The Genetic Evolution of Quantum Programs Using The ZX-Calculus

Project Dissertation

University of Oxford
Department of Computer Science

Kenton Barnes

MSc in Computer Science 2019-20
September 2020

Abstract

This project is the first to use the ZX-calculus as an internal representation for an evolutionary computation system that synthesises quantum programs. Specifically, we use the open graph representation of ZX-diagrams in a genetic programming system to solve a series of benchmark problems. We found that when compared to previous works that used the quantum circuit model as their internal representation, the naturally reduced redundancy of the open graph representation provided significant advantage to our system. We evolved solutions to some problems much more often than the related work. However, the mutations defined over the open graph representation proved less effective than those used in previous works defined over the circuit model.

Keywords

Quantum Computing, Automatic Programming, Quantum Circuits, Genetic Algorithms, ZX Calculus, Evolutionary Computation,

Abbreviations

GP *Genetic Programming*, QFT *Quantum Fourier Transform*, GDO *Grovers Diffusion Operator*, SND *Semantic Neutral Drift*, Randomised Unitary (RND)

Contents

Chapter 1

Introduction

Quantum computers make use of quantum physics to achieve an advantage over classical computation. Such devices have been theoretical up until the last few years, but now quantum hardware groups report to have devices that can perform some computations faster than classical computers [1, 38]. Although widespread commercial use is still years away [39], research efforts continue on both the hardware and software side to make the supposed benefits of quantum computing, in areas such as drug-design, cryptography and artificial intelligence, a reality.

Historically, programs for quantum computers have been created manually by concatenating small units of computation, called quantum gates, together. However, the hardware that quantum scientists are writing programs for is improving in quality and scale. This pattern is set to continue as hardware developers continue to achieve new levels of gate fidelity and qubit counts [40, 38]. This means larger programs that use more qubits, the smallest unit of quantum information, can be run. It will no longer be the case that only a handful of instructions can be executed. Programs with hundreds of operations across dozens of qubits will soon become the norm. Consequently, it will become in-feasible for programmers to control their quantum devices by writing a few lines in a low-level instruction language. Instead, to best make use of the available hardware, automatic methods that can help to produce larger and more optimal programs are needed.

Existing tools for assisting quantum programmers vary in function from optimisation of low-level code written by humans [45], to languages that allow users to only engage with high-level descriptors of quantum algorithms [49]. The type of automated assistance this project will focus on are systems which automatically synthesise entire quantum programs. Given a list of input cases and expected out-

puts, a quantum program synthesis system should produce a quantum program that returns the expected outputs and is as small as possible.

More specifically this project deals with approximate program synthesis, where an acceptable threshold of error in the outputs from the produced program is tolerated. This approximate version of the program synthesis problem also allows systems to explore a trade-off between error in its outputs and the size of programs produced. The ability to choose smaller programs with a small rate of error could be of great advantage to near-term programmers of Noisy Intermediate-Scale Quantum devices, where the accuracy of running a program on the device can drastically decrease when running larger programs [39].

Evolutionary computation is a collection of heuristic driven optimisation methods that are inspired by genetics and evolution. One of the applications of evolutionary computation methods has been the creation of novel classical programs to solve problems defined by a given fitness function. While the quantum physics that underpins quantum computation is not intuitive to humans, a computer's 'understanding' of a candidate program in some evolutionary computation system is purely numerical; it takes the form of an assigned fitness value. Along with this, the actual representation of a quantum program can be similar in form to that of a classical program: namely a list of small instructions to be executed in order. These similarities in how an automated system might deal with classical and quantum programs have meant that evolutionary computation methods are just as applicable to the synthesis of quantum programs as they are to classical programs.

There have been many previous works that have created successful evolutionary computation systems for generating quantum programs. The most common method has been to use genetic programming (GP) [6, 31, 42, 47], but other works have also used Ant Colony optimisation techniques [2]. One thing that these works have in common is the use of the circuit model for quantum computation as the way of representing candidate solutions in their systems. The circuit model is similar in form to that of Boolean logic circuits in classical computing, made up of a group of interconnected gates acting on a finite number of qubits. It is used by these works because it is easy to manipulate using automatic methods. Operations such as gate addition and removal are both simple to implement and allow a smooth traversal of the semantic space for quantum programs.

The ZX-calculus is a diagrammatic way of representing quantum programs and has been an important area of study since its inception in 2008 [11]. It has been gaining popularity among quantum computing research groups because of results that show how ZX-diagrams can be reduced down using well-defined transition rules. For example, optimisation methods using the ZX-calculus are some of the best for reducing the T-count of quantum circuits [28]. These transition rules also make it easy to remove redundancy from a ZX based representation compared to the circuit model. This reduced redundancy, and therefore increased semantic efficiency of the representation, makes it a good candidate for being the representation used in evolutionary computation software for quantum program synthesis.

This project aims to create a system that uses GP to evolve quantum programs, where all candidate programs are represented as ZX-diagrams. This project is the first work to use the ZX-calculus for this purpose. By making comparisons between our system and previous works that used the circuit model, we evaluate how well the ZX-calculus functions as the basis for candidate representation in evolutionary computation systems. We use the open graph representation of ZX-diagrams, which offers reduced redundancy compared to the circuit model. We assess whether any performance gained by our system is through the reduced redundancy native to open graphs, or through the mutation operations we introduce which have no analogue in the circuit model. We also investigate how well using the ZX-calculus allows the system to reduce the size of the evolved programs, and whether the system can trade-off accuracy in the evolved program outputs for decreased program size.

As not all ZX-diagrams represent valid quantum programs, this project goes some way towards investigating what methods are best to deal with this weakness of a ZX based representation in GP systems. We propose four different ways to mitigate against the introduction of these invalid individuals into the system, and then include an evaluation of these methods to justify our choice in the method we use for the rest of the work.

This report provides background information for quantum computing, the ZX-calculus and genetic programming in Chapter 2. It then goes on to detail our methodology for investigating the research aims of this project and implementation of the designed system in Chapters 3 and 4. In Chapter 5 we discuss our evaluation methodology and give results comparing different versions of our system to each-other and previous works in this area of research. Our conclusions, detailed in Chapter 6, are that the main advantage of the ZX based representation is not the different mutation operations that it allows, but rather the reduced redundancy it

can enable compared to the circuit model. Overall the ZX-calculus was a successful internal representation that allowed this project to achieve better results than previous work, solving problems quicker and more reliably. However, more research should be done to find mutations defined over the ZX-calculus that can efficiently traverse the semantic space of quantum programs.

Chapter 2

Background

This background section is indented for those who have a firm basis in computer science but whom may not have experience with quantum computing or evolutionary computation. While some linear algebra is used to build up the basic concepts in quantum computing, a detailed understanding of it is not needed to engage with the content of this report.

For a more complete resource introducing the topic of quantum computation we recommend Nielsen and Chuang (2002) [35], while for diagrammatic methods and the ZX-calculus we recommend Coecke and Kissinger (2018) [12] as the best introductory resource.

2.1 Quantum Computation

Since the inception of modern computer science in the 20th century, the capabilities of most computers have not exceed that of the Turing machine, methods that can be represented by paper-and-pen workings or classical physics. The machines and associated computational methods bound by these limitations are referred to as being *classical* throughout this work.

In 1982 Richard Feynman proposed that the physics seen in quantum theory could be harnessed to somehow provide a benefit to the field of computation [16], and take computers beyond the methods of classical computing. We have since seen that quantum computing can indeed provide a speed-up and allow previously intractable problems to be solved in polynomial time. Peter Shor proved the most famous example of such a speed up, with the creation of his famous quantum algorithm for factorising integers in polynomial time [44].

Quantum computation is defined as any computation that is achieved through the use of quantum theoretical laws such as entanglement and superposition. This broad definition encompasses many different models for computation, but initially section will focus on the one of the most common approaches: the quantum circuit model [35].

In classical physics, the simplest physical system is the bit. This is an abstract notion with two possible states, often referred to as '0' and '1'. In real computers there are several physical implementations of a bit, such as directions of magnetisation on a disk, or the value of a voltage in some part of an electronic circuit. A bit is the smallest element of data in classical computing, and the smallest units of computation are defined over small numbers of these bits. An example of a small computational unit would be a logical gate like AND or NOT.

In quantum physics, the simplest system is the qubit. Like the bit, this is just an abstract notion that has two possible observable states, we will label these states $|0\rangle$ and $|1\rangle$. These labels are from Dirac notation, which is a way of hiding the matrix notation that we will also be engaging with. In this notation, states correspond to column vectors.

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

$$|1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Unlike the bit, these observable states are not the only states a qubit can be in. Let's use a a small quantum operation, called the Hadamard (H) gate, and apply it to a qubit in the $|0\rangle$ state. In matrix notation, quantum gates are all unitary matrices, and then the application or sequential composition of gates is done through matrix multiplication. Here is the unitary matrix for the Hadamard gate.

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

$$H|0\rangle = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

It is not immediately obvious how we interpret the resultant state. It seems to be both a $|0\rangle$ and $|1\rangle$ added together. When we measure this state we will observe $|0\rangle$ half of the time and observe $|1\rangle$ the other half; we have encountered the proba-

bilistic nature of quantum computing. We say that the system we measured was in superposition of states. In general, a single qubit superposition state $|\psi\rangle$ takes the following form:

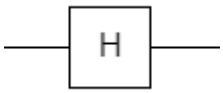
$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle = \begin{bmatrix}\alpha\\ \beta\end{bmatrix}$$

$$\text{s.t. } |\alpha|^2 + |\beta|^2 = 1$$

The probability that $|0\rangle$ is observed when measuring $|\psi\rangle$ is $|\alpha|^2$ and the probability that $|1\rangle$ is observed is $|\beta|^2$. An important caveat of making an observation of a system in quantum computing is that this has an effect on the state of the system. Going forward the system is no longer in a superposition state, but rather has *collapsed* to the state which was observed. It is this destructive nature of measurement that has made it so hard to implement reliable quantum computers.

There are some important superposition states we need to define for use later on. We have already seen $|+\rangle$, defined as $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and we can also define $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

Superposition is an important concept in quantum computing, but perhaps more important for the computational speed-up that quantum computing offers is quantum entanglement. To introduce entanglement we will use a quantum circuit [35], which makes it easier to visualise the application of multiple quantum gates across multiple qubits. The circuit model is one of the most widely-used methods for representing quantum programs. It is similar to logical circuits from Boolean logic in classical computing. Logical circuits are made up of 'gates', which describe a simple Boolean operation on a small number of bits, and 'wires', which connect gates together and represent the flow of information for a single bit. This section has already been using the gates defined in the circuit model, and we will now go on to visualise these and connect them with wires that represent qubits. Often quantum circuits are more static than Boolean logic circuits, in that the number of qubits visible throughout a diagram does not change. Quantum circuits are read from left to right, with the input being given at the far left of the circuit. For the purposes of this project, the reader can assume that all measurement takes place at the end (far right) of the circuit. Most gates are pictures as a labelled boxes. For example, here we can see the Hadamard gate on a single wire.



To use the circuit model to produce an entangled state we will need to use 2 qubits. The states of a 2-qubit system are constructed from the different combinations of Kronecker (tensor) products of states $|0\rangle$ and $|1\rangle$. This is because the Kronecker product corresponds to parallel composition. We have already seen that matrix multiplication corresponds to sequential composition; note that both of these operations preserve the unitary nature of gates, meaning the circuits that gates build up are also unitary operations.

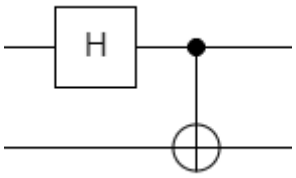
The observable states for a 2-qubit system are as follows.

$$|00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} \quad |01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} \quad |10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} \quad |11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Next we need to define a gate that acts on 2 qubits called CNOT. The CNOT gate is the conditional NOT operation. It has a control qubit and when this is in the $|1\rangle$ state it performs a NOT operation on the defined target qubit.

$$CNOT = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

In the circuit model the CNOT gates gets a more descriptive visualisation than just a labelled box. We represent the control qubit using $\bullet$, the target is represented by $\bigoplus$, and these are connected by a vertical line. Let's now consider the following circuit, and assume that the input given is $|00\rangle$.



If we were to use multiplication and Kronecker products to work through the matrix notation for this circuit we would get the following resultant state.

$$\frac{1}{\sqrt{2}}\begin{bmatrix}1\\0\\0\\1\end{bmatrix} = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

If we were to measure this system we would have an equal chance of observing $|00\rangle$ or $|11\rangle$. In classical computing, any state can always be represented as the parallel composition of each of the component bits. The state we have from the above circuit breaks away from that concept, as there are no two single qubit states that can composed in parallel with each-other to make this state. We say that the two qubits we have are entangled. If we were to measure just one of them we would know which of the two possibilities we had forced the system to collapse to, and therefore know what we would observe if we were to measure the second qubit.

While we have managed to construct an interesting entangled state with just the Hadamard and CNOT gates, this *gate set* is not universal. That is to say that there are quantum operations that we cannot represent using any number of Hadamard and CNOT gates composed together. The most common example of a universal gate set is Clifford+T. This is the set {Hadamard, CNOT, S, T} where S and T are both different types of the parameterised Z rotation gate: Z_α.

$$Z_\alpha = \begin{bmatrix}1 & 0\\0 & e^{i\alpha}\end{bmatrix}$$

For S we have $\alpha = \pi/2$ and for T we have $\alpha = \pi/4$.

2.1.1 The Quantum Speedup

With the background given so far in this report, it may be hard to see why quantum computers are thought to provide a performance benefit over classical computers. To understand this, we can think about how classical computers may simulate quantum computers. Essentially they have to go through the operations we explored in the previous section. We can remind ourselves that when we had a 2-qubit gate we had to use a 4×4 matrix to describe the operation said gate. Extending this to larger circuits we see that for an n-qubit circuit we need a $2^n \times 2^n$ matrix to describe it.

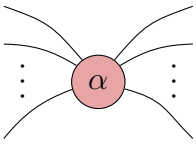
We need resources exponential in the number of qubits to simulate a given quantum circuit. While there are certain families of quantum programs that can be simulated in polynomial time [22], in general, the worst case is thought to be exponential.

While the hardware for quantum computers is improving [40, 38], for general purpose applications classical computers simulating quantum computers are still the fastest and most accessible way for researchers to engage with quantum computation, despite the exponential worst-case performance cost. For this project this means that, like all related work, the quantum programs our system produces will be simulated on a classical computer rather than being run on a real quantum computer. As this simulation time of thousands of candidate programs takes up a majority of the run-time for current quantum program synthesis systems, it will be interesting to see how this area of study changes if quantum hardware can reduce this run-time in the future.

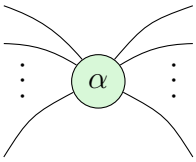
2.2 The ZX Calculus

While the circuit model reasons about quantum programs as unitary transformations on qubit registers, it is possible to generalise this to any linear map between qubit registers. The ZX-calculus [11] achieves such a generalisation.

The ZX-calculus is a diagrammatic approach to reasoning about quantum programmes. Like the quantum circuit diagram approach, we have wires that connect some basic units of computation. However, rather than quantum gates, the computational units are called spiders, of which there are only two kinds: Z and X. Spiders are more flexible than gates in that they can have any number of inputs and outputs, but we can still give them a definition in terms of a linear map between these inputs and outputs. This definition features an expansion of the Dirac notation already seen where for a column vector $|\psi\rangle$: $\langle\psi|$ is the adjugate of that vector.



$$= \quad |+...+\rangle\langle+...+| + e^{i\alpha}|-...-\rangle\langle-...-|$$



$$= \quad |0...0\rangle\langle 0...0| + e^{i\alpha}|1...1\rangle\langle 1...1|$$

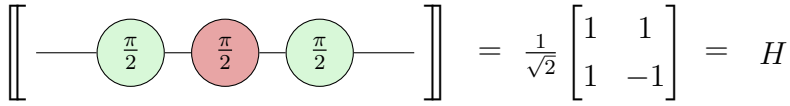
We distinguish between spiders by their colour, green meaning the Z spider and red meaning the X spider. We can see that each spider takes a single parameter that is called the phase. When a spider has a phase of 0 we leave the spider blank when drawn.

To give meaning to our ZX-diagrams we will be thinking about their *interpretation* [25], which is the linear map that a given diagram represents. For some diagram D we notate the interpretation as $[[D]]$. We will also use this to define a more flexible version of equivalence on diagrams. For two diagrams D_1 and D_2 and some complex number z s.t. $z \neq 0$: $D_1 = D_2$ when $[[D_1]] = z[[D_2]]$. We call this equivalence up to a global scalar. These scalars are implementation details that only get in the way for the purposes of this paper. There are several resources in literature that do give the values for these scalars [26, 34].

We can represent gates from the previous section using these spiders. An example is the Z_α gate.

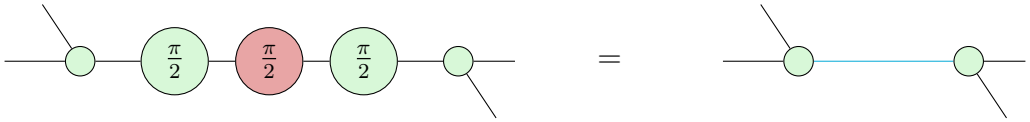
$$[\![\; \text{—}\alpha\text{—} \;]\!] \quad = \quad \begin{bmatrix} 1 & 0 \\ 0 & e^{i\alpha} \end{bmatrix} \quad = \quad Z_\alpha$$

Some equivalences between gates and spiders are more complex, requiring multiple spiders. Spiders and wires can be composed in parallel and in sequence to build up these larger diagrams using the tensor product and multiplication operations that we used in the circuit model. An example of a multi-spider gate equivalence is the Hadamard gate.

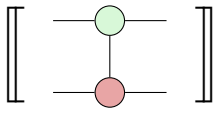


$$= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = H$$

Such is the commonality of this gate, we often use the convention of just drawing a blue edge between spiders where there is a Hadamard gate.

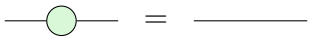


Another multi-spider gate equivalence is for the CNOT gate.

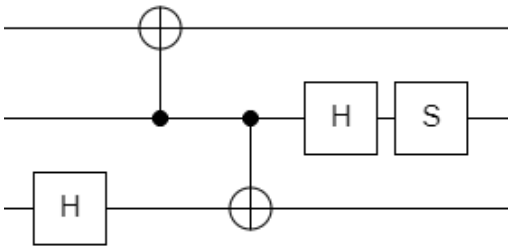


$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \text{CNOT}$$

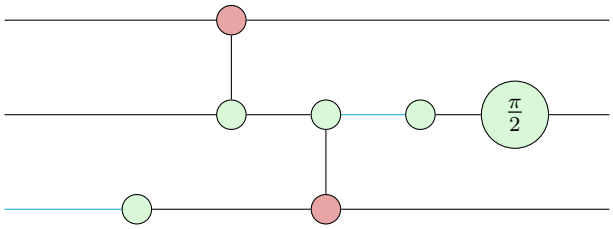
At this stage it is worth noting two identities in the ZX-calculus. Firstly that a 0-phase Z or X spider with one input and one output is just a wire. And secondly that two Hadamard gates composed with each-other is also just a wire.



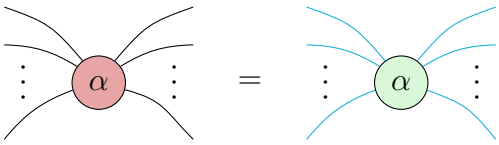
We have been building up equivalences between elements of the circuit model and ZX-calculus with the aim of being able to convert any circuit into a ZX-diagram. Consider the following example circuit.



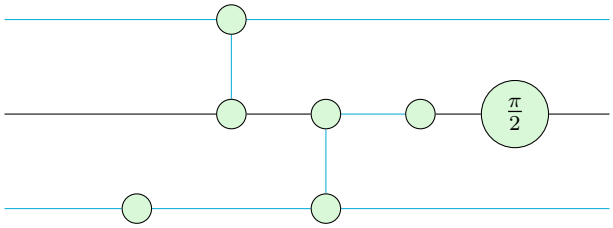
Using what we have seen so far we can use the gate equivalences to convert the example circuit to a ZX-diagram.



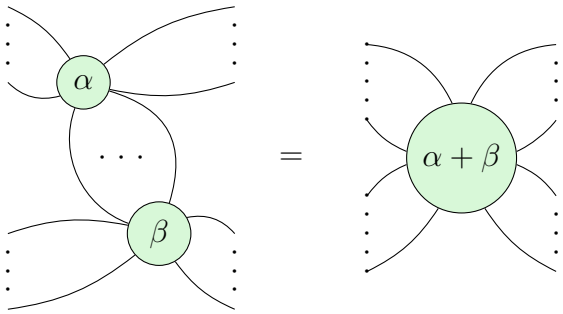
There are now some transformations that it might benefit us to define and apply to our very circuit-like ZX-diagram in an attempt to put it into a simpler form. The first is the rule of colour change. This introduces a relationship between Z and X spiders via the Hadamard gate.



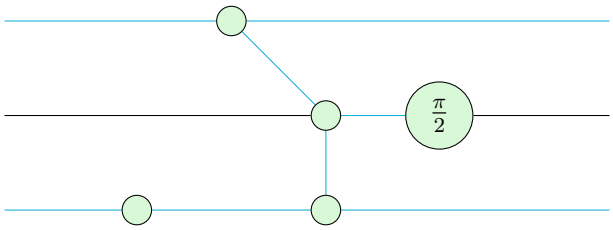
Applied to our circuit-like ZX-diagram on the X spiders, colour change gives us the following.



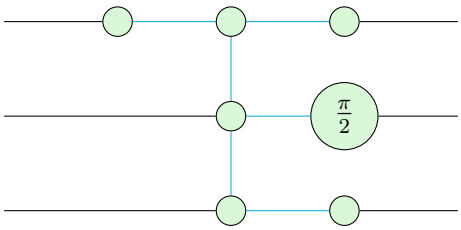
The next rule is spider fusion, where connected spiders of the same colour are allowed to fuse together by adding their phases.



Applied to our circuit-like ZX-diagram, spider fusion gives us the following.



In the simplified representation of ZX-diagrams we are trying to transform our diagram into, we cannot have Hadamard edges attaching to the edges of the diagram. So we can now use the inverse of colour change and the previously stated identities for Hadamards and the 0-phase spiders to get rid of the bottom left spider and produce the following form.



This now meets the definition of a *graph-like* ZX-diagram, because: all spiders are Z-spiders, these spiders are only connected to each-other via Hadamard edges, the adjacency matrix of edges is lower- or upper-triangular as there are no self loops or multiple edges between the same two spiders, each input and output is connected to a single spider, and each spider is only connected to at most one input and one output. While in this example the number of spiders in the diagram has not been reduced since we first converted our example circuit to a ZX-diagram, the amount of information needed to store this diagram has been. This is because there is now only one type of edge between nodes (the edges that go to the inputs and outputs of the diagram can be assumed), and there is now only one colour of spider. Using equivalences between gates and spiders as well as the spider fusion and colour change transitions we have shown how any circuit can become a graph-like ZX-diagram. The work of Kissinger and Wetering (2020) [28] further shows that any ZX-diagram, not just those derived from circuits, has an equivalent graph-like representation. That is to say that the space of graph-like ZX-diagrams is not a restriction on what can be represented.

A related concept to graph-like diagrams is that of the *open graph*. Open graphs represent graph-like ZX-diagrams in the following structure; some triple (G, I, O)

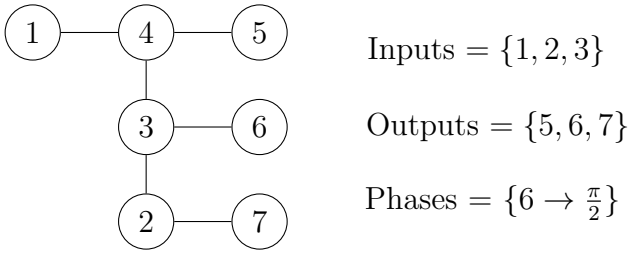


Figure 2.1: Open graph representation of ZX-diagrams. Consisting of connectivity graph, input set, output set and phase map.

where G is the undirected graph with edge set E and vertex set V. With vertices representing the Z-spiders and edges representing the Hadamard edges in a graph-like ZX-diagram. $I \subseteq V$ is the set of inputs and $O \subseteq V$ are the outputs. An open graph, along with an assignment between vertices and phases, completely describes a given graph-like ZX-diagram. In this report we will display open graphs as seen in Figure 2.1. We show the underlying graph representing the connections between Z-spiders via Hadamard edges. Each spider is given a numerical label. The set of inputs and outputs is also given. If needed, the map between spider index and phase is given, where if a spider is not represented in the map the phase is assumed to be 0.

2.2.1 Diagram Reduction

The work of Kissinger and Wetering (2020) [28] is part of an area of research that use the ZX-calculus, and transition rules defined over it, to optimise quantum circuits. Specifically, Kissinger and Wetering take quantum circuits, convert them to graph-like ZX-diagrams, run a series of semantic-preserving transition rules defined over graph-like diagrams that further reduce the size of the ZX-diagrams, and finally convert the diagrams back to quantum circuits (Section 2.2.2). The type of optimisation this process aims to achieve is T-count reduction. The T-count of a circuit is how many Z_α gates there are such that $\alpha \neq \frac{k\pi}{2}$ for any $k \in \mathbb{Z}$. In the previously mentioned Clifford+T gate set, the only gate that meets such a definition is the T gate. Therefore, when run on Clifford+T circuits, the work of Kissinger and Wetering attempts to minimise the number of T gates. This is an important task because the T gates are harder to implement on quantum hardware, and simulate on classical computers, than the other gates in Clifford+T [10].

The main result of the Kissinger and Wetering paper is a reduction scheme that manged to achieve the lowest known T-count for 72% of benchmark problems. This area of research using the ZX-calculus to optimise circuits continues to be one of the most successful applications of this quantum program representation [15, 14].

While the details of the transition rules they define over graph-like diagrams for the purpose of reducing their size will not be discussed here, readers should know that they only really work when the quantum circuit, and corresponding graph-like diagram, contain gates with specific phases. That is, some of the rules used do not work on spiders of arbitrary phase, rather they necessitate the phase be some multiple of a fraction of π in order to be applicable.

2.2.2 Circuit Extraction

It is not currently possible for all ZX-diagrams to be 'run' by a quantum computer. There are often ambiguities about what 'running' a given diagram would mean, about when operations would be carried out and on which qubits they would apply to. While in the future this may change, current quantum computing hardware and classical simulators of quantum computers take input in the form of a quantum circuit. This less ambiguous representation of a quantum program can simply be interpreted as a list of instructions to execute, so is easier for computers to deal with.

It is therefore desirable to convert ZX-diagrams to quantum circuits, for example this was a step required in the circuit optimisation process in Section 2.2.1. This problem of *circuit extraction* from ZX-diagrams is not trivial. In Section 2.2 we detailed how to convert circuits to ZX-diagrams and this used simple equivalence between gates and small ZX-diagrams. However, to then get into the open graph representation we transformed our diagram using rules such as colour change and spider fusion. These rules changed the diagram into something that we could no longer just apply the inverse of our gate equivalences to in order to get a circuit back out of the diagram. Circuit extraction is a difficult problem because of these rules like spider fusion that have, in general, no equivalent in the circuit model.

The most extensive work on circuit extraction is that of Backens et al. (2020) [4]. They prove the worst case performance of their extraction method to be $O(n^2k^2+k^3)$ where k is the number of qubits in the system and n is the number of inputs to the diagram[i]. Their method works by progressing forward a *frontier*, one side of which

[i]In general, $k \neq n$ but in this project we always have exactly one input for each qubit.

contains the circuit extracted thus far and the other side contains the diagram yet to be extracted. They define a collection of rules than can be applied to progress the frontier, with different rules being applied depending on the diagrammatic structures seen at the frontier.

It is not possible for all ZX-diagrams, or even graph-like ZX-diagrams, to be converted into quantum circuits. If we were to run the circuit extraction procedure on a diagram that was not extractable, eventually the frontier would reach a state where none of the defined rules to progress the frontier were applicable. As mentioned previously, while quantum circuits represent unitary transformations between registers of qubits, the ZX-calculus can be used to represent any linear-map between qubits; so it makes sense that not all ZX-diagrams have an equivalent circuit.

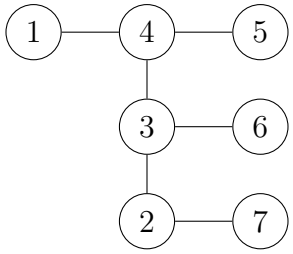
Characterising the class of ZX-diagrams that can be extracted into circuits is an important task. Often researchers want ensure certain transformations performed on extractable ZX-diagrams retain extractability. Works such as that of Backens et al. achieve such a characterisation by proving that their circuit extraction method works exactly when the given open graph has a generalised flow, or *gFlow* [9]. This is a mathematical property that is defined on open graphs, and it has been a great tool for researchers in this area.

For the purposes of this project we will consider a simplified version of gFlow, that we will understand as a recipe to follow to win a game played on an open graph. Consider a game where we start by associating a random bit with each node in an open graph. A player wins the game when they get the system into a state where there are no 1's on the graph apart from at the output nodes of the graph. A player does this through the use of the $\mathbf{flip}_v$ operation. When $\mathbf{flip}_v$ is performed on some vertex v in the graph, the NOT operation is applied to all the bits on all nodes that neighbour (share an edge with) v.

A recipe for how to solve this game for all possible starting configurations (values of the bits associated with each node) on a given open graph would consist of two things. Firstly an ordering $\prec$ over all nodes where the ordering starts with inputs to the diagram and ends with the outputs. Secondly, a *correction set* for each node which will tell us what to do if we find a '1' on a given node. We will call this set $g(v)$ for some node v. Each node $w \in g(v)$ must adhere to $v \prec w$. The definition of $g(v)$ is thata after performing $\mathbf{flip}_w$ for all nodes $w \in g(v)$ the end result must be that v has flipped, and that the only other bits that have changed are in $g(v)$ or are output nodes. To follow this recipe to solve a game we first find the minimum node according to $\prec$ that is marked with a 1, call this node v. Then we perform $\mathbf{flip}_w$ for

all nodes $w \in g(v)$. We repeat these two steps until all the 1's present are only on the outputs nodes.

We can give an example using the open graph we created in Section 2.2. We can define $\prec$ using the index within each node. Note that the output set is $\{5, 6, 7\}$.



The correction sets are then $g(1) = \{4, 6\}$, $g(2) = \{7\}$, $g(3) = \{6\}$, $g(4) = \{5\}$. Correction sets are not needed for outputs. We could follow this recipe and it would win the game for any starting state on this graph. The presence of this recipe for solving this game for this open graph means that this open graph has a gFlow and thus could be extracted back into a circuit using the methods seen in Backens et al. (2020) [4]. If an open graph does not have such a recipe, it does not have a gFlow and is therefore currently thought to be non-extractable.

2.3 Evolutionary computation

Evolutionary computation is an umbrella term for a number of algorithms and methods inspired by genetics, evolution, and biology in general. They are optimisation methods often centred around the randomised iterative improvement of a collection of candidate solutions. This improvement is guided by some heuristic. In the case of using evolutionary computation for creating computer programs, this heuristic represents how close a given candidate is to the desired output program.

2.3.1 Genetic Algorithms

One of the most popular methods in evolutionary computation is the genetic algorithm [19]. Inspired by Darwin's theory of evolution, the process revolves around some selection function that only allows certain candidates to 'survive' into future iterations of the algorithm. Another key part of genetic algorithms is that the elements we want to optimise about a candidate solution are encoded in *genes*, collectively known as *genetic material*.

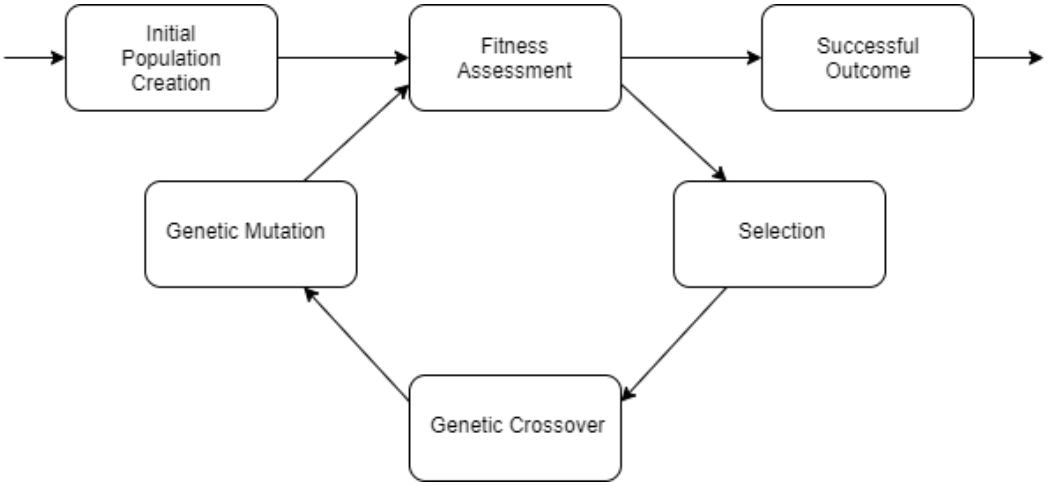


Figure 2.2: Overview of a Genetic Algorithm

The algorithm starts with an initial *population* of randomly generated candidate solutions or *individuals*. These are then each assigned a numerical *fitness* that represents how close a candidate solution is to the target solution. A selection algorithm is then run. This selects the individuals that will make up the population in the next iteration (*generation)* of the algorithm. It uses the fitness of the individuals to make this selection, with it more commonly selecting fitter individuals. There are a number of selection algorithms [21], and nearly all employ stochastic elements to not just select the fittest individuals. This is because the system can benefit from the increased genetic diversity of the population if some of the less fit individuals are selected.

The next stage is to take the selected solutions and perform genetic crossover on them. This pairs up the candidates and takes some features from one parent and some features from the other to create offspring. These offspring then undergo genetic mutation. These are random changes to characteristics about the individual. For example, if an individual had a characteristic encoded by some numerical gene, a mutation might be to add a random value to this gene. To allow for some stability in the system, the crossover and mutation operations are not al-

ways performed, rather performed on an individual-by-individual basis with a given probability. These probabilities, known as the mutation rate and crossover rate, are important hyper-parameters for the system. To get the most out of a genetic algorithm, these need to be optimised along with other parameters that can be found throughout the system, for example many selection functions have them [21].

After crossover and mutation, fitness assessment takes place again and we start looping with successive iterations of the algorithm. This process repeats itself usually until either a fixed number of generations has passed or an individual has achieved a sufficiently high fitness threshold and the run of the algorithm is deemed to be successful.

In general genetic algorithms are good methods for exploring complex fitness landscapes and are often able to avoid getting stuck in local maxima because of the stochastic nature of the method. While gradient-based methods [43] can also achieve this, and often attain results quicker than genetic algorithms, it is not always possible to quickly attain the gradient of a given point in a fitness landscape. Program synthesis is a great example of a problem domain with a complex fitness landscape where it is hard to attain the gradient.

Genetic Programming

The use of genetic algorithms for evolving computer programs is known as genetic programming (GP). Pioneered by Koza (1990) [30], this approach was initially used to manipulate tree-based programs to solve problems defined by some fitness function representing the target program. The tree structure consisted of nodes that were either operators or terminals. Terminals would represent some absolute value or program input and then operators would take arguments from their children and pass results of an operation on these arguments up to their parent nodes. The value at the top of the tree would be used as the result of the program. Work in this area has gone on to develop a range of program representations. For example there are linear methods that use simple lists of instructions that store and operate on data in registers [8]. Another popular method are stack-based systems where data and program instructions are stored in a number of stack data structures [37, 48].

Common mutations defined over the domain of computer programs are to add an instruction, remove an instruction or to slightly change an existing instruction. Common crossover operations involve splitting programs up and re-combining them with their co-parent. More semantic crossover operations that perform one of the two parents programs depending on a conditional statement [33].

Some of the advantages of genetic algorithms make them very appropriate for use in the area of automatic programming. One advantage being the flexibility in how the fitness function can be defined, meaning a wide variety of programs can be solved by the same algorithm by just changing the fitness function. Most fitness functions for GP systems work by having the user give the system a list of input and expected output pairs. This test-case based fitness function will go through and run the candidate programs on each input, and evaluate how close the given output is to the expected output.

Another advantage of the flexibility of fitness functions for GP systems is the potential for multi-objective optimisation [29]. While getting the expected outputs for each test-case might be one objective, making sure the evolved programs remain as small as possible might be another. This could be achieved by linearly combining the test-case fitness and program size fitness or a producing a tuple of the different fitness values to give one a priority over the other in a lexicographic fitness.

There are some common disadvantages a GP system may suffer from. One being the size of the hyper-parameter space a system user might need to optimise over. With parameters such as population size, crossover rate, mutation rates and selection algorithm parameters; the hyper-parameter space can be very large. Systems might also take a while to run so evaluating many points in this space could take a long time. Optimal parameters are also likely to vary from problem to problem, meaning optimisation needs to be re-done when a new programming problem is tackled.

2.3.2 Ant Colony Optimisation

Another example of an evolutionary computation method, is Ant Colony Optimisation [17]. It is a method inspired by the way ants find the quickest path to food sources, by leaving pheromones along a their walked path. It was first adapted for programming tasks by Rout et al. (2000) [41] and later improved by Green et al. (2004) [23], where they used it to solve symbolic regression problems. The program representation used was the tree-based one as seen in GP. At each node in the tree there would be a pheromone table, with an associated pheromone value for each of the operators and terminals that could be used at this point in the program. Higher pheromones values are more attractive to ants, making it more likely that a given operator or terminal is selected to be used in a program. Initially all pheromones are set to 0.5 which corresponds to a uniform distribution across all possible operators and terminals at each position. Then the fitness of a program - where fitness metrics are worked out in a similar way to GP systems - increases the values of the selected operators' and terminals' pheromones. The value of all pheromones also decreases slightly with each iteration of the system, as way of encouraging other 'routes' to continue to be tried throughout the process and avoid getting the system stuck in local maxima. The literature also describes how simulated ants walking through the program tree can be used to perform genetic operators such as crossover [41, 23].

Chapter 3

Methodology

This chapter details the design of our GP system, describing what all the key components are and and why design decisions were made. This section also explains how different versions of our system have been created to investigate the primary research aim of this project; whether the ZX-calculus is an effective internal representation for evolutionary computation approaches for synthesising quantum programs.

3.1 Quantum Program Representation

Our GP system will mutate populations of quantum programs, but there are different ways of representing these programs. Related works have used the circuit model as a simple and easily mutable representation for their evolutionary programming systems [31, 47, 2]. But, this work is the first to use an internal representation based on the ZX-calculus instead.

We will be using the open graph representation of ZX-diagrams (Section 2.2) to represent our quantum programs. There are several reasons for this. Firstly, not all ZX-diagrams represent quantum systems that can be run as a quantum program. The first barrier to knowing how to execute a ZX-diagram is having no guarantee of knowing where the starting state of a qubit is fed into the diagram and where the measurement is made for each qubit. The open graph representation solves this by identifying a set of input and output nodes in the diagram, with one input and output marked for every qubit.

Another advantage of the open graph representation is the reduced redundancy it offers compared to the circuit model. Essentially this means many circuits can be reduced to the same open graph. Figure 3.1 shows several circuits which are all equivalent to the one we converted into an open graph in Section 2.2. All of these circuits reduce to the same, previously seen, open graph.

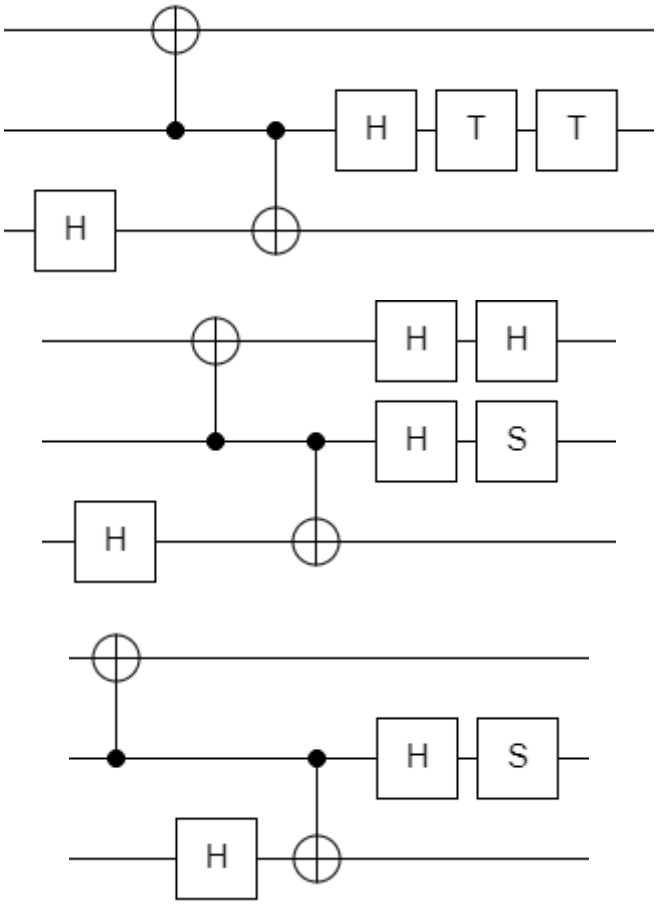


Figure 3.1: Three different circuits that are all equivalent, and all of which reduce to the same open graph

While the first two circuits become the same open graph due to the spider fusion transformation, the third demonstrates another kind of redundancy: indifference to some gate orderings when such orderings do not matter. Our hypothesis is that the reduced redundancy of open graphs compared to the circuit model means our program synthesis system will be able to traverse the fitness landscape quicker, as we have effectively reduced the size of the search space.

Note that we have applied a small restriction on the open graphs we use, in that we force the number of inputs to be the same as the number of outputs. This is because of how our fitness function (Section 3.4.2) is defined, as it provides input for all qubits and reads output for all qubits.

3.2 Genetic Operators

The design of good genetic operators (mutation and crossover) is important for the functioning of a GP system. The operators dictate the systems ability to change the semantics of individuals and thus explore the fitness landscape. In this work, we want to trial new mutation operators that mutate open graphs. We want to assess the capability of these mutations and compare them to the mutations used in previous works that operate on the circuit model. To do this we will run benchmarks on two versions of our system which vary in the mutation operators they can use. The first, *pure_zx*, will feature genetic operators that can only work on open graphs. An example might be adding an edge between two nodes, which in general has no simple analogue in the circuit model. The second system, *pure_cm*, will use mutation operators that feature in many of the previous works [31, 47]: the addition and removal of gates at random positions in a circuit. A comparison between these two systems will go someway towards answering the research aim for the project, but a further comparison is needed to assess the advantage that may be provided through the reduced redundancy that naturally comes from using open graphs. This reduced redundancy will feature in both the *pure_zx* and *pure_cm* systems as it happens regardless of the genetic operators used. Therefore, comparisons will be made between these two systems and a third system from a related work that uses evolutionary computation for quantum program synthesis.

3.2.1 Managing Unsafe Operators

A unique challenge for the design of genetic operators in the domain of open graphs is that not all open graphs have a gFlow. Individuals that have no gFlow cannot be extracted to a quantum circuit and hence cannot be run as a quantum program for the purpose of fitness assessment. While the initial population (Section 3.3) will only contain extractable individuals, some genetic operators may sometimes turn extractable open graphs into open graphs that are not extractable. We will call the genetic operators that have the potential to do this *unsafe*. This paper proposes several possible strategies to mitigate the effects of unsafe operators.

1. **Discarding:** When an unsafe operation results in a non-extractable circuit, discard the result and try another operation. While this stops the introduction of non-extractable individuals into the system, it also increases the time taken to perform genetic operators. For example, if we run an unsafe mutation and the result is non-extractable, not only has the computation required to perform the mutation so-far been wasted, but additional work is needed to undo the mutation, select another mutation, and perform another mutation.

2. **Non-selection:** Allow unsafe operations to happen, but never select the non-extractable circuits produced. This achieves a very similar effect to the discarding method, by not letting bad individuals propagate far into the run of the algorithm. However, this method effectively reduces the size of the population in the system. If at every generation a certain percentage of the pool of candidates are not valid for selection, then the effective size of the population and hence diversity of genetic material across the population is reduced. We also effectively waste fitness assessments, as the fitness assessment function is called on individuals which we know are non-extractable. We have already seen how circuit extraction is a polynomial cost operation (Section 2.2.2) and how circuit simulation is exponential (2.1.1), so these wasted fitness evaluations actually take less time than evaluations of extractable candidates. This means that comparisons between systems based on the number of calls to the fitness function, which is a common metric in previous work [31, 46], will disadvantage this approach compared to comparisons based on the real run-time of the system.

3. **Avoidance:** Do not use any unsafe operations. Note that the *pure_cm* version of our system features only safe operators, but this is because all of the mutations in that system are defined over the circuit model, not over open graphs. During this project attempts were made to look into creating a version of the system that used safe operators defined over open graphs. However, while there is a wealth of literature on semantic preserving safe operations for ZX-diagrams, operations that are safe and good at shifting the semantics of a ZX-diagram in a meaningful way are uncommon. It is possible to make small changes to a semantic preserving transformation to enable it change semantics, but the ways in which these operations can change the semantics of a program has been found to be insufficient to enable the genetic algorithm to fully explore the fitness landscape. This is an area where we encourage further work in, to find a set of effective safe operators that can efficiently traverse the semantic space of quantum programs.

4. **Acceptance:** Use unsafe operations, and allow their non-extractable results to be selected: albeit uncommonly because of the low fitness value they will be assigned by the fitness function. This approach could be effective in two scenarios. Firstly, if the unsafe operators used only produce non-extractable individuals very rarely, and the natural selection pressure against these badly-preforming individuals is enough to remove them from the population quickly, without the need to implement specific non-selection. Secondly, it is theorised that it is possible this approach could provide benefit to a system under certain circumstances. Namely, this less restrictive approach could end up taking a candidate out of extractable space, select this broken individual, then perform another mutation that puts it back into extractable space but with different semantics. In Section 3.3, this report shows that extractable open-graphs are sparse in larger open graphs, and therefore the occurrence of an individual travelling through non-extractable space to another more useful point in the solution space will not be common.

The *discarding* method has been used throughout the experiments in the main results section of this paper. It was chosen for use in our system over *non-selection* because the only disadvantage it has compared to *non-selection* is the increased run-time taken to repeat genetic operators. Section 5.1.1 details how our systems are assessed in terms of the number of fitness evaluations performed rather than absolute run-time, in order to be able to compare to previous works. This evaluation method heavily disadvantages *non-selection*.

We used the *discarding* method over *acceptance* because we did not believe our genetic operators would be able to take non-extractable individuals back into extractable space at any useful rate.

We can see these hypothesised advantages of *discarding* confirmed in Section 5.3, where we compared systems that used *discarding*, *non-selection* and *acceptance*.

3.2.2 Mutation Operators

In our system the amount of mutations that occur is configured by the mutation rate. This is the probability that an individual will undergo a mutation at the start of each generation. When this chance occurs, a particular type of mutation to perform must be chosen. In our systems each mutation is selected with an equal probability.

We will now describe all mutations implemented for use in our system. Some examples will be given of mutations executed on the open graph built up in Section 2.2,

the circuit and produced open graph can be seen in Figure 3.2. Where the input set, output set or phase map are not included in the result of a mutation the reader may assume they have not changed.

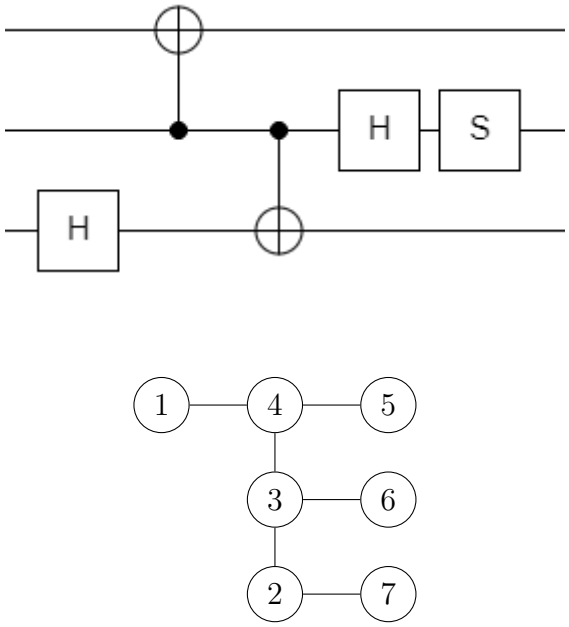


Figure 3.2: Example circuit and open graph

Edge Removal and Addition

One of the simplest ways to change an open graph is to add an edge between two previously unconnected nodes, or remove an existing edge. In this work we have split these add and remove edge operations up into several different mutations. We define *internal* nodes to be the nodes that are neither an output or input, and similarly *boundary* nodes as those that are an input or output. We can then consider three possible edge types, defined by the types of the two nodes they connect; internal-internal, internal-boundary and boundary-boundary. We then have the add edge and remove edge mutation for each of these types. These mutations work by identifying a list of valid edges that they could add or remove, and then randomly selecting one item from the list as the actual mutation to perform. These are all unsafe operations, and all are included in the *pure_zx* system.

An example of a successful edge addition on the example open graph leads to the open graph seen in Figure 3.3. This has added a boundary-boundary edge between nodes 6 and 7. As this only effects connections between output edges, the gFlow seen in Section 2.2 still holds.

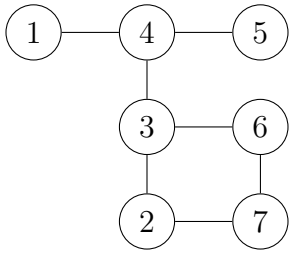


Figure 3.3: Result of boundary-boundary edge addition mutation

Another example can be seen in Figure 3.4, where the result of an internal-boundary edge requires us to change the recipe to solve the bit flipping game. We need to change the correction set for 2 to now be $g(2) = \{6, 7\}$.

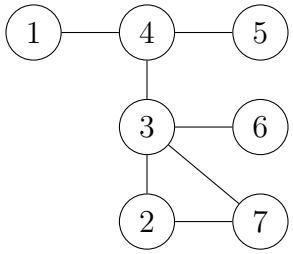


Figure 3.4: Result of internal-boundary edge addition mutation

We can see the result of a mutation that has broken the gFlow in Figure 3.5. Here, the remove mutation has removed internal-boundary edge between 1 and 4, and thus there is no valid correction set for 1. Such a diagram does therefore not have gFlow and is not extractable.

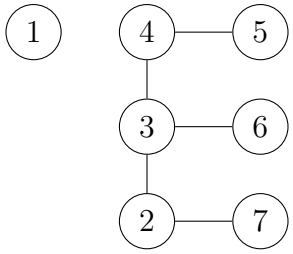


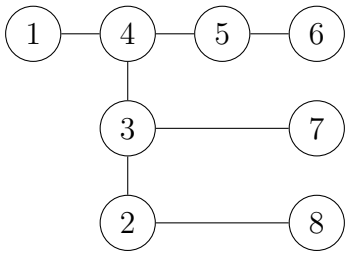
Figure 3.5: Result of internal-boundary edge removal mutation

Phase Change

The phase change mutation selects a node at random. It will then change the phase of the selected node by a value between $-\pi$ and $+\pi$, where all values are selected uniformly at random. It is a property of open graphs that all phase changes do not remove gFlow, hence the phase change mutation is safe. We can see this trivially using in our definition of gFlow, as the phase map was not used at any point. This mutation is used in the *pure_zx* system.

Add Node

This mutation selects an edge in the graph connecting some nodes (x, y). It then creates some node z and assigns it a random phase between 0 and 2π. It then removes edge (x, y) and adds edges (x, z) and (z, y). This mutation is used in the *pure_zx* system. An example of this mutation on the example open graph can be seen in Figure 3.6. The correction sets needed to show that this has gFlow are as follows: $g(1) = \{4, 6, 7\}$, $g(2) = \{8\}$, $g(3) = \{7\}$, $g(4) = \{5\}$ and $g(5) = \{6\}$.



$\{6 \to \frac{\pi}{2}, 7 \to \frac{\sim\pi}{20}\}$

Figure 3.6: Result of node addition mutation

Swap Edge

The edge swapping mutation is like an edge removal and addition at the same time. It first chooses some node x to be the pivot node, and then chooses one its neighbouring nodes, y. Then the method selects some node z such that $z \neq x$, $z \neq y$, and there is currently no edge (x, z). The mutation then adds the edge (x, z). This mutation is used in the *pure_zx* system. An example of this mutation can be seen in Figure 3.7. The result has gFlow if we modify the correction sets such that $g(1) = \{4, 6, 7\}$, $g(2) = \{7\}$, $g(3) = \{6, 7\}$ and $g(4) = \{5\}$.

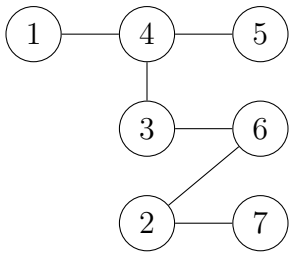


Figure 3.7: Result of swap edge mutation

Full Reduce

Results from previous papers studying GP systems suggests that using Semantic Neutral Drift (SND) mutations may benefit the system [3, 20]. SND mutations are defined as being those which do not change the semantics of a candidate, rather they perform some transformation that will change how the program is represented without changing what the program does. While they will not immediately effect the fitness of individuals, other than to perhaps reduce the size of a given program, they are thought to be advantageous for the overall exploration of the semantic space in subsequent generations. There has already been much work to find operations that could be used as SND mutations in the ZX-calculus. The works discussed in Section 2.2.1 find suitable operations that often have the goal of reducing the size of a ZX-diagram without changing its semantics [28]. Such is the wealth of research in this area, there is a large scope of possible SND mutations to try to use in a ZX based GP system. In this work we have only implemented one such SND mutation, and thus there is an opportunity for future experimentation in this area.

The SND mutation our *pure_zx* system will use is the scheme for reducing ZX-diagrams proposed by Kissinger and Wetering (2020) [28], which we will call *Full Reduce*. The first part of their approach is to get the diagram into a graph-like form, which we have already have as open graphs are graph-like. Subsequent operations in the Full Reduce method rely on the ZX-diagram containing specific structures that transformations can be performed upon. Many of these structures have to feature specific phases, such as integer multiples of π. The mutations discussed in this section are not conducive to such structures occurring regularly in our candidate solutions, with several mutations introducing randomly generated floating point phases. Therefore, a modified version of the system will be produced where all phases introduced will be multiples of $\frac{\pi}{4}$. This better matches the domain of

ZX-diagrams the Full Reduce method was designed to optimise, and hence in this version of the system the Full Reduce mutation should be more effective. This modified version of system is evaluated in Section 5.2, where we test it on a subset of the benchmark problems and compare it to the main *pure_zx* floating-point phase version of the system. Restricted phases were not used in the *pure_zx* system as we hypothesise that restricting the phases to multiples of a fraction of π leads to less smooth traversal of the fitness landscape and hence increases the likelihood of the system getting stuck in a local maxima.

Gate Addition

The *pure_cm* version of our system will trial mutations that have previously been used in works that had the circuit model internal representation [31, 47]. One of the most important types of mutation that these works have used is the addition of a single specified gate at a random point into the program. For us to implement this, we first use circuit extraction[i] to convert from ZX-diagram to circuit. Then a random position and qubit in the circuit is selected, and the specified gate is inserted there. If the gate requires further arguments like phase or additional qubits to act on, then these are also randomly generated. The gates we have chosen to add via this mutation are as followed: CNOT, X_π, X_α, Z_α, Hadamard, SWAP. For gates not discussed previously, gate definitions can be seen in Appendix A. Each of these gates is added to circuits by its own gate addition mutation. These operations are all safe, as adding a gate to a circuit just creates another circuit.

Figure 3.8 shows the result of adding a X_π gate on the middle qubit of the circuit used to build our example in Section x, and as we can see this single gate addition has caused a more complicated change in the open graph with the addition of a new internal node and a change of phase for another.

Gate Removal

The other mutation in the *pure_cm* system is simply to extract the open graph to a circuit, randomly select one of the gates in the circuit and remove it. An example could be taking our example circuit and removing the last H gate. Figure 3.9 shows what the circuit for this looks like and then gives the resultant open graph. The output set for this graph is now $\{3, 5, 7\}$.

[i]Where this paper talks about genetic operators that rely on being able to extract circuits, the operation simply does not take place if the input is non-extractable.

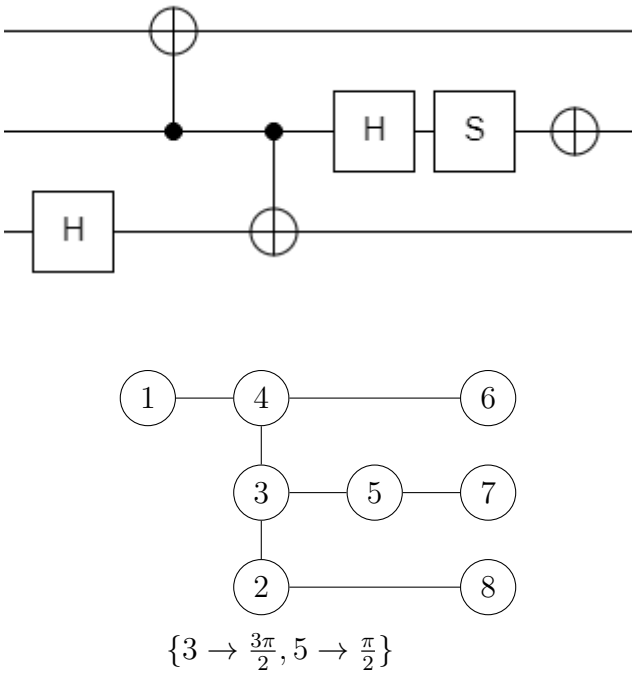


Figure 3.8: Result of gate addition, featuring resultant circuit and open graph

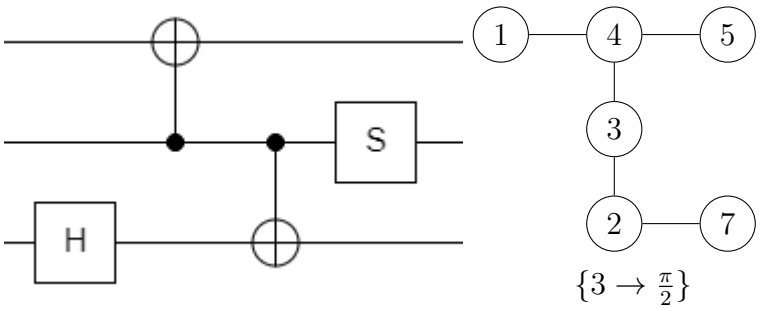


Figure 3.9: Result of gate removal, featuring resultant circuit and open graph

3.2.3 Crossover Operator

We use the common crossover operation from literature. Multi-point crossover over for quantum circuits. This meant extracting a circuit from both parent ZX-diagrams, converting these to a lists of gates and selecting 2 points in each list. Then we would produce two offspring from splicing and combining the gate lists at the selected crossover points. An example of this can be seen in Figure 3.10, where we start with our individuals already being converted to gate-lists and show the resultant gate lists after crossover. Gates are represented in this list using their name followed by

the indices of the qubits they act on. Colouring has been used to mark where the selected crossover points were and to indicate which child each gate went into.

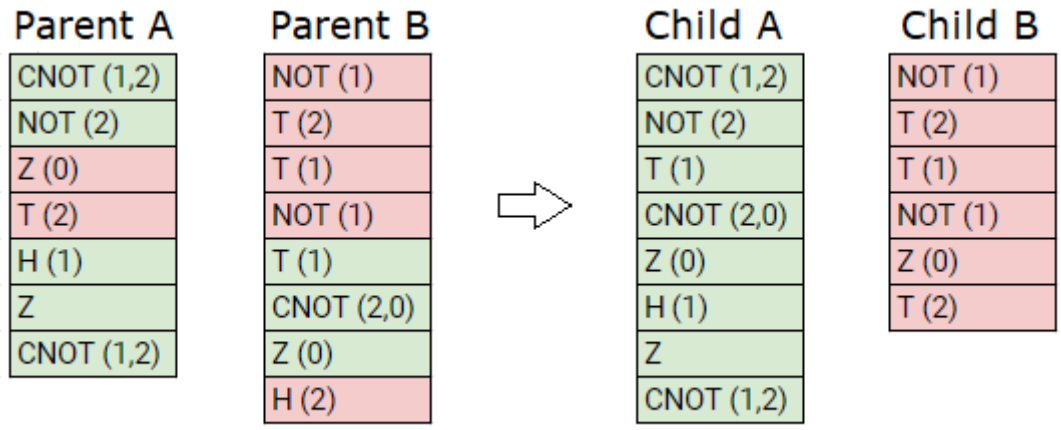


Figure 3.10: Example of crossover operation on gate lists

After this both children are converted into the open graph representation. If either parent is not extractable, then copies of both parents are returned instead of any crossover being performed.

In our system crossover is controlled by a crossover rate parameter. Given a list of pairs of individuals to become the parents of the next generation, the parents are crossed over with at the rate specified; otherwise they are copied into the next generation without crossover. Mutation then occurs after crossover, and manifests independently of whether crossover occurred or not.

3.3 Population Creation

The first population of an evolutionary computational method needs to be randomly instantiated, sometimes with a bias towards where you think a solution might be. The population creation method can be very influential for the success of a system, with the best methods producing a varied population that are distributed across the semantic space for the problem domain.

Throughout this work we have used one of the population creation methods from previous work. This works by having some set of quantum gates, we used Clifford-T, and to repeatedly randomly select some given number of these and place them

randomly into a quantum circuit. We then convert this circuit to our open graph representation. We chose a size of 10 gates for our randomly instantiated initial circuits.

Another population creation method was trialled during the design phase of this project. This was to randomly generate open graphs, and filter out those that were not extractable. The method took as input the number of qubits generated diagrams should have and generated an input node and an output node for each qubit. A second parameter was the number of internal nodes to have in the diagram. Lastly, the method would randomly add edges between each pair of nodes with a configurable probability. For example if given a probability of 10%, then on average 10% of all pairs of nodes would have an edge between them. This probability can therefore be thought of as the average connectivity of the generated open graphs.

To assess the suitability of this method we investigated how commonly the randomly created open graphs had gFlow. We varied two parameters while doing this: the number of internal nodes in the open graph and the connectivity of the graph generated. The number of qubits used in all diagrams was fixed at 4, though results were found to be similar for other numbers of qubits. Figure 3.11 shows the percentage of randomly generated graphs that had gFlow across the different settings for our random open graph generation method. 1000 repeat tests were used at each point to attain this percentage.

This experiment shows the sparsity of gFlow in open graphs at higher internal node numbers. As higher numbers of internal nodes corresponds to more complex extracted circuits, this area of the space was the most important to explore. Out of all 27,000 repeat tests with more than 5 internal nodes, across all graph connectivity values, only 5 had a gFlow. So while this method could be used to reliably create open graphs with few internal nodes, it was deemed unsuitable for use as an initial population creation method.

3.4 Individual Selection

The following section describes the full process of how individuals are selected to be the parents of the next generation in the genetic algorithm. As the individuals in the open graph representation are assessed in a very similar way to individuals in the circuit model, this component of the system, including the selection algorithm

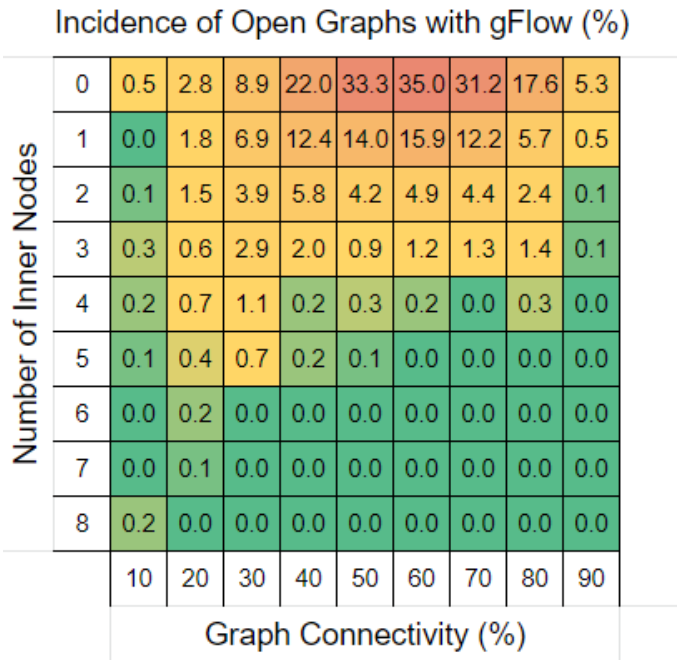


Figure 3.11: Heatmap to show the incidence of open graphs which have gFlow. Each data-point represents a percentage of randomly generated open graphs that have a gFlow out of 1000 repeat tests.

and fitness function, are similar to approaches seen in previous works [31, 47, 2].

3.4.1 Selection Algorithm

Tournament selection has featured in several related works [47]. It is a selection algorithm that values the ranking of fitnesses rather than absolute fitness values. It is a highly configurable selection algorithm because of its use of two variables: tournament size k and selection chance p. To make a selection it chooses k individuals from the population at random. It then sorts these into fitness order. It then selects the fittest individual with probability p. If this selection does not occur, then it selects second fittest with probability $p * (1 - p)$. If this further selection did not take place then we continue down the tournament and select the ith fittest individual with probability $p * (1 - p)^{i-1}$. If no individual is selected from the tournament, we restart the selection process. The configurability gained by varying k and p makes it easy to change the selection pressure, which is important in the overall running of a GP system.

3.4.2 Fitness Assessment

For program synthesis software, fitness functions are usually based on a collection of test cases, each consisting of an input to the system and the expected output from the program. This is no different for quantum programs. Each problem is defined by several test cases, where inputs are defined by some sequence of qubits in either the $|0\rangle$ or $|1\rangle$ state. Desired outputs are then given as the expected probability distribution that someone would observe if they were to repeatedly measure the output.

Before fitness assessment can take place in our system, we must first extract a circuit from our open graphs, and then simulate it this circuit on the test cases. If an individual cannot be extracted to a circuit, then we allocate the lowest possible fitness value for each fitness metric.

Fitness functions from previous works have used the test case information in a variety of ways to make a numerical measure of fitness. We will be combining some of the most common approaches to result in a tuple of fitness values, which are then compared lexicographically. The first two metrics in our fitness tuple are based on the concept of Mean Squared Fidelity (MSF) [2]. To compute this we work out the Bhattacharyya coefficient [7] between the expected output probability distribution and the actual probability distribution given by the candidate for each test case. If this coefficent is greater than 0.51 for a test case, we add one to a count of *passed_tests*, which is the fist component of our fitness tuple. This method was inspired by Spector et al. (1999) [46], who proposed that works should reward each test cases being passed in a way that was better than random, over any linear measure of fitness on a single test case. This deters the algorithm from finding easy ways to get a lot of fitness without solving the problem, i.e. by putting the system into a superposition of all states or just solving the easy test cases with high accuracy but very low accuracy for other more difficult test cases. The *passed_tests* fitness component discourages such behaviours and helps the genetic algorithm avoid local maxima.

Next we take the square of each of these calculated Bhattacharyya coefficients and sum them together, before dividing by the total number of test cases to get an average of the coefficients. This gives us the MSF which is used as the second component of our fitness tuple. This is a floating point number that represents the distance between the candidate and target solution, standardised between 0 and 1. Values of 1 mean that the expected output is observed across all test cases. It is this component

of fitness that we define success on. As we will be evolving programs on quite small numbers of qubits, its quite easy for individuals to score the maximum value in the *passed_tests* metric, so this is unsuitable for classing a candidate as successful. Instead we define success to be individuals with MSF ≥ 0.98, which is a level of accuracy used in prior works [2].

One component of fitness that has featured in some works, but we believe is not studied as often as it should be, is some measure of the cost of running a quantum program. As we are evolving static circuits that cannot change the number of qubits they operate on, we use the gate count of the circuits extracted from our individuals to approximate the cost of running the candidate solutions on a real quantum device. This forms the third part of our fitness metric. We ask our system to minimise this fitness component, but as this is the last metric it is only taken into account when the other two metrics are the same.

Works in the area of approximate program synthesis present an opportunity to explore a trade- off between approximating a desired circuit and reducing the cost of the circuit. Our system however only be optimises the size of individuals when it is at no cost to the accuracy. We do this because comparisons between our work and related works will primarily be on the accuracy metrics, as related works do not include detailed circuit size information. Yet, we still believe our system presents some opportunity to the user to have a trade-off between accuracy and performance. During the running of our system we observed that after successful evolving a candidate with MSF ≥ 0.98 the system would go on to increase accuracy to be closer to 1, and as it did the gate count of solutions would increase. We therefore started to record the gate count of any candidate with MSF ≥ 0.98, which would often be lots less than the best performing candidate. A system user could consider both these smaller individuals closer to MSF $= 0.98$ and see how their gate counts compare to the best performing individuals closer to MSF $= 1$, and in this way our system offers the user a trade-off between accuracy and size, without ever prioritising the optimisation of size over accuracy. In Section 5.1.3 we see this data collected and discussed.

If the user of a program synthesis system knew exactly the trade-off between accuracy and size they wanted to make, then they could instead have a fitness component that was a linear component of MSF and gate-count directly in line with the trade-off they wanted to make.

3.4.3 Elitism

Elitism is the process of automatically selecting the best performing individuals when creating the next generation, rather than leaving it to randomised selection methods [5]. It is usually configured as a percentage of the next generation the system will select using elitism. This process useful when the semantics of individuals are very sensitive to changes during mutation or crossover, as these changes can quickly destroy useful evolved solutions before selection pressure manages to propagate their advantageous genetic material throughout the system. Elitism means that if a breakthrough mutation occurs that leads to some individual with a fitness higher than all others there is no chance of the individual being randomly lost by selection process. Elitism was used by all versions of our system.

Chapter 4

Implementation

This chapter outlines our Python implementation of the designed GP system. It briefly justifies the use of Python as the programming language for the system, describes the key software components and gives an overview of some of the main data-structures.

4.1 Implementation Language

We used Python to implement the system because this gave the best access to a key library: PyZX [27]. PyZX implements some useful methods from literature, which saved time during this project. The key methods used are *extract_circuit* [18, 4] and *full_reduce* [28]. This library was also useful for testing and prototyping the system by enabling the us to easily visualise ZX-diagrams embedded in Jupyter notebooks.

Python also provides access to many libraries commonly used for research software, such as Numpy [36], which gives useful functions over large collections of multi-dimensional data. such methods were used for linear algebra in the fitness function, and data-processing for attaining figures for results.

4.2 Genetic Algorithm

There were existing libraries that implemented genetic algorithms in Python. The decision was made not to use any of these libraries because they did not contain the features that enabled the recording of the data we needed to evaluate of our system.

It was also best to implement our own genetic algorithm suite, to avoid a bloat of features we did not need and slow the system down.

An interface was created such that all the interchangeable components of the system could be swapped in and out to enable different versions of the system to be quickly tested. These are components such as the genetic operators, fitness function, and test problem. The system was also implemented such that all hyper-parameters were configurable at the top level of the interface, to enable the quick testing of different parameter settings.

4.3 Data Structures

The PyZX library gives us some of the key data structures already implemented. Namely, a representation of quantum circuits, ZX-diagrams and functions for converting between the two. While open graphs can be represented using PyZX's ZX diagram class, we created our own version that was more optimised to just represent open graphs. This simpler data structure was smaller and easier to implement mutations over. Functions to convert between this open graph data structure and the PyZX ZX-diagram class were made so we could use the methods from literature already implemented in PyZX.

We chose to represent open-graphs using four smaller data-structures.

1. An adjacency matrix. This is triangular due to the bi-directional nature of ZX-diagrams. Nodes are identified by the index they are stored at in the adjacency matrix.

2. A list of node-ids for the nodes that are regarded as inputs to the diagram

3. A list of node-ids for the nodes that are regarded as outputs to the diagram

4. An array, indexed by node-id, for the phases of each node. These are stored as a floating point number or fraction which is interpreted as a multiple of π.

An alternative approach for representing an open-graph could be linked-object based, with node objects storing all associated data for said node, such as its connections and phase. This object-based structure would require traversal, such as an breadth-first-search, just to enquire about the connectivity or phase of a node. This is why we have selected an array-based approach, where we can access connectivity and phase data with a constant time lookup.

Chapter 5

Results

This chapter is split into three sections. The first goes through the main results of the project, which pertain to the central research aim of the project; whether ZX-diagrams are an effective internal representation in evolutionary computation systems for the synthesis of quantum programs. We evaluate the ability of the open graph representation and mutations defined over open graphs to solve a variety of benchmark problems, and their ability to produce solutions that are small in terms of their gate count. The following two sections then deal with other research questions raised within Chapter 3. The first of these sections evaluates evaluates our system when the phases of all nodes are restricted to the form $k\pi/4$ for $k \in \mathbb{Z}$ to make better use of the Full Reduce mutation. Finally, we evaluate ways of mitigating unsafe genetic operators. Each of the three sections in this chapter describes our methodology for collecting results, presents the relevant data collected and discusses this data to reach conclusions about the systems we are assessing.

5.1 Open Graph Redundancy and Mutations

This section aims to investigate whether ZX-diagrams are an effective internal representation in evolutionary computation systems for the synthesis of quantum programs. This is primarily achieved through a comparison between our created systems, *pure_zx* and *pure_cm*, as well as a third that used the circuit model from related work. Both of our systems use the open graph internal representation and hence may be advantaged by the reduced redundancy this gives compared to the circuit model. But additionally, performance of *pure_zx* could be due to the mutations we define over the ZX-calculus rather than the open graph representations natural reduced redundancy. So we can investigate which mutations perform best,

those defined over the circuit model or the ZX-calculus, with a comparison between *pure_zx* and *pure_cm*.

5.1.1 Evaluation Methodology

Rigorous analysis of quantum program synthesis systems consists of repeatedly running them on a collection of test problems, with different random seeds for each run. Data like the best fitness achieved, or the number of generations it took to each run to achieve a pre-defined level of fitness, are recorded. However, up until recently, there has been little consensus on the specifics of this evaluation process. Previous works vary in the data they collect, how they are collect it, and most importantly the test problems they use. This means many related works are hard to compare to each-other. The work of Atkinson et al. (2019) [2] is the first to propose a set of test problems to unify the bench-marking of works in this area. The work justifies a list of proposed test problems for which they say all future systems should also be evaluated on to make related works more comparable. Not only this, but the results said work goes on to give are some of the most comprehensive in terms of the amount of data reported on and in terms of giving an appropriate level of detail in how the results were collected. Our project will be using the proposed benchmark problems, as well as providing the same evaluation data that the work of Atkinson et al. provides. This is not only to be able to directly compare to said work, but also in the hope that it makes this body of work more comparable to future research efforts if this becomes the standard for quantum program synthesis evaluation.

The bench-marking problems that Atkinson et al. propose are: the Quantum Fourier Transform (QFT) [13] on 2, 3 and 4 qubits; the Grover's Diffusion Operator (GDO) [24] on 2, 3 and 4 qubits; Bell Pair (BP) generator [35] and the Toffoli (TOF) gate [50]. These problems were chosen as they are regularly occurring in literature both on their own and as components in larger circuits.

The primary metric for comparison in the work of Atkinson et al. is the success rate for solving a problem. They run a system 100 times and allow each run a maximum of 1,000,000 quantum program evaluations[i], recording how many runs produce an individual with MSF ≥ 0.98. As MSF is a component of our lexicographic fitness it will be easy to reproduce this metric for our system.

[i]For GP systems, a quantum program is evaluated every time the fitness function is called. So evaluations = generations × population size.

Another set of important metrics they use is based on the best preforming individual in each of the 100 runs. They report the median best fitness and the maximum best fitness across all runs. This can aid understanding in how close a system is getting to solving a problem, and hence help compare systems when there is a low success rate. Additionally the interquartile range (IQR) of best fitnesses is given, which indicates the consistency or variance of the best fitnesses attained by a system.

A slight deficiency in data given by the work of Atkinson et al. is a statistic that could help compare systems when both are achieving a high success rate for a problem. We propose the inclusion of the median evaluations until success statistic for this purpose. For example it could be the case that although systems are given 1,000,000 evaluations to solve a problem, they achieve the success threshold in the first 10,000. Systems could record the first time they saw an individual that was successful, according to the definition of success in the success rate metric. The median value for this across all runs could then be reported. For problems which a system is solving very commonly, knowing how quickly the system is solving it, on average, will allow a more detailed comparison. In our system we have given the median generations until success (Gens) data which can easily be converted to the amount of evaluations until success by multiplying by our population size of 1000.

We also propose the inclusion of another important piece of data that is often overlooked in previous works. The potential for evolutionary computation methods to perform multi-objective optimisation has been studied [29]. In the area of quantum program synthesis, multi-objective fitness functions lend themselves not only to solving problems but solving them efficiently, i.e with as few gates as possible. The ZX-calculus has the potential to be very adept at producing solutions with reduced size, something we have tried to explore by adding a gate-count component to our fitness metric. Our results section will include data on the gate counts of evolved solutions to each problem, for both of our trialled systems. We will take the best performing individual from each 100 run and report on the median gate count of these individuals. To investigate the trade-off between performance and gate-count, we also have our the systems report on the lowest gate-count for any individual within the defined success range of MSF ≥ 0.98 for each run. A median of this figure is given as well as the lowest gate count seen within MSF ≥ 0.98 across all runs.

The last improvement we propose for the evaluation methodology is to add a randomised benchmark problem. In this work we have used the work of Mezzadri (2006)

[32] to generate random unitary matrices to represent target 2-qubit (RND-2) and 3-qubit (RND-3) circuits. While Atkinson et al. propose a varied set of test problems, it may still be possible for related works to in some way optimise their system specifically towards the problems in this test set. It is much more of a challenge to optimise towards randomly generated unitaries. A disadvantage of these test problems is that they are likely to have an increased variance due to different generated unitaries being harder or easier to evolve than others. This could be counteracted by performing more repeat tests for these test cases.

All systems were evaluated using the hyper-parameters seen in Table 5.1. Hyper-parameters were attained through trial and error as well as being influenced by related works. More comprehensive approaches to optimisation such as Grid Search were not possible because of the number of parameters and the time it takes to comprehensibly evaluate a setting of parameters.

Parameter	Value
Crossover Rate	0.5
Mutation rate	0.7
Tournament k	9
Tournament p	0.6
Elitism	0.02

Table 5.1: Hyper-parameters used for our systems.

5.1.2 Evaluation Data

Evaluation data for the *pure_zx* system being run on the bench-marking problems can be seen in Table 5.2. All data points are the result of 100 repeat tests. All runs were allowed to run for 1,000,000 circuit simulations. As populations of size 1000 were used in all tests, this means a maximum of 1000 generations were allowed to run. All values are given to 3 significant figures. The best success rates across the *pure_zx*, *pure_cm* and Atkinson et al. (2019) [2] systems are highlighted in bold for each problem.

Evaluation data for the *pure_cm* system being run on the bench-marking problems can be seen in Table 5.3. All data points are the result of 100 repeat tests. All runs were allowed to run for 1,000,000 circuit simulations. As populations of size 1000 were used in all tests, this means a maximum of 1000 generations were allowed to run. All values are given to 3 significant figures.

Problem	Success Rate	Best Fitness Per Run			Gens
		Median	Best	IQR	
BP	**100%**	1.00	1.00	0.00	2
TOF	58%	1.00	1.00	0.00	182
QFT-2	**100%**	1.00	1.00	0.00	8
QFT-3	93%	1.00	1.00	0.00	111
QFT-4	15%	0.941	0.994	0.06	681
GDO-2	**100%**	1.00	1.00	0.00	79
GDO-3	**0%**	0.750	0.875	0.00	N/A
GDO-4	**0%**	0.900	0.900	0.00	N/A
RND-2	**100%**	1.00	1.00	0.00	31
RND-3	0%	0.919	0.944	0.01	N/A

Table 5.2: ***pure_zx*** system results. 100 runs for each problem, each with 1,000,000 circuit evaluations. Success rate of runs with MSF ≥ 0.98. Best individual taken from each run and median, best and interquartile range (IQR) of their fitnesses. Median number of generations until success (Gens) is first reached in successful runs.

Problem	Success Rate	Best Fitness Per Run			Gens
		Median	Best	IQR	
BP	**100%**	1.00	1.00	0.00	2
TOF	**78%**	1.00	1.00	0.00	371
QFT-2	**100%**	1.00	1.00	0.00	14
QFT-3	**95%**	1.00	1.00	0.00	76
QFT-4	**68%**	0.993	0.996	0.00	450
GDO-2	**100%**	1.00	1.00	0.00	2
GDO-3	**0%**	0.750	0.875	0.00	N/A
GDO-4	**0%**	0.900	0.900	0.00	N/A
RND-2	**100%**	1.00	1.00	0.00	13
RND-3	**59%**	0.985	0.999	0.00	464

Table 5.3: ***pure_cm*** system results. 100 runs for each problem, each with 1,000,000 circuit evaluations. Success rate of runs with MSF ≥ 0.98. Best individual taken from each run and median, best and interquartile range (IQR) of their fitnesses. Median number of generations until success is first reached in successful runs (Gens).

Table 5.4 shows the results from Atkinson et al. (2019) [2]. Data given to as many significant figures as were available. All data points are the result of 100 repeat tests.

Problem	Success Rate	Best Fitness Per Run		
		Median	Best	IQR
BP	**100%**	1.00	1.00	0.00
TOF	9%	1.00	1.00	0.06
QFT-2	**100%**	1.00	1.00	0.00
QFT-3	68%	1.00	1.00	0.06
QFT-4	2%	0.84	0.98	0.07
GDO-2	**100%**	1.00	1.00	0.00
GDO-3	**0%**	0.67	0.85	0.06
GDO-4	**0%**	0.90	0.90	0.00

Table 5.4: Data as given in Atkinson et al. (2019) [2]. 100 runs for each problem, each with 1,000,000 circuit evaluations. Success rate of runs with MSF ≥ 0.98. Best individual taken from each run and median, best and interquartile range (IQR) of their fitnesses.

During the same experiments that was used to collect data for Tables 5.3 and 5.2, we recorded data on the size of the individuals being created. Table 5.5 shows these results for the *pure_zx* system while Table 5.6 shows these for the *pure_cm* system. The best minimum gate counts within MSF ≥ 0.98 across both systems are highlighted in bold. The same data for the Atkinson system is not available.

Problem	Best MSF Per Run Median Gate Num	Smallest Successful Per Run Median Gate Num	Smallest Successful Per Run Minimum Gate Num
BP	5	4	**4**
TOF	34	30	26
QFT-2	23	17	14
QFT-3	38	30	26
QFT-4	126	85	74
GDO-2	9	6	**6**
GDO-3	N/A	N/A	N/A
GDO-4	N/A	N/A	N/A
RND-2	28	15	**7**
RND-3	N/A	N/A	N/A

Table 5.5: ***pure_zx*** system solution size results. 100 runs for each problem, each with 1,000,000 circuit evaluations. Best performing individual taken from each run, median gate count of these is given. Lowest gate count individual with MSF ≥ 0.98 taken for each run and the median of these is given. Minimum gate count is the smallest result seen across all runs that has MSF ≥ 0.98.

Problem	Best MSF Per Run Median Gate Num	Smallest Successful Per Run Median Gate Num	Smallest Successful Per Run Minimum Gate Num
BP	5	4	**4**
TOF	34	25	**23**
QFT-2	22	15	**13**
QFT-3	52	26	**23**
QFT-4	117	76	**68**
GDO-2	9	6	**6**
GDO-3	N/A	N/A	N/A
GDO-4	N/A	N/A	N/A
RND-2	60	17	12
RND-3	144	118	**87**

Table 5.6: ***pure_cm*** system solution size results. 100 runs for each problem, each with 1,000,000 circuit evaluations. Best performing individual taken for each run, median gate count of these given. Lowest gate count individual with MSF ≥ 0.98 taken for each run and the median of these is given. Minimum gate count is the smallest result seen across all runs that has MSF ≥ 0.98.

5.1.3 Results Discussion

The results show that both systems produced in this work using the ZX-calculus internal representation out-perform the previous work of Atkinson et al. (2019) [2]. This can primarily be seen by an increase in the success rate for some of the problems that the Atkinson system struggled to solve, with our systems seeing a success rate increase of at-least 49% for the Toffoli problem, and 25% for the QFT-3 problem. The variance of our systems were also lower for these problems, both having an IQR of 0 compared to 0.06 for the Atkinson system. This implies our systems were much more consistent in their operation.

An example of a problem that our systems solved better than previous work is the QFT-4 problem. Our *pure_cm* system solved the problem 68% of the time while the Atkinson system solved it 2% of the time. This is a drastic increase, where you would expect our *pure_cm* system to solve the problem given a single run but it would take many runs before expecting the Atkinson system to find a solution.

While the set of genetic operators the *pure_zx* and *pure_cm* systems use differs entirely from each-other, what they do have in common is the use of the open graph internal representation. Because of the spider-fusion needed to get quantum circuits into this representation, open graphs have decreased redundancy compared to the circuit model, and this seems to have helped both our systems explore their solution space quicker and more reliably than the previous work across all test problems.

Despite the observed success rate increase across the QFT-3, QFT-4 and TOF problems, our systems did not solve the GDO-3 and GDO-4 problems. However the *pure_cm* system did get closer to solving GDO-3 than the previous work, as can be seen by increased the median and best fitness attained.

To further investigate the validity of the ZX-calculus as the basis for internal representations in evolutionary computation systems, we have experimented with the effectiveness of the mutations based on an open graph compared to mutations based on a circuit. Both the success rate and median generations until success metrics across multiple test problems support the conclusion that the mutations based on the circuit model are more effective. There was no problem where *pure_zx* outperformed *pure_cm*, although the difference between their success rates was smaller than the difference observed between our systems and the prior work. In particular, we can highlight the RND-3 test problem, which had systems trying to evolve solutions to randomly generated unitary matrices. The *pure_cm* system managed to solve this problem 59% of the time while the *pure_zx* system failed to solve it. This

disparity between the two systems on the randomised test problem is the greatest out of all test problems, and this helps justify our proposition for it to be added to the benchmark problem suite for similar systems. We also did not observe a large increase in IQR for the randomised test problems compared to other test problems, also helping justify that the tests are not too random in that the unitaries they produce do not differ too greatly in how easy they are to solve.

Not only was the *pure_cm* system able to evolve solutions to the test problems quicker and more reliably, but the solutions it created were smaller too. On average across all test problems that both systems solved, the smallest solution created by the *pure_cm* that is still within the desired MSF fitness range, was a gate smaller than those in the *pure_zx* system. The *pure_cm* achieved the smallest valid solution on all problems, with the exception of RND-2. Overall, the sizes of the solutions produced were unimpressive, with hand-written code being smaller in all cases. However, one positive in the size data is the difference between the sizes of the absolute best performing individuals and the individuals that are just somewhere within MSF ≥ 0.98. For example, on the QFT-3 problem when solved by *pure_cm* the median gate count for the best performing individual of each run was 52, but if we allow any individual within MSF ≥ 0.98 the median gate count is 26, a 50% reduction in size that may well be worth the trade-off for a maximum of 0.02 MSF.

5.2 Open Graphs Limited to $k\pi/4$ Phases

In Section 2.2.1 we introduced the work of Kissinger and Wetering (2020) [28], which describes a scheme for reducing the size of ZX-diagrams, with a focus on reducing the count of gates that have phases which are not in the form $k\pi/2$ for $k \in \mathbb{Z}$. A mutation that performs this procedure, *Full Reduce*, was included in our system with the goal of further reducing down the size of the open graphs of our candidate solutions in order to have the extract to smaller circuits and increase the semantic efficiency of our approach. As we have several mutations that add floating point multiples of π, it is unlikely the system is using much of some of the methods in the Full Reduce mutation. For this reason we have performed an experiment to see how the system would perform differently if we limited all phases to the form $k\pi/4$, and allow the Full Reduce to perform more transformations on our candidate solutions.

We define the $k\pi/4$ system as being the *pure_zx* system, with the same hyper parameters seen in Table X. The difference is that the Add Node and Phase change

mutations are changed such that they can only introduce phases that are in the form $k\pi/4$, for $k \in \mathbb{Z}$, into the system. As the initial population generation method converted Clifford-T circuits into open graphs (see Section X) this also only introduces phases of the form $k\pi/4$ into the system.

Note that the $k\pi/4$ system could be based on the better performing *pure_cm* system, but as Full Reduce was not used in the original *pure_cm* system this could lead to misleading results. We would not know if differences between the $k\pi/4$ system and *pure_cm* were down to the phase limitation or the inclusion of Full Reduce in the *pure_cm* mutation set.

5.2.1 Evaluation Methodology

We are using the same evaluation methodology as discussed in Section 5.1.1. However, due to time constraints and the computational resources accessible during this project we were unable to evaluate another system across all benchmark problems with 100 repeat tests each. Therefore we chose to evaluate the $k\pi/4$ system on the TOF and QFT-3 benchmark problems, with 100 repeat tests. These problem were selected the original *pure_zx* and *pure_cm* systems solved them commonly but not all the time. There is also room to improve on the size of the solutions produced.

5.2.2 Evaluation Data

Tables 5.7 and 5.8 display the data from evaluating the $k\pi/4$ system as well as the previously seen data from the *pure_zx* system on the same problems. The best success rate and minimum gate count within MSF ≥ 0.98 are highlighted in bold for each problem.

System	Problem	Success Rate	Best Fitness Per Run			Gens
			Median	Best	IQR	
pure_zx	TOF	**58%**	1.00	1.00	0.00	182
pure_zx	QFT-3	**92%**	1.00	1.00	0.00	111
$k\pi/4$	TOF	12%	0.750	1.00	0.00	545
$k\pi/4$	QFT-3	52%	0.995	1.00	0.01	260

Table 5.7: ***pure_zx*** system results. 100 runs for each problem, each with 1,000,000 circuit evaluations. Success rate of runs with MSF ≥ 0.98. Best individual taken from each run and median, best and interquartile range (IQR) of their fitnesses. Median number of generations until success (Gens) is first reached in successful runs.

System	Problem	Best MSF Per Run Median Gate Num	Smallest Successful Per Run Median Gate Num	Smallest Successful Per Run Minimum Gate Num
pure_zx	TOF	34	30	**26**
pure_zx	QFT-3	38	30	26
$k\pi/4$	TOF	36	30	27
$k\pi/4$	QFT-3	35	29	**22**

Table 5.8: $k\pi/4$ phase restricted system results. 100 runs for each problem, each with 1,000,000 circuit evaluations. Best performing individual taken from each run, median gate count of these is given. Lowest gate count individual with MSF ≥ 0.98 taken for each run and the median of these is given. Minimum gate count is the smallest result seen across all runs that has MSF ≥ 0.98.

5.2.3 Results Discussion

We can see what the $k\pi/4$ system is less good at solving the test problems than the *pure_zx* system as of the lower success rates and median best fitnesses. This was expected due to the limitation we placed on the mutations the system could use.

The results regarding the size of the solutions produced are mixed, with the $k\pi/4$ system unable to beat *pure_zx* on the TOF problem but achieving a small optimisation over *pure_zx* on QFT-3. This could indicate that the $k\pi/4$ system is indeed better able to optimise the size of individuals, but is being held back by its poor ability to traverse semantic space. For example, if we consider that the *pure_zx* solved TOF 58 times, compared to the 12 times $k\pi/4$ solved it, that's 46 more opportunities *pure_zx* had to find smaller solutions compared to the $k\pi/4$ system.

5.3 Unsafe Mutation Mitigation

In Section 3.2.1 we proposed four methods for mitigating against the individuals with no gFlow that are added to the system via unsafe mutations. While we explained the decision to use the *discarding* method for most of this project, we would like to do a small investigation to justify this choice and to assess the other methods.

Note that the *avoidance* method has not been assessed, as no successful *avoidance* system could be created for this project. There needs to be more work to look in to mutations that traverse semantic safe efficiently but are also safe to perform. One could class the *pure_cm* system as being an *avoidance* system, as it only uses safe mutations, but the original intent behind the proposed methods was to allow mutations defined over the ZX-calculus to be used rather than to revert to safe methods defined on the circuit model.

The *non-selection* system is a version of the *pure_zx*, where all the mutations will not discard their results if they produce an individual with no gFlow. The selection algorithm has been modified such that it never selects individuals with no gFlow.

The *acceptance* system is a version of the *pure_zx* that has no mitigation against unsafe individuals. It has the potential to traverse through the space of non-extractable diagrams to get to other parts of extractable space.

5.3.1 Evaluation Methodology

We are using the same evaluation methodology as discussed in Section 5.1.1. Due to time constraints and the computational resources accessible during this project we were unable to evaluate more systems across all benchmark problems with 100 repeat tests each. We chose to evaluate the $k\pi/4$ system on the TOF and QFT-3 benchmark problems, with 100 repeat tests. These problems were selected because the original *pure_zx* and *pure_cm* systems solved them commonly but not all the time. We have not included evaluation of the size of produced results, because primarily we want to evaluate the ability of these systems to solve the problems, and because there were no notable conclusions to draw from this data.

5.3.2 Evaluation Data

Table 5.9 displays the data from evaluating the *non-selection* system and *acceptance* system as well as the previously seen data from the *pure_zx* system that uses the

discarding method on the same problem. The best success rate is highlighted in bold for each problem.

System	Problem	Success Rate	Best Fitness Per Run			Gens
			Median	Best	IQR	
pure_zx	TOF	**58%**	1.00	1.00	0.00	182
pure_zx	QFT-3	**92%**	1.00	1.00	0.00	111
non-selection	TOF	53%	1.00	1.00	0.00	212
non-selection	QFT-3	85%	1.00	1.00	0.00	140
acceptance	TOF	25%	0.750	1.00	0.00	413
acceptance	QFT-3	34%	0.966	1.00	0.00	360

Table 5.9: Unsafe mitigation systems results. 100 runs for each problem, each with 1,000,000 circuit evaluations. Success rate of runs with MSF ≥ 0.98. Best individual taken from each run and median, best and interquartile range (IQR) of their fitnesses. Median number of generations until success (Gens) is first reached in successful runs.

5.3.3 Results Discussion

We can see that the *non-selection* method is performing slightly worse than *pure_zx*, seeing an average of a 6% success rate difference in favour of *pure_zx*. These approaches are likely to be even more similar than these results suggest, but our evaluation methodology, which was chosen to be comparable to previous works, disadvantages the *non-selection* system by scoring performance based on the number of fitness evaluations rather than run-time. It is likely this system would be more comparable to *pure_zx* in terms of run-time for the reasons discussed in Section 3.2.1. The closeness between the two *discarding* and *non-selection* approaches suggests that *non-selection* is a valid mitigation technique, and that more research should go in to comparing these two similar approaches.

The *acceptance* method performs much worse than *pure_zx*, with lower success rates, lower median fitnesses and higher number of generations until success on both test problems. This confirms our hypothesis that the system would be unable to use unsafe operations to its advantage, to better traverse the fitness landscape by using the non-extractable space it now has access to compared to the other systems.

Chapter 6

Conclusions

This work has successfully created a system that uses genetic programming to synthesise quantum programs. It is the first to use an internal representation based on the ZX-calculus. The work has shown that the advantage of using such a representation is the reduced redundancy it can offer compared to the quantum circuit model which related works use [31, 47, 2]. The best version of our system solves the QFT-3, QFT-4 and Toffoli benchmark problems at a success rate an average of 54% higher than the work of Atkinson et al (2019) [2].

While the semantic efficiency of our ZX based internal representation was useful, using mutations defined over the circuit model was more successful than using our novel mutations that operate on ZX-diagrams. The system that used circuit based mutations was always more successful than the system that used ZX based mutations, with higher (or equal) success rates and median best fitnesses on all problems. One of the benchmark problems was unsolved by the ZX-mutation based system while the circuit model mutation based system solved it 59% of the time.

Up until recently there was little standardisation in how systems similar to ours were evaluated and compared. The work of Atkinson et al. (2019) [2] changes this by proposing a set of benchmark problems and metrics to measure when getting systems to evolve solutions to the problems. We have adhered to this benchmark problem set as well as proposing improvements to the method to further improve how works are evaluated and compared. One such contribution was the inclusion experiments to try and create randomly generated unitary matrices. This proved to be a worthy inclusion as it was the most distinguishing problem for the comparison of two of our systems.

Another addition to the data collected for evaluation was the inclusion of gate counts of the evolved programs, so the trade-off between evolving accurate solutions and small solutions can be explored. Our system managed to exhibit a large difference in gate count between individuals with the best accuracy and individuals with lower accuracy but still within a defined success bound, with the difference between the median gate counts showing the lower accuracy solutions were often 50% the size of the best fitness individuals. This shows the potential of GP systems to explore the gate count verses accuracy trade-off that may be important to near-term quantum devices [39].

A weakness of our work has come from the Python-based implementation. While using Python and the PyZX [27] library gave us access to already implemented methods from literature, these technologies were not optimised for quickly simulating our candidate quantum circuits. Spending more time on implementing a faster solution in another language or integrating a faster simulation system into our existing system might have benefit us in the long term of the project. This is because some of the runs of the system took several hours on the machines we had available. Considering hundreds of repeat tests are needed to be run for system evaluation, this limited our ability to evaluate more systems across the whole benchmark suite and meant we were less able to experiment with different settings of hyper-parameters. Further optimisation of the hyper-parameters could have increased performance for our systems.

6.1 Future Work

We believe this work has been a successful first attempt at using ZX based internal representations for GP systems, but throughout the project have found many areas that warrant further exploration.

One of the secondary objectives of this project was to propose and evaluate methods for dealing with the unsafe mutations that can be defined over the ZX-calculus. We used the *discarding* method throughout the work, with a small investigation into the other proposed methods. These evaluations suggested that the individuals with no gFlow served no benefit to our system, and hence we believe that a system that uses only safe mutations would be best. Future works could look for effective mutations that are also safe in the fact that they preserve gFlow. Up until now, works on the

ZX-calculus looking at preserving gFlow have also been concerned with preserving semantics for the purpose of creating transformations that help reduce the size of a quantum circuit and keeping it extractable. More work could go into to looking at gFlow preserving operations that also better traverse the semantic space of quantum programs. We believe there is still potential for mutations defined over the ZX-calculus to be very useful, despite them performing worse than the circuit based mutations in this project.

Our system shows that GP systems can be used to allow users a trade-off between accuracy or size in their evolved solutions. However, our system always prioritised accuracy over size when evolving its solutions as we used a lexicographic fitness metric where gate count was the last element. Future works could use a linear combination of fitness metrics that would be informed by the accuracy to gate count trade-off that would actually be of benefit to quantum devices. Different hardware will be able to run different sizes of program at with different accruacy, knowledge of this could help inform the way we linearly combine the fitness and gate count metrics. Further to this, gate count is a primitive measure of the cost of a quantum program, and in future we propose more informed metrics that could be device specific be used to represent the real cost of running a given quantum program on a quantum device.

This work used the Full Reduce [28] operation as a SND mutation to attempt to allow the system to optimise the size of its candidate solutions. Our results using this mutation were mixed, as when we limited the phases introduced into our candidates we noticed a considerable loss of accuracy for our system, and only limited improvements to gate counts. There is a wealth literature on semantic-preserving ZX-diagram reducing transformations, including the components that make up the Full Reduce operation. Experiments that use these as mutations could get more success than just the Full Reduce on its own.

Bibliography

[1] Frank Arute, Kunal Arya, Ryan Babbush, Dave Bacon, Joseph C Bardin, Rami Barends, Rupak Biswas, Sergio Boixo, Fernando GSL Brandao, David A Buell, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574(7779):505–510, 2019.

[2] Timothy Atkinson, Athena Karsa, John Drake, and Jerry Swan. Quantum program synthesis: Swarm algorithms and benchmarks. In *European Conference on Genetic Programming*, pages 19–34. Springer, 2019.

[3] Timothy Atkinson, Detlef Plump, and Susan Stepney. Evolving graphs with semantic neutral drift. *Natural Computing*, pages 1–17, 2019.

[4] Miriam Backens, Hector Miller-Bakewell, Giovanni de Felice, and John van de Wetering. There and back again: A circuit extraction tale. *arXiv preprint arXiv:2003.01664*, 2020.

[5] Shumeet Baluja and Rich Caruana. Removing the genetics from the standard genetic algorithm. In *Machine Learning Proceedings 1995*, pages 38–46. Elsevier, 1995.

[6] Kenton M. Barnes and Michael B. Gale. Meta-genetic programming for static quantum circuits. In *Proceedings of the Genetic and Evolutionary Computation Conference Companion*, GECCO '19, page 2016–2019. Association for Computing Machinery, 2019.

[7] Anil Bhattacharyya. On a measure of divergence between two multinomial populations. *Sankhyā: the indian journal of statistics*, pages 401–406, 1946.

[8] Markus F Brameier and Wolfgang Banzhaf. *Linear genetic programming*. Springer Science & Business Media, 2007.

[9] Daniel E Browne, Elham Kashefi, Mehdi Mhalla, and Simon Perdrix. Generalized flow and determinism in measurement-based quantum computation. *New Journal of Physics*, 9(8):250, 2007.

[10] Earl T Campbell, Barbara M Terhal, and Christophe Vuillot. Roads towards fault-tolerant universal quantum computation. *Nature*, 549(7671):172–179, 2017.

[11] Bob Coecke and Ross Duncan. Interacting quantum observables. In *Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP)*, Lecture Notes in Computer Science, 2008.

[12] Bob Coecke and Aleks Kissinger. Picturing quantum processes: A first course in quantum theory and diagrammatic reasoning. In *Picturing Quantum Processes: A First Course in Quantum Theory and Diagrammatic Reasoning*. Cambridge University Press, 2017.

[13] Don Coppersmith. An approximate fourier transform useful in quantum factoring. *arXiv preprint quant-ph/0201067*, 2002.

[14] Niel de Beaudrap, Xiaoning Bian, and Quanlong Wang. Techniques to reduce $\pi/4$-parity phase circuits, motivated by the zx calculus. *arXiv preprint arXiv:1911.09039*, 2019.

[15] Niel de Beaudrap, Xiaoning Bian, and Quanlong Wang. Fast and effective techniques for t-count reduction via spider nest identities. *arXiv preprint arXiv:2004.05164*, 2020.

[16] David Deutsch. Quantum computation. *Physics World*, 5(6):57, 1992.

[17] Marco Dorigo, Mauro Birattari, and Thomas Stutzle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, 2006.

[18] Ross Duncan, Aleks Kissinger, Simon Perdrix, and John Van De Wetering. Graph-theoretic simplification of quantum circuits with the zx-calculus. *Quantum*, 4:279, 2020.

[19] Alex S Fraser. Simulation of genetic systems by automatic digital computers i. introduction. *Australian Journal of Biological Sciences*, 10(4):484–491, 1957.

[20] Edgar Galván-López, Riccardo Poli, Ahmed Kattan, Michael O'Neill, and Anthony Brabazon. Neutrality in evolutionary algorithms... what do we know? *Evolving Systems*, 2(3):145–163, 2011.

[21] David E Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *Foundations of genetic algorithms*, volume 1, pages 69–93. Elsevier, 1991.

[22] Daniel Gottesman. The heisenberg representation of quantum computers. *arXiv preprint quant-ph/9807006*, 1998.

[23] Jennifer Green, Jacqueline L Whalley, and Colin G Johnson. Automatic programming with ant colony optimization. In *Proceedings of the 2004 UK Workshop on Computational Intelligence*, pages 70–77. Loughborough University, 2004.

[24] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[25] Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. A complete axiomatisation of the zx-calculus for clifford+ t quantum mechanics. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 559–568, 2018.

[26] Emmanuel Jeandel, Simon Perdrix, and Renaud Vilmart. Diagrammatic reasoning beyond clifford+ t quantum mechanics. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 569–578, 2018.

[27] Aleks Kissinger and John van de Wetering. PyZX: Large Scale Automated Diagrammatic Reasoning. In Bob Coecke and Matthew Leifer, editors, Proceedings 16th International Conference on *Quantum Physics and Logic,* Chapman University, Orange, CA, USA., 10-14 June 2019, volume 318 of *Electronic Proceedings in Theoretical Computer Science*, pages 229–241. Open Publishing Association, 2020.

[28] Aleks Kissinger and John van de Wetering. Reducing the number of non-clifford gates in quantum circuits. *Phys. Rev. A*, 102:022406, Aug 2020.

[29] Abdullah Konak, David W Coit, and Alice E Smith. Multi-objective optimization using genetic algorithms: A tutorial. *Reliability Engineering & System Safety*, 91(9):992–1007, 2006.

[30] John R. Koza. Non-linear genetic algorithms for solving problems. United States Patent 4935877, 19 June 1990. filed may 20, 1988, issued june 19, 1990, 4,935,877. Australian patent 611,350 issued september 21, 1991. Canadian patent 1,311,561 issued december 15, 1992.

[31] Paul Massey, John A. Clark, and Susan Stepney. Human-competitive evolution of quantum computing artefacts by genetic programming. *Evolutionary Computation*, 14(1):21–40, 2006.

[32] Francesco Mezzadri. How to generate random matrices from the classical compact groups. *arXiv preprint math-ph/0609050*, 2006.

[33] Alberto Moraglio, Krzysztof Krawiec, and Colin G Johnson. Geometric semantic genetic programming. In *International Conference on Parallel Problem Solving from Nature*, pages 21–31. Springer, 2012.

[34] Kang Feng Ng and Quanlong Wang. A universal completion of the zx-calculus. *arXiv preprint arXiv:1706.09877*, 2017.

[35] Michael A Nielsen and Isaac Chuang. Quantum computation and quantum information, 2002.

[36] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[37] Tim Perkis. Stack-based genetic programming. volume 1, pages 148 – 153 vol.1, 07 1994.

[38] JM Pino, JM Dreiling, C Figgatt, JP Gaebler, SA Moses, CH Baldwin, M Foss-Feig, D Hayes, K Mayer, C Ryan-Anderson, et al. Demonstration of the qccd trapped-ion quantum computer architecture. *arXiv preprint arXiv:2003.01293*, 2020.

[39] John Preskill. Quantum computing in the nisq era and beyond. *Quantum*, 2:79, 2018.

[40] Google AI Quantum et al. Hartree-fock on a superconducting qubit quantum computer. *Science*, 369(6507):1084–1089, 2020.

[41] Olivier Roux and Cyril Fonlupt. Ant programming: Or how to use ants for automatic programming. In *Proceedings of ANTS*, volume 2000, pages 121–129. Springer Berlin, 2000.

[42] Ben IP Rubinstein. Evolving quantum circuits using genetic programming. In *Proceedings of the 2001 Congress on Evolutionary Computation (IEEE Cat. No. 01TH8546)*, volume 1, pages 144–151. IEEE, 2001.

[43] Sebastian Ruder. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747*, 2016.

[44] Peter W Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[45] Seyon Sivarajah, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. t— ket¿: A retargetable compiler for nisq devices. *Quantum Science and Technology*, 2020.

[46] Lee Spector, Howard Barnum, Herbert J Bernstein, and Nikhil Swamy. Quantum computing applications of genetic programming. *Advances in genetic programming*, 3:135–160, 1999.

[47] Lee Spector and Jon Klein. Machine invention of quantum computing circuits by means of genetic programming. *AI EDAM*, 22(3):275–283, 2008.

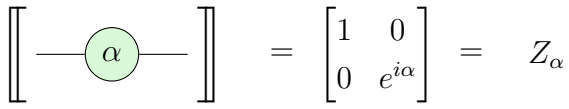
[48] Kilian Stoffel and Lee Spector. High-performance, parallel, stack-based genetic programming. *Genetic Programming*, pages 224–229, 1996.

[49] Krysta M Svore, Alan Geller, Matthias Troyer, John Azariah, Christopher Granade, Bettina Heim, Vadym Kliuchnikov, Mariia Mykhailova, Andres Paz, and Martin Roetteler. Q#: Enabling scalable quantum computing and development with a high-level domain-specific language. *arXiv preprint arXiv:1803.00652*, 2018.

[50] Tommaso Toffoli. Reversible computing. In *International Colloquium on Automata, Languages, and Programming*, pages 632–644. Springer, 1980.

Appendix A
Extended Gate Definitions and ZX Equivalences

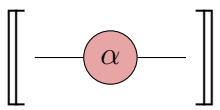
Arbitrary Phase Shift Gate



$$= \begin{bmatrix} 1 & 0 \\ 0 & e^{i\alpha} \end{bmatrix} = Z_\alpha$$

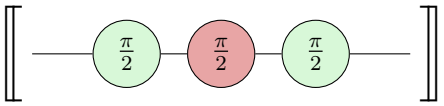
Common instances are $S = Z_{\pi/2}$ and $T = Z_{\pi/4}$

Arbitrary X Rotation



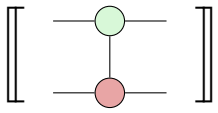
$$= \begin{bmatrix} cos(\frac{\alpha}{2}) & -i\ sin(\frac{\alpha}{2}) \\ -i\ sin(\frac{\alpha}{2}) & cos(\frac{\alpha}{2}) \end{bmatrix} = X_\alpha$$

Hadamard Gate



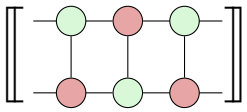
$$= \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = H$$

CNOT Gate



$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} = \text{CNOT}$$

SWAP Gate



$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \text{SWAP}$$