintained. The three items that shall be changed are the number of generations, the mutation rate and the number of search iterations. Using the mouse, left click in the text area to the right of the “Number of Iterations” label and enter 3. Unselect all instructions except *Create\_H*, *Create\_CH*, *Create\_X*, *Create\_CX*, *Create\_CCX*, *Create\_R*, *Create\_CR*, *Iterate*, *RevIterate* and *Body*.
7. To start the search, click on the button labelled “Evolve Now”. This shall initiate the search and the dialog box shall close. All the selection drop down lists will be disabled so that changes cannot be made during a search. During the search, a statistics panel is provided as can be seen in Figure A.9. If the JPPF version of the search engine were selected, the statistics panel would provide much less information due to the amount of information available due to the distribution.
8. When the search is complete, the results shall be displayed in the central panel as can be seen in Figure A.10.

## A.5 Analysing the Search Results

Once the search is completed, the results are shown in the central panel as can be seen in Figure A.10. This section assumes that the results are produced by following the steps in Section A.4.

### **A.5.1 Which Search Result?**

In Step A.4 of Section A.4 the number of search iterations is set to 3. This means that three completely separate searches are carried out. Obviously this means that there are three separate search results produced. All search results are available for the user to analyse.

The user interface provides a drop down list of all the available search results. Below the drop down list are the details of the selected results. The results of the three searches are shown in Figures A.10, A.11 and A.12.

### **A.6 Analysing the Search Results**

The client provides access to some search statistics. The dialog box containing this information can be launched by the “Statistics” button shown alongside the drop down list in the central area. The dialog that is launched can be seen in Figure A.13.

Figure A.14 shows the statistics after 100 iterations of a search to find the entanglement circuit, a Hadamard on gate 1 followed by a Pauli-X gate on qubit 2 controlled by qubit 1. The statistics show the histogram of which generation an ideal solution was found.



A.7 Referenced Figures

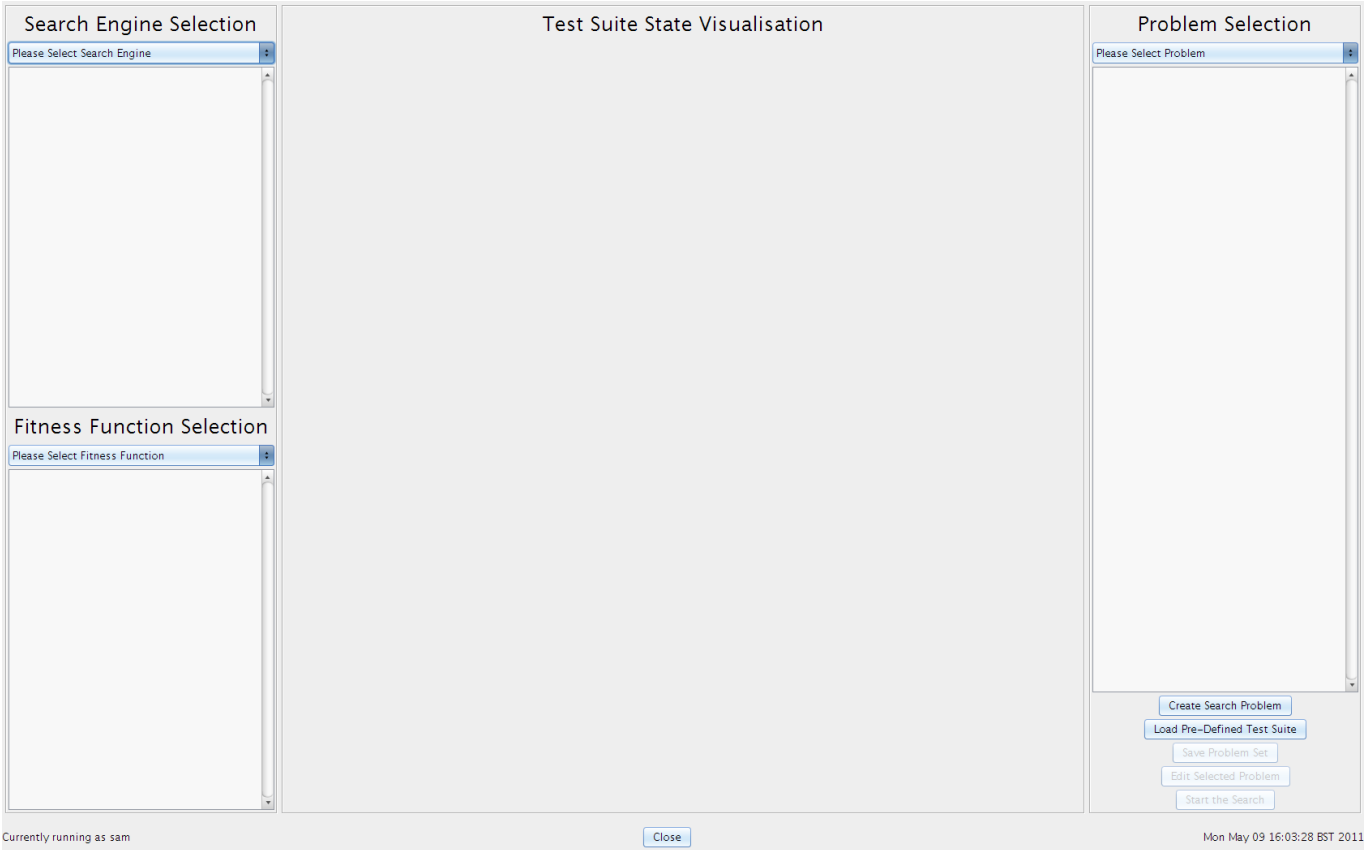


Figure A.2: Initial State of the Client GUI

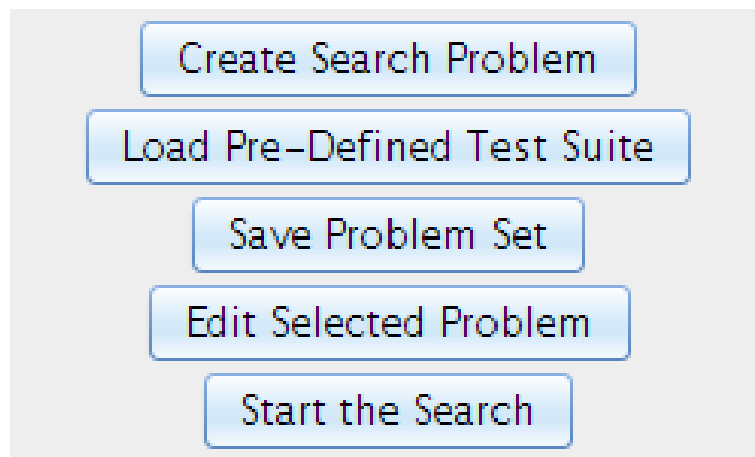


Figure A.3: Right hand menu

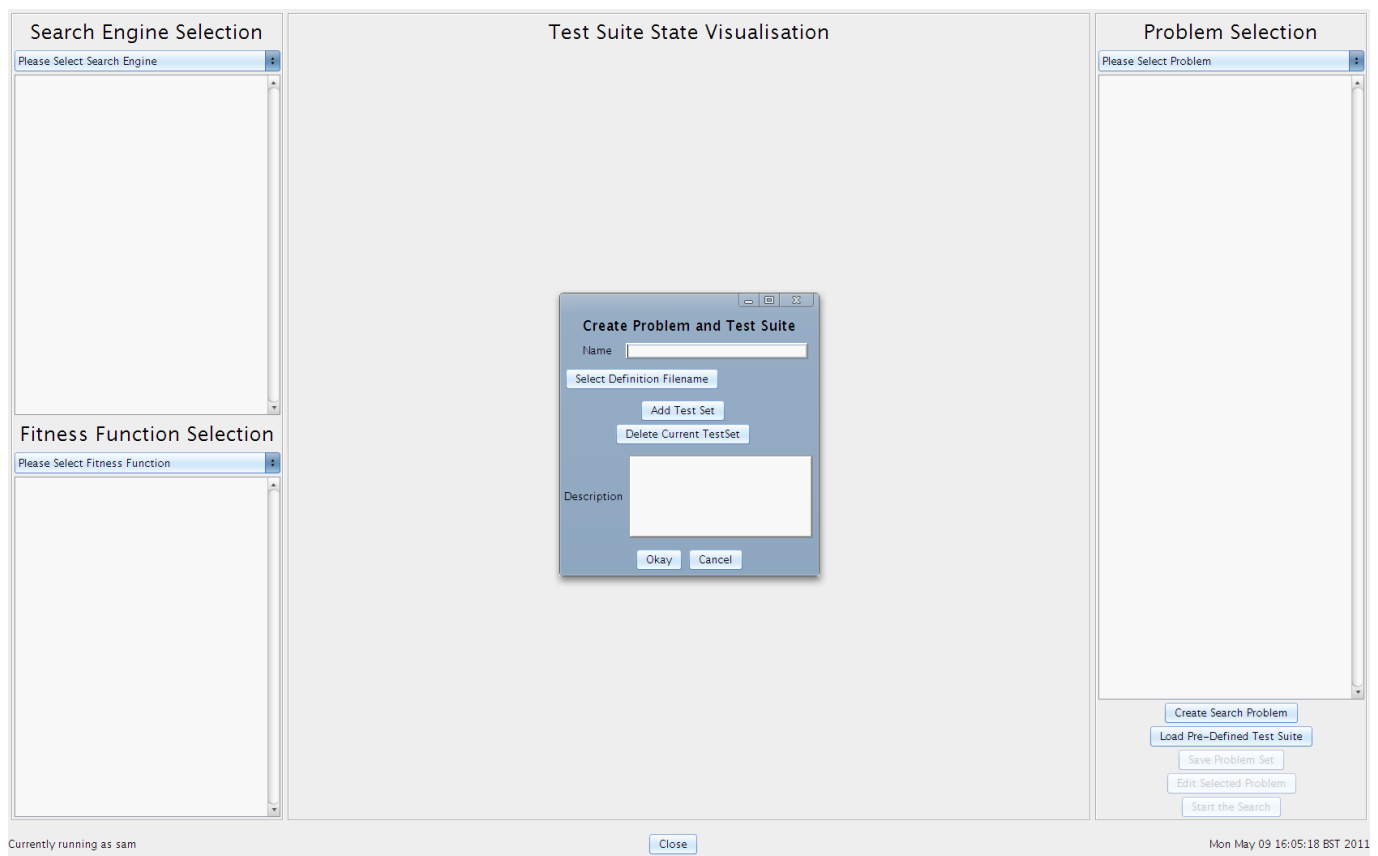


Figure A.4: Create Problem and Test Suite dialog box

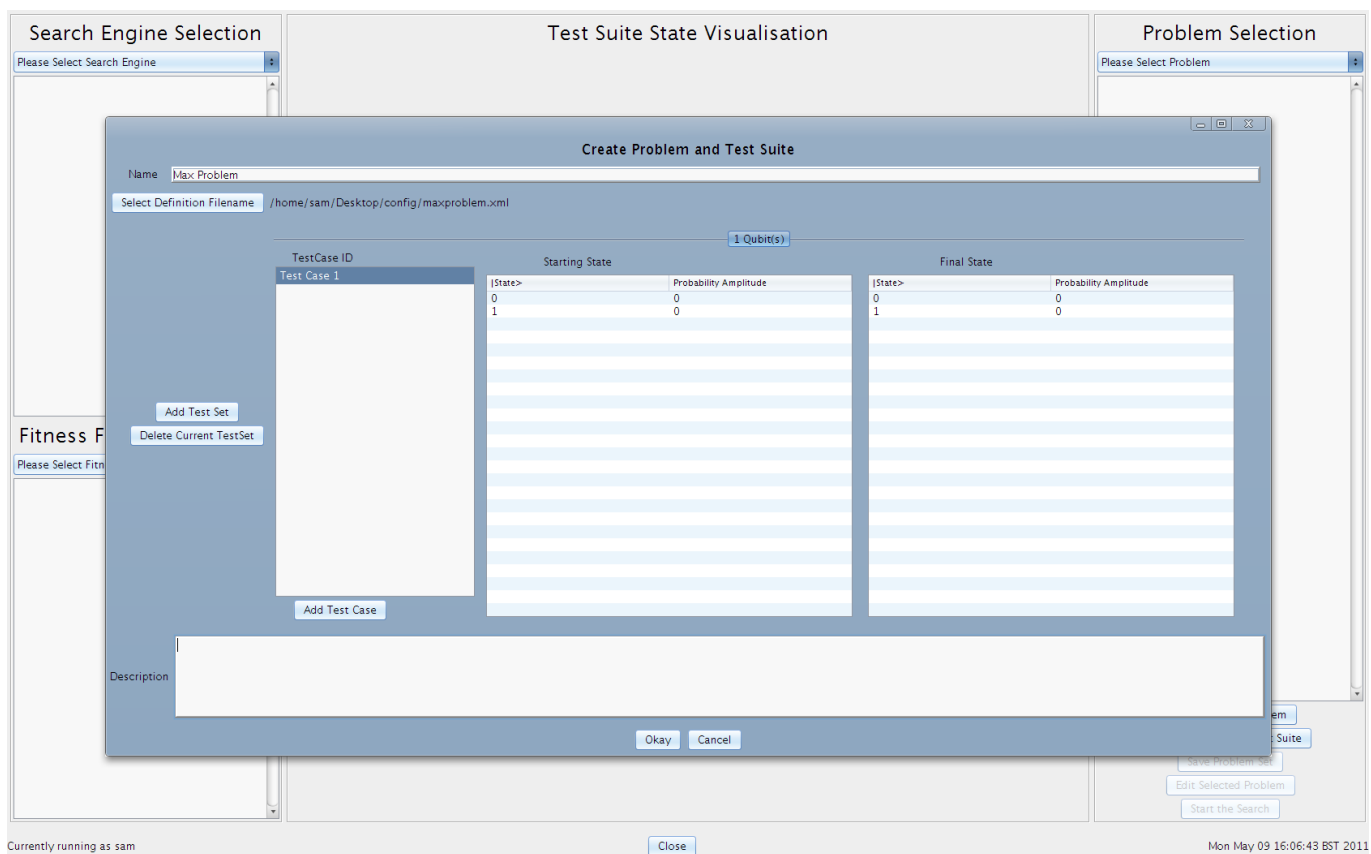


Figure A.5: Full Create Problem and Test Suite dialog box

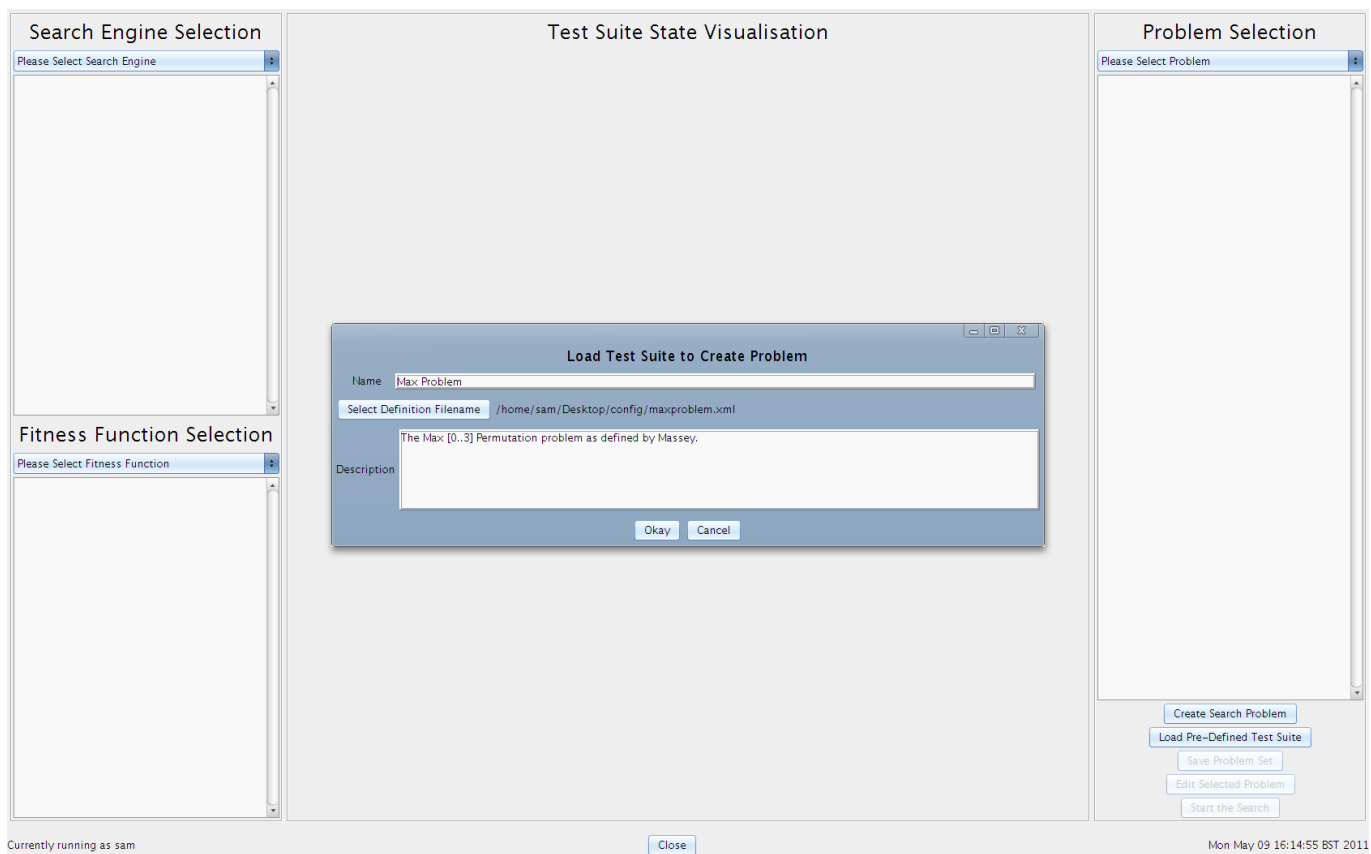


Figure A.6: Load Test Suite to Create Problem dialog box

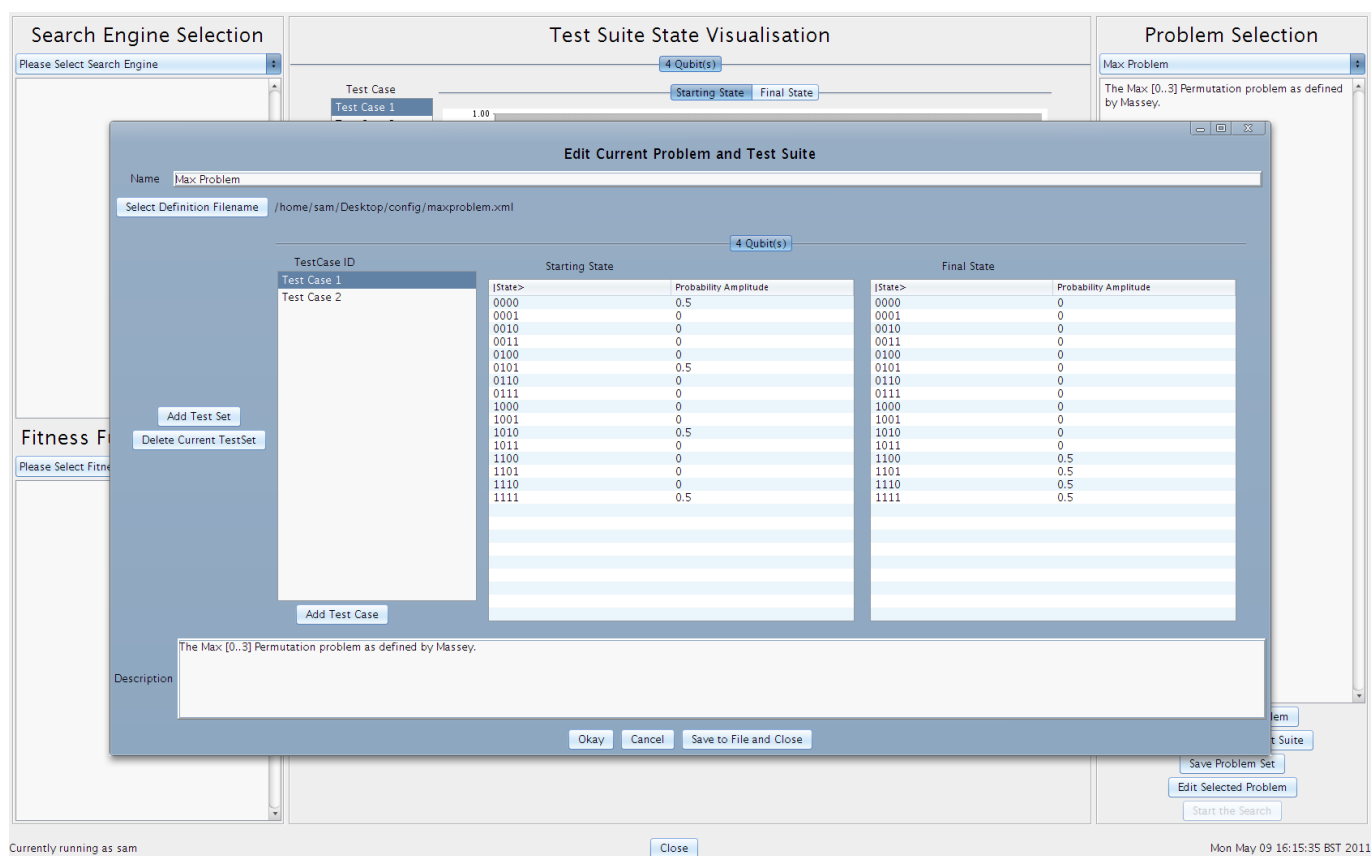


Figure A.7: Edit Current Problem and Test Suite dialog box

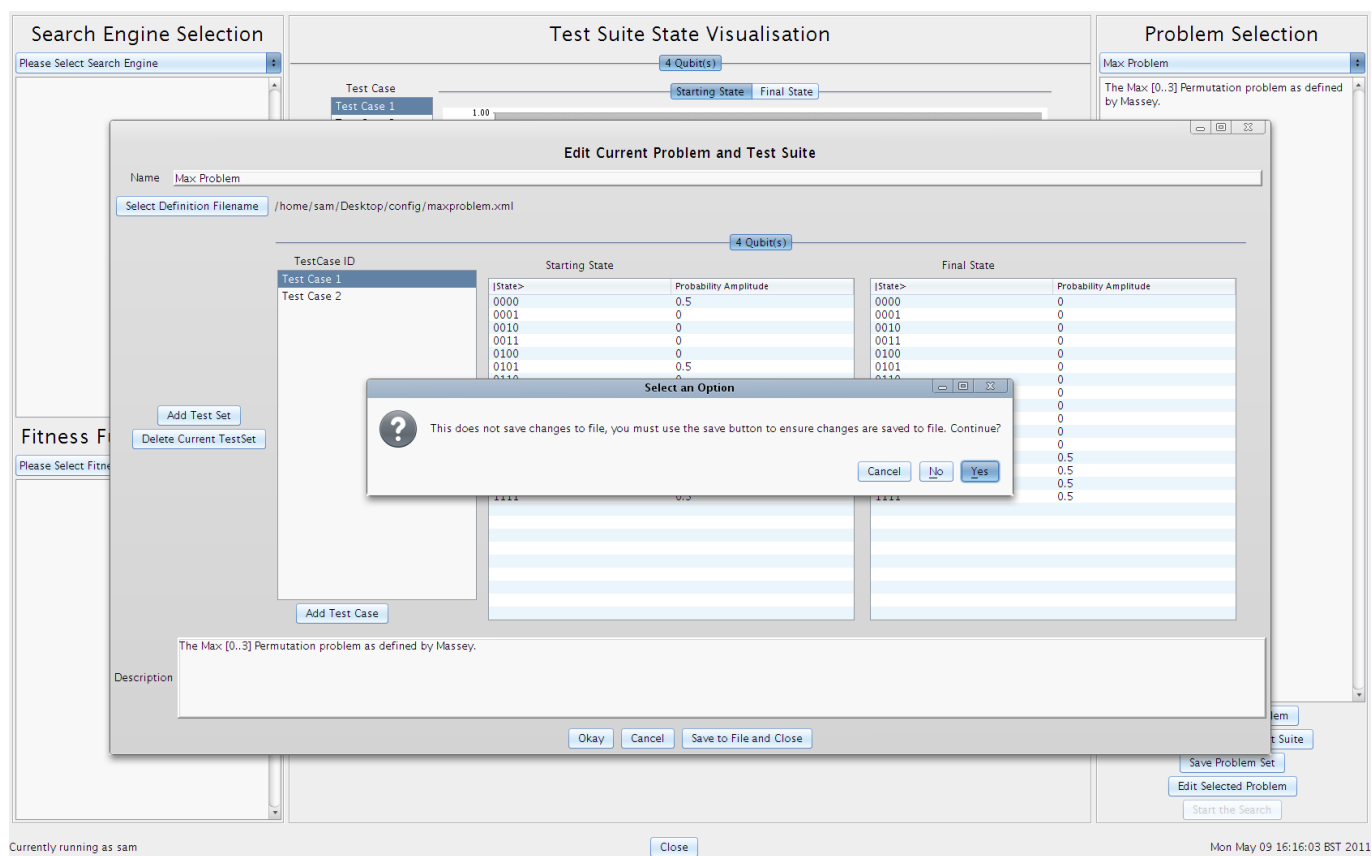


Figure A.8: Warning Produced by Pressing Okay

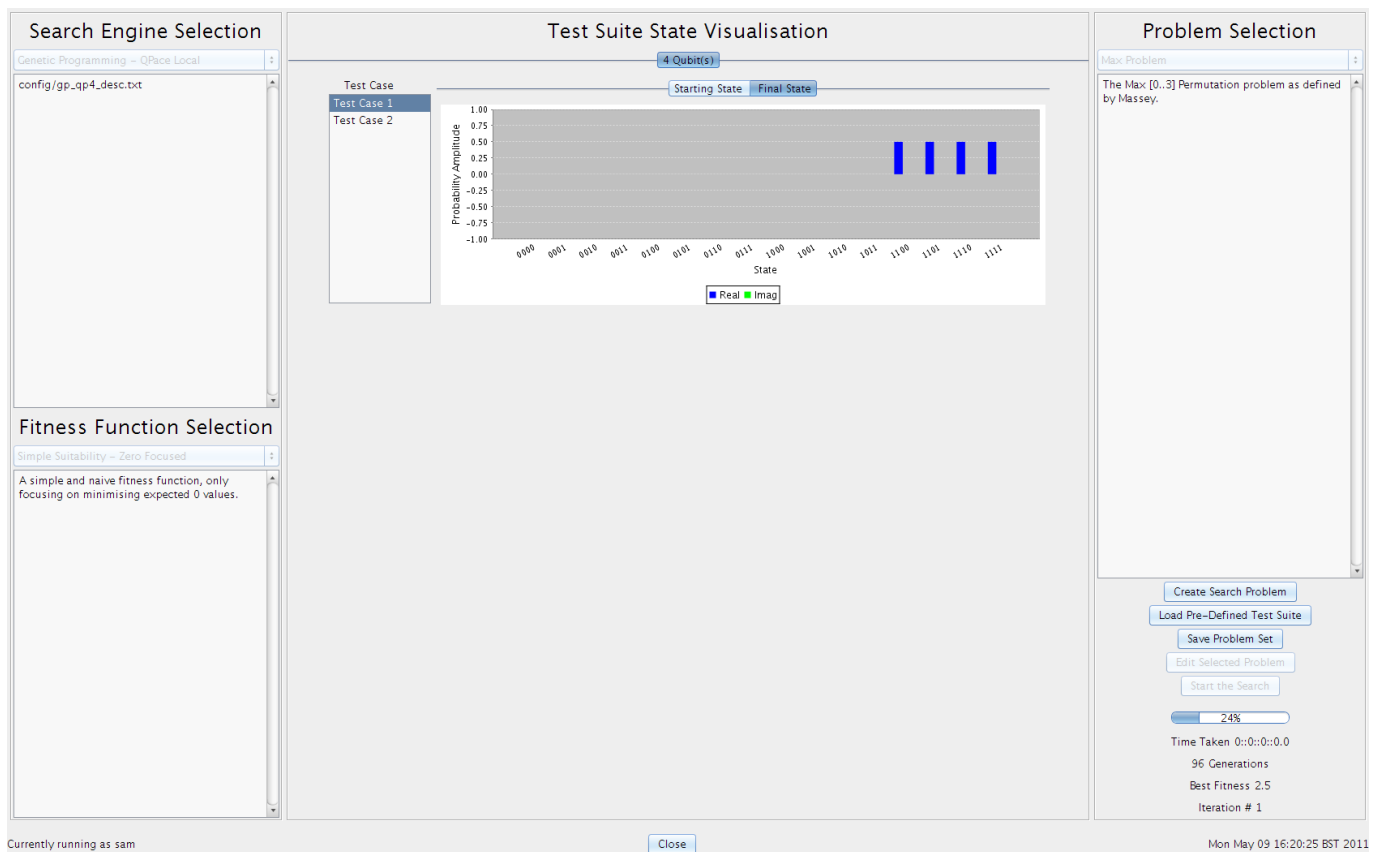


Figure A.9: Search Progress Statistics

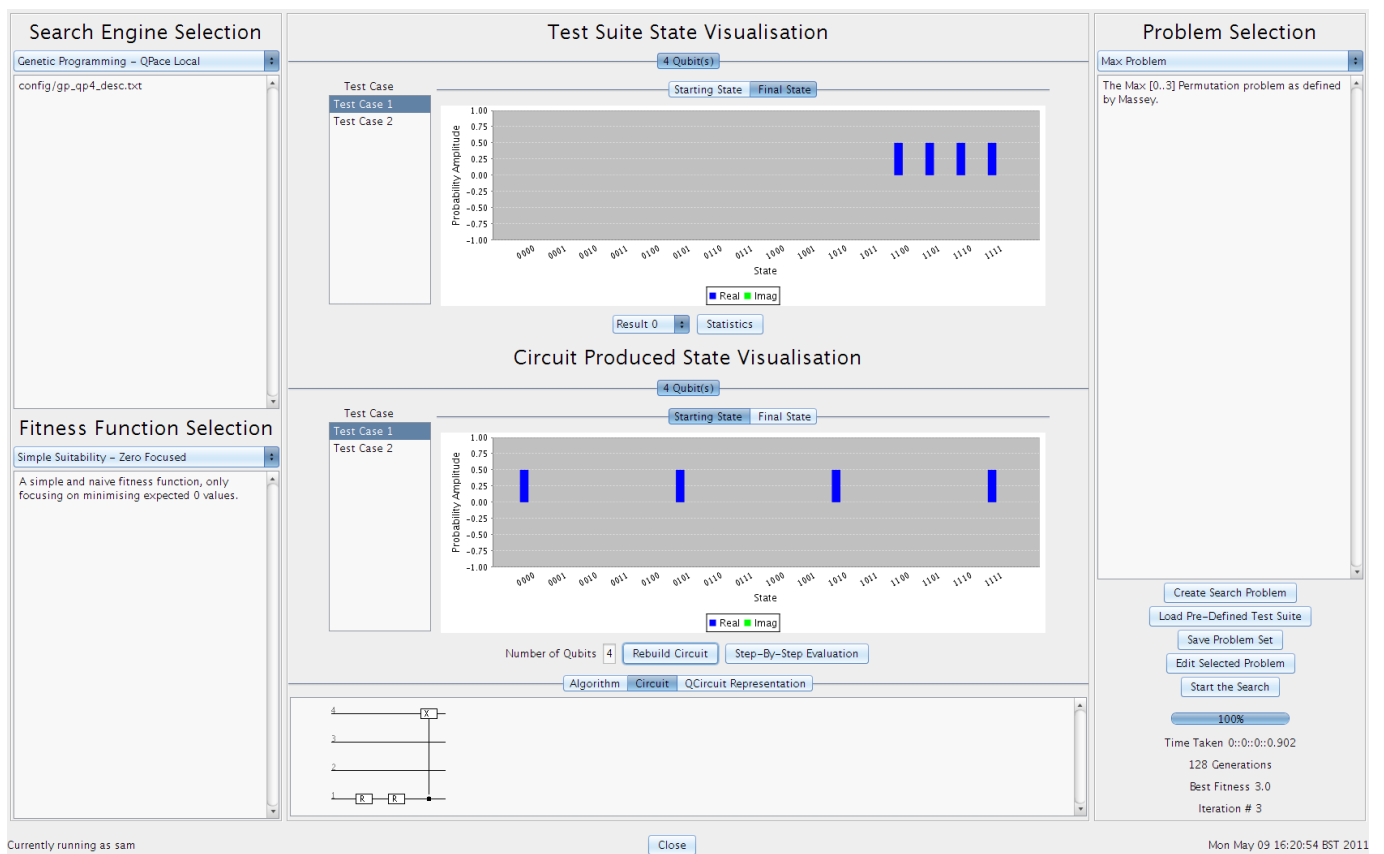


Figure A.10: Client GUI after Search is Complete - Result 0

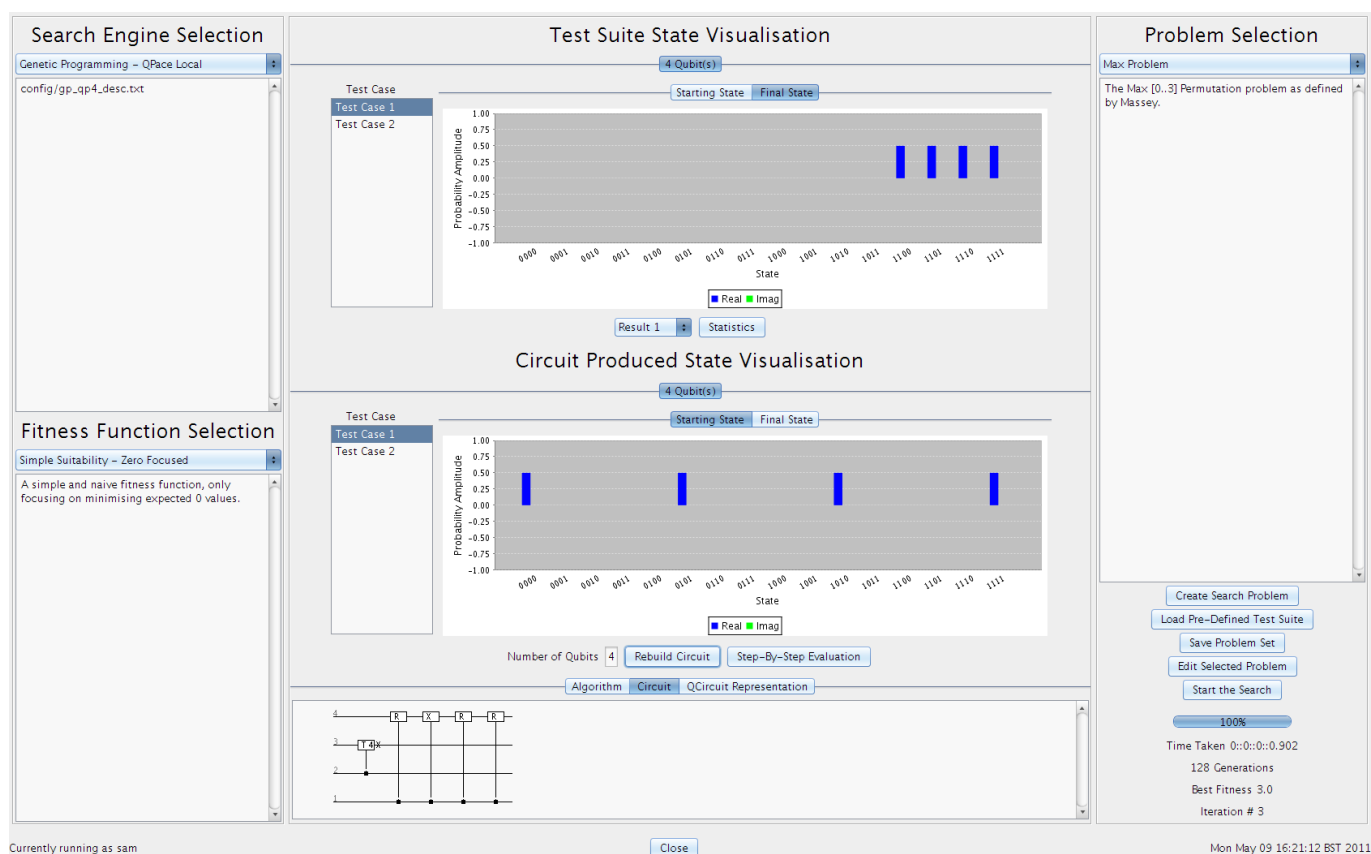


Figure A.11: Client GUI after Search is Complete - Result 1

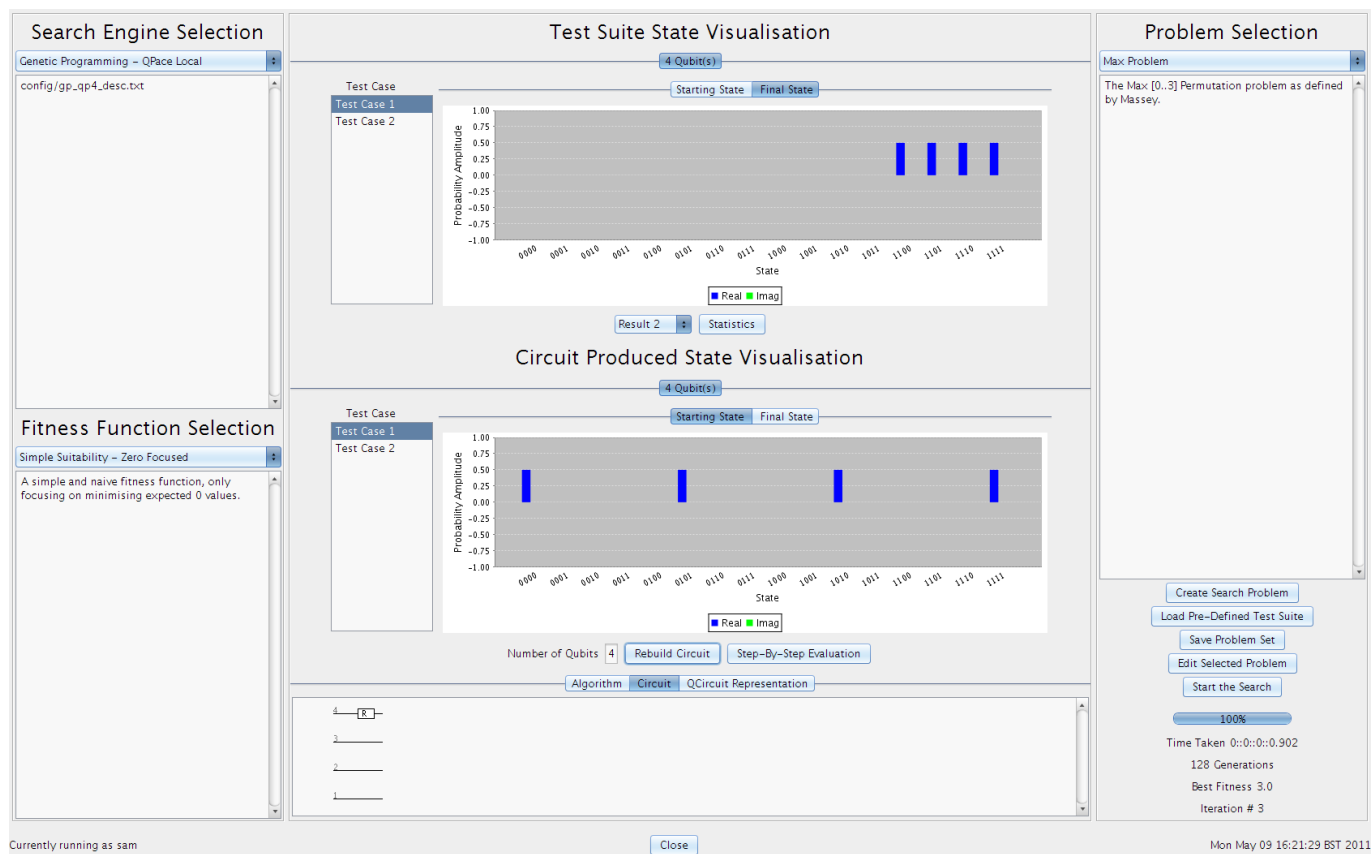


Figure A.12: Client GUI after Search is Complete - Result 2

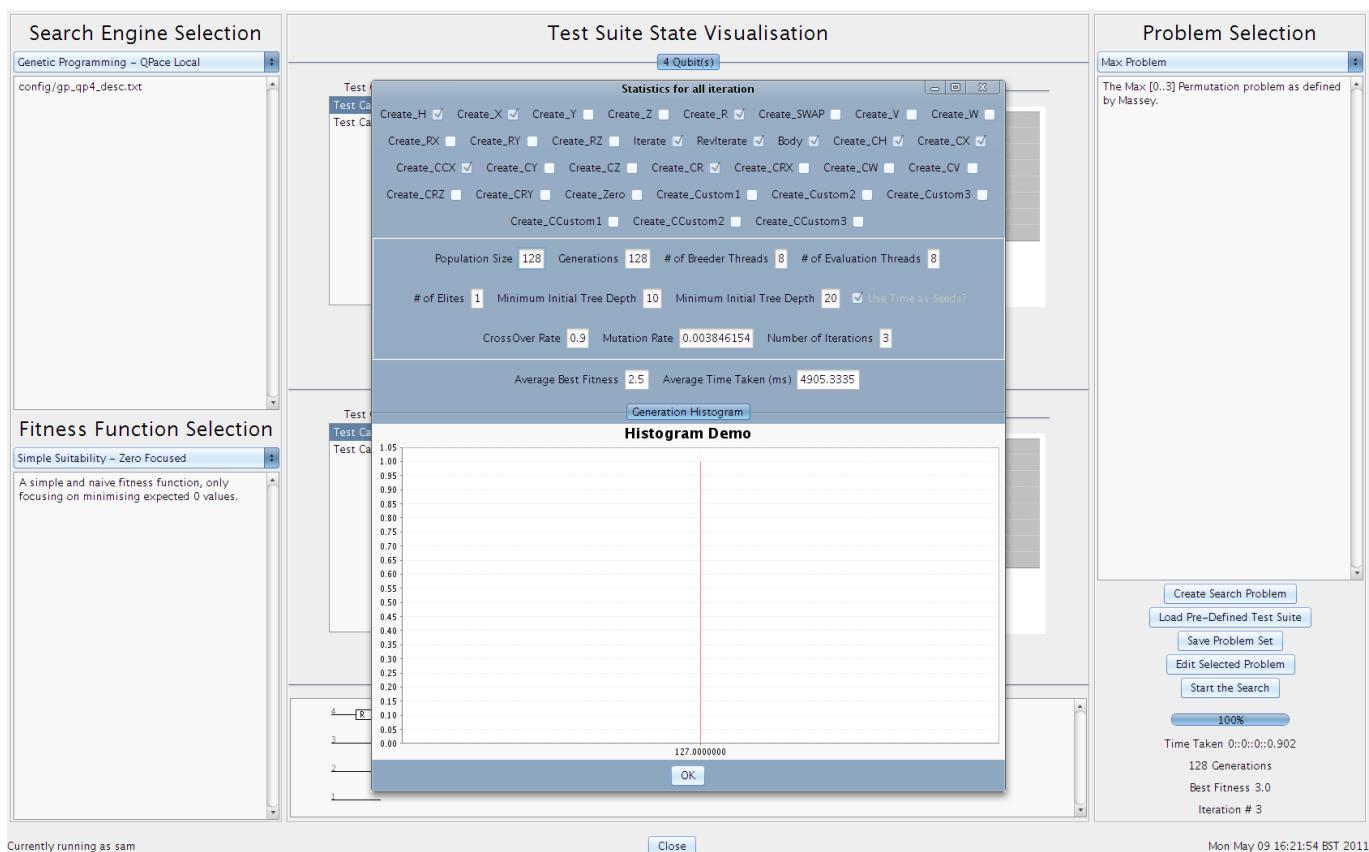


Figure A.13: Client GUI Showing the Search Statistics

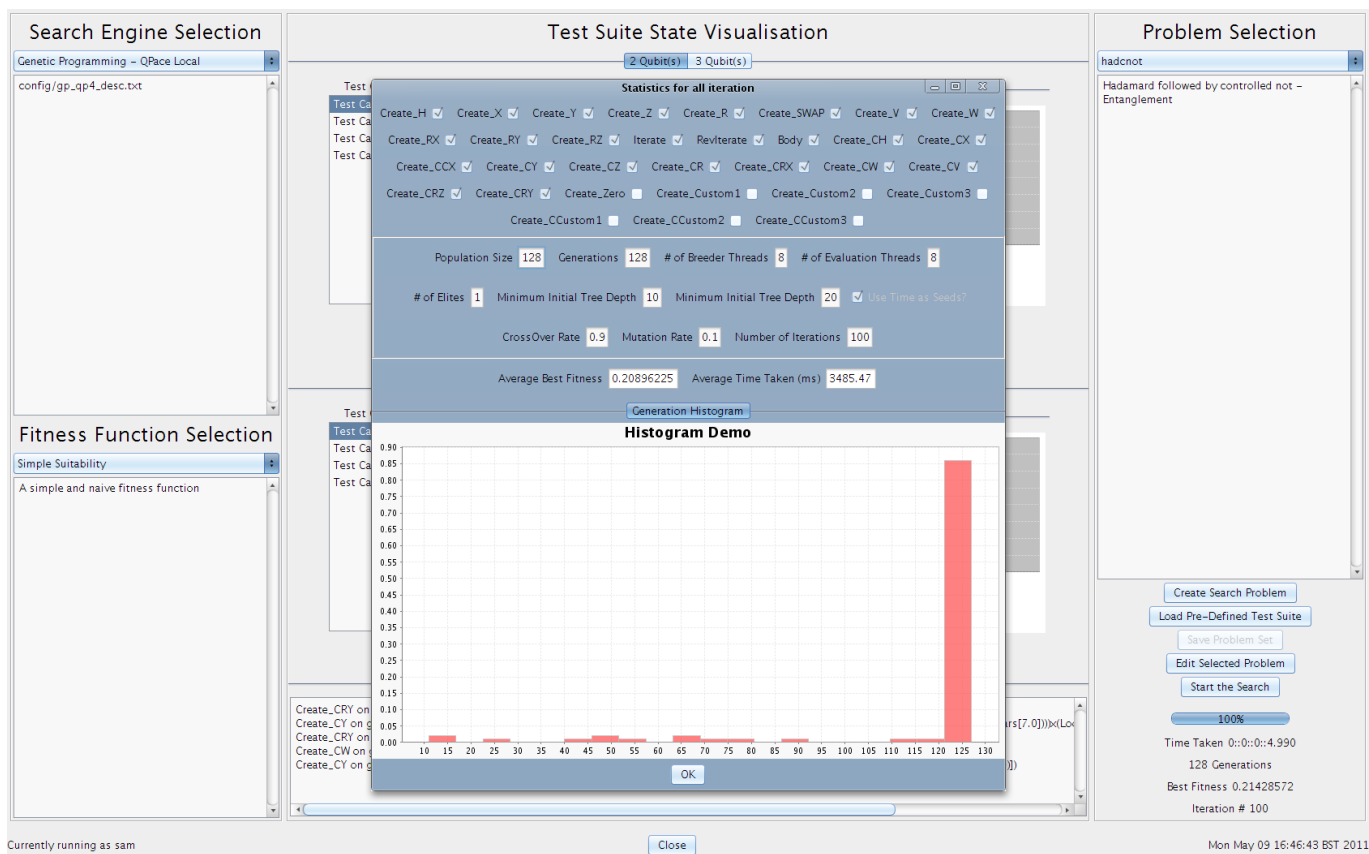


Figure A.14: Client GUI Showing the Search Statistics of Entanglement Search

## B Suitability Measure Definitions

All suitability measures are defined using the polar form  $re^{i\theta}$ . Subscript F indicates the Final state produced by the circuit generated from the algorithm under evaluation. Subscript E indicated the Expected final state as defined in the current test case.  $N$  is the system size.

$$\delta(a, b) = \begin{cases} 1, & \text{if } |a - b| < ACCURACY \\ 0, & \text{otherwise} \end{cases} \quad (B.1)$$

$$\delta(a, b, c, d) = \begin{cases} 1, & \text{if } |a - b| < ACCURACY \wedge |c - d| < ACCURACY \\ 0, & \text{otherwise} \end{cases} \quad (B.2)$$

### B.1 Simple Suitability Measure

$$ACCURACY = 10^{-8}$$

$$fitness = \sum_{i=0}^{2^N-1} |r_{Fi} - r_{Ei}| \quad (B.3)$$

$$hits = \sum_{i=0}^{2^N-1} \delta(r_{Fi}, r_{Ei}) \quad (B.4)$$

### B.2 Phase Aware Suitability Measure

$$ACCURACY = 10^{-8}$$

$$fitness = \sum_{i=0}^{2^N-1} |r_{Fi} - r_{Ei}| + |\theta_{Fi} - \theta_{Ei}| \quad (B.5)$$

$$hits = \sum_{i=0}^{2^N-1} \delta(r_{Fi}, r_{Ei}, \theta_{Fi}, \theta_{Ei}) \quad (B.6)$$



## C Experiment Data

### C.1 Deutsch Experiment

#### C.1.1 Full Algorithm

---

**Algorithm 8** Evolved Solution for Deutsch Problem

---

Create\_Custom1 on gate Loop\_Vars[54.0]  
Create\_H on gate Loop\_Vars[5.0]  
Create\_CCustom1 on gate 1.0 and gate System\_Size  
Create\_H on gate 1.0  
Create\_V on gate 54.0  
Create\_CCustom1 on gate 1.0 and gate System\_Size  
Create\_H on gate 1.0  
Create\_CX on gate 54.0 and gate 1.0

---



### C.2.1 Search Parameters

Population Size	128	Generations	300
# of Breeder Threads	8	# of Evaluation Threads	8
# of Elites	1	Number of Iterations	1
Minimum Initial Tree Depth	10	Maximum Initial Tree Depth	20
CrossOver Rate	0.9	Mutation Rate	0.0033846154
Search Engine		Genetic Programming - QPace Local	
Suitability Measure		Simple Suitability - Non-Zero Count	

Create_H	Create_X	Create_Y	Create_Z	Create_R	Create_Custom1	Body
Create_CH	Create_CX	Create_CY	Create_CZ	Create_CR	Create_CCustom1	Iterate
	Create_CCX					

Figure C.2: Search Parameters used for Deutsch Jozsa Search

### C.2.2 Full Algorithm

---

#### Algorithm 9 Evolved Solution for Deutsch Jozsa Problem

---

```

Create_CR on gate 2.0 and gate 6.0 with phase 1.0
Create_H on gate 2.0
Create_H on gate 3.0
Create_H on gate 1.0
Create_Custom1 on gate 1.0
Create_CR on gate Loop_Vars[3.0] and gate 48.0 with phase Loop_Vars[2.0]
Create_CR on gate 2.0 and gate 6.0 with phase 1.0
Create_H on gate 39.0
Create_CR on gate 2.0 and gate 6.0 with phase 1.0
Create_H on gate 2.0
Create_H on gate 3.0
Create_H on gate 31.0
Create_CR on gate 2.0 and gate 6.0 with phase 1.0
Create_H on gate Loop_Vars[2.0]
Create_H on gate 1.0
Create_CR on gate 79.0 and gate Loop_Vars[10.0] with phase Loop_Vars[2.0]

```

---

### C.2.3 Full Circuits

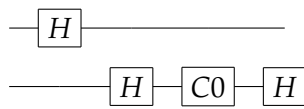


Figure C.3: Deutsch Jozsa Solution - Two Qubits

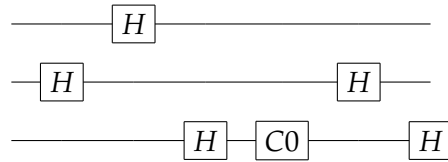


Figure C.4: Deutsch Jozsa Solution - Three Qubits

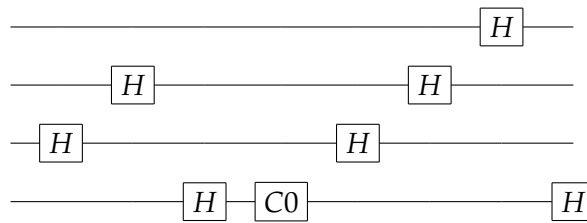


Figure C.5: Deutsch Jozsa Solution - Four Qubits

### C.3 0..3 Max Permutation Problem

#### C.3.1 Search Parameters

Population Size	128	Generations	128	
# of Breeder Threads	8	# of Evaluation Threads	8	
# of Elites	1	Number of Iterations	1	
Minimum Initial Tree Depth	1	Maximum Initial Tree Depth	30	
CrossOver Rate	0.9	Mutation Rate	0.003846154	
Search Engine	Genetic Programming - QPace Local			
Suitability Measure	Simple Suitability - Zero Focessed			
Create_H	Create_X	Create_R	Body	Iterate
Create_CH	Create_CX	Create_CR	Create_CCX	RevIterate

Figure C.6: Search Parameters Used for 0..3 Max Permutation Problem

### C.3.2 Algorithm

---

**Algorithm 10** Full Algorithm to Produce the Solution for the Max Permutation Problem

---

```
{  
  Iterate for 10.0 iterations {  
  }  
}  
Create_X on gate Loop_Vars[8.0]  
Create_CX on gate 17.0 and gate 45.0  
Create_CH on gate 7.0 and gate 17.0  
Iterate for 95.0 iterations {  
  Create_R on gate Loop_Vars[4.0] with phase Loop_Vars[1.0]  
}  
Create_H on gate 102.0  
{  
{  
  Create_CCX on gate 2.0 and gate 1.0 with phase 2.0  
}  
  Create_X on gate 2.0  
}  
  Iterate for 10.0 iterations {  
  }  
  Create_CCX on gate System_Size and gate 1.0 with phase 2.0
```

---

### C.3.3 Measurement Probabilities

Test Case	$ 00\rangle$	$ 01\rangle$	$ 10\rangle$	$ 11\rangle$
1	0.25	0.00	0.25	<b>0.50</b>
2	0.00	0.25	<b>0.50</b>	0.25
3	0.25	0.00	0.25	<b>0.50</b>
4	0.00	0.25	<b>0.50</b>	0.25
5	0.25	<b>0.50</b>	0.25	0.00
6	0.25	<b>0.50</b>	0.25	0.00
7	0.25	0.00	0.25	<b>0.50</b>
8	0.00	0.25	<b>0.50</b>	0.25
9	0.25	0.00	0.25	<b>0.50</b>
10	0.00	0.25	<b>0.50</b>	0.25
11	0.25	<b>0.50</b>	0.25	0.00
12	0.25	<b>0.50</b>	0.25	0.00
13	0.25	0.00	0.25	<b>0.50</b>
14	0.00	0.25	<b>0.50</b>	0.25
15	0.25	0.00	0.25	<b>0.50</b>
16	0.00	0.25	<b>0.50</b>	0.25
17	0.25	<b>0.50</b>	0.25	0.00
18	0.25	<b>0.50</b>	0.25	0.00
19	<b>0.50</b>	0.25	0.00	0.25
20	<b>0.50</b>	0.25	0.00	0.25
21	<b>0.50</b>	0.25	0.00	0.25
22	<b>0.50</b>	0.25	0.00	0.25
23	<b>0.50</b>	0.25	0.00	0.25
24	<b>0.50</b>	0.25	0.00	0.25

The bold values are the probability of the desired states, the solution states.

## C.4 Quantum Fourier Transform Experiment

$$\begin{aligned}
 & \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0.50000 \\ 0.50000 \\ 0.50000 \\ 0.50000 \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0.50000 + 0.00000i \\ 0.00000 + 0.50000i \\ -0.50000 + 0.00000i \\ -0.00000 - 0.50000i \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0.50000 + 0.00000i \\ -0.50000 + 0.00000i \\ 0.50000 - 0.00000i \\ -0.50000 + 0.00000i \end{pmatrix} \\
 & \text{(a) Test Case 1} & \text{(b) Test Case 2} & \text{(c) Test Case 3} \\
 & & \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} 0.50000 + 0.00000i \\ -0.00000 - 0.50000i \\ -0.50000 + 0.00000i \\ 0.00000 + 0.50000i \end{pmatrix} & \\
 & & \text{(d) Test Case 4} & 
 \end{aligned}$$

Figure C.7: Test Cases for Two Qubits

$$\begin{aligned}
 & \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0.35355 + 0.00000i \\ 0.35355 + 0.00000i \\ 0.35355 + 0.00000i \\ 0.35355 + 0.00000i \\ 0.35355 + 0.00000i \\ 0.35355 + 0.00000i \\ 0.35355 + 0.00000i \\ 0.35355 + 0.00000i \end{pmatrix} & \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0.35355 + 0.00000i \\ 0.25000 + 0.25000i \\ 0.00000 + 0.35355i \\ -0.25000 + 0.25000i \\ -0.35355 + 0.00000i \\ -0.25000 - 0.25000i \\ -0.00000 - 0.35355i \\ 0.25000 - 0.25000i \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0.35355 + 0.00000i \\ 0.00000 + 0.35355i \\ -0.35355 + 0.00000i \\ -0.00000 - 0.35355i \\ 0.35355 - 0.00000i \\ 0.00000 + 0.35355i \\ -0.35355 + 0.00000i \\ -0.00000 - 0.35355i \end{pmatrix} \\
 & \text{(a) Test Case 1} & \text{(b) Test Case 2} & \text{(c) Test Case 3} \\
 & \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0.35355 + 0.00000i \\ -0.25000 + 0.25000i \\ -0.00000 - 0.35355i \\ 0.25000 + 0.25000i \\ -0.35355 + 0.00000i \\ 0.25000 - 0.25000i \\ 0.00000 + 0.35355i \\ -0.25000 - 0.25000i \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0.35355 + 0.00000i \\ -0.35355 + 0.00000i \\ 0.35355 - 0.00000i \\ -0.35355 + 0.00000i \\ 0.35355 - 0.00000i \\ -0.35355 + 0.00000i \\ 0.35355 - 0.00000i \\ -0.35355 + 0.00000i \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0.35355 + 0.00000i \\ -0.25000 - 0.25000i \\ 0.00000 + 0.35355i \\ 0.25000 - 0.25000i \\ -0.35355 + 0.00000i \\ 0.25000 + 0.25000i \\ -0.00000 - 0.35355i \\ -0.25000 + 0.25000i \end{pmatrix} \\
 & \text{(d) Test Case 4} & \text{(e) Test Case 5} & \text{(f) Test Case 6} \\
 & \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0.35355 + 0.00000i \\ -0.00000 - 0.35355i \\ -0.35355 + 0.00000i \\ 0.00000 + 0.35355i \\ 0.35355 - 0.00000i \\ -0.00000 - 0.35355i \\ -0.35355 + 0.00000i \\ 0.00000 + 0.35355i \end{pmatrix} & \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \rightarrow \begin{pmatrix} 0.35355 + 0.00000i \\ 0.25000 - 0.25000i \\ -0.00000 - 0.35355i \\ -0.25000 - 0.25000i \\ -0.35355 + 0.00000i \\ -0.25000 + 0.25000i \\ 0.00000 + 0.35355i \\ 0.25000 + 0.25000i \end{pmatrix} & \\
 & \text{(g) Test Case 7} & \text{(h) Test Case 8} & 
 \end{aligned}$$

Figure C.8: Test Cases for Two Qubits



#### C.4.1 Search Parameters

Population Size	1280	Generations	1280
# of Breeder Threads	8	# of Evaluation Threads	8
# of Elites	1	Number of Iterations	1
Minimum Initial Tree Depth	10	Maximum Initial Tree Depth	20
CrossOver Rate	0.9	Mutation Rate	0.003846154
<b>Search Engine</b>		Genetic Programming - QPace Local	
<b>Suitability Measure</b>		Phase Sensitive	
Create_H	Create_R	Create_SWAP	Body
Create_CH	Create_CR	Iterate	RevIterate

Figure C.9: Algorithm Instructions used for Quantum Fourier Transform Search

#### C.4.2 Algorithm

---

**Algorithm 11** Full Algorithm to Produce the Quantum Fourier Transform

---

```
Create_SWAP on gate 4.0 and gate 1.0
Create_CH on gate 1.0 and gate 5.0
Create_SWAP on gate 5.0 and gate (Loop_Vars[4.0])+(16.0)
Create_CH on gate 1.0 and gate 5.0
Iterate for 5.0 iterations {
  Create_CH on gate 1.0 and gate 6.0
}
Create_SWAP on gate 14.0 and gate 1.0
Create_CR on gate (14.0)x(Loop_Vars[30.0]) and gate Loop_Vars[Loop_Vars[5.0]] with phase
Loop_Vars[3.0]
Create_CH on gate 1.0 and gate 5.0
Create_CH on gate 1.0 and gate 5.0
Create_CR on gate 20.0 and gate Loop_Vars[Loop_Vars[8.0]] with phase (Loop_Vars[3.0])x(1.0)
Create_CH on gate 5.0 and gate 5.0
Create_SWAP on gate 20.0 and gate 5.0
Create_CH on gate 1.0 and gate 6.0
Create_SWAP on gate Loop_Vars[(Loop_Vars[38.0])-((80.0)x(Loop_Vars[9.0]))] and gate 1.0
Create_SWAP on gate 9.0 and gate 1.0
{
  Create_CH on gate 1.0 and gate 5.0
  Create_H on gate 16.0
}
Create_SWAP on gate ((Loop_Vars[16.0])-(Loop_Vars[Loop_Vars[9.0]]))-(14.0) and gate 16.0
Create_H on gate 23.0
Create_CH on gate 9.0 and gate 5.0
Create_H on gate (Loop_Vars[9.0])+(7.0)
Create_SWAP on gate Loop_Vars[9.0] and gate (Loop_Vars[9.0])+(7.0)
Iterate for Loop_Vars[9.0] iterations {
  Create_CR on gate Loop_Vars[Loop_Vars[4.0]] and gate 20.0 with phase 3.0
  Create_CH on gate 1.0 and gate 5.0
  Create_SWAP on gate 4.0 and gate 1.0
  Create_H on gate 16.0
  Create_SWAP on gate 9.0 and gate 23.0
  Create_SWAP on gate Loop_Vars[4.0] and gate 14.0
  Create_CR on gate 0.0 and gate (5.0)-(Loop_Vars[4.0]) with phase 2.0
  Create_CH on gate 1.0 and gate 5.0
  Create_CR on gate 7.0 and gate 12.0 with phase Loop_Vars[3.0]
  Create_H on gate 10.0
  Create_CH on gate 1.0 and gate 5.0
  Create_CR on gate 1.0 and gate Loop_Vars[Loop_Vars[9.0]] with phase Loop_Vars[3.0]
  Create_H on gate Loop_Vars[((Loop_Vars[6.0])-((9.0)-((9.0)-(Loop_Vars[Loop_Vars[34.0]]))))-(-19.0)]
}
```

---

### C.4.3 Circuits

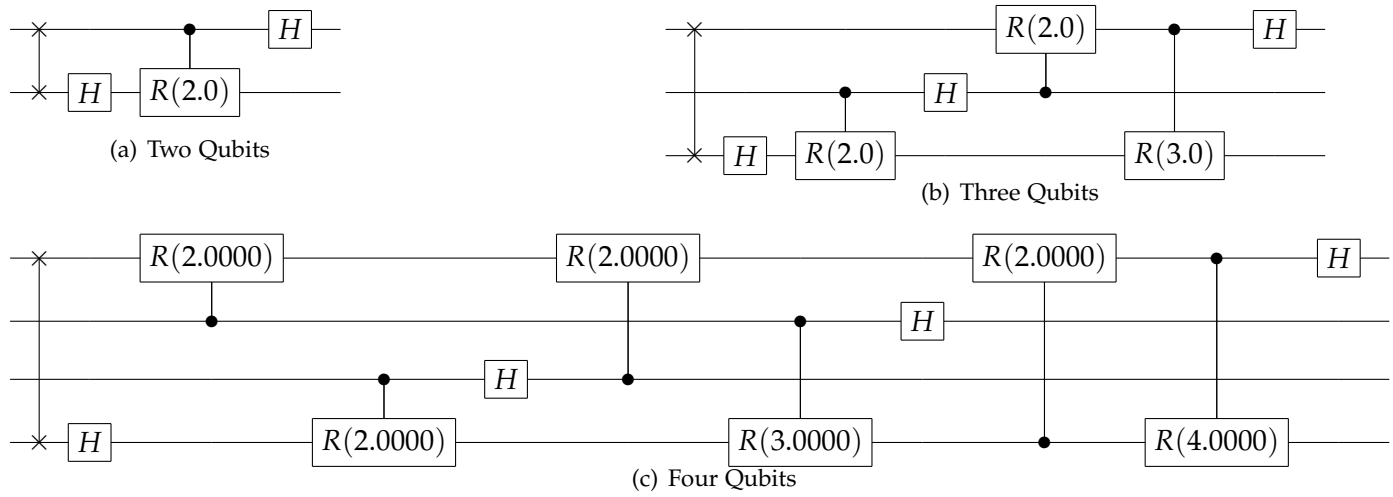


Figure C.10: Evolved Quantum Fourier Transform Solution

## **D Libraries Used**

### **D.1 Software**

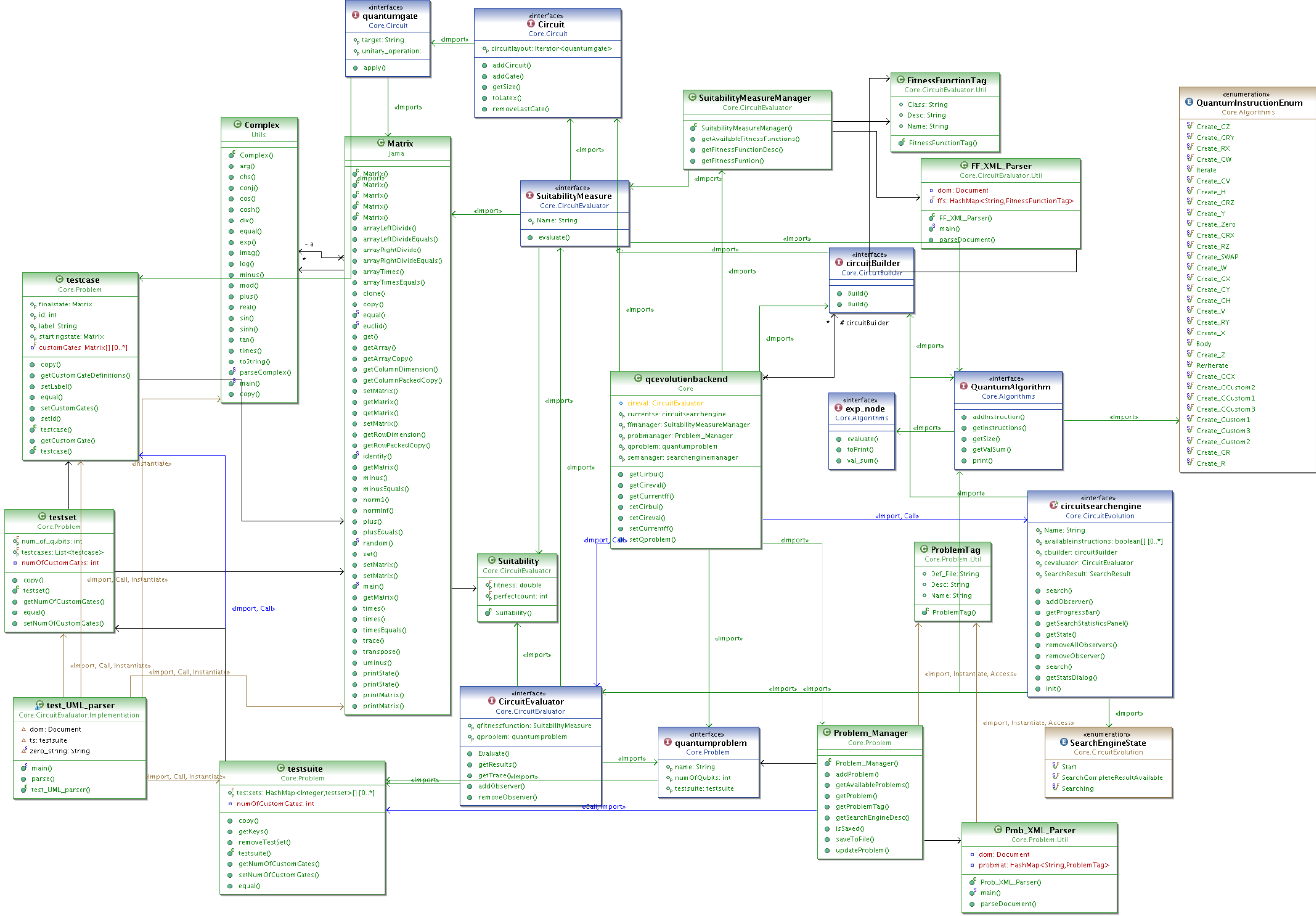
- Apache Commons Logging[44]
- Apache Log4J[41]
- Apache Xerces-J[45]
- ECJ[33]
- Jama[30]
- JFreeChart[46]
- JPPF[34]
- JQuantum[32]
- JUnit[42]
- Oracle JavaMail[47]
- SeaGlassLookAndFeel[48]

### **D.2 Report**

- algorithm[49]
- algorithmic[49]
- appendix[50]
- array[51]
- color[52]
- emp[53]
- geometry[54]
- graphicx[55]
- inputenc[56]
- listings[57]
- longtable[58]
- metapost[59]
- pdfscape[60]
- QCircuit[31]
- qtree[61]
- subfigure[62]
- tikz[63]
- xypic[64]



E Architecture Diagram







## F XML Outlines

### F.1 Search Engine XML Outline

```
<searchengine>
  <se>
    <Name>SEARCH ENGINE NAMES</Name>
    <Class>IMPLEMENTING FULLY QUALIFIED CLASS NAME</Class>
    <Desc>SEARCH ENGINE DESCRIPTION</Desc>
  </se>
</searchengine>
```

### F.2 Search Problem Definition XML Outline

```
<Problems>
  <prob>
    <Name>PROBLEM NAME</Name>
    <DefFile>PROBLEM DEFINITION FILE</DefFile>
    <Desc>PROBLEM DESCRIPTION</Desc>
  </prob>
</Problems>
```

### F.3 Suitability Measure Definition XML Outline

```
<FitnessFunc>
  <FitnessFunctionTag>
    <Name>SUITABILITY MEASURE NAME</Name>
    <Class>IMPLEMENTING FULLY QUALIFIED CLASS NAME</Class>
    <Desc>SUITABILITY MEASURE DESCRIPTION</Desc>
  </FitnessFunctionTag>
</FitnessFunc>
```

### F.4 Matrix Definition XML Outline

```
<Matrix NumQubits="NUM_OF_QUBITS">
  <MatrixElement Row="ROW_INDEX" Column="COLUMN_INDEX">
    <Real>REAL COMPONENT</Real>
    <Imag>IMAGINARY COMPONENT</Imag>
  </MatrixElement>
</Matrix>
```

### F.5 Test Suit Definition XML Outline

```
<testsuite NumCustomGates="NUMBER_OF_CUSTOM_GATE">
  <testset NumQubits="SYSTEM_SIZE_FOR_THIS_TEST_SET">
    <testcase><!--o-->
      <custom_gate>CUSTOM GATE XML FILE NAME</custom_gate>
      <starting_state>
        <matrix_element><!-- o-->
          <Real>REAL COMPONENT</Real>
          <Imag>IMAGINARY COMPONENT</Imag>
        </matrix_element>
        .
        .
        .
      </starting_state>
      <final_state>
        <matrix_element><!-- o-->
          <Real>REAL COMPONENT</Real>
          <Imag>IMAGINARY COMPONENT</Imag>
        </matrix_element>
        .
        .
        .
      </final_state>
    </testcase>
  </testset>
</testsuite>
```

## G Available Algorithm Instructions

Instruction	Action		Instruction	Action
<b>Create_H</b>	Create a Hadamard Gate		<b>Create_CH</b>	Create a Controlled Hadamard Gate
<b>Create_X</b>	Create a Pauli-X Gate		<b>Create_CX</b>	Create a Controlled Pauli-X Gate
<b>Create_Y</b>	Create a Pauli-Y Gate		<b>Create_CY</b>	Create a Controlled Pauli-Y Gate
<b>Create_Z</b>	Create a Pauli-Z Gate		<b>Create_CZ</b>	Create a Controlled Pauli-Z Gate
<b>Create_R</b>	Create a R(k) Phase Gate		<b>Create_CR</b>	Create a Controlled R(k) Phase Gate
<b>Create_V</b>	Create a V Gate		<b>Create_CV</b>	Create a Controlled V Gate
<b>Create_W</b>	Create a W Gate		<b>Create_CW</b>	Create a Controlled W Gate
<b>Create_RX</b>	Create a Rotate-X Gate		<b>Create_CRX</b>	Create a Controlled Rotate-X Gate
<b>Create_RY</b>	Create a Rotate-Y Gate		<b>Create_CRY</b>	Create a Controlled Rotate-Y Gate
<b>Create_RZ</b>	Create a Rotate-Z Gate		<b>Create_CRZ</b>	Create a Controlled Rotate-Z Gate
<b>Create_SWAP</b>	Create a SWAP Gate			
<b>Iterate</b>	Run Sub-Algorithm[0] for $n$ iterations		<b>RevIterate</b>	Run Sub-Algorithm[0] for $n$ iterations in reverse order
<b>Body</b>	Perform each Sub-Algorithm in turn			
<b>Create_Custom1</b>	Create Custom Gate Number 1		<b>Create_CCCustom1</b>	Create a Controlled Custom Gate Number 1
<b>Create_Custom2</b>	Create Custom Gate Number 2		<b>Create_CCCustom2</b>	Create a Controlled Custom Gate Number 2
<b>Create_Custom3</b>	Create Custom Gate Number 3		<b>Create_CCCustom3</b>	Create a Controlled Custom Gate Number 3