UTS UNIVERSITY OF TECHNOLOGY SYDNEY

A Toolchain for Cluster-State Architecture

by Gregory Michael BOWEN

Thesis submitted in fulfilment of the requirements for the degree of

Doctor of Philosophy

under the supervision of Associate Professor Simon Devitt

University of Technology Sydney
Faculty of Engineering and Information Technology

February 2025

Certificate of Original Authorship

I, Gregory Michael Bowen, declare that this thesis is submitted in fulfilment of the requirements for the award of Doctor of Philosophy in the Faculty of Engineering and Information Technology at the University of Technology Sydney.

This thesis is wholly my own work unless otherwise referenced or acknowledged. In addition, I certify that all information sources and literature used are indicated in the thesis.

This document has not been submitted for qualifications at any other academic institution.

This research is supported by the Australian Government Research Training Program (RTP) Scholarship doi.org/10.82133/C42F-K220.

signature: Production Note:
Signature removed prior to publication.

date: 28-FEB-2025

For Emma, Scarlett, Rafael.

φύσις κρύπτεσθαι φιλεῖ

Diels-Kranz, Ch. 22, Fr. 123

Acknowledgements

I thank,

- my supervisor, Simon Devitt, for his guidance and his willingness to punt;
- Chris Ferrie, for underwriting this research at its beginning;
- Michael Steel, for occasions of discussion and advice;
- Madhav Krishnan Vijayan, for his help and encouragement;
- Alan Robertson, for comment on early drafts of this dissertation.

Abstract

This dissertation is a demonstration of tūQ, a classical GUI-based toolchain for modelling, optimising and compiling an algorithm as an expression of the cluster-state model. The cluster-state model and its equivalent graph-state are explored as an alternative to the widespread circuit model of quantum computing. The tūQ toolchain consists of two main modules, Modeller and Simulator as well as functions that integrate the modules. Modeller was created as a sandpit environment, in which its user can experiment with a graph state through measurement functions that are consistent with the stabiliser formalism. Simulator was created for expressing an algorithm in graph-state syntax; and for reading a circuit and converting it to a graph state. The user can combine Modeller with Simulator in iterative workflow to express and test an algorithm before transpiling it to the quantum assembly language, OpenQASM. The tūQ toolchain is comparable in scope and power with the many (classical) circuit modelling tools that are currently available.

Artefacts

The thesis discussed herein is the source of a number of artefacts, which are listed below for reference.

Publications

Bowen, Greg and Simon Devitt (2022) 'Q2Graph: a modelling tool for measurement-based quantum computing'. **arXiv**, DOI: 10.48550/arXiv.2210.00657.

Bowen, Greg and Simon Devitt (2025) 'tūQ: a design and modelling tool for cluster-state algorithms'. **arXiv**, DOI: 10.48550/arXiv.2502.18991.

Bowen, Greg, Athena Caesura, Simon Devitt and Madhav Krishnan Vijayan (2025) 'Design and efficiency in graph-state computation'. **arXiv**, DOI: 10.48550/arXiv.2502.18985.

Conference presentation

'Design and efficiency in a graph-state compiler'. **ANZCOP-AIP** summer meeting 03-08 December 2023, Canberra, Australia, URL: `https://virtual.oxfordabstracts.com/event/4612/homepage`.

Conference posters

'Q2Graph: a modelling tool for measurement-based quantum computing'. **Quantum Australia** 23-25 February 2022, Sydney, Australia.

‘Dynamic provisioning of a measurement-based quantum computing cluster state’. **Quantum Australia** 21-23 February 2023, Sydney, Australia.

Source code

Q2Graph, URL: `https://github.com/QSI-BAQS/Q2Graph`.

Etch, URL: `https://github.com/QSI-BAQS/Etch`.

tūQ, URL: `https://github.com/QSI-BAQS/tuQ`.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The circuit model is widely viewed within the quantum computing community as the *de facto* standard for expressing and computing quantum algorithms. As at December 2024, most publicly-accessible quantum computing services and quantum computing emulators have an architecture founded upon the circuit model. Many quantum computing services also supply a proprietary emulator, generally a graphical user interface ('GUI') for expressing a circuit-based algorithm or a domain-specific language organised around circuit model principles. In practice, these quantum computing services and emulators are tightly coupled. Even the open source quantum assembly language, OpenQASM, for those quantum computing services that accept it as algorithmic input, enforces computation by quantum logic gates of the circuit model. This dominance of the circuit model may be partly due to the limited availability and coherence times of qubit resources accessible through the so-called noisy, intermediate-scale quantum ('NISQ') computers of the current era (Preskill 2018). Notwithstanding any possible link to hardware, dominance of the circuit model imposes a certain regimentation of thinking upon ongoing research into quantum algorithms and computation.

The following dissertation is an extended proof-of-concept for the '`tūQ`' compiling toolchain, which advances the cluster-state quantum

computing model as an alternative to the circuit. The tūQ toolchain is a classical GUI-based platform for modelling, optimising and compiling an algorithm as an expression of the cluster-state model (e.g. Raußendorf and Briegel 2001, 2002; Raußendorf, Browne, and Briegel 2003; Nielsen 2004, 2006; Briegel et al. 2009). The user of the tūQ toolchain can,

- read a circuit and convert it to a cluster state,
- draft an algorithm in a syntax compatible with cluster-state principles,
- reduce a cluster state in a sandpit environment to identify possible optimisations of an algorithm; and gain insights of how the computation might resolve, given a processor built to implement the cluster-state model,
- transpile an algorithm to OpenQASM 3.0, to implement on a quantum computer or another emulator.

The remainder of this dissertation will introduce this functionality through contextual analysis and worked examples.

A cluster state can describe two or more dimensions but canonically a two-dimensional lattice, with its corollary in a *graph state*, is the common expression. The tūQ user can model a graph state as well as draft a graph-based algorithm for formal computation. Input in graph format stands in place of a high-level programming language traditionally associated with a classical compiler. Indeed, the graph state also functions as a quantum intermediate representation ('QIR') to tūQ and is similarly responsive to optimisation. Finally, tūQ provides for transpiling an algorithm to the OpenQASM standard, facilitating an end-to-end workflow of iterative drafting-modelling of a graph-state algorithm then computation through any quantum computing service that accepts the standard as input.

In all, this dissertation is a solution to the research question of how to design a sufficient quantum compiling toolchain that supports the cluster-state computing paradigm. The significance of this research is that `tūQ` is a compiling toolchain specifically engineered for a cluster-state architecture and thus stands apart from a toolchain engineered for traditional circuit-based NISQ architectures.

1.1 Terminology and layout

Certain terms appear repeatedly in this dissertation and require a specific interpretation, unless explicitly stated to the contrary. The reader should interpret the occasional reference to 'operating system', 'kernel' or 'process' as being an unspecified distribution that is otherwise compatible with the POSIX standard[1]. Moreover, a reference specifically to operating system conflates the otherwise specialised functions of personal computer and server operating systems.

In keeping with established convention, the computer architecture of binary based on relative voltage is denoted as 'classical' and all feasible computational applications of such architecture as 'classical computing'. As will become apparent, demarcating the classical computing domain from the quantum on the basis of 'feasible' computations is arbitrary and unavoidable. The terms 'quantum computer', 'quantum machine' and 'quantum hardware' will be interchangeable; a 'NISQ computer' is an instance of quantum computer/machine/hardware although *not conversely*. Fuller definitions of and distinctions between these terms appears in Section 2.4, below.

Occasionally a quantum compiler phase may require illustrating with a similar phase of a classical compiler. The classical compiler GNU `gcc` (`https://gcc.gnu.org/`) is the standard in such instances. Note

[1]Requirements for POSIX compliance appear at URL: `https://posix.opengroup.org`.

also that a reference to a software package, classical or otherwise, will appear in `TrueType` font.

From this introduction to the research, Chapter 2 opens with a review of the principles of quantum computing then continues with a detailed analysis of the cluster-state and graph-state models. A review of the quantum computing landscape as at December 2024 will set the context for the itemised requirements for a cluster-state compiler that close the chapter.

The remaining chapters of this dissertation form an extended case study of the `tūQ` toolchain. Chapter 3 is an introduction to `tūQ` as a tool for modelling graph states. The chapter begins with an in-depth analysis of graph states as both an expression of the cluster state and the format for encoding algorithms through `tūQ`. Graph states are a substitute for the text-based languages of classical compilers. Modeller's purpose as a sandpit for reducing a lattice in a demonstration of cluster-state processes lays the groundwork for a companion tool for expressing algorithms in graph-state format. There is also a video component to this chapter: the first of three separate video attachments prepared for this dissertation is a demonstration of `tūQ`'s Modeller mode.

Chapter 4 continues the exploration of graph states, in this case the graph state as syntax for expressing a cluster-state algorithm. A brief review of quantum emulators and their use case is used to position the `tūQ` toolchain's Simulator mode. Simulator's origins lie with an enquiry into the practicalities of 'circuit etching' as an alternative to the algorithm-specific graph. A workflow combining Simulator and Modeller for the purposes of optimising algorithms is considered. This chapter has a demonstration video, being the second of the three attachments prepared for this dissertation and focused on `tūQ`'s Simulator mode.

Chapter 5 is an introduction to functions that further integrate `tūQ`'s Modeller and Simulator modes and facilitate the actual computing of a graph-state algorithm drafted in Simulator. Worked examples of `tūQ`'s

‘Open Algorithm’ and ‘Compile’ functions give details of how Modeller and Simulator combine in workflow for optimising algorithms. The third of the three videos prepared for this dissertation is an attachment to this chapter and provides a demonstration of the practicalities of such workflows. The chapter concludes with an evaluation of tūQ’s claim as a compiler toolchain.

Chapter 6 is an appraisal of the tūQ toolchain in totality and concluding remarks. Directions on how to install tūQ and an introduction to functions of the toolchain’s Modeller and Simulator modes appear respectively as Appendix A and Appendix B of this dissertation.

1.2 Summary

The tūQ toolchain is a proof-of-concept compiler for an as-yet unrealised cluster-state architecture. As such, the toolchain promotes an alternative to the widespread circuit model for modelling, optimising and compiling an algorithm. The research question considered in this dissertation is thus how to assemble a toolchain that is sufficient for compiling an algorithm that expresses the cluster-state model. As stated, the significance of this research is that it presents a complete and alternative compiler framework to those that support the circuit model.

Chapter 2

Background

This chapter serves to introduce two paradigms of quantum computing and the fundamentals of compiling as found in the classical and quantum domains of computation. A general introduction to the principles of quantum computing is the foundation for analysis of circuit-based then, cluster-state models of quantum computing. Of the two models, the circuit is arguably simpler and its instances indisputably more widespread while a cluster-state, constituting a stabiliser state and also grounded in graph theory, may be more complete in terms of its processes. A review of the landscape of NISQ hardware and quantum computing emulators as at December 2024 lends further context to the respective penetration of these models into ongoing research in quantum computing.

The remaining consideration is compiler technology, well-established in classical computing but emerging in quantum computing. For the foreseeable future, quantum computing including processes associated with compiling will depend upon classical computing architecture. A minimal blueprint is set out for a cluster-state compiler within a mixed quantum-classical architecture. Although compiling is a single function of the `tūQ` toolchain showcased in this research, it is closely tied with a revival of the second model for quantum computation.

Original material

- This chapter sets context to research and development of compiling and compilers for quantum computers. It includes a review of principles of quantum computing and relevant concepts of classical compiling as well as an overview of the state of quantum computing services as at December 2024.

2.1 Principles of quantum computing

The basic unit of quantum computing is the *qubit*, a namesake of the bit of classical computing. A qubit is a mathematical abstraction from properties observed in such elementary particles as electrons, photons, trapped ions and neutral atoms. A qubit presents as a two-state system within a complex 2^n-dimensional vector space. Adopting the Dirac notation of a *ket* to designate an element of this vector space, the general state of a qubit, $|\psi\rangle$ is

$$|\psi\rangle = \alpha_0 |0\rangle + \beta_1 |1\rangle = \begin{pmatrix} \alpha_0 \\ \beta_0 \end{pmatrix} \tag{2.1}$$

for which α_0 and β_1 are complex numbers that must meet the normalisation condition, namely,

$$|\alpha_0|^2 + |\beta_1|^2 = 1. \tag{2.2}$$

The two states $|0\rangle$ and $|1\rangle$ are orthogonal by their inner product,

$$\langle 0|1\rangle = 0 \tag{2.3}$$

and therefore $|\psi\rangle$ has an orthonormal basis. Note also, α_0 and β_1 are probability amplitudes and when squared, equate to the probability of observing or 'measuring' $|\psi\rangle$ in either state $|0\rangle$ or $|1\rangle$, respectively.

Measurement is a stochastic process: the qubit, $|\psi\rangle$ of equation (2.1) is a *superposition* of states $|0\rangle$ and $|1\rangle$ and only at measurement does

it irreversibly collapse to either state $|0\rangle$ or $|1\rangle$. As a reference, measured state $|0\rangle$ and $|1\rangle$ is termed the 'computational basis'. If qubit a is measured as $|0\rangle$ along the z-axis then, each subsequent measurement of qubit a along the z-axis will return $|0\rangle$. If qubit a is next measured along the x-axis then, qubit a returns to state $|\psi\rangle$ along the z-axis and any subsequent measurement on that axis is again stochastic.

The general state of two qubits, $|\Psi\rangle$ is also a normalised superposition of four states,

$$|\Psi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle = \begin{pmatrix} \alpha_{00} \\ \alpha_{01} \\ \alpha_{10} \\ \alpha_{11} \end{pmatrix}, \tag{2.4}$$

with probability amplitudes that sum to unity,

$$\sum_{0 \leq x < 2^n} |\alpha_x|^2 = 1. \tag{2.5}$$

The state of two qubits, $|\psi\rangle = \alpha_0|0\rangle + \alpha_1|1\rangle$ and $|\phi\rangle = \beta_0|0\rangle + \beta_1|1\rangle$ as a pair described by state $|\Psi\rangle$ is the *product state* of respective states for which the normalisation condition of equation (2.5) holds,

$$|\Psi\rangle = |\psi\rangle \otimes |\phi\rangle = (\alpha_0|0\rangle + \alpha_1|1\rangle) \otimes (\beta_0|0\rangle + \beta_1|1\rangle). \tag{2.6}$$

Not every state of two or more qubits is a product state. If it is possible to express a combined state of two qubits as a tensor product then, the state is separable; if expressing it as a tensor product is impossible, the state is *entangled*. For example,

$$|\Psi\rangle = |01\rangle, \tag{2.7}$$

is separable while,

$$|\Psi\rangle = |01\rangle + |10\rangle, \tag{2.8}$$

is not separable and hence, entangled. Note entanglement as a system is not restricted to two-qubits states: the Greenberger-Horne-Zeilinger ('GHZ') state,

$$|GHZ\rangle = \frac{|000\rangle + |111\rangle}{\sqrt{2}}, \tag{2.9}$$

is an example of a three-qubits entangled system. The significance of entangled systems to quantum information (and therefore, quantum computing) lies in the associated phenomenon of quantum *teleportation*. If a two qubit system is entangled, the qubits are correlated: measuring one of the qubits will simultaneously destroy entanglement and prepare ('teleport') the state of the measured qubit to the other, unmeasured qubit. Quantum teleportation is foundational to the proposition of computing via a cluster state.

On the basis that a qubit is described as a vector, linear transformation is the mechanism for modelling change of state. A *unitary* linear transformation, $U|\psi\rangle$ preserves the magnitudes of all vectors; it also follows from linearity,

$$U(\alpha|\psi\rangle + \beta|\phi\rangle) = \alpha U|\psi\rangle + \beta U|\phi\rangle \tag{2.10}$$

that unitary linear transformation preserves the inner product of arbitrary pairs of vectors and that probability is therefore conserved. An *eigenvector* $|\psi\rangle$ is any vector for which transformation by linear operator M is identical to the product of $|\psi\rangle$ and scalar, a termed the *eigenvalue* hence,

$$M|\psi\rangle = a|\psi\rangle\,. \tag{2.11}$$

Ordinarily the eigenvalue, a is a complex number however if M is self-adjoint or 'Hermitian' that is,

$$M = M^{\dagger}, \tag{2.12}$$

then, a is a real number. Note, each of the Pauli matrices,

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{2.13}$$

is unitary and Hermitian. As will be discussed in Section 2.2.2, real number eigenvalues are significant in the equivalence of cluster states and stabiliser states.

In effect, each quantum logic gate is a linear transformation operator. There are minimal sets of quantum logic gates that bestow *universality* of operations in quantum computing. Each of these sets of gates can approximate, albeit efficiently, any unitary operator, U to transform state (Deutsch, Barenco, and Ekert 1995; Barenco et al. 1995; also, Nielsen and Chuang 2010). This property of quantum logic gate sets is similar to the set of logic gates NOT and AND ('NAND') in classical computing, which can effect all possible transformations of state.

The universal set of quantum logic gates considered in this dissertation includes the so-called Clifford group of single-qubit operators, Hadamard- (H) and S- ('phase') gates; and control operator, CNOT,

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \quad S = \begin{bmatrix}1 & 0\\ 0 & e^{i\frac{\pi}{2}}\end{bmatrix} \quad CNOT = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\\ 0 & 0 & 1 & 0\end{bmatrix}; \tag{2.14}$$

each of which will map a Pauli matrix to a Pauli matrix. The Clifford group acts to normalise the N-qubits Pauli group, $\mathcal{P}^N$,

$$\mathcal{P}^N \coloneqq \{\pm I, \pm iI, \pm X, \pm iX, \pm Y, \pm iY, \pm Z, \pm iZ\}^N, \tag{2.15}$$

in which I is the identity-gate and X, Y, Z are the Pauli matrices of equation (2.13), above; with superscript, † denoting a conjugate transpose of the matrix, it follows,

$$\begin{aligned} HXH^\dagger = Z,\ \ HZH^\dagger = X,\ \ HYH^\dagger = -Y;\\ SXS^\dagger = Y,\ \ SZS^\dagger = Z,\ \ SYS^\dagger = -X;\\ \text{CNOT}\,(X\otimes I)\,\text{CNOT}^\dagger = X\otimes X,\\ \text{CNOT}\,(I\otimes X)\,\text{CNOT}^\dagger = I\otimes X,\\ \text{CNOT}\,(Z\otimes I)\,\text{CNOT}^\dagger = Z\otimes I,\\ \text{CNOT}\,(I\otimes Z)\,\text{CNOT}^\dagger = Z\otimes Z. \end{aligned} \tag{2.16}$$

Adding the non-Clifford $\frac{\pi}{4}$ ('T') gate,

$$T = \begin{bmatrix}1 & 0\\ 0 & e^{i\frac{\pi}{4}}\end{bmatrix} = \sqrt{S}, \tag{2.17}$$

to the Clifford group completes a universal gate set although unlike Clifford group operators, T gate does not map Pauli to Pauli. For example,

$$\begin{aligned} TZT^\dagger &= \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix} \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\frac{\pi}{4}} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix} \notin \mathcal{P}^N, \\ TXT^\dagger &= \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix} \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \begin{bmatrix} 1 & 0 \\ 0 & e^{-i\frac{\pi}{4}} \end{bmatrix} = \begin{bmatrix} 0 & e^{-i\frac{\pi}{4}} \\ e^{i\frac{\pi}{4}} & 0 \end{bmatrix} \notin \mathcal{P}^N. \end{aligned} \tag{2.18}$$

Finally, it is a requirement that every quantum logic gate is unitary and therefore *reversible* to conserve probability of quantum states, that is

$$UU^\dagger = U^\dagger U = 1. \tag{2.19}$$

2.2 Quantum computing models

NISQ computers are characteristic of the present era of quantum hardware development; an in-depth review of these devices by qubit resources and front ends appears in Section 2.4, below. A limited supply of physical qubits (e.g. Arute et al. 2019; Gold et al. 2021; Wack et al. 2021) and sensitivity of qubits to errors arising from environmental decoherence or 'noise' (e.g. Preskill 2018; Kandala et al. 2019) are typical of present-day NISQ computers. Noise has an inverse relationship with the coherence times of qubits so that noisy qubits with short coherence times will necessarily increase the probability of errors occurring at one or more points in a computation. For this reason, many quantum computing service providers also offer *emulator* packages, commonly in the form of a software development kit (SDK) or a platform as a service (PaaS) offering, that mimic the workings of an idealised quantum computer on a classical computer; further details of these quantum emulators appear in Section 2.4.1, below. Due to their ready availability, quantum emulators are a regular component of experimentation in

quantum computing: the user of a given emulator can expect to prepare the state of (simulated) qubits and transform them by means of standard operators to then observe an output free of errors.

More than one framework exists in the quantum computing literature to represent a system state of qubits and the permissible operators to transform state. The quantum circuit is the prevalent method for modelling computation through change of state. The cluster state and its modelling standard, the graph state, stand in contrast to the circuit. Reviews of each framework appear below. It is imperative first to place these frameworks within a proper context: the limitations of current quantum computers are a brake on features of a quantum computing language, at least as compared to such classical languages as C, TypeScript or Elixir. Neither of the following quantum computing frameworks includes a type system or operator precedence or such syntactic features as control flow, iteration or memory allocation. In short, both the circuit- and cluster state frameworks that appear below are marginally more expressive than assembly languages of classical computing. An algorithm formalised through either of the circuit- and cluster state models is redolent of those associated with punched cards and classical mainframes of the 1940s through 1960s. The effects of this ceiling on language abstraction will reappear at points throughout this dissertation.

2.2.1 Circuit model

The use of logic gate notations was an illustrative device adopted in the proof-of-concept for quantum computing (Feynman 1986; also, Feynman 1982). Syntactically, a circuit consists of n qubits represented as horizontal 'wires', oriented east-west, with pictograms denoting logic gate operations mounted at various points upon the wires. The qubits are initialised in an arbitrary state, $|\psi\rangle$ although commonly this initial state is $|0\rangle$. Representing each qubit as a wire denotes progress of the computation through time.

Semantically, each logic gate on a wire represents a unitary transformation applied to the qubit and therefore the combined effect of all gates is equivalent to a unitary transformation of the input qubits. After DiVincenzo (1997), a quantum circuit should meet the following requirements,

(i) a suitable state space of n qubits to fulfil the computation,

(ii) ability to prepare states in the computational basis,

(iii) ability to perform quantum gates on any subset of (i), as required; and

(iv) ability to perform measurements in the computational basis.

A quantum circuit evolves, left-to-right, through successive gate operations. In other words, the circuit model 'builds' its output through incremental transformations of its element qubits. Figure 2.1 is a demonstration of the quantum circuit to obtain the GHZ entangled state of equation (2.9), above. Note, the circuit model requires only a *suitable*

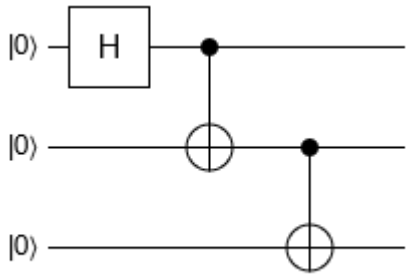


Figure 2.1: The circuit to create the Greenberger-Horne-Zeilinger ('GHZ') entangled state of equation (2.9), above. The initial state, visible at the left-most terminus of the qubit wires, is set at $|0\rangle$. The circuit of three qubits evolves, left-to-right, through a Hadamard gate on qubit 1 then two successive CNOT gate operations, first on qubits 1 and 2 and second on qubits 2 and 3. Only after applying the last gate of the circuit will each qubit be measured.

state space of qubits as a condition of computation; that is, the circuit may begin as a separable tensor product of qubits. A specific state space

of qubits, as is a requirement of the cluster-state model, is not *ex ante* fundamental to the circuit model. A quantum computer will measure each qubit of the circuit to obtain the computational output, only after applying the last gate of the algorithm. The relative simplicity of the model has contributed to circuits becoming the *de facto* standard for encoding a quantum computing algorithm.

In effect, the majority of publicly accessible NISQ computing services, including any emulator package the service provider might also provide, only admit input organised in circuit format. As shall become apparent from Section 2.4, below, all publicly accessible NISQ computing services as at December 2024 work from a (quantum) processing unit that enforces the circuit model as the syntax for inputting an algorithm. These same NISQ computers also have a limited availability of qubit resources, making it hard to compute through a cluster-state which typically consumes more qubits on average than an equivalent circuit for the same output. That access to present day NISQ computers or their emulators dictates it is thus another factor in the dominance of the circuit model.

A circuit-based algorithm relies upon each and every gate transformation completing without error and further, no publicly accessible NISQ computing service as at December 2024 will support real-time feedforward of state. A researcher cannot make such assumptions of an algorithm processed in a NISQ computer, which does not correct for quantum errors and is not fault tolerant. By the Central Limit Theorem (cf. Feller 1991), repeated samples ($n \geq 30$) of an algorithm's output from a NISQ computer are sufficient for the output to tend towards the normal distribution, with the mean output being a reliable estimator of the 'true' computational output. However, computing through a NISQ facility is inescapably an exercise in probability and any circuit-based algorithm processed through a NISQ computer is better read as a statement of procedure than as a guarantee of outcome (cf. Google 2023a).

2.2.2 Cluster-state model

The problem of randomness of measurement output is not unique to the circuit model rather, cluster-state models are a different method for dealing with it. To begin, quantum entanglement and the property of quantum teleportation are necessary to the model of a cluster state as a computational medium. This requirement for an initial state space of entangled qubits is in contrast with the looser requirements for state space of qubits under the quantum circuit model.

More than one cluster-state model appears in the literature (e.g. Childs, Leung, and Nielsen 2005; Browne and Rudolph 2005) but discussion in this dissertation as well as the design principles of its accompanying compiler toolchain, `tūQ` is based on the measurement-based quantum computing model (Raußendorf and Briegel 2002; Raußendorf, Browne, and Briegel 2003; Raußendorf, Harrington, and Goyal 2006). Direct reference to the measurement-based quantum computing (henceforth, 'MBQC') model will be made only as necessary.

A cluster state, $|\Phi\rangle$, including the 'substrate' of MBQC (Raußendorf and Briegel 2001), is also known as a *graph state*, designated $|G\rangle$ and the titles are interchangeable (e.g. Hein, Eisert, and Briegel 2004; Nielsen 2006), that is

$$|\Phi\rangle \equiv |G\rangle\,. \tag{2.20}$$

As a summary of its fundamentals as a graph, G, a graph-state is an ordered pair consisting of a non-empty, finite set of vertices, $V\{G\}$ and a finite set of edges, $E\{G\}$. Vertices are qubits, edges are interactions between qubits and $|G\rangle$ must be a *simple* graph, prohibiting more than one edge between vertices and an edge as a 'self-join' of a qubit. Further discussion of graph states appears in Chapter 3 of this dissertation.

A cluster state is a system of n-interacting qubits, with a topology of dimensions $\mathbb{Z}^d$, where d (dimension) ≥ 1. Raußendorf and Briegel (2002) denote the MBQC cluster state as $QC_{\mathcal{C}}$. As noted above, this dissertation proceeds with instances of $QC_{\mathcal{C}}$ (or, $|G\rangle$) configured as a two-dimensional lattice (cf. Bolt et al. 2016). Preparing $QC_{\mathcal{C}}$ requires,

1. a set of qubits in arbitrary state, $|\psi\rangle$ (the 'input qubits')[1]; and,

2. all qubits that are not elements of the set of 1. in state, $|+\rangle = \frac{|0\rangle+|1\rangle}{\sqrt{2}}$; then,

3. entangling the qubits of 1. and 2. through a controlled-phase (CZ) interaction, where

$$CZ = diag(1, 1, 1, -1). \tag{2.21}$$

   The particulars of implementing the CZ interactions of a cluster state, $|\Phi\rangle$ are often unspecified (e.g. Nielsen 2004; Cabello et al. 2011; Adcock et al. 2020; Dahlberg, Helsen, and Wehner 2020) whereas Briegel and Raußendorf (2001; also, Raußendorf and Briegel 2002) stipulate the Ising nearest-neighbour interaction to obtain entanglement of $QC_{\mathcal{C}}$.

The algorithm is the variable that dictates the total number of qubits sufficient for $QC_{\mathcal{C}}$ as shall become apparent in Chapter 4 of this dissertation. A general cluster state represented as $|G\rangle$ with qubits ($v \in V\{G\}$) in state $|+\rangle$ and CZ interactions between qubits ($(u, v) \in E\{G\}$) is therefore,

$$|G\rangle = \prod_{(u,v)\in E\{G\}} CZ^{(u,v)} \left(\bigotimes_{v\in V\{G\}} |+\rangle_v \right). \tag{2.22}$$

An example of a $QC_{\mathcal{C}}$ lattice appears as Figure 2.2.

Cluster states and therefore, graph-states are also *stabiliser* states. An n-qubits stabiliser state, $|\phi\rangle$ is a simultaneous eigenvector with eigenvalue $+1$ of n-commuting and independent elements of the Pauli group, $\mathcal{P}^N$ of equation (2.15), above (Gottesman 1997; Van den Nest, Dehaene, and De Moor 2004a; Hein, Eisert, and Briegel 2004). The stabilis-

[1] Raußendorf and Briegel (2002) are subtle on this point, showing that MBQC proceeds effectively with a *known* quantum input state; nonetheless, requirement 1. as stated is not inconsistent with MBQC but merely amounts to a prescriptive instance of it (Raußendorf and Briegel 2001).

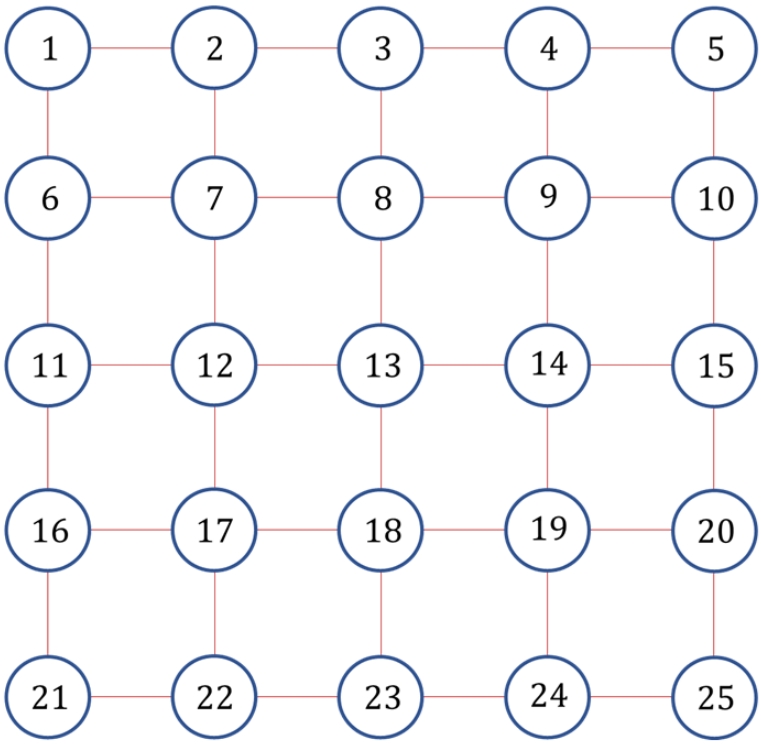


Figure 2.2: A 5 x 5 lattice. Vertices represent qubits while edges signify CZ interactions. Interpreting the lattice as $QC_{\mathcal{C}}$, input qubits form the left-most column (IDs 1, 6, 11, 16 and 21) while readout qubits form the right-most column (IDs 5, 10, 15, 20 and 25). The state of $QC_{\mathcal{C}}$ will evolve left-to-right with successive single-qubit measurements.

ers of $|\phi\rangle$ are defined as,

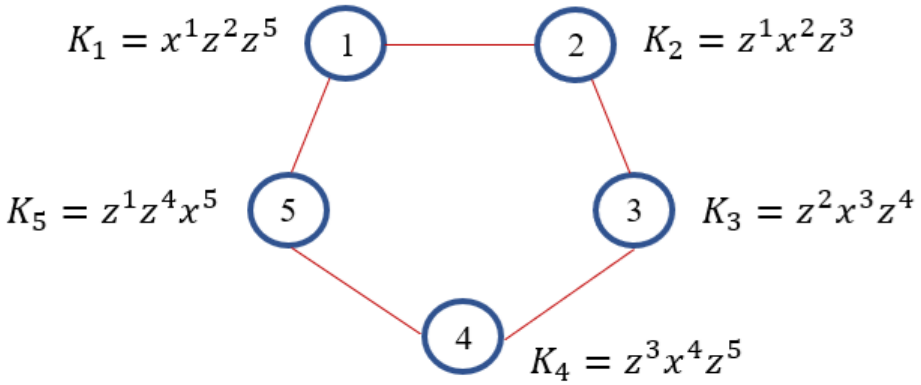
$$\mathcal{S} := \{M \in \mathcal{P}^N | M\,|\phi\rangle = |\phi\rangle\}, \tag{2.23}$$

being a group of operators with real overall phase $+1$ and M is a generator of $\mathcal{S}$. By extension, every vertex, i of a graph state, $|\psi\rangle$ has a stabiliser, $K^{(i)}$ such that,

$$K^{(i)} = \sigma_x^{(i)} \bigotimes_{j \in E} \sigma_z^{(j)}, \tag{2.24}$$

with E denoting the set of all vertices neighbouring i; thus $\mathcal{S}$ for G is the set, M, where,

$$\{M\} = K^{(i)}, i = (1 \ldots N). \tag{2.25}$$



$$\begin{bmatrix} X & Z & I & I & Z \\ Z & X & Z & I & I \\ I & Z & X & Z & I \\ I & I & Z & X & Z \\ Z & I & I & Z & X \end{bmatrix}$$

Figure 2.3: A representation of a five-vertices graph by its stabilisers and its equivalent adjacency matrix. Each vertex is prepared in state $|+\rangle$ and $X|+\rangle = |+\rangle$ holds.

Graph-state $|\psi\rangle$ and the stabiliser state as encoded in an adjacency matrix are equivalent (see Hein, Eisert and Briegel (2004) for a detailed discussion).

Figure 2.3 is an example of a simple five-vertices graph and its equivalent adjacency matrix. Assume each vertex of the graph is prepared in state $|+\rangle$, as consistent with $QC_{\mathcal{C}}$. Observing from equation (2.23) that

$$X|+\rangle = |+\rangle\,, \tag{2.26}$$

the stabiliser of each vertex and its neighbours describes a [5, 5] matrix with vertex coordinates, X on the main diagonal, all neighbouring qubits designated Z and the identity, I elsewhere.

Computation through $QC_{\mathcal{C}}$ proceeds by measuring single qubits in a certain order and basis with overall quantum information propagat-

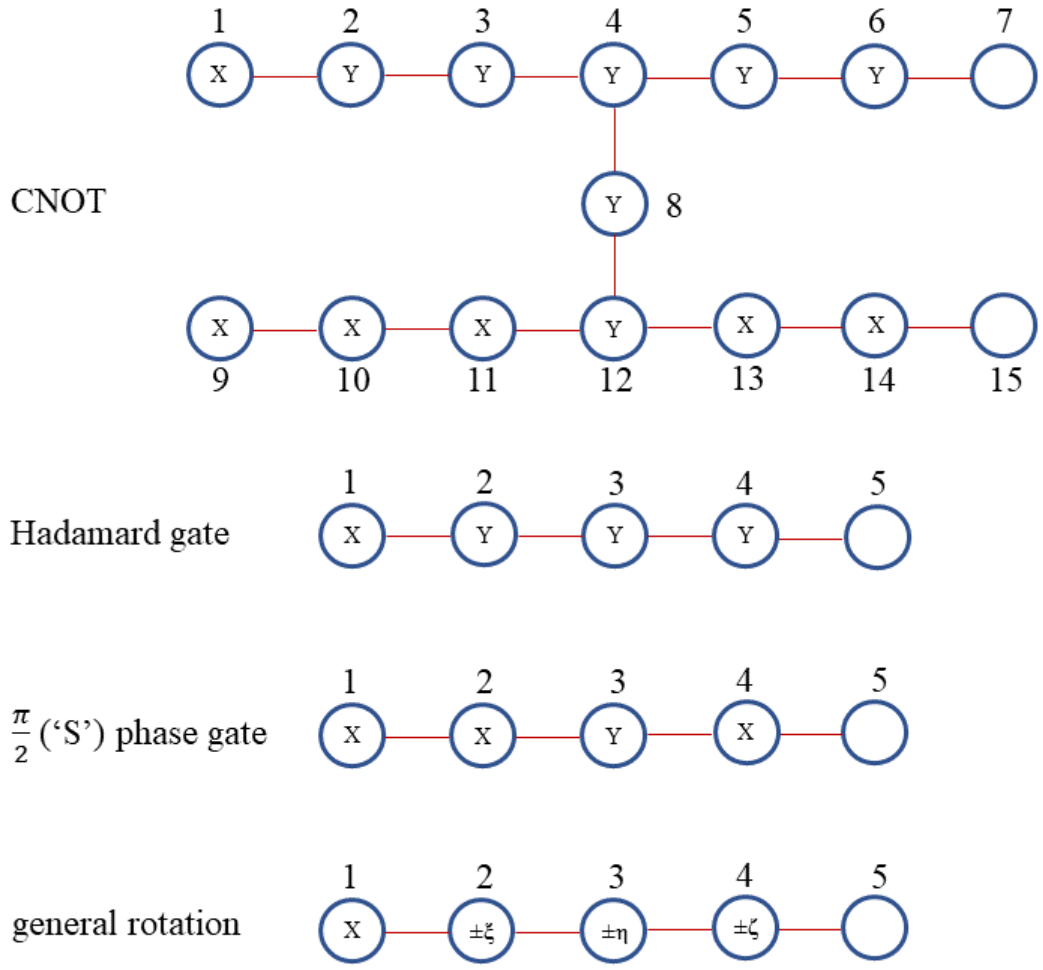


Figure 2.4: Individual X, Y and Z Pauli measurements composed to replicate Clifford group operations (after Raußendorf and Briegel (2002), Figure 1). Each vertex label is the required Pauli measurement; an unlabelled vertex signifies a readout qubit to be measured in the computational basis. The integer above a vertex is its identification. Applying Euler angles, ξ, η, ζ effects the appropriate rotation, leaving any superfluous Euler angle to be measured in basis σ_x.

ing left-to-right. Figure 2.4 is a demonstration of Clifford group operators and rotations about axes $x/y/z$ as single-qubit measurements of $QC_{\mathcal{C}}$. In general, the basis of measurement for qubit i depends upon the results of preceding measurements. In regard to the Clifford group operators Hadamard, S and CNOT, the measurements are prescribed combinations in basis σ_x or σ_y and excepting the readout qubits, these

measurements can occur simultaneously. The general rotation, U_{Rot} is more illustrative of the 'in general' dependency of measurement. After the extended example of Raußendorf and Briegel (2002), qubits 1-4 are measured in basis,

$$\mathcal{B}_j(\varphi_{j,meas}) = \left\{ \frac{|0\rangle_j + e^{i\varphi_{j,meas}} |1\rangle_j}{\sqrt{2}}, \frac{|0\rangle_j - e^{i\varphi_{j,meas}} |1\rangle_j}{\sqrt{2}} \right\} \tag{2.27}$$

to obtain measurement outcome $s_j \in \{0, 1\}$; further, measurement angle $\varphi_{j,meas}$, the measurement basis for qubit j, equals the angle between the measurement at j and the positive x-axis. If $s_j = 0$, project qubit j into the first state of $\mathcal{B}_j(\varphi_{j,meas})$. Substituting in the appropriate Euler angles, the procedure to rotate a qubit is measure,

1. qubit 1 in $\mathcal{B}_1(0)$,
2. qubit 2 in $\mathcal{B}_2\left(-\xi(-1)^{s_1+\kappa_{1,I}}\right)$,[2]
3. qubit 3 in $\mathcal{B}_3\left(-\eta(-1)^{s_2+\kappa_2}\right)$,
4. qubit 4 in $\mathcal{B}_4\left(-\zeta(-1)^{s_1+s_3+\kappa_{1,I}+\kappa_3}\right)$,

where $\kappa_{1,I}$ denotes the input qubit and for this example, assume that qubits 1-5 are prepared in state $|+\rangle$ so that $\kappa_{1,I} = 0$. Quantum information teleports to 'readout', qubit 5. Steps 1-4 result in the unitary transformation,

$$U'_{Rot}\left[\xi, \eta, \zeta\right] = U_{\sum,Rot} U_{Rot}\left[\xi, \eta, \zeta\right], \tag{2.28}$$

in which $U_{\sum,Rot}$ represents the *byproduct* of measurement. Any and all readout qubits work in essence as a quantum register and measurement is contingent upon their bases; if

[2] Raußendorf and Briegel (2002) use notation κ' in their example of the general qubit rotation, to distinguish it from notation κ which they reserve for describing the cluster state $|\phi\rangle_{\mathcal{C}}$ on the whole cluster, $\mathcal{C}$. Observing the same distinction is superfluous to this example.

- readout measurements are in basis σ_x, σ_y or σ_z, they are 'interpreted' according to their byproduct; otherwise
- previous measurements determine the basis of readout measurement.

Irrespective of the stochastic outcome of measuring qubit j of $QC_{\mathcal{C}}$, MBQC is a deterministic computing model. Any qubit measurement that moves $QC_{\mathcal{C}}$ away from a +1 eigenvalue necessarily creates a byproduct and which then determines the method of measuring a readout qubit. These byproducts are unavoidable but they can be offset (Raußendorf, Browne, and Briegel 2003; Browne and Briegel 2006; Browne et al. 2007). Once again after Raußendorf and Briegel (2002), in the case of the general qubit rotation outlined above, the byproduct is,

$$U_{\sum,Rot} = \sigma_x^{s_2+s_4+\kappa_2+\kappa_4}\sigma_z^{s_1+s_3+\kappa_{1,I}+\kappa_3+\kappa_{5,\mathcal{O}}}. \tag{2.29}$$

where $\kappa_{5,\mathcal{O}}$ denotes the readout qubit. In the case of a Clifford group operator say, the 15-qubits CNOT, U_{CNOT}, its byproduct is,

$$U_{\sum,CNOT} = \sigma_x^{(c)^{\gamma_x^{(c)}}}\sigma_x^{(t)^{\gamma_x^{(t)}}}\sigma_z^{(c)^{\gamma_z^{(c)}}}\sigma_z^{(t)^{\gamma_z^{(t)}}} \tag{2.30}$$

for which,

$$\begin{aligned}
\gamma_x^{(c)} =& s_2 + s_3 + s_5 + s_6 + \kappa_2 + \kappa_3 + \kappa_5 + \kappa_6,\\
\gamma_x^{(t)} =& s_2 + s_3 + s_8 + s_{10} + s_{12} + s_{14} + \kappa_2 + \kappa_3 + \kappa_8 + \kappa_{10} + \kappa_{12} + \kappa_{14},\\
\gamma_z^{(c)} =& s_1 + s_3 + s_4 + s_5 + s_8 + s_9 + s_{11} + \kappa_{1,I} + \kappa_3 + \kappa_4 + \kappa_5 + \kappa_{7,\mathcal{O}} +\\
& \kappa_8 + \kappa_{9,I} + \kappa_{11} + 1,\\
\gamma_z^{(t)} =& s_9 + s_{11} + s_{13} + \kappa_{9,I} + \kappa_{11} + \kappa_{15,\mathcal{O}}.
\end{aligned} \tag{2.31}$$

Additional rotations in basis σ_x or σ_z, depending upon the measurement result, can be propagated through the readout qubit of any operation. The propagation relation for the examples of the general qubit rotation and CNOT are, respectively,

$$U_R\left[\xi,\eta,\zeta\right]\sigma_z^s\sigma_x^{s'} = \sigma_z^s\sigma_x^{s'}U_R\left[(-1)^s\xi,(-1)^{s'}\eta,(-1)^s\zeta\right]; \tag{2.32}$$

and

$$\text{CNOT}(c,t)\sigma_z^{(t)^{s_t}}\sigma_z^{(c)^{s_c}}\sigma_x^{(t)^{s'_t}}\sigma_x^{(c)^{s'_c}} = \sigma_z^{(t)^{s_t}}\sigma_z^{(c)^{s_c+s_t}}\sigma_x^{(t)^{s'_c+s'_t}}\sigma_x^{(c)^{s'_c}}\text{CNOT}(c,t). \tag{2.33}$$

Moreover, a (classical) Pauli tracker makes it possible to log the additional rotations in basis σ_x or σ_z through the course of processing the computation, to correct or interpret computational basis measurement of the readout qubit(s); see Ruh and Devitt (2024) for a recent implementation of Pauli tracking through a classical application.

A review of cluster-state hardware or emulator packages is appropriate in closing this analysis of the cluster-state model. As at December 2024, no publicly accessible NISQ platform will accept an algorithm encoded as a cluster state or, by implication, will process a computation through a cluster state[3]. That said, it is possible to draft an algorithm in circuit format to construct a graph on a NISQ computer; otherwise, cluster state computation is represented with at least one:

- (alpha release, Python) library, Mentpy, for 'creating and training quantum machine learning models in the measurement-based quantum computing framework,' at `https://github.com/BestQuark/mentpy`; and
- emulator, Graphix, 'a measurement-based quantum computing compiler to generate, optimize and simulate MBQC measurement patterns,' (Sunami and Fukushima 2022).

. Chapter 3 will include a closer appraisal of these analytical tools.

[3]fusion-based quantum computing (Zhang et al. 2023; Bartolucci et al. 2023) is an exception to this order, albeit qualified by the technology being,

- grounded exclusively in photonic systems,
- distinguished by fusion gates, which are intrinsically non-deterministic; and
- not publicly accessible.

2.3 Classical compiling

Programs that translate human-friendly ('high-level') programming languages to the low-level instruction sets required by classical computers are a well-established feature of the classical computing landscape. This family of programs, which includes compilers, serves as the blueprint for a similar translating program in a quantum computing facility. There are a number of aspects to the quantum computing tool kit as at December 2024 that make simply adapting a classical compiler of choice a not-straightforward task. Limitations imposed by hardware and the open question of designing a combined classical-quantum architecture, both considered below, militate against a carbon copy of a classical compiler into a quantum computing environment. Despite these problems, a classical compiler remains a starting point for designing a cluster-state compiler.

The purpose of a compiler is to simplify the act of operating a central processing unit of the (classical) computer. This statement prompts further questions: what is the role of a central processing unit, within (classical) computer architecture, that makes a compiler necessary in the first place? If a compiler's purpose is to be an interface between human and central processing unit, for which entity is it simplifying the interaction? Which specific properties of a central processing unit make operating it so complex that it requires a compiler? Beginning with its role, a central processing unit, commonly shortened to processor or CPU, is hardware that conducts the actual computing of any task submitted to a (classical) computer. A processor is often likened to the 'brain' of the computer, fetching instructions from memory then carrying out those instructions (Tanenbaum and Bos 2015). In general, a CPU has set operations it can perform, which include arithmetic/logic operations and fetch-decode-coordinate operations; further, a processor holds a number of registers for the temporary storage and fast recall of data fetched from memory for the purposes of fulfilling these operations. The 'bitness' rating of a processor, for example the 64-bits of

an Intel Core i9 CPU, indicates the ceiling to bits that it can manipulate in any single operation. While the duties or purpose of processors are comparatively standardised, it is important to recognise that CPU architecture is not universal in design: manufacturers set their own criteria for what makes their processors computationally efficient. Perhaps inevitably, the instruction set available for a CPU from manufacturer X may not be the same as those for a CPU from manufacturer Y. If the instruction set recognised by the first CPU does happen to align with that recognised by the second CPU, there is still no guarantee of the two CPUs executing instructions in the same way.

Activating any of a processor's operations requires machine code, a form of binary code. Most humans consider writing instructions in binary to be prohibitive in complexity, even for a trivial computation. Machine code is exact in its application; for example, a single string of CPU instructions might,

i. Confirm status of instruction, conditional or unconditional,

ii. If unconditional, is the value required for this step located (a) in this instruction or (b) in a register?,

iii. If the required operation is ii.(a), jump to the relevant bits in this instruction ELSE if ii.(b), identify the required register;

iv. If there is a target register, go to another part of the instruction to retrieve the address...

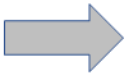
and so forth, with each successive string of CPU instructions exhibiting similarly precise directives. An example of the machine code required by an Intel Core i5 CPU to output the string 'hello world!' to the display appears as Figure 2.5, below. It all but invites error for most humans to work at such incremental levels of operations, in a non-intuitive language like binary, at least in the short term.

The compilation strategies that began appearing in the mid-1950s were solutions to the problem of:

```
#include <stdio.h>

int main (void){
   printf("hello world!\n");

   return 0;
}
```



```
00000000: 00110001 11101101 01001001 10001001 11010001 01011110  1.I..^
00000006: 01001000 10001001 11100010 01001000 10000011 11100100  H..H..
0000000c: 11110000 01010000 01010100 01001001 11000111 11000000  .PTI..
00000012: 10010000 00010001 01000000 00000000 01001000 11000111  ..@.H.
00000018: 11000001 00110000 00010001 01000000 00000000 01001000  .0.@.H
0000001e: 11000111 11000111 00010110 00010001 01000000 00000000  ....@.
00000024: 11111111 00010101 01111110 00101111 00000000 00000000  ..~/..
0000002a: 11110100 00001111 00011111 01000100 00000000 00000000  ...D..
00000030: 01001000 10001101 00111101 10111001 00101111 00000000  H.=./.
00000036: 00000000 01001000 10001101 00000101 10110010 00101111  .H.../
0000003c: 00000000 00000000 01001000 00111001 11111000 01110100  ..H9.t
00000042: 00010101 01001000 10001011 00000101 01010110 00101111  .H..V/
00000048: 00000000 00000000 01001000 10000101 11000000 01110100  ..H..t
0000004e: 00001001 11111111 11100000 00001111 00011111 10000000  ......
00000054: 00000000 00000000 00000000 00000000 11000011 00001111  ......
0000005a: 00011111 10000000 00000000 00000000 00000000 00000000  ......
00000060: 01001000 10001101 00111101 10001001 00101111 00000000  H.=./.
00000066: 00000000 01001000 10001101 00110101 10000010 00101111  .H.5./
0000006c: 00000000 00000000 01001000 00101001 11111110 01001000  ..H).H
00000072: 10001001 11110000 01001000 11000001 11101110 00111111  ..H..?
00000078: 01001000 11000001 11111000 00000011 01001000 00000001  H...H.
0000007e: 11000110 01001000 11010001 11111110 01110100 00010100  .H..t.
00000084: 01001000 10001011 00000101 00101101 00101111 00000000  H..-/.
0000008a: 00000000 01001000 10000101 11000000 01110100 00001000  .H..t.
00000090: 11111111 11100000 01100110 00001111 00011111 01000100  ..f..D
00000096: 00000000 00000000 11000011 00001111 00011111 10000000  ......
0000009c: 00000000 00000000 00000000 00000000 11110011 00001111  ......
000000a2: 00011110 11111010 10000000 00111101 01000101 00101111  ...=E/
000000a8: 00000000 00000000 00000000 01110101 00010011 01010101  ...u.U
000000ae: 01001000 10001001 11100101 11101000 01111010 11111111  H...z.
000000b4: 11111111 11111111 11000110 00000101 00110011 00101111  ....3/
000000ba: 00000000 00000000 00000001 01011101 11000011 10010000  ...]..
000000c0: 11000011 01100110 01100110 00101110 00001111 00011111  .ff...
000000c6: 10000100 00000000 00000000 00000000 00000000 00000000  ......
000000cc: 00001111 00011111 01000000 00000000 11110011 00001111  ..@...
000000d2: 00011110 11111010 11101011 10001010 01010101 01001000  ....UH
000000d8: 10001001 11100101 10111111 00000100 00100000 01000000  .... @
000000de: 00000000 11101000 00001100 11111111 11111111 11111111  ......
000000e4: 10111000 00000000 00000000 00000000 00000000 01011101  .....]
000000ea: 11000011 00001111 00011111 01000100 00000000 00000000  ...D..
000000f0: 01000001 01010111 01001100 10001101 00111101 10110111  AWL.=.
000000f6: 00101100 00000000 00000000 01000001 01010110 01001001  ,..AVI
000000fc: 10001001 11010110 01000001 01010101 01001001 10001001  ..AUI.
00000102: 11110101 01000001 01010100 01000001 10001001 11111100  .ATA..
00000108: 01010101 01001000 10001101 00101101 10101000 00101100  UH.-.,
0000010e: 00000000 00000000 01010011 01001100 00101001 11111101  ..SL).
00000114: 01001000 10000011 11101100 00001000 11101000 10100011  H.....|
0000011a: 11111110 11111111 11111111 01001000 11000001 11111101  ...H..
00000120: 00000011 01110100 00011011 00110001 11011011 00001111  .t.1..
00000126: 00011111 00000000 01001100 10001001 11110010 01001100  ..L..L
0000012c: 10001001 11101110 01000100 10001001 11100111 01000001  ..D..A
00000132: 11111111 00010100 11011111 01001000 10000011 11000011  ...H..
00000138: 00000001 01001000 00111001 11011101 01110101 11101010  .H9.u.
0000013e: 01001000 10000011 11000100 00001000 01011011 01011101  H...[]
00000144: 01000001 01011100 01000001 01011101 01000001 01011110  A\A]A^
0000014a: 01000001 01011111 11000011 00001111 00011111 00000000  A_....
00000150: 11000011                                               .
```

Figure 2.5: (author generated) C syntax to write the string 'hello world' to the display, at left contrasted with the machine code for an Intel Core i5 CPU to execute that computation, at right.

1. supplying instructions to the processor, in machine code and at a suitable scale, through,

2. a medium that is intelligible to humans and reduces the risk of errors of commission or omission.

A solution, therefore, was to create an interface between human and processor. In synopsis, an interface - the compilation strategy – provides:

(I) a *set* of predominantly text-based operators and data structures, expressly intended for a human user and for the purpose of com-

posing instructions to the processor. The set of operators and data structures is more commonly known as a programming language. Each element of the set is native to and defined by the compiler. Moreover, the operators and data structures are cast at a high level of abstraction, essentially to group the common, low-level operations intrinsic to a processor. The set's target audience and high-level abstraction will invariably involve a trade-off between such design criteria as brevity, expressiveness and tractability (e.g. Brachman, Levesque, and Pagnucco 2004; Halpin and Morgan 2008),

(II) quality assurance of the input algorithm, including checks of syntax (e.g. operator or structure is an element of the set) and semantics (e.g. the syntax is organised correctly) to catch so-called compile-time errors and some runtime errors. Note, quality assurance is extensive but has practical limits: no compilation strategy can pick up every possible compile- or runtime error;

(III) One or more optimisation procedures applicable specifically to the data structures of (I), above. These procedures optimise the time- and space dimensions of the computation ultimately passed to the processor as machine code;

(IV) machine code output matched to the processor's instruction set. A compilation strategy may or may not interface with the processor directly and in real time.

Many interfaces, including compilers, interpreters or bytecode through a virtual machine ('VM'), fulfil these same provisions. A compiler is complex in design but as compared to many interpreters or VMs,

- requires fewer working parts; and
- permits fine control of processor operations, with provisions for allocating and managing (main) memory operations.

For these reasons, a compiler is the exclusive focus of this dissertation, as the model for an interface between human and quantum processor. As for the question above regarding which of the human user or processor has its interaction with the other simplified, the compiler creates a virtuous circle. Obviously a human user has a 'natural' entry point to specifying the computational task without the need for any change to formatting of input as received by the processor. Further reflection also makes apparent the quality assurance function of a compiler and the discipline it imposes upon a programmer, bolstering the efficiency of computing, which, of course, directly involves the CPU.

The convenience brought by a compiler comes with a caveat: a compiler outputs machine code that is specific to the host processor, which means executable machine code is not portable between computers. Put plainly, executable machine code compiled against processor X cannot practically be imported then run on a computer with processor Y. This is a direct result of the variability of design among processors and their respective instruction sets. Source code, on the other hand, can be imported and compiled for processor Y, as long as the necessary compiler is installed on the target computer. The non-portability of executable machine code applies even to levels of abstraction as low as assembly language.

The literature on compilers commonly breaks down their inner workings into phases. Cooper and Torczon (2012) use three-phases abstraction of a compiler, reproduced as Figure 2.6, below, consisting of Front- and Back End phases separated by an Optimisation phase. The purpose of each phase summarises as,

- Front end: enforces the programming language's grammar to check program syntax is well-formed then 'lexing' and parsing the program to an intermediate representation (see below).
- Back end: map the optimised intermediate representation to the instructions set of the processor. A compiler's use case ends at the CPU.

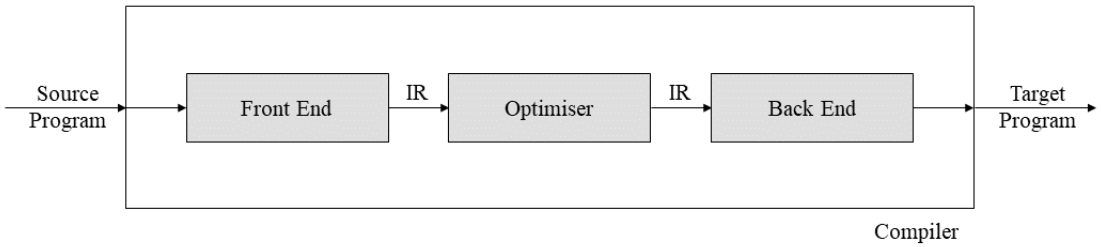


Figure 2.6: Abstracting a classical compiler into phases (after untitled figure, Cooper and Torczon 2012: p. 8).

- Optimisation: improve spatial/temporal efficiency of the intermediate representation while *losing none of the meaning of the input algorithm*. This is a subtle point with implications for architectures predicated upon multiple processors.

The data structure known as an intermediate representation ('IR') is the factor common to each of these phases. A program that is syntactically and semantically compliant with the grammar of a programming language is recast to an IR, as syntax that will ultimately map to the processor's instructions set. To repeat, the high-level syntax of classical programming languages are meant for the human operator, not the CPU. This is not the same as stating that programming language and IR are unrelated as clearly all operations permissible under a programming language must be workable for the processor and therefore, any intermediate representation of the procedure. Balancing the operations of the input programming language and viable operations of the processor as they do, intermediate representations take different forms that group loosely as,

- tree-based (e.g. Abstract Syntax Tree or 'AST'),
- graph-based (e.g. Dependence Graph),
- linear, essentially assembly language executed imperatively; and
- hybrid, a mixture of the above.

A modern compiler may incorporate more than one IR component. Chapter 4 of this dissertation includes a worked example of deriving a graph-based IR as applied to a quantum compiler.

An IR undergoes an iterative procedure of analysis-optimisation-trans- lation to arrive at a data structure that minimises consumption of both time and space (i.e. bits, of main memory) in processor operations. Each such iteration, termed a *pass* cannot result in information loss from the original input program. The role of an operating system in facilitating a pass comes to the fore at this point. There can be no doubt that an operating system greatly enhances the doing of computation. In most cases, machine code targeting the processor is *augmented* with operating system routines, among others (Fischer, Cytron, and LeBlanc 2010). It is possible though onerous, to write a program at a low enough level (i.e. *pure* machine code, assembly code) to operate a CPU directly thus bypassing an operating system if not, dispensing with it altogether. In this scenario, the program must account for tasks ordinarily covered by an operating system, such as managing memory and calling device drivers. The converse of writing an algorithm for a machine with an operating system but no CPU is an unviable proposition.

It is timely to consider the role of algorithms in computing. An *algorithm* is the logic to resolve a computational problem, built from a language's data structures and organised with its control structures (Rabhi and Lapalme 1999). In other words, an algorithm is the organising principle to source code, the high-level syntax of a computer program. Drawing this distinction is, necessarily, arbitrary because code without an algorithm may be syntactically correct yet nonsensical while algorithm without code, at any level of abstraction, is impossible. Further to the restriction on IR transformations not involving information loss, the compiler serves only to check, optimise-transform and translate the algorithm it receives. Defining an algorithm occurs externally to a compiler. This separation of roles in algorithm design in turn raises the question of algorithm efficiency. 'Efficiency' of a solution and the

complexity of a computational problem are both quantified with the metrics of time and space. There can be multiple algorithms to solve a particular computation, which can be ranked by the time taken and the space consumed to reach the (correct) answer. Further, whether a classical algorithm exists to solve it in polynomial time/space is the basis of attributing complexity to a computational problem (see, Aaronson 2013). To come full circle, a compiler optimises code to minimise consumption of both space and time in processor operations but it is the programmer who determines efficiency in an algorithm.

2.4 Quantum computation in 2024

To be explicit with terms identified in Section 1.1 above, the term 'quantum computing facility' shall reference a hypothetical construct that nevertheless *excludes* the NISQ computer. NISQ computers, as already discussed, are the existing proofs-of-concept of quantum computing; in contrast, the criteria of a quantum computing facility shall include:

1. fault tolerant: rate of error occurrence and pace of error correction make a polylogarithmic impact on computation time and space,
2. technology neutral: can be trapped ion, photonics, silicon etc. but not annealing,
3. paradigm neutral: the architecture accommodates input in circuit or cluster-state format,
4. architecture: may integrate with a classical computing facility,
5. resource management routines: bespoke middleware or a (classical) operating system interact with the facility through remote procedure call ('RPC'); all compiler functions would be local to the middleware or OS,

6. quantum processing unit ('QPU'): a QPU is an independent processor, specific to a quantum computing facility and therefore notwithstanding 4., a QPU wholly resolves any computation the compiler passes to the facility.

Repeated references to this hypothetical shall appear in this dissertation.

The landscape of quantum computers, as at December 2024 consists entirely of NISQ computers and is dominated by circuit-model architecture. The number of qubits accessible for a given computation is also limited. Moreover, a clear use case for quantum computers is work-in-progress as is specifying problems within the domain (e.g. Rubin et al. 2023; Chen et al. 2023). The following review of first, NISQ computers as at December 2024 then second, features of a combined classical-quantum computing architecture is context for considering those criteria of the quantum computing facility defined above that are relevant to a quantum compiler. Particular points of focus include the compiler solutions attaching to NISQ computers and the possible points of convergence for the circuit and cluster-state paradigms in QPU design.

2.4.1 NISQ computers

Ranked by QPU (qu-)bitness as at December 2024, the publicly accessible NISQ quantum computers[4] are:

- IBM quantum platform, Heron R2 processor at 156-qubits[5],
- Rigetti Systems Ankaa-3 at 84-qubits (cf. Dupont et al. 2025),

[4]Note, neither of Google's superconductors, 'Willow' (Google 2024); and 'Sycamore' (Google 2023b) at 105- and 72-qubits, respectively, is publicly accessible.

[5]Announced at IBM Developer Conference, held 13-15 November 2024 at Watson Research Centre, Yorktown Heights, New York (URL: `https://learning.quantum.ibm.com/course/qdc2024`).

- Quantinuum Model H2, trapped-ion architecture at 56-qubits (Crippa, Jansen, and Rinaldi 2024; also, Moses et al. 2023),
- IonQ, Forte at 36-qubits (Chen et al. 2024).

In general, access to these services works through contractual agreement or through established third-party providers. NISQ computing service providers mitigate the risk of environmental noise resulting in decoherence but give no assurance of eliminating it.

Most of the NISQ-computer parent companies or their service providers also actively maintain open-source emulators. As noted above, all of the publicly-accessible NISQ platforms will only accept algorithms encoded in circuit format and this applies also to their open source emulator applications. In order:

- IBM offers Qiskit (`https://github.com/Qiskit`),
- Google offers Cirq (`https://github.com/quantumlib/cirq`),
- Rigetti offers the Forest SDK (`https://docs.rigetti.com/qcs/#forest-sdk`), which makes use of the pyquil ecosystem (`https://github.com/orgs/rigetti/repositories`; also, `https://github.com/quil-lang`),
- Quantinuum is a NISQ service provider. Microsoft's Quantum Development Kit (QDK) and its proprietary language, Q# (`https://docs.microsoft.com/en-gb/azure/quantum/install-overview-qdk`) are the access point through Microsoft Azure,
- IonQ also is a NISQ service provider, licensing through Microsoft Azure, Amazon Braket and Google Cloud services or its own cloud service, 'IonQ Quantum Cloud'. Each cloud service provider applies its own restrictions to encoding of algorithms.

Each of these emulators is also an application programming interface ('API') to integrate with the parent company's NISQ computer, which

means the emulators double as proxy compilers; for example, a circuit built through Qiskit is compatible with the Heron processor.

Quantum computing emulators use classical computing hardware to mimic the workings of a quantum computing facility. There is an abundance of quantum emulator libraries or online emulator resources beyond those offered by the major service providers[6]. These resources are either GUI-based or SDK's but at a minimum, they enable the user to manipulate the state of simulated qubits through quantum logic gates then to measure the results. Emulators in general are especially useful as tools for visualising formalisms of circuits (or pulses).

Apart from their (qu-)bitness ceilings and the technology of qubits construction, the publicly available information regarding NISQ processors barely hints at the specifics of their instruction sets. It is possible to reverse-engineer the workings of a NISQ processor from standardised quantum assembly languages (Cross et al. 2017; Khammassi et al. 2018; Cross et al. 2022) but only to a point. Assembly language is a low-level syntax which translates to processor instructions (i.e. machine code) by means of an assembler, for example GNU `as` (`www.gnu.org/software/binutils/`). In effect, quantum assembly languages, like the service providers' SDKs, act as an API to the QPU so specifics of the instructions to compute an arbitrary quantum logic gate, for example, remain guesswork. Similarly, there is a question of the autonomy of a NISQ processor. Classical hardware often works to manage a NISQ computer (e.g. Wack et al. 2021; Bartolucci et al. 2023) including each NISQ computer service listed above and the load balancing of computational routines between CPU and QPU as often as not, goes unspecified. The degree to which a CPU optimises circuit input and more importantly, the nature of optimisation within a combined classical-quantum architecture are relevant when evaluating a NISQ processor.

[6]see `https://qosf.org/project_list/` for an extensive list of quantum computing emulator libraries or online resources

2.4.2 Combined classical-quantum architectures

The consensus is that, for the foreseeable future, classical and quantum computing facilities will combine as an architecture. In the short term, the classical and quantum machines will form a parent-child architecture with the classical machine as parent and the child quantum machine probably as an I/O device. It is harder to predict a classical-quantum architecture of the medium term because of the many difficult and as yet unresolved issues besetting NISQ computers (e.g. mitigating environmental noise). Some form of parallel computing architecture seems to be the inevitable model, in part because classical memory may have to compensate for limited advances in solid quantum memory. Presumably attaining true fault-tolerance in the quantum machine will also be a prerequisite to a parallel architecture. The appeal of a classical-quantum parallel architecture lies not only in the performance gains of parallel processing but also in the potential to combine classical and quantum computational problems in a single algorithm, in contrast to the either-or restrictions on algorithms under a parent-child architecture.

Which options of parallel architecture might work with classical and quantum hardware in the medium term? Until a manufacturer of classical processors expands its production also to include a quantum processor, a CPU and QPU will be heterogeneous. Assuming that processors of classical and quantum machines are heterogeneous effectively rules out a shared-memory multiprocessor architecture. A *distributed memory* architecture (a.k.a., 'multicomputer') that operates within an RPC framework is plausible. Note, a distributed memory architecture is not inconsistent with classical memory acting as an offset to quantum memory but does mean that classical and quantum processors will not directly share memory. An asynchronous request-reply messaging protocol would be the most parsimonious of RPC frameworks for this distributed memory classical-quantum architecture. Briefly, request-reply messaging is similar to a client-server relationship, whereby the server

listens for a message from the client then replies in kind. Request-reply is usually the only message channel between client and server although a client may have channels with many (other) servers and conversely. Further, a synchronous transaction is atomic in that both the request and the reply messages must execute in their entirety before either server or client can begin any subsequent action whereas an asynchronous transaction means a delayed (but not aborted) reply does not block other actions from executing.

A classical-quantum parallel architecture will task the quantum machine with those computational problems of a complexity that remain prohibitive to classical computing[7]. The proposition is a quantum computer will deliver both temporally and spatially efficient solutions to this subclass of computational problems. At a minimum, the classical machine will fulfil decoding, logging and memory management tasks of computations that engage the quantum machine. Of most relevance to this dissertation, a parallel architecture, even of two or more classical machines, bears significantly upon how compiling must work.

While much research has gone into quantum compiler design, in addition to the quantum emulators described above (e.g. Litteken et al. 2020; Sivarajah et al. 2021; Patel, Silver, and Tiwari 2022; Kreppel et al. 2023; Krishnan Vijayan et al. 2024), the practicalities of compiling within a classical-quantum parallel architecture are largely unexplored (Svore et al. 2006; Chong, Franklin, and Martonosi 2017). As just one problem specific to a compiler: the difficulties of realising 'automatic parallelisation' in classical shared-memory multiprocessor architecture will emerge in a classical-quantum RPC architecture. The goal of automatic parallelisation in compiling, which continues to elude computer scientists, is to achieve the performance benefits of parallel processing

[7] In March 2024, Google announced a $5m prize for the entrant who could quantify a use case for a quantum computer (`https://blog.google/technology/research/google-gesda-and-xprize-launch-new-competition-in-quantum-applications/`).

without adding overhead to the developer (e.g. Marlow 2013).

As noted above, a compiler is a receiver of algorithms, which it then optimises by minimising the number of processor operations required to execute the computation. At the same time, the point of a parallel classical-quantum architecture is to permit algorithms that draw upon either or both machines to complete a given computation: the classical machine computes those parts of the algorithm for which it is best suited and likewise, the quantum machine. Therein lies the automatic parallelisation problem peculiar to this architecture: either the developer writes the algorithm at a low level, to allocate tasks between machines and in so doing incurs the overhead of effectively bypassing the compiler; or the architecture requires a bespoke compiler that efficiently allocates tasks to machines, with 'efficient' including the provision to supersede the algorithm as received. Constructing such a compiler would be prohibitively difficult and almost certainly would involve an artificial intelligence component. While such a compiler may compromise the optimisation rule of preserving the meaning of an input algorithm, it certainly contravenes the protocol of receiver-of-algorithms.

An algorithm in circuit- or cluster-state format is at a low level of abstraction compared to the text-based language that front ends a classical compiler. The syntactic expressiveness of a circuit or cluster state approximates that of an intermediate representation associated with classical compilers. This inflexible limit to syntax resolves the automatic parallelisation problem for the classical-quantum architecture insofar as it also limits the possible optimisations for any compiler solution: precisely because circuit- and cluster-state syntax are low-level means they restrict diversity of expression in an algorithm and *ipso facto*, the variability of possible optimisation. As noted before, expressiveness is one criterion among many in the design of a high-level programming language. If the designer is prepared to limit expressiveness to a level equivalent with circuits or cluster states, it remains possible to create a high level programming language that reduces the overhead

placed on a developer, without removing it completely. Of more significance to established compiling protocols, this high-level language also lessens the need for optimisation passes that may overrule an algorithm (cf. Ittah et al. 2022). A path to making a parallel architecture *efficient enough* is to split text-based syntax between commands that are specific to each processor, in other words, create a parallel language of two lexicons but one grammar. A parallel language does mean the developer must explicitly determine how an algorithm divides between machines but not at an onerous low level. The classical lexicon should remain essentially unchanged from contemporary language standards and a quantum lexicon that includes operator precedence and devices of control flow, iteration and memory allocation is theoretically possible. These language features alleviate the overhead of explicitly allocating routines to machine. Striking a balance between high-level language and optimisation of its intermediate representation is more art than science so the practicalities of a language higher than circuit- or cluster-state encoding will be touched upon in Chapters 4 and 6 of this dissertation.

2.5 A cluster-state compiler

Having set out the purpose of a compiler and itemised the various proxy compilers for NISQ computers, each instance of which implements the circuit model exclusively, it remains to outline the requirements for a cluster-state compiler. The remainder of this dissertation and the artefacts arising from it will fulfil these requirements.

Some assumptions apply to the target QPU of this cluster-state compiler namely, the processor will have,

1. access to a number of qubits per clock cycle at least comparable with that of a publicly-accessible NISQ computer,

2. an instructions set that includes but is not restricted to operations for entangling qubits, preparing qubit state and incremental measuring of a qubit's state,

3. I/O access to classical memory.

The point to make explicit is that a processor meeting these criteria could work on a NISQ computer.

The front-end language of a cluster-state compiler will be graph-based. A detailed review of graphs and graph states appears in Chapter 3 of this dissertation but for present purposes, the compiler will accept only simple graphs as *expressions of graph states*. All such graphs comprise of no fewer than one vertex, which may connect to another vertex by a single edge to represent an entanglement interaction. This restriction on input format is not contrary to the feasibility of a high-level language to apply the cluster-state model to a parallel architecture, as posited above. A graph-based compiler is instead a proof-of-concept for a larger project, the logical culmination of which would be fabricating the hypothetical QPU defined above.

As noted above, the Clifford group operators plus the non-Clifford T operator constitutes a universal quantum gates set. Within the setting of the quantum computing facility defined in Section 2.4, which is a fault-tolerant and error-correcting facility, transforming a graph with a T operation imposes an extra resourcing cost of the noiseless qubits needed for the procedure known as magic state distillation (Bravyi and Kitaev 2005). Briefly, preparing a number of ancillary magic state qubits with an equal number of noiseless qubits works to encode ('distil') one qubit in the required T state, at a low probability of error. Each instance of a T operator in an algorithm therefore increases the size of the graph state by the extra qubits required to distil magic state (Raußendorf, Harrington, and Goyal 2007). The input algorithm must account for all qubits necessary to the computation. This is a question considered in more depth in Chapter 4 of this dissertation.

A graph input format constrains possible optimisation of the input algorithm to a manageable set. In most cases, those optimisation schemes proposed for a quantum compiler are methods to minimise the number of qubits required for a computation in order to lessen exposure to indefinite coherence times (e.g. Nam et al. 2018; Sung et al. 2020; Amaro et al. 2022; Hietala et al. 2023; Liu et al. 2023). Such schemes overwhelmingly represent a circuit model compiler. Cluster states too are sensitive to decoherence but unlike a circuit, a cluster state is *composable* making it possible to assemble an n-qubits cluster by increment, in other words composing the required graph from subgraphs (Browne and Briegel 2006). Of course, optimising the size of each subgraph in an effort to reduce the impact of decoherence on the graph overall is a non-trivial procedure. The practicalities of optimising graph states will feature in a worked example in Chapter 4 of this dissertation.

The lack of a QPU in production that can natively implement the cluster state model means output of the compiler will be OpenQASM assembly language. Canonically, compiler and assembler are part of a toolchain in which the latter receives the assembly code output of the former to then create the object or machine code for the processor (e.g. Fischer, Cytron, and LeBlanc 2010). The aforementioned GNU `as` assembler places within this toolchain model. A linker to access any libraries called in the original source code also often forms part of this toolchain. A benefit of OpenQASM output from the compiler is the NISQ emulators listed above will accept input in this format, which in turn permits 'testing' of a graph-based algorithm on the NISQ service. Of course, it is debatable whether such an approach qualifies as a test of the cluster-state model given its significant divergence from the computational means of circuit model processors of a NISQ computer.

2.6 Summary

A general introduction to the principles of quantum computing was the foundation for a review of the circuit- and cluster-state models. Reviewing the cluster-state model was also the first step in the case for the tūQ toolchain. Computing through a cluster-state works by measuring single qubits in a specific order and basis and relies upon propagating the desired state though the cluster by means of both entanglement and quantum teleportation. This model differs enough from the gate-based operations of the circuit model to warrant a bespoke compiler. Analysis of the purpose of a compiler in a classical computing framework and how a compiler fulfils that purpose set the background for a review of the quantum computing landscape as at December 2024.

An architecture to process algorithms encoded in circuit format is the common defining feature of publicly-accessible NISQ computers, including their proprietary quantum emulators. Integrating with a circuit-based architecture also informs the design of the many standalone emulator projects available as SDKs or hosted services. A practical capacity to operate a QPU may be a sufficient design requirement for contemporary NISQ computers but a quantum computing facility of the future will work in parallel with a classical computing facility, which will make new demands on compiler technology. A high-level programming language to operate a classical-quantum parallel architecture may require two lexicons and one grammar, a potential obsolescence and seldom addressed in current compiler offerings. Under such circumstances, a new compiler model promoting an alternative paradigm of quantum computation seems timely. To that end, a list of requirements for a cluster-state compiler rounds out the chapter.

In closing, the following points raised in this chapter are pivotal to the dissertation as a whole:

1. A cluster state, $|\Phi\rangle$ is equivalent to a graph state, $|G\rangle$ thus, $|\Phi\rangle \equiv |G\rangle$;

2. circuit-based and measurement-based quantum algorithms are approximately as expressive as intermediate representations used in classical compliers;
3. a graph state is the format of both algorithm input and quantum intermediate representation (QIR);
4. the developer determines efficiency in an algorithm whereas a compiler optimises the received algorithm;
5. a parallel classical-quantum architecture must bear upon the design of a compiler.

Chapter 3

Modelling graph states with tūQ

The focus of this chapter and the two that follow is the executable `tūQ` (`https://github.com/QSI-BAQS/tuQ`), an application for modelling, simulating and compiling graph states. This chapter is a discussion of `tūQ` Modeller, the toolchain component for modelling cluster states through the use of (simple) graphs. The so-called 'substrate' of highly entangled qubits arranged as a two-dimensional lattice, which is central to the MBQC model, may be assembled by increments (e.g. Raußendorf and Briegel 2001), making a clear use case for a modelling tool. A designer of algorithms equipped with such a tool can readily expand or shrink a graph as well as consider any subgraph in finer detail. The value of such modelling is self-evident as a computation based on a cluster state becomes more complex. A more ambitious use case to realise an entire MBQC architecture is founded upon basic graph-state modelling functionality; chapters 4 and 5 then are case studies of `tūQ`'s Simulator mode and functions to integrate Modeller-Simulator in workflow, respectively.

The rationale for a modelling component within the `tūQ` toolchain is to minimise the number of qubits required for an algorithm either to be simulated or compiled. `tūQ` Modeller and Simulator then have an iter-

ative relationship in which Modeller is the staging point for a first cut of any algorithm that is then passed through Simulator to evaluate by its computational output. The following introduction to Modeller as the first component of the tūQ toolchain begins by examining the connection between cluster states and graphs, including a review of the latter's nomenclature. This overview then leads to the concept of a graph state, a data structure that is closely connected with cluster states. The chapter closes with a worked example of drawing then reducing the 15-qubits CNOT pattern with tūQ Modeller.

Original material

- The source code for Q2Graph may be inspected at URL: `https://github.com/QSI-BAQS/Q2Graph`; the source code for tūQ Modeller may be inspected at URL: `https://github.com/QSI-BAQS/tuQ`. References to both applications appear in this chapter.
- The majority of Section 3.1, part of Section 3.2 and all of Appendix B represents original work that appears in Bowen and Devitt (2022).
- A supplementary video for this introduction to tūQ Modeller entitled,
  - A Toolchain for Cluster-State Architecture Modeller.

  References to specific passages in the video will appear in this chapter. Citation will be of the form 'Modeller video' and may include a starting point in format Minute.Second, as required.
- This chapter gives further context to tūQ Modeller with a worked example of the 15-qubits CNOT gate as proposed by Raußendorf and Briegel (2002).

3.1 Graphs and graph states

Chapter 2 of this dissertation included a detailed introduction to the cluster state but a brief evaluation of the cluster-state's equivalent, the graph state. Any cluster state is fundamentally a multipartite system, characterised by *interactions* of quantum systems. *Syntactically* the graph notation lends itself to this characterisation while *semantically* the abstraction of graph state captures the point-in-time identity and transformation of state in the cluster; it also accommodates the abstraction of the stabiliser state, which is significant in reasoning about algorithms based on cluster-state computation. The Modeller mode of the tūQ toolchain is an interactive sandpit environment for the user to draw or transform graphs and therefore graph-states in an effort to gain insights of how a computational process acting through a cluster state would work. As a tool, tūQ Modeller has clear applications to drafting quantum algorithms or for reasoning about the engineering complexity involved in constructing cluster-state computing architecture. A discussion of the workings and nomenclature of graphs lays the groundwork for analysis of graph states, both of which in turn inform the requirements for tūQ Modeller.

3.1.1 Graphs

Chapter 2 included an introduction to the cluster state with particular focus upon $QC_{\mathcal{C}}$, which is fundamental to any MBQC architecture. A two-dimensional, nearest-neighbour graph, specifically a *simple* graph, is the medium used to model the computational properties of $QC_{\mathcal{C}}$. An understanding of the workings and nomenclature of graphs is the first step to an understanding of graph states.

To rehash the summary of graphs appearing in Section 2.2.2, above, a simple graph, G is an ordered pair consisting of a non-empty, finite set of vertices, $V(G)$ and a finite set of edges, $E(G)$. An edge signifies an interaction between vertices and in a simple graph,

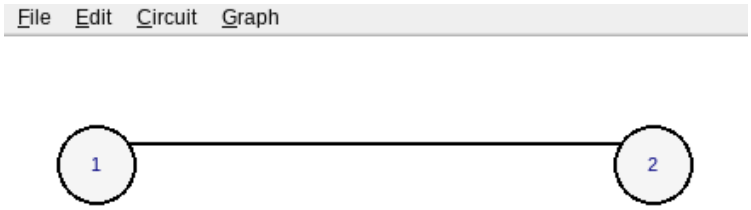


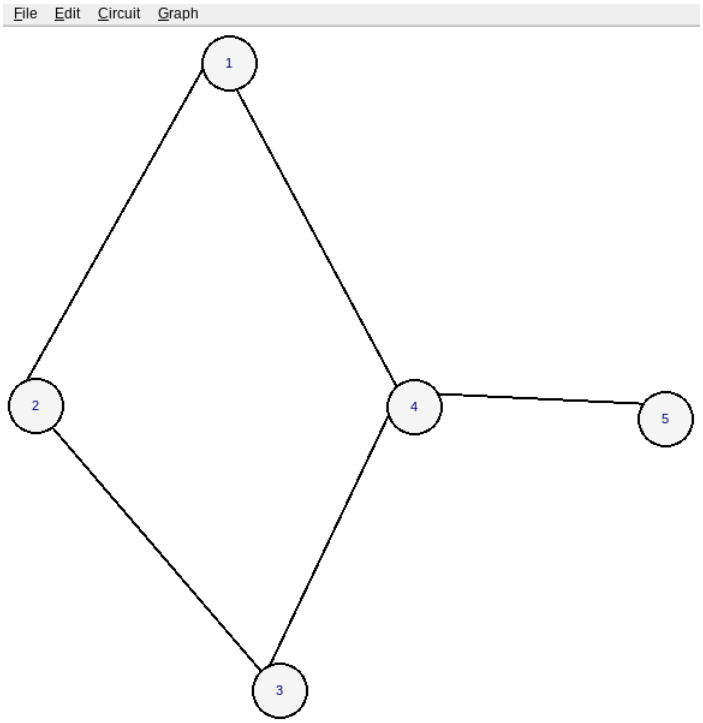
Figure 3.1: A graph of two adjacent vertices.

1. an edge 'self-joining' a vertex; and
2. more than one edge joining vertex *i* to vertex *j*.

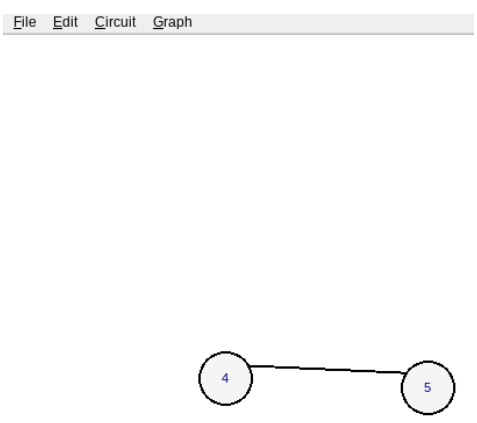
is prohibited. Recall that a graph state is also equivalent to a stabiliser state and that according to the MBQC model, it must remain a stabiliser state through every measurement (see Section 2.2.2, above). A graph that violates either or both of these restrictions is not a graph state. As is consistent with $QC_{\mathcal{C}}$, a vertex is a qubit prepared in state $|+\rangle$ and an edge is a CZ gate between qubits. The Modeller video opens with a demonstration of laying out a simple graph.

When arbitrary vertices a and b are linked by an edge ab as per Figure 3.1, they are deemed *adjacent*. The *degree* of a vertex is the number of edges joined to it and therefore, the number of vertices with which it interacts; for example, each of vertex 1 and vertex 2 of Figure 3.1 has a degree of 1. Any and all vertices adjacent to arbitrary vertex, a form a subgraph of G, termed the *neighbourhood* of vertex a and denoted as N_a. Removing arbitrary vertex a from G at once removes all edges connected to a and therefore all of its adjacencies to create a new graph, denoted as $G \setminus a$; similarly, removing a subset of vertices, U is denoted $G \setminus U$; an example of these transformations of G appears as Figure 3.2, below.

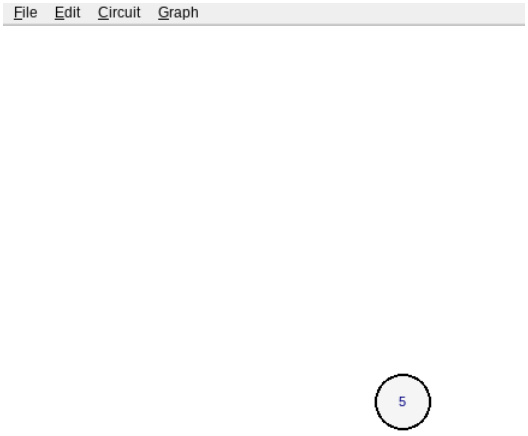
The *complement* of a simple graph, G of vertex set $V(G)$, is itself a simple graph, G' of vertex set $V(G)$ but with inverted edges. In other words, if vertices b and c are neighbours of vertex a in G but b and c are not adjacent to each other then, they are adjacent in G'; Figure 3.3 is an



(a) A five-vertices graph, G.

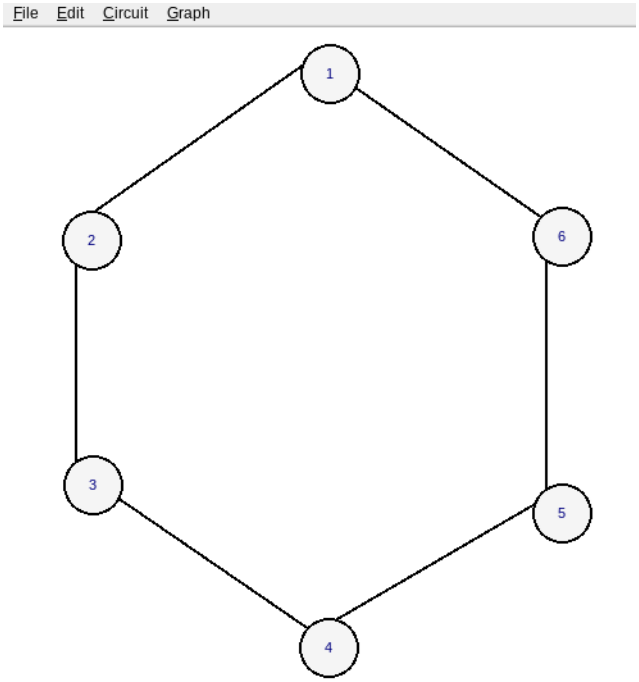


(b) Removing vertex 2 from G, removes all edges of vertex 2 and all of its adjacencies to create graph, $G \setminus 2$.

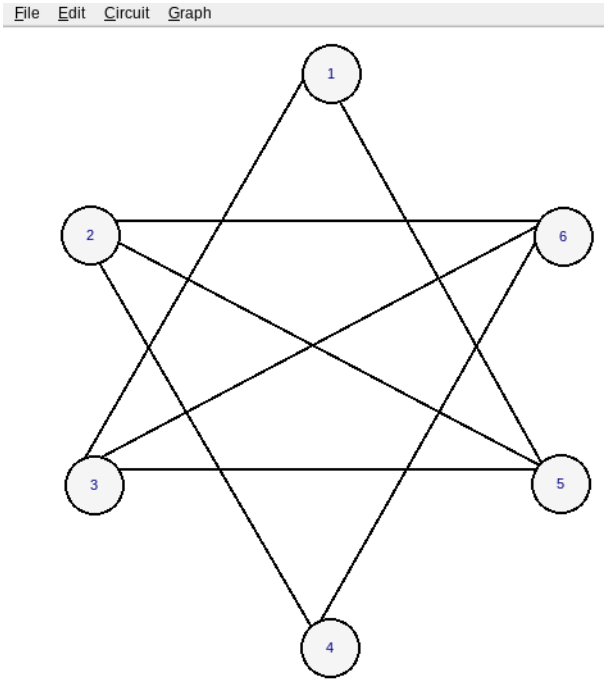


(c) Removing vertices 1 and 2 as a subset, U creates graph, $G \setminus U$.

Figure 3.2: Transforming a graph, G, by removing an arbitrary vertex $a \rightarrow G \setminus a$; or a subset of vertices $U \rightarrow G \setminus U$.



(a) A graph of six vertices, each with two neighbour vertices.



(b) The complement of the graph of Figure 3.3a is itself a simple graph with the sites of vertices unchanged but the edges inverted.

Figure 3.3: A graph of six vertices and its complement.

illustration of a six-vertices graph and its complement. It is also possible to restrict a complement to a neighbourhood of G, leaving the rest of G unaffected. This transformation, as appears in Figure 3.4, below, is termed 'local complementation' and appears frequently in modelling of graph states (Van den Nest, Dehaene, and De Moor 2004a; Adcock et al. 2020).

3.1.2 Graph states

The graph state, $|G\rangle$ has received significant attention as a modelling framework (Van den Nest, Dehaene, and De Moor 2004a; Hein, Eisert, and Briegel 2004; Anders and Briegel 2006; Hein et al. 2006; Briegel et al. 2009; Dahlberg, Helsen, and Wehner 2020). As the name suggests, a graph state is a (simple) graph representation of a quantum system. It is important to be clear in regard to this nexus between graph and quantum system. Graphs are rule-based abstractions of the interactions between arbitrary entities. Similarly, the qubit is both a physical system and a mathematical object derived from properties of that system (see Section 2.1, above; for broader discussion, Mermin (2007) or Nielsen and Chuang (2010)). The abstractions 'graph' and 'qubit' conform with compatible rules hence the concordance of graph and (quantum) state.

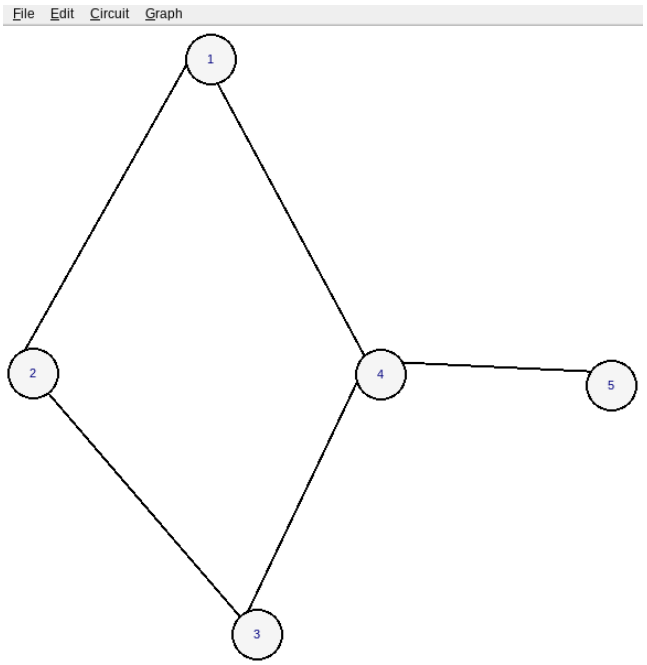
Local complementation ('LC') of Figure 3.4 is a component of $\mathcal{P}^N$ operations on a graph state and therefore underpins $|G\rangle$ as a stabiliser state. After equation (2.24), above, the root of the stabiliser on vertex i,

$$\sqrt{K^{(i)}} = \sqrt{\sigma_x^{(i)}} \bigotimes_{j \in n^i} \sqrt{\sigma_z^{(j)}}, \tag{3.1}$$

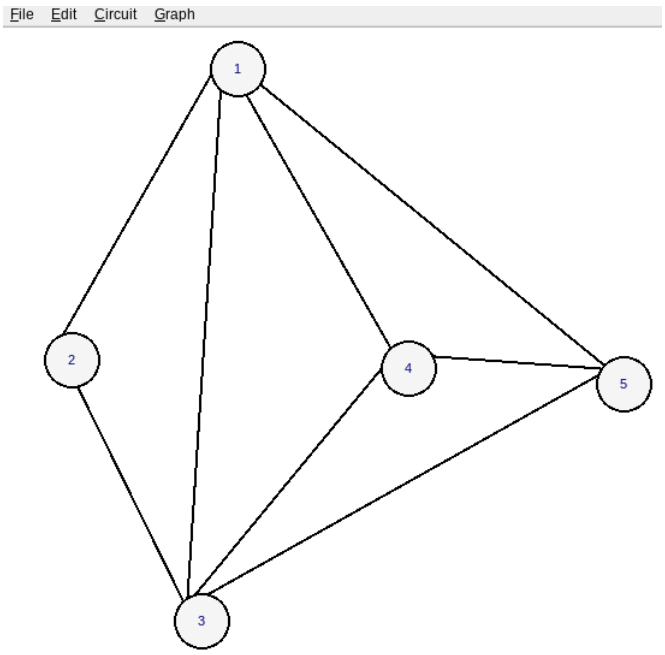
where n^i denotes the neighbourhood of vertex i, corresponds directly with local complementation on vertex i,

$$\sqrt{K^{(i)}}\,|G\rangle \equiv |\tau_i(G)\rangle\,. \tag{3.2}$$

where τ_i, denotes local complementation. Hein *et al.* (2006, proposition 5) show that iff $|G'\rangle$ is $|G\rangle$ after at least one local complementation



(a) A five-vertices graph.



(b) Local complementation of the neighbourhood of vertex 4 of Figure 3.4a.

Figure 3.4: A five-vertices graph, G and local complementation of the neighbourhood of vertex 4. Note how $V(G)$ and the degree of non-adjacent vertex, 2 are unaffected by local complementation

operation, $|G\rangle$ and $|G'\rangle$ are termed *LC-equivalent*. The significance of LC-equivalence lies in the fact that it guarantees continuity of state-space through successive measurements of $|G\rangle$ (Schlingemann 2004; Anders and Briegel 2006). As shall be shown, a graph-state modelled in `tūQ` also has continuity of state-space through one or more transformations[1]. Note, the term 'LC-equivalent' is occasionally shortened to 'local equivalence' (e.g. Claudet and Perdrix 2024) but context is important in interpreting such abbreviation. Two graph states are also equivalent through corresponding equivalence of state vectors (Hein, Eisert, and Briegel 2004). Such local-unitary equivalence ('LU-equivalence') obtains if there exists a local unitary, U such that,

$$|G\rangle = U\,|G'\rangle\,, \tag{3.3}$$

although LU-equivalence, unlike LC-equivalence need not map a graph to a graph: if G' of $U\,|G'\rangle$ is a graph then, G is a graph but not conversely. Locality in the case of LU-equivalence refers to the systems associated with vertices of $G = (V, E)$ and $G' = (V, E')$, for example

$$U = U_1 \otimes U_2 \otimes U_3 \otimes U_4, \tag{3.4}$$

where U_i act on each vertex independently. With specific reference to discussion of adjacency matrices in Section 2.2.2, above, if the adjacency matrices of G and G' are identical (i.e. $G \equiv G'$) then, LU-equivalence holds only if $U \equiv I$, where I is the identity matrix (cf. Van den Nest, Dehaene, and De Moor 2004a). Necessarily then, identity of the adjacency matrices of G and G' does not imply LC-equivalence of G and G'. Note, it is also not necessary that LU-equivalence between $|G\rangle$ and $|G'\rangle$ makes the graph-states isomorphic (i.e. $|G\rangle \cong |G'\rangle$), where graph-state isomorphism is a bijection, $f : V1 \to V2$, between vertices

[1]Local complementation may also be indispensable to modelling linear-optical (Browne and Rudolph 2005; Kieling, Rudolph, and Eisert 2007) or fusion-based (Bartolucci et al. 2023) computing architectures, both of which derive from MBQC principles.

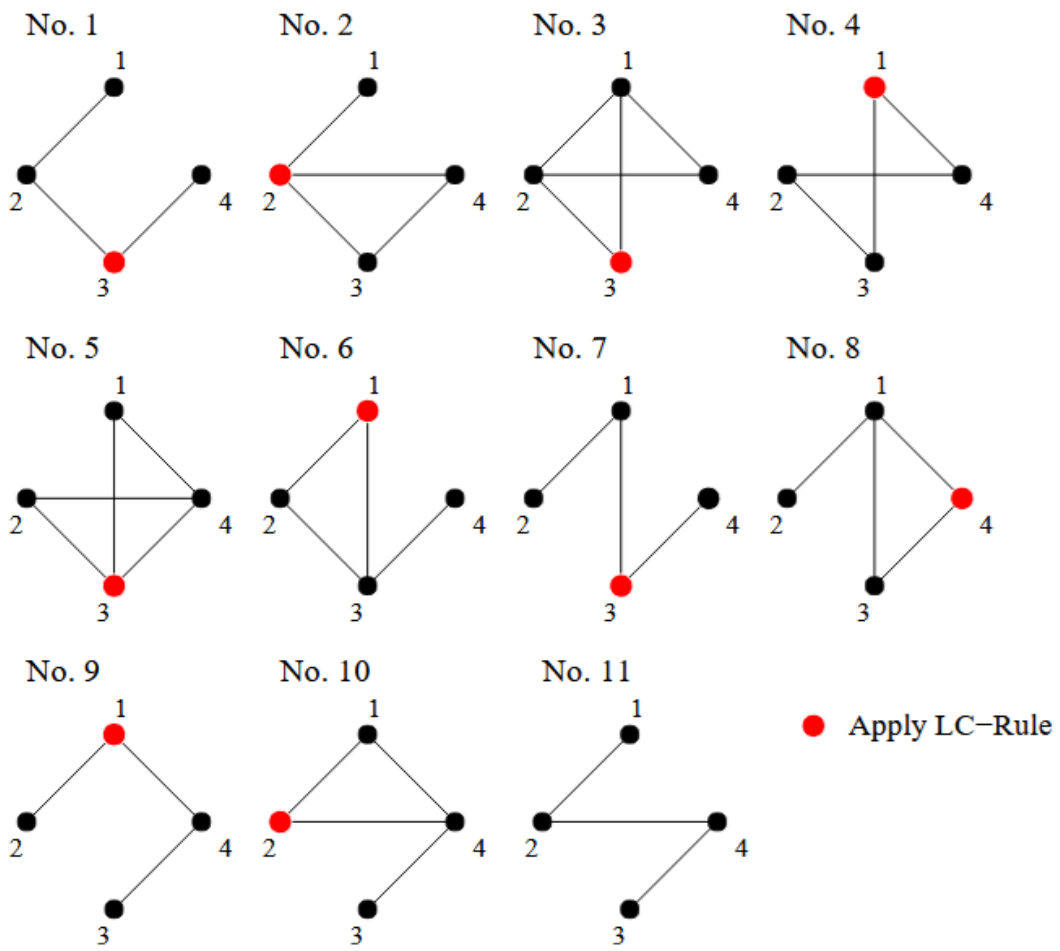


Figure 3.5: Successive applications of local complementation to G as a demonstration that $G \cong G'$ is not a necessary condition of LC-equivalence, **source**: Hein *et al.* (2006), Figure 4.

of two graphs, G_1 and G_2 such that $\{a, b\}$ is an edge in G_1 iff $\{f(a), f(b)\}$ is an edge in G_2. Furthermore with reference to Figure 3.5, isomorphism of $|G\rangle$ and $|G'\rangle$ is not a necessary condition of LC-equivalence. More shall be made of LC-equivalence in relation to optimising $|G\rangle$ in Chapter 4 of this dissertation.

Pauli measurements in the σ_x, σ_y or σ_z basis, along with LC unitaries, are the mechanism for transforming $QC_{\mathcal{C}}$. After Hein *et al.* (2006, proposition 7), measuring the qubit represented by vertex a in basis σ_x, σ_y or

σ_z will transform $|G\rangle$ to $|G'\rangle$ on the remaining vertices thus,

$$\begin{aligned} Z : P^a_{z,\pm} |G\rangle &= \frac{1}{\sqrt{2}} |z,\pm\rangle^a \otimes U^a_{z,\pm} |G - a\rangle \,, \\ Y : P^a_{y,\pm} |G\rangle &= \frac{1}{\sqrt{2}} |y,\pm\rangle^a \otimes U^a_{y,\pm} |\tau_a(G) - a\rangle \,, \\ X : P^a_{x,\pm} |G\rangle &= \frac{1}{\sqrt{2}} |x,\pm\rangle^a \otimes U^a_{x,\pm} |\tau_{b_0}(\tau_a \circ \tau_{b_0}(G) - a)\rangle \,, \end{aligned} \tag{3.5}$$

where P^a signifies the projection of vertex a into the designated $_{x/y/z}$ basis and τ_a designates local complementation of vertex a's neighbourhood. By way of further clarifying the effect of a measurement on G and where N_a denotes the neighbourhood of vertex a,

- Z: delete vertex a from G,
- Y: invert $G[N_a]$ and delete vertex a from G,
- X: choosing any $b_0 \in N_a$, invert $G[N_{b_0}]$, applying the rule for Y then, invert $\tilde{G}[N_{b_0}]$ again.

A preview of the $Z/Y/X$ standard operators and of local complementation as functions of `tūQ` Modeller appear at 03.06 in the Modeller video. Furthermore, Section 3.2.1, below is a demonstration of these operators as implemented in `tūQ` Modeller.

As identified in Figure 2.4, above, specific combinations or *patterns* of Pauli measurements are consistent with the Clifford group operators in transforming $|G\rangle$. Those patterns also confer the commutable properties of Clifford group operators itemised in Section 2.1. More broadly, patterns derived from elements of $\mathcal{P}^N$ may be composed and are tractable (Raußendorf, Browne, and Briegel 2003; Danos et al. 2009). By comparison, the gates-based encoding of state through a circuit and its lack of feedforward of state does not admit composition. The practicalities of introducing the non-Clifford operator, T to $|G\rangle$ to achieve a universal gate set; and those of optimising $|G\rangle$ will be further explored in Chapter 4. The focus of the remainder of this chapter is modelling $|G\rangle$ through `tūQ`.

3.2 Modelling graph states

The origin of `tūQ` Modeller mode is `Q2Graph`, a standalone application for modelling $|G\rangle$ (Bowen and Devitt 2022). Upon launching `tūQ`, the user selects Modeller mode to obtain the functionality discussed here. An outline of the functions to model $|G\rangle$ through `tūQ` Modeller appears as Appendix B of this dissertation; and the Modeller video includes options of how to assemble larger lattices quickly. Note also that Modeller as a proof-of-concept module can create a lattice of dimensions no greater than $[121, 121]$.

The requirements for `tūQ`/`Q2Graph` as a tool for modelling $|G\rangle$ were,

1. an interactive tool for drawing a planar/non-planar graph,
2. simple graphs only,
3. restricting operators to $\mathcal{P}^2$ to model $|G\rangle$ transformations,
4. demonstrate LC-equivalence.

The following conventions apply to `tūQ` Modeller,

- a vertex appears as a circle and signifies a qubit. Each vertex is labelled with a unique integer for identification,
- an edge, ab, between vertex a and vertex b appears as an unbroken line.

Two further design principles inform the `tūQ` requirements and differentiate the application from any circuit-based SDK alternatives:

i. a platform wherein (measurement) patterns are the standard syntax for expressing algorithms; as noted above, the Clifford patterns are composites of $\mathcal{P}^N$ operations, and

ii. a vehicle for deriving a graph-based algorithm to pass to a quantum computing facility.

The use case for tūQ is twofold namely to demonstrate,

- the inner workings of cluster-state architecture by enabling a user to interact with $|G\rangle$;
- an alternative computational model to the circuit.

There are parallels between tūQ and the quantum circuit emulator, Quirk (https://algassert.com/2016/05/22/quirk.html), which appeared at a time in quantum computing research when tools both to model a circuit and to compute output through it were scarce[2]. Like Quirk, which in many ways laid the groundwork for more sophisticated circuit-model emulation platforms, the desired outcome for tūQ is to prompt wider evaluation of cluster-state computing models, perhaps even extending to prototyping of compatible QPU technology.

tūQ differs from other well-known graph visualisation/design packages (e.g. Gansner and North 2000; Hagberg, Schult, and Swart 2008; Bastian, Heymann, and Jacomy 2009; Coene 2018; The Sage Developers 2024; Tantau 2024) insofar as its primary purpose is the design and modelling of a graph as an algorithm for an MBQC system. Specifically, tūQ enables its user to model $|G\rangle$ *directly*. Other bespoke MBQC modelling tools do exist with Graphix (Sunami and Fukushima 2022) and McBeth (Evans et al. 2023) perhaps the most widely known. Graphix is a Python library for writing text-based MBQC algorithms while McBeth is a complete syntax for reading, writing or simulating quantum programs (see, https://github.com/seunomonije/mcbeth). Moreover, both Graphix and McBeth offer syntax for the user to encode its intended algorithm directly, with cluster state, $|\phi\rangle$ created as a corollary of expressing the algorithm: $|\phi\rangle$ (and thus, $|G\rangle$) are the end product of the user's algorithm. In a manner similar to the SDKs of circuit-based quantum computing platforms discussed in the previous chapter, Graphix

[2]Quirk released in 2016; by way of comparison, both Qiskit and Q# released in 2017 while Cirq released in 2018.

and McBeth are vehicles for expressing algorithms compatible with an as-yet unrealised cluster-state QPU. They are initial steps to an MBQC API. While their respective use cases align with tūQ's, the design decisions underpinning Graphix and McBeth have no overlap with tūQ *as a modelling tool*. Graphix and McBeth abstract away from cluster state, $|\phi\rangle$ in the expressing of algorithms, which has the advantage of simplifying the process of writing algorithms but almost certainly at the cost of complexity in the optimisation layer of any compiler. tūQ Modeller on the other hand is a sandpit environment in which its user expresses an algorithm directly as $|G\rangle$ and can exploit LC-equivalence to refine it.

Before considering a worked example in tūQ Modeller below, there is further discussion to be had regarding the assumed properties of the qubit of Modeller mode, namely whether the qubit is a *physical* or a *logical* representation. More will be said on logical qubits in Chapter 4 of this dissertation but for present purposes, the logical qubit is defined as a system of physical qubits created for the purposes of mitigating errors that might arise through decoherence during a (quantum) computing process. There is no precise rule to the number of physical qubits required for constructing a logical qubit as the number scales in relation to more than one variable not least of which includes the nebulous complexity of the computation under consideration (e.g. Fowler et al. 2012). Regardless the seemingly unavoidable need for error mitigation in any foreseeable quantum computing facility may lead to accusations of inefficiency of tūQ-based models when the 'real' number of qubits required to realise such models is never stipulated. Such criticism however is not well-formed. To begin, tūQ Modeller is not an architectural design tool; as stated above, it is a sandpit environment for modelling $|G\rangle$, testing LC-equivalence thereof and reasoning about algorithms to be realised through cluster-state computation. Indeed, resolving hardware-side design problems of error mitigation or fault-tolerance is out of scope for tūQ Modeller so by that criterion alone, providence of physical or logical qubits is irrelevant to its models. The value of tūQ Modeller lies in its user being free to reason about the prop-

agation of state without having to account for the impact of (quantum) errors. As signalled before, the real consideration for tūQ Modeller is preserving a stabiliser state as a condition of local equivalence of $|G\rangle$, which tūQ achieves by admitting only $\mathcal{P}^2$ operators.

3.2.1 Worked example: tūQ Modeller

The following is an annotated demonstration of drawing then reducing the CNOT pattern, as specified by Raußendorf and Briegel (2002), with the use of tūQ Modeller. Recall that all MBQC patterns are simply aggregations of individual qubit measurement operators, the details of which can be found in Appendix B of this dissertation. While the CNOT pattern would ordinarily appear as a subgraph of G, it is necessarily a self-contained G and hence running through its vertex transformations will provide a reasonable demonstration of tūQ Modeller.

State is neither prepared in nor tracked through tūQ Modeller although the propagation of state is readily apparent through a graph-state modelled in tūQ; nevertheless, for completeness MBQC would prepare any $QC_{\mathcal{C}}$ as described in Section 2.2.2, above. Figure 3.6, below, is a schematic diagram of a prepared $QC_{\mathcal{C}}$. Computing with $QC_{\mathcal{C}}$ requires measuring its component qubits in a specific order and basis. The rightmost column of the cluster forms the readout qubits, the ultimate recipients of incremental state change caused by each qubit measurement and propagated through the lattice, via quantum teleportation. Readout qubits are measured in the computational basis.

The 15-qubits CNOT pattern reproduced as Figure 3.7a, below, is specified in Raußendorf and Briegel (2002). A circuit-model CNOT ('controlled NOT') is a two-qubits gate that makes the state of the target qubit dependent upon that of the control qubit: if the control qubit is in the excited state (i.e. $|1\rangle$) then, the target qubit is flipped (i.e. $|0\rangle$ becomes $|1\rangle$ and conversely). CNOT in matrix notation appears at Equation (2.14), above, while an example of the gate's conditional effect is as follows,

$$|00\rangle \to |00\rangle\,;\;\; |01\rangle \to |01\rangle\,;\;\; |10\rangle \to |11\rangle\,;\;\; |11\rangle \to |10\rangle\,. \tag{3.6}$$

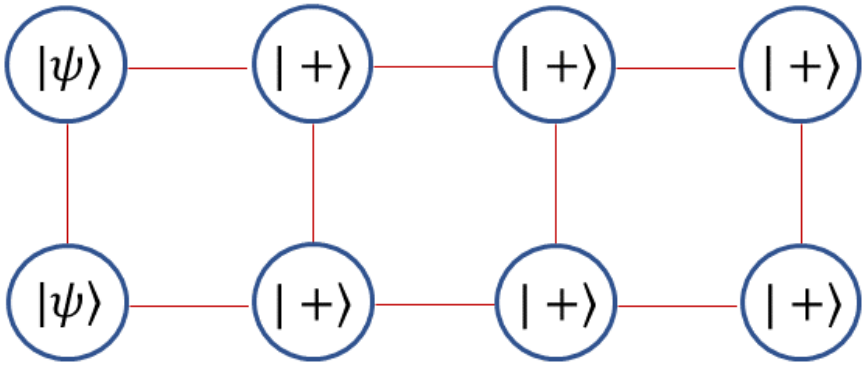
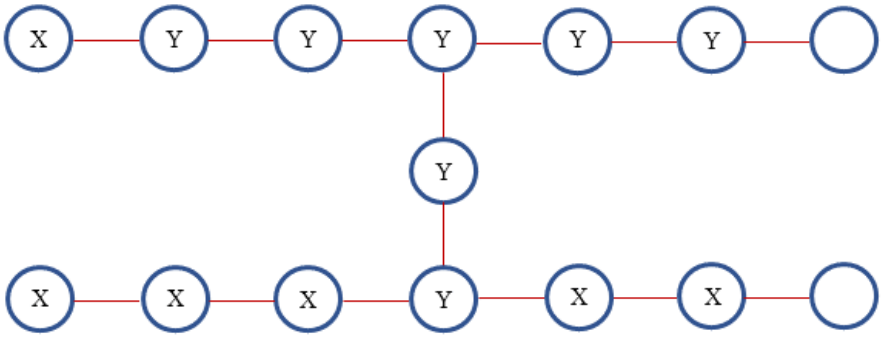


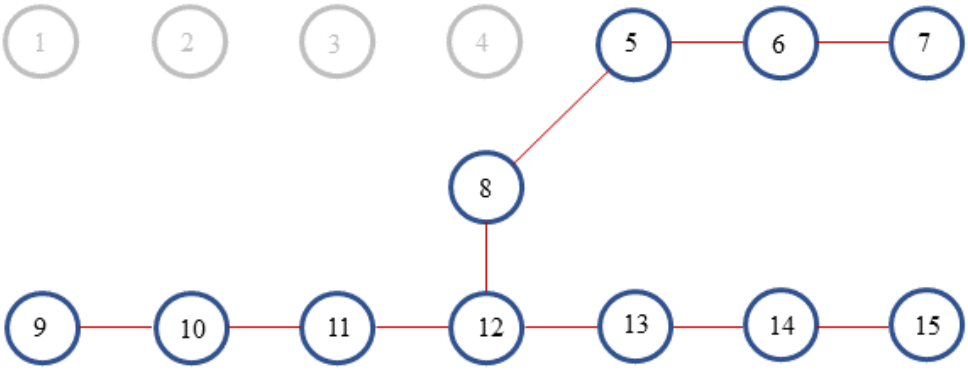
Figure 3.6: MBQC substrate, $QC_{\mathcal{C}}$. The left-most qubits are initialised with the required state, $|\psi\rangle$ then all remaining qubits are prepared in state $|+\rangle$ before all qubits are entangled. Once all other qubits are measured or were removed through Z-operations before computation began, measuring the right-most qubits in basis σ_z will supply the computational readout.

Under MBQC, measurement proceeds in a certain order and basis: for the CNOT pattern, qubits one and nine are prepared in a known state as input while measuring qubits seven and fifteen will deliver the required computational readout. The `tūQ` user can observe the inner workings of a graph-based algorithm as it would transform its substrate lattice. Figure 3.7, below, is a representation of the CNOT pattern at (a) initialisation then (b) at the instant after qubit four is measured but its neighbour is as yet undisturbed. Cumulative state change has propagated to readout qubit seven with measurements one through three. Successive measurements from qubit two have been in the basis σ_y, which performs local complementation upon any neighbour before removing the qubit from the lattice. As a consequence, incremental state change will accrue also to qubit eight from measurement four.

Figure 3.8, below, captures the point at which readout qubit seven will be measured in basis σ_z. Previous measurements five and six in basis σ_y propagated state both to readout qubit seven and to qubit eight; measuring qubit seven in basis σ_z will remove it from the lattice but its state at measurement will teleport to qubit eight. This incremen-



(a) the 15-qubits CNOT pattern as specified in Raußendorf and Briegel (2002). Each qubit of the pattern is labelled with the basis σ_x, σ_y measurements required to effect a CNOT transformation. As always, the leftmost column of qubits is for input and the rightmost column is the readout qubits.



(b) Qubits 1-4 of the CNOT pattern are measured, note the local complementation of neighbouring qubits 5 and 8 from applying operator Y to qubit 4.

Figure 3.7: The CNOT pattern at (a) initialisation and at (b) measurement 5, the instant at which qubit 4 is measured and qubit 5 is undisturbed.

tal emerging of state stands in juxtaposition to the gate-based model of control qubit triggering a state change in the target qubit.

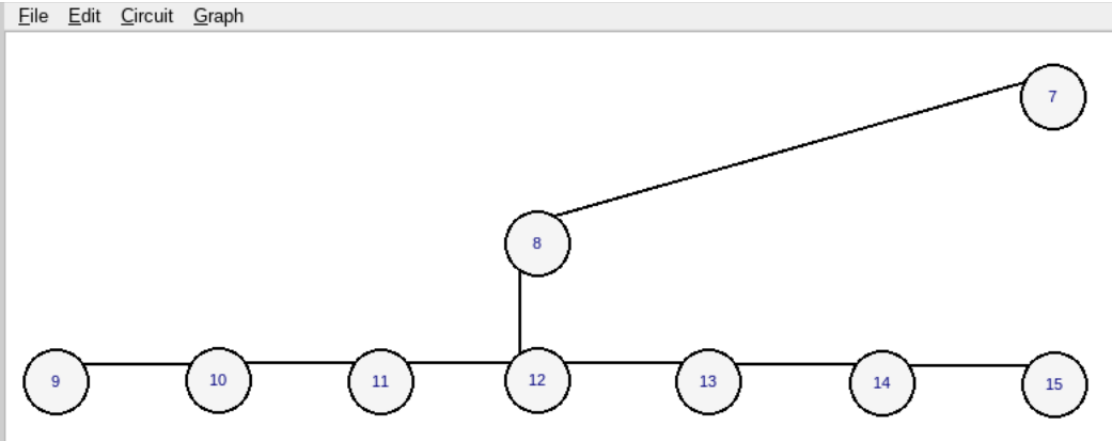


Figure 3.8: The CNOT pattern as rendered in `tūQ` at measurement 7, the instant at which qubit 6 is measured and qubit 7 is undisturbed.

The pattern finishes with measurements of the 'target' qubits. Unlike the 'control' qubits, most of the measurements are in the basis σ_x. The X-operator of $|G\rangle$ applies two consecutive local complementations that require the user to designate a 'special neighbour' vertex to determine the neighbourhood transformed by the second local complementation (see Appendix B). The basis σ_x measurements of qubits 10 and 14 are noteworthy because of this second local complementation. The configuration of their respective neighbourhoods is such that the consecutive local complementation works to isolate those qubits within the remaining lattice (Figure 3.9, below). State teleports to the two rightward neighbours of these qubits before entanglement with their immediate rightward neighbour is destroyed. At readout, then, these two qubits are left as remainders of the computation. The possibility of recycling such excess qubits into subsequent or even parallel computations will be explored in the first worked example of Chapter 4.

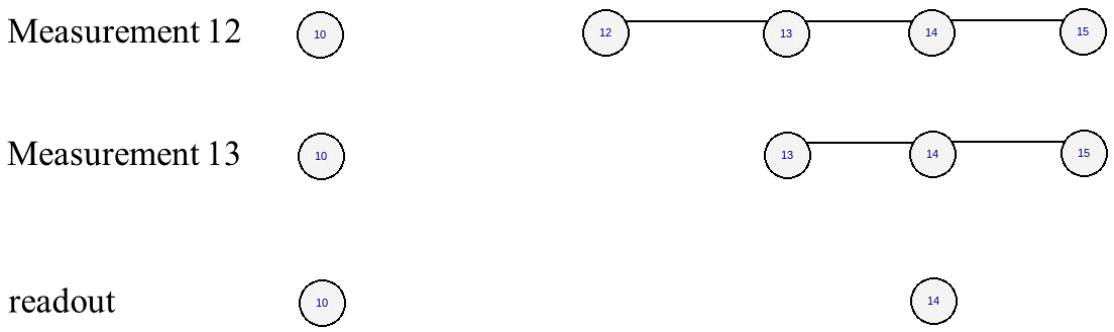


Figure 3.9: The CNOT pattern as rendered in tūQ at measurements 12 and 13 and at readout. Consecutive local complementation as part of the X-operator of $|G\rangle$ teleports state to the two rightward neighbours of qubits 10 and 14 then isolates those qubits within the remaining lattice by destroying entanglement with their immediate rightward neighbour.

3.3 Summary

The first of tūQ's three core functions is modelling of graph-based quantum algorithms using classical computing facilities. The user of tūQ Modeller is able to create and transform a graph state, $|G\rangle$ by:

1. adding or removing a vertex or an edge,
2. executing local complementation on a target subgraph,
3. reproducing Pauli measurement operators X, Y, Z through functions that combine elements of 1. with 2.

In turn, the restrictions on format and transformation of $|G\rangle$ are the guarantee of normalised state in any algorithm passed to a downstream quantum computing facility. In totality, tūQ Modeller enables the user to reason about consecutive reductions of $QC_{\mathcal{C}}$ without having simultaneously to account for the impact of (quantum) errors.

Measurement patterns make $|G\rangle$ equivalent to the circuit for expressiveness and tractability in encoding algorithms. By implication, the tūQ user can assemble and test a graph-based algorithm then reverse-engineer an equivalent circuit to run on a NISQ computer if so desired.

By reducing the graph, the tūQ user can see what an (ideal) circuit does. Moreover, tūQ is a suitable medium for formal proofs of algorithms on $QC_{\mathcal{C}}$ precisely because the application is free of quantum errors.

Ultimately the reason that Modeller is part of the tūQ toolchain is to minimise the spatial complexity of an algorithm, in this case the number of qubits required to resolve a computation. An iterative relationship between tūQ Modeller and Simulator, the subject of the following chapter, is thus implied, in which Modeller is a sandpit for the user to draft its algorithm for encoding in Simulator and evaluation of the computational output. If one sets aside the fact that the worked example of Section 3.2.1 assembled then transformed the (known) CNOT pattern, that interaction with Modeller might also be thought of as step one of this iterative relationship, as an exercise in sizing qubit requirements for an algorithm to be simulated. In many ways, creating tūQ Modeller necessitated creating its Simulator mode too. There are no obvious means to 'pipe' a graph-encoded algorithm through any emulator that is currently (or freely) available. As mentioned previously, the majority of emulators presently available accept only the circuit model as their algorithm. Furthermore, many emulators are either an SDK, most often tightly-coupled front- and back-end source code, or a PaaS, often a limited licence, proprietary GUI + API combination (cf. Litteken et al. 2020). Put plainly, it would be a non-trivial undertaking to reconfigure an open-source emulator in order for it to accept $|G\rangle$-as-algorithm. One solution would be to specify a transpiler of graph-to-QASM or graph-to-JSON in order to evaluate a graph-encoded algorithm through a SDK like Cirq or a platform like Qiskit. While such a transpiler approach does present a solution of sorts, it is also potentially tortuous and raises the risk of information loss. Provisions to encode and optimise an algorithm as part of the tūQ toolchain obviates the need for any reverse-engineering of algorithms drafted in Modeller.

The use case to realise an entire MBQC architecture as founded upon basic graph-state modelling functionality begins with tūQ Modeller.

Chapters 4 then 5 are case studies of `tūQ`'s Simulator mode and functions to integrate Modeller-Simulator in workflow, respectively.

Chapter 4

Simulating graph states with tūQ

As noted in the previous chapter, the rationale for tūQ's Modeller mode is to minimise the number of qubits needed to complete an arbitrary computation. That same assumption applies to tūQ's Simulator mode, the subject of this chapter. A brief review of quantum emulators and the general use case for such applications is used to position Simulator within the tūQ toolchain. Of the core innovations brought by Simulator perhaps the most significant to this dissertation is that it presents an alternative to the circuit notation for encoding $|G\rangle$.

The origins of tūQ Simulator lie with a research tool named Etch, which was bespoke to an enquiry into circuit etching as an alternative optimising strategy to the algorithm-specific graph. The details of this enquiry are discussed as an extended use case for Etch and subsequently, tūQ Simulator. The proportion of Pauli qubits to non-Pauli qubits situated in a given $|G\rangle$ and how they are obtained will be the focus of this enquiry.

Although tūQ Simulator shares a modified version of Etch's circuit reading function, Simulator is by functionality a superset of Etch. tūQ's Simulator and Modeller modes will be shown to be complementary, bringing finer or coarser abstraction to the same $|G\rangle$. Simulator ab-

stracts away unnecessary detail to enable its user to concentrate on the *logic* of an algorithm while among Modeller's uses is to be a drill-down facility, enabling the user to track how an algorithm resolves in part or in its entirety.

Original material

- The source code for Etch may be inspected at URL: `https://github.com/QSI-BAQS/Etch`; the source code for `tūQ` Simulator may be inspected at URL: `https://github.com/QSI-BAQS/tuQ`. References to both applications appear in this chapter.
- Sections 4.2 and 4.3 represent original work that appears in Bowen, Caesura, Devitt and Krishnan Vijayan (2025).
- Parts of Section 4.4 represent original work that appears in Bowen and Devitt (2025).
- As with Chapter 3, there is a supplementary video for this introduction to `tūQ` Simulator, entitled
  - A Toolchain for Cluster-State Architecture Simulator.

  The video is a standalone introduction to Simulator but references to specific passages in the video will appear in this chapter. Citation will be of the form 'Simulator video' and may include a starting point in format Minute.Second, as required.

4.1 Emulators and compilers

The previous chapter was in part a demonstration of a graph state, $|G\rangle$ and its equivalence to a cluster state, $|\phi\rangle$ for the purposes of modelling a (quantum) computation. To restate the application's use case, `tūQ` Modeller is a sandpit environment for modelling and transforming a graph

state, or a subgraph of it, by means of defined functions. Indeed, if one thinks of such transformation as a proxy for a computational *process* then, $|G\rangle$ is the medium for n iterations of a basic cycle of initialising, transforming then reading of system state. Indeed, through quantum teleportation as discussed in Section 2.1, above, measuring its preceding nearest-neighbour, $|\phi\rangle^{i-1}$ determines the state of qubit, $|\phi\rangle^{i}$ just as measuring it determines state in the succeeding nearest-neighbour qubit, $|\phi\rangle^{i+1}$ and so on, *cumulatively*. Reading the state of any qubit within this cycle of transformation is a matter of measuring the qubit in the computational basis, with Pauli corrections accounting for determinism in the computation overall (see Chapter 2). Initialising and reading of computational state are not functions of tūQ Modeller, as stipulated in Chapter 3. Traditionally such functionality is the prevail of a quantum *emulator*, an application for encoding a quantum algorithm and computing its state outcome.

A quantum emulator is a classical computing application, with IBM's Qiskit as perhaps the most widely recognised example. Most instances of quantum emulators encode an algorithm using circuit notation and 'compute' the output as a sequence of matrix transformations that mimic the requisite quantum gates. An emulator's operations are limited only by the classical framework itself, in this case the amount of main memory at the user's disposal[1] and those quantum operations a classical computer can replicate viz., the Clifford group operations (Gottesman 2008). As noted in Section 2.1, above, the Clifford group does not confer universality, which necessarily limits the scope of algorithms accessible to an emulator.

Apart from the limitations to (quantum) algorithms, it is also instructive to compare the function of an emulator with a compiler's. As noted at several points above, an emulator is not a compiler, an assertion that extends to IBM's Qiskit, Microsoft's QDK/Q# and Google's Cirq, each of

[1]The unitary matrix, U required to transform a quantum system grows exponentially with the number of qubits that comprise the system.

which doubles as an API to a NISQ service. Among other differences, an emulator does not

- perform compile-time checks of input syntax,
- optimise an algorithm into an IR,
- transpile to assembly language or machine code[2].

There can be many reasons why an emulator does not optimise an algorithm although the near-universal requirement to encode in circuit format may be the common component. A circuit as syntax is 'close to the metal', all but directly transpiling to QASM in its specifying and sequencing of quantum gate operations. Simplicity of its syntax necessarily reduces the expressiveness of a circuit and also limits the utility of syntax-checking functions in a circuit-based compiler. That a circuit has no parallel among IRs of classical compilers leaves only its output and runtime as metrics to evaluate an algorithm encoded in that notation scheme. Concomitantly, efforts at optimising the circuit-based algorithm depend to some degree upon guesswork.

Its Simulator mode, as outlined in depth below, extends the `tūQ` toolchain in two ways, enabling the user to:

1. write algorithms expressly for $|G\rangle$, in an alternative notation to the circuit; and
2. derive the size of the lattice medium dictated by 1., in real time.

As such, this `tūQ` mode partly fulfils the role of both quantum emulator and (compiling) optimiser and thus the name, *Simulator*. The graph-state algorithm notation introduced in 1. above is a tile-based syntax

[2]Those emulators with an API provision make an RPC to NISQ computer and part of that procedure includes transpiling the input algorithm to QASM; whether this procedure makes the emulator a compiler is arguable.

that obeys the same 'specific order and basis' rule to the placing of patterns as tūQ Modeller but at a higher-level of abstraction. tūQ Simulator's tile-based notation scheme is an effort to redress a perception of drafting algorithms for a graph-state model as being less intuitive than drafting algorithms in circuit notation. As shall be made apparent, the benefit of tūQ Simulator is that the user can transfer its algorithm directly to tūQ Modeller where it can manually evolve $|G\rangle$ as a quantum process. From that perspective, the tūQ toolchain presents as less of a black box than gate-based operations typical of circuit-based APIs and emulators. Similarly, a proof of concept optimiser was a priority for tūQ as a compiler toolchain. The fact that $|G\rangle$ may be optimised through the property of LC-equivalence, however incrementally such optimisations may be realised, must put $|G\rangle$ on a competitive footing with the circuit model as an efficient computing paradigm. As noted above, optimising a circuit is impractical but that does not make optimising $|G\rangle$ a straightforward exercise: a disinterested appraisal of tūQ Simulator's optimising functions is required. The second of two worked examples involving lattice specification below is a fuller reckoning with optimising a graph-state algorithm. For now it is enough to observe that tūQ Simulator offers no automated optimising but rather a user can draft an algorithm therein then look at how it forms as a lattice through Modeller mode. It is in this capacity that Modeller's local complementation function[3] comes to the fore as a tool for reducing lattice size through the property of LC-equivalence. Any consistencies discovered in such manual minimisation procedures are the germ of automated optimisation in future versions of graph-state compilers.

[3]Appendix B has details of tūQ Modeller's LC function and the Modeller video includes a demonstration of LC applied to $|G\rangle$ at 03.48.

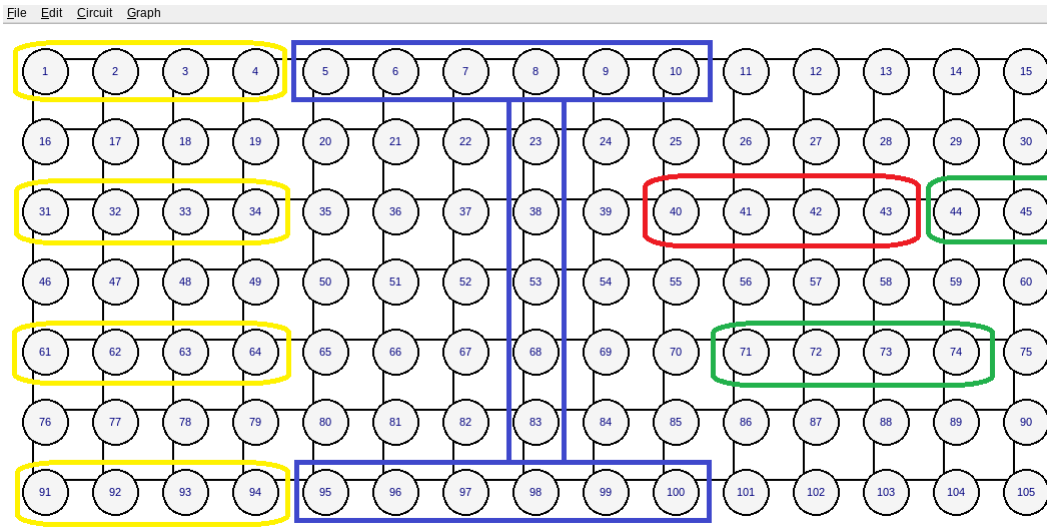
4.2 Before tūQ Simulator: circuit etching

The precursor to tūQ Simulator was a project artefact with the working title of 'Etch'. As the name would suggest, Etch was created for the purpose of 'circuit etching', which is the procedure of transposing a circuit-based algorithm to its graph-state equivalent. From its comparatively focused use case, Etch as an application runs leaner than tūQ Simulator.

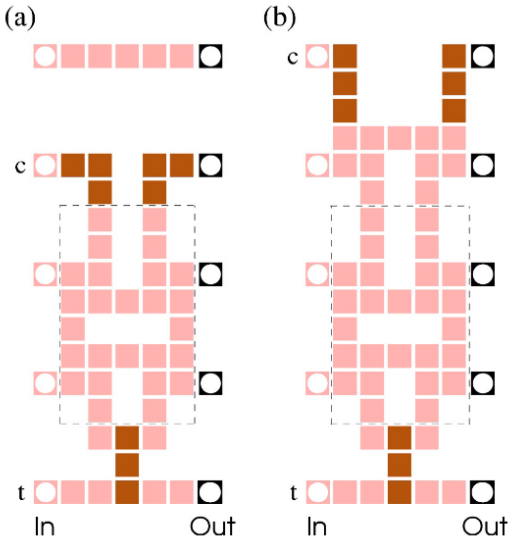
In regard to input, Etch is able to read the following gate encodings as input to estimate the size of an equivalent graph state:

- adjacent row CNOTs, exclusively,
- $R_x(\theta)$, $R_y(\theta)$, $R_z(\theta)$, which describe single-qubit rotations by θ radians about axis $x/y/z$, respectively;
- Hadamard (H),
- $S/S^\dagger$,
- 'general rotation', U_{Rot},
- $T/T^\dagger$.

A significant point of difference between the two applications is that Etch can read circuits which include gate operations that are otherwise inadmissible to tūQ Simulator. That said, circuit input that includes a CNOT spanning across non-adjacent rows poses two problems for both Etch and as will be seen, Simulator and hence the adjacent rows only restriction on gate encodings. First, the non-neighbouring CNOT obstructs left-to-right teleportation of state of any qubit wire situated between the CNOT control and target qubit wires. Notionally, this problem is soluble with a crossing sub-pattern proposed in Raußendorf, Browne and Briegel (2003) but that solution involves a greater number of qubits than the standard 15-qubits CNOT; see Figure 4.1 for examples of the non-adjacent CNOT dilemma and the non-neighbouring CNOT



(a) A non-adjacent CNOT (blue highlight) as it impacts other patterns - H(yellow), S (red) and Z-rotation (green) - of the lattice. The 'nexus' qubits of the CNOT (IDs 23, 38, 53, 68, 83) also correlate with wire qubits teleporting state left-to-right from the Hadamards at rows 2 and 4 to the $S \circ$ Z-rotation composition and the isolated Z-rotation pattern, respectively.



(b) CNOT pattern for non-neighbouring qubits. Qubits at light pink sites are measured in basis σ_x and those at dark brown, in basis σ_y. Patterns (a) and (b) apply to control and target separated by an odd or even number of qubits, respectively. The pattern can extend across any separation by repeating the qubits configuration within dashed lines; **source**: Raußendorf, Browne and Briegel (2003), Figure 11.

Figure 4.1: The impact of a non-adjacent CNOT on a cluster-state computation and a resolving pattern as proposed by Raußendorf, Browne and Briegel (2003).

pattern proposed by Raußendorf, Browne and Briegel (2003). Such a fix increases the ratio of Pauli to non-Pauli measurements in $|G\rangle$ and therefore hinders the desired outcome of Etch's research question. More will be said on this topic in Worked Example 1, below. Second, including non-adjacent CNOTs also increases the count of qubits to be removed through measurement in basis σ_z, increasing qubits wastage. Although any excised qubits do not count towards what will be termed the distillation ratio in Worked Example 1, profligacy with qubits is ill-advised in general at this point in the evolution of quantum computing facilities.

A demonstration of typical Etch output appears as Figure 4.2a, below. When interpreting Etch text output,

- it is a text-based format and consists of vital statistics of $|G\rangle$ namely, lattice dimensions and a zero-based coordinates system for patterns found within the lattice. If one thinks of a pattern as a 'block' of qubits (cf. Figure 2.4, above), Etch's coordinates represent the pattern's location as [ID of pattern's bottom row, south-western qubit of pattern] with all patterns bar CNOT occupying a single row and CNOT occupying three rows;
- as noted above, the output lattice is two-dimensional (e.g. Raußendorf and Briegel 2001, 2002; Raußendorf, Browne, and Briegel 2003) not three-dimensional as is consistent with error-correcting strategies (i.e. Raußendorf, Harrington, and Goyal 2006). Figure 4.2b is a representation of Etch output as constructed in `tūQ` Modeller;
- recall, it is possible to compose patterns of $|G\rangle$. After Danos et al. (2009), if pattern A is left of pattern B and is composed with pattern B, pattern B's initialisation qubit doubles as the readout qubit of pattern A, which is dropped altogether (see Section 3.2.1 above as an example of measuring $|G\rangle$ in a specific order and basis); and

- Etch does not optimise the specification of its output lattice: configuration of the input circuit strictly specifies the situating of patterns within $|G\rangle$ and thus the size of the lattice overall.

The first of the two worked examples below focuses on Etch and will show that it encoded a lattice to address the question of relative efficiency of different $|G\rangle$ inputs to a quantum computing facility. To define that statement further, 'lattice' is the same two-dimensional $|G\rangle$ as `tūQ` Modeller generates, while 'encoding' means *explicitly*, the situating of patterns within $|G\rangle$ and *implicitly*, the ratio of Clifford group patterns to non-Clifford group patterns. Recall also that Etch recognises as input a modestly broader range of gate operations than `tūQ` Simulator. In that and other senses to be made explicit between the following two worked examples, Etch and `tūQ` Simulator are similar, not the same, in design and execution.

4.3 Worked example 1: comparing the circuit etching and algorithm-specific graph strategies

Etch was created to answer the question, is there a lattice configuration for an algorithm that draws on a universal quantum gate set, which is comparable to the optimisation known as the *algorithm-specific graph* in terms of its resource efficiency? By way of introducing the research question, some of these terms require further explanation to wit,

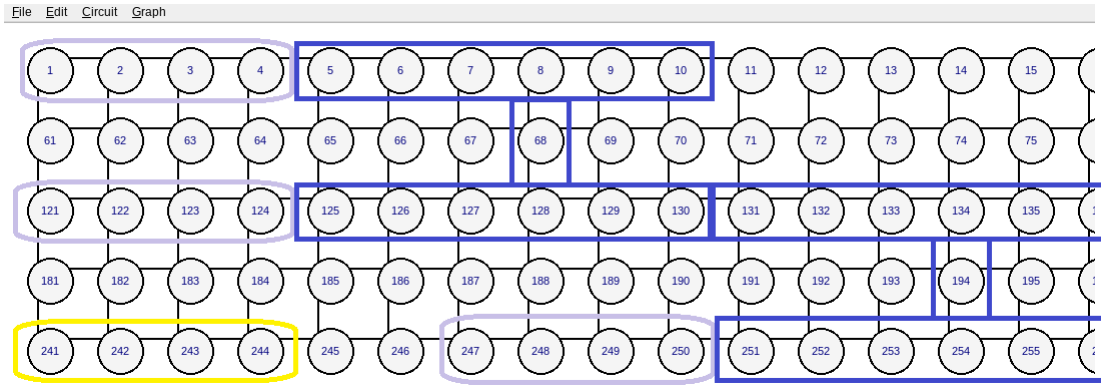
- *universal quantum gate set* is the Clifford group operators plus the non-Clifford $\frac{\pi}{4}$ (T) rotation stipulated in Section 2.1;[4]

[4]Note, Raußendorf and Briegel (2002) propose an arbitrary rotation operator as their non-Clifford pattern; see Boykin *et al.* (1999) for alternative configurations.

The input circuit specifies a $[5, 60]$ cluster state.

gate → [SE, SW] coordinates:
t → [0, 3]
t → [2, 3]
h → [4, 3]
cnot → [2, 9]
t → [4, 9]
cnot → [4, 15]
$t^\dagger$ → [2, 19]
t → [4, 19]
cnot → [2, 25]
cnot → [4, 31]
cnot → [2, 37]
$t^\dagger$ → [4, 37]
cnot → [4, 43]
cnot → [2, 49]
$t^\dagger$ → [4, 49]
cnot → [4, 55]
h → [4, 59]

(a) Lattice specifications output of Etch. Etch returns the necessary information for creating a lattice from an input circuit including the $M \cdot N$ dimensions of the lattice and zero-based coordinates of patterns [ID of pattern's bottom row, southwestern qubit of pattern]. Note, configuration of the input circuit strictly specifies the situating of patterns and thus the size of the 2D-lattice.



(b) Western extremity of the $[5, 60]$ lattice specified by Etch, as constructed in Modeller. The coloured outlines superimposed upon the lattice indicate the position of each pattern as specified by Etch coordinates. The colour scheme in this instance is lilac (T), yellow (H) and blue (CNOT).

Figure 4.2: Etch text output and its representation as a lattice and respective patterns as constructed in tūQ Modeller.

- *resource efficiency*: a minimised $|G\rangle$ by dimensions and necessarily, an equally minimised number of qubit elements to that $|G\rangle$. The number of physical qubits available through NISQ computers make resource efficiency a critical consideration of algorithm design (see Section 2.4 above for examples).

Algorithm-specific graph (henceforth, ASG) merits a longer introduction. An ASG is a QIR that divides the input algorithm between its Clifford and non-Clifford components. Recall from Chapter 2 that a circuit or $|G\rangle$ is as syntactically expressive as an intermediate representation of classical compilers. The practicalities of this equivalence are such that any optimisation required for compiling a quantum algorithm must be encoded as steps in the algorithm. As noted above, a circuit as syntax all but directly transpiles to QASM. In comparison, optimising a graph-state requires a reduction strategy like ASG or LC-/LU-equivalence between the graph-state and another $|G\rangle$ with fewer vertices/qubits. As a consequence, computing state for the Clifford components of the algorithm occurs in a classical facility, in a manner consistent with the Gottesman-Knill theorem (Gottesman 2008) and thereby reducing the overall number of qubits required to compute the input algorithm. Any non-Clifford components of the algorithm are input to a quantum computing facility. In other words, the ASG proceeds thus:

1. In a *classical* computing facility,
   (a) create a lattice sufficient for the required graph-state, $|G\rangle$,
   (b) remove any qubit from (a) that is superfluous to $|G\rangle$ by measuring it in basis σ_z
   (c) measure all qubits of (b) in Pauli bases, $\mathcal{P}^N$ with the exception of any non-Clifford T pattern(s).
2. As required, in a *quantum* computing facility,
   (d) process any T patterns of 1(c).

Steps (a) through (c) effectively trim an algorithm of computation that can be efficiently handled on a classical computer and most importantly, prepare the input qubit(s) of any residual T pattern(s). From the perspective of this combined classical-quantum stack, step (d) is, in effect, the intermediate representation of steps (a) through (c) (cf. Cross et al. 2017; Häner et al. 2018). As an approach, the ASG is both an accommodation with the resource limitations of current NISQ computers and recognition that facilities of quantum computing and classical computing will coexist and interact, rather than the former supplanting the latter.

Jabalizer (Krishnan Vijayan et al. 2024) is a proto-compiler that optimises its (graph-state) algorithm input with an ASG. The case for an alternative graph-state optimising strategy came when early versions of Jabalizer,

- returned ASGs with an irregular structure. This can make it difficult to construct in a quantum computing facility at least as compared to the two-dimensional graph of MBQC; and
- experienced insuperable performance limitations, as documented below.

The possibility of *reusing* the measured qubits of $|G\rangle$ as an alternative strategy for restricting qubit numbers overall became the impetus for the application, Etch. Consistent with the ASG optimisation, Jabalizer splits an input circuit between Clifford patterns, which it routes to a classical computer, and any T patterns, which go to the quantum computing facility. It is vital to be clear at this point in comparing Etch to Jabalizer: Etch is a standalone tool for quantifying circuit etching, it is not part of a compiler toolchain. To repeat, the input to Etch is a circuit encoded in JSON format while its output consists of (1), the dimensions of an equivalent graph-state and (2), the coordinates within the graph-state of patterns to replicate the gates of the circuit, as per

Figure 4.2a, above. Furthermore, as stated above, Etch does not optimise the specification of its output graph state: Etch passes a graph state to the quantum computing facility, precisely as it was set by the circuit input.

Let λ represent an arbitrary quantum algorithm expressed in the universal quantum gate set named above (i.e. Clifford group operators plus T) and passed to Jabalizer or Etch. The algorithm λ will have a proportion of Clifford patterns, $\lambda_{\mathcal{P}}$ and T patterns, λ_T such that

$$\lambda = \lambda_{\mathcal{P}} + \lambda_T = 1. \tag{4.1}$$

Ignoring,

$$\lambda = \lambda_T; \lambda = \lambda_{\mathcal{P}}, \tag{4.2}$$

it seems obvious that processing λ (i.e. circuit etching) will always consume more qubits than processing λ_T (i.e. algorithm-specific graph). The circuit-etching project, for which Etch was made, was a test of this proposition. Circuit etching as an alternative to ASG hinges upon the *relative* efficiency of initialising T-states and specifically, the total number of qubits required. Both ASG and circuit-etching require the same number of qubits to process λ_T. For its part, Jabalizer processes λ_T with 'single-use' qubits because a classical computing facility will process $\lambda_{\mathcal{P}}$. In comparison, circuit etching is a proposal to recycle any qubit consumed in processing $\lambda_{\mathcal{P}}$ for the purpose of then processing λ_T. As shall be made clear, Etch was created to quantify this proposition.

A few working assumptions of circuit etching require further comment:

- The T operation of rotating a qubit by $\frac{\pi}{4}$ radians around the z-axis cannot be directly implemented in a fault-tolerant manner for error-corrected systems. Processing a T operation at an acceptable threshold of error is achievable through a procedure known as magic state distillation, which also increases the overall number of qubit resources required to resolve a computation. The term 'distillation ratio' will refer to the proportion of qubits recycled from

a cluster state to the number of qubits required to effect the magic state distillation necessary to the computation. The ratio itself is a target for Etch either to meet or approximate within an arbitrary level of tolerance, as a metric of resource allocation efficiency;

- Unless explicitly stated to the contrary, 'qubit' is short for *logical* qubit. As prefaced in Chapter 3, the logical qubit is a construct that appears in quantum error correction research which includes the literature surrounding magic state distillation. A logical qubit is a system of n physical qubits, bearing in mind that a qubit is itself an abstraction for example, of the quantum property of spin. The purpose of a logical qubit is to spread computational state over the component physical qubits to facilitate the correcting of errors in state that invariably arise. Essentially, a logical qubit exists to mitigate errors that may arise from environmental decoherence (e.g. Preskill 1998; Fowler et al. 2012; Horsman et al. 2012; Terhal 2015; Litinski 2019a). The logical qubit is relevant to this worked example inasmuch as it is a modelling device of magic state distillation but a broader investigation of the field of quantum error correction is outside the scope of this dissertation.[5]

The particulars of magic state distillation as a procedure are the last background details to cover before beginning a more detailed comparison of ASG as executed by Jabalizer and circuit etching as quantified by Etch. Transforming $|G\rangle$ by means of a T pattern imposes an extra resourcing cost in the form of the noiseless qubits needed for magic state distillation (Bravyi and Kitaev 2005). Briefly, the required T-state is 'distilled' by preparing a number of ancillary magic state (physical)

[5]See (Bravyi and Raußendorf 2007; Gottesman 2009; Devitt, Nemoto, and Munro 2009; Fowler, Stephens, and Groszkowski 2009; Fowler et al. 2012; Terhal 2015) as examples of discussion of the logical qubit as an elementary component of error correction strategies and the surface code in particular, to the extent that these strategies bear upon cluster-state computation.

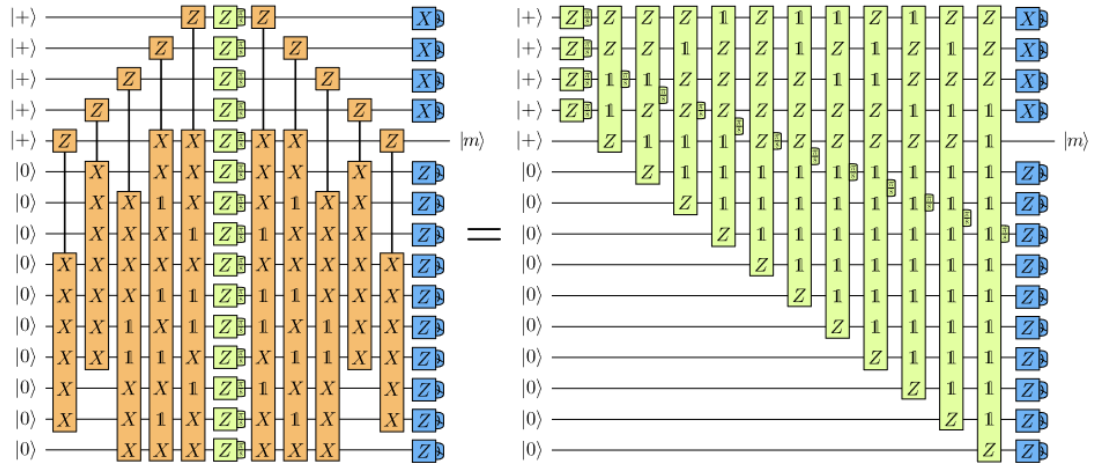


Figure 4.3: The encode-T-decode circuit to effect 15-to-1 magic state distillation, **source**: Litinski (2019a), Figure 14.

qubits with an equal number of noiseless physical qubits to encode one qubit at a low probability of error. Recall that in the two-dimensional lattice of MBQC, the T pattern should reduce to a single encoded readout qubit that initialises any subsequent pattern. The ancillary qubits required to process the T pattern will comprise their own lattice, a 'T factory', separate from the main lattice of computation and reserved for such operations. As the initial step, it takes 15 physical qubits to encode one logical qubit in a T-state (Litinski 2019a, 2019b). This 15-to-1 magic state distillation effects the T-state initialisation at probability of error, $35p^3$ to leading order; see Figure 4.3 as a reproduction of Litinski's 15-to-1 protocol. As might be expected, it is possible to realise magic states of a higher fidelity at the cost of more qubits. The important consideration is that each instance of a T pattern in a graph-state algorithm requires a separate T factory, which necessarily increases the qubit requirements of the cluster (Raußendorf, Harrington, and Goyal 2007). The qubits necessary to processing an algorithm must either be part of the algorithm's lattice at inception or be allocated at some stage of processing the algorithm. Having set the scene, it now remains to introduce Jabalizer, the circuit-etching strategy as exemplified by Etch and their respective resource allocation models.

A full introduction to Jabalizer's functionality is available at (Krishnan Vijayan et al. 2024) but at its most elementary, the application takes a circuit as input to output a graph state consistent with the fault-tolerant ICM form (Paler et al. 2017). ICM is a technique of decomposition and measurement to facilitate fault-tolerant computation from an input algorithm/circuit. Jabalizer realises the ICM form through an ASG. To fulfill the ASG strategy, Jabalizer passes the Clifford gates component of its input circuit to a classical computing facility, to resolve using the stabiliser formalism (Gottesman 2008; Raußendorf, Harrington, and Goyal 2006); the application passes all remaining T operators to quantum hardware as a specification for the requisite (magic state) distillation lattice. The Clifford/non-Clifford outputs are reconciled as the final stage of computation.

Zapata Computing (Boston, MA) ran tests of Jabalizer as part of an exercise in benchmarking compilation times. The tests were to simulate a hydrogen chain of length two to 30 through quantum signal processing (QSP) (e.g. Low, Yoder, and Chuang 2016; Low and Chuang 2017). These tests exposed potentially insurmountable limitations to global compiling on Jabalizer. Test input circuits of size nine to 45 qubits, each with 41-76K gate operations, were passed to Jabalizer. Memory crashes when writing to IRs or prohibitive runtimes from processing the Clifford gate component resulted from input circuits of 30 or more qubits in size (cf. Litteken et al. 2020). A snapshot of these issues appears as Figures 4.4 and 4.5. Efforts to resolve Jabalizer's compile- and runtime problems stalled at the fact that circuits do not compose, as noted above; and that any workaround of streaming a circuit would be hampered by Jabalizer's requirement that its input qubits be prepared at $|0\rangle$. This was the context of the decision to build Etch as a test of a circuit etching strategy.

If compilation occurs at the error-corrected level, a qubit measured as part of a cluster state computation is effectively discarded (cf. Browne and Rudolph 2005). Under circuit etching, that consumed qubit might be repurposed as a component of a magic state distillation

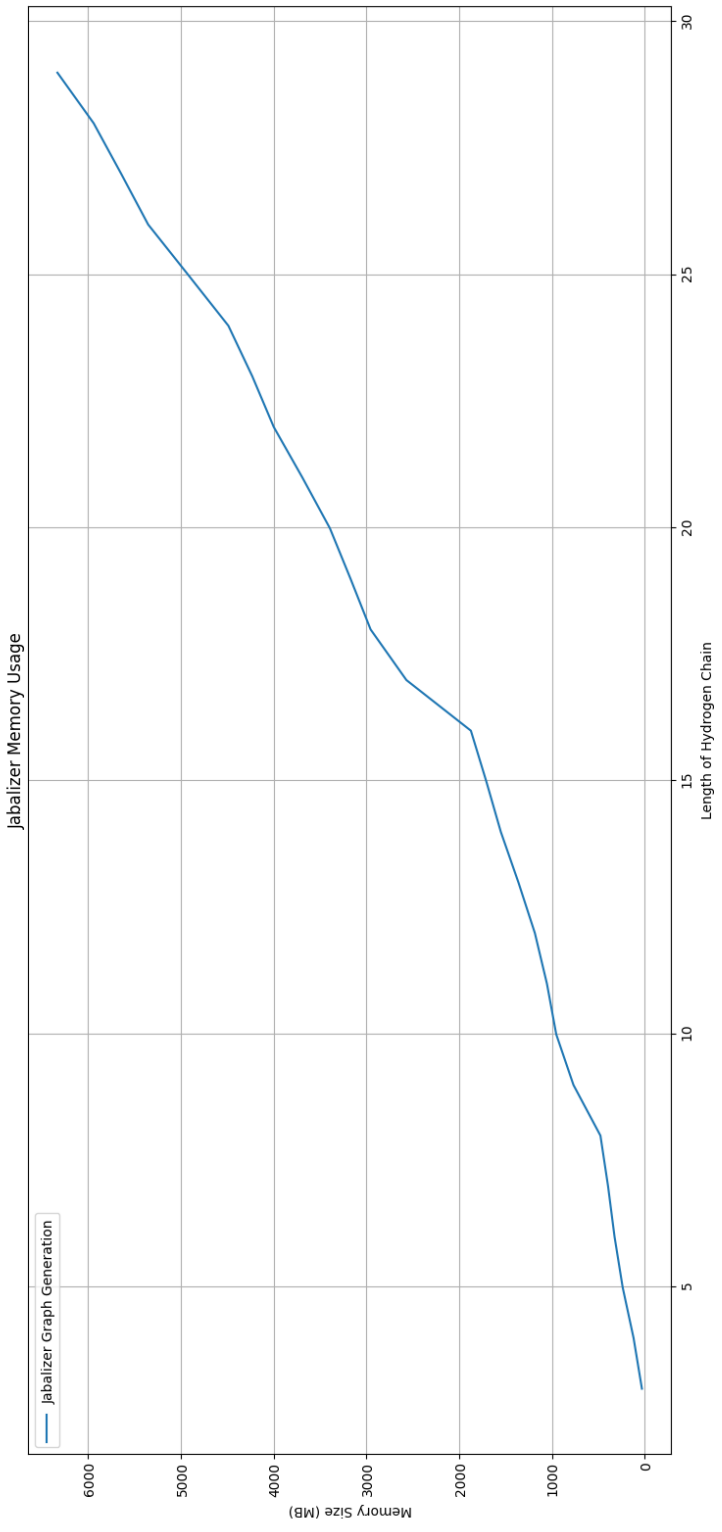


Figure 4.4: Jabalizer memory use from quantum signal processing circuits, **source**: Zapata Computing, Boston MA.

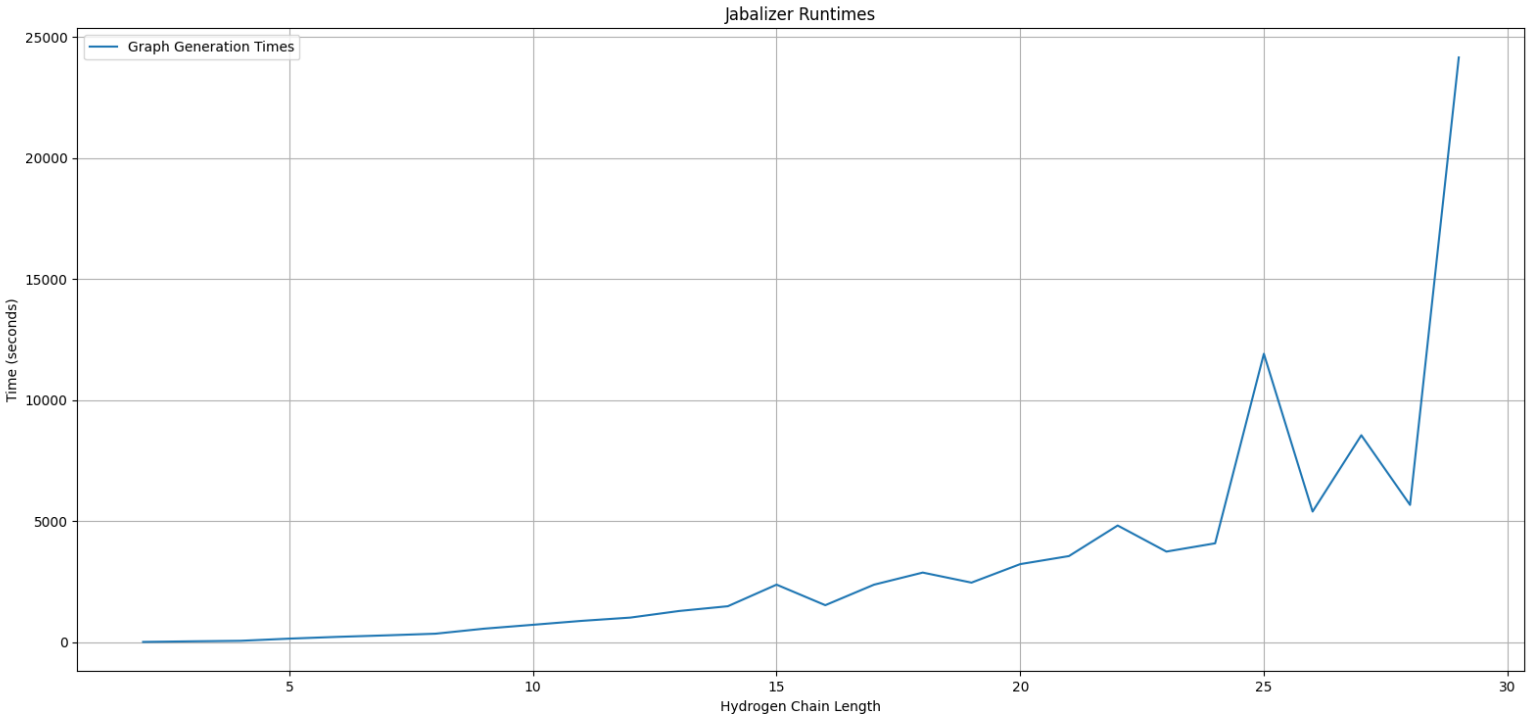


Figure 4.5: Jabalizer runtimes from quantum signal processing circuits, **source**: Zapata Computing, Boston MA.

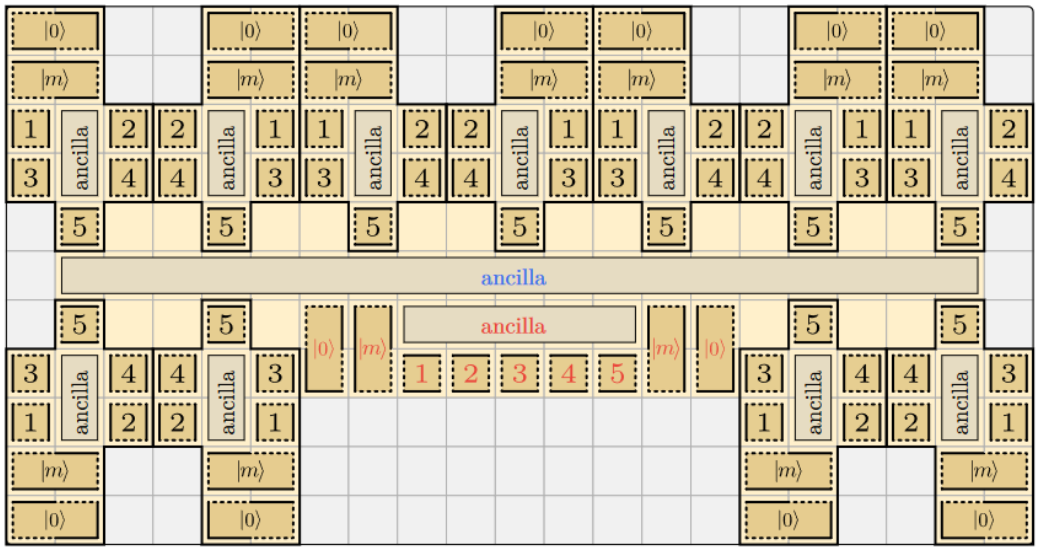


Figure 4.6: The 176-tiles block proposed for obtaining the concatenation protocol **source**: Litinski (2019a), Figure 19

circuit; it also bears repeating that circuit etching works with logical qubits and the only hard-and-fast number concerning physical qubits required to make a logical qubit is a *floor* number of seven (Kraglund Andersen et al. 2020).

The concatenation protocol is an instance of the higher fidelity magic state distillation circuits also mentioned above. The protocol takes 15 *logical* qubits, each encoded under 15-to-1 distillation, to apply the same 15-to-1 encoding process to them in turn and thereby realise a probability of error of $35(35p^3)^3$ to leading order. On paper, a single concatenation protocol would amount to a 225-tiles block (i.e. 15^2) however Litinski (2019a, Figure 19) proposes a concatenation protocol of a 176-tiles block of dimensions 11 rows by 21 columns; see Figure 4.6 as a reproduction of Litinski's proposed 176-tiles block. This 176-tiles block is an instance of a T factory, as identified above.

Consider again the two-dimensional lattice of $|G\rangle$, that can be described as an amalgam of qubits arranged in consecutive columns or, equivalently arranged in rows. If $|G\rangle$ is the resource for an algorithm

$\lambda = \lambda_{\mathcal{P}} + \lambda_T$ then, at least one T pattern is sited within $|G\rangle$ and will require a sufficient number of unused qubits to encode it through a T factory in this case, Litinski's 176-tiles concatenation protocol (cf. Elman, Gavriel, and Mann 2024). To encode the T pattern's readout qubit will require 11 other unused qubits as a contiguous block, in the same column as it from which might be formed a column of Litinski's 176-tiles concatenation protocol. The site of the T pattern within the lattice is irrelevant so long as there are sufficient unused qubits to create Litinski's [11, 21] tile block adjacent to it. In other words, circuit etching must repurpose 11 qubits that were measured in the Pauli basis, for every one non-Clifford readout qubit or be within an arbitrary level of tolerance to this ratio (say, no greater than 25 Pauli : 1 non-Pauli). *The distillation ratio mooted above is thus 11 : 1* and represents the target of testing outlined in section 4.3.1, below. Furthermore, circuit etching must approximate the 11 : 1 distillation ratio regardless of the proportion of λ_T and $\lambda_{\mathcal{P}}$ operators. Iff circuit etching satisfies both of these conditions, it is an efficient resourcing alternative to ASG.

4.3.1 Evaluating circuit etching with Etch

Restating the application's functions, Etch specifies the:

1. minimum dimensions of $|G\rangle$ from a circuit model input; and
2. coordinates of measurement patterns within 1. in a layout strictly equivalent to the input circuit.

The instantaneous quantum polynomial (IQP) circuit (Shepherd and Bremner 2009) was the archetype for input circuits passed to Etch as part of its distillation ratio evaluation. A non-varying set of gates defines the IQP circuit so notwithstanding randomness of circuit size, the combinations of circuit gates is consistent: the same gates, in different proportions, appear in each IQP circuit instance. A circuit entirely generated at random will seldom align with an algorithm drafted to solve

a defined problem and indeed, algorithms working in the same hypothetical problem space commonly comprise of similar if not, the same combinations of gates. The qubits count of each circuit was generated at random[6] from within a range, 5-120.

Etch processed thirty instances of IQP circuit as a test of circuit etching's capacity to achieve the distillation ratio. The number of qubits constituting the individual patterns in Etch's $|G\rangle$ output is consistent with those of Raußendorf, Browne and Briegel (2003) *except* that a pattern's property of composition means the readout qubits are dropped from it. For example, the 15-qubits CNOT ordinarily configured as 7-1-7 (see Figure 3.7a) becomes 6-1-6 under circuit etching unless that CNOT's rightmost qubits would fall on the right boundary of $|G\rangle$ and therefore become readout qubits of the lattice (cf. Danos et al. 2009). Data of the graph state output, including breakdown by gates count and Pauli to non-Pauli ratios appear as Table 4.1.

Interpreting columns of Table 4.1 proceeds as follows,

- **circuit id**, unique id of the IQP test circuit;
- **specification**, dimensions by [ID of pattern's bottom row, south-western qubit of pattern] of Etch's $|G\rangle$ output. Recall that Etch specifies $|G\rangle$ output as a two-dimensional lattice and strictly in accordance with layout of the input circuit. In other words, layout of Etch's $|G\rangle$ output is not optimised;
- **lattice**, the total number of qubits in Etch's $|G\rangle$ output, as derived from **specification**, prior to superfluous qubits being removed by basis σ_z measurements (see Figure 4.7, below);
- **Clifford$_4$**, **CNOT$_{6\text{-}1\text{-}6}$**, **arbitrary Z-rotation$_4$**, **T/T$^\dagger{}_4$**, the count of this pattern type to appear in $|G\rangle$ as derived from IQP circuit input. The subscript to each column heading denotes the number of

[6]The Python 3.9.2 random library, used to set each circuit's qubits size, is actually a pseudo-random number generator.

circuit id	specification	lattice	Clifford$_4$	CNOT$_{6-1-6}$	arbitrary Z-rotation$_4$	$T_4/T^\dagger{}_4$	excised Z-measurements	graph state	Pauli	Pauli : non-Pauli
1	[8, 76]	608	14	6	4	7	222	386	375	34.1
2	[38, 348]	13,224	66	32	12	40	5,536	7,688	7,636	146.8
3	[22, 204]	4,488	41	18	6	22	1,818	2,670	2,642	94.4
4	[161, 1500]	241,500	323	138	39	148	103,362	138,138	137,951	737.7
5	[14, 136]	1,904	23	12	4	16	804	1,100	1,080	54.0
6	[145, 1232]	178,640	259	116	34	154	71,340	107,300	107,112	569.7
7	[154, 1420]	218,680	286	132	39	159	93,588	125,092	124,894	630.8
8	[99, 928]	91,872	182	86	26	103	39,818	52,054	51,925	402.5
9	[134, 1228]	164,552	256	114	27	135	69,882	94,670	94,508	583.4
10	[95, 892]	84,740	166	84	18	111	37,380	47,360	47,231	366.1
11	[47, 448]	21,056	76	42	14	55	9,366	11,690	11,621	168.4
12	[139, 1280]	177,920	268	118	33	135	75,402	102,518	102,350	609.2
13	[37, 376]	13,912	70	34	10	37	6,358	7,554	7,507	159.7
14	[60, 524]	31,440	112	48	13	60	12,528	18,912	18,839	258.1
15	[105, 976]	102,480	199	90	25	107	43,830	58,650	58,518	443.3
16	[121, 1100]	133,100	245	100	33	104	54,900	78,200	78,063	569.8
17	[47, 344]	16,168	86	32	8	50	5,472	10,696	10,638	183.4
18	[76, 708]	53,808	161	64	16	65	22,592	31,216	31,135	384.4
19	[138, 1196]	165,048	255	112	32	140	66,864	98,184	98,012	569.8
20	[171, 1540]	263,340	311	144	49	175	110,736	152,604	152,380	680.3
21	[63, 544]	34,272	118	50	20	56	13,550	20,722	20,646	271.7
22	[117, 1096]	128,232	227	100	32	110	54,700	73,532	73,390	516.8
23	[97, 932]	90,404	180	86	25	98	39,990	50,414	50,291	408.9
24	[109, 1048]	114,232	211	96	33	99	50,208	64,024	63,892	484.0
25	[36, 324]	11,664	56	30	9	43	4,830	6,834	6,782	130.4
26	[18, 156]	2,808	28	14	6	21	1,078	1,730	1,703	63.1
27	[87, 836]	72,732	176	76	20	81	31,692	41,040	40,939	405.3
28	[30, 268]	8,040	54	24	10	27	3,192	4,848	4,811	130.0
29	[79, 760]	60,040	150	70	12	84	26,530	33,510	33,414	348.1
30	[128, 1188]	152,064	220	112	30	148	66,416	85,648	85,470	480.2

Table 4.1: Etch layouts of graph states from randomly generated IQP circuits. Columns **lattice**, **Clifford$_4$**, **CNOT$_{6\text{-}1\text{-}6}$**, **arbitrary Z-rotation$_4$**, **T/T$^\dagger{}_4$**, **excised Z-measurements** are raw counts by property, subscripts denote the number of qubits in a pattern; columns **graph state** and **Pauli** are derived values. Each instance of input IQP circuit features every (IQP circuit) gate but in different proportions and therefore the same *combinations* of patterns appear in each graph state output. The distillation ratio of each $|G\rangle$ appears as column **Pauli : non-Pauli**.

qubits in a pattern (e.g. **CNOT$_{6\text{-}1\text{-}6}$** has 13 qubits spread over three rows: six on row 1, one on row 2, six on row 3, as discussed above). Qubits of a pattern appear consecutively in a single row with the exception of CNOT, which spreads over three rows (cf. Figure 2.4, above) and hence the separation of **Clifford$_4$** and **CNOT$_{6\text{-}1\text{-}6}$**. **T/T$^\dagger{}_4$** and **arbitrary Z-rotation$_4$** are separate on the grounds that the former is a specific instance of the latter;

- **excised Z-measurements**, the number of qubits that are superfluous to $|G\rangle$ and are to be removed by basis σ_z measurement. Column **lattice** counts include these qubits;
- **graph state**, the number of the computational qubits in $|G\rangle$, derived as **lattice** *less* **excised z-measurements**;
- **Pauli**, the number of Clifford operation qubits in **graph state**, derived as **graph state** *less* (**arbitrary Z-rotation$_4$** *plus* **T$_4$/T$^\dagger{}_4$**);
- **Pauli : non-Pauli**, ratio of Pauli to non-Pauli measurements, derived as **Pauli** over (**arbitrary Z-rotation$_4$** *plus* **T$_4$/T$^\dagger{}_4$**). This value is the distillation ratio.

The 30 circuits of Table 4.1 are a reliable sample size for the purposes of extrapolating to the population of circuits specified by circuit-etching (Gujarati 1999). With $\bar{X}$ denoting the arithmetic mean, the following statistics thus derive from the respective column headings:

- $\bar{X}$, **graph state**: 50,966.1
- $\bar{X}$, **Pauli**: 50,858.5
- $\bar{X}$, **Pauli : non-Pauli**: 362.8

None of the 30 input circuits of Table 4.1 satisfies or approximates the distillation ratio of 11:1 indeed, $\bar{X}$, **Pauli : non-Pauli** is greater than 360 and far exceeds the desired 11:1 ratio. As an indication of the number

of qubits required to satisfy the average circuit of Table 4.1, $\bar{X}$, **graph state**, comprising almost 51,000 vertices (qubits) is approximately 97-99 rows by 928-932 columns, depending upon the combination of input gates. Assuming the population of circuits specified by circuit-etching is normally distributed and $\bar{X}$ **Pauli : non-Pauli** and its standard deviation (209.0) are reliable estimators of it, the probability of a given instance of circuit-etching achieving the distillation ratio amounts to 4.63E-04.

4.3.2 Viability of circuit etching

When assessing an application to create $|G\rangle$ from the universal gates set of Clifford group + T, the central question is, 'does it efficiently process T transformations?' by which 'efficiently' is held to mean space-efficient (cost in qubits), time-efficient (cost in time) or ideally, minimised in both criteria. By the **Pauli : non-Pauli** data of Table 4.1, circuit etching is an inefficient strategy: the gap between the target distillation ratio and observed **Pauli : non-Pauli** values is invariably large. The data resolve to a minuscule probability that circuit etching could scale with or even attain the target distillation ratio. In brief, nearly all cases of circuit etching captured as Table 4.1 produce more qubits eligible for recycling than would ever be required by distillation in an error-corrected environment. Indeed, one could *halve* each reported **Pauli** value to then derive an adjusted **Pauli : non-Pauli** value and still none of the graph states would return the desired distillation ratio[7]. Moreover, each and every future improvement of magic state distillation will only lower the number of qubits required (e.g. Litinski 2019b) making circuit etching still more inefficient as a strategy for distillation. In the interest of completeness, it is worth exploring whether optimising Etch's layout patterns might have made circuit etching more competitive with ASG.

[7]Halving the **Pauli** value of two test IQP circuits - ids 1 and 5 - would place their (adjusted) **Pauli : non-Pauli** ratios at 17:1 and 27:1, respectively.

As stated above, Etch does not optimise the specification of its graph state layout but strictly adheres to configuration of the input circuit. It is worth noting that many, if not all, of the patterns laid out by Etch could be spatially or compositionally optimised. It was a deliberate decision to not manually optimise Etch's output patterns before lattice specification and resultant qubit counts: any optimisation put in place, or combination thereof, may not have been 'the best' optimisation (cf. Hietala et al. 2023) and thereby making a straw man of circuit etching. It is prudent however to estimate the benefits of 'what if' criticisms. For example, rearranging patterns to run them in parallel would increase time efficiency and reduce the size of the lattice (i.e. **specification**, Table 4.1) but would do nothing to affect the **Pauli : non-Pauli** value. Reducing the lattice size does not reduce the absolute count of patterns, which actually determines the ratio's numerator. Only reducing the number of Pauli qubits will optimise circuit etching with respect to the distillation ratio and conceivably, marginally increase space efficiency of the lattice. Taking a different tack, the **Pauli** numerator could be reduced by substituting patterns, the most obvious being the eight qubits CZ pattern replacing the (composing) 13 qubits CNOT pattern (Raußendorf, Browne, and Briegel 2003). Again the answer must be a qualified 'yes' because CNOT is not the commonest pattern in Table 4.1. Only five circuit instances have CNOT counts which tally within 50-55 per cent of counts of **Clifford$_4$** pattern in the same circuit: ids 5, 10, 11, 25 and 30. Every other circuit has CNOT pattern instances numbering fewer than half that of **Clifford$_4$** patterns. By qubits, a CNOT pattern does contribute more to the **Pauli** count than a CZ pattern but in absolute number, the CNOT pattern (and by proxy, the CZ pattern) was not the biggest contributor to **Pauli** to begin. It would seem that optimisation alone is not the key to a more competitive circuit etching strategy.

Before considering the options left for graph-state computation strategies, it must be stressed that real limitations to such strategies do not repudiate the graph-state computation model any more than they implicitly validate the circuit model. The tests of Jabalizer and Etch

outlined above only show that neither application in its current format brings an inexpensive route to resourcing T operations. The compile- and runtime difficulties revealed in Jabalizer and the inability of circuit etching to attain the target distillation ratio demonstrate the inherent complexity of creating an application that caps qubit numbers by integrating classical computing facilities or that resources T patterns with recycled qubits.

A possible future direction for Jabalizer lies in combining the ASG and circuit etching strategies. A hybrid compilation strategy in Jabalizer would be predicated upon,

1. ASG not being bound to established patterns of Clifford and non-Clifford gates; and
2. Circuit etching's capacity for composition of its input patterns.

As an application of this hybrid stategy, Jabalizer could be retooled as a *pattern factory* to assemble graph states. Recall that a pattern in cluster-state computing is simply a series of measurements that replicates the workings of a quantum gate. A pattern factory would make widgets that are not bound in scope to the canonical patterns defined by Raußendorf and Briegel (2002) but can form sequences of such patterns or if necessary, entire segments of $|G\rangle$ overall. As the worked example of Section 4.5 below will demonstrate, `tūQ` Simulator is a tool well-suited to abstracting $|G\rangle$ into its component processes. The only practical limit to size of an individual widget created by Jabalizer is the requirement for coherence times of its component qubits. This proposal shifts Jabalizer more clearly into the optimising space, as a receiver of algorithms in circuit form which it then transpiles to 'meta-patterns' similar in purpose to subroutines of classical computing. In the event that a meta-pattern/subroutine does become prohibitive by compile- or runtime, Jabalizer would split its output into smaller components, assuming that the meta-pattern/subroutine is not already in an atomic form.

The new Jabalizer would require a scheduler extension that,

(i) composes Jabalizer's widget output(s) to encode $|G\rangle$; then

(ii) passes or streams (i) to a quantum computing facility for processing.

As long as the order and basis of measurement of (ii) is consistent with that necessary to (i), the scheduling tool is consistent with MBQC principles. It is the composition property of patterns that enables streaming to work. Rules to parallel process $|G\rangle$ as an optimisation of time efficiency could be introduced (e.g. Evans et al. 2023). Whether (ii) could pass synchronously or asynchronously to a quantum computing facility is an open question.

4.4 From Etch to tūQ Simulator

The process of building then using Etch to generate metrics for the circuit-etching experiment showed the utility of,

1. an explicit graph-state syntax for directly encoding an algorithm intended for a quantum computing facility; and
2. a medium for optimising 1. by hand with the use of `tūQ` Modeller, in an iterative manner,

as addenda to the `tūQ` toolchain. Figure 4.7 is a simple demonstration of how `tūQ` Simulator and Modeller respectively layout the same input circuit; the worked example of Section 4.5, below will demonstrate how iterative use of these two `tūQ` modes can encode and optimise $|G\rangle$.

As with `tūQ` Modeller, Simulator launches as a blank canvas with a palette from which a user can encode its algorithm (see Figure 4.8, below). The user clicks on the required button of the palette to add individual measurement instructions or established patterns of the same, in any combination it requires up to a sequence of no greater than 21 x 21

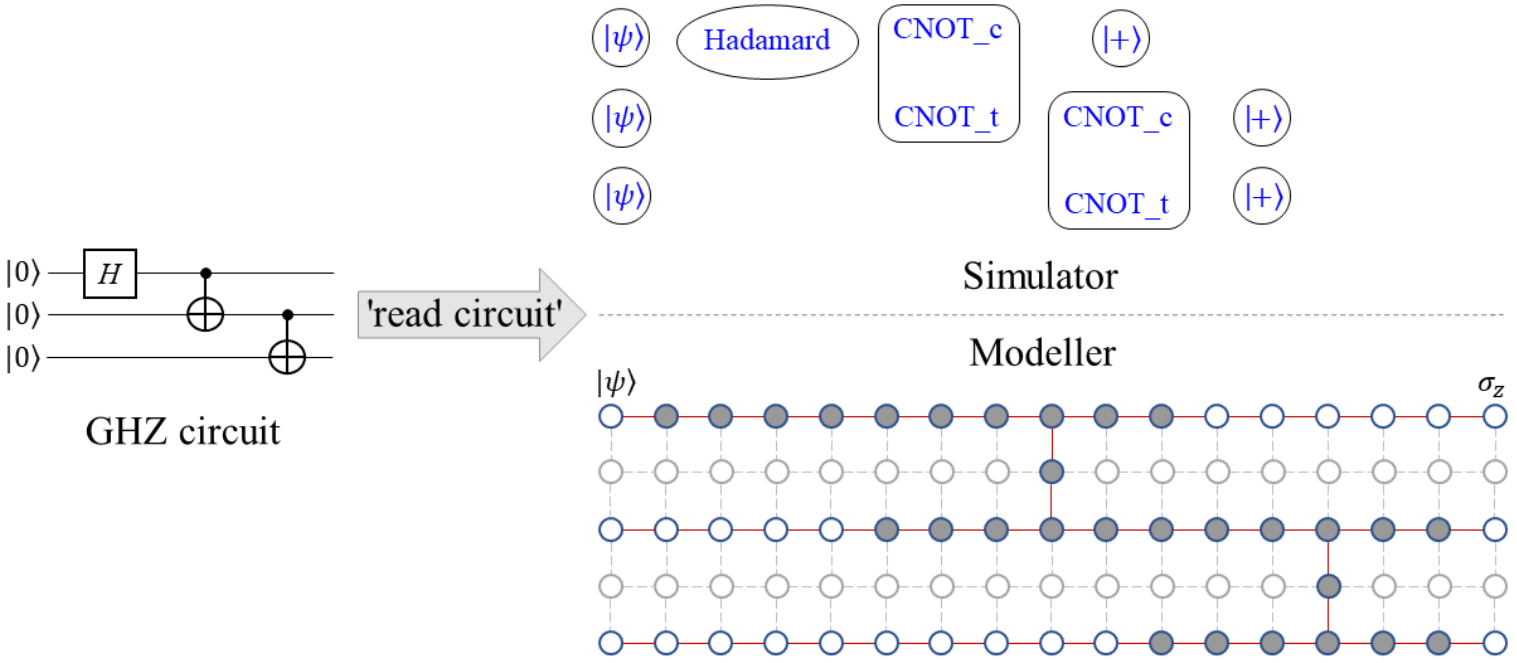


Figure 4.7: The GHZ circuit as rendered by menu function, Circuit > Read Circuit of tūQ Simulator and Modeller, respectively. Simulator and Modeller layouts are equivalent in expression but differ in abstraction. In *Simulator* layout of $|G\rangle$, circuit gates become patterns, represented in a tile-based format of rounded rectangles. Initialisation and readout columns are marked as circles tagged $|\psi\rangle$ and $|+\rangle$, respectively. Simulator's syntax is sparse and abstract denoting only the left-to-right order and basis of an algorithm's measurement. In *Modeller* layout of $|G\rangle$, qubits (vertices) appear as circles while an edge between neighbouring qubits represents entanglement. The leftmost column, labelled $|\psi\rangle$, consists of $|G\rangle$ input qubits and the rightmost column, labelled σ_z, consists of readout qubits to be measured in the computational basis. Qubits with a solid fill indicate basis σ_x/σ_y measurements while qubits with no fill, excepting the readout qubits, are measurements in basis σ_x. The fainter, 'greyed-out' qubits are removed from $|G\rangle$ by basis σ_z measurements in advance of measuring computational qubits.

tiles. Note, individual measurement instructions and patterns each represent a single tile, in other words measurements and patterns are the same scale. Simulator's tile-based syntax denotes only the order and basis of an algorithm's measurements although data of minimum $|G\rangle$ dimensions, the number of qubits in $|G\rangle$ and a count of T patterns update in real time at the application's top margin. These data provide the user a less abstracted insight to the size of the lattice required to fulfil its algorithm. Simulator writes exclusively left-to-right and north-to-south upon inserting 'CNOT ↓' or clicking 'add row'. Toggling the 'switch row' drop-down of the pattern palette allows the user to move between existing rows of its algorithm.

Simulator builds upon Etch's capacity to read a circuit-based algorithm that is encoded in JSON format. Simulator can read Google's Cirq JSON schema although its default is a JSON schema derived from IonQ's schema (`https://docs.ionq.com/api-reference/v0.3/writing-quantum-programs`) but expanded to include a 'moments' number type similar to Cirq's. A 'moment' datum is a proxy for the left-to-right ordering of measurement patterns as read into Simulator. The following gate encodings are valid input to a graph-state algorithm rendered through Simulator:

- adjacent row CNOTs, exclusively,
- $R_x(\theta)$, $R_y(\theta)$, $R_z(\theta)$,
- Hadamard (H),
- S,
- T.

As noted above, these gates are a subset of the valid input to Etch. Further, Simulator will not read a Swap gate although a user can encode three successive CNOTs as a proxy for a Swap gate. Note also the

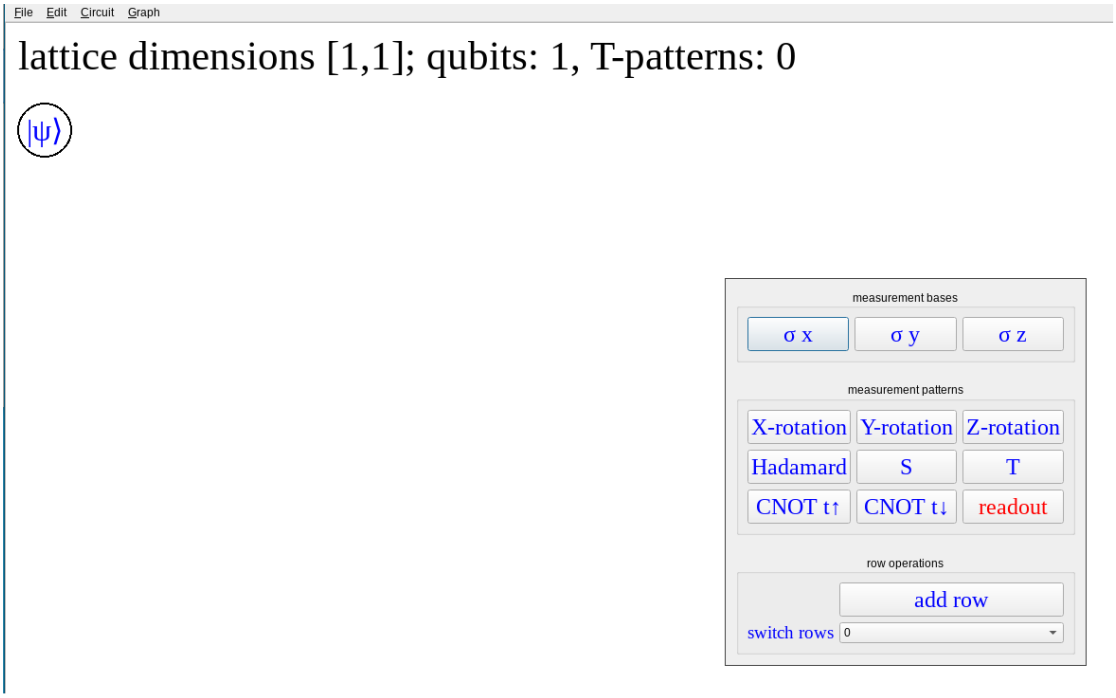


Figure 4.8: Launching `tūQ` Simulator returns a blank canvas with a pattern palette from which a user can encode its algorithm. Text at the top margin of the canvas is real time data of minimum $|G\rangle$ dimensions, the number of qubits in $|G\rangle$ and a count of T patterns. These data enable a user to visualise the size of the lattice required to fulfil the algorithm.

same 'adjacent rows only' restriction on CNOTs in Simulator as applies in Etch.

Both `tūQ` Simulator and Modeller are applications to construct $|G\rangle$ as the substrate of a quantum computation. Simulator is a tool for the user to encode an algorithm and the same algorithm can be replicated in Modeller for the user then to test reduction strategies or LC-equivalence. As noted above, the arranging and spacing of Simulator tiles is an abstraction from the (hypothetical) lattice substrate. Real time updates of $|G\rangle$ dimensions and qubits count in Simulator calibrate with the same configuration of qubits and measurement counts as stipulated by Raußendorf and Briegel (2002). Features to be discussed in Chapter 5 enable a precise conversion between Simulator and Modeller, thereby enhancing the more mechanistic worked example, below

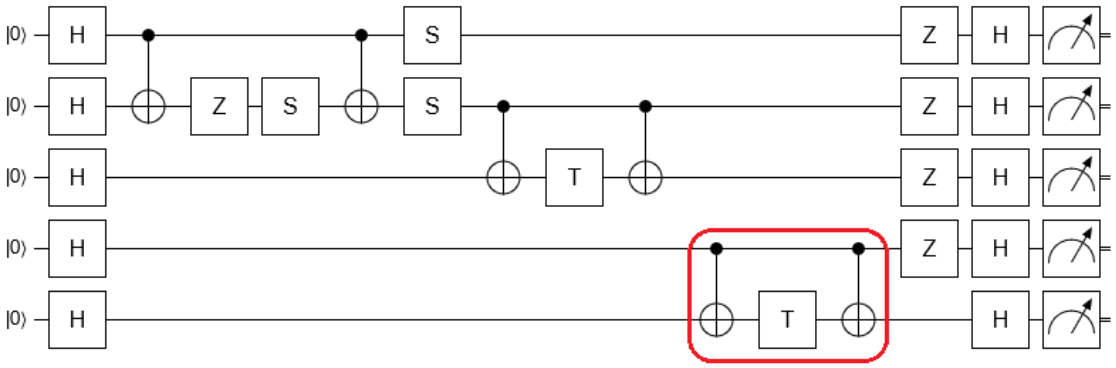


Figure 4.9: IQP circuit input read into tūQ Simulator. The red boundary marking indicates a 'floating' sequence of gates as a potential spatial optimisation of Simulator's $|G\rangle$ output.

of manually integrating output of the two modes to optimise $|G\rangle$.

4.5 Worked example 2: spatially optimising $|G\rangle$

The following is a brief example of how `tūQ` Simulator is used to draft a graph-state algorithm and to improve spatial resourcing of its lattice resource. In this instance, Simulator reads an input circuit then replaced with an optimised algorithm, written directly to Simulator using the application's tile-based syntax. Spatial optimising of the circuit-based algorithm input comes through minimising the distance of its equivalent $|G\rangle$. The Simulator video includes a demonstration of this worked example beginning at 04.09.

A randomly generated IQP circuit (Figure 4.9) serves as input to Simulator to obtain a tile syntax algorithm as output. Simulator's tile syntax version of the input circuit describes $|G\rangle$ output of dimensions [8,70] or 560 qubits, which is too large to attach as a complete image herein. Excerpts of the overall tile-based algorithm are used below to sight in specific patterns within Simulator's output of the example IQP circuit. At

pattern	**coordinates**
H	[0, 4]
H	[2, 4]
H	[4, 4]
H	[5, 4]
H	[7, 4]
CNOT	[2, 10]
Z-rotation	[2, 14]
S	[0, 28]
S	[2, 28]
CNOT	[4, 34]
T	[4, 38]
CNOT	[4, 44]
CNOT	[7, 50]
T	[7, 54]
CNOT	[7, 60]
Z-rotation	[0, 64]
Z-rotation	[2, 64]
Z-rotation	[4, 64]
Z-rotation	[5, 64]
H	[0, 68]
H	[2, 68]
H	[4, 68]
H	[5, 68]
H	[7, 68]

Table 4.2: Pattern coordinates of the input IQP circuit as read into `tūQ` Simulator by menu function, Circuit > Read Circuit.

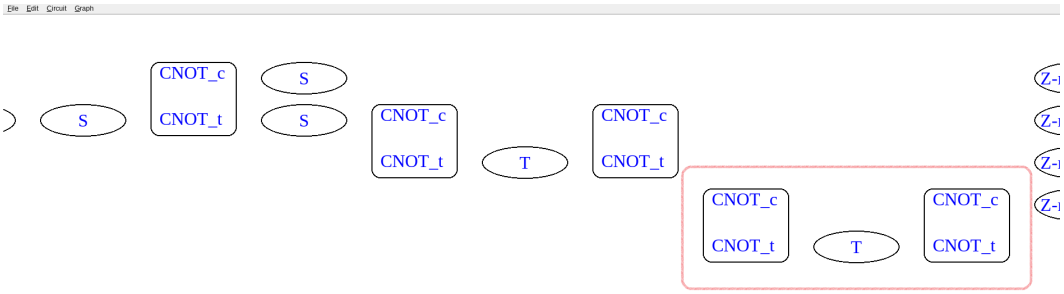
this stage of the worked example, the same zero-based coordinates system found in Etch is sufficient to gauge dimensions and configuration of the IQP circuit as transpiled to $|G\rangle$; the relevant coordinates of the input circuit's patterns appear as Table 4.2. Note, these coordinates reflect composing of patterns, after the same convention as Etch, inasmuch as they drop the readout qubits of individual patterns. As with Etch, only those patterns abutting the rightmost column of the lattice have their readout qubits counted. Finally, for the sake of the demonstration, all reported qubits numbers are *physical* qubits. This assumption does not

undermine any broader point made regarding spatial optimisation and possible information loss within $|G\rangle$.

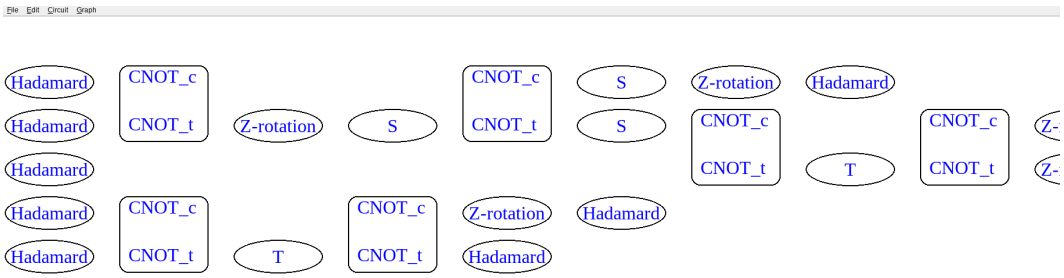
Figure 4.10a, below, is an excerpt of $|G\rangle$ as defined by Simulator's tile syntax, in turn transpiled from the IQP circuit input of Figure 4.9. The red boundary marking of Figure 4.10a encloses the patterns at coordinates [7,50], [7,54] and [7,60] of Table 4.2. This sequence of patterns 'floats' by which is meant it is non-contiguous with any pattern to precede it. By virtue of its floating property, this sequence of patterns can run as a parallel sub-process of $|G\rangle$. Rewriting the circuit algorithm directly into Simulator returns the layout as partly reproduced in Figure 4.10b. No information is lost by reformatting the IQP circuit input to the new configuration and by so doing, shrinks $|G\rangle$ to dimensions [8,53], with patterns distributed as Table 4.3, below. The reformatted $|G\rangle$ has 424 qubits or 126 fewer qubits than that read into Simulator from the IQP circuit.

A detailed discussion of integrating tūQ Simulator and Modeller appears in Chapter 5 but as a preview relevant to this spatial optimisation example, the following is a demonstration both of the initial steps of reducing the [8,53] lattice in tūQ Modeller and of the LC-equivalence discussed in Section 3.1.2, above. Figure 4.11, below, is an image of the western extremity of an [8,53] lattice as constructed in tūQ Modeller, superimposed with coloured outlines to denote the patterns of Figure 4.10b. The outline colouring in Figure 4.11 is,

- yellow = Hadamard (H),
- blue = CNOT,
- green = Z-rotation,
- red = S,
- lilac = T.



(a) An excerpt of Simulator's output of tile syntax as transpiled from the IQP circuit input. The layout of tiles is consistent both with the algorithm as defined in the input circuit and the coordinates captured in Table 4.2. The red boundary marking indicates the same 'floating' sequence of gates as highlighted in Figure 4.9.



(b) Reformatting the IQP circuit input directly using Simulator's tile syntax. By sliding the floating sequence of Figure 4.10a leftwards, the lattice substrate of the algorithm also shrinks, in this case from dimensions [8,70] to [8,53] and a reduction of 126 qubits.

Figure 4.10: Input and refit of an IQP circuit. The red boundary marking of sub-figure 4.10a encloses 'floating' patterns that can run as a parallel sub-process of $|G\rangle$. Rewriting the circuit algorithm directly into Simulator returns the layout of sub-figure 4.10b. No information is lost by reformatting the IQP circuit input to the new configuration.

pattern	**coordinates**
H	[0, 4]
H	[2, 4]
H	[4, 4]
H	[5, 4]
H	[7, 4]
CNOT	[2, 10]
Z-rotation	[2, 14]
S	[0, 28]
S	[2, 28]
CNOT	[4, 34]
T	[4, 38]
CNOT	[4, 44]
CNOT	[7, 10]
T	[7, 14]
CNOT	[7, 24]
Z-rotation	[0, 48]
Z-rotation	[2, 48]
Z-rotation	[4, 48]
Z-rotation	[5, 48]
H	[0, 53]
H	[2, 53]
H	[4, 53]
H	[5, 53]
H	[7, 53]

Table 4.3: Pattern coordinates from writing the algorithm encoded in IQP circuit directly into Simulator. This is a more compact representation of the circuit without information loss.

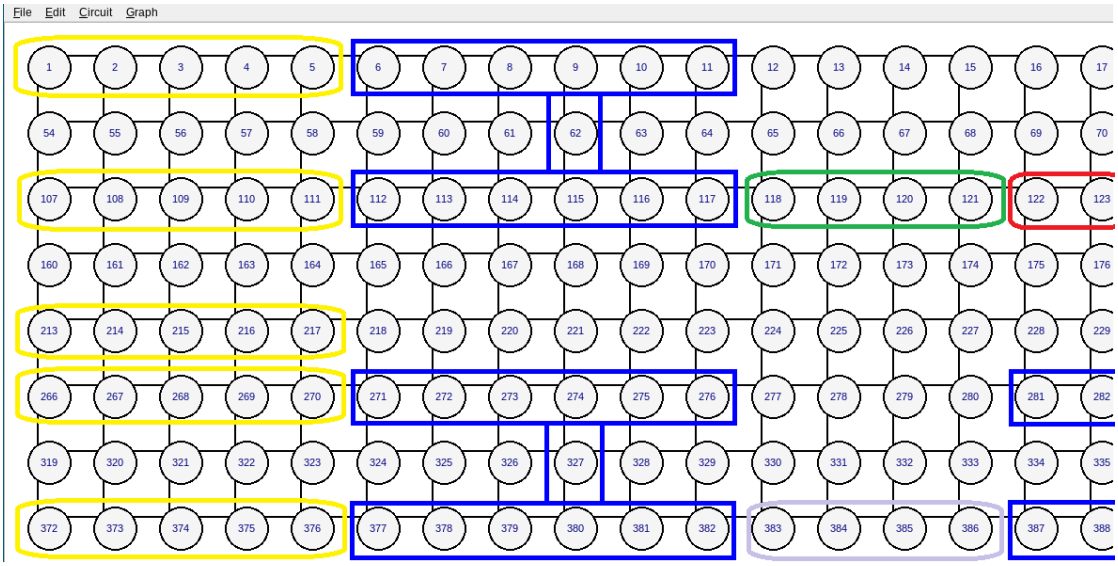


Figure 4.11: Western extremity of an [8,53] lattice as constructed in tūQ Modeller. The lattice is a requirement of computing the algorithm written directly in Simulator's tile syntax. The coloured outlines superimposed upon the lattice indicate the patterns specified in Simulator's algorithm.

As demonstrated in Section 3.2.1, above, tūQ Modeller enables the user to reduce $|G\rangle$ through measurement functions upon individual vertices (qubits). Modeller redraws its edges with each measurement function in a demonstration of teleporting state. Continuously reducing $|G\rangle$ by individual measurements is the crux of MBQC and an approximation of how a processor of a quantum computing facility would effect computation. Figure 4.12, below, is a snapshot during consecutive measurements of the Hadamard patterns at coordinates [0,4], [2,4] and [4,4] of Table 4.3 and at the corresponding rows of Figure 4.11. Note also how measurement in basis σ_z has removed qubits and entanglements superfluous to $|G\rangle$ in advance of computation beginning.

The point of this manual exchange from tūQ Simulator to Modeller is to demonstrate how Simulator-Modeller integrate as part of the same toolchain: read a circuit into Simulator, rewrite it, try the rewrite in Modeller and repeat with each iteration an exercise in reducing the number of required qubits. Each iteration of the cycle initialise-

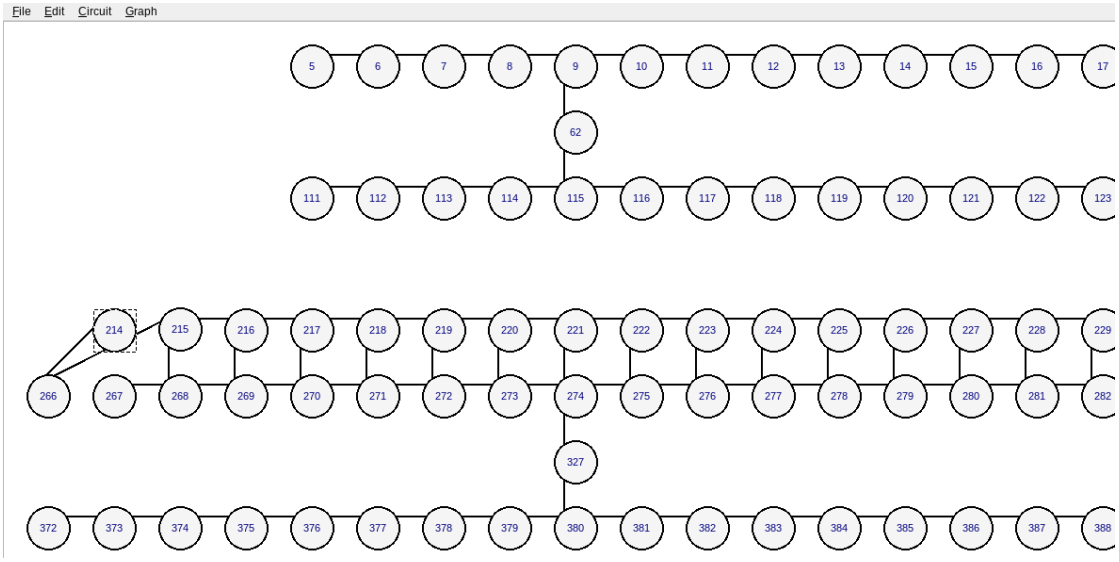


Figure 4.12: Reducing the Hadamard (H) at coordinates [r,c] through measurements of individual qubits. Measurements are consistent with those prescribed by Raußendorf and Briegel (2002). Note how measurement in basis σ_z has removed qubits and entanglements superfluous to $|G\rangle$ before computation has begun.

transform-read reproduced by tūQ Simulator and Modeller brings its user closer to runtimes of exact initialisation of smaller graphs and therefore, fewer qubits.

4.6 Summary

As with Modeller, tūQ Simulator's core purpose is to minimise the number of qubits needed to resource $|G\rangle$. Whereas traditional quantum emulators exist to mimic the workings of a quantum computing facility, tūQ Simulator is a module of a toolchain and exists both to write algorithms and to specify the graph state required to compute such algorithms, in real time. Perhaps the most notable point of divergence of Simulator from traditional emulators is that it works in an alternative notation to the circuit.

Simulator originated with another research tool named Etch, which was bespoke to an enquiry into circuit etching as an alternative optimising strategy to the algorithm-specific graph (ASG). While it is possible to think of Simulator as a later version of Etch, there are three fundamental differences between the applications namely, Etch:

1. reads gate encodings that are otherwise inadmissible to Simulator;
2. input is limited to circuits encoded in JSON format;
3. output consists solely of $|G\rangle$ dimensions and a zero-based coordinates system for patterns found within it, with no optimisation.

The proposition that circuit-etching could be an alternative to ASG turned on what was termed the 'distillation ratio' or the proportion of Pauli qubits to non-Pauli qubits situated in a given $|G\rangle$. Data obtained by processing thirty (30) randomly-generated IQP circuits through Etch do not make a case for circuit-etching as a viable alternative to ASG for error-corrected compilation and assuming the sample's distribution of $|G\rangle$ reflects a population that is normally distributed, the probability is negligible that circuit etching could attain the desired distillation ratio.

Simulator shares a modified version of Etch's circuit reading function but otherwise, Simulator by functionality is a superset of Etch. The user of `tūQ` Simulator can,

- read circuit-based input, albeit circuits comprised only of Clifford group operators plus the non-Clifford operator, T ($\frac{\pi}{4}$ rotation),
- directly encode an algorithm through a tile-based syntax,
- by transpiling a given $|G\rangle$ to Modeller, manually toggle between `tūQ`'s Simulator and Modeller modes to optimise an algorithm by minimising the distance of $|G\rangle$.

A brief demonstration of the last point in which a lattice of dimensions [8,70] as defined by an input IQP circuit was reduced to a lattice of dimensions [8,53] *without information loss* closed this introduction to tūQ Simulator.

Basic manual operations by a user make it clear that tūQ's Simulator and Modeller modes are complementary: Simulator abstracts away a lot of the detail and thereby enables the user to concentrate on the *logic* of an algorithm. At the same time, Modeller can work as a kind of drill-down facility, enabling the user to track how an algorithm resolves 'under the bonnet', in part or in its entirety. Modeller and Simulator lend finer or coarser abstraction to the same $|G\rangle$. Chapter 5 is a discussion of functions that integrate the tūQ toolchain and facilitate a less labour-intensive version of Worked Example 2.

Chapter 5

Synthesis: tūQ Modeller and Simulator modes

Each of the two previous chapters introduced a mode of the `tūQ` compiler toolchain, engineered to address a fundamental element of graph-state computing. Modeller mode enables the user to manipulate graph states of vertices (qubits) and edges (interactions) as a representation of bare metal transformation of computational state. Simulator mode enables the user to draft graph-state algorithms and focus on their logic by abstracting away from the component level native to Modeller. These use cases imply a degree of integration between `tūQ`'s Modeller and Simulator modes that may not have been evident from the introductory worked examples of Chapters 3 and 4 or the function documentation in Appendix B, 'Using tūQ'; nevertheless `tūQ` Modeller and Simulator are complementary.

This chapter begins with worked examples of the as-yet-unexamined menu functions,

- (Modeller mode) Open Algorithm; and
- (Simulator mode) Compile.

Both of the Worked Examples below begin with a graph-state algorithm as drafted in Simulator, consistent with the the worked examples of

Chapter 4. Worked Example 1 is a demonstration of transposing the algorithm to and reducing it within Modeller while Worked Example 2 is a demonstration of transpiling the algorithm to OpenQASM 3.0 format. These demonstrations then lay the basis for a broader evaluation of whether the integrated `tūQ` toolchain satisfies the minimum requirements to qualify as a compiler.

The two worked examples to follow will start with the optimised IQP circuit of Figure 4.10b and its lattice coordinates of Table 4.3 above. This is called the 'base algorithm' for convenience. The Modeller mode function, 'Open Algorithm' will read the base algorithm from its saved .txt file format, an abridged version of which appears as Figure 5.1, below. The Simulator mode function, 'Compile' takes the base algorithm as an input stream for transpiling to OpenQASM.

Original material

- The source code for `tūQ` Modeller and Simulator may be inspected at URL: `https://github.com/QSI-BAQS/tuQ`. References to the integrating functions 'Open Algorithm' and 'Compile' appear in this chapter.
- Sections 5.1 and 5.2 represent original work that appears in Bowen and Devitt (2025).
- As with Chapters 3 and 4, there is a supplementary video for this introduction to `tūQ`'s integrating functions, entitled
  - A Toolchain for Cluster-State Architecture Integrate.

  The video is a 'live' introduction to each of the integrating functions appearing as worked examples of this chapter.

```
south 8
east 53
ψ 0 0
Hadamard 0 1
CNOT t↓ 0 2
CNOT t↓ 0 5
S 0 6
Z-rotation 0 7
Hadamard 0 8
ψ 1 0
Hadamard 1 1
Z-rotation 1 3
S 1 4
S 1 6
CNOT t↓ 1 7
CNOT t↓ 1 9
Z-rotation 1 10
Hadamard 1 11
ψ 2 0
Hadamard 2 1
T 2 8
Z-rotation 2 10
Hadamard 2 11
ψ 3 0
Hadamard 3 1
CNOT t↓ 3 2
CNOT t↓ 3 4
Z-rotation 3 5
Hadamard 3 6
ψ 4 0
Hadamard 4 1
T 4 3
Hadamard 4 5
```

Figure 5.1: An abridged version of the algorithm of the refitted IQP circuit of Figure 4.10b, above which serves as the base algorithm for the worked examples of Sections 5.1 and 5.2, below. Notations are as follows: 'south 8' and 'east 53' are the [row, column] dimensions, [8, 53] of the lattice; all other rows have the format, ψ/**pattern row column** (e.g. ψ 1 0, is input tile at row 1, column 0; CNOT t↓ 0 2, is CNOT tile with *control* at row 0, column 2).

5.1 Worked example 1: Simulator-Modeller integration

The worked example of Section 4.5, above was a preview of integrating tūQ Simulator output with Modeller's functions. In that example, output of Simulator, saved in .txt format, specified both the minimum underlying lattice needed to resource the algorithm and the position of the algorithm's patterns within the lattice. Figure 4.11 was a demonstration of a (Modeller) lattice as defined by a Simulator algorithm but with the patterns of its defining algorithm manually superimposed as coloured outlines. Modeller's menu function, 'Open Algorithm' exists to automate and extend the same procedure.

Opening tūQ Modeller then, selecting function File > Open Algorithm from the menu bar then further selecting the base algorithm introduced above returns a canvas of which the western extremity appears as Figure 5.2a, below. The Modeller function 'Open Algorithm' reads an algorithm drafted and saved (.txt format) in Simulator to reproduce it as the equivalent $|G\rangle$. The lattice layout is modified to highlight target qubits of the algorithm to allow the user to reduce it manually as a preview of how a quantum computing facility would process the same.

The steps of the Open Algorithm function include,

i. creating the minimum required lattice, then

ii. situating the pattern or basis $\sigma_x/\sigma_y/\sigma_z$ measurement qubits within the lattice; and

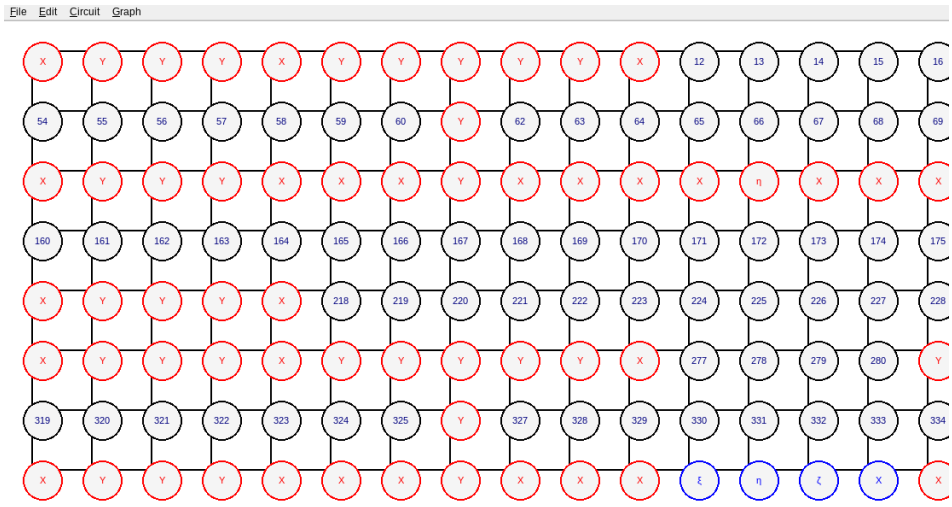
iii. highlighting each Clifford pattern or basis $\sigma_x/\sigma_y/\sigma_z$ measurement qubits in *red*. The single non-Clifford T-pattern highlights in *light blue*; and

iv. specifying each measurement of the pattern that is needed to replicate the equivalent (circuit) gate, in place of a vertex (qubit) integer ID. Rotation patterns around the x-/y-/z-axis accept a

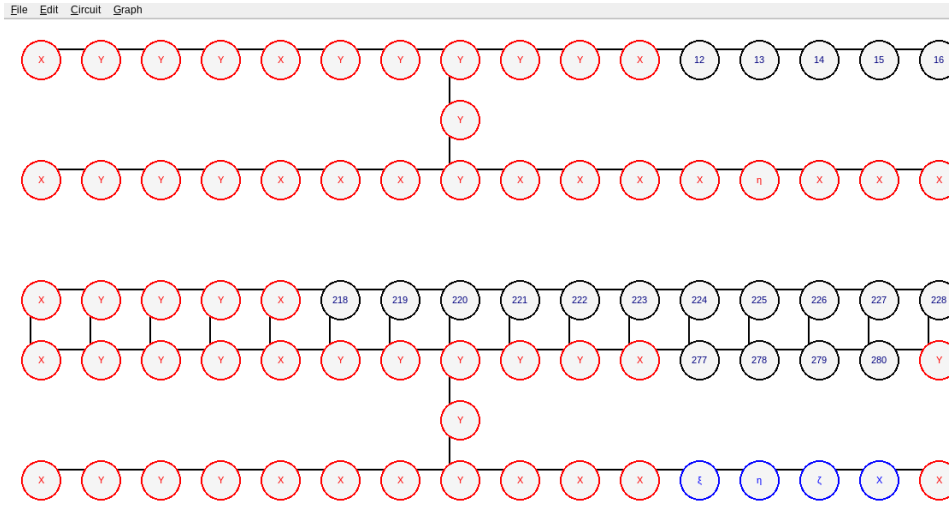
manual specification of θ (see Section 5.2, below) while the 'general rotation' pattern as supplied by Raußendorf and Briegel (2002) stands in for the non-Clifford T-pattern.

All other vertices (qubits) appear in Modeller's standard format with the customary integer ID. The user must decide on the appropriate measurement of these 'standard' qubits and therefore it is a manual process in tūQ Modeller. After Raußendorf and Briegel (2001), any non-target, 'surplus' qubit is removed from the lattice in advance of computational measurements, by measuring the qubit in basis σ_z. This treatment includes qubits on the same row as the 'nexus' qubit that connects the upper and lower rows of a CNOT pattern. The example of Figure 5.2b includes the second row of Figure 5.2a after successive measurements in basis σ_z has removed qubit IDs 54 through 60 then 62 through 69 at either side of the basis σ_y nexus qubit. Any standard qubit that is left after removing surplus qubits through measuring in basis σ_z is to be measured in basis σ_x, bearing in mind that qubits in a lattice are ordinarily initialised in state $|+\rangle$ (see Section 3.2.1, above). Qubit IDs 12 through 16 of row one of Figure 5.2b are targets for measurement in basis σ_x. The sole contribution to the overall computation realised by measuring standard qubits in basis σ_x is to teleport state left-to-right through the lattice.

For all of the features listed above, the worked example of Section 5.2, below will also expose some of the limits to the integration that 'Open Algorithm' function brings to tūQ Modeller and Simulator modes. For example, one step of Simulator's 'Compile' function provides for the user directly to input radians of a rotation pattern about the x-/y-/z-axis whereas Modeller's operations do not extend to rotating a qubit. Similarly, the presence of a T-pattern in a Simulator algorithm is a standard gate instruction to OpenQASM for the processor then to action. In contrast, Modeller's operations are predicated upon $|G\rangle$ as a stabiliser state, which leaves the non-Clifford T-pattern effectively insuperable and further, Modeller has no capacity for enacting magic state distilla-



(a) The 'Open Algorithm' function lays out pattern or basis $\sigma_x/\sigma_y/\sigma_z$ measurement qubits within the lattice, highlights patterns as Clifford (red perimeter) or non-Clifford (light blue perimeter) and displays the component measurements of a pattern in place of the usual vertex (qubit) integer ID.



(b) Manual steps to follow the Open Algorithm function include removing superfluous qubits of the lattice by measuring in basis σ_z (cf. Figure 4.12, above). Any 'standard' qubit that precedes or follows a qubit highlighted by the function is to be measured in basis σ_x as part of computing through the lattice. This operation replicates the left-to-right teleporting of state through the lattice, assuming that qubits of the lattice were initialised in state $|+\rangle$.

Figure 5.2: Western extremity of the lattice specified by the base algorithm and rendered by tūQ Modeller function, 'Open Algorithm', showing automated and manual steps of preparation.

tion. When evaluating these limits to integration, the user should recall that one of Modeller's use cases is to enable reasoning about the accumulation of state; it is not intended as a tool for initialising or specifying state (see Section 3.2, above). As a practical instruction, Modeller's user should resolve non-compliance of a qubit by measuring in basis σ_z then keep going. Finally and as already implied, Modeller has no provision to encode an algorithm imported to, or created or modified within it in a low-level language compatible with a processor: that function lies solely in the domain of tūQ Simulator.

5.2 Worked example 2: compiling $|G\rangle$ to OpenQASM

To paraphrase a statement from Chapter 2 introducing the concept, a compiler exists to simplify the act of operating a processor. As an approximation of the same, tūQ Simulator's 'Compile' function will encode a tile-based algorithm in the quantum assembly language, OpenQASM. Recall that classical assembly language is a low-level syntax, which translates directly to machine code and that NISQ service providers use its quantum equivalent, usually OpenQASM, like an API for direct instructing of their respective QPUs. The aim for the Compile function is thus to render an algorithm drafted for $|G\rangle$ as computable on a QPU configured for OpenQASM. This is necessarily a proxy step as all quantum computers do not natively use graphs. Compiling to an OpenQASM circuit formalism is the one option for computation of an algorithm as at December 2024. The practicalities of computing a graph-based algorithm through a circuit-based architecture are considered in Section 5.3, below.

Having opened Simulator then either opened a saved algorithm or drafted one within the live session, the user can select function Circuit > Compile from the menu bar. For this example, opening the (saved)

```
// tuQ graph state to OpenQASM
OPENQASM 3.0;

include "stdgates.inc";

qubit q0;
qubit q1;
qubit q2;
qubit q3;
qubit q4;

// initialise qbits
reset q0;
reset q1;
reset q2;
reset q3;
reset q4;

h q0;
h q1;
h q2;
h q3;
h q4;
cx q0 q1;
cx q3 q4;
rz (0.75) q1;
t q4;
s q1;
cx q3 q4;
cx q0 q1;
rz (0.2) q3;
h q4;
s q0;
s q1;
h q3;
rz (0.5) q0;
cx q1 q2;
h q0;
t q2;
cx q1 q2;
rz (0.5) q1;
rz (0.25) q2;
h q1;
h q2;

// measure qbits
measure q0;
measure q1;
measure q2;
measure q3;
measure q4;
```

Figure 5.3: The base algorithm transpiled to OpenQASM 3.0 and linked to the library 'stdgates.inc' for gates logic.

base algorithm from .txt format will return the same configuration of tiles as appears in Figure 4.10b, above. Assuming the function is fully configured[1], the Compile function will,

1. encode the base algorithm to OpenQASM and a copy of the script will write to a .txt file for inspection. This is the default setting of the Compile function. Figure 5.3, below is a demonstration of the script for the base algorithm; and

2. POST the script to the supplied (REST) API. This is an optional setting of the Compile function. Briefly, a POST request is an HTTP method to obtain data from a server without updating the server's resource in part or in full. With reference to this function, the API must conform with OpenQASM 3.0 syntax for POST to succeed.

With reference to Figure 5.3, the script specifies 'OpenQASM 3.0' as the version in use and links to the library 'stdgates.inc' for gates logic (Cross et al. 2022). Note that OpenQASM 3.0 requires a qubit variable is first, declared then second, initialised ('reset') as separate steps. Note also that x-/y-/z-rotation patterns are specified as rotation gates rx/ry/rz with the relevant θ of the rotation also defined. The Compile function enables the user to supply the rotation θ through a dialog box, as shown in Figure 5.4. Text that specifies the x-/y-/z-rotation pattern and its [row, column] position in the algorithm appears in the Compile dialog to remove ambiguity that might otherwise arise when transpiling a large algorithm with multiple rotations. Refer to the video link of this chapter for a walk-through of passing the base algorithm to `tūQ` Simulator's 'Compile' function and the output returned by the Quokka emulator (`www.quokkacomputing.com`).

[1] As per Appendix A, the extent of output returned from passing an algorithm to the Compile function will depend upon how the user configured `tūQ`'s source code at installation.

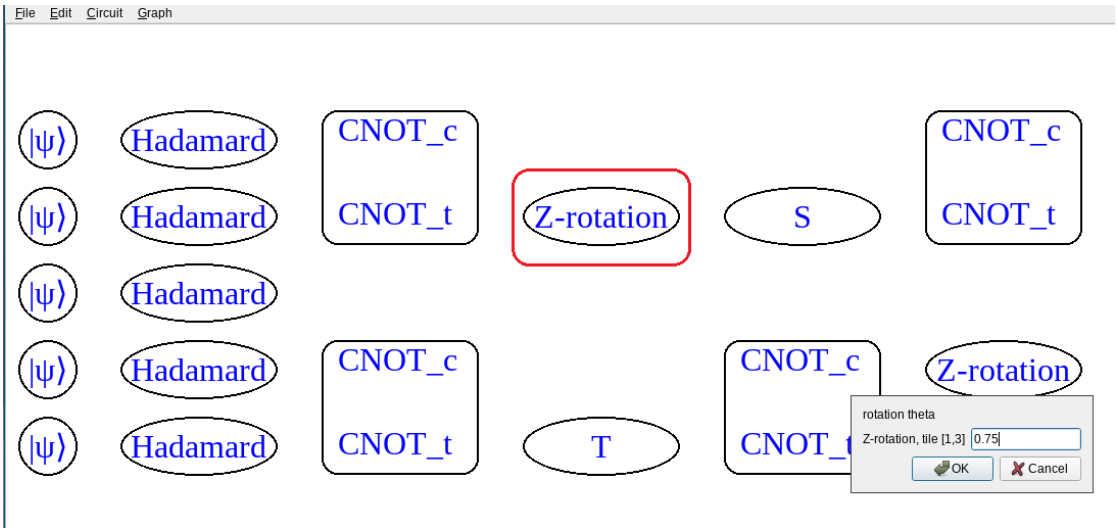


Figure 5.4: Direct input of θ as part of the Simulator function, 'Compile'. In this example, the input dialog matches the Z-rotation tile at position [1, 3], as outlined in red, with an input field for the user to set the requisite θ.

5.3 A working compiler?

The essential functions of a compiler, as represented in Figure 2.6 above, divide between,

- Front end: check input syntax is well-formed, lex and parse input to an intermediate representation;
- Optimisation: iteratively improve spatial/temporal efficiency of the IR;
- Back end: map the optimised IR to the instructions set of the processor.

As a high-level list of requirements, to realise a compiling framework requires *at least*,

1. a set of text-based operators and data structures intended for a human to compose instructions to the processor;

2. quality assurance of input formed from 1. including checks of syntax and semantics;

3. machine code output matched to the processor's instructions set.

These are minimal requirements and merely hint at the desirability of any optimising processes of Figure 2.6. The question considered below is whether the integrated `tūQ` toolchain satisfies requirements 1-3 in full, in part or at all.

When considering the effectiveness of the `tūQ` toolchain, it serves to recap the state of cluster-state computing as an alternative to circuit-based architecture at the time of writing. As at December 2024, a researcher wishing to pursue cluster-state computing has no access to,

(i) a QPU in production,

(ii) assembly language with which to instruct (i),

(iii) any optimising algorithms conducive to cluster-state operations; and further has,

(iv) few formalised higher-level languages for drafting an algorithm while none transpiles to an assembly language of any description.

Itemising the reasons for this under-representation of the cluster-state computing paradigm in theory and design is beyond the scope of this discussion but if nothing else, `tūQ` as a project demonstrates that impracticality of building a compiling toolchain is not one of them.

With the `tūQ` toolchain a user can,

- read a circuit and convert it to $|G\rangle$,
- draft an algorithm in a syntax compatible with cluster-state principles,

- reduce $|G\rangle$ in a sandpit environment to identify possible optimisations of an algorithm; and gain insights of how the computation might resolve, given a processor built to implement the graph-state model,
- transpile an algorithm to OpenQASM 3.0.

Much of this functionality overlaps with that of the architecture layer[2] of a compiling stack (cf. Chong, Franklin, and Martonosi 2017; Beverland et al. 2022), operating at the level of individual measurements of qubits or such measurements grouped as patterns. As a *proof-of-concept* toolchain and with caveats discussed below, `tūQ` does satisfy compiler framework requirements 1 and 3 but partly satisfies requirement 2.

A classical compiler relies upon data structures, especially IRs and a capacity to copy them as part of the optimising process. In comparison, a wholly quantum computer-based compiler is more restricted in its architecture. The no-cloning theorem precludes copying of quantum state (cf. Selinger and Valiron 2009), which frustrates efforts to realise a quantum IR and an optimising strategy based upon it as devices of a quantum compiler. The no-cloning problem means that any quantum state datum required at a later point in processing *must* write to classical memory. Consequently, a classical computer is needed as a control mechanism to a NISQ 'co-processor' (e.g. Nguyen et al. 2020). Such efforts to extend a classical compiler to work with a NISQ co-processor are often architecture-specific (e.g. Murali et al. 2020; Zhang et al. 2023), which begs the question of whether a compiler that is not architecture-portable counts as a compiler. Regardless, the no-cloning restriction equally imposes limits upon the `tūQ` toolchain. As shown above and in Chapter 4, optimising an algorithm through `tūQ` is an iterative process: the user can toggle between Simulator and Modeller to tighten an algorithm but it is inescapably manual in effort.

[2] Also known as the microarchitecture layer.

As noted above, tūQ Simulator admits only single-qubit measurements, Clifford patterns and the non-Clifford t-pattern to its algorithm syntax and further, tūQ Modeller allows only measurements in basis σ_x, σ_y or σ_z; nevertheless, these restrictions are not equivalent to the lexing and parsing functions of a classical compiler. That said, lexing and parsing are necessary to a higher abstraction of syntax than that presently available for drafting a quantum algorithm in general. As follows from statements made above concerning optimisation in a quantum compiler, there is no recourse to such abstractions as data structures or conditional statements when drafting a quantum algorithm. The expressiveness of an algorithm drafted in Simulator-Modeller is closer to the lower-level instructions of OpenQASM than an algorithm written in a classical language like C. In short, the front end classical compiling functions of checking for compile-time errors, lexing and parsing are superfluous to expressing an algorithm at circuit- or graph-level. This is not to say that errors of logic or expression cannot arise in a quantum algorithm, only that these errors tend to manifest at runtime and against which classical compilers also provide limited protection. Therefore, it is reasonable to assert the tūQ toolchain partly satisfies requirement 2: the limited expressiveness of its syntax also limits but does not eliminate the means of admitting a compile-time error.

The degree to which the toolchain satisfies an output of machine code is more nuanced: tūQ does transpile to OpenQASM but clearly that instructions set is intended for a gate-based processor. Transpiling a graph-state algorithm to a circuit-based processor at the very least carries a risk of information loss from the algorithm. As an example of such, it is possible to compose individual measurements into patterns or even the pattern factories broached in Section 4.3.2, above; gates, on the other hand, do not compose. OpenQASM necessarily imposes circuit-model algorithms on a cluster-state toolchain. A broader view is to subordinate the discrepancy between graph-state algorithm and circuit-based processor to the outcome of an algorithm obtaining meaningful output *within these restrictions*. Furthermore, an assembly

language organised around the cluster-state model may not exist as at December 2024 but should that variant eventuate, tūQ's Compile function can refactor from OpenQASM to it. In other words, tūQ paired with OpenQASM is proof-of-concept compiling that showcases output from a graph-state algorithm.

The final, overarching consideration, for quantum compilers in general, is compensating for the probabilistic nature of measuring quantum state. In regard to the MBQC framework, applying extra rotations in basis σ_x or σ_z to the relevant readout qubit is sufficient to correct $|G\rangle$ for randomness of measurements (Raußendorf and Briegel 2002). To effect the correcting σ_x/σ_z rotations to readout qubits requires a log of measurements of every qubit in the system. This log, known alternately as a Pauli- *tracker* (Paler et al. 2014) or *frame* (Riesebos et al. 2017), comprises of (Pauli) records of every measurement of any qubit to teleport state to the readout qubit[3] under consideration (cf. Knill 2005; Gottesman 2008; Terhal 2015). A Pauli tracker library, compatible with the graph-state computing model, is available (Ruh and Devitt 2024) and although it can be linked to the toolchain, it is not included in version 0.1 of tūQ. This is in large part because of time constraints in assembling the tūQ source code but is also a result of tūQ's Compile function being paired with OpenQASM and by extension, a gates-based processor that essentially renders Pauli tracking redundant.

5.4 Summary

This chapter began with worked examples of the tūQ menu functions, 'Open Algorithm' and 'Compile'. The Modeller function, Open Algorithm automates and extends the manual procedure demonstrated in Section 4.5, above whereas Simulator's Compile function renders an algorithm drafted for $|G\rangle$ computable on a QPU configured for OpenQASM.

[3]Danos *et al.* (2009) refer to this aggregating of measurements as a *signal*.

Both worked examples draw on the base algorithm from the optimised IQP circuit of Figure 4.10b, above as demonstrations of how tūQ Modeller and Simulator modes are complementary in their use cases. These integrating functions also make clear the delineation of Modeller and Simulator's use cases: the user can test optimisations of an algorithm through Modeller but drafting and compiling an algorithm, at least as far as transpiling it to assembly language, is restricted to Simulator only. These worked examples wrap up the tour of tūQ functionality but also prompt a review of the toolchain in totality with respect to its status as a compiler.

The minimal set of requirements both to realise a compiler and to evaluate tūQ's legitimacy as such includes,

1. a set of text-based operators and data structures intended for a human to compose instructions to the processor;
2. quality assurance of input formed from 1. including checks of syntax and semantics;
3. machine code output matched to the processor's instructions set;

whereas the tūQ toolchain,

- delivers tile-based operators intended for a human to compose an algorithm,
- works at a lower level of operations with fewer and more-limited operators than would necessitate the lexing and parsing functions of a classical compiler,
- transpiles an algorithm to OpenQASM 3.0, showcasing the possibility of output from a graph-state algorithm.

Moreover, linking to a Pauli tracker library such as that proposed by Ruh and Devitt (2024) would ensure a computation through the toolchain

is deterministic and would be meaningful at some point in the future when an assembly language that encompasses graph-state operators becomes available. All of this within an industry and research community in which the circuit-model is unequivocally dominant and the graph-state model under-represented. As at December 2024, research into graph-state computing is hindered by an absence of for-purpose hardware, assembly language and optimised algorithms; and comparatively few formalised higher-level languages with which to draft a graph-state algorithm.

Chapter 6

Conclusion

Innovation relies on competition. In the September 2024 product update entitled, 'Qiskit: The most performant (*sic*) quantum software development kit'[1], IBM announced,

> 'Using a new open-source suite of over 1,000 benchmarking tests developed by leading universities, national labs, and researchers at IBM, we found that Qiskit is second-to-none in terms of the speed and quality demonstrated for most test tasks.
>
> 'When measured against the closest leading quantum software development kits, Qiskit showed its overall speed in both its ability to build and manipulate quantum circuits, as well its speed in synthesizing and transpiling them.
>
> 'In regards to quality, Qiskit transpiled circuits with the lowest number of 2Q gates — meaning it needs to perform far fewer operations when running circuits on hardware, thereby generating less noise and superior results.

[1]URL: `https://docs.quantum.ibm.com/announcements/product-updates/2024-09-17-performance`.

> 'Finally, Qiskit was the only software that was able to complete all tests that can be completed using current quantum hardware...'

The product update closed with advice of a new open-source package, `Benchpress` as the benchmarking suite used for these tests.

The `tūQ` toolchain introduced in this dissertation is a GUI-based package for modelling, optimising and compiling a cluster-state algorithm. Clearly, `tūQ` is a proof-of-concept toolchain: it is not a production-grade release and has neither the presence of Qiskit across NISQ computing service providers nor a crowd-sourced team of developers and testers to manage a schedule of regular releases. What the `tūQ` toolchain does bring to quantum computing is competition: version 0.1 is a stable package and like Qiskit, it is open source; most importantly, `tūQ` may yet foster alternative avenues of quantum computing research and perhaps lay the groundwork for similarly alternative engineering solutions, such as a processor with instructions that include operations fundamental to the MBQC model.

6.1 Summary

This dissertation, in the shape of its main artefact, `tūQ`, is a solution to the research question of how to design a sufficient quantum compiling toolchain that supports the cluster-state computing paradigm. The significance of this research, as noted above, lies in `tūQ` being a proof-of-concept compiling toolchain, specifically engineered for a cluster-state computing architecture.

Following a discussion of terminology and nomenclature, Chapter 2 opened with a review of the principles of quantum computing. This included defining the set of quantum gates required to obtain universality of operations namely, Clifford group operators plus the non-Clifford $\frac{\pi}{4}$

(T) rotation. This set was a component of all subsequent worked examples of this dissertation.

The principles of quantum computing were a foundation for a detailed analysis of the equivalent cluster-state and graph-state models. Reviewing the cluster-state model was the first step in making a case for the tūQ toolchain. Computing through a cluster-state, $|\phi\rangle$ works by measuring single qubits in a specific order and basis and relies upon propagating state though $|\phi\rangle$ by means of CZ gates and quantum teleportation. The cluster-state model and specifically MBQC is sufficiently different from the gate-based operations of the circuit model to warrant a bespoke compiler.

A compiler of a classical computing facility exists to translate human-friendly programming languages to the machine code required by processors. A compiler will check an input algorithm for syntax and semantics at compile-time but provides limited cover against errors at runtime. For the most part, a compiler works with intermediate representations, data structures for the purpose of improving the spatial and temporal efficiency of the algorithm. Optimising through IRs cannot involve information loss from the input algorithm. It was also observed that algorithms in circuit- or cluster-state format are approximately as expressive as IRs.

This analysis of classical compiling frameworks set the background for a review of the quantum computing landscape as at December 2024, which consisted entirely of NISQ computers and was dominated by circuit-model architecture, extending to the proprietary quantum emulators of NISQ computing service providers. Furthermore, many standalone emulator projects, available as SDKs or hosted services, are configured to integrate with a circuit-based architecture. The hypothetical 'quantum computing facility' was defined in counterpoint to contemporary NISQ computers and appears repeatedly as a point of reference for the remainder of this dissertation.

A quantum computing facility of the future will work in parallel

with a classical computing facility, which will make new demands on compiler technology. A high-level programming language to operate a classical-quantum parallel architecture may require two lexicons and one grammar, a potential obsolescence and seldom addressed in current compiler offerings. Under such circumstances, a new compiler model promoting an alternative paradigm of quantum computation seems timely and Chapter 2 closed with a list of requirements for that compiler.

Chapters 3 to 5 are an extended case study of the tūQ toolchain. Chapter 3 is an introduction to tūQ Modeller, a tool for modelling graph-based quantum algorithms using classical computing facilities. Furthermore, the first of three videos accompanying this dissertation is an introduction to tūQ's Modeller mode. Chapter 3 is a continuation of the discussion of the cluster state but in this instance as an analysis of its equivalent, the graph state, $|G\rangle$. The graph-state properties of local complementation ('LC') and LC-equivalence and their significance to transforming $|G\rangle$ are reviewed; as is the concept of local unitary equivalence.

Modeller enables its user to create and transform a graph state, $|G\rangle$ by:

1. adding or removing a vertex or an edge,
2. executing local complementation on a target subgraph,
3. reproducing Pauli group operators through functions that combine elements of 1. with 2.

Modeller is a sandpit environment in which its user expresses an algorithm directly as $|G\rangle$ and can exploit LC-equivalence to refine the algorithm. State is neither prepared nor tracked through tūQ Modeller although the propagation of state is readily apparent through a graph-state model. The user is free to reason about consecutive reductions of

$QC_{\mathcal{C}}$ without having simultaneously to account for the impact of (quantum) errors; indeed, this makes Modeller suitable for formal proofs of algorithms on $QC_{\mathcal{C}}$. Other MBQC modelling tools such as Graphix and McBeth abstract away from $QC_{\mathcal{C}}$ in the expressing of algorithms, which makes it simple to draft an algorithm but inevitably increases the complexity of engineering required for the optimisation layer of any graph-state compiler.

Modeller is part of the tūQ toolchain in order to minimise the spatial complexity of an algorithm, in this case the number of qubits required to resolve a computation. Referencing Appendix B of this dissertation as well as the Modeller video, this chapter's worked example is a demonstration of drawing then reducing Raußendorf and Briegel's 15-qubits CNOT pattern in Modeller. An iterative workflow between Modeller and tūQ Simulator, the subject of the Chapter 4, is thus implied.

Chapter 4 provides further analysis of graph states, in this case the graph state as syntax for expressing a cluster-state algorithm. As with Chapter 3, the second video accompanying this dissertation is an introduction to tūQ's Simulator mode. In much the same way as Modeller, tūQ Simulator's core purpose is to minimise the number of qubits needed to resource $|G\rangle$. Whereas traditional quantum emulators exist to mimic the workings of a quantum computing facility, Simulator exists to write algorithms and to specify the $QC_{\mathcal{C}}$ required to compute such algorithms.

Simulator originated with another research tool named Etch, which was bespoke to an enquiry into circuit etching as an alternative optimising strategy to the algorithm-specific graph (ASG). The proposition that circuit-etching could be an alternative to ASG turned on the proportion of Pauli qubits to non-Pauli qubits situated in a given $|G\rangle$. Worked example 1 was a demonstration of Etch transpiling 30 IQP circuits to $|G\rangle$ as a test of this proposition. Data from the Etch experiment do not make a case for circuit-etching as a viable alternative to ASG: assuming normal distribution of the experiment's sample data, there is negligible prob-

ability of circuit etching ever realising the desired proportion of Pauli qubits to non-Pauli qubits.

By its functionality, Simulator is a superset of Etch. The user of tūQ Simulator can,

- read circuits encoded in the universal quantum gates set identified in Chapter 2,
- directly encode an algorithm through a tile-based syntax,
- combine in workflow with tūQ Modeller to optimise an algorithm by minimising the distance of $|G\rangle$.

Worked example 2 and the Simulator video are an example of Simulator paired with Modeller in workflow to optimise a graph-state as read from an IQP circuit. The [8,70] lattice as defined by an input IQP circuit was reduced through simple rearrangement and without information loss to a lattice of dimensions [8,53].

Basic manual operations by a user make it clear that tūQ's Simulator and Modeller modes are complementary: Simulator abstracts away a lot of the detail and thereby enables the user to concentrate on the *logic* of an algorithm. At the same time, Modeller can work as a kind of drill-down facility, enabling the user to track how an algorithm resolves, in part or in its entirety. Modeller and Simulator lend finer or coarser abstraction to the same $|G\rangle$.

Chapter 5 is an introduction to functions that further integrate tūQ's Modeller and Simulator modes and facilitate the actual computing of a graph-state algorithm as drafted in Simulator. As such, this chapter is mainly two worked examples of the tūQ menu functions, 'Open Algorithm' and 'Compile'. The third and final video accompanying this dissertation is a demonstration of these same integration functions. The Modeller function, Open Algorithm automates and extends the manual procedure demonstrated in the second worked example of Chapter 4.

Simulator's Compile function renders an algorithm drafted for $|G\rangle$ computable on a QPU configured for OpenQASM. These integrating functions also make clear the delineation of Modeller and Simulator's use cases: the user can test optimisations of an algorithm through Modeller but drafting and compiling an algorithm, at least as far as transpiling it to assembly language, is restricted to Simulator only.

A review of the tūQ toolchain overall, with respect to its status as a compiler follows the worked examples. The minimal set of requirements both to realise a compiler and to evaluate tūQ's legitimacy as such includes,

1. a set of text-based operators and data structures intended for a human to compose instructions to the processor;
2. quality assurance of input formed from 1. including checks of syntax and semantics;
3. machine code output matched to the processor's instructions set;

whereas the tūQ toolchain,

- delivers tile-based operators intended for a human to compose an algorithm,
- works at a lower level of operations with fewer and more-limited operators than would necessitate the lexing and parsing functions of a classical compiler,
- transpiles an algorithm to OpenQASM 3.0, showcasing the possibility of output from a graph-state algorithm.

Moreover, linking to a Pauli tracker library as mooted in Chapter 2 would ensure a computation through the toolchain is deterministic and would be meaningful at some future date when an assembly language that encompasses graph-state operators becomes available.

6.2 Future research

Research into cluster-state computing as at December 2024, is hindered by an absence of for-purpose hardware, assembly language and optimised algorithms; and comparatively few formalised higher-level languages with which to draft a graph-state algorithm. Version 0.1 of the t$\bar{u}$Q toolchain is a first step to addressing many of these deficiencies but there are other, concise projects that could progress the toolchain further towards a complete compiler.

There is little immediate reward to drafting a cluster-state assembly language given the absence of a QPU engineered to process cluster-state computations. Neither however is there any penalty to expanding or further abstracting OpenQASM syntax to admit cluster-state computations. This proposition is a variation on the 'two lexicons, one grammar' discussion covered in Chapter 2. The two OpenQASM lexicons would need to be orthogonal to avoid side-effects in processor calls and the processor would have to be engineered to ignore unrecognised instructions rather than percolating compile-time errors. Ensuring this latter point is made all the more difficult by the reluctance of industry-leaders like IBM or Google to release data on how their processors work. In the same vein as arguments made in this dissertation for the significance of t$\bar{u}$Q, an assembly language aligned with the cluster-state model may be all the proof-of-concept needed for investment in a compatible QPU.

Arguably, the real prize for advancing cluster-state computation would be optimisation strategies that unambiguously mark cluster-state computing as more efficient in its use of physical qubits than the circuit. This will always be a hard case to make given how an algorithm processed through $QC_{\mathcal{C}}$ will in most, if not all, cases require many more qubits than processing it through a circuit. Here is a moment to take pause. If an algorithm processed through $QC_{\mathcal{C}}$ can be optimised to require on average fewer shots than its circuit equivalent, is it resource inefficient? Alternatively, as a variation on the circuit-etching exper-

iment of Chapter 4, perhaps an optimisation does lay in a procedure to prepare-and-recycle qubits. It may be that none of these enquiries bears fruit but as at December 2024, research into optimising cluster-state computation was under-represented and such questions, unanswered.

Finally, an argument can be made for a front-end programming language, at a higher level of abstraction than the cluster-state. Pioneering work by Selinger and Valiron (2009) would suggest a fit between the lambda calculus and graph-state algorithms. The constants measurement, `meas`, and unitary operations, U, if restricted to the universal quantum gates set, could make the lambda calculus inclusive of cluster-state computation with each measurement of cluster state, $|\phi\rangle$ an exercise in LU-equivalence, $U|\phi\rangle$.

Appendix A

Installing tūQ

A.1 Setting up

Installing and building `tūQ` requires the following environment,

- a Linux machine; `tūQ` is not configured to run on either Windows or Mac,
- Git, version 2.35+,
- CMake, version 3.28+.

The user must install nlohmann::json (header library) where tūQ/src/layout can read it before compiling.

A.2 Installing

To obtain source code for `tūQ`, the user should run

- git clone https://github.com/QSI-BAQS/tuQ.

then run the Cmake file

- CLI

The user should run cmake processes through any of bash, ksh or zsh shells.

The method, toQASM of class SimulatorView requires some customising.

- SimulatorView::toQASM will output the OpenQASM script to a .txt file and at a minimum, the user should set a path to the required output destination as (QFile) variable, filePOST in format '/path/to/file';
- SimulatorView::toQASM default is *not* to write to an API although the necessary code to do so is part of the method but commented out. The user should specify the API URL as (std::string) variable, reqStrQASM. Note, the POST syntax of this method is necessarily generic and thus simple: it omits such precautions as error checking or managing bad links; furthermore, it does not cater for an API that requires JSON format.

Appendix B

Using tūQ

B.1 tūQ Modeller

The five-vertices graph of Figure B.1, below is the baseline for demonstrations of `tūQ` Modeller functions to follow. Keystrokes activate most

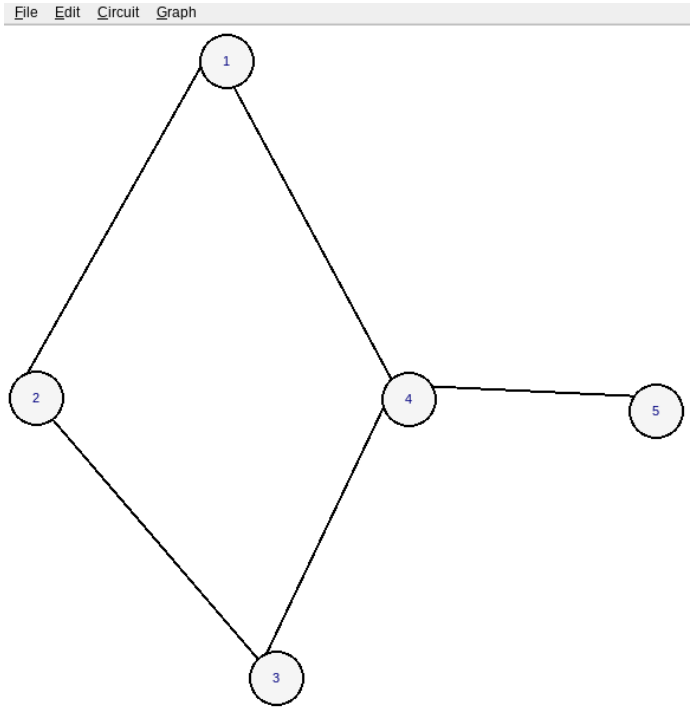


Figure B.1: A five-vertices graph, G.

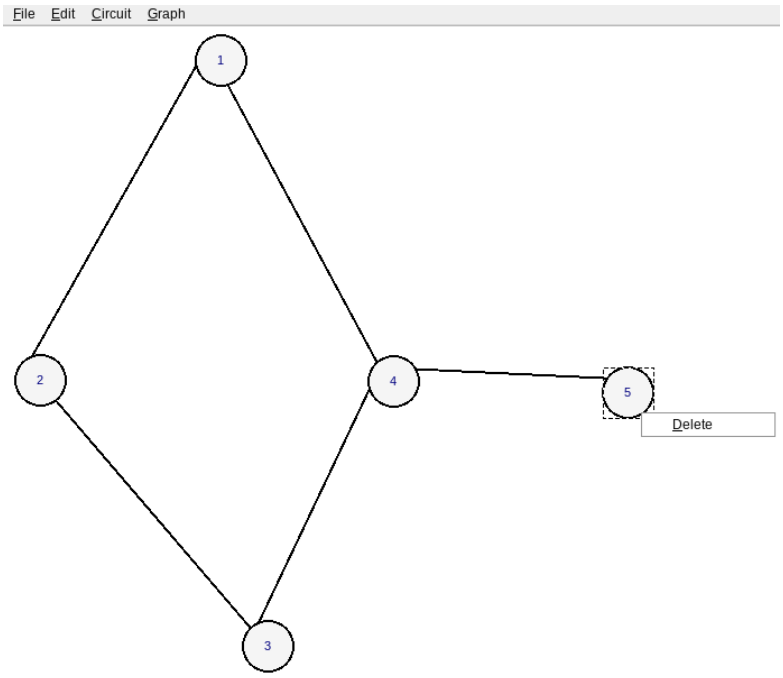


Figure B.2: Deleting a vertex of G.

tūQ Modeller functions while features of a graph created by the user are reorganised either through further functions or by drag-and-drop with a mouse. Note, Modeller can create a lattice of dimensions no greater than $[121, 121]$.

Functions to add or remove a vertex or an edge are the backbone to constructing and editing of $|G\rangle$. tūQ Modeller functions to add, move or change a graph are as follows:

1. Add/remove vertex: the user can,
    - **add** a vertex by punching the 'V' key then, left-clicking the mouse at the desired position on the canvas. At any point, the user may click-and-drag a vertex instance to change its position on the canvas.
    - **remove** a vertex by right-clicking the target vertex then, selecting option 'Delete' (or, keystroke 'D'), as per Figure B.2.

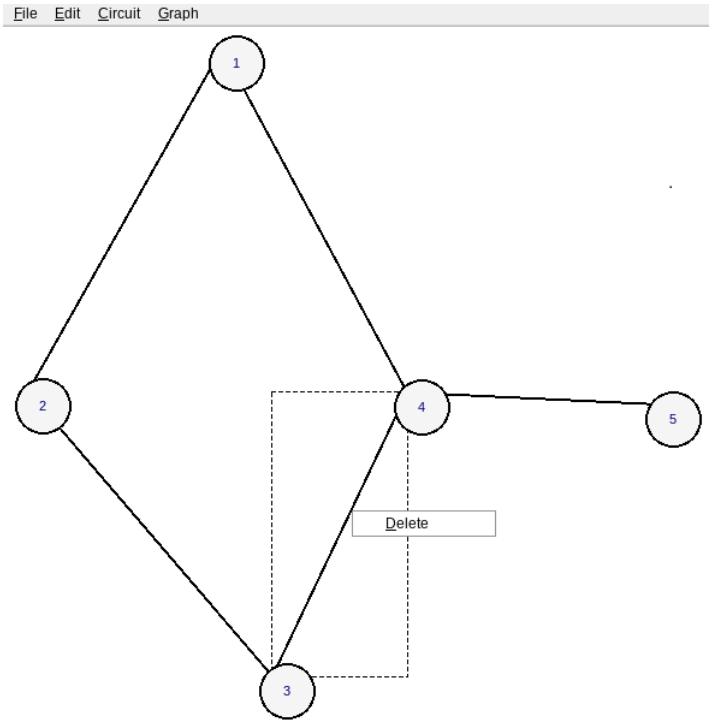


Figure B.3: Deleting an edge of G.

2. Add/remove edge: an edge, ab, signifies entanglement between vertex a and vertex b. The user can,

   - **add** an edge by punching the 'E' key then, left-clicking first on (nominated) vertex a then on vertex b.
   - **remove** an edge by right-clicking on the edge itself then, selecting 'Delete' (or, keystroke 'D'), as per Figure B.3.

   As noted in Section 3.1.1, above, both a 'loop' - an edge that connects a vertex to itself - and multiple edges between two vertices are prohibited as options in `tūQ`.

3. Local complementation. To perform local complementation as a standalone function, punch the 'O' key then, left-click on (nominated) vertex a. Figure B.4 is a demonstration of local complementation applied to vertex 4 as it appears in the graph of Figure B.1. Strictly speaking, this function should be reserved for testing

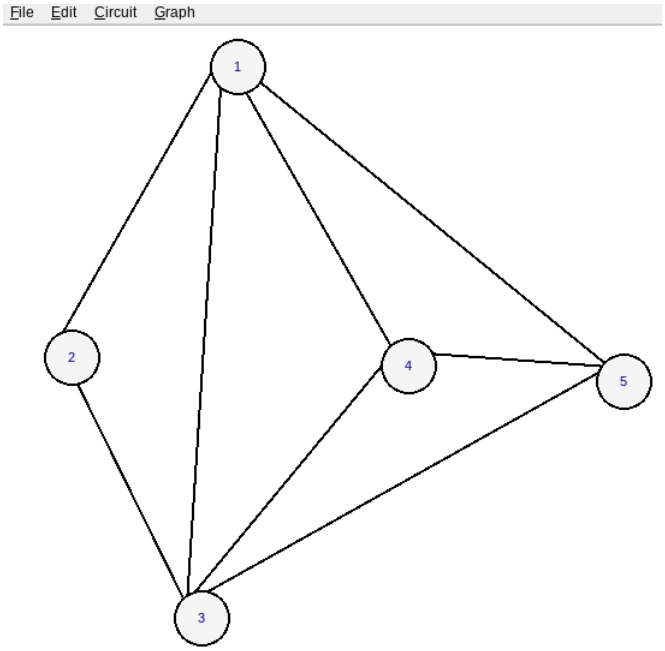


Figure B.4: Local complementation at vertex 4 of G.

purposes only; if $|G\rangle$ is to be preserved as a stabiliser state then, all transformation should occur using only the operators X, Y or Z outlined below.

The Pauli measurements (henceforth, LPMs) in bases σ_z (Z), σ_y (Y) and σ_x (X) are operations for transforming $|G\rangle$. After Hein *et al.* (2006), each of Z, Y and X respectively transforms a graph thus:

4. operator Z: to apply this operation, punch the 'Z' key then, left-click on (nominated) vertex a to remove it and each of its edges from G. Figure B.5 is a demonstration of this LPM applied to vertex 3 as it appears in the graph of Figure B.1.

5. operator Y: to apply this operation, punch the 'Y' key then, left-click on (nominated) vertex a to execute local complementation on it; a Z-operation on vertex a then completes the Y-operation. Figure B.6 is a demonstration of this LPM applied to vertex 3 as it appears in the graph of Figure B.1.

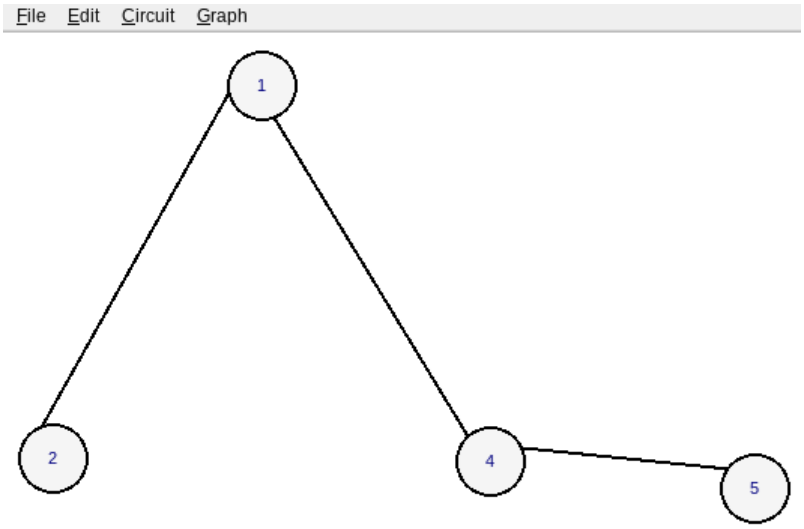


Figure B.5: Z-operation applied to vertex 3 of G.

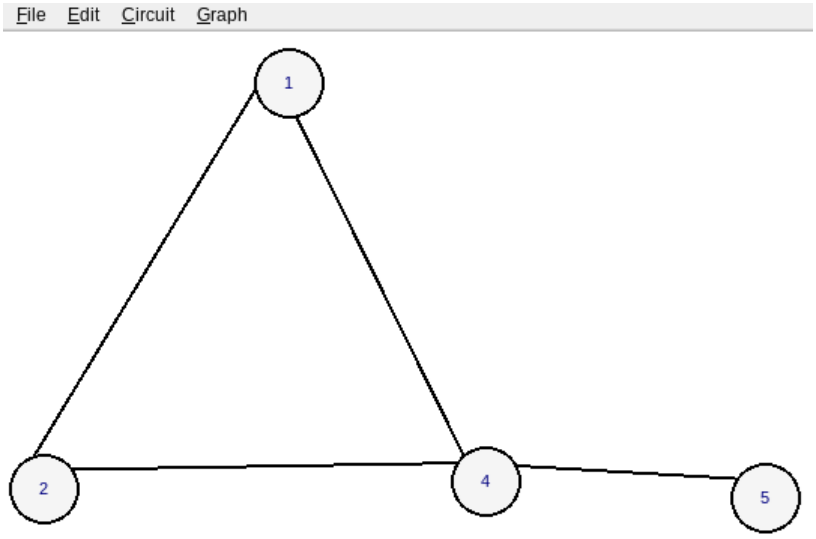
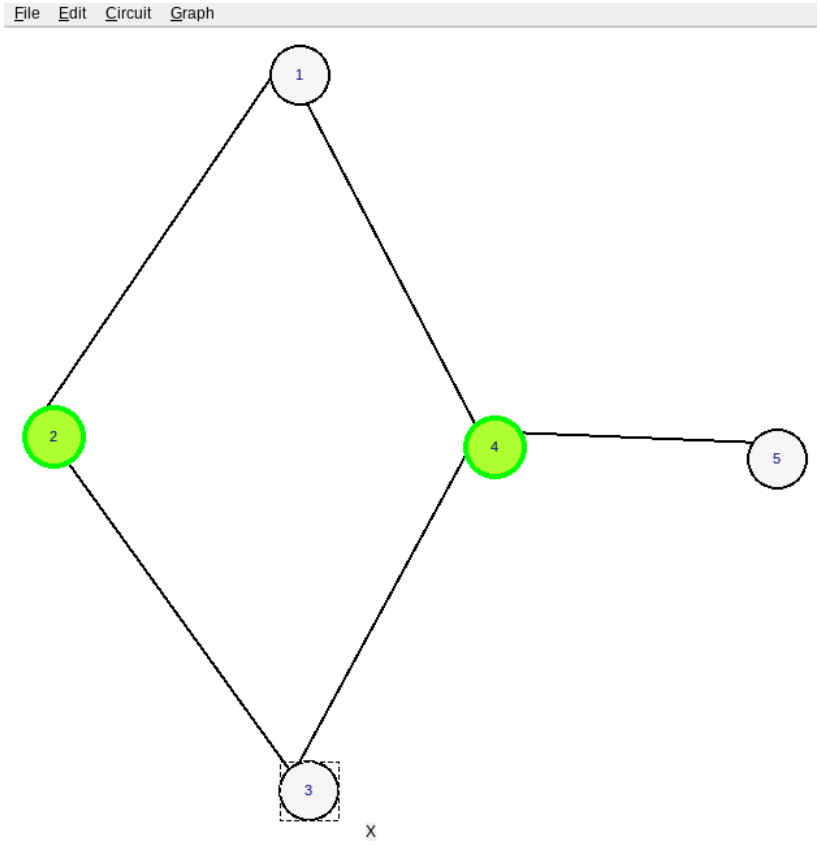
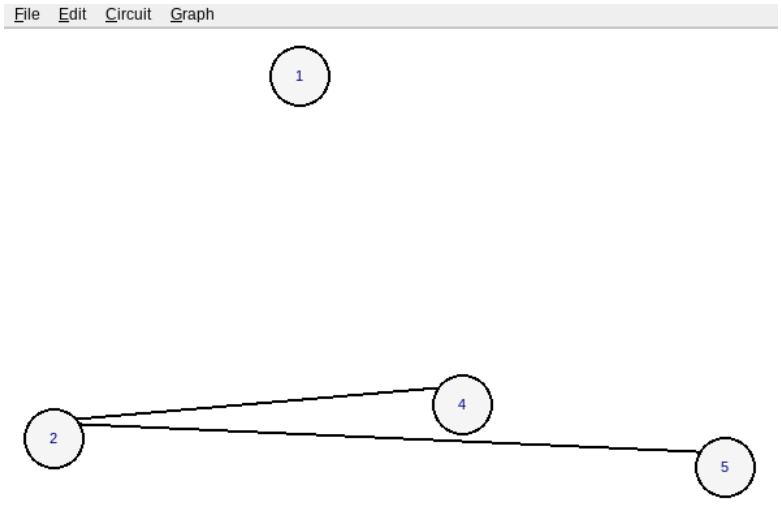


Figure B.6: Y-operation applied to vertex 3 of G.

6. operator X: is an operation requiring two consecutive left-clicks. To effect the X-operation, first punch the 'X' key then, left-click on (nominated) vertex a to execute local complementation on it. Figure B.7 is a demonstration of the local complementation of vertex a, in this case vertex 3 of the graph in Figure B.1. Note how the neighbourhood of vertex 3 is highlighted as a prompt to the user to select the 'special neighbour' vertex (Hein, Eisert, and Briegel



(a) X-operation, the neighbourhood of vertex 3 of G will undergo local complementation operations.



(b) Completed X-operation with vertex 4 selected as the special neighbour of the graph of Figure B.7a.

Figure B.7: X-operation applied to G.

2004; Hein et al. 2006). To conclude the operation, select and left-click on the special neighbour vertex, *b* to execute local complementation on it; σ_z transformation of vertex *a* then completes the X measurement. Figure B.7b is a demonstration of these concluding operations, in this instance vertex 4 as it appears in Figure B.7a is the nominated vertex *b*.

B.2 tūQ Simulator

In standard use,

- the 'Esc' key will exit the operator palette;
- button 'CNOT t ↑' does not insert a target row: the user must click the palette button 'CNOT t ↑' while positioned in the (existing) target row;
- it is not a requirement to close an algorithm row with a readout tile but it is good practice: tūQ Modeller *will not backfill a readout column* that is absent from the source algorithm.

Bibliography

Aaronson, Scott. *Quantum Computing Since Democritus*. Cambridge: Cambridge University Press, 2013. isbn: 1-107-23299-6.

Adcock, Jeremy C., Sam Morley-Short, Axel Dahlberg, and Joshua W. Silverstone. 'Mapping graph state orbits under local complementation'. In: *Quantum* 4 (2020), p. 305. doi: 10.22331/q-2020-08-07-305.

Amaro, David, Carlo Modica, Matthias Rosenkranz, Mattia Fiorentini, Marcello Benedetti, and Michael Lubasch. 'Filtering variational quantum algorithms for combinatorial optimization'. In: *Quantum Science and Technology* 7.1 (2022), p. 015021. doi: 10.1088/2058-9565/ac3e54.

Anders, Simon and Hans J. Briegel. 'Fast simulation of stabilizer circuits using a graph-state representation'. In: *Physical Review A* 73.2 (2006). doi: 10.1103/PhysRevA.73.022334.

Arute, Frank et al. 'Quantum supremacy using a programmable superconducting processor'. In: *Nature* 574.7779 (2019), pp. 505–510. doi: 10.1038/s41586-019-1666-5.

Barenco, Adriano et al. 'Elementary gates for quantum computation'. In: *Physical Review A* 52.5 (1995), pp. 3457–3467. doi: 10.1103/PhysRevA.52.3457.

Bartolucci, Sara et al. 'Fusion-based quantum computation'. In: *Nature Communications* 14.1 (2023), p. 912. doi: 10.1038/s41467-023-36493-1.

Bastian, Mathieu, Sebastien Heymann, and Mathieu Jacomy. 'Gephi: an open source software for exploring and manipulating networks'. In:

2009. url: http://www.aaai.org/ocs/index.php/ICWSM/09/paper/view/154.

Beverland, Michael E. et al. 'Assessing requirements to scale to practical quantum advantage'. In: *arXiv* (2022). doi: 10.48550/arXiv.2211.07629. url: https://arxiv.org/abs/2211.07629.

Bolt, Andrew, Guillaume Duclos-Cianci, David Poulin, and Tom M. Stace. 'Foliated Quantum Error-Correcting Codes'. In: *Physical Review Letters* 117.7 (2016), pp. 070501–070501. doi: 10.1103/PhysRevLett.117.070501.

Bowen, Greg, Athena Caesura, Simon J. Devitt, and Madhav Krishnan Vijayan. 'Design and efficiency in graph-state computation'. In: *arXiv* (2025). doi: 10.48550/arXiv.2502.18985. url: https://arxiv.org/abs/2502.18985.

Bowen, Greg and Simon J. Devitt. 'Q2Graph: a modelling tool for measurement-based quantum computing'. In: *arXiv* (2022). doi: 10.48550/arXiv.2210.00657. url: https://arxiv.org/abs/2210.00657.

— 'tūQ: a design and modelling tool for cluster-state algorithms'. In: *arXiv* (2025). doi: 10.48550/arXiv.2502.18991. url: https://arxiv.org/abs/2502.18991.

Boykin, P. Oscar, Tal Mor, Matthew Pulver, Vwani Roychowdhury, and Farrokh Vatan. 'On universal and fault-tolerant quantum computing'. In: *arXiv* (1999). doi: 10.48550/arxiv.quant-ph/9906054. url: https://arxiv.org/abs/quant-ph/9906054.

Brachman, Ronald, Hector Levesque, and Maurice Pagnucco. *Knowledge Representation and Reasoning*. San Francisco: Elsevier Science & Technology, 2004. isbn: 978-0-08-048932-2.

Bravyi, Sergey and Alexei Kitaev. 'Universal quantum computation with ideal Clifford gates and noisy ancillas'. In: *Physical Review A* 71.2 (2005). doi: 10.1103/PhysRevA.71.022316.

Bravyi, Sergey and Robert Raußendorf. 'Measurement-based quantum computation with the toric code states'. In: *Physical Review A* 76.2 (2007), p. 022304. doi: 10.1103/PhysRevA.76.022304.

Briegel, Hans J., Daniel E. Browne, Wolfgang Dür, Robert Raußendorf, and Maarten Van den Nest. 'Measurement-based quantum computation'. In: *Nature Physics* 5.1 (2009), pp. 19–26. doi: 10.1038/nphys1157.

Briegel, Hans J. and Robert Raußendorf. 'Persistent entanglement in arrays of interacting particles'. In: *Physical Review Letters* 86.5 (2001), pp. 910–913. doi: 10.1103/PhysRevLett.86.910.

Browne, Daniel E. and Hans J. Briegel. 'One-way quantum computation - a tutorial introduction'. In: *arXiv* (2006). doi: 10.48550/arxiv.quant-ph/0603226. url: https://arxiv.org/abs/quant-ph/0603226.

Browne, Daniel E., Elham Kashefi, Mehdi Mhalla, and Simon Perdrix. 'Generalized flow and determinism in measurement-based quantum computation'. In: *New Journal of Physics* 9.8 (2007), p. 250. doi: 10.1088/1367-2630/9/8/250.

Browne, Daniel E. and Terry Rudolph. 'Resource-efficient linear optical quantum computation'. In: *Physical Review Letters* 95.1 (2005), pp. 010501.1–010501.4. doi: 10.1103/PhysRevLett.95.010501.

Cabello, Adán, Lars Eirik Danielsen, Antonio J. López-Tarrida, and José R. Portillo. 'Optimal preparation of graph states'. In: *Physical Review A* 83.4 (2011), p. 042314. doi: 10.1103/PhysRevA.83.042314.

Chen, Chi-Fang Anthony, Hsin-Yuan Huang, John Preskill, and Leo Zhou. 'Local minima in quantum systems'. In: *arXiv* (2023). doi: 10.48550/arXiv.2309.16596. url: https://arxiv.org/abs/2309.16596.

Chen, Jwo-Sy et al. 'Benchmarking a trapped-ion quantum computer with 30 qubits'. In: *Quantum* 8 (2024), p. 1516. doi: 10.22331/q-2024-11-07-1516.

Childs, Andrew M., Debbie W. Leung, and Michael A. Nielsen. 'Unified derivations of measurement-based schemes for quantum computation'. In: *Physical Review A* 71.3 (2005), p. 032318. doi: 10.1103/PhysRevA.71.032318.

Chong, Frederic T., Diana Franklin, and Margaret Martonosi. 'Programming languages and compiler design for realistic quantum hard-

ware'. In: *Nature* 549.7671 (2017), pp. 180–187. doi: 10.1038/nature23459.

Claudet, Nathan and Simon Perdrix. 'Local equivalence of stabilizer states: a graphical characterisation'. In: *arXiv* (2024). doi: 10.48550/arxiv.2409.20183. url: https://arxiv.org/abs/2409.20183.

Coene, Jean-Philippe. 'sigmajs: An R htmlwidget interface to the sigma.js visualization library'. In: *Journal of Open Source Software* 3.28 (2018), p. 814. doi: 10.21105/joss.00814.

Cooper, Keith D. and Linda Torczon. *Engineering a Compiler*. 2nd ed. San Francisco: Morgan Kaufmann, 2012. isbn: 978-0-12-088478-0.

Crippa, Arianna, Karl Jansen, and Enrico Rinaldi. 'Analysis of the confinement string in (2 + 1)-dimensional quantum electrodynamics with a trapped-ion quantum computer'. In: *arXiv* (2024). doi: 10.48550/arXiv.2411.05628. url: https://arxiv.org/abs/2411.05628.

Cross, Andrew W., Lev S. Bishop, John A. Smolin, and Jay M. Gambetta. 'Open quantum assembly language'. In: *arXiv* (2017). doi: 10.48550/ARXIV.1707.03429. url: https://arxiv.org/abs/1707.03429.

Cross, Andrew W. et al. 'OpenQASM 3: a broader and deeper quantum assembly language'. In: *ACM Transactions on Quantum Computing* 3.3 (2022), pp. 1–50. doi: 10.1145/3505636.

Dahlberg, Axel, Jonas Helsen, and Stephanie Wehner. 'How to transform graph states using single-qubit operations: computational complexity and algorithms'. In: *Quantum Science and Technology* 5.4 (2020), p. 45016. doi: 10.1088/2058-9565/aba763.

Danos, Vincent, Elham Kashefi, Prakash Panangaden, and Simon Perdrix. 'Extended measurement calculus'. In: *Semantic Techniques in Quantum Computation*. Ed. by Ian Mackie and Simon Gay. Cambridge: Cambridge University Press, 2009, pp. 235–310. isbn: 978-0-52-151374-6.

Deutsch, David, Adriano Barenco, and Artur Ekert. 'Universality in quantum computation'. In: *Proceedings of the Royal Society A* 449.1937 (1995), pp. 669–677. doi: 10.1098/rspa.1995.0065.

Devitt, Simon J., Kae Nemoto, and William J. Munro. 'Quantum error correction for beginners'. In: *Reports on Progress in Physics* 76.076001 (2009). doi: 10.1088/0034-4885/76/7/076001.

Divincenzo, D. P. 'Topics in quantum computers'. In: *Mesoscopic Electron Transport*. Ed. by Lydia L. Sohn, Leo P. Kouwenhoven, and Gerd Schön. Dordrecht: Springer Netherlands, 1997, pp. 657–677. isbn: 978-9-40-158839-3.

Dupont, Maxime et al. 'Benchmarking quantum optimization for the maximum-cut problem on a superconducting quantum computer'. In: *Physical Review Applied* 23.1 (2025), p. 014045. doi: 10.1103/PhysRevApplied.23.014045.

Elman, Samuel J., Jason Gavriel, and Ryan L. Mann. 'Optimal scheduling of graph states via path decompositions'. In: *arXiv* (2024). doi: 10.48550/arXiv.2403.04126. url: https://arxiv.org/abs/2403.04126.

Evans, Aidan, Seun Omonije, Robert Soule, and Robert Rand. 'MCBeth: a measurement-based quantum programming language'. In: *IEEE/ACM 4th International Workshop on Quantum Software Engineering (Q-SE)*. IEEE, 2023, pp. 1–8. isbn: 979-8-35-030180-9. doi: 10.1109/Q-SE59154.2023.00007.

Feller, William S. *An Introduction to Probability Theory and its Applications*. 3rd ed. New York: Wiley, 1991. isbn: 978-0-47-125708-0.

Feynman, Richard P. 'Simulating physics with computers'. In: *International Journal of Theoretical Physics* 21.6 (1982), pp. 467–488. doi: 10.1007/BF02650179.

— 'Quantum mechanical computers'. In: *Foundations of Physics* 16.6 (1986), pp. 507–531. doi: 10.1007/BF01886518.

Fischer, Charles N., Ron K. Cytron, and Richard J. LeBlanc. *Crafting a Compiler*. Boston: Addison-Wesley, 2010. isbn: 978-0-13-606705-4.

Fowler, Austin G., Matteo Mariantoni, John M. Martinis, and Andrew N. Cleland. 'Surface codes: towards practical large-scale quantum computation'. In: *Physical Review A* 86.3 (2012). doi: 10.1103/PhysRevA.86.032324.

Fowler, Austin G., Ashley M. Stephens, and Peter Groszkowski. 'High-threshold universal quantum computation on the surface code'. In: *Physical Review A* 80.5 (2009). doi: 10.1103/PhysRevA.80.052312.

Gansner, Emden R. and Stephen C. North. 'An open graph visualization system and its applications to software engineering'. In: *Software - Practice and Experience* 30.11 (2000), pp. 1203–1233. doi: 10.5555/358668.358697.

Gold, Alysson et al. 'Entanglement across separate silicon dies in a modular superconducting qubit device'. In: *npj Quantum Information* 7.142 (2021). doi: 10.1038/s41534-021-00484-1. url: https://www.nature.com/articles/s41534-021-00484-1.

Google. 'Measurement-induced entanglement and teleportation on a noisy quantum processor'. In: *Nature* 622.7983 (2023a), pp. 481–486. doi: 10.1038/s41586-023-06505-7.

— 'Suppressing quantum errors by scaling a surface code logical qubit'. In: *Nature* 614.7949 (2023b), pp. 676–681. doi: 10.1038/s41586-022-05434-1.

— 'Quantum error correction below the surface code threshold'. In: *Nature* (2024). doi: 10.1038/s41586-024-08449-y.

Gottesman, Daniel. 'Stabilizer codes and quantum error correction'. In: *arXiv* (1997). doi: 10.48550/ARXIV.QUANT-PH/9705052. url: https://arxiv.org/abs/quant-ph/9705052.

— 'The Heisenberg representation of quantum computers'. In: *arXiv* (2008). doi: 10.48550/arXiv.quant-ph/9807006. url: http://arxiv.org/abs/quant-ph/9807006.

— 'An introduction to quantum error correction and fault-tolerant quantum computation'. In: *arXiv* (2009). doi: 10.48550/ARXIV.0904.2557. url: https://arxiv.org/abs/0904.2557.

Gujarati, Damodar. *Essentials of Econometrics, second edition*. Boston: McGraw-Hill, 1999.

Hagberg, Aric A., Daniel A. Schult, and Pieter J. Swart. 'Exploring network structure, dynamics, and function using NetworkX'. In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gael Varo-

quaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA: SciPy Conference, 2008, pp. 11–15. url: http://conference.scipy.org.s3-website-us-east-1.amazonaws.com/proceedings/SciPy2008/paper_2/.

Halpin, Terry and Tony Morgan. *Information Modeling and Relational Databases*. 2nd ed. Boston: Morgan Kaufmann, 2008. isbn: 978-0-12-373568-3.

Häner, Thomas, Damian S. Steiger, Krysta Svore, and Matthias Troyer. 'A software methodology for compiling quantum programs'. In: *Quantum Science and Technology* 3.2 (2018), p. 020501. doi: 10.1088/2058-9565/aaa5cc. url: http://dx.doi.org/10.1088/2058-9565/aaa5cc.

Hein, Marc, Wolfgang Dür, Jens Eisert, Robert Raußendorf, Maarten Van den Nest, and Hans J. Briegel. 'Entanglement in graph states and its applications'. In: *Quantum Computers, Algorithms and Chaos, Proceedings of the International School of Physics "Enrico Fermi"; course 162*. Ed. by G. Casati, D.L. Shepelyansky, P. Zoller, and G. Benenti. Amsterdam: IOS Press, 2006, pp. 115–218. isbn: 978-1-58-603660-7.

Hein, Marc, Jens Eisert, and Hans J. Briegel. 'Multiparty entanglement in graph states'. In: *Physical Review A* 69.6 (2004), pp. 1–62311. doi: 10.1103/PhysRevA.69.062311.

Hietala, Kesha, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 'A verified optimizer for quantum circuits'. In: *ACM Transactions on Programming Languages and Systems* 45.3 (2023), pp. 1–35. doi: 10.1145/3604630.

Horsman, Clare, Austin G. Fowler, Simon J. Devitt, and Rodney Van Meter. 'Surface code quantum computing by lattice surgery'. In: *New Journal of Physics* 14.12 (2012), p. 123011. doi: 10.1088/1367-2630/14/12/123011.

Ittah, David, Thomas Häner, Vadym Kliuchnikov, and Torsten Hoefler. 'QIRO: a static single assignment-based quantum program representation for optimization'. In: *ACM Transactions on Quantum Computing* 3.3 (2022), Article 14. doi: 10.1145/3491247.

Kandala, Abhinav, Kristan Temme, Antonio D. Córcoles, Antonio Mezzacapo, Jerry M. Chow, and Jay M. Gambetta. 'Error mitigation extends the computational reach of a noisy quantum processor'. In: *Nature* 567.7749 (2019), pp. 491–495. doi: 10.1038/s41586-019-1040-7.

Khammassi, Nader, Gian G. Guerreschi, Imran Ashraf, Justin W. Hogaboam, Carmen G. Almudever, and Koen Bertels. 'cQASM v1.0: towards a common quantum assembly language'. In: *arXiv* (2018). doi: 1805.09607. url: https://arxiv.org/abs/1805.09607.

Kieling, Konrad, Terry Rudolph, and Jens Eisert. 'Percolation, renormalization, and quantum computing with nondeterministic gates'. In: *Physical Review Letters* 99.13 (2007), p. 130501. doi: 10.1103/physrevlett.99.130501.

Knill, Emanuel. 'Quantum computing with realistically noisy devices'. In: *Nature* 434.7029 (2005), pp. 39–44. doi: 10.1038/nature03350.

Kraglund Andersen, Christian et al. 'Repeated quantum error detection in a surface code'. In: *Nature Physics* 16.8 (2020), pp. 875–880. doi: 10.1038/s41567-020-0920-y.

Kreppel, Fabian et al. 'Quantum circuit compiler for a shuttling-based trapped-ion quantum computer'. In: *Quantum* 7 (2023), p. 1176. doi: 10.22331/q-2023-11-08-1176.

Krishnan Vijayan, Madhav, Alexandru Paler, Jason Gavriel, Casey R. Myers, Peter P. Rohde, and Simon J. Devitt. 'Compilation of algorithm-specific graph states for quantum circuits'. In: *Quantum Science and Technology* 9.2 (2024), p. 025005. doi: 10.1088/2058-9565/ad1f39.

Litinski, Daniel. 'A game of surface codes: large-scale quantum computing with lattice surgery'. In: *Quantum* 3 (2019a), p. 128. doi: 10.22331/q-2019-03-05-128.

— 'Magic state distillation: not as costly as you think'. In: *Quantum* 3 (Dec. 2019b), p. 205. doi: 10.22331/q-2019-12-02-205. url: https://doi.org/10.22331/q-2019-12-02-205.

Litteken, Andrew, Yung-Ching Fan, Devina Singh, Margaret Martonosi, and Frederic T. Chong. 'An updated LLVM-based quantum research

compiler with further OpenQASM support'. In: *Quantum Science and Technology* 5.3 (2020), p. 34013. doi: 10.1088/2058-9565/ab8c2c.

Liu, Sitong, Naphan Benchasattabuse, Darcy Q. C. Morgan, Michal Hajdušek, Simon J. Devitt, and Rodney Van Meter. 'A substrate scheduler for compiling arbitrary fault-tolerant graph states'. In: *IEEE International Conference on Quantum Computing and Engineering (QCE)*. Vol. I. IEEE, 2023, pp. 870–880. isbn: 979-8-35-034323-6. doi: 10.1109/QCE57702.2023.00101.

Low, Guang Hao and Isaac L. Chuang. 'Optimal hamiltonian simulation by quantum signal processing'. In: *Physical Review Letters* 118.1 (2017). doi: 10.1103/physrevlett.118.010501.

Low, Guang Hao, Theodore J. Yoder, and Isaac L. Chuang. 'Methodology of resonant equiangular composite quantum gates'. In: *Physical Review X* 6.4 (2016). doi: 10.1103/physrevx.6.041067.

Marlow, Simon. *Parallel and Concurrent Programming in Haskell*. New York: O'Reilly, 2013. isbn: 978-1-44-933594-6.

Mermin, N. David. *Quantum Computer Science: An Introduction*. Cambridge: Cambridge University Press, 2007. isbn: 978-0-52-187658-2.

Moses, Steven A. et al. 'A race-track trapped-ion quantum processor'. In: *Physical Review X* 13.4 (2023). doi: 10.1103/physrevx.13.041052.

Murali, Prakash, Dripto M. Debroy, Kenneth R. Brown, and Margaret Martonosi. 'Architecting noisy intermediate-scale trapped ion quantum computers'. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE Press, 2020, pp. 529–542. isbn: 978-1-72-814661-4. doi: 10.1109/ISCA45697.2020.00051.

Nam, Yunseong, Neil J. Ross, Yuan Su, Andrew M. Childs, and Dmitri Maslov. 'Automated optimization of large quantum circuits with continuous parameters'. In: *npj Quantum Information* 4.1 (2018), p. 23. doi: 10.1038/s41534-018-0072-4.

Nguyen, Thien, Anthony Santana, Tyler Kharazi, Daniel Claudino, Hal Finkel, and Alexander McCaskey. 'Extending C++ for heterogeneous quantum-classical computing'. In: *arXiv* (2020). doi: 10.48550/arXiv.2010.03935. url: https://arxiv.org/abs/2010.03935.

Nielsen, Michael A. 'Optical quantum computation using cluster states'. In: *Physical Review Letters* 93.4 (2004), pp. 040503.1–040503.4. doi: 10.1103/PhysRevLett.93.040503.

— 'Cluster-state quantum computation'. In: *Reports on Mathematical Physics* 57.1 (2006), pp. 147–161. doi: 10.1016/S0034-4877(06)80014-5.

Nielsen, Michael A. and Isaac L. Chuang. *Quantum Computation and Quantum Information*. 10th anniversary. Cambridge: Cambridge University Press, 2010. isbn: 978-1-10-700217-3.

Paler, Alexandru, Simon J. Devitt, Kae Nemoto, and Ilia Polian. 'Software-based pauli tracking in fault-tolerant quantum circuits'. In: *Proceedings of the Conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014, pp. 1–4. isbn: 978-3-98-108013-1. doi: 10.7873/DATE.2014.137.

Paler, Alexandru, Ilia Polian, Kae Nemoto, and Simon J. Devitt. 'Fault-tolerant, high-level quantum circuits: form, compilation and description'. In: *Quantum Science and Technology* 2.2 (2017). doi: 10.1088/2058-9565/aa66eb.

Patel, Tirthak, Daniel Silver, and Devesh Tiwari. 'GEYSER: a compilation framework for quantum computing with neutral atoms'. In: *ISCA '22: Proceedings of the 49th Annual International Symposium on Computer Architecture*. Assoc Computing Machinery, 2022, pp. 383–395. doi: 10.1145/3470496.3527428.

Preskill, John. 'Reliable quantum computers'. In: *Proceedings of the Royal Society A* 454 (1998), pp. 385–410. doi: 10.1098/rspa.1998.0167.

— 'Quantum computing in the NISQ era and beyond'. In: *arXiv* (2018). doi: 10.22331/q-2018-08-06-79. url: https://arxiv.org/abs/1801.00862.

Rabhi, Fethi and Guy Lapalme. *Algorithms: a Functional Programming Approach*. 2nd ed. International computer science series. Harlow: Addison-Wesley, 1999. isbn: 978-0-20-159604-5.

Raußendorf, Robert and Hans J. Briegel. 'A one-way quantum computer'. In: *Physical Review Letters* 86.22 (2001), pp. 5188–5191. doi: 10.1103/PhysRevLett.86.5188.

— 'Computational model underlying the one-way quantum computer'. In: *Quantum Information & Computation* 2.6 (2002), pp. 443–86. doi: 10.26421/QIC2.6-3.

Raußendorf, Robert, Daniel E. Browne, and Hans J. Briegel. 'Measurement-based quantum computation on cluster states'. In: *Physical Review A* 68.2 (2003), pp. 223121–2231232. doi: 10.1103/PhysRevA.68.022312.

Raußendorf, Robert, Jim Harrington, and Kovid Goyal. 'A fault-tolerant one-way quantum computer'. In: *Annals of Physics* 321.9 (2006), pp. 2242–2270. doi: 10.1016/j.aop.2006.01.012.

— 'Topological fault-tolerance in cluster state quantum computation'. In: *New Journal of Physics* 9.6 (2007), pp. 199–199. doi: 10.1088/1367-2630/9/6/199.

Riesebos, L., X. Fu, S. Varsamopoulos, C. G. Almudever, and K. Bertels. 'Pauli frames for quantum computer architectures'. In: *DAC '17: Proceedings of the 54th Annual Design Automation Conference 2017*. Association for Computing Machinery, 2017, p. 6. isbn: 978-1-45-034927-7. doi: 10.1145/3061639.3062300.

Rubin, Nicholas C. et al. 'Quantum computation of stopping power for inertial fusion target design'. In: *arXiv* (2023). doi: 10.48550/arXiv.2308.12352. url: https://arxiv.org/abs/2308.12352v1.

Ruh, Jannis and Simon J. Devitt. 'Quantum circuit optimisation and MBQC scheduling with a pauli tracking library'. In: *arXiv* (2024). doi: 10.48550/arXiv.2405.03970. url: https://arxiv.org/abs/2405.03970.

Schlingemann, Dirk. 'Cluster states, algorithms and graphs'. In: *Quantum Information & Computation* 4.4 (2004), pp. 287–324. doi: 10.26421/QIC4.4-4.

Selinger, Peter and Benoît Valiron. 'Quantum lambda calculus'. In: *Semantic Techniques in Quantum Computation*. Ed. by Ian Mackie and

Simon Gay. Cambridge: Cambridge University Press, 2009, pp. 135–172. isbn: 978-0-52-151374-6.

Shepherd, Dan and Michael J. Bremner. 'Temporally unstructured quantum computation'. In: *Proceedings of the Royal Society A* 465.2105 (2009), pp. 1413–1439. doi: 10.1098/rspa.2008.0443.

Sivarajah, Seyon, Silas Dilkes, Alexander Cowtan, Will Simmons, Alec Edgington, and Ross Duncan. 't|ket⟩ a retargetable compiler for NISQ devices'. In: *Quantum Science and Technology* 6.1 (2021), p. 14003. doi: 10.1088/2058-9565/ab8e92.

Sunami, Shinichi and Masato Fukushima. 'Graphix: optimizing and simulating measurement-based quantum computation on local-Clifford decorated graph'. In: *arXiv* (2022). doi: 10.48550/arXiv.2212.11975. url: https://arxiv.org/abs/2212.11975.

Sung, Kevin J. et al. 'Using models to improve optimizers for variational quantum algorithms'. In: *Quantum Science and Technology* 5.4 (2020), p. 044008. doi: 10.1088/2058-9565/abb6d9.

Svore, Krysta M., Alfred V. Aho, Andrew W. Cross, Isaac L. Chuang, and Igor L. Markov. 'A layered software architecture for quantum computing design tools'. In: *Computer* 39.1 (2006), pp. 74–83. doi: 10.1109/MC.2006.4.

Tanenbaum, Andrew S. and Herbert Bos. *Modern Operating Systems*. 4th ed. New Jersey: Pearson, 2015. isbn: 978-0-13-359162-0.

Tantau, Till. *TikS & PGF (Version 3.1.10)*. Reference Manual. 2024. url: https://github.com/pgf-tikz/pgf.

Terhal, Barbara M. 'Quantum error correction for quantum memories'. In: *Reviews of Modern Physics* 87.2 (2015), pp. 307–346. doi: 10.1103/RevModPhys.87.307.

The Sage Developers. *SageMath, the Sage mathematics software system (version 10.3)*. Reference Manual. 2024. url: https://doc.sagemath.org/pdf/en/reference/reference.pdf.

Van den Nest, Maarten, Jeroen Dehaene, and Bart De Moor. 'Graphical description of the action of local Clifford transformations on graph

states'. In: *Physical Review A* 69.2 (2004a), pp. 223161–223167. doi: 10.1103/PhysRevA.69.022316.

Wack, Andrew et al. 'Quality, speed, and scale: three key attributes to measure the performance of near-term quantum computers'. In: *arXiv* (2021). doi: 10.48550/ARXIV.2110.14108. url: https://arxiv.org/abs/2110.14108.

Zhang, Hezi et al. 'OneQ: a compilation framework for photonic one-way quantum computation'. In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. Association for Computing Machinery, 2023, Article 12. isbn: 979-8-40-070095-8/23/06. doi: 10.1145/3579371.3589047.