A Panoply of Quantum Algorithms

by

Bartholomew Furrow

B.Sc., Queen's University, 2004

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Physics)

The University Of British Columbia

September, 2006

Abstract

This thesis' aim is to explore improvements to, and applications of, a fundamental quantum algorithm invented by Grover[1]. Grover's algorithm is a basic tool that can be applied to a large number of problems in computer science, creating quantum algorithms that are polynomially faster than fastest known and fastest possible classical algorithms that solve the same problems. Our goal in this thesis is to make these techniques readily accessible to those without a strong background in quantum physics: we achieve this by providing a set of tools, each of which makes use of Grover's algorithm or similar techniques, that can be used as subroutines in many quantum algorithms.

The tools we provide are carefully constructed: they are easy to use, and they are asymptotically faster than the best tools previously available. The tools that we supersede include algorithms by Boyer, Brassard, Høyer and Tapp[2], Buhrman, Cleve, de Witt and Zalka[3] and Dürr and Høyer[4].

After creating our tools, we create several new quantum algorithms, each of which is faster than the fastest known classical algorithm that accomplishes the same aim, and some of which are faster than the fastest possible classical algorithm. These algorithms come from graph theory, computational geometry and dynamic programming. We discuss a breadth-first search that is faster than $\Theta(\text{edges})$ (the classical limit) in a dense graph, maximum-points-on-a-line in $\Theta(N^{3/2} \lg N)$ (faster than the fastest classical algorithm known), as well as several other algorithms that are similarly illustrative of solutions in some class of problem.

Through these new algorithms we illustrate the use of our tools, working to encourage their use and the study of quantum algorithms in general.

Table of Contents

List of Tables

List of Figures

Acknowledgements

The author, being a neophyte to the art of writing scientific papers, would particularly like to thank his advisor, Bill Unruh, for being an excellent font of advice and encouragement, as well as being very supportive of the author's choice of topic. He would also like to thank his second reader, David Kirkpatrick, for a series of excellent suggestions and no small number of corrections. He would like to thank Yury Kholondyrev, Matthew Chan and others involved in the UBC programming team for some early ideas of problems to tackle, and plenty of practice explaining himself; and finally he would like to thank Dürr and Høyer, authors of [4], for being the first to show him an exciting field, full of potential.

1. Introduction

The field of quantum algorithms is very young, with the very notion of quantum computation having been introduced in the early 1980s[5, 6, 7]. This thesis' aim is to make the study of quantum algorithms easily accessible to those without a strong background in quantum physics; we will achieve this by providing a set of tools that can be used as subroutines in many quantum algorithms. Great care is taken in the construction of these tools, which are both easy to use and technologically superior to those that are currently available, featuring asymptotically faster running times.

We will accomplish this aim in three steps, firming up work that has already been done and introducing new algorithms along the way. Our first step is to investigate the properties of two quantum search algorithms: the $BBHT$ algorithm and the $BCWZ$ algorithm. With $BBHT$ we devote considerable effort to discovering its probability of error, which had not previously been done; we make use of the result in Chapter 3.

As quantum searches, $BBHT$ and $BCWZ$ take a function $F(x)$, with x taking the values $\{0, \ldots, N-1\}$ and $F(x)$ taking the values 0 and 1, and find a (typically rare) element of the domain x such that $F(x) = 1$. We call this a "solution" for F. Equivalently, we can say that F is the characteristic function for some subset of $\{0, \ldots, N-1\}$, and quantum search algorithms' goal is to find some element of that subset.

We will frequently make reference to F, N and M in this section: F refers to a function whose solutions we are trying to find, N is the size of its domain, and M is the number of solutions (equivalently: F refers to the characteristic function of some subset, M is the cardinality of that subset, and F's domain is $\{0, \ldots, N-1\}$). The two algorithms we will examine, $BBHT$ and $BCWZ$, while both quantum searches, are different in methodology and in their properties: $BBHT$ runs faster than $BCWZ$ when M is large (when there are multiple solutions), and $BCWZ$ runs faster than $BBHT$ when we require a high probability of success. We discuss what it means for these algorithms to fail, as well as the probability with which that happens, below.

The $BBHT$ algorithm[2] was invented by Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp in 1996. It performs a quantum search in $\Theta(\sqrt{N/M})$ time,[1] making the assumption that at least one solution exists. Since we often do not know *a priori* if solutions exist, however, *BBHT* has limited utility. It is easy to modify *BBHT* so that, instead of assuming that solutions exist, it gives up after a certain point; but this gives it some probability of failure, a chance that it might fail to find a solution when one exists. That modification, not original to this paper[8, 9], is required for most uses of *BBHT*; in Chapter 2 we are the first to analyze it in depth, calculating its probability of failure and rederiving its running time.

The *BCWZ* algorithm[3] was the invention of Harry Buhrman, Richard Cleve, Roland de Wolf and Christoph Zalka in 1999. Like *BBHT*, it performs a quantum search; it also accepts a parameter ϵ, an upper-bound on the probability of failure that we are willing to tolerate. Given that parameter, *BCWZ* runs in $\Theta(\sqrt{N \lg \epsilon^{-1}})$ time. The advantage of *BCWZ* is that the dependence of the running time on the probability of failure ϵ scales as $\sqrt{\lg \epsilon^{-1}}$: if one were to search with *BBHT* instead of *BCWZ*, trying it multiple times to reduce the probability of failure, one would arrive at $\Theta(\sqrt{N/M} \lg \epsilon^{-1})$ time. We re-derive *BCWZ* in Chapter 2, showing a little more detail than the authors did in their original derivation.

Our second step is the introduction of our new tools, founded on *BBHT* and *BCWZ*, that perform various kinds of quantum searches. We combine these two, yielding a superior search algorithm that shares the best features of both: the fast running time of *BBHT* when there are many solutions to be found, and the fast running time of *BCWZ* when we want a very low probability of failure. Using this improved algorithm, we then derive a new procedure for finding the minimum of a function $G(x)$, based on that of Dürr and Høyer[4]; we also analyze the complexity for finding all M solutions to a function F, and introduce a tool, *mindiff*, that will be useful for solving problems in graph theory. We do all this in Chapter 3.

Our third objective is to use these tools, showing their application to be straightforward. We invent several new quantum algorithms, each of which is polynomially faster than the fastest known classical algorithm that achieves the same goal, and some of which are polynomially faster than the fastest possible classical algorithm that achieves the same goal. Our new algorithms solve problems that come from the areas of dynamic programming, computational geometry, and graph theory. A summary of those speedups can be found in Chapter 6; perhaps our most impressive result is a dynamic programming algorithm that speeds up a problem that has an

[1]See Section 1.2 for an explanation of O, Θ, Ω notation.

$\Theta(N^3)$ classical solution[2] to $\Theta(N^2)$ quantumly.

We also look at quantum algorithms by Dürr, Heiligman, Høyer & Mhalla[8] and Ambainis & Špalek[9], examining how they deal with failure in *BBHT*. We improve their algorithms by factors of order $\log N$ by replacing their use of the *BBHT* search algorithm with our improved *BBHT/BCWZ* combination.

Through these steps we hope to introduce the field of quantum algorithms to as broad an audience as possible; there are many exciting problems that remain to be solved, and we hope that this thesis will motivate and aid in such research.

1.1 A Brief History of Quantum Computing

1.1.1 Fundamental Work

Our chronicle begins in the years 1980-1982, with work done independently by Yuri Manin and Richard Feynman[5, 6]. Both researchers considered the problem of simulating the natural world using standard computers, and both came to the conclusion that it was impossible to do efficiently: simulating quantum mechanics with a classical computer would require work exponential in the number of particles being simulated. Feynman observed that if one had a computer whose basic operations were quantum mechanical in nature—what he was the first to call a quantum computer—one could, presumably, simulate a quantum system while only doing work polynomial in the number of particles. What this means is that quantum physics itself is somehow exponentially more powerful than a classical computer; and that something capable of harnessing quantum physics' strengths might be similarly powerful.

In the early 1930s, a great deal of work was done on the notion of effective computation: what exactly computers can and cannot do. Several independent approaches were taken, pursued in work by Church, Turing and Kleene[11, 12, 13]; ultimately they all came to equivalent conclusions about computers' abilities, in some sense launching the formal study of computer science. A quantum analog to these conclusions was described in 1985 by David Deutsch, where he stated[14]: "Every finitely realizable physical system can be perfectly simulated by a universal model computing machine operating by finite means." In the same paper he introduced such a "universal model computing machine," and thus launched the formal study of

[2]There is also a $\Theta(N^3 \frac{\lg \lg N}{\lg N})$ algorithm due to Tamaki[10].

quantum computing.

There has been a great deal of study devoted to probabilistic algorithms: algorithms that employ randomness as part of their logic. Randomness permeates the algorithms in this thesis, since the procedures on which they are based involve operations on and measurements of quantum systems. Every algorithm presented here has some probability of failure: some probability that it will run, terminate, and return an incorrect response. There are different ways of dealing with this; we can repeat the algorithms, or parts of them, to reduce the probability of failure: this gives us a relationship between probability of failure and running time. Throughout this paper, the analyses of our algorithms' running times will frequently include the symbol ϵ, indicating how much slower our algorithms become as we require lower and lower probabilities of failure.

1.1.2 Exponential Speedups

Although this thesis focusses on on quantum algorithms that offer polynomial speedups over their classical (non-quantum) equivalents, we digress briefly here to discuss some early quantum algorithms that are exponentially faster than their classical counterparts—being either faster than the fastest possible classical algorithm to solve the same problem, or faster than the fastest known classical algorithm. These are, of course, what has made the field of quantum algorithms so exciting.

An early quantum algorithm was invented by Dan Simon in 1994[15]. Simon's algorithm solved an artificial problem, one whose solution lacked obvious application; however he was able to show that a quantum algorithm would solve it exponentially more rapidly than any classical algorithm could. Moreover, Simon's algorithm laid the groundwork for Peter Shor's subsequent development of quantum factoring.

In 1994, Peter Shor published an algorithm that solved two important problems in number theory, both of which are generally believed to be superpolynomial on a classical computer: prime factorization, and the discrete log problem[16]. This algorithm is perhaps the best-known product of quantum computing: because it performs factoring in polynomial time, it could in principle be used to crack modern cryptosystems, such as RSA.

This section, regrettably, contains the last mention of exponential speedups in this thesis. While the study of exponential speedups is exciting, thus far results have been hard to come by, and generalize poorly. Thus one can factor, and do useful things with factoring, but one can not solve more than one or two similar problems. We will instead focus on polynomial speedups brought by Grover's algorithm, a very general tool that can be used to solve a seemingly huge number of problems either faster than they can be solved classically, or faster than the fastest known classical solution.

1.1.3 Grover's Algorithm and *BBHT*

We will be comparatively brief in this section, since Chapter 2 expands on many of the developments we will discuss here.

With the exponential improvements over the fastest known classical algorithms offered by Simon and Shor, it is natural to ask whether quantum computers provide exponential speedups in general; one could ask if there is some general methodology, exponentially faster on quantum computers than on classical computers, for seeking solutions to some class of problems. In 1994, Charles Bennett, Ethan Bernstein, Gilles Brassard and Umesh Vazirani looked at searching spaces of size N, and concluded that an unstructured search for a unique solution would not be faster than $\Omega(\sqrt{N})$ on a quantum computer, as opposed to $\Theta(N)$ on a classical computer[17]. They did not have an algorithm that would achieve this quadratic speedup, but their lower bound would prove prophetic.

In 1996, Lov Grover introduced an algorithm for what he called "searching for a needle in a haystack": given (for example) a phone book with N randomly-ordered entries, Grover's algorithm could find a particular name in $\Theta(\sqrt{N})$ accesses to the book[1]. Classically this would take $\Theta(N)$ accesses: we would have to look at each entry of the phone book individually (note that we do not normally build phone books this way: traditionally they are alphabetized, allowing for a much more efficient search).

Grover's algorithm has potential applications that range far further than lookups in poorly-designed phonebooks, but in its original form it is quite limited: while it could find Mr. Zloklikovits in Vancouver without difficulty, it would have a great deal more difficulty with finding an arbitrary one of the many McDonald's restaurants[18]. Grover, although he did discuss it, did not go into detail about what would happen in the case where there was more than one needle in the haystack; and so his algorithm needed modification.

We now discuss two different enhancements to Grover's algorithm: the *BBHT* algorithm and the *BCWZ* algorithm. Both of them accomplish Grover's original goal, quickly finding a needle in a haystack; and both work even if there are multiple needles, finding one at random. *BBHT* is faster than *BCWZ* when there are many solutions to be found, while *BCWZ* is faster when a high probability of success is required.

Michel Boyer, Gilles Brassard, Peter Høyer, and Alain Tapp published the *BBHT* algorithm in 1996[2]. Consider Grover's phone book example, and our difficulty with finding any one of the M different McDonald's restaurants. Boyer, Brassard, Høyer and Tapp constructed an algorithm that would find a random McDonald's, after performing $\Theta(\sqrt{N/M})$ accesses to the phone book. More generally, the *BBHT* algorithm solves the following problem: given a function F ("is this entry a McDonald's?") that maps the domain $\{0, \ldots, N-1\}$ (the phonebook) to $\{0, 1\}$, find any x such that $F(x) = 1$ (find a McDonald's). The applications for this are very broad, as we shall see throughout this thesis. It is also worth noting that *BBHT* may fail; there is some probability that, even if a solution (needle, McDonald's) exists, *BBHT* will not find one. We will discuss this further in Chapter 2.

Harry Buhrman, Richard Cleve, Ronald de Wolf and Christof Zalka published the *BCWZ* algorithm in 1999[3]. Their algorithm used a technique known as amplitude amplification, invented by Brassard and Høyer and further explored by Brassard, Høyer, Mosca and Tapp, which we will discuss in Chapter 2. The *BCWZ* algorithm solves the same problem as *BBHT*; but unlike *BBHT*, *BCWZ*'s speed does not increase with the number of solutions. Like *BBHT*, it may fail to find a solution when one exists; *BCWZ*, however, is designed to minimize that probability in an efficient manner.

1.1.4 Algorithms Using *BBHT*

Shortly after the publication of *BBHT*, Christoph Dürr and Peter Høyer, also in 1996, demonstrated a use for the algorithm: minimum finding[4]. Given some function G that maps the domain $\{0, \ldots, N-1\}$ onto some totally ordered set, Dürr and Høyer's minimum-finding algorithm finds any x such that $G(x)$ is minimal, after $\Theta(\sqrt{N})$ calls to G.

BBHT also provides many speedups in graph theory. Christoph Dürr, Mark Heiligman, Peter Høyer and Mehdi Mhalla wrote the first paper on the topic, which gave quantum solutions to several classic graph problems[8]. The algorithms that they gave are polynomially faster than their classical counterparts. We will examine their algorithm for finding shortest paths in weighted graphs in Section 4.5.2.

Another paper speeding up graph problems was written by Ambainis and Špalek; they also addressed classic graph problems[9]. We will examine their algorithm for maximum bipartite matching in Section 4.5.3, speeding it up slightly. They also addressed problems similar to bipartite matching, such as maximum flow in integer networks, which we will not examine.

1.2 Conventions, Notation, and Notes

There are a few things that we need to discuss for those unfamiliar with the procedures of algorithmic analysis. We create several new algorithms in this thesis, and for each of them we are interested in analyzing its running time: how long it takes to compute a solution. Since our algorithms have randomness in them, we will be interested in the expected time it takes to run our algorithms on some input. Since any given problem can have a very large number of possible inputs, so we use a common formalism in complexity analysis:

For any problem, consider the set of all possible inputs and group them together by their size. For any given size, take the worst possible input of that size (the input that causes our algorithm to have the slowest expected running time). Now we have a mapping from input size to time; that tells us how fast our algorithm is as a function of input size, but this function may be erratic or hard to analyze. We need a mechanism for comparing these functions, so we can say that one algorithm is "faster" or "slower" than another. To do so, we compare these functions against well-behaved benchmark functions, such as N, N^2, $N \lg N$, and 2^N.

Consider such a well-behaved benchmark function, $b(N)$, and a function $f(N)$ that indicates (as discussed above) the worst-case expected running time of some algorithm on input of size N. We say that $f(N)$ is $O(b(N))$ if, for some constant c, $f(N) < cb(N)$ for all N; this indicates that our algorithm's running time is no slower than $b(N)$. We say that $f(N)$ is $\Omega(b(N))$ if, for some constant c, $f(N) > cb(N)$ for all N, indicating that our algorithm's running time is no faster than $b(N)$. Finally, we say that $f(N)$ is $\Theta(b(N))$ if it is both $O(b(N))$ and $\Omega(b(N))$, which indicates that our algorithm's running time goes approximately as $b(N)$.

This formalism is convenient for many reasons, not the least of which is that it allows us to dispense with constants and lower-order terms. $2N^2 + 10000$ is $\Theta(N^2)$, as is $5N^2 + 50$. This lets us dispense with many concerns about machine-specific issues: is division slower than addition? As long as it's only slower by a constant factor, it doesn't matter.

Following the convention found in Cormen et al.'s book[19], we assume that basic arithmetic and addressing conventions take constant time. This is consistent with other papers on quantum algorithms: see for example [1, 2, 3, 4, 8, 9].

A note on terminology: we frequently use the term "classical" to refer to algorithms that are designed for a classical model of computing. While this is a standard term, it is perhaps unfortunate: in computer science literature, a classical algorithm is one that has proven its worth and stood the test of time. Euclid's and Dijkstra's algorithms are perhaps the best examples, being algorithms in common use that were invented hundreds of years ago. We attempt to maintain some of this usage by using the term "classic" to refer to such algorithms.

Finally, wherever the character 'z' appears on its own in this document, it is pronounced "zed."

1.3 Capabilities of our Theoretical Quantum Computer

This thesis' algorithms are designed based on certain assumptions about what sort of machine they will be run on; we discuss those here. Fundamentally, we take our machine to be equivalent to a "universal model computing machine" by Deutsch's definition[14]. That assumption alone is insufficient, however, to let us analyze our algorithms: any classical computer is equivalent to a Turing Machine, but there are many operations that are asymptotically faster if one has a standard personal computer using C++.

As such, we assume throughout the paper that the computer on which our algorithms are running is fundamentally very similar to a modern personal computer: specifically that it follows the RAM model outlined by Cormen et al.[19]. To add to that model the sort of capabilities that we associate with an idealized quantum computer, we first say that all our bits are qubits, and make the assumption that we do not have to concern ourselves with decoherence (that the qubits will stay at whatever value they are given); furthermore, that an arbitrary number of qubits can be entangled together. While the assumption that decoherence will not be an issue is perhaps unrealistic, we hope that our algorithms can be appropriately modified, that error correction codes will be sufficiently sophisticated, or that, while not arbitrary, decoherence times will be long enough for many of our algorithms to work.

Our processor should be capable of quantum operations. In particular, it should be capable of some universal set of quantum logic gates as outlined in Nielsen and Chuang's text[20]. A universal set of quantum logic gates is capable of simulating an arbitrary quantum gate, though the operations in this thesis only require basic arithmetic operations, Hadamard gates, and single-qubit gates.

We require that our random-access memory be accessible in superposition: this means that if we pass a linear combination of number states to our RAM-accessor, such as[3] $(|0\rangle + |2\rangle + |5\rangle)/\sqrt{3}$, it can extract all the appropriate memory entries in parallel. To illustrate this, let us define the operator $\mathcal{M}$, which retrieves qubits from memory. To access the 5th qubit in memory, we would perform the following operation: $\mathcal{M}|5\rangle|0\rangle$. This would give us $|5\rangle|a\rangle$, where a is the content of the 5th qubit in memory. The result would be entangled with the original qubit, still sitting in the 5th slot of memory. By accessing memory in *superposition*, we mean performing the operation $\mathcal{M}(|5\rangle + |2\rangle)|0\rangle/\sqrt{2}$ and getting the result $(|5\rangle|a\rangle + |2\rangle|b\rangle)/\sqrt{2}$, where b is the content of the 2nd qubit in memory.

Other papers on quantum algorithms[8, 9] choose to focus on analyzing the number of queries to some oracle their quantum algorithms take, eschewing the analysis of running time. We will discuss this further in Section 4.5.1; our results can be similarly analysed, and for the most part our running times are equal to our query complexities. In some algorithms, particularly that of Section 4.5.2, there is a difference; it is because of such differences that we feel running time is a better measure than query complexity of algorithms' performance, despite a large degree of uncertainty about exactly how quantum computers will be implemented in the future (and what unknown factors will appear in running times).

1.4 New Material Presented in this Thesis

This section is intended to be a quick reference, providing ready access to material that is original to this thesis. We will list this material in the order in which it appears. Recall that F refers to a function whose solution we are trying to find, N refers to the size of F's domain, and M refers to the number of solutions.

Our analysis of *BBHT*'s probability of failure (Section 2.2 and appendix A) is original, and is one of the most important contributions of this thesis; also original is our detailed presentation of *BCWZ* (Section 2.4).

Section 3.1's *findsol* is original, and is perhaps the most important contribution of this thesis; also original is its application in Section 3.3, *findall*. Section 3.2's *minfind* is Dürr and Høyer's[4], modified to take advantage of *findsol*'s capabilities (they used an unmodified *BBHT*, and aborted it after $\Theta(\sqrt{N})$ time). Section 3.4's *mindiff* comes from Dürr, Heiligman, Høyer and

[3]This notation will be defined and discussed in Section 2.1. Intuitively, $|5\rangle$ indicates a series of qubits (each of which is indicated by $a_i|0\rangle + b_i|1\rangle$) set to $|0\rangle \ldots |0\rangle|1\rangle|0\rangle|1\rangle$, the binary representation of 5.

Mhalla's similar algorithm[8], though we provide several of the algorithm's specifics that did not impact its query complexity, and were thus not studied by the original authors (see Section 4.5.1).

The algorithms in Sections 4.1 and 4.2 (*BFS* and *DFS*) are our own, although Ambainis and Špalek have an almost-identical algorithm[9] for breadth-first search that used *BBHT* instead of *findsol*.

The algorithm in Section 4.3 (*SPNW*) is our own, as is Section 4.4's *APSP*. Our improvements to Dürr, Heiligman, Høyer and Mhalla's single-source shortest paths[8] and Ambainis and Špalek's bipartite matching[9] algorithms are original (see Section 4.5), and we add some additional detail to DHHM's single-source shortest paths.

The algorithms in Chapter 5, *maxpoints*, *coinchange* and *subarray-sum* are all original, though they are based on classical algorithms that are not original to this thesis.

2. Grover's Algorithm and Amplitude Amplification

2.1 Grover's Algorithm

To describe Grover's algorithm, we will use Grover's original metaphor: needle-finding. Grover's algorithm is a procedure to find a "needle" in a "haystack." More precisely, given a function F, with value 0 except at some unique q (the "needle") for which $F(q) = 1$, acting on the domain $\{0, \ldots, N-1\}$ ("haystack"), Grover's algorithm finds the needle q. In this section we will explain the physics of Grover's algorithm, taking the needle always to be the integer q.

First we need to define some notation. A qubit is a two-state system, and so we will represent the two states as $|0\rangle$ and $|1\rangle$. One qubit's state, then, can be written as $a|0\rangle + b|1\rangle$, where a and b are complex numbers such that $|a|^2 + |b|^2 = 1$. If we have two qubits, A and B, their joint state can be written as $c|0\rangle_A|0\rangle_B + d|0\rangle_A|1\rangle_B + e|1\rangle_A|0\rangle_B + f|1\rangle_A|1\rangle_B$. We change notation here to write this as $c|00\rangle + d|01\rangle + e|10\rangle + f|11\rangle$, and finally change notation one more time to write the above as $c|0\rangle + d|1\rangle + e|2\rangle + f|3\rangle$.[4] Note that we started by writing our qubits separately, and ended by representing the qubit-string (or "qudit" as it is sometimes known) as an integer. We will consistently use this final notation throughout the paper. We will use Roman characters and integers to represent eigenstates of the number basis (such as $|i\rangle$ and $|3\rangle$), and Greek characters to represent arbitrary states.

To implement Grover's algorithm, we assume that we have implemented F as a quantum black box $\mathcal{F}$. For these purposes, this means that we can pass a state $|i\rangle$ into $\mathcal{F}$, and end up with $\mathcal{F}|i\rangle \equiv (-1)^{F(i)}|i\rangle$. Because quantum physics is linear, we can evaluate $F(0), \ldots, F(N-1)$ "simultaneously" by implementing $\mathcal{F}$ and passing in $|\Psi\rangle \equiv (|0\rangle + |1\rangle + \ldots + |N-1\rangle)/\sqrt{N}$.

[4]There is clearly an ambiguity whenever we write $|0\rangle$ or $|1\rangle$, since each could represent the state of a single qubit or the state $|00\ldots000\rangle$ (respectively $|00\ldots001\rangle$) of some number of qubits. We hope which is meant will be clear from context.

The first step in Grover's algorithm is to put our system $|\phi\rangle$ into the following starting state, which we will refer to as $|\Psi\rangle$ all throughout this chapter:

$$|\Psi\rangle = \frac{1}{\sqrt{N}}|0\rangle + \frac{1}{\sqrt{N}}|1\rangle + \ldots + \frac{1}{\sqrt{N}}|q\rangle + \ldots + \frac{1}{\sqrt{N}}|N-1\rangle \tag{2.1}$$

Grover's algorithm works by gradually nudging the amplitude on the $|q\rangle$ term toward 1, and simultaneously nudging the other amplitudes toward 0. The last step of the algorithm is measurement: since $|q\rangle$'s amplitude is now near 1, it is likely that our measurement's result will be q. This series of nudges is called amplitude amplification on $|\phi\rangle$ (our notation for "our current state").

We begin our discussion of Grover iterations with a diagram of the vector space in which we will sit. Our starting state is $|\Psi\rangle$, the even superposition of all possible inputs; our desired state is $|q\rangle$. Considering the span of those two vectors, which is two-dimensional, we can now construct Figure 2.1.

The first step in Grover's algorithm is to take our initial state $|\Psi\rangle$ and apply $\mathcal{F}$ to it. Revisiting our definition of $\mathcal{F}$, we realize that this simply reflects our state about the vertical axis. That is the first step in a Grover iteration; the second step is to reflect about our initial state, $|\Psi\rangle$ (we will call the operator that does this $\mathcal{J}$, and provide its details later). We can see the results of this process in Figure 2.2.

It is easy to see from Figures 2.1 and 2.2 that a single iteration increases the angle between the current state $|\phi\rangle$ and the vertical axis by 2θ. θ was defined in Figure 2.1 to be the angle between the vertical axis and $|\Psi\rangle$, our starting state; since the probability of measuring $|q\rangle$ in our starting state is $\frac{1}{N}$, $\sin^2(\theta) = \frac{1}{N}$ and thus $\theta = \sin^{-1}\left(\frac{1}{\sqrt{N}}\right)$.

We are still missing a few details, but we can now write Grover's algorithm as the following procedure:

1. Create the black box $\mathcal{F}$ such that $\mathcal{F}|i\rangle = (-1)^{F(i)}|i\rangle$.
2. Let the state of our system be called $|\phi\rangle$. Initialize $|\phi\rangle$ to $|\Psi\rangle \equiv (|0\rangle + |1\rangle + \ldots + |N-1\rangle)/\sqrt{N}$.
3. Let $\theta \equiv \sin^{-1}\left(\frac{1}{\sqrt{N}}\right)$. Let $\alpha = \theta$.
4. Repeat the following, until $\alpha + 2\theta > \frac{\pi}{2}$:
   (a) Apply $\mathcal{F}$ to $|\phi\rangle$, followed by $\mathcal{J}$, the reflection about the starting state $|\Psi\rangle$.

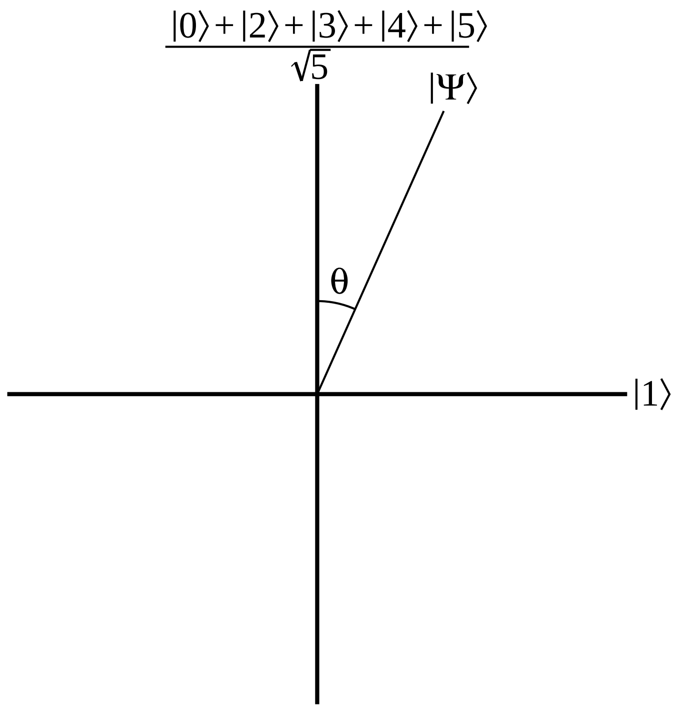

$\frac{|0\rangle+|2\rangle+|3\rangle+|4\rangle+|5\rangle}{\sqrt{5}}$
$|\Psi\rangle$
θ
$|1\rangle$

Figure 2.1: Starting state for Grover's algorithm: $N = 6$, $q = 1$. θ is defined to be the angle between $|\Psi\rangle$ and the vertical axis. Note that the vertical axis has no $|1\rangle$ term, and is thus orthogonal to the horizontal axis.

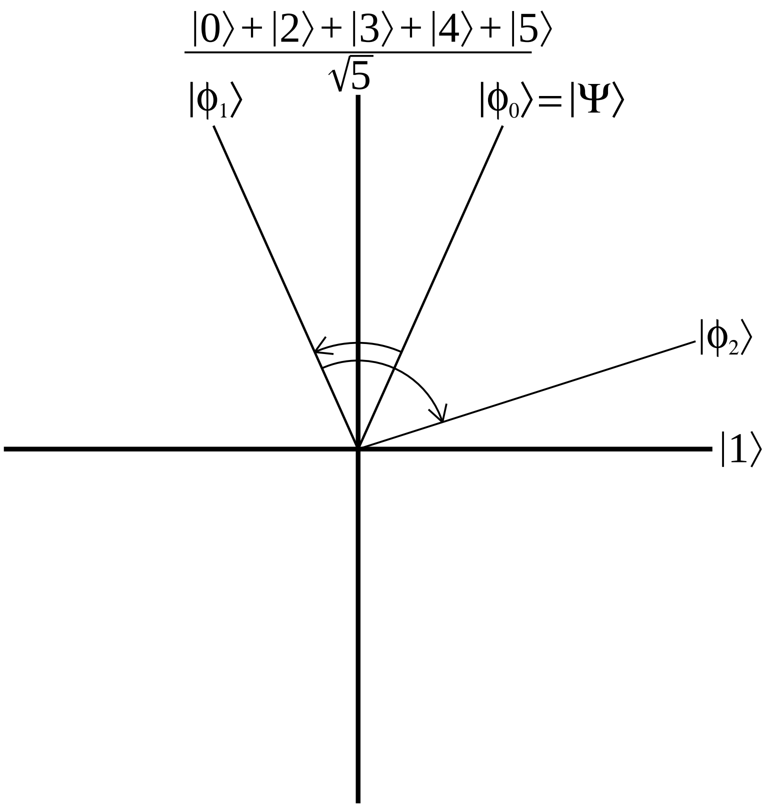
   (b) Let $\alpha = \alpha + 2\theta$.

5. Measure $|\phi\rangle$ in the number basis; call the result x.

6. If $F(x) = 1$, return x. Otherwise go to step 1.

The "goto" in step 6 is necessary because θ may be such that α never exactly reaches $\frac{\pi}{2}$, the point at which x would necessarily equal q. This means that every time we run through steps 1–6, there is some probability of "failure"—measuring the wrong x. Since we never measure unless our current angle from the vertical axis is $\geq \frac{\pi}{6}$—if the angle is less, we will do another Grover iteration—this probability is $\leq \frac{2}{3}$, and so in the worst case the number of times we repeat the body of the algorithm has expectation 3.

We still lack two major details: we have not discussed the implementation of $\mathcal{J}$; and while we have a procedure, we would like to know how long it takes (how often step 4 will repeat for given N). Implementing $\mathcal{J}$ is a



$\frac{|0\rangle + |2\rangle + |3\rangle + |4\rangle + |5\rangle}{\sqrt{5}}$
$|\phi_1\rangle$
$|\phi_0\rangle = |\Psi\rangle$
$|\phi_2\rangle$
$|1\rangle$

Figure 2.2: One Grover iteration: $N = 6$, $q = 1$.

matter of combining a few quantum gates. $\mathcal{J}$'s purpose is to reflect about $|\Psi\rangle$, our initial state; to implement it, we can switch bases so that $|\Psi\rangle = |0\rangle$, and then reflect about $|0\rangle$. We leave the details to Grover's original paper[1].

Finally, we need a running time. For starting states with a large number of solutions ($\theta > \frac{\pi}{6}$), we effectively choose a random input and check it against F, for which the expected number of applications of F is a constant. For $\theta \leq \frac{\pi}{6}$, we perform at least one Grover iteration. We first note that our number of iterations is $\leq \frac{\pi}{2\theta}$. Since $\theta \geq \sin\theta$ in that range, we have our number of iterations $\leq \frac{\pi}{2\sin\theta} = \frac{\pi\sqrt{N}}{2}$. We can take a lower bound similarly, and arrive at $\Theta(\sqrt{N})$ iterations through step 4, and thus $\Theta(\sqrt{N})$ calls to F.

As for running time that comes from sources other than the calls to F, we have $\Theta(\sqrt{N})$ calls to $\mathcal{J}$. $\mathcal{J}$ is a $\Theta(1)$ operation, and thus takes no more time than $\mathcal{F}$ (which must be $\Omega(1)$). One detail of $\mathcal{J}$ is that it only works on an integer number of qubits: executing it leaves us with entanglement over states that are not in the domain of F, and can thus break Grover's. To deal with that we can simply extend the domain of F to the nearest power of 2, defining $F(x) = 0$ for x not in the original domain. This increases N by a factor of two at most, and thus leaves us with the same asymptotic performance.

2.2 BBHT

Grover's algorithm has a major disadvantage, which is that it is designed for the one-needle case: where $F(x) = 1$ for only one x. It has the problem that every Grover iteration increases $|\phi\rangle$'s angle from the vertical axis by 2θ; after about $\frac{\pi}{4\theta}$ iterations, each iteration *reduces* the probability of obtaining the correct result.

When the number of solutions M is known, Grover's algorithm is easy to adapt. Recall that θ, the angle between the vertical axis and our starting state, was initially $\sin^{-1}\sqrt{\frac{1}{N}}$. Since we now have M possible solutions, the probability that measuring $|\Psi\rangle$ would give one of the solutions is $\frac{M}{N}$; therefore the projection of $|\Psi\rangle$ onto the horizontal axis, $\sin^2\theta$, must equal $\frac{M}{N}$. Thus we have that $\theta = \sin^{-1}\sqrt{\frac{M}{N}}$; aside from noting that change, we can proceed normally with Grover's algorithm, obtaining a random solution when it terminates. If the number of solutions is not known, however, then neither is θ; and Grover's algorithm as written will not consistently offer us a quadratic speedup.

The **BBHT** algorithm[2] solves the needle-finding problem when the number of solutions is not known. The concepts it makes use of were introduced in Section 2.1, and so we begin by presenting Boyer, Brassard, Høyer and Tapp's algorithm, slightly modified for our purposes:

1. Create the black box $\mathcal{F}$ such that $\mathcal{F}|i\rangle = (-1)^{F(i)}|i\rangle$.
2. Initialize $m = 1$ and set $\lambda = 1.31$.[5]
3. While $m \leq 2\sqrt{N}$, repeat the following unless instructed to return:
    (a) Let the state of our system be called $|\phi\rangle$. Initialize $|\phi\rangle$ to $|\Psi\rangle$, the equal superposition of all states.
    (b) Choose an integer j uniformly at random such that $0 \leq j < m$.

[5]This choice was made to minimize the running time of *BBHT* under a certain approximation (see Appendix A): any number strictly between 1 and 2 would work.

   (c) Apply j Grover iterations $\mathcal{JF}$ to $|\phi\rangle$.
   (d) Measure $|\phi\rangle$ in the number basis, and call the result x.
   (e) If $F(x) = 1$, return x; otherwise, set m to λm.

4. Return *false*.

Note that we return the special value *false* if no solution has been found after a sufficiently large number of iterations through step 3. Since M is unknown, it is possible that $M = 0$ and there are no solutions; thus it is important to give up at some point if no solution has yet been found. That is a modification we have made from the original algorithm: we allow our algorithm to give up if there is probably no solution, which also introduces the possibility of failure (which we explore below); it may give up too soon.

Intuitively, *BBHT* works by trying several different numbers of Grover iterations, which (depending on how many iterations there were) will yield different probabilities of success for different values of M. The algorithm as a whole will fail with probability $< .5M^{-.93}$, and its total number of calls to F has an expectation of $\Theta(\sqrt{N/M})$; we prove this in Appendix A.

BBHT is simple and powerful: given a binary function F and its domain, it finds x so that $F(x) = 1$ if such an x exists, returns *false* if it does not, and requires no additional information. Its one disadvantage is that it may fail, returning *false* incorrectly. *BBHT* is the primary tool used by Dürr, Heiligman, Høyer and Mhalla[8] and Ambainis and Špalek[9] in constructing their algorithms; we use it in combination with *BCWZ* (see Section 2.4) to ameliorate the effects of failure on running time.

2.3 Amplitude Amplification

Amplitude amplification generically refers to any process where one takes some state $a|\phi\rangle + b|\chi\rangle$ and selectively increases the magnitude of one of its amplitudes, a or b. Grover's algorithm is an amplitude amplification process: we start off with an even superposition of all input states, and amplify the amplitude of the states that satisfy some function F.

An amplitude amplification *process* is a search, much like Grover's algorithm or *BBHT*. Like Grover's algorithm, an amplitude amplification process starts in some state $|\beta\rangle$, gradually amplifies the amplitude of the terms we are interested in through a process much like a Grover iteration, and then terminates by measuring the state.

In Section 2.4 we will make use of an algorithm called **exact search**, an amplitude amplification process invented by Brassard and Høyer[21]. One of the difficulties with Grover's algorithm is that it does not necessarily succeed on the first try, even if we know M; because the probability of measuring a correct answer is $\sin^2((2m+1)\theta)$ after m Grover iterations, there may not be some integer number of iterations that would guarantee success. If we allow modifications to our Grover iterations, however, we can decrease the amount by which our angle increases, and make our required number of iterations an integer. Brassard and Høyer's *exact search* does just that, accepting a parameter $M_1 > 0$ and, if M_1 is the number of solutions to the given function F, returning a random such solution in $\Theta(\sqrt{N/M_1})$ calls to F. If M_1 is not the number of solutions to F, it will terminate in $\Theta(\sqrt{N/M_1})$ calls to F, and return either a solution or *false*.

Exact search's strength does not come from the fact that it never fails if the guess for M is right; after all, if M is known then it is easy to adapt Grover's algorithm to try until it succeeds. Its strength is that if we try exact search for a given M_1 and it fails, then the function definitively does not have exactly M solutions. We will find this to be useful in Section 2.4, when we explore the *BCWZ* algorithm.

Amplitude amplification has a number of other uses, such as working with heuristics to solve difficult decision problems; we will not explore those uses here. Suffice it to say that it is a useful procedure in general, and worthy of exploration in its own right.

2.4 BCWZ

The **BCWZ** algorithm, invented by Buhrman, Cleve, de Wolf and Zalka[3], is another general search algorithm; but where *BBHT* is speedy when there are many solutions, *BCWZ* deals well with errors.

Let us examine how we typically deal with errors classically. Consider a function G that finds a solution to some problem. If a solution exists, G finds that solution with probability $1-p$; otherwise it simply returns *false*. Suppose that we suspect there are no solutions: given that G is our only tool, how can we be sure to within a probability of ϵ that no solution exists?

The solution is to keep trying G until the probability is less than ϵ that, if a solution existed, it was not found. Since the probability that G fails to find a solution when one exists is p, if we try r times then the probability that we fail is p^r. Since we want $p^r = \epsilon$, we have $r = \log(\epsilon)/\log(p)$. We thus need to repeat G $\Omega(\lg \epsilon^{-1})$ times before we can be confident that our probability of failure is less than ϵ.

The *BBHT* algorithm, which we discussed in Section 2.2, has all of the properties that we assigned to G. If it finds a solution, we know immediately; if it fails to find one, or if there is none, it will claim no solution exists. It fails with some probability p if there is a solution; so if we want to ensure that a function F has no solutions (and be right with probability $1-\epsilon$) we must try *BBHT* $\Omega(\lg \epsilon^{-1})$ times.

BCWZ does better than that, and (as we will see) reduces our error-dependence to $\Theta(\sqrt{\lg \epsilon^{-1}})$. We have already introduced the ideas that we need, and so we will begin by writing out the algorithm. Note that step 1 has a logarithm taken in base 1.5; our choice of 1.5 is not entirely arbitrary, but is important to the analysis of the algorithm. Our ultimate requirement is that our logarithm's base X is such that $0.61^{\log_X \epsilon^{-1}} < \epsilon$ for all ϵ in $(0,1)$, and so we could have taken any base in the range $(1, 1.6]$.

1. Let $M_0 = \lceil \log_{1.5} \epsilon^{-1} \rceil$.
2. Apply *exact search*, taking its "guess" for M, M_1, to be all integers in $[1, M_0]$. If a result other than *false* is ever returned, return it.
3. Repeat the following steps M_0 times unless instructed to return:
   (a) Let the state of our system be called $|\phi\rangle$. Initialize $|\phi\rangle$ to $|\Psi\rangle$, the even superposition of states.
   (b) Choose an integer j uniformly at random such that $0 \le j < \sqrt{N/M_0}$.
   (c) Apply j Grover iterations $\mathcal{JF}$ to $|\phi\rangle$.
   (d) Measure $|\phi\rangle$ in the number basis, and call the result x.
   (e) If $F(x) = 1$, return x.
4. Return *false*.

After step 2, we can be sure that $M = 0$ or $M > M_0$; consequently, $\sqrt{N/M_0} > \sqrt{N/M}$. We found in our analysis of *BBHT* (see Appendix A) that every time we go through a series of steps such as steps 3 above, we have probability at least 0.39 of finding a solution if one exists. So either $M < M_0$, and we are guaranteed to find a solution in step 2 (the probability of failure is 0); or we arrive at step 3, and after each iteration our probability of failure is reduced by a factor of 0.61. Our probability of failure for $M > M_0$ is thus $0.61^{\log_{1.5} \epsilon^{-1}} < \epsilon$. As for running time, our number of calls to F is bounded by:

$$\sum_{M_1=1}^{M_0} \Theta(\sqrt{N/M}) + M_0 \Theta\sqrt{N/M_0} = \Theta(\sqrt{N \lg \epsilon^{-1}}) \qquad (2.2)$$

Thus demanding a low probability of failure from *BCWZ* is cheaper than demanding the same from *BBHT*; but since *BBHT* is a factor of $\sqrt{M}$ faster than *BCWZ* when is error probability is a constant, it is rarely practical to use *BCWZ* on its own. We will explore using the two algorithms jointly in Section 3.1, and thus create a tool that is more powerful than either alone.

It is worth noting that we have made a small modification to the original *BCWZ* algorithm: instead of our step 3, Buhrman, Cleve, de Wolf and Zalka have a step saying "conduct M_0 searches, each with $O(\sqrt{N/M_0})$ queries." They later conclude that "each of the searches in [that step] can be made to have error probability $\leq 1/2$." It is unclear what "can be made to have" means there, but such searches may have probability of success near 0 if an unfortunate choice of ϵ (and thus M_0) is made. Our step 3 avoids that difficulty, and ends up having error probability ≤ 0.61. This can be reduced to 0.5 by the simple expedient of trying it twice; this procedure is presumably what the authors intended, or close to it.

2.5 Inside an Iteration: Properties of F

Each of the search algorithms that we have discussed operates with some function F, and finds some solution x such that $F(x) = 1$; we have not given much discussion to what properties F may have, however. Can the function access tables of previously-stored information? Can it look things up in a hash table or search tree? Can it write in memory, and have those writings available in the next iteration? To some extent this discussion depends on the capabilities we have chosen for our quantum computer, but there will inevitably be some things we need and some things we can never have.

It is worth noting that in this thesis, the functions we evaluate inside Grover iterations are always fundamentally classical in nature: look up something in a table; multiply two numbers together; if A then do B; and so forth. F is executed as a quantum black box on a superposition of states, so it needs to be implemented with quantum gates—objects that are fundamentally reversible. Furthermore, after F has been evaluated and each term $a_i|i\rangle$ has become either $a_i|i\rangle$ or $-a_i|i\rangle$, we need to have no intermediate qubits from our computation of F tied up with that result. After all, if we took $\mathcal{F}(|i\rangle|0\rangle + |j\rangle|0\rangle)$ and somehow came out with $(-1)^{F(i)}|i\rangle|a\rangle + (-1)^{F(j)}|j\rangle|b\rangle$, then any further manipulation of our state with Grover's algorithm would not budge $|a\rangle$ and $|b\rangle$, and thus would not allow interference between $|i\rangle$ and $|j\rangle$.

Fortunately for us, decades of work have been done on reversible computing, and a paper by Bennett[22] contains what we need: any irreversible procedure may be executed reversibly, as long as we keep any intermediate results. This allows us to perform operations such as multiplication, which has no inverse—given ab we can not necessarily find a and b—reversibly, by storing ancillary information (given ab and a, we can find b).

Generalizing this process to an arbitrary quantum algorithm F, we undertake the following process: we start by executing F, taking care never to erase any information. If at any point we have to perform a fundamentally irreversible operation like multiplication, we keep ancillary information so that we are able to reverse the operation. Once we have finished our computation and determined $k = F(x)$, our second step is to apply a gate that gives us a total phase of $(-1)^k$. Finally we take every step involved in the execution of F and reverse it in order from last to first. Finally we end up in our initial state, but for the phase on $|x\rangle$.

The fact that we reverse each step has the effect of undoing any writing that we do to memory; so while we can perform an arbitrary classical computation inside F, and can read from memory set before the beginning of the computation (such as arrays, hash tables and binary search trees), any alterations that we make to that memory will be undone before the next Grover iteration.

2.6 Summary

We have discussed four search algorithms in this chapter. We summarize them here, using our notation from earlier: F is a binary-valued function that we are trying to find a solution for (x such that $F(x) = 1$), N is the size of F's domain, and M is the number of solutions. An algorithm is said to fail if it returns that there are no solutions when there is at least one.

Grover's algorithm is a search that, when $M = 1$, takes $\Theta(\sqrt{N})$ calls to F and never fails. It is made obsolete by the other three algorithms.

BBHT is a search that does not require knowledge of M, takes $\Theta(\sqrt{N/M})$ calls to F, and fails with probability less than $.5M^{-.93}$. If $M = 0$, it takes $\Theta(\sqrt{N})$ time but never fails.

Exact search is a search that takes a guess for M, which we call M_1; if that guess is correct, it takes $\Theta(\sqrt{N/M})$ calls to F, and has probability 0 of failure. Otherwise it takes $\Theta(\sqrt{N/M_1})$ calls to F, and fails with some probability.

BCWZ is a search that does not require knowledge of M, accepts a parameter ϵ, takes $\Theta(\sqrt{N \lg \epsilon^{-1}})$ calls to F, and fails with probability $\leq \epsilon$.

3. Tools

Here we present some basic algorithms, founded on *BBHT* and *BCWZ* (see Section 2.6). These serve as subroutines to be used throughout this thesis, where they will be referred to by their abbreviated names, found in the section headers.

Note that each of the following algorithms operates with some given function often called F, whose evaluation could have some arbitrary time complexity; as such, our standard unit of time for this chapter is the number of calls to the given function. There are, of course, operations in these algorithms outside of the function calls; in each of these algorithms, except for *mindiff* in Section 3.4, the extra operations are subdominant to the number of calls to the given function. In Section 3.4 we will use t_J and t_K to specify how the running time of the given functions J and K impact the total running time of the algorithm.

Each algorithm in this section is not only polynomially faster than the fastest known classical algorithm that achieves the same goal, but is also polynomially faster than the fastest possible classical algorithm.

3.1 Finding a Solution to F, *findsol*

Theorem 1 *Take a binary-valued function F over the domain $\{0, \ldots, N-1\}$. The following algorithm* **findsol** *searches for a solution x in the domain such that $F(x) = 1$. If there are $M > 0$ solutions, findsol will return a random one with probability $> 1 - 0.5M^{-1.86}\epsilon$, where ϵ is some probability of failure that we are willing to tolerate. If it successfully does so, findsol takes an expected $\Theta(\sqrt{N/M} + \sqrt{N \lg \epsilon^{-1}} M^{-1.86})$ calls to F; if it is unsuccessful, or if there are no solutions, findsol returns the special value false after $\Theta(\sqrt{N \lg \epsilon^{-1}})$ calls to F.*

In the following we use an extra parameter r, which allows tradeoffs in cost between the case where a solution exists and the case where there is no solution. We found $r = 2$ to be adequate for our purposes throughout this thesis; we include it as a parameter here in case someone else has use for it.

The principle we use here is very straightforward. First, we acknowledge that we can't do any better than $\sqrt{N \lg \epsilon^{-1}}$ (a single *BCWZ*) in the case where there are no solutions, so we try to optimize for the case where there are solutions and we can hope for $\Theta(\sqrt{N/M})$ calls to F. To do this, we try *BBHT* first due to its faster running time. Then if we have not found a solution, we check for one with *BCWZ* to make sure.

1. Run *BBHT* up to r times. If any of those returns a result that satisfies F, immediately return that result.

2. Run *BCWZ* with parameter ϵ. If it returns a result that satisfies F, return that result; otherwise return *false*.

The analysis for this is very straightforward. If there are solutions, step 1 takes an expectation of $\Theta(\sqrt{N/M})$ calls to F (we expect it to repeat less than twice). The *BBHT*s all fail with probability $\Theta(.5^r M^{-.93r})$; if they do we move on to step 2, which takes $\sqrt{N \lg \epsilon^{-1}}$ calls to F. This gives us an expectation of $\Theta(2\sqrt{N/M} + .5^r M^{-.93r}\sqrt{N \lg \epsilon^{-1}})$ total calls to F in the case where there are solutions; these reduce to to the promised quantities when $r = 2$. If there are no solutions, step 1 is $\Theta(r\sqrt{N})$ and step 2 is $\Theta(\sqrt{N \lg \epsilon^{-1}})$.

Looking at the probability of failure, we observe that the algorithm cannot possibly find a solution that does not exist, and therefore cannot fail when there are no solutions (the error is "one-sided"). If there are solutions, the probability of failure is $\leq .5^r M^{-.93r}\epsilon$, the probability that the *BBHT*s and *BCWZ* all fail.

We chose $r = 2$ because 2 is the smallest value that gives us a sufficiently small coefficient of $\sqrt{\lg \epsilon^{-1}}$ in the running time (one proportional to less than M^{-1}). That property is important in Section 3.3; in general, almost any natural number is a reasonable choice for r.

3.2 Minimum Finding, *minfind*

Theorem 2 *Take a function G that maps the domain $\{0, \ldots, N-1\}$ to some totally-ordered set. The following algorithm* **minfind** *finds x in the domain such that $G(x)$ is less than or equal to all $G(y)$ in expected time $\Theta\left(\sqrt{N \lg \epsilon^{-1}}\right)$ and with probability $\leq \epsilon$ of failure.*

This algorithm is based on one by Dürr and Høyer[4], and is identical in approach to theirs; but where we use *findsol* (step 2a, below) they use

BBHT $\lg \epsilon^{-1}$ times. The motivation for this algorithm, as with theirs, is repeatedly to find y with smaller and smaller values for $G(y)$. To do this efficiently, we use *findsol* as introduced in Section 3.1.

1. Pick y uniformly at random from the set $\{0, \dots, N-1\}$.
2. Repeat the following until instructed to return:
    (a) Run *findsol* with parameter ϵ to find an element $y' : G(y') < G(y)$.
    (b) If *findsol* returns an element, set $y = y'$; otherwise return y.

Dürr and Høyer show that the probability that y will ever take on the k^{th}-lowest value is $1/k$, and that for different k, those probabilities are independent. With that in mind, we can sum over all values of k to arrive at an expected running time and a probability of failure. For expected running time, we find:

$$\begin{aligned} t_{\mathit{minfind}} &= \sqrt{N \lg \epsilon^{-1}} + \sum_{k=2}^{N} \frac{1}{k} \sqrt{N \lg \epsilon^{-1}} k^{-1.86} \\ &\leq \sqrt{N \lg \epsilon^{-1}} + \int_{1}^{N} \frac{dk}{k} \sqrt{N \lg \epsilon^{-1}} k^{-1.86} \\ &\leq \sqrt{N \lg \epsilon^{-1}} + \sqrt{N \lg \epsilon^{-1}} \end{aligned}$$

calls to G. We calculate the probability of failure similarly, first noting that $P_{\mathit{fail}} \leq \sum_k P(k) P_{\mathit{fail}}(k)$:

$$P_{\mathit{fail}} \leq \sum_{k=2}^{N} \frac{1}{k} \epsilon k^{-1.86} \leq \int_{1}^{N} \frac{dk}{k} \epsilon k^{-1.86} \leq \epsilon$$

3.3 Finding all x that Satisfy F, *findall*

Theorem 3 *Take a binary function F over the domain $\{0, \dots, N-1\}$. Let any x for which $F(x) = 1$ be called a solution for F, and let the number of distinct solutions be called M. The following algorithm,* **findall***, finds all solutions for F in $\Theta(\sqrt{NM} + \sqrt{N \lg \epsilon^{-1}})$ expected calls to F, with probability $\leq \epsilon$ of failure.*

The idea behind this algorithm is to find successive solutions x, striking each off the search as we find it in order to guarantee that we find something different every time. We do this straightforwardly with *findsol*.

1. Create a direct-address table D to store results found so far.[6]
2. Repeat the following until instructed to return:
    (a) Run *findsol* with parameter ϵ to find an element x of $\{0,\dots,N-1\}$ such that $F(x)=1$ and D does not contain x.
    (b) If *findsol* returns an element, add it to the result set and D; otherwise, return the result set.

We calculate the number of calls to F with a straightforward integral:

$$\begin{aligned} t_{\mathit{findall}} &= \sqrt{N\lg\epsilon^{-1}} + \sum_{k=1}^{M}\left(\sqrt{N/k} + k^{-1.86}\sqrt{N\lg\epsilon^{-1}})\right) \\ &\approx 2\sqrt{N\lg\epsilon^{-1}} + \int_1^M dk\left(\sqrt{N/k} + k^{-1.86}\sqrt{N\lg\epsilon^{-1}})\right) \\ &\approx 2\sqrt{N\lg\epsilon^{-1}} + \sqrt{NM} + \sqrt{N\lg\epsilon^{-1}} \end{aligned}$$

We calculate the probability of failure similarly, noting that the probability of failure $P_{\mathit{fail}} \le \sum_k P_{\mathit{fail}}(k)$:

$$P_{\mathit{fail}} \le \sum_{k=1}^{M} \epsilon k^{-1.86} \le \int_1^M dk\epsilon k^{-1.86} \le \epsilon$$

3.4 Finding a Minimal d Objects of Different Types, *mindiff*

Suppose that we want to book d holidays to different destinations, and there are N flights y_i leaving our home airport to various destinations, with various costs $J(y_i)$. The following algorithm finds us the d cheapest destinations, and their respective cheapest flights.

Theorem 4 *Take a function J (our "cost") over the domain $\{0,\dots,N-1\}$, and some division of that domain into disjoint subsets (our "destinations"). The following algorithm,* **mindiff***, takes the lowest-cost element of each subset (breaking ties arbitrarily) and takes the lowest-cost d of those elements (again breaking ties arbitrarily), and returns the result. mindiff achieves*

[6]A direct-address table is a construct like a hash table, allowing us to keep track of already-found elements with $\Theta(1)$ operations. Details can be found in Cormen et al.'s book on algorithms[19].

this in $O\left((t_J + t_K)\left(\sqrt{Nd} + \sqrt{N \lg \epsilon^{-1}}\right) + d \lg N \lg d\right)$ *time, with probability* $\leq \epsilon$ *of failure.*

In terms of implementation, *mindiff* accepts a cost function J and a subset-division function K. We say that x and y are in the same subset iff $K(x) = K(y)$. We will call the d elements of *mindiff*'s result set x_i. To find the elements that we seek, we will start off with some "bad" set of d elements that are not a valid result set for *mindiff*, and repeatedly use *findsol* to find elements y that "improve" our current result set by satisfying either of the following conditions:

1. $J(y) < J(x_a)$ and $K(y) = K(x_a)$ for some a. This means y and x_a are in the same subset, and y is lower-cost than x_a.
2. $J(y) < J(x_a)$ for some a, $K(y) \neq K(x_i)$ for any i. This means y is in some subset that doesn't appear in our result set, but is lower-cost than something that does.

The basis for this algorithm comes from Dürr, Heiligman, Høyer and Mhalla[8]; their algorithm is roughly the same as our step 3 below, but it only outlines that step, and not the various data structures necessary for its implementation. The principle behind both their algorithm and ours is repeatedly to find y such that it meets either of the conditions above, and to replace the appropriate element of the result set with the new y.

1. Let x be the array of answers. Initially, let the x_i be "infinities," for which $J(x_i) = \infty$, and $K(x_i)$ is unique and not equal to $K(y)$ for any y in $\{0, \ldots, N-1\}$.
2. Let D be a direct-address table mapping $K(x_i)$ to i, and initialize it as such. Let T be a balanced binary search tree containing the pair $(J(x_i), i)$ for all i, sorted by $J(x_i)$, and initialize it as such.
3. Repeat the following until J has been evaluated $\Omega(\sqrt{Nd})$ times, or the loop has repeated $\Omega(d \lg N)$ times (whichever happens first):
   (a) Let τ be the largest $J(x_k)$ in T, and k the corresponding index.
   (b) Use *BBHT* to find some element of the domain y such that either $J(y) < \tau$ and $K(y) \notin D$ (condition 2), or $K(y) \in D$ and $J(y) < J(x_{D(K(y))})$ (condition 1). Note that $J(x_{D(K(y))})$ is the cost of the cheapest flight that we have found so far going to y's destination, if that is currently in our result set.

   (c) If condition 1 was met, set $x_{D(K(y))} = y$, and update D and T correspondingly. Otherwise, if condition 2 was met, set $x_k = y$, and update D and T accordingly.

4. Run *findsol* with parameter ϵ to check whether there is still a y that satisfies either condition as outlined in step 3b. If not, return x. If so, repeat step 3.

Terminating the loop in step 3 after $\Omega(\sqrt{dN})$ calls to J provides probability of success $> \frac{1}{2}$, which is shown by Dürr, Heiligman, Høyer and Mhalla. They also show that $\Omega(d)$ iterations suffice to eliminate a constant fraction of the domain from consideration, thus $\Omega(d \lg N)$ iterations will also provide probability of success $> \frac{1}{2}$. In order to improve the probability of success, we run *findsol* with parameter ϵ to check whether we are yet done; if we are not, we repeat step 3 until we are. Since the probability for step 3 to finish successfully after one pass is $\geq \frac{1}{2}$, we expect to repeat it – and *findsol* – an expectation of ≤ 2 times. We also have to consider the contribution of updating and accessing T, which will take $\Theta(\lg d)$ time with every iteration; thus our total running time is $O\left((t_J + t_K)\left(\sqrt{dN} + \sqrt{N \lg \epsilon^{-1}}\right) + d \lg N \lg d\right)$ with probability $\geq 1 - \epsilon$ of success.

Note that if d is greater than the number of distinct values for K ($\equiv \gamma$), we return γ valid elements and $d - \gamma$ infinities (fictitious elements of the domain as defined in step 1).

4. Applications in Graph Theory

A graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ consists of a finite set of vertices $\mathbb{V} = \{v_i\}$ and a finite set of edges $\mathbb{E} = \{e_i\}$, where $e_i = (v_j, v_k)$. In a directed graph, each edge is an ordered pair. In an undirected graph, each edge is an unordered pair. For convenience when analyzing algorithms that run on graphs, we write $|\mathbb{V}|$ as V, and $|\mathbb{E}|$ as E.

Many problems in computer science can be reduced to graph problems. For example, a subway system could be represented as an undirected graph, in which each stop is represented by a vertex and the each line connecting a pair of stops is represented by an edge.

A common graph problem is to find a path from one vertex to another along edges in the graph. More formally, a path p between vertices v_{start} and v_{end} is an ordered set of edges, $(e_{i_1}, e_{i_2}, \ldots, e_{i_n})$, where $e_{i_j} = (v_{i_j1}, v_{i_j2})$, such that $v_{i_11} = v_{\text{start}}$, $v_{i_n2} = v_{\text{end}}$, and $e_{i_j2} = e_{i_{j+1}1}$ for each $1 \leq i < n$ (the j^{th} edge in the path ends where the $j + 1^{\text{th}}$ edge begins). Often one is interested in finding the shortest path between two vertices: the path with the smallest number of edges n.

In a variant of the shortest path problem, each edge and/or vertex is assigned a weight, and the shortest path is considered to be the one for which the sum of weights of all vertices and edges in the path is minimized. A graph in which the vertices or edges have weights is referred to as a weighted graph.

The graph algorithms presented here assume the graphs they operate on will be simple graphs: they will have at most one edge between any two vertices (or two edges in opposite directions, in the directed case), and no "self-edges" that directly connect a vertex to itself. Most of these algorithms are very easy to generalize to graphs that do not have those properties, but we leave that task to the enterprising reader.

In this chapter we will focus on quantum versions of long-studied classic problems such as shortest paths, searching through graphs, and graph matchings (suppose you want to pair up vertices that are connected; what's the maximum number of disjoint pairs you can make?).

We present the algorithms here for two models of representing graphs, which we will discuss as quantum black boxes. If there is an edge between vertices v_i and v_j, we refer to it as e_{ij}. We present these two models because one, the edge array model, tends to yield more efficient algorithms; on the other hand, the adjacency matrix model is a common representation, and could be given to an algorithm as input. The models are:

- The **adjacency matrix** model, as a quantum black box, is passed i, j ($0 \leq i, j < V$) and returns whether e_{ij} exists. This is a mathematical function, often represented classically as a $V \times V$ matrix with entries in $\{0, 1\}$.
- The **edge array** model, as a quantum black box, is passed i, j and returns the destination of the j^{th} edge outgoing from vertex v_i (we assume for convenience that we know how many edges are outgoing from each vertex). Classically this is usually represented as a ragged array, but sometimes is generated mathematically as-needed. We call the set of edges outgoing from v_i d_i, and its cardinality $|d_i|$. The edge array model is sometimes called the *adjacency list* model; though when it is referred to by that name, the internal representation of edges may be a linked list rather than an array.

If the graph is weighted, the adjacency matrix and edge array models also return the weight of the edge queried.

For an excellent introduction to graph theory and algorithms therein, see Cormen, Leiserson, Rivest and Stein's classic introduction to algorithms[19]. It contains detailed discussions of breadth-first and depth-first searches, Dijkstra's algorithm and the Bellman-Ford algorithm, as well as all-pairs shortest paths. We look at all of these in this chapter, but leave the details to this reference.

4.1 Breadth-First Search, *BFS*

Breadth-first and depth-first search are two of the simplest algorithms for searching a graph, and find extensive use inside many important graph algorithms. The principle behind each is the same: starting at some source, we systematically explore the vertices of our graph, "visiting" each vertex connected to the origin in some order. By introducing quantum versions of each here, we tarnish their simplicity but maintain their strength and increase their speed.

As we mentioned above, *BFS* and *DFS* both see extensive use. Both can be used to determine whether a vertex is connected to the rest of the graph (if there is a path from that vertex to every other vertex), and breadth-first search in particular can be used to compute shortest paths in an unweighted graph. Depth-first search, on the other hand, can be used to detect "bridges" in a graph: edges which, if they were removed, would sever the graph into two pieces with no edges between them. There is a great deal of utility to be had from these two over and above what is discussed here, and both are well-studied techniques in classical computing; both run on classical computers in $\Theta(E)$ time, which is provably optimal.

To implement a breadth-first search here, we begin with classical BFS: we keep a list of vertices we want to visit, and every time we visit another of those vertices we add all of its unvisited neighbours to the list. Through use of a boolean array we ensure each vertex is only visited and added once. To choose the order in which the vertices are visited, we let our list be a "queue," wherein vertices added first are visited first; thus we end up visiting the vertices in order of how close they are to the origin of our search (breadth-first).

To speed up the process of finding all of the unvisited neighbours of each node, we use Section 3.3's *findall*. This algorithm is based on a BFS from Ambainis and Špalek[9], though they use repeated *BBHT*s rather than our *findall*.

Theorem 5 *The following algorithm* **BFS** *executes a breadth-first search through a graph* $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ *in* $O(\sqrt{V^3 \lg(V\epsilon^{-1})})$ *time in the matrix model,* $O(\sqrt{VE \lg(V\epsilon^{-1})})$ *in the edge array model. There is probability* $\leq \epsilon$ *that it fails, executing the breadth-first search incorrectly.*

1. Let the vertex from which we are searching be called v_a. Let there be a queue of vertices q, and let it initally contain only v_a. Let there be a boolean array vis of size V, with entries $vis[i] = \delta_{i,a}$.

2. Repeat the following until q is empty:

    (a) Remove the first element of q and call it v_i.

    (b) Visit v_i.

    (c) With Section 3.3's *findall*, find all neighbours v_j of v_i with $vis[j] =$ *false*.

    (d) For each such v_j, set $vis[j] = true$ and add v_j to q.

Since *BFS* makes a call to a function that may fail, *findall*, there is some probability that it fails. We may pass the *BFS* function some constant ϵ, a probability of failure that we are willing to tolerate; since *BFS* calls *findall* V times, if we require that *findall* has probability $\leq \epsilon/V$ of failure each time, the total probability with which *BFS* fails will be less than ϵ.

In the matrix model, each vertex v_i is processed at most once, and its *findall* contributes $\sqrt{Vn_i} + \sqrt{V \lg(V\epsilon^{-1})}$ to the running time in the matrix model, where n_i is the number of elements added to q. In the edge array model, each vertex is processed at most once and contributes $\sqrt{|d_i|\, n_i} + \sqrt{|d_i| \lg(V\epsilon^{-1})}$. By the Cauchy-Schwartz inequality, we have:

$$\sum_{v_i \in \mathbb{V}} \sqrt{n_i\, |d_i|} \leq \sqrt{\sum_{v_i \in \mathbb{V}} n_i} \sqrt{\sum_{v_i \in \mathbb{V}} |d_i|} = \sqrt{VE} \tag{4.1}$$

$$\sum_{v_i \in \mathbb{V}} \sqrt{|d_i| \lg(V\epsilon^{-1})} \leq \sqrt{\sum_{v_i \in \mathbb{V}} |d_i|} \sqrt{\sum_{v_i \in \mathbb{V}} \lg(V\epsilon^{-1})} \leq \sqrt{VE \lg(V\epsilon^{-1})} \tag{4.2}$$

We conclude that *BFS* in the edge array model runs in $O(\sqrt{VE \lg(V\epsilon^{-1})})$ time, and since $E < V^2$, *BFS* in the matrix model runs in $O(\sqrt{V^3 \lg V})$ time.

Classically the fastest possible breadth-first search algorithm takes $\Theta(E)$ time in the edge array model, $\Theta(V^2)$ time in the matrix model; so for bounded error probability, *BFS* is faster than its classical counterpart in the matrix model, and faster than its classical counterpart in the edge array model for $E \in \Omega(V \lg \epsilon^{-1})$.

4.2 Depth-First Search, *DFS*

Classically depth-first and breadth-first search can have very similar implementations, and the same is true in the quantum regime. The simplest implementation of depth-first search in both regimes, however, is a recursive one, which we show here.

Theorem 6 *The following algorithm* **DFS** *executes a depth-first search through a graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$ in $O(\sqrt{V^3 \lg V})$ time in the matrix model, $O(\sqrt{VE \lg(V\epsilon^{-1})})$ in the edge array model, with probability $\leq \epsilon$ that it will fail, performing the depth-first search incorrectly.*

1. Let the vertex from which we are searching be called v_a. Let there be a boolean array *vis* of size V, with its entries initialized to 0. Call DFS-BODY(v_a).

```
2. Function DFS-BODY(vertex $v_k$):
    (a) Visit $v_k$. Set $vis[k] = true$.
    (b) Repeat the following until instructed otherwise:
        i. Use Section 3.1's findsol to find a neighbour of $v_k$ that has
           not yet been visited, $v_i$. If findsol returns false, return.
        ii. Recursively call DFS-BODY($v_i$).
```

Since *DFS* makes a call to a function that may fail, *findsol*, there is some probability that it fails. We may pass the *DFS* function some constant ϵ, a probability of failure that we are willing to tolerate; since *DFS* calls *findsol* no more than $2V$ times—one call to find each vertex, and one call returning *false* from each vertex—if we require that *findsol* has probability $\leq \epsilon/2V$ of failure each time, the total probability with which *DFS* fails will be less than ϵ.

For each of the calls to *findsol* that returns *false*, we find a contribution (in the edge array model) of $\sqrt{|d_i| \lg(2V\epsilon^{-1})}$ to our running time, which sums over all vertices to $O(\sqrt{VE \lg(V\epsilon^{-1})})$ by the Cauchy-Schwartz inequality (see equation 4.2). We must also sum the running times of the successful *findsol*s: we note that for each vertex v_i, if we end up finding n_i of its neighbours through DFS-BODY(v_i), the running time of that will be $O\left(\sum_{k=1}^{n_i} \sqrt{(|d_i|/k) \lg(2V\epsilon^{-1})}\right) = O(\sqrt{|d_i|\, n_i \lg(V\epsilon^{-1})})$. Summing that contribution over each vertex, we again use the Cauchy-Schwartz inequality (equation 4.2) to arrive at $O(\sqrt{VE \lg(V\epsilon^{-1})})$. In the matrix model we simply replace E with V^2, arriving at $O(\sqrt{V^3 \lg(V\epsilon^{-1})})$.

Classically the fastest possible depth-first search algorithm takes $\Theta(E)$ time in the edge array model, and $\Theta(V^2)$ time in the adjacency matrix model; so for bounded error probability, *BFS* is faster than its classical counterpart in the matrix model, and faster than its classical counterpart in the edge array model for $E \in \Omega(V \lg \epsilon^{-1})$.

4.3 Single-Source Shortest Paths with Negative Edge Weights, *SPNW*

Classically, the problem of single-source shortest paths with negative edge weights is solved using an algorithm called Bellman-Ford[23, 24], on which we base our algorithm.[7] Our algorithm returns an array of shortest distances

[7]Thanks to Yury Kholondyrev for a discussion that led to this algorithm.

to points, or the special value *false* if there exists a negative-weight cycle in the graph that can be reached from the source. It also computes an array *from*, whose i^{th} element is the index of the vertex previous to v_i on the shortest path from v_a to v_i; this allows the shortest path from v_a to v_i to be recovered.

Intuitively, we are going to take each edge in turn and see if it helps our current shortest path to each point (a technique called "relaxation"); we repeat that process V times, at which point each edge will have helped all it can.

Theorem 7 *Given a graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, the following algorithm* **SPNW** *returns an array whose i^{th} element is the shortest distance from the source v_a to vertex v_i, or ∞ if no such path exists. If there is a negative weight cycle that can be reached from v_a, instead of an array it returns the special value false. It does this in $O(\sqrt{V^5 \lg(V\epsilon^{-1})})$ time in the matrix model, $O(\sqrt{V^3 E \lg(V\epsilon^{-1})})$ in the edge array model, with probability $\leq \epsilon$ that it fails, returning an incorrect result.*

1. If we are using the edge array model, set up an array f such that $f[i][j]$ is the source of the j^{th} edge incident on i.

2. Initialize an array *dist*, such that *dist*$[i] = \infty$ for $i \neq a$, 0 for $i = a$.

3. Initialize an array *from*, such that *from*$[i] = -1$.

4. Repeat the following $V - 1$ times:

   (a) For each vertex v_i, using the algorithm of Section 3.2, *minfind* a vertex v_j such that e_{ji} exists, and *dist*$[j] + \text{length}(e_{ji})$ is minimized. Execute the minfind by searching over $f[i]$ in the edge array model, $\mathbb{V}$ in the matrix model.

   (b) If *dist*$[j] + \text{length}(e_{ji}) <$ *dist*$[i]$, set *dist*$[i] =$ *dist*$[j] + \text{length}(e_{ji})$ and set *from*$[i] = j$.

5. Repeat step 4a one more time. If it changes *dist*, return *false*. Otherwise return *dist*.

Since *SPNW* makes a call to a function that may fail, *minfind*, there is some probability that it fails. We may pass the *SPNW* function some constant ϵ, a probability of failure that we are willing to tolerate; since *SPNW* calls *minfind* V^2 times, if we require that *minfind* has probability

$\leq \epsilon/V^2$ of failure each time, the total probability with which *SPNW* fails will be less than ϵ.

This algorithm, like Bellman-Ford, works due to the fact that all shortest paths in a graph without negative weight cycles must use fewer than V edges. Each time through step 4, we ask "could the path to vertex v_i be shorter if we were allowed to use one more edge?" Repeating this $V-1$ times lets us use $V-1$ edges, and repeating it a last time lets us check whether there is a negative weight cycle. Meanwhile we keep our array *from*, which tells us how we got to v_i and allows us to recover the whole path. In the edge array model, the running time is $V\sum_i \sqrt{|d_i|\lg(V\epsilon^{-1})} = O(\sqrt{V^3E\lg(V\epsilon^{-1})})$, again making use of the Cauchy-Schwartz inequality (equation 4.2). In the matrix model, our E becomes a V^2 as usual, and we have $O(\sqrt{V^5\lg(V\epsilon^{-1})})$. Since this is greater than V^2, if the graph is sparse it may be worth first converting to the edge array model.

Classically there are two fastest known algorithms for single-source shortest paths: Bellman-Ford, discussed above, takes $\Theta(VE)$ time in the edge array model, $\Theta(V^3)$ time in the adjacency matrix model; and an algorithm by Zwick[25], which computes all-pairs shortest paths, runs in $\Theta(V^{2.575})$ in both models. For bounded error probability, *SPNW* is faster than both in the matrix model, faster than Zwick's algorithm in the edge array model, and faster than Bellman-Ford in the edge array model when $E \in \Omega(V\lg V)$.

4.4 All-Pairs Shortest Paths with Negative Edge Weights, *APSP*

Theorem 8 *Given a graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, the following algorithm* **APSP** *returns an array whose i, j^{th} element is the length of the shortest path between vertices v_i and v_j, ∞ if no such path exists. If there is a negative weight cycle in the graph, instead of an array it returns the special value false. It does this in $\Theta(\sqrt{V^5}\left(\lg V + \sqrt{\lg(V\epsilon^{-1})}\right) + V^2\lg^3 V)$ in the matrix model, $\Theta(\sqrt{V^3E}\left(\lg V + \sqrt{\lg(V\epsilon^{-1})}\right) + V^2\lg^3 V)$ in the edge array model.*

We can do this directly with Johnson's algorithm[19, 26][8], which we will lay out in classical terms and then make obvious substitutions with our own quantum subroutines. Johnson's works by running Dijkstra's algorithm

[8]Thanks to Wei-Lung Dustin Tseng, Michael Li and Man-Hon "Matthew" Chan for simultaneously suggesting that looking at Johnson's might yield something better than than the algorithm I was presenting to them at the time.

from every origin point, which gives the shortest paths from all points to all other points; the difficulty is that Dijkstra's does not work in graphs with negative-weight edges, so first it is necessary to reweight edges so that all of their weights are positive.

When reweighting the edges, we need to reweight them in such a way that running Dijkstra's algorithm will return a genuine shortest path. It is not enough, for example, to pick a large number and add it to each edge; this would cause Dijkstra's algorithm to favour paths that used few edges. We will instead apply a label $a[i]$ to each vertex v_i, and let $\text{weight}'(e_{ij}) = \text{weight}(e_{ij}) + a[i] - a[j]$. If we run Dijkstra's algorithm using the weight$'$s, any path from v_x to v_y will be reweighted by $a[x] - a[y]$; every vertex in between will have its a value subtracted and then added. Thus genuine shortest paths will be found, with the only caveat being that the paths' lengths need to be reweighted.

Now that we know we can label vertices and reweight edges appropriately, we need to decide how to label the vertices. Let $a[i]$ be the length of the shortest path from *anywhere* to v_i. This will be 0 at most, since we define the distance from v_i to itself to be 0. If there is a negative-weight edge going to v_i, it may be the weight of that edge. We will find these distances by inserting an imaginary node s into the graph with weight-0 edges going everywhere, and then using Bellman-Ford to find the shortest path to each node from there. If the shortest path to v_i comes from v_j, then there is a path of identical length that comes from s. Using Bellman-Ford also tells us right away whether there is a negative-weight cycle in the graph.

Now we need to show that this labelling causes our reweighting to give only nonnegative weights to edges. Let there be an edge e_{ij} between v_i and v_j, with weight w. Our reweighted edge has weight $w' = w + a[i] - a[j]$. $a[i]$ is the shortest distance to v_i from anywhere, and so the shortest distance to v_j from anywhere is at most $a[i] + w$; so we have $a[j] \leq a[i] + w$, and $w' \geq 0$.

Now we have a way of altering our graph that preserves shortest paths and gives nonnegative weights to all edges. From there, we can run Dijkstra's algorithm from each vertex, fixing our distances only when we're done, and so find the shortest paths (and distances) to each vertex from each vertex.

The quantum version for this algorithm is a straightforward adaptation. We replace Bellman-Ford with our *SPNW* algorithm from Section 4.3, and Dijkstra's algorithm with our adaptation of Ambainis and Špalek's single-source shortest path algorithm from Section 4.5.2.

Since *APSP* makes calls to functions that may fail, *SPNW* and single-source shortest paths, there is some probability that it fails. We may pass the *APSP* function some constant ϵ, a probability of failure that we are willing to tolerate; since *APSP* calls *SPNW* and single-source shortest paths a total of $V+1$ times, if we require that both have probability $\leq \epsilon/(V+1)$ of failure each time, the total probability with which *APSP* fails will be less than ϵ.

One *SPNW* and V single-source shortest paths, each with failure probability $\leq \epsilon/(V+1)$, take a total of $\Theta\left(\sqrt{V^5}\left(\lg V + \sqrt{\lg(V\epsilon^{-1})}\right) + V^2 \lg^3 V\right)$ in the matrix model, $\Theta\left(\sqrt{V^3 E}\left(\lg V + \sqrt{\lg(V\epsilon^{-1})}\right) + V^2 \lg^3 V\right)$ in the edge array model.

Classically there are two fastest known algorithms for all-pairs shortest paths with negative edge weights: Johnson's algorithm takes $\Theta(VE + V^2 \lg V)$ time in the edge array model, $\Theta(V^3)$ time in the matrix model; Zwick's algorithm runs in $\Theta(V^{2.575})$ in both models. So for bounded error probability, *APSP* is faster than Johnson's algorithm in the edge array model for $E \in \Omega(V \lg^4 V)$, is always faster in the matrix model than Johnson's algorithm, and is faster than Zwick's algorithm in both models.

4.5 Improvements to Existing Quantum Graph Algorithms

It has quickly become to the tradition in the literature[8, 9] to devise quantum algorithms with *BBHT* as though there were no probability that it could fail, and then to throw a factor of $\log(N)$ into the running time at the end to take the probability of failure into account. Here we give two examples of algorithms that can be given faster asymptotic behaviour with careful treatment of errors (done by using the tools introduced in Chapter 3).

4.5.1 Query Complexity

In the algorithms we discuss here, the authors chose to find the quantum *query complexity* of their algorithms, rather than the running time. The query complexity is the number of times a graph is "queried": how many times the algorithm has to ask "is there an edge between v_i and v_j?" in the adjacency matrix model, or ask "what is the j^{th} edge coming out of vertex v_i?" in the edge array model.

We have chosen to analyze running time rather than query complexity for the algorithms here, as well as the algorithms elsewhere in the paper, because we hope the day will come when quantum and classical algorithms will be run on the same computer, at the same speed; and so an $\Theta(\sqrt{V^3 \lg V})$ quantum algorithm might practically be said to be faster than an $\Theta(V^2)$ algorithm. There are of course many reasons to prefer both, but this is the one we have chosen.

4.5.2 Single-Source Shortest Paths

Dürr, Heiligman, Høyer and Mhalla (DHHM) discuss algorithms for single-source shortest paths, minimum spanning tree, connectivity and strong connectivity[8]. The quantum query complexity for their single-source shortest paths, $O(\sqrt{VE} \lg^2 V)$, can be improved by using *mindiff*, whereupon it becomes $O(\sqrt{VE} \lg V)$. The explanation follows.

We will begin by trying to motivate this algorithm as a speedier version of Dijkstra's $\Theta(E \lg V)$ algorithm. That algorithm involves iteratively building up a set T of nodes, the shortest path to each of which we already know; each iteration expands that set by one element by finding the next-closest node to a, the origin of the search. First we will present a classical Dijkstra's algorithm, then discuss how that can be improved using quantum methods.

Our Dijkstra's and quantum single-source shortest paths will make use of priority queues: these will be implemented as balanced binary search trees with $\Theta(\lg N)$ access, insertion and deletion. The knowledgeable reader will note that Fibonacci heaps are generally more appropriate for Dijkstra's algorithm; while that is true, they do no better in the quantum single-source shortest paths algorithm, and so we avoid them for simplicity's sake. The reader who is used to C++ may imagine our priority queue to be a set⟨pair⟨int, pair⟨int, int⟩⟩⟩.

We begin with the classical Dijkstra's algorithm:

1. Construct an array *Trace* of size V, and initialize its entries to -1.
2. Construct an array *LeastCost* of size V, and initialize its entries to ∞.
3. Let there be a priority queue Q of edges, initially containing the (*priority*, *value*) pair $(0, (a, -1))$.
4. While Q is not empty, repeat the following:
    (a) Remove the least value from Q: call its priority *cost,* and its value $(\mathit{loc}, \mathit{from})$.
    (b) If $\mathit{LeastCost}[\mathit{loc}] \neq \infty$, we have already been to *dest*; go back to step 4a.

(c) Set $\mathit{LeastCost}[\mathit{loc}] = \mathit{cost}$ and $\mathit{Trace}[\mathit{loc}] = \mathit{from}$.

(d) For each element $d_{loc}[i]$ of d_{loc}, call its destination v and its weight c, and do the following: if $\mathit{LeastCost}[v] = \infty$, insert the $(\mathit{priority}, \mathit{value})$ pair $(\mathit{cost} + c, (v, \mathit{loc}))$ into Q.

5. $\mathit{LeastCost}[i]$ now contains the length of the shortest path from a to i, or ∞ if no such path exists; $\mathit{Trace}[i]$ is the predecessor to i in a shortest path from a to i (i.e. the path goes $a, \ldots, \mathit{Trace}[i], i$).

Step 4 will repeat V times at most: every time the loop runs, a new element of *LeastCost* will be set. Our most time-consuming step is thus step 4d, which is at the core of the algorithm: inserting edges into a priority queue so that the edge of least cost may be removed. A total of E edges will be inserted, and since the priority queue will be of size $O(E)$ this will cost $\Theta(E \lg E) = \Theta(E \lg V)$. Step 4a shares this running time.

Our goal in constructing the quantum single-source shortest path, then, will be to reduce the length of time taken by the equivalent of step 4d. Since most of our graph algorithms seem to offer speedups on the order of $\sqrt{E/V}$ speedups, we will shoot for something similar (and will in fact hit $\sqrt{E/V}/\lg V$).

Since the Fibonacci heap implementation of Dijkstra's algorithm takes $\Theta(E + V \lg V)$ time, if we wish to outdo that then we may not look at each edge classically; instead we must do some sort of *minfind* over edges. For the first few vertices, that seems to work: finding the closest vertex to the origin vertex a is roughly a $\Theta(\sqrt{V})$ operation. Of course, finding the next-closest vertex in the same way is $\Theta(\sqrt{2V})$, and so on up until the last vertex, which will cost $\Theta(\sqrt{E})$. This gives us something roughly $\Theta(V\sqrt{E})$, which is slower than Dijkstra's algorithm; somehow we need to reduce the number of *minfind*s that gets done.

The solution, due to Dürr et al., is to group vertices together, run a sort of *minfind* from the group, and then use that result as many times as possible. Consider the following: initially we just need to find the closest vertex to a, b. Now we can find the closest ("closest" always means "closest to a") two vertices that use edges coming out of $\{a, b\}$, by using *mindiff*; call the closer of those c, and the further of them d. We still have the next-closest vertex from $\{a, b\}$, and can re-use that result: so we find the closest vertex that uses edges coming from $\{c\}$, and see whether it or d is closest (assume for the sake of example that this is d). Now we can merge d into $\{c\}$, which becomes the same size as $\{a, b\}$, so we merge them together to get $\{a, b, c, d\}$ and find the closest 4 vertices that use edges coming from that set. We continue in this fashion, merging sets whenever two sets are the same size, and only ever re-computing the closest 2^x vertices to a set when it changes. The procedure is formalized below:

1. Construct an array *Trace* of size V, and initialize its entries to -1.
2. Construct an array *LeastCost* of size V, and initialize its entries to ∞. Set *LeastCost*$[a]$, the cost to get to the origin of the search, to 0.
3. Let there be a priority queue Q of edges, initially empty.
4. Let there be a stack of vertex sets S, initially containing $\{a\}$, the origin vertex. We will refer to the current top element of the stack by *S.top*.
5. Repeat the following until instructed to exit the loop:
   (a) Call the function *SSSP–Mindiff*, defined below; call the resulting set R.
   (b) For each element $R[i]$ of R, call its destination v, its source u and its weight c, and insert the (*priority*, *value*) pair $(c, (v, u))$ into Q.
   (c) Remove the least value from Q: call its priority *cost,* and its value (*loc*, *from*). Repeat this until *LeastCost*$[from] = \infty$ or Q is empty.
   (d) If Q is empty, go to step 6.
   (e) Set *LeastCost*$[loc]$ = *cost* and *Trace*$[loc]$ = *from*. Push $\{loc\}$ onto S.
   (f) As long as S has at least two elements, and the top two elements of S have the same size, merge the top two elements of S.
6. *LeastCost*$[i]$ now contains the length of the shortest path from a to i, or ∞ if no such path exists; *Trace*$[i]$ is the predecessor to i in a shortest path from a to i (i.e. the path goes $a, \ldots,$ *Trace*$[i], i$).

We would like to run *mindiff* on the edges outgoing from our sets, but *mindiff* operates on the domain $\{0, \ldots, N-1\}$; as such, we have to adapt it to work on several disjoint collections of objects. This adds a factor of $\lg V$ to our algorithm and to DHHM's (see below).

Function **SSSP–Mindiff:**

1. Create a table T of size $|S.top|$ such that $T[i]$ is $\sum_{j=0}^{i} \left|d_{S.top[j]}\right|$.

2. Let $k = |S.top|$, and $N = T[k-1]$ (the number of edges out from all vertices in $S.top$).

3. Define the mapping $H(i) =$ the i^{th} edge out from all vertices in $S.top$. $H(i)$ can be computed by doing a binary search on T, followed by some basic arithmetic.

4. Define $c(i)$ to be edge $H(i)$'s weight, $u(i)$ to be its source, and $v(i)$ to be its destination.

5. Run Section 3.4's *mindiff*, using N as above, $d = k$, $F(i) = c(i) + \mathit{LeastCost}[i]$, and $G(i) = v(i)$. Return the resulting set.

This algorithm is different from DHHM's in that our *mindiff* algorithm is more carefully constructed than the version they use, in two respects. The first is that theirs has constant probability of failure, and they repeat it $\Theta(\lg(V))$ times, leading to slower performance than ours overall. The second is that they do not make some of the specifics of their algorithm clear: for example, they do not describe how they maintain their list of best answers so far. This will inevitably add to the total running time of their algorithm (though not its queries to the graph, which is what they chose to analyze), and so their *mindiff*'s running time ends up as $O\left((t_F + t_G)\sqrt{Nd}\lg N + \text{some quantity}\right)$, at a contrast to our faster $O\left((t_F + t_G)\left(\sqrt{Nd} + \sqrt{N\lg\epsilon^{-1}}\right) + d\lg N\lg d\right)$.

The running time analysis for our *mindiff* and theirs is not complete; the evaluation of F and G as defined in *SSSP–Mindiff* is a $\Theta(\lg N)$ process. Our *mindiff* thus takes $O\left(\sqrt{Nd}\lg N + \sqrt{N\lg\epsilon^{-1}}\lg N + d\lg N\lg d\right)$ operations here, whereas theirs takes $O\left(\sqrt{Nd}\lg^2 N + \text{some quantity}\right)$.[9]

Since this algorithm makes calls to a function that may fail, *mindiff*, there is some probability that it fails. We may pass the function some constant ϵ, a probability of failure that we are willing to tolerate; since it calls *mindiff* no more than V times, if we require that each call to *mindiff* has probability $\leq \epsilon/V$ of failure each time, the total probability with which our algorithm fails is less than ϵ.

We may now analyze our number of graph queries compared to theirs: as far as querying the graphs go, our algorithms are identical but for the

[9]Rather than performing a binary search as we do, DHHM iterate through all of the edges connected to $S.top$ and label each one individually. Had they cared about running time, they doubtless would not have taken this approach.

extra factor of $\lg V$ on their *mindiff* that arises due to errors. This becomes an extra factor of $\lg V$ in the algorithm's running time, for a total query complexity of $O(\sqrt{VE}(\lg V + \sqrt{\lg(V\epsilon^{-1})}))$ for our algorithm (vs. $O(\sqrt{VE}\lg V \lg(V\epsilon^{-1})x)$ for theirs).

Now that we know how long our *mindiff* step takes, we may evaluate our running time. We perform this calculation only for our own algorithm: DHHM's time complexity is identical but for the extra $\lg V$ on the leading term, the verification of which is an exercise that the reader is welcome to perform. Our time complexity has two major parts: insertion/removal of elements in Q, and the function *SSSP–Mindiff*. Merging sets is briefly worth consideration, but recall that *mindiff* is immediately run on sets after they are merged, and that takes at least $\Omega(\text{size})$ time.

In order to determine the time complexity for inserting/removing elements from Q, we will first examine how Q is updated. Q is updated every time a new *S.top* appears, which is at the beginning of every iteration through step 5. The sequence of update sizes follows:

$$1, 2, 1, 4, 1, 2, 1, 8, 1, 2, 1, 4, 1, 2, 1, 16, 1, \ldots \tag{4.3}$$

Note that 1 shows up as every second digit, 2 as every fourth, etc. We can thereby say that for each size s, it appears no more than V/s times in the sequence. Since the sizes are powers of two, we have a total of $\leq \sum_{i=0}^{\lg V} V/2^i \times V = V \lg V$ elements in Q. Since insertion takes an average of $\Theta(\lg V)$ time, the time used in the maintenance of the priority queue is bounded from above by $O(V \lg^2 V)$.

Next we need the time used by *mindiff*. The frequency of set sizes is identical, so for each size we will sum over all sets of that size that we encounter throughout the algorithm. Let m_{ij} be the number of edges coming from the j^{th} set of size 2^i:

$$t_{\text{mindiff}} \leq \sum_{i=0}^{\lg V} \sum_{j=0}^{V/2^i} \left(\sqrt{m_{ij} 2^i} \lg m_{ij} + \sqrt{2^i \lg(V\epsilon^{-1})} \lg m_{ij} + i 2^i \lg m_{ij} \right) \tag{4.4}$$

Note that no vertex will be in two different sets of the same size, and so $\sum_{j=0}^{V/2^i} m_{ij} = E$.

$$t_{\text{mindiff}} \leq \sum_{i=0}^{\lg V} \left(\sqrt{E 2^i \frac{V}{2^i}} \lg V + \sqrt{2^i \lg(V\epsilon^{-1})} \frac{V}{2^i} \lg V + i 2^i \frac{V}{2^i} \lg V \right) \tag{4.5}$$

$$t_{\text{mindiff}} \leq \sqrt{EV} \lg^2 V + V \lg V \sqrt{\lg(V\epsilon^{-1})} + V \lg^3 V \tag{4.6}$$

This is the dominant term, and so our algorithm takes $O(\sqrt{EV}\lg^2 V + V\lg^3 V + V\lg V\sqrt{\lg(V\epsilon^{-1})})$ time, and DHHM's takes $O(\sqrt{EV}\lg^2 V\lg(V\epsilon^{-1}) + V\lg^3 V)$ time.

The fastest known classical solution to this problem, Dijkstra's algorithm, runs in $\Theta(E + V\lg V)$, $\Theta(V^2)$ in the matrix model; so for bounded error probability, our algorithm is faster than its classical counterpart in the matrix model, and faster in the edge array model for $E \in \Omega(V\lg^4 V)$. Note that the fastest possible classical algorithm must be $\Omega(E)$, so our algorithm is also faster than that for $E \in \Omega(V\lg^4 V)$ in the edge array model, and always in the matrix model.

4.5.3 Bipartite Matching

Ambainis and Špalek[9] address bipartite matching, non-bipartite matching and maximum flow. Their algorithm for bipartite matching runs in $O(V\sqrt{E+V}\lg V)$ time, and is a quantum adaptation of Hopcroft and Karp's classical $\Theta((E+V)\sqrt{V})$ algorithm[27]; we further modify their algorithm, solving the problem here in $O(V\sqrt{(E+V)\lg V})$ time.

The problem of bipartite matching can be described in several ways: for example, consider a collection of boys and girls to be vertices of a graph, and have an edge in the graph for each $(\mathit{boy}, \mathit{girl})$ pair that would make a good couple. In bipartite matching, we pair off the boys and girls in such a way that only compatible couples are paired, each person has at most one partner, and there is a maximum number of pairings.

Some basic principles underlie most solutions to this problem. Consider some (non-maximum) matching-so-far $\mathbb{M}$ between boys and girls; if we can construct a path $\mathbb{P}$ starting at an unmatched boy and ending at an unmatched girl such that all edges in the path are either unmatched $(\mathit{boy}, \mathit{girl})$ edges or matched $(\mathit{girl}, \mathit{boy})$ edges, then the old matching can be expanded by one more pair by taking $\mathbb{M}' = \mathbb{M} \oplus \mathbb{P}$ (where $\mathbb{M} \oplus \mathbb{P}$ means taking all edges in either $\mathbb{M}$ or $\mathbb{P}$, but not both). Intuitively, where $\mathbb{M}$ and $\mathbb{P}$ have an edge in common, we are "unmatching" that $(\mathit{boy}, \mathit{girl})$ pair, and "rematching" the two using the surrounding edges in the path. Because this path augments $\mathbb{M}$ by adding one to its size, it is called an augmenting path. Figure 4.1 illustrates a single execution of this procedure.

The principle behind Hopcroft and Karp's algorithm is as follows: suppose that every time we want to find an augmenting path $\mathbb{P}$, we find the shortest such path. They proved that if we do that, we will see at most $2\sqrt{V}$ different path lengths in the whole process of constructing a maximum matching. So if we devise a process to find a *maximal* set of disjoint aug-

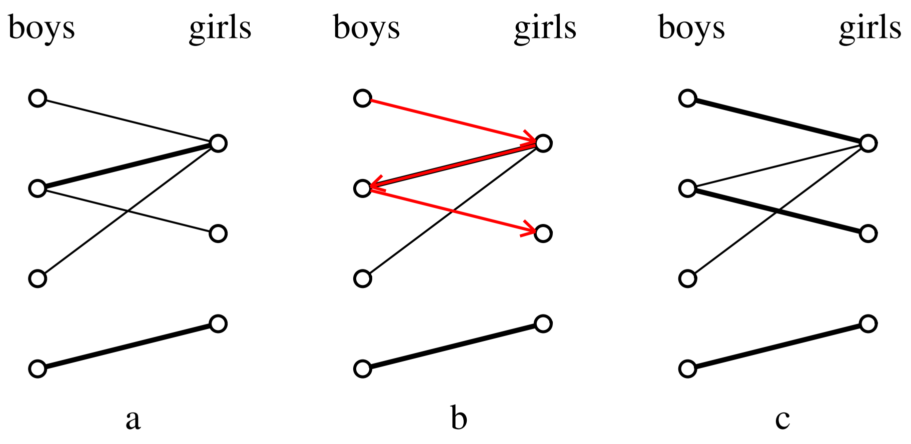


Figure 4.1: (a) A non-maximum matching $\mathbb{M}$. (b) An augmenting path $\mathbb{P}$, using unmatched (*boy*, *girl*) edges and matched (*girl*, *boy*) edges. (c) A new matching, $\mathbb{M}' = \mathbb{M} \oplus \mathbb{P}$. Dark edges indicate edges that are part of the current matching.

menting paths of minimal length (maximal means here that the set cannot be expanded by adding more paths of the same length), each time we execute that procedure, the next set will have longer length. We can thus repeat that process $O(\sqrt{V})$ times, seeing that number of distinct lengths, and end up having constructed a maximum matching.

The construction of such a set of augmenting paths can be accomplished through the use of a breadth-first search followed by a modified depth-first search, which we will replace by our own *BFS* and *DFS*. We outline these steps below, and show them diagrammatically in Figures 4.2; we suggest following the steps in the figures in parallel with those written below.

1. Find the set of all unmatched girls. Consider there to be an edge to each of those nodes from a new "fictional" node t.

2. Find the set of all unmatched boys. Consider there to be an edge from each of those nodes to a new "fictional" node s.

3. Let there be an array a of size V whose entries are initialized to ∞.

4. Conduct a breadth-first search originating at t, where the only edges permitted for use are *matched* (*boy*, *girl*) edges, *unmatched* (*girl*, *boy*) edges, and the edges incident on s and t. When visiting a vertex v_i,

set $a[i]$ to v_i's distance from t. If the breadth-first search terminates without reaching s, return that no augmenting path exists; otherwise stop as soon as the search reaches s. Perform this search using *BFS*.

5. Reverse all the edges incident on s and t.

6. Any path running from s to t that uses unmatched (*boy*, *girl*) edges and matched (*girl*, *boy*) edges will be an augmenting path. If we also require that each step $v_i \to v_j$ brings us closer to t ($a[i] = a[j] + 1$), any such path will be an augmenting path of minimal length.

7. Conduct a depth-first search originating at s, where an edge e_{ij} can only be used if $a[i] = a[j] + 1$, and the only such edges permitted for use are unmatched (*boy*, *girl*) edges, matched (*girl*, *boy*) edges, and edges incident on s and t. When visiting an unmatched girl (whose distance from t will be 1), trace back the path taken, mark it as an augmenting path, and resume the search from s (not visiting any of the nodes on the path again).

Hopcroft and Karp have done the work for us of proving that the algorithm works in principle, leaving us to show that the procedure outlined above finds a maximal set of disjoint augmenting paths of minimum length. To show that, we need to show that the paths we find are all of minimum length; and that after this procedure, there is no augmenting path remaining that has the same length and is completely disjoint.

The latter condition is straightforward. Assume there is such a path $\mathbb{P}$: then at some point it must reach an unmatched girl, and it will connect that unmatched girl to t.

Since our initial breadth-first search computes the distance from each vertex to t (and thus from each vertex to some unmatched girl), the requirement that each step reduces that number ensures that all paths found in our depth-first search are of minimum length.

The running time for our algorithm is straightforward to compute given the results for *BFS* and *DFS*. Hopcroft and Karp proved that our procedure (above) needs to be run no more than $2\sqrt{V}$ times. The procedure itself comprises a *BFS*, a *DFS*, and some number of backtraces. The backtraces use each node found in the *DFS* at most once, and so do not add to the *DFS*' running time.

Since this algorithm makes calls to functions that may fail, *BFS* and *DFS*, there is some probability that it fails. We may pass our function some constant ϵ, a probability of failure that we are willing to tolerate; since it calls

BFS and *DFS* no more than a total of $4\sqrt{V}$ times, if we require that *BFS* and *DFS* have probability $\leq \epsilon/4\sqrt{V}$ of failure each time, the total probability with which bipartite matching fails will be less than ϵ. Our breadth-first and depth-first searches thus have a running time of $O(\sqrt{VE \lg(V\epsilon^{-1})})$.

Ambainis and Špalek's algorithm, on which ours is based, is almost identical; the difference is that where our *BFS* and *DFS* use *findsol* and *findall*, theirs use repeated applications of *BBHT*. This causes their breath-first and depth-first searches to run in $O(\sqrt{VE} \lg \epsilon^{-1})$ time, whereas ours run in $O(\sqrt{VE \lg \epsilon^{-1}})$ time. Our total running time is thus $O(V\sqrt{E \lg(V\epsilon^{-1})})$, a factor of $\sqrt{\lg V}$ faster than Ambainis and Špalek's $O(V\sqrt{E} \lg(V\epsilon^{-1}))$.

Ambainis and Špalek also discuss non-bipartite matching and maximum flow in the same paper; in both cases they ignore errors for the body of their algorithms, and throw on an extra factor of $\lg V$ at the end in order to reduce the probability of failure to a constant. While that works, this section shows that it is not necessarily optimal for bipartite matching; and due to the similarity of bipartite matching to the other problems they consider, it is reasonable to conjecture that one could also achieve a speedup on the order of $\sqrt{\lg V}$ for general matching and flow.

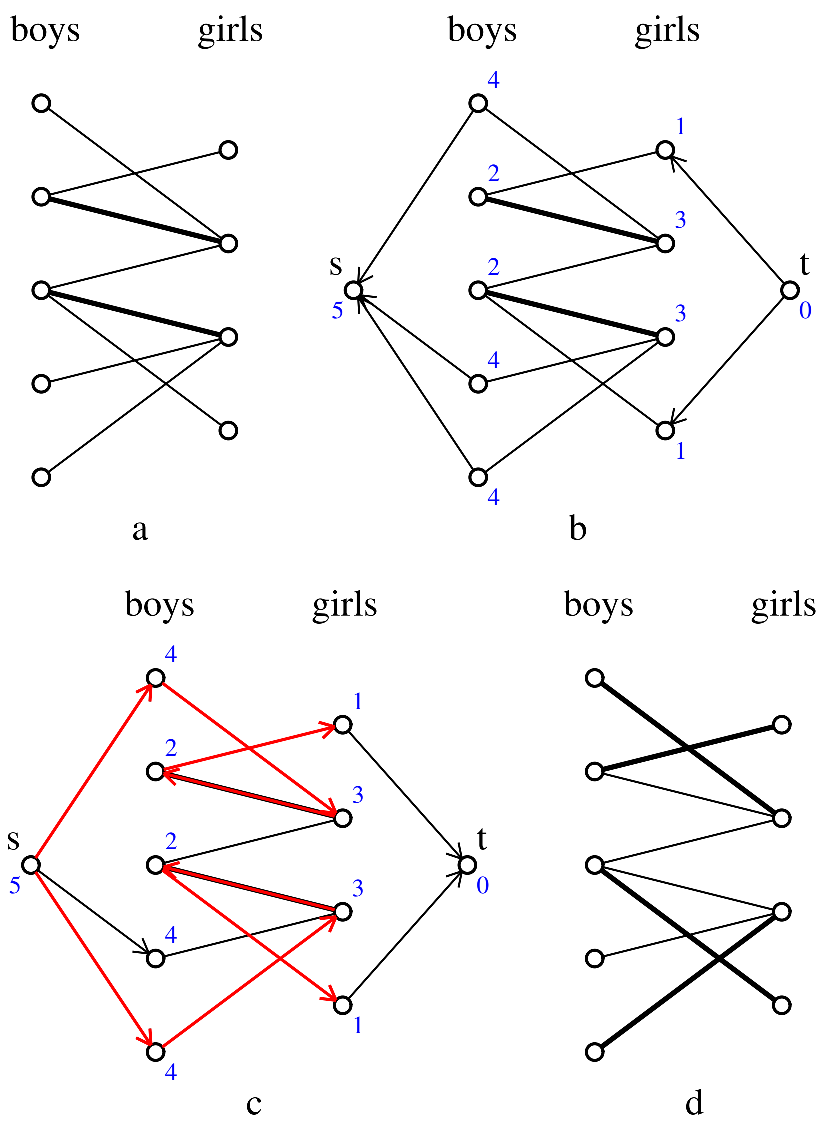


Figure 4.2: (a) A non-maximum matching $\mathbb{M}$. (b) After steps 1-4. Nodes are labelled by their distance from t; for the purposes of this step, matched (thick) edges go left-to-right, and unmatched edges go right-to-left. (c) After steps 5-7. Two possible augmenting paths are shown in red. Here, matched edges go left and unmatched edges go right; the thin edge in the middle cannot be used, however, because it would mean increasing distance from t (and thus lead to an augmenting path not of minimal length). (d) A new matching, $\mathbb{M}' = \mathbb{M} \oplus \{paths\}$.

5. Applications in Computational Geometry and Dynamic Programming

5.1 Computational Geometry Algorithms

Geometry problems are a natural area of attack for quantum algorithms, because by defining N points we have implicitly defined $\Theta(N^2)$ relationships between those points, making it very natural to ask questions whose answers use information $\Theta(N^2)$ in the size of the question. We will address points, which will be in either $\mathbb{Z}^d$ or $\mathbb{R}^2$, as p_i.

5.1.1 Maximum Points on a Line, *maxpoints*

This problem is, in all of its generality, a very simple one: given N points, find the line that goes exactly through the maximum number of them. We differentiate here between a solution that is practical for integers[10] and a slightly slower solution that is practical for real numbers; acknowledging that practical computers, however quantum, have precision issues where real numbers are concerned.

Intuitively each algorithm works by taking a single point p and finding out how many points are on the best line that goes through p. We then use *minfind* to find the best such p. In the $\mathbb{Z}^d$ case, our method is to find the vector from p to each other point, canonicalize it using GCD, and then stick all those vectors into a hash table so that we can quickly count repeats. In the $\mathbb{R}^2$ case, our method is to sort the points in counterclockwise order about p and see look for collinear points, which should now be ordered consecutively.

This is a particularly interesting problem to solve in $\mathbb{Z}^2$ because it is a member of a class of classical problems called "3SUM-hard"[28]. Of the

[10] Thanks to Kory Stevens for a productive conversation that removed a factor of $\lg N$.

problems belonging to this class, all of the known ones have classical lower-bounds of at most $\Omega(N)$, and upper bounds of at least $O(N^2)$. All problems in the class are reducible from the 3SUM problem: given a set S of N integers, is there some triplet a, b, c in that set such that $a + b + c = 0$? That is quite a straightforward problem to solve with *findsol* in $\Theta(N)$, while we will solve *maxpoints* in $N^{1.5}$; thus we open a gap of $\sqrt{N}$ between two similar problems where no such gap existed before. This raises interesting questions about the maximum points on a line problem, and a number of other problems in 3SUM-hard. This in turn suggests that many of the algorithms in 3SUM-hard may be amenable to sub-N^2 quantum solutions.

5.1.2 Maximum Points on a Line: $\mathbb{Z}^d$

Theorem 9 *Let there be N points in $\mathbb{Z}^d$, whose coordinates are bounded by $\pm U$. The following algorithm* **maxpoints** *finds the straight line on which lies the maximum number of those points, in $\Theta(N^{3/2} d \lg U \sqrt{\lg \epsilon^{-1}})$ time and with probability $\leq \epsilon$ that it will fail, returning an incorrect result.*

1. Use Section 3.2's *minfind* to maximize the following function, *mup* (maximum using p), over all points p. Call the result P.
2. Function **mup(p)**:
   - (a) Create an empty hash table H, mapping vectors in $\mathbb{Z}^d$ (keys) to integers (values).
   - (b) For each point p_i:
     - i. Define $\overrightarrow{a} = \overrightarrow{p_i} - \overrightarrow{p}$.
     - ii. Normalize $\overrightarrow{a}$, keeping its entries in the integers, so that the first nonzero component is positive and the gcd of the absolute values of the components is 1.
     - iii. If $\overrightarrow{a}$ is not yet in H, insert it in H mapping to value 1; if $\overrightarrow{a}$ is already in H, increment its value.
   - (c) Return the maximum value in H: the number of points on the best line going through p.
3. Run *mup* on P, but instead of returning the maximum value in the hash table return its corresponding key, and call it $\overrightarrow{V}$.
4. The answer to return is the line $\overrightarrow{X(t)} = \overrightarrow{P} + t\overrightarrow{V}$.

In *mup*, all vectors to other points from p are canonicalized in such a way that any pair of points collinear with p will have the same direction vector $\overrightarrow{a}$. *mup* repeats d *gcd*s N times, for a total of $\Theta(Nd\lg U)$, and our main function's most costly operation is one *minfind* that evaluates *mup* $\Theta(\sqrt{N\lg\epsilon^{-1}})$ times. Thus our total running time is $\Theta(N^{3/2}d\lg U\sqrt{\lg\epsilon^{-1}})$, and our probability of failure is ϵ. Classically the problem can be solved in $\Theta(N^2 d\lg U)$.

5.1.3 Maximum Points on a Line: $\mathbb{R}^2$

Theorem 10 *Let there be N points in $\mathbb{R}^2$. The following algorithm finds the straight line on which lies the maximum number of those points in $\Theta(N^{3/2}\lg N\sqrt{\lg\epsilon^{-1}})$, with probability $\leq \epsilon$ that it will fail, returning some incorrect result.*

1. Use *minfind* to maximize the following function, *mup2*, over all points p. Call the result P.

2. Function **mup2(p)**:

   (a) Let $\overrightarrow{a_i} = \overrightarrow{p_i} - \overrightarrow{p}$. If $\overrightarrow{a_i}.x < 0$, or $\overrightarrow{a_i}.x = 0$ and $\overrightarrow{a_i}.y < 0$, then reverse $\overrightarrow{a_i}$. This puts all points to the right of p.

   (b) Sort the $\overrightarrow{a_i}$ as follows: $\overrightarrow{a_i} < \overrightarrow{a_j}$ iff $(\overrightarrow{a_i} \times \overrightarrow{a_j}) \cdot \widehat{z} > 0$. This has the effect of sorting the p_i in counter-clockwise order about p.

   (c) Iterate over the sorted array, keeping a running total of how many consecutive $\overrightarrow{a_i}$ have cross product of 0 with one another. Return the maximum such total. (Practically, we should see how many consecutive $\overrightarrow{a_i}$ have cross product $< \delta$ for some small δ, and loop through a second time to catch the nearly-straight-up and nearly-straight-down $\overrightarrow{a_i}$).

3. Run *mup*2 on P, but instead of returning the maximum total, return some point (other than P) on the line giving that total. Call it P'.

4. The answer to return is the line $\overrightarrow{X(t)} = \overrightarrow{P} + t(\overrightarrow{P'} - \overrightarrow{P})$.

This algorithm sorts the points about each point p, which has the effect of grouping collinear points together. Then it simply counts how many consecutive collinear points it can find. *mup2* is $\Theta(N\lg N)$, and our most costly operation is one *minfind* that evaluates *mup*2 $\Theta(\sqrt{N\lg\epsilon^{-1}})$ times, for a total running time of $\Theta(N^{3/2}\lg N\sqrt{\lg\epsilon^{-1}})$ and probability of failure ϵ. Classically this problem can be solved in $\Theta(N^2\lg N)$.

5.2 Dynamic Programming Algorithms

Dynamic programming (DP) is a technique for solving problems by combining the solutions to subproblems. DP algorithms achieve this by partitioning their problems into subproblems, solving the subproblems recursively, and then combining the solutions to solve the original problem. What distinguishes dynamic programming from other approaches is that the subproblems are not independent: subproblems share sub-subproblems with one another. A dynamic programming algorithm solves every sub-subproblem only once and saves its result in a table, thus eliminating the need to recompute the answer for a sub-subproblem every time it is needed[19].

Dynamic programming is often used to solve optimization problems. Given some situation (a problem), come up with a choice (each possible choice leads to a subproblem) that optimizes some final quantity (way down at the sub-. . . -subproblem level). We will see an example of this in Section 5.2.1. Since DP is often used to make some sort of optimal choice, DP algorithms in general are obvious candidates for Section 3.2's *minfind*, which reduces the time taken to check all our options.

5.2.1 Coin Changer, *coinchange*

Given a monetary system with some set of coins and bills, we may wish to make some precise amount of money – the **coin changer** problem is to use as few coins and bills as possible. Intuitively, this is easy: with Canadian or American money, for example, to make D cents one can simply take the largest bill/coin of value $v \leq D$, then make $D - v$ cents in the same way. For example, to make 40¢ one would take the largest coin less than 40¢ (25¢), then the largest coin less than the remaining 15¢ (10¢), and finally a 5¢ coin. This is a *greedy* approach that works for most real currencies, but it is not always optimal: for example, should a 20¢ piece be added to the Canadian system, then making 40¢ only takes two coins, but the greedy approach will still cause us to use three. Should the reader ever travel to Costa Rica or Bhutan, he or she will encounter a non-greedy currency system.[11]

Theorem 11 *Given a length C integer array of coin denominations V, as well as an integer D, the following algorithm* **coinchange** *returns the mini-*

[11]The Bhutanese ngultrum is divided into 100 chertrum. The three least valuable Bhutanese coins in modern circulation are the 20 chertrum coin, the 25 chertrum coin, and the 50 chertrum coin. Making 40 chertrum is not possible using the greedy technique; even with a 5-chertrum coin added, the greedy technique does not yield an optimal solution.

mum number of coins required to make D units, or ∞ *if making D units of currency is impossible. It achieves this in* $\Theta(D\sqrt{C}\lg D)$ *time.*

Since we are trying to minimize a quantity, the number of coins used, making D units optimally is a matter of choosing one coin $V[i]$ to use, then making $D - V[i]$ units optimally. To do so we build up a table T, where $T[i]$ is the minimum number of coins needed to make i units. We start by filling in $T[i]$ with i small, since later entries will depend on earlier ones.

```
1. Let there be an array $T$ of size $D + 1$, such that initially $T[0] = 0$, and
   $T[i \neq 0] = \infty$.
2. For $d$ from 1 to $D$, DO:
   (a) Use the algorithm of Section 3.2 to minfind one of the coins $V[i]$
       such that $d - V[i] \geq 0$, and $1 + T[d - V[i]]$ is minimal.
   (b) If such a coin was found, let $T[d] = 1 + T[d - V[i]]$.
   DONE.
3. Return $T[D]$.
```

Here we simply fill in the table as discussed above, by using *minfind* to determine which coin should be taken first. Since *coinchange* makes a call to a function that may fail, *minfind*, there is some probability that it fails. We may pass the *coinchange* function some constant ϵ, a probability of failure that we are willing to tolerate; since *coinchange* calls *minfind* D times, if we require that *minfind* has probability ϵ/D of failure each time, the total probability with which *BFS* fails will be less than ϵ. Each *minfind* takes $\Theta(\sqrt{C}\lg(D\epsilon^{-1}))$ time, and *minfind* is repeated D times for a total time complexity of $\Theta(D\sqrt{C}\lg(D\epsilon^{-1}))$.

The reason we discuss this example is because it is very representative of how one can improve dynamic programming algorithms in general using quantum techniques. Many other problems, for example minimum-operation matrix chain multiplication, can be solved quickly with a quantum algorithm in much this manner.

5.2.2 Maximum Subarray Sum, *subarray-sum*

Theorem 12 *Given an* $N \times N$ *array of real numbers* A, *the following algorithm* **subarray-sum** *finds a rectangular subarray such that the sum of the* *subarray's elements is maximized, in* $\Theta(N^2\sqrt{\lg \epsilon^{-1}})$ *time and with probability of failure* ϵ*. We will address the result by its limits: (miny, minx, maxy, maxx).*

This is another classic problem, for which the fastest known classical solution runs in $\Theta\left(N^3\sqrt{\frac{\log\log N}{\log N}}\right)$ and was found by Tamaki[10]. There is a much more straightforward (though still clever) $\Theta(N^3)$ solution, which involves maximizing the sum of all $\Theta(N^2)$ possible column ranges, each in $\Theta(N)$.

Our algorithm begins by creating a table T that makes checking the sum for an arbitrary rectangle $\Theta(1)$, and then simply *minfind*s over all rectangles. This algorithm, like the classical one, is really greedy rather than dynamic programming; we include it in this section because the construction of T is DP.

1. Let there be an $N \times N$ array T, whose i, j element will hold the sum for subarray $(0, 0, i, j)$. Initialize its entries to 0, and define $T[i][j] = 0$ if i or j is negative. The next step will fill in T as desired.

2. For i from 0 to $n-1$, For j from 0 to $n-1$ DO:

   (a) $T[i][j] = A[i][j] + (T[i-1][j] + T[i][j-1] - T[i-1][j-1])$.

   DONE.

3. There are N^4 possible rectangular subarrays. The summation over any such array is $T[maxy][maxx] - T[maxy][minx-1] - T[miny-1][maxx] + T[miny-1][minx-1]$, which is a $\Theta(1)$ calculation. Use the algorithm of Section 3.2 to *minfind* over all such *(miny, minx, maxy, maxx)* and find the subarray with the maximum summation, and then return it.

The creation of T takes $\Theta(N^2)$, and the *minfind* takes $\Theta(N^2\sqrt{\lg \epsilon^{-1}})$ and has probability of failure ϵ. The dynamic programming part of this algorithm is the construction of T.

6. Summary and Conclusions

We began the body of this thesis with careful analysis of the *BBHT* algorithm, finding its probability of failure; the result allowed us to construct an algorithm, *findsol*, that solves unstructured search (the same problem as Grover's algorithm solves) faster than previous algorithms. The benefit of this tool is widespread: any algorithm making use of unstructured search may use *findsol*, often saving $\sqrt{\lg}$ time factors over *BBHT*, the previous tool of choice.

We summarize *findsol* in Table 6.1, contrasting it with the best pre-existing quantum algorithm (*BBHT*) as well as the fastest possible probabilistic and deterministic classical algorithms that achieve the same goals.

Unstructured Search	Solution Exists	No Solution
Quantum	$\Theta(\sqrt{N/M} + \sqrt{N} \lg \epsilon^{-1}/M^{1.86})$	$\Theta(\sqrt{N \lg \epsilon^{-1}})$
Previous Quantum	$\Theta(\sqrt{N/M})$	$\Theta(\sqrt{N} \lg \epsilon^{-1})$
Classical Probab.	$\Theta(N/M)$	$\Theta(N \lg \epsilon^{-1})$
Classical Determ.	$\Theta(N)$	$\Theta(N)$

Table 6.1: Section 3.1's *findsol* compared to alternatives. The unit of time is the number of calls to F; N is the size of the domain, M is the number of solutions, and ϵ is the maximum probability of failure we will tolerate.

After analyzing *BBHT* and introducing *findsol*, we discussed a series of other tools designed to be used in the construction of algorithms. The tools are various kinds of quantum searches, allowing us to perform operations like minimum-finding faster than we could do with a classical computer: they are summarized in Table 6.2, and contrasted with the pre-existing quantum algorithms as well as the fastest possible deterministic classical algorithms achieve the same goals.

After constructing these tools, we made use of them, finding applications in graph theory. The tools are presumably applicable in a wide range of algorithms, and we chose to target some of the more central, classic algorithms in graph theory: breadth-first search, depth-first search, single-source short-

	Minimum Finding	Finding all M solutions
Quantum	$\Theta(\sqrt{N}\lg\epsilon^{-1})$	$\Theta(\sqrt{NM}+\sqrt{N}\lg\epsilon^{-1})$
Previous Quantum	$\Theta(\sqrt{N}\lg\epsilon^{-1})$	$\Theta(\sqrt{NM}+\sqrt{N}\lg\epsilon^{-1})$
Classical Determ.	$\Theta(N)$	$\Theta(N)$

	Finding d Minimal, Different Objects
Quantum	$\Theta(\sqrt{Nd}+\sqrt{N}\lg\epsilon^{-1}+d\lg N\lg d)$
Previous Quantum	$\Theta(\sqrt{Nd}+\sqrt{N}\lg\epsilon^{-1}+?)$
Classical Determ.	$\Theta(N)$

Table 6.2: Section 3.2-3.4's algorithms *minfind*, *findall* and *mindiff* compared to alternatives. The unit of time is the number of calls to F, except in the $d\lg N\lg d$ term of *mindiff*, where the unit is time. N is the size of the domain, M is the number of solutions, and ϵ is the maximum probability of failure we will tolerate.

est paths (with negative edge weights allowed), and all-pairs shortest paths. We summarize those results in Table 6.3, contrasting with the fastest possible classical algorithms in the case of breadth-first and depth-first search, and the fastest known classical algorithms in the case of single-source and all-pairs shortest paths. Note that several of our graph algorithms can run more slowly than their classical counterparts for E sufficiently small; in each such case there is some a such that the quantum algorithm is faster if $E\in\Omega(V\lg^a V)$.

Problem	Quantum Complexity	Classical
BFS	$O(\sqrt{VE}\lg(V\epsilon^{-1}))$	$\Theta(E)$
DFS	$O(\sqrt{VE}\lg(V\epsilon^{-1}))$	$\Theta(E)$
Single-Src. S.P.	$O(\sqrt{V^3E}\lg(V\epsilon^{-1}))$	$\Theta(VE)$
All-Pairs S.P.	$O(\sqrt{V^3E}(\lg V+\sqrt{\lg(V\epsilon^{-1})})+V^2\lg^3 V)$	$\Theta(V^{2.575})$

Table 6.3: Chapter 4's algorithms, *BFS* (breadth-first search), *DFS* (depth-first search), *SPNW* (single-source shortest paths with negative weights) and *APSP* (all-pairs shortest paths with negative edge weights) compared to classical alternatives. Results are presented in the edge array model; V is the number of vertices in the graph, E is the number of edges, and ϵ is the maximum probability of failure we will tolerate. In this table, to convert the complexity from the edge array model to the adjacency matrix model, change E to V^2.

We have also adapted a graph algorithm by Dürr, Heiligman, Høyer and Mhalla, as well as one by Ambainis and Špalek, to use our tools; we have thus given them slight speedups by improving how they deal with the possibility of failure. The results are summarized in Table 6.4, and contrasted with the fastest known classical algorithms.

	Single-Source Shortest Path (+ve edge weights)
Quantum Adaptation	$O(\sqrt{EV}\lg^2 V + V\lg^3 V + V\lg V\sqrt{\lg(V\epsilon^{-1})})$
Previous Quantum	$O(\sqrt{EV}\lg^2 V\lg(V\epsilon^{-1}) + V\lg^3 V + ?)$
Classical	$\Theta(E + V\lg V)$
	Bipartite Matching
Quantum Adaptation	$O(V\sqrt{E\lg(V\epsilon^{-1})})$
Previous Quantum	$O(V\sqrt{E}\lg(V\epsilon^{-1}))$
Classical	$\Theta(\sqrt{V}E)$

Table 6.4: Section 4.5's algorithms, compared to the algorithms they were adapted from and the fastest known classical algorithms. Results are presented in the edge array model; V is the number of vertices in the graph, E is the number of edges, and ϵ is the maximum probability of failure we will tolerate. In this table, to convert the complexity from the edge array model to the adjacency matrix model, change E to V^2.

Finally we addressed a few problems in computational geometry and dynamic programming. In Section 5.1 we took a problem (*maxpoints*) whose best classical limits are $\Omega(N)$ and $O(N^2)$, and invented a bounded-error $\Theta(N^{3/2}\lg N)$ algorithm for it; hopefully there are other problems that can be addressed similarly. In Section 5.2.1 we addressed the coin changer problem, a problem that is commonly used when introducing dynamic programming. This example is illustrative of what one can do for dynamic programming with judicious use of *minfind*; meanwhile the maximum subarray-sum problem in Section 5.2.2 is a good illustration of how one can make minor adaptations to a classical algorithm that make it amenable to quantum techniques.

6.1 Future Directions

In this thesis we have chosen to focus on deriving new algorithms rather than proving lower bounds. As such, it is possible that the algorithms presented here are not optimal, which presents clear directions for future research:

Problem	Quantum Complexity	Classical
Points on a Line ($\mathbb{Z}^d$)	$N^{3/2} d \lg U \sqrt{\lg \epsilon^{-1}}$	$N^2 d \lg U$
Points on a Line ($\mathbb{R}^2$)	$N^{3/2} \lg N \sqrt{\lg \epsilon^{-1}}$	$N^2 \lg N$
Coin Changer	$D \sqrt{C \lg(D\epsilon^{-1})}$	DC
Maximum Subarray Sum	$N^2 \sqrt{\lg \epsilon^{-1}}$	N^3

Table 6.5: Chapter 5's applications in computational geometry and dynamic programming, compared to fastest known classical algorithms. ϵ is the maximum probability of failure we will tolerate.

searching for lower bounds that approach the upper-bounds presented here, and finding faster algorithms.

There are few published quantum algorithms (at least when viewed in the context of the number of published classical algorithms!), which means that there is a great deal of exciting work to be done; and with so many classical algorithms with no quantum counterparts, much of the low-hanging fruit remains untouched.

A. BBHT: Running Time and Probability of Failure

The probability of failure for *BBHT* is the probability that, for each m up to $2\sqrt{N}$, we never successfully find a result when there is one to be found. To calculate that probability, first we need a result originally derived by Boyer, Brassard, Høyer and Tapp[2]: first, note that after j Grover iterations, the probability of returning a valid result is $\sin^2((2j+1)\theta)$. For a given m, j could be any of $0, \ldots, m-1$; averaging over those values, we see the following for the probability of failure for any given iteration through step 3:

$$\begin{aligned}
P_{\text{fail},m} &= \sum_{j=0}^{m-1} \frac{1}{m} \sin^2((2j+1)\theta) \\
&= \frac{1}{2m} \sum_{j=0}^{m-1} 1 - \cos((2j+1)2\theta) \\
&= \frac{1}{2} - \frac{1}{4m} \sum_{j=0}^{m-1} \left(e^{i4j\theta} e^{i2\theta} + e^{-i4j\theta} e^{-i2\theta} \right) \\
&= \frac{1}{2} - \frac{1}{4m} \left(e^{i2\theta} \frac{1 - e^{i4\theta m}}{1 - e^{i4\theta}} + e^{-i2\theta} \frac{1 - e^{-i4\theta m}}{1 - e^{-i4\theta}} \right) \\
&= \frac{1}{2} - \frac{\sin(4m\theta)}{4m \sin(2\theta)}
\end{aligned}$$

Which is the probability that an invalid result will be returned, for m an integer. m is of course not actually an integer, but by choosing a random integer $0 \leq j < m$, we treat it as one and can consider it to be one for the purposes of that formula.

We wish to upper-bound the probability of error for *BBHT* as a whole, and we will start by differentiating between the cases $0 < \theta \leq \frac{\pi}{4}$ $(M \leq N/2)$ and $\frac{\pi}{4} < \theta \leq \frac{\pi}{2}$ $(M > N/2)$. For any $M \leq N/2$, we wish to find an m_0 such that for each repetition of the outer loop when $m > m_0$, the probability of failure is less than or equal to some constant. For $M > N/2$, we will find that the probability of failure is always less than or equal to some constant.

We begin by considering $M \leq N/2$. In order to find m_0, first we have to find critical points of $f_\theta(m) \equiv \frac{1}{2} + \frac{\sin(4m\theta)}{4m\sin(2\theta)}$, the probability that an invalid result will be returned:

$$\begin{aligned} \frac{df_\theta(m)}{dm} &= 0 \\ \frac{4\theta\cos(4m\theta)}{4m\sin(2\theta)} &= \frac{\sin(4m\theta)}{4m^2\sin(2\theta)} \\ 4m\theta &= \tan(4m\theta) \\ 4m\theta &= 0, 4.49, 7.73, \ldots \end{aligned}$$

Now we consider the form of $f_\theta(m)$. It starts off at $f_\theta(0) = \frac{1}{2} + \frac{\theta}{\sin(2\theta)}$ and decreases from there; we want to find the first maximum it will return to after dipping down, meaning $4m\theta = 7.73$. Since $0 < \theta \leq \frac{\pi}{4}$, we use $\sin(2\theta) \geq \frac{4}{\pi}\theta$, and arrive at (when $4m\theta = 7.73$) $f_\theta(m_0) \leq \frac{1}{2} + \frac{\sin(7.73)}{\frac{4}{\pi}\times 7.73} \approx 0.6$. That does not give us m_0, however: m_0 is when $f_\theta(m)$ first dips that low. Solving numerically and using $\sin\theta \leq \theta$:

$$\begin{aligned} 0.6 &= \frac{1}{2} + \frac{\sin(4m_0\theta)}{\frac{4}{\pi}4m_0\theta} \\ 4m_0\theta &\leq 2.78 \\ m_0 &\leq 0.69/\sin\theta \\ m_0 &\leq 0.69\sqrt{N/M} \end{aligned}$$

For $\frac{\pi}{4} < \theta \leq \frac{\pi}{2}$, although $f_\theta(m)$ is well-behaved and slowly-oscillating over the space of integer values of m, it oscillates wildly in between; so our previous approach, based on considering f_θ as a function acting on the continuum, will not work. To fix this problem, instead of considering θ, we now consider the angle $\phi \equiv \frac{\pi}{2} - \theta$; first noting that $f_\theta(m) = 1 - f_\phi(m)$, meaning that success for θ corresponds to failure for ϕ:

$$P_{fail}(m) = \frac{1}{2} + \frac{\sin(4m\theta)}{4m\sin(2\theta)} = \frac{1}{2} + \frac{\sin(4m(\frac{\pi}{2} - \phi))}{4m\sin(\pi - 2\phi)} = \frac{1}{2} - \frac{\sin(4m\phi)}{4m\sin(2\phi)}$$

Now we are back in the elysian realm of $0 \leq \phi < \frac{\pi}{2}$, and we can bound the probability of failure for ϕ from below and use that result. The procedure here is as before, but instead of 7.73 we use the first root of $\tan(4m\phi) = 4m\phi$, 4.49. For $\phi < \frac{\pi}{4}$ we use $\sin(2\phi) \leq 2\phi$, and arrive at (when $4m\phi = 4.49$)

$P_{fail} \geq \frac{1}{2} + \frac{\sin(4.49)}{2\times 4.49} \approx 0.39$. That is the lowest the probability of failure $f_\phi(m)$ ever gets, and correspondingly it is the lowest the probability of success $1 - f_\theta(m)$ ever gets.

We now have that, for any given iteration of the outer loop, the probability of failure for $M > N/2$ is less than or equal to 0.61 for all m, and the probability of failure for $M \leq N/2$ is less than or equal to 0.6 for $m \geq m_0 = 0.69\sqrt{N/M}$. We now compute the total probability of failure and running time for each case.

For $M > N/2$ the total probability of failure is simply $0.61^{\lg_\lambda(2\sqrt{N})} \approx .5N^{\frac{-0.26}{\ln\lambda}}$, and the probability of getting to the k^{th} iteration through the main loop is 0.61^k. This gives us a total running time of $\sum_{k=0}^{\lg_\lambda(2\sqrt{N})} \frac{\lambda^k}{2}(0.61)^k < \frac{1}{2}\frac{1}{1-0.61\lambda}$.

For $M < N/2$ the total probability of failure is $0.6^{\lg_\lambda(2\sqrt{N}) - \lg_\lambda(0.69\sqrt{N/M})}$, which gives us $0.6^{\lg_\lambda(2.8\sqrt{M})} \approx (2.8M)^{-0.25/\ln\lambda}$. The running time is the sum:

$$\begin{aligned}
t &= \sum_{k=0}^{\lg_\lambda(0.69\sqrt{N/M})} \frac{\lambda^k}{2} + \sum_{k=\lg_\lambda(0.69\sqrt{N/M})}^{\lg_\lambda(2\sqrt{N})} \frac{\lambda^k}{2}(0.6)^{k-\lg_\lambda(0.69\sqrt{N/M})} \\
&\approx \int_0^{\lg_\lambda(0.69\sqrt{N/M})} \frac{\lambda^k}{2}dk + \int_{\lg_\lambda(0.69\sqrt{N/M})}^{\lg_\lambda(2\sqrt{N})} \frac{\lambda^k}{2}(0.6)^{k-\lg_\lambda(0.69\sqrt{N/M})}dk \\
&= \frac{0.69\sqrt{N/M}}{2\ln\lambda} + (0.69\sqrt{N/M})^{-\lg_\lambda 0.6} \int_{0.69\sqrt{N/M}}^{2\sqrt{N}} \frac{dx}{2} x^{\lg_\lambda 0.6} \\
&= \frac{0.69\sqrt{N/M}}{2\ln\lambda} + (0.69\sqrt{N/M})^{-\lg_\lambda 0.6} \left[\frac{dx}{2}\frac{x^{1+\lg_\lambda 0.6}}{1+\lg_\lambda 0.6}\right]_{0.69\sqrt{N/M}}^{2\sqrt{N}} \\
&= \frac{0.69\sqrt{N/M}}{2\ln\lambda} + \frac{(3\sqrt{M})^{\lg_\lambda 0.6}}{1+\lg_\lambda 0.6}\sqrt{N} - \frac{1}{2}\frac{\sqrt{N/M}}{1+\lg_\lambda 0.6}
\end{aligned}$$

Since we have $\sqrt{N/M}$ dependence from the first term, we should choose λ such that the second term contributes no worse, which gives us the condition $\lg_\lambda 0.6 < 1$, or $\lambda < 1.64$. We now have:

$$t \leq \frac{0.69\sqrt{N/M}}{2\ln\lambda} - \frac{1}{2}\frac{\sqrt{N/M}}{1+\lg_\lambda 0.6}$$

which is minimal for $\lambda \approx 1.31$, and more importantly is $\Theta(\sqrt{N/M})$. Boyer, Brassard, Høyer and Tapp arbitrarily chose $\lambda = \frac{8}{7}$, which we would like to say makes the algorithm 50% slower than our choice of λ; but that is only true in this approximation. Furthermore, the optimal value for λ is actually a function of M/N, so there is no one optimal λ in general.

Using $\lambda = 1.31$, our results can be summarized in Table A.1. Most important to us is that our running time is $\Theta(\sqrt{N/M})$ calls to F, and our probability of failure is less than $.5M^{-.93}$. It is also worth noting that our earlier restriction, $\lambda < 1.64$, came because we chose a small root for $\tan(x) = x$. If we had chosen a larger root, λ could have been larger, up to an asymptotic maximum of 2.

Case	Probability of Failure	Expected Running Time
$M \leq N/2$	$\leq .4M^{-.93}$	$\leq 1.9\sqrt{N/M}$
$M > N/2$	$\leq .5N^{-.96}$	≤ 2.3

Table A.1: Probability of failure and expected running time for *BBHT* (for $\lambda = 1.31$)

Bibliography

[1] L. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the 28th Annual ACM Symposium on Theory of Computing (STOC)*, pages 212–219, 1996.

[2] M. Boyer, G. Brassard, P. Høyer, and A. Tapp. Tight bounds on quantum searching. *Fortschritte Der Physik*, 46:493–505, 1998.

[3] H. Buhrman, R. Cleve, R. de Wolf, and Ch. Zalka. Bounds for small-error and zero-error quantum algorithms. In *Proceedings of the 40th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 358–368, 1999.

[4] C. Dürr and P. Høyer. A quantum algorithm for finding the minimum. quant-ph/9607014, 1996.

[5] Yu.I. Manin. Computable and uncomputable (in Russian). Moscow, Sovetskoye Radio, 1980.

[6] R. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6/7):467–488, 1982.

[7] P. Benioff. Quantum mechanical hamiltonian models of Turing machines that dissipate no energy. *Journal of Mathematical Physics*, 22:495.

[8] C. Dürr, M. Heiligman, P. Høyer, and M. Mhalla. Quantum query complexity of some graph problems. In *Proceedings of ICALP 2004, Turku, Finland*, 2004.

[9] A. Ambainis and R. Špalek. Quantum algorithms for matching and network flows. cs/0508205, 2005.

[10] H. Tamaki and T. Tokuyama. Algorithms for the maximum subarray problem based on matrix multiplication. In *Proceedings of the 9th Symposium on Discrete Algorithms (SODA)*, volume 12, pages 446–452, 1998.

[11] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[12] A. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

[13] S.C. Kleene. Lambda-definability and recursiveness. *Duke Mathematical Journal*, 2:340–353, 1936.

[14] D. Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London Ser. A*, A400:97–117, 1985.

[15] D.R. Simon. On the power of quantum computation. In *Proceedings of the 35th Annual IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 116–123, 1994.

[16] P. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS)*, 1994.

[17] C.H. Bennett, E. Bernstein, G. Brassard, and U. Vazirani. Strengths and weaknesses of quantum computing. *SIAM Journal on Computing*, 26(5):1510–1523, 1997.

[18] Yellow Pages Group Co. Canada 411. http://www.canada411.com, 2006.

[19] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms.* MIT Press, 2001.

[20] M.A. Nielsen and I.L. Chuang. *Quantum Computation and Quantum Information.* Cambridge University Press, Cambridge, 2000.

[21] Gilles Brassard and Peter Høyer. An exact quantum polynomial-time algorithm for Simon's Problem. In *Israeli Symposium on Theory of Computing and Systems*, pages 12–23, 1997.

[22] C.H. Bennett. Logical reversibility of computation. *IBM Journal of Research and Development*, 17(6):525–532, 1973.

[23] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[24] L. Ford and D. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[25] U. Zwick. All pairs shortest paths in weighted directed graphs: exact and almost-exact algorithms. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1998.

[26] D. Johnson. Efficient algorithms for shortest paths in sparse networks. In *Proceedings of the IEEE Symposium on Foundations of Computer Science (FOCS)*, 1998.

[27] J. Hopcroft and R. Karp. An $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM Journal of Computing*, 2(4), December 1973.

[28] A. Gajentaan and M. Overmars. On a class of $O(n^2)$ problems in computational geometry. *CGTA: Computational Geometry: Theory and Applications*, 5, 1995.