Quantum Cost Models for Cryptanalysis of Isogenies

by

Samuel Jaques

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Combinatorics and Optimization

Waterloo, Ontario, Canada, 2019

Statement of Contributions

I am the sole author of Chapters 1 and 5. Chapter 2 is partly based on "Surface codes: Towards practical large-scale quantum computation" by A.G. Fowler, M. Mariantoni, J.M. Martinis, and A.N. Clelaland. Chapters 2, 6, and 7 contain material from "Quantum Cryptanalysis in the RAM model: Claw finding attacks on SIKE", co-authored by myself and J. Schanck. Chapter 3 is based on "Search via quantum walk" by F. Magniez, A. Nayak, J. Roland, and M. Santha. Chapters 4 and 6 use methods from "Efficient distributed quantum computing" by R. Beals, S. Brierley, O. Gray, A.W. Harrow, S. Kutin, N. Linden, D. Shepherd, and M. Stather.

Abstract

Isogeny-based cryptography uses keys large enough to resist a far-future attack from Tani's algorithm, a quantum random walk on Johnson graphs. The key size is based on an analysis in the query model. Queries do not reflect the full cost of an algorithm, and this thesis considers other cost models. These models fit in a *memory peripheral* framework, which focuses on the classical control costs of a quantum computer. Rather than queries, we use the costs of individual gates, error correction, and latency. Primarily, these costs make quantum memory access expensive and thus Tani's memory-intensive algorithm is no longer the best attack against isogeny-based cryptography. A classical algorithm due to van Oorschot and Wiener can be faster and cheaper, depending on the model used and the availability of time and hardware. This means that isogeny-based cryptography is more secure than previously thought.

Acknowledgements

I would like to thank my supervisors, Alfred Menezes and Michele Mosca, for their advice and help throughout this project, my master's program, and beyond.

I give particular thanks to John Schanck, for many discussions and explanations and for a lot of the philosophy behind this thesis.

For helping me understand random walks, I thank Aswhin Nayak. Thank you to all of the other students who made my time more fun, interesting, and educational, and thank you to all of the staff in the Combinatorics and Optimization department and the Institute for Quantum Computing.

Special thanks to my partner Tiffany Cassidy for supporting me during my education.

Table of Contents

List of Figures

List of Tables

List of Costs

Introduction

Asymmetric cryptography was invented in the 1970s, and quantum computing was envisioned in the 1980s. In the 1990s, these two fields collided with Shor's algorithm — a fast quantum algorithm to factor integers. This work put an expiry date on the current methods of asymmetric cryptography. At some point we will need to switch to new methods, though we still don't know *when*. Anticipating a long transition, the United States' National Institute of Standards and Technology (NIST) began a process of standardizing alternative, "post-quantum" approaches to asymmetric cryptography that will withstand attacks from quantum computers. In the first round of submissions, they received 82 proposals. Each proposal was either much slower than previous asymmetric cryptography, or used much larger keys, or both. Everyone who needs secure communication now faces a choice: keep the old methods and risk a quantum attack (potentially decades in the future), or switch to the new methods and suffer substantial performance loss.

Part of the reason for the loss in performance is that the new methods must be secure against both classical *and* quantum attacks. For almost every submission to NIST, the "best" attack is quantum. Hence, each system needs to use keys large enough that even a far future quantum computer will be unable to break it. This raises the bar for cryptanalysis: Cryptographers need to analyze the difficulty of breaking a system with a type of computer *that does not exist yet*.

Luckily for cryptographers, quantum algorithm analysis is a thriving research area. We have well-studied models of the capabilities of quantum computers, several families of quantum algorithms, and many results in complexity theory relating classical algorithm complexity to quantum algorithm complexity. A basic result in this field is that quantum computers are "more powerful" than classical computers. This means that any classical algorithm can be converted to a quantum algorithm with only a constant overhead, but many experts believe there are quantum algorithms that a classical computer cannot efficiently simulate.

Many quantum algorithm analysts seem to take this result as *carte blanche* to use any classical algorithm they want, at any point they want, in any quantum circuit they want. This will not be possible. First, the conversion from a classical algorithm to a quantum algorithm often produces many so-called "ancilla" qubits. These are extra pieces of data that are essentially "garbage" but which cannot be removed without ruining the computation. Many quantum algorithms rely on different computations interfering with each other, but ancilla qubits can prevent this interference.

Second, interference in a quantum algorithm is a very physical phenomenon. Any data must have a unique representation for interference to occur reliably. For example, there will be interference between two linked lists containing the same data but with elements at different memory addresses. Classical computers do not often worry about this issue, and hence not every classical algorithm will work as part of a quantum algorithm.

These issues will come up again and again in this thesis, and show that many quantum algorithms do not work as advertised. Further, all of the analyses require a theoretical model of a quantum computer, and the model itself may be insufficient.

First, the most common model of quantum costs is the *query* model. Here, we have some specific circuit called an *oracle*, treated as a "black box", which performs some quantum circuit. We build an algorithm that uses this oracle and we count how many times we needed to perform the oracle circuit. We must interpret query complexity with care, as it excludes costs that may be critical in certain applications. For some algorithms, the query complexity is almost exactly the same as any other cost. For other algorithms, the difference is enormous.

Even slightly more realistic models seem to just be based on a classical computer that has extra quantum capabilities. Classical computers have a very particular architecture, with a CPU performing sequential instructions and with access to a large array of random access memory. With today's technology there is no reason to expect quantum computers to follow the same path, and in fact the quantum technology we *do* have suggests that they will behave very differently. In particular, a major difference is that quantum computers might not have physical "gates" printed onto chips like classical computers do, but that gates (`AND`, `OR`, etc.) will remain *processes* that are applied to quantum data. This is a subtle distinction but would have drastic implications for quantum random access memory.

Finally, quantum computers are noisy. This is why we do not have quantum computers today. Early classical computers were noisy too, but the errors in a quantum computer are fundamentally different and fundamentally more complicated. To protect against noise, either an external process must intervene and repeatedly correct errors, or the system must be cold enough that the noise is small enough not to cause problems. A classical hard drive can use a two-dimensional magnetic disk to store memory and it will retain that data for a long time without external intervention, even at room temperature. For quantum systems, all known methods to encode data in an analogous way in two dimensions require impossibly low temperatures, and many possible approaches are proven to have the same issue [17, §4]. Thus, quantum data storage will likely need a continuous, costly error correction process.

In short, post-quantum cryptography has created a clash of realism. Cryptographers are comparing schemes on the scale of microseconds, where the fastest algorithm can depend on just a few bits of keys. On the other hand, they are choosing keys based on quantum algorithm complexity, a field based on more abstract and long-term thinking.

Considering how much classical computing technology changed since its creation, quantum algorithm analysts have good reason to postpone analyses of specific architectures. Especially in the realm of decades-long cryptography, we don't want to be surprised by a radical breakthrough in 30 years. However, we know enough now about quantum computers that we can start to narrow down the possibilities, and draw reasonable – though not iron-clad – conclusions about the costs of quantum algorithms.

This thesis will show what such an analysis might look like. Adding only a few extra assumptions about quantum computing ends up radically changing the costs of several algorithms. For SIKE, a particular post-quantum asymmetric protocol, we show that we can retain the same security but reduce key sizes by 42%, just by taking a closer look at the quantum attacks used to estimate security. Figure 1 shows the conclusions for SIKE, showing that under many assumptions of constraints and architecture, classical algorithms are the best option. There are almost certainly other post-quantum protocols which are more secure than current estimates suggest, for very similar reasons.

Outline

Chapter 1 introduces the notation and basic concepts of quantum computing, highlighting some of the problems mentioned above.

In Chapter 2, we define the memory peripheral model of quantum computers. This model emphasizes that quantum gates are processes that are enacted on data, rather than objects that data passes through. This further highlights the costs of the classical computation to control these gates. The formal definition gives a broad framework that can accomodate many different physical assumptions. We choose five models within this framework, based on whether error correction is "passive" or "active", whether the model

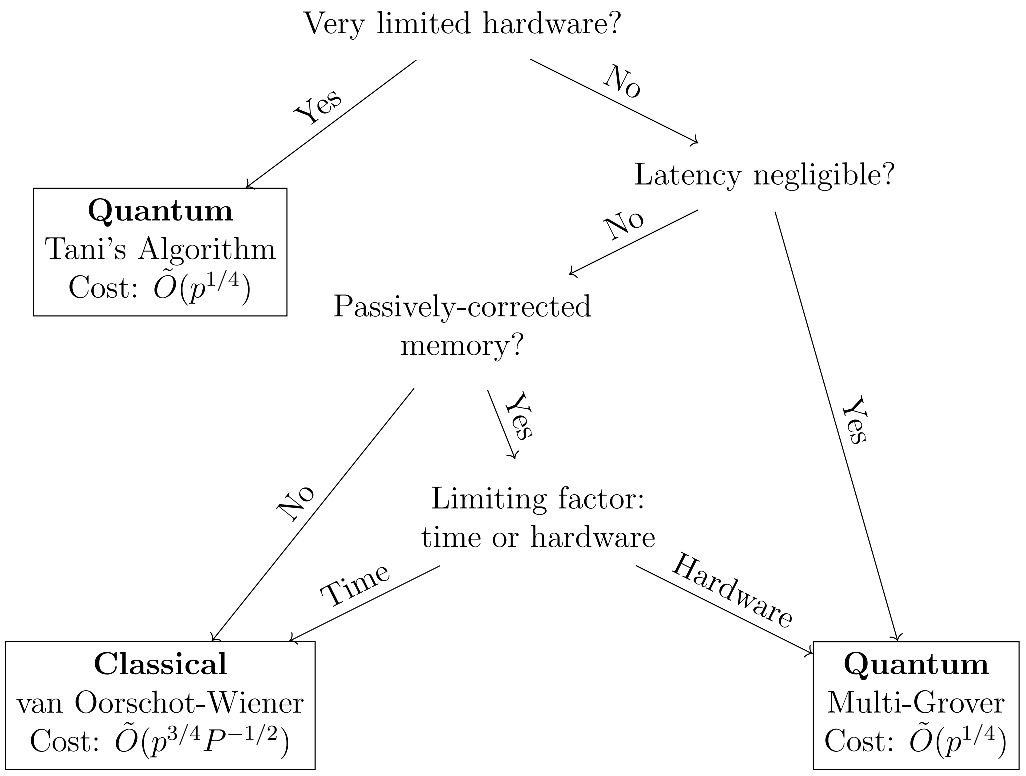


Figure 1: Cost-optimal algorithms to attack SIKE under different constraints and assumptions for. Costs are given in terms of p, a prime used as a parameter for SIKE (typically between 400 to 900 bits) and P, the number of parallel processors. Chapter 6 gives the full analysis, including the units of cost.

accounts for the physical geometry of the computer, and whether we assume cheap quantum random access memory.

We then explain the algorithms that we will analyze using these computational models. Chapter 3 introduces search algorithms based on quantum random walks, including Grover's algorithm and the Magniez-Roland-Santha-Nayak framework. The goal is to motivate quantum random walks with analogies to classical random walks, and describe the main steps of the algorithms without proving correctness.

Chapter 4 gives different methods for quantum data structures and memory access. Most quantum random walk applications use *Johnson graphs*, a type of graph where vertices are sets. Hence, they need efficient, history-independent data structures. Basic operations on these data structures end up surprisingly expensive compared to their classical analogues. This chapter shows that adapting basic data structures to the strengths and weaknesses of a quantum computer is underexplored and the optimal approaches barely resemble classical methods.

The focus is protocols based on supersingular isogenies, and we give a basic description of isogeny-based cryptography in Chapter 5. This frames the schemes in a way where attacks can be viewed as a *claw finding* problem. We also give estimates for classical and quantum costs to compute an isogeny.

Since claw-finding can break isogeny-based cryptography, we analyze the classical and quantum algorithms in Chapter 6. Specifically, we compare Grover, Tani, Multi-Grover, meet-in-the-middle, and van Oorschot-Wiener. Accounting for the costs given in the previous chapters, all the algorithms have the same exponential relationship between the total cost and the problem size. However, there are substantial differences in memory requirements and parallelism that complicate the analysis. We conclude that the optimal algorithm and the cost to run it will depend on the computational model and the budget of memory, processors, and time. In most parameterizations, the quantum Multi-Grover algorithm has the lowest cost, but accounting for physical geometry adds significant costs that favour the classical van Oorschot-Wiener algorithm.

Chapter 7 uses these cost estimates. First we argue that claw-finding is the most effective naive attack on supersingular isogeny-based cryptography and then we analyze NIST's security categories. We take an approach where security is based on comparisons: when is one system as secure as another "benchmark" system, such as AES. With this approach we evaluate the costs of attacks on isogeny-based cryptography and conclude that SIKE, the protocol under consideration for standardization, uses keys that are too large. Certain applications could use keys 42% smaller at the same security level.

Chapter 1

Quantum Computing Background

Section 1.1 gives the basic details of quantum computing, including the notation and techniques used throughout this thesis. Section 1.2 describes a few basic techniques of quantum computing that we will implicitly use later. Section 1.3 explains some of the difficulties quantum computers face, and how they can be "more powerful" than classical computers but be restricted in other ways. We discuss the particular issue of error correction in Section 1.4 and why it is fundamentally more difficult for quantum computers and why it may be a permanent component of quantum computers.

1.1 Notation and Basic Concepts

1.1.1 Classical Computing

Assuming the reader is familiar with classical computers, this section gives an unusual introduction of a classical computer as a vector space.

A classical binary computer contains some physical components, called bits, that exist in one of two *states*, denoted 0 or 1. These could be the magnetisation of a hard disc, the voltage in a wire, the frequency of a radio signal, etc. We can consider these as basis vectors of a 2-dimensional Hilbert space $\mathbb{C}^2$, denoted $\|\mathtt{0}\rangle\!\rangle$ and $\|\mathtt{1}\rangle\!\rangle$. This means we can define unusual things like $\frac{1}{2}\|\mathtt{0}\rangle\!\rangle - 5i\|\mathtt{1}\rangle\!\rangle$, although such mathematical objects have no physical meaning.

If we have two bits, then we can represent them as a tensor product: $\|\mathtt{b}_1\rangle\!\rangle \otimes \|\mathtt{b}_2\rangle\!\rangle$, which can now exist in four distinct states: `00`, `01`, `10`, and `11`. For notational convenience, we write this as $\|\mathtt{b}_1\rangle\!\rangle\,\|\mathtt{b}_2\rangle\!\rangle$ or even $\|\mathtt{b}_1\mathtt{b}_2\rangle\!\rangle$, where the tensor product is implied. This means n bits live in a vector space $\mathbb{C}^{2^n}$. Our computer can still only store one of 2^n different basis vectors, each of which represents a different possible bitstring. We call these basis vectors the "computational basis".

On these bits, we have gates. A gate maps input bits to output bits, with the output determined precisely by the input bits. In the vector space, this means it maps basis vectors to basis vectors, so we can extend it linearly and turn any gate into a linear transformation.

So far the vector notation adds nothing to the analysis. However, it helps with probabilitistic algorithms. Suppose we don't have a specific bitstring in the computer, but a probability distribution of bitstrings. We can represent this as a state

$$\|p\rangle\!\rangle = \sum_{\mathtt{b}\in\{\mathtt{0},\mathtt{1}\}^n} p(\mathtt{b})\ \|\mathtt{b}\rangle\!\rangle\,. \tag{1.1}$$

Suppose we have a gate G, represented as a linear transformation. Then we have

$$G\,\|p\rangle\!\rangle = \sum_{\mathtt{b}\in\{\mathtt{0},\mathtt{1}\}^n} p(\mathtt{b})G\,\|\mathtt{b}\rangle\!\rangle\,. \tag{1.2}$$

In other words, if we had probability $p(\mathtt{b})$ of bitstring $\mathtt{b}$, then after applying G, we still have probability $p(\mathtt{b})$ of the output bitstring when G is applied to $\mathtt{b}$, as we should.

Further, we could define a probabilistic gate by defining the output to be a convex combination of basis states. This implies that a gate must be column-stochastic.

Hence, we can capture any probabilistic algorithm with the convex hull of the computational basis. Anything outside the hull is still physically unrealistic. The vector notation means that we can use the same objects to represent any probability distribution of bitstrings, which we can call a "state". It's important to remember that for a basis state like $\|\mathtt{01101}\rangle\!\rangle$, the "01101" is just a label for the state, and that $\|\mathtt{01101}\rangle\!\rangle$ represents, in some sense, the physical system storing that data, rather than the data itself.

Example 1.1.1. *Let* $\|\mathtt{0}\rangle\!\rangle = \begin{pmatrix}1\\0\end{pmatrix}$ *and* $\|\mathtt{1}\rangle\!\rangle = \begin{pmatrix}0\\1\end{pmatrix}$. *We can represent an* AND gate as the following matrix, acting on $\mathbb{C}^2\otimes\mathbb{C}^2$:

$$\mathtt{AND} = \begin{pmatrix}1 & 1 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix}. \tag{1.3}$$

We can represent the state of a bitstring formed by flipping 2 coins as:

$$\left(\tfrac{1}{2}\,\|\mathtt{0}\rangle\!\rangle + \tfrac{1}{2}\,\|\mathtt{1}\rangle\!\rangle\right)\otimes\left(\tfrac{1}{2}\,\|\mathtt{0}\rangle\!\rangle + \tfrac{1}{2}\,\|\mathtt{1}\rangle\!\rangle\right) = \tfrac{1}{4}\left(\|\mathtt{00}\rangle\!\rangle + \|\mathtt{01}\rangle\!\rangle + \|\mathtt{10}\rangle\!\rangle + \|\mathtt{11}\rangle\!\rangle\right). \tag{1.4}$$

Applying AND to such a state will give $\frac{3}{4}\,\|\mathtt{0}\rangle\!\rangle + \frac{1}{4}\,\|\mathtt{1}\rangle\!\rangle$. In other words, flipping two coins and taking the "AND" of the result will give 0 with probability 3/4 and 1 with probability 1/4.

Any deterministic circuit can be generated by a single gate, NAND:

$$\mathtt{NAND} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}. \tag{1.5}$$

This means that for any matrix with entries in $\{0,1\}$ for which each column has precisely one entry equal to 1, there is some arrangement of NAND gates (and possibly extra bits initialized to 0) which will be equivalent to that matrix.

Any column-stochastic matrix will be a convex combination of such $\{0,1\}$ matrices. Adding a single "coin-flip" gate, which sends $\|\mathtt{0}\rangle\!\rangle$ to $\frac{1}{2}(\,\|\mathtt{0}\rangle\!\rangle + \|\mathtt{1}\rangle\!\rangle)$, we could use rejection sampling to produce any distribution we want, and use that distribution to decide which deterministic circuit to apply. This will create the necessary convex combination. Thus, NAND+"coin-flip" can generate any column-stochastic matrix.

Physically, we know many ways to construct NAND gates out of transistors, and many computers have ways to extract randomness from their environment.

1.1.2 Quantum States

In the classical case we could only physically construct states in the convex hull of the computational basis states. For a quantum computer, instead of that convex hull, states are limited to the surface of the unit sphere in the Hilbert space spanned by the computational basis. This implies we can have exotic states like

$$\frac{1}{\sqrt{2}}(|\mathtt{0}\rangle + i\,|\mathtt{1}\rangle), \tag{1.6}$$

which raises the important question: What *is* such a state? The classical states were easy to interpret, as probability distributions of physical phenomena like combinations of voltage levels. Quantum states still describe physical systems, but answering what a specific state physically represents is a heated, ongoing philosophical debate.

For our purposes, we use a Hilbert space because a quantum state must be a solution to a linear equation called Schrödinger's equation:

$$i\hbar\frac{d}{dt}\,|\psi\rangle = \hat{H}\,|\psi\rangle\,. \tag{1.7}$$

Here $\hat{H}$ is a linear operator called the Hamiltonian and $\hbar$ is Planck's constant.

We restrict to the unit sphere because we can perform a physical process that *measures* a quantum state according to an orthonormal basis. If we measure a state $|\psi\rangle$ according to a basis $\{|\psi_1\rangle, \cdots, |\psi_n\rangle\}$, then the probability of reading the result "i" is $|\langle\psi_i|\psi\rangle|^2$. In other words, the squared modulus of the coefficients of a quantum state form a probability distribution of measurement results, and hence the state must be a unit vector in the ℓ^2-norm.

If we consider the state in Equation 1.6, measuring it in the basis $\{|\mathtt{0}\rangle, |\mathtt{1}\rangle\}$ gives a result of `0` with probability 1/2 and a `1` with probability 1/2.

We can also build quantum states out of a basis besides the computational basis, and we can construct measurements that are not defined by a particular basis. These generalizations do not impact the quantum algorithms we analyze in this thesis, so we will ignore them.

1.1.3 Operations

Classically, preserving the convex hull meant restricting to column-stochastic linear transformations. To preserve the unit sphere, transformations must be unitary. In an analogy to universal `NAND` gates for classical computing, we define the following:

Definition 1.1.1. *A set $\mathcal{G}$ of quantum gates is universal if, for any unitary U and any $\epsilon > 0$, there is a finite set of gates $G_1, \cdots, G_n \in \mathcal{G}$ such that*

$$\|U - G_nG_{n-1}\cdots G_2G_1\| < \epsilon, \tag{1.8}$$

where $\|\cdot\|$ is the operator norm.

There are physically realizable universal sets of quantum gates, meaning that we can apply any unitary we wish to a quantum computer by some set of gates. However, there are practical concerns about how many gates we will need and how difficult it is to apply each gate.

Clifford + T Gates

The most prominent universal gate set is the Clifford+T set.

It contains the single-qubit Pauli gates:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \; X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \; Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \; Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}. \tag{1.9}$$

These have some important properties:

1. Each non-identity gate has order 2, is unitary, and is Hermitian (it equals its own conjugate transpose).
2. Adding in all scalar multiples by $\{\pm 1, \pm i\}$, the Pauli gates form a multiplicative group.
3. $XY = -YX = iZ$, $YZ = -ZY = iX$, and $ZX = -XZ = iY$.

These are all single-qubit gates, but we can construct the n-qubit Pauli group by independently acting on different qubits with Pauli gates:

$$\mathcal{P}_n = \{P_1 \otimes \cdots \otimes P_n \,|\, P_i \text{ is a Pauli gate for all } i\}. \tag{1.10}$$

The Clifford group is defined as the *commutator* of the n-qubit Paulis. This is the group defined as

$$\{U \text{ unitary} \,|\, UPU^* \text{ is an } n\text{-qubit Pauli gate, for all } n\text{-qubit Pauli gates } P\}. \tag{1.11}$$

Together with the Pauli group, the following extra gates, acting on any qubit or pairs of qubits, generate the Clifford group:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \; S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, \; \texttt{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{1.12}$$

Two things make the Clifford group special. First, it can be classically simulated in polynomial time (the "Gottesman-Knill Theorem"). Second, even though the Cliffords are not a universal gate set, adding *any* other gate to it will give a universal gate set. The typical choice is the T-gate:

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{pmatrix}. \tag{1.13}$$

Adding the T-gate to the Clifford group makes it hard to simulate classically. High-fidelity physical methods (those with error correction, Section 1.4) for Clifford gates typically have trouble with T gates, and vice versa.

1.1.4 Computing

Circuits

By initializing the quantum states as bitstrings, and receiving output as bitstrings, we hope to perform some computation with quantum states. We describe these using quantum circuits, which might look like Figure 1.1.

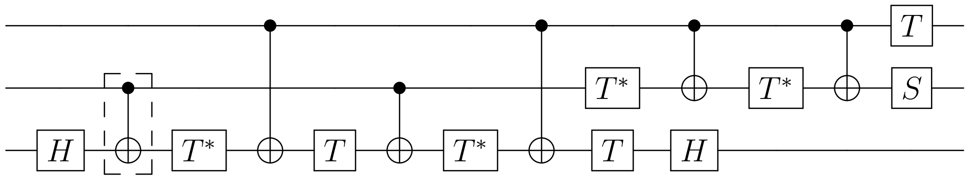


Figure 1.1: A quantum circuit

The wires represent individual qubits, and the boxes represent gates. Most gates are drawn with a box with the name of the gate in the box, like the T and H gates. `CNOT` gates are drawn with a $\bullet$ for the first qubit, the control, and a $\oplus$ for the second qubit, the target, as the box in Figure 1.1 shows.

In a quantum circuit diagram, the horizontal axis represents time, flowing from left to right. In Figure 1.1, the first thing to happen is an H gate is applied to the third qubit, then a `CNOT` to the second and third qubits, etc.

Quantum circuit diagrams deliberately resemble classical circuit diagrams such as Figure 1.2, though classical circuit diagrams are an abstraction of a 2-dimensional layout of wires and gates. To emphasize this distinction:

- Gates in a quantum circuit diagram represent a *process* enacted on the data;
- Gates in a classical circuit diagram represent an *object*.

In a classical circuit, data exists as electrical signals that move through the gates. Thus we often equivocate between the gate as an *object* and the gate as a *process* of signals moving through the gate.

We could interpret the horizontal axis in Figure 1.2 as time and hence the gates as processes. One can imagine that the circuit diagram is printed as wires and gates onto a physical chip, and signals enter at the left and propagate through the circuit as time progresses.

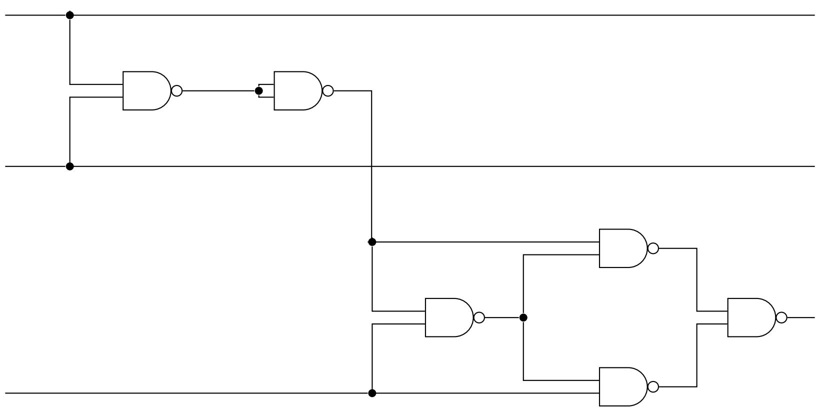


Figure 1.2: A classical circuit made from `NAND` gates

This means that, classically, we can incur a fixed cost to construct a circuit, then each time we use the circuit we only incur some cost to propagate the signal through. Often this cost is energy or time. As an example, random access memory requires a large number of gates, but once these gates are built, we can use them repeatedly, quickly, and cheaply.

Promising quantum computing technologies are different. The data is stationary and the gates are applied to it; there is no object that corresponds to a gate. In this way, quantum data more closely resembles a magnetic hard drive, where bits are stationary magnetized regions, and the "gates" are patterns of bit flipping performed by a moving disk head. The memory peripheral model of quantum computers (Section 2.2) formalizes this analogy.

The implication is that if we design a quantum algorithm that uses one subroutine many, many times (such as memory access), we have to pay the same cost each time we use it. Whatever each gate costs, we must "pay" this cost every time we apply the gate.

Computing Power

Since all quantum gates are unitaries, which are invertible, every quantum computation must be reversible, so we cannot directly apply a simple classical universal gate like `NAND`.

Instead we use a `TOFFOLI` gate, which looks like the following:

$$\texttt{TOFFOLI} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}. \tag{1.14}$$

Note that:

1. On computational basis inputs, the `TOFFOLI` gate has no effect on the first 2 bits, and flips the final bit if and only if the first two bits are `1`. This gate is also called a "controlled controlled `NOT`", because it is a `NOT` gate controlled by the first two bits.

2. This is a deterministic classical gate. In fact, Figure 1.2 gives a classical circuit to implement it.

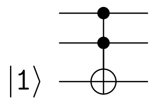
3. This is a unitary matrix, and Figure 1.1 gives a quantum circuit to implement it with Clifford+T. Technically that circuit also uses T^* gates, but $T^* = T^7$, so it could be implemented with just Clifford+T.

Importantly, `TOFFOLI` is universal for reversible classical computation [1]. Any non-reversible classical computation can be simulated with a reversible computation. To do this, we use ancillae. An ancilla (qu)bit is a (qu)bit that stores extra information before, during, or after a computation, and is not involved in the full computation.

For example, Figure 1.3 shows how to implement a `NAND` gate with a `TOFFOLI` gate. We start with the first two inputs, initialize an ancilla qubit to $|\texttt{1}\rangle$, then use the `TOFFOLI`. The ancilla becomes the output data.

The two bits of input are left over and useless, since a `NAND` gate does not need them, but we cannot remove them in a reversible computation. Hence, they must remain as ancillae.

Since a `TOFFOLI` gate can simulate a `NAND` gate, anything we can do on a non-reversible classical computer we can do reversibly, and anything we can do reversibly we can do with quantum data. Thus, in an abstract sense, a quantum computer is at least as powerful as a classical computer.



(a) Circuit

Input	Data
001	001
011	011
101	101
111	110

(b) Truth table. "Ancillae" are in grey.

Figure 1.3: Reversibly simulating a `NAND` gate with a `TOFFOLI` gate.

Superpositions and Interference

Arguably the two most important extra capabilities of quantum computers are superposition and interference.

Looking at the H gate in Section 1.1.3, called the `HADAMARD` gate, its action on the $|0\rangle$ state produces $\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. We call this a uniform superposition: We have both computational basis states with an equal probability of measuring either one.

If we apply H gates to many qubits in parallel, we get the following:

$$(H \otimes \cdots \otimes H)(|0\rangle \cdots |0\rangle) = \frac{1}{2^{n/2}} \sum_{b \in \{0,1\}^n} |b\rangle \,. \tag{1.15}$$

Here we have every possible n-bit string in superposition, after applying only n gates. If we have some computation represented as a unitary U, we can apply U to this state:

$$U \frac{1}{2^{n/2}} \sum_{b \in \{0,1\}^n} |b\rangle = \frac{1}{2^{n/2}} \sum_{b \in \{0,1\}^n} U |b\rangle \,. \tag{1.16}$$

Now our state is a uniform superposition over all possible outputs. It's almost as though we have computed every possible input at once. This *seems* very powerful for search problems, however this is still no better than a probabilistic classical algorithm.

Suppose we define a classical "Hadamard":

$$H_c = \frac{1}{2} \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix} . \tag{1.17}$$

This takes any bit as input, and outputs $\mathtt{0}$ or $\mathtt{1}$ with equal probability. If we apply this to n bits, we get the state

$$\frac{1}{2^n}\sum_{\mathtt{b}\in\{\mathtt{0},\mathtt{1}\}^n} \| \mathtt{b} \rangle\rangle \,. \tag{1.18}$$

and we could apply any circuit U and get

$$\frac{1}{2^n}\sum_{\mathtt{b}\in\{\mathtt{0},\mathtt{1}\}^n} U \| \mathtt{b} \rangle\rangle \,. \tag{1.19}$$

This represents a distribution where we have equal probability of every possible output of U. Note that this is exactly the distribution we would get from measuring the state in Equation 1.16, so there is no quantum advantage yet.

The quantum advantage is *interference*. Unlike probabilistic classical computations, states can have negative coefficients. This means we can have states that cancel out.

As a very basic example, consider that

$$H|\mathtt{0}\rangle = \tfrac{1}{\sqrt{2}}(|\mathtt{0}\rangle + |\mathtt{1}\rangle). \tag{1.20}$$

If we apply H again, we can compute its action on each basis state:

$$\begin{aligned} H\tfrac{1}{\sqrt{2}}(|\mathtt{0}\rangle + |\mathtt{1}\rangle) =& \tfrac{1}{\sqrt{2}}(H|\mathtt{0}\rangle + H|\mathtt{1}\rangle) & (1.21)\\ =& \tfrac{1}{2}\left((|\mathtt{0}\rangle + |\mathtt{1}\rangle) + (|\mathtt{0}\rangle - |\mathtt{1}\rangle)\right). & (1.22)\end{aligned}$$

Looking at this final state, we see that the $|\mathtt{1}\rangle$ states cancel, so the final output is $|\mathtt{0}\rangle$. This means $H^2|\mathtt{0}\rangle = |\mathtt{0}\rangle$; in fact, $H^2 = I$.

In an extremely rough sense, all the useful quantum algorithms create a superposition of desirable and undesirable outputs, then they find some way to cause good outputs to interfere constructively and bad outputs to interfere destructively.

Gate requirements

If we build complicated circuits out of a small set of universal gates, it's natural to ask how many gates we need to build a circuit that will perform a specific task. For instance, how many gates do we need to sort an array of elements?

There is a specific, obvious lemma that will be useful later. It essentially says that if you have a circuit that has the potential to alter any one of n qubits, then it uses $\Omega(n)$ gates.

Lemma 1.1.1. *Let U be a circuit that acts on n qubits, built from gates that act on at most k qubits. Then either:*

- *U uses as least n/k gates.*
- *Up to some fixed permutation of the qubits, U can be written as $U = U' \otimes I_2^{\otimes m}$ for some U' and $m \geq 1$.*

Proof. Suppose the first statement does not hold, and U uses $\ell < n/k$ gates. Each gate acts on at most k qubits, so together the gates act on at most $\ell k < n$ qubits. The action of U on the remaining qubits must be the identity, so we can write $U = U' \otimes I_2^{\otimes(n-\ell k)}$. □

Lemma 1.1.1 only applies to a model like the circuit model where any gate that *might* be necessary must be used. In a different model, the computer may be able to use the input to only apply the gates needed just for that one input. A dynamic model like this is a reasonable way to model a classical computer. For this reason, the RAM model (Definition 2.2.4) gives only unit cost to memory access, even though Lemma 1.1.1 states that a random memory access circuit to n bits needs n/k gates of arity k.

Should we model quantum computations in the same way? In most quantum technologies, a classical controller must apply the gates. Since the classical controller cannot read the contents of a quantum state without modifying or destroying the state, it cannot "dynamically generate" a quantum circuit in the same way that it can for a classical circuit. Methods for a quantum controller to generate a circuit rely on controlled gates, but the issue remains if a classical controller must apply the controlled gates.

1.2 Basic Techniques

1.2.1 Uncomputing Ancillae

Suppose we have some computable function $f : \{\texttt{0},\texttt{1}\}^n \to \{\texttt{0},\texttt{1}\}^n$ that is not injective. Then we cannot construct a quantum circuit U_f such that

$$U_f |\texttt{b}\rangle = |f(\texttt{b})\rangle \tag{1.23}$$

since this is not reversible. We *must* have some ancillary data that would, in principle, allow us to reconstruct $\texttt{b}$ from $f(\texttt{b})$. Looking at the simulation of a `NAND` gate in Figure 1.3, the well-known method is to simply keep the input $\mathtt{b}$. This means that for any classically computable function f, we would like to construct a quantum circuit U_f such that

$$U_f \left|\mathtt{b}\right\rangle \left|\mathtt{0}\right\rangle^n = \left|\mathtt{b}\right\rangle \left|f(\mathtt{b})\right\rangle^n . \tag{1.24}$$

However, many times we will instead have a circuit U_f such that

$$U_f \left|\mathtt{b}\right\rangle \left|\mathtt{0}\right\rangle^n \left|\mathtt{0}\right\rangle^m = \left|\mathtt{b}\right\rangle \left|f(\mathtt{b})\right\rangle^n \left|g(\mathtt{b})\right\rangle^m . \tag{1.25}$$

Here, g is a function that produces ancilla qubits. To remove them, we introduce a fourth register of n bits, and then apply U_f as normal:

$$U_f(\left|\mathtt{b}\right\rangle \left|\mathtt{0}\right\rangle^n \left|\mathtt{0}\right\rangle^m \left|\mathtt{0}\right\rangle^n) = \left|\mathtt{b}\right\rangle \left|f(\mathtt{b})\right\rangle \left|g(\mathtt{b})\right\rangle \left|\mathtt{0}\right\rangle^n . \tag{1.26}$$

Then use a `CNOT` from the second register to the fourth:

$$\left|\mathtt{b}\right\rangle \left|f(\mathtt{b})\right\rangle \left|g(\mathtt{b})\right\rangle \left|\mathtt{0}\right\rangle^n \mapsto \left|\mathtt{b}\right\rangle \left|f(\mathtt{b})\right\rangle \left|g(\mathtt{b})\right\rangle \left|f(\mathtt{b})\right\rangle . \tag{1.27}$$

Then reverse U_f on the first three registers:

$$U_f^{-1}(\left|\mathtt{b}\right\rangle \left|f(\mathtt{b})\right\rangle \left|g(\mathtt{b})\right\rangle) \left|f(\mathtt{b})\right\rangle = \left|\mathtt{b}\right\rangle \left|\mathtt{0}\right\rangle^n \left|\mathtt{0}\right\rangle^m \left|f(\mathtt{b})\right\rangle . \tag{1.28}$$

This has an overall action of

$$\left|\mathtt{b}\right\rangle \left|\mathtt{0}\right\rangle^n \mapsto \left|\mathtt{b}\right\rangle \left|f(\mathtt{b})\right\rangle , \tag{1.29}$$

and it only doubles the computation and the space that U_f requires.

We can summarize this result as:

Theorem 1.2.1. *Given a classical circuit of N* `NAND` *gates to compute a function $f : \{\mathtt{0},\mathtt{1}\}^n \to \{\mathtt{0},\mathtt{1}\}^m$, we can construct a quantum circuit U_f with $N + n + 2m$ qubits and $2N + m$ gates with action on computational basis states:*

$$U_f \left|\mathtt{b}\right\rangle \left|\mathtt{0}\right\rangle^{2m} \left|\mathtt{1}\right\rangle^N = \left|\mathtt{b}\right\rangle \left|f(\mathtt{b})\right\rangle \left|\mathtt{0}\right\rangle^m \left|\mathtt{1}\right\rangle^N . \tag{1.30}$$

The $\left|\mathtt{1}\right\rangle$ states are used as ancilla to create `NAND` out of `TOFFOLI`, as in Figure 1.3. The specific overhead will change for a classical circuit that uses different gates.

1.2.2 Arbitrary Qubit Rotations

Many algorithms need states of the form $|\psi_p\rangle := \sqrt{p}\,|\mathtt{0}\rangle + \sqrt{1-p}\,|\mathtt{1}\rangle$. To construct this from Clifford+T gates, we use Kliuchnikov, Maslov, and Mosca's method [38] to map

$$\tfrac{1}{\sqrt{2}}(|\mathtt{0}\rangle + |\mathtt{1}\rangle) \mapsto \tfrac{1}{\sqrt{2}}(|\mathtt{0}\rangle + e^{i\theta}\,|\mathtt{1}\rangle) \tag{1.31}$$

for some angle θ. The fidelity (Section 1.4.2) of this process is exponential in a parameter k, and the circuit uses $O(k)$ gates.

We then apply ZT^2H to the state, which maps it to

$$\cos\left(\tfrac{\theta}{2}\right)|\mathtt{0}\rangle + \sin\left(\tfrac{\theta}{2}\right)|\mathtt{1}\rangle\,. \tag{1.32}$$

We will use G_p to denote the circuit with action $G_p\,|\mathtt{0}\rangle = |\psi_p\rangle$.

The method to find the state in 1.31 follows a very general construction, and there may be a more efficient method specific to this problem.

1.2.3 Superpositions of Arbitrary Lists

Applying n Hadamard gates to $|\mathtt{0}\rangle^n$ produces the following superposition:

$$\frac{1}{2^{n/2}}\sum_{\mathtt{b}\in\{\mathtt{0},\mathtt{1}\}^n}|\mathtt{b}\rangle\,. \tag{1.33}$$

In many applications, we may want to produce a superposition such as

$$\frac{1}{\sqrt{N}}\sum_{b<N}|\mathtt{b}\rangle \tag{1.34}$$

where N is not a power of 2. Here b is the integer that $\mathtt{b}$ represents.

One approach is to use arbitrary qubit rotations. When constructing these superpositions for powers of 2, we can consider that each Hadamard gate "splits the amplitude" in half for each bit. With arbitrary superpositions, we want to "split the amplitude" proportional to how many states have each bit equal to $\mathtt{0}$ or $\mathtt{1}$.

Suppose N is n bits long, meaning $2^n < N < 2^{n+1}$. Let $\mathtt{N}_n \cdots \mathtt{N}_1$ be the binary representation of N, and let $p_m = 2^{\mathtt{N}_n\cdots\mathtt{N}_m\mathtt{0}\cdots\mathtt{0}}/N$. We start by creating the state $\sqrt{p_n}\,|0\rangle + \sqrt{1-p_n}\,|1\rangle$ in the first qubit.

We then iterate left to right through the remaining qubits, applying the following controlled circuit to the mth qubit:

- If the current bitstring $\mathtt{b}_n \cdots \mathtt{b}_m$ is different than $\mathtt{N}_n \cdots \mathtt{N}_m$, apply a Hadamard to the mth qubit.
- If the current bitstring is the same, then control based on $\mathtt{N}_{m-1}$:
    - If $\mathtt{N}_{m-1} = \mathtt{0}$, do nothing.
    - If $\mathtt{N}_{m-1} = \mathtt{1}$, apply the rotation circuit $G_{p_{m-1}}$.

Controlled gates will execute the "if" statements: A circuit will compare the current bitstring, in superposition, and use the result to control the necessary circuit.

The comparison circuit, which compares to a fixed classical string of m bits, can be done with $O(m)$ gates. Since G_p requires $O(k)$ gates for the precision parameter k, this circuit will require $O(\lg N(\lg N + k))$ gates.

1.2.4 Probabilistic Algorithms

Often we design algorithms that only succeed with some probability. Suppose we want to compute some function f and we have a probabilistic, classical circuit C_f that succeeds with probability $1-p$, and otherwise computes some undesirable function g. Assume that C_f has some detection circuit that will detect if the computation of f failed. The action of C_f will look like

$$C_f \left\| x \right\rangle\!\!\rangle = (1-p) \left\| f(x) \right\rangle\!\!\rangle \left\| \mathtt{0} \right\rangle\!\!\rangle + p \left\| g(x) \right\rangle\!\!\rangle \left\| \mathtt{1} \right\rangle\!\!\rangle , \tag{1.35}$$

where the final bit indicates whether U_f succeeded or not.

Classically, we can apply another circuit that checks if the final bit is $\mathtt{1}$ — meaning the computation failed — and if so, it erases everything, removes the check bit, and applies C_f again:

$$\begin{aligned}(1-p) \left\| f(x) \right\rangle\!\!\rangle \left\| \mathtt{0} \right\rangle\!\!\rangle + p \left\| g(x) \right\rangle\!\!\rangle \left\| \mathtt{1} \right\rangle\!\!\rangle \mapsto & (1-p) \left\| f(x) \right\rangle\!\!\rangle + p(C_f \left\| \mathtt{0} \right\rangle\!\!\rangle) & (1.36)\\ = & (1-p^2) \left\| f(x) \right\rangle\!\!\rangle \left\| \mathtt{0} \right\rangle\!\!\rangle + p^2 \left\| g(x) \right\rangle\!\!\rangle \left\| \mathtt{1} \right\rangle\!\!\rangle . & (1.37)\end{aligned}$$

We can repeat this k times and drive the probability of failure down to p^k, which is exponential in the number of repetitions.

We would like a similar quantum procedure for a quantum circuit U_f, but the classical method required erasing $g(x)$. In the quantum setting we cannot erase. We must "uncompute" $g(x)$ with some circuit U_g^{-1}, raising the following issues:

1. This may be computationally expensive or difficult to design.
2. It may be impossible. We may have two orthogonal states $|x\rangle$ and $|y\rangle$ such that $U_f|x\rangle$ is orthogonal to $U_f|y\rangle$ (it must be, since U_f is unitary) but for which $|g(x)\rangle$ is not orthogonal to $|g(y)\rangle$.
3. Even if we have some circuit U_g^{-1} that maps $|g(x)\rangle$ to $|x\rangle$, if we apply it to $U_f|x\rangle$ controlled by the failure detection qubit, we would get

$$\sqrt{1-p}\,|f(x)\rangle\,|\mathtt{0}\rangle + \sqrt{p}\,|x\rangle\,|\mathtt{1}\rangle \tag{1.38}$$

and if we then apply U_f again based on the control qubit, we end up with

$$\begin{aligned}\sqrt{1-p}\,|f(x)\rangle\,|\mathtt{00}\rangle &+ \sqrt{p}\left(\sqrt{1-p}\,|f(x)\rangle\,|\mathtt{0}\rangle + \sqrt{p}\,|g(x)\rangle\,|\mathtt{1}\rangle\right)|\mathtt{1}\rangle \\ &= \sqrt{1-p^2}\,|f(x)\rangle\,|0\rangle\,\underbrace{\tfrac{1}{\sqrt{1+p}}(|0\rangle + \sqrt{p}\,|1\rangle)}_{=|\psi_p\rangle} + p\,|g(x)\rangle\,|\mathtt{11}\rangle\,. \end{aligned} \tag{1.39}$$

Now we are stuck with an extra $|\psi_p\rangle$ state entangled with our result.

If we repeat the procedure k times, we end up with k garbage ancilla qubits in the same state $|\psi_p\rangle$. We will give a classical and quantum approach to remove this.

Classical garbage removal

We can remove the extra qubits one at a time by measuring. After applying U_f once to get the state in Equation 1.38, we measure the failure detection qubit.

With probability $1-p$, we measure 0, telling us the remaining register is $|f(x)\rangle$, and we can continue the computation.

With probability p we measure 1 and we have $|g(x)\rangle$. Then we apply our uncomputation U_g^{-1}, which restores our state $|x\rangle$, and then we can simply repeat.

Once we measure a 0, we stop and continue, with a guarantee that we have the right output. This will take an average of $1/(1-p)$ repetitions.

The problem with this approach is that some applications do not “allow” the classical computer to know whether U_f was applied at all. If we control the application of U_f with another qubit and we measure 1 in the garbage removal, we know that the control qubit must have been 1, and this destroys any superposition of the control qubit. Hence, for many applications, classical garbage removal is insufficient.

Quantum garbage removal

The circuit G_p from Section 1.2.2 is made of Clifford+T gates, so its inverse has the same cost. Then we can apply it to each ancillae. Since $G_p^{-1}|\psi_p\rangle = |\mathtt{0}\rangle$, this uncomputes the effect of the circuit on the ancillae.

One drawback is that we need to know the precise value of p to construct the circuit G_p^{-1}.

1.3 Difficulties

Because quantum computers can simulate classical computers, we might wish to build a unitary to do anything a classical computer can, then combine it with a quantum circuit. But there is no guarantee that our simulation of a classical computation will work well with the quantum phenomena of interference and superposition.

This is frequently ignored or given little attention in the literature. Works on random walks have included steps that are not reversible, set-ups not guaranteed to be free of garbage ancillae, and sampling from potentially difficult-to-sample distributions. Some of these "problems" are not problems in their original context using an oracle model, but this is easy to forget when using the results in other contexts.

1.3.1 Reversibility

Consider the ancillae removal of Section 1.2.1. Suppose we are in the opposite situation: We would like to keep the ancillae $g(\mathtt{b})$, and "uncompute" the input $\mathtt{b}$. There is no general way to do this. We need a function $h : \{\mathtt{0},\mathtt{1}\}^{n+m} \to \{\mathtt{0},\mathtt{1}\}^n$ such that $h(f(\mathtt{b}), g(\mathtt{b})) = \mathtt{b}$. This function might *exist*, but it might not be computationally feasible.

Further, we may have a function f that is bijective. It is then theoretically possible to produce a circuit U_f such that

$$U_f |\mathtt{b}\rangle = |f(\mathtt{b})\rangle \tag{1.40}$$

but again, this may not be computationally feasible. There is no general technique to produce such a U_f.

As an example, consider the problem of shuffling an array of n elements. One method to do this is a Knuth shuffle: Start at the first element and work towards the end, and for each element i, swap it with a random element j between i and the end of the array (inclusive). We imagine this as a function whose input is a list of n elements and a list of n random numbers from the appropriate range (the first number is from 1 to n, the second is from 2 to n, etc.), and whose output is a shuffled list of n elements.

Immediately we see that since the original Knuth shuffle can shuffle any initial ordering into any other ordering, it would be non-reversible if we output *just* the shuffled list; we will assume we want the output to be the shuffled list and the initial list, but *not* the list of n numbers.

This is possible with no asymptotic overhead but the solution is not trivial. Despite the lack of such a technique in the literature, many algorithms rely on initializing a uniform superposition of random subsets without leaving any "garbage" ancillae. Section 4.3.3 explores this issue in more detail.

1.3.2 No-Cloning Theorem

The "no-cloning" theorem states that there is no unitary that can take an arbitrary state $|\psi\rangle|0\rangle$ and map it to $|\psi\rangle|\psi\rangle$. This is a simple consequence of the fact that unitaries preserve distance. But it creates challenges for quantum computing, because we cannot arbitrarily duplicate quantum data. This is something we are used to doing with classical data.

This should sound contradictory, since classical data is a subset of quantum data. If we can duplicate classical data, shouldn't we be able to duplicate classical data as represented with quantum states? This is not a paradox because the no-cloning theorem states that we cannot duplicate an *arbitrary* state $|\psi\rangle$. However, it is possible to construct a unitary which duplicates a single fixed orthogonal basis.

For example, the `CNOT` gate from Section 1.1.3 has the following action:

$$\mathtt{CNOT}\,|\mathtt{0}\rangle\,|\mathtt{0}\rangle = |\mathtt{0}\rangle\,|\mathtt{0}\rangle\,,\ \mathtt{CNOT}\,|\mathtt{1}\rangle\,|\mathtt{0}\rangle = |\mathtt{1}\rangle\,|\mathtt{1}\rangle\,. \tag{1.41}$$

Thus, it copies the computational basis states. On other states, the behaviour is the linear extension, but it doesn't copy:

$$\mathtt{CNOT}\tfrac{1}{\sqrt{2}}(|\mathtt{0}\rangle + |\mathtt{1}\rangle)\,|\mathtt{0}\rangle = \tfrac{1}{\sqrt{2}}(|\mathtt{00}\rangle + |\mathtt{11}\rangle). \tag{1.42}$$

This is not equal to two copies of the first qubit, which would instead be equal to:

$$\tfrac{1}{\sqrt{2}}(|\mathtt{0}\rangle + |\mathtt{1}\rangle) \otimes \tfrac{1}{\sqrt{2}}(|\mathtt{0}\rangle + |\mathtt{1}\rangle) = \tfrac{1}{2}(|\mathtt{00}\rangle + |\mathtt{01}\rangle + |\mathtt{10}\rangle + |\mathtt{11}\rangle), \tag{1.43}$$

though the state in Equation 1.42 is called a Bell state and is very important in its own right.

Frequently, descriptions of quantum algorithms will say that a state is "copied" to another register. This just means that the computational basis states are copied with `CNOT` gates.

1.3.3 Entanglement

Formally, a state $|\psi\rangle \in \mathcal{H}_A \otimes \mathcal{H}_B$ is called *entangled* if it cannot be written as $|\psi_A\rangle \otimes |\psi_B\rangle$. For quantum computing, this means we have a state that maybe looks like this:

$$|\mathtt{00}\rangle\,|\mathtt{0}\rangle + |\mathtt{01}\rangle\,|\mathtt{1}\rangle + |\mathtt{10}\rangle\,|\mathtt{1}\rangle + |\mathtt{11}\rangle\,|\mathtt{0}\rangle\,. \tag{1.44}$$

The last register is the `XOR` of the first 2. We cannot write this as a product of two states in each system. In such a situation, we will say that register 3 is *entangled* with registers 2 and 3. This thesis will sometimes also refer to a specific state in superposition, like $|\mathtt{01}\rangle\,|\mathtt{1}\rangle$, and say that $|\mathtt{01}\rangle$ is entangled with $|\mathtt{1}\rangle$. This is imprecise since, as a single state, $|\mathtt{01}\rangle\,|\mathtt{1}\rangle$ is *not* entangled, but this use implicitly refers to a state in superposition.

Entanglement can be very useful but it can also be a problem. Recall that $HH\,|\mathtt{0}\rangle = |\mathtt{0}\rangle$. But what if, after the first `HADAMARD` gate, we applied a `CNOT` gate like Equation 1.42, and *then* tried to invert H. We would get:

$$(H \otimes I)\mathtt{CNOT}(H \otimes I)\,|\mathtt{00}\rangle = (H \otimes I)(|\mathtt{00}\rangle + |\mathtt{11}\rangle \tag{1.45}$$

$$= |\mathtt{00}\rangle + |\mathtt{10}\rangle + |\mathtt{01}\rangle - |\mathtt{11}\rangle\,. \tag{1.46}$$

This is not equal to $|\mathtt{00}\rangle$! After the `CNOT` gate, the two registers are entangled, and the desired interference does not occur.

This is especially problematic for reversible circuits. Suppose we create some superposition of inputs x with coefficients α_x and compute some function $f(x)$, such as

$$\sum_{x \in X} \alpha_x\,|x\rangle\,|f(x)\rangle\,. \tag{1.47}$$

We might wish to cause interference among the states $|f(x)\rangle$ based on some property of f, but they are entangled with the inputs x, and so such interference cannot occur. We would need to somehow uncompute $|x\rangle$ to remove the entanglement, and as discussed in Section 1.2.1, there is no efficient general method to do this.

1.4 Error Correction

Computers exist in the real world, where they are imperfect and subject to noise. We can model noise as a set of operations called *noise operations*. These are the same kind of operations that we use to do computation: stochastic matrices for classical, unitaries for quantum.

For example, we could imagine our set of noise operations is $\{I\} \cup \{E_i | 1 \leq i \leq n\}$, where E_i acts on n-bit strings and flips the ith bit. We imagine that during certain points of our computation, a random noise operator (possibly just the identity!) is applied to our state according to some probability distribution.

Classically, there is an enormous field of research in *error correcting codes*. An error correcting code provides a way to encode a binary string s into a larger string $\overline{s}$, with a method to decode $\overline{s}$ back into s. We want our code to correct a set of noise operations, meaning if we take any noise operator E in our set, apply E to $\overline{s}$, then decode it, it should still return s.

A quantum error-correcting code (QECC) is a similar process that encodes states $|x\rangle$ into states $|\overline{x}\rangle$ in a higher-dimensional space, with some projection P that maps $|\overline{x}\rangle$ back to $|x\rangle$. We call an encoded qubit a "logical qubit". For every QECC, we have some set $\mathcal{N}$ of quantum noise operations, and we want:

$$P(E\,|\overline{x}\rangle) = |x\rangle\,, \text{ for all } E \in \mathcal{N}. \tag{1.48}$$

This is very similar to classical error correction, with a few important differences.

1.4.1 Quantum Channels

Recall that the classical computing formalism we introduced can handle probabilistic operations. Thus, given a set of noise operations N and some probability distribution P of different noise operators, we can define

$$\Phi_N = \sum_{E \in N} P(E)E \tag{1.49}$$

and Φ_N will still be a valid operator, which represents the entire noisy process, which we could call a *channel*.

If we try this in the quantum setting, with noise operations $\mathcal{N}$:

$$\Phi_{\mathcal{N}} = \sum_{U \in \mathcal{N}} P(U)U \tag{1.50}$$

this is no longer a unitary operator and thus is not a valid quantum operation in the framework described so far.

The problem is that we have mixed classical probabilities with quantum operations. Such operations require a more general formalism (density matrices and quantum channels). The definition of a memory peripheral (Definition 2.2.2) uses quantum channels, but nothing else in this thesis needs these tools.

1.4.2 Fidelity

Classically if we try to compute some string x but our circuit produces a state such as

$$(1-p)\,\|x\rangle\!\rangle + p\,\|y\rangle\!\rangle \tag{1.51}$$

we say that the circuit succeeded with probability $1-p$. If we use this state as the input to another algorithm that expects x as an input, then the probability of success will remain $1-p$.

The same concept holds for quantum states. We define the *fidelity* between two quantum states $|\psi\rangle$ and $|\phi\rangle$ as

$$F(|\psi\rangle, |\phi\rangle) = |\langle\psi|\,\phi\rangle|^2. \tag{1.52}$$

This takes a value between 0 and 1. The fidelity is 1 if and only if $|\psi\rangle = |\phi\rangle$, and it is 0 if and only if they are orthogonal.

To interpret fidelity, note that if we measure $|\phi\rangle$ in a basis that contains $|\psi\rangle$, we will measure ψ with probability equal to $F(|\phi\rangle, |\psi\rangle)$.

Since unitary matrices preserve the inner product, we have

$$F(|\psi\rangle, |\phi\rangle) = F(U|\psi\rangle, U|\phi\rangle). \tag{1.53}$$

This means that if we design a circuit U with well-defined behaviour on input $|\psi\rangle$, but we give it some approximation $|\phi\rangle$ as an input instead, the resulting state will have the same fidelity with the intended output. This means we do not gain approximation error, no matter the size of the circuit.

Fidelity plays a larger role with density matrices, where it captures both classical and quantum senses of error. We will mostly use it in an imprecise sense to refer to how close a state is to some desired state.

1.4.3 Error Complexity

A single-bit classical error operator can only be a bit flip. Single-qubit quantum errors are more complex, since they could be *any* 2×2 unitary. However, we can simplify this somewhat, noting that quantum operations are linear and the Pauli matrices span the set of 2×2 matrices. Thus, we can focus on only two errors: X errors and Z errors.

An X error is analogous to a bit flip, since its action is $|0\rangle \mapsto |1\rangle$ and $|1\rangle \mapsto |0\rangle$. A Z error is called a "phase flip", and it's action is $|0\rangle \mapsto |0\rangle$ and $|1\rangle \mapsto -|1\rangle$. This has no classical analogue. This extra dimension of errors makes error correction fundamentally more complicated; see Section 2.3.

Since the Pauli Y gate is equal to iXZ, we can treat it as a Z error followed by an X error. Since any single-qubit error can be written as a linear combination of I, X, Z, and Y, and quantum operations are linear, then if our error correction circuit can correct an X error, a Z error, and both, then it can correct any single-qubit error.

1.4.4 Error Rates

Though classical ECCs have extensive use in noisy applications like Wi-Fi or space probes, they are less necessary within the components of a computer because the components have very low rates of noise. A consumer-grade laptop can use a code that can correct 1 error per 2048 bits and reasonably expect to never see a problem in its lifetime [30].

In contrast, quantum computing technologies are plagued by noise and it appears this will not change. Multiple qubit errors are very likely in very short time spans, so we need large error correcting codes and we need to apply the error correction very quickly.

Hence, we typically talk about "physical" versus "logical" qubits in quantum computing. The physical qubits are the actual physical objects that have the desired quantum behaviour, such as a trapped ion or a superconducting loop of metal. A logical qubit is a collection of physical qubits, together with some error correction circuitry, that stores an encoding of a single qubit. We will need logical qubits to overcome noise and hence to perform any quantum computation.

Devoret and Schoelkopf [24] created a diagram showing the path to scalable quantum computing technologies that quantum scientists are pursuing today, which is included as Figure 1.4.

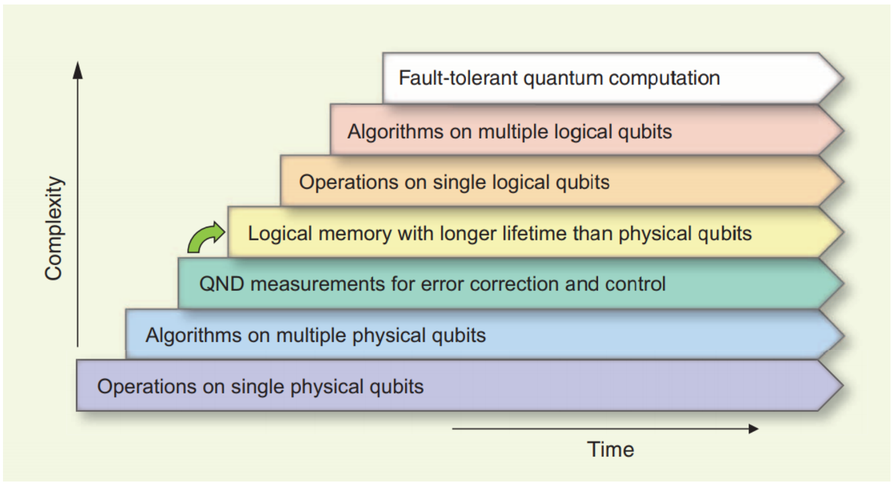

Fault-tolerant quantum computation
Algorithms on multiple logical qubits
Operations on single logical qubits
Logical memory with longer lifetime than physical qubits
QND measurements for error correction and control
Algorithms on multiple physical qubits
Operations on single physical qubits
Complexity
Time

Figure 1.4: The "staircase" of quantum computing technologies, from [24].

1.4.5 Logical Operators

Because the entire computation process is noisy, we don't want to decode a logical qubit until it must be measured. Thus, we want gates that apply to *logical* qubits, rather than just physical qubits. A logical gate will be more complex than a physical gate.

For example, consider a code which uses 5 physical qubits to represent one logical qubit, as follows:

$$\begin{aligned}|\overline{0}\rangle &= \tfrac{1}{\sqrt{8}}(\,|00000\rangle + |11100\rangle - |10011\rangle - |01111\rangle \\ &\quad + |11010\rangle + |00110\rangle + |01001\rangle + |10101\rangle) \\ |\overline{1}\rangle &= \tfrac{1}{\sqrt{8}}(\,|11111\rangle - |00011\rangle + |01100\rangle - |10000\rangle \\ &\quad - |00101\rangle + |11001\rangle + |10110\rangle - |01010\rangle).\end{aligned} \tag{1.54}$$

A logical X operator to flip these two states is

$$\overline{X} = (X \otimes X \otimes X \otimes X \otimes X)(Z \otimes Z \otimes Z \otimes Z \otimes Z), \tag{1.55}$$

and a logical Z operator is

$$\overline{Z} = (Z \otimes I \otimes Z \otimes Z \otimes I). \tag{1.56}$$

Some general methods to construct logical gates include:

- Applying physical gates to each physical qubit in the logical qubit. This is called a "transveral" gate.
- Using known commutation relations among gates to rearrange the quantum circuit to move all instances of a particular gate to the end, and then using the result to change how the measurement results are interpreted. We cannot do this for every gate, otherwise quantum circuits would be easy to classicallly simulate. For example, in a Clifford+T circuit, the X and Z operators can commute through the circuit to the end.
- Preparing physical qubits to particular configurations (so-called "magic" or "resource" states), using error-correcting codes to refine these physical qubits into a logical qubit in the resource state, then using this extra resource qubit to help perform a particular gate. This may require measuring the resource qubit and thus destroying the resource state.

An important point is that, from the code's perspective, there is no difference between a logical operator and an uncorrectable error. If the code could "correct" the logical operation and undo it, then we would be unable to change the encoded state and thus could not perform any computation. Thus, there is some balance to strike such that the controller can easily apply logical gates, but they are still unlikely to happen as a result of noise.

1.4.6 Passive versus Active Correction

Correcting a classical ECC requires reading some or all of the extra bits to decide how to correct it. This is because we need different operations to correct different errors. For QECCs we cannot always "read" extra bits since measurement is destructive.

Suppose we have some circuit D that detects which error occured. For some set of errors $\{E_i\}$, the circuit D has the action

$$D(E_i |\psi\rangle) |0\rangle = E_i |\psi\rangle |E_i\rangle \tag{1.57}$$

where $|E_i\rangle$ is some representation of the error E_i. We call this the *syndrome*. We have two main design choices to correct this error:

Passive Correction: The correction is done entirely within the "quantum realm". That is, any classical controller or output does not know what error occurred.

Active Correction: We measure the syndrome and use a classical circuit to decide which quantum error correction circuit to apply.

Passive error correction is often envisioned as a carefully-engineered Hamiltonian such that its natural time evolution will correct any noise. This approach is preferable because we would like the error correction to be a fixed cost. We would use extra resources to construct a more complicated physical qubit, but afterwards it would be more resilient to noise, without any additional effort. A classical magnetic hard drive has such a Hamiltonian (see Section 2.3).

Active correction would be more difficult in the long run, because measuring the state will heat the system and introduce more noise, and it will also require extra classical computation to correct every error. However, it may be better in practice because it moves most of the work to the classical realm, where we are already capable of high fidelity computation. This reduces the number of noisy quantum operations.

1.4.7 Syndrome Measurement

Considering Equation 1.57, we can ask what happens when we measure the syndrome during active error correction. Suppose our set of errors is the Pauli gates, and suppose we represent our error as a linear combination of Paulis:

$$E = e_I I + e_X X + e_Z Z + e_Y Y. \tag{1.58}$$

If we apply a circuit that detects Pauli errors, it will produce the state:

$$e_I \left|\psi\right\rangle \left|I\right\rangle + e_X X \left|\psi\right\rangle \left|X\right\rangle + e_Z Z \left|\psi\right\rangle \left|Z\right\rangle + e_Y Y \left|\psi\right\rangle \left|Y\right\rangle . \tag{1.59}$$

If we measure the syndrome, this will "collapse" our state into one of the Pauli errors. For example, with probability $|e_X|^2$, we will measure X and the resulting state will be $X\left|\psi\right\rangle$. From there, we can apply the X correction circuit.

More importantly, for "small" errors, the largest component will be e_I. With probability $|e_I|^2$, we will measure I — "no errors" — and the resulting state will just be $\left|\psi\right\rangle$. In summary: Just measuring the syndrome will, with some probability, correct the error.

This fact is important for the function of surface codes, in Section 2.1.

Chapter 2

Quantum Computational Models

Only a rash person would declare that there will be no useful quantum computers by the year 2050, but only a rash person would predict that there will be.

-N. David Mermin, *Quantum Computer Science: An introduction* [45]

To motivate this chapter, consider that Turing machines, Lambda calculus, and the WORD RAM model are all equivalent abstractions of computers. Yet of these, the WORD RAM model is the most realistic and useful, not for any fundamental reason, but because computers engineers happened to find a way to build large and cheap random-access memory.

This is somewhat surprising: Consider the architecture of a 2-dimensional DRAM chip. The bits are stored in a rectangular grid, and to access or write a bit in location (i, j), a current is sent along a wiring running at horizontal position i, and another at vertical position j, and they will only affect the memory at the point of intersection.

To select the ith wire out of n wires requires at least $\log n$ binary decisions about where to send the current, which implies that there are n "gates" of some kind in the RAM chip. If there are n wires on each side, there are n^2 bits of memory; thus, the total number of gates for N bits of memory scales as $O(\sqrt{N})$. This seems expensive, but miraculously, we can build these gates cheaply. Even more surprising, these gates are substantially cheaper than the gates inside a CPU.

For quantum computers, there is also a zoo of equivalent computational models: Circuit models, distributed quantum computing, quantum Turing machines, measurement-based computers, and others. At this point, it seems premature to choose one model. Yet, for cryptanalysis, we *need* to choose a model. If we make a statement like "Grover's algorithm solves brute-force search with cost $O(\sqrt{N})$", we are implicitly using *some* model of computation. Hence, we would like to make the best guess we can.

With that in mind, in this chapter we will start with a motivating example of surface codes (Section 2.1). These show clearly how quantum gates are *processes*, not physical objects, as well as highlighting the large overhead for error correction.

In Section 2.2 we give the *peripheral model* of quantum computation. In this model, we treat the quantum computer as a peripheral for a classical computer, in principle no different than a monitor or printer. This emphasizes the costs of classical control, which are asymptotically the dominant cost of quantum computing.

The peripheral model is very general and can accomodate a range of features. Section 2.3 gives several examples. We choose five specific models in Section 2.4, which we use throughout this thesis. Two of the models come from previous work, while the other three are first attempts at more realistic model of quantum computing.

2.1 Surface Codes

Surface codes are a particular type of *topological* error-correcting codes. This section describes some basic features of surface codes to give a sense of what a quantum computer might be like and to motivate some of the assumptions of the models that will follow. Fowler et al. [25] provide at excellent survey of surface codes. They use qubits on the vertices of a large 2-dimensional grid, though other lattices would work. Half the qubits are designated "measurement" qubits and the other half are "data" qubits, arranged in a checkerboard pattern.

Active Correction: To correct errors, we repeat a "measurement cycle", where the measurement qubits are entangled with the data qubits and then measured. This has the effect of syndrome measurement as in Section 1.4.7, so it will either correct errors or force X, Y, or Z errors in the data qubits. If an X error occurs on a data qubit, we detect it because it flips the measurement results of two adjacent measurement qubits.

If two X errors occur next to the same measurement qubit, the errors will "cancel out" and be undetectable by that qubit. To be completely undetectable we would need an X error on every qubit from one edge of the code to another — hence, the probability of having so many errors decreases exponentially with the size of the qubit grid. The same logic applies to the other Pauli errors, and by linearity to all single-qubit errors.

Physical Overhead: The error correction keeps the system in a fixed logical state, but we must be able to deliberately change the state to do any computation. To do this, we turn off small sections of qubits to form "holes". A series of errors from the edge of one hole to the edge of another hole would be undetectable, for the same reason that errors from one edge of the grid to the other are undetectable. Such a string of "errors" acts as a logical operator. Each pair of holes induces such an operator, and hence acts as a single logical qubit.

We need the holes to be sufficiently far from each other, and from other pairs of holes, that these logical operators do not occur from noise. The number of qubits between two holes is the *distance* of the code.

We can see from this construction that the probability of error in each logical qubit in each time step decreases exponentially wtih the distance, and the number of physical qubits for every logical qubit increases quadratically with the distance. If we have a computation on Q logical qubits for depth D, we need to suppress the per-qubit-per-time-step error below $O(\mathsf{QD})$. This means the physical:logical qubit ratio is $\Theta(\log^2(\mathsf{QD}))$. This is optimal for a family of codes called "stabilizer codes" [15].

Computational Overhead: In every measurement cycle, a classical controller must entangle and measure half of all the physical qubits. It needs to either store that data or process it immediately. This implies at least $\Omega(\mathsf{Q}\log^2(\mathsf{QD}))$ operations in every time step. Further, the control hardware must solve a perfect matching problem in every time step for every logical qubit. This control can be done in $O(1)$ time with $\Omega(1)$ parallel processors per physical qubit [26].

Stationary Qubits: Gates are applied "on top" of a surface code, mostly by changing the pattern of measurements. Different measurement patterns can move logical qubits without moving any of the physical qubits. Hence, we cannot pay a fixed cost to build a gate, then use it many times for free; we need to pay the same computational cost every time we use a gate.

The control cost ends up the same for doing nothing to the qubit as it is for performing a gate. This means the identity "gate" has approximately the same cost as any other Clifford gate.

It happens that T gates are unusually expensive for a surface code. These are a significant component of the total computation, but are asymptotically the same cost as the rest of the circuit.

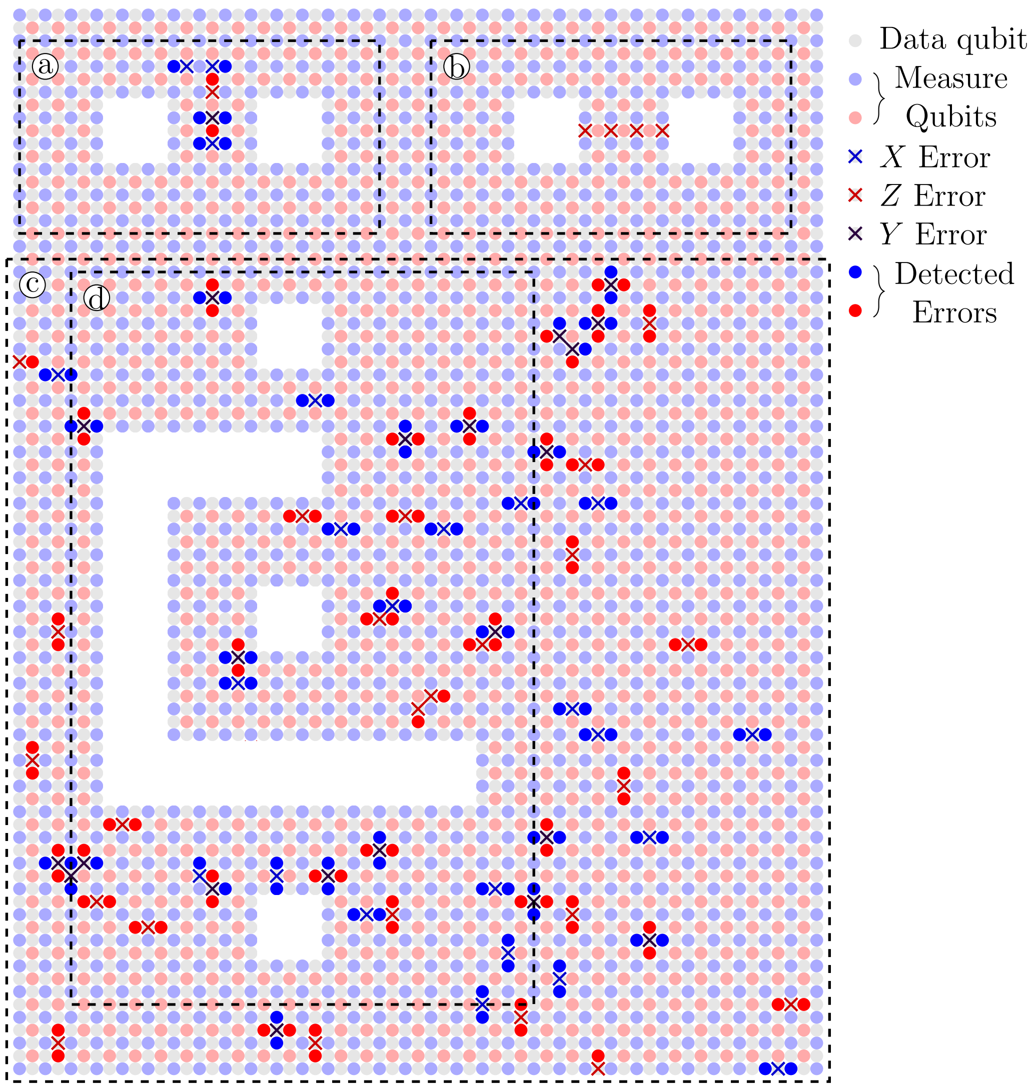


Figure 2.1: An illustration of a surface code of distance 3. a: A single logical qubit with a detectable set of errors. b: A logical qubit with an undetectable (logical) error. c: A patch showing errors occuring at a rate of 2%. d: Two qubits showing part of the process of a `CNOT` gate.

2.2 Peripheral Models

2.2.1 Description

Jaques and Schanck [34] introduced the peripheral model of quantum computation. Our goal was to formalize a model of quantum computation where a classical controller must perform computational work to manage a quantum computer, and to emphasize the costs of this classical computation. This section summarizes the definitions in our paper.

The Hamiltonian of a quantum state governs its evolution through time. For the quantum state to perform some computation, either some external "force" acts on it, or it evolves naturally by its Hamiltonian. The latter is known as "ballistic" computation. The peripheral model captures both approaches: Either the controller queues and applies transformations, or it performs some actions to change the Hamiltonian. In both cases the controller performs significant computations.

The first part of the model is the *memory peripheral*, which captures the physical state of the quantum computer at a specific point in time.

Definition 2.2.1 (Definition 1, [34]). *A* memory peripheral *is a tuple* $\mathsf{A} = (\mathcal{H}^{\mathsf{A}}, H^{\mathsf{A}}_{sys})$*, where* $\mathcal{H}^{\mathsf{A}}$ *is a Hilbert space and* H^{A}_{sys} *is a Hamiltonian acting on* $\mathcal{H}^{\mathsf{A}}$*, referred to as the system Hamiltonian.*

Here the "state" of the computer just describes which qubits are active, where the qubits are, and what Hamiltonians are acting on them, but this is distinct from the quantum "state" of the computer, which would be a vector in $\mathcal{H}^{\mathsf{A}}$. Hence, the state of the peripheral governs what quantum states it can possibly hold.

The controller can apply *memory operations* to the peripheral.

Definition 2.2.2 (Definition 2, [34]). *A* memory operation *is a morphism between peripherals,* $(\mathcal{H}^{\mathsf{A}}, H^{\mathsf{A}}_{sys}) \to (\mathcal{H}^{\mathsf{B}}, H^{\mathsf{B}}_{sys})$*, where the map from* $\mathcal{H}^{\mathsf{A}}$ *to* $\mathcal{H}^{\mathsf{B}}$ *is a quantum channel.*

On this, we define a parallel composition of memory peripherals, denoted $\mathsf{A} \otimes \mathsf{B}$. If $\mathsf{A} = (\mathcal{H}^{\mathsf{A}}, H^{\mathsf{A}}_{sys})$ and $\mathsf{B} = (\mathcal{H}^{\mathsf{B}}, H^{\mathsf{B}}_{sys})$, then

$$\mathcal{H}^{\mathsf{A}\otimes\mathsf{B}} = \mathcal{H}^{\mathsf{A}} \otimes \mathcal{H}^{\mathsf{B}} \tag{2.1}$$

$$H^{\mathsf{A}\otimes\mathsf{B}}_{sys} = H^{\mathsf{A}}_{sys} \otimes I^{\mathsf{B}} + I^{\mathsf{A}} \otimes H^{\mathsf{B}}_{sys}. \tag{2.2}$$

We can extend morphisms to parallel composition: Given $f : \mathsf{A} \to \mathsf{C}$ and $g : \mathsf{B} \to \mathsf{D}$, then we can define a morphism $f \otimes g : \mathsf{A} \otimes \mathsf{B} \to \mathsf{C} \otimes \mathsf{D}$, with the natural action.

There are many possible morphisms we can define, but only some of them may be physically realizable, such as Clifford+T gates. This leads us to the full definition of a memory peripheral *model*:

Definition 2.2.3 (Definition 3, [34]). *A* memory peripheral model *is a tuple* $(\mathcal{C}, \circ, \otimes, 1)$ *where:*

- $\mathcal{C}$ *is a collection of memory peripherals.*
- *The morphisms between the objects of* $\mathcal{C}$ *are memory operations.*
- $\circ$ *represents sequential composition of morphisms.*
- $\otimes$ *represents parallel composition of objects and morphisms.*
- *There is a "void" memory peripheral* 1 *such that* $\mathsf{A} \otimes 1 = 1 \otimes \mathsf{A} = \mathsf{A}$ *for all* A *in* $\mathcal{C}$*.*

This definition naturally leads to the idea of *irreducible* memory peripherals and morphisms. A memory peripheral A is irreducible if it cannot be written as a parallel composition of two non-void memory peripherals. A similar definition applies to morphisms.

In many cases we can describe a memory peripheral model by a finite set of irreducible memory peripherals and morphisms. For example, a qubit memory with Clifford + T gates can be described this way: There is one irreducible memory peripheral, $\mathsf{Q} = (\mathbb{C}^2, 0)$. A trivial Hamiltonian means it undergoes no time evolution. The morphisms are Clifford+T gates applied to single qubits or pairs of qubits. These can be composed in sequence and parallel to produce an infinite set of memory peripherals and morphisms.

In the qubit memory and Clifford+T gate example, the set of irreducible morphisms acted on at most 2 irreducible memory peripherals. Hence, we call it a 2-ary memory peripheral model. In general, we say a memory peripheral model is k-ary if every irreducible morphism acts on a composition of at most k irreducible memory peripherals.

For measuring an algorithm on a memory peripheral, we refer to the *depth* and *width*. The width is the total number of irreducible memory peripherals that form a specific memory peripheral. In most cases this is simply the number of active qubits. The depth is the total number of sequential memory operations.

"Depth" will also refer to the total run-time in cases where different operations take different times. This is technically incorrect, but it is a reasonable equivocation because:

- some gates with a large run-time are actually abstractions of a sequence of smaller gates;

- the focus is opportunity cost, so the run-time will reflect the depth of some circuit that *could* be run.

2.2.2 Control Costs

In a memory peripheral model, a computation is a series of morphisms on an initial peripheral. We assume that there is a classical computer that executes these morphisms somehow.

The simplest and most natural model for classical control is a word-RAM model. This attempts to model a typical classical computer, with a CPU that performs basic arithmetic on elements of a larger random access memory.

The following definition is based on Homer and Selman [29], though with a more general set of functions and relations.

Definition 2.2.4. *A Word-RAM computer consists of:*

- *an array A of n words of ω bits, where $2^\omega \geq n$;*
- *a finite set $\mathcal{F}$ of functions $f : \{\mathtt{0},\mathtt{1}\}^{k\omega} \to \{\mathtt{0},\mathtt{1}\}^\omega$, each with some arity $k \in \mathbb{N}$;*
- *a finite set $\mathcal{R}$ of relations $r : \{\mathtt{0},\mathtt{1}\}^{k\omega} \to \{\mathtt{0},\mathtt{1}\}$, each with some arity $k \in \mathbb{N}$.*

A RAM program *for this computer is a finite sequence $(o_1, \cdots, o_N)$ of RAM* operations. *The computer sequentially executes the operations, which have one of the following forms:*

- *A function $f \in \mathcal{F}$ and $k+1$ addresses $a_1, \cdots, a_{k+1}$. The operation computes $f(A[a_1], \cdots, A[a_k])$ and writes the result to $A[a_{k+1}]$.*
- *A relation $r \in \mathcal{R}$, k addresses $a_1, \cdots, a_k$, and an integer $i \in [N]$. The operation computes $r(A[a_1], \cdots, A[a_k])$, and if the result is $\mathtt{1}$, skips to operation o_i, and otherwise continues as normal.*
- *A `WAIT` operation that does no computation.*

Every operation takes unit time and unit cost, except `WAIT`, which has unit time but no cost.

Typical choices for functions are bitwise `AND`, `OR`, `XOR`, right- and left-shifts, addition, and subtraction. Typical choices for relations are equality, $\leq$, and $\geq$.

We further consider a *parallel* word-RAM model, where we have P distinct word-RAM computers ("processors") with access to a *shared* memory. They each execute one operation at every time step. If two processors attempt to access the same memory location in the same step, there is some method that grants access to only one processor.

The memory peripheral model adds special operations to the usual word-RAM model. For each k-ary memory peripheral A, the controller has an *instruction set*, which is a list of the possible irreducible morphisms. We add an `APPLY` operation, which takes $k+1$ arguments: k different memory peripheral addresses, and one argument for the morphism. The controller executes this operation, which queues that particular morphism. For morphisms on separate systems in the memory peripheral, the controller is allowed to schedule parallel operations. For example, if it schedules a `CNOT` gate between qubits 1 and 2, it could also schedule a `CNOT` gate between qubits 3 and 4.

The other instruction is "`STEP`". This starts the physical process that will apply all of the scheduled morphisms to the memory peripheral.

2.2.3 Advantages

The peripheral model has three main advantages over the circuit model:

Highlights physical assumptions:

By including the Hamiltonian in the model, we bring the physical assumptions to the foreground. The simplest and least realistic assumption is that the Hamiltonian for every memory peripheral is 0, in which case we recover the circuit model. Including a Hamiltonian of 0 is a reminder that such a model is a very idealistic abstraction of the physical system.

With other Hamiltonians, we can capture noise terms. Since a quantum channel can represent any noise terms, by the Stinespring Dilation Theorem we can represent these as interactions with an external system. Hence, the Hamiltonian can produce the noise terms.

From a noisy Hamiltonian, we can either add another passively error-correcting Hamiltonian or add some active error correction. In the former case, we are again making an explicit physical assumption. In the latter case, we could group the active error correction memory peripherals together into a larger peripheral which behaves like a single logical qubit. In doing so, we would need to include all the operations we used for the active error correction into the costs for the basic operations on the logical qubit.

Captures ballistic computations:

If we have a non-trivial Hamiltonian then it is time–independent for each memory peripheral. Thus, the quantum state evolves according to the Schrodinger equation as:

$$i\hbar\frac{\partial}{\partial t}\left|\psi(t)\right\rangle = H^{\mathsf{A}}_{sys}\left|\psi(t)\right\rangle \tag{2.3}$$

which has a solution of

$$\left|\psi(t)\right\rangle = e^{iH^{\mathsf{A}}_{sys}t/\hbar}\left|\psi(0)\right\rangle . \tag{2.4}$$

Here the exponent is the usual matrix exponent: $e^{iH^{\mathsf{A}}_{sys}t/\hbar} = \sum_{n=0}^{\infty}\frac{(iH^{\mathsf{A}}_{sys}t/\hbar)^n}{n!}$, which is a unitary. We assume that there is necessarily some small time interval δ_0 between memory operations, so that the operator $e^{iH^{\mathsf{A}}_{sys}\delta_0/\hbar}$ acts on the state between each memory operation.

While the circuit model focuses on time-dependent Hamiltonians created by interactions with an external system, such as shooting a laser at a qubit to alter the state, this formalism allows us to model *ballistic* computations. The name is meant to evoke a computer where computation is done by billiard balls bouncing around inside of it. We perform substantial work to set up the state of the computer, then we drop a ball in and let it evolve on its own. Similarly, a ballistic quantum computation would involve carefully creating a precise Hamiltonian, then simply letting the system evolve by the action of that Hamiltonian.

One can easily show (since the Hamiltonian evolution is unitary) that ballistic computation is robust against small errors in the initial *state*, but the effect of noise in the initial *Hamiltonian* is under-explored. This thesis will mostly ignore ballistic computation, since there are few serious proposals for ballistic quantum computation.

Permits direct comparison of classical costs:

In the circuit model, we can give a cost in gates or depth or number of qubits, but it's difficult to compare to classical algorithms. We need some sort of common unit between the two. In a practical setting, this will probably be some currency: is it cheaper to run the algorithm on the quantum or classical computer? But at this point, it's nearly impossible to guess at the costs of quantum computing.

The memory peripheral model has a fundamental assumption that any quantum technology will rely on a classical controller. We then ignore all of the costs to build and maintain the quantum aspects, and focus solely on the work that the classical controller does. This turns the cost into an opportunity cost: What else could we do with the classical controller if it were not busy running our quantum computation?

This is necessarily an underestimate of quantum computation costs, but probably not by much. More importantly, it gives a common unit for the costs of classical and quantum algorithms. This allows us to compare them and to give reasonable analyses of algorithms with a large mix of quantum and classical computation.

2.3 Architectural Features

This section gives various features that could be included in a computational model. We will not use all of them, and some of them simply demonstrate different ways to think about quantum computing and how one might account for extra costs in a memory peripheral model.

2.3.1 Error Correction

Section 1.4.6 introduced passive and active error correction, and here we will follow previous work [34] and suggest that passive error correction is unlikely.

For classical computers, passive error correction is possible in 2 physical dimensions. A magnetic hard disk demonstrates this and Figure 2.2 shows the principle. The system's energy is lowest when all the magnetism lines up, and if one component gets flipped, it naturally "falls" back down to the lowest energy configuration. To flip the "logical" bit means flipping all the bits, which is only energetically neutral if a full row of magnetization is flipped. The probability of such a large error, under an independent, random noise model, decreases exponentially with the lattice size [48].

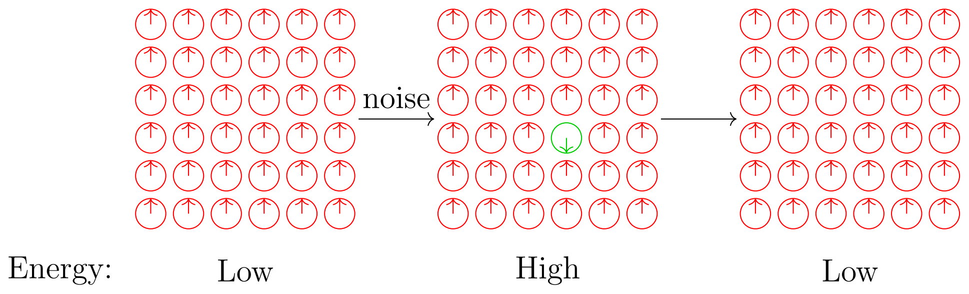


Figure 2.2: An idealization of a lattice of magnets that passively corrects small errors.

For a quantum code, this is more difficult. Bravyi and Terhal prove that for a lattice of qubits with a stabilizer code, two-dimensional quantum memory cannot passively correct itself [15]. There is an explicit construction of six-dimensional self-correcting memory [23], and without a universal gate set, it can be done in four dimensions [4]. The three dimensional case is open.

In practice we can build, at most, a three-dimensional memory. Cooling and control issues make a two-dimensional memory much more likely. For three-dimensional memory, we may be able to "stack" layers of two-dimensional memory, but for this to scale in the same way as truly three-dimensional memory will require difficult engineering.

Thus, a passively-corrected memory is not just an assumption about future engineering capabilities, but an assumption of fundamental physical results. It may be proven impossible in the future.

In a memory peripheral model, passive correction is contained in the Hamiltonian, and hence it induces no extra costs for the classical controller.

We can view active correction as four distinct steps to apply a single memory operation f, shown in Algorithm 1. This is what a surface code does. We suppose that we wish to apply f to the qubits in addresses $R_{i_1}, \cdots, R_{i_k}$, and the error correction will use ancillae at addresses $R_{j_1}, \cdots, R_{j_m}$.

Algorithm 1 Active error correction in a memory peripheral.

```
APPLY $f$ to $R_{i_1} \cdots R_{i_k}$
STEP
for $\ell = 1$ to $m$ do
  APPLY measure to $R_{j_\ell}$ and save to the result to $S_{j_\ell}$
end for
STEP
CORRECT($S_{j_1}, \cdots, S_{j_\ell}, R_{i_1}, \ldots, R_{i_k}, R_{j_1}, \cdots, R_{j_m}$)
```

The `CORRECT` operation in Algorithm 1 takes the results of the measurements of the ancillae, the addresses of the data qubits and the ancillae and computes the necessary steps to correct the errors. This may involve several `SCHEDULE` operations to queue memory operations that will physically correct the errors, followed by a `STEP` operation. Alternatively, like a surface code, it may simply use the control hardware to account for the errors.

In Lines 2 and 6, there will be some time intervals δ_1 and δ_2 when the system will undergo some evolution, which may include errors. Hence, even if the morphism f is just the identity, the error correction is still necessary, just like the surface code.

To add these correction costs in a memory peripheral model, we need to decide

- what kind of errors occur, and how frequently, and
- the necessary steps to correct them.

Though this should technically be described with Hamiltonians and explicit gates, it is easier and just as thorough to take an average error correction cost for each gate, and then use that for the cost of the gate.

For example, in a surface code of distance d a Hadamard gate requires $\Theta(d)$ sequential operations. Correcting a single qubit requires $\Theta(\log^2(\mathsf{DQ}))$ operations on average, so any "idle" qubits will also incur this cost for their error correction while the Hadamard gate is applied.

2.3.2 Computational Qubits

Are we able to apply any gate we want to any qubit we want, and apply as many as we can at the same time? In the surface code, this is true for all Clifford gates. Yet this is distinctly untrue for classical computers, and explicitly modelled in a Word-RAM model, which can only apply arbitrary operations to CPU memory, and must work hard to copy memory between different cache levels to perform well.

Many quantum computing papers explicitly make this assumption (e.g., [25, 7]), and it is implicit in others. Others ([16]) consider things like a "QPU" (quantum processing unit) and describe ways to teleport data in and out of the QPU for computations.

In a memory peripheral model, we could make two different types of irreducible peripheral, a computational qubit Q_C and a storage qubit Q_S. The model would contain only some operations on Q_S and would have some operation to swap the state between the two types.

2.3.3 QRAM

Suppose we have a "quantum memory", consisting of a large array of qubits $|x_1\rangle \cdots |x_N\rangle$. We want random access, meaning we can take a state $|i\rangle$ as input and have some operation such that

$$|i\rangle |y\rangle |x_1\rangle \cdots |x_N\rangle \mapsto |i\rangle |x_i\rangle |x_1\rangle \cdots |x_{i-1}\rangle |y\rangle |x_{i+1}\rangle \cdots |x_N\rangle , \tag{2.5}$$

analogous to classical RAM. Adding this circuit to a gate set is equivalent to adding the following two circuits:

$$|i\rangle |y\rangle |x_1\rangle \cdots |x_N\rangle \mapsto |i\rangle |y\rangle |x_1\rangle \cdots |x_{i-1}\rangle |x_i \oplus y\rangle |x_{i+1}\rangle \cdots |x_N\rangle \tag{2.6}$$

$$|i\rangle\,|y \oplus x_i\rangle\,|x_1\rangle \cdots |x_N\rangle \mapsto |i\rangle\,|y\rangle\,|x_1\rangle \cdots |x_{i-1}\rangle\,|x_i\rangle\,|x_{i+1}\rangle \cdots |x_N\rangle\,. \tag{2.7}$$

Classical random access can be done with a classical controller, which can decide which gates to apply. For quantum data, the classical controller cannot read the input $|i\rangle$ to decide which gates to apply without also destroying the quantum state. Thus, the classical controller must apply all gates that may be necessary for *any* input state, and use gates like `TOFFOLI` or `CNOT` to implement any conditional logic.

The controller will need to apply some sort of operation to each qubit $|x_j\rangle$ in memory. In a k-ary memory peripheral model, a single operation can only apply to k objects at once, so the total cost of the operation in Equation 2.5 will grow linearly with N, since k does not scale. In a circuit model, the same result holds. The proof is trivial but for completeness it will follow.

Theorem 2.3.1. *In a circuit model with a finite set of gates of fanin less than some constant, a random access circuit to an N-bit array requires $\Omega(N)$ gates.*

Proof. Let U be the random access circuit as a unitary. Considering its action in Equation 2.5 we can see that, for every bit in memory, there is some input such that U alters that bit. Thus, U cannot be written as $U = U' \otimes I_2^{\otimes m}$ for $m \geq 1$. By Lemma 1.1.1, U requires at least $N/k = \Omega(N)$ gates. $\square$

There is a slight trick here: By using the circuit model, we require U to be built entirely out of gates, and in between applying gates, the quantum state does not change. It has not been proven impossible for memory access to be fast, cheap, and ballistic, where some initial set-up would cause the quantum Hamiltonian to do the hard work of moving memory. It remains a topic for future research to decide if there are fundamental requirements on the precision of the set-up of such a Hamiltonian that force ballistic memory to have linear cost as well.

Ballistic memory access resembles the "bucket-brigade" QRAM proposal from Giovannetti, Lloyd, and Maccone (GLM)[27], which imagines photons travelling through a series of gates. The controller may be able to send a single photon into the system and have it propagate without external intervention.

Bucket-brigade QRAM faces some issues. It is mostly interested in memory access in small superpositions, while our interest is a more general QRAM. Arunachalam et al. [6] show that error rates for the gates involved must decrease linearly with the memory size for a quantum search involving the QRAM. It remains an interesting challenge to model both the computational cost and the error rate of the gates in bucket-brigade QRAM.

Alternatively, we could imagine a future situation where we simply have separate sections of hardware that are dedicated specifically to memory access, analogous to classical RAM. Even though we require $O(N)$ gates for memory access, perhaps these can be assembled and controlled by hardware that is much less expensive than the rest of the quantum computer. Our primary cost in a memory peripheral model is the opportunity cost of the control hardware, so here we assume that the QRAM hardware, because it's dedicated to this one task, would be incapable of performing any other computations.

With passive correction this may be reasonable, but not with active correction, where we would need substantial computation just to store memory.

Cheap QRAM is unlikely. Still, we can include it in a peripheral memory model in several ways:

- We could assume that we have some random access operation that is either not k-ary or ballistic, requiring only $O(1)$ control operations to execute. This also models QRAM built from $O(N)$ gates that are controlled by specialized hardware.
- We could give a cost of $O(N)$ for the random access gate, using that gate as shorthand for $O(N)$ k-ary memory operations that must be performed to execute the gate. Section 4.1 gives explicit circuits for this.
- We could ignore random access and build alternative memory access circuits for different use cases, as we did in Jaques and Schanck [34] and which Section 4.2.2 will describe.

2.3.4 Locality

In the surface code, both logical `CNOT` and T gates require the qubits to be physically near to each other. Moreover, the physical `CNOT` gates only affect physically adjacent qubits. In this way, operations on the surface code are *local*.

This sense of locality is different than the idea of a "k-local Hamiltonian". A k-local Hamiltonian is a Hamiltonian $H = \sum_{i=1}^{n} H_i$ where each H_i acts only on k different systems, but there are no restrictions on the physical proximity of those systems.

If the gates of a quantum computer are local, then we need some way to physically move qubits. Some papers refer to "flying qubits", which are physically moved around. Other options are quantum teleportation or movements like the surface code. All require intervention by the controller.

A memory peripheral model does not naturally have a spatial layout, but we can account for it by restricting the set of memory options to those that act only on physically close objects. From there, to give the precise costs of a computation, we would need to describe the physical layout of the architecture. For many algorithms and analyses, such precision is unnecessary. Instead we could give approximations. For example, if there are N objects in a memory peripheral model of a 2-dimensional architecture, a 2-ary memory operation could have a cost of $O(\sqrt{N})$, which would be the average cost to move the qubits near to each other.

2.3.5 Latency

Latency is subtly distinct from locality. Here we do not give restrictions on which parts of the computer can interact, but we account for the time it takes for the signals to propagate between them. For example, with quantum teleportation we may be able to move any qubit to any other part of the computer in a constant number of operations, but we still need both locations to have one qubit out of a pair of entangled qubits. Sending the entangled qubits may take some time.

Including latency costs switches the unit of cost from bit operations to time, and hence the RAM model is no longer appropriate. We could simply give a constant time cost for each sequential RAM operation, but then we are accounting for latency in the quantum computer and not its classical controller. For consistency, we should use another model for the classical controller that accounts for latency.

For a formal model, Candidate Type Architecture accounts for latency, but focuses on a subtly different application, where we have many parallel machines that have high-latency communication costs between them. Bernstein approaches similar questions to this thesis [8] and simply states, absent a formal model, that "random access to an N-element array takes time $N^{1/2}$".

We will add some formality and define a model of "RAM with latency", where the cost is in units of processor-hours. Basic arithmetic operations have cost 1, and random access to an N-element array has cost $N^{1/d}$ for d-dimensional memory. The idea is that if a RAM machine must make a lengthy query to a large memory array, it's wasting time that it could use to do some other computation.

A rigorous version of this model is a task for future work.

2.4 Models

Here we give a list of the models we will use, and the cost metrics that are most natural.

2.4.1 Passively-Corrected Clifford+T Circuit

In this model, taken from Jaques and Schanck [34], we have a unique irreducible memory peripheral $\hat{\mathsf{Q}} = (C^2, 0)$, which is a single qubit with a trivial Hamiltonian, so it undergoes no time evolution. The irreducible memory operations are the Clifford+T gate set.

The physical assumption is that we have somehow built logical qubits that passively correct themselves.

This model ignores locality and latency and assume uniform gate costs. This means the cost metric is just the gate count. We refer to this model as "**Passive Circuit**".

2.4.2 Actively-Corrected Clifford+T Circuit

This is the second model from Jaques and Schanck [34]. It's the same as the previous model, but with active correction, so that the identity gate has $O(1)$ RAM operation cost.

This makes the cost metric equal to the depth of an algorithm times its "width", measured in logical qubits. We may need to cost different stages of an algorithm separately, if the width changes drastically. We refer to this model as "**Active Circuit**".

2.4.3 Passively-Corrected Circuit with Latency

This is the same as the passively corrected model, but accounts for latency. A full treatment of latency would require specifying a physical layout, but instead we will simply assume that, unless otherwise specified, a gate between two qubits in a memory of width N takes time $O(N^{1/d})$, where $d > 1$ is the dimension of the architecture.

To equate time with the classical control, we will assume that the control is formed of parallel processors that would be capable of some computation during the time that a high-width gate operates. In some sense this contradicts the spirit of the passively-corrected model, since we are not giving a cost to any idle time of the classical control. To justify this cost, we assume that applying a gate that takes time T requires $O(T)$ RAM operations, meaning that controlling a long gate requires more computation.

This gives a total cost metric that is proportional to "gate-time": the sum of the time taken by all gates used in the circuit. This means that using many gates simultaneously is still expensive, even if they are used for a very short amount of time.

We refer to this model as "**Passive Latency**".

2.4.4 Actively-Corrected Circuit with Locality

This is the actively-corrected analog of the previous model. Again, if we don't specify a particular physical layout for a circuit, we will simply assume that gates require $O(N^{1/d})$ time to execute when used on memory of size N in dimension d. This could be signal propagation time or physical rearrangement. We assume the architecture is similar to a surface code, where the qubits require error correction at regular, frequent intervals. Thus, the $O(N^{1/d})$ time translates to $O(N^{1/d})$ error correction operations on each qubit.

For a cost metric, we assume something that scales similarly to a surface code. Using Bravyi and Terhal's result [15] and the same property of surface codes, we assume that for a circuit of Q qubits active for D time steps, each logical qubit requires $O(\log^d(\mathsf{DQ}))$ physical qubits, each of which requires, on average, one RAM operation per time step. Thus, once the full time D of the circuit is computed, including latency, the cost becomes $O(\mathsf{DQ}\log^d(\mathsf{DQ}))$.

We refer to this model as "**Active Local**".

2.4.5 QRAM with Latency

Here we assume passively-corrected "computational" qubits, and a distinct quantum memory. We assume access to N-bit quantum memory in dimension d is a single gate that takes time $O(N^{1/d})$. There are two natural cost metrics here: The gate count and the time. The final cost will be the maximum of the two.

We will refer to this model as "**QRAM**".

Chapter 3

Quantum Walks

Quantum walks are a major family of quantum algorithms that perform searches by repeatedly taking random steps, in superposition, on a graph or Markov process. In most applications, they give exponential costs with smaller exponents than the corresponding classical algorithms, though many of these results are only *query* complexity.

Section 3.1 explains Grover's algorithm, which is the simplest unstructured quantum search algorithm. Quantum random walks are an extension of Grover's algorithm and share many properties.

To show how random walks can be used as a search algorithm, Section 3.2 explains a classical search by random walk. The relationship between a classical brute force search and a random walk is almost the same as the relationship between Grover's algorithm and a quantum random walk.

Section 3.3 gives two approaches to random walks, Szegedy's algorithm (Section 3.3.1) and the Magniez, Nayak, Roland, and Santha (MNRS) algorithm (Section 3.3.2).

These algorithms can only improve on a naive Grover search when the cost to take a random step in a graph is substantially less than the cost to sample a vertex completely at random.

3.1 Grover's Algorithm

Grover's algorithm solves a very generic search problem. There is some small subset S of the set of n-bit strings. We have a reversible circuit U_S which tests membership; its action is:

$$U_S\left|\mathtt{b}\right\rangle\left|\mathtt{c}\right\rangle=\begin{cases}\left|\mathtt{b}\right\rangle\left|\mathtt{c}\right\rangle & ,\mathtt{b}\notin S\\ \left|\mathtt{b}\right\rangle\left|\mathtt{c}\oplus\mathtt{1}\right\rangle & ,\mathtt{b}\in S\end{cases}. \tag{3.1}$$

where $\mathtt{c}$ is another bit. We want a circuit to output some $\mathtt{b}$ from the set S.

Grover's algorithm uses U_S and another operator U_0 that flips a bit if $\mathtt{b}$ is not the all-zeros string:

$$U_0\left|\mathtt{b}\right\rangle\left|\mathtt{c}\right\rangle=\begin{cases}\left|\mathtt{b}\right\rangle\left|\mathtt{c}\right\rangle & ,\mathtt{b}=\mathtt{0}^n\\ \left|\mathtt{b}\right\rangle\left|\mathtt{c}\oplus\mathtt{1}\right\rangle & ,\mathtt{b}\neq\mathtt{0}^n\end{cases}. \tag{3.2}$$

Actually, Grover's algorithm needs reflections instead, but these are easy to construct using a single ancilla initialized to $\left|-\right\rangle:=\frac{1}{\sqrt{2}}(\left|\mathtt{0}\right\rangle-\left|\mathtt{1}\right\rangle)=H\left|\mathtt{1}\right\rangle$. First note that if $\mathtt{b}\notin S$, then $U_S\left|\mathtt{b}\right\rangle\left|-\right\rangle=\left|\mathtt{b}\right\rangle\left|-\right\rangle$. If $\mathtt{b}\in S$, then:

$$U_S\left|\mathtt{b}\right\rangle\left|-\right\rangle=\left|b\right\rangle\frac{1}{\sqrt{2}}(\left|0\oplus 1\right\rangle-\left|1\oplus 1\right\rangle) \tag{3.3}$$

$$=\frac{1}{\sqrt{2}}\left|\mathtt{b}\right\rangle(\left|\mathtt{1}\right\rangle-\left|\mathtt{0}\right\rangle) \tag{3.4}$$

$$=-\left|\mathtt{b}\right\rangle\left|-\right\rangle. \tag{3.5}$$

Thus, the full action is:

$$U_S\left|\mathtt{b}\right\rangle\left|-\right\rangle=\begin{cases}-\left|\mathtt{b}\right\rangle\left|-\right\rangle & ,b\in S\\ \left|\mathtt{b}\right\rangle\left|-\right\rangle & ,b\notin S\end{cases}. \tag{3.6}$$

Since the ancilla $\left|-\right\rangle$ is unchanged, there is no unwanted entanglement. We use a similar circuit to turn U_0 into a reflection.

With these two operators, Figure 3.1 shows a circuit for Grover's algorithm.

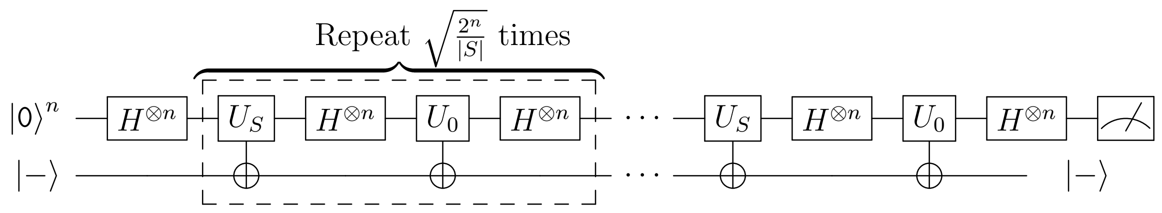


Figure 3.1: Grover's algorithm. The dashed line circles a single "step", which we will call a *Grover step*.

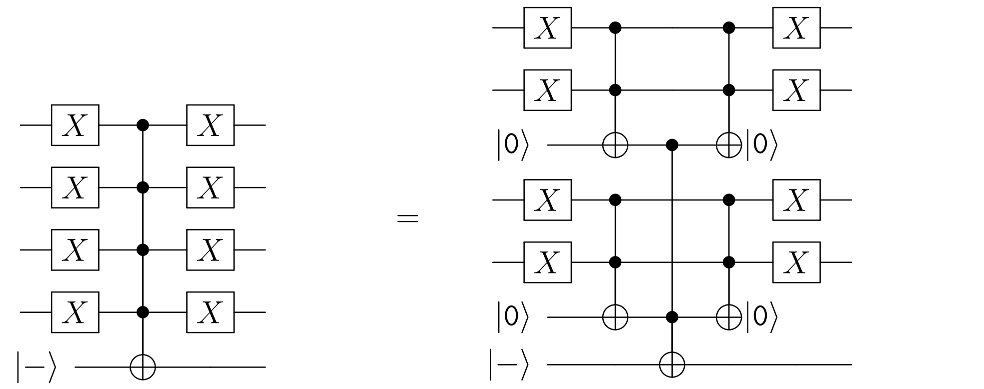
The rest of this section gives two intuitive explanations. Both rely on viewing Grover's algorithm as a series of 2 reflections, with analysis due to Boyer, Brassard, Høyer, and

Tapp [13]. The operation U_S is clearly a reflection over the subspace spanned by $|\mathtt{s}\rangle$ for $\mathtt{s} \in S$.

The circuit U_0 reflects over the uniform superposition. This would be difficult to detect directly, so we use Hadamard gates to transform the entire space. This maps the uniform superposition to the state $|\mathtt{0}\rangle^n$, which is easy to detect. We denote the uniform superposition as

$$|\phi_0\rangle := \frac{1}{\sqrt{2^n}} \sum_{\mathtt{b} \in \{\mathtt{0},\mathtt{1}\}^n} |\mathtt{b}\rangle \,. \tag{3.7}$$

The operator U_0 flips the sign (the phase) over $|\mathtt{0}\rangle$ and acts as the identity on orthogonal states, with a simple circuit (Figure 3.2). Since H is a unitary, $H^{\otimes n} U_0 H^{\otimes n}$ will flip the sign on $|\phi_0\rangle$ and act as the identity on orthogonal states.



(a) Using an unbounded fanin `CNOT`.

(b) Using `TOFFOLI` and ancilla qubits.

Figure 3.2: The operator U_0 in Grover's algorithm, for 4-bit strings. In general this circuit uses $\Theta(n)$ X gates, $\Theta(n)$ `TOFFOLI` gates, and $\Theta(\log n)$ ancilla qubits initialized to $|0\rangle$.

Viewing Grover's algorithms as reflections about an average value is more sensible as a computation, but viewing it as rotations is more geometrically intuitive.

3.1.1 Reflections about the Average

In a uniform superposition over n-bit strings, the probability of measuring a string in S is $|S|/2^n$, the same as classically picking a random string. We can think of the steps in Grover's algorithm as a method to concentrate the amplitude of the state onto the strings in S.

Lemma 3.1.1. *The operation $H^{\otimes n}U_0H^{\otimes n}$ will reflect a state about its average coefficient with respect to the computational basis. That is, suppose we have a superposition of n-bit strings, as*

$$|\phi\rangle = \sum_{\mathtt{b}\in\{\mathtt{0},\mathtt{1}\}^n} \alpha_{\mathtt{b}} |\mathtt{b}\rangle \tag{3.8}$$

for some $\alpha_{\mathtt{b}} \in \mathbb{C}$. Then

$$H^{\otimes n}U_0H^{\otimes n} |\phi\rangle = \sum_{\mathtt{b}\in\{\mathtt{0},\mathtt{1}\}^n} (\alpha_{\mathtt{b}} - 2\overline{\alpha}) |\mathtt{b}\rangle \tag{3.9}$$

where $\overline{\alpha}$ is the average value of $\alpha_{\mathtt{b}}$.

The proof is omitted. Figure 3.3 demonstrates how reflections about the average increase the necessary amplitude(s).

Part 3.3e is the maximum possible amplitude for the marked state, yet there is still a 13% chance of measuring a non-marked state. This is a normal property of Grover's algorithm: It is inherently probabilistic. Since each step is a discrete jump in amplitude, for most values of n we cannot achieve 100% probability of measuring a marked state.

3.1.2 Rotations

We can take the state $|\phi_0\rangle$ and decompose it as:

$$|\phi_0\rangle = \underbrace{\tfrac{1}{\sqrt{|S|}} \sum_{\mathtt{b}\in S} |\mathtt{b}\rangle}_{:=|S\rangle} + \underbrace{\tfrac{1}{\sqrt{2^n-|S|}} \sum_{\mathtt{b}\notin S} |\mathtt{b}\rangle}_{:=\left|S^\perp\right\rangle} . \tag{3.10}$$

The state $|S\rangle$ is the uniform superposition of elements in S, and $\left|S^\perp\right\rangle$ is the uniform superposition of elements not in S.

The states reached by Grover's algorithm will remain in the two-dimensional subspace spanned by $|S\rangle$ and $\left|S^\perp\right\rangle$. This means that U_S is really a reflection over $\left|S^\perp\right\rangle$ and $H^{\otimes n}U_0H^{\otimes n}$ is a reflection over $|\phi_0\rangle$.

We recall a fact of geometry: In a two-dimensional vector space, given two vectors $\vec{x}$ and $\vec{y}$, reflecting a third vector $\vec{z}$ over $\vec{x}$ and then over $\vec{y}$ will have the effect of rotating $\vec{z}$ by twice the angle between $\vec{x}$ and $\vec{y}$. This means we can visualize Grover's algorithm with Figure 3.4.

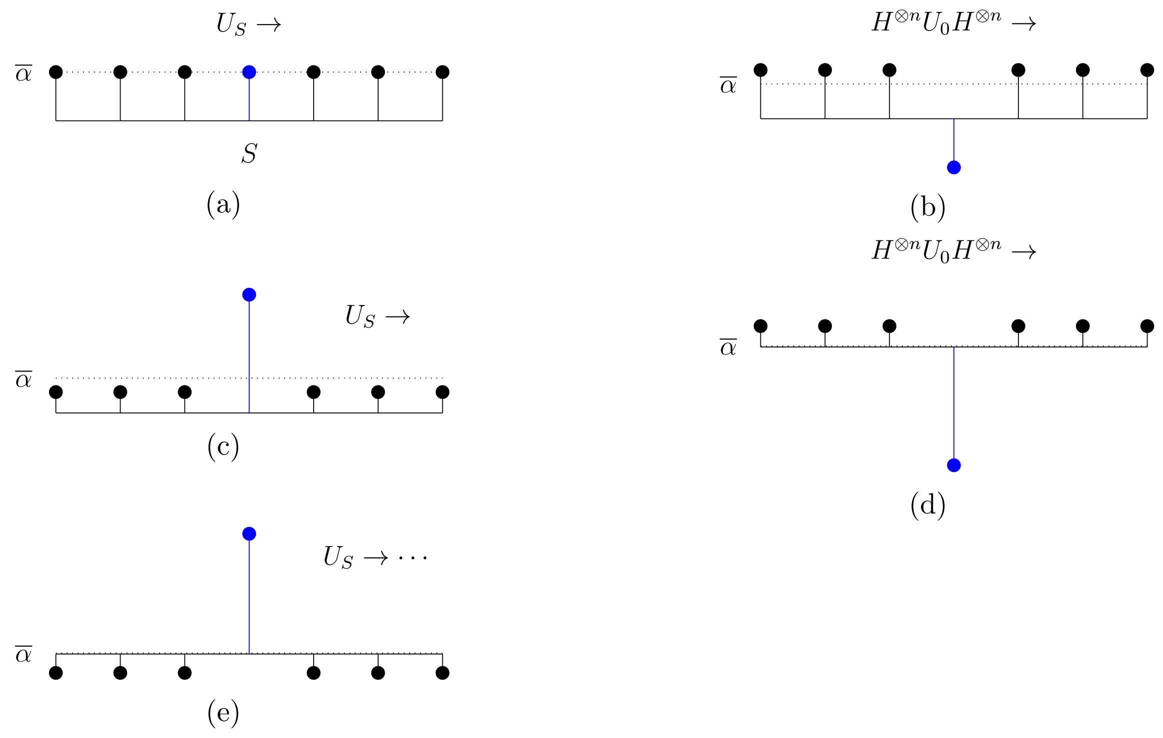
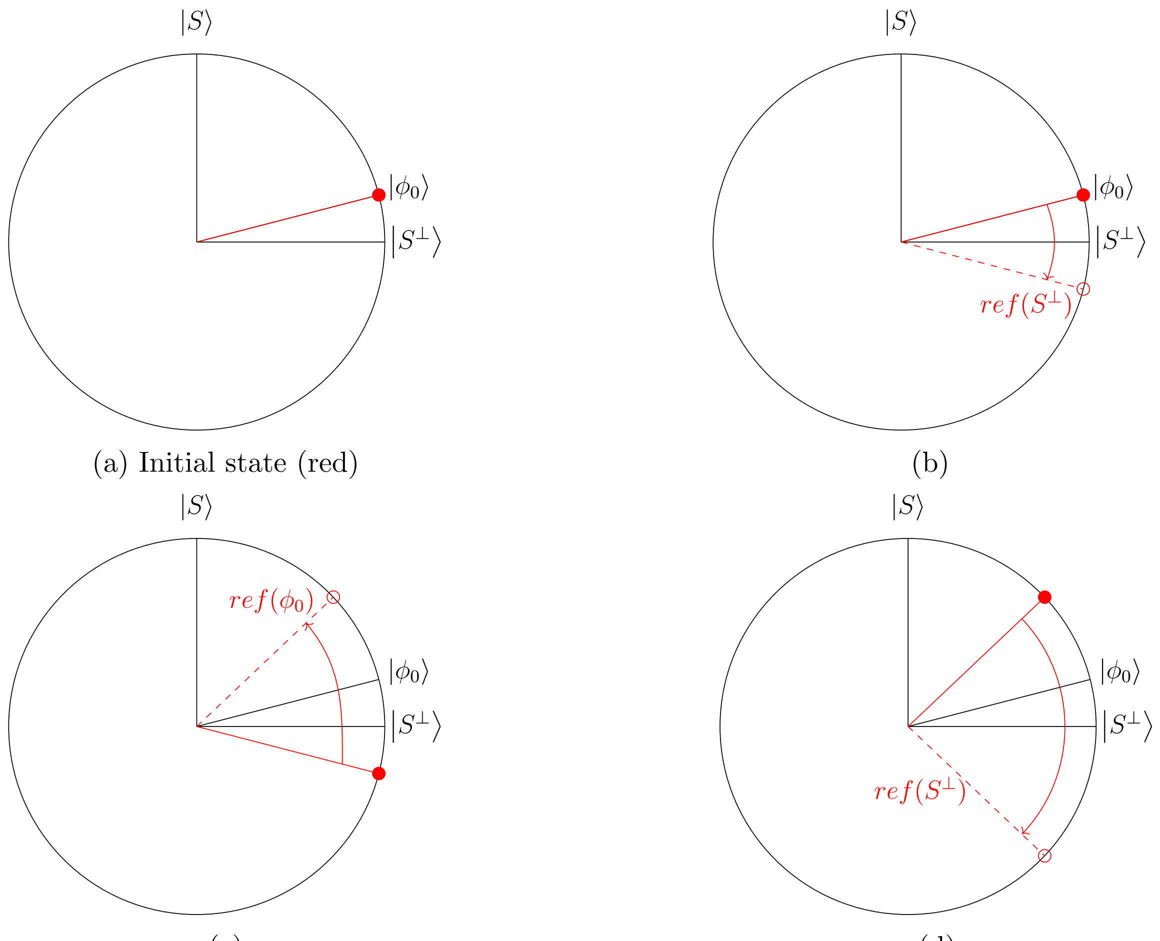


Figure 3.3: Grover's algorithm applied to a superposition of 7 elements. Each dot is a different state, and the height represents the amplitude of that state in the superposition. The reflection U_S ensures that the average value (the dotted line) stays between the marked state, which is in S, and the other states. This ensures that the amplitude of the marked state is further from the average. This means that the reflection over the average will exaggerate the differences in amplitudes. If the process were to continue past step 3.3e, it would reverse, and shrink the difference in amplitude between the marked state and the average.



(a) Initial state (red) (b) (c) (d)

Figure 3.4: Grover's algorithm in the subspace spanned by $|S\rangle$ and $|S^\perp\rangle$. The red dot is the current state of the algorithm. By reflecting over $|S^\perp\rangle$ and $|\phi_0\rangle$, the state rotates towards $|S\rangle$.

This perspective makes the complexity analysis easy. The angle between $|\phi_0\rangle$ and $|S^\perp\rangle$ is

$$\theta_0 := \cos^{-1}(\langle \phi_0 | S^\perp \rangle) = \cos^{-1}\left(\sqrt{\frac{2^n - |S|}{2^n}}\right) \approx \sqrt{\frac{|S|}{2^n}}. \tag{3.11}$$

To reach $|S\rangle$, we need to rotate by $\frac{\pi}{2} - \theta_0$. Hence the total number of rotations is

$$\frac{\frac{\pi}{2} - \theta_0}{\theta_0} \approx \frac{\pi}{2}\sqrt{\frac{2^n}{|S|}}. \tag{3.12}$$

If $|S| = 1$, then this gives us the familiar $O(\sqrt{2^n})$ run-time, assuming each rotation takes time $O(1)$. As $|S|$ increases, the required number of rotations decreases.

A single Grover step will always rotate by the same value, meaning that once we reach a state close to $|S\rangle$, more Grover steps will overshoot the marked elements! To get close to $|S\rangle$, we need to know $|S|$, yet sometimes we don't have this information.

Since the state is rotating around, its projection onto $|S\rangle$ forms a sinusoidal wave, shown in Figure 3.5. Picking a random t between 2 and $2^{n/2}$ picks a random point on a wave with at least 1/4 of a full rotation. The probability of measuring something in S will thus be the integral of this wave, which is approximately 1/2. Thus, repeating this procedure with different random t will exponentially decrease the chance of failure.

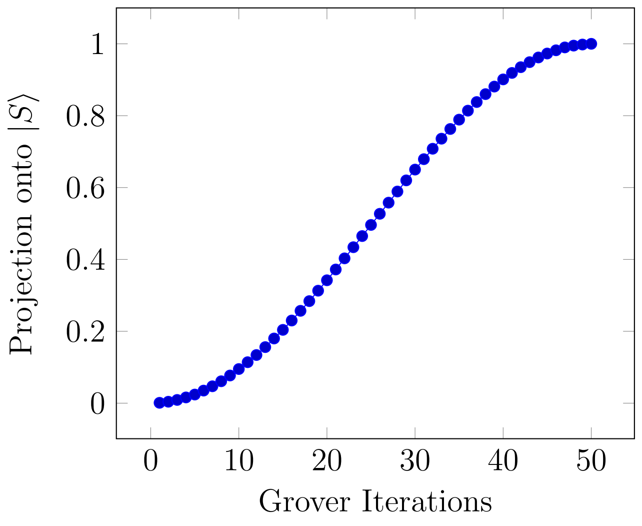


Figure 3.5: Probability of measuring an element of S, as a function of the number of Grover steps.

3.1.3 Important Points

Grover's algorithm requires $O(2^{n/2})$ applications of the "oracle" operator U_S. This is provably optimal for such an oracle [58]. Unless we exploit some underlying feature of our search problem, there is no algorithm that can outperform Grover.

Looking carefully at the algorithm, we need to uncompute any intermediate computations that U_S uses to ensure the proper interference when using $H^{\otimes n} U_0 H^{\otimes n}$. We may find an application where the total search space is small but U_S is expensive. This is the motivation between quantum random walks.

A second approach for an unknown size of S is Figure 3.6. We run the algorithm repeatedly, with a different number of total iterations t each time. If we triple t each time, we cannot "overshoot" an ideal range (shaded in the figure) where the probability of success is at least 86%. This method is necessary for quantum random walks.

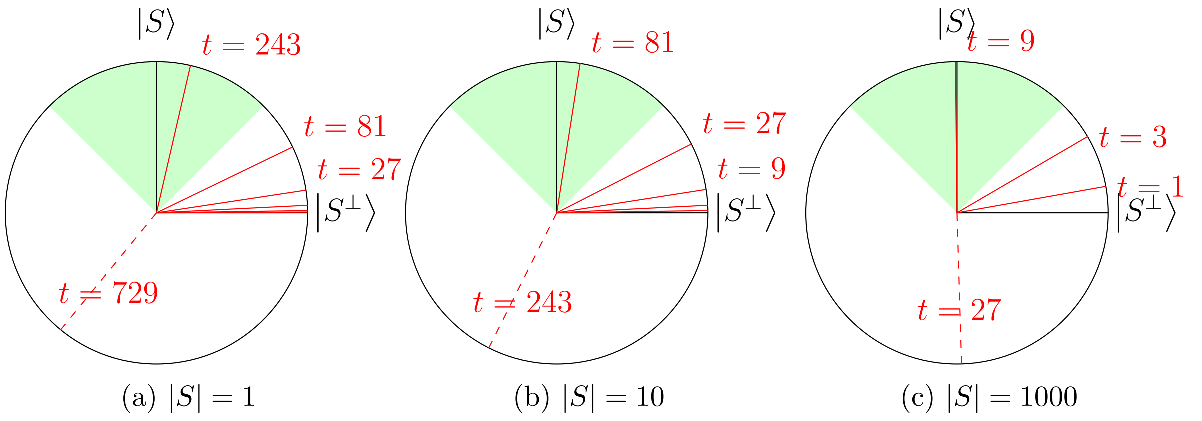


Figure 3.6: Exponentially increasing iterations for Grover's algorithm for $X = \{\texttt{0}, \texttt{1}\}^{15}$

3.2 Classical Random Walks

Let X be any finite set. A Markov process $\mathcal{M}$ on X is a sequence of random variables $X_1, X_2, \cdots$ on X with the "Markov property", which means that the process is forgetful:

$$p(X_t = x | X_{t-1} = y_{t-1}, \cdots, X_1 = y_1) = p(X_t = x | X_{t-1} = y_{t-1}). \tag{3.13}$$

It's time-homogeneous if $p(X_t = x | X_{t-1} = y) = p(X_2 = x | X_1 = y)$, for all $t \geq 2$. This is a random walk: We can start at any point $x \in X$, and imagine walking to different, random elements in X at every time step.

We can capture all the information with a $|X| \times |X|$ transition matrix P, where $P_{xy} = P(X_2 = x | X_1 = y)$. Following the notation for classical computation from Section 1.1.1, we represent a probability distribution on X as a vector $\|x\rangle\rangle$. Then $P\|x\rangle\rangle$ gives the probability distribution after one step. Hence, $P^n\|x\rangle\rangle$ gives the probability distribution after n steps. P will be row-stochastic.

We say $\mathcal{M}$ is *irreducible* if every state is reachable from every other state, meaning for any $x, y \in X$, there is some n such that $(P^n)_{xy} > 0$. By the Perron-Frobenius theorem, this implies P will have an eigenvalue 1 with a unique eigenvector with positive coordinates that we denote $\|\pi\rangle\rangle$. This is called the *stationary distribution*, because it stays "stationary" under the Markov Process, and because we can normalize the positive coordinates to make it a valid probability distribution.

3.2.1 Random Walks as Search

Suppose we have the same set-up as Grover's algorithm: A subset $S \subseteq X$, a circuit to decide if $\mathtt{b} \in S$ for a string $\mathtt{b}$, and we want to output an element $\mathtt{b}$ from S.

The classical analogue to Grover's algorithm is Algorithm 2, a brute force search.

Algorithm 2 A classical brute force search.

```
1: Initialize the state $x$ as a random element of $X$
2: for $t$ steps do
3:     Check if the current state $x$ is in $S$; if so, output $x$ and halt
4:     Choose another random element of $X$
5: end for
6: If the previous steps did not halt, declare $S = \emptyset$
```

For Algorithm 2 to succeed with high probability, we need $t = \Omega(1/\epsilon)$, where $\epsilon = |S|/|X|$. If we let S be the cost of **s**ampling a random element, and C the cost of **c**hecking a random element, then Algorithm 2 costs

$$\mathsf{S} + \frac{1}{\epsilon}(\mathsf{S} + \mathsf{C}). \tag{3.14}$$

If we add a Markov process structure to X, then in Step 4 we could choose a new element according to the transition probabilities. As an example, perhaps from an element x the Markov chain will preferentially choose new elements y such that y can be computed more efficiently using information from x.

This leads us to Algorithm 3.

Algorithm 3 A classical random walk search.

```
1: Initialize the state $x$ as a random element of $X$
2: for $t_2$ steps do
3:     Check if the current state $x$ is in $S$; if so, output $x$ and halt
4:     Take $t_1$ random steps in the Markov chain
5: end for
6: If the previous steps did not halt, declare $S = \emptyset$
```

We might expect that we could take $t_1 = 1$ and $t_2 = \Omega(1/\epsilon)$, the same as before. This will not work because we are not selecting new elements *independently* anymore. If we pick an x particularly "far" from S, then in the next step we will still be "far" from S.

To fix this, we use an important fact of Markov processes: Starting from any distribution on X, if we take enough random steps, we will approach the stationary distribution. Since the stationary distribution is unique, then after enough steps, *the resulting distribution will be independent of where we started.*

Hence, in Step 4, we take t_1 large enough to get close to the stationary distribution and "forget" where we started. Then we move to Step 3 and check. The algorithm now acts like repeated independent random samples from the stationary distribution, so we would need to set $t_2 = \Omega(1/\epsilon)$ for

$$\epsilon = \sum_{x \in S} \pi_x, \tag{3.15}$$

i.e., ϵ is the probability of selecting an element of S when sampling from the stationary distribution.

To find t_1, we need to introduce the spectral gap. The spectral gap of P is the difference between the largest and second-largest eigenvalue of P, and will be denoted δ. Since the largest eigenvalue of P is 1, $\delta = 1 - \lambda_2$ for second-largest eigenvalue λ_2. Perron-Frobenius says $\delta > 0$, and it could be arbitrarily small though not arbitrarily large.

Proposition 3.2.1. *The spectral gap of a Markov chain is at most 1, and this minimum is achieved if and only if $P = \frac{1}{|X|}J$, where J is the all-ones matrix.*

Proof. We know that the trace is the sum of eigenvalues, so $\mathrm{Tr}(P) = 1 + \sum_{\lambda<1} \lambda$. For a fixed trace, we maximize $1 - \lambda_2$ by setting all eigenvalues equal. There are at most $|X|$ eigenvalues, so this gives $\mathrm{Tr}(P) = 1 + (|X| - 1)\lambda_2$. Thus

$$\delta = 1 - \lambda_2 \leq 1 - \frac{\mathrm{Tr}(P) - 1}{|X| - 1} = \frac{|X| - \mathrm{Tr}(P)}{|X|}. \tag{3.16}$$

Every entry of P is non-negative, so $\mathrm{Tr}(P) \geq 0$. Thus, $\delta \leq 1$. $\square$

The spectral gap is important for the mixing time: Starting from any distribution, how many steps do we need to take to be close to the stationary distribution?

Proposition 3.2.2. *Let $\mathcal{M}$ be a Markov process on X such that the transition matrix P is diagonalizable. Let $|p\rangle$ be any probability distribution on X and define $d_0 = \| \, \|p\rangle\!\rangle - \|\pi\rangle\!\rangle \, \|$. Then $\|P^n \|p\rangle\!\rangle - \|\pi\rangle\!\rangle \| \leq d_0(1-\delta)^n$.*

Here we are using $\|p\rangle\!\rangle$ to represent a *classical* probability distribution, following the notation of Section 1.1.1.

Proof. Since P is diagonalizable, we can decompose the input probability distribution $\|p\rangle\!\rangle$ into orthonormal eigenvectors $\|\pi_\lambda\rangle\!\rangle$ of P for each eigenvalue λ, with coefficients a_λ:

$$\|p\rangle\!\rangle = a_1 \|\pi\rangle\!\rangle + \sum_\lambda a_\lambda \|\pi_\lambda\rangle\!\rangle . \tag{3.17}$$

Since $\|\pi_\lambda\rangle\!\rangle$ is orthogonal to $\|\pi\rangle\!\rangle$ for all λ, then the sum of components of $\|\pi_\lambda\rangle\!\rangle$ is 0 for each λ. Hence, the sum of components of $\|p\rangle\!\rangle$ is just a_1. Since $\|p\rangle\!\rangle$ is a probability distribution, we have $a_1 = 1$.

After n steps of the Markov process:

$$P^n \|p\rangle\!\rangle = P^n \|\pi\rangle\!\rangle + \sum_\lambda a_\lambda (P^n \|\pi_\lambda\rangle\!\rangle) \tag{3.18}$$

$$= \|\pi\rangle\!\rangle + \sum_\lambda a_\lambda \lambda^n \|\pi_\lambda\rangle\!\rangle . \tag{3.19}$$

Then we have

$$\|P^n \|p\rangle\!\rangle - \|\pi\rangle\!\rangle \|^2 = \sum_\lambda a_\lambda^2 \lambda^{2n} \tag{3.20}$$

$$\leq \sum_\lambda a_\lambda^2 (1-\delta)^{2n} \tag{3.21}$$

$$= d_0^2 (1-\delta)^{2n}. \tag{3.22}$$

$\square$

If $n = \Omega(\frac{1}{\delta})$, the distance between $P^n \, \|p\rangle\rangle$ and $\|\pi\rangle\rangle$ is approximately 0, for any input probability distribution $\|p\rangle\rangle$. Thus, the "mixing time" of a Markov process, which we will not formally define, will be approximately equal to $\frac{1}{\delta}$, the inverse of the spectral gap.

Thus in Algorithm 3, we take $t_1 = \Theta(\frac{1}{\delta})$. Let U be the cost of selecting a new element according to the Markov process, and S and C be defined as for the brute force search. Then Algorithm 3 costs

$$\mathsf{S} + \frac{1}{\epsilon}\left(\frac{1}{\delta}\mathsf{U} + \mathsf{C}\right). \tag{3.23}$$

Equation 3.14 states that a naive search costs

$$\mathsf{S} + \frac{1}{\epsilon}(\mathsf{S} + \mathsf{C}). \tag{3.24}$$

Thus if $\frac{1}{\delta}\mathsf{U} \leq \mathsf{S}$, a random walk is cheaper than a naive search.

3.2.2 Greedy Random Walk

Algorithm 3 might seem inefficient; why not do a greedy algorithm?

Algorithm 4 A greedy classical random walk.

```
1: Initialize the state $x$ as a random element of $X$
2: for $t_3$ steps do
3:     Check if the current state $x$ is in $S$; if so, output $x$ and halt
4:     Take one random step in the Markov chain
5: end for
6: If the previous steps did not halt, declare $S = \emptyset$
```

Algorithm 4 will succeed with constant probability if $t_3 \geq \frac{1}{\delta\epsilon}$, since by that point it has done all the same steps as Algorithm 3, but with more check steps. Thus we can bound the cost as

$$O\left(\mathsf{S} + \tfrac{1}{\epsilon\delta}(\mathsf{U} + \mathsf{C})\right). \tag{3.25}$$

If the check cost C is substantial, this bound is higher than that of Algorithm 3. However, it's possible that it will take many fewer steps if there is a high probability of finding marked elements in all of the extra check steps. In general it is diffitcult to decide whether the greedy version is cheaper. The answer will depend on the underlying Markov process. Hence, we avoid Algorithm 4 only because there is no proof that it will be an improvement.

3.3 Quantum Random Walks

Quantum walks are the natural quantum analog of the classical walks we just described. Despite many previous results, the first improvement over Grover's algorithm was not until Ambainis in 2003 [5], after which there has been substantial work. The seminal paper now is Magniez, Nayak, Roland, and Santha [44], who unified previous results, proved a general algorithm, and introduced the standard notation.

The analogies between classical walks, quantum walks, and Grover's algorithm are as follows:

- In place of a "check", we use a conditional phase flip, just like U_S in Grover's algorithm.
- In place of a random step, we reflect over a subspace based on the transition probabilities, like $H^{\otimes n}U_0H^{\otimes n}$ in Grover's algorithm.
- Where Grover's algorithm "diffused" uniformly to *every* element, in a random walk we diffuse according to the underlying Markov process.

Figure 3.7 visualizes these relationships for random walks on graphs.

$$\begin{tikzcd}
\boxed{\text{Brute force}} \arrow[r, "\text{Only randomly}", "\text{select neighbours}"'] \arrow[d, "\substack{\text{diffuse to}\\ \text{superposition}\\ \text{phase flip}}", "\substack{\text{Random}\mapsto\\ \\ \text{Check}\mapsto}"'] & \boxed{\substack{\text{Classical}\\ \text{Random Walk}}} \arrow[l, bend right, "\text{Use the complete graph}"'] \arrow[d, "\substack{\text{diffuse to}\\ \text{superposition}\\ \text{phase flip}}", "\substack{\text{Random}\mapsto\\ \\ \text{Check}\mapsto}"'] \\
\boxed{\text{Grover}} \arrow[r, "\text{Only diffuse}", "\text{to neighbours}"'] & \boxed{\substack{\text{Quantum}\\ \text{Random Walk}}} \arrow[l, bend left, "\text{Use the complete graph}"]
\end{tikzcd}$$

Figure 3.7: A diagram of classical and quantum random walk algorithms. In the category of hand-waving analogies, this diagram commutes.

More formally, to convert the classical algorithm to quantum, we first need to make it reversible. To do this, we use "edges", rather than elements, and think about our Markov process as a graph, with edges weighted by probability, and edges of probability 0 removed. The quantum walks use pairs $(x, y) \in X^2$. Thus, at every point in the algorithm, we have a state like

$$\sum_{x,y \in X} \alpha_{xy} |x\rangle |y\rangle , \tag{3.26}$$

where $\alpha_{xy} \in \mathbb{C}$. Denoting the set of edges incident to a vertex x as $I(x)$, we could also use states of the form

$$\sum_{x \in X} \sum_{e \in I(X)} \alpha_{xe} |x\rangle |e\rangle . \tag{3.27}$$

These two equations store the same information, so there is no difference to the function of the algorithm which one we use, so long as all the operations work properly. We use the vertex-and-edge method in Section 4.3.

In almost all applications there is also data associated to each element of x, so we also store the data for each element, denoted $|x\rangle_d$.

As in most quantum analogues of classical algorithms, we replace probability distributions with superpositions. Thus we have a natural stationary "state":

$$|\pi\rangle = \sum_{x \in X} \sqrt{\pi_x} |x\rangle_d . \tag{3.28}$$

The outcome of measuring this state precisely follows the classical probability distribution $\|\pi\rangle\!\rangle$.

We want to initialize the algorithm to this state, so we let $\mathcal{S}$ be an operator that does this. Many applications have uniform stationary distributions, which makes this easier: $\mathcal{S}$ would be a parallel set of Hadamard gates, then the bijection from bitstrings to elements of X, and then a computation of the associated data.

In Grover's algorithm, where we might classically use a check operation, we instead used a conditional phase flip. We do the same thing here. The classical random walk, Algorithm 3, was really two simultaneous searches: One for elements of S within the stationary distribution, and another search for the stationary distribution. Thus, we do the same thing in the quantum setting.

We use walk steps to search for the stationary distribution. A walk step $\mathcal{W}$ involves two subroutines. First, we apply an update operator $\mathcal{U}$, whose action on each state is like a random walk step:

$$\mathcal{U}\left|x\right\rangle_d \left|0\right\rangle = \sum_{y\in X} \sqrt{P_{yx}}\left|x\right\rangle_d \left|y\right\rangle_d . \tag{3.29}$$

But we also include, in analogy with the operator U_0 in Grover's algorithm, a reflection $\mathcal{R}_0$ about $\left|0\right\rangle^n$, whose action is defined on computational basis states $\left|y\right\rangle$ as:

$$\mathcal{R}_0\left|x\right\rangle_d\left|y\right\rangle = \begin{cases} -\left|x\right\rangle_d\left|y\right\rangle & , y \neq 0^n \\ \left|x\right\rangle_d\left|y\right\rangle & , y = 0^n \end{cases} . \tag{3.30}$$

Algorithm 5 describes a full quantum walk step.

Algorithm 5 A quantum random walk step, due to Szegedy [53].

```
1: Apply $\mathcal{U}^{-1}$
2: Apply $\mathcal{R}_0$
3: Apply $\mathcal{U}$
4: Swap the first and second registers
5: Repeat Steps 1 to 3.
```

In this, $\mathcal{U}$ is analogous to $H^{\otimes n}$ in Grover's algorithm. Together with $\mathcal{R}_0$ it has the effect of reflecting about the average coefficient of "neighbours" of x.

For the check step, we flip the phase of those $\left|x\right\rangle_d$ for $x \in S$ with an operator $\mathcal{C}$:

$$\mathcal{C}\left|x\right\rangle_d\left|y\right\rangle_d = \begin{cases} -\left|x\right\rangle_d\left|y\right\rangle_d & , x \in S \\ \left|x\right\rangle_d\left|y\right\rangle_d & , x \notin S \end{cases} . \tag{3.31}$$

Algorithm 6 A quantum random walk in the style of Ambainis.

```
1: Apply $\mathcal{S}$ to $|0\rangle|0\rangle$ to create $|\pi\rangle|0\rangle$.
2: **for** $t_1$ times **do**
3:     Apply $\mathcal{C}$
4:     Apply $\mathcal{W}$ for $t_2$ repetitions
5: **end for**
6: Measure the state.
```

Our first attempt at a quantum random walk is to assemble these operators in the same way as a classical random walk, which is Algorithm 6. This is the original quantum random walk that Ambainis described [5]. The analysis is tricky: It's not clear what $\mathcal{W}$ actually does. Ideally, it would reflect over $|\pi\rangle$ and rotate in a two-dimensional subspace just like Grover's algorithm. However, $\mathcal{W}$ does not reflect over $|\pi\rangle$, nor is it constrained to a two-dimensional subspace. Thus, it may not solve the search problem and measuring the final state may not yield an element of S.

Ambainis designed a particular Markov process to solve the element distinctness problem, then proved that Algorithm 6 *will* solve the search problem for that particular Markov process. However, there is no proof that Algorithm 6 works for other Markov processes.

3.3.1 Szegedy's Algorithm

Algorithm 7 Szegedy's random walk algorithm.

```
1: Apply $\mathcal{S}$ to $|0\rangle|0\rangle$ to create $|\pi\rangle|0\rangle$.
2: for $t_3$ times do
3:     Apply $\mathcal{C}$
4:     Apply $\mathcal{W}$
5: end for
6: Measure the state.
```

Szegedy expanded on Ambainis' approach [53] and analyzed a quantum version of the Algorithm 4, the greedy walk. Szegedy proved that we can use Algorithm 7 to solve the *decision* problem, where we want to decide if S is empty or not. If S is empty, then $\mathcal{C}$ acts as the identity. It should also be clear that $\mathcal{W}$ acts as the identity on $|\pi\rangle|0\rangle$. Thus, if S is empty, the output will always be $|\pi\rangle|0\rangle$. Szegedy showed that if S is non-empty, if $t_3 = O(\frac{1}{\sqrt{\epsilon\delta}})$, with ϵ and δ defined in the same way as for classical random walks, then the output state will be *different* than $|\pi\rangle|0\rangle$. More precisely, the inner product between the two states will be below a constant value. Thus, we can use entanglement to solve the decision problem, giving Algorithm 8.

Algorithm 8 has cost

$$O\left(\mathsf{S} + \frac{1}{\sqrt{\epsilon\delta}}(\mathsf{U} + \mathsf{C})\right). \tag{3.32}$$

For any Markov process, if $t_3 = O(\frac{1}{\sqrt{\epsilon\delta}})$, then if M is empty, Algorithm 8 will always measure $|+\rangle$, and if M is non-empty, there is a constant probability of measuring $|-\rangle$.

Algorithm 8 Szegedy's random walk decision algorithm.

```
1: Initialize an ancilla qubit to $|+\rangle$
2: Use the ancilla qubit to control the following steps:
3:     Apply $\mathcal{S}$ to $|0\rangle|0\rangle$ to create $|\pi\rangle|0\rangle$.
4:     for $t_3$ times do
5:         Apply $\mathcal{C}$
6:         Apply $\mathcal{W}$
7:     end for
8: Measure the ancilla qubit in the $\{|+\rangle, |-\rangle\}$ basis.
```

In many cases it is straightforward to divide the set X to perform a binary search, thus solving the search problem with Szegedy's decision algorithm. The total cost will depend on how S, U, and C scale as the search space gets smaller.

3.3.2 MNRS Algorithm

MNRS manged to constrain the algorithm to the 2-dimensional subspace spanned by $|\pi\rangle$ and the marked vertices. To do this they reflect over $|\pi\rangle$; to do that, they use phase estimation. They wrapped the phase estimation in a Recursive Amplitude Amplification to keep the errors low throughout the algorithm.

Phase Estimation

Phase estimation is a common quantum technique. It's famously used in Shor's algorithm, and Cleve, Ekert, Macchievello, and Mosca (CEMM) [22] gave a general form, Algorithm 9. Figure 3.8 shows a circuit for it.

Algorithm 9 Phase estimation.

```
Require: A circuit $\mathcal{W}$, a state $|\phi\rangle$, and an integer $s$
1: Apply $H^{\otimes s}$ to an ancilla register of $|0\rangle^s$
2: Use the ancilla to control the number of applications of $\mathcal{W}$, i.e., $|n\rangle|\phi\rangle \mapsto |n\rangle \mathcal{W}^n |\phi\rangle$
3: Apply an $s$-bit inverse quantum Fourier transform to the ancilla
```

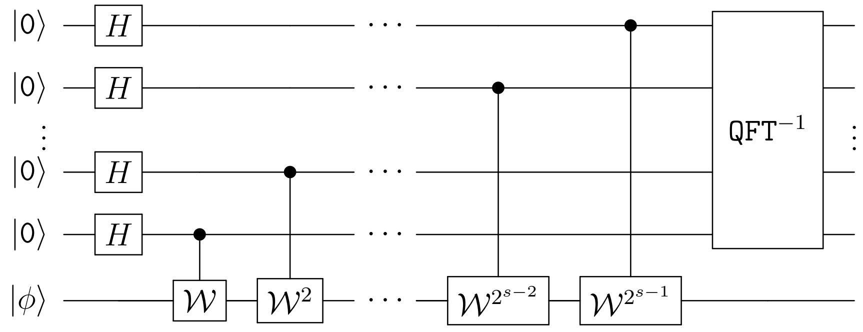


Figure 3.8: Circuit for phase estimation.

Since $\mathcal{W}$ is a unitary, all its eigenvalues have the form $e^{i\theta}$; we will let spec$\mathcal{W}$ denote the set of all such θ. We can decompose any input $|\phi\rangle$ as

$$|\phi\rangle = \sum_{\theta \in \mathrm{spec}\mathcal{W}} \alpha_\theta |\psi_\theta\rangle \,, \tag{3.33}$$

where $|\psi_\theta\rangle$ are the eigenvectors of $\mathcal{W}$.

Theorem 3.3.1 ([22])**.** *Applying phase estimation to $|\phi\rangle$ will produce state 3.34 in the ancillae.*

$$\sum_{\theta \in spec\mathcal{W}} \alpha_\theta |b_\theta\rangle \,, \tag{3.34}$$

The value b_θ is close to an s-bit approximation to θ, and satisfies

$$\left|\left\langle b\right| \left\lfloor \tfrac{2^s\theta}{2\pi} \right\rceil \right\rangle\right| \geq 2/\pi. \tag{3.35}$$

Proof. Let $|\phi\rangle$ be an eigenvector of $\mathcal{W}$ with eigenvalue $e^{i\theta}$. After Step 2, we have the following state:

$$\frac{1}{2^{s/2}} \sum_{n\in\{0,1\}^s} |n\rangle \mathcal{W}^n |\phi\rangle = \frac{1}{2^{s/2}} \sum_{n\in\{0,1\}^s} |n\rangle e^{in\theta} |\phi\rangle \,. \tag{3.36}$$

Applying an inverse quantum Fourier transform gives:

$$\mathtt{QFT}^{-1} \frac{1}{2^{s/2}} \sum_{n\in\{0,1\}^s} |n\rangle e^{in\theta} |\phi\rangle = \frac{1}{2^s} \sum_{n\in\{0,1\}^s} \sum_{m\in\{0,1\}^s} e^{2\pi i \frac{nm}{2^s}} |m\rangle e^{in\theta} |\phi\rangle \tag{3.37}$$

$$= \frac{1}{2^s} \sum_{m\in\{0,1\}^s} \underbrace{\left(\sum_{n\in\{0,1\}^s} e^{in\left(-\frac{2\pi m}{2^s}+\theta\right)} \right)}_{(A)} |\phi\rangle \tag{3.38}$$

$$\approx \left| \left\lfloor \tfrac{2^s\theta}{2\pi} \right\rceil \right\rangle |\phi\rangle \tag{3.39}$$

This works because the sum in 3.38 (A) is approximately a sum of roots of unity if $\frac{2\pi m}{2^s} \not\approx \theta$, and it's approximately a sum of `1`s if $\frac{2\pi m}{2^s} \approx \theta$. We will omit the precise details. $\square$

Our goal was to use the eigenvalues to distinguish $|\pi\rangle$ from orthogonal states in order to reflect over $|\pi\rangle$. The phase θ for $|\pi\rangle$ will be 0, and for any orthogonal eigenvector of $\mathcal{W}$, the phase θ will be non-zero. We need the minimum value of s such that for all $\theta \neq 0$, $\lfloor 2^s\theta/2\pi \rceil \neq 0$. This resembles the spectral gap, and Szegedy made this connection precise.

Theorem 3.3.2 (Theorem 1, [53])**.** *Let P be an irreducible Markov chain, and $\mathcal{W}$ be defined as before. Let $\cos(\theta_1), \cdots, \cos(\theta_\ell)$ be the singular values of P, such that $\delta = 1 - \cos(\theta_1)$. Then the eigenvalues of $\mathcal{W}$ are*

$$\left\{\pm 1, e^{\pm 2i\theta_1}, \cdots, e^{\pm 2i\theta_\ell}\right\}. \tag{3.40}$$

Theorem 3.3.2 implies that the maximum non-unit eigenvalue of $\mathcal{W}$, $e^{2\theta_1 i}$, has θ_1 equal to $2\cos^{-1}(1-\delta) \approx 2\sqrt{\delta}$. Thus, we need $2^s 2\sqrt{\delta}/2\pi > 1/2$, and thus $2^s > \frac{\pi}{2\sqrt{\delta}}$. Phase estimation requires applying $\mathcal{W}$ up to 2^s times, so the bound of 2^s means we need to apply it $O(\frac{1}{\sqrt{\delta}})$ times.

Equation 3.35 implies that after phase estimation the fidelity is only $2/\pi$. We could increase fidelity by increasing s, but the cost of phase estimation, dominated by the repeated applications of $\mathcal{W}$, increases exponentially with s. Instead, we can simply repeat the phase estimation k times. If the output phases are stored in registers $|\omega_1\rangle, \cdots, |\omega_k\rangle$, then after phase estimation we have

$$|\psi\rangle |\omega_1\rangle \cdots |\omega_k\rangle. \tag{3.41}$$

Suppose $|\psi\rangle$ is an eigenvector of $\mathcal{W}$ for a non-unit eigenvalue. We want to flip its phase. This uses the following operator:

$$\mathcal{R}_{\pi,s,k} |\psi\rangle |\omega_1\rangle \cdots |\omega_k\rangle = \begin{cases} -|\psi\rangle |\omega_1\rangle \cdots |\omega_k\rangle & ,\omega_i \neq 0 \text{ for any } i \\ |\psi\rangle |\omega_1\rangle \cdots |\omega_k\rangle & ,\omega_1 = \cdots = \omega_k = 0. \end{cases} \tag{3.42}$$

We can construct $\mathcal{R}_{\pi,s,k}$ by combining Toffoli gates to make a multi-`OR` gate of "$\omega_i \neq 0$", and use the output to control a `NOT` on an ancilla $|-\rangle$ state, just like with Grover's algorithm.

Since each $|\omega_i\rangle$ has projection $2/\pi$ onto the space orthogonal to $|0\rangle$, the projection of $|\psi\rangle |\omega_1\rangle \cdots |\omega_k\rangle$ onto the -1 eigenspace of $\mathcal{R}_{\pi,s,k}$ will be $1-(\frac{2}{\pi})^k$. Thus, repeating the phase estimation k times, which only increases the cost by a linear factor, reduces the error by an exponential factor.

If we denote s-bit phase estimation on $\mathcal{W}$ repeated k times as $\mathcal{P}(\mathcal{W}, s, k)$, then we define the full phase-estimation-and-flip subroutine as

$$\mathcal{R}(\mathcal{W}, s, k) = \mathcal{P}(\mathcal{W}, s, k)\mathcal{R}_{\pi,s,k}\mathcal{P}(\mathcal{W}, s, k). \tag{3.43}$$

Analysis:

MNRS showed that their algorithm solves the search problem after $1/\sqrt{\epsilon}$ iterations. If W is the cost to reflect over the stationary distribution, the total cost becomes:

$$O\left(\mathsf{S} + \frac{1}{\sqrt{\epsilon}}(\mathsf{W} + \mathsf{C})\right). \tag{3.44}$$

Since the reflection uses phase estimation, which uses $O(\frac{1}{\sqrt{\delta}})$ repetitions of the update plus some negligible extra gates (the `QFT` and `HADAMARD`s), the total cost becomes

$$O\left(\mathsf{S} + \frac{1}{\sqrt{\epsilon}}\left(\frac{1}{\sqrt{\delta}}\mathsf{U} + \mathsf{C}\right)\right). \tag{3.45}$$

This should remind you of the cost of Algorithm 3, Equation 3.23. If the analogy holds, it suggests that phase estimation and reflecting over the uniform distribution is analogous to the classical technique of taking enough random steps to sample from the stationary distribution. We do not have any intuition for the connection between these processes.

Recursive Amplitude Amplification:

The algorithm we described from MNRS uses ancilla qubits for each phase estimation, but these would need to be maintained after the phase estimation. This would result in $1/\sqrt{\epsilon}$ different, useless ancilla registers. To avoid this, MNRS invented Recursive Amplitude Amplification (RAA).

For RAA we use ℓ different "layers" of phase estimation, each using increasing precision and each using distinct ancilla qubits to hold the phase estimations. Thus, we define s_i and k_i for each layer, and we have states of the following form:

$$|\psi\rangle := |v_d\rangle\, |u_d\rangle\, |\omega_1\rangle^{s_1 k_1}\, |\omega_2\rangle^{s_2 k_2} \cdots |\omega_\ell\rangle^{s_\ell k_\ell}\,. \tag{3.46}$$

That is, each state $|\omega_i\rangle^{s_i k_i}$ contains k_i separate s_i-bit phase estimation results. Then we define $\mathcal{R}_{\pi,i}$:

$$\mathcal{R}_{\pi,i}|\psi\rangle = \begin{cases} -|\psi\rangle, & \omega_j \neq 0 \text{ for any } j \leq i \\ |\psi\rangle & ,\omega_1 = \cdots = \omega_i = 0 \end{cases}. \tag{3.47}$$

This lets us define the circuit $\mathcal{A}_i$ recursively: $\mathcal{A}_0$ is the identity, and for $i \geq 0$:

$$\mathcal{A}_{i+1} = \mathcal{A}_i \mathcal{C} \mathcal{A}_i^{-1} \mathcal{R}_{\pi,i} \mathcal{P}(\mathcal{W}, s_i, k_i) \mathcal{A}_i. \tag{3.48}$$

Careful accounting will show that $\mathcal{R}_{\pi,i}$ and $\mathcal{P}(\mathcal{W}, s_i, k_i)$ are each called $3^{\ell-i}$ times over the course of RAA, and $\mathcal{C}$ is called $\frac{1}{2}(3^{\ell+1}-1)$ times.

From MNRS, $k_i = \log\left(\frac{18}{4\pi^3}\frac{\gamma}{i^2}\right)$, for a positive precision parameter $\gamma \leq 1/40$. Adding up the size of the ancillas shows that we need $O(\log(1/\epsilon\delta))$ ancilla qubits for all the layers of phase estimation.

3.3.3 Grover vs. Other Random Walks

Grover's algorithm can be viewed as a random walk [52]. The underlying Markov process is uniform: starting from any $x \in X$, the probability of walking to any y is constant, equal to $\frac{1}{|X|}$. This means δ takes its maximum value of 1. We can further conclude that the update cost for Grover's algorithm is equal to the set-up cost, since in both cases, a random element is selected. Thus, in the MNRS language, Grover's algorithm has cost

$$O\left(\mathsf{S} + \frac{1}{\sqrt{\epsilon}}(\mathsf{S} + \mathsf{C})\right). \tag{3.49}$$

Hence, if we have a random walk where $\mathsf{S} \leq \frac{1}{\sqrt{\delta}}\mathsf{U}$, we are better off ignoring whatever original Markov structure was present and just using Grover's algorithm. This is analogous to the classical case, though since $\frac{1}{\sqrt{\delta}} \leq \frac{1}{\delta}$, this is less restrictive than the classical requirement.

This is still a strong requirement and limits the practical applicability of random walks. The most common method to achieve the required inequality is to use random walks on a Johnson graph.

Definition 3.3.1. *A Johnson graph on a set X, for an integer R, is a graph $J(X, R)$ whose vertices are all the subsets of X of size R, and two vertices v and u are adjacent, denoted $v \sim u$, if and only if $|v \cap u| = R - 1$.*

Algorithms for information set decoding [37], triangle-finding [43], subset-sum [9], claw-finding [55], and element distinctness [5] use random walks on a Johnson graph, and we are unaware of any applications of quantum random walks that do not use Johnson graphs in some way. The rationale is that the vertices can be kept as sorted lists, and then insertion and deletion are much faster than the construction of the full list. However, the idea that insertion into a list is cheaper than constructing a list is a heuristic borrowed from classical algorithms, and as we argued in Section 1.3, not every classical heuristic will transfer to quantum.

Chapter 4

Quantum Data

As most quantum walks are memory-intensive algorithms, we need to discuss quantum memory and data structures.

With the perspective of "gates as processes", random memory access becomes very expensive because access to N bits of memory requires $\Omega(N)$ gates, and every memory access incurs the full gate cost. Sections 4.1.1 and 4.1.2 give two potential random access circuits which nearly match the lower bounds.

Section 4.2 gives details on specific data structures necessary for quantum random walks. The primary difficulty is that a quantum data structure must be *history independent* for it to interfere properly in quantum algorithms. This means the physical memory layout cannot depend on how the data was constructed. Most classical data structures fail this requirement.

Quantum Radix Trees (Section 4.2.1) rely on many random memory accesses, and accounting for the full cost makes radix trees expensive. Instead, the easiest way to maintain a history independent list is to keep it physically sorted. Classically this is rare because every insertion and deletion must move most of the elements. In a quantum computer, we already pay a linear cost for memory accees, so this approach is more palatable. Section 4.2.2 explains a "sliding sorted array" that uses this method and ends up cheaper than a quantum radix tree.

Section 4.3 gives some specific details for how these data structures work in Johnson graphs.

4.1 Quantum Memory Access

Using QRAM is common in random walks. In his seminal paper on random walks on Johnson graphs, Ambainis adds an extra "random access" gate to the usual gate set, the same as in Section 2.3.3. Jeffery takes the cost at $O(\log N)$ for N bits of memory [35], while Beals et al. [7] define memory access as a single time step, though they later show that it does require $O(\log N)$ steps if the memory access gate is built from smaller gates.

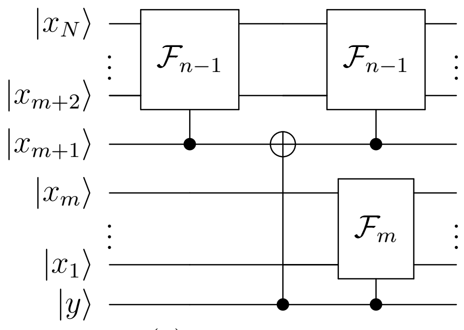
These are reasonable estimates of time, but Theorem 2.3.1 shows that the gate costs are high. Here we give two different memory access approaches that nearly match this lower bound, with logarithmic depth.

4.1.1 Fanout Memory

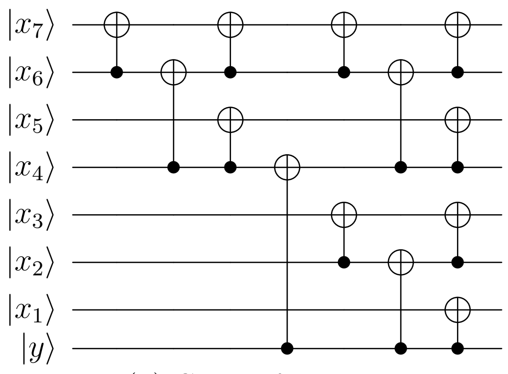
A *fanout* gate of size N has the following action on computational basis states:

$$|y\rangle |x_1\rangle \cdots |x_N\rangle \mapsto |y\rangle |x_1 \oplus y\rangle \cdots |x_N \oplus y\rangle . \tag{4.1}$$

Fanout acts like a multi-target `CNOT`. We can construct a fanout circuit recursively using only `CNOT`. Clearly when $N = 1$, a single `CNOT` suffices. For any other N, we use the circuit in Figure 4.1, letting $n + m = N$ and denoting an n-element fanout with $\mathcal{F}_n$:



(a) Recursive circuit



(b) Circuit for 7 memory elements.

Figure 4.1: General and specific circuits for fanout.

This tree structure is due to Moore [46]. This gives a recursive relationship that lets us conclude that `FANOUT` requires $O(N)$ `CNOT` gates, in depth $O(\log N)$. If we assume an optimal memory layout in d-dimensional space, we can conclude that the CNOT gate requires $O(N^{1/d})$ time; this gives a total time of $O(N^{1/d})$ as well.

For the passive local model, the ith layer in the recursion uses 2^i CNOT gates, each of distance $N/2^i$, so the gate-time cost is $2^i(N/2^i)^{1/d}$. Summing this over all $\log N$ layers gives a gate-time cost of $O(N)$. This assumes the qubits are laid out in an efficient way.

Cost 4.1 summarizes the costs.

We can use half as many gates if we have a parallel array of memory; this does not change the asymptotics, but it's the circuit we show in Figure 4.2.

Cost 4.1 Fanout circuit

Model	Measure	Cost
Passive Circuit (2.4.1)	Gates	$O(N)$
Active Circuit (2.4.2)	Total	$O(N \log N)$
	Depth	$O(\log N)$
	Width	$O(N)$
Passive Latency (2.4.3)	Gate-Time	$O(N)$
Active Local (2.4.4)	Total	$O(N^{1+1/d}(\log N)^d)$
	Depth	$O(N^{1/d})$
	Width	$O(N)$

Moore also notes that if we conjugate a FANOUT with HADAMARD gates, it swaps the roles of control and target. That is, it will have the action

$$|y\rangle |x_1\rangle \cdots |x_N\rangle \mapsto |y \oplus x_1 \oplus \cdots \oplus x_N\rangle |x_1\rangle \cdots |x_N\rangle \,, \tag{4.2}$$

what he calls a PARITY gate - an enormous, multi-input XOR. The use of HADAMARD gates does not change the asymptotics of any cost measure we consider, so PARITY has the same cost as FANOUT.

A fanout memory access (from [34]) to an array A first fans out the address i to a second array A'. Then at each cell, we can copy the contents, in parallel, conditioned on whether the address of that cell equals the address in A'. This will only be true for one cell, i, so we only copy out the necessary contents. This method is very similar to a sliding sorted array [34].

Figure 4.2 summarizes the following steps. We want to access the ith element of an N-cell array A. We use three ancilla arrays A', A'', and A''', all initialized to zero.

Fanout: Use a fanout to copy the input address i to every cell of A'.

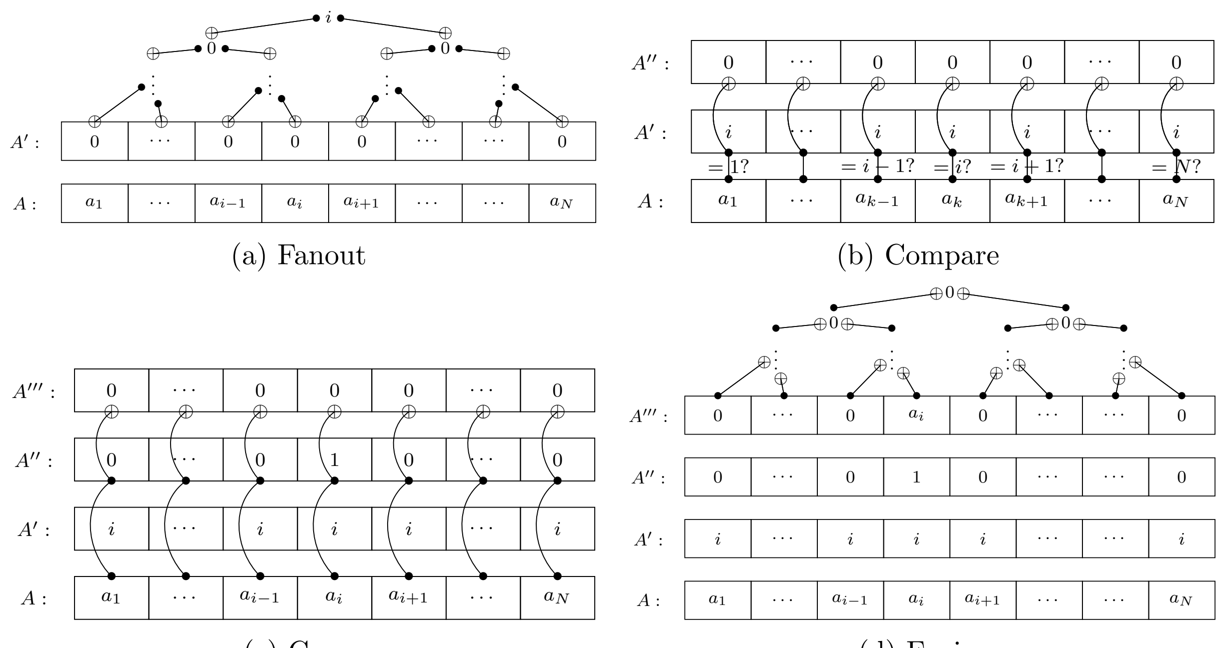
Compare: For each cell j, compute the boolean value of $(A'[j] == j)$ and copy the result to A''. Note that this can use custom circuits for each cell. Since $A'[j] = i$ for all j, then after this step $A''[j] = \delta_{ij}$.

We also want to use smaller fanout circuits to copy the result to all m bits in each cell of A''.

Copy: For each cell j, use $A''[j]$ to control a copy from $A[j]$ to $A'''[j]$ (i.e., m Toffoli gates). After this step, $A'''[j] = 0$ for all $j \neq i$, and $A'''[i] = A[i]$.

Fanin: Use a `PARITY` gate from all cells of A''' to the output register. Since all but one cell of A''' are zero, this will simply copy the non-zero cell, which is $A'''[i] = A[i]$.

Uncompute: Uncompute all the previous steps to clear the ancilla arrays back to zero.



(a) Fanout

(b) Compare

(c) Copy

(d) Fanin

Figure 4.2: Memory access with fanout, drawing heavily from [34]. See text for full description.

Cost 4.2 gives all the costs for m-bit memory. We assume that m is small enough that latency is irrelevant, and use a comparator circuit of $O(m)$ gates, depth $O(\log m)$ and with $O(m)$ [56].

Cost 4.2 Fanout memory access

Model	Measure	Cost
Passive Circuit (2.4.1)	Gates	$O(Nm)$
Active Circuit (2.4.2)	Total	$O(Nm \log N)$
	Depth	$O(\log N)$
	Width	$O(Nm)$
Passive Latency (2.4.3)	Gate-Time	$O(Nm)$
Active Local (2.4.4)	Total	$O((Nm)^{1+1/d}(\log(Nm))^d)$
	Depth	$O((Nm)^{1/d})$
	Width	$O(Nm)$
QRAM (2.4.5)	Gates	1
	Time	$O((Nm)^{1/d})$

4.1.2 Sorting Network Memory Access

It's clear looking at Figure 4.2 that large portions of the memory and gates are doing very little useful work. This is what Beals et al. [7] decided about single-element memory access, and set out with the loftier goal of a multi-element memory access circuit $U_{N,N}$, with action:

$$\begin{aligned} &|j_1\rangle \cdots |j_N\rangle |y_1\rangle \cdots |y_N\rangle |x_1\rangle \cdots |x_N\rangle \\ \mapsto &|j_1\rangle \cdots |j_N\rangle |y_1 \oplus x_{j_1}\rangle \cdots |y_N \oplus x_{j_N}\rangle |x_1\rangle \cdots |x_N\rangle . \end{aligned} \tag{4.3}$$

The key to their approach is that since we can apply gates simultaneously to all qubits, we can turn them into a sorting network. That is, each memory cell can compare itself to a neighbouring cell, and they will swap if need be. The depth of such a sorting network depends on how many "neighbours" a qubit has. Ignoring physical layout and letting each qubit connect to every other qubit leads to $O(\log N)$ depth.

We can also achieve $O(\log N)$ depth with "hypercube" connectivity. This means each qubit has $O(\log N)$ neighbours. If the neighbours are within a bounded physical distance, then the number of cells within graph distance D grows exponentially with D - but the physical space they can occupy only grows as D^d in d-dimensional space. Hence, the density of cells must increase exponentially to fit into Euclidean space, which is not realistic.

In a two-dimensional grid, a sort has depth $O(N^{1/2})$; we assume that in d dimensions a sort has depth $O(N^{1/d})$.

Figure 4.3 shows the memory access. First we make tuples $(i, d_{out}, d_{in}, \mathtt{b})$, where $\mathtt{b}$ is a boolean flag indicating either "query" ($\mathtt{0}$) or "answer" ($\mathtt{1}$). The registers d_{out} and d_{in} store data inputs and outputs, and i stores the memory address.

Each index register $|j_i\rangle$ means we want to query the data in location j_i and `XOR` it with the data y_i. Thus, we format a "query" tuple $(j_i, y_i, \mathtt{0}, \mathtt{0})$ for each register $|y_i\rangle$.

Each data register $|x_i\rangle$ may need to be an answer to a query, so we format each one into an "answer" tuple $(i, \mathtt{0}, x_i, \mathtt{1})$.

Then we sort, based on the following order: First sort based on the address register, and if multiple tuples have the same address, sort on the anser/query flag. After the sort, we will end up with data ordered as:

$$\cdots |(j_k, y_k, \mathtt{0}, \mathtt{0})\rangle \, |(i, \mathtt{0}, x_i, \mathtt{1})\rangle \cdots \tag{4.4}$$

where $j_k = i$. Then the data can simply be copied from d_{in} of the answer tuple to d_{out} of (potentially several) query tuple(s). Then the sort is reversed, putting everything back where it is supposed to be.

There are more complications to do this properly, but we will skip these and focus on the cost of this circuit. We follow Beals et al. [7] and use the comparator circuit of [56], which compares m-bit strings using a binary tree of $O(m)$ gates, $O(\log m)$ depth, and $O(m)$ width. We assume m is small enough that $\log m$ dominates any signal propagation time.

The sorting step dominates the costs for all metrics. We use their proof of Theorem 5 [7] to get the full costs. For locality and latency, we assume that the d-dimensional case is limited to a d-dimensional lattice connectivity, such that a sorting network takes $O(N^{1/d})$ depth and $O(N^{1+1/d})$ comparisons.

A sort necessarily requires $O(\log N)$ ancilla qubits to store a bitstring which represents the permutation. Without these ancillae, sorting is obviously irreversible. It's unclear how many ancillae a specific sort will need. For example, Cheng and Wang [20] describe a quantum merge sort based on quantum comparators, each of which performs a single swap but requires its own ancilla qubit. This would make the number of ancillae equal to the

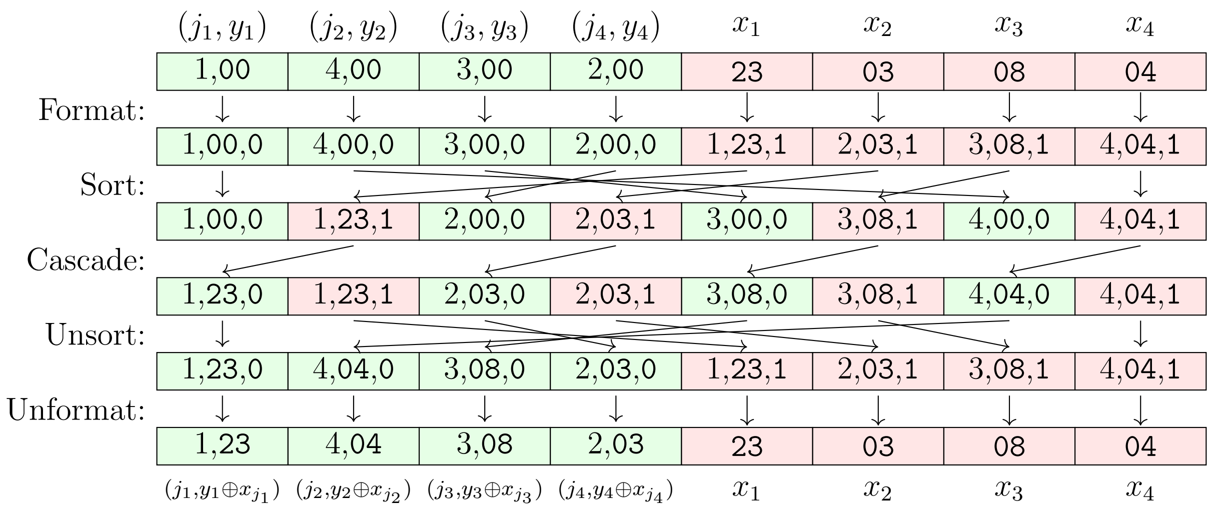


Figure 4.3: Multi-address memory access with a sorting network. Only one of d_{in} and d_{out} is shown.

number of comparisons, which is what Beals et al. assume [7]. This is far from optimal, but without a method to “compress” the ancillae, we will use the same conclusion.

This means a d-dimensional sorting network needs $O(N^{1+1/d})$ ancillae. Each processor now has a latency of $O((N^{1/d}+m)^{1/d})$ to send its data to a neighbour to do the comparison. Arguably, we could simulate a higher connectivity with a series of predetermined swaps between adjacent qubits. This would not improve the total depth, but it would reduce the number of comparisons and hence the number of ancillae. We will not analyze this approach.

Cost 4.3 shows the cost of a single sort, which is asymptotically the same as a sorting network memory access.

Cost 4.3 Sorting N elements of m bits each

Model	Measure	Cost
Passive Circuit (2.4.1)	Gates	$O(Nm \log N)$
Active Circuit (2.4.2)	Total	$O(N \log N \log m(m + \log N))$
	Depth	$O(\log N \log m)$
	Width	$O(N(m + \log N))$
Passive Latency (2.4.3)	Gate-Time	$O(N^{1+1/d}(m + N^{1/d^2}))$
Active Local (2.4.4)	Total	$O(N^{1+2/d+1/d^2}) + o(N^{1+2/d+1/d^2})$
	Depth	$O(N^{1/d}(\log m + N^{1/d^2}))$
	Width	$O(N(N^{1/d} + m))$
QRAM (2.4.5)	Gates	$O(N \log N(m))$
	Time	$O(N \log N(Nm)^{1/d})$

Comparing fanout to a sorting network, we see that fanout is simpler and costs less. However, the sorting network is much more powerful, since it can perform N simultaneous memory accesses. As Beals et al. [7] put it:

> There is a sense in which the gates in a typical circuit for [single memory access] can be said to be "not working very hard" (although this idea is hard to quantify precisely), and this inefficiency points to the need for a parallel algorithm.

Physical layout, and hence reduced connectivity, makes the sorting network noticeably more expensive. This may be more palatable for particular applications.

4.2 Quantum Data Structures

A sorted list is a basic building block of many classical algorithms. It is also essential to quantum random walks on graphs whose vertices are sets.

In our paper [34], we used the requirements that Jeffery set out [35]. The most basic way to define a quantum sorted list is as follows:

Definition 4.2.1. *A quantum sorted list data structure for a set X is a map from subsets $S \subseteq X$ to unique states $|S\rangle$, along with a sequence of unitaries $U_I(N)$ and $U_L(N)$ parametrized by an integer N, that satisfy the following properties for all $x \in X$ and all $S \subseteq X$ with $|S| = N$:*

Look-up: $U_L(N)\left|S\right\rangle\left|x\right\rangle\left|0\right\rangle = \begin{cases} \left|S\right\rangle\left|x\right\rangle\left|1\right\rangle & ,x\in S \\ \left|S\right\rangle\left|x\right\rangle\left|0\right\rangle & ,x\notin S \end{cases}.$

Insertion: $U_I(N)\left|S\right\rangle\left|x\right\rangle = \left|S\cup\{x\}\right\rangle\left|x\right\rangle$.

A popular classical approach uses a linked list that implements a binary tree. That is, for some total ordering of X, we store S as a set of memory cells, each containing an element x and pointers to other nodes in the tree. We ensure that the pointer to the "right" points to a tree of elements y such that $y > x$, and that the pointer on the "left" points to a tree of elements y such that $y \leq x$.

In many classical applications, we leave the particulars of the memory layout to to the system. It's largely irrelevant which particular RAM address the nodes happen to occupy. In a quantum algorithm we do not have this luxury. Two different lists, $\left|S\right\rangle$ and $\left|S'\right\rangle$, storing the same *set* S, but in a different order, will not interfere properly: $\left|S\right\rangle - \left|S'\right\rangle \neq 0$. For quantum walks and many other algorithms we need interference, hence why we require a *unique* state $\left|S\right\rangle$.

This is known as "history independence" and it was known to Ambainis [5] who solved it with a hash-table and skip-list. Bernstein, Jeffery, Lange, and Meurer [9] introduced the quantum radix tree, which Jeffery [35] expanded on. Jaques and Schanck [34] simply translated a classically history independant data structure, a dynamic sorted array, to the quantum setting. There we called it a "Johnson Vertex", after our main application, but in this thesis we call it a "sliding sorted array" for more generality.

The quantum radix tree takes a history dependent data structure, and uses a uniform superposition over all possible layouts, while the hash-table and skip-list uses a randomized but history independent layout with poor worst-case performance, then uses a superposition of possible hash functions to approach the average-case performance.

By the same logic as Theorem 2.3.1, insertion, deletion, and lookup will require $\Omega(N)$ gates. The quantum radix tree exceeds this bound by a logarithmic factor, but a sliding sorted array meets it.

For the claw-finding random walks in Chapter 6, we need to support three other operations: selection, claw check, and complement sampling [34]. We assume that we have two sets, X and Y, with a quantum data structure on each, and functions $f : X \to S$ and $g : Y \to S$. Then we require the following circuits:

Selection $U_S(N)$: For any set $S \subseteq X$ with $|S| = N$,

$$U_S(N)\left|S\right\rangle = \frac{1}{\sqrt{N}}\sum_{x\in S}\left|S\right\rangle\left|x\right\rangle . \tag{4.5}$$

Claw check $U_C(N_1, N_2)$: For any two sets $S_1 \subseteq X$ and $S_2 \subseteq Y$ with $|S_1| = N_1$ and $|S_2| = N_2$,

$$U_C(N_1, N_2) |S_1\rangle |S_2\rangle |0\rangle = \begin{cases} |S_1\rangle |S_2\rangle |1\rangle & , S_1, S_2 \text{ have a claw} \\ |S_1\rangle |S_2\rangle |0\rangle & , \text{otherwise.} \end{cases}, \tag{4.6}$$

where S_1 and S_2 "have a claw" if there is some $x \in S_1$ and $y \in S_2$ such that $f(x) = g(y)$.

Complement sampling $U_K(N)$: For any set $S \subseteq X$ with $|S| = N$:

$$U_K(N) |S\rangle = \frac{1}{\sqrt{|X| - N}} |S\rangle \sum_{x \in X \setminus S} |x\rangle . \tag{4.7}$$

We also need Selection and Complement sampling circuits for Y as well.

Jaques and Schanck introduced complement sampling [34]. Previous approaches did not include this; in a query model, this can be done in the naive way by iterating over the entire data structure without using any queries. Jaques and Schanck avoided the issue by showing that sampling from the full set introduces errors small enough to ignore.

Here we will give a method to perform complement sampling directly. What we do is first create a uniform superposition over the full set X:

$$|S\rangle \mapsto |S\rangle \sum_{x \in X} |x\rangle . \tag{4.8}$$

After this, we use the lookup circuit:

$$|S\rangle \sum_{x \in X} |x\rangle \mapsto |S\rangle \sum_{x \in X \setminus S} |x\rangle |0\rangle + |S\rangle \sum_{x \in S} |x\rangle |1\rangle . \tag{4.9}$$

This matches the probabilistic algorithm setup of Section 1.2.4. Here, the "garbage" ancilla could be uncomputed by the inverse of the Selection circuit. Hence, we can use any method listed in Section 1.2.4 to get arbitrarily close the intended action of complement sampling.

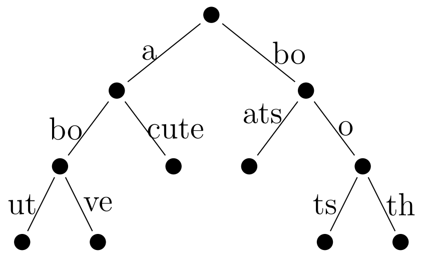
With the assumption of arbitrary superpositions, this can be done with roughly the same cost as a single look-up.

4.2.1 Quantum Radix Trees

We follow the description of quantum radix trees from Jeffery [35]. Our conclusion, from [34], is that a radix tree is less efficient than a sliding sorted array (Section 4.2.2), though we will give explicit details for the radix tree. Many of these details are missing from the literature, and they serve as an illustration of the subtle and difficult issues of reversibly implementing traditional classical algorithms.

Classical Radix Trees

A radix tree is binary tree based on strings. Each edge denotes a substring. A path from the root node to a leaf node stores the string formed by concatenating all the edges in that path. Figure 4.4 gives an example of a classical radix tree, though the radix trees we will consider are assumed to store only binary strings.



(a) The data as a tree.

1: (a,4),(bo,2)
2: (ats,⊥),(o,5)
3: (ut,⊥),(ve,⊥)
4: (bo,3),(cute,⊥)
5: (ts,⊥),(th,⊥)

(b) Memory cells storing the radix tree. "⊥" indicates the end of the string.

Figure 4.4: An example of the modelled layout and the actual memory layout of a radix tree storing the words *about, above, acute, boats, booth*, and *boots.* Note that the memory layout is non-unique for this tree: The root cell is fixed, but there are 4! = 24 possible arrangements of the other 4 cells.

Rather than describe insertion and look-up in full generality, we will simply walk through the example tree.

To look up "acute" in the tree, we would first check the root node, with two outgoing edges, (a,4) and (bo,2). Since "a" is the prefix of "acute", we go to memory location 4 and find the node with edges (bo,3) and (cute,⊥). "cute" matches the remainder of our search string, so we found the correct string.

To insert "actor", we would look up in the same way, and find the cell with edges (bo,3) and (cute,⊥). Since the first character of "cute" matches our input string, we change the edge from (cute,$\perp$) to (c,6), then insert a new node at location 6 with edges (ute,$\perp$) and (tor,$\perp$).

To delete "acute", we would look it up first, then notice that memory location 4 only contains (bo,3) and (cute,$\perp$), so we would need to delete all data in memory location 4 and move "bo" up the tree. We would go back to memory location 1 and combine (a,4) with (bo,3) to become (abo,3). Note that this fragments the memory, since location 4 is empty. We would rely on some sort of garbage collection to eventually fix this.

In short, a radix tree is just a binary tree, but with the information stored in the edges.

Quantum Modifications

To make a radix tree history-independent, we use a superposition over all possible memory layouts. The fragmentation problem is worse now, because classically, a radix tree cannot be fragmented until there have been deletions. This creates a difference in layout between a freshly-initialized radix tree and one produced by several insertions and deletions. Hence, we either need to pre-fragment the quantum radix tree, or prevent fragmentation. Previous literature opts to pre-fragment by choosing a random memory location for insertion.

This means we need some method to store a list of the empty cells. Here we propose storing this tree within the empty nodes themselves.

The left half of Figure 4.5 shows how this might look. Since memory cells are not "empty", we will use "data-free" to refer to a cell that does not contain the useful data. It may either be blank or contain a node of the tree which tracks data-free cells. The data-free-cell tree will store memory locations as binary strings. In Figure 4.5, we have entire substrings for the edges at each node, rather than the next available substring as in a classical radix tree. Section 4.2.1 explains this.

A radix tree needs $N-1$ memory cells to store N strings. Hence, if it is allocated to store N strings but currently holds only n, there will be $N-n$ data-free cells. Then $N-n-1$ of these cells will store the tree of data-free cells, meaning there is one completely empty cell.

Insertion

Inserting a new element x into the tree will require a new data node. Whatever the address of this node, we will need to delete this address from the data-free-cell tree. This will remove one node from the data-free-cell tree, which is the node we want to use for the data tree. We need to ensure that if we delete the node at address i from the data-free-cell tree, that we also delete the *string* "i" from the content of the tree. These will not always match.

To make this work, we first select the random memory address i from the data-free-cell tree. To do this, we start at the root of the tree, then randomly select a branch, weighted by the number of nodes under each branch. Hence, we need to append a "size" to each node of the tree. In Figure 4.5 the root, cell 12, has a left branch of size 3 and a right branch of size 4. Thus, we want to follow the left branch with probability 3/7 and the right branch with probability 4/7.

To actually accomplish this, we would initialize a superposition

$$\frac{1}{\sqrt{N-n-1}} \sum_{\mathtt{b}=0}^{N-n-1} |\mathtt{b}\rangle \tag{4.10}$$

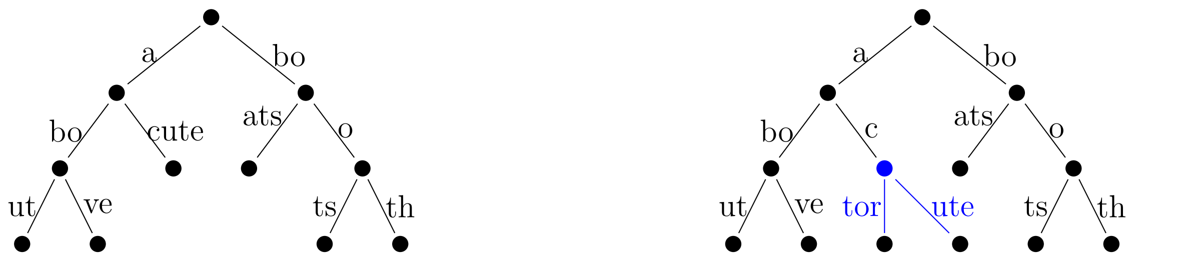
and compare `b` to the size of the left branch. If `b` is greater, we would subtract the size of the left branch from `b`, take the right branch, and recurse. If `b` is smaller, we would take the left branch and recurse. This will select a random element of the data-free-cell tree with uniform probability. This will be a memory location; call it i.

Since our goal is for i to contain data, we need to delete the string "i" from the data-free-cell tree. This alters the structure of the data-free-cell tree; in particular, we will need to delete some node. Let j be the memory location of this node. Once this node is deleted, we swap memory location i with memory location j, updating the appropriate pointers. This means i will be blank, so we can add the new data node to location i and update the tree as necessary.

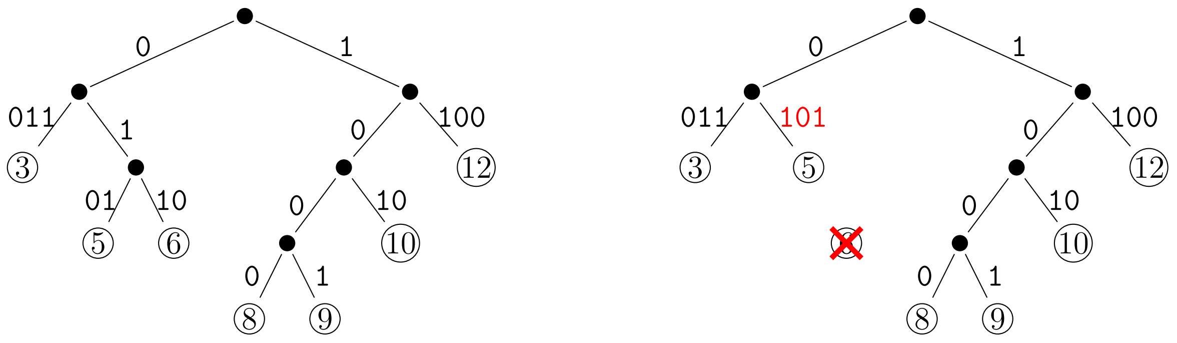
Figure 4.5 shows this process. In the figure, the new element x is "actor". We randomly chose $i = 6$. To delete the string "6" from the data-free-cell tree, memory location 10 must change so that the substring "`101`" terminates, meaning we will need to delete memory location 3 from the data-free-cell tree. Once it is deleted, we move the data in memory location 6 to memory location 3, which means we need to update the pointer in the parent of the node at address 6, which in this case is at address 5. Then we can save the new data node in location 6.

Deletion is just the inverse of insertion.

Assuming the objects in the tree have m bits, this process involved, at worst $O(m)$ memory accesses, comparisons, and additions. As in Section 4.1.1, m-bit comparisons use $O(\log m)$ depth, $O(m)$ gates and $O(m)$ width. Takahashi, Tani, and Kunihiro [54] provide



(a) The data tree.



(b) The data-free-cell tree. Leaf labels are for ease of interpretation only.

	Left	Right
1.	**(a,2)**	**(bo,4)**
2.	**(abo,7)**	**(acute,⊥)**
3.	(0101,⊥)	(0110,⊥)
4.	**(boats,⊥)**	**(boo,11)**
5.	(10,6)	(1100,⊥)
6.	(100,8)	(1010,⊥)
7.	**(about,⊥)**	**(above,⊥)**
8.	(1000,⊥)	(1001,⊥)
9.	blank	
10.	(0011,⊥)	(01,3)
11.	**(boots,⊥)**	**(booth,⊥)**
12.	(0,10)	(1,5)

	Left	Right
1.	**(a,2)**	**(bo,4)**
2.	**(abo,7)**	**(ac,6)**
3.	(100,8)	(1010,⊥)
4.	**(boats,⊥)**	**(boo,11)**
5.	(10,3)	(1100,⊥)
6.	**(actor,⊥)**	**(acute,⊥)**
7.	**(about,⊥)**	**(above,⊥)**
8.	(1000,⊥)	(1001,⊥)
9.	blank	
10.	(0011,⊥)	(0101,⊥)
11.	**(boots,⊥)**	**(booth,⊥)**
12.	(0,10)	(1,5)

(c) Memory cells as tuples of (substring, memory location), for edges to the right and left. "⊥" indicates the end of the string. Data is in bold.

Figure 4.5: Left: A radix tree with data and data-free-cells. For this data, the tree layout is fixed, but the memory layot is not. Right: Inserting the string "actor" into the tree. Changes to the data are shown in blue and changes to the data-free-cells are shown in red.

a circuit for m-bit addition with $O(\log m)$ depth and $O(m)$ gates. Using the fanout memory access gives Cost 4.4.

Cost 4.4 Quantum radix tree insertion

Model	Measure	Cost
Passive Circuit (2.4.1)	Gates	$O(Nm^2)$
Active Circuit (2.4.2)	Total	$O(Nm^2 \log(N)$
	Depth	$O(m \log(N))$
	Width	$O(Nm)$
Passive Latency (2.4.3)	Gate-Time	$O(Nm^2)$
Active Local (2.4.4)	Total	$O((Nm)^{1+1/d} m \log(Nm))^d m)$
	Depth	$O((Nm)^{1/d} m)$
	Width	$O(Nm)$
QRAM (2.4.5)	Gates	$O(m^2)$
	Time	$N^{1/d} m$

Initialization

We can initialize a quantum radix tree with a Knuth shuffle (see Section 4.3.3). The total cost is

$$O(N(\mathsf{C}_{RAM} + \mathsf{A})). \tag{4.11}$$

where C_{RAM} is the cost of a random access and A is the cost of m-bit addition. Combining these with the standard random access costs and addition costs gives Cost 4.5.

Cost 4.5 Initializing a quantum radix tree with a Knuth shuffle

Model	Measure	Cost
Passive Circuit (2.4.1)	Gates	$O(N^2 m)$
Active Circuit (2.4.2)	Total	$O(N^2 m \log N)$
	Depth	$O(N \log N)$
	Width	$O(Nm)$
Passive Latency (2.4.3)	Gate-Time	$O(N^2 m)$
Active Local (2.4.4)	Total	$O(N^{2+1/d} m^{1+1/d} \log(Nm))^d)$
	Depth	$O(N(Nm)^{1/d})$
	Width	$O(Nm)$
QRAM (2.4.5)	Gates	$O(Nm)$
	Time	$O(N^{1+1/d} m^{1/d})$

Augmented Radix Trees

For claw-finding, we have functions $f : X \to S$ and $g : Y \to S$, and we want to be able to find if there is some x in our list such that $f(x) = z$, for some input string z. We will need a similar structure for both X and Y but we will focus on X. To do this, the tree will store strings of $(f(x), x)$. The function value $f(x)$ is the prefix of this string so that we can search for z in the tree, and simply stop after the length of z. Further, since each node stores the weight of the subtree underneath itself, once we find $f(x)$ we can read the weight to learn how many $x \in S$ have this value of f.

The following method is Schanck's take on Jeffery's data structure [51, 35]. For a state $|S_1\rangle |S_2\rangle$ storing subsets of X and Y, respectively, we will keep another register $|C\rangle$, which stores a claw counter:

$$C = |\{(x, y) \in S_1 \times S_2 \,|\, f(x) = g(y)\}| . \tag{4.12}$$

This counts unordered pairs that form claws. This means if we have three elements x_1, x_2, and y such that $f(x_1) = f(x_2) = g(y)$, we count this as 2 claws.

To maintain this counter, when we insert an element x into S_1, we need to check if $f(x)$ matches any $g(y)$ in S_2. This is a straightforward query. If there is a match, we add the number of such y to the claw counter. This is where we use the weight of the node with the prefix $g(y)$.

This adds some complexity to the insertion operation. However, this is only $O(\log m)$ memory accesses and $O(\log m)$ additions, which the radix tree already needed to do, so the asymptotic complexity of insertion, Cost 4.4, will not change.

Selection: Since the tree includes the "weight" of each node (the number of strings stored in that branch), it is straightforward to use this weight to select random nodes in superposition, just as we did to randomly insert.

Check: With the claw counters, a check is a single comparison: We look at the claw counter of the root node, and check if it is equal to 0 or not.

4.2.2 Sliding Sorted Arrays

A classical *dynamic sorted array* is an array of elements that are physically in order, meaning the ith element in the array is the ith largest element (or smallest, depending on the ordering used). This is clearly history independent, and search is fast, but inserting a new element means potentially moving all elements in the array. Classically, this is tremendously costly.

For a quantum computer, Theorem 2.3.1 means we are already prepared to spend a linear number of gates for insertion. In the sliding sorted array, insertion into an n-element array can require moving up to n elements, but each element moves only 1 "cell". Hence, this can actually be implemented with a low gate cost. In the QRAM model, it's the same issue as classical computers, so the cost is enormous; we will not consider the QRAM model for sliding sorted arrays.

For this section we assume that the elements are arranged in non-increasing order. The details come from our paper [34], though there we called this structure a "Johnson vertex", because we limited ourselves to lists without duplicates and focused only on applications to Johnson graphs. However, the structure works more generally. We will use "cell" to refer to a specific piece of data stored in the array, and any extra data it requires for the array to function.

Initialization

To initialize we only need to set every cell to a specified bottom string $\perp$, which is defined so that $x \geq \perp$ for all x that we may want to insert in the array. For later steps we require $\perp$ to be the all-zeros string.

Insertion and Deletion

Here we will describe the full steps for insertion, following [34]. We are inserting an element x into an array A which contains elements $a_1, \cdots, a_R$. We will require two ancilla arrays A' and A'' both initialized to `0`; A' stores cells of the same length as A, and A'' stores binary cells. Figure 4.6 shows the insertion process.

Fan-out: The first step uses a fan-out circuit (as in Equation 4.1) to fill A' with x, Figure 4.6a. We also use `CNOT` gates to copy x to the final, empty cell in A; we explain why in the swap and un-fan-out steps.

Compare: We compare every cell $A[i]$ in A to the corresponding $A'[i]$, and save the boolean evaluation of "$A[i] \leq A'[i]$" to $A''[i]$, Figure 4.6b. Since every $A'[i]$ is x, this means $A''[i] = (A[i] \leq x)$.

Swap: The bits in A'' are used as controls for two swaps: one diagonal swap and one swap straight down, Figure 4.6c and 4.6d. This moves all the necessary elements to the right by one cell, and puts x in the new cell that this opens in the middle of the array.

Because the diagonal swap will not change the last element of the array, Figure 4.6 shows how, if we had not copied x over the bottom string in the fan-out step, we would end up with the bottom string in $A'[N]$ after this step.

Un-compare: We do the same comparion as the compare step, and this will clear the comparison array A'', Figure 4.6e. It was necessary to use $\leq$, so that the newly inserted x will still clear the comparison bit above itself.

Un-fan-out: Starting from the state shown in Figure 4.6f, we invert the fanout circuit to clear A'. This finishes the insertion. Here it's clear that the final element of A' must be x to be properly cleared.

We assume that we build this circuit such $A[i]$, $A'[i]$, and $A''[i]$ are all physically close, so that the swap step does not have any latency or locality concerns. Using Cost 4.1 and the cost of a comparison, we give the full sliding sorted array insertion cost as Cost 4.6. This is noticeably less expensive than the radix tree. Memory access is "non-local", in the sense that the input is physically far from everywhere it needs to be for the computation. Hence for sorted lists, the expensive operation is the non-local memory access, and the sliding sorted array saves cost by only doing 2 non-local fanouts, and using local operations for the rest.

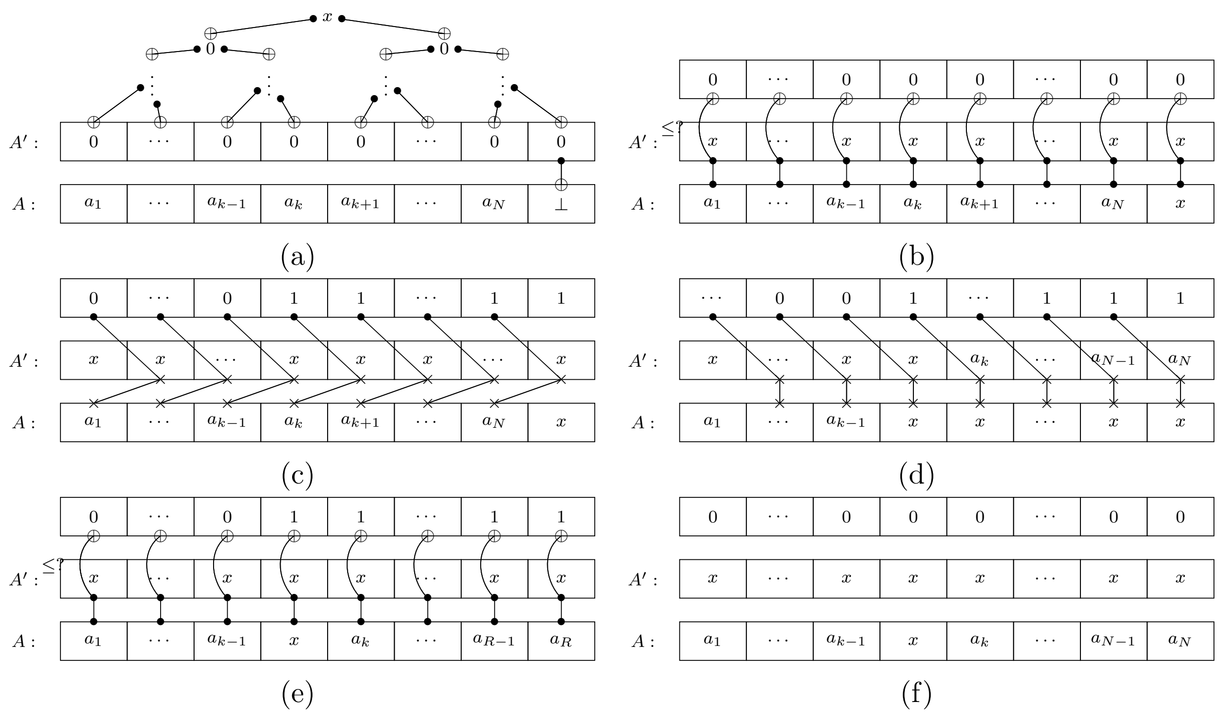


Figure 4.6: Insertion into a sliding sorted array A, from [34]. See text for full description.

Cost 4.6 Insertion into a sliding sorted array

Model	Measure	Cost
Passive Circuit (2.4.1)	Gates	$O(Nm)$
Active Circuit (2.4.2)	Total Depth Width	$O(Nm \log N)$ $O(\log N)$ $O(Nm)$
Passive Latency (2.4.3)	Gate-Time	$O(Nm)$
Active Local (2.4.4)	Total Depth Width	$O((Nm)^{1+1/d}\log(Nm))^d)$ $O((Nm)^{1/d})$ $O(Nm)$

This data structure works perfectly well with repeated elements (i.e., to store a multiset). If x is already in the array, a newly inserted x will always be inserted into the front.

This means that the comparator must use full string equality to check $\leq$, or else we lose history independence.

Search

Search is essentially the same as insertion. For an element x, we still need ancilla arrays A' and A'' initialized to 0. The steps are as follows:

Fanout: Fanout x to A', Figure 4.7a.

Compare: For all i, set $A''[i]$ to the result of $A[i] == A'[i]$, Figure 4.7b.

`OR`-tree: We use a binary tree of logical `OR` gates to find the logical `OR` of all elements of A'', Figure 4.7c. A quantum `OR` requires one ancilla, initialized to 0, which will store the result. Hence each layer of the `OR`-tree will need new ancillae, for a total of N new ancillae bits. Once the `OR`-tree is finished, the final result is copied to some output register, then the tree is uncomputed.

Uncompute: Reverse the comparison and fanout circuits.

This has the same cost as insertion. An interesting point here is that the search does not require the array to be sorted. Sorting the array is actually only necessary for history independence.

Augmented Sorted Arrays

Augmenting a sliding sorted array follows many of the same methods as the radix tree. The cells will now store tuples $(x, f(x))$ or $(y, g(y))$. We also add a counter the start of the array which counts the number of unordered pairs that form claws, the same as Equation 4.12. From now on, assume we have two parallel arrays $S_1 \subseteq X$ and $S_2 \subseteq Y$.

Selection: Selection is easy, since the classical controller knows the size N of the array. It produces a superposition of integers up to N, and uses these as input to a fanout memory access.

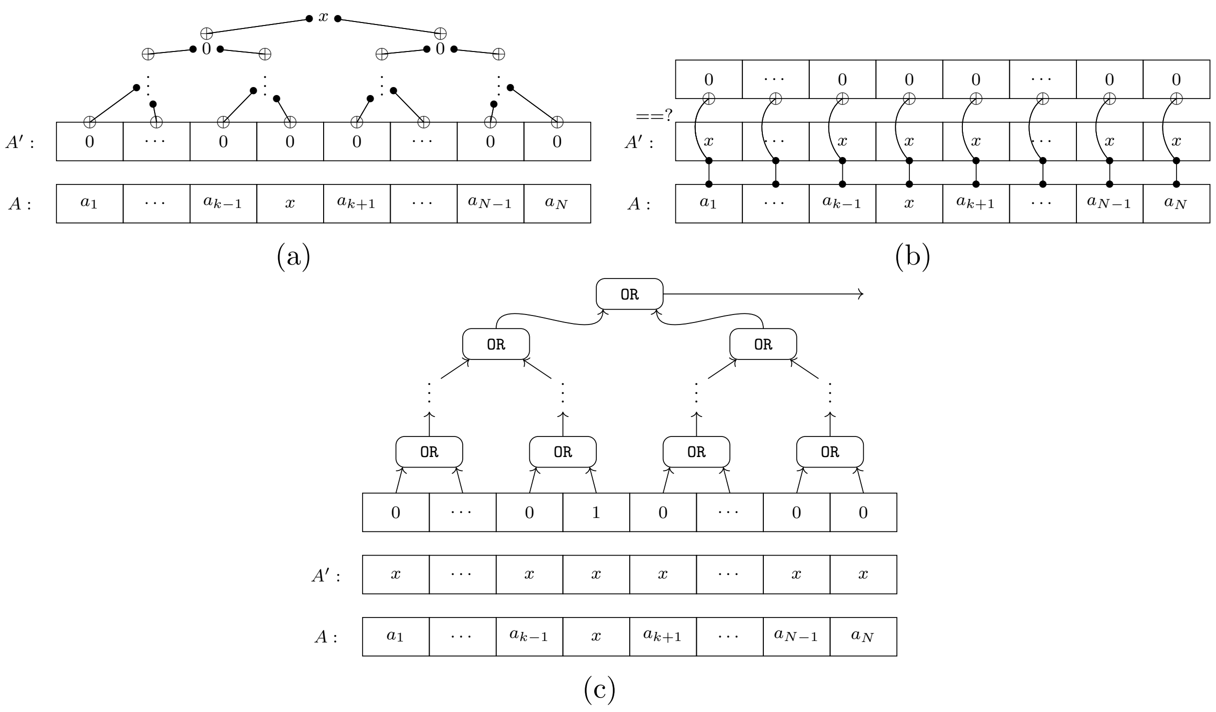


Figure 4.7: Searching into a sliding sorted array A. See text for full description

Claw detection: If we properly maintain the claw counter, we can simply check that counter to decide if the pair (S_1, S_2) contains a claw. To maintain the counter, we augment the insertion operation.

When inserting an element x into S_1, we count the number of $y \in S_2$ such that $f(x) = g(y)$. The following steps will accomplish this, using two parallel arrays A' and A'', although now we require A'' to have $\log N$ bits, where N is the size of the array.

Fanout: Fanout $f_1(x)$ to A'.

Compare: Compare $A'[i]$ to $A[i]$ for all i. In particular, since $A[i] = (y_i, g(y_i))$, we only compare to $g(y_i)$. The result is saved in A'' *as an integer.*

Sum: Use a binary tree of addition circuits to add up all elements of A''. An in-place addition circuit will map (a, b) to $(a, a + b)$ [54]. A tree of such circuits can add up all N numbers in depth $O(\log^2 N)$, since each integer has $\log N$ digits, and this tree will require $O(N \log N)$ gates. The result is added to the claw counter.

We apply this counting circuit with the roles of S_1 and S_2 reversed when we insert an element into S_2.

Performing this count dominates the insertion cost and gives Cost 4.7. We assume $\log N$ is in $O(m)$, otherwise there are fewer possible strings to insert into the array than cells allocated in the array. When accounting for latency, we assume the $N^{1/d}$ terms dominate the extra cost of addition. It is still cheaper than the radix tree by logarithmic factors in all the cost models.

Remark 4.2.1. *In the tree of integer addition, we know that in the ith layer, the results will have at most $i+1$ bits. Hence, we can use smaller addition circuits in the first layer, which requires the most addition circuits. Unfortunately, this optimization has no asymptotic impact on the depth or gate count.*

Cost 4.7 Insertion into an augmented sliding sorted array

Model	Measure	Cost
Passive Circuit (2.4.1)	Gates	$O(Nm)$
Active Circuit (2.4.2)	Total	$O(Nm(\log m + \log^2 N))$
	Depth	$O(\log m + \log^2 N)$
	Width	$O(Nm)$
Passive Latency (2.4.3)	Gate-Time	$O(Nm)$
Active Local (2.4.4)	Total	$O((Nm)^{1+1/d} \log^d(Nm)))$
	Depth	$O((Nm)^{1/d})$
	Width	$O(Nm)$

Physical Layout

For geometric models, there is some question about the layout of this array. Figure 4.6 shows it as a linear list, but we would like a d-dimensional memory array.

Considering the various processes, the primary requirement is that sequential elements are physically next to each other. Many layouts have this property; in 2-dimensions, we could simply lay out the memory in a zig-zag pattern.

This makes the comparison steps local and fast. For the fanout, we are already prepared to spend $O(N^{1/d})$ depth and $O(N)$ gates, so we could either find some way to build a depth-

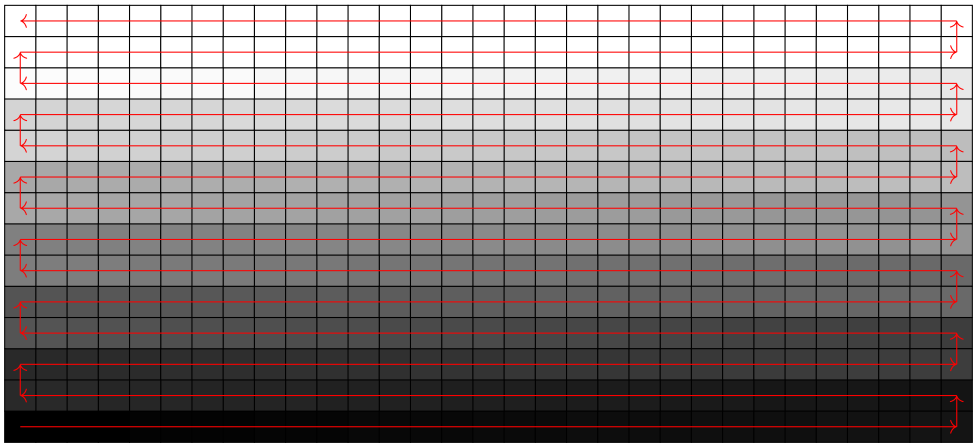


Figure 4.8: An example of the physical layout of the elements in a sliding sorted array, sorted from lightest to darkest.

optimal tree, or we could do it the naive way like Figure 4.9. Both will have the same asymptotic costs.

4.3 Johnson Graphs

There are a few special tricks to help with walks on Johnson graphs. Recall Definition 3.3.1, which states that two vertices u and v are adjacent if they differ by exactly two elements, i.e., $|u \setminus v| = |v \setminus u| = 1$.

4.3.1 Symmetric Differences

The states in a random walk are of the form $|v\rangle\,|u\rangle$, with v adjacent to u. Since u is almost exactly the same data as v, we can define

$$\Delta(v,u) := (v \setminus u, u \setminus v). \tag{4.13}$$

There is an easily-computable bijection from states of the form $|v\rangle\,|u\rangle$ to states $|v\rangle\,|\Delta(v,u)\rangle$. Hence, we can simply store the symmetric difference.

We need to store the symmetric difference anyway. For the update step, we start with $|v\rangle\,|\mathtt{0}\rangle$ and we want to construct adjacent vertices. The steps are:

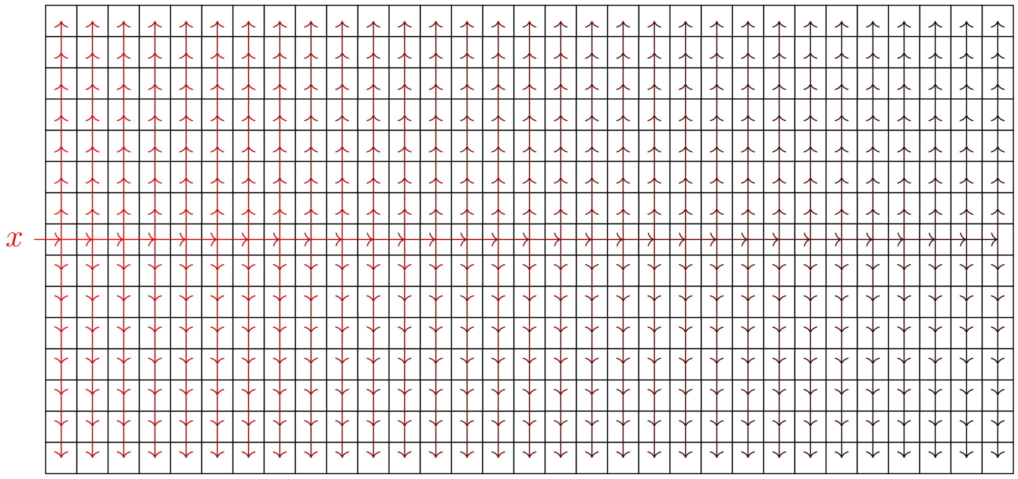


Figure 4.9: Naive fanout in a 2-dimensional array. Arrows indicate a `CNOT`, red is early in the process, black is later.

1. Copy v to the second register: $|v\rangle\,|v\rangle$.
2. Perform a selection and a complement sample on the second register:

$$|v\rangle\,|v\rangle \sum_{x\in X\setminus v} |x\rangle \sum_{y\in v} |y\rangle\,. \tag{4.14}$$

3. Insert x into the second register and delete y from the second register:

$$|v\rangle \sum_{x\in X\setminus v} \sum_{y\in v} |(v\cup\{x\})\setminus\{y\}\rangle\,|x\rangle\,|y\rangle\,. \tag{4.15}$$

The final state can be rewritten:

$$|v\rangle \sum_{x\in X\setminus v} \sum_{y\in v} |(v\cup\{x\})\setminus\{y\}\rangle\,|x\rangle\,|y\rangle = |v\rangle \sum_{u\sim v} |u\rangle\,|\Delta(v,u)\rangle\,. \tag{4.16}$$

If we want to get rid of the symmetric difference, we would need to uncompute it from $|v\rangle\,|u\rangle$, which is straightforward with a sliding sorted array, but unnecessary. We can simply choose to keep $|\Delta(v,u)\rangle$ instead of $|u\rangle$.

Swaps

The walk step requires swapping $|u\rangle$ and $|v\rangle$. The original method, using $|u\rangle|v\rangle$, can easily accomodate swapping by just using bit-wise swaps on every bit. Alternatively, one can simply reverse the action of all the components of the update step.

Using the symmetric difference means we have to change $|v\rangle|\Delta(v,u)\rangle$ to $|u\rangle|\Delta(u,v)\rangle$. For this, let $\Delta(v,u)=(x,y)$. We first delete y from v, then insert x into v, then swap the order of x and y. This does not change which edge is represented, but changes the order in which v and u are represented.

4.3.2 Self loops

For both MNRS and Szegedy, the walk steps are controlled by ancilla so we cannot use the measurement-based garbage collection of Section 1.2.4. Instead of using quantum garbage collection, we could ignore the problem.

Our solution in Jaques and Schanck [34] ignored complement sampling and modified the graph structure to include edges of the form $\{v,v\}$ (called "*loops*"). The design challenge was to ensure that the sets stay the same size and maintain history independence.

We ned to give the self edges a weight R times greater than the other edges. This means

$$p_{uv}=\begin{cases}\frac{1}{R(|X|-R)} & ,u\neq v\\ \frac{1}{X-R} & ,u=v\end{cases}. \tag{4.17}$$

To represent edges, we use $|\Delta(v,u)\rangle=|x,y\rangle$ for edges between distinct vertices, and we set

$$|\Delta(v,v)\rangle:=\frac{1}{\sqrt{R}}\sum_{x\in v}|x\rangle|x\rangle \tag{4.18}$$

to be the state representing a loop. These edges are all still orthogonal.

We defined the edges in this way because it is easy to modify the update operation. The new update does the following:

1. From a state $|v\rangle|0\rangle$, create a uniform superposition of elements of X: $|v\rangle|0\rangle\sum_{x\in X}|x\rangle$.
2. Perform a membership check and save the result in an ancilla:

$$|v\rangle\left(\sum_{x\in X\setminus v}|x\rangle|0\rangle+\sum_{x\in v}|x\rangle|1\rangle\right). \tag{4.19}$$

3. Use the negation of the final ancilla to control a selection circuit:

$$|v\rangle \left(\sum_{x \in X \setminus v} \sum_{y \in v} |x\rangle |y\rangle |\mathsf{0}\rangle + \sum_{x \in v} |x\rangle |\mathsf{0}\rangle |\mathsf{1}\rangle \right). \tag{4.20}$$

4. Copy x to the blank register, controlled by the final ancilla:

$$|v\rangle \left(\sum_{x \in X \setminus v} \sum_{y \in v} |x\rangle |y\rangle |\mathsf{0}\rangle + \sum_{x \in v} |x\rangle |x\rangle |\mathsf{1}\rangle \right). \tag{4.21}$$

5. Perform a membership text on the second-last register to clear the final ancilla:

$$|v\rangle \left(\sum_{x \in X \setminus v} \sum_{y \in v} |x\rangle |y\rangle |\mathsf{0}\rangle + \sum_{x \in v} |x\rangle |x\rangle |\mathsf{0}\rangle \right). \tag{4.22}$$

The left two registers are $\sqrt{p_{vu}} |\Delta(v,u)\rangle$ for all u, including $u = v$. Hence, this is the correct update operation.

Spectral Gap

Loops cause an important change in the random walk algorithms. Let P be the adjacency matrix of the original graph, including normalization. Our new walk essentially decides to take a loop with probability $R/|X|$ and otherwise walks according to P, so the new transition matrix P' will be

$$P' = \frac{R}{|X|} I + \left(1 - \frac{R}{|X|}\right) P. \tag{4.23}$$

This gives us all the eigenvalues of P' in terms of the eigenvalues of P. Since the spectral gap of P is $\delta = 1 - \lambda_2$, we can find the new spectral gap as

$$\delta' = 1 - \lambda_2' = 1 - \left(\frac{R}{|X|} + \left(1 - \frac{R}{|X|}\right) \lambda_2 \right) \tag{4.24}$$

$$= \delta \left(1 - \frac{R}{|X|}\right) \tag{4.25}$$

If we set $R \leq \frac{|X|}{2}$, then we have $\delta' \geq \frac{1}{2}\delta$. Considering Equation 3.45, this change to the graph will at most double the the number of walk steps. For the applications in this thesis, $R/|X|$ is negligible and $\delta' \approx \delta$.

4.3.3 Set-up

The set-up step needs to generate a random subset. It's natural to think we could just generate random elements of X and insert them sequentially, but this runs into two problems:

1. For large R, there is an overwhelming probability of picking duplicate elements at some point;
2. We need to preserve history dependence, but our insertion operation is only $|v\rangle\,|x\rangle \mapsto |v \cup \{x\}\rangle\,|x\rangle$. That is, it keeps the new element as an ancilla, and it *must* do this to be reversible.

Solving the complement sampling problem will solve the first issue but not the second. To solve them both at once, we constructed two new algorithms: A repeated sort-and-measure and a reversible Knuth shuffle.

A Knuth shuffle uses a list A of N random integers to randomly permute a list S of size N. The ith element of A, a_i, is a random integer between i and N; we iterate from $i = 1$ to N and swap $S[i]$ with $S[a_i]$. Note that if we stop after j steps, the first j elements are in their final positions. Thus, to construct a random subset of some set X, we let a_i between a random integer between i and $|X|$, and initialize S to the first N elements of $|X|$. If we want to swap $S[i]$ with $S[a_i]$ for some $a_i > N$, we simply simulate having the extra elements in the array. This requires an ancilla array of size at most N.

Uncomputing the ancilla A is tricky but ultimately the same cost as the original shuffle. If we initialize S in a fixed order, then this is reversible and leaves no ancillae. We will not give the full details here.

To initialize a random list with a "sort-and-measure" technique, we independently construct a superposition of random elements in every cell. This will almost certainly generate duplicates, so we sort the array, check for duplicates, clear the duplicates, then measure the check bits. This tells us where to recreate superpositions. We repeat this until there are no duplicates. The trick in the analysis is to ensure that the measurements leave a *uniform* superposition of the remaining list; again, we omit the full details.

We assume the QRAM model uses the Knuth shuffle to initialize a radix tree, which has cost

$$O(R(\mathsf{C}_{RAM} \log R + \mathsf{A} + \mathsf{F})) \tag{4.26}$$

where F is the cost to compute any associated data with the vertices.

The other models use a sort-and-measure to initialize a sorted sliding array; this requires $O(\log R)$ sorts, $O(R\log R)$ comparisons, and $O(R)$ elements initialized. Note that if the size of each element in the array m is smaller than the number of ancillae that the oracle needs, F_W, then we cannot simultaneously initialize all R elements for the sort-and-measure, if only Rm qubits are available. Thus, we assume that the sort-and-shuffle is used to initialize an array of *integers* with no duplicates, and then these integers are *sequentially* mapped to function values.

This gives Cost 4.8, showing that set-up is roughly a linear cost for passively-corrected models, but quadratic for actively-corrected.

Cost 4.8 The set-up step for a Johnson graph $J(X,R)$.

Model	Measure	Cost
Passive Circuit (2.4.1)	Gates	$O(Rm\log^2 R + R\mathsf{F}_G)$
Active Circuit (2.4.2)	Total	$O(R\mathsf{F}_D(Rm+\mathsf{F}_W))$
	Depth	$O(R\mathsf{F}_D)$
	Width	$O(Rm+\mathsf{F}_W)$
Passive Latency (2.4.3)	Gate-Time	$O(R^{1+1/d}\log R(m+R^{1/d^2})+R\mathsf{F}_G)$
Active Local (2.4.4)	Total	$O(R^{2+1/d}\mathsf{F}_D\log^d(R^{2+1/d}\mathsf{F}_D))+o(R^2)$
	Depth	$O(R^{1/d}\log R(\log m+R^{1/d^2})+R\mathsf{F}_D))$
	Width	$O(R(R^{1/d}+m)+\mathsf{F}_W)$
QRAM (2.4.5)	Gates	$O(R(\log R+m+\mathsf{F}_G))$
	Time	$O(R(R^{1/d}\log R+m+\mathsf{F}_D))$

Chapter 5

Isogenies

Here we introduce two cryptographic protocols, Supersingular Isogeny-based Diffie-Hellman (SIDH) and Supersingular Isogeny-based Key Encapsulation (SIKE). These are intended to be safe against attack from both quantum and classical computers. Section 5.1 gives a basic outline of the protocols, just enough to outline some of the naive attacks in Section 5.2. This frames the attack as a "claw-finding" problem, which Chapter 6 will analyze.

Section 5.3 gives estimates for the costs of quantum and classical computers to compute isogenies. We use these costs in later chapters to give costs for attacks on SIDH and SIKE. The quantum costs are based on previous work on elliptic curve arithmetic on a quantum computer, while the classical costs come from a specification for SIKE.

5.1 Isogeny-based Cryptography

We assume the reader is familiar with the basics of elliptic curves. We denote elliptic curves as

$$E : y^2 = x^3 + ax + b \tag{5.1}$$

for $a, b \in \mathbb{F}$. By "E" we mean the equation. We use $E(\mathbb{F})$ to denote the set of all pairs $(x, y) \in \mathbb{F}^2$ that satisfy the equation, plus the point at infinity $\mathcal{O}$.

Since elliptic curves are groups, they can have group homomorphisms between them. Isogenies are a special kind of homomorphism between elliptic curves, and *separable* isogenies are a special kind of isogeny and the only kind we consider. As a homomorphism, an isogeny's kernel is a subgroup. This uniquely defines the isogeny:

Theorem 5.1.1 (Velú)**.** *Let E be an elliptic curve and let G be a finite subgroup of $E(\mathbb{F})$. Then there is an isogeny with G as its kernel. We denote this isogeny as $^{\phi}/_{G}$, and the image curve as $^{E}/_{G}$. For separable isogenies, the isogeny and image curve are unique up to isomorphism.*

The size of the kernel is called the *degree* of the isogeny. Theorem 5.1.1 comes with an algorithm to compute the action of $^{\phi}/_{G}$:

Theorem 5.1.2 (Velú)**.** *Let E be an elliptic curve and G a finite subgroup of $E(\mathbb{F})$ of size d. There is an algorithm to compute $^{\phi}/_{G}(P)$ for any point $P \in E(\mathbb{F})$ that requires $d-1$ elliptic curve point additions and $d+1$ finite field additions.*

Isogenies add a graph structure to the set of all isomorphism classes of elliptic curves.

Definition 5.1.1. *For a natural number ℓ, the ℓ-*isogeny graph *is the graph formed by:*

Vertices: *All isomorphism classes of elliptic curves*

Edges: *(E_1, E_2) are adjacent if there is a degree-ℓ isogeny from E_1 to E_2.*

For this definition, we consider all isomorphisms and isogenies defined over the algebraic closure.

Every isogeny $\phi : E_1 \to E_2$ has a *dual* isogeny $\hat{\phi} : E_2 \to E_1$, so we can treat the graph as undirected[1]. If ℓ is a prime that does not divide the characteristic of $\mathbb{F}$, then the ℓ-isogeny graph is $\ell + 1$ regular and connected.

5.1.1 Supersingular Isogeny-based Diffie-Hellman

The Supersingular Isogeny-based Diffie Hellman (SIDH, [33]) protocol uses isogenies over a special class of elliptic curves, the *supersingular* curves. The protocol starts with a public elliptic curve E_0, and has public parameters ℓ_A, ℓ_B, e_A, and e_B.

Alice takes a random walk in the ℓ_A-isogeny graph of length e_A, starting from E_0, to reach some curve E_A. Her public key is E_A and her private key is the path she used to reach it. Bob takes a similar random walk of length e_B from E_0 in the ℓ_B-isogeny graph to

[1]Curves with j-invariants of 0 or 1728 are slightly unusual (see [2]) but that does not impact this thesis.

reach E_B. Then Alice does her "same" secret walk starting from E_B, Bob does his secret walk from E_A, and they arrive at the same curve E_{AB}, which is the shared secret.

In general it does not make sense to move a path in a graph from one vertex to another. For SIDH, the parameters of the curve and the paths are chosen in a precise way to make this possible. More specifically:

- The curve is defined over $\mathbb{F}_{p^2}$ for a prime p such that $p = \ell_A^{e_A}\ell_B^{e_B}f - 1$.
- We choose $\ell_A^{e_A} \approx \ell_B^{e_B} \approx p^{1/2}$.
- E_0 has a group structure of:

$$E_0(\mathbb{F}_{p^2}) \cong \left(\mathbb{Z}_{\ell_A^{e_A}}\right)^2 \oplus \left(\mathbb{Z}_{\ell_B^{e_B}}\right)^2 \oplus (\mathbb{Z}_f)^2 \tag{5.2}$$

  where $\mathbb{Z}_n$ is the additive group of integers modulo n.
- All curves with isogenies to E_0 will have same group structure.
- Denoting Alice's secret isogeny as ϕ_A, she must output $\phi_A(P_B)$ and $\phi_A(Q_B)$, where P_B and Q_B generate the subgroup isomorphic to $(\mathbb{Z}_{\ell_B^{e_B}})^2$.

Because Alice's image curve is public, it is easy for an adversary to pick a random path in the isogeny graph, compute the image curve, and compare it to Alice's public key. For a given elliptic curve $E : y^2 = x^3 + ax + b$, we define the j-invariant to be

$$j(E) = \frac{1728(4a^3)}{4a^3 + 27b^2}. \tag{5.3}$$

Two curves are isomorphic if and only if they have the same j-invariant. Hence, it is easy to check if two curves are isomorphic.

5.1.2 SIKE

Supersingular Isogeny-based Key Encapsulation (SIKE) is a specific variant of SIDH that was submitted for standardization [32]. For SIKE, we take $\ell_A = 2$ and $\ell_B = 3$.

Throughout this thesis, "SIKE-n" will refer to an instantiation of SIKE that uses an n-bit prime p.

5.2 Attacks

Jao and de Feo define the following:

Problem 5.2.1 (Computational Super-Singular Isogeny Problem (CSSI), [33])**.** *Given public parameters E_0, ℓ_A, ℓ_B, e_A, and e_B for an instance of SIDH, and a public key E_A, determine a path of length e_A in the ℓ_A-isogeny graph from E_0 to E_A.*

We will assume that breaking SIDH or SIKE is equivalent to solving CSSI.

5.2.1 Naive Attack

For a naive attack on SIKE, we would enumerate all possible paths in the isogeny graph that start at E_0, compute the isogeny they define, and check if the j-invariant of the output curve matches E_A.

There are $\ell + 1$ degree-ℓ isogenies starting from E_0 and each leads to a separate curve. Each of these curves will have another $\ell + 1$ isogenies, but one will be the dual of the first and will go back to E_0. In SIDH, we never go backwards. Thus, the total number of paths of length e is at most $(\ell + 1)\ell^{e-1} \approx \ell^e + \ell^{e-1}$. Since $\ell^e \approx p^{1/2}$, that is how many paths we need to check, illustrated in Figure 5.1.

5.2.2 Meet in the Middle

Instead of checking every path individually, we could instead enumerate all paths of *half* the total length that start from E_0, and all paths of half the total length that start from E_A, shown in Figure 5.2. There should be only one point where they intersect, and this is easy to check with the j-invariant of the image curve.

There will be $(\ell + 1)\ell^{e/2} \approx p^{1/4}$ paths from each curve, so there are only $2p^{1/4}$ paths to check in total. However, we run into a problem: Suppose we have a path p_1 from E_0 that leads to a curve E_1. The only known way to decide if p_1 is the first half of Alice's secret path is to find another path p_2 from E_A that also leads to E_1. But finding this second curve is difficult in its own right.

This is a type of *claw-finding* problem, which Chapter 6 will address.

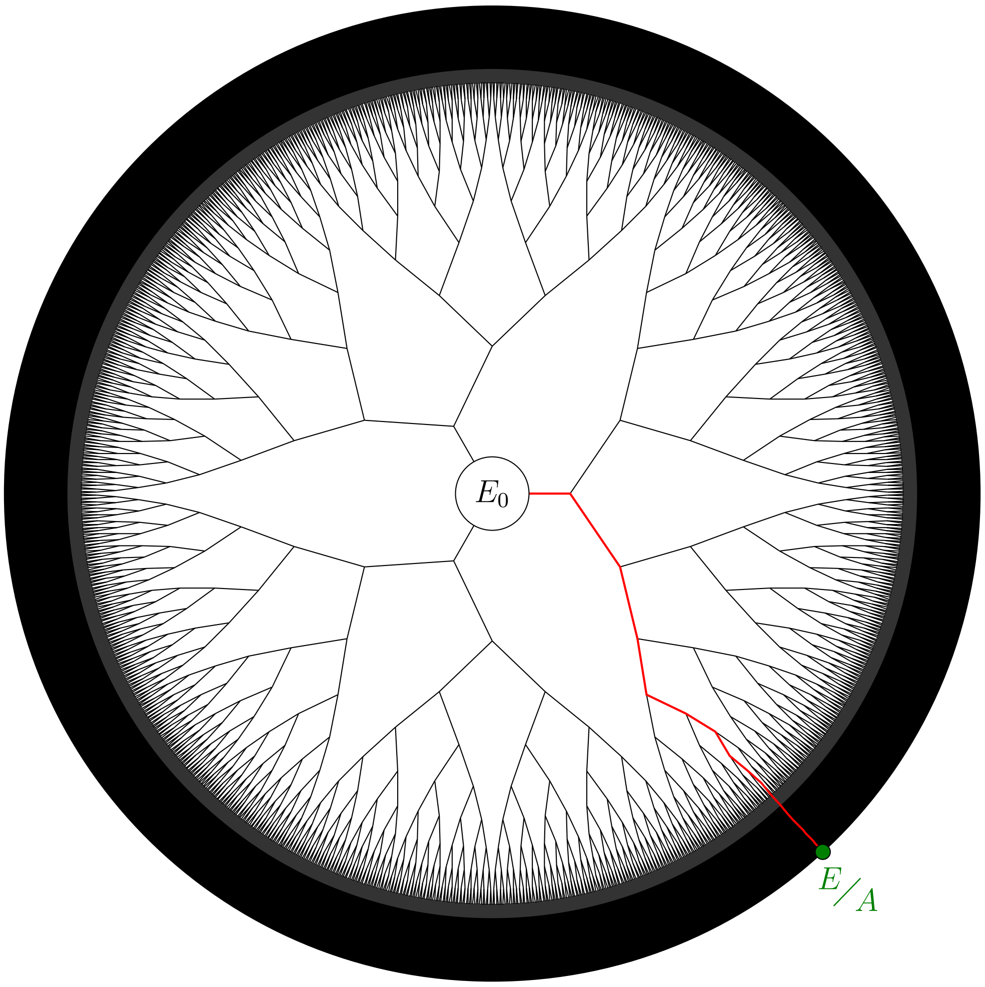


Figure 5.1: The subgraph of the 2-isogeny graph used in SIKE, induced by all paths of length at most 128. In this diagram, the length of each edge decreases exponentially with its distance from E. Vertices are at the intersections between edges. The red path is Alice's secret isogeny, a series of random degree-2 isogenies.

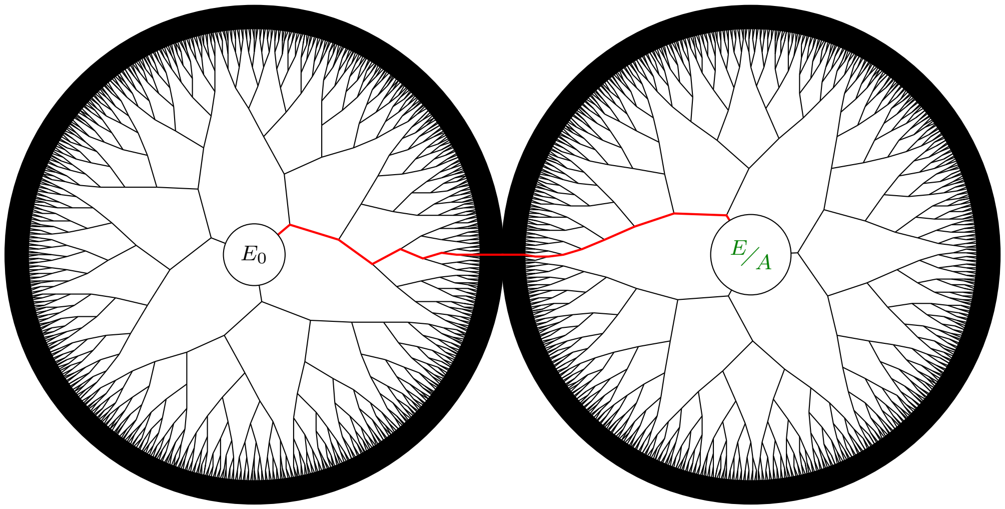


Figure 5.2: Paths of length 64 from E_0 and paths of length 64 from $^{E}/_{A}$, in the style of Figure 5.1. The secret isogeny from E_0 to $^{E}/_{A}$ is in red.

5.3 Isogeny Computations

5.3.1 Quantum Estimates

To estimate quantum costs, we need to know how much it will cost a quantum computer to compute an isogeny. We previously used a very conservative estimate [34]; here, we will use specific figures from specific circuits.

Theorem 5.1.2 states that we need $\ell - 1$ elliptic curve point additions and $\ell + 1$ finite field additions. Roetteler et al. [50] give the cost of a single point addition over a field of an n-bit prime as

$$224n^2 \lg n + 2045n^2 \tag{5.4}$$

gates and $7n + 2\lg n + 9$ qubits. A single finite field addition costs

$$16n \lg n - 26.9n \tag{5.5}$$

gates and uses $2n$ qubits. Thus a degree-2 isogeny has cost

$$224n^2 \lg n + 2045n^2 + 48n \lg n - 54n \tag{5.6}$$

and uses $8n + 2\lg n + 9$ qubits. Roetteler et al. find that the depth is almost exactly the same as the gate count.

In the SIKE specification, they give a method to compute the full isogeny which requires $e \lg e$ point additions, $e \lg e$ isogeny computations, has depth $e \lg e$ and requires storing $\lg e$ points. We know that $e = \frac{1}{2} \lg p$ and that over $\mathbb{F}_{p^2}$, $n = 2 \lg p$.

For concrete cost estimates, we will assemble these costs automatically. The expressions are unwieldy so Cost 5.1 only shows the dominant terms.

Cost 5.1 Quantum isogeny computation

Model	Measure	Cost
	Gates	$448 \lg^3 p (\lg\lg p)^2 + o(\lg^3 p (\lg\lg p)^2)$
Clifford+T	Depth	$448 \lg^3 p (\lg\lg p)^2 + o(\lg^3 p (\lg\lg p)^2)$
	Width	$2 \lg p \lg\lg p + o(p \lg\lg p)$

Is this a good estimate? It might be an under-estimate because Roetteler et al. base their costs on circuits that are tailor-made to add one specific point to an input point, rather than any two inputs. It may be an over-estimate because this was the first paper with a full circuit.

5.3.2 Classical Estimates

Classical methods can be irreversible, which seems to give them an advantage. The SIKE specification [32] gives implementation results in thousands of CPU cycles for various values of the prime p. We will assume that a CPU cycle involves 4 RAM operations, since the CPU used has 4 cores.

The specification gives concrete values instead of asymptotics. Hence, we will assume the cost, in RAM operations, scales according to

$$a_2 \lg^{2.5} p (\lg\lg p) + a_1 \lg p + a_0 \tag{5.7}$$

and extrapolate. This is based on a $O(\lg^{1.5} p)$ cost for $\mathbb{F}_{p^2}$ multiplication. This gives a cost of

$$0.0809 \lg^{2.5} p (\lg\lg p) - 9082 \lg p + 3.27 \times 10^6. \tag{5.8}$$

This is for the reference implementation. A similar approach to the memory used gives

$$76.4 \lg p + 82,527 \tag{5.9}$$

bits of memory.

Chapter 6

Claw Finding

The meet-in-the-middle attack on SIDH can be framed as a *claw finding* problem. This chapter will introduce and analyze several classical and quantum algorithms.

Section 6.1 introduces the problem and show the close connection between claw finding and collision finding. After that, we give two classical algorithms, Meet-in-the-middle (Section 6.2) and van Oorschot-Wiener (Section 6.3), and conclude that van Oorschot-Wiener is cheaper in both RAM operations and run-time.

Following this are three quantum algorithms, Grover's algorithm (Section 6.4), Tani's Algorithm (Section 6.5), and Multi-Grover search (Section 6.6). Using the memory costs from Chapter 4, the latter two algorithms are significantly more expensive than previous analyses.

Section 6.7 compares the algorithms. Though they all scale in the same way with respect to the problem size, Tani's algorithm does slightly better than Grover's algorithm because it saves in oracle computations. Multi-Grover parallelizes better than Tani's algorithm, but requires a more complicated architecture and suffers heavily from latency and locality costs.

Comparing classical and quantum algorithms, the overall pattern is that with large memory, classical algorithms can run faster and cheaper. When memory is limited but time is available, quantum algorithms are cheaper.

6.1 The Claw Finding Problem

Problem 6.1.1 (Claw Finding)**.** *Given two functions $f : \mathcal{X} \to S$ and $g : \mathcal{Y} \to S$, find a claw: A pair $(x, y) \in \mathcal{X} \times \mathcal{Y}$ such that $f(x) = g(y)$.*

Throughout this chapter, X and Y will refer to the sizes $|\mathcal{X}|$ and $|\mathcal{Y}|$.

Claw finding is closely related to collision finding:

Problem 6.1.2 (Collision Finding)**.** *Given a function $h : V \to W$, find a collision: Two distinct elements $v_1, v_2 \in V$ such that $h(v_1) = h(v_2)$.*

Obviously the difficulty of these problems will depend heavily on the functions and the image set; e.g. if the set S or W has two elements, both problems are easy. We distinguish a sub-problem:

Problem 6.1.3 (Golden Claw/Collision Finding)**.** *A claw finding (resp. collision finding) problem is called* golden *if there are $O(1)$ claws (resp. collisions).*

If we assume that f and g are random, there is a simple, imprecise method to compute the expected number of claws or collisions. For claws, consider that any pair (x, y) *could* be a claw, and the probability is $1/|S|$. The probability that a different pair (x', y) is a claw will depend slightly on whether (x, y) is a claw, but the dependence is small and so we can assume these are independent events. Thus, the expected number of claws is

$$\mathbb{E}[\text{number of claws}] = \frac{|\mathcal{X}||\mathcal{Y}|}{|S|}. \tag{6.1}$$

Similar logic shows that for collision finding,

$$\mathbb{E}[\text{number of collisions}] = \frac{|V|^2}{|W|}. \tag{6.2}$$

This quickly gives some intuition behind the "birthday paradox": V only needs to be of size $\approx \sqrt{|W|}$ to expect at least one collision, because $\sqrt{|W|}$ elements gives $|W|$ *pairs*, each of which has (approximately) a $1/|W|$ probability of making a collision.

Hence, if the image set is much larger than the domain set, this will give us a golden claw or collision problem.

Because of these size constraints, we can parameterize an algorithm to solve these problems by the sizes of $\mathcal{X}$, $\mathcal{Y}$, and S. Then we can make precise comparisons between claw finding and collision finding. Importantly, these lemmas depend on claw and collision problems on functions that are indistinguishable from random.

Lemma 6.1.1. *Let $\mathcal{A}$ be an algorithm that solves claw-finding for $|\mathcal{X}| = n_x$, $|\mathcal{Y}| = n_y$ and $|S| = n_s$. Then we can solve a collision finding problem for a function $h : V \to W$ with $|V| = n_x + n_y$ and $|W| = n_s$ with $O(1)$ calls to $\mathcal{A}$.*

Proof. Divide the set V into two sets $\mathcal{X}$ and $\mathcal{Y}$ with $\mathcal{X} \cup \mathcal{Y} = V$ and $|\mathcal{X}| = n_x$ and $|\mathcal{Y}| = n_y$. Use $S = W$, $f : \mathcal{X} \to S$ as $f(v) = h(v)$ and similarly for $g : \mathcal{Y} \to S$, and give these parameters to the claw-finding algorithm.

If $\mathcal{A}$ returns a pair $v_1 \in \mathcal{X}$ and $v_2 \in \mathcal{Y}$ such that $f(v_1) = g(v_2)$, this is a collision for h. If $\mathcal{A}$ returns no claw, there may still be a collision. Let v_1, v_2 be a pair with $h(v_1) = h(v_2)$. If our partition of V puts both v_1 and v_2 into $\mathcal{X}$ or both into $\mathcal{Y}$, $\mathcal{A}$ will not find them. Since we chose $\mathcal{X}$ and $\mathcal{Y}$ randomly, this has probability $1/2$ for each collision. We can repeat $O(1)$ times with different $\mathcal{X}$ and $\mathcal{Y}$ to find the collision; if all iterations return no claw, we conclude there is no collision. $\square$

Lemma 6.1.2. *Let $\mathcal{A}$ be an algorithm that solves collision-finding for $h : V \to W$ with $|V| = h_v$ and $|W| = h_w$. Assuming heuristics 6.1 and 6.2, we can solve a claw-finding problem for functions $f : \mathcal{X} \to S$ and $g : \mathcal{Y} \to S$ for $|\mathcal{X}| = ch_v$ and $|\mathcal{Y}| = (1-c)h_v$ with $c \in (\frac{1}{2}, 1)$, $|S| = h_w$ with $O(\frac{c}{1-c})$ calls to the collision oracle.*

Proof. Let V equal the disjoint union of $\mathcal{X}$ and $\mathcal{Y}$ (i.e., we may need to append a 0 to all elements of $\mathcal{X}$ and 1 to all elements of $\mathcal{Y}$) and let $W = S$. Let $h : V \to W$ be defined as $h(v) = f(v)$ if $v \in \mathcal{X}$ and $h(v) = g(v)$ if $v \in Y$. Then we give this to the collision-finding oracle.

The collision-finding oracle will either return a claw or a collision of either f or g. The expect number of collisions of f is $\frac{|\mathcal{X}|^2}{|S|}$, the expected number of collisions of g is $\frac{|\mathcal{Y}|^2}{|S|}$, and the expected number of claws is $\frac{|\mathcal{Y}||\mathcal{X}|}{|S|}$. Hence, we expect to need

$$\frac{|\mathcal{X}|^2 + |\mathcal{X}||\mathcal{Y}| + |\mathcal{Y}|^2}{|\mathcal{X}||\mathcal{Y}|} = 1 + \frac{ch_v}{(1-c)h_v} + \frac{(1-c)h_v}{ch_v} = O\left(\frac{c}{1-c}\right) \tag{6.3}$$

calls to the collision oracle. $\square$

Collision finding for large image sets can be reduced to small image sets with a linear cost:

Lemma 6.1.3. *Let $f(n)$ be the cost function for a collision finding algorithm for $h : V \to W$ where $|V| \approx |W| = n$. If the heuristic of 6.2 holds, then there is an algorithm for $|W| = N_W > n$ with cost $O(\frac{N_W}{n} f(n))$.*

Proof. The idea is that we construct a random function $h' : W \to W'$ where $|W'| = |V|$. Then $h' \circ h : V \to W'$ has the appropriate size so we run our original collision finding algorithm. Every collision of h is also a collision for $h' \circ h$, though there will be more. We expect roughly $\frac{|V|^2}{|W'|} = n$ collisions for $h' \circ h$. We expected $\frac{|V|^2}{|W|} = n^2/N_W$ collisions originally, so the probability that a $h' \circ h$ collision is an h collision is $\frac{n}{N_W}$. Hence, we run the original collision finder for $\frac{N_W}{n}$ times and check each collision it finds. □

We can also reduce collision finding for large domain and range to smaller domain and range, thanks to the technique of van Oorschot and Wiener [57]:

Lemma 6.1.4. *Suppose we have a collision-finding algorithm with cost $f(n, \mathsf{H}, p)$ for functions $h : \{0,1\}^n \to \{0,1\}^n$ that cost H to evaluate and have $p2^{2n}$ collisions (i.e., the probability of a random pair $(x, y) \in \{0,1\}^{2n}$ colliding is p). Then we can construct a collision-finding algorithm for any $h : \{0,1\}^* \to \{0,1\}^N$ with $N \geq n$ at cost*

$$O\left(f\left(N\theta, \frac{\mathsf{H}}{\theta}, \frac{p}{\theta^2}\right) + \frac{\mathsf{H}}{\theta}\right) \tag{6.4}$$

for any $\theta \in (0, 1)$.

Proof. There are two tricks here: The first is that the domain of h contains its range, so we can compose h with itself. Let $W = \{0,1\}^N$; we can then treat h as a function $h : W \to W$.

The second trick is to create a random subset $W_d \subseteq W$ formed of "distinguished" points. In practice we often just pick a certain number of leading 0s — if h is random, this will produce random subsets.

Then we construct a function $h_d : W_d \to W_d$ as follows: The function $h : W \to W$ also acts as $h : W_d \to W$. So we apply this function repeatedly until we find an element of W that is contained in W_d. We assume $|W_d| = \theta|W|$ for some $\theta \in (0, 1)$, so we expect $O(1/\theta)$ applications of h.

Note that if there is any collision for h, it "lifts" to a collision in h_d: Once we collide in h, subsequent applications of h will maintain that collision. Further, any collision for h_d is necessarily a collision for h. Hence, we have $p2^{2n}$ collisions for h_d, but its domain and range have size $2^n\theta$, so h_d has a much higher probability of collision, p/θ^2.

Hence, we apply our collision-finding algorithm on h_d, at cost $f(N\theta, \mathsf{H}/\theta, p/\theta^2)$. It returns a collision w_1 and w_2 with $h_d(w_1) = h_d(w_2)$. We use h to recreate the "path" of function applications to find the collision of h with a binary search, with cost H/θ. □

Corollary 1 (van Oorschot–Wiener Algorithm)**.** *Claw finding for a function $h : \{0,1\}^* \to \{0,1\}^n$ with probability of collision p costs at most*

$$O\left(\mathsf{H}\frac{1}{\sqrt{pM}} + \log M\right) \tag{6.5}$$

when using $O(Mn)$ bits of memory.

Proof. The simplest collision-finding algorithm is to build a list of M random elements and the evaluation of h on each element, sort this list, then check for collisions. This has cost $O(M(\mathsf{H}+\log M))$, roughly; there are potential time-memory trade-offs and parallelizations that we will leave to Section 6.3. The probability of a collision in the list is pM^2, so we repeat the entire process $O(1/pM^2)$ times and get a total cost of

$$O\left(\frac{1}{pM}(\mathsf{H} + \log M)\right). \tag{6.6}$$

Arguably, instead of repeating the full process, we should replace a single element at a time. This now functions as a random walk on a Johnson graph, as analyzed in Section 3.2, which concluded that both approaches have the same asymptotic cost.

Using the naive collision-finding as the oracle f, we apply Lemma 6.1.4 and get a cost of

$$\frac{\theta^2}{pM}\left(\frac{\mathsf{H}}{\theta} + \log M\right) + \frac{\mathsf{H}}{\theta}. \tag{6.7}$$

We optimize by setting $\theta = \sqrt{pM}$ to give the total cost.

□

Lemmas 6.1.1 to 6.1.4 show that claw-finding and collision finding are essentially the same problem. For individual algorithms, often there are simple modifications that allow a direct conversion from collision finding to claw finding and vice versa, rather than using the reductions in this chapter. Further, these lemmas show that collision finding can mostly be reduced to collision finding for a random function.

6.2 Meet in the Middle

To perform a meet-in-the-middle attack on the claw-finding problem, we enumerate a list S_x consisting of pairs $(x, f(x))$ for $x \in \mathcal{X}$, sorted by the value of $f(x)$. Then for each

$y \in \mathcal{Y}$, we compute $g(y)$ and search in S_x for a pair $(x, f(x))$ with $g(y) = f(x)$. When we find this, we have found the claw.

With unlimited memory, we would enumerate all of $\mathcal{X}$, then we only need to check each y once. With limited memory, we would repeat the procedure with disjoint subsets of $\mathcal{X}$. For each subset, we need to check all of $\mathcal{Y}$.

Let R_x be the size of the set S_x and H be the number of RAM operations to compute f or g. Without latency, the total number of RAM operations for each subset S_x is:

- $O(R_x(\log R_x + \mathsf{H}))$ operations to construct each list S_x.
- $O(|Y|(\log R_x + \mathsf{H}))$ operations to look for a claw that matches S_x.

There will be $\frac{|X|}{R_x}$ subsets, making the total cost

$$O\left(\frac{|\mathcal{X}|}{R_x}\left(R_x + |\mathcal{Y}|\right)\left(\log R_x + \mathsf{H}\right)\right). \tag{6.8}$$

The optimal here is to take $R_x = |\mathcal{X}|$. Choosing $\mathcal{X}$ to be the smaller set (without loss of generality) saves logarithmic factors.

When we account for latency in dimension d, using Section 2.3.5, reading or writing to the list S_x costs $R_x^{1/d}$. This leads to:

- $O(R_x(R_x^{1/d} + \mathsf{H}))$ time to construct each list.
- $O(|\mathcal{Y}|(R_x^{1/d} + \mathsf{H}))$ time to look for the claw.

Assuming H is dominated by $R_x^{1/d}$, this leads to a total cost of

$$O\left(\frac{|\mathcal{X}|}{R_x^{1-1/d}}\left(R_x + |\mathcal{Y}|\right)\right). \tag{6.9}$$

Again, the optimal is to choose $\mathcal{X}$ to be the smaller set and take $R_x = |\mathcal{X}|$.

Parallelism

To parallelize, assume we have P processors with *shared* memory of size M. The obvious parallelization is to build the list S_x in parallel, then search it in parallel.

We store S_x as a hash table to avoid simultaneous memory access issues. We initialize the memory into approximately $M/\log M$ sorted lists. Each processor computes a tuple $(x, f(x))$, hashes $f(x)$ somehow, and uses that to decide which list to send the tuple to. As long as M is in $\Omega(P^2)$, then collisions are unlikely and each processor can insert into its list without issue, using any standard technique to handle shared memory access.

Once the list is constructed, we can similarly divide $\mathcal{Y}$ into subsets and have each processor search through a different subset of $\mathcal{Y}$. The same arguments apply to show that we will not suffer significant latency from processors accessing the same hash list.

This takes the costs without latency to:

- $O\left(\frac{R_x}{P}(\log R_x + \mathsf{H})\right)$ for construction;
- $O\left(\frac{|\mathcal{Y}|}{P}(\log R_x + \mathsf{H})\right)$ to search $\mathcal{Y}$.

In other words, this parallelizes perfectly, up to $P = O(\sqrt{M})$.

Cost 6.1 gives the costs to search $\mathcal{X}$ and $\mathcal{Y}$ for claws with a meet-in-the-middle using P processors with M total memory (in units of the bit-length of $(x, f(x))$) with M in $O(|\mathcal{Y}|)$. For the depth of this algorithm, simply divide the cost by P.

Cost 6.1 Classical meet-in-the-middle.

Model	Measure	Cost
No latency	RAM ops	$O\left(\frac{\lvert\mathcal{X}\rvert\lvert\mathcal{Y}\rvert}{M}(\log M + \mathsf{H})\right)$
Latency	proc. hrs.	$O\left(\frac{\lvert\mathcal{X}\rvert\lvert\mathcal{Y}\rvert}{M^{1-1/d}}\mathsf{H}\right)$

6.3 van Oorschot–Wiener

van Oorschot–Wiener (VW) collision finding is a parallelization of Pollard's rho algorithm [57]. Pollard originally designed the rho algorithm to find collisions of hash functions. The idea is that a hash function $H : \{0,1\}^* \to \{0,1\}^n$ can be composed with itself. From any input x, this gives a chain

$$x \underset{H}{\rightarrow} H(x) \underset{H}{\rightarrow} H^2(x) \underset{H}{\rightarrow} H^3(x) \underset{H}{\rightarrow} \cdots \quad (6.10)$$

This chain will eventually cycle, meaning there will be integers m and n such that $H^m(x) = H^{m+n}(x)$. If m is the least such integer, then the two strings $H^{m-1}(x)$ and $H^{m+n-1}(x)$ have the same hash value. There are various methods to detect when such a cycle has occured.

To parallelize, van Oorschot-Wiener used the idea of "distinguished points". This is the idea behind Lemma 6.1.4, and here we will give more precise details from their paper [57].

The easiest way to produce distinguished points is to use the hash function itself, which acts as a random function. Then we consider the first s bits: If they are all zero, we call the point distinguished. Each hash has probability $\theta := 2^{-s}$ of producing a distinguished point. We store distinguished points in a shared list, and check for collisions within this list. If we find a collision of distinguished points, we go back to the last distinguished point on either side and carefully check each hash for collisions.

We store distinguished points as (x_1, x_2, n), where x_1 is the starting point, x_2 is the next distinguished point found, and n is the number of hashes such that $x_2 = H^n(x_1)$. Given two pairs (x_1, x_2, n) and (y_1, x_2, m), we compute $H^{m-n}(y_1)$ (assuming WLOG $m \geq n$) and sequentially hash this value and x_1 until we find the actual collision. This requires $O(1/\theta)$ hashes and comparisons.

To parallelize, we simply have each parallel processor perform its own chain of hash function iterations, but save the distinguished points in a shared memory. If the memory gets full before finding a collision, we simply discard a value. Assume we store distinguished points in a hash table to facilitate parallel memory insertions.

For claw-finding, we use Lemmas 6.1.2 and 6.1.4. We need to define a random function $H : \mathcal{X} \cup \mathcal{Y} \to \mathcal{X} \cup \mathcal{Y}$ such that if $f(x) = g(y)$, then $H(x) = H(y)$. We actually use the set $\mathcal{X} \sqcup \mathcal{Y} = (\mathcal{X} \times \{\mathtt{0}\}) \cup (\mathcal{Y} \times \mathtt{1})$, but the effect is the same. We take a suitably random hash function $h : \{\mathtt{0}, \mathtt{1}\}^* \to \mathcal{X} \sqcup \mathcal{Y}$, and construct H as follows for an input string $z = (x, \mathtt{b})$:

$$H(z) = \begin{cases} h(f(x)) & , \mathtt{b} = \mathtt{0} \text{ and } x \in \mathcal{X} \\ h(g(x)) & , \mathtt{b} = \mathtt{1} \text{ and } x \in \mathcal{Y} \end{cases}. \tag{6.11}$$

This H satisfies our requirements and we can perform van Oorschot-Wiener collision finding. We can construct h out of a hash function with an n-bit output by simply iterating it until it the output is in the required set.

As in Lemma 6.1.2, this reduction produces a lot of collisions that do not represent claws. On average, we expect $O(|\mathcal{X}| + |\mathcal{Y}|)$ collisions in this problem. We perform a brute force search on all collisions, checking each collision we find to see if it is the solution of the original claw problem. Thus, we expect to need to find about $O(|\mathcal{X}| + |\mathcal{Y}|)$ collisions.

Here we notice that VW collision finding is more likely to find certain collisions than others. Figure 6.1 shows this phenomenon. Our random function h that we used to construct H might put our golden collision in a bad place, so we need to regularly change to a different random function h'. Adj et al. [3] go over the full analysis and provide some experimental evidence that this works, and that the heuristic constants are accurate, in particular for isogenies.

6.3.1 Analysis

We follow VW's analysis here [57]. Consider the "trail" of elements leading to a distinguished point (e.g., in Figure 6.1a, `31` and `45` are on the trail of `40`). If distinguished points occur with probability θ, we expect $1/\theta$ points in each distinguished point's trail. Hence, if we have M distinguished points in memory, there are roughly $O(M/\theta)$ points that would lead to one of these distinguised points. Hence, each processor has a

$$\frac{M/\theta}{|\mathcal{X}| + |\mathcal{Y}|} \tag{6.12}$$

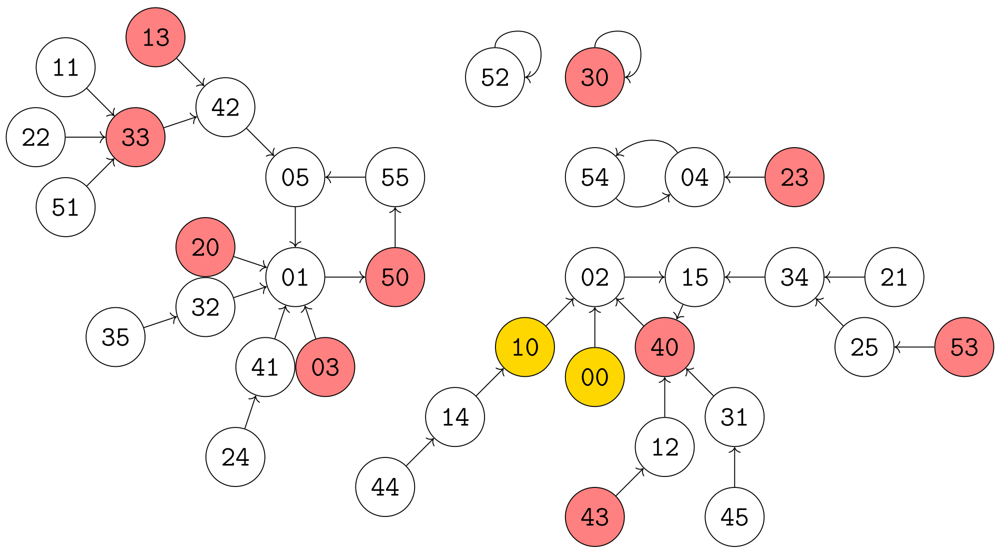
probability of selecting a point that will lead to a distinguished point already in memory, and thus probably a collision. Once a collision is found among the distinguised points, we need $2/\theta$ steps to find the actual collision. This leads to a total cost of

$$\frac{|\mathcal{X}| + |\mathcal{Y}|}{M/\theta} + \frac{2}{\theta} \tag{6.13}$$

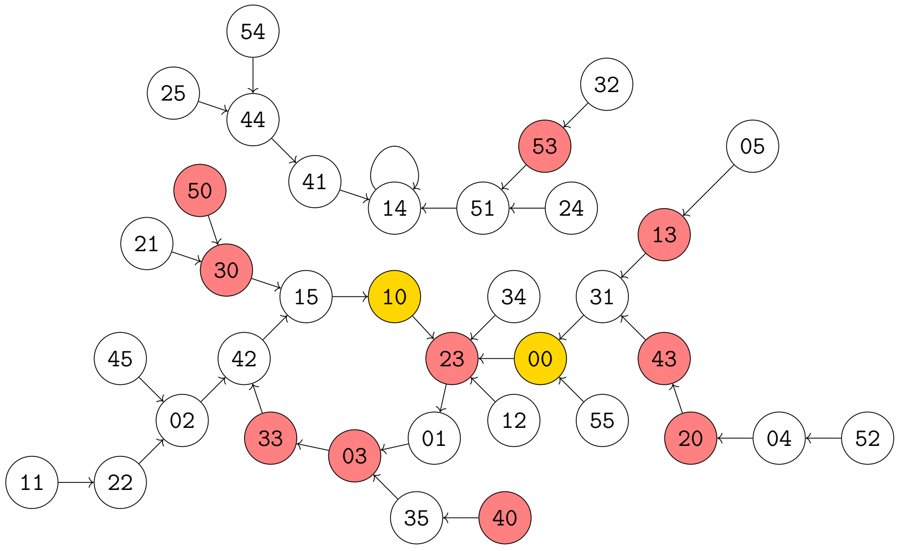
RAM operations to find each collision. There are some issues with this analysis that VW address, but they only change constant factors. Distinguished point finding parallelizes perfectly but trail retracing does not parallelize, so Equation 6.13 does not change with parallelization. We can optimize for RAM operations and get an optimal

$$\theta = \sqrt{\frac{M}{|\mathcal{X}| + |\mathcal{Y}|}}. \tag{6.14}$$

To crudely estimate latency, consider that the algorithm computes roughly $\frac{|\mathcal{X}|+|\mathcal{Y}|}{M/\theta}$ points before finding a collision. Roughly θ of those points will be distinguished and need to be



(a) Function graph under one choice of h.



(b) Function graph under one choice of h.

Figure 6.1: Function graphs of two random functions on the same data. Distinguished points (red) are those that end in 0 or 3; the golden collision is golden orange. In 6.1a, the golden collision ends up in a bad place and will only be found if one processor starts with "`00`" and another starts with `44`, `14`, or `10`. In contrast, 6.1b shows an excellent random function, where the majority of starting points will lead to the golden collision.

inserted into memory. Thus, we end up with a memory insertion latency term of

$$O\left(\frac{|\mathcal{X}|+|\mathcal{Y}|}{M/\theta}\theta M^{1/d}\right) = O\left(\theta^2\frac{|\mathcal{X}|+|\mathcal{Y}|}{M^{1-1/d}}\right), \tag{6.15}$$

where memory insertion take time $O(M^{1/d})$.

Assume that each processor perfroms the sequence of "iterate function, find distinguished point, insert to memory, retrace trail to find collision" independently of the other processors. That is, one might be retracing its steps while most of the others are still iterating the function. This assumption means that we can average the time for each step over all of the processors. Hence, the total cost in processor-hours is independent of the number of processors. Thus, the total cost with latency is

$$O\left(\theta^2\frac{|\mathcal{X}|+|\mathcal{Y}|}{M^{1-1/d}} + \theta\frac{|\mathcal{X}|+|\mathcal{Y}|}{M}\mathsf{H} + \frac{2}{\theta}\mathsf{H}\right). \tag{6.16}$$

The two right-hand terms include H, the cost to compute the function H, which is not included in the memory insertion. Optimizing this directly is cumbersome. Instead we can note:

- If $1/\theta > M^{1/d}/\mathsf{H}$, memory is low so the middle term of computation time dominates;
- If $1/\theta < M^{1/d}/\mathsf{H}$, memory is large and the left term of insertion time dominates.

Using X and Y to denote $|\mathcal{X}|$ and $|\mathcal{Y}|$, optimizing each case separately yields:

$$\theta = \begin{cases} O\left(\sqrt[3]{\frac{M^{1-1/d}\mathsf{H}}{X+Y}}\right) & , M \text{ in } \Omega\left((X+Y)^{\frac{d}{2+d}}\mathsf{H}^{\frac{3d}{2+d}}\right) \\ O\left(\sqrt{\frac{M}{X+Y}}\right) & , M \text{ in } O\left((X+Y)^{\frac{d}{2+d}}\right). \end{cases} \tag{6.17}$$

Astute readers may notice that these equations are inconsistent: If M is in $\Theta((X+Y)^{\frac{d}{2+d}})$, the recommended values of θ are different. Since this is a rough approximation, we will ignore this issue.

Storing distinguished points with a hash table means we will overwhelm the memory insertion with too many points at once. Thus, we only want to find roughly $O(M)$ distinguished points in every time step. We expect to find $O(\theta P)$ distinguished points in every step, so we need P in $O(M/\theta)$, or

$$P \text{ in } O\left(\sqrt{M(|\mathcal{X}|+|\mathcal{Y}|)}\right). \tag{6.18}$$

This gives the costs with and without latency for finding a single collision; see Cost 6.2. This cost looks odd because the cost with latency seems to scale slower than the cost in RAM operations, but this is only because memory is assumed to be so large in the latency case; with that much memory, the RAM operation cost scales better.

Cost 6.2 Finding a single collision with VW. The latency cost only applies so long as M is in $\Omega\left((X+Y)^{\frac{d}{2+d}}\right)$. Here $X = |\mathcal{X}|$ and $Y = |\mathcal{Y}|$.

Model	Measure	Cost
No latency	RAM ops.	$O\left(\sqrt{\frac{X+Y}{M}}\mathsf{H}\right)$
Latency	proc. hrs.	$O\left(\sqrt[3]{\frac{X+Y}{M^{1-1/d}}}\right)$

We will need to find $O(X+Y)$ collisions, on average, to find the golden collision. This gives Cost 6.3. This cost includes extra terms for the total expected number of memory insertions: this is the total number of iterations multiplied by θ. This is necessary if we consider cases with more than $X+Y$ memory cells.

Cost 6.3 Finding a golden collision with VW, with the same notation and limitations as Cost 6.2.

Model	Measure	Cost
No latency	RAM ops.	$O\left(\sqrt{\frac{(X+Y)^3}{M}}\mathsf{H} + X + Y\right)$
Latency	proc. hrs.	$O\left(\sqrt[3]{\frac{(X+Y)^4}{M^{1-1/d}}} + M^{1/d}(X+Y)\right)$

For both Costs 6.2 and 6.3, we can find the depth by dividing by the number of processors.

6.3.2 Application to SIDH

For SIDH, $|\mathcal{X}| = |\mathcal{Y}| = p^{1/4}$ and H is $O((\log p)^3)$. This leads to Cost 6.4. Figure 6.2 compares the two algorithms, showing that VW has a lower cost except for extremely high memory. Since both algorithms parallelize perfectly, this means that VW also has lower depth. Hence, we will ignore meet-in-the-middle and assume VW is the better algorithm.

In Figure 6.2, “SIKE-610” means an instantiation of SIKE with a 610 bit prime.

Cost 6.4 Comparison of VW and Meet-in-the-middle for SIDH.

Model	Algorithm	Measure	Cost
No Latency	VW	RAM ops.	$O\left(\frac{p^{3/8}}{\sqrt{M}}\log^3 p + p^{1/4}\right)$
	MitM	RAM ops.	$O\left(\frac{p^{1/2}}{M}\left(\log^3 p + \log M\right)\right)$
Latency	VW	proc. hrs.	$O\left(\frac{p^{1/3}}{M^{\frac{d-1}{3d}}} + M^{1/d}p^{1/4}\right)$
	MitM	proc. hrs.	$O\left(\frac{p^{1/2}}{M^{\frac{d-1}{d}}}\right)$

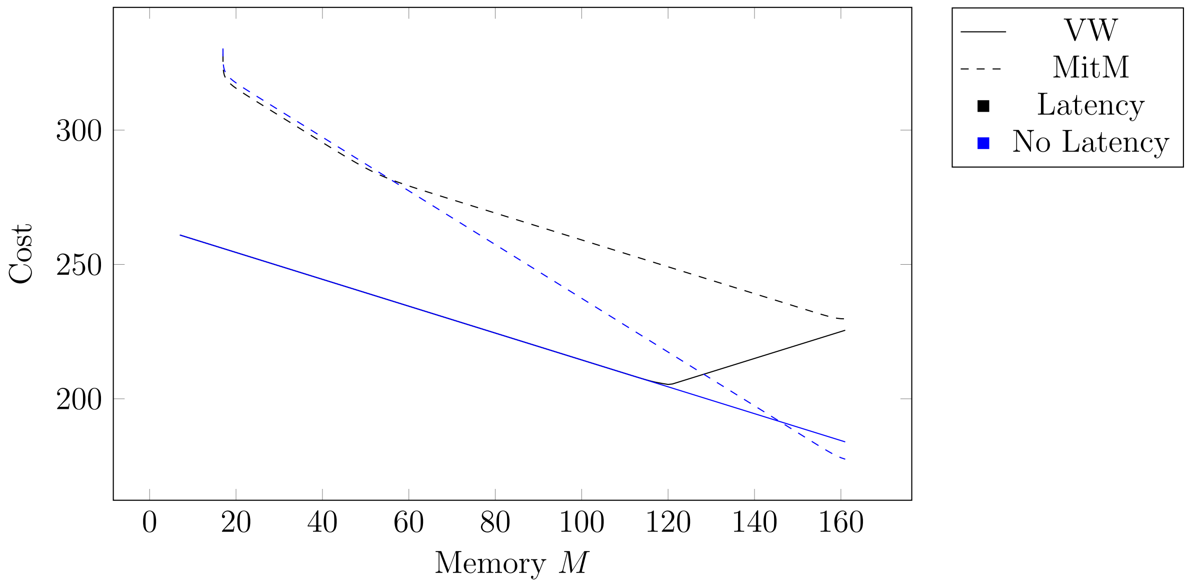


Figure 6.2: van Oorschot-Wiener vs. Meet-in-the-middle costs for attacking SIKE-610. For latency, the dimension is 2 and costs are processor-hours; without latency, costs are RAM operations. Axes are log base 2.

6.4 Grover's Algorithm

To apply Grover's algorithm to claw-finding, we consider a boolean function on $\mathcal{X} \times \mathcal{Y}$, with value `0` on (x, y) if $f(x) \neq g(y)$, and `1` otherwise. This will search for claws. The space has size $|\mathcal{X}||\mathcal{Y}|$, so we need

$$\sqrt{|\mathcal{X}||\mathcal{Y}|} \tag{6.19}$$

oracle calls.

To parallelize Grover's algorithm, the optimal method [58] just divides the search space and assign different pieces to different independent processors. With P processors, each processor needs

$$\sqrt{\frac{|\mathcal{X}||\mathcal{Y}|}{P}} \tag{6.20}$$

oracle calls. Since all processors do this, the total number of oracle calls *increases* proportional to $\sqrt{P}$. Suppose that calling f and g cost F_G gates, requires F_D depth, and F_W qubits. Then we can compute the total cost in all the different models and give Cost 6.5. Assume the costs to compute the functions f and g are logarithmic in the size of the search space, meaning we can say that in all the cost models we consider, the cost is

$$\tilde{O}(\sqrt{P|\mathcal{X}||\mathcal{Y}|}). \tag{6.21}$$

Cost 6.5 Claw finding with Grover's algorithm. This ignores latency or locality costs because each processor only uses F_W. Here $X = |\mathcal{X}|$ and $Y = |\mathcal{Y}|$.

Model	Measure	Cost
Passive Circuit (2.4.1)	Gates	$O\left(\sqrt{PXY}\mathsf{F}_G\right)$
Active Circuit (2.4.2)	Total	$O\left(\sqrt{PXY}\mathsf{F}_D\mathsf{F}_W\right)$
	Depth	$O\left(\sqrt{\frac{XY}{P}}\mathsf{F}_D\right)$
	Width	$O(P\mathsf{F}_W)$
Passive Latency (2.4.3)	Gate-Time	$O\left(\sqrt{PXY}\mathsf{F}_G\right)$
Active Local (2.4.4)	Total	$O\left(\sqrt{PXY}\mathsf{F}_D\mathsf{F}_W \log^d\left(\sqrt{PXY}\mathsf{F}_D\mathsf{F}_W\right)\right)$
	Depth	$O\left(\sqrt{\frac{XY}{P}}\mathsf{F}_D\right)$
	Width	$O(P\mathsf{F}_W)$
QRAM (2.4.5)	Gates	$O\left(\sqrt{PXY}\mathsf{F}_G\right)$
	Time	$O\left(\sqrt{\frac{XY|}{P}}\mathsf{F}_D\right)$

6.5 Tani's Algorithm

Tani's algorithm applies Szegedy's quantum random walk to claw finding [55]. We need to build a special graph to do this.

Definition 6.5.1. *Given two graphs G_1 and G_2, the* direct graph product *is the graph $G_1 \times G_2$ with:*

- $V(G_1 \times G_2) = V(G_1) \times V(G_2)$.
- *Two tuples (v_1, v_2) and (u_1, u_2) are adjacent in $G_1 \times G_2$ if and only if $v_1 \sim u_1$ and $v_2 \sim u_2$.*

Figure 6.3 shows an example of a graph product. To take a random step on a graph product $G_1 \times G_2$, we take a random step in the first graph, then a random step in the second graph (or take both steps simultaneously).

For Tani's algorithm, we take the product of two Johnson graphs: $J(\mathcal{X}, R_x) \times J(\mathcal{Y}, R_y)$. A single vertex is a pair of lists (S_x, S_y). To take a single step in the graph from such a vertex, first we choose some $x \in S_x$ and some $y \in S_y$. Then we remove x and y from their respective lists, choose a new $x' \notin S_x$ and $y' \notin S_y$, and insert the new elements into each list.

6.5.1 Basic Costs

Recalling Section 3.3, the full cost of Tani's algorithm needs the setup cost S, the update cost U, and check cost C, the spectral gap δ, and the fraction of marked vertices ϵ.

Marked Vertices:

We define a vertex (S_x, S_y) to be marked if there is a claw between S_x and S_y: Some $x \in S_x$ and $y \in S_y$ such that $f(x) = g(y)$. Assuming a unique claw (x, y), there will be $\binom{R_x-1}{|\mathcal{X}|-1}$ lists containing x out of $\binom{R_x}{|\mathcal{X}|}$ total, which gives

$$\epsilon = \frac{\binom{R_x-1}{|\mathcal{X}|-1}}{\binom{R_x}{|\mathcal{X}|}} \frac{\binom{R_y-1}{|\mathcal{Y}|-1}}{\binom{R_y}{|\mathcal{Y}|}} = \frac{R_x R_y}{|\mathcal{X}||\mathcal{Y}|}. \tag{6.22}$$

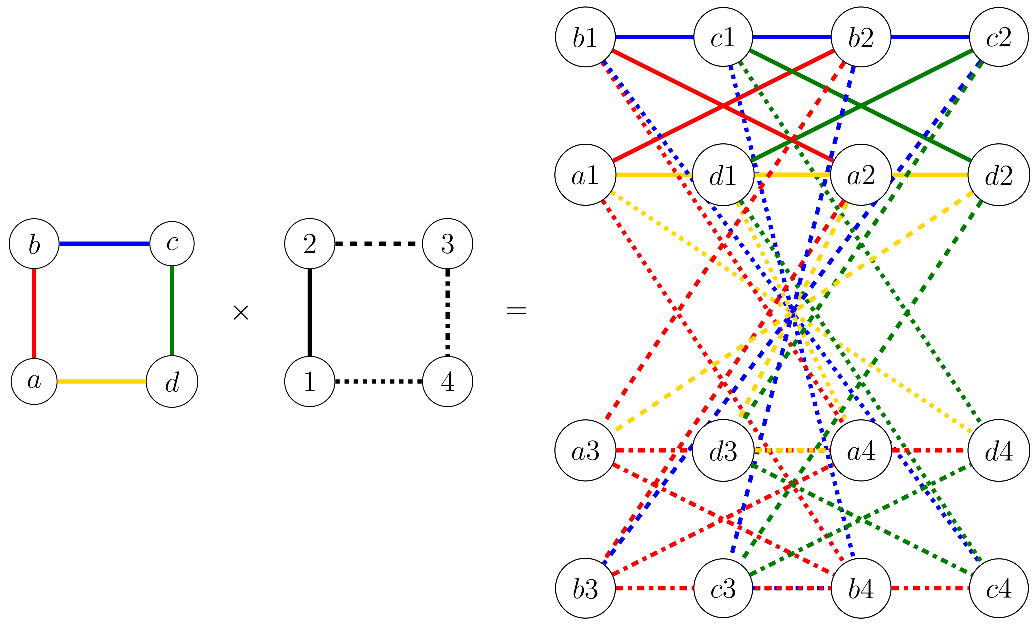


Figure 6.3: An example graph product of C_4 with itself. Even for these simple graphs, the resulting product is nearly impossible to comprehend visually.

Spectral gap:

The spectral gap of a Johnson graph $J(\mathcal{X}, R)$ is $O(1/R)$. The spectral gap of a graph product is the minimum of the two spectral gaps. We can see this by noting that the adjacency matrix of a graph product is the tensor product of the two constituent adjacency matrices. Thus, the spectral gap for Tani's algorithm

$$\delta = \min\left\{\frac{1}{R_x}, \frac{1}{R_y}\right\} = O\left(\frac{1}{\max\{R_x, R_y\}}\right). \tag{6.23}$$

As Section 4.3.2 showed, this differs by only a constant factor if we add loops and assume $\max\{R_x, R_y\} \leq |\mathcal{X}|/2$.

Set-up:

To initialize Tani's algorithm, we need to initialize two lists that satisfy the data structure requirements of Section 4.2. This raises two issues:

Element Selection: As defined, Tani's algorithm requires the lists to be *sets* without repeated elements. Thus, we need some way to choose a random subset of $\mathcal{X}$ (and $\mathcal{Y}$) without choosing duplicates.

Setting Claw Check Bit: The data structures use extra claw-checking bits for the entire set. If we intialize the lists S_x and S_y separately, then we have a separate claw finding problem to solve for S_x and S_y.

Section 4.3.3 describes the set-up routine, but we must also search for claws.

In the extended Knuth shuffle, we initialize one list as normal. As we initialize the second list, every time we swap a new element y into the list, we search for $g(y)$ in the first list. Asymptotically, this adds no cost, since we already needed to perform a search in a sorted list.

With a sample-and-sort, once we have initialized both lists, we can sort them *together* by the values of $f(x)$ and $g(y)$. Simultaneous comparisons of sequential elements can detect any claws, and the results can be summarized with a binary tree of additions (like the "Sum" step from Section 4.2.2).

Altogether this gives Cost 6.6.

Cost 6.6 Set-up for Tani's algorithm. The QRAM model uses an extended Knuth shuffle to populate a quantum radix tree; the rest uses a sample-and-sort to populate a sliding sorted array.

Model	Measure	Cost
Passive Circuit (2.4.1)	Gates	$O\left(R(m\log R+\log^2 R+\mathsf{F}_G)\right)$
Active Circuit (2.4.2)	Total	$O\left(R(m+\mathsf{F}_W)\left(\log^2 R\log m+\mathsf{F}_D\right)\right)$
	Depth	$O\left(\log^2 R\log m+\mathsf{F}_D\right)$
	Width	$O\left(R(m+\mathsf{F}_W)\right)$
Passive Latency (2.4.3)	Gate-Time	$O\left((Rm)^{1+2/d}\log^2 R\right)$
Active Local (2.4.4)	Total	$O\left(R^{1+2/d}m^{1/d}\log^{2+d} R(m+\mathsf{F}_W)\right)+o(R^{1+2/d})$
	Depth	$O\left((Rm)^{1/d}\log^2 R+\mathsf{F}_D\right)$
	Width	$O\left(R(R^{1/d}+m+\mathsf{F}_W)\right)$
QRAM (2.4.5)	Gates	$O\left(R(\log R+m^2+\mathsf{F}_G)\right)$
	Time	$O\left(R^{1+1/d}\log R\right)$

Update:

For the update, we follow the approach with loops from Section 4.3.2. This is dominated by Cost 4.7, updating the claw counter, so Tani's algorithm's update cost, 6.7, is the same.

Cost 6.7 Update step in Tani's algorithm

Model	Measure	Cost
Passive Circuit (2.4.1)	Gates	$O(Rm + \mathsf{F}_G)$
Active Circuit (2.4.2)	Total	$O((Rm + \mathsf{F}_W)(\log R \log\log R + \mathsf{F}_D))$
	Depth	$O(\log R \log\log R + \mathsf{F}_D)$
	Width	$O(Rm + \mathsf{F}_W))$
Passive Latency (2.4.3)	Gate-Time	$O(Rm + \mathsf{F}_G)$
Active Local (2.4.4)	Total	$O((Rm)^{1+1/d} + o((Rm)^{1+1/d})$
	Depth	$O((Rm)^{1/d} + \mathsf{F}_D)$
	Width	$O(Rm + \mathsf{F}_W)$
QRAM (2.4.5)	Gates	$O(m^2 + \mathsf{F}_G)$
	Time	$O(R^{1/d}m + \mathsf{F}_T)$

Check:

By the construction of the augmented data structure, the check uses a single control bit. This is one local gate, and hence it's cost is $O(1)$ in each cost model.

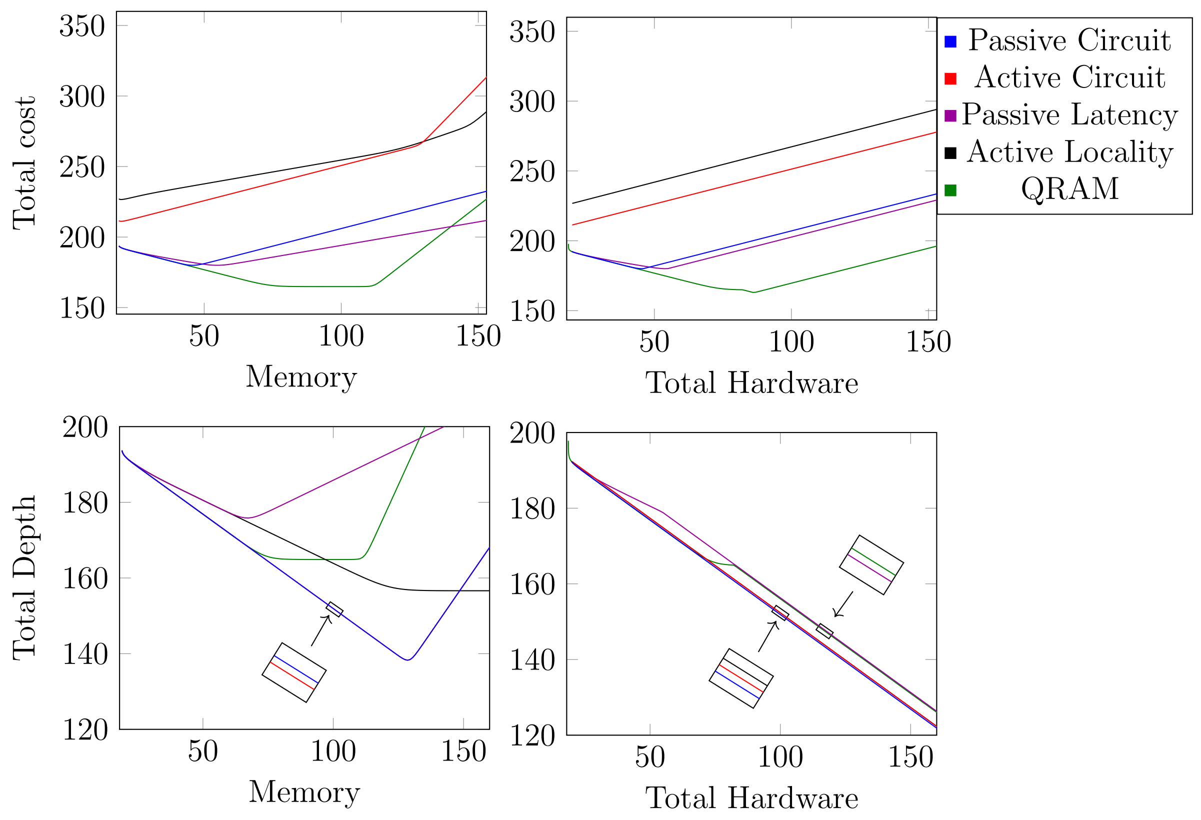
6.5.2 Analysis

Assembling the costs in each model gives completely inscrutable equations. To explain what's happening, Figure 6.4a shows how the total cost changes with R in each model. Appendix B analyzes the costs under each model, and chooses an R to minimize costs. The optimal values of R are:

Passively-Corrected: $R = \mathsf{F}_G/m$.

Actively-Corrected: $R = \mathsf{F}_W/m$.

QRAM: $R = (|\mathcal{X}||\mathcal{Y}|)^{1/d}$.



(a) Increasing memory (b) Increasing parallelism after optimal memory

Figure 6.4: Costs and depths of Tani's algorithm for CSSI for SIKE-610 at different memory levels. For 6.4a, the increased memory is used for increased R in a single instance. For 6.4b, the increased hardware (memory over all parallel processors) is used to parallelize instances that use the cost-optimal R. The dimension d is 2. Axes are log base 2.

To explain these optima, the total number of walk steps will decrease as R increases. As long as the oracle computation is the dominant cost in the insertion step, this makes the total number of gates decrease as R increases. When R gets too large, the memory operations become the dominant cost. The cost of each memory operation increases faster than the number of walk steps decreases, so the cost starts to increase with R. Roughly speaking, the optimal is always at the point where a memory operation is the same cost as an oracle computation.

A few other features of Figure 6.4a:

- The set-up step, using a sort-and-measure technique, requires many ancillae for the sorting algorithm. This increases the necessary number of qubits, and much more for geometric models with locality or latency. This makes the geometric models seem cheaper in Figure 6.4a, because the costs grow slower with increased memory. In reality, the costs grow slower because the algorithm gets less efficient: it's "wasting" all of its extra memory just storing ancilla qubits for the set-up sort. Figure 6.4a shows that the geometric models have a much slower decrease in depth.

This also explains the sudden jump of the active circuit model at $R \approx 2^{125}$: at this memory level, set-up costs start to dominate. The active locality model will show a similar jump but it will occur at higher memory levels.

- In the QRAM model, the cost plateaus at $R \approx 2^{73}$. This is only because we chose $d = 2$, so that once latency terms dominate in the update step, the increase in latency of each step, proportional to $\sqrt{R}$, precisely cancels out the reduction in the number of steps, also proportional to $\sqrt{R}$.

Cost 6.8 Total costs of Tani's algorithm, with the set size optimized. Here $X = |\mathcal{X}|$ and $Y = |\mathcal{Y}|$

Model	Measure	Cost
Passive Circuit (2.4.1)	Gates	$O(\sqrt{XY\mathsf{F}_G m})$
Active Circuit (2.4.2)	Total	$O\left(\sqrt{XY\mathsf{F}_W m}\mathsf{F}_D\right)$
	Depth	$O\left(\sqrt{\frac{XYm}{\mathsf{F}_W}}\mathsf{F}_D\right)$
	Width	$O(\mathsf{F}_W))$
Passive Latency (2.4.3)	Gate-Time	$O\left(\sqrt{XY\mathsf{F}_G m}\right)$
Active Local (2.4.4)	Total	$O\left(\sqrt{XY\mathsf{F}_W m}\mathsf{F}_D \log^d(\sqrt{XY\mathsf{F}_W m}\mathsf{F}_D)\right)$
	Depth	$O(\sqrt{\frac{XYm}{\mathsf{F}_W}}\mathsf{F}_D)$
	Width	$O(\mathsf{F}_W)$
QRAM (2.4.5)	Gates	$O\left((XY)^{1/3}(m^2 + \mathsf{F}_G)\right)$
	Time	$O\left((XY)^{\frac{d+1}{3d}} m\right)$

Cost 6.8 summarizes all of the costs at the optimal value of R. Here we see that in all the cost models except QRAM, the cost scales as $\tilde{O}(\sqrt{|\mathcal{X}||\mathcal{Y}|})$ with $|\mathcal{X}|$ and $|\mathcal{Y}|$, the same as Grover's algorithm. Tani's algorithm ends up slightly cheaper than Grover's algorithm, by factors of roughly $\sqrt{\mathsf{F}_G/m}$ (passively-corrected) or $\sqrt{\mathsf{F}_W/m}$ (actively-corrected). For isogeny computations, this is actually noticeable.

In the QRAM model, the optimal R is $(|\mathcal{X}||\mathcal{Y}|)^{1/3}$, the same as Tani's original parameters. The latency terms dominate, but the optimal memory size R is when the set-up cost equals the cost of the main loop of the random walk. At this optimal, the total cost depends on d; with $d = 2$, the cost scales as $\tilde{O}(\sqrt{|\mathcal{X}||\mathcal{Y}|})$, and as d approaches infinity, the cost approaches $\tilde{O}((|\mathcal{X}||\mathcal{Y}|)^{1/3})$, which was Tani's original result.

6.5.3 Parallelization

Naive

The naive parallelization is to simply divide the search space, the same way that Grover's algorithm parallelizes. With P parallel processors, each one will search a space of size $|\mathcal{X}||\mathcal{Y}|/P$. Using this in Cost 6.8 gives Cost 6.9.

Cost 6.9 Costs of Tani's algorithm with naive parallelization. Here $X = |\mathcal{X}|$ and $Y = |\mathcal{Y}|$.

Model	Measure	Cost
Passive Circuit (2.4.1)	Gates	$O(\sqrt{PXY\mathsf{F}_G m})$
Active Circuit (2.4.2)	Total	$O\left(\sqrt{PXY\mathsf{F}_W m}\mathsf{F}_D\right)$
	Depth	$O\left(\sqrt{\frac{XYm}{P\mathsf{F}_W}}\mathsf{F}_D\right)$
	Width	$O(P\mathsf{F}_W)$
Passive Latency (2.4.3)	Gate-Time	$O\left(\sqrt{PXY\mathsf{F}_G m}\right)$
Active Local (2.4.4)	Total	$O\left(\sqrt{PXY\mathsf{F}_W m}\mathsf{F}_D \log^d(\sqrt{PXY\mathsf{F}_W m}\mathsf{F}_D)\right)$
	Depth	$O(\sqrt{\frac{XYm}{P\mathsf{F}_W}}\mathsf{F}_D)$
	Width	$O(P\mathsf{F}_W)$
QRAM (2.4.5)	Gates	$O\left(P^{2/3}(XY)^{1/3}(m^2 + \mathsf{F}_G)\right)$
	Time	$O\left((XY/P)^{\frac{d+1}{3d}}\right)$

As with Grover's algorithm, the depth decreases while the total cost increases, both proportional to $\sqrt{P}$, for all non-QRAM models. In the QRAM model, the total time decreases in proportion to $P^{\frac{d+1}{3d}}$, while the gates increase as $P^{2/3}$. Thus, when $P \leq (|\mathcal{X}||\mathcal{Y}|)^{\frac{1}{3d+1}}$, the time cost dominates, and above that the gate costs. Hence, this is in some sense the "optimal" parallelization, as it minimizes the cost.

Figure 6.4b shows how the costs generally increase with parallelization. The curves have roughly the same shape as Figure 6.4a, since costs increase with the square root of either memory or parallelism. The depth, on the other hand, is much lower with naive parallelization than increased memory, since there is less time wasted with a longer sort in the update step.

In the QRAM model, we parallelized at the memory level where latency matched computation. This prevents the plateau of depth, but increases the costs.

Increased Memory

This is not really a method of parallelization, but another depth-width tradeoff. We could use larger values of R to decrease the depth, but this would increase both the width and total cost. Looking at the costs in Appendix A shows that in all (non-QRAM) cost models, the total cost increases roughly proportional to $\sqrt{R}$ and the depth decreases proportional to $\sqrt{R}$ as well. This means the trade-off scales approximately in the same way as naive parallelization.

Increasing the memory requires all of the memory to be capable of quantum communication with the rest of the memory, while for naive parallelization, each parallel quantum computer can act independently. Further, logarithmic factors in R lead to slightly higher costs when using higher memory. Thus, using increased memory is an inferior approach to a depth-width tradeoff.

Jeffery, Magniez, and de Wolf

Jeffery, Magniez, and de Wolf (JMW) [36] give a query-optimal parallelization of Ambainis' algorithm for element-distinctness that is easily modified for claw-finding. For P processors, we construct a product of Johnson graphs:

$$G = \underbrace{J(\mathcal{X}, R_x) \times \cdots J(\mathcal{X}, R_x)}_{P \text{ copies}} \times \underbrace{J(\mathcal{Y}, R_y) \times \cdots J(\mathcal{Y}, R_y)}_{P \text{ copies}}. \tag{6.24}$$

Each vertex is a p-tuple of subsets of $\mathcal{X}$ and a p-tuble of subsets of $\mathcal{Y}$, i.e.,

$$v = (v_{x,1}, \cdots, v_{x,P}, v_{y,1}, \cdots, v_{y,P}), \tag{6.25}$$

and each subset $v_{x,i}$ has the same size R_x, and each $v_{y,j}$ has size R_y. Two vertices u and v are adjacent if $|v_{x,i} \setminus u_{x,i}| = 1$ for all i, and similarly for y. The spectral gap of a graph product is the minimum of the spectral gaps, so the spectral gap will be $\Omega(1/R)$, for $R = \max\{R_x, R_y\}$. Each processor stores a separate set, for RP total memory.

We set marked vertices to be the those where there is some claw in *any* of the subsets. That means there is some i and j, with $x \in v_{x,i}$ and $y \in v_{y,j}$ such that $f(x) = g(y)$. This makes the fraction of marked vertices ϵ equal to $\Omega((RP)^2/|\mathcal{X}||\mathcal{Y}|)$. Hence, the cost per processor (letting $X = |\mathcal{X}|$ and $Y = |\mathcal{Y}|$) is

$$\mathsf{S}_{JMW} + \frac{\sqrt{XY}}{RP}\left(\sqrt{R}\mathsf{U}_{JMW} + \mathsf{C}_{JMW}\right). \tag{6.26}$$

The naive parallelization gives the same spectral gap, but ϵ changes to $\Omega(R^2P/XY)$, so the cost per processor ends up as

$$\mathsf{S} + \frac{\sqrt{XY}}{R\sqrt{P}}\left(\sqrt{R}\mathsf{U} + \mathsf{C}\right). \tag{6.27}$$

An update step in the JMW parallelization involves insertion and deletion into $2P$ Johnson graphs, each with size R, but this is the *entire* update step for the naive parallelization. A similar argument applies to the set-up step; the check step for naive parallelization is $O(1)$. Thus, each graph operation in the JMW parallelization is at least as expensive as naive parallelization.

The check step is much more difficult for JMW because when each processor adds a new element to its list, it needs to check $P - 1$ other lists to see if it forms a claw with any of the other lists. We will assume that for the check step, they work together as a sorting network and sort all the lists together, treating them as a single list. Ignoring latency this gives a $O(PR\log(PR))$ gate cost to the check step, with depth $O(\log(PR))$.

This gives a total depth of roughly

$$\log R + \frac{\sqrt{XY}}{P\sqrt{R}}\log R + \frac{\sqrt{XY}}{PR}\log(PR) \tag{6.28}$$

The middle term will dominate. To fit in a given depth D, we need $P = \sqrt{XY/RD^2}$. Substituting for gate cost over all processors gives

$$\frac{\sqrt{XYR}}{D}\log R + \sqrt{XYR} + \sqrt{XY}\log(PR) \tag{6.29}$$

Hence, it parallelizes well enough that decreasing depth adds no extra cost. However, this was a very approximate derivation. Because JMW relies on highly connected quantum processors, we will ignore it in favour of the Multi-Grover search, which also uses sorting networks and gives a similar cost.

6.6 Multi-Grover Search

In "Efficient Distributed Quantum Computing" [7], Beals et al. use a computational model where many small quantum computers interact in a "low-degree" graph. They show that this is equivalent *in depth* to a QRAM model, or to a circuit model where bounded fanin gates can be simultaneously applied to any non-overlapping sets of qubits in memory.

There are two main issues with this model:

- They use a hypercube as a "low-degree" graph, where the degree of each vertex grows as $O(\log N)$ with the number of vertices. As Section 4.1.2 argued, this is not embeddable into Euclidean space.
- They focus on depth, not on gate count. This leads to some bizarre algorithms that seem to have extraordinarily good run-times, but careful analysis shows enormous gate costs.

Specifically, they give an algorithm for element distinctness that is easily adapted to claw-finding. The algorithm resembles the BHT algorithm [14] for collision finding.

The basic idea is that with P processors, each processor can store some piece of data while simultaneously searching within the data that they themselves store, using the multi-memory access circuit from Section 4.1.2. This means that if each processor i stores $(x_i, f(x_i))$, then it can also compute $(y_i, g(y_i))$, and each one can simultaneously search for some j such that $(y_i, g(y_i)) = (x_j, f(x_j))$.

Let S be the set $\{(x_1, f(x_1)), \cdots, (x_P, f(x_P))\}$ of tuples stored by each processor. The method just described allows each processor to compute the following boolean function:

$$g(y) = \begin{cases} 1 & \text{, there exists some } i \text{ such that } g(y) = f(x_i) \\ 0 & \text{, otherwise.} \end{cases} \tag{6.30}$$

Hence, each processor can perform a Grover search on the set $\mathcal{Y}$ for such a claw. In fact, they can divide the set $\mathcal{Y}$ into $|\mathcal{Y}|/P$ pieces, and each one will search that subset. If one half of the claw is in the set S a Grover search will require $\sqrt{|\mathcal{Y}|/P}$ iterations to find it.

The function g requires P calls to the oracle and then one sort. If S is the sorting cost, the cost of g is

$$\mathsf{S} + P\mathsf{F} \tag{6.31}$$

so, given some initial list S, the cost of Grover search on $\mathcal{Y}$ for a claw in S will cost

$$O\left(\sqrt{\frac{|\mathcal{Y}|}{P}}(\mathsf{S}+P\mathsf{F})\right). \tag{6.32}$$

If each processor picks a random x, the probability that one half of the claw is in S will be $P/|\mathcal{X}|$. We can then wrap the *entire* algorithm in another Grover search, where we search through these random sets. This will require $\sqrt{|\mathcal{X}|/P}$ repetitions of the Grover search subroutine. Hence, the total cost will be

$$O\left(\frac{\sqrt{|\mathcal{X}||\mathcal{Y}|}}{P}(\mathsf{S}+P\mathsf{F})\right), \tag{6.33}$$

where S is the cost of the sort used in the middle of the algorithm. This uses Cost 4.3 for the sort cost with Pm as the size of the list to be sorting.

Cost 6.10 Claw-finding with a Multi-Grover search, with $X = |\mathcal{X}|$ and $Y = |\mathcal{Y}|$.

Model	Measure	Cost
Passive Circuit (2.4.1)	Gates	$O(\sqrt{XY}(m\log P + \mathsf{F}_G))$
Active Circuit (2.4.2)	Total	$O(\sqrt{XY}(\log^2 P\log m) + o(\sqrt{XY}\log^2 P)$
	Depth	$O\left(\frac{\sqrt{XY}}{P}(\log P\log m + \mathsf{F}_D)\right)$
	Width	$O(P(\log P + \mathsf{F}_W))$
Passive Latency (2.4.3)	Gate-Time	$O(\sqrt{XY}(P^{1/d}(P^{1/d^2} + \mathsf{F}_G))$
Active Local (2.4.4)	Total	$O(\sqrt{XY}P^{2/d+1/d^2}\log^d(XYP)) + o(\sqrt{XY}P^{2/d+1/d^2})$
	Depth	$O(\frac{\sqrt{XY}}{P}(P^{1/d}(\log m + P^{1/d^2}) + \mathsf{F}_D))$
	Width	$O(P(P^{1/d} + \mathsf{F}_W)$

When we ignore latency, multi-Grover parallelization is much more efficient than Tani's or Grover's algorithms. In passive latency, parallelization is better if the dimension is 3 or more; with active locality, parallelization is better if the dimension is at least 5.

Recall that for the sorting network used in multi-Grover, the dimension plays a big role because the depth of the sort (in gates) depends on the connectivity of the qubits, which must be embedded into some physical device. That is, one can imagine using a model without latency for Tani's algorithm, relying upon the signal speeds of the control circuitry to be so fast that they can be ignored, but it's much harder to believe that one could connect qubits into a graph with more than constant connectivity.

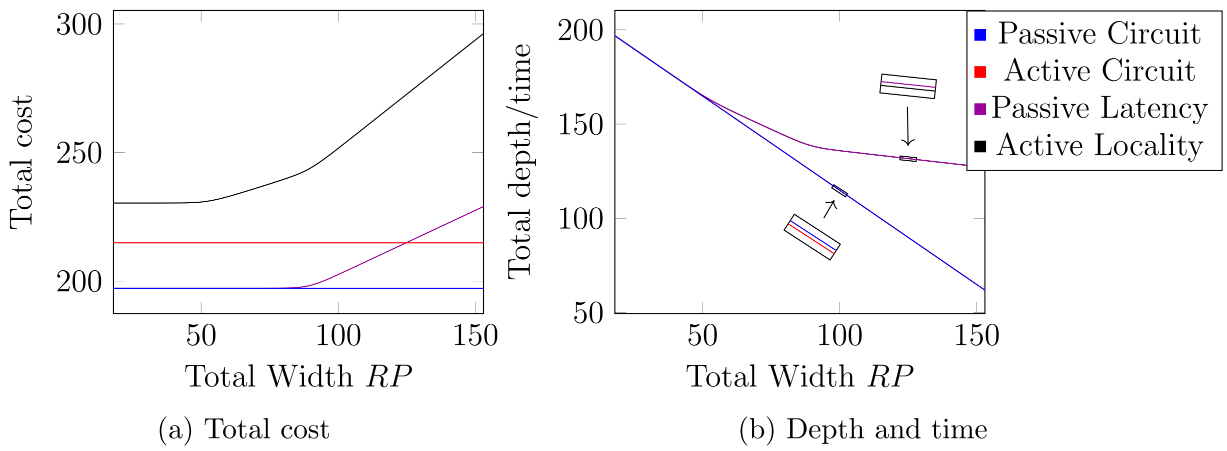


Figure 6.5: Multi-Grover search for SIKE-610, with dimension 2. Axes are in log base 2.

Figure 6.5 shows the costs and depth including both sorting and oracle costs. With latency, there is an optimal parallelization where latency costs match oracle costs. Without latency, the costs stay constant.

6.7 Comparisons

6.7.1 Tani vs. Grover

Figure 6.6 compares Tani's algorithm to Grover's algorithm. This is identical to Figure 6.4b but with Grover's algorithm added. Though this figure is specific to SIKE, the patterns in it are more general.

Grover's algorithm is more expensive. In the passively-corrected models, memory is only expensive to access, not to maintain, so Tani's algorithm provides modest improvements by balancing memory access with oracle calls. In contrast, in the actively-corrected models, it is expensive just to store the memory while the oracle computation runs, so there is less room for advantage by increasing memory, and the gap between Tani and Grover is smaller. In the QRAM model, Tani's algorithm is vastly superior.

If we set $R = 1$ in Tani's algorithm, we get a Johnson graph $J(\mathcal{X}, 1)$, which is the complete graph. Grover's algorithm is a random walk on the complete graph, so we can view Grover's algorithm as a limiting case of Tani's algorithm. Massive parallelization of small instances of Tani's algorithm is not much different than parallel Grover.

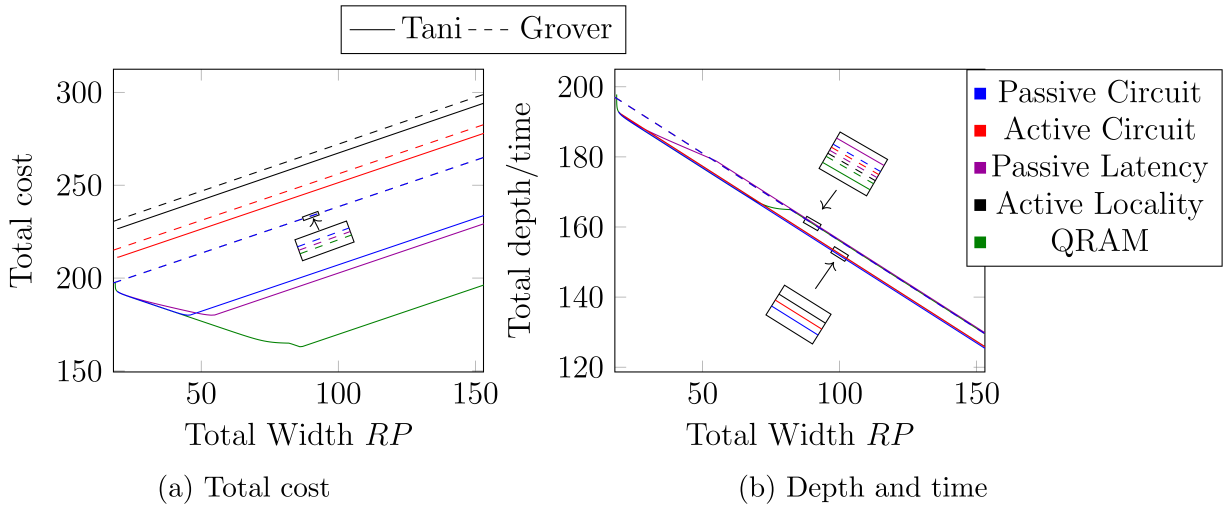


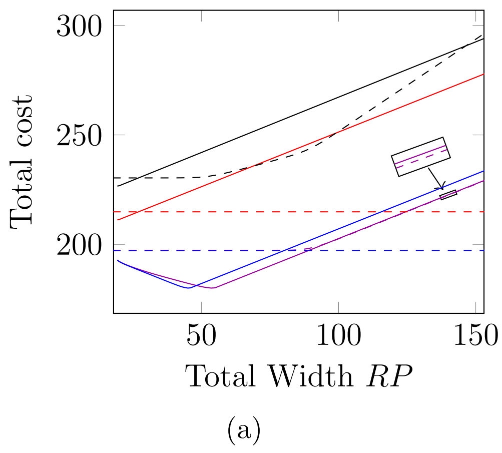
Figure 6.6: Costs of Tani's algorithm vs. Grover's algorithm for SIKE-610, with dimension 2. Axes are in log base 2.

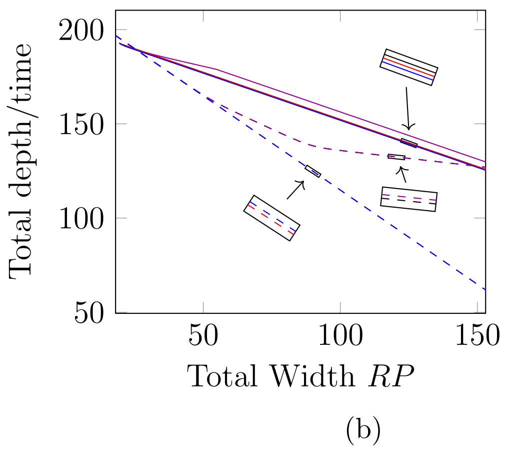
6.7.2 Tani vs. Multi-Grover

Here there is no comparison in the QRAM model, since Multi-Grover does not use any QRAM.

Figure 6.7 compares the algorithms. Like Grover's algorithm, the total cost of Tani's algorithm increases as it parallelizes. Since it must parallelize to reduce depth, the cost increases as depth decreases (6.7c). Multi-Grover parallelizes much more efficiently, so it only sees the same effect in models with latency where sorting costs much more.

We see that for passively-corrected models, Tani's algorithm has an advantage for small memory. Multi-Grover calls the oracle for every processor in every iteration, but Tani only calls it once per iteration. Hence, the increase in oracle costs precisely cancels the reductions in steps for Multi-Grover. For Tani's algorithm, increasing memory is "free" up to about 2^{45} bits, since the memory costs are still dwarfed by oracle calls, but Tani does not need to increase the number of oracle calls. Once memory insertions start to dominate





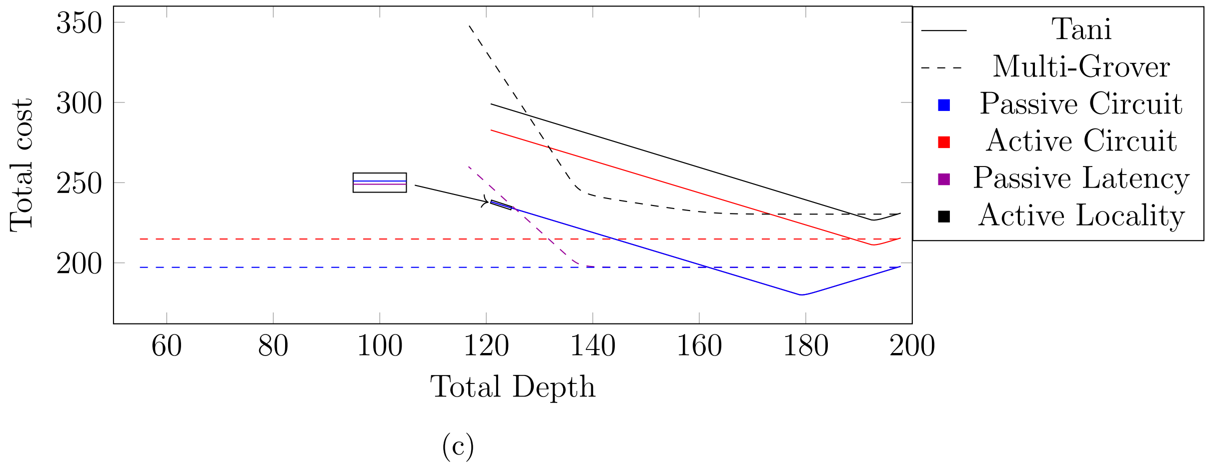


Figure 6.7: Tani and Multi-Grover for SIKE-610. The dimension d is 2. Axes are log base 2.

and Tani parallelizes, it parallelizes less efficiently than Multi-Grover and soon loses the advantage.

In actively-corrected models, Tani's algorithm almost immediately suffers increased costs from increased memory, because maintaining the memory dominates the costs. This means it is almost immediately more expensive than Multi-Grover. Accounting for latency, Tani's algorithm is eventually cheaper beacuse Multi-Grover relies on so many sorts.

We assumed neither algorithm can use more than $\sqrt{|\mathcal{X}||\mathcal{Y}|} + \mathsf{F}_W$ memory. Since Multi-Grover parallelizes better, in non-geometric models it can achieve much lower depths. At the extreme, it's roughly equal to a single oracle call and a single sort, which is essentially the optimal high-memory approach: Initialize a list of all of $\mathcal{X}$ and another list of all of $\mathcal{Y}$, compute the oracle for every element in each list, sort the lists together, and look for any collisions.

The best algorithm depends on the depth, memory, and model assumptions.

6.7.3 Quantum vs. Classical

To compare quantum and classical, we will use the best quantum algorithm and compare it to van Oorschot-Wiener, which Section 6.3 showed was the best classical algorithm. Here there is some ambiguity on how to compare, since we have underspecified the computational resources. The costs of each algorithm vary greatly depending on the amount of parallelization, the maximum depth allowed, and for VW, the amount of shared memory.

One way to make the comparisons "fair" is to assume that if a quantum algorithm uses M qubits of memory, then VW has access to M bits of memory, since the quantum memory will have some classical control that could store the data for VW. In Jaques and Schanck [34], we also assumed that VW has one processor for each qubit of memory. We will continue to make this assumption for all models except the QRAM.

To compute the costs of parallel VW with latency, we used Equation 6.16 directly, taking whichever value of θ from Equation 6.17 produces the lower total cost. Figure 6.8a shows the results and compares them to the quantum algorithms. This figure shows that increased memory *reduces* the total cost of VW, while it increases the cost of Tani's algorithm and does not change the cost of Multi-Grover. The depth of each algorithm decreases when they use more hardware (Figure 6.8b), but for Tani's algorithm, the depth reduction comes at a cost of increased gates. For VW, reduced depth also reduces the total amount of computation. We can also see that latency has a negligible impact on VW until it uses $\approx 2^{140}$ bits of shared memory.

Figure 6.8a may seem odd, since costs increase with the available resources. Could we not simply choose to ignore the extra memory for Tani's algorithm to minimize costs? We could, but the underlying assumption is that we want to minimize depth. Hence, depth decreases as costs increase. We can see that in Figure 6.8c.

Let $X = |\mathcal{X}|$ and $Y = |\mathcal{Y}|$. A single instance of Tani's algorithm scales as $O(\sqrt{XY})$ while VW scales as $O((XY)^{3/4})$, which explains why Tani's algorithm has a lower depth when parallelization is limited. By assumption, increased "hardware" means both processors and memory increase for VW. This means if Q is the total number of qubits, Tani's algorithm decreases in depth by $O(Q^{1/2})$ while VW decreases in depth by $O(Q^{3/2})$. Hence,

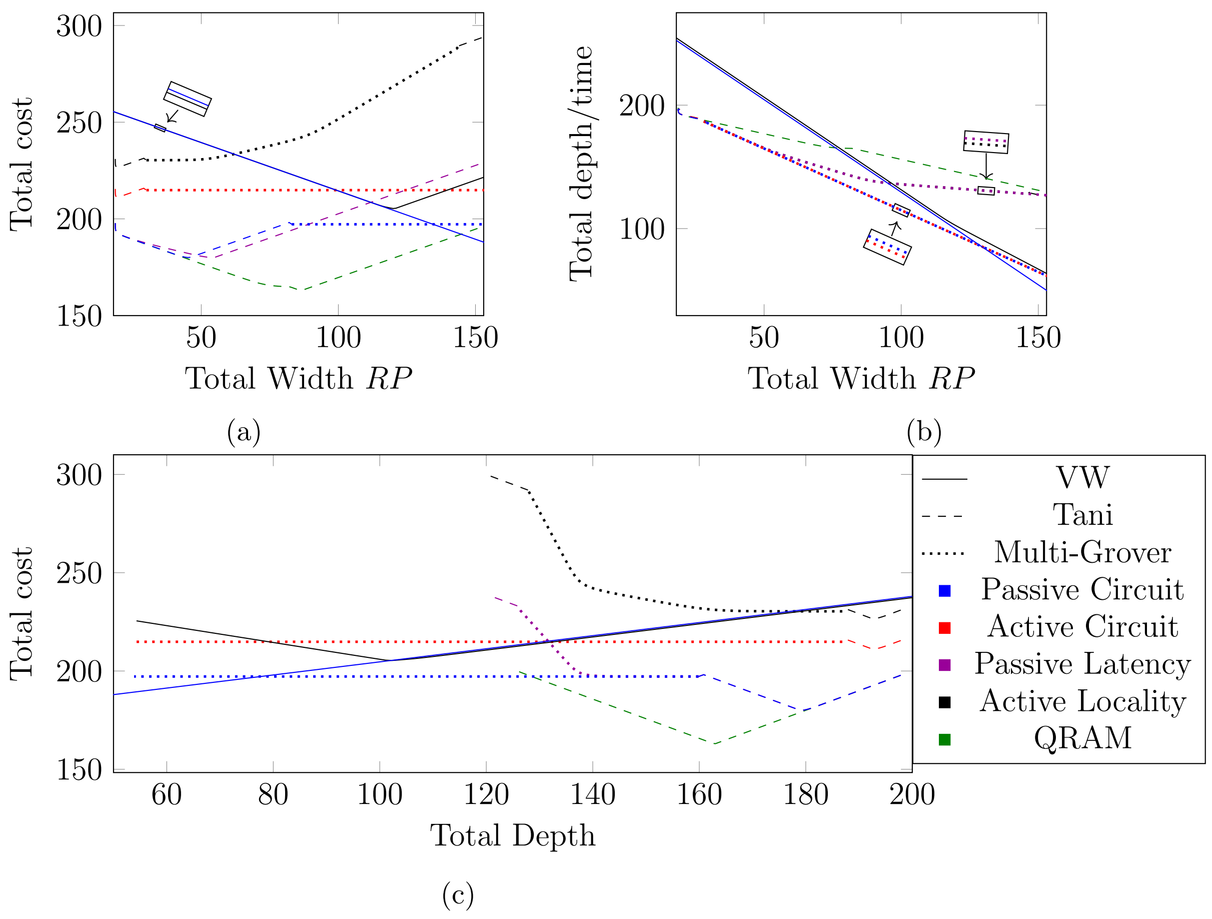


Figure 6.8: Classical vs. quantum algorithms for SIKE-610. The dimension d is 2. Axes are log base 2.

VW becomes faster than Tani's algorithm when there is plenty of hardware available. In general, the crossover will be where the following equation holds:

$$\underbrace{\frac{(XY)^{3/4}}{Q^{3/2}}}_{\text{VW depth}} \approx \underbrace{\frac{(XY)^{1/2}}{Q^{1/2}}}_{\text{Tani depth}} \tag{6.34}$$

which occurs when $Q = (XY)^{1/4}$. This is approximate since we ignored logarithmic factors. If Q is below this threshold, then Tani's algorithm requires at least $(XY)^{1/4}$ depth and VW requires at least $(XY)^{3/8}$ depth.

Similarly, Multi-Grover scales as $O(\sqrt{XY})$ and the depth decreases by $O(Q)$. This is still slower than VW; however, if we solve the same equation,

$$\underbrace{\frac{(XY)^{3/4}}{Q^{3/2}}}_{\text{VW depth}} \approx \underbrace{\frac{(XY)^{1/2}}{Q}}_{\text{Multi-Grover depth}}, \tag{6.35}$$

we conclude that the crossover, where VW becomes faster than Multi-Grover, occurs when $Q = (XY)^{1/2}$. This is an absurd memory requirement. Many of the assumptions we used in the analysis start to break down with that much memory.

One last perspective on these costs is to compare the total cost to the depth, shown in Figure 6.7c. This figure asks: How much hardware would be necessary to fit in the required depth, and hence what is the total cost? This shows that costs decrease with depth or stay the same for quantum algorithms, but increase with depth for VW.

Multi-Grover outperforms VW in the passive circuit model and for high depth in the active circuit model. VW is better in the active locality model, and at low depths is better than the passive latency model. VW is better than QRAM, because the QRAM model assumes a quantum processor is very expensive, but a single qubit has the same cost as a classical processor.

Our final conclusions about these algorithms (P is the total hardware available):

1. With very limited memory, Tani's algorithm is the best, with cost scaling approximately as $O((|\mathcal{X}||\mathcal{Y}|)^{1/2})$.
2. If memory is available and latency is not an issue (i.e., one can connect one's qubits in a hypercube) then Multi-Grover is the best algorithm. The cost scales as $O((|\mathcal{X}||\mathcal{Y}|)^{1/2}\log P)$.
3. If latency is an issue, Multi-Grover generally performs better until the sort costs dominate the oracle costs.
   (a) With actively-corrected memory, VW is better, with cost scaling roughly as $O((|\mathcal{X}||\mathcal{Y}|)^{3/4}P^{-1/2})$.
   (b) With passively-corrected memory (including QRAM), VW is better for low depth, but quantum algorithms, either Tani or Multi-Grover, perform better with low memory.
   (c) If one has less than $O((|\mathcal{X}||\mathcal{Y}|)^{1/4})$ hardware and less than $O((|\mathcal{X}||\mathcal{Y}|)^{1/4})$ time, the problem cannot be solved.

That third conclusion is something we should really emphasize. Consider SIKE-751. To break it with claw-finding, one needs either 2^{94} time (roughly the age of the universe in nanoseconds) or 2^{94} hardware (roughly the surface area of the earth in square micrometers). So covering the Earth's surface in 1 micrometer^2-sized qubits, running Tani, VW, or Multi-Grover for the age of the universe, would just barely solve CSSI for SIKE-751.

Smaller parameters have feasible limits, like 2^{54} for time and hardware for SIKE-434, though the total cost is still high at roughly 2^{108} RAM operations.

Chapter 7

Security of SIKE and SIDH

The United States' National Institute of Standards and Technology (NIST) is standardizing post-quantum cryptography and is considering SIKE. Hence, we need analyses of SIKE's security. The goal of the previous analyses was to present a more realistic picture of attacks on isogeny-based cryptography.

"Security" is a nebulous idea, since there are many avenues of attack. There may be implementation issues that allow side-channel attacks, fault attacks, or attacks on the protocols. The point when an adversary would use their quantum computer to attack SIKE will probably be *after* they have tried and failed to use any simpler and cheaper attack. Since our focus is quantum costs, we will assume all the implementation issues are fixed and that an adversary is forced to solve the Computational Super-Singular Isogeny problem (CSSI, Problem 5.2.1) if they want to break SIKE.

This chapter will first address why other potential attacks do not work for SIKE and SIDH, thus making claw-finding the best attack. Since the classical and quantum algorithms have different "cost landscapes" of hardware, depth, parallelization, and cost, we consider NIST's security definitions and conclude that the appropriate way to give costs is *relative* to other algorithms. Combining the results of all the previous chapters, we conclude that:

1. The cost of breaking SIKE or SIDH with claw-finding grows on the order of $p^{1/4}$;
2. In a realistic computational model, SIKE-434 gives as much security as AES-128 or SHA-256.

These are lower than previous estimates, meaning one can use smaller keys – and hence improve performance – without sacrificing security.

7.1 Isogeny-specific Attacks

Chapter 6 only focused on generic claw-finding attacks, and ignored the structure of the isogeny problem. There have been a few attempts to provide a specific quantum attack to find secret isogenies; we list them here.

7.1.1 Ordinary Isogenies

Childs, Jao, and Soukharev (CJS) [21] provided a subexponential time and subexponential space quantum algorithm to find hidden *ordinary* isogenies. They frame the problem as an abelian hidden shift problem. They use either Kuperberg's [41] or Regev's [49] algorithm to solve it. Since their paper, Kuperberg refined his method with a more general algorithm that subsumes Regev's technique [40]. There is an oracle-call vs. space tradeoff, where the space ranges from polynomial to subexponential. The number of oracle calls is superpolynomial under any parameterization.

The oracle in question must compute an isogeny from its representation as an arbitrary element of the ideal class group. CJS provided a method to compute this, but it used subexponential time and space.

There is a new protocol known as CSIDH [18] that uses supersingular curves over $\mathbb{F}_p$. These have similar algebraic structure to ordinary curves, so CJS is the best attack. This resulted in a sudden surge of results improving on the isogeny oracle. Two methods have exponential asymptotic complexity but are fast for practical key sizes [12, 31], and another method optimizes constants [10]. The exponential component is to solve a lattice problem, but the lattices for concrete CSIDH instances are low-dimensional. Supersingular curves over $\mathbb{F}_{p^2}$ lack some of the algebraic structure of ordinary curves, so the CJS attack will not work on SIDH.

7.1.2 Supersingular Isogenies

Biasse, Jao, and Sankar (BJS) [11] provided a quantum method to compute a supersingular isogeny over $\mathbb{F}_{p^2}$ between some E_1 and E_2. Their method uses a Grover search to find an isogeny from E_1 to some curve defined over $\mathbb{F}_p$, denoted E_1'. A similar isogeny can be found between E_2 and some E_2'. Then the CJS algorithm can find an isogeny between E_1' and E_2'. Composing these three isogenies, as in Figure 7.1, gives an isogeny between E_1 and E_2.

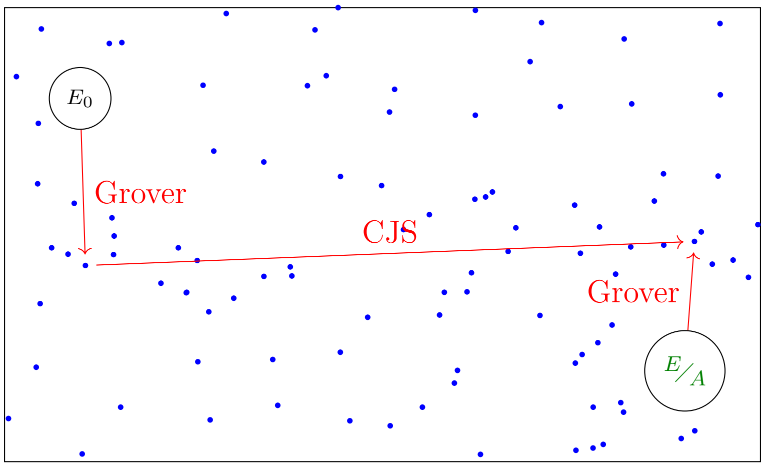


Figure 7.1: The BJS algorithm. The rectangle represents all supersingular curves over $\mathbb{F}_{p^2}$, and the blue dots represent curves defined over $\mathbb{F}_p$.

There are two problems with this approach. The first is that this technique finds *some* isogeny between E_1 and E_2, but not necessarily the secret isogeny of a specified degree that would solve CSSI. This issue may be inconsequential [39].

More pressing is the Grover search. There are about $p/12$ supersingular elliptic curves over $\mathbb{F}_{p^2}$, and there are roughly $p^{1/2}$ supersingular curves over $\mathbb{F}_p$. Thus, the fraction of curves that the Grover step needs to find is $O(p^{-1/2})$. In CSSI, if one searches the $p^{1/2}$ isogenies reached by paths of length $\frac{1}{2}\log p$ from the starting curve E_1, precisely one path should be the correct isogeny, meaning the fraction of correct curves is also $O(p^{-1/2})$. Thus, the number of iterations in a Grover search will be the same for these two searches. The oracle cost to compute a single isogeny will be almost exactly the same. Hence, the BJS algorithm starts with a Grover search that has the same cost as simply solving the CSSI problem directly!

There are no other isogeny-specific quantum algorithms that do better, or even close, to Grover's algorithm. It remains an open question whether this is because CSSI is genuinely a hard problem and will remain exponential, or whether it is simply because few people have enough expertise in both isogenies and quantum algorithms to solve the problem.

From this, we will conclude that for now generic claw-finding attacks are the best attack against CSSI, so they will define the security of SIDH and SIKE.

7.2 Security Definitions

7.2.1 Cost Parameters

The conclusions at the end of Chapter 6 are fairly compact and complete, but they leave a hard task for someone trying to decide on a secure set of parameters for a cryptosystem. What computational model is the most realistic? How much hardware will an adversary have?

The reason this is more difficult for quantum algorithms may be that many cryptographically relevant classical algorithms parallelize perfectly. This includes:

- a brute force key search,
- van Oorschot-Wiener applied to collision finding,
- VW applied to discrete log,
- the number field sieve (more or less).

Hence, if we conclude that the "complexity" of an attack is 2^{128}, then we know we need to do 2^{128} operations somehow, whether we spread that out over many machines or use only a few machines for a very long time. Thus cryptographers seem to ignore this issue, and simply try to hit benchmarks of complexity. In contrast, a quantum algorithm like Grover's has a gate count that varies with the number of machines.

One reason that cryptographers might ignore the classical trade-off is that the attacks all have the *same* tradeoff. Thus, if you have already chosen a 128-bit symmetric cipher, then it's only reasonable to choose a 256-bit hash function, a 256-bit elliptic curve, or a 3072-bit RSA key. This consistency has two nice features:

- If you've already chosen one component of a cryptosystem, and it has n bits of security, you only need to choose n bits of security for your other systems, since any attack powerful enough to break one will be powerful enough to break the other.
- If you want to compare two cryptosystems, you only need to compare them at the same "complexity", for the same reason.

Moving forward, we think this consistency is the important feature we should retain to compare systems with different attack landscapes.

7.2.2 NIST's Approach

Consistency is an important feature and seems to be the philosophy behind NIST's security levels. They define each security level as

> Any attack that breaks the relevant security definition must require computational resources comparable to or greater than those required for [breaking various protocols].

Each security level is defined by which protocol is the benchmark of computational resources:

Level 1: Key recovery for AES-128

Level 2: Collision for SHA-256

Level 3: Key recovery for AES-192

Level 4: Collision for SHA-384

Level 5: Key recovery for AES-256

NIST claims that these are in increasing order, which says something about their process. BHT [14] give a quantum algorithm for collision-finding of an n-bit hash that runs in time $O(2^{n/3})$ but uses space $O(2^{n/3})$. This has been controversial (q.v. [8]), and if we focused only on run-time, this fits NIST's ordering: the required time for each task is then 2^{64}, 2^{85}, 2^{96}, 2^{128}, and 2^{128}.

However, in terms of gate cost without QRAM, BHT's algorithm costs approximately $O(2^{n/3})$. This means level 2 requires 2^{170} gates, while level 3 only requires 2^{96} gates (ignoring the gates to actually compute AES or SHA). One might conclude that NIST is thus assuming an adversary will have QRAM. However, NIST may be totally agnostic on the issue, since they go on to define each category in terms of gates. Categories 2 and 4 are *only defined* by classical gates. Our guess is that NIST did not want to take a stance on whether BHT is a realistic threat or not.

Category	Computational Task	Quantum Gates	F_G	F_D	Classical Gates	H
1	AES-128	$2^{170}/\text{MAXDEPTH}$	$2^{23.8}$	$2^{18.1}$	2^{143}	2^{25}
2	SHA-256	–	–	–	2^{146}	2^{28}
3	AES-192	$2^{233}/\text{MAXDEPTH}$	$2^{24.4}$	$2^{18.3}$	2^{207}	2^{25}
4	SHA-384	–	–	–	2^{210}	2^{28}
5	AES-256	$2^{298}/\text{MAXDEPTH}$	$2^{25.0}$	$2^{18.5}$	2^{272}	2^{25}

Table 7.1: Gate counts for NIST categories. The quantum gate counts and depth for individual AES calls (F_G and F_D) come from Grassl et al. [28] and we deduce the classical gate counts below.

7.2.3 NIST Category Explanations

It's worth explaining how NIST derived their categories. They only cite Grassl et al. [28], so the following is a reconstruction of their logic. Table 7.1 shows their figures. The "MAXDEPTH" is the value of the maximum depth of a quantum circuit, which they assume will be either 2^{48}, 2^{64}, or 2^{96}.

The classical gate counts are straightforward. For n-bit AES, the gate counts are 2^{n+25}, suggesting NIST envisions a brute force attack that requires 2^{25} gates for a single AES-evaluation. For an n-bit hash, the gate counts are $2^{n/2+18}$. This scales like van Oorschot-Wiener collision finding, where a hash computation requires 2^{18} gates. Note that this ignores any shared memory available, since VW doesn't benefit from extra memory if it only needs to find a single collision.

For the quantum figures, we have to consider parallel Grover. If we want to search a space of size N, and computing a single oracle call costs F_G, then we need $\sqrt{N}\mathsf{F}_G$ gates. If we parallelize to P machines, each machine will search a space of size N/P and thus it will require $\sqrt{N/P}\mathsf{F}_G$ gates. Counting the gates over all the machines totals $\sqrt{PN}\mathsf{F}_G$ gates.

The total depth for each machine will be $\sqrt{N/P}\mathsf{F}_D$, where F_D is the depth of a single oracle call. This will be the total depth of the algorithm. If we assume that, without parallelization, the depth would be greater than MAXDEPTH, then we are forced to parallelize. We want to parallelize as little as possible to fit in the depth limit, so we want

$$\sqrt{\frac{N}{P}}\mathsf{F}_D = \text{MAXDEPTH}. \tag{7.1}$$

Solving this equation gives $\sqrt{P} = \frac{\sqrt{N}\mathsf{F}_D}{\text{MAXDEPTH}}$. We can substitute that into the total gate count to get

$$\frac{N\mathsf{F}_G\mathsf{F}_D}{\text{MAXDEPTH}}. \tag{7.2}$$

If we use the values from Grassl et al. [28] to find F_G and F_D for AES, we get $2^{169.9}/\textsc{maxdepth}$, $2^{234.7}/\textsc{maxdepth}$, and $2^{299.6}/\textsc{maxdepth}$ for AES 128, 192, and 256. Hence, this was probably how NIST came up with these results.

This means that, in NIST's view, quantum algorithms will be depth-limited.

7.2.4 Reductions for SIKE

If we have SIKE-n for some prime p of n bits then this induces a claw-finding problem with $|\mathcal{X}| = |\mathcal{Y}| = 2^{n/4}$ and $|S| = 2^{n/2}$. By Lemma 6.1.1, solving this claw-finding problem is as hard as finding a collision (likely unique) for a hash function $h : V \to W$ with $|V| = 2^{n/4+1}$ and $|W| = 2^{n/2}$. Since restricting the *input* size cannot increase the difficulty (we can just ignore some inputs), this means that SIKE-n is at least as hard as collision finding for an $n/2$ bit hash. Hence, we can definitely meet NIST categories 2 and 4 with primes of length 512 and 768, respectively.

Conversely, we can only reduce a golden collision search to claw-finding, not a generic collision. However, security levels 2 and 4 are defined in terms of a generic collision. We can apply Lemma 6.1.3 but this does not give a tight bound.

The concrete analysis shows that the first bound, that SIKE-n is as hard as collisions for an $n/2$-bit hash, is essentially tight, up to differences in difficulty between computing a hash and computing isogenies.

7.3 SIKE Security Tables

For each cost model, we will provide a table showing the cost of each algorithm for collision finding for SHA-256 and SHA-384. This will provide the benchmarks for NIST categories 2 and 4. For categories 1, 3, and 5, we will compute the cost of a Grover search, based on the calculations described in the last section.

To properly analyze NIST categories 2 and 4, we would need to do the same analysis as Chapter 6, but applied to collision finding. Instead, we will simply re-use the analysis but with $|\mathcal{X}| = |\mathcal{Y}| = N^{1/2}$, when the hash's output space has size N. This works for Tani's algorithm and Multi-Grover search – in essence, it turns Tani's algorithm into Ambainis' algorithm – but it does not work for VW, which only needs *one* collision. Equation 6.13 gives the *average* cost per collision, but the cost for a single collision will be different. We

Prime Length	Model	NIST Level	Best Attack
434	Passive Circuit	1	VW/Multi-Grover
	Active Circuit	1	VW
	Passive Latency	2	VW/Multi-Grover
	Active Locality	2	VW
	QRAM	2	VW
610	Passive Circuit	3	VW/Multi-Grover
	Active Circuit	2	VW
	Passive Latency	3	VW
	Active Locality	3	VW
	QRAM	3	VW
751	Passive Circuit	4	VW/Multi-Grover
	Active Circuit	4	VW
	Passive Latency	5	VW
	Active Locality	5	VW
	QRAM	5	VW

Table 7.2: Security levels of SIKE under different choices of primes and cost models.

will need to first fill the memory, which has size M. Thus, the total cost will be

$$\frac{M}{\theta} + \frac{N\theta}{M} + \frac{1}{\theta} \tag{7.3}$$

hash evaluations. Optimizing θ gives $\theta = M/\sqrt{N}$, and substituting gives a total cost of $\sqrt{N}$. Hence, we will simply use NIST's figures for VW.

Appendix B gives tables detailing the results for each model. Table 7.2 summarizes the results.

The main difference in security is between models with or without latency. The security benchmarks do not suffer any losses from latency, because VW does not need high memory to find a single collision, and key search parallelizes without communication. However, latency has a big impact in the high-memory attacks against CSSI. Hence, accounting for latency raises the security of CSSI *relative to* the security of the benchmarks.

Thus, it seems reasonable to recommend SIKE-434, -610, and -751 for levels 1, 3, and 4 (respectively) if we optimistically assume latency is no issue, and for levels 2, 3, and 5 otherwise.

NIST Level	SIKE Prime Length Recommendation		
	This Thesis		SIKE Submission [32]
	No Latency	Latency	
1	434	–	503
2	610	434	–
3	–	610	751
4	751	–	–
5	–	751	964

Table 7.3: Recommendations for SIKE prime lengths for NIST's security levels.

7.3.1 Previous Analyses

The original description of SIDH used Tani's algorithm as the best attack and hence used a 768-bit prime for 128 bits of security. In this sense, AES-256 only has 128 bits of security because of Grover's algorithm, so this nearly matches our conclusions here. The SIKE submission (shown in Table 7.3) also used Tani's algorithm and compared it to a single instance of Grover's algorithm. They left substantial safety margins, and if they "tightly" matched the security levels, they would have recommend primes of 389, 583, and 778 bits, respectively.

These analyses gave similar recommendations to mine because they used lower costs for both the attack *and* the benchmark for the attack. Grover's algorithm becomes substantially more expensive under depth limits because it parallelizes so badly, so our analysis had a higher security threshold for the NIST levels than the SIKE submission. We included Multi-Grover and VW, which do better under depth limits than Grover, reducing the *relative* security of SIKE.

For absolute security, our most conservative analysis gave 223 "bits of security" to SIKE-751, though "bits of security" is undefined without specifying limits for either processors, memory, or depth. One only needs a 434 bit prime to achieve 128 bits of security, which is 42% smaller than the previous 768 bit recommendation.

Adj et al. [3] consider just classical attacks, and give a fixed memory limit of 2^{80} words for VW. They also used "bits of security" rather than comparing to the NIST categories. The final conclusions end up essentially the same.

Most of this thesis came from Jaques and Schanck [34]. That paper ignored Multi-Grover because of the locality issues in a sorting network, but it made much more conservative assumptions about the cost of computing an isogeny. That is, in this thesis we traded realism in the search algorithm for realism in the oracle cost. The results are mostly the same, though the geometric models are likely more realistic in both aspects and the security is correspondingly higher.

7.3.2 Discussion

It's worth noting that most of the attacks are hopelessly unrealistic. In the most realistic model — active locality — with a max depth of 2^{64}, VW needs 2^{116} bits of memory to break SIKE-434, which is absurd. Of course, with max depth 2^{96}, it "only" uses 2^{79} bits of memory — but then the depth is "the approximate number of gates that atomic scale qubits with speed of light propagation times could perform in a millennium" [47].

In short, all of the calculations in Appendix B show attacks that are frankly ridiculous. Even with extremely optimistic assumptions about the scale and speed of future qubits, the attacks would need planet-sized networks of quantum computers running for hundreds of years. It's reasonable, though somewhat bold, to say that none of the attacks in this thesis will ever break SIKE-610. A more likely threat is an improved algorithm. None of the attacks in this thesis use the isogeny structure, and the CSSI problem (originating with [19]) has only withstood 13 years of analysis. Further, this chapter's assumption may be wrong and SIDH and SIKE may be easier than CSSI.

Conclusions and Open Problems

We showed that most previous analyses underestimated the security of SIDH, but we suspect other cryptosystems understimated their security as well. The “best” attack on code-based cryptography is quantum information set decoding [37], which also uses random walks on a Johnson graph. Quantum lattice sieving [42] is the best attack on lattice-based cryptography and it too uses exponential memory. If we can confidently increase our security estimates for these schemes, we can shrink the parameters and improve performance.

The cost of the claw-finding problem scales as $O(\sqrt{XY})$ for all of the quantum and classical algorithms we analyzed. However, these algorithms were designed with other cost metrics in mind; there may be other algorithms that perform better under the metrics we used. In a query model, it’s pointless to trade total iterations for query costs in each update operation; however, in other models, this would permit optimal parameterizations with more expensive memory operations. Using distinguished points in Tani’s algorithm may provide such a tradeoff.

As far as we are aware, all the quantum random walk search algorithms use Johnson graphs. Since query costs drastically underestimate memory access costs, in other cost models these algorithms will probably lose their advantage. Further work could determine which algorithms still offer an improvement over classical algorithms, and could try to find random walk applications that do not use a Johnson graph.

Most of our conclusions sprang from an assumption that quantum random access memory is expensive. This is true if memory is actively-corrected or if the random access gate must be constructed from bounded fan-in gates; however, either assumption might be wrong. Further work could prove that passively-corrected quantum memory is impossible in greater dimensions or with larger families of error-correcting codes. There is also no work that we are aware of that tries to quantify the computationally difficulty of engineering precise Hamiltonians to perform complicated operations like memory access. This is essentially the opposite approach of quantum chemical simulations. Lower bounds on this problem would give more weight to our assumptions on memory access.

The models we used were ad-hoc and there is substantial room for more rigour and justification. As a basic example, we equated latency with gate costs, so the time to send a signal past one qubit was the same as the time to perform a gate on that qubit. The difference between these will be a constant factor, but the constant may be large enough to affect non-asymptotic analyses.

Further, the algorithms we analyzed used quantum processors that were either highly connected and communicating very frequently (Multi-Grover) or had no quantum communication (parallel Grover and parallel Tani). An algorithm in between these two extremes would need a model with a more precise description of quantum communication between processors. The equivalence between the Distributed Quantum Computing, circuit, and QRAM models [7] only holds under a depth metric and non-Euclidean physical layouts.

As quantum cryptanalysis becomes more important for real parameter choices, we need to take a closer look at the results from complexity theory, which often hide aspects of the computation that are irrelevant for the theory, but which will be important in practice. We need more quantum algorithm analysis at a medium level of realism, where we can trust the conclusions to hold under many possible routes of quantum architectures, but which include as many costs as we can.

References

[1] S. Aaronson, D. Grier, and L. Schaeffer. The classification of reversible bit operations. arXiv:1504.05155, 2015. available at https://arxiv.org/abs/1504.05155.

[2] G. Adj, O. Ahmadi, and A. Menezes. On isogeny graphs of supersingular elliptic curves over finite fields. *Finite Fields and Their Applications*, 55:268 – 283, 2019.

[3] G. Adj, D. Cervantes-Vázquez, J.-J. Chi-Domínguez, A. Menezes, and F. Rodríguez-Henríquez. On the cost of computing isogenies between supersingular elliptic curves. In *Selected Areas in Cryptography – SAC 2018*, pages 322–343. LNCS 11349.

[4] R. Alicki, M. Horodecki, P. Horodecki, and R. Horodecki. On thermal stability of topological qubit in Kitaev's 4d model. *Open Systems & Information Dynamics*, 17:1–20, 2010.

[5] A. Ambainis. Quantum walk algorithm for element distinctness. *SIAM J. Computing*, 37:210–239, 2007.

[6] S. Arunachalam, V. Gheorghiu, T. Jochym-OConnor, M. Mosca, and P. Varshinee Srinivasan. On the robustness of bucket brigade quantum ram. *New J. Physics*, 17(12):123010, 2015.

[7] R. Beals, S. Brierley, O. Gray, A. W. Harrow, S. Kutin, N. Linden, D. Shepherd, and M. Stather. Efficient distributed quantum computing. *Proc. Royal Soc. London A: Mathematical, Physical and Engineering Sciences*, 469, 2013.

[8] D. J. Bernstein. Cost analysis of hash collisions: Will quantum computers make SHARCS obsolete? *Workshop Record of SHARCS09: Special-purpose Hardware for Attacking Cryptographic Systems*, 2009.

[9] D. J. Bernstein, S. Jeffery, T. Lange, and A. Meurer. Quantum algorithms for the subset-sum problem. In *Post-Quantum Cryptography – PQCrypto 2013*, pages 16–33. LNCS 7932.

[10] D.J. Bernstein, T. Lange, C. Martindale, and L. Panny. Quantum circuits for the CSIDH: optimizing quantum evaluation of isogenies. In *Eurocrypt 2019.* To appear. Available at https://eprint.iacr.org/2018/1059.

[11] J.-F. Biasse, D. Jao, and A. Sankar. A quantum algorithm for computing isogenies between supersingular elliptic curves. In *Progress in Cryptology – INDOCRYPT 2014*, pages 428–442. LNCS 8885.

[12] X. Bonnetain and A. Schrottenloher. Quantum security analysis of CSIDH and ordinary isogeny-based schemes. Version 20181219:085722. Available at https://eprint.iacr.org/2018/537.

[13] M. Boyer, G. Brassard, P. Høyer, and A. Tapp. Tight bounds on quantum searching. *Fortschritte der Physik*, 46(45):493–505, 1998.

[14] G. Brassard, P. Hoyer, and A. Tapp. Quantum algorithm for the collision problem. Third Latin American Symp. on Theoretical Informatics (LATIN'98), pp. 163-169, 1998. LNCS 1380, 1997.

[15] S. Bravyi and B. Terhal. A no-go theorem for a two-dimensional self-correcting quantum memory based on stabilizer codes. *New J. Physics*, 11, 2009.

[16] K. A. Britt and T.S. Humble. High-performance computing with quantum processing units. *J. Emerg. Technol. Comput. Syst.*, 13(3):39:1–39:13, March 2017.

[17] B.J. Brown, D. Loss, J.K. Pachos, C.N. Self, and J.R. Wootton. Quantum memories at finite temperature. *Rev. Mod. Phys.*, 88:045005, Nov 2016.

[18] W. Castryck, T. Lange, C. Martindale, L. Panny, and J. Renes. CSIDH: An efficient post-quantum commutative group action. In *ASIACRYPT 2018*, pages 395–427. LNCS 11274.

[19] D. X. Charles, K. E. Lauter, and E. Z. Goren. Cryptographic hash functions from expander graphs. *J. Cryptology*, 22(1):93–113, Jan 2009.

[20] S.-T. Cheng and C.-Y. Wang. Quantum switching and quantum merge sorting. *IEEE Trans. Circuits and Systems I*, 53(2):316–325, Feb 2006.

[21] A. M. Childs, D. Jao, and V. Soukharev. Constructing elliptic curve isogenies in quantum subexponential time. *J. Math. Cryptology*, 8:1–29, 2014.

[22] R. Cleve, A. Ekert, C. Macchiavello, and M. Mosca. Quantum algorithms revisited. *Proc. Royal Society A*, 454(1969), 1998.

[23] E. Dennis, A. Kitaev, A. Landahl, and J. Preskill. Topological quantum memory. *J. Mathematical Physics*, 43, 2002.

[24] M. H. Devoret and R. J. Schoelkopf. Superconducting circuits for quantum information: An outlook. *Science*, 339:1169–1174, 2013.

[25] A. G. Fowler, M. Mariantoni, J. M. Martinis, and A. N. Cleland. Surface codes: Towards practical large-scale quantum computation. *Physical Rev. A*, 86, Sep 2012.

[26] A. G. Fowler, A. C. Whiteside, A.C. Hollenberg, and C. L. Lloyd. Towards practical classical processing for the surface code. *Phys. Rev. Lett.*, 108:180501, May 2012.

[27] V. Giovannetti, S. Lloyd, and L. Maccone. Architectures for a quantum random access memory. *Phys. Rev. A*, 78, Nov 2008.

[28] M. Grassl, B. Langenberg, M. Roetteler, and R. Steinwandt. Applying Grover's algorithm to AES: Quantum resource estimates. In *Post-Quantum Cryptography – PQCrypto 2016*, pages 29–43. LNCS 9606.

[29] S. Homer and A.L. Selman. *Computability and Complexity Theory, 2e.* Springer Science and Business Media, 2011.

[30] Micron Technology Inc. Technical note: Error correction code (ECC) in Micron single-level cell (SLC) NAND. Micron Technology Inc. technical note, 2011. available at `https://www.micron.com/-/media/client/global/documents/products/technical-note/nand-flash/tn2963_ecc_in_slc_nand.pdf`.

[31] M.J. Jacobson Jr. J.-F. Biasse, A. Iezzi. A note on the security of CSIDH. arXiv preprint, 2018. Available at `https://arxiv.org/abs/1806.03656`.

[32] D. Jao, R. Azarderakhsh, M. Campagna, C. Costello, L. De Feo, B. Hess, A. Jalali, B. Koziel, B. LaMacchia, P. Longa, M. Naehrig, J. Renes, V. Soukharev, and D. Urbanik. Supersingular isogeny key encapsulation. *Submission to NIST post-quantum project*, November 2017. Available at `https://sike.org/#nist-submission`.

[33] D. Jao and L. De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *Post-Quantum Cryptography – PQCrypto 2011*, pages 19–34. LNCS 7071.

[34] S. Jaques and J. Schanck. Quantum cryptanalysis in the RAM model: Claw-finding attacks on SIKE. Version 20190205:012259. Available at https://eprint.iacr.org/2019/103.

[35] S. Jeffery. *Frameworks for Quantum Algorithms.* PhD thesis, University of Waterloo, 2014.

[36] S. Jeffery, F. Magniez, and R. De Wolf. Optimal parallel quantum query algorithms. *Algorithmica*, 79(2):509–529, Oct 2017.

[37] G. Kachigar and J.-P. Tillich. Quantum information set decoding algorithms. In *Post-Quantum Cryptography – PQCrypto 2017*, LNCS 10346, pages 69–89. Springer.

[38] V. Kliuchnkikov, D. Maslov, and M. Mosca. Asymptotically optimal approximation of single qubit unitaries by Clifford and T circuits using a constant number of ancillary qubits. *Physical Rev. Letters*, 110, May 2013.

[39] D. Kohel, K. Lauter, C. Petit, and J.-P. Tignol. On the quaternion ℓ-isogeny path problem. *LMS J. Computation and Mathematics*, 17A:418–432, 2014.

[40] G. Kuperberg. Another Subexponential-time Quantum Algorithm for the Dihedral Hidden Subgroup Problem. In *Theory of Quantum Computation, Communication and Cryptography – TQC 2013*, LIPIcs 22, pages 20–34.

[41] G. Kuperberg. A subexponential-time quantum algorithm for the dihedral hidden subgroup problem. *SIAM J. Comput.*, 35:170–188, 2005.

[42] T. Laarhoven, M. Mosca, and J. van de Pol. Finding shortest lattice vectors faster using quantum search. *Designs, Codes and Cryptography*, 77:375–400, Dec 2015.

[43] F. Le Gall and S. Nakajima. Quantum algorithm for triangle finding in sparse graphs. *Algorithmica*, 79:941–959, Nov 2017.

[44] F. Magniez, A. Nayak, J. Roland, and M. Santha. Search via quantum walk. *SIAM J. on Computing*, 40:142–164, 2011.

[45] N. David Mermin. *Quantum Computer Science: An Introduction.* Cambridge University Press, 2007.

[46] C. Moore. Quantum circuits: Fanout, parity, and counting. arXiv preprint, 1999. available at https://arxiv.org/abs/quant-ph/9903046.

[47] National Institute of Standards and Technology. Submission requirements and evaluation criteria of the post-quantum cryptography standardization process. 2017. available at https://csrc.nist.gov/csrc/media/projects/post-quantum-cryptography/documents/call-for-proposals-final-dec-2016.pdf.

[48] R. Peierls. On Ising's model of ferromagnetism. In *Mathematical Proc. Cambridge Philosophical Society*, volume 32, pages 477–481. Cambridge University Press, 1936.

[49] O. Regev. A subexponential time algorithm for the dihedral hidden subgroup problem with polynomial space. arXiv:quant-ph/0406151, 2004. available at https://arxiv.org/abs/quant-ph/0406151.

[50] M. Roetteler, M. Naehrig, K. M. Svore, and K. Lauter. Quantum resource estimates for computing elliptic curve discrete logarithms. In *Advances in Cryptology – ASIACRYPT 2017*, pages 241–270. LNCS 10625, 2017.

[51] J. Schanck, 2019. personal communication.

[52] N. Shenvi, J. Kempe, and K. B. Whaley. Quantum random-walk search algorithm. *Phys. Rev. A*, 67:052307, May 2003.

[53] M. Szegedy. Quantum speed-up of markov chain based algorithms. In *2004 IEEE Symposium on Foundations of Computer Science*, pages 32–41, Oct.

[54] Y. Takahashi, S. Tani, and N. Kunihiro. Quantum addition circuits and unbounded fan-out. *Quantum Info. Comput.*, 10:872–890, September 2010.

[55] S. Tani. An improved claw finding algorithm using quantum walk. In *Mathematical Foundations of Computer Science – MFCS 2007*, pages 548–558. LNCS 4708.

[56] H. Thapliyal, N. Ranganathan, and R. Ferreira. Design of a comparator tree based on reversible logic. In *2010 IEEE International Conference on Nanotechnology*, pages 1113–1116.

[57] P.C. van Oorschot and M.J. Wiener. Parallel collision search with cryptanalytic applications. *J.Cryptology*, 12(1):1–28, Jan 1999.

[58] C. Zalka. Grover's quantum searching algorithm is optimal. *Physical Rev. A*, 60, Oct 1999.

APPENDICES

Appendix A

Analyses of Tani's Algorithm

These are the costs of Tani's algorithm in each model. Here I have set $R = R_x = R_y$; I omitted the middle step which shows that they will always be equal at the optimal values. For each model the cost is unwieldy, so I will analyze each individually. I further assume that $R \leq \sqrt{|X||Y|}$, and I will use X and Y to refer to the sizes of the sets X and Y.

Passive Circuit

The total cost is

$$O\left(R(m\log^2 R + \mathsf{F}_G) + \sqrt{\frac{XY}{R}}(Rm + \mathsf{F}_G)\right). \tag{A.1}$$

It's clear to see that the update steps dominate the cost. Costs will be minimized when $Rm = \mathsf{F}_G$, and after that, increasing R will increase total costs, at a rate proportional to $\sqrt{R}$.

At the cost-minimizing R, assuming $\sqrt{XY} \geq \mathsf{F}_G$, the total cost is approximately

$$O\left(\sqrt{XY\mathsf{F}_G m}\right). \tag{A.2}$$

Comparing to Cost 6.5, Tani's algorithm saves a factor of $\sqrt{\mathsf{F}_G/m}$. For something like isogeny computations, this can be substantial. Essentially, the more expensive F_G becomes, the closer we get to the idealized oracle model where Tani originally proposed the algorithm.

At the optimal value of R, Tani's algorithm scales as Grover's algorithm with respect to X and Y.

Active Circuit

The total cost is

$$O\left(R\mathsf{F}_W \log R(\log R \log m + \mathsf{F}_D) + \sqrt{\frac{XY}{R}}(Rm + \mathsf{F}_W)(\log R \log\log R + \mathsf{F}_D)\right). \quad \text{(A.3)}$$

Here, the optimal value of R is F_W/m. Assuming $\mathsf{F}_D \geq \log(\mathsf{F}_W/m)$, this gives an optimized cost of

$$O\left(\sqrt{XY\mathsf{F}_W m}\mathsf{F}_D\right). \quad \text{(A.4)}$$

Here the savings are only $\sqrt{\mathsf{F}_W/m}$ compared to Grover's algorithm. This factor is at least 1 since computing the functions must use at least m bits, so $\mathsf{F}_W \geq m$.

Passive Latency

The total cost is

$$O\left(R(R^{1/d}\log R(m + R^{1/d^2} + \mathsf{F}_G) + \sqrt{\frac{XY}{R}}(Rm + \mathsf{F}_G)\right). \quad \text{(A.5)}$$

The optimal is the same as the passive circuit, with $Rm = \mathsf{F}_G$, for an optimal cost of

$$O\left(\sqrt{XY\mathsf{F}_G m}\right). \quad \text{(A.6)}$$

Active Locality

Ignoring the logarithmic factors for error correction, which will scale in the same was as depth$\times$width, the cost would be

$$O\left(R^{1+2/d+1/d^2}\log^{d+1}(R) + \sqrt{\frac{XY}{R}}((Rm)^{1/d} + \mathsf{F}_D)(Rm + \mathsf{F}_W)\right). \quad \text{(A.7)}$$

Here the optimal will be $Rm = \mathsf{F}_W$. Note that if $(Rm)^{1/d} \geq \mathsf{F}_D$ but $Rm \leq \mathsf{F}_W$, then the right-hand term will still decrease with increasing R. At the optimal, if we assume the set-up costs are negligible and that $\mathsf{F}_D \geq \mathsf{F}_W^{1/d}$, the total cost ends up as

$$O\left(\sqrt{XY\mathsf{F}_W m}\mathsf{F}_D \log^2(\sqrt{XY\mathsf{F}_W m}\mathsf{F}_D)\right). \quad \text{(A.8)}$$

QRAM

The total cost is

$$O\left(\max\begin{Bmatrix} R(\log R + \mathsf{F}_G) + \sqrt{\frac{XY}{R}}(m^2 + \mathsf{F}_G) \\ R(R^{1/d}\log R + m + \mathsf{F}_T) + \sqrt{\frac{XY}{R}}(R^{1/d}m + \mathsf{F}_T)\end{Bmatrix}\right). \tag{A.9}$$

Since the right-hand term always decreases with increasing R, each term is minimized by setting terms of the sum equal. The gate-cost minimum is when

$$R(\log R + \mathsf{F}_G) = \sqrt{\frac{XY}{R}}(m^2 + \mathsf{F}_G). \tag{A.10}$$

Assuming $\log R \approx m$, we get a optimal R of

$$R = (XY)^{1/3}\left(\frac{m^2 + \mathsf{F}_G}{m + \mathsf{F}_G}\right)^{2/3} \tag{A.11}$$

leading to a minimum gate cost of

$$O\left((XY)^{1/3}(m^2 + \mathsf{F}_G)^{2/3}(m + \mathsf{F}_G)^{1/3}\right). \tag{A.12}$$

To minimize time is more complicated. If we assume $R^{1/d}m \leq \mathsf{F}_T$ then the minimum would be $R = (XY)^{1/4}$, which will probably contradict the assumption, unless F_T is enormous. I will ignore the edge case where $R^{1/d}\log R \leq \mathsf{F}_T \leq R^{1/d}m$, and focus on $R^{1/d}m \geq R^{1/d}logR \geq \mathsf{F}_T$. This gives us

$$R^{1+1/d}\log R = \sqrt{XY}R^{1/d-1/2}m. \tag{A.13}$$

Assuming $\log R \approx m$, we get $R = (XY)^{1/3}$. This was Tani's original parameterization, and produces a total time of

$$O\left((XY)^{\frac{d+1}{3d}}m\right). \tag{A.14}$$

Except for exceptionally high dimension d or cost F_G, the time will be higher than the gate cost, so the optimal cost for Tani's algorithm is Equation A.14 in the QRAM model.

Appendix B

Security Tables for SIKE

For each cost model, I calculate the cost of attacks on CSSI, attacks on AES, and attacks on SHA. Comparing the costs of CSSI to the other costs gives the NIST security levels.

In some models, the costs permute the order of the security levels. This means some parameters may achieve NIST level 4 but not level 3, for example. I will denote such a case as "Level 2/4", and similarly for other ambiguous levels.

Because Grover's algorithm suffers no penalties from latency, the costs of AES key search are the same in the passive circuit, passive latency, and QRAM models. I included all the tables for completeness.

Where an algorithm couldn't meet the depth limits, I included the lowest-depth parameterization, but made the font gray. I used bold font to indicate the lowest-cost attack.

For classical SHA collisions and AES, the depth and width are excluded from the table, since these can be made (almost) arbitrarily small or large as needed.

Passive Circuit Model

SHA Collision Finding

Max.	Hash	Tani			Multi-Grover			VW			NIST
Depth	Length	C	D	W	C	D	W	C	D	W	Level
48	256	203	76	141	153	48	105	**146**	–	–	2
48	384	299	108	205	217	48	170	**210**	–	–	4
64	256	203	76	141	153	64	89	**146**	–	–	2
64	384	299	108	205	217	64	153	**210**	–	–	4
96	256	182	96	100	153	96	57	**146**	–	–	2
96	384	299	108	205	217	96	122	**210**	–	–	4

AES Key Finding

Max.	AES	–			Grover			Classical			NIST
Depth	Size				C	D	W	C	D	W	Level
48	128	–	–	–	**122**	48	80	143	–	–	1
48	192	–	–	–	**187**	48	144	207	–	–	3
48	256	–	–	–	**252**	48	210	272	–	–	5
64	128	–	–	–	**106**	64	48	143	–	–	1
64	192	–	–	–	**171**	64	113	207	–	–	3
64	256	–	–	–	**236**	64	178	272	–	–	5
96	128	–	–	–	**88**	82	12	143	–	–	1
96	192	–	–	–	**139**	96	48	207	–	–	3
96	256	–	–	–	**204**	96	114	272	–	–	5

CSSI

Max.	Prime	Tani			Multi-Grover			VW			NIST
Depth	Length	C	D	W	C	D	W	C	D	W	Level
48	434	176	93	126	152	48	121	**142**	48	110	1
48	610	243	116	171	197	48	167	**187**	48	154	3
48	751	296	135	207	233	48	203	**223**	48	191	4
64	434	176	93	126	152	64	104	**148**	64	99	2
64	610	243	116	171	197	64	151	**193**	64	144	3
64	751	296	135	207	233	64	188	**228**	64	180	4
96	434	173	96	120	**152**	96	72	159	96	77	3
96	610	243	116	171	**197**	96	119	203	96	122	3
96	751	296	135	207	**233**	96	156	239	96	158	5

Active Circuit Model

SHA Collision Finding											
Max.	Hash	Tani			Multi-Grover			VW			NIST
Depth	Length	C	D	W	C	D	W	C	D	W	Level
48	256	216	76	142	153	48	105	**146**	–	–	2
48	384	313	108	206	217	48	170	**210**	–	–	4
64	256	216	76	142	153	64	89	**146**	–	–	2
64	384	313	108	206	217	64	153	**210**	–	–	4
96	256	196	96	101	153	96	57	**146**	–	–	2
96	384	313	108	206	217	96	122	**210**	–	–	4

AES Key Finding											
Max.	AES	–			Grover			Classical			NIST
Depth	Size				C	D	W	C	D	W	Level
48	128	–	–	–	**128**	48	80	143	–	–	1
48	192	–	–	–	**193**	48	144	207	–	–	3
48	256	–	–	–	**258**	48	210	272	–	–	5
64	128	–	–	–	**112**	64	48	143	–	–	1
64	192	–	–	–	**177**	64	113	207	–	–	3
64	256	–	–	–	**242**	64	178	272	–	–	5
96	128	–	–	–	**94**	82	12	143	–	–	1
96	192	–	–	–	**145**	96	48	207	–	–	3
96	256	–	–	–	**210**	96	114	272	–	–	5

CSSI											
Max.	Prime	Tani			Multi-Grover			VW			NIST
Depth	Length	C	D	W	C	D	W	C	D	W	Level
48	434	219	93	127	168	48	121	**142**	48	110	1
48	610	287	116	172	215	48	167	**187**	48	154	2
48	751	342	135	208	252	48	203	**223**	48	191	4
64	434	219	93	127	168	64	104	**148**	64	99	2
64	610	287	116	172	215	64	151	**193**	64	144	3
64	751	342	135	208	252	64	188	**228**	64	180	4
96	434	216	96	121	168	96	72	**159**	96	77	3
96	610	287	116	172	215	96	119	**203**	96	122	3
96	751	342	135	208	252	96	156	**239**	96	158	5

Passive Latency Model

SHA Collision Finding

Max.	Hash	Tani			Multi-Grover			VW			NIST
Depth	Length	C	D	W	C	D	W	C	D	W	Level
48	256	203	76	141	224	96	192	**146**	–	–	2
48	384	299	108	205	336	144	288	**210**	–	–	4
64	256	203	76	141	224	96	192	**146**	–	–	2
64	384	299	108	205	336	144	288	**210**	–	–	4
96	256	183	96	101	224	96	192	**146**	–	–	2
96	384	299	108	205	336	144	288	**210**	–	–	4

AES Key Finding

Max.	AES	–			Grover			Classical			NIST
Depth	Size				C	D	W	C	D	W	Level
48	128	–	–	–	**122**	48	80	143	–	–	1
48	192	–	–	–	**187**	48	144	207	–	–	3
48	256	–	–	–	**252**	48	210	272	–	–	5
64	128	–	–	–	**106**	64	48	143	–	–	1
64	192	–	–	–	**171**	64	113	207	–	–	3
64	256	–	–	–	**236**	64	178	272	–	–	5
96	128	–	–	–	**88**	82	12	143	–	–	1
96	192	–	–	–	**139**	96	48	207	–	–	3
96	256	–	–	–	**204**	96	114	272	–	–	5

CSSI

Max.	Prime	Tani			Multi-Grover			VW			NIST
Depth	Length	C	D	W	C	D	W	C	D	W	Level
48	434	176	93	135	190	81	163	**159**	48	116	2
48	610	243	116	180	267	114	229	**228**	48	166	4
48	751	296	135	216	329	141	282	**283**	51	205	∞
64	434	176	93	135	190	81	163	**152**	64	102	2
64	610	243	116	180	267	114	229	**221**	64	153	4
64	751	296	135	216	329	141	282	**277**	64	193	5
96	434	173	96	129	**152**	96	85	158	95	79	3
96	610	243	116	180	267	114	229	**208**	96	125	3/5
96	751	296	135	216	329	141	282	**263**	96	166	5

Active Locality Model

SHA Collision Finding											
Max.	Hash	Tani			Multi-Grover			VW			NIST
Depth	Length	C	D	W	C	D	W	C	D	W	Level
48	256	232	76	142	304	96	192	**146**	–	–	2
48	384	329	108	206	449	144	288	**210**	–	–	4
64	256	232	76	142	304	96	192	**146**	–	–	2
64	384	329	108	206	449	144	288	**210**	–	–	4
96	256	211	96	101	304	96	192	**146**	–	–	2
96	384	329	108	206	449	144	288	**210**	–	–	4

AES Key Finding											
Max.	AES	–			Grover			Classical			NIST
Depth	Size				C	D	W	C	D	W	Level
48	128	–	–	–	**142**	48	80	143	–	–	1
48	192	–	–	–	**208**	48	144	207	–	–	3
48	256	–	–	–	**274**	48	210	272	–	–	5
64	128	–	–	–	**125**	64	48	143	–	–	1
64	192	–	–	–	**192**	64	113	207	–	–	3
64	256	–	–	–	**258**	64	178	272	–	–	5
96	128	–	–	–	**107**	82	12	143	–	–	1
96	192	–	–	–	**159**	96	48	207	–	–	3
96	256	–	–	–	**225**	96	114	272	–	–	5

CSSI											
Max.	Prime	Tani			Multi-Grover			VW			NIST
Depth	Length	C	D	W	C	D	W	C	D	W	Level
48	434	235	93	127	260	81	163	**159**	48	116	2
48	610	304	116	172	360	114	229	**228**	48	166	4
48	751	359	135	208	440	141	282	**283**	51	205	∞
64	434	235	93	127	260	81	163	**152**	64	102	2
64	610	304	116	172	360	114	229	**221**	64	153	4
64	751	359	135	208	440	141	282	**277**	64	193	5
96	434	231	96	121	195	96	85	**158**	95	79	3
96	610	304	116	172	360	114	229	**208**	96	125	3
96	751	359	135	208	440	141	282	**263**	96	166	5

QRAM Model

SHA Collision Finding											
Max.	Hash	Tani			Multi-Grover			VW			NIST
Depth	Length	C	D	W	C	D	W	C	D	W	Level
48	256	208	80	141	203	203	143	**146**	–	–	2
48	384	305	112	205	300	300	208	**210**	–	–	4
64	256	208	80	141	203	203	143	**146**	–	–	2
64	384	305	112	205	300	300	208	**210**	–	–	4
96	256	192	96	108	203	203	143	**146**	–	–	2
96	384	305	112	205	300	300	208	**210**	–	–	4

AES Key Finding											
Max.	AES	–			Grover			Classical			NIST
Depth	Size				C	D	W	C	D	W	Level
48	128	–	–	–	**122**	48	80	143	–	–	1
48	192	–	–	–	**187**	48	144	207	–	–	3
48	256	–	–	–	**252**	48	210	272	–	–	5
64	128	–	–	–	**106**	64	48	143	–	–	1
64	192	–	–	–	**171**	64	113	207	–	–	3
64	256	–	–	–	**236**	64	178	272	–	–	5
96	128	–	–	–	**88**	82	12	143	–	–	1
96	192	–	–	–	**139**	96	48	207	–	–	3
96	256	–	–	–	**204**	96	114	272	–	–	5

CSSI											
Max.	Prime	Tani			Multi-Grover			VW			NIST
Depth	Length	C	D	W	C	D	W	C	D	W	Level
48	434	161	116	126	173	173	126	**159**	48	116	2
48	610	207	121	171	240	240	170	**228**	48	166	4
48	751	258	139	207	293	293	206	283	51	205	∞
64	434	161	116	126	173	173	126	**152**	64	102	2
64	610	207	121	171	240	240	170	**221**	64	153	4
64	751	258	139	207	293	293	206	**277**	64	193	5
96	434	161	116	126	173	173	126	**158**	95	79	3
96	610	207	121	171	240	240	170	**208**	96	125	3/5
96	751	258	139	207	293	293	206	**263**	96	166	5